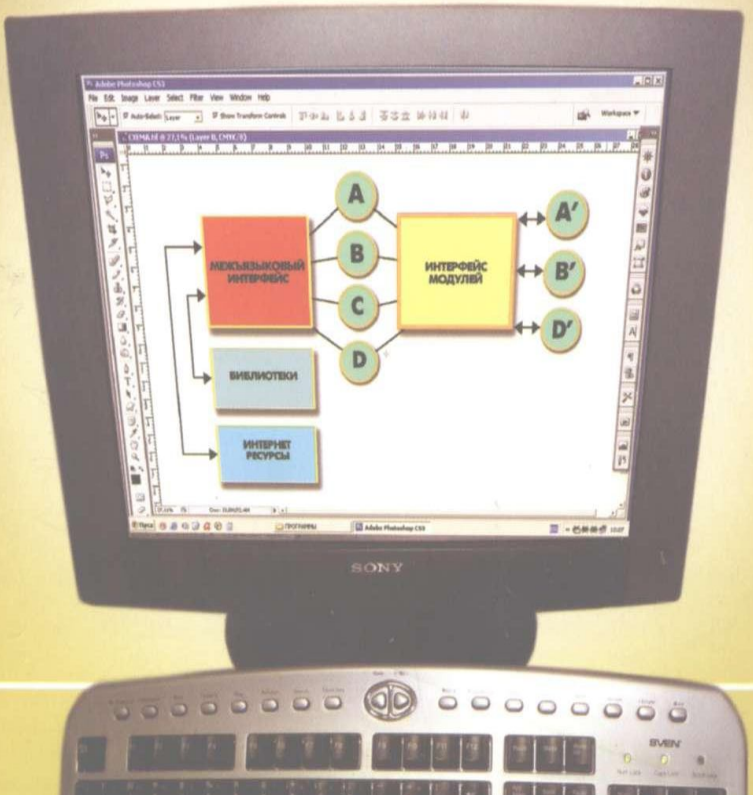
Е.М. Лаврищева
В.Н. Грищенко

СБОРОЧНОЕ ПРОГРАММИРОВАНИЕ

Основы индустрии программных продуктов



НАЦИОНАЛЬНАЯ АКАДЕМИЯ НАУК УКРАИНЫ
ИНСТИТУТ ПРОГРАММНЫХ СИСТЕМ

Е.М.Лаврищева
В.Н.Грищенко

СБОРОЧНОЕ ПРОГРАММИРОВАНИЕ

Основы индустрии программных продуктов

Второе издание
Дополненное и переработанное

КИЕВ
НАУКОВА ДУМКА
2009

УДК 519.681.2

Лаврищева Е.М., Грищенко В.Н.

Сборочное программирование. Основы индустрии программных продуктов: 2-изд. Дополненное и переработанное.–Киев: Наук. думка, 2009.–372с.–ISBN 978-966-00-0848-1.

В монографии систематизированы существующие подходы и методы сборки сложных программ из более простых программных ресурсов (модулей, компонентов, компонентов повторного использования – КПИ и программ). Приведено теоретическое обобщение и обоснование метода сборочного программирования программных систем из программных ресурсов. Определены основные операции над ними. Разработаны формальные основы компонентного программирования: модели компонентов, интерфейсов, среды, а также внешней и внутренней алгебры. Дана классификация программных ресурсов и подходов к применению информационных ресурсов. Рассмотрены технологические аспекты метода сборки модулей, КПИ и других готовых ресурсов, а также предложена новая концепция инженерии сборки технологий для реализации программных приложений СОД на их основе. Приведено общее описание средств автоматизации метода сборки программных ресурсов, определены фундаментальные основы (теоретические, инженерные, экономические, управленческие) индустрии программных продуктов различного назначения.

Для специалистов, занимающихся автоматизированным созданием программных систем из готовых ресурсов в разных предметных областях с использованием современных инструментально-технологических средств и сред, а также студентов и аспирантов вузов специальностей – прикладная математика и программная инженерия.

Ответственный редактор академик НАН Украины Ф. И. Андон

Утверждено к печати ученым советом
Института программных систем НАН Украины

Научно-издательский отдел физико-математической
и технической литературы
Редактор М.К.Пунина

ISBN 978-966-00-0848-1

ОГЛАВЛЕНИЕ

СПИСОК СОКРАЩЕНИЙ

АИС	–	автоматизированная информационная система
АСНИ	–	автоматизированная система научных исследований
АСУ	–	автоматизированная система управления
БИМ	–	библиотека исходных модулей
ВБМ	–	временная библиотека модулей
ГКНТ	–	Государственный комитет по науке и технике
ЖЦ	–	жизненный цикл
ЕСКД	–	единая система конструкторской документации
ЕСПД	–	единая система программной документации
ИВ	–	информационный вектор
ИК	–	интегрированный комплекс
ИР	–	информационный ресурс
ИС	–	информационная система
КП	–	комплекс программ
КТП	–	карта технологического процесса
ЛБ	–	личная библиотека
МТЛ	–	модель технологической линии
МТП	–	модель технологического процесса
МЯИ	–	межъязыковый интерфейс
ОКМ	–	объектно-компонентный метод
ОМ	–	объектная модель
ООП	–	объектно-ориентированный подход
ПИ	–	программная инженерия
ПК	–	программный компонент
ПО	–	программное обеспечение
ПП	–	программный продукт
ППО	–	прикладное программное обеспечение
ППП	–	пакет прикладных программ
ПС	–	программная система
ПТД	–	проектно-технологический документ
ПрО	–	предметная область
СА	–	системная архитектура
САА	–	система алгоритмических алгебр
САМП	–	система автоматизации модульного программирования
САПР	–	система автоматизации проектирования
СОД	–	система обработки данных
СППО	–	система проблемно-ориентированного обеспечения
ССПМ	–	система сборки программ из модулей
ТД	–	технологический документ
ТЛ	–	технологическая линия
ТИ	–	технологический интерфейс
ТМ	–	технологический модуль
ТО	–	технологическая операция
ТП	–	технологический процесс
ТПР	–	технология подготовки разработки
ТЗ	–	техническое задание
ФА	–	функциональная архитектура
ФОТ	–	функционально-ориентированная технология

ЭПД	–	эксплуатационный документ
ЯМК	–	язык модульного конструирования
ЯОК	–	язык описания классов
ЯОТ	–	язык описания типов
ЯОУ	–	язык описания модели управления
ЯП	–	язык программирования
ACM	–	Association for Computing Machinery
CBD	–	Component-Based Development
CMM	–	Capability Maturity Models
CS	–	Computer science
GDM	–	Generative Domain Model
DSL	–	Domain specific language
SWEBOK	–	Software Engineering Body Knowledge
V&V	–	верификация и валидация

ПРЕДИСЛОВИЕ

Широкое распространение видов и типов новой компьютерной техники ставит перед специалистами важные задачи по эффективному её использованию при создании программных и прикладных систем массового применения.

Анализ потребности разных отраслей хозяйства и промышленности в программных системах (ПС) показывает, что на практике их требуется более 3,5 млн. различного наименования, назначения и сложности. Например, затраты на объем выпуска в 2005 г. составили 800 млн. дол. США, это на порядок меньше необходимого.

В ближайшее годы создание и адаптация разрабатываемых ПС к различным задачам отраслей значительно улучшатся за счет широкомасштабного применения готовых программных ресурсов повторного использования, аналогично тому, как это происходит в машиностроении или автомобильной промышленности при сборке крупных изделий из готовых деталей.

Повышению уровня качества автоматизации ПС, а также увеличению в 4–6 раз производительности труда программистов за последние годы способствовали:

– фундаментальные и прикладные научные исследования по созданию теоретических, прикладных и технологических основ индустрии разработки и производства ПС;

– разработка организационно-методических и стандартных руководств для управления коллективной разработкой программных проектов как инструментов эффективного создания типовых высокопроизводительных технологий производства ПС;

– практические разработки программных систем специального назначения, в том числе и бизнес-систем, в небольших и крупных коммерческих организациях;

– обучение студентов ВУЗов новым концепциям программной инженерии.

Один из путей решения первой задачи – совершенствование технологии программирования в целях перехода на промышленные методы создания ПС, которые характеризуются технологическими линиями (ТЛ) по производству программных продуктов (ПП), удовлетворяющих определенным экономическим критериям. Основная цель технологии программирования, представленной ТЛ, – четкая регламентация и организация работ по проектированию, разработке и изготовлению программных продуктов высокого качества и высокой производительности.

Условием повышения производительности труда разработчиков ПС является использование готовых производственных ресурсов, а именно, компонентов повторного использования (КПИ) со стандартными или унифицированными интерфейсами, способствующих производить формализованный сбор разнородных ПП согласно заданным требованиям заказчиков. Сборочный процесс производства ПП априори основывается на контроле деятельности исполнителей и оценке

показателей качества как отдельных КПИ, так и продукта в целом.

Иными словами, новыми тенденциями в современной технологии программирования являются:

– распространение на сферу программирования промышленных методов организации (планирование трудозатрат, определение трудоемкости, учет, контроль результатов труда и др.) проведения работ при изготовлении ПС;

– перенос акцента с процесса программирования ПС на более ранние процессы, обеспечивающие анализ автоматизируемой предметной области и формирование требований к создаваемому продукту;

– введение в практику разработки ПС таких понятий, как ТЛ, модель жизненного цикла (ЖЦ), технологическая карта, маршрут и т. п., являющихся основным фундаментом организации управления разработкой промышленных изделий.

Наиболее быстрый переход к промышленной организации разработки ПС может быть обеспечен путем систематизации и формализации сборочного программирования, основанного на многократном использовании КПИ как готовых наборов модулей для разных областей, по словам А.П.Ершова: «чистой экономии труда, которая должна составлять 2/3 годового производства программных средств.».

Цель настоящей монографии состоит в том, чтобы теоретически сформулировать, проверить практически и реализовать модели, методы и средства сборочного программирования, позволяющие на промышленной основе изготавливать сложные ПС из более простых программных ресурсов, а так же определить перспективные пути развития технологии программирования и индустриальных основ производства ПП.

Монография состоит из десяти глав.

Первая глава содержит описание основных форм программирования (синтезирующего, конкретизирующего, сборочного, компонентного и др.) и описание проблем представления и реализации ПС в среде сборочного программирования. Даны основные определения, сформулированы цели и задачи сборочного программирования, его место в современном процессе создания ПС.

Во второй главе проанализированы существующие методы разработки ПС сложной структуры, базирующиеся на концепции сборки (комплексирование, интеграция, композиция), определены объекты и основные модели сборочного программирования: модель информационного сопряжения модулей (экспорт и импорт данных); модель управления и выбора готовых ресурсов из разных источников (библиотек, системы Интернет и др.), необходимых для сборки. Дан анализ современных стилей программирования, поддерживающих парадигму сборки.

В третьей главе описаны задачи комплексирования модулей и определены формальные механизмы преобразования типов и структур передаваемых данных между программными объектами. На основе теории структурной организации данных К. Хоара и В. Вирта представлен алгебраический подход к определению и преобразованию типов данных, заданных в разных языках программирования (ЯП). Такой подход основан на утверждениях, доказывающих необходимые и достаточные условия преобразования типов данных объектов, которые записаны в разных языках программирования (ЯП) и подлежат сборке в сложные ПС.

В четвертой главе сформулированы принципы выполнения операций над модульными и программными структурами, а также задачи проверки их правильности. При реализации типов структур программ применен аппарат теории графов и матричный способ представления графов сложных объектов матрицей смежности и достижимости. Введены операции (объединения, соединения, проекции и разности) над объектами с модульной организацией и описаны утверждения, определяющие действия над ними в матричном представлении.

Пятая глава посвящена подходу к классификации и типизации компонентов и других программных ресурсов. Определены классификационные характеристики и признаки для класса программных и информационных ресурсов Интернета. Обоснован базис типизации компонентов и решений по их повторному использованию. Проведена типизация компонентов веб-приложений, метод доступа к информационным ресурсам и их интеграции при создании веб-сайтов, баз данных, XML-документов и др.

В шестой главе представлен новый объектно-компонентный метод создания ПС из готовых КПИ. Представлен теоретический фундамент объектного и компонентного программирования, включающий в себя базовую терминологию, модели (объектная и компонентная), внешнюю и внутреннюю компонентные алгебры. Приведены утверждения относительно алгебры компонентного программирования, показана коммутативность операций в ней и определено место алгебры в среде разработки ПС. Продемонстрировано применение концепции сборки разноязыковых компонентов в современных средах и системах: CORBA, JAVA, MS.NET и др.

Седьмая глава посвящена методам построения интегрированных комплексов программ, применяемых в процессе проектирования и сборки. Рассмотрен ряд готовых инструментальных систем, используемых при интеграции программ. Показано использование аппарата логического проектирования интегрированных комплексов и применение экспертных систем для представления функциональных, программно-технических и технологических характеристик ПС, влияющих на процесс интеграции/сборки программных ресурсов. Предложены языки описания типов данных, классов их эквивалентности и управления программными объектами.

В восьмой главе сформулированы основные понятия технологии сборочного программирования: объект разработки, методы и инструменты разработки, методы организации и управления разработкой. Сформулирована задача, принципы и способы сборочного формирования конкретных технологических линий (ТЛ), реализующих задачи и функции СОД на производственной основе. Дано описание процессов и действий конкретно, созданных ТЛ применительно к производству по ним классов задач обработки данных, в том числе научно-исследовательских и математических.

В девятой главе подробно рассмотрены вопросы организации и управления разработками программ на основе ТЛ. Описаны методы планирования трудозатрат и трудоемкости, снижения сложности и управления качеством ПС на процессах ЖЦ. Приведена методика оценки показателей качества ПС. Дана характеристика моделей надежности и процесса тестирования ПС, обеспечивающего подготовку данных для определения надежности по оценочным моделям.

В десятой главе представлен системный анализ аспектов индустриального производства ПП (ЖЦ, разделы ядра знаний SWEBOK, PMBOK, стандарты

качества и оценивания ПП). Рассмотрен базовый процесс инженерии производства компьютерных программ с использованием ядра SWEBOK, стандартов ЖЦ, инфраструктуры и менеджмента. Предложена система дисциплин (научная, инженерная, управленческая, экономическая), охватывающая основные аспекты в производственном цикле индустриального изготовления ПП. Сформулированы теоретические основы и роль в пространстве задач информатики.

В приложении 1 приведена структура онтологической системы КПИ, в приложении 2 дан пример описания веб-приложения, а в приложении 3 – интерфейс компонентов в ЯП Паскаль и Дельфи.

Данное издание монографии переработано и дополнено. В нее включены основные направления становления и развития сборочного программирования с участием авторов. Идею сборки как конвейерного механизма фабрик программ предлагали В.М.Глушков и А.П.Ершов (1967г.). Далее (1985–1988гг.) формирование нового вида программирования, а именно сборочного, происходило при участии Е.Л.Ющенко. Средства сборки модулей, программ и компонентов развиваются и совершенствуются в настоящее время. Базовая концепция организации сборки разных видов и типов программных объектов – интерфейс, как связывающее звено разноязыковых программ в любых средах, – получила полную формализацию в языках описания интерфейса IDL (Interface Definition Language) и стандарта описания общих типов данных ISO/IEC 11404–1996, 2007 независимо от ЯП.

В монографию вошли новые результаты исследований и разработок, касающиеся развития процессов сборки применительно к компонентам для современных систем и сред, а также определения новой системы дисциплин программной инженерии, охватывающих все аспекты производственной деятельности по созданию программных продуктов. Научно-технической поддержкой парадигмы сборочного программирования являются результаты аспирантов и соискателей (с.н.с. Н.Т.Задорожная, Г.И.Коваль, Т.М.Коротун, О.А.Слабоспицкая), защитивших кандидатские диссертации в 2004–2008 гг., а также подготовленной к защите докторской диссертации В.Н.Грищенко и кандидатской диссертации С.Л.Поляничко (под руководством Е.М.Лаврищевой). К ним относятся ключевые основополагающие концепции теории сборки КПИ, управления программными проектами, инженерии тестирования и качества, оценивания процессов и продуктов, а также моделей представления знаний о КПИ. Таким образом, был сделан существенный вклад в развитие и совершенствование базовых положений программной инженерии, что способствовало формированию основ индустриального производства программных продуктов высокого качества.

Авторы благодарны академику НАН Украины Ф.И.Андону за полезные советы, замечания и поддержку в процессе подготовки новой редакции монографии, а также сотрудникам научного отдела Института программных систем (с.н.с. Л.П.Бабенко, Е.В.Карпусь, Г.И.Коваль, Л.И.Куцаченко и др.) за помощь в подготовке рукописи и реализации инструментально-технологических средств сборочного и компонентного программирования.

Глава 1

ПРОБЛЕМАТИКА СБОРОЧНОГО ПРОГРАММИРОВАНИЯ ПРОГРАММНЫХ СИСТЕМ

1.1. ОСНОВНОЕ СОДЕРЖАНИЕ МЕТОДА СБОРОЧНОГО ПРОГРАММИРОВАНИЯ

Одна из главных задач современного программирования – создание теоретических и прикладных основ построения сложных программ из более простых программных элементов, которые записаны в современных ЯП. Фактически решение этой задачи осуществляется путем сборки, объединения или интеграции разнородных программных ресурсов, КПИ, включая модули и программы реализации некоторой предметной области (ПрО). Цель метода сборки – интеграция в новые программные структуры КПИ, программ, готовых ресурсов, накопленных в Интернете.

1.1.1. РАННИЕ МЕТОДЫ ИНТЕГРАЦИИ ПРОГРАММНЫХ ОБЪЕКТОВ

Интеграция программных структур сначала выполнялась с помощью готовых подпрограмм библиотек разного назначения путем их вставки в интегрируемую программную систему. Затем появлялись разные стили программирования (конкретизирующее, синтезирующее, композиционное и др.), которые решали проблему комплексирования программных объектов методами, близкими к сборке разнородных объектов (табл. 1.1). Они способствовали постепенному развитию индустриальных основ изготовления программных систем (ПС): КПИ, жизненный цикл (ЖЦ), измерение и оценивание работ на процессах ЖЦ и самого продукта на отдельные показатели качества. Позднее сформировались и другие стили программирования, которые используют готовые компоненты в процессе создания ПС методами сборочного типа – комплексирование, интеграция, композиция. К ним можно отнести: объектно-ориентированное, компонентное, генерирующее, аспектное, агентное и др.

Рассмотрим ранние методы программирования с элементами интеграции, комплексирования и синтеза.

Конкретизирующее программирование базируется на выделении из некоторой универсальной программы отдельной ее части, настроенной на особые, определенные условия выполнения. Можно отметить два типа такого выделения. Первый характеризуется формированием конкретной программы и аналогичен процессу макрогенерации. Второй тип связан с конкретизацией информационных структур в используемом программном средстве.

Таблица 1.1

Вид программирования	Основа	Метод	Средство автоматизации
Конкретизирующее программирование (происходит от наличия некоторого универсального ПС)	Универсальная многопараметрическая структура ПС. (СУБД, ППП и т.п.)	Метод адаптации с техникой смешанных вычислений, оптимизационных преобразований и генерации ПК, как конкретизация более общей программы к условиям ее применения (к структуре данных, функциям конкретной ПрО)	СКС технология для адаптации СУБД к условиям применения с помощью: языка описания типов данных и средств манипулирования; разбиения ПС на мелкие программы; нестандартные функции преобразования
Синтезирующее программирование (происходит от постановки задачи, составляемой в виде модели вычислений или спецификации программы решения задачи)	Модель вычислений (Э.Х.Тыугу [195]), отображающая понятия и отношения ПрО в виде модели программы, содержащей описание данных и макроопределений. Спецификация программы (В.П.Агафонов [3, 4]) – описание задачи в терминах математичес-ких понятий, используемое в качестве задания при разработке программы	Метод доказательной генерации на основе управляющей программы, заданной последовательностью операторов вызовов, реализующих отношения между объектами модели вычислений. Метод пошагового уточнения (символический метод решения задачи) – доказательное рассуждение (программа) о сути вычисления задачи	Система ПРИЗ, в которой по описанию модели вычислений выбирается путь и строится программа решения задачи путем синтеза отдельных программ задач. Система АТЛАНТ (СО АН СССР) со встроенными типами данных для создания БД. Система СПОРА содержит спецификации задачи и сборщик программ
Композиционное программирование (композиция функций и данных в программы)	Функции и операции композиции в логико-математической системе	Процесс композиции включает: в себя данные, функции, операции композиции и десрипторы (выражения, термы, формулы)	Система построения композиционных программ
Сборочное программирование (происходит от Банка модулей, программ, программных систем, КП и др.)	Схема (модель) сборки – это ориентированный нагруженный граф, в вершинах которого находятся элементы банка, а дуги задают связи между ними и режим функционирования	Метод комплексирования разноязыковых модулей по схеме, задающей их взаимосвязь по управлению и по данным, преобразуемым к релевантным типам данных. Интеграция ПК по схеме из объектов Банка модулей в среде интеграции и преобразования разных типов данных и передач управления	Система АПРОП реализует связь разноязыковых модулей в ЯП ОС ЕС, через модули-посредники, обеспечивающие необходимое преобразование типов и структур несовпадающих данных. Система ДИСУПП устанавливает связи между программными компонентами по маршрутной схеме

При **синтезирующем** программировании строится модель программы по

спецификации задачи, по которой будет синтезирована программа ее решения. Спецификация задается в терминах некоторого формального языка. На основе ее и правил построения алгоритмов описания конкретной предметной области происходит формирование требуемой программы.

Композиционное программирование базируется на принципах функциональности и композиционности, который рассматривает программы как набор функций, строящихся из других функций с помощью специальных операций, названных композициями. На основе композиционного уточнения (экспликации – explication) создана логико-математическая система композиционного построения программ, которая объединяет современные парадигмы программирования (структурное, функциональное, объектно-ориентированное и др.) в рамках единой концептуальной, экспликативной платформы. Ее основу составляют три типа объектов: сами объекты, методы композиции и средства построения из одних объектов других объектов или функций.

Сборочное программирование характеризуется сборочным построением программ из готовых «деталей», которыми являются программные объекты различной степени сложности. Элементы процесса сборки присутствуют во многих методах программирования: сверху–вниз, снизу–вверх и т. д.

Программисты, разрабатывая программы без применения каких-либо методов программирования, выделяют повторно используемые операторы и оформляют их в виде отдельных, самостоятельных фрагментов или подпрограмм для дальнейшего использования.

Возникают вопросы: в чем суть сборочного программирования и что позволяет выделить его в виде отдельного метода. Для ответа на поставленный вопрос прежде всего отметим, что сборочное программирование:

– является одним из методов программирования и подчиняется общим закономерностям;

– представляет одну из форм повторного использования программных средств;

– качественно отличаться от процессов сборки в других методах.

Под **методом сборки** понимается способ сопряжения разноязыковых программных объектов в ЯП, основанный на теории спецификации, и отображения (mapping) типов и структур данных ЯП, представленных алгебраической системой. Основу алгебраического формализма составляют типы данных, операции над ними и функции релевантного, эквивалентного преобразования одних типов в другие. Сопряжение пар объектов в ЯП осуществляется через оператор вызова, в списке параметров которого задаются значения формальным параметрам, которые проверяются на соответствие типов данных утверждениями алгебраической системы, доказывающими необходимые и достаточные условия отображения данных в классе ЯП. Результат отображения – операторы релевантного преобразования типов данных в специальном модуле-посреднике для каждой сопрягаемой пары объектов.

Методом, близким к сборке, является генерация разных объектов к одному общему выходному коду и среде функционирования. Понятие генерации программ возникло почти одновременно с понятием сборки, и на сегодняшний день оно получило новое развитие в связи с ориентацией на описание модели предметной области (домена) средствами языка DSL (Domain Specific Language), отображающими специфику этой области. Такое новое направление еще не имеет

стандартных решений относительно самой проблемы постепенной трансформации описания в этом языке и выполнения инструментов генерации, отладки, интеграции и др. для получения конечной ПС.

Жизненный цикл программного продукта в сборочном программировании – это период времени между началом разработки и завершением его использования. Он характеризуется несколькими процессами, отражающими закономерности при разработке и применении программных средств. Различные подходы, методы, инструментальные средства могут вносить отличительные особенности в отдельные процессы, однако само содержание процессов жизненного цикла (ЖЦ) программного продукта (ПП) вполне определенное. Наиболее общими являются следующие процессы (рис 1.1).

Постановка задачи. На данном процессе формулируется задача, для решения которой необходимо разработать программный продукт. Определяются целесообразность разработки, ее стоимость, сфера применения, эффект от внедрения. Условие решаемой задачи оформляется в виде ее спецификации – технического задания, постановки задачи и т. д.

Проектирование ПП. Этот процесс может делиться на отдельные подпроцессы, отражающие иерархическую структуру ПП, включая общее проектирование, проектирование подсистем, программ и отдельных модулей. Проектирование состоит в разработке алгоритмов решения поставленных задач, абстрактных структур данных, операционной среды функционирования. Результат проектирования – спецификации программных объектов средствами ЯП и описание информационных структур. Здесь решается также вопрос о подборе готовых программ, выполняющих требуемые функции и подфункции некоторой задачи предметной области.

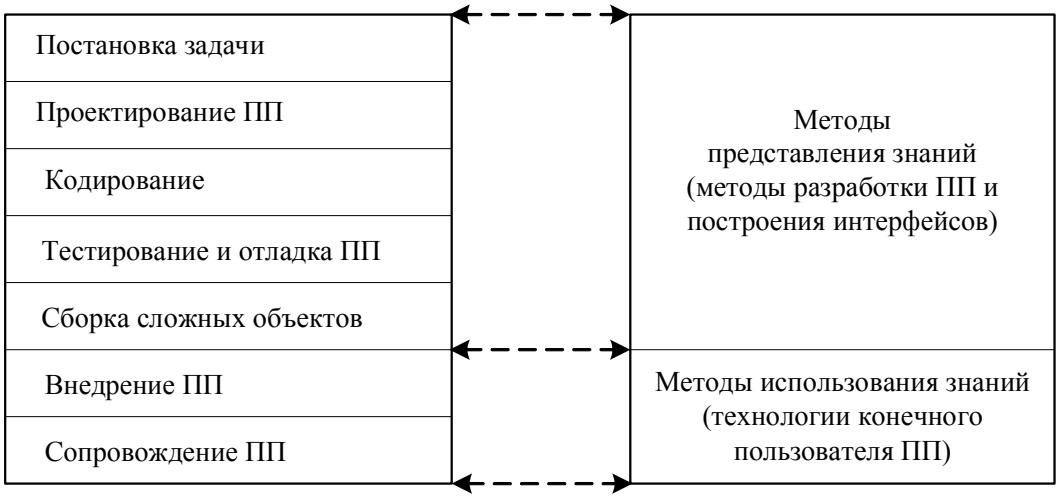


Рис.1.1. Схема представления методов разработки и использования ПС

Проектирование и апробирование готовых ПП. Спецификации проектирования для вновь разрабатываемых компонентов ПП переводятся в тексты на языках программирования. Структуры данных конкретизируются и отображаются в реальную память. Исходные тексты переносятся на внешние носители информации. Готовые компоненты проверяются на выполнение функций в наборе исходных данных.

Тестирование и отладка ПП. Этот процесс делится на несколько подпроцессов, связанных с тестированием и отладкой разных видов программных объектов, входящих в состав ПП, а также с устранением различных типов ошибок, обнаруженных при кодировании или проектировании. Ошибочные результаты данного процесса приводят к выполнению предыдущих действий и необходимых корректирующих изменений в ПП.

Сборка и тестирование сложных программных объектов. Осуществляется сборка готовых ПП из числа повторно используемых и вновь разрабатываемых методом комплексирования или интеграции. В результате создается интерфейсная среда, включающая в себя программы преобразования передаваемых данных и передач управлений соответствующим компонентам объекта сборки. Последний тестируется на множестве тестов, проверяющих созданные интерфейсы (правильность передач данных между объектами) и функционирование всего объекта. Неудовлетворительные результаты тестирования приводят к замене отдельных элементов или к исправлению в них типов и структур передаваемых данных. Эти действия вызывают выполнение указанных выше процессов ЖЦ. Результат данного процесса – опытный образец ПП.

Внедрение ПП. Составляется технологическая документация, разрабатываются контрольные примеры, проводится опытная эксплуатация ПП. После завершения опытной эксплуатации и исправления обнаруженных ошибок опытный образец ПП считается готовым к промышленной эксплуатации.

Сопровождение. Этот процесс – последний в жизненном цикле и длится до завершения использования программного продукта. Окончание применения связано с:

– разработкой более качественного и совершенного ПП;

– завершением решаемой задачи и моральным старением самого ПП или операционной среды его функционирования.

В процессе сопровождения происходит устранение ранее не обнаруженных ошибок и изменение ПП, связанные с совершенствованием технологии его применения.

Использование конкретного метода программирования сказывается на длительности каждого процесса ЖЦ, его сложности и применяемых инструментальных средствах. В рассмотренных выше процессах ЖЦ создаваемого ПС явно не отражены процессы его документирования и изготовления. Это связано с тем, что технология программирования регламентирует постепенную разработку документации, ее корректировку во всех процессах проектирования и изготовления ПП. Поэтому к моменту изготовления и внедрения значительная часть документации уже подготовлена. В процессе изготовления осуществляется копирование носителей программ, доработка документации и подготовка носителей ПП. Применение инструментальных средств поддержки разработки упрощает процесс изготовления документации, и поэтому отпадает необходимость в выделении его как отдельного процесса в рамках технологии сборочного программирования.

ПС как форма представления знаний. Программные средства – это фактически один из способов представления знаний о решаемых задачах, конкретных предметных областях. С данной позиции рассмотрим более подробно процесс разработки и использования ПП.

При разработке баз знаний необходимо рассмотреть два ключевых вопроса:

представление знаний и их использование. Методами представления знаний в рассматриваемой области по сути являются методы и средства разработки ПП как знания специалистов-разработчиков. При этом языками представления знаний служат языки: спецификаций, программирования, описания данных, управления заданиями и т. д. Сами разработанные ПС представляются как часть знаний о предметной области, к которой относится решаемая задача.

Использование знаний связано с процессами внедрения (частично) и сопровождения ЖЦ ПП, т. е. его применения. Метод использования ПП – технология его применения. Результаты рассмотренного подхода к представлению знаний о повторно используемых ПС отражены на рис. 1.1. В частности, этой схеме удовлетворяют все три отмеченных выше виды программирования систем на основе готовых компонентов.

Отличие сборочного программирования от других методов. Главное отличие сборочного программирования от других методов повторного использования ПС состоит в наличии «устойчивой обратной связи», что отражено на рис. 1.2.

Рис.1.2. Схема представления знаний в методе сборочного программирования

«Обратная связь» означает, что в технологию использования разработанного ПП входят методы его применения при проектировании других ПС, где ПП входит как составной элемент. Термин «устойчивый» подчеркивает, что методы применения разработанных ПП в дальнейшей разработке ПС являются необходимым условием использования метода сборочного программирования.

Из рис.1.2. следует, что метод сборочного программирования является обобщением традиционных методов разработки, так как в него дополнительно входят и методы построения интерфейсов между отдельно разработанными ПП.

Следовательно, изменяется и ЖЦ. Процесс проектирования включает выбор готовых ПП, которые полностью или частично решают поставленную задачу, и решение проблем интерфейсов между компонентами. Если требуется дополнительная разработка новых и сложных ПС, то формируется частная задача, решаемая с применением традиционных методов программирования. Процесс кодирования упрощается (отсутствует) при наличии готовых нескольких или всех необходимых компонентов. Основное внимание при этом уделяется комплексной отладке программных объектов, созданных из готовых компонентов. Главные направления процесса объединения накопленных в этих базах знаний разноплановых программных ресурсов – развивающиеся современные методы программирования.

1.1.2. СОВРЕМЕННЫЕ МЕТОДЫ ИНТЕГРАЦИИ ПРОГРАММНЫХ ОБЪЕКТОВ

Наряду с рассмотренными методами интеграционного типа сформировались и другие, которые используют готовые компоненты в процессе создания ПС методами сборочного типа путем комплексирования, интеграции, композиции и др. К ним относятся современные широко применяемые подходы, а именно, объектно-ориентированный, компонентный, генерирующий, аспектный, агентный и др. (табл.1.2).

Рассмотрим современные виды программирования с элементами интеграции – комплексирование, синтез, генерация.

Таблица 1.2

Вид программирования	**Объект**	**Модель**	**Метод**	**Интерфейс и ЯП**	**Среда и инструменты**	**Результат**
Объектно-ориентированное	Объект, класс, интерфейс	Объектная модель	Интеграции, взаимодействия	IDL, RPC, RMI, C++, Visual C++, Java	CORBA, COM, Rational Rose, MS.NET, MSF	ООПС, приложение
Компонентное	Компонент, каркас, контейнер, приложение Reuse	Компонентная модель, модель приложения	Интеграции, композиции, взаимодействия	IDL, RPC, PDL, JNI, RMI, DOM. Платформо-ориетированный интерфейс, трансформация в C++, брокер	COM, CORBA, Windows API, API Viewer, Matlab, RMI	Компонентная система
Порождающее	Объект, класс, компонент, каркас, контейнер, приложение, член семейства	Порождающая доменная модель, объектная, компонентная модель, Checking model	Генерация, взаимодействие, интеграция, сборка, встраивания	IDL, XML, RSL, DSL, точки вариантности MX, UML, сообщения, современные ЯП.	Windows API, API Viewer, Demral, Visual Works – DLL, IDE Eclipse, Rational Rose, Jaspect, DMS	Приложение, член семейства, семейство систем

Объектно–ориентированный подход (ООП). Он включает в себя стратегию разработки, в рамках которой разработчики системы вместо операций и функций мыслят объектами. Объект – это предмет внешнего мира, некоторая сущность, пребывающая в различных состояниях и имеющая множество операций. Операции задают сервисы, предоставляемые объектам для выполнения определенных вычислений, а состояние – это набор атрибутов объекта, поведение которых изменяет это состояние [35, 36, 47].

Объекты группируются в класс, который служит шаблоном для включения описания всех атрибутов и операций, связанных с объектами данного класса.

Класс - по сути дела это тип объекта, в котором кроме данных (свойств, параметров объекта), содержатся методы их обработки. Объект относительно

своего класса является *экземпляром*. Класс определяет структуру и функциональность (*поведение*), одинаковую для всех экземпляров данного класса. Один класс отличается от других набором поддерживаемых интерфейсов, т.е. набором сообщений, которые посылаются объектам.

Экземпляр обладает *состоянием*, меняющимся в соответствии с поведением, заданным классом. Объекты могут обмениваться между собой *сообщениями,* по . которым запускается соответствующий ему метод**.** Если метод для обработки сообщения не найден, то сообщение перенаправляется объекту-*предку*.

Основные принципы ООП такие:

наследование – механизм установления отношений «потомок–предок» (возможность порождать один класс от другого с сохранением всех свойств и методов класса-предка);

инкапсуляция – свойство сокрытия реализации класса;

абстракция – описание взаимодействия объектов в терминах сообщений/событий;

полиморфизм – возможность замены объектов одного объекта другим объектом со сходной структурой.

Многие современные языки (Smalltalk, C++, Java, Python, PHP, Object Pascal, Delphi и др.) включили в себя понятие класса и другие принципы ООП.

Программная система в ООП – это взаимодействующие объекты, имеющие собственное локальное состояние и набор операций для определения состояний объектов. Объекты могут инкапсулировать информацию о представлении состояний и ограничивают к ним доступ.

Проектирование ПС заключается в проектировании классов объектов и взаимоотношений между ними. Оно выполняется с помощью следующих процессов:

анализ – создание объектной модели (ОМ) ПрО, в которой объекты отражают реальные ее сущности и операции над ними;

проектирование – уточнение ОМ с учетом описания требований для реализации конкретных задач системы;

программирование – реализация ОМ средствами C++, JAVA и др.

сопровождение – использование, развитие системы, внесение изменений, как в состав объектов, так и в методы их реализации;

модификация ПС – изменение системы в процессе ее сопровождения путем добавления новых функциональных возможностей, интерфейсов и операций.

Приведенные процессы могут выполняться итерационно друг за другом и с возвратом к предыдущему. На каждом из них может использоваться одна и та же система нотаций.

Переход к следующему процессу приводит к усовершенствованию результатов предыдущего процесса путем более детальной реализации ранее определенных классов объектов и добавления новых.

Результат процесса анализа ЖЦ – модель ПрО и набор других моделей (модель архитектуры, модель окружения и использования), полученных на процессах ЖЦ. Они отображают связи между объектами, их состояния и набор операций для динамического изменения состояния других объектов, а также взаимоотношения со средой. Объекты инкапсулируют информацию об их состоянии и ограничивают к ним доступ. Модель окружения и использования

системы – это две взаимно дополняющие друг друга модели связи системы со средой.

Модель окружения системы – это статическая модель, которая описывает другие системы из пространства разрабатываемого ПО, а модель использования системы – это динамическая модель, которая определяет взаимодействие системы со своей средой.

Взаимодействие определяется последовательностью запросов к сервисам объектов и реакцией системы после выполнения запроса. Когда взаимодействие между объектами проектируемой системы и ее окружением определены, эти данные используются для разработки архитектуры системы, которая может разрабатываться с помощью объектов, созданных в предыдущих проектах.

Основные модели системной архитектуры такие:

– *статическая модель* описывает статическую структуру системы в терминах классов объектов и взаимоотношений между ними (обобщения, расширения, использования и др.);

– *динамическая модель* представляет динамическую структуру системы и взаимодействие между объектами системы в процессе выполнения.

Результат проектирования – это ПС, в которой все необходимые объекты определены статически или динамически с помощью классов и соответствующих методов их реализации. Полученная объектно-ориентированная система проверяется на показатели качества на основе результатов тестирования и сбора данных об ошибках. Такую систему можно рассматривать как совокупность автономных и независимых объектов. Изменение метода реализации объекта или добавление новых функций не влияет на другие объекты системы. Объекты могут быть повторно используемыми.

Компонентно-ориентированная разработка (Component-Based Development – CBD). Это самостоятельная методология, появление которой наряду с модульной технологией связано с такими тенденциями в программировании [132, 133]:

– переход к распределенным вычислениям и обработке информации, при которых различные вычислительные процессы выполняются на разных компьютерах и средах, для их разработки применяются разные ЯП и механизмы взаимодействия между собою отдельных программных объектов;

– широкое распространение готовых компонентов разного назначения и их применение в процессе разработки новых ПС при условии соответствия определенным требованиям к структуре, интерфейсам и правилам интеграции;

– визуализация типовой структуры объектов и их интерфейсов, проверка возможности их применения в целях генерации и связывания в единое целое.

Данные тенденции основываются на унификации и стандартизации программных компонентов, их выбора и построения из них сложных программных продуктов.

Особенность компонентной разработки – применение компонентов на всех процессах ЖЦ. Иными словами, все элементы и объекты компонентной разработки – это модули, компоненты, программы, данные и т.п., представляемые программными компонентами.

Использование компонентов в программировании имеет ряд преимуществ:

– снижение затрат на разработку ПС;

– сокращение времени разработки отдельных компонентов;

– улучшение показателей качества и надежности программ из компонентов;

– упрощение процедур и снижение затрат на поддержку и сопровождение программных продуктов;

– создание определенной инфраструктуры в разработке ПС, основанной на унификации и стандартизации компонентов в целях практического применения;

– создание рыночных компонентов за счет их стандартизации и ориентации на повторное использование.

С формальной точки зрения программный компонент (component) – это независимый от ЯП самостоятельно реализованный программный объект, доступ к которому возможен лишь с помощью интерфейсов, определяющих его функции и порядок обращения к его операциям.

В компонентной CBD разработке обеспечен механизм взаимодействия компонентов через контракт, в котором определены правила взаимодействия с другими компонентами, входящими в контейнер или каркас.

Интеграция (сборка) компонентов и их развертывание с среде не зависят от ЖЦ разработки компонентов. Замена любого компонента новым компонентом не должна приводить к перекомпиляции или перенастройке связей в ПС.

Интерфейс компонента может быть определен в виде спецификации точек доступа к компоненту, используя которые, клиент получает сервис в клиент-серверной среде. Так как интерфейс не предоставляет реализацию операций, то можно изменять реализацию без изменения интерфейса и таким образом улучшать исполнительные или функциональные свойства компонента без перестройки ПС в целом, а также прибавлять новые интерфейсы (и реализацию) без изменения существующей реализации ПС.

Семантика интерфейса может быть представлена с помощью контрактов, которые определяют внешние ограничения и поддерживают компонент-инвариант. Кроме того, для каждой операции компонента контракт может определять ограничения, которые должны быть выполнены клиентом перед вызовом операции (предусловие), и условия, которое компонент выполнит после завершения операции (постусловие). Предусловие и постусловие определяют спецификацию поведения компонента и зависят от состояния, которое поддерживает компонент, а также интерфейсов, связанных с набором инвариантов.

Контракты и интерфейс связаны между собою, но содержат разные сущности. Интерфейс представляет собою коллекцию операций или функциональных свойств спецификации сервисов, которые поддерживает компонент. Контракт задает описание поведения компонента, нацеленное на взаимодействие с другими компонентами, и отражает семантику функциональных свойств компонента.

Таким образом, спецификация семантики компонента определяет его интерфейс и ограничения. Каждый интерфейс состоит из набора операций (сервисов, которые он предлагает или требует). Поведение компонента определяют также и нефункциональные свойства, которые обеспечивают принципы взаимодействия с другими компонентами или средой, а также механизмы защиты и безопасности выполнения.

Компонент может использоваться многократно, как КПИ, к которым относятся формализованные артефакты деятельности разработчиков ПС при реализации некоторых функций и которые могут применяться в новых разработках.

Артефактом может быть реальная порция информации, создаваемая при выполнении деятельности, связанной с разработкой ПС систем различного назначения, в частности промежуточные продукты процесса разработки ПС: требования, постановки задач, заготовки программ, программы, комплексы, системы и т.п. Ими также могут быть: спецификации, модели, архитектура, каркас (framework) и т.п., а также готовые ПС.

Для объединения разных видов компонентов применяются шаблоны развертывания, которые сохраняются в скрытой части абстракции компонента. К спецификации компонента могут прибавляться новые шаблоны интеграции или изменяться старые. В некоторых классах КПИ параметры интеграции в новую среду включаются в интерфейс компонента, что сужает круг задач компонента для его повторного использования. Интеграция компонентов в архитектуру целевой ПС в CBSE состоит из нескольких типичных методов объединения, среди которых наибольшее распространение получили паттерны, каркасы и контейнеры [8, 26].

Паттерн – абстракция, которая содержит описание взаимодействий совокупности объектов, ролей участников и их ответственности. Он практически определяет повторяемое решение в проблеме объединения КПИ в программную структуру. Для каждого объединения определяется взаимодействие объектов при совместной кооперативной деятельности с заданием абстрактных участников, их ролей, распределения полномочий (или обязанностей). Паттерны классифицируются по трем уровням абстракции:

– высокий уровень связан с глобальными свойствами и архитектурой системы, скомпонованной из компонентов в виде архитектурного паттерна, который охватывает общую структуру и организацию ПС в виде набора подсистем с определением их ролей и отношений между ними;

– средний уровень абстракции уточняет структуру, поведение отдельных подсистем и компонентов ПС, а также взаимодействие между ними;

– нижний уровень представляет собой абстракцию определенного вида (например, объединение компонентов по аналогии), которая зависит от выбранной парадигмы и ЯП.

Кроме паттернов, в сборочной стратегии широко используется *каркас* – типовая повторно возникающая ситуация на уровне модели ПС, которая определяет структуру проекта и имеет недоопределенные элементы, обозначенные пустыми слотами для расположения в них новых определенных компонентов. В этом смысле каркас становится КПИ со свойством экземпляризации и представленный высокоуровневой абстракциею проекта ПС, в которой отделены функции компонентов от задач управления ими. В бизнес-функциях каркас задает надежное управление ими. Каркас объединяет множество взаимодействующих между собою объектов в некоторую интегрированную среду, предназначенную для решения заданной конечной цели типа “белый” или “черный ящик”.

Каркас типа “белый ящик” включает в себя абстрактные классы для представления цели объекта и его интерфейсов. При реализации эти классы наследуются в конкретные классы с указанием соответствующих методов реализации, как это принято в ООП. Каркас типа «черный ящик» характеризуется тем, что в видимой, информационной, части имеется описание точек входа и выхода. Через эти точки можно входить и выходить из компонента не обязательно через конечный оператор. Таким образом, каркас определяет контекст интеграции

отдельно построенных программных частей. Физически он реализуется с помощью одного или больше паттернов, которые в свою очередь выступают в роли инструкций по реализации проектных решений.

Заполненный компонентами каркас становится *контейнером* и сам может быть компонентом ПС, обеспечивает инкапсуляцию компонентов и доступ к компонентам через каркас, который определяет возможные варианты наполнения контейнера. В нем заданы функции, порядок их выполнения, вызываемые события, сервис, интерфейс, необходимый для обращения к провайдеру сервиса. При этом контракты выражают общие принципы и определяют спецификацию отношений между конкретными компонентами, которые могут отличаться от спецификации компонентов как частей композиции.

Создание компонентной системы начинается с построения *компонентной модели,* включающей в себя проектные решения по композиции программных компонентов, разные типы паттернов, связи между ними, способы взаимодействия (интерфейсы) и операции развертки ПС в среде функционирования.

Схема модели компонентной системы представлена на рис. 1.3 [6].

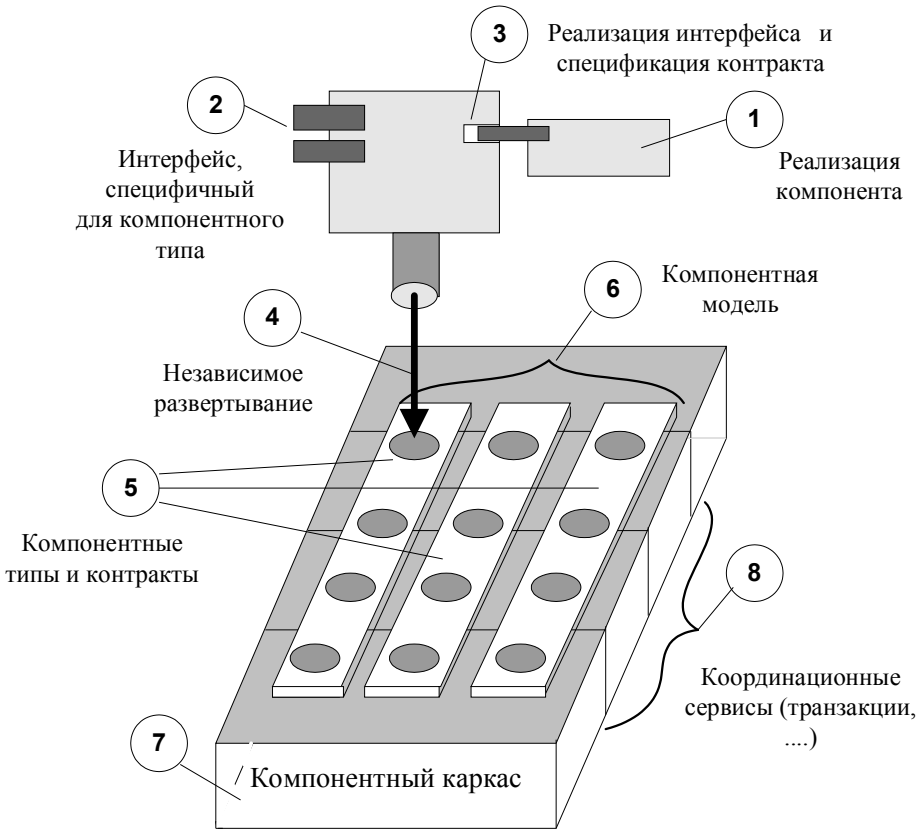


Рис. 1.3. Модель компонентной системы

Здесь: 1 – компонент, имеющий функциональность; 2 – один или более интерфейсов, описанных внутри контракта; 3 – контрактное взаимодействие компонентов друг с другом; 4 – развертывание компонентов во время выполнения в заданной среде; 5 – разные типы компонентов системы; 6 – компонентная

модель из множества типов компонентов, их интерфейсов; 7 – спецификаций шаблонов взаимодействия этих компонентов; 8 – компонентный каркас обеспечивает выполнение компонентной модели, множество сервисов.

Сами программные компоненты в данной модели могут быть изготовлены с помощью любой методологии разработки ПС с учетом требований и условий их применения при построении из них новых систем.

Порождающее программирование. Данное программирование ориентировано на создание семейств программных систем путем трансформации описания членов семейства спецификациями высокого уровня типа языка DSL (Domain Specific Language) с отображением специфики предметной области в терминах соответствующей базовой терминологии. Результат последовательной трансформации генерируется и собирается в выходной код [142, 204].

Его основа – общая порождающая доменная модель GDM (Generative Domain Model) семейства систем. В ней содержатся:

– описание членов семейства в языках (DSL, RSL, CSP, Clear и др.);

– компоненты реализации функций членов семейства и их характеристики;

– знания о конфигурации (Configuration knowledge) в виде правил конструирования, вариантов оптимизации, сочетания и зависимости между членами семейства и др.;

– модель общих, изменяемых характеристик членов семейства и их взаимодействия;

– механизмы обеспечения изменчивости, синхронизации и безопасности как компонентов, так и членов семейства средствами описания связывающих аспектов.

Знания о конфигурации упакованы в виде абстракций общего и специального назначения и хранятся в активной библиотеке многократного использования. Там же хранятся и производственные знания о средствах тестирования, измерения, планирования и оценки отдельных компонентов и членов семейства. В наборе этих знаний определяются задачи ПрО и подходы к ее описанию и реализации.

Каждая модель члена семейства представляется в некотором конкретном синтаксисе, например DSL, XML, UML и др., которая трансформируется к выходному целевому представлению по следующей схеме (рис.1.4) [185]:

– преобразование исходной модели к промежуточному (объектному) виду;

– преобразование исходной модели в выходную (целевую);

– преобразование целевой модели в конкретный код.

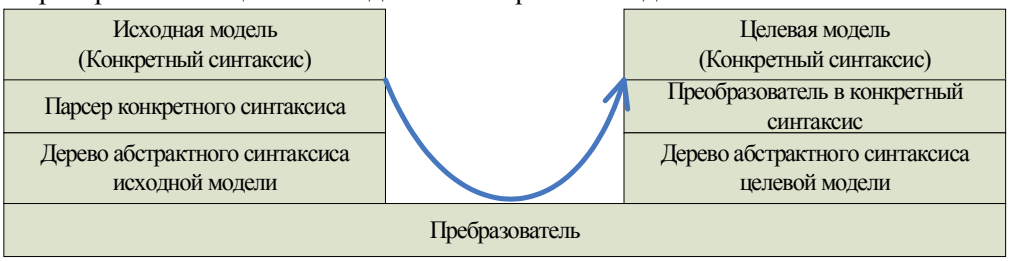


Рис. 1.4. Схема преобразования модели системы

Генерация выходного кода осуществляется такими способами:

– итеративным, при котором перечисляются все узлы исходной модели и они преобразуются в соответствующее представление целевой модели;

– шаблонным, позволяющим вставлять макросы, т.е. участки кода в любую

часть заданного шаблона;

– поисковым, состоящим в нахождении места в исходной модели и ее преобразования.

Шаблон – это некоторое регулярное выражение концептуальной модели системы. Для его описания создается специальный язык шаблона, основанный на исходном языке программирования и включающий в себя критерии поиска сложных совпадений в исходной модели.

Цель генерации кода – обеспечение возможности его настройки на целевую платформу для получения выходного кода осуществляется в два этапа. Первый этап – обеспечивает чтение конфигурации, по которой создается соответствующий генератор текущего преобразования, второй – использует созданный генератор для выполнения необходимых преобразований модели.

Генерация исполняемого кода осуществляется посредством:

– использования шаблонов проектирования;

– сохранения сгенерированного и исправленного вручную (рукописного кода) в отдельных файлах;

– дизайна архитектуры системы, в которой определены сгенерированные артефакты;

– использования гибких методик склеивания сгенерированных и рукописных частей кода.

Интеграция (сборка) сгенерированного кода и рукописного состоит в написании вручную программ для незаполненных участков шаблона путем вставки.

Моделирование в UML. Основу составляет стандартный язык UML lkz моделирования программ. Язык предназначен для визуализации, спецификации, конструирования и документирования артефактов ПС. Описание программ в этом языке осуществляется с помощью стандартных диаграмм, словаря и механизмов расширения.

Словарь UML содержит три категории элементов: предметы (things), связи (relationships) и диаграммы (diagrams). Предметы имеют такие разновидности: структурные (use case, class, active class, interface, component, collobaration, node); поведенческие (interaction, state machine); групповые (package, model, subsystem, framework); аннотационные (note). Связи различаются как зависимые (dependency), ассоциативные (association) и те, которые обеспечивают обобщение (generalization).

В состав диаграмм входят: class, use case, object, sequence, component, colloboration, statechart, activity, deployment. Они используются для графического изображения соответствующих элементов ПС. Графические нотации – это наглядные структуры, которые задают представление предметов, связей и механизмов расширения языка UML. Механизмы расширения предназначены для уточнения синтаксиса и семантики языка и включаютв себя: стереотипы и констрейны. Метод UML включает в себя:

– формальную метамодель для описания синтаксиса и семантики, а также для унификации разных аспектов проектирования объектных приложений;

– совокупный набор нотаций графического представления объектов проектирования;

– множество разнообразных идиом для отображения конкретной модели ПС.

В рамках UML развивается модельно-ориентированный подход, обеспечивающий трансформацию исполняемых моделей системы, основанную на выполнении таких действий:

– создание среды разработки ПС в UML с помощью сценариев и шаблонов проектирования;

– проектирование модели интеграции ПС и системы образцов средствами UML;

– выбор метрик и методов оценивания показателей качества объектов в проектируемой системе;

– модификация моделей с учетом функциональных и нефункциональных требований к системе;

– изменение системы путем повторного использования КПИ и методик преобразования передаваемых данных;

– проверка поведения системы осуществляется после выполнения разного рода преобразований;

– обеспечение целостности модели путем внесения изменений в модель.

Иными словами, модели в UML, а также модели, спроектированные на языке SDL, трансформируются специальными утилитами для исполняемого кода.

Инсерционное программирование. Термин «инсерционное» происходит от английского *insert* – вставить, поместить, погрузить. Имеется в виду вставка в программу или среду объекта, называемого агентом. Агент обладает поведением и погружается в информационную среду, которая меняет ее поведение относительно других агентов этой среды [147, 148]. Описание инсерционной программы состоит в определении функции погружения, начального состояния среды и агента, погруженного в эту среду. Поведение среды меняется в результате воздействия на расположенные в ней объекты взаимодействующих агентов.

Основа инсерционного программирования – модель поведения агентов в средах, базирующихся на понятиях транзиционной системы, теории взаимодействующих процессов и отношений бисимуляционной эквивалентности агентов относительно среды. В отличие от агентного программирования, инсерционное программирование ориентируется исключительно на поведение агентов.

Представление состояний агентов и сред как структур данных, а также функций погружения осуществляется средствами алгебраического программирования APS [147]. Инсерционная программа описывается на языке AL и имеет три уровня:

– описание поведения инициализированной многоуровневой среды с расположенными в ней агентами;

– задание функции, определяющей отношения переходов агентов;

– описание ядра функции погружения или развертывания агентной программы.

Инсерционными программами моделируются реальные системы с недетерминированным поведением агентов и сред. Реализация систем инсерционного программирования требует применения программ-симуляторов, а не интерпретаторов, а также постановки целей для получения конкретных результатов.

Таким образом, в современных средах программирования решается задача взаимодействия объектов через контракты в компонентном программировании, через шаблоны проектирования в порождающем программировании или через

механизмы погружения в инсерционном программировании. Эта особенность соответствует реализации интерфейса объектов посредством их взаимодействия.

1.2. ОСНОВНЫЕ ЭЛЕМЕНТЫ И ЗАДАЧИ СБОРОЧНОГО ПРОГРАММИРОВАНИЯ

Процесс сборки любых изделий характеризуется: комплектующими деталями и узлами, схемой сборки (взаимосвязями отдельных компонентов и правилами взаимодействия), сборочным конвейером (технологией сборки). Конкретизируем эти понятия с точки зрения метода сборочного программирования. При этом будем предполагать, что используются только готовые программные продукты [125, 133, 140].

Комплектующие элементы. Комплектующими в методе сборочного программирования являются простые программные элементы (модули, объекты, компоненты, сервисы и др. (табл.1.3).

Таблица 1.3

Элемент сборки	**Описание элемента**	**Схема взаимодействия**	**Представление, хранение**	**Результат композиции**
Процедура, подпрограмма, функция	Идентификатор	Непосредственное обращение, оператор вызова	Библиотеки подпрограмм и функций	Программа
Модуль	Паспорт модуля связи	Вызов модулей, интеграция модулей	Банк, библиотеки модулей	Программа с модульной структурой
Объект	Описание класса	Создание экземпляров классов, вызов методов	Библиотеки классов	Объектно-ориентированная программа
Компонент	Описание логики (бизнес), интерфейсов (APL, IDL), схемы развертки	Удаленный вызов в компонентных моделях (COM, CORBA, OSF, …)	Репозитарий компонентов, серверы и контейнеры компонентов	Распределенное компонентно-ориентированное приложение
Сервис	Описание бизнес-логики и интерфейсов сервиса (XML, WSDL, …)	Удаленный вызов (RPS, HTTP, SOAR, …)	Индексация и каталогизация сервисов (XML, UDDI, …)	Распределенное сервисно-ориентированное приложение

Из них собираются программные объекты различной сложности и степени готовности: программы, комплексы, пакеты прикладных программ, программные системы и т.д.

Модуль – независимая функциональная часть программы, к которой можно обращаться как к самостоятельной единице через внешний интерфейс.

Объект – базовое понятие в объектно-ориентированном программировании, которое обладает свойствами наследования, инкапсуляции и полиморфизма. Объединяет данные и операции (методы). Объекты взаимодействуют между собой через сообщения. Объекты с общими свойствами и методами образуют класс, являясь в нем экземплярами класса. В языке C++ имеется несколько библиотек классов общего применения.

Компонент – программный объект, который реализует некоторую функциональность и является базовым понятием компонентного программирования и компонентно-ориентированной разработки (component-based development – CBD). Основная форма представления компонента – каркас и контейнер. *Каркас* – высокоуровневая абстракция, в которой функции отделены от задач управления.

Контейнер – оболочка, внутри которой реализованы функции в виде экземпляров компонентов, обеспечивает взаимодействие с сервером через стандартные интерфейсы (function, home interface). Экземпляры обращаются друг к другу через системные сервисы данного контейнера или другого.

Сервис – это программный ресурс, который реализует некоторую функцию, в том числе и бизнес функцию. Содержит независимый интерфейс с другими сервисами и ресурсами. Веб-сервис обеспечивает реализацию задач интеграции приложений разной природы и используется как провайдер. Совокупность взаимодействующих сервисов, веб-сервиса и их интерфейсов образует сервисно-ориентированную архитектуру (service oriented architecture), доступ к которым происходит через веб-языки и протоколы.

Результатом сборки может быть любой программный объект, за исключением элементарного элемента – модуля, компонента, сервиса и др. Однако для практического применения выбирается несколько базовых типов программных объектов, рассматривающихся как готовые компоненты, и результат использования метода сборочного программирования.

Для технологичности сборки все объекты должны иметь паспорта, содержащие данные, необходимые для информационного сопряжения и организации совместного функционирования в рамках программного объекта более сложной структуры. Важное условие сборки состоит в наличии большого числа разнообразных комплектующих, т.е. ПС, обеспечивающих решение широкого спектра задач из различных предметных областей.

Схема сборки. Под схемой сборки программных объектов понимается схема их взаимодействия, определяющаяся непосредственными обращениями к компонентам (типа оператора CALL) или последовательностью их выполнения. При этом взаимодействие каждой пары объектов зависит от совместного использования данных. В общем случае схема сборки состоит из совокупности моделей, отражающих различные типы связей между компонентами: передача управления, обмен информацией, условия совместного функционирования и т. д.

Операции сборки выполняются согласно паспортам объектов и правилам сопряжения. Информация в паспортах должна быть систематизирована и выделена в такие отдельные группы: передаваемые данные и их типы, вызываемые объекты, совместно используемые файлы и т.д. Правила сопряжения определяют совместимость объединяемых объектов и содержат описание функций, необходимых для согласования различных их характеристик, представленных в паспортах.

Сборочный конвейер. Процесс сборки может производиться ручным, автоматизированным и автоматическим способами. Как правило, последний способ практически неосуществим, что связано с недостаточно формальным определением программных объектов и их интерфейсов. Ручной способ нецелесообразен, так как сборка готовых компонентов представляет собой большой объем действий, носящих скорее рутинный, чем творческий характер. Наиболее приемлемый – автоматизированный способ сборки в среде системы, которая по заданным спе-

цификациям (моделям) программ осуществляет сборку с помощью стандартных правил сопряжения под управлением человека. Средства, поддерживающие данный способ сборки, называются инструментальными средствами сборочного программирования. К ним относятся средства комплексирования (объединения компонентов в более сложный объект); средства интерфейсов (реализация сопряжения объектов согласно их паспортам и стандартным правилам связи); средства описания и использования моделей сборочного программирования (совокупность моделей сборки различных программных объектов).

Из рассмотренной схемы сборки выделим задачи сборочного программирования и условия его применения:

– выбор компонентов из числа готовых программных объектов для обеспечения процесса решения класса задач из определенной предметной области;

– проектирование структуры (моделей) создаваемого объекта, элементами которого являются готовые программные элементы, определенные на множестве данных выбранной предметной области;

– сборка согласно моделям программного продукта, реализующего функции, соответствующие целям и задачам автоматизируемой предметной области.

Необходимые условия применения данного метода программирования включают в себя:

– наличие большого числа разнообразных программных продуктов, как объектов сборки;

– паспортизация программных объектов сборки;

– наличие достаточно полного набора стандартных правил сопряжения и алгоритмов их реализации и средств автоматизации процесса сборки;

– создание технологий применения разработанных объектов для использования в более сложных программных продуктах.

Последнее условие означает, что должны существовать определенные формы представления ПС как знаний о предметных областях, универсальные с точки зрения проектирования и разработки программных систем.

Основной вопрос сборки – это реализация *интерфейсов* между отдельными модулями и/или компонентами, обеспечивающими их «стыковку» или связь.

1.3. ОПРЕДЕЛЕНИЕ ИНТЕРФЕЙСА ПРОГРАММНЫХ ОБЪЕКТОВ

Общее определение. *Интерфейс* – это связь двух отдельных сущностей. Виды интерфейсов: программные, аппаратные, языковые, пользовательские, цифровые и т.п. Программный (API) и/или аппаратный интерфейс (port) – это способы преобразования входных/выходных данных во время объединения компьютера с периферийным оборудованием. Языковый интерфейс определяет константы, переменные, параметры и структуры данных программы в некотором ЯП, которые передаются другим программам или модулям [116, 127, 132].

Иными словами, термин интерфейс в программировании олицетворяет собой набор операций с параметрами, обеспечивающими определение видов услуг и способов их передачи от одного программного объекта другому. На начальном процессе программирования в роли интерфейса выступали операторы обращения к процедурам и функциям программы с соответствующими формальными параметрами. Программы, процедуры и функции записывались в ЯП, а операторы обращения к ним включали в себя имена вызываемых процедур и функций и

список фактических параметров, задающих значения формальным параметрам для получения результатов. Последовательность и число формальных параметров соответствовало фактическим параметрам. Выполнение функции в среде программы на одном ЯП не вызывало проблем, так как типы данных параметров совпадали.

Когда один из элементов – программа, процедура или функция записаны на разных ЯП и, кроме того, если они располагаются на разных компьютерах, то возникают проблемы неоднородности представления типов данных параметров в этих ЯП, структур памяти платформ компьютеров, компиляторов и операционных сред их выполнения.

Понятие интерфейса, как самостоятельного объекта, сформировалось в связи со сборкой или объединением *разноязыковых* программ и модулей в одну монолитную систему на больших ЭВМ (mainframes), память которых позволяла получать программы соответственно до 100–200тыс. команд и более.

Интерфейс играет роль посредника по передаче информации от вызываемого модуля вызывающему, записанные на разных ЯП. В нем дано описание формальных и фактических параметров, произведена проверка соответствия передаваемых параметров (количества и порядка расположения), а также типов данных параметров. Если типы данных параметров оказывались не релевантными (например, передается целое, а результат функции – вещественное, или наоборот), то в посреднике необходимо произвести прямое и обратное преобразование передаваемых типов данных с учетом структуры памяти их компьютеров.

На рис.1.5. приведена схема некоторой программы С, в которой содержатся два оператора вызова – Call A ( ) и Call B ( ) с некоторыми параметрами, которые первоначально передаются интерфейсным посредникам A’ и B’. В них проводится преобразование данных к нужному виду модулей A, B и их передача для выполнения. После обработки этих данных результаты выполнения проходят данную схему в обратном порядке. Данные, полученные в модулях A и B, преобразуются в интерфейсных посредниках A’ и B’ к соответствующему виду их описания в программе С.

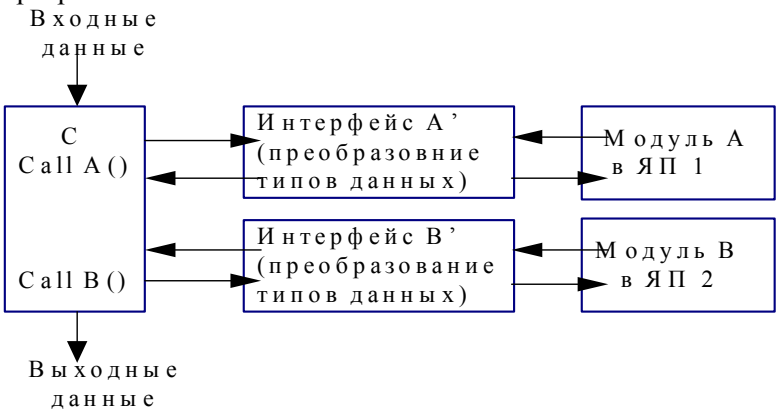


Рис. 1.5. Схема вызова модулей A и B через интерфейсы A` и B`

Интерфейс, как способ передачи информации, используется в разных видах программирования с некоторыми особенностями и различиями. Приведем их.

Интерфейс и ООП

Главный элемент ООП – класс. Интерфейс описывает поведение класса объектов. Класс поддерживает несколько интерфейсов, которые содержат операции. *Интерфейс* – это множество операций, которые используются для определения услуг класса или компонента (рис.1.6), именует это множество операций или определяет их сигнатуру и результирующие действия.

Класс	
Внешнее представление	**Внутреннее представление**
Интерфейсные операции: – публичные, доступные всем клиентам; – защищенные, доступные классу и подклассу; – частные, доступные классу.	Реализация операций для определения поведения объектов класса.

Рис.1.6. Структура представления класса и интерфейса

Если интерфейс реализуется с помощью класса, то он наследует все операции интерфейса. Одни и те же операции могут появляться в различных интерфейсах. Если их сигнатуры совпадают, то они задают одну и ту же операцию, соответствующую поведению системы.

Операции и сигналы могут быть связаны отношениями обобщения. Интерфейс – потомок, включающий в себя все операции и сигналы своих предков, может добавлять собственные. Интерфейс-потомок наследует все операции прямого предка. Его реализацию можно рассматривать как наследование поведения. Класс может реализовывать другой класс через интерфейс.

Интерфейс в современных средах и сетях. Появление разных типов компонентов и необходимость их объединения в более сложные структуры в локальных и глобальных сетях привело к определению понятия интерфейса как механизма удаленного вызова компонентов, располагающихся в разных узлах сети или среды, который реализуется через передачу сообщений.

Сеть строится на основе стандартной семиуровневой модели открытых систем OSI (Open Systems Interconnection). Объекты этих уровней связываются между собой по горизонтали и вертикали. Запросы от приложений поступают на уровень представления данных этой модели для их кодирования (перекодирования) к виду используемой платформы. Открытые, распределенные системы предоставляют разным приложениям услуги по обработке интерфейсов, управлению удаленными объектами, обслуживанию очередей и запросов и т.п.

Интерфейс программных объектов может задаваться одним из способов:

– вызов удаленных процедур RPC (Remote Procedure Call) в системах ONC SUN, OSF DSE [5, 133];

– связывание распределенных объектов и документов в системе DCOM;

– взаимодействие объектов, интерфейс которых описывается в языке IDL (Interface Definition Language) и обрабатывается брокером объектных запросов ORB (Object Request Broker) в системе CORBA [166];

– удаленный вызов RMI (Remote Methods Invocation) в системе JAVA для

объектов, описанных в языках JAVA, Паскаль, C++, C# [25, 191] и др.

RPC– вызов задает интерфейс удаленных компонентов в языках высокого или низкого уровней. На языке высокого уровня RPC–вызов передается с помощью сообщения, содержащего параметры для удаленной процедуры. В языке низкого уровня в сообщении указывается более подробная информация: тип протокола, размер буфера данных и т.п.

Взаимосвязь одного процесса с удаленно расположенным от него другим процессом (например, сервером) на другом компьютере можно выполнить также протоколами UDP или TCP/IP. В них посредник stub передает интерфейс от компонентов клиента серверу с параметрами, которые обеспечивают выполнение удаленной серверной процедуры.

Механизм посылки запроса в системе CORBA – это описание запроса в языке IDL для посредника stub/skeleton и передача его протоколом IIOP или GIOP брокеру ORB. Запрос обрабатывается в среде брокера через stub/ skeleton сервером объектного сервиса (Common Object Services) или общими средствами (Common Facilities) системы CORBA. Брокер реализован в разных распределенных системах (CORBA, COM, SOM, Nextstep и др.) и предназначен для обеспечения взаимодействия объектов, которые располагаются в разных сетевых средах.

Вызов метода RMI в системе JAVA выполняет виртуальная машина VM (virtual machine), которая интерпретирует byte–коды компонентов, созданных разными системами программирования с ЯП (JAVA, Pascal, C++) на компьютерах. Функции RMI в этой системе аналогичны функциям брокера ORB.

Описание интерфейса объектов в распределенных средах

В промежуточной среде рассмотренных систем реализуется два способа связывания объектов: на уровне ЯП через интерфейсы прикладного программирования и компиляторов IDL, которые генерируют клиентские и серверные stub. Интерфейсы определяются в IDL или APL, а динамический вызов осуществляет ORB для объекта-клиента и объекта-сервера. Эти интерфейсы имеют отдельную реализацию на ЯП и доступны из программ, записанных в разных ЯП. Компиляторы IDL, как часть промежуточного слоя, сами реализуют связывание с соответствующим языком программирования ЯП механизма ссылок.

Интерфейс в IDL или в API обеспечивает взаимодействие компонентов на основе описания их формальных и фактических параметров и порядка задания операций передачи параметров для получения результатов. Это описание – ничто иное, как спецификация интерфейсного посредника двух разноязыковых объектов, которые взаимодействуют между собой через механизм вызова интерфейса и могут выполняться на разных процессах сервера или клиента.

В функции интерфейсного посредника клиента входят:

– подготовка внешних параметров клиента для обращения к сервису сервера,

– посылка параметров компонента серверу, его выполнение для получения результата или выдача сведений об ошибках.

Общие функции интерфейсного посредника сервера состоят в следующем:

– получение сообщения от клиента, запуск удаленной процедуры, вычисление результата и подготовка его для передачи клиенту;

– возврат результата клиенту в параметрах сообщения, уничтожение удаленной процедуры и др.

Преобразование интерфейсных параметров

Программы, расположенные на разных типах компьютеров, передают друг другу данные через протоколы, форматы которых преобразуются к формату данных принимающей серверной платформы (так называемый маршалинг данных) с учетом порядка и стратегии выравнивания, принятой на этой платформе. Демаршалинг данных – это обратное преобразование данных (полученного результата) к виду передавшей клиентской программы. Если среди передаваемых параметров оператора вызова содержатся нерелевантные типы или структуры данных, которые не соответствуют параметрам вызванного объекта, то производится их преобразование. Стандарт ISO/IEC 11404–2007 регламентирует описание общих типов данных, которые могут использоваться любыми современными ЯП [133, 134].

К средствам преобразования данных и их форматов также относятся:

– стандарты кодировки данных (XDR – eXternal Data Representation, CDR – Common Representation Data [4]), NDR – Net Data Representation) и методы их преобразования;

– ЯП и механизмы обращения компонентов друг к другу;

– языки описания интерфейсов – RPC, IDL и RMI.

На каждой платформе компьютера используются соглашения о кодировке символов (например, ASCII), о форматах целых чисел и чисел с плавающей точкой (например, IEEE, VAX и др.). Для представления целых, как правило, используется дополнительный код, а для типов float и double – стандарт ANSI / IEEE и др.

Порядок расположения байтов зависит от структуры платформы (Big Endian или Little Endian) и от старшего к младшему байту и от младшего байта к старшему. Процессоры UltraSPARC и PowerPC поддерживают обе возможности. При передаче данных с одной платформы на другую учитывается возможное несовпадение порядка байтов. Маршалинг данных поддерживается несколькими стандартами, некоторые из них рассмотрим ниже.

XDR-стандарт содержит язык описания структур данных произвольной сложности и средства преобразования данных, передаваемых на платформы (Sun, VAX, IBM и др.). Программы, написанные в ЯП, могут использовать данные в XDR-формате, несмотря на то, что компиляторы выравнивают эти данные по-разному в памяти машины. В XDR-стандарте целые числа с порядком "от младшего" приводятся к порядку байтов от «старшего» и обратно. Преобразование данных – это кодирование (code) или декодирование (decode) простых и сложных типов данных. Кодирование – это преобразование из локального представления в XDR-представление и запись в XDR-блок. Декодирование – это чтение данных из XDR-блока и преобразование в локальное представление заданной платформы. Выравнивание данных – это размещение значений базовых типов с адреса, кратного действительному размеру в байтах (2, 4, 8, 16). Границы данных выравниваются по наибольшей длине (например, 16). Системные процедуры оптимизируют расположение полей памяти под сложные структуры данных и преобразуют их к формату принимающей платформы. Обработанные данные декодируются обратно к виду формата платформы, отправившей эти данные.

CDR-стандарт в среде CORBA обеспечивает преобразование данных в форматы передающей и принимающей платформ. Маршалинг данных выполняет

интерпретатор Type Code и брокер ORB. Процедуры преобразования сложных типов включают в себя:

– дополнительные коды для представления целых чисел и чисел с плавающей точкой (стандарт ANSI / IEEE);

– схему выравнивания значений базовых типов в среде компилятора;

– базовые типы (signed и unsigned) языка IDL и плавающий тип двойной точности и др.

Преобразование данных выполняются процедурами encoder ( ) и decoder ( ) интерпретатора TypeCode, который используют базовые примитивы при выравнивании информации и помещении ее в буфер. Для сложного типа вычисляется размер и границы выравнивания, а также их размещение в таблице с индексами значений, используемых при инициализации ORB.

XML-стандарт предназначен для устранения неоднородности во взаимосвязях компонентов в разных ЯП с помощью XML-формата данных, учитывающего разные платформ и среды. Промежуточные среды (CORBA, DCOM, JAVA и др.) имеют в своем составе специальные функции, аналогичные XML и обеспечивают взаимосвязь разноязыковых программ. XML имеет различные системные поддержки: браузер – Internet Explorer для визуализации документов, объектная модель DOM (Document Object Model) для отображения XML–документов и интерфейс IDL в системе CORBA. Для перехода ПС к XML осуществляется переформатирование данных системы в формат XML и обратно.

Иными словами, XML – средство представления объектов для разных объектных моделей на единой основе. Он не зависит от платформы и среды модели взаимодействия компонентов прикладного уровня, упрощает обработку документов, работу с БД с помощью средств (XML-парсеры, DOM-интерфейсы, XSL-отображение XML в HTML и др.).

Общая схема связи ЯП в распределенной среде

Характерная особенность ЯП в распределенных средах – неоднородность как в смысле представления типов данных, так и платформ компьютеров, где реализованы соответствующие системы программирования из множества языков $L_1,\ldots, L_n$. Причина неоднородности, как это указывалось выше, это различные способы передачи параметров объектам для разных сред, разнообразие объектных моделей, разных видов удаленного вызова модулей в современных ЯП (рис.1.7).

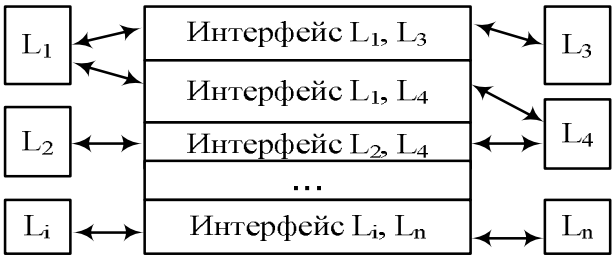


Рис.1.7. Схема связи $L_1, L_2, \ldots, L_n$ через интерфейсы

Системы программирования с ЯП имеют следующие особенности:

– различающиеся двоичные представления результатов компилирования для одного и того же ЯП на разных компьютерах;

– двунаправленность связей между ЯП и их зависимость от среды и платформы;

– отображение параметров вызовов объектов в операции методов;

– связь с разными ЯП через ссылки на указатели в компиляторах;

– связь ЯП через интерфейсы каждой пары модулей во множестве L_1, …, L_n промежуточной среды.

Иными словами, связь между различными языками L_1,…, L_n осуществляется через интерфейс каждой пары L_i, L_j, взаимодействующих между собой с помощью соответствующих конструкций языка L_i , операций интерфейса и наоборот.

Принципы взаимодействия ЯП в среде CORBA

Основной принцип взаимодействия объектов в среде CORBA – это запрос от клиента для выполнения метода объекта через интерфейс. Взаимодействие ЯП производится путем отображения типов данных модулей в типы данных клиентских и серверных стабов (stub) средствами брокера ORB.

Для всех ЯП системы CORBA (C++, JAVA, Smalltalk, Visual C++, COBOL, Ada-95) предусмотрен общий механизм связи и расположения параметров для методов объектов в промежуточном слое [25, 166]. Связь между объектными моделями каждого ЯП системы COM и JAVA выполняет брокер ORB (рис.1.8).

Если в общую объектную модель CORBA входит объектная модель COM, то в ней типы данных определяются статически, а конструирование сложных типов данных осуществляется для массивов и записей. Методы объектов используются в двоичном виде и допускается совместимость машинного кода объекта, созданного в одной среде разработки, коду другой среды, а также совместимость разных ЯП за счет отделения интерфейсов объектов от реализаций.

В случае вхождения в состав модели CORBA объектной модели JAVA/RMI, вызов удаленного метода объекта осуществляется ссылками на объекты, задаваемые указателями на адреса памяти.

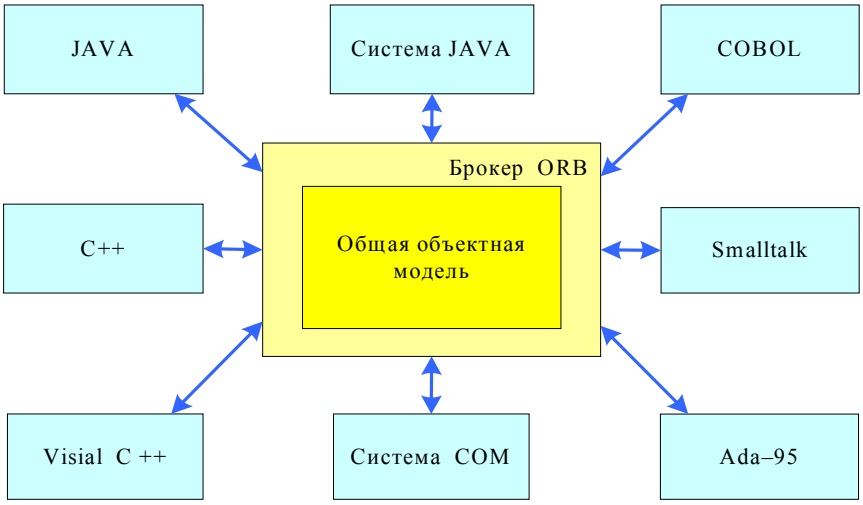


Рис.1.8. Интегрированная среда системы CORBA

Интерфейс как объектный тип реализуется классами и предоставляет

удаленный доступ к нему сервера. Компилятор JAVA создает байт-код, который интерпретируется виртуальной машиной, обеспечивающей переносимость байт-кодов и однородность представления данных на всех платформах среды CORBA.

1.4. ПРЕДСТАВЛЕНИЕ ЗНАНИЙ О ПРЕДМЕТНЫХ ОБЛАСТЯХ

Сложились устойчивые понятия о методах представления и использования знаний: продукционные правила; фреймы; семантические сети; исчисление логики предикатов; нечеткие знания. Данные методы неприменимы в методе сборочного программирования, несмотря на то, что существуют языки представления знаний, которые одновременно являются и языками программирования (например, Пролог, Лисп и др.)

В сборочном программировании знания о задачах предметной области встроены в алгоритмы самих модулей, компонентов, ПС и т.п. Промышленный подход к процессу сборки требует еще дополнительные знания, связанные с правилами их объединения в разные конфигурации для решения задач ПрО. Этот процесс соответствует выбору готовых деталей, стыковки их в некоторые мелкие технические узлы, а из них сборке в более крупные части изделия. Каждая деталь имеет конструкторскую документацию, включающую в себя документацию на комплектующие сборки и на результирующий продукт. Аналогично используются объекты и в сборочном программировании. Формой представления знаний об элементарных объектах является их описание в ЯП, а знаний о результатах сборки – более сложные структуры программ и ПС, приобретенные в процессе сборки. Последние включают в себя дополнительные знания о главной цели и задачах разработки ПС, а также и о схемах сборки, интерфейсах и правилах преобразования несовпадающих типов данных.

Таким образом, применение сборочного программирования позволяет получить новые знания не только о реализуемых предметных областях, но и о процессе разработки ПС из готовых объектов. Использование метода сборки расширяет сферу его применения. Это является еще одним отличием от традиционных методов разработки ПС, главная цель которых заключается в приобретении или накоплении новых знаний о предметных областях.

Возвратимся к формам представления ПС как знаний. Знания об элементарных объектах сборки включают в себя спецификации, исходные тексты на ЯП, описания данных (при работе с внешними файлами и базами данных), процедуры на языках управления заданиями для вызова ПП и т. д. Дополнительные знания, полученные в процессе сборки, представляются в виде совокупности моделей и программных интерфейсов сопряжения отдельных компонентов.

Автоматизированный подход к сборке ПП позволяет упорядочить процесс приобретения новых знаний на основе применения специальных средств описания и обработки моделей, а также генерации интерфейсов согласно набору стандартных правил. Эти средства – основа создания технологий применения разработанных ПП для их использования в более сложных программных продуктах. Каждый ПП в рамках сборочного программирования (с соответствующими средствами автоматизации) создается на единой методологической основе и без существенных изменений может использоваться в других процессах сборки.

На современном этапе развития программирования в качестве базы знаний

выступают репозитарии, каталоги информационных систем, ресурсы Интернета и др. Их рассмотрению посвящены специальные разделы данной работы.

Представление знаний о готовых программных ресурсах. В настоящее время главные ресурсы – это КПИ и разные атрефакты проектирования, полученные на более ранних процессах ЖЦ при создании ПС. Они недостаточно используются, как готовые, и потому в рамках научного проекта института созданы новые подходы, модели и методы их системного описания и сохранения в репозитарии компонентов, упорядоченных по разным предметным областям ПИ. Представленные знания обеспечивают более широкое их применение и дают значительное сокращение сроков и стоимости создания новых ПС.

Эти знания, заданные в репозитарии, обеспечивают сохранение, локализацию знаний о КПИ, их нахождение и контроль логической связанности, целостности и согласованности представленных в них знаний.

Для адекватного раскрытия семантики; описания разных параметров и аспектов функционирования КПИ разработана концепция многоаспектного представления знаний о КПИ в виде онтологической модели, как способа стандартизированного описания разных свойств и возможностей современных программных ресурсов, накопленных в разных хранилищах и библиотеках Интернета, в частности и в построенном нами репозитарии.

Данная модель практически реализована в диссертационной работе [19–21] и задается графовой структурой с концептами домена предметной области в вершинах и дугами, задающими отношениям между этими концептами. В КПИ концептами является информация о функциях, сервисах, интерфейсах, сценариях общения, вариантности развертки и т.п. На их основе можно провести аттестацию КПИ, изменить или добавить новы концепты, формулировать требования к новым ПС и по ним находить нужные КПИ, сопоставить найденные КПИ с теми, которые запрашивает пользователь и принимает решение о целесообразности их использования в новой разработке. В приложении 1 приведено краткое описание реализации онтологической модели на компьютере.

Вывод. В данной главе рассмотрен метод сборочного программирования на концептуальном уровне без конкретизации его составляющих. На общем уровне рассмотрена схема сборки программных объектов, компоновки и генерации разных программных ресурсов на основе модельного подхода. В последующих главах уточняется их содержание применительно к разным видам объектов (модулей и компонентов) и правила их сопряжения. Описывается несколько практических реализаций, приводятся методики применения данного метода, дано описание концепции развития и совершенствования сборки в среде компонентного и генерирующего программирования.

Глава 2

ОБЩИЕ ВОПРОСЫ РЕАЛИЗАЦИИ МЕТОДА СБОРОЧНОГО ПРОГРАММИРОВАНИЯ

2.1. МЕТОДЫ КОМПЛЕКСИРОВАНИЯ ПРОГРАММНЫХ ОБЪЕКТОВ

Из анализа существующих методов разработки ПО, изложенных в [5, 130–135] следует, что интерфейсам для комплексирования или интеграции программных объектов как самостоятельным инструментальным средствам в последние годы уделялось много внимание. Все вопросы комплексирования решаются на процессе проектирования ПО, что не всегда согласуется с практическими результатами. Задачи сопряжения программных объектов решаются различными методами и с помощью разных программных средств. Можно выделить следующие действующие группы методов решения задач сопряжения объектов:

1) комплексирование на основе аппаратных средств ЭВМ;
2) изменение структуры исходных текстов модулей;
3) использование средств компиляторов и редакторов связей для сопряжения модулей;
4) использование промежуточных языков систем программирования;
5) организация связи с помощью управляющей программы;
6) использование при комплектовании специальных модулей связи программных объектов;
7) системное комплексирование готовых программных объектов.

Рассмотрим эти методы более подробно.

1. Комплексирование с использованием аппаратных средств ЭВМ основано на специальной системе команд и способе представления данных. Архитектура таких ЭВМ отлична от модели Неймана. Оперативная память дает возможность хранить различные типы данных, при этом каждая ячейка содержит специальные признаки (тэги, дескрипторы и т. д.), позволяющие правильно интерпретировать их значения и выбирать соответствующие режимы обработки. Система команд, учитывающая эти признаки [12], позволяет не рассматривать задачи, связанные с преобразованием внутреннего представления данных в модулях.

В более совершенных ЭВМ, таких, как система Интел 432, ориентация направлена на использование конструкций ЯП Ада, дескрипторы существуют не только для простых типов данных, но и для сложных. В ее архитектуру входят специальные команды:

CREATE OBJECT (создать объект).

CREATE TYPE OBJECT (создать типизированный объект),

При выполнении этих команд создаются как обычные объекты, так и объекты «расширенного типа». Типы объектов проверяются на процессе трансляции и при выполнении команд, использующих эти объекты.

В отличие от системы Интел 432 архитектура системы В-1700 фирмы BURROUGHS ориентирована на несколько ЯП – Бейсик, Фортран, Кобол, РПГ/2 [133]. Система динамически меняет свою архитектуру во время диспетчеризации различных прикладных программ. При выполнении программы аппаратными средствами осуществляется переключение микропрограмм на режим ЯП данной программы. Аналогичным образом происходит обработка данных.

Среди известных разработок следует отметить МВК «Эльбрус», архитектура которого ориентирована на язык высокого уровня [76]. Процесс комплексирования в этом комплексе имеет следующие особенности. Преобразование относительных адресов готовых программ в машинные осуществляется на аппаратном уровне. Необходимость преобразования типов данных отсутствует за счет введения тэгов и возможности выбора алгоритма обработки данных на аппаратном уровне. Основная задача комплексатора (системной программы объединения модулей) состоит в добавлении новых процедур в базовую программу и/или замене существующих новыми версиями.

2. При комплексировании, основанном на изменении структуры и исходного текста модулей, проблемы сопряжения решаются путем внесения дополнительных операторов и команд в исходный текст для каждой пары взаимодействующих модулей в соответствии с подходами [61, 125, 165]. Основное преимущество данного метода состоит в том, что разработчик вносит наиболее оптимальные (по времени выполнения или объему дополнительной памяти) изменения. Любой метод, являющийся универсальным, потребует больше ресурсов, но и имеет следующие недостатки: разработчик должен самостоятельно осуществлять все вопросы комплексирования, особенности передачи управления, структуры передаваемых данных; объем работ по разработке программных объектов увеличивается, так как требуется дополнительное время на написание и отладку изменений, вносимых в отдельные модули; модули являются зависимыми от своего окружения, их замена и/или включение в другие объекты являются сложной задачей; при разработке модулей на ЯП высокого уровня не все проблемы сопряжения могут быть решены одними языковыми средствами – часть проблем необходимо реализовывать на языке типа Ассемблера.

Рассматриваемый метод комплексирования целесообразно применять при объединении небольшого числа модулей, если использование средств универсального интерфейса не оправдано.

3. Применение содержащихся в компиляторах средств для связи модулей также связано с изменениями в исходных текстах. Однако по сравнению с предыдущим методом изменения эти незначительные. Разработчик только указывает место в модуле и дополнительные средства, включаемые в это место, для вызова другого модуля. Примером рассматриваемого подхода может служить комплексирование с применением оптимизирующего транслятора с языка ПЛ/1. При вызове модуля, написанного на ЯП Фортран, из Паскаль-модуля последний будет содержать следующие дополнительные операторы:

DCL FOR OPTIONS (FORTRAN)

ENTRY (<описание типов данных передаваемых параметров модулю FOR>);

операторы ЯП CALL FOR (<список передаваемых параметров>).

При вызове Паскаль-модуля из Фортран-модуля его заголовок будет иметь вид:

Pascal: PROC (<список параметров>) OPTIONS (FORTRAN).

Реализацию аналогичного подхода к комплексированию объектов также

предусматривают трансляторы с ЯП Паскаль, LAVA в соответствующих операционных системах и средах.

К достоинствам данного метода комплексирования следует отнести незначительные дополнительные затраты на внесение изменений в исходный текст и освобождение разработчика от знаний особенностей сопряжения. Основные его недостатки состоят в следующем: модули зависят от своего окружения; невозможно полностью отделить процесс трансляции модулей от процесса комплексирования; не все трансляторы обеспечивают возможности по сопряжению разных модулей.

Последний недостаток довольно существенный, так как требует применения других методов комплексирования.

4. Комплектование на основе использования промежуточных языков систем программирования является следствием применения специальных промежуточных кодов трансляции, называемых Р-кодами [199]. Входными языками трансляторов в таких системах служат обычные ЯП высокого уровня. Благодаря применению единого промежуточного Р-кода достигается совместимость объектных программ. Наиболее известные промежуточные коды – UCSD-код, Р-4, М-код. Система UCSD поддерживает программирование на ЯП Паскаль, Фортран, Бейсик, АПЛ, Лисп, Модула-2. Разработаны специальные компьютеры, внутренними языками которых являются UCSD-код [125? 214] и М-код. Система UCSD представляет собой абстрактную машину, а система команд ее ориентирована на работу со стеком. Стек используется для хранения общих и локальных данных, включая параметры процедур и временные переменные, а также коды процедур активных сегментов, т.е объектов, загружаемых в память из внешнего устройства. В момент вызова процедуры создается ее сегмент данных, состоящий из параметров, локальных данных и маркера, определяющего начало области стека для вызванной процедуры. Проверка совместимости типов проводится при компиляции.

К достоинствам подобных систем относятся: простота подхода к комплексированию; отсутствие специфики сопряжения и высокая степень переносимости ПС. Главным недостатком применения Р-кодов является снижение скорости выполнения программы, так как она выполняется в режиме интерпретации. Использование ЭВМ, внутренним языком которых являются Р-коды [199], устраняет этот недостаток.

5. Организация связи с помощью управляющей программы требует наличия универсальной программы сопряжения. Взаимодействие между парой модулей не прямое, а осуществляется через управляющую программу, которая выполняет все необходимые операции сопряжения для данной пары. Пример такого подхода приведен в [172].

Для взаимодействия модулей, написанных на языках Фортран, ПЛ/1, Ассемблер, разработана управляющая программа, содержащая 6 точек входа. Каждая соответствует определенному типу взаимодействия модулей. Необходимо отметить, что данная программа позволяет решать вопросы сопряжения, связанные с передачами управления, не затрагивая преобразования передаваемых данных.

К достоинствам метода комплексирования следует отнести наличие готовых средств для реализации интерфейса (управляющая программа), единый подход к решению вопросов сопряжения и незначительные усилия для освоения управляющей программы, требуемые от разработчика проблемных модулей. К недостаткам - наличие готовой управляющей программы, которая могла бы решать все задачи

сопряжения; необходимость изменения модульной структуры программы и зависимость модулей от окружения, проявляющаяся в привязке к конкретным точкам входа при вызове других модулей.

6. Метод комплексирования, основанный на применении специальных модулей сопряжения, имеет много общего с методом использования управляющей программы. Взаимодействие между модулями не прямое, а осуществляется через специальный модуль сопряжения, называемый модулем связи или посредником. Модуль связи разрабатывается для каждой пары взаимодействующих модулей в программном комплексе. Данный метод основывается на библиотеке интерфейса, содержащей макроопределения и модули преобразования типов данных. Каждое макроопределение выполняет определенную функцию сопряжения, связанную с передачей управления и данных между объединяемыми модулями. Последовательность макрокоманд, составленная согласно используемому подходу, образует модуль связи и включается в состав готового агрегата для каждой пары связываемых разноязыковых объектов. Модуль связи состоит из:

– оператора идентификации имени модуля связи;

– макрокоманды пролога, включающей подготовительные операции для выполнения модуля связи;

– последовательности макрокоманд, обеспечивающей преобразование передаваемых данных с учетом их форматов, определенных в вызываемом модуле;

– последовательности макрокоманд, обеспечивающей создание функциональной среды вызываемого модуля и передачи ему управления;

– последовательности макрокоманд, обеспечивающей обратное преобразование возвращаемых вызывающему модулю данных;

– макрокоманды эпилога модуля связи, обеспечивающей возврат управления вызывающему модулю;

– оператора завершения модуля связи.

К достоинствам данного подхода следует отнести: освобождение разработчика от знаний особенностей сопряжения модулей; наличие заготовок в библиотеке интерфейса для генерации модуля связи; независимость модулей от окружающей среды; возможность реализации новых функций сопряжения за счет расширения библиотеки новыми функциями преобразования типов данных, передаваемых от одного модуля другому.

К недостаткам отнесем: большой объем работ, связанный с программированием модулей связи; значительное увеличение числа модулей, включаемых в программный агрегат, усложняет гибкость управления модульными структурами. Эти недостатки в значительной степени устраняются при автоматизированном подходе к процессу комплексирования. Пример реализации такого подхода описан в работах по системе АПРОП [44, 56, 118, 121, 135].

7. Метод комплексирования готовых программных компонентов существенно отличается от ранее рассмотренных тем, что в нем переход к новому программному компоненту осуществляется не стандартным механизмом вызова, а с помощью средств ОС после завершения работы предыдущей компоненты. Применение данного подхода описано в [73]. Механизм вызова представляется на языке управления заданиями ОС ЕС в виде пакета задания, состоящего из нескольких шагов. Первые шаги описывают обращения к прикладным программам, а на последнем из них происходит вызов специальной программы, осуществляющей

обращение к программе системного ввода. Выбор нужного компонента состоит в модификации пакета задания (изменение имени программы, описания ресурсов и др.) и повторном обращении к программе системного ввода.

К достоинствам данного метода следует отнести простоту использования средств комплексирования, полную независимость программных компонентов друг от друга и минимум знаний о процессе комплексирования.

Самым существенным недостатком является то, что все программные объекты должны быть доведены до уровня готовности к выполнению (т. е. пройти процессы трансляции, редактирования связей, занесения в библиотеку готовых программ), а замена программных объектов описанным способом требует значительных вычислительных ресурсов, особенно времени выполнения.

Обобщенные результаты анализа методов комплексирования на разных стадиях разработки программных комплексов представлены в табл. 2.1.

Таблица 2.1

Метод комплексирования	**Выбор архитектуры ЭВМ**	**Разработка модулей**	**Разработка дополнительных ПС**	**Трансляция модулей**	**Сборка модулей**	**Выполнение ПП**	**Средства автоматизации**
1. Использование аппаратных средств	~				V	~ V	+
2. Изменение структуры текста и модулей		~	~ V		V		–
3. Использование средств компиляции				~	V		+ -
4. Использование промежуточных ЯП	~			~	V	~ V	+
5. Использование управляющей программы	V				V		+ -
6. Использование модулей связи	~		~		V		+
7.Комплексирование готовых программ						V	+

Здесь приняты следующие обозначения:

«~» – отметка о необходимости решения задачи сопряжения пар модулей (по управлению и по данным);

«V» – отметка о необходимости решения проблемы объединения программных компонентов в единый агрегат;

«+» – возможность автоматизации данного метода комплексирования, а « - » – ее отсутствие;

«+» – возможность автоматизации частных задач и отсутствие общих путей автоматизации.

Вопросы автоматизации построения интерфейсов тесно связаны с задачей

повторного использования программных средств, которой в настоящее время уделяется большое внимание. В работе [227] предлагается язык LIL (Library Interconnection Language) для использования компонентов, хранящихся в библиотеках общего пользования. Эффективность такого подхода подтверждает опыт японских фабрик разработки ПО, где производительность труда программистов за год при использовании ранее разработанных программных средств возросла на 20 % [125].

Необходимость повторного использования программных средств привела к созданию метода сборочного программирования [116, 125–127]. В этом случае основные затраты на разработку связаны с созданием новых программных компонентов и комплексирование их со старыми. В [195] приводится описание инструментальных средств для разработки программных компонентов многоразового использования. В качестве иллюстрации выбран класс задач обработки данных.

2.2. СРЕДСТВА АВТОМАТИЗАЦИИ МЕТОДОВ КОМПЛЕКСИРОВАНИЯ

На базе описанных выше методов комплексирования создано множество систем автоматизации программирования и комплексирования, разработаны специальные языки и технологии. Необходимо отметить, что в большинстве подходов с понятием комплексирования связывают и синтез программ. В общем случае это справедливо, так как в процессе комплексирования есть множество черт, сходных с процессом синтеза, и наоборот.

В частном случае для систем модульного программирования такие процессы имеют определенные границы. Ниже приведен обзор известных методов и систем синтеза и комплексирования программных средств.

1. В [178–180] рассмотрен метод композиционного программирования. Суть его состоит в уточнении понятия семантической структуры программ. С этой точки зрения данные уточняются как именные данные, запрашивающие программы (выражения и операторы ЯП с семантической точки зрения рассматриваются как именные функции). Средства конструирования запрашивающих программ рассматриваются как композиции именных функций. Программы отождествляются с выводами в универсальной программной логике, роль аксиом в которой выполняют именные функции, а роль правил вывода – композиции.

2. Метод концептуального программирования рассмотрен в [195]. Для конкретной предметной области описываются понятия, достаточные для выражения смысла решаемых задач. В терминах этих понятий выполняется описание каждой задачи, по которому осуществляется автоматический синтез программы и выполняются вычисления. Алгоритмы структурного синтеза программ основаны на доказательстве существования решения, которое находится программой. Алгоритмы поиска рассматриваются для трех различных классов задач – линейной структуры, ветвящейся структуры и структуры с подпрограммами. Проблемы интерфейсов решаются на процессе синтеза программ. Большинство принципов, положенных в основу концептуального программирования, реализовано в системе ПРИЗ [195]. Развитием ее явилась мобильная инструментальная система, предназначенная для автоматизации построения пакетов прикладных программ [170].

3. Системой, близкой к системе ПРИЗ по концепциям синтеза программ, является СПОРА [144]. В ней синтез основан на использовании баз знаний. Данная

система обеспечивает построение программы по спецификации задачи. Сам процесс построения представляется как доказательства теоремы существования решения задачи в некоторой формальной теории. Архитектура базы знаний ориентирована на модульный принцип построения системы в целом.

4. Целый ряд систем автоматизации программирования построен на принципе использования промежуточных языков. В качестве примера можно привести язык АЛМО [94], систему СИМПР [82, 83], язык MESA [228] и др. Особый интерес представляет система СИМПР, являющаяся многоязыковой системой модульного программирования. В основу промежуточного языка положен абстрактный семантический уровень интерпретации программ, который содержит понятия, необходимые для описания внешних свойств модулей. В качестве формы записи языка используется польская инверсная запись. Слогами этой записи являются операции языка или указатели объектов транслируемой программы, к которым применяются данные операции.

Абстрактный семантический уровень открыт для расширения, что позволяет включать в систему новые ЯП. Включение нового языка – это фиксация всех языковых понятий, которые могут выступать в качестве внешних свойств модуля, в терминах уровня интерпретации программ. Интерфейс между разноязыковыми модулями не требуется, так как внешние свойства каждого модуля определены на едином для системы СИМПР абстрактном семантическом уровне.

Язык MESA предназначен для создания больших программ по модульному принципу. Он ориентирован на использование ЯП с блочной структурой типа Алгол, Паскаль и др.

5. Подход, связанный со структурной интерпретацией языков высокого уровня, нашел применение в R-технологии, описанной в [39]. При программировании в терминах специального R-языка создается структура исходной предметной области. Затем проектируется логическая структура данных, связанная с соответствующими алгоритмами обработки. Работы по визуализации R-языка и распространению графического стиля программирования, расширили его применение в языках высокого уровня (Модула-2, Си, Паскаль и др.) и современных языков С и С++.

Практически во всех системах автоматизации программирования приходится решать задачи комплексирования программных компонентов и управления модульными структурами программ.

6. Программная система BPS предназначена для разработки больших программных комплексов. Входные языки – Ассемблер и специально разработанный язык BPS/L. За основу разработки BPS/L был взят язык МОДУЛА, дополненный специальными инструкциями, предоставляющими необходимую информацию для комплексирования. В состав этой информации входят: список модулей, с которыми связан данный модуль, описание объектов, используемых вне модуля, описание локальных объектов и т. д. Специальные средства связи обеспечивают построение графа программного агрегата и управление комплексированием на основе указанной информации. При программировании модуля с использованием языка Ассемблера его структура дополняется специальными инструкциями, аналогичными таковым в языке BPS/L.

7. В основу системы автоматизации модульного программирования (САПМ) [183] положен принцип инверсного преобразования графа достижения заданной цели. Для множества пакетов прикладных программ (ППП) строятся орграфы

предметных областей. По ним создается совокупность матриц связей, описывающих взаимосвязи программных модулей. Каждой вершине графов ставится в соответствие цель, которую необходимо достигнуть, например номер задачи, выполнение которой обеспечит достижение данной цели. Для решения задачи на графе указываются конечные цели и с помощью матриц связи строятся обратные цепочки с фиксацией промежуточных целей. Вопросы интерфейсов между модулями не рассматриваются.

8. Методика проектирования модульных программных структур [203] заключается в преобразовании программной структуры в модульную структуру с помощью четырех типов модулей: функционального; модуля данных; абстрактной структуры данных; абстрактного типа данных. Каждый из них содержит информацию, которая описывает его внешнее поведение. Все связи осуществляются модульным интерфейсом, проводится классификация видов связей модулей, разработаны объективные критерии создания модульных структур и методика их использования. Методология проверена практикой.

9. В инструментальном комплексе ДИСУППП [173] комплексирование модулей осуществляется на основе многоуровневого графа модели ПрО. Сами функциональные модули реализуют отдельные шаги вычислений и хранятся в специальной базе. Комплекс ДИСУППП эффективно применяется при конструировании диалоговых систем.

10. В рамках системы SARA [255] разработан язык модульного взаимодействия MIL, который позволяет описывать взаимодействие не только между программными модулями, но и между программным модулем и модулем данных, примером которого может служить файл. В последнем случае взаимодействие эквивалентно операциям обмена. С помощью средств языка MIL описываются как сами модули, так и интерфейсы между ними.

11. Описание взаимодействия модулей дается в языке мультимодульного программирования, являющегося составной частью алгоритмического языка Маяк [160]. Программа в этом языке – это сеть алгоритмических модулей переменной структуры. Язык позволяет описывать динамическое взаимодействие между модулями и обеспечивает возможность параллельного их вычисления.

Другие языки и системы, обладающие аналогичными возможностями, кратко изложены в [125].

2.3. ОБЪЕКТЫ СБОРОЧНОГО ПРОГРАММИРОВАНИЯ

Построение любой формальной теории или системы основано на определении объектов, рассматриваемых этой теорией, их характеристик и свойств, а также на выборе операций над данными объектами. Следуя этой схеме, необходимо определить объекты сборочного программирования, понимая под операциями различные функции комплексирования – информационное и программное сопряжение, обеспечение интерфейса по управлению, создание интегрированных сред, языков общения с пользователями и т. д.

Строгих формальных определений для программных объектов в настоящее время не существует. В практике программирования используются такие термины, как модуль (включая подпрограмму, функцию, процедуру и др.), макромодуль, программа, комплекс программ, система, подсистема и т. д. Каждое понятие в контексте избранных вопросов исследований может иметь тот или иной смысл, что связано с выбором конкретных концепций, методов, моделей для анализа. На

практике указанные термины употребляются для сходных по признакам программных объектов, интуитивно понятных большинству специалистов. Из этого следует, что объекты, их свойства, операции над ними, рассматриваемые в данной книге, будут анализироваться с позиций сборочного программирования и полученные результаты не претендуют на всеобщность применения.

Рассмотрим два типа программных объектов сборочного программирования. К первому отнесем объекты, которые могут менять формы своего представления при разработке ПС. В частности, к ним относятся макромодуль, модуль, программный сегмент. Все эти объекты изменяют форму представления (исходную, объектную, выходную) на различных процессах разработки ПС. Макромодуль используется на процессе макрогенерации, результат которой – исходный или объектный текст модуля. Модуль может проходить процессы трансляции и сборки (редактирования связей). Программный сегмент включается в более сложный объект на процессе редактирования связей его элементов. К этим объектам относятся также, приведенные в главе1: компонент, сервис, агент и др. Среди объектов первого типа выберем в качестве базового – модуль. Все проблемы комплексирования для объектов данного типа сводятся к проблемам взаимодействия объектов и построения из них более сложных программных структур.

Ко второму типу относятся производные программные объекты, не меняющие форму представления при их использовании: программы, комплексы, системы и т.п. Все эти объекты автономны, решают определенные классы задач, и для их использования не требуется дополнительная обработка. Базовый объект этого типа – программа. Предполагается, что более сложные объекты – комплексы, системы и т.д. представляют собой множество взаимодействующих программ. Сама программа как конечный результат процесса разработки ПС состоит из некоторого множества модулей, компонентов, сервисов и др.

Определение понятий модуль и программы не достаточно формализовано. Более строгое определение этих понятий в рамках метода сборочного программирования будет приведено ниже.

В программных системах новых поколений в качестве объектов используются также пакеты прикладных программ, а любая система обработки данных представляется как открытая система. Сюда относится архитектура ISA и SAA.

Выбор модуля и программы в качестве базовых программных объектов сборочного программирования оказывает принципиальное влияние на реализацию функций комплексирования, что вызвано следующими особенностями этих объектов.

1. Возможность изменения объектов при реализации интерфейсов между ними. Для модулей такие изменения допустимы. К ним, в частности, относятся изменения в описаниях данных и порядке следования передаваемых параметров, особые режимы трансляции и редактирования связей. Программы такой возможности не имеют. Поэтому, если программа формирует файл определенной структуры, то для ее изменения необходимо вносить изменения в модули, входящие в программу, что противоречит требованию неизменяемости формы представления.

2. Уровень реализации интерфейсных функций. Для реализации интерфейсов между модулями используются механизмы, определяемые ЯП и соответствующими трансляторами. Для информационного сопряжения необходимо наличие средств преобразования данных (типов и внутреннего представления) ЯП. Решение

проблемы связи модулей зависит от особенностей механизмов управления, используемых трансляторами в конкретных ЯП. Реализация интерфейсов между программами использует механизмы ОС, в которой обмен информацией происходит в основном на уровне файлов (включая файлы баз данных) и командных строк. Для вызова программы используются специальные средства супервизора ОС.

Выделение двух типов программных объектов позволяет установить связь сборочного программирования с методом модульного программирования и методами построения интегрированных комплексов. Иными словами, метод сборочного программирования – обобщение метода модульного программирования и результаты, полученные при его исследовании и анализе, будут справедливы для частных случаев.

2.4. ОСНОВНЫЕ МОДЕЛИ ПРОГРАММНЫХ СИСТЕМ (ПС)

Модель – это математическое или другое представление системы, процессов ее изготовления из готовых объектов (путем их выбора, определения интерфейсов, интеграции и др.) и управления этими процессами.

В сборочном программировании – это модель сопряжения программных объектов и модель управления ими. Указанные модели названы основными, так как проблемы передачи данных и управления – главные задачами при объединении любых программных объектов с применением соответствующих методов сборочного программирования и инструментальных средств.

В объектно-ориентированном программировании сформировались другие модели: объектная модель, информационная модель, модель состояний, модель процессов и др. Общее, что их объединяет, – это обеспечение интерфейса объектов при создании из них ПС.

2.4.1. МОДЕЛИ СБОРОЧНОГО ПРОГРАММИРОВАНИЯ

Модель информационного сопряжения. Под *информационным сопряжением* двух или нескольких программных объектов понимается процесс преобразования множества их общих данных к форме, согласующейся с представлением структуры каждого из объектов. В дальнейшем, не снижая общности процесса анализа, рассмотрим информационное сопряжение только пары разных объектов. Модель информационного сопряжения – это совокупность формальных описаний данных взаимодействующих объектов и функций их преобразования к релевантному виду.

Процесс сборки объектов в интегрированный комплекс (ИК) – это создание сложного программного объекта путем объединения готовых более простых программных элементов.

Пусть $P = \{p^i\}_{i=\overline{1,s}}$ – множество программных компонентов, входящих в состав создаваемого ИК. С каждым p^i связано множество данных D^i, с помощью которых осуществляется информационный обмен между интегрируемыми компонентами.

Множество $D^i = \{d_j^i\}_{j=\overline{1,t}^i}$ состоит из переменных d_j^i, каждая из которых характеризуется тройкой: именем (идентификатором переменной) N_j^i, типом T_j^i и текущим значением V_j^i).

Рассмотрим два программных компонента p^i и p^k (p^k выполняется после p^i) с множествами данных D^i и D^k соответственно. В общем случае в D^i и D^k могут входить переменные, общие для p^i и p^k с точки зрения их семантической обработки. Эти переменные образуют подмножества $\tilde{D}^i$ и $\tilde{D}^k$. Задача информационного

сопряжения состоит в преобразовании подмножества данных $\tilde{D}^i$ в представление, согласующееся с $\tilde{D}^k$.

Введем следующие обозначения: $N^i = \{ N_j^i \}_{j=1,\ldots t}^{i}$, $T^i = \{T_j^i\}_{j=1,\ldots t}^{i}$, $V^i = \{V_j^i\}_{j=1,\ldots t}^{i}$. В $\tilde{D}^i$ им соответствуют множества из троек – $\tilde{N}^i$, $\tilde{T}^i$ и $\tilde{V}^i$. В общем случае для преобразования множества данных D^i необходимо построить преобразование для этих имен $\tilde{N}^i$, $\tilde{T}^i$ и $\tilde{V}^i$. Имеет место следующие два случая.

1. Каждой переменной $d_j^i \in \tilde{D}^i$ соответствует только одна переменная $d_j^i \in \tilde{D}^k$. Тогда преобразование (отображение)

$$F^{ik}: \tilde{D}^i \rightarrow \tilde{D}^k \qquad (2.1)$$

состоит из множества преобразований для отдельных переменных: $F^{ik}=\{F^{ik}_{ji}\}$. При этом формально $F^{ik}_{ji} = (FN^{ik}_{ji}, FT^{ik}_{ji}, FV^{ik}_{ji})$. Вводя обозначения $FN^{ik} =\{FN^{ik}\}$, $FT^{ik} = \{FT^{ik}_{ji}\}$, $FV^{ik} = \{FV^{ik}_{ji}\}$, определяем преобразования:

$$\begin{aligned} FN^{ik} &: \tilde{N}^i \rightarrow \tilde{N}^k \\ FT^{ik} &: \tilde{T}^i \rightarrow \tilde{T}^k \\ FV^{ik} &: \tilde{V}^i \rightarrow \tilde{V}^k \end{aligned} \qquad (2.2)$$

соответственно для множеств идентификаторов, типов данных и значений.

2. Между переменными d_j^i и d_j^k не существует однозначного соответствия. Это тогда, когда несколько элементов из $\tilde{D}^i$ соответствуют одному элементу из $\tilde{D}^k$ и наоборот. Сложная связь, при которой несколько элементов из $\tilde{D}^i$ соответствуют нескольким элементам из $\tilde{D}^k$, в практике сборочного программирования, как правило, отсутствует, что связано с раздельной разработкой отдельных программных компонентов.

Соответствие нескольких переменных одной и наоборот свидетельствует об изменении уровня структурирования данных. Пусть $\tilde{d}^i_j$ соответствует несколько элементов из $\tilde{D}^k$. Обозначим их через $\tilde{d}^k_{j1, \ldots\ldots}\tilde{d}^k_{jr}$, а через S – функцию селектора, снижающую уровень структурирования данных:

$$S(\tilde{d}^i_{j1}) = (\tilde{d}^i_{j1, \ldots\ldots}\tilde{d}^i_{jr}), \qquad (2.3)$$

где каждому $\tilde{d}^l_{jv}$ соответствует $\tilde{d}^k_{jv}$, при $v = 1, 2, \ldots, r$. Замещая $\tilde{d}^l_j$ в $\tilde{D}^i$ элементами $d^i_{j1},\ldots, d^i_{jr}$ получаем множество $\tilde{D}^i$. Построение отображения F^{ik}: $\tilde{\tilde{D}}^i \rightarrow \tilde{D}^k$ производится аналогично, как в случаи 1.

При соответствии нескольких элементов из $\tilde{D}^i$ одному элементу из $\tilde{D}^k$ поступаем следующим образом. Вместо функции селектора вводим функцию конструирования вида

$$C(\tilde{d}^i_{j1, \ldots\ldots}\tilde{d}^i_{jr}) = \tilde{d}^i_j, \qquad (2.4)$$

где $\tilde{d}^i_j$; соответствует единственному элементу из $\tilde{D}^k$. Модифицируя элементы множества $\tilde{D}^i$ и рассматривая отображение F^{ik}: $\tilde{\tilde{D}}^i \rightarrow \tilde{D}^k$, приходим к аналогичному результату.

Проведем анализ отображений *FN, FT* и *FV* (индексы для простоты опущены). Из построения следует, что множества $\tilde{N}^i$ и $\tilde{N}^k$ содержат одинаковое количество элементов. Поэтому *FN* только переупорядочивает идентификаторы переменных в соответствии с последовательностью, принятой при описании программного компонента p^k.

Отображение преобразования множества типов данных *FT* более сложное, оно связано с наличием практически неограниченного количества типов. По определению тип данных характеризуется парой

$$T = (X, \Omega),$$

где X – множество значений, которые могут принимать переменные рассматриваемого типа, Ω – множество операций, выполняемых над этими переменными. T можно рассматривать как алгебраическую систему. В ней преобразование типа $T_j^i = (X_j^i, \Omega_j^i)$ в тип $T_j^k = (X_j^k, \Omega_j^k)$ соответствует преобразованию множества значений X_j^i в X_j^k, при котором семантическое содержание операций из Ω_j^i эквивалентно операциям из Ω_j^k.

В общем случае преобразование T^i_j в T^k_j может быть односторонним. Однако для повторного использования данных, что характерно для многократного вызова программных компонентов, обрабатывающих одни и те же структуры данных, требуется и прямое и обратное преобразования. Для достижения этого необходимо, чтобы отображение между T^i_j и T^k_j было изоморфизмом. Иными словами, построению преобразования между двумя типами данных будет соответствовать нахождению изоморфного отображения между двумя алгебраическими системами.

При практической реализации модель информационного сопряжения целесообразно рассматривать как совокупность моделей для пар программных компонентов $P = \{P^{ik}\}$ в создаваемой программе.

Модель для каждой из них имеет вид

$$M^{ik} = (\tilde{N}^i, \tilde{T}^i, \tilde{V}^i, N^k, T^k, V^k, FN^{ik}, FT^{ik}, FV^{ik}). \qquad (2.5)$$

Отдельные составляющие данной модели и принципы их построения описаны выше.

Модель управления программными объектами. Под *управлением программными объектами* в рамках метода сборочного программирования понимается процесс выбора объекта из библиотеки готовых модулей, планирование его использования в программном агрегате и активизация в целях выполнения. Условие выбора любого объекта – необходимость выполнения конкретной функции ПрО, вытекающей из алгоритма решаемой задачи, и готовность входных данных. Модель управления – это формальное описание условий для выбора объектов и его выполнения.

Анализируя критерии выбора программных объектов, необходимо отметить два случая.

1. Более приоритетный – критерий выполнения необходимой функции согласно алгоритму решения задачи. В этом случае происходит целенаправленная подготовка входных данных для выбранного программного объекта. Такая ситуация характерна для большинства методов передачи управления. Примерами могут служить оператор вызова ЯП, где с именем вызываемого модуля указывается список передаваемых параметров, и пакет операторов языка управления заданиями ОС, содержащий последовательность вызываемых программ и описания необходимых файлов. Вызов объекта происходит из предположения, что необходимые входные данные уже подготовлены ранее.

2. Приоритетный также критерий готовности данных. Вызывается тот объект, для которого подготовлена входная информация. Данный случай встречается значительно реже, и его целесообразно рассмотреть на конкретном примере.

На рис. 2.1. представлена схема интегрированного комплекса, состоящего из четырех программ (P_1, P_2, P_3, P_4). P_1 создает файлы F_1 и F_3. Файл F_1 служит входной информацией для программы P_2,, результат выполнения которой – файл F_2. Аналогично, результат работы F_3 – файл F_4. Программа P_4 использует файлы F_2 и F_4..

Рассмотрим последовательность вызовов программ, основываясь на критерии готовности данных.

Первой будет выполняться программа P_1 , так как она единственная не требует входной информации. После ее выполнения может быть вызвана P_2 или P_3,.

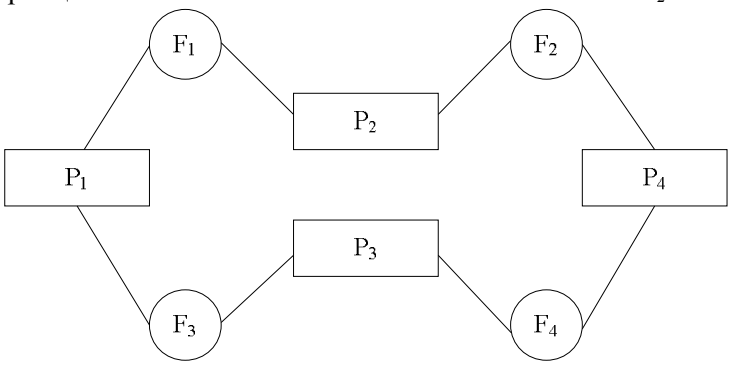


Рис. 2.1. Пример схемы интегрированного комплекса

Выберем программу P_2. После ее выполнения согласно второму критерию будет вызвана P_3 , так как для P_4 требуется файл F_4. Последней выполняется программа P_4. Аналогичный результат будет, если программа P_3 выполнится раньше, чем P_2. Таким образом, критерию готовности данных может удовлетворять несколько программных объектов, и любой из них может быть выбран в качестве следующего.

Оба рассмотренных случая можно объединить в рамках единой модели. Для этого введем специальный тип данных – «программные» переменные (PROGVAR). Этим переменным могут присваиваться имена выполняемых объектов. Переменные типа PROGVAR объединяются с данными для программных объектов, и рассматриваемый критерий применяется к процедуре объединения. Перейдем к формальному описанию модели управления.

Пусть $P = \{p^i\}_{i=\overline{1,s}}$ – множество программных компонентов и множество D^i соответствует p^i, как и для модели информационного сопряжения. Рассмотрим множество данных интегрированного комплекса, определяемое как D^i

$$D = (\bigcup_{i=1}^{s} D^i) \bigcup D^c, \qquad (2.6)$$

где D^c обозначает множество управляющих данных. С каждым компонентом p^i свяжем предусловие R^i и постусловие Q^i, которые задаются на множестве данных D. R^i проверяется перед вызовом p^i и определяет условие вызова данного компонента. Q^i проверяется после вызова p^i и определяет условие завершения решаемой задачи или выбора следующего компонента.

Пусть $R = \{R^i\}_{i=\overline{1,s}}$ и $Q = \{Q^i\}_{i=\overline{1,s}}$. Суть модели управления программными объектами состоит в следующем. При выборе очередного компонента просматривается множество Q до первого истинного условия, затем проверяется соответствующее условие из R. Если последнее условие истинно или оно может быть к нему приведено (например, вводом необходимой информации с терминала пользователя), то происходит вызов соответствующего компонента. Если условие из R ложно, то происходит поиск следующего истинного условия в Q и т. д.

Последовательность вызовов компонентов в общем случае не детерминирована, что обеспечивает динамические связи между компонентами.

Множество управляющих данных D^c содержит переменные, входящие в условия и не связанные с множествами D^i. В частности, к ним относятся переменные:

– содержащие имена программных компонентов интегрированного комплекса;

– определяющие характеристики среды выполнения (технические условия, версия ОС и т. д.);

– задаваемые пользователем.

Наличие «программных» переменных вводит элементы детерминизма путем упорядочивания вызовов программных компонентов. Подробный анализ данной модели показывает, что задача выбора очередного программного компонента аналогична задаче логического программирования. При этом условия R^i и Q^i соответствуют правилам для логического вывода, содержащим предикаты, функции и переменные из множества D. Множество предикатов и функций – не фиксировано и оно может содержать арифметические операции, операции над файлами и т. д. В состав данного множества также входят специальные функции, позволяющие проверять готовность данных, обращение к пользователю для ввода информации с терминала или выбора одного из множества компонентов, для которых выполняется истинность условий и т. д. С точки зрения логического программирования, условия R^i и Q^i равноценны, а порядок их проверки определяется выбранной стратегией обработки.

Средствами логического программирования опишем модель управления для ИК, схема которого приведена на рис. 2.1 (в нем для простоты верхний индекс переменной P заменен нижним). Введем следующие предикаты и функции:

select *(x)* – выбор компонента с именем *x*;
def *(y)* – анализ готовности данного *y;*
undex (*y*) – анализ отсутствия готовности данного *y;*
exec *(x)* – выполнение компонента с именем *x;*
setdef *(y)* – установка признака готовности для данного *y;*
cleardef *(y)* – сброс признака готовности для данного *y*.

Рассмотрим следующее описание:

1: select (P_4): – def (F_2), def (F_4), exec (P_4), cleardef (F_2), cleardef (F_4)
2: select (P_3): – def (F_3), exec (P_3), setdef (F_4), cleardef (F_3)
3: select (P_2): – def (F_1), exec (P_2), setdef (F_2), cleardef (F_1)
4: select (P_1): – undef (F_1), undef (F_3), exec (P_1), setdef (F_1), setdef (F_3).

Запрос на выбор очередного компонента имеет вид select *(x)*. Ему будет удовлетворять правило 4. Его выполнение связано с вызовом программы P_1 и установкой признаков готовности для файлов F_1 и F_3. Второму запросу удовлетворяют правила 2 и 3, третьему – правило 4. После этого все признаки приведены в начальное состояние, и четвертому запросу будет удовлетворять правило 4.

Выбор и выполнение очередного компонента можно автоматизировать. Для этого логическая программа будет иметь следующий вид:

1: select *(x):– S (x)*
2: *S* (P_4): – def (F_2), def (F_4), exec (P_4), cleardef (F_2), cleardef (F_4)
3: *S* (P_3): – def (F_3), exec (P_3), setdef (F_4), cleardef (F_4), *S(x)*

4: S (P2) : – def (*F1*), exec (*P2*), setdef (*F2*), cleardef *(F1), S(x)*

5: *S (P1):* – undef *(F1*), under *(F3),* exec *(P1),* setdef (*F1*), setdef (*F3*), *S(x).*

Программа выполняет аналогичные действия, а смена компонента происходит автоматически. Рассмотренные логические программы построены на основе второго критерия – критерия готовности данных. Приведем два варианта программы, основанные на первом критерии:

1: select *(x):– S (P4)*

2: *S (P4): – S (P3),* exec *(P4)*

3: *S (P3):– S (P2),* exec *(P3)*

4: *S (P2): – S (P1)*, exec *(P2)*

5: *S (P1): –* exec *(P1)* .

Согласно этому описанию происходит последовательное выполнение компонентов P_1 , P_2,, P_3 и P_4. Во втором варианте используется «программная» переменная. Для операций над ней введены предикат eq(*Y,P),* проверяющий равенство значения переменной *Y* имени переменной *P,* и функция set *(Y, P),* присваивающая переменной *Y* имя *P:*

1: select *(x)*: – set *(Y, P1)*, *S (Y)*

2: *S(Y): –* eq *(Y, P4),* exec *(P4)*

3: *S(Y): –* eq *(Y, P3),* exec *(P3)*, set *(Y, P4)*, *S (Y)*

4: *S(Y): –* eq *(Y, P2),* exec *(P2)*, set *(Y, P3)*, *S (Y)*

5: *S(Y): –* eq *(Y, P1)* exec *(P1)*, set *(Y, P2), S (Y)*

Практическая реализация описанного механизма выбора и вызова компонента может выражаться в виде программы, написанной на языке ПРОЛОГ или в другом ЯП высокого уровня. Окончательно модель управления интегрированным комплексом примет следующий вид:

$$M = (P, D, R, Q, I), \qquad (2.7)$$

где *P* – множество программных компонентов, имена которых включены в пред– и постусловия; *D* – множество данных, определяемое (2.6); *R = R (D)* – множество предусловий, определенных на *D; Q=Q(D)* – множество постусловий, определенных на *D; I* – средства, определяющие стратегию обработки правил (в практической реализации – это средства транслятора или интерпретатора).

Приведенная модель может использоваться для определения параллельно выполняемых программных объектов. Программы P_2 и P_3 (см. рис. 2.1) могут выполняться параллельно (при наличии соответствующих вычислительных ресурсов). Этот факт подтверждается тем, что второму запросу select *(x)* удовлетворяют два правила.

Таким образом, средствами логического программирования могут быть описаны условия параллельного выполнения программных компонентов.

2.4.2. БАЗОВЫЕ МОДЕЛИ РАЗРАБОТКИ СОВРЕМЕННЫХ ПС

Развитие сборочного, объектно-ориентированного и других методов программирования шло также по пути создания разного типа моделей, как один из способов определения разных сторон реализуемой предметной области и их трансформации к конечному результату. Приведем набор моделей, возникших в объектно-ориентированном программировании (ООП) и в других видах программирования:

– объектная модель, модель архитектуры (статическая и динамическая), модель окружения (взаимодействия со средой) и использования [35, 36];

– модели метода Шлеера и Меллора (информационная модель, модель поведения системы, модель процессов) [208];

– модели UML (use case), используемые для описания требований, структуры системы и ее преобразование в исполняемый код [35];

– модели процессов разработки ПС в RUP [125] и др.;

– модель ПрО;

– модельно-управляемая разработка – MDD (Model Driven Development), ориентированная на непосредственный перевод описания модели в исполняемый код [140];

– модельно-управляемая архитектура – MDA (Model Driven Architecture) для создания ПС на основе построенных моделей, которые транслируются в конкретную реализацию системы [140].

Объектная модель – одна из первых моделей ООП, получившая реализацию в разных инструментальных системах (CORBA, RUP и др.) В ООП процессы проектирования выполняют формирование моделей на каждом процессе.

Данная модель создается в процессе анализа предметной области, объекты которой определяют реальные ее сущности и операции над ними. В процессе проектирования эта модель дополняется требованиями и функциями, которые программируются средствами C++, Java и др., и передаются заказчику для решения задач ПрО, поиска ошибок и внесения изменений как в состав объектов, так и в методы их реализации.

Средствами ООП создаются: модель ПрО, модель архитектуры, модель окружения и использования. Эти модели воплощаются в программный продукт, который реализует связи между объектами, набор операций и состояний, порождаемых взаимосвязями объектов в модели окружения.

Модель архитектуры включает в себя:

– статическую модель описания структуру системы в терминах классов объектов и взаимоотношений (обобщения, расширения, использования и др.) между ними;

– динамическую модель, которая определяет взаимодействие между объектами системы во время выполнения, инициируется запросами к сервисам объектов и реакцией после их выполнения.

К статической модели относится модель окружения, а к динамической – модель использования. Обе модели взаимно дополнят друг друга через модель связи со средой.

Результат проектирования – ПС, в которой определены все объекты, вызываемые статически или динамически, методы их реализации и выполнения.

Характерная особенность данных моделей – частичное, непоследовательное их преобразование для интеграции выходного кода.

Модели Шлеера и Меллора – это модели (информационная модель, модель поведения системы, модель процессов), которые отображают структуру информации ПрО, поведение системы и процессов обработки данных в системе.

Под информационной моделью понимается совокупность объектов (сущностей) ПрО, их характеристики (атрибуты) и связи между ними. Модель состояний предназначена для отображения динамического поведения системы, связанного с

изменением состояний объектов информационной модели и ЖЦ поведения объектов. Состояние модели зависит от ситуации, обусловленной правилами и линией поведения входящих в нее объектов. Все экземпляры одного класса объектов имеют одинаковое поведение. Модель процессов определяет действия системы, которые связаны с изменениями модели состояний при выполнении некоторых функций. Действие – это реакция на событие, которое инициирует выполняемая функция системы. По каждому действию модели состояний создается диаграмма процесса, на которой реализуются события при выполнении заданных функций системы. Построенные модели реализуют средствами ЯП отдельными частями системы, между которыми устанавливаются связи, зависящие от передаваемых данных или методов объектов.

Модели процессов разработки ПС в RUP. Главный элемент проектирования систем – модель вариантов использования, на основе которой разрабатываются модели анализа, проектирования, реализации, размещения и тестирования системы (рис.2.2).

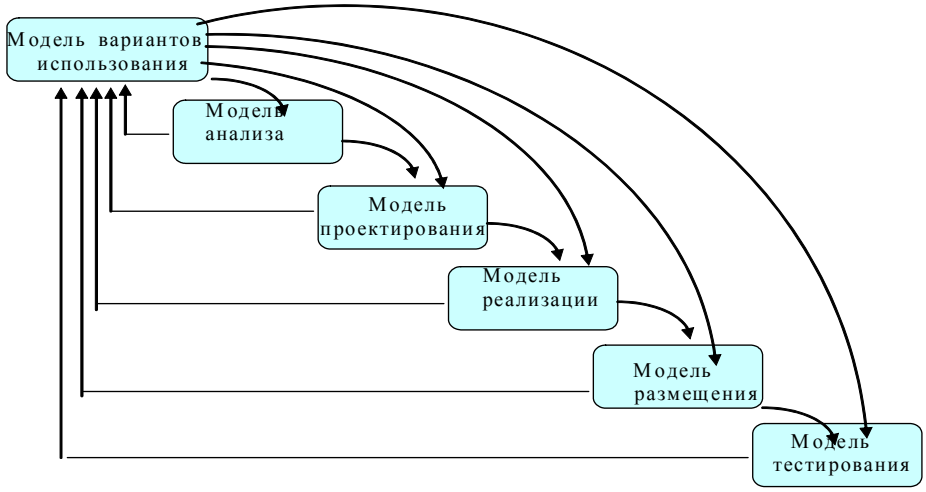


Рис.2.2. Связь моделей в системе RUP

Каждая модель включает в себя входные данные для поиска и спецификации классов и подсистем, подбора и спецификации тестов, а также процесс планирования итераций разработки и интеграции ПС.

Модели представляются разными видами диаграмм, которые могут уточняться или расширять модели предыдущих итераций процесса. Артефакты моделей связаны между собой и совместимы друг с другом. Отношения между моделями не полностью формальные, поскольку отдельные части моделей специфицированы на языке метамодели или UML, а другие описаны на естественном языке. Варианты использования в моделях специфицируют тип отношений между действующим лицом (актером), пользователем и системой. На высоком уровне абстракции методы задают реализацию операций в виде последовательности действий или альтернатив, выполняемых экземплярами вариантов использования.

Модельно-управляемая разработка MDD – современная парадигма разработки ПС, ориентирована на непосредственный перевод описания модели в исполняемый код. Данная парадигма использует:

– модельно-управляемую архитектуру MDA, проектируется средствами UML или его конкретных профилей и выполняется на разнообразных платформах;

– модельно-интегрированные вычисления MIC (Model-Integrated Computing), которые задаются на языке моделирования DSML (Domain-Specific Modeling Language) и преобразуются в платформо-зависимые артефакты [140, 142, 185, 217].

MIC объединяет языки моделирования DSML с системой типов, ассоциируемых с понятиями ПрО, средствами описания требований и поведения объектов, а также трансформационные процессоры и генераторы, которые синтезируют код и XML-описание в выходной код для развертывания (рис.2.3).

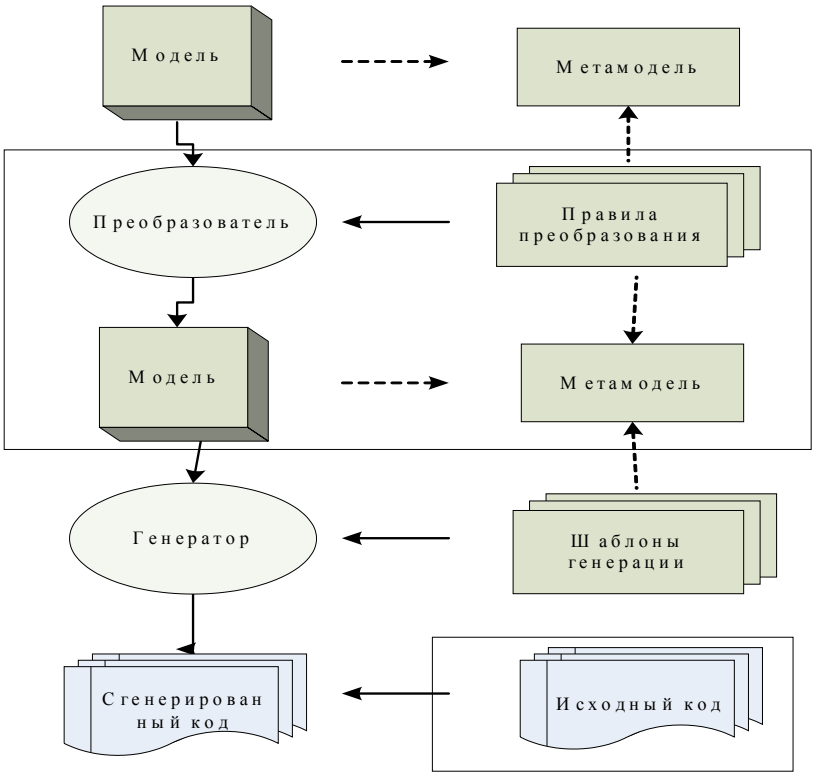


Рис. 2.3. Схема трансформации моделей в DSL в выходной код

Модели преобразуются в исполнимый код для конкретной платформы с помощью специальных преобразователей. Каждая модель преобразуется в другую модель с понижением уровня абстракции, т.е. в модель более конкретную и менее абстрактную. Серия таких преобразований в конечном итоге приводит к исполнимому коду и при последнем преобразовании создается модель-код.

Модель предметной области – это модель домена GDM (Generative Domain Model), которая может включать в себя нескольких моделей для отдельных подобластей ПрО, соответствующих отдельным членам семейства программ.

Каждая из этих моделей отображает понятия и специфику соответствующего члена из семейства систем. На их пересечении формируются общие понятия, характеристики и ограничения в модели характеристик ПрО. Описание каждой модели осуществляется в соответствующем предметно-ориентированном DSL-

языке. Данное описание по специальным правилам трансформируется непосредственно в соответствующий ЯП реализации этой модели (рис. 2.4) или в другой DSL-язык, который затем трансформируется в требуемый ЯП.

Полученное описание в ЯП модели ПрО транслируется к исходному коду среды или платформы, в которой этот код будет функционировать. Иными словами, постепенная трансформация описаний отдельных моделей $ПрО_1$,…, $ПрО_N$ к исполняемому коду определяется платформой выполнения ПС.

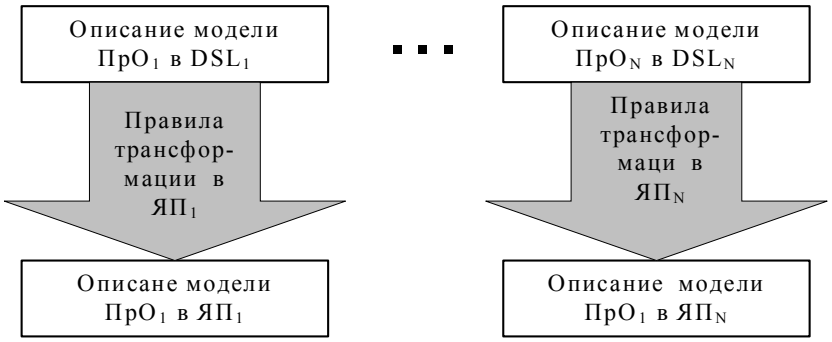


Рис. 2.4. Схема трансформации моделей ПрО

Представление моделей ПрО выполняется по известной модельно-управляемой разработкой MDD. В ней архитектура системы моделируется на двух уровнях – платформо-независимом уровне по модели PIM (Platform Independent Model) и платформо-зависимом уровне по модели PSM (Platform Specific Models). Эта концепция моделирования архитектуры согласно MDA и отображения PIM→PSM составляют идеологию построения семейства систем по модели прикладных систем (Application model) в DSL-языке (рис. 2.5) [140, 142].

Соответственно подходу MDD модели отдельные прикладные системы как члены семейства могут иметь общие задачи и различаться платформами реализации, которые отмечаются точками вариантности в модели семейства систем, полученной путем автоматической трансформации PIM→PSM и без участия разработчика.

Эти точки определяют альтернативные концепции для моделирования отдельных особенностей модели GDM. Источником моделирования являются инвариантные понятия, характеристики членов системы в соответствующей модели характеристик и их критерии, извлекаемые из описания этой модели в DSL.

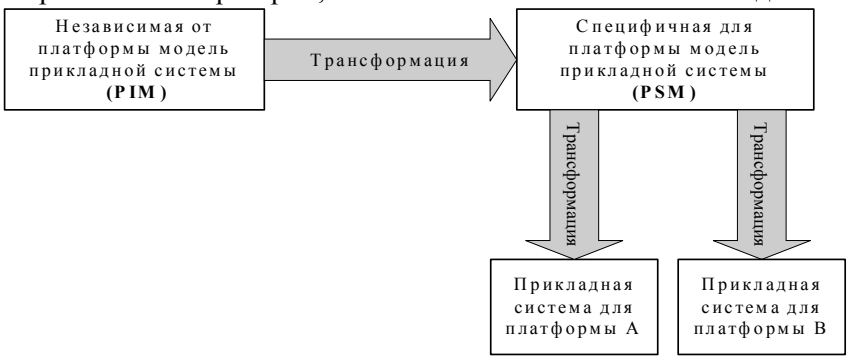


Рис.2.5. Схема трансформации модели прикладной системы к платформам А, В

Модели в UML. В рамках UML модельно-ориентированный подход развивается в направлении трансформации исполняемых моделей (рис.2.6).

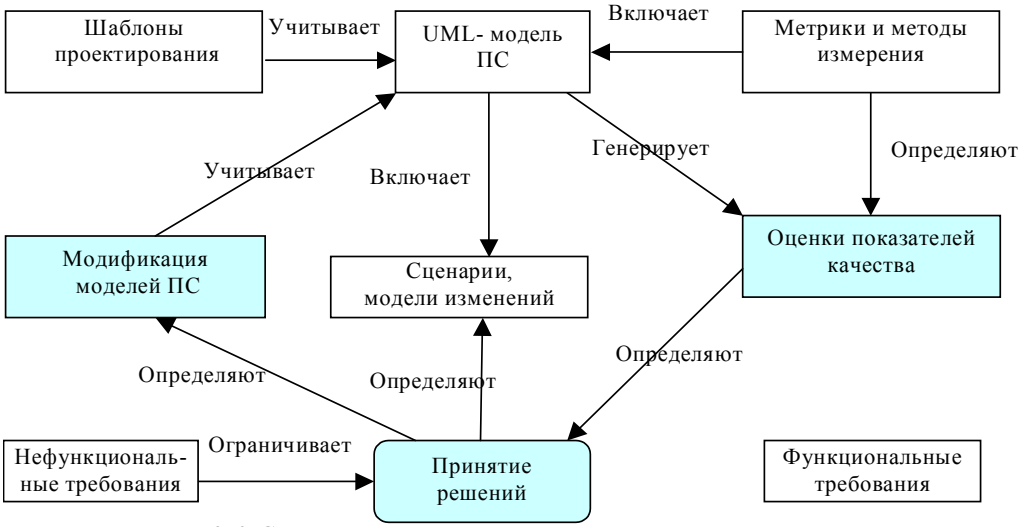


Рис.2.6. Схема моделирования системы с применением UML

Рассматривается модель ПС в языке UML, которая строится на основе шаблонов проектирования с учетом функциональных и нефункциональных требований, которые будут проверяться и оцениваться как показатели качества. Модель – это сценарии работы системы, по которым принимаются решения о результатах выполнения отдельных сценариев. При неудовлетворительном выполнении сценария, проводиться изменение (модификация) соответствующих элементов модели ПС и уточнение требований.

Иными словами, сущность моделирования состоит в выполнение таких действий:

– создание среды разработки ПС в UML с помощью сценариев и шаблонов проектирования;

– проектирование модели интеграции ПС и системы образцов средствами UML;

– выбор метрик и методов оценивания показателей качества объектов в проектируемой системе;

– модификация моделей с учетом функциональных и нефункциональных требований к системе;

– изменение системы путем повторного использования КПИ и методик преобразования передаваемых данных;

– проверка поведения системы осуществляется после выполнения разного рода преобразований;

– обеспечение целостности модели путем внесения изменений в модель.

В настоящее время активно развивается моделирование процессов, их семантика и принципы выполнения. Процессы касаются КПИ и потоков управления при взаимодействии компонентов в среде функционирования. Для

этих целей для UML, обеспечивающего создание модели системы (архитектуры) ПС, а создан язык моделирования процессов SDL (Specification and Description language), как язык графического программирования и воспроизведения моделей процессов сложных систем (www.sdl-forum.org, www.itu.int/itu.doc.itu-t/tec). Обе модели объединяются между собой и трансформируются специальными утилитами к исполняемому коду.

Вывод. В данной главе представлен материал, связанный с проблемами объединения (комплексирования, интеграции) модулей в более сложные программные образования. Основу процесса объединения составляют модели сборочного программирования. Показан новый подход к разработке программных систем – модельный. Дано короткое описание ряда современных моделей, включая объектную модель, модель архитектуры системы MDA, модельно-управляемая разработка MDD, модель генерации домена GDM, UML-модель и др. Основное средство их реализации – трансформация к другому ЯП, промежуточному языку или к выходному. Модели UML трансформируются к новому языку SDL.

Глава 3

КОМПЛЕКСИРОВАНИЕ МОДУЛЕЙ

Модульный принцип – основополагающий в процессе проектирования и разработки программных средств различного назначения. Он обеспечивает декомпозицию исходной задачи на ряд функций, каждая из которых реализуется программным модулем, осуществляющим преобразование исходных данных функций в выходные результаты. Модуль представляет собой самостоятельную, элементарную единицу процесса конструирования по методу сборочного программирования.

При сборке ПС из большого числа модулей с различными характеристиками и свойствами возникает проблема интерфейса модулей, состоящая в организации передачи данных (их импортирование и экспортирование) и управлений между объединяемыми модулями [113, 121, 122].

3.1. ОПРЕДЕЛЕНИЕ МОДУЛЯ И ЕГО СВОЙСТВ

Ранее было отмечено, что общепринятого формального определения понятия модуль не существует. В общем случае под модулем понимается преобразование множества исходных данных X во множество выходных данных Y, задаваемое в виде отображения

$$M : X \rightarrow Y. \qquad (3.1)$$

На сами множества X и Y, а также на отображение M накладывается ряд ограничений и дополнительных условий, позволяющих отделить модуль как самостоятельный программный объект от других классов программных объектов. Теперь будем говорить о том, что модуль обладает множеством свойств и характеристик. Рассмотрим эти свойства относительно трех основных процессов ЖЦ ПС: проектирования, разработки и выполнения.

Процесс проектирования структуры ПС определяет следующие свойства модулей:

– необходимость выполнения одной или нескольких взаимосвязанных функций;

– логическая законченность и обособленность относительно процесса проектирования (модуль должен обеспечивать выполнение необходимой функции независимо от результатов проектирования остальных компонентов ПС);

– независимость одного модуля от других (внутренняя логика данного модуля явно не связана с логикой работы других);

– замена отдельного модуля в ПС без нарушения всей структуры;

– возможность вызова других модулей и возврат управления вызвавшему модулю;

– уникальность именования модуля;

– доступ к общим данным.

На процессе разработки отдельных модулей для них характерны следующие свойства:

раздельная компиляция;

описание на одном из ЯП и представление в виде процедуры, подпрограммы, программной секции;

один вход в модуль;

информация о характеристиках модуля должна содержаться в его описании;

ограничения на размер модуля.

Процесс выполнения ПС предъявляет следующие требования к свойствам модуля:

возможность использования в различных местах программной системы;

отсутствие необходимости сохранять историю своих вызовов и обеспечение повторного входа;

передача данных между модулями через параметры вызова;

изменение как можно меньшего количества передаваемых модулю данных;

унификация механизмов передачи управления и данных между модулями.

Не все свойства одинаково важны в процессе разработки ПС. Некоторые из них, как, например, ограничение на размер модуля, имеют различные оценки. Однако для модульного процесса проектирования характерны две обобщенные тенденции – усиление внутренних связей в модуле и ослабление внешних связей. Хорошо спроектированный модуль обладает значительно более сильными внутренними связями, чем внешними. При выполнении этого условия в процессе комплексирования модули можно рассматривать как неделимые программные единицы с определенными свойствами без учёта их внутренней структуры. Исходя из этого выделим свойства модулей, наиболее важные для процесса комплексирования.

Свойство 1. Логическая законченность и обособленность относительно процесса проектирования, позволяющая представить комплексирование как композицию независимых функций.

Свойство 2. Заменяемость отдельного модуля в программной структуре, позволяющая рассматривать комплексирование как отдельный процесс разработки ПС, не связанный с существенными изменениями всей структуры.

Свойство 3. Свойство возврата управления непосредственно вызывающему модулю, что позволяет сводить процесс комплексирования к объединению пар модулей, взаимодействующих по передаче и возврату управления.

Свойство 4. Обращение одного модуля к другим, что позволяет строить программные структуры, состоящие из модулей, – модульные структуры.

*Свойство 5***.** Раздельная компиляция, при которой все модули в создаваемой программной структуре должны быть внешними. Данное условие позволяет отделить процесс трансляции отдельных модулей от процесса комплексирования**.**

Свойство 6. Возможность использования модуля в различных точках ПС. Наличие этого свойства позволяет избегать дублирования объектов на процессе комплексирования, если к модулю имеется более одного обращения.

*Свойство 7***.** Унификация механизмов передачи управления и данных между модулями. Выполнение этого условия обеспечивает возможность решения задач сопряжения отдельных модулей без учета их внутренней структуры, используя стандартизованные механизмы взаимодействия модулей.

Приведенные свойства позволяют выбрать аксиоматический подход к

определению модуля.

Модулем в рамках метода сборочного программирования называется программный объект, характеризующийся свойствами 1-7.

Данное определение задает границы применения понятия модуль и справедливо для метода сборочного программирования. Операциями над объектами типа *модуль* являются различные функции комплексирования, характеристика которых будет приведена ниже.

3.2. ОПРЕДЕЛЕНИЕ ТИПОВ СВЯЗЕЙ МОДУЛЕЙ

В данном главе рассматриваются определения и характеристики взаимосвязей между модулями, используемые в дальнейшем для описания функций межмодульного интерфейса.

Под *видом связи* между модулями будем понимать: 1) связь по управлению; 2) информационную связь (связь по данным).

1. **Связь по управлению** характеризуется наличием или отсутствием среды ЯП и механизмом вызова.

Среда ЯП определяется как совокупность дополнительных программно-информационных средств окружения модулей, генерируемых компилятором с ЯП и включающих:

системные программные средства процессов выполнения, состоящие из библиотечных программ взаимодействия с операционной системой, вычисления стандартных функций, обработки внутренних структур данных и т. д.;

внутренние структуры данных, включающие различные системные данные, списки ассоциаций идентификаторов для программ блочной структуры, стек в виде последовательности точек передачи и возврата управления, и т. д.;

динамические данные, включающие списки областей используемой и свободной памяти, областей сохранения регистров и т. д.

Механизм вызова определяется способом обращения к вызываемому модулю. Различают стандартный и нестандартный механизмы вызова. Стандартный определяется оператором вызова CALL ЯП, процедурным вызовом и вызовом функций средствами ЯП. Нестандартный представляет собой любой механизм, отличный от стандартного. Используется только при программировании на Ассемблере.

Связь по управлению обладает определенной сложностью, описываемой следующей функцией:

$$CP = K_1 + K_2 \qquad (3.2)$$

где K_1, – коэффициент типа механизма вызова; K_2 – коэффициент перехода от среды ЯП вызывающего модуля к среде ЯП вызываемого.

Для стандартного механизма вызова $K_1=1$, для нестандартного – $K_1= 1 + a$ ($a > 0$). Здесь a зависит от количества отличных от стандартного характеристик вызова. К характеристикам вызова относятся: способ определения точки входа в вызываемый модуль; механизм передачи управления; формирование адреса возврата в вызывающий модуль; доступ к списку параметров; метод сохранения и восстановления регистров вызывающего модуля.

Коэффициент K_2 зависит от количества операций, необходимых для перехода от среды вызывающего модуля к среде вызываемого и наоборот. Аналитического выражения для K_2 не существует, но можно указать параметры, от которых он

зависит. К ним относятся; количество библиотечных модулей, входящих в среду; количество выполняемых ими функций; количество структур данных, входящих в среду; общий объем памяти, занимаемой программно-информационными средствами среды; ЯП вызывающего и вызываемого модулей.

Если вызывающий и вызываемый модули написаны на одном *ЯП,* то $K_2 = 0$. Для остальных случаев $K_2 > 0$.

2. **Информационная связь** определяется как обмен данными между модулями. Различают регулярную и нерегулярную информационную связь. Регулярная характеризуется целенаправленным обменом постоянно определенного множества данных при каждой активизации вызываемого модуля. Этот тип связи реализуется посредством списка передаваемых параметров в операторах вызова. Нерегулярная – непостоянством множества обмениваемых данных или косвенным доступом (через посредников) к информации.

К этому типу связи относится обмен данными посредством глобальных имен или через внешние файлы.

Информационная связь характеризует *сложность* ПС, описываемую следующей функцией:

$$CI = \sum_{i=1}^{n} K_i F(x_i), \qquad (3.3)$$

где K_i – весовой коэффициент для i-го параметра; $F(x_i)$ – функция количества элементов простых типов для параметра x_i.

Коэффициенты $K_i = 1$ – для простых переменных и $K_i > 1$ – для сложных типов данных. $F(x_i) = 1$, если x_i – простая переменная, и $F(x_i) > 1$ – для сложных типов данных. Необходимо отметить, что в (3.3) входят данные, принадлежащие к регулярной и нерегулярной информационной связи. Определение простых и сложных типов данных будет приведено ниже.

3.3. ИНТЕРФЕЙСЫ ПРОГРАММ И ИХ ФУНКЦИИ

В сборочном программировании важное место занимает интерфейс, под которым понимается взаимосвязь программных объектов (модулей, программ, языков и т. п.), обеспечивающая условия их совместного функционирования.

В зависимости от способа реализации интерфейсы могут быть встроенными, внешними и комбинированными [60–63]. Функции встроенного интерфейса реализуются в самих программных объектах. Внешний интерфейс разрабатывается как самостоятельный программный продукт. Функции комбинированного интерфейса частично реализуются в программных компонентах, а частично в виде отдельных программных средств.

В основе построения интерфейсов лежит:

– стандартизация объектов, их свойств и характеристик;

– разработка формализованных правил (механизмов) сопряжения стандартизованных объектов;

– средства автоматизации механизмов связи объектов.

При этом имеет место несколько видов интерфейсов: межъязыковый, межмодульный, межпрограммный, пользовательский.

Межъязыковый интерфейс — это связь различных ЯП по типам, содержащихся

в них данных, методам их организации и способам отображения этих средств соответствующими системами программирования.

Основными функциями межъязыкового интерфейса является решение следующих четырех задач.

1. Обеспечение перехода от среды функционирования одного ЯП к среде функционирования другого. При связи разноязыковых модулей необходимо осуществить переход от среды ЯП вызывающего модуля к среде ЯП вызываемого. Перед возвратом управления необходимы обратные операции.

2. Обеспечение передачи управления между разноязыковыми модулями. Решение данной задачи связано с решением предыдущей.

Если реализованы средства перехода от одной среды функционирования к другой, то передача управления между разноязыковыми модулями не отличается от передачи управления между одноязыковыми. В этом случае задача для межъязыкового интерфейса сводится к контролю за последовательностью обращения от вызывающего модуля к вызываемому.

Необходимо отметить, что решение данной задачи предполагает механизм непосредственного взаимодействия модулей без дополнительных средств обеспечения передачи управления (например, операционной системы).

3. Обеспечение доступа к общим данным. Механизмы доступа относятся к механизмам нерегулярной информационной связи, отличаются и зависят от места расположения информации.

Данные могут находиться в оперативной памяти, и тогда для доступа к ним используются такие средства ЯП, как общие области для ЯП Фортран [41], данные типа EXTERNAL в ПЛ/1 [42, 43] и др. Если данные имеют файловую структуру, то они обычно хранятся во внешней памяти. В этом случае для доступа к ним используются операторы ввода - вывода ЯП высокого уровня и макрокоманды обмена языка типа Ассемблер.

Использование общих данных, расположенных в оперативной памяти, всегда сопряжено с опасностью их разрушения (намеренного или случайного характера) и с неверным выполнением программы. Поэтому при модульном подходе основным механизмом информационного обмена является аппарат передачи данных через параметры оператора вызова CALL ЯП высокого уровня и соответствующей макрокоманды языка Ассемблер. В случае особых режимов обработки общие данные могут передаваться через внешнюю память.

4. Реализация механизма передачи данных через параметры вызова. Этот механизм передачи данных относится к механизму регулярной информационной связи и наиболее стандартизован, так как он применяется практически во всех ЯП, допускающих секционирование программ. Задачей межъязыкового интерфейса в этом случае будет преобразование данных из списка фактических параметров вызывающего модуля к представлению, согласующемуся с описанием формальных параметров вызываемого модуля.

Преобразование направлено на устранение отличий как языковых, так и связанных с реализацией (представлением) данных системами программирования. Оно выполняется перед и после выполнения вызываемого модуля (т. е. осуществляется прямое и обратное преобразования).

Межмодульный интерфейс – это обеспечение связи модулей, записанных в разных ЯП, и управление модульными структурами программ.

Результат анализа свойств модуля для комплексирования позволяет сделать вывод о существовании двух больших групп задач для межмодульного интерфейса. К первой относятся проблемы локального взаимодействия каждой пары модулей (свойства 3, 4, 7, 9), ко второй – проблемы построения и обработки модульных структур (свойства 1, 2, 5, 6, 8). Такое деление упорядочивает разработку межмодульного интерфейса, отделяя задачи сопряжения пар модулей от задач управления модульными структурами в целом. В соответствии с этим в задачу реализации межмодульного интерфейса входят разработка межъязыкового интерфейса как программного компонента, автоматизирующего сопряжение пар модулей, и управление построением и обработкой модульных структур программ.

Сопряжение пар разноязыковых модулей сводится к устранению отличий в:

1) языковых средствах ЯП;

2) описании отдельных модулей;

3) способах представления модулей системами программирования с ЯП.

1. Отличия в языковых средствах ЯП являются следствием неодинаковости синтаксического и семантического представления типов данных ЯП, их функциональных возможностей. К ним необходимо отнести:

механизм конструирования новых типов данных отсутствует в языках Фортран, ПЛ/1, Кобол и имеется в языках Паскаль, Ада, Симула-67, Модула-2, Си, Альфард, CLU [13, 37, 38, 69, 92];

некоторые предопределенные типы отсутствуют в определенных ЯП (например, символьный тип в ЯП Фортран);

представление некоторых предопределенных типов отличается в разных ЯП (логический тип в языке ПЛ/1 представлен как битовая строка);

динамические типы данных отсутствуют в Фортране и Коболе и имеются в языках Паскаль, ПЛ/1, Ада, Симула-67 и др.;

организация внешних файлов различна (ПЛ/1 допускает последовательную, индексно-последовательную и прямую организацию файлов, Фортран не обеспечивает индексно-последовательной организации файлов);

дескрипторы для представления структурных типов данных имеются в некоторых ЯП и не требуются в Фортране и Коболе;

представление некоторых структурных типов отличается в различных ЯП (массивы в Фортране располагаются по столбцам, в других ЯП – по строкам).

2. Проблемы сопряжения, связанные с описаниями модулей, вызваны несоответствием задания формальных и фактических параметров и состоят в следующем:

описания типов данных, областей значений переменных, индексов массивов и т. д. задаются неоднозначно;

с одним формальным параметром сопоставляется несколько фактических и наоборот, что порождается отсутствием структурных типов данных в некоторых ЯП и их обработкой в виде нескольких отдельных компонентов;

изменение порядка следования параметров.

3. К проблемам, связанным с реализацией систем программирования, относятся:

особенности передачи управления (наличие среды функционирования);

различия во внутреннем представлении однородных типов данных для различных систем программирования;

различия в структуре и организации внешней памяти для однородных файлов.

Из приведенного выше следует, что проблема передачи управления между разноязыковыми модулями носит не принципиальный характер, а является следствием реализации конкретных систем программирования с ЯП. Это подтверждается также тем, что аналогичные проблемы для ЯП, реализованных на мини- и микро- ЭВМ, практически отсутствуют или незначительны.

Управление построением и обработкой модульных структур программ как одной из задач межпрограммного интерфейса основывается на реализации следующих четырех функций.

1. Комплектование программных средств различного уровня сложности. Этот процесс является довольно ответственным процессом в разработке ПС и выполняется обычно в несколько шагов. На каждом приходится иметь дело с программными агрегатами различного уровня сложности, готовности, прошедшим различные стадии отладки и т. д. Комплексирование таких разнородных программных компонентов должно происходить на единой методологической основе, составляющей содержание данной задачи межпрограммного интерфейса.

2. Обеспечение построения различных видов программных структур. Существует несколько видов программных структур, каждая из них характеризуется особенностями построения и выполнения программного агрегата. Наибольшее распространение в практике программирования получили простая, оверлейная и динамическая структуры. Построение этих программных структур и составляет вторую задачу управления модульными структурами.

Выполнение операций над модульными структурами. Перед процессом комплексирования необходимо исследовать модульные структуры объектов для обеспечения правильности их функционирования. В частности, эти исследования состоят в выполнении операций определения доступности модулей и их именования, анализа циклов в последовательностях обращений между модулями и др. Поэтому в управлении обработкой модульных структур заключается третья задача межпрограммного интерфейса.

3. Обеспечение тестирования и отладки межмодульных переходов. Данная задача – комплексная, она способствует построению правильных модульных структур. При ее решении используются как средства межъязыкового интерфейса, так и средства управления модульными структурами. Средства сопряжения модулей позволяют обеспечить тестирование передаваемых параметров, а средства управления модульными структурами – отслеживание цепочек выполняемых модулей.

3.4. МОДЕЛИ КОМПЛЕКСИРОВАНИЯ МОДУЛЕЙ

Во второй главе приведены основные модели сборочного программирования – модель информационного сопряжения и модель управления программными объектами. Комплексирование модулей как частная проблема сборочного программирования также может, в общем, описываться этими моделями. Однако конкретное определение объектов – модулей – позволяет детализировать данные модели. Детализация основывается на следующих основных факторах.

1. Использование оператора вызова CALL или ему подобного для обращения к модулю. Этот способ соответствует первому случаю выбора программных объектов в модели управления (см. п.2.4.1).

2. Обмен данными через параметры оператора вызова как основной способ информационного сопряжения. Передаваемые через параметры данные можно

разделить на два типа – входные и выходные. Входные формируются в вызывающем модуле и используются в вызываемом без изменения их значений. Выходные данные формируются в результате работы вызываемого модуля и возвращаются вызывающему.

3. Описание средствами одного из ЯП высокого уровня или Ассемблера всех передаваемых данных. Это также относится к описаниям данных с различным уровнем структурирования (см. п.2.4).

Эти факторы позволяют формализовать описание проблемы комплексирования модулей и построить алгоритм ее решения. Решение проблемы комплексирования модулей заключается в нахождении формализованного метода построения программных интерфейсов и управления модульными структурами.

Рассмотрим формальное описание задачи разработки межъязыкового интерфейса (МЯИ). Зафиксируем класс языков программирования L, состоящий из n ЯП: $L = \{l_a\}$, $\alpha = 1, ..., n$. Пусть имеется пара взаимодействующих модулей, из которых вызывающий написан на l_a-языке, а вызываемый – на l_β-языке. Обозначим через $V = [v^1_f, v^2, \ldots, v^k]$ список фактических параметров вызывающего модуля, а через $F = \{f_1, f_2, \ldots, f^{k_1}\}$ – список формальных параметров вызываемого модуля. В общем случае $k \neq k_1$.

Разделим V и F в соответствии со множествами входных и выходных параметров: $V = V_i \bigcup V_0$, $F = F_i \bigcup F_0$ (i соответствует множеству входных, а 0 – множеству выходных параметров). При этом $V_i \sim F_{ti}$, а $V_0 \sim F_0$ (~ обозначает знак соответствия).

Исходя из этих обозначений задача МЯИ заключается в преобразовании типов данных из V_i в соответствующее представление в F_i и F_0 - в представление V_0. При этом принципиальной разницы между преобразованиями типов данных от V_i к F_i и от F_0 к V_0 не существует (в дальнейшем индексы i и 0 при соответствующих множествах будем опускать).

В общем случае, как отмечено в п. 2.4.1, элементу множества F может соответствовать несколько элементов множества V и наоборот, что объясняется различием в типах данных для фактических и формальных параметров. Поэтому отображение для отдельных элементов множеств $V=\{v^1, v^2, ..., v^k\}$ и $F = \{f^1, f^2, \ldots, f^k\}$ может оказаться неоднозначным. Для построения однозначного отображения необходимо провести разбиение множеств таким образом, чтобы каждому подмножеству из V соответствовало только одно подмножество из F. Проведем построение этого разбиения. Для каждого $f^i \in F$ рассмотрим его полный прообраз $V^i \in V$. Различные прообразы V^i и V^j могут иметь или не иметь одинаковые элементы. Если $V^i \cap V^j \neq \Theta$, то объединим V^i и V^j в одно подмножество. Соответственно будет проведено объединение в одно подмножество элементов F^i и F^j. Данная процедура применяется до тех пор, пока не будет исчерпано множество V. В результате получим два семейства подмножеств $\Pi = \{V^1, V^2 \ldots, V^m\}$ и $\Phi = \{F^1, F^2, \ldots, F^m\}$ таких, что

$$\bigcup_{t=1}^{m} V^t = V, \quad V^t \bigcap V^{t_1} = \Theta \quad \text{при} \quad t \neq t_1 \tag{3.4}$$

$$\bigcup_{t=1}^{m} F^t = F, \quad F^t \bigcap F^{t_1} = \Theta \quad \text{при} \quad t \neq t_1 \tag{3.5}$$

для которых существует однозначное отображение, записываемое в виде

$$A : \Pi \rightarrow \Phi. \tag{3.6}$$

В зависимости от количества элементов во множествах V^t и F^t имеют место следующие случаи:

1) $|V^t| = |F^t| = 1$. Отображение A для данных подмножеств включает операции преобразования типов данных.

2) $|F^t| > 1$ и $|V^t| = 1$. Это означает, что одному фактическому параметру структурного типа данных соответствует несколько формальных параметров скалярных типов или структурных с меньшим уровнем структурирования. Отображение A включает операции селектора отдельных компонентов ипреобразования типов данных.

3) $|V^t| > 1$ и $|F^t| = 1$. Это означает соответствие нескольких фактических параметров одному формальному. Отображение A содержит операции преобразования типов и конструирование структурного типа с более высоким уровнем структурирования, чем у передаваемых параметров.

4) $|V^t| > 1$ и $|F^t| > 1$. Это свидетельствует о существовании глубокой связи вызывающего и вызываемого модулей, которая зависит от внутренней логики функционирования модулей. Такие связи противоречат свойствам модулей, и поэтому данный случай не поддается формальному анализу при комплексировании модулей.

На основе проведенного анализа свойств отображения A выделим три класса операций для информационного сопряжения модулей.

1. **Операции преобразования типов данных (P).** Позволяют осуществлять непосредственное преобразование типа данных T_a^t в T_β^t без дополнительных операций изменения уровня структурирования [85, 86]. Операцию преобразования типов данных запишем в виде : $P^{tq}{}_{a\beta} = (T_a{}^t , T^q{}_\beta)$

Здесь данные типа T_a^t преобразуются в $T^q{}_\beta$, a и β соответствуют языкам l_a и l_β. Предполагается, что множество типов данных каждого ЯП упорядочено и индексы t и q определяют конкретные элементы этого множества. Для ЯП, имеющих средства конструирования новых типов, t и q будут функциями от других индексов, и упорядоченность типов может определяться тем, что новый тип t будет конструироваться из типов, для которых индексы не больше t.

Каждый ЯП имеет определенное множество предопределенных типов данных и базовых операций конструирования, что определяет основу всего множества типов. Новый тип будет иметь индекс, функционально зависящий от индексов предопределенных типов и конкретных операций конструирования.

2. **Операции селектора (S).** Используются для выбора из структурного типа его отдельных компонентов с меньшим уровнем структурирования. Механизмы реализации этих операций отличны от аналогичных, имеющихся в ЯП, так как они не должны изменять структуры данных, непосредственно обрабатываемых в модулях (выполняются свойства модулей, связанные с неизменностью их внутренней структуры при комплексировании) [47].

3. **Операции конструирования структурных типов** (C). Данные операции являются обратными по отношению к операциям селектора. Их механизмы конструирования отличны от аналогичных операций, имеющихся в ЯП.

Множество рассматриваемых операций охватывает как преобразования типов данных для ЯП из класса *L,* так и необходимые функции конструирования структурных типов и выбора их отдельных компонентов. Детальное рассмотрение показывает, что для данного множества операций полнота отсутствует. Причины этого рассматриваются ниже.

Исходя из приведенных определений сформулируем постановку задачи создания МЯИ [61–63, 85, 86, 112]..

Дано: класс ЯП $L = \{l_1, l_2, \ldots, l_n\}$ и для каждого из ЯП известны множества типов данных и операций конструирования новых типов.

Необходимо:

1. Построить множества операций преобразования типов данных $P = \{ P^{tq}_{\alpha\beta} \}$, операций селектора S и конструирования C для структурных типов.

2. Для каждой пары взаимодействующих модулей провести разбиения множеств фактических и формальных параметров в соответствии с формулами (3.4) и (3.5) и построить отображение A на основе P, S и C.

Если отображение A построить не удается, то это означает, что МЯИ не обеспечивает сопряжение данной пары модулей с соблюдением свойств модулей. Возможно, сопряжение может быть реализовано с нарушением рассматриваемых свойств. Последнее замечание позволяет использовать МЯИ для определения качества создаваемой модульной структуры.

Определим аппарат описания типов данных ЯП. Каждый тип данных характеризуется множеством значений, которые могут принимать переменные этого типа, и множеством операций над этими переменными. Поэтому наиболее подходящим методом, как было отмечено в гл. 2, является описание типов данных как алгебраических систем.

Введем обозначение для алгебраической системы, соответствующей некоторому типу данных: $U = <X, \Omega>$. Здесь X – множество значений рассматриваемого типа, Ω – множество операций над объектами данного типа.

Тип самой алгебраической системы определяется в соответствии со структурой множества Ω. В общем случае результаты некоторых операций из Ω могут не принадлежать X, и тогда U может рассматриваться как частичная алгебраическая система.

Выбор данного метода описания обусловлен содержанием задачи информационного сопряжения для МЯИ. Операции преобразования типов $P^{tq}_{\alpha\beta}$ должны обеспечивать не только однозначное соответствие множеств значений преобразуемых типов данных в вызывающем и вызываемом модулях, но и одинаковую интерпретацию операций над данными в этих модулях. При этом должно осуществляться как прямое преобразование данных от вызывающего к вызываемому модулю, так и обратное.

При таком подходе операции преобразования $P^{tq}_{\alpha\beta}$ соответствуют изоморфным отображениям одной алгебраической системы в другую. Отображение A, рассмотренное ранее, есть набор изоморфизмов, включающий изоморфное отображение множеств фактических и формальных параметров и изоморфные отображения алгебраических систем для типов передаваемых параметров. Исходя из этого первую часть постановки задачи формулируем в следующем виде.

Для заданного множества алгебраических систем $\sum\{U_a^t\}$ (U_a^t - соответствует типу данных t в α-м ЯП из рассматриваемого класса L языков) построить все возможные изоморфные отображения между элементами множества $\sum$.

Таким образом, задача построения МЯИ будет заключаться в выполнении следующих действий.

1. Определение класса ЯП и построение множества $\sum\{U_a^t\}$ при условии, что типы данных T^t_a описываются как алгебраические системы U^t_a.

2. Нахождение изоморфных отображений между элементами множества $\sum$, результатом которого для U^t_a в U^t_β могут быть;

явный вид изоморфизма;

множество изоморфных отображений с определенными общими свойствами и ограничениями;

доказательство отсутствия существования изоморфного соответствия между анализируемыми алгебраическими системами.

Если имеет место последний результат, то полученное множество операций $P = \{P^{tq}_{\alpha\beta}\}$, соответствующее множеству изоморфных отображений, будет характеризоваться неполнотой, как было отмечено выше.

3. Для всех структурных типов данных строятся множества операций селектора S и конструирования C. Эти операции базируются на общем подходе к структурной организации данных без учета их особенностей в конкретных ЯП. Необходимость построения множеств S и C определяется задачами и возможностями конкретного МЯИ, и в частном случае эти множества могут быть пустыми.

4. Для каждой пары взаимодействующих модулей должно быть выполнено:

построение изоморфного отображения между множествами фактических V и формальных F параметров (построение множеств П и Ф);

выбор необходимых операций селектора и конструирования из множеств S и C;

выбор необходимых изоморфных преобразований из множества P.

Предложенная схема построения МЯИ базируется на типах данных ЯП, которые рассматриваются ниже.

3.5. ТИПЫ ДАННЫХ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ (ЯП)

Для анализа основных типов используется теория структурной организации данных [41, 201]. Она базируется на формализованном подходе к определению типов, основанном на аксиоматизации каждого типа и правилах выполнения операций над объектами. Система аксиом определяет структуру множества значений типа, принадлежность ему отдельных элементов, их основные свойства, связь с другими типами данных.

Для каждой операции, выполняемой над переменными рассматриваемого типа, определяются типы операндов и результата. Теория структурной организации данных не ориентирована на применение в конкретном ЯП. Среди существующих ЯП основные положения данной теории воплощены в Паскале, Модула-2, Ада и др. В дальнейшем для обозначения типов данных используются синтаксические конструкции языка Паскаль.

Существуют четыре предопределенных типа данных: целый (INTEGER); вещественный (REAL); булевой (BOOLEAN); символьный (CHARACTER). Они характеризуются тем, что на уровне архитектуры большинства ЭВМ имеются средства для обработки соответствующих типов данных и они существуют практически во всех ЯП. Остальные являются производными и образуются с помощью средств конструирования новых типов данных.

Все типы данных делятся на простые и структурные. К простым относятся перечислимые и числовые, к структурным – массивы, записи, множества, списки,

последовательности и т. д. В настоящей работе подробно рассматриваются типы данных: перечислимые на основе булевого и символьного типов; числовые на основе целого и вещественного типов; массивы и записи как объекты структурных типов.

Другие типы, представленные в теории структурной организации данных, являются менее распространенными в ЯП и рассматриваются без подробного анализа.

3.5.1. ПРОСТЫЕ ТИПЫ ДАННЫХ

К простым типам относятся перечислимые и числовые. Общее обозначение перечислимого типа имеет вид type $T = \{x_1, x_2, ... x_n\}$. Здесь T – имя типа, а $x_1, x_2, ..., x_n$ – имена всех значений типа T, образующих множество значений X. Операции над перечислимыми типами включают бинарные операции отношения и унарные операции pred и succ, определяющие соответственно предыдущий и последующий элементы во множестве X. Все операции отношения ($<, \leq, >, \geq, =, \neq$) при построении алгебраических систем будут заменены одной ($\leq$), определяющей линейную упорядоченность множества X.

Примерами перечислимых типов могут служить булевый и символьный типы. Множество значений булевого типа состоит из двух элементов – false и true. Множество операций Ω кроме перечисленных выше, включают операции булевой алгебры &, V, ¬. Запишем алгебраическую систему, соответствующую булевому типу:

$$U^b = <X^b, \Omega^b>,$$
$$X^b = \{\text{false, true}\}, \qquad (3.7)$$
$$\Omega = \{\&, V, \neg, \text{pred, succ}, \leq\},$$

U^b имеет тип (2, 2, 1, 1, 1;2) согласно арности соответствующих операций и предикатов.

Булевый тип в ЯП записывается в виде type T^b = (false, true). Если он стандартный (Boolean), то это описание в текстах модулей может опускаться.

Множество значений X символьного типа состоит из букв, цифр и специальных символов (знаков арифметических операций, знаков препинания и т. д.). Множество операций совпадает со множеством операций для любого перечисленного типа. Алгебраическая система U^c, соответствующая символьному типу, имеет вид

$$U^c = <X^c, \Omega^c>,$$
$$X^c = \{\ldots, \text{'A'} \ldots, \text{'X'} \ldots, \text{'0'}, \text{'1'}, \ldots, \text{'9'}\}, \qquad (3.8)$$
$$\Omega^c = \{\text{pred, succ}, \leq\}.$$

Множество X упорядочено согласно внутреннему представлению символов для ОС ЕС.

Алгебраическая система U^c имеет тип <1,1;2> согласно арности соответствующих операций и предикатов.

Символьный тип средствами ЯП записывается следующим образом: type T^c = (..., 'A', ..., 'X' ...., '0', ..., '9'). Если он стандартный (CHAR), то это описание в модулях может быть опущено. Операция ord, присваивающая каждому символу его порядковый номер во множестве X^c, и char, определяющая по порядковому номеру символа его значение, являются по существу операциями преобразования типов и поэтому во множество Ω^c не включены.

Для перечисленных типов характерны следующие аксиомы [201], которые в дальнейшем будут использоваться для анализа операций преобразования типов

данных:

$$X.\min \in X,$$
$$X.\max \in X, \qquad (3.9)$$
$$(\forall x \in X) \& (x \neq X.\max) \Rightarrow \text{succ}(x) \in X.$$
$$(\forall x \in X) \& (x \neq X.\max) \Rightarrow \text{succ}(x) \neq X.\min$$

Здесь X.min и X.max обозначают соответственно минимальный и максимальный элементы множества X.

Практическое использование числовых типов всегда содержит ограничения, определяемые архитектурой ЭВМ (конечное значение количества разрядов слова памяти для представления чисел) или явным описанием в модулях (для ограничения диапазона значений отдельных элементов). Поэтому все числовые типы без нарушения общности анализа могут быть рассмотрены как отрезки. В общей форме отрезок записывается в виде type T = (X.min … X.max). Здесь X.min и X.max обозначает соответственно минимальный и максимальный элементы отрезка. Для любого $x \in X$ выполняется условие x.min < x < x.max. Для стандартных числовых типов (INTEGER и REAL) приведенное выше описание в модулях может быть опущено. В этом случае элементы X.min и X.max не определены и зависят от конкретной реализации транслятора с ЯП на конкретном типе ЭВМ.

Над переменными целого типа и типов, для которых целый тип является базовым, выполняются те же операции, что и в случае перечисленных типов. Кроме того, добавляются операции целочисленной арифметики: унарный минус, +, -, ×, div (целочисленное деление) и mod (получение остатка от деления). Алгебраическая система G^i, соответствующая целому типу, будет иметь вид

$$U^i = \langle X^i, \Omega^i \rangle,$$
$$X^i = \{X^i.\min, X^i.\max + 1, \ldots X^i.\max\}, \qquad (3.10)$$
$$\Omega^i = \{+, \times, \text{div}, -, \leq\}.$$

Во множестве Ω^i операция « - » соответствует унарному минусу. Остальные операции выражаются через операции, включенные в Ω. Алгебраическая система G^i имеет тип (2, 2, 2, 1; 2) согласно арности операций и предикатов. Форма записи целого и отрезков целого типа в ЯП имеет вид

$$\text{type } T^i = (X^i.\min, \ldots, X^i.\max).$$

Над переменными вещественного типа и типов, для которых вещественный тип является базовым, выполняются операции отношения и обычные арифметические операции для действительных чисел (унарный минус, +, -, ×, /). Алгебраическую систему G^r, соответствующую вещественному типу, запишем следующим образом:

$$U^r = \langle X^r, \Omega^r \rangle,$$
$$X^r = \{x \mid X^r.\min \leq x \leq X^r.\max\} \qquad (3.11)$$
$$\Omega^r = \{+, \times, /, -, \leq\}.$$

Во множестве Ω^r операция «–» соответствует унарному минусу. Алгебраическая система Ω^r имеет тип (2, 2, 2, 1; 2) согласно арности операций и предикатов.

Форма записи вещественного и отрезков вещественного типа в ЯП имеет вид

$$\text{type } T^r = (X^r.\min, \ldots, X^r.\max).$$

Необходимо сделать замечание относительно порядка выполнения операций над любыми типами:

все операнды приводятся к базовому типу;

операция выполняется, как над объектами базового типа;

для полученного результата выполняется обратный переход от базового к исходному типу.

Если результат принадлежит множеству значений данного типа, то операция выполняется верно. В противном случае результат операции не определен.

В дополнение к аксиомам (3.9) для числовых типов будут использоваться следующие:

$$(\forall x \in X) \Rightarrow T(T^0(x)) = x,$$
$$(\forall x_1 \in X) \& (\forall x_2 \in X) \Rightarrow (x_1 \le x_2) \equiv (T^0(x_1) \le T^0(x_2)). \quad (3.12)$$

Здесь T^0 обозначает базовый тип для типа T. Операции $T^0(x)$ и $T(x)$ определяют преобразование значения к соответствующему типу. В этих обозначениях выполнение арифметических операций для числовых типов будет определяться следующим образом ($\oplus$ – любая двухместная арифметическая операция):

$$(\forall x_1 \in X) \& (\forall x_2 \in X) \Rightarrow (x_1 \oplus x_2) \equiv T(T^0(x_1) \oplus T^0(x_2)). \quad (3.13)$$

Операции сравнения числовых типов выполняются согласно аксиоме (3.12).

3.5.2. СТРУКТУРНЫЕ ТИПЫ ДАННЫХ

Отличительной особенностью структурных типов данных в сравнении с простыми является то, что они содержат несколько упорядоченных элементов, обработка которых проводится как над целыми объектами, так и на уровне отдельных элементов. Структурные типы строятся из базовых типов и отличаются функциями конструирования и механизмами обработки. В качестве основных структурных типов в работе рассматриваются массивы и записи [201].

Массивы

Функция конструирования *массива* на основе базовых типов состоит в определении отображения из множества индексов на множество значений его элементов:

$$M : I \rightarrow Y. \quad (3.14)$$

Здесь I – множество индексов, Y – множество значений элементов массива. В общем случае отображение M может не быть взаимно однозначным, если в различных элементах массива (элементы с разными индексами) содержатся одинаковые значения. Множество I является множеством значений перечислимого типа или отрезка целого типа. Элементы множества Y могут быть элементами любого типа, допускаемого в теории структурной организации данных.

Над массивами могут выполняться следующие операции:

отношение для упорядоченных массивов (определяется как совокупность операций отношения для всех элементов массивов);

сложение и вычитание однотипных массивов, т. е. массивов с одним и тем же множеством индексов (определяется как совокупность соответствующих операций над всеми элементами массивов с одинаковыми индексами);

умножение двухмерных массивов по правилам умножения матриц.

Операция умножения накладывает ограничения на область значений индексов массивов, связанных с правилами умножения матриц, и не является общей для всех

типов массивов. Операции сложения и вычитания выполняются только для числовых массивов. Поэтому они не входят в состав множества общих операций над массивами. Алгебраическая система, соответствующая типу данных *массив,* имеет следующий вид:

$$U^{a} = < X^{a}, \Omega^{a} >,$$

$$X^{a} = \{ x \mid (\forall x_1 \in X^{a}) \,\&\, (\forall x_2 \in X^{a}) \Rightarrow I(x_1) =$$

$$I(x_2)) \,\&\, (Y(x_1) \cup Y(x_2) \subset \overline{Y}(X^{a}))\}, \; \Omega^{a} = \{ \leq \}. \qquad (3.15)$$

Здесь *I{x)* обозначает множество индексов для массива *x, Y(x)* - множество значений элементов массива *x,* $\overline{Y}(X^{a})$ определяет множество значений элементов для всех массивов, принадлежащих рассматриваемому типу. Второе выражение (3.15) обозначает, что к данному типу принадлежат только те массивы, у которых множества индексов совпадают, а множество значений их элементов принадлежат одному и тому же множеству, характеризующему рассматриваемый тип. В обозначениях (3.15) отображение (3.14) примет следующий вид *(I* - постоянно для всех *x):*

$$x : I \rightarrow Y(x), \quad Y(x) \subset \overline{Y}(x). \qquad (3.16)$$

Множество Ω^{a} состоит только из одного предиката. Поэтому алгебраическая система G^{a} фактически является алгебраической моделью типа (2).

Тип данных *массив* в ЯП записывается в вид:

$$\text{type } T^{a} = \text{array } T(I) \text{ of } T(\overline{Y}),$$

где T^{a} – тип данных *массив; T (I)* – тип данных индексов массива; $T(\overline{Y})$ – тип данных для множества значений элементов массивов типа T^{a}.

Операции, определенные в (3.15), выполняются над массивами как над единым структурным значением. Кроме того, над элементами множеств *I* и *Y* могут выполняться операции, соответствующие их типам данных. Определим операцию селектора для элементов массива. Пусть $I' \subset I$. Через *E* обозначим вложение $I' \rightarrow I$. Отображение $E \bullet M : I' \rightarrow Y$ называется ограничением отображения *M* на *x* и обозначается $M \mid I'$ [112–115]. Если *I'* состоит из одного элемента $I' = = \{k\}$, то $M|\{k\}$ будет определять элемент множества *Y,* соответствующий индексу *k.* Заменяя *M* на *x* (согласно определению массива), получим обозначение операции селектора для элементов массива $x|\{k\}$. В ЯП обычно элемент массива обозначается в виде *x [k].*

Необходимо отметить, что в предыдущем анализе рассматривались одномерные массивы. Многомерные массивы определяются рекурсивно. Данные типа *T (Y)* в описании массива T^{a} в свою очередь могут быть массивами и т. д. При этом подходе последовательность предложений

$$\text{type } T^{a} = \text{array } T(I^{1}) \text{ of } T(Y^{1}),$$

$$\text{type } T(Y^{1}) = \text{array } T(I^{2}) \text{ of } T(Y^{2})$$

эквивалентна следующей записи:

$$\text{type } T^{a} = \text{array } T(I^{1} \times I^{2}) \text{ of } T(Y^{2}).$$

В последнем предложении множество индексов массивов, принадлежащих типу T^{a}, представлено в виде прямого произведения множеств значений для типов $T(I^{1})$ и $T(I^{2})$. Операция селектора будет определяться в виде $\{\{x\}|\{i\}\}|\{j\}$, если $i \notin I^{1}$ и $j \notin I^{2}$. В ЯП элемент массива в этом случае будет обозначаться через *x[i][j]* или *x[i,j].*

Запись

Структурный тип *запись,* как и *массив,* состоит из нескольких компонентов, которые могут быть разнородными, т.е. принадлежать различным простым или структурным типам. Функция конструирования записей представляет конкатенацию отдельных компонентов. Множество значений типа *запись* является прямым произведением множества значений ее компонентов. Ко множеству операций, выполняемых над записями, относятся только операции отношения. При этом предполагается, что сравниваться могут только однотипные структуры (типы компонентов сравниваемых записей и их порядок следования одинаковые) и для каждого компонента записи выполняемая операция отношения интерпретируется как соответствующая операция для типа, соответствующего данному компоненту.

Пусть запись состоит из *n* компонентов. Каждый *m*– компонент *(m* = 1, 2, ... , *n)* имеет тип T^{v_m} и ей соответствует алгебраическая система $U^{v_m} = <X^{v_m}, \Omega^{v_m}>$. Индекс v_m соответствует одному из индексов для рассматриваемых в работе типов данных. Алгебраическая система для *записи* будет иметь вид

$$U^z = <X^z, \Omega^z>,$$

$$X^z = \{ x \mid (x=x^{v_1} \times \ \dots \ \times x^{v_n}) \ \& \ (x^{v_1} \in X^{v_1}) \ \& \dots \& \ (x^{v_n} \in X^{v_n}) \}, \ \Omega^z = \{ \le \}. \qquad (3.17)$$

Аналогично массиву алгебраическая система G^z является моделью типа (2).

Общая форма представления типа для записи имеет вид

$$\text{type } T^z = (S_{v1}: T^{v_1}; \dots S_{vn}: T^{v_n}).$$

Здесь S_{v_1}, ..., S_{v_n} – селекторы, а T^{v_1}, ..., T^{v_n} – типы данных для компонентов записи. Средствами ЯП запись $(S_{v_1}$, ..., $S_{v_n})$ - имена компонентов записи) описывается следующим образом:

$$\text{type } T^z = \text{record}$$
$$S_{v_1} : T^{v_1}$$
$$:$$
$$S_{v_n} : T^{v_n}$$
$$\text{end.}$$

Операции, введенные в (3.17), выполняются над записью как над единым структурным значением. Для обработки отдельных компонентов введем операцию селектора, аналогично соответствующей операция для массива. Пусть I = $\{ S_{v_1}, \dots, S_{v_n} \}$, $I' \subset I$. Обозначим через *E* вложение $I' \rightarrow I$. Пусть *x* - переменная типа *запись* и определяется согласно (3.17). Введем в рассмотрение множество $X^v = \{ X^{v_1}, \dots, X^{v_n} \}$. Тогда между *I* и X^v существует однозначное соответствие $M : I \rightarrow X^v$. Ограничение отображения *M* на *I′* обозначим через $M \mid \{ S_{v_m} \}$. Если *I* состоит из одного элемента $I' = \{ S_{v_m} \}$, то $M \mid \{ S_{v_m} \}$ будет определять S_{v_m}–ю запись компонента. Заменяя *M* на *x,* получаем $x \mid \{ S_{v_m} \}$. В ЯП компонентные записи обозначаются в виде $x \bullet S_{v_m}$, где S_{v_m} имя соответствующего компонента.

Проведенный анализ относится к фиксированным записям. Для вариантных

записей, последовательность компонентов которых определяется специальным признаком, можно поступить следующим образом. Признак является переменной перечислимого типа с конечным множеством значений. Каждому конкретному значению признака соответствует определенный вид записи. Поэтому вместо одной вариантной записи рассматривается семейство фиксированных записей для каждого значения признака. Семейство конечное, так как конечно число элементов перечислимого типа. Анализ вариантной записи будет сведен к перебору полученного семейства и обработке конкретной фиксированной записи. Поэтому в дальнейшем без снижения общности результатов будут рассматриваться только фиксированные записи.

3.5.3. СЛОЖНЫЕ ТИПЫ ДАННЫХ

Выше были рассмотрены основные структурные типы данных – массивы и записи, которые встречаются в большинстве ЯП. Кроме них находят применение и другие: множества, объединения, динамические объекты данных, списки, последовательности, стеки, деревья и др. Некоторые из этих типов являются стандартными в конкретных ЯП, другие реализуются путем программного моделирования соответствующих структур и операций над ними. В данной работе для дополнительных структурных типов приводится информация описательного характера и детальный анализ не производится. Это вызвано следующими причинами [3, 4, 201]:

1. Некоторые типы являются собственностью одного или малого числа ЯП и не имеют аналогов в других ЯП.
2. Анализ некоторых типов, моделируемых средствами определенного ЯП, может быть сведен к анализу базовых типов, подробно рассмотренных в данной работе.
3. Реализация некоторых типов и выполнение операций над ними недостаточно формализованы.

Необходимо отметить, что чем сложнее тип данных, тем разнообразнее операции над объектами этого типа и тем менее формализовано представление этих операций в ЯП.

Множества

Примером ЯП, в котором реализован аппарат множеств, является Паскаль [48]. Общая форма записи типа данных *множество* следующая:

$$\text{type } T = \text{powerset } T^0,$$

где T определяет тип множества; T^0 является базовым типом для элементов множества.

Обычно T – перечислимый или целый тип. Для типа T реализованы все основные операции над множествами как математическими объектами – объединение, пересечение, разность, операции включения, определения тождественности и нетождественности. Операции селектора представляют собой выбор элемента типа T^0 из объекта типа T. Операцией конструирования является формирование из одного или нескольких элементов типа T^0 объекта типа T.

Объединения

Общая форма записи для объединения имеет вид [82]

$$\text{type } T = \text{union } (T^{v_1}, \ldots, T^{v_n}),$$

где T – тип объединения; $T^{v_1}, \ldots, T^{v_n}$ – базовые типы.

В общем, на базовые типы ограничения не накладываются. Любой объект типа T имеет два компонента – значение и признак, по которому определяется один из типов $T^{v_1}, \ldots, T^{v_n}$ для данного значения. Механизм реализации объединения подобен механизму реализации вариантных записей. Отличие состоит в том, что сам признак скрыт в отличие от признака вариантной записи, в которую он входит в качестве отдельного компонента. Все операции над объектами типа аналогичны операциям над вариантными записями.

Динамические объекты данных

Этот тип данных в различных вариантах реализован во многих ЯП: Паскаль, Ада, ПЛ/1, Си и др. Общая форма записи для этого типа имеет вид

$$\text{type } T = \text{pointer to } T^0,$$

где T – определяет ссылочный тип; T^0 – базовый тип.

Базовым может быть любой тип. Объект типа T представляет адрес объекта типа T^0. Фактически ссылочный тип не является структурным, так как переменные этого типа содержат только одно значение, как и объекты простых типов. Однако использование ссылочного типа отличается от использования простых типов. Операции над ссылочными типами не формализованы. Например, язык Паскаль допускает только одну операцию – настройку на элемент базового типа (аналогично операции присваивания). В то же время язык Си допускает над ссылочными типами операции адресной арифметики.

Списки

Списки являются конструкциями ЯП Лисп или могут реализовываться программным моделированием. Во втором случае элемент списка описывается как запись, содержащая одну или несколько компонентов ссылочного типа, которые обеспечивают связь между элементами списка. К последним в этом случае могут быть применены операции, которые аналогичны операциям над фиксированными записями. Кроме них существуют операции, применяемые к целому списку: выбор начального элемента, получение остатка списка, соединение списков, их сравнение, инвертирование, поиск элементов (атомов) в списке и др. Необходимо отметить, что в ЯП Лисп эти операции принадлежат к средствам самого языка. Для языков, не имеющих стандартных средств обработки списков, аналогичные операции должны быть реализованы в виде отдельных процедур. Все это не позволяет определить фиксированное множество стандартных операций над списками, характерное для всех или большинства ЯП.

Последовательности

Общая форма имеет вид: $\text{type } T = \text{sequence } T^0$,

где T – тип последовательности; T^0 – базовый тип.

Последовательность является одним из вариантов списка, у которого каждый элемент содержит только одну ссылочную переменную, что обеспечивает

одностороннюю связь. Операции над последовательностями аналогичны операциям над списками. Одной из разновидностей последовательности является строка. Для строки каждый элемент кроме ссылочной переменной, содержит компонент символьного типа. Обработка символьных строк допускается в языке Снобол.

Стеки

Стек представляет собой специальным образом организованную память с дисциплиной обработки LIFO (последним прошел – первым обработан). Обычно отдельный элемент стека принадлежит простому типу. Однако, используя средства программного моделирования, можно реализовать обработку стеков, содержащих элементы любых типов. На практике стеки реализуются в виде массивов или списков. В отличие от других дополнительных структурных типов, множество операций над стеками фиксировано и включает: инициализацию стека, занесение элемента в стек, выбор из стека, анализ элемента, находящегося на вершине стека.

Операции над стеками могут быть реализованы па аппаратном уровне, стандартными средствами ЯП или программным моделированием.

Деревья

Деревья являются списковыми структурами и служат для представления графов или других аналогичных объектов. Множество операций над деревьями аналогично множеству операций над списками. Реализация этих операций зависит от конкретных приложений.

Кроме рассмотренных дополнительных структурных типов используются и другие – таблицы, файлы, всевозможные комбинации, перечисленных выше типов и др. Полный анализ всех типов данных и их преобразований выходит за рамки настоящей книги. Предлагаемый подход может применяться и для использования этих типов.

3.6. МЕТОДЫ ПРЕОБРАЗОВАНИЯ НЕРЕЛЕВАНТНЫХ ТИПОВ ДАННЫХ

Выше построено множество $\sum$, которое состоит из двух групп алгебраических систем: $\sum_1 = \{U^b, U^c\ U^i, U^r\}$,

$$\sum_2 = \{U^a, U^z, U^u, U^e\},$$

где $\sum_1$ включает наборы простых типов данных ($t = b$ (bool), c (char), i (int), r (real)) и $\sum_2$ – сложных (структурных) типов данных ($t = a$ (array), z (record), u (union), e (enum),…), как комбинаций простых типов данных.

Следующим процессом в разработке межьязыкового интерфейса, согласно постановке задачи, будет построение изоморфных отображений между элементами множества описывающими типы данных в различных ЯП. Поэтому в общем случаи множество $\sum$ будет состоять из нескольких подмножеств:

$$\sum = \bigcup_{\alpha=1}^{\eta} \Sigma_\alpha , \qquad (3.18)$$

где Σ_α описывает множество типов данных в языке l_a, принадлежащем классу L.

При построении изоморфных отображений между алгебраическими системами необходимо определить их свойства, основанные на сохранении интерпретации однотипных операций в различных системах. Для полноты анализа необходимо рассмотреть все возможные комбинации между элементами каждого из множеств

$\Sigma\alpha$ и между элементами различных $\Sigma\alpha$ и $\Sigma\beta$, где $\alpha \neq \beta$. При строгом анализе значительное число отображений между системами U_a^t и U_β^q не рассматривается, что связано с отличиями некоторых множеств Ω_a^t и Ω_β^q и отсутствием изоморфных отображений между определенными основными множествами X_a^t и X_β^q. В связи с этим, чтобы не ограничивать множество операций преобразований типов данных, будут рассмотрены следующие случаи:

1. Алгебраические системы U_a^t и U_β^q изоморфны. Существуют изоморфизмы между X_a^t и X_β^q. Множества Ω_a^t и Ω_β^q совпадают.

2. Существуют изоморфизмы между X_a^t и X_β^q. Множества Ω_a^t и Ω_β^q различные. Если $\Omega_a^t \cap \Omega_\beta^q = \Omega$ и Ω не пусто, то рассматриваются отображения между модифицированными алгебраическими системами $U_a^t = < X_a^t, \Omega >$ и $U_\beta^q = X_\beta^q < \Omega >$.

3. Между множествами X_a^t и X_β^q отсутствует изоморфное отображение. Рассмотрено несколько частных случаев для преобразования типов данных, принадлежащих этой группе.

Переходя к строгому анализу изоморфных отображений, отметим следующий очевидный факт: мощности алгебраических систем должны быть равны:

$$|U_a^t| = |U_\beta^q| \qquad (3.19)$$

Под мощностью алгебраической системы U_a^t понимается мощность множества X_a^t [6]. Несмотря на тривиальность результата, этот факт может служить очень сильным критерием оценки правильности сопряжения модулей для случая, если в разных ЯП одинаковые типы данных имеют различные множества значений. Другим применением этого критерия может служить перенос программного обеспечения с одного типа ЭВМ на другой с различными архитектурами и диапазонами целых и вещественных чисел. Для получения строгих результатов условие (3.19) всегда должно выполняться.

Анализируя множества Ω_a^t можно отметить, что они все содержат операции отношения. Поэтому на самих множествах X_a^t задано отношение порядка. Учитывая, что любые два элемента из X_a^t сравнимы, это отношение определяет линейный порядок. В этом случае изоморфное отображение должно сохранять отношение линейного порядка. В дальнейшем всегда будет предполагаться выполнение этого требования.

Пусть $\sum$ – множество алгебраических систем, построенных выше. Тогда имеет место следующая лемма.

Лемма 3.1. *Для любого изоморфного отображения φ между системами U_a^t и U_β^q выполняются равенства $\varphi(X_{a.\min}^t) = X_{\beta.\min}^q$, $\varphi(X_{a.\max}^t) = X_{\beta.\max}^q$.*

Доказательство данной леммы простое. Для всех построенных алгебраических систем, описывающих простые типы данных, множества значений ограниченны (числовые типы рассматриваются как отрезки). Согласно аксиомам (3.9) минимальные и максимальные элементы этих множеств им принадлежат. Структурные типы этих данных строятся из простых с помощью конечного числа операций, а их множества значений также конечны. Поэтому $X_{a.\min}^t$, $X_{a.\max}^t$, $X_{\beta.\min}^q$,

$X^q_{\beta.\max}$ существуют и принадлежат множествам X^t_a и X^t_β соответственно. Учитывая линейную упорядоченность этих множеств, необходимо, чтобы выполнялось условие леммы. Если бы оно не выполнялось, то изоморфное отображение не сохраняло бы линейный порядок, что противоречит сделанному ранее выводу. Лемма доказана.

3.6.1. ПРЕОБРАЗОВАНИЕ ПРОСТЫХ ТИПОВ ДАННЫХ

Исследуем следующие типы преобразований:

перечислимый – в перечислимый;
булевый – в булевый;
целый – в целый;
вещественный – в вещественный.

Рассмотрим операцию преобразования между *перечислимыми* и типами на примере символьных типов. Пусть $U^c_a = U^c_\beta$, описывают два символьных типа в некоторых ЯП. Ранее были получены результаты о том, что изоморфное отображение должно сохранять порядок и $|U^c_a| = |U^c_\beta|$. Имеет место следующая теорема.

Теорема 3.1. *Пусть φ – отображение системы U^c_a в систему U^c_β множества $\sum_1$. Для того чтобы отображение φ было изоморфным, необходимо и достаточно, чтобы φ изоморфно отображало X^c_α и X^c_β с сохранением линейного порядка.*

Необходимость. Пусть φ – изоморфизм. Тогда при отображении сохраняются все операции множества $\Omega = \Omega^c_\alpha = \Omega^c_\beta$, в том числе и операция отношения, которая определяет линейный порядок на множествах X^c_α и X^c_β.

Достаточность. Пусть φ изоморфно отображает X^c_α на X^c_β с сохранением линейного порядка. Необходимо проверить сохранность операций, указанных в (3.8). Операция отношения выполняется согласно упорядоченности. Рассмотрим операцию succ.

Согласно лемме 3.1 $\varphi(X^c_{\alpha.\min}) = X^c_{\beta.\min}$. Последовательно применяя операцию succ к данному равенству и учитывая линейную упорядоченность множества X^c_α на X^c_β *(x < succ(x))*, мы получим, что для любого $x^c_\alpha \in X^c_\alpha$ и $x^c_\alpha \neq X^c_{\alpha.\max}$ из равенства $\varphi(x^c_\alpha) = x^c_\beta$ следует равенство

$$\varphi(\text{succ}(x^c_\alpha)) = \text{succ}(x^c_\beta). \qquad (3.20)$$

Для операции pred рассмотрение аналогичное. Согласно лемме 3.1 $\varphi(X^c_{\alpha.\max}) = X^c_{\beta.\max}$. Последовательно применяя операцию pred, получим, что для любого $x^c_\alpha \in X^c_\alpha$ и $x^c_\alpha \neq X^c_{\alpha.\min}$ из равенства $\varphi(x^c_\alpha) = x^c_\beta$, где $x^c_\beta \in X^c_\beta$ следует равенство

$$\varphi(\text{pred}(x^c_\alpha)) = \text{pred}(x^c_\beta). \qquad (3.21)$$

Теорема доказана. Она позволяет использовать в качестве операции преобразования между перечисленными типами любое изоморфное отображение, удовлетворяющее условиям теоремы, независимо от природы элементов множеств X^c_α и X^c_β. Если в вызывающем модуле фактический параметр имеет тип

type *TV* = ('A', 'B', 'C'),

а в вызываемом модуле формальный параметр описан в виде

type *TF* = (1, *2,* 3),

то в качестве операции преобразования можно выбрать любое изоморфное отображение вида

$$\varphi\ (\text{'A'}) = 1,$$
$$\varphi\ (\text{'B'}) = 2,$$
$$\varphi\ (\text{'C'}) = 3.$$

При этом необходимо помнить, что над объектами типа *TF* можно использовать только операции перечислимого типа, определенные в (3.8).

Рассмотрим операцию преобразования между б у л е в ы м и типами. Пусть U_a^b и U_β^b – алгебраические системы, описывающие эти типы. Тогда справедлива следующая теорема.

Теорема 3.2. *Любой изоморфизм* φ *между системами* U_a^b и U_β^b *является тождественным изоморфизмом:*

$$\varphi\ (X^\beta_{\alpha.false}) = X^b_{\beta.false}$$
$$\varphi\ (X^b_{\alpha.true}) = X^b_{\beta.true} \qquad (3.22)$$

где через $x^\beta_{\alpha.false}(X^b_{\beta.false})$ и $x^\beta_{\alpha.true}(X^b_{\beta.true})$ обозначены соответственно элементы false и true *в множестве* $X^b_\alpha(X^b_\beta)$.

Д о к а з а т е л ь с т в о. Согласно теореме 3.1 для отображения между U^b_α и U^b_β выполняются все операции из (3.7), кроме операций булевой алгебры. При выполнении последних операций всегда справедливо неравенство $X^\beta_{\alpha.false} < X^b_{\alpha.true}$. Поэтому, учитывая, что φ сохраняет порядок, единственно возможным изоморфизмом является отображение вида (3.22). Теорема доказана.

В общем случае X^β_α и X^b_β – различные множества (например, в ЯП ПЛ/1 в качестве булевых переменных используются битовые строки). Если $X^\beta_\alpha = X^b$, то φ является тождественным автоморфизмом.

Рассмотрим операции преобразования для *числовых* типов. Введем стандартные обозначения для множеств целых чисел Z, рациональных Q и действительных (вещественных) R. Ранее предполагалось, что основные множества алгебраических систем, соответствующие числовым типам, ограничены. Для дальнейшего анализа снимем это ограничение и будем считать, что при необходимости границы числового диапазона могут быть расширены.

Рассмотрим алгебраические системы U^i_α и U^i_b, соответствующие целым типам, и изоморфизм φ между ними. Изучим свойства этого изоморфизма. Для операции сложения выполняется равенство $\varphi(x^i_\alpha + y^i_\alpha) = \varphi(x^i_\alpha) + \varphi(y^i_\alpha)$. Полагая $y^i_\alpha = 0$, получаем, что $\varphi(x^i_\alpha) = \varphi(x^i_a) + 0$. Так как x^i_α – произвольное, то данное равенство будет выполняться только при $\varphi(0) = 0$.

Для операции умножения необходимо выполнение равенства $\varphi(x^i_\alpha \cdot y^i_\alpha) = \varphi(x^i_\alpha) \cdot \varphi(y^i_\alpha)$. Полагая $y^i_\alpha = 1$, получаем, что $\varphi(x^i_\alpha) = \varphi(x^i_\alpha) \cdot \varphi(1)$. Здесь $\varphi(x^i_\alpha)$ – целое число. Если оно не равно нулю, то выполним операцию целочисленного деления и получим, что $\varphi(1) = 1$. Окончательно можно записать

$$\varphi\ (0) = 0, \quad \varphi\ (1) = 1, \qquad (3.23)$$

где φ – изоморфизм между U^i_α и U^i_β.

Пусть $X^i_\alpha \subset Z$ и $X^i_\beta \subset Z$. Тогда $\forall\ x^i_\alpha \in X^i_\alpha\ \exists\ x^i_\beta \in X^i_\beta : \varphi(x^i_\alpha) = x^i_\beta$.

Рассмотрим произвольное $x^i_\alpha < 1$. Последовательно применяя результаты (3.23), получаем

$$\varphi(x^i_\alpha) = \varphi(x^i_\alpha - 1) + \varphi(1) = \varphi(x^i_\alpha - 1) + 1 = \ldots = \varphi(1) + x^i_\alpha - 1 = x^i_\alpha .$$

Пусть $x^i_\alpha < 0$ и $|x^i_\alpha| = y^i_\alpha > 0$. Отметим следующий результат:

$$0 = \varphi(y^i_\alpha + (-y^i_\alpha)) = \varphi(y^i_\alpha) + \varphi(-y^i_\alpha) = y^i_\alpha + \varphi(-y^i_\alpha).$$

Отсюда следует, что $\varphi(-y^i_\alpha) = -y^i_\alpha$ или $\varphi(x^i_\alpha) = x^i_\alpha$. Окончательно можно отметить, что для любого целого x^i_α необходимо равенство

$$\varphi(x^i_\alpha) = x^i_\alpha \qquad (3.24)$$

Рассмотрим изоморфное отображение φ между алгебраическими системами U^r_α и U^r_β, соответствующие *вещественным* типам, где X^r_α и X^r_β – подмножества *R*. Так как $Z \subset R$, то результат (3.24) справедлив и для подмножества вещественных чисел, совпадающего с *Z*. Пусть $x^r_\alpha \in Q$, т. е. $x^r_\alpha = m/n$, где *m* и *n* –целые числа. Тогда отметим следующее:

$$m = \varphi(m) = \varphi(n\frac{m}{n}) = \varphi(n) * \varphi(\frac{m}{n}) = n * \varphi(\frac{m}{n}).$$

Отсюда следует равенство

$$\varphi(\frac{m}{n}) = \frac{m}{n}. \qquad (3.25)$$

Это соотношение выполняется для любых рациональных чисел. На основании известной теоремы из теории множеств (см., например, [6, 28]) Любое вещественное число *x* можно с любой заранее определенной точностью ε представить в виде рационального числа *m/n* так, что $|x - \frac{m}{n}| < \varepsilon$. Это тем более справедливо для представления вещественных чисел в ЭВМ, так как точность представления зависит от количества разрядов машинного слова памяти. Таким образом, для любых $x \notin R$ выполняется

$$\varphi(x) = x. \qquad (3.26)$$

Учитывая, что множества *Z* и *R* являются базовыми для основных множеств X^i_α и X^r_α алгебраических систем U^i_α *и* U^r_α, определенных в п. 2.4, мы доказали следующую теорему.

Теорема 3.3. *Любой изоморфизм между алгебраическими системами соответствующими числовым типам, является тождественным автоморфизмом.*

В доказательстве использовался тот факт, что 0 и 1 принадлежат основным множествам. Это существенное предположение, так как некоторые числовые отрезки не содержат этих значений. Такие типы данных должны анализироваться в частном порядке.

Выше получены результаты, следующие из анализа изоморфных отображений алгебраических систем. Перейдем к рассмотрению случая изоморфизма основных множеств X^t_α и X^q_β при условии отличия Ω^t_α и Ω^q_β Пусть $\Omega^t_\alpha \cap \Omega^q_\beta \neq 0$. Возможны

следующие виды преобразований между типами данных:

символьный – в целый;

символьный – в булевый;

целый – в символьный;

целый – в булевый;

булевый – в целый;

булевый – в символьный.

В приведенных преобразованиях отсутствует вещественный тип. Это связано с тем, что множества значений символьных, булевых и целых чисел имеют дискретный характер, множество вещественных чисел по своей сути непрерывно (при реализации на ЭВМ это множество также дискретно, что связано с особенностями структуры основной памяти).

Рассмотрим преобразования символьного типа в целый и целого в символьной. Пусть U^c_α и U^i_β – алгебраические системы, описывающие эти типы. Пересечение множеств операций $\Omega = \Omega^c_\alpha \cap \Omega^i_\beta$ имеет вид

$$\Omega = \{\text{pred, succ}, \leq\}. \qquad (3.27)$$

Но оно в точности совпадает со множеством операций для любого перечислимого типа. Поэтому мы можем воспользоваться результатами теоремы 3.1 и выбрать любое изоморфное отображение множеств X^c_α и X^i_β, сохраняющее линейный порядок. В качестве множества X^i_β может быть выбран любой отрезок целого типа, для которого

$$|X^c_\alpha| = X^i_{\beta.\max} - X^i_{\beta.\min} + 1, \qquad (3.28)$$

т. е. должно выполняться условие (3.19) о равенстве мощностей основных множеств для алгебраических систем. Особый интерес представляют отображение, ставящее каждому символу его порядковый номер в алфавите (функция ord), и отображение, определяющее по порядковому номеру символа в алфавите его значение (функция chr). В этом случае функции ord и chr являются операциями преобразования между символьным и целым типами.

Рассмотрим преобразование целого в булевый и булевого в целый. Пусть алгебраические системы U^i_α и U^b_β описывают соответственно целый и булевый типы. Пересечение множеств операций $\Omega = \Omega^i_\alpha \cap \Omega^b_\beta$ совпадает с (3.27). Поэтому в данном случае применимы предыдущие результаты и над целым и булевым типами возможны только операции как над перечислимыми типами.

Преобразование символьного в булевый и булевого в символьный практического применения не имеет. Однако для них справедливы формальные выводы, как и в случае преобразования между символьными и целыми типами. Возможно только применение операций для перечислимых типов. Множества значений каждого из типов для данного случая будут содержать по 2 элемента.

3.6.2. ПРЕОБРАЗОВАНИЕ СТРУКТУРНЫХ ТИПОВ ДАННЫХ

Рассмотрим операции преобразования для массивов и записей. Анализируя множества операций для соответствующих алгебраических систем (3.15) и (3.17), можно отметить, что при преобразовании должен сохраняться только линейный порядок.

Возможны следующие виды преобразований: массив – в массив, запись – в

запись, массив – в запись, запись – в массив.

Рассмотрим первое из них. Пусть U^a_α и U^i_β – две алгебраические системы, описывающие массивы. Пусть φ – их изоморфизм, в котором сохраняется линейный порядок. Это означает, что

$$(\forall\ x^a_{\alpha_1} \in X^a_\alpha\ ) \,\&\, (\forall\ x^a_{\alpha_2} \in X^a_\alpha) \,\&\, (\ x^a_{\alpha_1} \le x^a_{\alpha_2}\ ) \ \Rightarrow\ \varphi\ (x^a_{\alpha_1}) \le \varphi(x^a_{\alpha_2}\ ) \qquad (3.29)$$

Согласно (3.16), $x^a_{\alpha_1}$ и $x^a_{\alpha_2}$ – функции отображения. Выполнение отношения $\le$ для функций означает выполнение этого отношения для всех элементов области определения этих функций.

Рассмотрим определение выражений для функций $\varphi\ (x^a_{\alpha_1}\ )$ и $\varphi(x^a_{\alpha_2}\ )$, из (3.16) следует

$$x^a_{\alpha_1} : I^a_\alpha \to Y(x^a_{\alpha_1}),\ \ Y(x^a_{\alpha_1}\ ) \subset \overline{Y}\ (\ X^a_\alpha),$$
$$x^a_{\alpha_2} : I^a_\alpha \to Y(x^a_{\alpha_2}),\ \ Y(x^a_{\alpha_2}\ ) \subset \overline{Y}\ (X^a_\alpha). \qquad (3.30)$$

Пусть выполняется

$$\varphi_i :\ I^a_\alpha \to I^a_\beta\ ,\qquad \varphi_y : \overline{Y}\ (X^a_\alpha) \to \overline{Y}\ (X^a_\beta) \qquad (3.31)$$

где φ_i и φ_y – изоморфные отображения, соответствующие множествам индексов i и значениям y элементов массива. Через E_1 обозначим вложение $Y\ (x^a_{\alpha_1}\ ) \to \overline{Y}\ (X^a_\alpha)$, а через E_2 – вложение $Y(\ x^a_{\alpha_2}\ ) \to \overline{Y}\ (X^a_\alpha)$. Тогда $\varphi(\ x^a_{\alpha_1}\ )$ и $\varphi(\ x^a_{\alpha_2}\ )$ будут определяться как ограничения отображения φ_y на соответствующие подмножества и обозначаться через $\varphi_y|Y(\ x^a_{\alpha_1}\ )$ и $\varphi_y|Y(\ x^a_{\alpha_2}\ )$ соответственно. Последнее неравенство в (3.29) будет эквивалентно

$$\varphi_y\ |\ Y(\ x^a_{\alpha_1}\ ) \le \varphi_y\ |\ Y(\ x^a_{\alpha_2}\ ) \qquad (3.32)$$

Расширим смысл обозначения в (3.31). Пусть теперь φ_i и φ_y будут обозначать изоморфные отображения между алгебраическими системами, для которых I^a_α, I^a_β, $\overline{Y}\ (X^a_\alpha), \overline{Y}\ (X^a_\beta)$ являются основными множествами. Тогда имеет место следующая теорема.

Теорема 3.4. *Пусть* U^a_α и U^a_β *– две системы из* $\sum_2$, *соответствующие типам данных массив, и* φ_i, φ_y *– изоморфные отображения, сохраняющие линейный порядок и определяемые* (3.31).

Тогда изоморфизм φ между U^a_α и U^a_α и U^a_β полностью определяется отображениями φ_i, и φ_y. Это означает, что анализ отображения φ может быть сведен к анализу φ_i и φ_y и значительно упрощается задача построения изоморфного отображения.

Доказательство. Пусть имеются отображения φ_i и φ_y. Отображение φ_i сохраняет линейный порядок, и, следовательно, сохраняется взаимная упорядоченность отдельных элементов массивов относительно друг друга. Рассмотрим выражение (3.29). С учетом предыдущего результата выполнение (3.29) достаточно проверить только для одного элемента массива. Пусть $k \in I^a_\alpha$. Ему будет соответствовать $\varphi_i\ (k) \in I^a_\beta$. Пусть выполняется неравенство $x^a_{\alpha_1}\ (k) \le x^a_{\alpha_2}\ (k)$.

Применим к обеим частям неравенства отображение φ_y в смысле (3.32). Упорядоченность не нарушается при применении операций конструирования структурных типов. Поэтому независимо от типа данных для множеств $\overline{Y}(X^a_\alpha), \overline{Y}(X^a_\beta)$ отображение φ_y также сохраняет линейный порядок. Следовательно, справедливо неравенство

$$\varphi_y \mid Y(x^a_{\alpha_1})(k)) \le \varphi_y \mid Y(x^a_{\alpha_2})(k)),$$

при этом каждому значению $\varphi_i(k)$ соответствует $\varphi_y \mid Y(x^a_\alpha)(k))$. Теорема доказана.

Таким образом, можно записать, что φ является двухкомпонентной последовательностью $\varphi = (\varphi_i, \varphi_y)$. Преобразование массивов определяется преобразованиями типов данных для индексов и множеств значений элементов массивов, принадлежащих к рассматриваемому типу.

Рассмотрим преобразование типа запись – в запись. Пусть U^z_α и U^z_β – две алгебраические системы, описывающие записи. В теории структурной организации данных [151] множество значений для записей определяется как прямое произведение множеств значений их отдельных компонентов. Данный подход к описанию записей содержит существенный недостаток. Рассмотрим множество значений алгебраической системы (3.17). Пусть $n = 3$ и имеется три описания:

а) type $T^{z_1} = (S_{v_1} : T^{v_1};\ S_{v_2} : T^{v_2};\ S_{v_3} : T^{v_3})$,

б) type $T^{z_2} = (S_{a_1} :\ T^{v_1};\ S' : T')$,
type $T' = (S_{v_2} : T^{v_2};\ S_{v_3} : T^{v_3})$,

в) type $T^{z_3} = (S' : T';\ S_{v_3} : T^{v_3})$,
type $T' = (T_{v_1} : T^{v_1};\ S_{v_2} : T^{v_2})$.

Несмотря на различия в описании трех типов записей T^{z_1}, T^{z_2}, T^{v_3} множества их значений совпадают относительно задачи преобразования данных. Поэтому имеется неопределенность при реализации операции преобразования вида запись – в запись. Для устранения этой неопределенности введем дополнительное ограничение, которое состоит в том, что между множествами типов данных обеих записей можно установить взаимно однозначное соответствие. Из этого ограничения следует:

– количество компонентов в обеих записях одинаково;

– для каждого из компонентов первой записи существует только один соответствующий компонент второй записи.

Операции отношения над записями будут выполняться с учетом выбранного соответствия (последовательности сравниваемых компонентов определяются данным соответствием). В этом случае анализ преобразования записи сводится к анализу преобразований типов отдельных компонентов. Если изоморфные отображения между отдельными компонентами сохраняют линейный порядок для самих компонентов, то с учетом сказанного выше мы доказали следующую теорему.

Теорема 3.5. *Пусть U^z_α и U^z_β – две алгебраические системы множества $\sum_2$,*

соответствующие типам данных запись, и $x_\alpha^z \in X_\beta^z$, $x_\beta^z \in X_\beta^z$.

Тогда, если между последовательностями компонентов x_α^z и x_β^z существует взаимно однозначное соответствие, то изоморфизм между U_α^z и U_β^z определяется изоморфными отображениями алгебраических систем, соответствующих компонентов записей.

Рассмотрим смешанные виды преобразований между структурными типами. Пусть U_α^a и U_β^z – алгебраические системы, описывающие массив и запись соответственно. Множества операций для этих систем совпадают. Поэтому определим условия изоморфного отображения для множеств значений этих типов. Как и в случае преобразования записей, для данного вида преобразования имеется некоторая неопределенность, которую можно устранить, введя следующее ограничение: количество элементов массива должно совпадать с количеством элементов записи. Тогда массив рассматривается как запись, у которой множество селекторов совпадает со множеством индексов, а типы всех компонентов одинаковы. Поэтому для преобразования массива – в запись можно использовать результаты теоремы 3.5.

Преобразование вида запись – в массив требует, чтобы типы всех компонентов записи были одинаковыми, а сама запись была представлена в виде массива. Множество значений компонентов образует множество значений элементов массива *Y*. Множество индексов строится простым переупорядочиванием селекторов компонентов записи: *i*-й компонент записи ставится в соответствие *φ(i)*-й элемент во множестве *J* (*φ* – взаимно-однозначное отображение). Само множество *I* будет множеством значений перечислимого типа. В этом случае можно использовать результаты теоремы 3.4.

3.6.3. ИЗМЕНЕНИЕ УРОВНЯ СТРУКТУРИРОВАНИЯ ДАННЫХ

К этим операциям относятся операции селектора и конструирования. Операции селектора обеспечивают выбор из объекта структурного типа отдельных компонентов: для массива – отдельные элементы, для записи– компоненты записи. В. п. 3.5 при анализе структурных типов данных дано определение этих операций. Операция выбора элемента массива определяется как ограничение $x|\{k\}$, где x – объект типа массив, представленный отображением (3.16), а k – элемент из множества индексов. Введем обозначение для этой операции

$$S^a = S^a(x, K). \qquad (3.33)$$

Аналогично была определена операция селектора для записей как ограничение $x|\{S_{w_m}\}$, где x определяет соответствие между множеством селекторов записи $I = \{ S_{w_1}, ..., S_{w_n} \}$ и компонентов записи $X^v = \{ x^{v_1}, ..., x^{v_n} \}$, а S_{w_m} – селектор выбираемого компонента. Введем обозначение для этой операции

$$S^c = S^c(x, S_{w_m}). \qquad (3.34)$$

Такая интерпретация операций селектора позволяет не учитывать типы отдельных элементов и компонентов структурных типов, а рассматривать их как множество объектов произвольной природы. Окончательно множество операций

селектора будет иметь вид:

$$S = \bigcup_{\alpha=1}^{n} S_\alpha, \quad S_\alpha = \{ S_\alpha^a, S_\alpha^z \}, \tag{3.35}$$

где S_a – множество операций селектора для структурных типов в l_a; S^a_a – операция селектора для массивов в l_a; S^z_α – операция селектора записей в l_a.

Определим операции конструирования структурных типов данных. Как и для операций селектора, будем полагать, что множество объектов, из которых конструируется структурный тип, имеет произвольную природу. Для конструирования типа *массив* необходимо, чтобы все его элементы имели одинаковый тип. В этом случае для операции конструирования необходимо:

упорядочить множество элементов;

построить множество индексов для элементов массива;

установить однозначное отображение между множеством индексов и множеством элементов.

Пусть имеется конечное множество объектов $X' = \{x_1, ..., x_n\}$ некоторого типа T^t. Запишем операцию конструирования массива:

$$C^a = C^a(x_1 \ldots . \ x_n) \tag{3.36}$$

Множество элементов упорядочено естественным образом в порядке их следования в описании (при необходимости естественную упорядоченность можно изменить). Множество индексов будет иметь вид $I = (1, ..., n)$, что соответствует множеству значений перечислимого или отрезка целого типа. Отображение между множествами I и X' определяется тем, что каждому $k \in I$ соответствует $x_k \in X'$. Обозначим это отображение через x^a. Окончательно получим

$$x^a: I \rightarrow Y(x^a),$$

где $Y(x^a)$ определяет множество значений элементов $x_1, ..., x_n$. При этом $Y(x^a) \subset X^t$ (X^t – множество значений для типа T^t). Этот массив может быть описан следующим образом:

$$\text{type } T^a = \text{array } T(I) \text{ of } T^t,$$

где T^a – тип данных массива; $T(I)$ – тип данных для множества индексов (перечислимый или отрезок целого типа).

Определим операцию конструирования записи. Метод построения аналогичен методу построения массивов. Имеется конечное множество объектов $X^v = \{ x^{v_1}, ..., x^{v_n} \}$, типов T^{v_1} ..., T^{v_n} соответственно. Операцию конструирования записи представим в виде

$$C^z = C^z\{ x^{v_1}, ..., x^{v_n} \} \tag{3.37}$$

Множество элементов упорядочено естественным образом в порядке их следования в описании операции (при необходимости естественную упорядоченность можно изменить). Каждому компоненту x_{v_m} поставим в соответствие селектор S_{v_m}, который может быть номером компонента в списке (3.37). Этим полностью определяется операция конструирования записи

$$\text{type } T^z = (S_{v_1} : T^{v_1} ; \ldots \ S_{v_n} : T^{v_n}),$$

где T^z – тип записи .

Окончательно запишем множество операций конструирования:

$$C = \bigcup_{\alpha=1}^{n} C_\alpha \ , \ C_\alpha = \{ C_\alpha^a \ , \ C_\alpha^z \}, \qquad (3.38)$$

где C_α –множество операций конструирования для структурных типов в ЯП; C_α^a – оператор конструирования массива в l_a , C_α^z – оператор конструирования записи в l_a.

3.7. ВОПРОСЫ ОПРЕДЕЛЕНИЯ ИНТЕРФЕЙСА ЯП И ПРОГРАММ

Проведенный выше анализ определяет необходимые условия, методы, правила и т. д., служащие основой построения межъязыкового интерфейса. Однако на практике часто встречаются задачи сопряжения модулей, которые не могут быть рассмотрены в рамках приведенных моделей, методов и средств. Для их решения необходима разработка методов, отличных от рассмотренных. Они обычно содержат допущения, противоречающие условиям формального анализа, – нарушение свойств модулей, изоморфности алгебраических систем и т. д.

Случаи нарушения условий построения МЯИ

1. Типы данных, описанные в п. 3.5, являются наиболее общими. При разработке модулей часто приходится вводить такие типы, множества значений которых являются подмножествами множеств значений для этих типов. Рассмотренный подход может применяться и для их анализа. Однако результаты в этом случае будут другими. Проиллюстрируем это на примере отрезков числовых типов.

Предыдущий анализ над множествами значений для числовых типов обязательно включает 0 и 1. В некоторых случаях отрезки типов могут не содержать этих значений. Поэтому результаты предыдущего анализа для числовых типов использоваться не могут. Однако выполняемые операции для объектов данных типов могут быть корректными за счет неявных дополнительных условий. Например, для отрезка целого типа вида *m...n,* где $n > m > 1$, могут не применяться операции вычитания div и mod, а операции сложения и умножения будут давать корректные результаты. Данную особенность можно легко объяснить. Алгебраическая система для целого типа (3.10) характеризуется множеством Ω^i, включающим все возможные операции для целого типа. Исключение из этого множества некоторых операций дает нам совершенно иную алгебраическую систему, и преобразование таких типов приводит к другим результатам. Однако в модулях соответствующий тип будет описан как отрезок целого, поэтому для аналогичных случаев анализ преобразования типов должен проводиться в частном порядке. Этот пример иллюстрирует проблему несоответствия в описаниях типов, относящуюся к классу проблем сопряжения, связанных с отличиями в описаниях модулей.

2. Другим аспектом практической реализации может служить несоответствие областей значений одинаковых типов фактического и формального параметров. Пусть фактический параметр имеет вещественный тип с областью значений вида – *a..a,* а формальный – вещественный тип с областью значений – *b..a,* где $a > b > 0$. Несмотря на различие в множествах значений, операции над объектами будут корректными, если значения объектов и результатов операций будут принадлежать пересечению рассмотренных отрезков. В этом случае при построении алгебраических систем необходимо в качестве множества значений рассматривать пересечение отрезков, чтобы обеспечить построение изоморфного

отображения.

3. При анализе преобразований между простыми типами было отмечено, что с вещественными типами возможны только преобразования вида *вещественный – в вещественный.* Это очень сильное ограничение, так как часто в практике программирования возникает преобразование типов вида *целый – в вещественный* и наоборот. Строгий анализ таких преобразований не может быть проведен ввиду отсутствия изоморфного соответствия между множествами значений этих типов – одному целому числу будет соответствовать некоторое множество вещественных чисел. Анализ преобразований для этих типов должен производиться следующим образом. Отображения между множествами целых и вещественных чисел являются частичными мульти отображениями. Пусть φ – отображение множества X^r на X^i, где X^r и X^i – множества вещественных и целых чисел соответственно. Для каждого $x^i \in X^i$ определим прообраз $\varphi^{-1}(x^i) \subset X^r$. Различным x^i будут соответствовать различные прообразы. Введем фактор-множество X^r/φ, элементами которого являются прообразы элементов x^i. В алгебраической системе U^r, описывающей вещественный тип, заменим множество X^r фактор-множеством X^r/φ. В этом случае отображение между алгебраическими системами сохраняет линейный порядок, но корректность арифметических операций не выполняется. Так, если преобразование вещественного числа в целое состоит в отбрасывании дробной части, то 1,6 + 1,6 = 3,2 и φ(3,2)=3. В то же время φ(1,6)+ φ(1,6) = 1 + 1 = 2 ≠ 3. Эти особенности необходимо учитывать при практической реализации подобных преобразований.

Данный пример относится к анализу преобразований, при которых отсутствует изоморфное соответствие между основными множествами алгебраических систем, описывающих преобразуемые типы. Анализ подобных преобразований должен проводиться в частном порядке.

4. Среди преобразований, рассмотренных в п. 3.7, не рассматривался случай, когда строка символов, содержащая последовательность цифр, должна преобразовываться в объект целого типа с соответствующим значением.

Как известно, в теории структурной организации данных строка символов описывается в виде массива символьных элементов. Поэтому такое преобразование переводит весь массив (структурный тип) в целое число (простой тип). Аналогичные преобразования, связанные с отображениями структурного типа в простой и наоборот, не входят в задачи межъязыкового интерфейса и поэтому не рассматриваются. Четыре приведенных выше примера показывают многообразие проблем и задач сопряжения модулей. Часть из них удается свести к строгому анализу с помощью дополнительных ограничений. Другие задачи не входят в состав функций межъязыкового интерфейса. Однако рассматриваемый подход с соответствующими дополнениями может использоваться для построения любых межъязыковых интерфейсов, предназначенных для комплексирования модулей.

Формальный подход к построению межъязыковых интерфейсов, рассмотренный в данной главе, позволяет упорядочить разработку МЯИ, выделить основные процессы его создания и особые ситуации, возникающие при его построении.

Аспекты практической реализации МЯИ

1. Выделены два основных процесса в построении МЯИ: разработка множества интерфейсных функций (операций преобразования,

селектора, конструирования);

реализация сопряжения для каждой пары модулей.

Данные процессы в значительной мере независимы. Поэтому практическая реализация соответствующих программных компонентов может осуществляться параллельно. Кроме того, фиксированное множество интерфейсных функций может использоваться в различных алгоритмах сопряжения, которые могут соответствовать разным прикладным программным системам. В этом случае множество функций является связующим звеном для данных систем.

2. Формальный подход к интерфейсным функциям позволяет на процессе проектирования выделить множество допустимых операций, исключая операции, которые не могут быть реализованы или не относятся к функциям МЯИ.

3. Результаты формального подхода позволяют осуществлять контроль на совместимость типов передаваемых параметров. Если в вызывающем и вызываемом модулях описаны типы данных фактических и формальных параметров с указанием множеств значений и операций над типами, то на процессе сопряжения можно определять допустимость преобразования типов для соответствующих параметров.

4. Анализ, основанный на формальном подходе, позволяет определить наиболее типичные ошибки сопряжения модулей (при отсутствии МЯИ). К их числу следует отнести ошибки:

несоответствия числа параметров в списках фактических и формальных параметров;

несогласованности типов передаваемых параметров в вызывающем и вызываемом модулях;

несоответствия во множествах значений типов фактических и формальных параметров;

адаптации программного обеспечения на ЭВМ с другой архитектурой (меняются множества значений);

связанные с использованием операций, которые не включены в описания типа данных (чаще всего это характерно для применения разного рода функций);

связанные с различным уровнем структурирования фактических и формальных параметров;

основанные на неверном описании типов данных передаваемых параметров в вызывающем и вызываемом модулях;

вызванные отсутствием обратных преобразований типов данных после работы вызываемого модуля.

Все эти классы ошибок выделены на основании анализа, проведенного в данной главе.

5. Реализация множества интерфейсных функций и алгоритма комплексирования позволяет автоматизировать процесс сопряжения модулей. Такой процесс автоматизированного сопряжения реализован в системе АПРОП, рассмотренной ниже.

В заключение рассмотрим сводку правил сопряжения модулей, обобщающих полученные ранее выводы и результаты.

Правило 1. Списки фактических и формальных параметров должны быть упорядочены – сначала следуют входные, затем выходные параметры. Если какой-либо параметр является и входным и выходным, то он должен присутствовать дважды в соответствующих частях списка.

Правило 2. Провести разбиения списков параметров на подмножества для построения однозначного отображения между списками фактических и формальных параметров. Упорядочить списки после проведения разбиения так, чтобы каждому *t*-му подмножеству V^t из списка фактических параметров соответствовало *t-е* подмножество F^t списка формальных параметров.

Для каждого *t* из списка входных параметров выполнить:

Правило 3. Если $|F^t| > 1$ и $|V^t| = 1$, то следует выбрать из множества операций селектора *S* необходимую операцию для соответствующей пары ЯП вызывающего и вызываемого модулей. Если операция существует, то следует применить ее к параметру V^t. Если такой операции нет, то необходимо ее построить, включить в состав множества *S* и применить к параметру V^t. Установить соответствие между каждого компонентного структурного типа (результат применения операции селектора) и соответствующим параметрам из F^t. Изменить отображение между списками параметров, соответствующее входным параметрам, за счет исключения соответствия между V^t и F^t и включения полученных новых соответствий.

Правило 4. Если $|V^t| > 1$ и $|F^t| = 1$, то выбрать из множества операций конструирования *C* необходимую операцию для соответствующей пары ЯП вызывающего и вызываемого модулей. Если операция существует, то применить ее ко множеству параметров V^t. Если такой операции нет, то необходимо ее построить, включить в состав множества *C* и применить ко множеству параметров V^t. Изменить отображение между списками параметров, соответствующих входным и выходным параметрам, за счет исключения соответствия между множеством V^t и параметром F^t и включения полученного нового соответствия для структурного типа.

Правило 5. Если $|V^t|>1$ и $|F^t|>1$, то данное преобразование не входит в состав задач межъязыкового интерфейса. Эта задача решается в частном порядке другими средствами.

После применения правил 3–5 получены модифицированные списки формальных и фактических параметров, они соответствуют входным параметрам, содержащим одинаковое количество элементов, и устанавливают между ними взаимно однозначное соответствие.

Для каждой пары фактических и формальных параметров выполнить:

Правило 6. Из множества *P* выбрать необходимую операцию преобразования типов данных. Если она существует, то применить ее к данной паре параметров. Если она отсутствует, то построить ее, включить во множество *P* и применить к рассматриваемой паре параметров. Если построить операцию не удается, то решить данную задачу в частном порядке с помощью других методов.

После обработки входных параметров вызываемым модулем правила 3–6 необходимо применить к выходным параметрам. При этом фактические параметры, а также множества V^t и F^t соответственно меняются местами.

Перечисленные шесть правил определяют содержание последнего процесса построения межъязыкового интерфейса относительно первых двух классов проблем, приведенных в п.2.1. Необходимо также отметить, что множества операций *P, S* и *C* постоянно расширяются за счет включения новых видов преобразований.

3.7.1. ПОДХОДЫ К РЕАЛИЗАЦИИ МЕЖЪЯЗЫКОВОГО И МЕЖМОДУЛЬНОГО ИНТЕРФЕЙСОВ

Примером реализации межъязыкового интерфейса, основанным на рассмотренных методах, могут служить соответствующие средства системы АПРОП [44]. Эти средства предназначались для решения следующих задач обеспечения:

– перехода от среды функционирования одного ЯП к среде функционирования другого;

– передачи управления между разноязыковыми модулями;

– доступа к общим данным для разноязыковых модулей;

– реализации механизма передачи данных через формальные параметры.

Предварительный анализ, проведенный в данной главе, позволяет выделить следующие процессы в разработке МЯИ: выбор класса ЯП; определение типов данных в этих ЯП; построение множества операций преобразования типов данных; построение множества операций конструирования и селектора для структурных типов; разработка правил сопряжения пар модулей с использованием построенных множеств операций. Рассмотрим подробнее каждый процесс.

Выбор класса ЯП. В класс ЯП межъязыкового интерфейса (МЯИ), реализованного в рамках системы АПРОП, входили ПЛ/1, Фортран, Кобол и Ассемблер ОС ЕС ЭВМ. Выбор этих языков обусловлен их широким распространением, различными сферами применения, и их особенностью является то, что все они не имеют механизмов конструирования новых типов данных и основные проблемы сопряжения возникают как различия в реализациях систем программирования с ЯП. Результаты исследования показывают, что проблемы сопряжения делятся на проблемы передачи управления и проблемы передачи данных между разноязыковыми модулями.

Предложенные операции преобразования типов данных, конструирования, селектора, передачи управления и передачи данных реализованы в системе АПРОП в виде макроопределений и загрузочных модулей.

Типы данных ЯП ПЛ/1, Фортран, Кобол. Существующее множество типов данных в этих ЯП небольшое. Простые типы представлены как предопределенные с небольшими видоизменениями. Структурные типы – массивы, записи и строки.

При рассмотрении типов данных ЯП, средств их описания и особенностей их представления алгебраическими системами значительное внимание уделяется архитектуре ЕС ЭВМ (структуре памяти, системе команд и др.). В ней под число целого типа отводится слово (4 байта) или полуслово (2 байта) памяти. Числа вещественного типа могут занимать слово или двойное слово (8 байт). Существуют вещественные числа, занимающие 16 байт памяти. Но в практике они встречаются редко и могут быть обработаны только средствами языков низкого уровня. Символьные данные представляются в виде последовательности байт, каждый из которых содержит один символ. Данные булевого типа представляются и обрабатываются как битовые строки. Числа, кроме целого и вещественного типов, могут быть представлены как последовательности десятичных цифр.

Язык ПЛ/1. К простым типам относятся целые (FIXED), вещественные (REAL), десятичные (DECIMAL) числа. Комплексные числа (COMPLEX) представлены в виде прямого произведения и занимают в памяти последовательность из двух чисел.

Символьные и логические данные описываются как строки (байт и бит соответственно). Этот тип данных в интерпретации для ЯП ПЛ/1 не входит в теорию

структурной организации данных и занимает промежуточное положение между простым и структурным типами. С точки зрения реализации строки необходимо отнести к структурным, а с точки зрения использования – к простым.

К структурным типам относятся массивы и записи (в терминологии ЯП ПЛ/1 – структуры). Такие типы данных, как базированные переменные, переменные типа метки и др., при построении МЯИ системы АПРОП

Язык Фортран. К простым типам относятся целые (INTEGER) и вещественные (REAL) числа, а также булевые (BOOLEAN) переменные. Комплексные числа (COMPLEX) могут быть представлены последовательностью только из двух вещественных чисел. Символьный тип в языке Фортран отсутствует, однако под расположение символов могут быть отведены переменные любых типов. В Фортране имеется возможность использовать шестнадцатиричные данные. Как и символьные, они могут быть расположены в переменных любых типов.

Структурные типы языка Фортран представлены только массивами. При этом в памяти они расположены по столбцам в отличие от других ЯП, где массивы располагаются по строкам.

Язык Кобол. Данные простых типов могут быть с фиксированной точкой (в том числе целые), вещественные, десятичные и типа DISPLAY. В последнем случае каждый символ данных занимает 1 байт.

Структурные типы представлены массивами и записями.

Необходимость преобразования данных для связи разноязыковых модулей обусловлена следующими причинами:

1. Язык ПЛ/1 для представления сложных типов данных (строк, массивов, записей) содержит дескрипторы – информационные векторы (ИВ). Другие ЯП дескрипторов не имеют. При передаче сложных типов данных в ПЛ-модуль необходимо построить ИВ, а из ПЛ-модуля по ИВ определить непосредственный адрес данных.

Массивы в Фортран-модулях расположены по столбцам, а для других ЯП – по строкам. Необходимы средства для транспортирования массивов.

2. Отсутствие в ЯП Фортран типа данных *запись* (или структура). При передаче данных такого типа необходимо наличие операций селектора и конструирования.

3. Отсутствие в ЯП Фортран и Кобол типов данных, эквивалентных строкам ЯП ПЛ/1. В некоторых случаях символьные строки могут быть представлены в виде массивов, а битовые – в виде массивов или простых типов данных.

4. Проблема, аналогичная преобразованию строк, возникает при передаче массивов символьных и битовых строк языка ПЛ/1.

5. Компоненты комплексных чисел в ЯП ПЛ/1 могут быть любыми числовыми значениями, в ЯП Фортран – только вещественными; в ЯП Кобол комплексные числа отсутствуют. Возможны передача и обработка комплексных чисел как массива или записи из двух компонентов.

6. В некоторых случаях возникает необходимость преобразования числовых величин в различные представления.

Проблемы передачи управления между разноязыковыми модулями вызваны следующими причинами:

а) при входе в ПЛ-модуль или Фортран-модуль необходимо установить среду функционирования для соответствующего ЯП;

б) при сопряжении разноязыковой пары модулей, одним из которых является ПЛ-модуль, необходимо специальным образом осуществлять связь цепочек областей

сохранения регистров.

Рассмотрим особенности применения операций над перечислимыми типами данных применительно к рассматриваемым ЯП:

1. Для целого типа множество операций совпадает с Ω^i (см. гл.2), кроме операции mod, которая может быть выражена через остальные.
2. Для вещественного типа множество операций совпадает с Ω^r.
3. Для символьного и булевого типов отсутствуют операции pred и succ.
4. При обработке массивов допустимо использование только отдельных элементов. В ЯП ПЛ/1 возможны операции «+» и «·» Для одномерных и двухмерных массивов, интерпретируемые как соответствующие операции над векторами и матрицами.
5. При обработке записей допустимо использование отдельных компонентов и в особых случаях всей записи.
6. Множество операций над строками в языке ПЛ/1 включает операции отношения и выбора подстроки.

Необходимо отметить особенности ЯП, упрощающие процесс информационного сопряжения модулей:

а) для простых типов данных в разных ЯП множества значений совпадают и определяются размером памяти, отводимой под переменные этих типов;

б) множества значений индексов массивов могут быть только отрезками целых типов;

в) в рассматриваемых языках отсутствует полный контроль типов операндов в операциях.

Это позволяет, например, для символьной строки ЯП ПЛ/1 использовать описание любого массива ЯП Фортран с соблюдением необходимого соотношения длины строки и размера массива.

Проведенный анализ типов данных ЯП ОС ЕС включает перечень типов данных ЯП, особенности их представления и выполнения операций над объектами рассмотренных типов, условия и особенности сопряжения разноязыковых модулей.

Полученные данные были использованы для разработки программных средств реализации функций МЯИ системы АПРОП.

Программные средства реализации функций МЯИ

Данные средства состоят из набора макроопределений и загрузочных модулей, предназначенных для сопряжения разноязыковых модулей по передаче управления и данных. Эти модули образуют библиотеку МЯИ. Подробное их описание дано в [44, 56, 121, 125].

Межъязыковый и межмодульный интерфейсы реализованы в системе АПРОП [97]. Они – базовые компоненты в системе сборке разноязыковых модулей. Далее приведены структура системы и краткое описание ее функций (рис.3.1).

В состав системы АПРОП входят компоненты реализации сборки разноязыковых программ, которые выполняют следующие функции:

– обработка паспортных данных модулей в ЯП;

– анализ описания параметров модулей, нерелевантных типов данных и проверка правильности передачи параметров по их количеству и семантике;

– преобразование простых (b–boolean, c–character, i–integer, r–real,) и сложных

a–array, z–record и др.) типов данных ЯП с помощью функций библиотеки интерфейса;

– генерация модулей-посредников и составление таблицы связи пар компонентов;

– интеграция пар модулей и их размещение в структуре программного агрегата;

– трансляция и компиляция модулей в ЯП агрегата в виде готовой программной структуры;

– трассировка интерфейсов и отладка функций модулей в каждой паре агрегата;

– тестирование программного агрегата в целом;

– формирование программ запуска программного агрегата и документации.

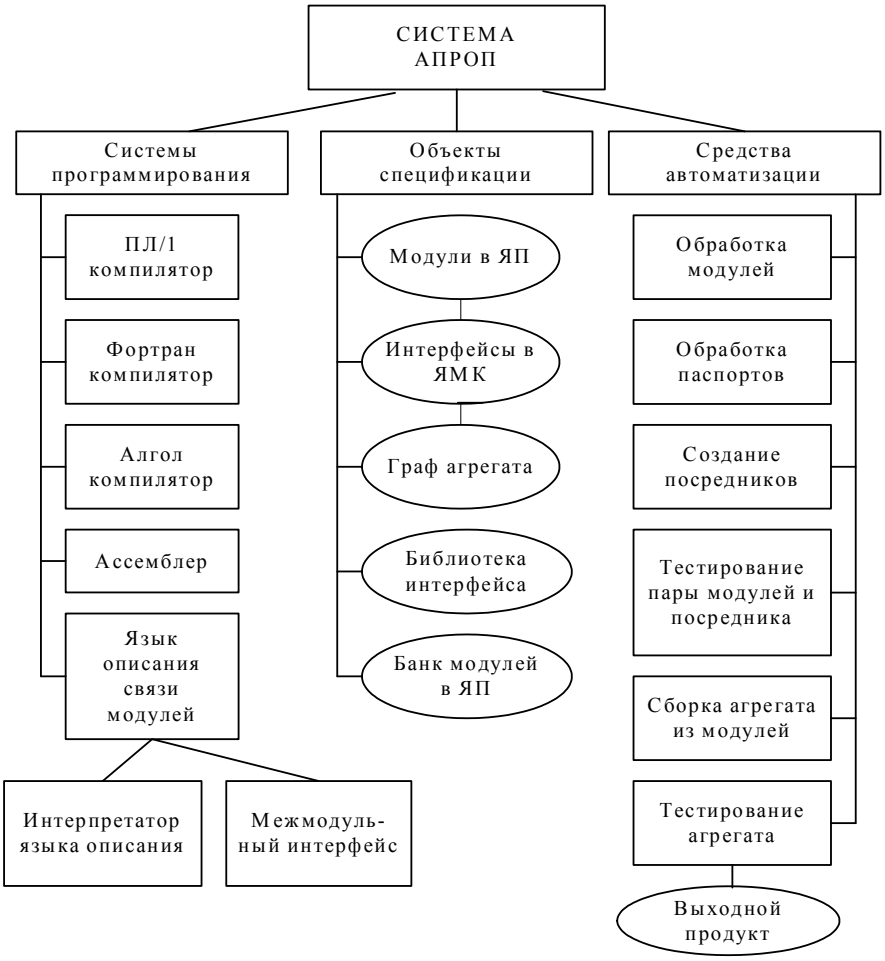


Рис 3.1. Структура системы автоматизации производства программ

В системе реализовано два типа интерфейса – интерфейс модулей и пары ЯП, т.е. межмодульный и межъязыковой интерфейсы.

Межмодульный интерфейс в системе – это компонент для генерации интерфейсных модулей-связок взаимодействующих между собой модулей.

Межъязыковой интерфейс системы – это компонент, содержащий набор

функций и макроопределений в классе типов данных и структур множества ЯП по их релевантному преобразованию, описанному выше.

Интерфейс модулей, как средство связи разных разноязыковых объектов в ЯП – первая реализация парадигмы интерфейса в программировании (1974–1985 гг.).

В задачу *межмодульного интерфейса* входит генерация интерфейсных модулей-посредников для сопрягаемых объектов с операторами конвертирования передаваемых данных и передач управления (туда и обратно).

Пример применения библиотечных функций преобразования при трансформации (TRAN) матрицы, описанной в *модуле А* на языке ПЛ1, заданной по столбцам, и в *модуле В* – на Фортране по строкам (табл.3.1).

Таблица 3.1

Модуль А	**Модуль–посредник (А', В')**	**Модуль В**
PROC OPTION **(**main**);** DCL K (4, 10), M (5, 5); **do** J=1 to 4; **do** L=1 to 10; K(J,L) = (J-1)*10+L; **end; end;** **do** J=1 to 5; **do** L=1 to 5; M(J, L)=(J-1)*5+L; **end; end;** CALL B' (K, M); **end A;**	SCECT **B': SAVEP 2;** **PLAS1** (4, 10); **PLAS1** (5, 5); **TRAN** (4, 10), 2, 1, p=0; **TRAN** (5, 5), 2, 2, p=0; ASFT CALL **B;** **A': TRAN** (4, 10), 2, 1, p=0; **TRAN** (5, 5), 2, 2, p=0; **return A; end;**	SUBROUTINE B (K, M); **integer** K(4, 10), M (5, 5); **write** (6, 1), ((K (J, L), L=1, 10, J =1, 4); **write** (6, 1) ((M (J, L), L=1,5, J =1, 5); **return** A'; **Format** (4013) **end;**

Модуль-посредник (А' или В') содержит сгенерированные операторы обращения к функциям транспонирования матрицы по строкам модуля А (точка входа В') в матрицу по столбцам для В и CALL В. После выполнения модуля В – обратные преобразования (А') для модуля А и возврат в него [132].

Система АПРОП реализована для ЯП ОС ЕС ЭВМ, внедрена в 52 организациях СССР (в Москве, Ленинграде, Минске, Риге, Ереване и др.) и у многочисленных потребителей ЯП ОС ЕС. Она была также использована, как базовая сборочная система в инструментально-технологических комплексах Прометей и РУЗА, обеспечивающих производство встроенных программ в рамках организаций министерства радиопромышленности СССР [151]. По решению этого министерства система передана в составе указанных комплексов в отраслевой Ереванский НУЦ (1984г.), которые поставлялись в разные организации страны до 1991г., где использовалась для целей автоматизации предприятий.

3.7.2. ОТЕЧЕСТВЕННЫЕ И ЗАРУБЕЖНЫЕ РАЗРАБОТКИ ИНТЕРФЕЙСА В 80-Х ГОДАХ

В период широкого использования разных ЯП ОС ЕС многие программисты сталкивались с проблемами интерфейса разноязыковых модулей в ОС ЕС. Подходы к решению задач сборки модулей отражены в многочисленных отечественных и зарубежных работах. Некоторые из них описаны ниже.

К **отечественным разработкам** относятся: интегральный подход к индустрии систем А.П.Ершова [80, 81], концепция структурного конструирования систем Е.Л.Ющенко [51], принцип многоязычности систем модульного программирования

Е.А.Жоголева [82, 83], метод комплексирования модулей В.Н.Орлова [165], модель сборки И.Н.Парасюка [169, 170] и многие другие.

Интегральный подход основывался на концепции систематизации и использования огромного запаса модулей и программ – «чистой экономии труда, которая должна дать стране 2/3 годового производства программных средств за счет совершенствования методов и средств сборочного создания программ, подобно сборки, достигнутой во многих отраслях промышленности»

Концепция структурного конструирования систем реализована в системе Мультипроцесист, в которой программы описывались алгебра алгоритмическими спецификациями системы алгоритмических алгебр [51, 206]. Основные компоненты алгебры алгоритмов – схемы программ, клоны, инструменты проектирования и синтеза программ. На данном этапе развития этого направления разработана концепция каркаса для синтеза программ и установления связей между ними, реализуется средствами системе Rational Rose.

Принцип построения многоязычной системы. Главная его особенность – это сборка многоязычной модульной структуры из разноязычных модулей на семантическом уровне, в котором накапливались основные их понятия (типы данных и их значения) в языках Фортран и Алгол. Этот уровень – база описания семантики этих ЯП. Все вопросы сопряжения разноязычных модулей описывались в машинно-зависимом языке – Автокод, что приносило определенные трудности при реализации управляющей программы для модульной структуры.

Метод комплексирования объектов идеологически – аналог метода сборки в АПРОП. В нем определены все соглашения о связях и управлении разных модулей в ЯП ОС ЕС, принятые в различных трансляторах, и дано методическое руководство по написанию *модулей-переходников* для каждой пары ЯП. Рассмотрено шесть видов пар ЯП: ПЛ1–Ассемблер, ПЛ1–Фортран, Фортран–Ассемблер и обратно. Приведены конкретные примеры модулей –переходников, как образцов для практической сборки.

Модель сборки семейства ППП ДЕЛЬТАСТАТ, которые наполнялись функциональными модулями в языке Фортран из области прикладной статистики. Для сборки ППП разработано семейство языков спецификаций моделей предметных областей и языков общения. На их основе осуществлялась автоматическая сборка Δ-систем семейства ППП, каждая из которых использовала сборочную модель вычислений программ в *динамике* выполнения.

Приведенный краткий анонс работ отражает близкие нам подходы и концепции решения задач интеграции (сборки, синтеза, объединения, комплексирования и др.) разных видов программных ресурсов. Однако ими данная проблема не исчерпывается. Необходимо упомянуть и целый ряд работ, посвященных описанию оригинальных подходов к решению данной проблемы, а также новых предложений по реализации интерфейса модулей, блоков и разнородных программ [46, 75, 76, 90, 94, 95, 106, 108, 156, 164, 184].

Зарубежные работы этого периода. Вопросами обеспечения интерфейса разнотипных модулей занимались в рамках проектов MIL, SAA, OBERON (www.oberon.ethz.ch/archives/documentation_new). Характерная особенность этих подходов – создание языков описания интерфейсов сопрягаемых объектов и реализация на их основе задач отображения типов и структур данных передаваемых через параметры. Они предвосхитили появление в 90-х годах языков

спецификации интерфейса IDL (Interface Defiinion Language), связей программных компонентов APL (Application Program Language) и др. Среди названных проектов наиболее близким к решению задач сборки разноязыковых модулей на примере языков Pascal, Modula-2 является система OBERON. Она расширена новыми возможностями по изготовлению программных продуктов в современных операционных средах.

Проблема интерфейса модулей в течение многих лет была на повестке дня многих международных и отечественных конференций. ГКНТ СССР провел международный конкурс «Интерфейс СЭВ» (1987г.), в котором принимали участие и специалисты Института кибернетики АН УССР (группы авторов от АПРОП [101] и РТК [39]). За реализацию программного и графического интерфейсов авторы были награждены специальными грамотами.

3.7.3. КРАТКИЙ ОБЗОР СОВРЕМЕННЫХ ПОДХОДОВ К РЕАЛИЗАЦИИ ИНТЕРФЕЙСОВ ЯП

Взаимодействие разноязыковых программ. Проблеме взаимодействия программ в новых, современных ЯП (C/C++, Visual C++, Visual Basic, Matlab, Smalltalk, Lava, LabView, Perl) посвящена работа И. Бея [28]. В ней отражена аналогичная концепция создания интерфейсных программ (более 60 вариантов) для практического решения проблем преобразования несовпадающих типов данных для каждой пары разноязыковых программ в этих ЯП, как средства взаимодействия между собой программ в современных средах. Этим подтверждается живучесть идеи отображения типов данных, в частности, и для новых ЯП.

Практически установлены конкретные виды связей каждой допустимой пары программ во множестве приведенных ЯП и разработаны необходимые функции отображения данных и их структур, передаваемых через операторы вызова программ в разных ЯП. Фактически описаны разные виды схем описания разноязыковых программ в ЯП, которыми могут пользоваться многочисленные программисты. Примеры обеспечения взаимодействия разноязыковых программ проиллюстрированы в наглядной форме (панели, сценарии, иконки, образцы и др.) и могут служить конкретными шаблонами для применения многими программистами (Подробнее см.п.7.4.2).

Взаимосвязь ЯП Паскаля и Дельфи. ЯП Де́льфи используется в одноимённой среде Borland. Его основу составляет Object Pascal, который сам наследник языка Паскаль. После добавления в эти языки объектно-ориентированных средств с возможностью доступа к метаданным классов (классам и их экземплярам). Так как все классы наследуют функции базового класса TObject, то любой указатель на объект можно преобразовать и воспользоваться методом ClassType и функцией TypeInfo. Объекты располагаются в динамической памяти.

Таким образом, Object Pascal и язык Delphi имеет общий базис – Turbo Pascal. Сам язык Delphi оказал огромное влияние на создание концепции языка C# для платформы .Net. Многие его элементы, функции, а также концептуальные решения по типам данных вошли в состав языка программирования C#. Иными словами, перечисленные языки имеют во многом совпадающие типы данных и поэтому существенных проблем обеспечения интерфейса программ в этих языках в основном не возникает. Кроме проблемы расположения типов данных в 16 и 32 разрядной памяти машины после компиляторов этих ЯП (См.Приложение2).

Принципы взаимодействия в среде MS.Net. Общая система типов CTS (Common Type System) в Microsoft .NET включает две группы типов данных для всех ЯП: статические типы-*значений (*value type), динамические, *ссылочные типы (*reference type), объектные, интерфейсные типы и указатели. Система CTS обрабатывает типы данных конкретных ЯП и методом отображения в системный вид и, наоборот, из системного в тип конкретного ЯП. Ссылочные типы выполняют роль формальных параметров, в которых задаются интерфейсные данные для связи разноязыковых компонентов и запоминаются в сборочном выполнимом файле, версия которого используется при преобразовании, развертывании или повторном использовании программ. Взаимно однозначное соответствие между именами *простых* типов в C# и именами FCL-*типов* (Framework Class Library) обеспечивает пространства имен с возможными 13 видами типов данных. CTS и данная библиотека – базис преобразования несовпадающих типов современных ЯП и компиляции сложных программ из программ во множестве ЯП (Подробнее см.п.7.4.3).

Об общих типах в стандарте ISO/IEC 11404. Концепция интерфейса ЯП, основанная на отображении типов и структур данных, нашла отражение в стандарте ISO/IEC 11404-2007 (General Purpose DataTypes). Данный стандарт определяет общие типы данных, включающие все существующие типы данных ЯП, а также новые и генерируемые. Эти данные описываются в универсальном языке LIDT (Language Independent DataTypes). В него включены все известные типы данных, средства их описания и преобразования, независимо от ЯП. В некотором смысле эти средства аналогичны описанию интерфейсных данных IDL, RPC и API. Язык LIDT – универсальный язык, он обеспечивает внешнее отображение типов данных ЯП в LIDT–типы данных, внутреннее отображение LIDT–типов данных в тип данных ЯП и обратное отображение.

Внешнее отображение – это преобразование типа в любом ЯП в допустимые LIDT–типы данных с проверкой их принадлежности к этому типу данных. Внутреннее отображение – это преобразование примитивных типов данных из семейства языка LIDT в конкретный внутренний тип ЯП. (в стандарте дан пример преобразования типов данных языка Паскаль в LIDT). Обратное внутреннее отображение – это преобразование одного внутреннего LIDT–типа в другое.

Данный стандарт ориентирован на программную поддержку общих типов и структур данных для вновь появляющихся версий ЯП (подробнее см.7.3.5).

Вывод. Проблема преобразования типов данных – важная и жизнедействующая на каждом этапе развития компьютеров и ЯП, используемых для описания программ в различных областях. В главе рассмотрен формальный аппарат (аксиомы, теоремы) представления типов данных ЯП в виде алгебраических систем и метод отображения несовпадающих типов данных при обращении одной разноязыковой программы к другой, использующий функции трансформации типов данных для каждой пары ЯП. Изложена методика преобразования типов данных ЯП четвертого и современного поколения (Приложение 2), а также подходы к ее реализации в современных системах и средах (CORBA, COM, .NET и др.). Кроме того, данная проблема нашла отражение в новой версии стандарта ISO/IEC 11404: 2007, ориентированного на перспективное использование общих типов данных в новых ЯП в качестве их надстройки на высоком уровне.

Глава 4

МЕТОДЫ УПРАВЛЕНИЯ МОДУЛЬНЫМИ СТРУКТУРАМИ

На основе анализа и определения задач управления модульными структурами выделено одно общее свойство – связь с операциями над модульными структурами программ. Взяв это свойство за основу межмодульного интерфейса, определим окончательный перечень задач в управлении модульными структурами:

1) выполнение операций над модульными структурами;
2) построение программных структур на основе их модульных структур;
3) построение отладочной среды модульной структуры;
4) сборка модулей в единый агрегат с учетом решения задач 1–3.

При решении первых трех задач четвертая является тривиальной, и ее обычно выполняет специальный компонент операционной системы – редактор связей в ОС ЕС, построитель задач в ОС РВ для СМ ЭВМ, комплексатор для МВК «Эльбрус» и т.д. Описание этого процесса можно найти в [10]. Поэтому подробно рассмотрим только три первые задачи.

4.1. ОПРЕДЕЛЕНИЕ МОДУЛЬНОЙ СТРУКТУРЫ ПРОГРАММНЫХ АГРЕГАТОВ

Для представления модульных структур используется математический аппарат теории графов, в котором графом G называется пара объектов $G = (X, \Gamma)$, где X – конечное множество, называемое множеством вершин, а Γ – конечное подмножество прямого произведения $X \times X \times Z$, называемое множеством дуг графа [154]. Из данного определения следует, что граф G фактически является мульти графом, так как две его вершины могут быть соединены несколькими дугами. Для отличия таких дуг вводится их нумерация целыми положительными числами.

Программная структура, состоящая из модулей (программный агрегат), описывается ориентированным графом. В нем каждая дуга соответствует оператору CALL и соединяет вершину, соответствующую вызывающему модулю, с вершиной, соответствующей вызываемому модулю. Для формального подхода к управлению модульными структурами введем несколько определений.

Определение 4.1. Модель модульной структуры программного агрегата – это объект, описываемый тройкой $T = (G, Y, F)$, где $G = (X, \Gamma)$ – ориентированный граф, являющийся графом модульной структуры; Y – множество модулей, входящих в программный агрегат; F – функция соответствия, ставящая каждой вершине X-графа элемент множества Y.

Функция F отображает X на Y:

$$F: X \rightarrow Y. \qquad (4.1)$$

В общем случае элементу из Y может соответствовать несколько вершин из X (данная ситуация, как показано ниже, характерна для динамической структуры программного агрегата).

Определение 4.2. Модульной структурой программного агрегата называется пара $S = (T, \chi)$, где T – модель модульной структуры программного агрегата; χ – характеристическая функция, определенная на множестве вершин X графа модульной структуры G.

Значение функции χ определяется следующим образом:

$\chi(x) = 1$, если модуль, соответствующий вершине $x \in X$, включен в состав программного агрегата;

$\chi(x) = 0$, если модуль, соответствующий вершине $x \in X$, не включен в состав программного агрегата, но к нему имеются обращения из других модулей, ранее включенных.

Определение 4.3. Две модели модульных структур $T_1 = (G_1, Y_1, F_1)$ и $T_2 = (G_2, Y_2, F_2)$ тождественны, если $G_1 = G_2$, $Y_1 = Y_2$, $F_1 = F_2$. Модель T_1 изоморфна модели T_2, если $G_1 = G_2$ между множествами Y_1 и Y_2 существует изоморфизм φ, а для любого $x \in X \Rightarrow F_2(x) = \varphi(f_1(x))$.

Определение 4.4. Две модульные структуры $S_1 = (T_1, \chi_1)$ и $S_2 = (T_2, \chi_2)$ тождественны, если $T_1 = T_2$ и $\chi_1 = \chi_2$ модульные структуры S_1 и S_2 называются изоморфными, если T_1 изоморфна T_2 и $\chi_1 = \chi_2$.

Понятие изоморфности модульных структур и их моделей необходимо для введения уровня абстракции, на котором определяются операции над модульными структурами. Для изоморфных объектов операции будут интерпретироваться одинаково без ориентации на конкретный модульный состав. В этом случае данные операции определяются над парами (G, χ).

Введенные выше определения описывают модульные структуры общего вида. В настоящей работе рассматриваются структуры специального вида, особенности которых состоят в следующем:

1) граф модульной структуры G имеет один или несколько компонентов связности, каждая из которых представляет ациклический граф, т. е. не содержит ориентированных циклов;

2) в каждом компоненте выделена единственная вершина, которая называется корневой и характеризуется тем, что не существует входящих в нее дуг и соответствующий ей модуль программного агрегата выполняется первым (для данного компонента связности);

3) циклы допускаются только для случая, когда соответствующий некоторой вершине модуль имеет рекурсивное обращение к самому себе. Обычно такая возможность реализуется компилятором с соответствующего ЯП и данный тип связи не рассматривается межмодульным интерфейсом. Поэтому такие дуги не включаются в граф. Исключение из рассмотрения других типов циклов связано с тем, что некоторые модули должны будут помнить историю своих вызовов, чтобы правильно вернуть управление. А это противоречит свойствам модулей, рассмотренным в гл.3;

4) пустой граф G_0 соответствует пустой модульной структуре.

В дальнейшем под терминами *модульная структура, граф модульной структуры* и т. д. будут пониматься объекты, удовлетворяющие указанным выше условиям. Типичный пример графа модульной структуры приведен на рис. 4.1.

Вершины x_1, x_2, ..., x_8 составляют множество X. Все дуги пронумерованы. Из модуля, соответствующего вершине x_5, имеются два обращения (два оператора вызова) к модулю, соответствующему вершине x_8. Множество дуг графа имеет вид $\Gamma = \{(x_1, x_2, 1), (x_1, x_3, 1), ..., (x_5, x_8, 1), (x_5, x_8, 2)\}$. В дальнейшем этот граф будет использоваться для иллюстрации операций над модульными структурами. Так, на рис.4.2. показана структура графа, соответствующая модулям x_5 и x_6, а на рис.4.3. – модульные структуры для трех видов сегментов.

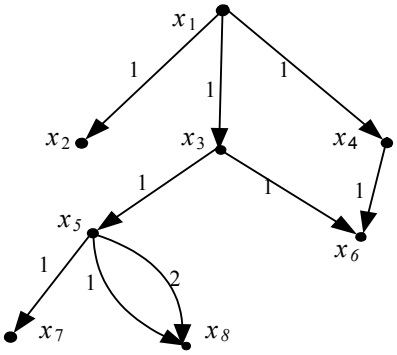


Рис. 4.1. Пример графа модульной структуры программы

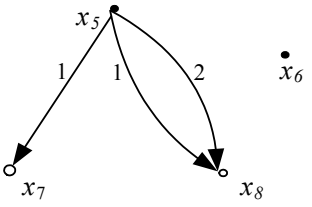


Рис. 4.2. Граф модульной структуры для модулей x_5 и x_6.

4.2. ТИПЫ ПРОГРАММНЫХ АГРЕГАТОВ

Ранее отмечено, что одной из функций межмодульного интерфейса является комплексирование различных типов программных агрегатов. Любой из них на различных процессах обработки характеризуется двумя признаками: завершенностью построения модульной структуры агрегата и признаком подчиненности. Первый признак определяет включение всех модулей в структуру. Построение модульной структуры не окончено, если имеются вершины графа с нулевым значением характеристической функции. Второй признак определяет, является ли данный программный агрегат самостоятельным объектом для выполнения или его модульная структура входит как составная часть в модульную структуру более высокого уровня.

Введем две функции согласно указанным признакам. Для признака завершенности построения функция $C(S)$ определяется следующим образом:

$C(S) = 1$, если $\chi(x) = 1$ для любых x из X;

$C(S) = 0$, если существует x такое, что $\chi(x) = 0$.

Для признака подчиненности функция $R(S)$ определяется так: $R(S)=1$, если соответствующий программный агрегат готов к выполнению; $R(S)=0$, если соответствующая модульная структура входит в состав структуры более высокого уровня.

Определим следующие типы программных агрегатов на основе введенных ранее определений.

Модуль. Модуль является программным агрегатом с графом модульной структуры $G^m = (X^m, \Gamma^m)$ с единственной вершиной $x \in X^m$, для которой $\chi(x_i)=1$. Данная вершина является корневой. Дуга вида (x_i, x_e, k), если она существует, означает

обращение из модуля, соответствующего вершине x_j , к модулю, соответствующему вершине x_i. На рис. 4.2 приведены графы двух модульных структур, соответствующие модулям x_5 и x_6.

Темный кружок соответствует вершине, для которой $\chi(x) = 1$; светлый – $\chi(x)=0$.

Сегмент. Сегмент является программным агрегатом с графом модульной структуры $G^s = (X^s, \Gamma^s)$, для которого выполняется одно из двух условий:

$C(S^s) = 0$, $C(S^s) = 1$ и $R(S^s) = 0$.

В зависимости от комбинации C и R различаются следующие виды сегментов: открытый сегмент $(C = 0, R = 0)$; сегмент, замкнутый сверху $(C = 0, R = 1)$; сегмент, замкнутый снизу $(C = 1, R = 0)$. На рис. 4.3 приведены графы модульных структур для трех видов сегментов соответственно.

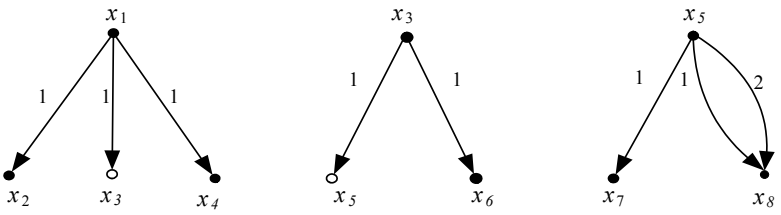


Рис 4.3. Графы модульных структур для трех видов сегментов

Программа. Программа является агрегатом с графом модульной структуры $G^p = (X^p, \Gamma^p)$, для которого выполняется $C(S^p) = 1$; $R(S^p) = 1$. Пример графа модульной структуры для программы приведен на рис. 4.1.

Комплекс. Комплекс является программным агрегатом с графом модульной структуры $G^c = (X^c, \Gamma^c)$, состоящим из n компонентов связности $(n > 1)$, каждая из которых является графом модульной структуры для соответствующей программы. Для комплекса выполняются

$$G^c = G_1^p \cup G_2^p \cup \ldots \cup G_n^p, \text{ где } X^c = X_1^p \cup X_2^p \cup \ldots \cup X_n^p,$$

$$\text{и } \Gamma^c = \Gamma_1^p \cup \Gamma_2^p \cup \ldots \cup \Gamma_n^p.$$

Данные определения модуля, сегмента, программы и комплекса не носят абсолютный характер, а введены для обозначения объектов, используемых на процессе комплексирования. Поэтому эти понятия могут отличаться от аналогичных, рассматриваемых в других контекстах.

4.3. МАТРИЧНОЕ ПРЕДСТАВЛЕНИЕ ГРАФОВ АГРЕГАТОВ ИЗ МОДУЛЕЙ

Для определения основных операций над модульными структурами используем матричное представление их графов. Матричные представления часто используются в различных работах. Например, матрицы вызовов и матрицы достижимости в [173, 183, 202, 203] эквивалентны матрицам смежности ориентированных графов.

В настоящей работе в качестве матричного представления используется матрица вызовов. Элемент матрицы m_{ij} определяет количество обращений (операторов вызова) из модуля, соответствующего индексу i, к модулю, соответствующему индексу j. Кроме матрицы вызовов для дальнейшего анализа понадобится

характеристический вектор, для каждого компонента i которого $V_i = \chi(x_i)$. Для графа модульной структуры, приведенной на рис. 4.1, характеристический вектор и матрица вызовов будут иметь вид

$$V = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \qquad M = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{4.2}$$

Проведем анализ матриц вызовов и характеристических векторов для графов модульных структур, соответствующих различным типам программных агрегатов.

Для модулей с графами, представленными на рис. 4.2 векторы и матрицы имеют следующий вид:

$$V^m_5 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad M^m_5 = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}; \quad V^m_6 = (1), \quad M^m_6 = (0). \tag{4.3}$$

Только один элемент характеристического вектора равен единице, и только в одной строке матрицы могут находиться ненулевые элементы.

Для сегментов с графами, представленными на рис. 4.3, векторы и матрицы записываются в таком виде:

$$V^s_3 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \qquad M^s_3 = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}; \qquad V^s_1 = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix},$$

$$M^s_1 = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}; \quad V^s_5 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \qquad M^s_5 = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \tag{4.4}$$

Для программы с графом, представленным на рис. 4.1 характеристический вектор и матрица вызовов совпадают с V и M соответственно и определяются (4.2). Все элементы V равны единице.

В комплексе программ характеристический вектор и матрица вызовов имеют следующий вид:

$$V^c = \begin{pmatrix} V^p_1 \\ V^p_2 \\ \dots \\ V^p_n \end{pmatrix}, \quad M^c = \begin{pmatrix} M^p_1 & 0 & \dots & 0 \\ 0 & M^p_2 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & M^p_n \end{pmatrix} \tag{4.5}$$

Здесь V^p_i и M^p_i $(i = \overline{1, \mathrm{n}})$ обозначают характеристический вектор и матрицу

вызовов для графа *i-й* программы, входящей в комплекс.

В дальнейшем матричное представление используется для анализа операций над модульными структурами.

4.4. ОТНОШЕНИЕ ДОСТИЖИМОСТИ ДЛЯ ГРАФОВ МОДУЛЬНЫХ СТРУКТУР

Пусть $G = (X, Г)$ – граф модульной структуры, x_i, x_j – вершины, принадлежащие X. Если в графе G существует ориентированная цепь от x_i к x_j, то вершина x_j - достижима из вершины x_i. Справедливо следующее утверждение: если вершина x_j достижима из x_i, а x_l – из x_j, то x_l достижима из x_i. Доказательство этого факта очевидно. Рассмотрим бинарное отношение на множестве X, которое определяет достижимость между вершинами графа. Введем обозначение $x_i \rightarrow x_j$ для достижимости вершины x_j из x_i. Отношение транзитивно. Обозначим через $D(x_i)$ множество вершин графа G, достижимых из x_i. Тогда равенство

$$\overline{x_i} = \{x_i\} \cup D(x_i) \qquad (4.6)$$

определяет транзитивное замыкание для x_i по отношению достижимости. Докажем следующую теорему.

Теорема 4.1. *Для выбранного компонента связности графа модульной структуры любая вершина достижима из корневой, соответствующей данному компоненту, т. е. выполняется равенство (x_1 – корневая вершина)*

$$\overline{x_1} = \{x_1\} \cup D(x_1) = X \qquad (4.7)$$

Доказательство. Предположим, вершина x_i $(x_i \in X)$ недостижима из x_1. Тогда $x_i \notin \overline{x_1}$ и множество $X' = X \setminus \overline{x_1}$, непусто. Поскольку выбранный компонент графа связанный, то существуют вершина $x_j \in \overline{x_1}$ и цепь $H(x_i, x_j)$, ведущая от x_i к x_j. Исходя из ацикличности графа G, в X' должна существовать простая цепь $H(x_l, x_j)$, где в вершину x_l не входят дуги (данная цепь может быть пустой, если X' состоит только из x_i). Рассмотрим цепь $H(x_l, x_j) = H(x_l, x_i) \cup H(x_i, x_j)$. Это означает, что модуль x_j достижим из вершин x_1 и x_i и обе вершины не содержат входящих дуг. А это противоречит определению графа модульной структуры с единственной корневой вершиной. Теорема доказана.

Результаты данной теоремы важны для обоснования требования отсутствия ориентированных циклов в графе модульной структуры относительно понятия достижимости. Рассмотрим граф, приведенный на рис. 4.4.Из этого рисунка ясно, что граф содержит ориентированный цикл и модули, соответствующие вершинам x_4, x_5, x_6 никогда выполняться не будут.

Таким образом, результаты теоремы 4.1 усиливают требование необходимости отсутствия ориентированных циклов в графе модульной структуры.

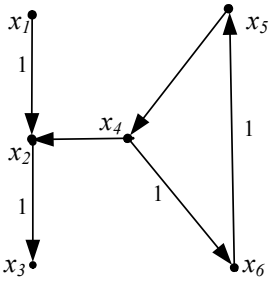


Рис. 4.4. Граф, содержащий ориентированный цикл

Проведем анализ матричного представления отношения достижимости для графов модульной структуры. В качестве примера рассмотрим граф для матрицы достижимости А, приведенной ранее на рис. 4.1.

Коэффициент a_{ij} = 1, если модуль, соответствующий индексу l, достижим из модуля, соответствующего индексу i Следующие результаты основаны на известной теореме из теории графов.

$$\mathbf{A}=\begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad (4.8)$$

Теорема 4.2. *Коэффициент m_{ij} l-й степени матрицы смежности M^l определяет количество различных маршрутов, содержащих l дуг и связывающих вершину x_i с вершиной x_j –ориентированного графа.*

Доказательство этой теоремы приводится в [99]. Рассмотрим следующие три следствия из этой теоремы.

Следствие 4.2.1. Матрица $\overline{M} = \sum_{l=1}^{n} M^i$, где M – матрица смежности ориентированного графа с n вершинами, совпадает с точностью до числовых значений коэффициентов с матрицей достижимости A.

Доказательство. В ориентированном графе, содержащем n вершин, максимальная длина пути без повторяющихся дуг не может превышать n. Поэтому последовательность степеней матрицы смежности M^i, где i = 1,2, …, n определяет количество всех возможных путей в графе с количеством дуг $\le n$. Пусть коэффициент $\overline{m_{ij}}$ матрицы M отличен от нуля. Это означает, что существует степень матрицы M^i, у которой соответствующий коэффициент m_{ij} также отличен от нуля. Следовательно, существует путь, идущий от вершины x_i к x_j, т. Е. вершина x_j достижима из x_i. Данное следствие определяет связь матрицы вызовов графа модульной структуры, совпадающей с матрицей смежности M, с матрицей достижимости A и определяет алгоритм построения последней.

Следствие 4.2.2. Пусть для некоторого i в последовательности степеней матрицы смежности M^i существует коэффициент $m_{ii} > 0$. Тогда в исходном графе существует ориентированный цикл.

Доказательство. Пусть $m_{ii} > 0$ для некоторого l. Следовательно, x_l достижима из x_i, т.е. существует цикл. Согласно теореме 4.2 данный цикл имеет l дуг (в общем случае повторяющихся).

Следствие 4.2.3. Пусть n-я степень матрицы смежности M^n ациклического графа совпадает с нулевой матрицей (все коэффициенты равны нулю).

Доказательство. Если граф ациклический, то в нем максимально простой путь не может иметь больше чем n – 1 дуг. Если в M^n имеется коэффициент, отличный от нуля, то должен существовать путь, состоящий из n дуг. А этим путем может быть только ориентированный цикл. Следовательно, все коэффициенты M^n для ациклического графа равны нулю. Данное следствие предоставляет необходимое и достаточное условие отсутствия циклов в графе модульной структуры.

Для ациклических графов отношение достижимости эквивалентно частичному строгому порядку. Транзитивность отношения достижимости рассмотрена выше.

Антисимметричность следует из отсутствия ориентированных циклов: если вершина x_j достижима из x^l, то обратное неверно. Введем обозначение $x_i > x_j$ если вершина x_j достижима из вершины x_i.

Пусть G = (X, Γ) – ациклический граф, соответствующий некоторой модульной структуре. Рассмотрим убывающую цепь элементов частично упорядоченного множества X:

$x_{i1} > x_{i2} > \ldots > x_{in}$. …,

где через ">" обозначено отношение достижимости. Поскольку X конечно, то цепь обрывается $x_{i1} > x_{i2} > \ldots > x_{in}$. Вершина x_{in} не имеет исходящих дуг, т. е. элемент x_{in} минимальный (ему соответствует модуль, который не содержит обращения к другим модулям). Максимальный элемент во множестве X – корневая вершина.

4.5. ОПЕРАЦИИ ПОСТРОЕННИЯ МОДУЛЬНЫХ СТРУКТУР

В п. 4.1 было отмечено, что операции над модульными структурами выполняются на уровне абстракции, определяемом понятием изоморфизма модульных структур. Над изоморфными структурами операции выполняются одинаково, поэтому преобразование модульных структур будет рассматриваться как изменение их графов и характеристических функций, т.е. $S = (G, \chi)$

Пусть $S_1 = (G_1, \chi_1)$ *и* $S_2 = (G_2, \chi_2)$ – две модульные структуры с графами $G_1 = (X_1, \Gamma_1)$ и $G_2 = (X_2, \Gamma_2)$ соответственно. Введем следующие обозначения:

$D(x)$ – множество вершин, достижимых из вершины x;

$D^*(x)$ – множество вершин, из которых достижима вершина x.

Для одинаковых вершин, входящих в графы G_1 и G_2, будут использоваться одинаковые обозначения.

Рассмотрим основные операции над модульными структурами. Операция *объединения*

$$S = S_1 \bigcup S_2 \tag{4.9}$$

предназначена для формирования модульной структуры комплекса и формально определяется следующим образом (S_1 и S_2 – любые модульные структуры, удовлетворяющие определению в п. 4.1):

$$G = G_1 \oplus G_2, \quad X = X_1 \oplus X_2, \quad \Gamma_1 \oplus \Gamma_2, \tag{4.10}$$

где символ $\oplus$ обозначает прямую сумму, и

$\chi(x) = \chi_1(x)$, если $\chi \in X_1$,

$\chi(x) = \chi_2(x)$, если $\chi \in X_2$.

Одинаковые вершины, входящие в G_1 и G_2, операцией объединения модульных структур рассматриваются как разные объекты. Поэтому характеристический вектор и матрица вызовов для модульной структуры S определяются так:

$$V_{1,2} = \begin{pmatrix} V_1 \\ V_2 \end{pmatrix}, \; M_{1,2} = \begin{pmatrix} M_1 & 0 \\ 0 & M_2 \end{pmatrix}, \tag{4.11}$$

где $V_{1,2}$ и $M_{1,2}$ – характеристические векторы и матрицы вызовов для модульных структур S_1 и S_2 соответственно. Операция объединения ассоциативна, но не коммутативна – порядок следования операндов определяет порядок выполнения компонентов комплекса. Необходимо отметить, что если операнды S_1 и S_2 удовлетворяют условиям определения модульных структур, то результат S также будет удовлетворять тем же требованиям.

Операция объединения модульных структур увеличивает число компонентов связности соответствующего графа. Кроме того, графы структур, соответствующие операндам, сами могут иметь несколько компонентов связности. Для остальных операций графы модульных структур операндов и результата имеют единственный компонент связности.

Рассмотрим операцию *соединения.* Через x_i и x_j обозначим корневые вершины для графов G_1 и G_2 модульных структур S_1 и S_2 соответственно. Если данные структуры удовлетворяют условиям :

множество $X' = X_1 \bigcap X_2$ непусто;

вершина $x_j \in X'$ и $\chi(x_j) = 0$;

$D^*(x) \bigcap D(x) = 0$ для каждого $x \in X'$, где $D^*(x) \in X_1$ и $D(x) \in X_2$, то операция соединения, обозначаемая

$$S = S_1 + S_2, \tag{4.12}$$

определяется следующим образом:

$$G = G_1 \bigcup G_2, \quad X = X_1 \bigcup X_2, \quad \Gamma = \Gamma_1 \bigcup \Gamma_2, \tag{4.13}$$

и для характеристической функции χ выполняется:

$\chi(x) = \chi_1(x)$, если $x \in X_1 \setminus X'$;

$X(x) = max\,(\chi_1(x), \chi_2(x))>$ если $x \in X'$,

$\chi(x) = \chi_2(x)$, если $x \in X_2 \setminus X'$.

Первое условие означает, что в графах G_1 и G_2 имеются общие вершины. Согласно второму условию корневая вершина G_2 принадлежит общей части и для S_1 объект, соответствующий x_j, еще не включен в модульную структуру. Третье условие запрещает существование циклов в графе результата. Действительно, если существует $x_n \in D^*(x) \bigcap D(x)$, то $x_n > x$ и $x > x_n$, что означает существование цикла. Так как не все S_1 и S_2 удовлетворяют приведенным выше условиям, то операция частичная.

Определим принадлежность результата операции соединения к классу рассматриваемых модульных структур. Поскольку X' непусто, то граф G имеет один компонент связности. Корневой вершиной графа G является x_i. Сам граф G не имеет ориентированных циклов, т. е. ацикличен. Таким образом, S принадлежит к классу рассматриваемых модульных структур.

Операция соединения не коммутативная и в общем случае не ассоциативна. Чтобы показать последний факт, рассмотрим результат $S = (S_1 + S_2) + S_3$, где корневые вершины графов G_2 и G_3 входят в состав вершин графа G_1 и $X_2 \bigcap X_3 \neq 0$. Тогда результат $S_2 + S_3$ не определен.

Рассмотрим операцию **проекции**. Пусть $S_1 = (G_1, \chi_1)$ – модульная структура и $x_i \in X_1$. Операция проекции модульной структуры на вершину графа S_1, обозначаемая как $S = P_{rxi}(S_1)$, определяется следующим образом:

$$G(X, \Gamma), \quad X = \overline{x}_i \ , \ \Gamma = \{(x_i, x_j, K) | \ x_i, x_j \in X\}, \tag{4.14}$$

и для характеристической функции

$\chi(x) = \chi_1(x)$, если $x \in X$.

Операция проекции определяет из модульной структуры S_1 подструктуру S. Проверим принадлежность S классу рассматриваемых модульных структур. Если граф модульной структуры S_1 связан и ацикличен, то теми же свойствами будет обладать и граф S. Существует единственная корневая вершина x_i в графе G. Таким образом, модульная структура S принадлежит рассматриваемому классу.

Операция **разности** для модульных структур определяется следующим образом. Пусть $S_1 = (G_1, \chi_1)$ – модульная структура и $x_i \in X_1$. Операция разности выполняется между модульной структурой и ее проекцией на вершину x_i соответствующего графа (x_i не является корневой вершиной графа G_1). Формально операция разности модульной структуры

$$S = S_1 - P_{r_{xi}}(S_1), \tag{4.15}$$

определяется следующим образом:

$$G = \{X, \Gamma), \quad X = (X_1 \setminus \overline{x_i}) \cup X' \tag{4.16}$$

$$\Gamma = \{(x_i, x_j, K) \mid x_i, x_j \in X\},$$

где множество X' состоит из таких элементов, для которых

$$X' = \{x'_j \mid (x_1 \in X_1 \setminus x_i) \,\&\, (x'_j \in \overline{x_i}) \,\&\, (x_1, x'_j, K) \in \Gamma\} \tag{4.17}$$

Характеристическая функция χ определяется так:

$$\chi(x) = \chi_1(x), \text{ если } x \in X_1 \setminus \overline{x_i};$$

$$\chi(x) = 0), \text{ если } x \in X'.$$

Во множество X включаются вершины, которые не вошли во множество $\overline{x_i}$, и те из вершин $\overline{x_i}$ в которые входят дуги из вершины $X_1 \setminus \overline{x_i}$ (множество X'). Характеристическая функция для элементов $x' \in X'$ равна нулю. Операция разности модульных структур определена так, чтобы быть обратной к операции соединения, т. е. чтобы выполнялось равенство

$$S - P_{r_{xi}}(S) + - P_{r_{xi}}(S) = S. \tag{4.18}$$

Проверим принадлежность S, определяемой в (4.15), к классу рассматриваемых модульных структур. Если граф G, связан и ацикличен, то этими же свойствами будет обладать граф G_1. Корневая вершина G совпадает с корневой вершиной G_1. Таким образом, S удовлетворяет условиям определения модульной структуры, приведенным в п. 4.1.

Пусть S^* обозначает множество модульных структур, заданное на прямом произведении $G^* \times \chi^*$. где G^* и χ^* – соответственно множество графов и характеристических функций. Обозначим через Ω множество введенных операций над модульными структурами и предикаты C и R, рассмотренные в п. 4.2:

$$\Omega = \{\cup, +, -, P_r, C, R\}. \tag{4.19}$$

Этим мы определяем частичную алгебраическую систему $U = (S^*, \Omega)$ над множеством модульных структур. Согласно определению типов программных агрегатов и операций над модульными структурами необходимо отметить следующее:

операция объединения применяется для программ и комплексов, результатом является комплекс;

операция соединения применяется для модулей и сегментов, результатом могут быть сегмент или программа;

операция проекции используется для программ и сегментов, результатом является модуль или сегмент;

операция разности применяется для программ и сегментов, результатом может быть модуль или сегмент.

4.6. ПРОЦЕСС ПОСТРОЕНИЯ МОДУЛЬНЫХ СТРУКТУР

Рассмотрим основные задачи построения программных агрегатов и алгоритмы их решения. Они взяты из практики комплексирования программных средств и этим во многом определилась реализация алгоритмов решения этих задач.

Задача 1. Дано множество модулей, входящих в состав программы, и имя главного (корневого) модуля. Построить модульную структуру программы.

Задача 2. Дано множество модулей, входящих в состав программы. Имя главного модуля неизвестно. Построить модульную структуру.

Задача 3. Дано множество модулей, реализующих некоторые функции предметной области, и имя главного модуля для одной из программ. Построить модульную структуру программы.

Задача 4. Дано множество модулей реализации функции предметной области и последовательность имен главных модулей нескольких программ. Построить модульную структуру для комплекса программ.

Задача 5. Заменить в модульной структуре один или несколько модулей новыми.

Задача 6. Выделить из модульной структуры объекты и включить их в другую структуру. В этой задаче под объектами понимается любая часть модульной структуры, т.е. любой подграф ее графа.

При решении данных задач будут использоваться введенные операции над модульными структурами. Для практического применения используется матричное представление объектов, а в качестве операций рассматриваются преобразования соответствующих матриц. Так, множества, характеристические функции представляются векторами, графы – матрицами вызовов и т. д. Аналогичные представления применяются и для объектов, используемых в алгоритмах решения задач. Технические аспекты реализации матричных представлений объектов и операций принципиальных трудностей не представляют и в данной работе не рассматриваются.

Рассмотрим решение каждой из поставленных задач, используя операции над модульными структурами.

1. Пусть X^p – множество вершин графа, соответствующее множеству модулей программы. Упорядочим его так, чтобы для каждых x_i, $x_j \in X^p$ из условия, что модуль, соответствующий x_i, вызывает модуль, соответствующий x_j, следует, что $j > i$. Если X^p таким образом не удается упорядочить, то в графе модульной структуры возникнут циклы, что противоречит необходимости ацикличности графа.

Обозначим вершину, соответствующую главному модулю, через x_1. Основной метод решения данной задача состоит в постепенном наращивании модульной структуры «сверху-вниз». С каждым модулем, соответствующим вершине x_i, связана модульная структура $S_i^m = (G_i^m, \chi_i^m)$, где $G_i^m = (\chi_i^m, \Gamma_i^m)$. Множество χ_i^m включает x_i и вершины, соответствующие модулям, к которым имеются обращения из данного модуля. Множество Γ_i^m соответствует множеству вызовов из модуля, соответствующего x_i. Характеристическая функция для x_i равна единице и нулю – для остальных вершин. Если из модуля, соответствующего x_i, нет обращения к другим модулям, то $X_i^m = \{x_i\}$, $\Gamma_i^m = 0$, $\chi(x_i) = 1$. Пусть Y – множество вершин графа, для которых $\chi = 0$.

Шаг I. Вводим начальные значения: $S = S_i^m$; $Y := X_1^m \setminus \{x_1\}$.

Шаг 2. Если $C(S) = 1$, то перейти на шаг 6.

Шаг 3. Выберем первый элемент $x_i \in Y$. Множество Y непусто и конечно.

Шаг 4. S:= S + S_i^m, $Y := (Y \setminus \{x_i\}) \cup (x_i^m \setminus \{x_i\})$. На данном шаге происходит наращивание модульной структуры и изменение множества Y.

Шаг 5. Переход на шаг 2.

Шаг 6. Выход.

Докажем корректность и сходимость данного алгоритма. На шаге 4 выполняется операция соединения структур. Данная операция корректна, так как ее условия выполняются:

$$x_i \in X \cap X_i^m,$$

$$\chi(x_i)=0, \quad X_i^m(x_i)=1,$$

циклы отсутствуют согласно упорядоченности множества X^p.

Рассмотрим изменение множества Y на шаге 4. Согласно результатам п. 4.4, в графе модульной структуры обязательно существуют вершины, из которых исходят дуги. Исходя из этого факта и конечности множества X^p, следует, что существует x_i такое, что $Y\setminus\{x_i\}=0$ и $x_i^m \setminus \{x_i\}=0$. В результате условие шага 2 истинно и осуществляется переход на шаг 6 – завершение работы алгоритма.

Необходимо отметить, что данный алгоритм имеет практическое воплощение в процессе комплексирования, основанном на применении специального компонента операционной системы – редактора связей, построителя задач, комплексатора и т. д.

2. Для решения второй задачи достаточно упорядочить множество X^p, как и для условия первой. Тогда первый элемент X^p соответствует главному модулю и можно воспользоваться алгоритмом решения первой задачи.

3. Для решения третьей задачи проведем аналогичное упорядочение модулей предметной области. Единственное условие состоит в том, чтобы $x_1 \in X$ соответствовал главный модуль программы. Тогда можно воспользоваться алгоритмом решения первой задачи.

4. Пусть X^c – множество вершин, соответствующих модулям реализации функции предметной области, и $x' = \{x_1, x_2, ..., x_n\}$ – множество вершин, соответствующих главным модулям программ $\{X' \subset X^c\}$. Алгоритм решения следующий.

Шаг 1. Начальное значение: $i := 1$, $S := 0$.

Шаг 2. Выбрать $x_i \in X'$ и построить модульную структуру с главным модулем, соответствующим корневой вершине x_i . Построение выполняется согласно алгоритму третьей задачи. Пусть S_i^p – соответствующая модульная структура.

Шаг 3. $S := S \cup S_i^p$, На данном шаге используется операция объединения модульных структур.

Шаг 4. $i := i + 1$. Если $i \le n$, то переход на шаг 2.

Шаг 5. Выход.

Корректность и сходимость данного алгоритма очевидны и не требуют доказательства.

5. Пусть $S = (G, \chi)$ – модульная структура с графом $G = (X, \Gamma)$. Необходимо заменить модуль программы, соответствующий вершине $x_i \in X$, новым модулем x'_i. Сложность данной задачи состоит в том, что с заменой модуля может измениться модульная структура S, так как x_i в общем случае является корневой вершиной для

некоторого графа. Данный алгоритм будет иметь следующий вид.

Шаг 1. Построить модульную структуру S'_i для корневой вершины x'_i согласно алгоритмам задач 1 – 3 (предполагается, что множество модулей для данной структуры известно до ее построения).

Шаг 2. $S: = S - P_{rx_i}(S)$. Исключается соответствующая часть модульной структуры S.

Шаг 3. Переобозначим x'_i через x_i, а S'_i – через S_i. Выполним операцию: $S = S + S_i$.

Шаг 4. В результате предыдущей операции в графе G модульной структуры могут существовать вершины с $\chi = 0$, т. е. $C(S) = 0$. Тогда необходимо применить алгоритм первой задачи, полагая в качестве множества Y множество таких вершин.

Выполнение операции соединения на шаге 3 должно удовлетворять необходимым условиям.

6. Данная задача подобна задаче 5. Пусть $S_1 = (G_1, \chi_1)$ – исходная модульная структура с графом $G_1 = (X_1, \Gamma_1)$ и $x_i \in X_1$. Граф $G_2 = (X_2, \Gamma_2)$ модульной структуры $S_2 = (G_2, \chi_2)$ имеет вершину x_i с $\chi(x_i) = 0$. Тогда алгоритм решения данной задачи следующий.

Шаг 1. $S_i = P_{rxi}(S_1)$.

Шаг 2. $S: = S_2 + S_i$. Операция соединения модульных структур должна удовлетворять необходимым условиям.

Шаг 3. Если $C(S) = 0$, то выполнять действия, аналогичные шагу 4 пятой задачи.

Как видно из приведенных выше алгоритмов, для решения задач 1– 6 использовались все четыре операции над модульными структурами.

4.6.1. ТИПЫ ПРОГРАММНЫХ СТРУКТУР

Среди многообразия программных структур выделяются три основные – простая, сложная (структура с динамическим вызовом модулей из внешней среды) и динамическая. Основное назначение различных структур – наиболее оптимальное использование основной памяти во время выполнения программного агрегата. Поэтому тип программной структуры относится к динамическим характеристикам соответствующей модульной структуры.

Простая структура. Программный агрегат с простой структурой создается на процессе комплексирования, где все связи между модулями фиксированы и не меняются во время выполнения. Объем основной памяти, занимаемой программным агрегатом с данным типом структуры, постоянен и равен сумме объемов отдельных модулей: $V_s = \sum_{i=1}^{n} v_i$, где v_i – объем памяти, занимаемый i-м модулем. Соответствующий граф модульной структуры всегда связан.

Сложная структура. Программный агрегат сложной структуры с динамическим вызовом модулей на одну общую память создается на процессе комплексирования модулей. В таком агрегате связи между модулями не такие жесткие, и их последовательность определяется относительно входящих в одну цепочку модулей. Модули в процессе выполнения загружаются в основную память в момент обращения к ним. После завершения работы память освобождается и может быть использована для загрузки другого модуля.

Компонент программного агрегата, загружаемый при однократном обращении, называется оверлейным сегментом. Его граф соответствует графу модульной структуры для сегмента с перекрытием памяти. В один такой сегмент включаются модули, между которыми существуют частые обращения, что экономит время выполнения программного агрегата за счет уменьшения количества операций удаленного ввода с устройств внешней памяти компьютера.

Как и для случая простой структуры, граф соответствующей сложной модульной структуры также связный (рис.4.5), и матрица вызовов имеет обычный вид, например (4.2). Объем требуемой основной памяти зависит от количества и состава модулей в сегменте. Максимальный объем памяти равен сумме объемов памяти отдельных

модулей: $v_0^{\max} = V_s = \sum_{i=1}^{n} v_i$.

Минимальный объем памяти, требуемый при выполнении, равен максимальному объему памяти среди всех возможных цепочек программного агрегата. Объем памяти проводится по алгоритму Флойда, предназначенного для определения кратчайшего пути в графе, в котором каждой дуге соответствует весовой коэффициент, называемый длиной дуги.

Для рассматриваемого случая коэффициенты соответствуют вершинам графа модульной структуры. Для применения алгоритма Флойда выполняем следующие преобразования.

1. Дополним граф модульной структуры новыми вершинами и дугами. Вершинами являются $x_0, x_{n+1}, \ldots, x_{n+m}$, где m-количество концевых вершин.

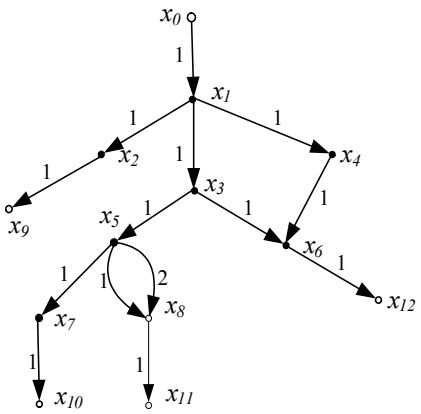


Рис. 4.5. Граф модифицированной модульной структуры

Новые дуги – это $(x_0, x_1, 1)$, $(x_{r1}, x_{n+1}, 1)$, ..., $(x_{rn}, x_{n+rn}, 1$. В них x_1 соответствует главному модулю, а все x_i – концевым вершинам. После выполнения таких операций граф модульной структуры на рис. 4.1 приведен к графу рис. 4.5. В нем новые вершины – $x_0, x_9, x_{10}, x_{11}, x_{12}$. Всем им ставим в соответствие весовые коэффициенты, равные нулю:

$$v_0 = v_9 = v_{10} = v_{11} = v_{12} = 0 .$$

2. Каждой дуге вида (x_i, x_j, k) ставим в соответствие коэффициент $v_{ij} = \dfrac{v_i + v_j}{2}$.

Рассмотрим все маршруты, ведущие от x_0 к одной из остальных дополнительных

вершин. Длина маршрута кратчайшего пути вычисляется так

$$l_{0,n+p}=v_{01}+\ \dots\ +v_{rp,n+p}=\frac{v_0+v_1}{2}+\dots+\frac{v_{2p}+v_{n+p}}{2}=\frac{v_0}{2}\ +v_1+\dots+v_{rp}+\frac{v_{n+p}}{2}\ =v_{1+\ \dots\ +}\ v_{rp}.$$

Эта длина $l_{0,\ n+p}$ будет равна сумме объемов памяти модулей для пути $x_1, \dots, x_{rp}$. Таким образом, применял алгоритм Флойда к графу, изображенному на рис. 4.2, мы решаем задачу вычисления объема памяти для максимальной цепочки.

3. Матрицу вызовов заменим матрицей путей. Для каждого $m_{ij}>0$ на соответствующем месте будет находиться значение v_{ij}. Значения $m_{ij} = Ø$ заменяются на $-\infty$. Программа, реализующая алгоритм Флойда, имеет следующий вид (предполагается, что матрица путей описана как двухмерная матрица $n\times n$):

```
for k = 1 to n do
for i = 1 to n do
for j = 1  to n do
if M[i, j ] < M [ i , k] + M[k, j] then
M[i, j]: = M [ i , k] + M[k, j].
```

В результате работы этого алгоритма будет построена матрица максимальных путей. Максимальное из значений $l_{0,n+p}$ будет определять необходимый минимальный объем памяти V_0^{min} для программного агрегата с перекрытием памяти. Саму сложную структуру для значений $V_0^{min} \le V_0 \le V_0^{max}$ можно построить, следуя алгоритмам, предложенным в [101, 102].

Качественная зависимость V_0 от числа динамических сегментов представлена на рис.4.6. Здесь n – число модулей в программном агрегате. Несмотря на различный вид кривых, они имеют общую закономерность– любое V_0 заключено между значениями v_0^{max} и v_0^{min}.

Динамическая структура. Механизм динамической связи между модулями отличен от механизма обращения с помощью оператора вызова CALL. Поэтому для описания графа модульной структуры необходимы дополнительные средства. Как и для оверлейной структуры, загрузка в основную память динамических объектов происходит при обращении к ним. По аналогии назовем объем, загружаемый при однократном обращении, динамическим сегментом. Каждый динамический сегмент имеет свою собственную модульную структуру, для которой составляется матрица вызовов. Если в разных динамических сегментах встречаются одинаковые модули, то они являются разными объектами в графе модульной структуры. Для иллюстрации используем исходный граф (см. рис.4.1). Пусть из модуля, соответствующего вершине x_1, динамически вызывается модуль, соответствующий вершине x_3. Полученный измененный граф приведен на рис. 4.7. Пунктирная стрелка обозначает динамический вызов. Модуль, соответствующий вершине x_6, встречается дважды.

Построим матрицу вызовов для данного программного агрегата. Каждому динамическому сегменту будет соответствовать своя клетка. Чтобы отличить динамический вызов от вызовов типа CALL, соответствующие элементы матрицы будут содержать отрицательные числа, абсолютные значения которых будут определять количество динамических вызовов между данными парами модулей. Матрица вызовов будет иметь следующий вид.

$$M=\begin{pmatrix} x_1 & x_2 & x_4 & x_6 & x_3 & x_5 & x_6 & x_7 & x_8 \\ 0 & 1 & 1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad (4.20)$$

Исследуем качественную зависимость объема оперативной памяти от количества динамических сегментов (рис.4.6. и 4.7). При одном сегменте в программном агрегате простой структуры имеем $V^1_d = V_s$. Если каждый динамический сегмент состоит из одного модуля, то по модифицированному алгоритму Флойда находится максимальный путь и $V_d^n = V_0^{min}$.

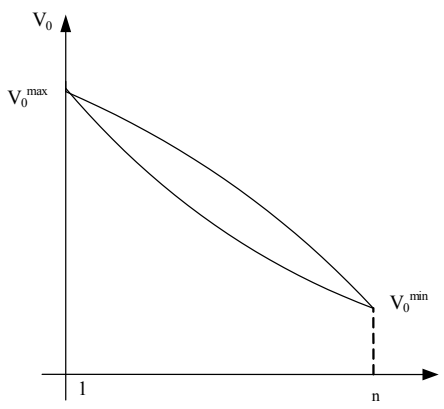


Рис. 4.6. Графики качественной зависимости V_a от количества сегментов

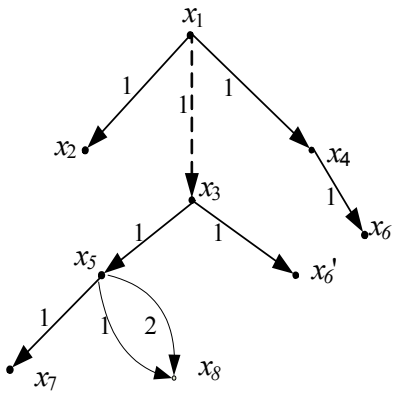


Рис. 4.7. Граф модульной структуры для динамического вызова

Для промежуточных значений зависимость имеет более сложный характер. На рис. 4.8 представлены две кривые и n – число модулей в программном агрегате.

Кривая *1* описывает зависимость, при которой в разных сегментах нет одинаковых модулей. Кривая *2* описывает зависимость для случая, когда у разных сегментов имеются одинаковые модули. Требуемая для них память увеличивается за счет дублирования таких модулей, аналогично рассмотренного выше примера. Однако вторая зависимость характерна и для случая, когда в динамических сегментах нет одинаковых модулей, но сами модули написаны в ЯП высокого уровня.

Это вызвано тем, что в состав каждой динамической части программы включаются одинаковые служебные утилиты – управление памятью, вводом-выводом, обработки аварийных ситуаций и т.д.

За счет дублирования этих программ происходит увеличение необходимого объема основной памяти.

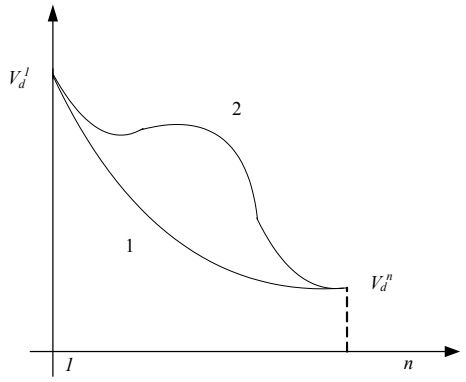


Рис. 4.8. Графики зависимости V_a от количества динамических сегментов

Таким образом, кривая под номером 1 характерна только для программных агрегатов, состоящих из модулей, написанных на языках типа Ассемблера, и графов модульной структуры, представленных в виде дерева. Это гарантирует отсутствие в разных динамических сегментах одинаковых модулей.

Иными словами, динамическая структура имеет существенное преимущество, так как она не зависит от редактирования связей.

4.6.2. МЕТОД РЕАЛИЗАЦИИ СВЯЗЕЙ В МОДУЛЬНОЙ СТРУКТУРЕ

Модульная структура, представленная графом G и определенная на множестве модулей $Y – \{y_1, y_2, ..., y_m\}$, описываемых в классе языков L (ПЛ/1, Фортран, Кобол, Ассемблер для ОС ЕС), служит основой для автоматизированного комплексирования по ней модулей в программный агрегат. При этом для каждой пары модулей y_i, y_j $(i, j$ – языки из множества $L)$, связанных на графе отношением вызова CALL, формируется модуль связи y'_{ij}. В общем случае для простых структур программ он содержит операторы прямого и обратного преобразований типов данных, передаваемых от вызывающего модуля (в i-языке) вызываемому (в j-языке) и обратно.

Для построения программных структур, в графе которых вершины – модули отмечены еще и специальными символами ρ, указывающими на тип вызова (динамический, сложной, подзадачный), в модуле связи, кроме указанных операторов преобразования типов данных, размещаются и операторы формирования среды выполнения соответствующей части агрегата.

Символ ρ может принимать следующие значения:

ρ = ¤ – означает формирование оверлейного сегмента, начиная с имени модуля, который этот символ помечает;

ρ = * – определяет начало динамического сегмента с вершины, помеченной в графе этим символом;

ρ = + – отмечает в графе G модуль, который должен быть сформулирован как главный модуль подзадачи;

ρ = / – означает включение отладочной среды в агрегат.

Используя эти обозначения, граф, ранее приведенный на рис. 4.1, примет новый вид, изображенный на рис.4.9. В нем для сегмента, заданного графом

$$\Gamma = \{(x_5, x_7, 1),\ (x_5, x_8, 1),\ (x_5, x_8, 2)\}$$

и помеченного именем x_5 со знаком ¤, в модуле связи x'_{58} будет сформирован фрагмент из операторов, обеспечивающих вызов с перекрытием памяти;

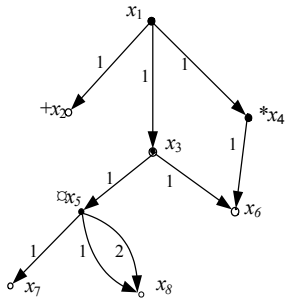


Рис.4.9.Граф программного агрегата с управляющими отметками

для сегмента заданного графом Г = $\{(x_4, x_6, 1)\}$, в модуле связи x'_{46} будет сформирован фрагмент из операторов, обеспечивающих динамический вызов.

Таким образом, для пары модулей x_i, x_j создается модуль связи x_{ij} вида

$$x'_{ij} = S_0 * (S_1 \times S_1^T) * (S_2 \times S_2^T) * S_0^1,$$

где S_0 – фрагмент агрегата, задающий среду функционирования модуля x_j;

S_1 – фрагмент агрегата, включающий последовательность обращений к функциям из множества {P, C, S} каждая из которых осуществляет необходимое преобразование фактических параметров при обращении на x_j-модуль;

S_1^T, $S_2^T \neq \emptyset$ означает наличие средств создания отладочной среды (по ρ=1 в графе модульной программы) для вершин x_j и x_i соответственно;

S_2 – фрагмент операторов по обратному преобразованию типов данных, передаваемых из x_j в x_i после его выполнения;

S_0^1 – фрагмент операторов эпилога модуля вершина x_i для восстановления среды.

Пример. Для пары модулей, заданных на графе рис.4.9 вершинами x_4, x_6, структура соответствующей части программного агрегата, включая модуль связи, изображена на рис.4.10. Аналогично реализуются связи разноязыковых модулей и для других видов вызова. Автоматизация связи пары разноязыковых модулей в модульной структуре посредством модулей посредников осуществляется на основе унификации описания объектов модульного типа и операций над этими объектами, определенных на множестве функций преобразования простых и структурных типов данных.

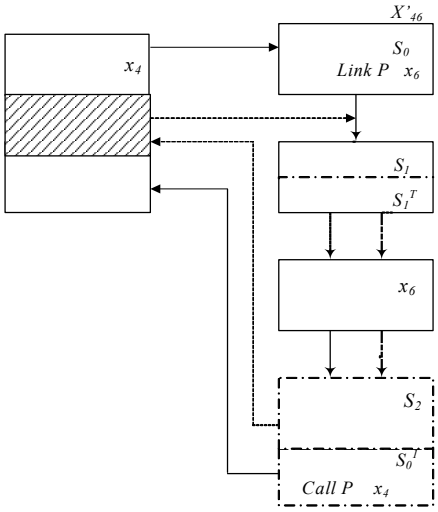


Рис. 4.10. Структура агрегата

Унификация объектов определяется посредством задания описаний их паспортов.

Паспорт модуля – это специальный раздел в описании модуля, содержащий следующие виды информации:

имя паспорта, которое совпадает с именем модуля;

язык программирования данного модуля;

список формальных параметров модуля;

разделение формальных параметров на входные и выходные;

список вызываемых модулей;

список фактических параметров для каждого вызываемого модуля;

описание типов данных для всех параметров, указанных в паспорте;

описание файлов печати, используемых в модуле.

Каждый вид информации образует подраздел, не зависящий от ЯП, кроме подраздела описания типов данных, на котором описан модуль. Паспорт описывается

на специальном *языке комплексирования* L', содержащем:

подмножества проекций языков из L, соответствующие описанию типов данных параметров, указанных в списках фактических и формальных параметров;

оператор описания графа и связи модулей в модульную структуру.

Язык L' позволяет описать паспорта модулей в классе языков L и графовую модель модульной структуры агрегата. При включении в класс L языков Модула-2, Си, Ада и др. в язык L', должны включаться средства конструирования новых типов данных, в том числе абстрактных типов данных, приведенных в [3, 4, 155, 201].

Модульная структура описывается в явном и неявном виде средствами языка L'. Неявное описание основано на том, что в паспорте модуля указываются имена вызываемых модулей. Это позволяет по паспорту главного (корневого) модуля построить для модульной структуры программный агрегат. Задачи построения агрегатов модульных структур основаны на этом методе. Его использование позволяет отождествить паспорт программного агрегата с паспортом его главного модуля.

Явное описание модульной структуры основано на представлении графа в виде матрицы вызовов. Особенности задания матриц вызовов приведены выше.

В качестве агрегатов выступают: сегмент – SEG; программа – PROG; комплекс – COMP; пакет – PAC. Их паспорта формируются в процессе комплексирования и состоят из совокупности паспортов модулей, входящих в состав графа программного агрегата.

В языке конструирования

имеются операторы, позволяющие описать указанные типы программных агрегатов с различными режимами их выполнения. Общая форма записи оператора связи имеет вид:

LINK <тип агрегата> <имя агрегата> (<имя главного модуля>, <дополнительный список имен модулей>) <режим выполнения>.

Тип программного агрегата принимает следующие значения: SEG, PROG, COMP, PAC.

Имя агрегата соответствует имени, под которым программный агрегат после комплексирования заносится в загрузочную библиотеку. Если создается сегмент, в дальнейшем включающийся в более сложный агрегат, то его имя должно совпадать с именем главного модуля. Для программы и комплекса это – уникальное имя, которое присваивается генерируемому корневому модулю.

Имя главного модуля модульной структуры указывается при комплексировании сегмента или программы. При построении комплекса этому имени соответствует имя первой выполняемой программы. Дополнительный список имен модулей указывается для построения в сегментах и программах с динамической структурой, а также для записи последовательности выполняемых программ комплекса.

Оператор позволяет записывать вид программных структур, отличных от простой. При этом знак ρ, заданный в вершинах графа, ставится перед именем модуля, что определяет тип вызова для данного модуля.

Режим выполнения определяется символом, принимающим следующие значения:

– построение графа модульной структуры и вывод на экран терминала;

0 – включение в модули связи средств отладки модульной структуры.

Если символ режима выполнения отсутствует, то предполагается обычный режим.

Средства управления процессом комплексирования. Данные средства

обеспечивают выполнение нескольких процессов построения модульных структур программ.

1. Ввод задания в языке L' на комплексирование программного агрегата и на выполнение синтаксического контроля операторов, содержащихся в задании.

2. Построение модульной структуры согласно используемым алгоритмам и операциям, описанным выше. На данном процессе проверяется наличие всех необходимых модулей, определяется их местонахождение (личная библиотека, банк модулей общего пользования и т. д.) и формируется матрица вызовов графа модульной структуры.

3. Генерация модулей связи необходима при:

сборки пар разноязыковых модулей;

построении динамической, сложной или подзадачной структуры;

формировании отладочной среды программного агрегата;

генерации корневых модулей для программ и комплексов.

4. Трансляция модулей связи и модулей в языке L', входящих в состав программного агрегата. На данном процессе выполняется:

трансляция сгенерированных модулей связи;

трансляция исходных модулей в ЯП для данного агрегата, если ранее эта операция к ним не применялась;

подготовка всех модулей для редактирования связей в модульной структуре агрегата загрузочного вида.

Данный процесс не имеет непосредственного отношения к комплексированию, но для автоматизации этого процесса он крайне необходим.

5. Организация построения различных программных структур путем генерации управляющих предложений для различных структур программ и выполнения редактирования связей для отдельных сегментов. Обычно в качестве редактора связей используется соответствующая программа операционной системы.

Практическая реализация методов управления модульными структурами нашла воплощение в средствах комплексирования системы АПРОП, которые предназначены для:

представления модульных структур, в том числе матричной формы;

реализации алгебраической системы с операциями над множеством модульных структур;

построения различных типов программных агрегатов и программных структур;

отладки модульных структур.

4.6.3. ОТЛАДКА И ТЕСТИРОВАНИЕ ПРОГРАММ ИЗ МОДУЛЕЙ

Применение данных методов предполагает, что отдельные модули, входящие в модульную структуру, прошли процесс автономной отладки и тестирования. На средства автономной отладки и тестирования никаких ограничений не накладывается.

Отладка модульной структуры

Задачи отладки модульных структур состоят в проверке правильности построения модульной структуры и выполнения программного агрегата, соответствующего данной модульной структуре. Рассмотрим эти задачи более подробно.

Проверка правильности построения модульной структуры. Существует два способа проверки. Первый основан на анализе результатов процесса редактирования

связей (сборки модулей), выполняемого специальной программой операционной системы. Данный способ позволяет выявить грубые ошибки – отсутствие модулей в модульной структуре, к которым есть обращения. Эти ошибки – следствие реального отсутствия модулей или неверного имени в операторе вызова LINK.

Второй способ основан на анализе самой модульной структуры, который заключается в следующем:

1. Визуальный анализ графа модульной структуры. Проверка правильности построения непосредственно осуществляется самим разработчиком ПС. Отображение графа модульной структуры на экране терминала или вывод его на печать.

2. Анализ матриц, описывающих модульные структуры и основанных на результатах п.4.4. Для проверки правильности построения модульных структур используются матрицы вызовов и достижимости. Результат проведенного анализа – установка существования циклов, числа маршрутов и достижимости между каждой парой модулей (данная информация используется для определения количества маршрутов при тестировании).

Перейдем к рассмотрению второй основной задачи отладки модульных структур.

Проверка правильности выполнения модульной структуры. Решение данной задачи тесно связано с реализацией проблем межъязыкового интерфейса. Проверка правильности выполнения модульной структуры предполагает отслеживание в динамике последовательности передач управления и данных между взаимодействующими модулями. Такая возможность позволяет фиксировать цепочки выполняемых модулей и выполнять трассировку передаваемых данных. Результаты этих операций отображаются на экран или печать. На основе их анализа определяются:

правильность последовательности вызовов модулей в соответствии с графом модульной структуры и в зависимости от входных данных;

правильность передаваемых данных между взаимодействующими модулями согласно описанию их типов в списке формальных и фактических параметров;

последний выполняемый модуль в момент аварийной ситуации при выполнении программного агрегата;

некоторые виды зацикливания в модулях при выполнении программных агрегатов.

Реализация этих функций возможна только при применении шестого метода комплексирования (см. гл.3) – использовании промежуточных модулей связи, в которые включаются средства для отладки модульных структур агрегатов. Они предназначены для фиксации момента входа в вызываемый модуль и выхода из вызываемого модуля; отображения и хранения значений входных параметров согласно описаниям их типов данных при передаче управления вызываемому модулю; отображения и хранения значений выходных (и, возможно, входных) параметров согласно описаниям их типов данных при возврате управления вызывающему модулю.

Средства отладки могут включать и дополнительные возможности: изменение значений передаваемых параметров при выполнении программного агрегата с использованием режима диалога; отображение файлов печати (значений параметров, цепочек вызовов модулей и т.п.) на экран после работы каждого модуля или агрегата в целом.

Тестирование модулей и структур агрегатов

Тестирование заключается в динамической проверке поведения программы на конечном множестве тестовых данных, выбранных из входного пространства, на соответствие ожидаемому поведению.

Динамическое тестирование всегда предполагает *выполнение* программы, а *конечное* – создание большого количества тестов, которые могут быть выполнены в ограниченные сроки наблюдения за поведением программы.

Методы тестирования, в основном, отличаются подходами к выбору множества тестовых данных из входного пространства. Основная цель тестирования – обнаружение дефектов в ПС и установление ее функциональной пригодности, удобства применения, производительности и др.

Тестирование на протяжении процесса разработки агрегата из модулей выполняется на нескольких уровнях. Для каждого уровня тестирования определяется категория объектов тестирования (весь агрегат ПС, программные компоненты, отдельные модули) и набор проверяемых тестируемых характеристик.

На каждом уровне тестирование повторяется многократно, образуя циклы: тестирование–исправление–повторное тестирование.

В современной практике тестирования все виды действий, начиная с планирования до оценки результатов тестирования, должны интегрироваться в четко определенный, документируемый и контролируемый процесс тестирования. Это облегчает взаимодействие между разработчиками, группой тестирования и руководством проекта, а также позволяет сделать процесс видимым, повторяемым и измеряемым.

Связь тестирования с другими видами проверки. Тестирование имеет много общего с процессами верификации и валидации (V&V). Общность процесса тестирования с процессами V&V заключается в единстве состава и структуры планов, рекомендуемых стандартом IEEE Std. 829, а также объектов и применяемых методов. Отличие этих процессов состоит в условиях их применения. Тестирование – основной процесс в ЖЦ, выполняемый *всегда,* для всех объектов ПО системы независимо от ее критичности. Процессы V&V, в современной трактовке стандартов IEEE Std. 1012 и ISO/IEC 12207 – поддерживающие процессы, которые могут применяться к выбранным объектам тестирования для проверки планов тестирования и подтверждения того, что выполненное тестирование адекватно уровню критичности ПС. По отношению к процессу тестирования V&V выполняет контрольную функцию и подтверждает соответствие объектов установленным требованиям.

Тестирование ПС тесно связано с отладкой и собственно программированием, но охватывает гораздо более широкий круг проблем и участников – программистов, тестировщиков, аналитиков и инженеров по качеству.

Виды и уровни тестирования. Традиционно выделяется три уровня тестирования ПО: автономное или модульное (unit testing), интеграционное (integrating testing) и системное (system testing). В стандарте ISO/IEC 12207 прослеживаются *четыре* уровня тестирования:

– модульное (в процессе «Построение ПО»),

– интеграционное (в процессе «Сборки ПО»),

– тестирование ПС (как процесс):

– системное (в процессе «Испытания ПС»).

Модульное тестирование предполагает проверку функционирования модулей в изоляции друг от друга. Оно обычно выполняется разработчиками с доступом к коду и чередуется с отладкой. Объектами тестирования являются отдельные процедуры, программные модули и компоненты, состоящие из тесно связанных модулей.

Цели модульного тестирования – обнаружение дефектов в реализации функций объектов и подтверждение соответствия объекта спецификациям проекта (техническому проекту).

Наиболее полно вопросы систематического модульного тестирования освещены в стандарте IEEE 1008-87 “Standard for Software Unit Testing”.

Интеграционное тестирование предназначено для проверки правильности взаимодействия между программными объектами, протестированными автономно.

Современные систематические стратегии интеграции определяются архитектурой ПС и моделью разработки (обычно итерационной с приращениями).

Интеграционное тестирование выполняется при каждой сборке новой версии ПС с целью выявления дефектов в интерфейсах между интегрируемыми компонентами и подтверждения их соответствия проекту архитектуры ПС.

Тестирование ПС заключается в проверке функционирования интегрированной версии системы в моделируемой среде. Цели тестирования на этом уровне – выявление дефектов в реализации внешних функций ПС и подтверждение его соответствия спецификации функциональных требований.

Системное тестирование ориентировано на оценку качества сопряжения модулей ПС с другими не программными компонентами современных систем.

На этом уровне тестирования производится проверка соответствия ПС целям качества, установленным в требованиях к системе , таким как надежность, устойчивость, производительность и др., а также *внешним интерфейсам* с другими системами, средой, аппаратным обеспечением. Большая часть функциональных отказов и дефектов должна быть идентифицирована и устранена на предыдущих уровнях тестирования. Здесь могут быть выявлены дефекты, связанные с неудовлетворительными техническими характеристиками функционирования всей системы.

Задачи системного тестирования часто объединяют в единый процесс, При этом тестирование технических характеристик ПС должно выполняться отдельно.

Вывод. Рассмотрен общий подход к управлению модульными структурами. Базовым формальным аппаратом для описания операций над модульными структурами является теория матричных преобразований для их представления в виде графа. Матричное представление используется для задания матрицы вызовов, задающей обращение одного модуля к другому. Проведено доказательство отношения достижимости на графе модульной структуры. Рассмотрены операции построения модульных структур и методы тестирования сложных программных агрегатов.

Глава 5

КЛАССИФИКАЦИЯ И ТИПИЗАЦИЯ ПРОГРАММНЫХ РЕСУРСОВ

Большое значение для проблематики современного сборочного программирования имеют компоненты повторного использования (КПИ) или reuses. Они образуют готовые *программные ресурсы*, включающие в себя модули, компоненты, объекты данных, информационные ресурсы Интернета и т.п.

Главная задача сборки разных программных ресурсов в конкретные программные системы (ПС) – обеспечение взаимоотношений или взаимодействий новых и готовых программных ресурсов широкого применения, которые создаются для разных платформ и в разные периоды развития компьютерных и информационных технологий.

Эффективное применение готовых программных ресурсов основывается на проведении их типизации и классификации. Особенно это касается нового вида ресурсов, так называемых менеджеров информационных ресурсов (ИР), используемых при построении прикладных систем и веб-приложений в среде Интернета.

Компоненты разработки, КПИ, программы и информационные ресурсы – объекты типизации и классификации в программной инженерии.

5.1. ОБЩИЕ МЕТОДЫ ТИПИЗАЦИИ И КЛАССИФИКАЦИИ ПРОГРАММНЫХ КОМПОНЕНТОВ

Сущность любой классификации состоит в разбивке множества элементов на классы эквивалентности согласно классификационным характеристикам и признакам. Классы эквивалентности не должны пересекаться, и все элементы конкретного класса должны иметь соответствующие признаки.

Для одного и того же множества может существовать несколько систем классификации, каждая из которых зависит от выбранных базовых признаков и способов группировки их в классификационные характеристики (например, программная инженерия как наука качественного производства). Основой такой системы является информация, которая сопровождает компоненты, и правила их упорядочивания соответственно определенным признакам – функциональность, системная среда и т.п.

Системы классификации компонентов не существует, поэтому ее построение основано на общих знаниях о компонентах и подходах к их применению. Эти знания распределяются среди таких групп общих свойств, как структурные признаки, функциональные возможности, характеристики поведения и т.п.

Классификация компонентов – это унификация представления информации о компонентах для дальнейшего поиска и отбора их из среды хранения, например репозитария. Она выполняется с учетом таких признаков: принадлежность к ПрО; тип компонента (модуль, класс, сервис и др.); интерфейс; готовность КПИ; тип структуры КПИ (каркас, паттерн, контейнер) и т.п.

Физическое размещение КПИ в репозитарии выполняется с помощью каталога, упорядочивающего их по именам и ссылкам на места их размещения. Компоненты, которые выбирают в репозитарии, требуют настройки на условия среды функционирования, в которой они будут использоваться, как элементы новой ПС.

5.1.1. ПОДХОД К ТИПИЗАЦИИ КОМПОНЕНТОВ

Классификация компонентов обусловлена типизацией, и, хотя классического определения понятия типизации компонентов пока не существует, оно связано с отдельными экземплярами компонентов, для которых могут применяться операции по определению их интерфейсов.

Согласно теории типов любой тип рассматривается, как множество элементов, для каждого из которых одна или совокупность предикатных функций принимают значение истины.

Если точно $d \in D$ (D – домен) и $P = \{P_1, P_2, \ldots, P_n\}$ – совокупность предикатных функций, то тип T определяется выражением

$$T = \{d: D \mid P\}. \qquad (5.1)$$

Исходя из этого выражения, сформулируем следующее определение.

Тип – это определенная характеристика свойства, в том числе структуры и поведения, которые относятся к определенной совокупности объектов. Если понятие типа рассматривают в контексте типа данных, то объекты – значения из определенного множества, а свойства – совокупность операций над ними.

Формально каждый тип данных задается абстрактной системой, которая состоит из множества значений (домена данных для этого типа) и совокупности операций над этим множеством (сигнатуры типа) [207]. Иными словами, выражение (5.1) принимает вид

$$T = (X, \Omega), \qquad (5.2)$$

где X – множество элементов типа данных; Ω – совокупность операций над элементами множества.

Как было сказано ранее (см. глава 2), типы данных делят на базовые или примитивные (например, integer, real) и сложные или производные (например, array, struct), которые состоят из упорядоченной совокупности данных с меньшим уровнем структурированности. Для обеспечения целостности обработки типизированных переменных в состав операций сложных типов должны входить операции непосредственно как над самой переменной, так и над отдельными составляющими. Это приводит к построению иерархической системы типов данных в рамках единой теории типов.

Каждый язык программирования (ЯП) имеет собственную совокупность типов данных и механизмы конструирования новых типов. Главное назначение типизации – представление абстракций таким образом, чтобы ЯП, который применяется для реализации программного проекта, поддерживал согласованность

принятых проектных решений, определяющихся соответствующей системой типов данных.

Обобщение понятия типа данных – класс в терминологии ООП [35, 36, 158].

Класс определяет множество объектов с общей структурой, состоящей из экземпляров класса, и поведения, которое определяется методами формирования совокупности операций над этими экземплярами (создание, уничтожение, сериализация и др.), а также над переменными класса и экземплярами класса.

Переменные могут иметь типы, простые или сложные, которые соответствуют другим классам и обеспечивают построение иерархической системы классов. Совокупность внешних переменных и методов класса определяют интерфейс, с помощью которого экземпляры других классов взаимодействуют с экземплярами данного класса.

Если сравнить наведенные определения типа и класса, то их сущности принципиально не различаются на абстрактном уровне. Поэтому эти понятия в современных ЯП часто рассматривают, как синонимы.

Любое взаимодействие с экземплярами компонентов в ООП происходит через интерфейсы, которые состоят из методов и атрибутов. Атрибут – логическое представление схемы взаимодействия экземпляра компонента с внешними переменными с помощью специальных методов, которые не относятся к операциям обработки данных, а обеспечивают доступ к этим переменным. Для каждой внешней переменной существует пара методов – выборка значения переменной (get-метод) и присвоение ей нового значения (set-метод).

С обобщающей точки зрения, интерфейс экземпляра компонента состоит из совокупности методов. Каждый компонент может иметь несколько таких интерфейсов, которые определяют его функциональные свойства. Кроме того, компонент может иметь специальный интерфейс, методы которого работают с его экземплярами (например, компонентная модель EJB языка JAVA имеет интерфейс, который называется “домашним” интерфейсом (Home-interface) [91].

В состав этих методов, в частности, входят методы поиска экземпляров компонента, их создание, уничтожение, и т.п. Для каждого компонента могут существовать несколько методов поиска, создания и т.д. в зависимости от алгоритма выполнения этой операции и соответствующей сигнатуры. Например, поиск экземпляра компонента может происходить по его уникальному идентификатору или значению определенной переменной.

Таким образом, каждый компонент характеризуется

– специальным интерфейсом, методы которого работают с экземплярами компонента, но не определяют непосредственно функциональные свойства экземпляров;

– одним или несколькими функциональными интерфейсами, которые реализуются в экземплярах компонентов и определяют их функциональные свойства.

Сравнивая этот факт с определением типа или класса и учитывая точку зрения классической теории делаем вывод, что понятие типизации компонентов не является определенным и принципиально отличается от аналогичного понятия для типов данных или классов. Сущность этого понятия лучше исследовать на примере операции приведения типов.

Для переменной определенного типа данных операцию приведения к другому

типу выполняют при условии, что существуют формальные алгоритмы преобразования множества значений типа и сохраняется семантика соответствующих операций.

Если существуют два типа: $T^1 = (X^1, \Omega^{1)}$ и $T^2 = (X^2, \Omega^2)$, то T^1 тип приводится к T^2 при следующем условии:

$T^1 \rightarrow T^2$, если $F(X^1)^2$ – функция преобразования множеств Ω^1, Ω^2.

Для экземпляра определенного класса операция приведения к экземпляру другого класса выполняется успешно, если его совокупность методов и полей удовлетворяют условиям описания другого класса. Как было указано выше, на абстрактном уровне равные понятия тип и класс используются как синонимы, и операция приведения для классов имеет семантику, похожую для приведения типов.

На абстрактном уровне операция приведения экземпляра класса в соответствии с заданным интерфейсом может рассматриваться как частный вид абстрактного класса. Поэтому операция семантически может быть сведена к операции приведения классов. Если выполняется операция приведения экземпляра класса в соответствии с определенным интерфейсом, то обязательное условие – реализация этим экземпляром всех методов соответствующего интерфейса. Иными словами, операции приведения для типов и классов – семантически эквивалентны.

Рассмотрим операцию приведения для компонентов. Приведем следующие выводы относительно программного компонента [64, 65]:

– компонент может иметь несколько интерфейсов;

– реализации одного и того же компонента могут иметь разную структуру и поведение в разных средах;

– статический аспект проверки типов для компонентов принципиально невозможен.

Для экземпляров компонентов операция приведения возможна лишь в соответствии с определенным интерфейсом компонента. С учетом того, что компонент может иметь несколько интерфейсов, тип экземпляра компонента – динамическая характеристика, которая определяется во время выполнения программы. Подобная аналогия имеется и у экземпляров класса, который реализуется несколькими интерфейсами. Вместе с тем, этот экземпляр всегда статически определен описанием своего класса, поэтому всегда имеет конкретный тип. Согласно определению понятия интерфейса операция приведения всегда выполняется, если класс имеет реализации всех методов интерфейсов. Это условие проверяется во время компиляции класса. Для ООП всегда выполняются следующие утверждения:

– любой экземпляр обязательно является экземпляром определенного класса и обладает всеми методами, заданными в этом классе;

– экземпляр класса может быть приведен в соответствие с одним из суперклассов или с интерфейсом, который реализуется в этом классе.

Аналогичных утверждений для экземпляров компонентов не существует. Понятие, которое было бы аналогично понятию класса для экземпляров компонентов, определить не удается (т.е. все экземпляры класса имеют типичную структура и поведение, а экземпляры компонента в общем случае не отвечают этому условию). Хотя сами интерфейсы компонента могут иметь родовидовые отношения с интерфейсами других компонентов, тем не менее, связи “класс–

суперкласс" и "наследование для интерфейсов компонентов" принципиально различаются как сущностью, так и методами реализации.

Аналогично имеется существенное различие на уровне классов компонентов. С точки зрения типизации, класс – это полностью определенное статические понятие. Наоборот, выражение "тип компонента" – это неопределенное понятие в статическом аспекте. Во время функционирования компонентной программы для компонента может существовать несколько экземпляров разной структуры и поведения. Поэтому операция приведения компонента с одним из его интерфейсов в динамическом аспекте также не определена и выполняется только для его отдельных экземпляров.

5.1.2. КЛАССИФИКАЦИОННЫЕ ПРИЗНАКИ КОМПОНЕНТОВ

Согласно основам теории сущность любой классификации состоит в разбиении множества из нескольких классов эквивалентности соответственно совокупности свойств, которые называются *классификационными признаками*. Классы эквивалентности не пересекаются, все элементы конкретного класса имеют соответствующие свойства.

Пусть C – множество элементов и $t^i_1, t^i_2, .., t^i_r$ – совокупность таких признаков, что для каждого $c \in C$ существует только один признак, присущий данному элементу.

Введем совокупность $T^i = \{t^i_1, t^i_2, \ldots t^i_r\}$. Тогда на множестве C существует эквивалентность относительно T^i. Элементам каждого из классов эквивалентности C^i_j присущ признак t^i_j, объединение C^i_j составляют множество C, т.е. $C = C^i_1 \in C^i_2 \in \ldots \in C^i_r \ldots$

Назовем T^i классификационной характеристикой, а каждое t^i_j – классификационным признаком характеристики T^i. Пусть $T = \{T^1, T^2, \ldots, T^n\}$ и каждое T^i – классификационная характеристика. Тогда T определяет классификационную систему для множества C.

Введем следующие определения.

Множество С – классификационное множество, если для него существует классификационная система T. Метод классификации для множества С – это способ построения классификационной системы T при условии выполнения требований к отношению эквивалентности на этом множестве.

Классификационная характеристика T^i является полной для множества С, если для каждого элемента $c \in C$ обязательно существует классификационный признак из T^i.

Классификационная характеристика T^i является частичной для множества С, если существует $C_1 \in C$ и для каждого элемента $c \in C_1$ не существует классификационного признака из T^i. Такой случай существует, если классификационные характеристики зависимы друг от друга. Например, характеристика T^1 является полной для множества C, и определяет классы эквивалентности $C^1_1\ C^1_2, \ldots, C^1_r$, а характеристика T^2 – лишь для одного или нескольких C^1_j.

Классификационные характеристики T^1 и T^2 иерархически упорядочены, если: T^1 – полная характеристика для определенного подмножества $C_1 \in C$ (или самой C); T^2 – частичная характеристика для C_1; T^2 – полная классификационная

характеристика для одного из классов эквивалентности $C^l_j \in C_l$.

Приведем несколько следствий.

1. Для одного и того же множества может существовать несколько систем классификации.

2. Система классификации – это понятие относительное, т.е. зависит от первичных, базовых признаков и способов группирования их в классификационные характеристики.

3. Каждый метод классификации определяется:

– способом группирования первичных признаков в классификационные характеристики;

– типом классификационных характеристик (полных или частичных);

– иерархическими связями между характеристиками.

Основные задачи классификации. Любая классификация рассматривается при решении следующих задач.

Во-первых, распределить элементы множества среди подмножеств согласно определенным знаниям о самом множестве и об отдельных элементах, а также их взаимосвязях, семантических признаках, и т.п. Решение этой задачи дает возможность по принадлежности конкретного элемента множества к определенному классу эквивалентности определить априори совокупность свойств данного элемента. Если определенный элемент $c \in C^i_j$, то c имеет признак t^i_j из классификационной характеристики T^i.

Во-вторых, обеспечить возможность быстрого поиска элементов множества согласно условиям, которые связаны с классификационными признаками. Результат поиска – класс эквивалентности, который отвечает заданным свойствам. Поиск происходит соответственно заданным классификационным характеристикам $T^l,\ldots, T^s$ и конкретным признакам этих классификационных характеристик.

В целом при классификации компонентов решаются аналогичные задачи. Особенность состоит в выборе системы классификационных признаков, характеристик и способов применения знаний о классификации компонентов.

Метод классификации компонентов – это метод классификации, который обеспечивает определенную систему представления информации и знаний о программных компонентах в целях быстрого поиска и отбора необходимых объектов. Основой такой системы является информация, которая сопровождает компоненты, и правила упорядочения соответственно определенным признакам: функциональности, системной среде, ресурсам и др. Результат применения метода классификации ПК – классы эквивалентности, которые определяют совокупности компонентов с аналогичными свойствами соответственно классификационным признакам.

Единой системы классификации компонентов на данный момент не существует. Ее построение основывается на общих знаниях о компонентах и их применении. Эти знания образуют следующие группы общих свойств компонентов :

– структурные признаки;

– функциональные возможности;

– характеристики поведения.

Детализация этих групп и объединение свойств из разных групп в единую совокупность компонентов обеспечивает создание целевой системы классификации.

5.1.3. КЛАССИФИКАЦИЯ КОМПОНЕНТНЫХ МОДЕЛЕЙ СОВРЕМЕННЫХ СИСТЕМ

Базис этой классификации – компонентная модель, на ее основе создаются компоненты и классификационная система с соответствующими классификационными характеристиками.

Рассмотрим наиболее распространенные в настоящее время практически применяемые виды компонентных моделей современных систем [40, 166, 176, 209]:

– модель COM в среде ОС Windows, основана на понятии сервера;

– модель DCOM, распределенная COM в сетевой среде Windows;

– компонентная модель .NET в среде .NET, которая является надстройкой к другим ОС, в частности Windows, Linux;

– много платформенная модель CORBA функционирует в различных средах ОС;

– компонентная модель EJB платформы JAVA, функционирует на любой ОС, для которой реализована виртуальная JAVA-машина.

С формальной точки зрения классификационная характеристика для множества компонентов (Comp) имеет такой вид:

$$T^{I} = \{\text{COM, DCOM, .NET, CORBA, EJB}\}. \quad (5.3)$$

Классы эквивалентности – это совокупности компонентов, которые созданы в рамках конкретной компонентной модели. Метод классификации – тривиальный (группировка всех признаков в единую характеристику).

Если есть компонент, который согласно приведенной классификации построен для конкретной модели, то с ним связана определенная совокупность знаний, а именно:

– ОС, на которых компонент может функционировать;

– особенности построения компонента;

– методы развертывания компонентов;

– особенности взаимодействия с другими компонентами и др.

Данная классификация наиболее распространена и имеет высокий уровень обобщения. Каждый из соответствующих классов эквивалентности может быть разделен на более детализированные совокупности. Например, на основе более глубокого анализа модели COM и ее свойств [209] существует следующая более детальная классификация.

Классификация COM-компонентов сервера автоматизации. Учитывая общую классификацию компонентов, для сервера выделяем классы:

– полные серверы (самостоятельные, завершенные приложения с возможностями встраивания компонента в другие приложения, т.е. они могут использоваться, как готовые при построении компонентных систем более высокого уровня структурирования);

– обычные серверы автоматизации (а также завершенные приложения, не имеющие возможности встраивания в другие приложения);

– мини-серверы (дополнительные компоненты, которые не имеют признаков самостоятельного функционирования в компонентной среде, могут применяться лишь в операционном контексте клиентского компонента);

– компоненты Active X (разновидность мини-серверов, которые подаются как самостоятельные библиотеки, с возможностью передаться в сеть, например, для функционирования в среде Интернет–браузера);

– серверы процессов (как фоновые задачи в ОС, постоянно функционируют и в общем случае могут не иметь связей с конечным пользователем).

Такая классификация – это ее расширение с учетом предыдущего примера, т.е. для классификационной характеристики T^1 уже определены классы эквивалентности и для одного из классов, который имеет признак *COM*, вводится новая классификационная характеристика T^2. Она является частичной характеристикой для всего множества компонентов Comp и полной для множества COM-компонентов. Классификационные характеристики T^1 и T^2 иерархически упорядочены.

Рассмотрим классификационные признаки для T^2. Анализ приведенной выше последовательности классов классификации показывает, что существует несколько характеристик, на основе которых построены соответствующие классы, а именно,

– уровень завершенности приложения;

– возможности встраивания в компоненты более высокого уровня структурированности;

– типы и виды среды функционирования;

– наличие связей с интерфейсом конечного пользователя.

Например, можно сформировать классификационную систему на основе всех четырех классификационных характеристиках. Однако для такой системы множество классов эквивалентности не имеют смысла или практически бесполезны. Таким образом, процесс есть самостоятельной фоновой задачей в ОС и поэтому нецелесообразно его рассматривать с точки зрения возможности встраивания в компоненты более высокого уровня структурированности. Учитывая, что практически полезными является лишь пять классов компонентов, одним из решений может быть непосредственное формирование единой характеристики, которая имеет пять признаков (каждый для соответствующего класса). При таком подходе существует вероятность потери информации, и, тем не менее, количество классов эквивалентности для классификационной системы значительно уменьшится. В этом случае имеем

T^2 = {полный сервер, обычный сервер автоматизации
мини-сервер, компонент Active X, сервер процесса}.

Выбор конкретного решения (расширенная или упрощенная система классификации) зависит от целевого назначения самой классификации как инструмента для практического применения.

Как и в предыдущем примере, если отнести конкретный COM компонент к определенному классу классификации, то можно получить информацию о его структуре, представлении, методах применения и др.

Классификация компонентов корпоративных приложений. Каждый из этих типов (веб-приложение, корпоративное приложение и др.) имеют типовую архитектуру. Реализация компонентной программы согласно выбранной архитектуре предусматривает распределение компонентов по уровням архитектуры с учетом необходимых архитектурных свойств.

Основа построения классификации по заданным признакам – архитектурная модель, которая состоит из четырех уровней, каждый из которых делится на

несколько логических архитектурных слоёв. Перечень уровней, слоев, и месте их расположения приведены в табл. 5.1.

Таблица 5.1

Уровень архитектуры	Слой уровня архитектуры	Место размещения
Представление	Отображение (клиентская часть)	Клиент
	Внешние службы	Веб-сервер
	Контролер приложения (серверная часть)	
Логика применения	Прикладные службы	Сервер приложений
	Бизнес-логика, модель домена	
Источник данных	Преобразование, отображение моделей данных	
	Менеджеры ИР, коннекторы	
Информационные ресурсы	Логика управления ресурсом	Сервер баз данных, файловая система
	Данные, информация	

Уровень представления. Цель – это подготовка, формирование и отображение результатов работы приложения для конечного пользователя, а также обеспечение интерфейса с этим приложением для взаимодействия с ним. В частности, на этом уровне находится графический интерфейс пользователя (GUI).

Уровень логики приложения. На нем находится основная функциональность приложения, бизнес процессы и алгоритмы обработки данных.

Уровень источников данных. Уровень предназначен для определения связи с информационными ресурсами (ИР) приложения. Каждый источник рассматривается как компонент с унифицированным интерфейсом, который инкапсулирует внутренние детали реализации и хранения данных.

Уровень информационных ресурсов. Здесь находятся такие ресурсы: базы данных, файловые структуры, отдельные документы и др.

При построении приложения на разных архитектурных уровнях могут быть использованы определенные ЯП, компонентные модели, методы, технологии и др., которые подходят для реализации соответствующих программных средств для создания корпоративных систем, приведенных в табл. 5.2.

Рассмотрим несколько базовых классификационных характеристик. Характеристика T^1 будет соответствовать уровню архитектуры, т.е. множество компонентов для построения корпоративных систем разделяем на количество уровней архитектуры:

T^1={Представление, логика приложения, источники данных, ИР).

Характеристика T^2 соответствует месту нахождения компонента согласно архитектурному уровню (из табл.5.1. видно, что компоненты уровня представления размещаются на клиенте и веб-сервере).

Данная характеристика имеет вид

T^2={Клиент, Веб-сервер, Сервер приложений, Сервер данных}. (5.4)

Приведенные характеристики – примеры формирования классификационных характеристик для построения значительно большей системы классификации.

В частности, в таблицу необходимо включить такие классификационные характеристики:

Таблица 5.2

Слой уровней архитектуры	Функциональность слоя архитектуры	Средство реализации
Отображение (клиентская часть)	Реализация отображения	Средства и языки разметки (HTML, XML)
	Логика формирования отображения	Скриптовые компоненты, аплеты (Java, JavaScript, VB Script)
	Управление функциями отображения	Интернет-браузер
Внешние службы	Веб-службы	XML, SOAP, WSDL, UDDI, др.
Контролер приложения (серверная часть)	Контролер, адаптер генерации представления	Сервлеты, JSP, ASP, PHP-script, Perl–script
Прикладные службы	Инкапсуляция бизнес логики, прикладные интерфейсы	Интерфейсы POJO, EJB, CORBA, .NET, COM, DCOM
Бизнес-логика, модель домена	Функциональные алгоритмы системы	Объектные и компонентные модели на основе POJO, EJB, CORBA, .NET, COM, DCOM
Преобразование, отображение моделей данных	Преобразование и отображение для обеспечения совместимости поведения	ORM-средства, Table Module, DataSet, DOM, др.
	Преобразование и отображение для обеспечения совместимости структуры	ORM-Средства, Table Module, DataSet, DOM, др.
Менеджеры информационных ресурсов, коннекторы	Унифицированные интерфейсы для информационных ресурсов	Интерфейсы ODBC, JDBC, специализированные интерфейсы
	Механизмы соединения с информационными ресурсами	Драйверы ODBC, JDBC, специализированные коннекторы
Логика, управление ресурсом	Управление ресурсом	СУБД, информационные оболочки
	Первичная обработка, обеспечение целостности	Триггеры, процедуры, которые сохраняются
Данные, информация	Представление структурированных данных	Таблицы СКБД, другие структуры данных
	Представление неструктурированных данных	Произвольные файлы и данные

– тип функциональности, которая реализуется на соответствующем слое архитектуры (для каждого слоя уровня архитектуры представляет группы функциональности для компонентов приложения соответствующего слоя);

– тип компонентной модели (частичной, так как в корпоративной системе существуют средства, которые разрабатываются без ориентации на компонентную модель);

– системная среда, ОС;

– ЯП компонентов;

– типовые компоненты, которые реализуют общие (типовые) шаблоны проектирования (например, ORM-средства, Table Module, DataSet) и др.

Совокупность приведенных характеристик – основа формирования системы классификации компонентов для построения корпоративных приложений.

5.1.4. ОСНОВЫ КЛАССИФИКАЦИИ КОМПОНЕНТОВ ВЕБ-ПРИЛОЖЕНИЙ

Схема классификации компонентов для корпоративных приложений используется при построении основ классификации компонентов веб-приложений с учетом архитектурных требований. К классификации компонентов относятся такие типовые решения и элементы:

– структура компонента;

– структура базовой компонентной среды (framework);

– схема и механизмы взаимодействия компонентов на основе интерфейсов;

– механизмы развертывания и конфигурации компонентов в каркасе;

– системные сервисы, которые поддерживают функционирование компонентной среды и взаимодействие компонентов.

Эти решения и элементы являются обязательными составляющими любого компонентного подхода и соответствующих компонентных моделей. Каждая система классификации по компонентным признакам должна учитывать эти типовые решения, элементы и каждый элемент классификации должен иметь классификационные свойства, которые связаны с такими решениями.

Архитектурные признаки классификации компонентов для веб-приложений определяются на основе следующих архитектурных и структурных решений [200]:

– общая архитектура веб-приложения;

– архитектура клиента приложения;

– архитектура серверной компоненты.

Обобщенная типовая архитектура веб-приложения. Такая архитектура – клиент серверная архитектура, а именно Интернет-браузер. Основа серверной части – веб-сервер, в среде которого реализуются компоненты обработки запросов и генерации ответов. Такие серверные компоненты реализуют дополнительную функциональность по взаимодействию с другими серверами, доступ к базам данных, обращение к разным классам приложений и др. Иными словами, веб-сервер – это компонент веб-приложения, который инкапсулирует у клиента все операции, выполняемые для обработки запросов. Определим следующие классификационные характеристики.

Характеристика T^1 соответствует местоположению компонента в клиент серверной архитектуре, т.е. T^1 = {Клиент, Сервер}.

Учитывая, что основу клиента составляет Интернет-браузер, определим характеристику, которая связана с типом браузера:

T^2 = {MS Internet Explorer, Mozilla, Opera, Netscape, Konqueror}.

T^2 – частичная характеристика, иерархически упорядочена относительно характеристики T^1.

Характеристика T^3 будет связана с типом веб-сервера:

T^3 = {Internet Information Server, Apache, Tomcat, JBoss, WebSphera, WebLogic, Cloudscape}.

Как и предыдущая характеристика T^3 – это частичная характеристика и иерархически упорядочена относительно характеристики T^1.

Типовая архитектура клиента веб-приложения. Эта архитектура построена на основе типовых моделей, объектов данных, операций, которые применяются в среде Интернет-браузера. Она состоит из нескольких логических уровней, каждый из которых определяется своими множествами структур данных и операций над ними. Этим уровням принадлежат:

– уровень распределенной среды;

– базовый уровень клиента;

– уровень формирования объектов данных клиента;

– функциональный уровень клиента;

– уровень отображения объектов данных.

Уровень распределенной среды. Этот уровень является логическим представлением сетевой среды, которая состоит из сети Интернета, корпоративных сетей, локальных вычислительных сетей и др.

Базовый уровень клиента. Типичная реализация этого уровня – Интернет-браузер, определяет среду клиента, обеспечивает формирование и отправку запросов к серверному компоненту, а также получение и базовую обработку ответа от сервера. Кроме этого, браузер предоставляет механизмы поддержки реализации других уровней архитектуры клиентов веб-приложений.

Уровень формирования объектов данных клиента. На этом уровне создаются и формируются объекты данных для реализации функциональности клиента. Объекты подаются как иерархически упорядоченное множество. Функции построения и упорядочения выполняет браузер. Каждый объект имеет совокупность атрибутов и свойств, а также методов, которые определяют функциональность этого объекта.

Функциональный уровень клиента. Этому уровню соответствуют два вида функций. К первому виду относятся те, которые выполняются во время формирования запроса к серверу приложения на основе данных, вводимых конечным пользователем. Функции второго вида обеспечивают обработку данных, которые получены в ответ со стороны сервера, и формирование визуального представления результатов обработки запроса.

Уровень отображения объектов данных. На этом уровне непосредственно происходит формирование визуального изображения, которое может видеть пользователь клиента веб-приложения. Оно строится согласно иерархической модели данных в соответствии с правилами отображения для каждого типа объектов данных.

Классификационная характеристика T^4 определяет функциональную группу, к которой принадлежат компонент:

T^4={формирование объектов данных, функциональные компоненты, отображение объектов данных}.

Примерами таких компонентов могут быть компоненты в одном из скриптовых ЯП, который воспринимает браузер:

– компоненты построения и работы с массивами, таблицами, списками (формирование объектов данных);

– функциональные компоненты для валидации, т.е. для проверки данных, которые вводит конечный пользователь;

– компоненты для работы с нестандартными окнами браузера для отображения объектов данных.

Характеристика T^5 определяет ЯП, в котором описан компонент:

$$T^5 = \{\text{JavaScript, VB Script, Java, HTML, XML}\}.$$

Характеристики T^4 и T^5 – частичные характеристики, иерархически упорядочены относительно характеристик T^2.

Дополнительные классификационные характеристики компонентов клиента связаны непосредственно с конкретной обработкой данных, и их выбор обусловлен субъективными факторами. Эти характеристики целесообразно вводить для некоторой предметной области, определенных методов обработки данных и др. Такие характеристики выходят за рамки архитектурных признаков.

Типовая архитектура сервера веб-приложения. Серверная часть значительно превосходит клиентскую по сложности и функциональности. Поэтому для нее существует больше классификационных характеристик. Типовая архитектура серверной части веб-приложения представлена на рис.5.1.

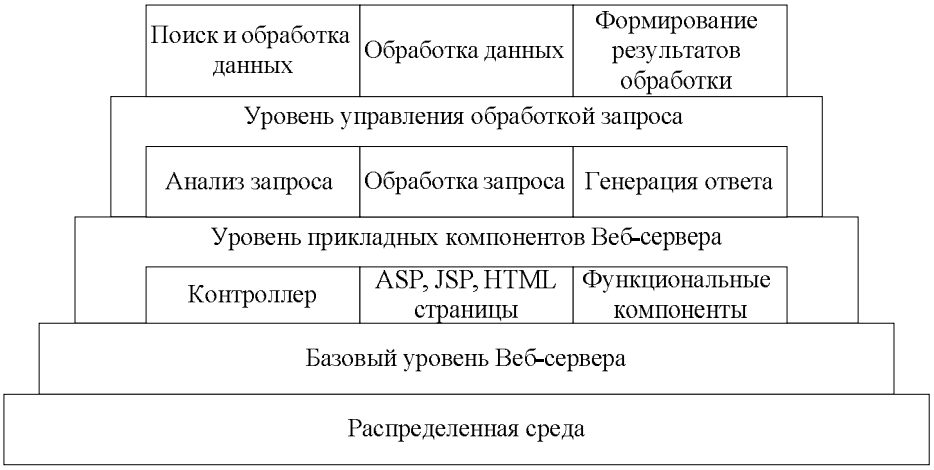


Рис. 5.1. Типовая архитектура серверной части веб-приложения

Уровень распределенной среды. Назначение и функции этого уровня полностью совпадают с аналогичным уровнем для клиентов.

Уровень прикладных компонентов веб-сервера. Этот уровень представляет собой совокупность компонентов, которые обеспечивают функциональность серверной части веб-приложений для обработки запросов со стороны клиентов и формирования соответствующих результатов. Современная архитектура обработки запросов, как правило, основана на шаблоне проектирования MVC (model-view-controller), согласно которому серверные компоненты выполняют функции контролера, представления и обработки данных, формирования и представления результатов клиентских запросов.

Уровень управления обработкой запроса. Этот уровень является логическим представлением модели обработки запросов клиентов, для которой существуют следующие группы функций обработки запросов:

– анализ запросов (проверка синтаксиса и семантики, определение типов и видов запросов);

– обработка запросов (поиск и отбор данных, операции над данными, внесение изменений в базы и файлы данных, определение объектов данных для генерации ответа, обеспечение целостности объектов данных);

– генерация ответа.

Уровень реализации операций обработки данных. Этот уровень является логическим уровнем операций обработки данных. Каждая операция инкапсулируется в определенном объекте данных. Типичные элементы этого уровня – объекты данных, которые выбирают из баз или файлов данных определенную информацию с критерием отбора, выполняют логическую обработку данных, генерируют фрагменты, как составляющие ответа клиенту.

Уровень представления данных. На этом уровне архитектуры находятся элементы, которые являются представлением данных веб-приложения и используются во время обработки запроса клиента.

Классификационные характеристики для серверной части. Первая важная характеристика – ЯП для описания серверных компонентов. Для формирования характеристик существует два решения. Первое решение – введение единой характеристики для всех возможных ЯП. Однако, такой подход имеет недостаток. ЯП серверных компонентов зависят от выбора конкретного веб-сервера и поэтому целесообразно применять второе решение – введение нескольких характеристик в зависимости от конкретного сервера. Кроме того, такое решение для классификации является информативнее первого.

Для сервера MS Internet Information Server эта характеристика имеет такой вид:

T^6_{IIS} = {ASP, JavaScript (серверный), VB Script (серверный), C, C++, ASP.NET, C#, C++ .NET, VB .NET, J# .NET, XML, SQL}.

Сервер Apache не поддерживает программирование в разных ЯП. Однако, для него существует много дополнительных средств функционирования компонентов, написанных в разных ЯП. Для наиболее распространенных языков

T^6_{APACHE} = {PERL, PHP, PYTHON, XML, SQL},

и характеристика Java-среды (Tomcat, JBoss, WebSphera, WebLogic и др.) имеет вид:

T^6_{JAVA} = {JAVA, JSP, JSTL, JSF,XML, SQL}.

Следующая классификационная характеристика связана с архитектурными уровнями серверной части веб-приложения. Она не охватывает уровень распределенной среды (это сетевой уровень и он существует независимо от веб-приложений) и базовый уровень веб-сервера

T^7 = {прикладные компоненты веб-сервера, управление обработкой запроса, операции обработки данных, представление данных}.

Компоненты уровня прикладных компонентов веб-сервера имеют следующую характеристику, что вводит признаки типового шаблона проектирования MVC:

T^8 ={контролер, серверные страницы, функциональные компоненты}.

К контролерам принадлежат компоненты, реализующие логику управления работой всех прикладных компонентов серверной части веб-приложения. Серверные страницы – это компоненты формирования HTML-ответа, который отправляется клиенту. Функциональные компоненты - это обобщенное название всех компонентов для реализации бизнес логики и обработки данных.

Для компонентов уровня управления обработкой запроса введем такую характеристику: T^9 = {анализ запроса, обработка запроса, генерация ответа}.

Компоненты этого уровня непосредственно реализуют логику обработки запросов. Они встроены в компоненты предыдущего уровня. Для своей работы такие компоненты используют функциональность компонентов, которые находятся на уровнях обработки данных и представления данных.

Для компонентов уровня реализации операций обработки данных введем следующую характеристику:T^{10}={отбор данных, обработка данных, формирование результатов}.

Компоненты этого уровня выполняют основные функции бизнес логики веб-приложения для операций обработки данных. Эквивалент таких компонентов – сессионные компоненты в модели EJB, основная задача которых состоит в анализе потребностей и инициирования выбора данных, необходимых для выполнения операции и формирования результатов, которые применяются при генерации ответа клиенту. Операции для доступа к данным и их сохранения в информационных ресурсах выполняют компоненты следующего уровня.

Для компонентов уровня представления данных введем следующую классификационную характеристику:

T^{11}={модели данных, описание доступа, операции обработки данных}.

Компоненты этого уровня непосредственно связаны с данными, находящимися в информационных ресурсах (базы данных, документы, файлы и др.), ответственны за доступ к данным и поддержку взаимодействия с ресурсами. Аналогом таких компонентов являются entity-компоненты в модели EJB.

Дополнительные классификационные характеристики серверных компонентов связаны непосредственно с конкретной бизнес–логикой, обработкой данных, их выбор обусловлен субъективными факторами. Такие характеристики целесообразно вводить для определенной предметной области, определенных методов обработки данных и др. Для серверных компонентов возможно введение качественных характеристик. Все эти характеристики выходят за рамки архитектурных признаков и потому не рассматриваются.

Совокупность классификационных характеристик $T = \{T^1, T^2, T^3, T^4, T^5, T^6_{IIS}, T^6_{APACHE}, T^6_{JAVA}, T^7, T^8, T^9, T^{10}, T^{11}\}$ определяют систему классификации компонентов для веб-приложений с архитектурными признаками (рис.5.2.).

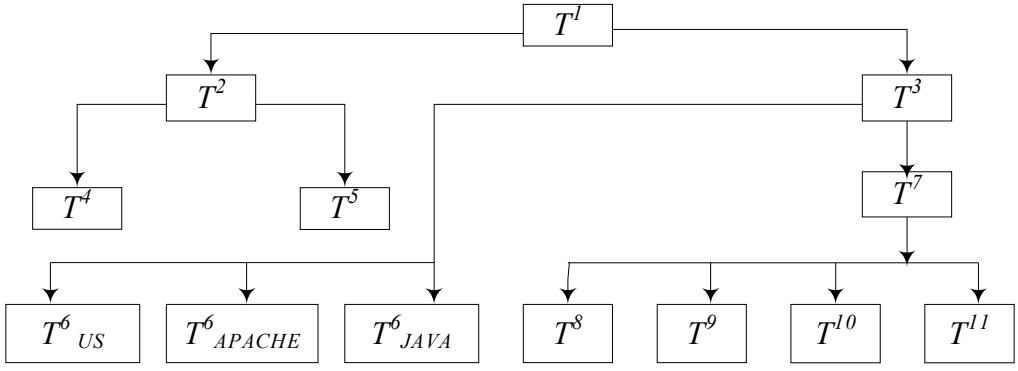


Рис. 5.2. Дерево классификационных характеристик компонентов веб-приложений

5.2. КЛАССИФИКАЦИЯ КОМПОНЕНТОВ ПОВТОРНОГО ИСПОЛЬЗОВАНИЯ

КПИ могут быть готовые программные элементы, как готовые «детали», которые

можно объединить в более сложную программную конструкцию. Именно эта особенность характеризует повторное использование как систематическую и целенаправленную деятельность, как по классификации и каталогизации КПИ в хранилищах, а также по применению их в новых разработках программных и информационных систем [20–22, 133, 221, 227] .

Систематическое повторное использование в программировании – это капиталоемкий подход, предусматривает построение, накопление КПИ и конструирование из них новых ПС.

Построение готовых ресурсов как КПИ исходит из решения типичных задач для семейства систем, описания их в виде обычных компонентов или КПИ и хранения их в каталогах хранилищ для организации поиска по запросам пользователей.

С общей точки зрения новый компонент или КПИ определяют по-разному в зависимости от среды и функций. Приведем некоторые определения.

Определение 1. *КПИ* – это некоторая функция с определенными свойствами и атрибутами, обеспечивающая взаимодействие со средой и задает поведение.

Определение 2. *Готовый КПИ* – это совокупность методов с определенной сигнатурой и типами данных, которые передаются и возвращаются после реализации соответствующего метода некоторым объектом.

Определение 3. *Компонент* – это самостоятельный программный элемент, который удовлетворяет определенным функциональным требованиям относительно архитектуры, структуры и организации взаимодействия в заданной среде, имеет спецификацию, которая помогает пользователю применять и объединять его с другими компонентами в сложную систему.

Определение 4. *Программный компонент* – это независимый от ЯП самостоятельно реализованный объект, обеспечивающий выполнение определенной совокупности прикладных сервисов, доступ к которым возможен через интерфейсы, указывающие функции и операции обращения к компоненту.

Данные определения отвечают разным условиям использования компонентов и КПИ и дают возможность создавать из них новые ПС.

Конструирование новых ПС из подобранных готовых ресурсов из разных хранилищ возможно осуществлять средствами межъязыкового и межмодульного интерфейса сборочного программирования.

Построение КПИ требует капиталовложений, а конструирование новых ПС с применением КПИ приносит прибыль за счет уменьшения трудозатрат на повторную разработку. Инвестиции в повторное использование требуют оценки вложения капитала, прогнозирования сроков и объемов возвращения этого вложения, оценки рисков и др. Прибыль при повторном использовании, как любой бизнес, требует специальных условий менеджмента инженерной деятельности создания продуктов из КПИ.

Сущность успеха такого бизнеса состоит в следующем:

1) повторное использование КПИ требует меньших трудозатрат, чем их новая разработка;

2) поиск пригодных для использования компонентов требует меньших усилий, чем реализация необходимых функций в проектируемой системе;

3) сборка или развертывание КПИ в новых условиях применения нуждается в меньших трудозатратах, чем полностью выполненная разработка.

Архитектура, в которую встраивается готовый КПИ, поддерживает стандартные механизмы работы с компонентами как со строительными блоками. Высокая эффективность повторного использования повышает функциональность, удобство и качество реализации будущей системы.

Классификация КПИ

КПИ с общей точки зрения – это артефакты деятельности разработчиков программных продуктов, которые необходимые другим для применения в новых разработках. Ими могут быть следующие:

– модели ПрО в терминах понятий и лексики этой области;

– готовые компоненты, системы или отдельные части систем;

– промежуточные продукты процесса разработки ПС (требования, постановки задач, архитектура и др.);

– диаграммы, паттерны объектов, которые могут использоваться другими и др.

Артефакты, как готовые ресурсы, должны удовлетворять таким требованиям, как независимость от конкретной платформы, наличие стандартного интерфейса и параметров настройки на новые условия выполнения, возможность взаимодействия с другими компонентами.

КПИ могут быть прикладными и общесистемными. *Прикладные компоненты* выполняют отдельные задачи и функции прикладной области (домены бизнеса, коммерции, экономики и т.п.), которые могут использоваться в дальнейшем как прикладные системы в других доменах с аналогичными функциями.

К *общесистемным компонентам* относятся компоненты общего и универсального назначения, а также общесистемные сервисные средства, которые обеспечивают системное обслуживание и предоставляют разные виды сервиса для многих создаваемых ПС различного назначения. Компоненты общего назначения – это трансляторы, редакторы текстов, системы генерации, интеграции, загрузчики и др. Они используются прикладными системами в процессе их проектирования и выполнения.

Универсальные системные компоненты обеспечивают функционирование любых (в том числе и прикладных) компонентов, обмен данными и передачу сообщений между всеми видами систем и компонентов, размещенных в разных операционных средах. К ним относятся: ОС, СУБД, сетевое обеспечение, электронная почта и др.

Связь между прикладными и общесистемными средствами осуществляется посредством стандартных интерфейсов, которые обеспечивают взаимодействие разных типов компонентов через механизмы передачи данных и сообщений.

Приведем классификацию готовых компонентов:

– процедуры и функции на ЯП высокого уровня;

– алгоритмы, программы;

– классы объектов и абстрактные классы;

– структуры данных и часто используемая информация (например, информационные ресурсы Интернета);

– API, IDL-модули в библиотеках (например, GUI, графика и др.);

– Веб-ресурсы и сервисы;

– средства развертывания систем и компонентов в операционной среде (например, CORBA, COM, .NET) [40, 166, 176, 209];

– готовые решения в виде абстракций – паттерны, фреймы и др.

Все разнообразие видов и типов готовых компонентов требует от разработчиков новых ПС поиска КПИ и изучения их особенностей для использования в этих ПС.

Разработка ПС из КПИ выполняется по следующим общим процессам:

– анализ функций КПИ присущих группам объектов в репозитарии;

– разработка новых функциональных компонентов и доводка их к уровню повторного использования;

– разработка или поиск интерфейсов КПИ в репозитарии;

– сборка КПИ в новую конфигурацию системы.

Спецификация КПИ

КПИ могут быть классами, созданными в рамках объектно-ориентированного программирования (например, огромная библиотека классов в C++) вместе с реализацией их методов.

В компонентном программировании наследуется реализация компонента и его интерфейсы. Компонент может изменяться и пополняться новыми функциональными возможностями и интерфейсами. Один компонент может содержать реализацию нескольких разных интерфейсов, а один интерфейс, в свою очередь, может быть реализован в разных компонентах. Замена одного компонента другим не приводит к перекомпиляции ПС или перенастройке связей в ней.

Пример КПИ – это контейнерные классы, которые сохраняют структуры данных с правилами их запоминания или выдачи элемента, который входит в контейнер. Механизм контейнеров реализован в C++ в виде так называемых шаблонов (templates) и их библиотек.

Каждый компонент имеет связь с внешними компонентами, наследует интерфейс и инкапсулирован как «черный ящик» без возможности вмешательства в его исходный код.

Модель спецификации имеет такой вид:

$$M_{КПИ} = (T, I, F, R, S),$$

где T – тип компонента; I – множество интерфейсов компонента; F – функциональность компонента; R – реализация, скрытая часть – программный код; S – сервис для взаимодействия со средой или набор правил развертки.

Каждый элемент спецификации компонента представляет собой видимую или скрытую от пользователя часть его абстракции.

Другой вид классификации КПИ определяется сложностью:

– простые компоненты (функция, модуль, класс и др.);

– компонент как объект, который имеет интерфейс, функцию и реализацию, а также возможность дополнять спецификацию шаблоном развертки и интеграции;

– готовые к использованию КПИ (например, beans компоненты в Java, AWT-компоненты и т.п.);

– КПИ типа каркасов, паттернов с элементами группирования нескольких простых КПИ в структуру, где они взаимодействуют в процессе решения задач системы.

Для удобства работы с КПИ создается метаинформация о том, какие классы совместимы с семантическими ограничениями, явным образом определенными в спецификации КПИ, содержащая описание информации вида:

– интерфейсов, которые реализуют компоненты;

– механизмов обеспечения повторного использования;
– среды развертки компонентов;
– сервиса, поддерживаемого компонентом;
– ролей, которые выполняют компоненты в системе;
– формализованных языков описания КПИ.

Применение КПИ проводится по такой технологии:

– отображение способности КПИ анализировать себя и динамически выполняться, т.е. управлять свойствами, событиями и методами, встроенными в него;

– стандартизированное описание для анализа и понимания его другим лицом, а не разработчиком;

– трансформация с возможным изменением структуры и исходного кода и с сохранением функций;

– сохранение параметров конфигурации в постоянной памяти для повторного использования;

– регистрация сообщений о событиях, полученных от других объектов через механизм ссылок (например, EJB-компоненты в JAVA),

– каталогизация КПИ в архивы (например, JAR-файлы) для дальнейшего повторного использования;

– адаптация КПИ к разным контекстам и средам использования, выделение свойств, которые мешают повторному их применению и модификации.

Репозитарий компонентов КПИ

КПИ и другие самостоятельные компоненты многоразового использования размещают в разных хранилищах. Ими могут быть библиотеки, репозитарии компонентов и ресурсов в Интернете (например, GreedStone, Matlab, библиотека классов C++ и др.).

Компоненты этих хранилищ используют много раз все желающие для реализации своих целей, в частности при построении ПС различного назначения. Репозитарий типа библиотеки GreenStone или Matlab предоставляет готовые программы научного, в частности математического, типа. Они ориентированны на математиков, физиков и других специалистов предметных областей.

В общем случае репозитарий – это система средств для хранения, пополнения наработанных КПИ, которая содержит инфраструктуру разработки ПС из компонентов, организацию доступа к КПИ из хранилища для дальнейшего использования в новых программных проектах.

С функциональной точки зрения репозитарий работает по принципу информационно-поисковой системы, объектами хранения которой есть различные типы документов, текстов и др. Система ставит запрос пользователя в соответствие ключевым понятиям, словам, правилам доступа, которые содержат коллекции документов.

Разделы репозитария могут содержать:

– готовые функциональные компоненты, КПИ;
– средства безопасности, защиты и изменений для КПИ;
– терминологию специфики представителей семейства ПС;
– средства взаимодействия, синхронизации компонентов;
– новые члены семейства ПС;

– сервисы для членов семейства и т.п.

В отличие от систем поиска информации в репозитарии компонентов, кроме КПИ, размещается семантическая информация в виде поискового образа, созданного на основе описания информационной модели каждого компонента. Эта модель–средство построения поискового образа для каталога КПИ, ориентированного на осмысление человеком функций КПИ и возможности их сопоставления с собственными потребностями.

Поисковый образ готовых компонентов в репозитарии может содержать:

– список ключевых слов, которые чаще всего упоминаются в тексте КПИ;

– ссылки на онтологию домена проблемной области, которой этот КПИ принадлежит.

Информационную потребность в КПИ формулирует пользователь в виде поискового запроса, который сопоставляют с описанием поискового образа КПИ в репозитарии. Поиск КПИ по заданному запросу выполняют до тех пор, пока не будет найден нужный КПИ или, наоборот, пользователем будет получает большое количество релевантных данному запросу компонентов для отбора нужных. При рассмотрении этих компонентов используется онтологическая, понятийная база репозитария, включающая в себе информационную модель каждого КПИ с унифицированной терминологией, ключевыми словами и т.п.

Информационная модель КПИ обеспечивает хранение, поиск и сопоставление КПИ. Репозитарий разделен на разделы согласно ПрО, перечень которых находится на первом уровне репозитария, на следующих – отдельные понятия ПрО. Разделы репозитария заполняют согласно проведенной *типизации и классификации* компонентов и КПИ.

Информационная модель поискового образа упрощает поиск необходимых КПИ и сокращает сроки разработки ПС за счет:

– отображения базовых функций и понятий компонента;

– засекречивания представления данных, операций обновления и получения доступа к этим данным;

– обработки исключительных ситуаций, которые происходят в процессе выполнения и др.

5.3. ИНФОРМАЦИОННЫЕ РЕСУРСЫ (ИР). КЛАССИФИКАЦИЯ И ТИПИЗАЦИЯ

Информационный ресурс – это логически упорядоченная совокупность данных и средств их представления, обеспечивает создание, хранение и использование информации об определенных предметных и проблемных областях, отдельных аспектах и свойствах, коллекциях объектов данных, количественных и качественных характеристик и показателей и т.п.

Это определение носит обобщенный характер и обеспечивает широкий спектр интерпретации ресурсов для сложных информационно-вычислительных систем. Главное назначение ресурса – предоставление информации клиенту по его запросу. Механизмы создания, формирования данных, а также хранения и представления ресурсов носит произвольный характер. Главное – получить ответ на запрос ресурса. В частности, ресурсом может быть прикладная программа, которая генерирует выходную информацию, воспринимающая извне как данные ресурса.

Классификация ИР – совокупность принципов типизации и упорядочения множества информационных ресурсов соответственно их функциональности, методов и средств представления, алгоритмов, схем и механизмов получения данных, доступа к информации. Согласно приведенного определения, классификаций информационных ресурсов существует довольно много. Наиболее распространенные строятся по таким признакам:

– типизация объектов данных (документы, таблицы, коллекции и т.п.);

– структурные признаки (файлы, каталоги, разделы и т.д.);

– методы формирования (базы данных, веб-сайты, прикладные программы и приложения и т.п.).

Тип ИР определяет класс информационных ресурсов, заданный конкретными значениями классификационных признаков относительно их определенной классификации. Типизация информационных ресурсов определяет методы и механизмы реализации ресурса, представление информации, а также доступа к ней. Пример – реляционные БД, где представление данных осуществляется на основе реляционной модели, механизмом реализации ресурса является реляционная СУБД, а методами доступа к информации – язык SQL.

Другим примером может быть файл данных в формате XML, основанный на языке разметки XML. Средством создания XML-файла является соответствующий генератор или редактор XML-данных, а механизмы доступа построены на специальных средствах, которые называются парсерами.

Типизация ИР заключается в унификации и стандартизации методов создания, представления и использования данных для определенного подмножества ИР согласно их функциональной сущности, техническим и технологическим особенностям.

Менеджер ИР – программный объект информационно-вычислительной системы, который обеспечивает управление и применение соответствующего информационного ресурса на основе стандартизированных правил и механизмов доступа к данным ресурса. Менеджер ИР по сути – это оболочка ресурса, реализующая стандартизованные интерфейсы для обеспечения процессов интеграции.

Тип менеджера ИР – класс программных объектов, который определяется совокупностью унифицированных и стандартизованных методов применения и доступа к ресурсам определенного типа. В частном случае конкретный тип может состоять только из одного экземпляра. Такая ситуация бывает, когда в состав интегрированной среды включена существующая система, которая создана на старых платформах и ЯП. В этом случае разрабатывается менеджер ресурсов для обеспечения взаимодействия только с такой системой.

Главная цель типизации менеджеров ИР состоит в упорядочении и унификации всех интерфейсов, которые соединяют менеджера с определенным классом ИР и с интегрированной средой. Пример типизации – это менеджер работы с XML-файлами в ОО программной системе. Такой менеджер взаимодействует с ресурсом на основе методов и средств языка XML. Со стороны интегрированной среды существует требование представления данных ресурса как отдельных объектов. Поэтому менеджер выбирает из XML-файла данные и подает в виде объектов, например, на основе DOM-модели.

5.3.1. ТИПОВАЯ СХЕМА СВЯЗИ С МЕНЕДЖЕРОМ ИР

Типовая схема соединения интегрированной среды с менеджером ИР представлена на рис.5.3.

Унифицированный запрос к менеджеру ресурса отображает структуру запроса со стороны интегрированной среды к менеджерам ресурсов определенного типа. При компонентной ориентации унифицированный запрос – это обращение к методу из входного интерфейса компонента–менеджера ресурса.

Типизированный запрос имеет структуру, которую непосредственно понимает конкретный ИР. Одна из функций менеджера ИР – трансформация унифицированного запроса в один или в последовательность типизированных запросов, которые необходимо выполнить, чтобы получить соответствующую информацию из ресурса. Например, унифицированный запрос к БД трансформируется в SQL-запрос для получения информации из реляционной БД. Реализация механизма типизированного запроса в целом может отличаться от механизма компонентного взаимодействия. Этот факт рассматривается как внутренний аспект в концепции представления менеджера ресурсов как обобщенного интерфейса, который инкапсулирует внутренние особенности реализации.

Ответ на типизированный запрос определяется структурой данных, которую формирует ИР согласно своей функциональности. Например, результат обработки SQL-запрос к БД подается в табличной форме (в случае, если ресурс содержит необходимую информацию).

Ответ на унифицированный запрос – структура данных, стандартизированная для определенной интегрированной среды, как типичный ответ для менеджера ресурсов. Например, таблица из предыдущего примера может оставаться без изменений или трансформироваться в XML-структуру данных, или быть представлена как HTML-страница и т.п. Механизм реализации ответа на унифицированный запрос – это составляющая механизма компонентного взаимодействия.

Проведем сравнительный анализ между схемой, которая приведена на рис.5. 3, и компонентной средой. Основу последнего составляет компонентный framework, представленный как совокупность серверов компонентов, взаимодействующих между собой, а также совокупность компонентных сервисов. Между компонентами и компонентным framework существуют стандартизированные механизмы подключения компонента к среде, организация взаимодействия и обмена данными. Как следствие, сервера интеграции – это компонентные серверы, а общесистемные сервисы именования, транзакций, безопасности реализуются для поддержки компонентов. Менеджер ресурсов – компонент, который независим от методов построения самого ресурса (последний может быть реализован, как на компонентной основе, так и на произвольной платформе).

Типичная процедура включения ресурса в интегрированную среду содержит следующие процессы:

– развертывание менеджера на одном из компонентных серверов согласно общим правилам развертывания компонентов;

– организация взаимодействия менеджера с ресурсом с учетом особенностей последнего (выбор адаптера или драйвера, связь с местонахождением ресурса и т.п.);

– конфигурирование менеджера в компонентной среде для обеспечения доступа

к нему со стороны других компонентов.

5.3.2. ТИПОВОСТЬ СТРУКТУРЫ И ИНТЕРФЕЙСА МЕНЕДЖЕРА ИР

Менеджер ИР – это подсистема, которая состоит из нескольких компонентов (рис.5.3), в которую входят следующие:

– контейнер, создающий и обеспечивающий управление вариантами соединений с ИР;

– интерфейсы с ИР, которые обеспечивают реализацию функциональности доступа к ресурсу;

– компонент управления взаимодействием для определенного соединения;

– адаптер доступа к ресурсу;

– дескриптор ИР, который является информационной базой для создания соединения и организации взаимодействия.

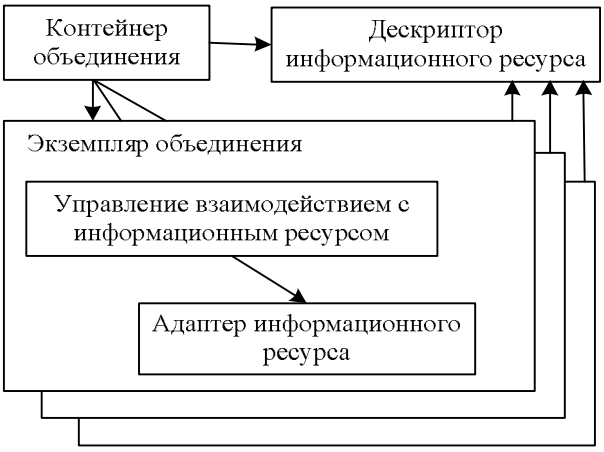


Рис.5.3. Типовая структура менеджера ИР

Роли и функциональность между компонентами менеджера ресурсов распределены следующим образом. Дескриптор ИР является информационной базой, которая поддерживает функционирование самого ресурса и его менеджера на протяжении всего ЖЦ. В частности, она содержит такую информацию:

– описание параметров развертывания менеджера как компонента;

– описание местонахождения или адреса ресурса, а также параметров доступа к нему;

– тип и характеристики менеджера ресурса;

– описание интерфейсов, методов, структур входных и выходных параметров;

– фрагменты документации менеджера и информационного ресурса.

Менеджер как типовой выступает в качестве контейнера экземпляров, каждый из которых является реализацией определенного соединения с ресурсом. Экземпляр создается во время инициации соединения и существует до вызова метода завершения соединения с ресурсом.

Экземпляр соединения состоит из нескольких компонентов, типовыми из которых есть компонент управления взаимодействием с ресурсом и адаптер ИР. Компонент управления взаимодействием с ресурсом обеспечивает реализацию входного интерфейса менеджера ресурсов, который определяет функциональность процедур доступа к ресурсу. К его функциям, в частности, относят:

– преобразование входных параметров унифицированного запроса к менеджеру в структуры данных адаптера ресурса;

– организация взаимодействия с адаптером ресурса;

– преобразование результатов выполнения запроса в структуры данных для выходных параметров.

В частном случае этот компонент может отсутствовать, если входные параметры унифицированного запроса не нуждаются в дополнительной обработке и воспринимаются адаптером ресурса непосредственно.

Адаптер ресурса взаимодействует с ИР соответственно механизмам доступа, реализованных в средствах управления ресурсом, например, средствах СУБД для доступа к БД.

Интерфейсы менеджера ИР. В компонентной структуре взаимодействие между всеми компонентами происходит с помощью интерфейсов. Поскольку менеджер ИР также представляется как компонент, то его функциональность рассматривается как реализация определенных интерфейсов.

Согласно общей архитектуре основными функциями и принципами построения функциональности менеджера ИР является реализация определенной совокупности типовых интерфейсов, обеспечивающих:

– соединение с менеджером ИР;

– взаимодействие с менеджером;

– представление данных.

Интерфейс соединения с менеджером ИР – это интерфейс контейнера соединений согласно типовой структуре менеджера. Он обеспечивает реализацию следующих основных типовых функций:

– поиск необходимого менеджера в интегрированной среде;

– создание экземпляра соединения для менеджера (одновременно с менеджером ресурсов могут быть связаны несколько экземпляров соединений согласно количеству обращений к нему);

– установка системных контрактов, в частности, для поддержки транзакций и безопасности.

Интерфейс обеспечения взаимодействия с менеджером – это прикладной интерфейс, который реализует экземпляр соединения и выполняет такие типовые функции:

– получение информации об описании экземпляра соединения, в частности перечня методов, которые реализуют функциональность для получения данных из ресурса;

– обращение к методам для получения необходимых данных из ИР.

Интерфейс представления данных – это прикладной интерфейс, он реализует экземпляр соединения и осуществляет:

– формирование типовых структур данных для информации, полученной из ресурса;

– преобразование между разными структурами данных и их представлением;

– операции с дескрипторами и метаданными для типовых структур данных.

Для отдельных или разных реализаций ПС концепция интеграции на основе менеджеров ресурсов может существовать с другими свойственными им интерфейсами.

5.3.3. АСПЕКТЫ ПРИМЕНЕНИЯ МЕНЕДЖЕРА ИР

Сущность концепции построения ПС на основе интеграции менеджеров ресурсов состоит в том, что разработчик отдельной программы или подсистемы вместе со средствами, которые непосредственно реализуют основную функциональность, создает специальный программный элемент - менеджер ресурсов, с помощью которого и происходит взаимодействие.

На свойства менеджера ресурсов влияют много факторов, например, основные из них:

– целевая среда, в рамках которой интегрируются программы и подсистемы;

– механизмы установления соединений и организации взаимодействия;

– наличие типовых и стандартизированных прикладных интерфейсов (API);

– способы и методы представления данных для работы с менеджерами ресурсов;

– обеспечение поддержки протоколов взаимодействия и общесистемных сервисов, на основе которых функционирует целевая среда;

– уровень унификации и типизации информационных ресурсов, которые интегрируются в среду.

В зависимости от выбора основных воздействующих факторов и уровня решения соответствующих проблем существуют много концепций применения менеджеров ИР.

Типичным примером архитектурных и структурных решений на основе применения менеджеров ИР является концепция связи открытых баз данных - ODBC (Open DataBase Connectivity), позволяющая единообразно обращаться к ресурсам БД. Данной концепции соответствует стандарт, который определяет архитектуру и принципы реализации ПС, взаимодействующих с БД в сетевой среде. В этой архитектуре типовым клиентом является конкретное приложение, которое с помощью прикладного интерфейса API обращается к специальному менеджеру – ODBC Driver Manager (рис.5.4). Этот менеджер, являясь типовым элементом, реализует интерфейс с разными СУБД на уровне типовых функций и структур данных с учетом конкретной БД. Фактически ODBC Driver Manager вместе с ODBC-драйвером по сути являются менеджером ресурса для соответствующей БД.

Другой пример подобной архитектуры – концепция JDBC (Java DataBase Connectivity — соединение с базами данных на Java) — платформенно-независимый промышленный стандарт взаимодействия Java-приложений с различными СУБД и основана на драйверах, позволяющих получать соединение с БД с помощью URL [25, 26, 161, 209].

Провайдеры данных в технологиях OLE DB [209] или ADO.NET [176] также являются примером менеджеров ресурсов для доступа к БД, документам, файлам и т.д.

Реализованная с помощью концепции менеджера ресурсов архитектура коннектора, является составляющей Java-среды J2EE [25]. Основная цель архитектуры коннектора – обеспечить унифицированный подход к интеграции различных программных средств и систем на основе типовых механизмов взаимодействия, стандартных интерфейсов и сервисов, единых методов построения интегрированных сред. Сфера применения – корпоративные системы, которые состоят из разнообразных программных средств и объектов, охватывающих

существующие корпоративные ИР, созданные в разное время и на разных платформах (рис.5.4.).

Для каждого ИР необходимо существование специальной подсистемы доступа к нему, принципы построения и функционирования которой определяются архитектурой конектора. Спецификация включает в себя также такие понятия, как ресурс корпоративной информационной системы (EIS Resource), менеджер ресурсов (Resource Manager), адаптер ресурсов (Resource Adapter), системный и прикладной контракты (System and Application Contracts) и др.

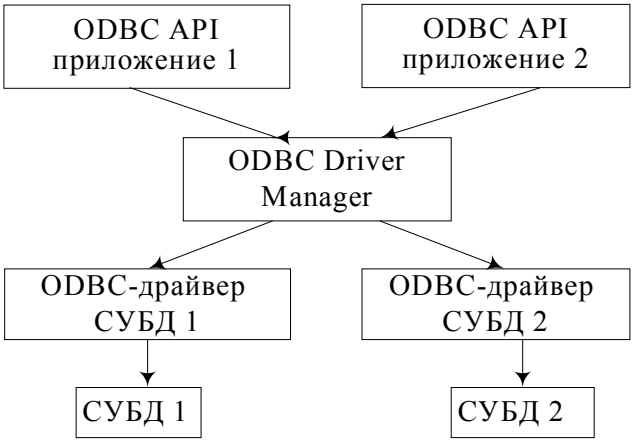


Рис.5.4. Архитектура ODBC-доступа к БД

Особенностями упомянутой архитектуры является спецификация таких элементов, как обеспечение условий соответствия адаптера ресурсов требованиям общих системных сервисов (соединений, транзакций, безопасности), ЖЦ адаптера ресурсов, интерфейсов управления соединениями, транзакциями, безопасностью, роботами и др.

Общая схема построения среды на основе интеграции ресурсов корпоративной системы приведена на рис.5.5. Ядро интеграции – сервер приложений, который функционирует как J2 EE-сервер. Он построен на основе Java-технологий, обеспечивающих поддержку стандартных механизмов взаимодействия Java-объектов и реализацию основных общесистемных сервисов, таких, как именование, транзакции, безопасность и др. В частности, сервер приложений имеет специальное расширение, которое поддерживает реализацию определенных API, обеспечивающих стандартные методы работы с адаптерами ИР.

Кроме того, адаптеры, непосредственно взаимодействующие с ресурсами, также должны удовлетворять требованиям их интерфейсов и условиям поддержки общесистемных сервисов. Стандартизация механизмов взаимодействия позволяет интегрировать одни серверы приложений с несколькими корпоративными ИР, а также интегрировать определенный ресурс с несколькими J2 EE-серверами.

5.4. ИНТЕГРАЦИЯ МЕНЕДЖЕРОВ РЕСУРСОВ

Существует много реализаций интеграции менеджеров ресурсов в интегрированных средах. Каждый из них имеет свои особенности, обусловленные

решением конкретных задач, выбором базовых объектов для интеграции, способов и механизмов взаимодействия и т.п. Сфера применения каждого подхода зависит от уровня унификации, типизации, стандартизации отдельных аспектов и составляющих этого подхода. Реализация концепции ODBC предназначена для решения проблемы интеграции баз данных, и сфера ее применения полностью определяется целью создания соответствующей архитектуры (рис.5.5).

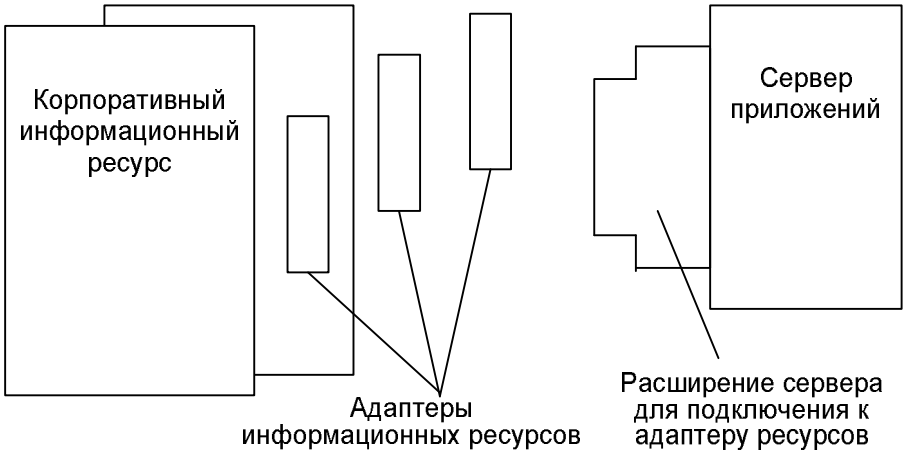


Рис.5.5. Общая схема интеграции ресурсов в архитектуре коннектора

Кроме различий в целях и задачах, отдельные подходы к интеграции могут иметь разные теоретические и методические основы и быть построены с применением разных моделей и платформ, функционировать на основе разных механизмов и технологий и т.п. Выше приведены основные факторы, которые влияют на менеджеров ресурсов на конкретные реализации разных подходов к интеграции. Детализация воздействия на процесс интеграции факторов и их свойств определяет конкретный метод интеграции на основе менеджеров ресурсов.

Разнообразие в методах подходов и реализациях создает определенные трудности при распространении и расширении сферы применения концепции менеджеров ресурсов и их интеграции. Различные среды интеграции, разные типы прикладных интерфейсов и правила взаимодействия – все это ограничивает возможности интеграции разнородных ресурсов. Например, если в состав конкретной среды входят разные типы серверов интеграции, то для каждого информационного ресурса должны быть разработаны несколько типов менеджеров ресурсов, что значительно усложняет применение упомянутой концепции, увеличивает объем работ программированию, усложняет также правила создания интегрированной среды и взаимодействия его отдельных элементов.

Выход из этой ситуации – применение таких теоретических и практических основ, которые бы максимальным образом способствовали решению проблемы, а также наиболее широко охватывали различные аспекты концепции в целях обеспечения необходимого уровня унификации и стандартизации структуры среды, правил интеграции, прикладных интерфейсов и др. Одним из более перспективных подходов является компонентный подход, который способствует решению вопросов практической реализации концепции интеграции менеджеров ИР на

единой теоретической, методической и прикладной основе.

Метод построения ПС на основе компонентной интеграции менеджеров ресурсов базируется на таких основных аспектах [209]:

– все объекты интеграции – суть компоненты, в том числе и менеджеры ИР;

– интегрированная среда – это компонентный framework ;

– серверы приложений и их общесистемные сервисы содержат необходимые операции по развертыванию, функционированию и взаимодействию компонентов;

– механизмом взаимодействия в интегрированной среде является их интерфейс;

– менеджеры ИР инкапсулируют ресурсы интегрированной среды и выступают как обобщенные интерфейсы;

– каждый компонент имеет типовую структуру, которая определяет его свойства, механизмы функционирования и ЖЦ (это, в частности, наличие контейнера и компонентных реализаций, существование механизма создания экземпляров компонента на основе фабрики компонентов, схема взаимодействия компонентов согласно интерфейсам и общесистемным сервисам и т.д.).

Концепции и принципы создания среды на основе интеграции менеджеров ресурсов положены в основу разработки веб-приложения для навигации и доступа к ИР в сети Интернета. Программные средства используют компонентный подход к разработке веб-приложений, который определяет, регламентирует и поддерживает выполнение типовых функций и операций в сети Интернета.

Основой интегрированной среды – веб-сервер, в рамках которого реализован компонентный framework, разработанный согласно компонентному подходу [67], веб-приложение для навигации и доступа к ИР реализовано как совокупность компонентов, которые функционируют в серверной компонентной среде согласно правилам и схемам обработки данных, определенных в компонентном framework. Пример описания веб-приложения методом интеграции менеджеров ИР приведен в Приложении 1.

Интегрированная среда содержит три типа менеджеров ИР:

– веб-сайтов;

– веб-приложений;

– удаленного доступа к БД.

Структура унифицированного запроса определяется стандартным для веб-среды CGI-механизмом. Аналогично форматом ответа на унифицированный запрос является HTML-страница.

Структуры типизированных запросов к ИР и ответы на них зависят от типа менеджера ресурсов.

Дескрипторы ресурсов подаются как XML-файлы, определяющие основные характеристики, по которым создаются соответствующие соединения с ресурсами, и их функциональность. Компонент, обеспечивающий обработку данных из дескриптора, применяет DOM-модель для XML-файлов. Перечни всех ресурсов, которые составляют общую систему ИР, также представлены в виде XML-файлов, распределенных между несколькими веб-серверами, на каждом из которых развернут отдельный экземпляр веб-приложения для навигации и доступа к ИР. Каждый перечень определяет совокупность ресурсов, связанных с конкретным сервером.

5.5. ПОДХОД К СТАНДАРТИЗАЦИИ РЕСУРСОВ ИНТЕРНЕТА

WWW (World Wide Web) – технология Интернета обеспечивает интеграцию мультимедийных объектов, форматов данных и разнообразных инструментальных сред через аппарат гипертекстов. Суть этой технологии состоит в том, что в тексте выделяется слово-ссылка (так называемая гиперссылка – фрагмент текста или изображения), с помощью которого выполняется переход к нужному документу или разделу документа [26].

К основным средствам описания разнообразных ресурсов Интернета относятся:

– язык гипертекстовой разметки документов HTML (HyperText Markup Language);

– универсальный способ адресации ресурсов в сети URL (Universal Resource Locator);

– протокол обмена гипертекстовой информацией HTTP (HyperText Transfer Protocol);

– универсальный интерфейс шлюзов CGI (Common Gateway Interface).

Язык HTML – это средство описания структуры документа и связей его с другими документами. Этот язык предназначен не для форматирования документа, а для его функциональной разметки, создания мультимедийной структуры.

Язык URL предоставляет универсальную форму адресации информационных ресурсов с помощью указателя на ресурс баз данных WWW. В нем задается HTTP протокол и полное имя машины или адрес порта/путь, в который принимается документ, как гипертекст. Протокол передачи файлов ftp используется для доступа к локальным файлам.

Протокол HTTP – это совокупность правил для общения навигатора браузера с веб–сервером. Он предназначен для обмена гипертекстовыми документами с учетом специфики информации, которая реализуется с помощью ASCII-команд. При этом используются общий интерфейс CGI (Common Gateway Interface), как универсальный интерфейс шлюзов, разработанный для расширения возможностей Интернета в целях подключения любого внешнего программного обеспечения, которое вносится в среду WWW как средство интерактивного диалога с пользователем для создания интерактивных форм и динамических документов.

Программа-клиент (браузер) выполняет функции интерфейса с пользователем и обеспечивает доступ ко всем информационным ресурсам Интернета. Клиент выполняет разные функции: запуска дополнительных внешних программ для работы с документами в форматах, отличных от HTML

Сервер протокола HTTP обеспечивает доступ к базе данных документов в формате HTML, управляемым сервером, программы которого разработаны в стандарте спецификации CGI. База данных HTML-документов является частью файловой системы, которая содержит текстовые файлы в формате HTML и связанные с ним другие ресурсы, в том числе и графические. Программы, которые обеспечивают взаимодействие сервера с серверами других протоколов или с распределенными на сети серверами баз данных, принимают данные от сервера и выполняют соответствующие действия (например, получение текущей даты, реализацию графических ссылок, доступ к локальным базам данных или производятся некоторые расчеты). На основе языка XML созданы: объектная модель документа DOM (Document Object Model); средства определения типа

документа DTD (Document Type Descpiption); каркас описания ресурса RDF (Resource Definition Framework).

Semantic Веб – это веб данных, которые управляются и поддерживаются самими приложениями. По существу, это среда общих форматов для обмена данными между различными источниками, а также для хранения, обработки и обмена документами. Semantic Веб обеспечивает интеграцию распределенных метаданных, используя описание объектов (или ресурсов), включающих описание атрибутов и значений. Ресурсы задаются URI идентификаторами, аналогично поэтому, как Web-страницы задаются URL. Каждый атрибут ресурса вместо коротких имен использует URI–идентификаторы. Это связано с необходимостью представлять распределенную информацию в Web, децентрализовано описывать некоторый ресурс в RDF документе, указывать ссылку URI на значение, аналогично тому, как в HTML указывается ссылка на другую веб-страницу. При этом описание ресурса может считаться не полным и окончательным, поскольку в Веб может содержаться и дополнительная информация об этом ресурсе в URI, либо в нескольких RDF-документах.

RDF – язык представления метаданных о веб-ресурсах в WWW, их сущностях, идентифицированных средствами веб-идентификаторов (URI), о данных приложений, которые обмениваются данными в обобщенной среде. Каждый ресурс описывается в терминах свойств и их значений. Ресурсы задаются в графическом виде с помощью вершин и дуг посредством синтаксиса описания, XML (RDF/XML) и URI. Вершинами графа могут быть субъекты и объекты. В качестве субъекта может выступать предикат, содержащий свойства. Объект может выступать в роли субъекта в другой тройке графа RDF. Объединение троек (субъект, объект, свойство) в графе означает их конъюнкцию. Квалифицированное имя Qname в XML обеспечивает запись идентификаторов в тройках и RDF/XML через префикс, которому присваивается URI и локальное имя. Иными словами, URI образует словарь терминов, совокупность взаимосвязанных терминов – URIref, как общие префиксы URI. URIref дает возможность отличать свойства, определяемые одним человеком, от свойств, определяемых другими (или в другом контексте). Например, name может означать ФИО человека или название сценария или переменной в некоторой программе. Так как свойства являются ресурсами, то о них можно записать дополнительную информацию (например, описание на русском языке свойства "name"). Создается предложение в RDF, в котором свойство URIref выступает в качестве субъекта. Значения свойств могут быть структурированными и содержать информацию типа: город, район, улица и др.

RDF представляет бинарные связи. Для задания n-арных связей вводится дополнительная сущность, которая разрешает встроенные типы данных (исключая URIref rdf:XMLLiteral для представления литерального значения XML-контента. Данные, представленные с помощью литерала, могут быть заданы и URI. Литералом может быть только объект и предикат. Простой литерал – это строка с факультативной меткой языка, а типизированный литерал – это строка с URI, т.е со значением из пространства значений, полученное отображением «лексика – значение» в строку литерала (т.е. лексическое значение).

RDF-схема обеспечивает описание словарей классов и атрибутов. Стандартизация RDF-словарей позволила включать свойства метаданных для конкретных предметных областей В стандарте зафиксировано использование

терминов (свойств, значений и т.п.), что дает возможность прикладным программам интегрироваться между собою и обмениваться понятной для них информацией. При получении данных из другой системы текущая программа может найти себе свойства, регламентированные стандартом, и соответственно правильно проинтерпретировать их содержание и семантику. Такая особенность называется *семантической интероперабельностью* в Semantic Веб.

*Стандарт W3C***.** Для спецификации веб-сервисов был создан консорциум W3C (World Wide Web Consortium), который провел взаимную увязку стандартов спецификации для описания веб-сервисов – *W3C*. Инструментом для согласования стандартов стала XML-схема, в форматах которой представляются стандарты и соответствующий набор тегов. Виртуальный способ интеграции веб-сервисов в прикладное приложение обеспечивает взаимодействие сервисов между собою через распределенные вызовы объектов в интегрированном пространстве. Стандарт W3C определяет обращение к веб-сервису в сетевой среде, передачу сообщений между разными сервисами, удаленные вызовы процедур с помощью стандартного протокола коммуникации SOAP (Simple Object Access Protocol). Информация представляется как XML-схема, в которой элемент XML определяется в виде заголовка и тела. Заголовок определяет цель сообщения, а тело – его содержание, типы параметров сообщения, ответа и т.п. Интерфейс веб-сервиса – это коллекции точек взаимодействия (end points), через каждую из которых можно получить от веб-сервиса определенную услугу средствами обмена сообщениями стандарта WSDL (Web Services Description Language) и форматом XML. Ресурсы описываются в стандарте WSDL в виде набора именованных интерфейсов и типов данных, изоморфных простым типам данных.

Вывод. Приведена классификация и типизация современных программных и информационные ресурсов, основанная на общей теории классификации. Ресурсы Интернета составляют базис для формирования новых конкретных систем классификации и типизации, которые могут использоваться при реализации новых предметных областей, доменов и различных приложений. Дан краткий анализ современных стандартных средств Интернета (XML, HTML, URL Semantic Web, RDF, W3C). Эти средства обеспечиваю доступ и интеграцию мультимедийных объектов, форматов данных и инструментальных сред через аппарат гипертекстов.

Глава 6

ОБЪЕКТНО-КОМПОНЕНТНЫЙ МЕТОД РАЗРАБОТКИ ПРОГРАММНЫХ СИСТЕМ

В течение последних двадцати лет в программировании доминировал ООП как основной способ разработки разных программных приложений. ООП внес в программисткую среду новый способ мышления и представления сущностей реального мира с помощью объектов, моделей взаимоотношений между ними, объединения их в классы (наследование) и использование самих объектов, как базисных понятий моделирования объектных программ (полиморфизм, визуализация и др.).

Каждая ПрО – это совокупность понятий, связанных некоторым множеством отношений, которое определяет ее поведение на протяжении времени существования. Классическое определение ОМ [36, 47, 65]:

«объектная ориентация = объекты + наследование».

В роли объектов выступают как абстрактные образы, так и конкретные физические предметы или группы предметов с указанным подмножеством их характеристик и функций. Объект как основа ОМ практически основывается на следующих предположениях:

– каждый объект однозначно соответствует некоторой сущности ПрО, имеет необходимые связи и особенности поведения;

– ОМ определяет меру полноты отображения всех сущностей и их поведение или меру адекватности его восприятия заказчиком.

Полнота отображения сущности объекта отвечает мере абстракции при ее описании.

Иной взгляд на проблемы проектирования ПрО предлагает компонентный метод программирования, называемый от готового, т.е. сборочного типа. В нем задачи ПрО декомпозируются на элементарные и относительно простые самостоятельные функции, которые затем программируются или их находят из числа реализованных самостоятельных КПИ для включения в состав компонентной модели. Понятие самостоятельности компонента, в общем случае, означает, что он создается без ориентации на конкретное применение, а лишь предполагается, что будет использоваться в разных ПС с выполнением требований относительно функциональных возможностей и свойств. Готовые КПИ объединяют или собирают в более сложные ПС в соответствующей композиционной (сборочной) среде.

Обеспечение условий широкого применения компонента – одно из главных целей в процессах разработки ПС в компонентном программировании. Кроме

унификации и стандартизации архитектурных, структурных, технологических характеристик компонента, важную роль играет выбор его функциональности и методов доступа к нему. Это, в свою очередь, требует более детального подхода к определению и спецификации компонента из определенной ПрО, задач, которые решаются в ней, конкретных функций обработки данных и т.п. Сложность этого подхода состоит в том, что для компонента не существует заведомо определенных функциональных требований, как в случае спецификации компонентов для конкретной целевой системы. Поэтому необходимое условие применения КПИ – существование определенных объективных концепций, принципов, критериев относительно выбора и спецификации их функциональных свойств.

Использование концепций и принципов концептуального моделирования ПрО для определения понятий и сущностей ПрО, их взаимоотношений и поведения в ОМ, это основополагающее свойство ООП, позволившее перейти от этой модели к компонентной путем добавления формального аппарата непосредственного программирование функций ПрО в виде отдельных компонентов и компонентных программ. Объединение базовых особенностей ОМ и компонентной модели позволили создать комплексный метод, так называемый объектно-компонентный метод ПС [64]. При этом между объектными и компонентными моделями устанавливают эквивалентные отображения, которые обеспечиваются формальными преобразованиями объектного представления ПрО в компонентное.

6.1. АСПЕКТЫ ТЕОРИИ ОБЪЕКТНОГО АНАЛИЗА

Теория объектного анализа ПрО для построения ОМ основана на формализме треугольника Фреге [197], который позволяет обобщить понятие объекта, как элемента реального мира и сформулировать его свойства и характеристики (рис.6.1).

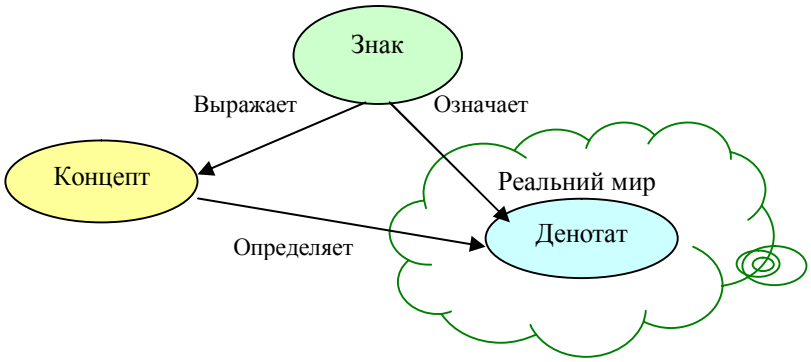


Рис. 6.1. Треугольник Фреге

В треугольнике Фреге:

– знак – идентификатор, который обозначает определенную сущность из действительной реальности, являющейся предметом, событием, понятием и т.п. с определенным содержанием;

– денотат – сущность из действительной реальности, обозначаемая знаком;

– концепт – сущность (семантика) денотата, которую постепенно уточняют на уровнях объектного анализа с привлечением математических формализмов описания и уточнения, отличая один объект от другого.

Главные концепции теории объектного анализа ПрО:

– обобщение понятийной структуры ПрО, которое отвечает формальному понятию треугольника Фреге и представляет объект как денотат с собственным именем и определенным смыслом (содержанием);

– теоретико-множественная концепция, которая основана на принципе различия и приведения в порядок отношений зависимости между множествами объектов и операциями над объектами (объединения, разности, декартова произведения и т.п.);

– логико-алгебраическая концепция для описания множества объектов в виде алгебраической системы с заданной совокупностью предикатов определенной сигнатуры;

– концепция поведения объектов в зависимости от времени их функционирования и формирования состояний, которые фиксируются в диаграмме переходов и используются при выполнении объектов.

Объектный анализ – это первая фаза метода объектно-компонентного проектирования, которая использует названные общие концепции проектирования. Однако, их сущность и содержание подчинены построению ОМ, с помощью которой обеспечивается ее дальнейшее отображение в компонентную модель [65, 66].

Главная цель объектного анализа – подать предметную область как множество объектов со свойствами и характеристиками, необходимыми и достаточными для определения и идентификации отдельных объектов, а также описание их поведения в рамках выбранной системы понятий и абстракций**.**

Каждое понятие ПрО вместе с его свойствами и особенностями поведения в среде это отдельный объект, а вся ПрО – совокупность объектов со связями, которые устанавливаются на базе отношений между этими объектами. В роли объекта выступают как абстрактные образы, так и конкретные физические предметы или группы предметов с указанным подмножеством их характеристик и функций.

При определении объекта применяют такие предположения:

– для каждого объекта устанавливают его соответствие сущности ПрО, а именно, необходимые связи и особенности поведения;

– мера полноты отображения сущности и поведения объекта, как мера абстракции и времени его описания, должна быть адекватна заказчику.

Объект – именованная часть действительной реальности с определенным уровнем абстракции из выбранной ПрО и с целиком обусловленным поведением. Представляется как понятийная структура в виде треугольника Фреге.

Процесс объектного анализа ПрО выполняется в соответствии со следующими принципами.

Принцип всеобщности объектного определения. На произвольном шаге объектного анализа все сущности – суть объекты.

Следствие 1. Предметная область, которая моделируется, сама является объектом.

Следствие 2. Моделируемая предметная область может быть отдельным объектом в составе другой предметной области (иерархия предметных областей).

Принцип существенности объектных различий. На произвольном шаге объектного анализа каждый объект – уникальный элемент.

Следствие 3. Каждый объект имеет, по крайней мере, одно свойство или

характеристику, которая определяет его уникальную идентификацию во множестве объектов предметной области.

Принцип объектной упорядоченности. На произвольном шаге объектного анализа все объекты упорядочены соответственно их отношениям.

Следствие 4. Каждый объект имеет, по крайней мере, одно отношение с другим объектом, которое обеспечивает упорядоченность этой пары объектов.

Принцип целостности объектной модели. На произвольном шаге объектного анализа совокупность объектов и отношений между ними однозначно определяет ОМ из выбранной предметной области с определенным уровнем абстракции и описания.

Следствие 5. Объектную модель можно представить в виде ориентированного связного графа, вершины которого – объекты, а дуги – отношения между объектами.

Конкретизация понятия объекта:

– *класс* – это объект, который представляет собой множество;

– *экземпляр класса* – это объект, являющийся элементом определенного множества, которое само является классом;

– *объединенный класс* – это множество, которое является прямой суммой нескольких других множеств;

– *класс-пересечение* – это множество, которое является общей частью других множеств;

– *агрегированный класс* – это множество, которое является подмножеством определенного декартова произведения нескольких других множеств.

6.1.1. ФОРМАЛЬНЫЕ УРОВНИ АБСТРАКТНОГО ПРЕДСТАВЛЕНИЯ ОБЪЕКТОВ МОДЕЛИ

Приведенные концепции и принципы общего определения объекта послужили основой для определения упорядоченной последовательности уровней представления объектов в ОМ и проектирования объектной системы в целом. Было введено четыре уровня абстрактного представления ОМ. На каждом из этих уровней необходимо применять определенный математический аппарат.

Обобщающий уровень. Получается путем обобщения понятийной структуры ПрО на основе формализма треугольника Фреге, согласно которому объект – денотат с собственным именем и определенным содержанием. Объект на данном уровне рассматривают как класс, а именно, как соответствующее математическое понятие, которое согласуется с аксиоматической теорией множеств Геделя–Бернайса.

Пусть $O=(O_0, O_1, \ldots, O_n)$ – совокупность объектов предметной области, где каждый O_0 соответствует самой предметной области (принцип всеобщности объектного определения). Тогда получаем выражение

$$\forall i \, \exists j \, [(i>0) \,\&\, (j\geq 0) \,\&\, (i \neq j) \,\&\, (O_i \in O_j)]. \qquad (6.1)$$

Между определенными элементами множества O существуют отношения принадлежности. На этом уровне выделяют сущности ПрО путем собственного субъективного восприятия последней и описывают их объектами в виде базовых понятий ОМ. Выделение сущностей осуществляют с учетом отличий, определяющих соответствующие им понятийные структуры, исходя из треугольника Фреге, и фиксируют объекты идентификаторами и концептами.

Результат этого уровня анализа – ОМ. В такой модели на данном уровне размещены только имена объектов ПрО.

Структурно-упорядоченный уровень. Этот уровень базируется на теоретико-множественной концепции, в соответствии с которой объекты, определенные на обобщающем уровне, предоставляют как множества или элементы определенного множества.

Исключая из множества O элемент O_0, который не принадлежит другим элементам, получаем множество $O'=(O_1, O_2, , \dots, O_n)$.

Выражение (6.1) на этом уровне трансформируют в такое:

$$\forall i \; \exists j \; [(i>0) \; \& \; (j>0) \; \& \; (i >\neq j) \; \& \; (O_i \in O_j)]. \qquad (6.2)$$

Здесь каждый из объектов – это множество или элемент определенного множества (согласно теории Геделя–Бернайса), и к ним применяются операции теоретико-множественной алгебры. Также это выражение определяет объектные отношения: "часть–целое", экземпляризация, агрегация и т.п.

Обозначим Ω – совокупность теоретико-множественных операций, а $\Sigma = (O', \Omega)$ – алгебраическая система для этого уровня объектного анализа. К теоретико-множественным операциям отнесены такие: объединения, пересечения, вычитания, сложения, симметричного вычитания, декартова произведения.

Результатом применения таких операций к каждой паре объектов корректное выявление структурной упорядоченности и определения новых объектов, которые не выявлены на обобщающем уровне.

Математические операции множества Ω имеют такой смысл:

– *операция объединения* $\cup (A, B)$, где A и B – объекты со статусом множеств. Результат применения операции: новый объект-множество, которое есть объединением множеств A и B;

– *операция пересечения* $\cap(A, B)$, где A и B – объекты-множества. Результат применения операций: новый объект-множество, который является пересечением множеств A и B;

– *операция вычитания* $D = A \cap B$ при $B \subset A$. Результат применения операции: новый объект-множество со всеми элементами A, которые не входят в B;

– *операция симметричного вычитания* $S = A \setminus B$, где $A \cap B$, и не $A \subset B$, не $B \subset A$. Результат применения операции – это новый объект-множество с множеством элементов, которые принадлежат A или ("или" – распределенное) B.

После выполнения этих операций новые объекты проверяются на их структурную упорядоченность по принципу восходящей технологии.

Таким образом, на базе субъективной оценки исполнителем статуса сущностей в ПрО указанные операции обеспечивают решение следующих задач:

– определение и фиксация структурной упорядоченности объектов;

– расширение описания объекта или конкретизация положения объекта в структуре ПрО;

– определение новых объектов, связанных с заданными;

– формирование объектного графа ПрО.

Принцип вычитания обеспечивает структурную упорядоченность объектов или системы их взаимодействий путем применения отношения принадлежности. Выделение объекта или фиксации его внешних различий осуществляется на основе анализа его структурного типа и определения места в объектном графе ПрО.

Результат проектирования на этом уровне – граф ОМ с установленными

отношениями между объектами с учетом следующих требований:

– множество вершин графа задает взаимно однозначное отображение множества объектов, определенных на обобщенном уровне, и полноту описания ПрО;

– для каждой вершины существует хотя бы одна связь (структурная) со второй вершиной графа;

– существует лишь одна вершина, которая имеет статус множества объекта, отображающего ПрО в целом.

Объекты на данном уровне отличаются друг от друга:

– статусом – множество, элемент или множество-элемент;

– взаимным порядком размещения объектов в графе, который определяется структурными связями и отношением принадлежности;

– взаимным порядком для элементов одного и того же множества в результате сопоставления элементов множества множеству натуральных чисел.

После построения графа ОМ выполняется его модификация (коррекция и расширение), реструктуризация и контроль с использованием приведенных теоретико-множественных операций. Все операции применяют только после завершения проектирования и проведения структурной упорядоченности объектов.

Модификация графа появляется тогда, когда количество объектов ОМ превышает субъективные возможности исполнителей корректного структурного упорядочения объектов. Она касается необходимости доопределения новых объектов в ОМ, которые не выявлены на обобщающем уровне.

Контроль полноты и неизбыточности ОМ ПрО обеспечивает устранение дублирования объектов в одном и том же множестве, а также циклов в объектном графе ПрО.

Характеристический уровень. Основу этого уровня составляет логико-алгебраическая концепция для модификации ОМ путем конкретизации объектов с учетом их свойств и характеристик. На этом уровне используется система предикатов над множеством объектов ПрО и логико-алгебраический математический аппарат для анализа объектов построенной на двух предыдущих уровнях модели ПрО.

На входе уровня объектный граф ПрО и спецификации объектов согласно следующей структуры

<Имя объекта> <Концепт> <Статус> [<Список элементов множества>]

Результат данного уровня – скорректированный и расширенный объектный граф ПрО. Для объектов, определенных на структурно-упорядоченном уровне, формируются их концепты.

Если $O'=(O_1, O_2, \ldots O_n)$ – совокупность объектов ПрО, а $P'=(P_1, P_2, \ldots P_r)$ – множество унарных предикатов, которые связаны со свойствами объектов ПрО, то концепт объекта O_i является множеством утверждений, построенных на основе предикатов из P', которые принимают значение истины для соответствующего объекта. Концепт $Con_i = \{P_{ik}\}$ при условии, что $P_k(O_i) = \text{true}$, где P_{ik} – утверждение для объекта O_i в соответствии с предикатом P_k. Этим правилом определяются свойства и характеристики объектов в рамках определенной системы абстракций для ПрО.

Выражение $\Lambda=(O', P')$ определяет алгебраическую систему концептов O' и предикатов определенной сигнатуры P', предназначенных для анализа сущности

объектов и выявления их особенностей в рамках модели ПрО.

Подмножеству предикатов соответствуют следующие требования:

– назначение предикатов и их количество – достаточно для удовлетворения условий концептуального моделирования ПрО;

– каждый конкретный предикат, его тип и сигнатура должны быть отображением субъективного восприятия сущности объекта исполнителем, который анализирует конкретную ПрО.

Предикаты алгебраической системы такие:

– *0–арные* операции соответствуют константам и определяют утвержденные характеристики ПрО;

– *унарные* операции соответствуют свойствам отдельных объектов;

– *бинарные* операции соответствуют взаимосвязям между отдельными парами объектов.

Объект, который имеет одновременно статус множества и элемента какого-нибудь множества, включает в себя два вида свойств – внешнего и внутреннего.

Совокупность свойств – подмножество множества, выделенных в системе предикатов, удовлетворяет условию, по которому каждый объект принимает значение истины одновременно не более, чем одним предикатом из совокупности, что соответствует характеристике объекта. Формирование множества предикатов для алгебраической системы может быть произвольным. Они вводятся соответственно характеристикам объектов.

Концепция вычитания объектов требует создания специальной системы предикатов для множества объектов. Если существуют несколько таких объектов, то система предикатов неполная, ее необходимо дополнить новыми предикатами (возможно, с помощью операции упорядочения этих объектов).

После определения характеристик и свойств (предикатов) алгебраическая система дополняется установленными исполнителем характеристиками объектов. Их можно расширять и модифицировать в процессе формирования модели ПрО на приведенных уровнях. Характеристики отображают проблематику ПрО, определяют типы свойств объектов, задают структурный статус объектов (элементов и множеств) с комплексом свойств внешнего и внутреннего видов:

– *внешняя характеристика объекта* – характеристика, каждое свойство которой является внешним и определяет все аспекты проблемной ориентации объекта в рамках ОМ ПрО;

– *внутренняя характеристика объекта* – характеристика объекта, которая отображает внутреннее свойство данного объекта и определяет вид объекта как множество.

Характеристики имеют область определения и значения. Область определения – упорядоченное множество, у которого существует взаимно однозначное соответствие между объектом и совокупностью свойств. Значение характеристики объекта – элемент области определения объекта, который отвечает его свойству, входит в состав данной характеристики и имеет значение истины для этого объекта.

Отношения между объектами задают бинарным предикатом, который определен на множестве объектов ПрО и принимает значение истины для рассматриваемой пары объектов.

Типы взаимоотношений включают в себя всевозможные комбинации двух элементов совокупности (множество, элемент множества) с учетом

упорядоченности пары, или первого и второго понятия. Взаимоотношение между объектами – это упорядоченная взаимосвязь объектов, которая возникает вследствие определения отношений между ними, и включают в себя:

1) множество – множество;

2) элемент множества – элемент множества;

3) элемент множества – множество;

4) множество – элемент множества.

Рассмотрим смысл взаимоотношений.

1. Множество-множество включает в себя:

обобщение – взаимоотношение, в соответствии с которыми каждое внутреннее свойство первого множества является внешним свойством второго множества.

специализация – взаимоотношение, обратное предыдущему;

агрегация – взаимоотношение, в соответствии с которым второе множество пары – агрегированное множество, а первое – один из сомножителей декартового произведения;

детализация – вариант взаимоотношений, где первое множество пары – агрегированное множество, а второе – сомножитель декартова произведения;

ассоциация – произвольное взаимоотношение двух объектов, структурная упорядоченность которых отсутствует.

2. Элемент множества – множество включает в себя такие понятия:

классификация – вариант взаимоотношений, отвечающий случаю, когда внешнее свойство соответствующего элемента множества является внутренним свойством множества;

экземпляризация – как вариант взаимоотношений, при котором внутреннее свойство множества является внешним свойством элемента множества. Такой тип имеет: *родовидовое отношение* (IS–A), которое определяет обобщение или специализацию; *часть–целое* (PART–OF), которое определяет агрегацию или классификацию.

Взаимоотношения базируются на структурной упорядоченности между парами объектов. Для полноты картины введем понятие ассоциации, которое достаточно часто используют в практических ОМ (например, "дружеские" функции в языке программирования С++).

Поведенческий уровень. Для объектов, которые определены на характеристическом уровне, концепция поведения обеспечивает рассмотрение характеристик объектов ПрО в зависимости от времени на отрезке оси, соответствующей времени существования данного объекта.

Понятие времени – конкретный параметр системы, значения которого упорядочены и каждый из них отвечает определенному *состоянию* отдельных объектов и системы в целом. Математический аппарат формализованного представления объектов в данной концепции – модель событий и сообщений. Отличительная особенность объектов – разные значения времени. Если объект и его характеристики не изменяются, то он не зависитот от времени и его отличие устанавливается другими концепциями.

Состояние объекта включает в себя:

– *атрибут,* который соответствует характеристике объекта и принимает значение истины при конкретном состоянии этого объекта;

– *статический атрибут* характеризует состояние с одним свойством;

– *динамический атрибут* характеризует состояние из нескольких свойств;

– *значения атрибута* состояния – это значения характеристики объекта, с помощью которого определяется данный атрибут.

Для статического атрибута состояние – постоянное значение, отвечающее существующему свойству. Для динамического атрибута состояние – значение, зависящее от параметра, моделирующего время, и отвечающее свойству объекта при определенном значении этого параметра

Существуют такие виды состояний:

– *состояние объекта* – это совокупность статических и динамических атрибутов состояния конкретного объекта, которое моделирует время;

– *пространство состояний* – это множество всех возможных состояний, в которых может находиться определенный объект модели;

– *жизненный цикл объекта* – это упорядоченная во времени последовательность состояний, которая принадлежит соответствующему пространству состояний и охватывает эволюцию объекта (рождение, существование, уничтожение);

– *диаграмма перехода состояний* – это совокупность всех возможных переходов между парами элементов пространства состояний и вариантами ЖЦ объекта;

– *метод* – это операция, которая обеспечивает переход между состояниями объекта соответственно диаграмме перехода состояний;

– *состояние объектной модели* – это совокупность состояний всех объектов модели при одном и том же значении параметра времени;

– *событие* – это реакция модели объектов, которая приводит к изменению состояния. Событие может включать в себя отправку и получение сообщения, определенное значению параметра времени;

– *сообщение* – это действие, которое предназначено для изменения состояния одного или нескольких объектов;

– *отправка сообщения* – это событие, связанное с необходимостью инициации сообщения;

– *получение сообщения* – это событие, связанное с реакцией объекта на необходимость изменения своего состояния.

Таким образом, соответственно концепции поведения объектов с учетом совокупности их атрибутов и значений определяют последовательности состояний объектов и строят их жизненные циклы, которые отображают в диаграммах переходов состояний. Взаимосвязи между объектами формируют бинарные предикаты, которые связаны со свойствами объектов ПрО, и детализируют уровень взаимосвязей между состояниями объектов. Зависимость от параметра времени достигается внесением в ОМ специального объекта Timer, основное назначение которого состоит в рассылке специальных сообщений текущего времени с определенным уровнем дискретности. Каждый объект имеет соответствующий метод, который анализирует полученное значение и выполняет переход к другому состоянию или оставляет его без изменения.

6.1.2. ФУНКЦИИ И АЛГЕБРА ОБЪЕКТНОГО АНАЛИЗА

К формальному аппарату объектного анализа отнесем множество базовых функций, которые обеспечивают декомпозиционные и композиционные изменения

денотатов и концептов объектов при концептуальном моделировании ПрО, расширение или сужение концептов отдельных объектов, а также увеличение или уменьшение количества объектов (детализация, экземпляризация, агрегация и др.). Такие изменения соответствуют определенным правилам и условиям, которые обеспечивают корректность выполнения функций, а также отражают приведенный многоуровневый подход в объектном анализе [64].

Рассмотрим следующие функции объектного анализа.

Декомпозиционные изменения денотата:

– соответствующая сущность действительной реальности, которая соответствует определенному объекту, подается как совокупность однородных предметов;

– соответствующая сущность действительной реальности, которая соответствует определенному объекту, подается как совокупность неоднородных предметов.

Композиционные изменения денотата:

– денотаты нескольких однородных предметов подаются в составе определенной сущности из выбранной предметной области;

– денотаты нескольких неоднородных предметов подаются в составе определенной сущности из выбранной предметной области.

Изменения концептов, связанных с декомпозиционным изменением денотатов:

– концепты новых детализированных объектов формируются на основе концепта начального объекта;

– концепты новых детализированных объектов формируются без учета концепта начального объекта.

Изменения концептов, связанных с композиционным изменениям денотатов:

– концепт нового композиционного объекта формируется на основе одинаковых концептов начальных объектов;

– концепт нового композиционного объекта формируется на основе разных концептов начальных объектов.

Изменения уровня детализации (абстракции) концептов:

– из концепта объекта исключается одно или несколько свойств или характеристик при формировании концепта нового объекта с одинаковым денотатом;

– в концепт объекта включается одно или несколько свойств или характеристик при формировании концепта нового объекта с одинаковым денотатом.

Алгебра объектного анализа

Пусть $O=(O_1, O_2,, \ldots, O_n)$ – множество объектов на определенном уровне объектного анализа и $O_i = O_i\,(Name_i, Den_i, Con_i)$,

где $Name_i$, Den_i, Con_i – знак (имя), денотат и концепт соответственно, $P = (P_1, P_2, \ldots, P_r)$ – множество предикатов, на основе которых определяют концепты объектов $Con_i = (P_{i1}, P_{i2}, \ldots, P_{is})$.

Базовые операции объектного анализа такие:

1. Декомпозиционное изменение денотата для формирования новых однородных объектов:

$$decds\ (O_i): O_i \rightarrow (O_{i1}, \ldots O_{ik}),$$

где $O_{ij} = O_{ij}(Name_{ij}, Den_{ij}, Con_{ij})$; $\forall j\ Con_{ij} = Con_i$; $Den_i = Den_{i1} \cup \ldots \cup Den_{ik}$.

2. Декомпозиционное изменение денотата для формирования новых неоднородных объектов:

$$decdn\ (O_i)\colon O_i \to (O_{i1}, \ldots O_{ik}),$$

где $O_{ij} = O_{ij}\ (Name_{ij}, Den_{ij}, Con_{ij});\ \forall j\ Con_{ij} = \varnothing;\ Den_i = Den_{i1} \cup \ldots \cup Den_{ik}$.

3. Композиционное изменение денотатов однородных объектов для формирования нового объекта:

$$comds\ (O_{i1}, \ldots, O_{ik})\colon (O_{i1}, \ldots O_{ik}) \to O_i,$$

где $O_i = O_i(Name_i, Den_i, Con_i);\ \forall j\ Con_i = Con_{ij};\ Den_{i1} \cup \ldots \cup Den_{ik} = Den_i$.

4. Композиционное изменение денотатов неоднородных объектов для формирования нового объекта:

$$comdn\ (O_{i1}, \ldots, O_{ik})\colon (O_{i1}, \ldots, O_{ik}) \to O_i,$$

где $O_i = O_i(Name_i, Den_i, Con_i);\ Con_i = \varnothing;\ Den_{i1} \cup \ldots \cup Den_{ik} = Den_i$.

5. Расширение концепта существующего объекта. Если предикат $P_t \in P$, $P_t \notin Con_i$ и $P_t(O_i)$ принимает значение истины, то

$$conexp\ (O_i, P_t)\colon O_i \to O_i',$$

где $O_i' = O_i'\ (Name_i, Den_i, Con_i`);\ Con_i \cup \{P_t\} = Con_i'$.

6. Сужение концепта существующего объекта. Если предикат $P_t \in Con_i$, то

$$connar\ (O_i, P_t)\colon O_i \to O_i',$$

где $O_i' = O_i'\ (Name_i, Den_i, Con_i');\ Con_i' = Con_i \setminus P_t$.

Это множество функций определяет алгебру объектного анализа $\Sigma = (O', \Psi)$, где $O' = (O_1, O_2, \ldots, O_n)$ – множество объектов,

$\Psi = \{decds, decdn, comds, comdn, conexp, connar\}$ – множество операций над элементами O'.

Каждая из операций имеет определенный приоритет и арность, а также связана с соответствующими допустимыми изменениями денотатов и концептов.

Переход от функций объектного анализа к операциям алгебры объектного анализа показан следующей теоремой.

Теорема 6.1. *Множество операций Ψ алгебры Σ – полная система операций относительно функций объектного анализа.*

Правила объектного анализа. Концептуальное моделирование определенной ПрО имеет итеративный характер и вытекает из определения самой ПрО как начального объекта. Сначала в моделировании применяют функции объектного анализа, которые приближают структуру и свойство ОМ к конечным целям. Каждая из функций рассматривается как последовательность выполнения операций алгебры объектного анализа, что поддерживает целостность представления ОМ. Процесс завершается формализованным описанием сущностей и модели ПрО с учетом каждого аспекта абстрагирования и применения соответствующего математического аппарата.

Рассмотрим основные правила объектного анализа:

– объектный анализ выполняется при условии минимизации потери информации из описания действительной реальности для выбранной ПрО;

– все изменения, которые происходят с объектной моделью, отвечают процессам детализации описания ПрО и определяются в рамках представления множества объектов, как совокупности треугольников Фреге;

– каждый шаг объектного анализа определяется изменениями денотата или концепта одного или нескольких объектов ОМ;

– новые объекты на определенном шаге объектного анализа определяют соответственно изменения денотатов существующих объектов;

– все изменения, которые происходят с объектной моделью, отвечают условиям существования и определения формальных уровней абстракции представления объектов;

– функции объектного анализа определяют преобразования в соответствии с допустимыми изменениями ОМ и ее отдельных элементов;

– каждый шаг объектного анализа обеспечивается условиями целостности ОМ.

Приведенные формализмы предназначены для определения и приведения в порядок базовой терминологии ООП, суть которой состоит в последовательной дефиниции терминов в соответствии с построением отношений между понятиями. Эти дефиниции выполняются на уровнях с постоянным уточнением и развитием характеристик и свойств объекта. Так, на обобщающем уровне определяется основной термин – объект. Структурно-упорядоченный уровень обеспечивает детализацию для объекта таких понятий, как класс, экземпляр класса, абстрактный класс и др. На характеристическим уровне дается определение следующих понятий: свойство объекта, отношения между объектами, агрегация; детализация, классификация, экземпляризация, ассоциация, характеристика объекта и т.п. На поведенческом уровне определяются понятия, которые связаны с состоянием объекта, а именно: атрибут состояния, статический или динамический атрибут состояния, пространство состояний, метод и другие.

Представление объектной модели в новых формализмах. Используя приведенные выше дефиниции объекта и его свойства, дадим формальное определение объектной модели ПС в следующем виде:

$$OMS = (OClass, G),$$

где $OClass = \{OClass^i\}$ – множество классов, G – объектный граф, в котором вершины – классы, а дуги задают отношения объектов.

Каждый класс из $OClass$ представляется как модель

$$OClass^i = (ClassName^i, Method^i, Field^i),$$

в которой $ClassName^i$ – имя класса; $Method^i = \{Method_j^i\}$ – множество методов класса; $Field^i = \{Field_n^i\}$ – множество переменных, которые определяют состояние экземпляров класса.

Пусть $PField^i \subset Field^i$ – множество внешних переменных (public) класса.

Каждому $PField_n^i \in PField^i$ поставим в соответствие методы $get<PField_n^i>$ и $set<PField_n^i>$ для доступа и модификации значений переменных, т.е. эти переменные являются атрибутами.

Новое множество методов имеет вид

$$IMethod^i = Method^i \cup \{get<PField_n^i>\} \cup \{set<PField_n^i>\}.$$

Каждому множеству $IMethod^i$ ставит в соответствие определенный интерфейс $IFunc^i$, состоящий из прототипов методов $IMethod^i$, реализация которых обеспечивает функциональность методов класса и их атрибуты.

Интерфейсная модель имеет следующий вид: $ISyst = (IFunc, IG)$,

где $IFunc = \{IFunc^i\}$ – множество интерфейсов для классов $OClass$; IG – интерфейсный граф, эквивалентный графу G. В IG вершины – это интерфейсы, а дуги – отношения между компонентами соответствующие отношениям между интерфейсами.

Таким образом, между графами G объектной модели и IG компонентной

модели существует изоморфное отображение, а функциональность реализаций для интерфейсов $IFunc^i$ эквивалентна функциональности класса $OClass^i$. Для классов *OClass* определяются условия, которые способствует формированию их представления в виде элементов множества интерфейсов в интерфейсном графе.

Определение 6.1. Декларированная в классе переменная называется управляемой по доступу к ее значению со стороны других классов, если она является public-переменной или для нее реализован доступ с помощью public-методов класса.

Если внешнее взаимодействие с классом происходит с помощью public-методов и управляемых переменных, то для него реализуется интерфейсный принцип доступа. Это означает, что не существует другого способа внешнего доступа к функциональности класса или его переменным.

Теорема 6.2. *Для каждой ОМ внешнее взаимодействие с классами которой происходит на основе public-методов и управляемых переменных, образуется единое интерфейсное представление ISyst с эквивалентной функциональностью.*

Эта теорема определяет условия существования единого эквивалентного отображения между объектным и интерфейсным представлениями компонентов.

6.2. ТЕОРИЯ КОМПОНЕНТНОГО ПРОГРАММИРОВАНИЯ

В связи с широким практическим использованием компонентного программирования далее рассмотрим основания компонентного программирования, а, именно, модели, формализмы представления компонентов, интерфейсы объектов, внешнюю и внутреннюю компонентные алгебры, операции которых предназначены для манипулирования компонентами среды, самой средой, а также для изменения, реконструирования и перепрограммирования наследуемых программных компонентов и КПИ [64–67].

Приведена обновленная парадигма сборки компонентов, базисом которой являются алгебраические системы преобразования типов данных и методы композиции компонентов в разные программные структуры.

Проведен анализ стандарта ГОСТ 3901–99, ориентированного на представление языков описания типов данных для всех действующих ЯП и использование для системного преобразования переданных другому компоненту параметров, данные которых могут отличаться от тех, что не входят в этот язык. Рассмотрены современные подходы к решению проблем взаимодействия разноязыковых компонентов и их реализация в общесистемных средах (CORBA, COM, JAVA).

Концепция компонента связана с фундаментальной идеей сборки, которая развивается соответственно уже сформированным тенденциям программирования. Усовершенствование, отождествление этих понятий и пополнение новыми для создания целостной теоретической базы компонентного программирования – суть этой главы.

Под **компонентным программированием** понимается метод создания ПС, базирующийся на применении концепций сборки (композиции) на всех процессах ЖЦ, где базовым элементом сборки является компонент.

Сборка компонентов – это процесс объединения отдельных компонентов в более сложную компонентную структуру, в которой принимают участие программные компоненты, каркасы, контейнеры и т.п. Сборка задается описанием связей (ассоциаций) между компонентными элементами, а также утверждениями о

компонентах и их связях. Сборку представляют теориями первого порядка, а правильность сборки зависит от языка спецификации компонентов. Представление сложного патерна включает в себя предикаты описания классов, состояний переменных, методов и их отношений. Класс и объект, а также патерн создают сорта объектов первого класса, могут также использоваться сорта bool и int.

Обеспечение условий повторного применения компонента – одно из основных целей процессов компонентной разработки. Кроме унификации и стандартизации архитектурных, структурных, технологических характеристик компонента, важную роль играет определение его функциональности и методов доступа. Это, в свою очередь, требует более детального подхода к определению спецификации компонента для определенной ПрО и задач, которые решаются ею, конкретных функций обработки данных и т.п. Сложность такого подхода состоит в том, что для компонента не существует заведомо определенных функциональных требований, как в случае спецификации компонентов для конкретной целевой системы. Поэтому необходимым условием разработки КПИ является существование определенных объективных концепций, принципов, критериев выбора и спецификации их функциональных свойств.

6.2.1. ОСНОВНЫЕ ПОНЯТИЯ КОМПОНЕНТНОГО ПРОГРАММИРОВАНИЯ

Компонентное программирование – это разновидность сборочного программирования, где роль элемента сборки играет программный компонент, далее – *компонент.* Сущность такого программирования определяют как процесс создания ПС из базовых объектов ОМ, целевых компонентов и КПИ. В соответствии с жизненным циклом построения таких систем согласно объектно-компонентному методу ОКМ проектирование новых систем выполняют как совокупность взаимодействующих объектов, компонентов и КПИ в рамках компонентной среды.

Соответственно этому методу компонентное проектирование связано с построением отдельных компонентов в рамках определенной ОМ ПрО. Для обеспечения перехода от объектов ОМ к компонентному представлению объектов установлена связь между объектно-ориентированным анализом и компонентным проектированием ПС. Для формализации процесса компонентного проектирования в рамках ОКМ разработан соответствующий формальный аппарат с базовой терминологией элементов (компонент, каркас, компонентная модель, компонентная среда и т.п.), их модели и компонентная алгебра.

Определения 6.2. *Компонент* – это независимый от ЯП, самостоятельно реализованный программный объект, который обеспечивает выполнение определенной совокупности прикладных сервисов, доступ к которым возможен только с помощью интерфейсов, определяющие функциональные возможности компонента и порядок обращения к его операциям.

В данном представлении компонент – отображение типового решения для некоторого фрагмента ПрО и имеет типовую архитектуру, структуру, характеристики и атрибуты интерфейсной части для обмена данными в компонентной среде. Иными словами компонент, представленный таким образом, становится неделимым и инкапсулированным программным элементом, который удовлетворяет функциональным требованиям, а также требованиям относительно архитектуры системы и компонентной среды.

Определения 6.3. *Компонентная программа* – это совокупность компонентов, которые необходимы для обеспечения функциональных и нефункциональных требований. Она строится и функционирует в соответствии с правилами определения компонентных конфигураций и компонентного взаимодействия в рамках объектных и компонентных моделей.

Обобщение компонентной архитектуры. Современная архитектура компонентной среды – один из основных источников идей и подходов компонентного программирования. Архитектура – расширение классической модели "клиент–сервер" с учетом специфики построения и функционирования компонентов, а также результатов практических реализаций и их апробаций.

Основа компонентной среды – множество серверов компонентов (или серверов приложений – application servers). Внутри сервера разворачиваются компоненты, представленные как контейнеры. Для каждого сервера может существовать произвольное количество контейнеров.

Контейнер – это оболочка, внутри которой реализуются функции компонентва. Взаимосвязь и взаимодействие контейнера с сервером строго регламентированы и осуществляются через стандартизированные интерфейсы. Контейнер управляет порождаемыми им экземплярами компонента, реализациями соответствующей функциональности. В общем случае внутри него может существовать произвольное число экземпляров-реализаций, каждая из которых имеет уникальный идентификатор.

С каждым контейнером связано два типа интерфейсов для взаимодействия с другими компонентами и интерфейс системных сервисов, необходимых для функционирования самого контейнера и реализации специальных функций, например, поддержка распределенных транзакций, в которых принимают участие несколько компонентов.

Первый тип интерфейса (в некоторых архитектурах называется Home interface) обеспечивает управленит экземплярами компонента с обязательными реализациями методов поиска, создания и удаления отдельных экземпляров.

Ко второму типу относятся интерфейсы, которые обеспечивают доступ к реализации функциональности компонента. Фактически с каждым экземпляром связан свой функциональный интерфейс.

Классы компонентов. Исходя из принципов выполнения систем компоненты можно разделить на три класса:

– компоненты развертывания (Deployment components), которые служат основой для формирования программы с развертыванием в компьютерной среде (библиотеки, фрагменты готовое для выполнения, файлы конфигурации, базы данных и др.);

– компоненты как рабочие продукты (Work product components), которые являются по сути промежуточными и вспомогательными элементами процесса программирования (файлы текстов с компонентами и данными, элементы с описаниями архитектуры, правилами генерации конечного кода и др.);

– компоненты среды выполнения (Execution components), которые создаются в процессе выполнения программы (временные программные объекты, файлы, таблицы БД и др.).

Кроме этого разделения, каждый класс имеет несколько других видов, а именно: программы, библиотеки, файлы, таблицы баз данных, документы и т.п.

Требования к компонентам. Основные требования такие**:**

1. Компонент имеет один или больше интерфейсов, которые имеют уникальные имена и описания, определяющие структуру интерфейсов. Компонент обязательно должен иметь хотя бы один интерфейс, так как компоненты используются с помощью их интерфейсов.

2. Каждый интерфейс состоит из определенного количества операций (методов). Каждая операция имеет собственный идентификатор, множество входных и выходных параметров, каждый из которых имеет тип из заведомо определенного множества простых и структурных типов данных. Совокупность параметров (с учетом их приведения в порядок) и их типов определяет сигнатуру операции, а совокупность сигнатур всех операций определяет сигнатуру соответствующего интерфейса. Каждый интерфейс должен иметь по крайней мере одну операцию.

3. Каждый компонент относительно схемы взаимодействия может быть как сервером, так и клиентом. Если к нему обращаются другие компоненты, то он – сервер. Для поддержки компонентом определенного сервиса в некоторых случаях необходимо обращаться к другим объектам, например, к компоненту, который отвечает за сохранение данных. В этом случае компонент – клиент. Последовательность их взаимодействий может быть сложной.

4. Совокупность интерфейсов можно разделить на два типа. К первому типу относятся интерфейсы, которые непосредственно описывают функциональность компонента, т.е. его сервисные возможности. Ко второму типу относятся ссылка на интерфейсы других компонентов, операции которых необходимо выполнить для обеспечения нормального функционирования компонента.

5. Все возможные взаимодействия между компонентами определяются на основе их интерфейсов. Такое требование запрещает существование скрытых (hidden) взаимодействий. Объекты, между которыми могут быть скрытые взаимодействия должны входить в состав одного компонента и рассматриваться как единое целое. Иными словами, компоненты поддерживают инкапсуляцию и их интерфейсы являются открытыми (public).

6. Информация, которая приведена в открытых интерфейсах, должна быть полной для описания внешнего поведения компонента и исключать неопределенность в при его применении. Все внутренние характеристики и особенно защита компонента в середине и извне – недоступны.

Общие свойства компонентов. Компоненты, как определенный тип программных объектов, имеют характерные и общие свойства, присущие всему множеству компонентов. Компонент имеет такие дополнительные свойства:

1. *Компонент - инкапсулированный* (encapsulated) объект. Инкапсуляция – процесс упрятывания (закрытости) реализации и кода, которые обеспечивают выполнение компонентом своих функций. Инкапсуляция компонента означает, что весь код закрытый, доступ к нему происходит с помощью специальных интерфейсов.

2. *Компонент - самоопределенный* (descriptive) объект. Пользователь компонента должен четко понимать как его назначение, так и особенности функционирования. Этого можно достичь лишь при наличии полной и всесторонней информации о нем. Соответственно структура компонента должна описывать три аспекта – интерфейсы, реализацию, развертывание.

3. *Компонент – взаимозаменяющий* (replaceable) объект. Это свойство является

следствием задания инкапсуляции и самоопределения. Первое свойство разрешает пользователю не учитывать внутренние особенности компонента, а второе – определяет его поведение, связи с другими компонентами и требования к внешней среде. В этом случае любой компонент с необходимой функциональностью и соответствующими интерфейсами может быть использован не только при построении программы, и при сопровождении и развитии (например, новый компонентов более качественно реализует функциональность или требует меньше ресурсов).

4. *Компонент – расширяющий объект* (extensible). Это свойство обуславливается необходимостью развивать и усовершенствовать компонент в целях улучшения существующих показателей и добавления новых функций.

Модели компонентного программирования. Формальные модели компонентного программирования обеспечивают простой переход от теории к практике с применением архитектуры и моделей современных систем CORBA, COM, EJB и реализацию базиса формальных методов построения компонентных программ и систем. Такие задачи не реализуются в рамках одной определенной модели. Предлагается семейство формальных моделей и других абстрактных структур с единой понятийной, терминологической и математической базой:

– модель компонента;

– модель компонентной среды;

– внешняя компонентная алгебра;

– модели сборки (композиции) компонентов.

Эти модели – типичны и ориентированы на применение их при теоретических исследованиях и построении элементов теории компонентного программирования.

6.2.2. МОДЕЛЬ КОМПОНЕНТА И ИНТЕРФЕЙСА

Под *моделью компонента* понимают абстрактное представление, которое отражает основные свойства, характеристики, типичную структуру, а также способность к взаимодействию с другими элементами компонентной среды.

Модель произвольного компонента в общем случае имеет вид

$$Comp = (CName, CInt, CFact, CImp, CServ), \qquad (6.3)$$

где *CName* – уникальное имя компонента; $CInt = \{CInt^i\}$ – множество интерфейсов, связанных с компонентом; *CFact* – интерфейс управления экземплярами компонента; $CImp = \{CImp^j\}$ – множество реализаций компонента; $CServ = \{CServ^r\}$ – интерфейс, который определяет множество системных сервисов, необходимых для поддержки функционирования компонента и взаимодействия с компонентной средой.

Суть данной концепции модели сводится к следующим аксиомам:

1) операции алгебры определяются над элементами данной модели и только на них;

2) операций алгебры обеспечивают необходимые условия целостности компонента.

Фактически это означает, что сложные модели (6.3) определяют основное множество элементов компонентной алгебры.

Имя компонента уникально в любом пространстве имен для компонентной среды. Для разрешения коллизий применяют метод квалификационных признаков (глава 5). Фактически это означает, что каждое имя входит в некоторое

пространство имен. Управление такими пространствами обеспечивается специальными алгоритмами их построения в виде иерархического дерева, где каждая вершина однозначно определяет маршрут (последовательность вершин) от корня дерева.

Компоненты определяют на множестве интерфейсов $CInt = CInt_1 \cup CInt_2$ двух типов. К первому типу (множество $CInt_1$) относят интерфейсы, которые реализуются в среде компонента и имеют соответствующие реализации методов. Ко второму типу (множество $CInt_2$) относят интерфейсы, реализованные в других компонентах, но функциональность которых нужна для выполнения методов данного компонента.

Модель интерфейса. Каждый *i-интерфейс* компонента имеет вид

$$CInt^i = (IntName^i , IntFunc^i , IntSpec^i) \quad (6.4)$$

где $IntName^i$ – имя интерфейса; $IntFunc^i$ – функциональность, реализованная данным интерфейсом (совокупность методов); $CInt^i$ – интерфейс управления экземплярами компонента; $IntSpec^i$ – спецификация интерфейса (описания типов, констант, других элементов данных, сигнатур методов и т.д.).

Необходимым требованием существования компонента является условие его целостности:

$$\forall CInt^i \in CInt \; \exists CImp^j \in CImp \; [Provide(CInt^i) \subseteq CImp^j], \quad (6.5)$$

где $Provide(CInt^i)$ означает функциональность, которая обеспечивает реализацию методов интерфейса $CInt^i$.

Наличие знака включения в данной формуле означает, что избранная реализация компонента может обеспечить поддержку не только необходимого интерфейса, но и других. Например, для этого практические технологии и ЯП (CORBA, Java, C++ и др.) содержат необходимые средства. Для каждого из таких интерфейсов может существовать несколько реализаций, которые различаются особенностями функционирования (например, операционной средой, средствами сохранения данных и т.д.).

Взаимодействие двух компонентов $Comp_1$ и $Comp_2$ определяется следующим необходимым условием: если $CInt^i{}_1 \in CInt_1$, то должен существовать $CInt^k{}_2 \in CInt_2$ такой, что

$$Sign (CInt^i{}_1) = Sign (CInt^k{}_2) \; \& \; Provide(CInt^i{}_1) \subseteq CImp^j{}_2,$$

где $Sign(...)$ означает сигнатуру соответствующего интерфейса.

Пары компонентов компонентной модели определенного типа (*CFact и CServ* – фиксированные) при сопоставлении имеют следующие свойства.

Определение 6.4. Два компонента $Comp_1$ и $Comp_2$ *тождественны* (равны), если тождественный их соответствующий состав. Как следствие, замена $Comp_1$ на $Comp_2$ не влияет на компонентную программу, которой принадлежит $Comp_1$.

Определение 6.5. Два компонента $Comp_1$ и $Comp_2$ *эквивалентны*, если тождественны их множества интерфейсов и реализаций. Замена $Comp_1$ на $Comp_2$ не меняет функциональность компонентной программы при условии установки соответствия между именами существующих и новых компонентов.

Определение 6.6. Два компонента $Comp_1$ и $Comp_2$ *подобны*, если тождественны их множества интерфейсов. Замена $Comp_1$ на $Comp_2$ сохраняет взаимосвязи компонентов, но функциональность компонентной программы может измениться.

Интерфейс $CInt^i$ из формулы (6.4) определяет необходимые методы управления экземплярами компонента, к которым относятся:

1) поиск и определение необходимого экземпляра компонента – *Locate*;

2) создание экземпляра компонента – *Create*;

3) удаление экземпляра компонента – *Remove*.

Эти методы составляют основу для любых интерфейсов управления экземплярами в рамках любых компонентных моделей.

В наиболее общем случае операции управления компонентами такие:

$$CInt^i = \{Locate, Create, Remove\}.$$

Каждая реализация компонента описывается так:

$$CImp^j = (ImpName^j, ImpFunc^j, ImpSpec^j), \quad (6.6)$$

где $ImpName^j$ – идентификатор имени реализации компонента; $ImpFunc^j$ – функциональность, соответствующая данной реализации (совокупность реализаций методов); $ImpSpec^j$ – спецификация реализации (описание условий выполнения, параметров настройки реализации и т.д.).

Реализация компонента – это совокупность методов определенной сигнатуры и типы данных для переданных параметров или параметров, которые возвращаются. По этим сигнатурам и типам данных сопоставляют реализации и интерфейсы в виде описания собственных методов, обеспечивающих процесс связывания. В отличие от объектно-ориентированного и других подходов связывание компонентном программировании происходит позднее, на заключительных процессах построения компонентной программы, а иногда и во время ее выполнения программы (например, это характерно для динамических интерфейсов в системе CORBA [97]).

Связь компонентной модели с объектной. Связь с ОМ прослеживается с помощью следующих суждений.

В процессе своего функционирования компонент с помощью метода *Create* из интерфейса *Cfact* порождает экземпляры:

$$Cfact.Create: Comp \rightarrow \{Cins_k^{ij}\},$$
$$Cins_k^{ij} = (Iins_k^{ij}, IntFunc^i, ImpFunc^j),$$

где $Cins_k^{ij}$ – экземпляр k компонента, который предоставляет свою функциональность с помощью интерфейса $IntFunc^i$ и обеспечивает реализацию этого интерфейса с помощью реализации $ImpFunc^j$; $Iins_k^{ij}$ – уникальный идентификатор экземпляра компонента.

Пусть существует некоторая объектная система, представленная в виде диаграммы классов: $Osyst = (Oclass, G)$,

где $Oclass = \{Oclass^i\}$ – множество классов; G - объектный граф, отражающий связи и отношения между классами и экземплярами.

Каждый класс представлен в виде: $Oclass^i = (ClassName^i, Method^i, Field^i\}$,

где $ClassName^i$ – имя класса; $Method^i = \{Method_j^i\}$ – множество методов; $Field^i = \{Field_n^i\}$ – множество переменных, определяющих состояние экземпляров класса.

Пусть $Pfield^i \subset Field^i$ – множество внешних переменных (public), которые доступные извне. Каждому $Pfield_n^i \in Pfield^i$ ставятся в соответствие методы $get<Pfield_n^i>$ и $set<Pfield_n^i>$ для присвоения и выборки значений соответствующей переменной, т.е. эти переменные становятся атрибутами в современных компонентных моделях

Соответственно в других классах вместо непосредственного обращения к таким переменным будут использоваться указанные методы.

Введем новое множество методов:

$$Imethod^i = Method^i \cup \{get<Pfield_n^i>\} \cup \{set<Pfield_n^i>\} ,$$

которым сопоставим интерфейс $Ifunc^i$, состоящий из прототипов методов, которые входят в $Imethod^i$.

Параллельно с *Osyst* будем рассматривать систему *Isyst = (Ifunc, IG)* ,

где *Ifunc = {Ifunc^i}* – множество интерфейсов; *IG* – интерфейсный граф, идентиченый графу *G*.

Класс $Oclass^i$ порождает свои экземпляры (объекты)

$$Obj_k^i = \{ObjName_k^i, Method^i, Field^i\},$$

которым в системе *Isyst* будут отвечать интерфейсные элементы $Iobj_k^i=\{Iname_k^i, Ifunc^i\}$.

Для каждого такого элемента не определена реализация соответствующего интерфейса. Сопоставив некоторому интерфейсу реализацию $ImpFunc^j$ (которая обеспечивает выполнения методов интерфейса), формируем элемент

$$Iobj_k^{ij} = \{Iname_k^i, Ifunc^i, ImpFunc^j\},$$

который, по своей сути, эквивалентен экземпляру компонента

$$Cins_k^{ij} = (Iins_k^{ij}, IntFunc^i, ImpFunc^j).$$

Основные расхождения определяют следующие факторы. Во-первых, выбор пригодной реализации происходит на процессе развертывания, а не на ранних процессах, как это нужно для ООП. Во-вторых, экземпляр объектного класса порождается его описанием и не может содержать элементов больше, чем существует в самом классе или в его суперклассах. Противоположно этому, реализация компонента может поддерживать несколько не связанных между собою интерфейсов. Эти две системы (объектная и интерфейсная) эквивалентны.

Таким образом, при построении компонентной программы применяют ООП и соответствующие инструментальные средства. Используя объектно-ориентированные методы проектирования, параллельно создают объектную систему и соответствующую ей интерфейсную систему без конкретизации реализации этих интерфейсов. Результат такого проектирования – совокупность интерфейсов, для которой рассматривается задача покрытия интерфейсов соответствующими компонентными реализациями. При этом нет необходимости учитывать реализацию функциональности ПС, которая выполняется компонентами путем интеграции (сборки) и развертывания компонентной ПС.

Типы отношений между компонентами. Пусть выражение вида $Comp_n=(Cname_n,Cint_n,Cfact_n,Cimp_n,Cserv)$ определяет перечень конкретных компонентов, которые имеют следующие типы отношений.

Отношение наследования двух компонентов определяется установлением отношения наследования для их входных интерфейсов (аналогично отношению наследования в ООП).

Отношение экземпляризации. Экземпляры компонента создаются соответственно определенному входному интерфейсу $CIntI^i$.

Выражение $CIns_k^{ij} = (IIns_k^{ij}, IntFunc^i, ImpFunc^j)$ описывает экземпляр компонента *Comp*. Здесь $IIns_k^{ij}$ – уникальный идентификатор экземпляра, $IntFunc^i$ – функциональность интерфейса $CIntI^i \in CInt$, $ImpFunc^j$ – программный элемент, который обеспечивает реализацию $CImp^i \in CImp$**.**

Отношение контракта. Контракт между компонентами $Comp_1$ и $Comp_2$ описывают выражением: $Cont_{12}^{im} = (CInt_1^u, CInt_2^m, IMap_{12}^{im})$,

где $CInt_1^u \in CInt_1$ – исходный интерфейс первого компонента, $CInt_2^m \in CInt_2$ – входной интерфейс второго компонента; $IMap_{12}^{im}$ – отображение соответствия между методами, входящими в состав обоих интерфейсов с учетом сигнатур и типов данных, которые передаются. Отношения контракта существует, если компонент $Comp_2$ имеет реализацию интерфейса $CInt_2^m$, выполняющего функциональность $IntFunc_1$ и интерфейс $CInt_1^n$.

Отношение связывания. Если между компонентами $Comp_1$ и $Comp_2$ существует отношения контракта $Cont_{12}^{im}$, то между их экземплярами $CIns_{1k}^{ij} = (IIns_{1k}^{ij}, IntFunc^i, ImpFuncj)$ и $CIns_{2p}^{mq} = (IIns_{2p}^{mq}, IntFunc^m, ImpFunc^q)$ существует отношение связывания относительно контракта $Cont_{12}^{im}$, которое описывается выражением $Bind\ (IIns_{1k}^{ij}, IIns_{2p}^{mq}, Cont_{12}^{im})$.

6.2.3. МОДЕЛЬ КОМПОНЕНТНОЙ СРЕДЫ

Компонентная среда *CE* (Component Enviroment) определяется как множество элементов

$$CE = (NameSpace, IntRep, ImpRep, CServ, CServImp), \qquad (6.7)$$

где $NameSpace = \{CName^m\}$ – пространство имен, представляющее собой множество имен компонентов в среде; $IntRep = \{IntRep^i\}$ – репозитарий интерфейсов, который содержит интерфейсы компонентов среды; $ImpRep = \{ImpRep^j\}$ – репозитарий реализаций, который содержит реализации для компонентов среды; $CServ = \{CServ^r\}$ – интерфейс, который определяет множество системных сервисов, необходимых для поддержки функционирования компонента; $CServImp = \{CServImp^r\}$ – множество реализаций для системных сервисов.

Из определения этой модели вытекает, что ее необходимо согласовывать с обобщенной архитектурой компонентной среды. Элемент репозитария интерфейсов определяется как пара:

$$IntRep^i = (CInt^i, CName^m),$$

где $CInt^i$ – интерфейс для определенного компонента; $CName^m$ – имя компонента, который реализует этот интерфейс.

Элемент репозитария реализации определяется как пара:

$$ImpRep = (CImp^j, CName^m),$$

где $CImp^j$ – реализация для определенного компонента; $CName^m$ – имя компонента, который содержит эту реализацию.

Введем понятие нейтрального (нулевого) элемента компонентной среды, называемой каркасом (framework)

$$FW = (\varnothing, \varnothing, \varnothing, CServ, CServImp),$$

где пространство имен, репозитарии интерфейсов и реализаций компонентов – пустые множества.

Элементы модели *CServ* и *CServImp* определяют конкретный тип компонентной среды. Совместимость компонентных сред разных типов определяется наличием отображения между соответствующими интерфейсами системных сервисов.

К множеству системных сервисов $CServ = \{CServ^r\}$ принадлежат сервисы, которые необходимы при организации построения и функционирования компонентных сред, а также управления компонентными конфигурациями. В частности, минимально необходимый набор сервисов:

1. Сервис наименования Naming, обеспечивающий возможность поиска компонентов в распределенной среде с учетом пространств имен.

2. Сервис связывания – Binding, предназначен для определения (связывания) соответствия “имя–объект” и применяется к найденным компонентам.

3. Сервис транзакций – Transaction, обеспечивает организацию и управление функционированием совокупности компонентов.

4. Сервис сообщения – Messaging, необходим для организации обмена информацией между компонентами и является сложным элементом в модели асинхронных транзакций.

В реальных средах могут быть реализованый и другие системные сервисы, например, сервис управления событиями, служба каталогов и т.д. Много таких сервисов определяют, в основном, условия упрощения организации компонентных сред и могут быть созданы, как базы других сервисов. В дальнейшем будем считать, что перечисленные выше четыре сервиса будут обязательными для любой компонентной модели и ее реализации. Они отражают реализацию базовых функций управления компонентными средами:

– поиск компонентов;

– доступ к их ресурсам;

– организация обмена информацией между компонентами;

– динамическое управление функционированием заданной совокупности компонентов (каркасов).

Определение 6.7. Каркасом компонентной среды называют среду, для которой совокупности имен компонентов, интерфейсов и реализаций – пустые множества, т.е. $FW = (\varnothing, \varnothing, \varnothing, CServ, CServImp)$.

Пусть $FW_1 = (\varnothing, \varnothing, \varnothing, CServ_1, CServImp_1)$ и $FW_2 = (\varnothing, \varnothing, \varnothing, CServ_2, CServImp_2)$ – два каркаса. разных типов. Соответственно этому, каждый тип компонентной среды имеет только один каркас.

Если существует отображения *SMap*:

$CServ_1 \rightarrow CServ_2$ такое, что $SMap(CServ_1) \subseteq CServ_2$, то этот каркас FW_1 объединяется с каркасом FW_2.

Определение 6.8. Каркас FW_1 совместимый с каркасом FW_2, если существует отображение $SMap: CServ_1 \rightarrow CServ_2$ такое, что $SMap(CServ_1) \subseteq CServ_2$.

Для совместимости каркаса FW_2 с каркасом FW_1 необходимо существование обратного отображения с аналогичными свойствами. Каждая компонентная среда определенного типа включает в себя соответствующий каркас. Тогда совместимость каркасов будет определять и совместимость компонентных сред рассмотренных типов.

Практически совместимость каркасов и компонентных сред означает следующее. Пусть в первую среду $CServ_1$ входит сервис именования. Тогда и в $CServ_2$ тоже должен входить аналогичный сервис. Кроме того, между этими сервисами должна существовать такая связь, при которой сервис первой среды должен понимать именование объектов из среды и обратно. Аналогичная ситуация должна существовать и для других системных сервисов, так как любая компонентная среда определенного типа использует те же сервисы, и их связи будут определять совместимость.

В частности, взаимодействие компонентов, расположенных в разных серверах, поддерживают сервисы этих серверов. Если совместимость между серверными сервисами существует, то такие компоненты могут входить в состав общего компонентного приложения.

Определение базовой компонентной среды. Пусть *CSet* – множество имеющихся компонентов и $\Psi = \{ CSet, CESet, \Omega\}$ - внешняя компонентная алгебра. Обозначим $CE^c \in CESet$ – компонентная среда, в которой развертываются все компоненты из CSet:

$$CE^c = CE_1 \oplus CE_2 \oplus \ldots \oplus FW.$$

В дальнейшем компонентные среды типа CE^c будем называть *базовыми* для соответствующего множества *CSet*.

Расширение базовой компонентной среды. Рассмотрим множество компонентов *CSet*. В результате применения операций внутренний компонентной алгебры и операций из обобщенной алгебры строим множество *CSet*,* которое объединяет именно *CSet* и все допустимые результаты применения этих операций над всеми элементами *CSet*. Будем говорить, что *CSet** зявляется замыканием *CSet* относительно множеств операций из указанных алгебр.

Для множества $CSet^*$ строится новая базовая компонентная среда CE^{*c}, которая является расширением предыдущего. Подобный процесс расширения может быть неоднократно повторяться с получением новых компонентов и базовых компонентных сред. Условия распределенной среды и проблемы возникновения и устранения угроз компонентам и каркасам ставят перед разработчиками ПС задачу управления атрибутами качества, безопасности и др. для обеспечения защиты компонентов от разрушений и выходов из тупика.

Теорема 6.3. *Каждая компонентная среда CE – это результат выполнения последовательности операций развертывания компонентов, входящих в ее состав и в компонентный каркас:* $CE = Comp_1 \oplus Comp_2 \oplus, \ldots, \oplus Comp_n \oplus FW$.

Теорема 6.4. *Построение компонентной среды не зависит от порядка инсталляции компонентов, которые входят в состав этой среды, т.е.:*

$$Comp_1 \oplus (Comp_2 \oplus CE) = Comp_2 \oplus (Comp_1 \oplus CE).$$

Теорема 6.5. *Операция объединения компонентных сред ассоциативна:*

$$(CE_1 \cup CE_2) \cup CE_3 = CE_1 \cup (CE_2 \cup CE_3).$$

Теорема 6.6. *Операция объединения компонентных сред коммутативна:*

$$CE_1 \cup CE_2 = CE_2 \cup CE_1.$$

Теорема 6.7. *Для любой компонентной среды* $CE \cup FW = FW \cup CE = CE$.

Теорема 6.8. *Для произвольных компонентных сред* CE_1 *и* CE_2 *и компонента Comp всегда выполняется:*

$$Comp \oplus (CE_1 \cup CE_2) = (Comp \oplus CE_1) \cup CE_2 = (Comp \oplus CE_2) \cup CE_1.$$

Теорема 6.9. *Для любого компонента* Comp *и компонентной среды* CE *всегда выполняется*: $(Comp \oplus CE) \setminus Comp = CE$.

Модель компонентной программы для определенной компонентной среды описывается выражением: $CP = (CE, Cont, Comp)$**,**

где *CE* – компонентная среда; *Cont* – множество контрактов для компонентов, входящих в состав *CE*; *Comp* – подмножество компонентов из *CE*, включающее в себя реализации, для которых отсутствуют входные интерфейсы и которые обращаются к другим компонентам с помощью своих исходных интерфейсов. Компоненты *Comp* – входные, т. е. с них начинается выполнение программы.

Условие целостности компонентной программы состоит в существовании для каждого компонента $Comp_1$ из *CE*, имеющего исходный интерфейс $CInt_1^u$, компонента $Comp_2$ с соответствующим входным интерфейсом $CInt_2^m$, и контракт $Cont_{12}^{im} = (CInt_1^u, CInt_2^m, IMap_{12}^{im})$ входит в состав множества *Cont*.

Процесс построения компонентной программы включает в себя развертывание компонентов и создание компонентной среды, определение начальных компонентов и формирование множества контрактов в соответствии с функциональными требованиями к компонентной программе.

6.2.4 ВНЕШНЯЯ И ВНУТРЕННЯЯ КОМПОНЕНТНЫЕ АЛГЕБРЫ

Внешняя алгебра. Алгебру будем называть *внешней*, если она определяет операции над компонентами и компонентными средами как над целевыми объектами.

Модель компонента и компонентных сред служит основой формирования внешней компонентной алгебры, которая определяет множество операций над соответствующими элементами и имеет такое выражение:

$$\Psi = \{ CSet, CESet, \Omega \}, \quad (6.8)$$

где $CSet$ – множество компонентов, каждый из которых имеет модель (6.3); $CESet$ – множество компонентных сред, каждое из которых описывается выражением (6.4); Ω – множество операций. В состав множества Ω входят такие операции ($Comp$ – компонент, CE_1, CE_2, CE_3 – компонентные среды):

– инсталляция (развертывание компонента) $CE_2 = Comp \oplus CE_1$;

– объединение компонентных сред $CE_3 = CE_1 \cup CE_2$;

– удаление компонента из компонентной среды $CE_2 = CE_1 \setminus Comp$.

Для операций внешней компонентной алгебры сформированы и доказаны теоремы.

Обозначим з $CSet = \{Comp_n\}$ – множество компонентов, а $CESet = \{CE_n\}$ – множество компонентных сред. Все компоненты отвечают условиям компонентной модели, а среды – условиям модели компонентной среды. Формальное определение базовых операций над компонентами в условиях среды приведены ниже.

Операция инсталляции (развертывания) компонента в компонентной среде $CE_2 = Comp \oplus CE_1$ имеет следующую семантику:

$$CE_2.NameSpace = \{Comp.CName\} \cup CE_1.NameSpace,$$
$$CE_2.IntRep = \{Comp.(CInt^i, CName)\} \cup CE_1.IntRep,$$
$$CE_2.ImpRep = \{Comp.(CImp^j, CName)\} \cup CE_1.ImpRep.$$

Операция объединения компонентных сред $CE_3 = CE_1 \cup CE_2$ имеет аналогичную семантику:

$$CE_3.NameSpace = CE_1.NameSpace \cup CE_2.NameSpace,$$
$$CE_3.IntRep = CE_1.IntRep \cup CE_2.IntRep,$$
$$CE_3.ImpRep = CE_1.ImpRep \cup CE_2.ImpRep.$$

Операция $\oplus$ имеет более высокий приоритет, чем операция $\cup$. Этот факт легко объясняется тем, что прежде, чем начать работать с компонентными средами, необходимо инсталлировать их компоненты. Отметим очевидные свойства операций:

$$\forall CE_1,\ CE_2 \cup FW = CE;$$
$$\forall CE_1,\ \forall CE_2\ \ CE_1 \cup CE_2 = CE_2 \cup CE_1;$$
$$\forall CE_1,\ \forall CE_2,\ \forall CE_3\ \ (CE_1 \cup CE_2) \cup CE_3 = CE_1 \cup (CE_2 \cup CE_3);$$
$$\forall Comp,\ \forall CE_1,\ \forall CE_2\ (Comp \oplus CE_1) \cup CE_2 = (Comp \oplus CE_2) \cup CE_1;$$
$$\forall Comp_1, \forall Comp_2, \forall CE\ Comp_1 \oplus (Comp_2 \oplus CE) = Comp_2 \oplus (Comp_1 \oplus CE).$$

Операция удаления компонента из компонентной среды $CE_2 = CE_1 \setminus Comp$ имеет

следующую семантику:

$$\exists\, CName^m \in NameSpace \& (CName^m = Comp.CName) \Rightarrow$$
$$CE_2.NameSpace = CE_1.NameSpace \setminus \{Comp.CName\} \&$$
$$CE_2.IntRep = CE_1.IntRep \setminus \{(\forall u \& IntRep^i.Cname = Comp.CName)\ IntRep^i\} \&$$
$$CE_2.ImpRep = CE_1.ImpRep \setminus \{(\forall j \& IntRep^j.Cname = Comp.CName)\ IntRep^j\}.$$

Для этой операции существует равенство на основе соответствующих операций над множествами, которые входят в определение компонента и среды $(Comp \oplus CE) \setminus Comp = CE$. При ином порядке скобок равенство не всегда выполняется. Это означает, что операция имеет более высокий приоритет, чем операция \.

Операция замещения компонента $Comp_1$ компонентом $Comp_2$ выражается через операции $\oplus$ и \: $CE_2 = Comp_2 \oplus (CE_1 \setminus Comp_1)$.

Пусть Ω обозначает множество операций над компонентами среды $\Omega = \{\oplus, \setminus, \cup\}$, расположенных в порядке уменьшения приоритетов.

Тогда $\Psi = \{CSet, CESet, \Omega\}$ определяет *внешнюю компонентную алгебру*, которая включает в себя множества компонентов, компонентных сред и операции над их элементами.

Внутренняя компонентная алгебра

Во внешней компонентной алгебре компоненты представляют собой целевые объекты. Однако модели компонента имеют собственную структуру. Поэтому целесообразно рассмотреть операции над отдельными элементами структуры, которые называются операциями эволюции (развития).

Суть этих операций – изменение имен, множеств интерфейсов, множеств реализаций, отношений и связей между этими множествами, а также преобразование структуры и функций. Особый интерес представляет собой множество таких операций эволюции, которые составляют целостность понятия компонента, т.е. условия определения и существования компонента в следдствии эволюции не изменяются.

Примеры таких операций:

1) добавление новой реализации для существующего интерфейса;

2) добавление нового интерфейса и новой реализации для него (этот пример характерный для программирования в модели COM);

3) объединение существующих интерфейсов в один и, если необходимо, то и объединение их реализаций в одну общую реализацию.

Множество элементов модели компонента и множество указанных операций рефакторинга определяют внутреннюю компонентную алгебру.

Структура алгебры. Современные языки и модели при практическом применении компонентного подхода (например, COM, CORBA, Java [209]), как правило, требуют, чтобы каждая реализация явным образом указывала на интерфейс, для которого она создана. Необходимость такой схемы обусловлена, в основном, практическими расчетами – проверка сигнатур методов и соответствия типов данных, поддержка модели безопасности, исключение возможных побочных эффектов и т.д. Теоретически не существует принципиальных ограничений, которые мешали бы устанавливать соответствие между интерфейсом и реализацией в динамике (некоторые модели, например, CORBA, поддерживают концепцию динамических интерфейсов, но она недостаточно формализована). Кроме того, произвольная совокупность методов реализации формально может определить

некоторый интерфейс, если ему предоставить уникальное имя.

Это замечание показывает, что, в наиболее общем случае, компонент может иметь реализации с большей функциональностью, чем того требуют явным образом определенные и описанные интерфейсы. А это, в свою очередь, означает, что реализации компонента могут носить избыточный характер, т.е. к компоненту может прибавляться новая функциональность со следующим определением для интерфейса (обратное замечание не верно).

Если к компоненту добавляют новый интерфейс, то он обязательно должен иметь реализацию. Это и есть одно из следствий ограничения целостности.

Среди множества компонентов существует особый компонент, для которого $CInt = \varnothing$ и $CImp = \varnothing$, то есть множества интерфейсов и реализаций – суть пустые множества. Пусть нулевым компонентом или шаблоном компонента является

$$TComp = (Template, \varnothing, CFact, \varnothing, CServ). \quad (6.9)$$

Условие целостности компонента (6.5) выполняется и для шаблона. Множество входных интерфейсов пустое и независимое от наличия или отсутствия в нем реализаций выражения (6.4), которое имеет истинное значение.

Представлениеэтого компонента фактически служит основой средств автоматизации создания компонентов, при применении которых разработчик может, в основном, сосредоточиться на интерфейсах и функциональности будущего компонента, функции же управления экземплярами, взаимодействие с системными сервисами, оформление компонента как целостной структуры включаются автоматически.

Пусть *OldComp* определяет базовый компонент для применения операций, а NewComp соответствует полученному компоненту после выполнения операций

$$OldComp = (OldCName, OldCInt, CFact, OldCImp, CServ),$$
$$NewComp = (NewCName, NewCInt, CFact, NewCImp, CServ).$$

Определим операцию добавления новой реализации. Особенность этой операции состоит в том, что множество интерфейсов компонентов может расширяться за счет добавления новых исходных интерфейсов, если реализация, которая прибавляется, требует дополнительной функциональности и предоставляется другим компонентам. Обозначим $NewCIntO^s = \{NewCIntO^{sq}\}$ – дополнительное множество исходных интерфейсов. В частном случае, $NewCIntO^s = \varnothing$, если дополнительная функциональность при реализации, не добавляется.

Операция добавления реализаций. Выше отмечалось, что существует две разновидности данной операции – это добавление операции для существующего интерфейса и для нового, что формально еще не определено.

Первая операция AddOImp имеет следующую форму записи

$$NewComp = AddOImp(OldComp, NewCImp^s, NewCIntO^s), \quad (6.10)$$

и семантику

$$NewCInt = OldCInt \cup NewCIntO^s,$$
$$NewCImp = OldCImp \cup \{ NewCImp^s \},$$
$$(\exists\, OldCInt^t \in OldCInt) \; Provide(OldCInt^t) \subseteq NewCImp^s,$$

где $NewCImp^s$ – реализация, которая добавляется; $OldCInt^t$ – множество входных интерфейсов с *OldCInt*.

Такая операция ассоциативна и коммутативна. Условие целостности компонента (6.6) выполняется автоматически, так как множество входных интерфейсов остается неизменным и из целостности старого компонента вытекает

целостность нового.

Согласно определению семантики операция *AddOImp* ассоциативна и коммутативна. Доказательство этих фактов является следствием анализа множества интерфейсов и реализаций, которые входят в состав соответствующих компонентов.

Вторая операции добавления реализации и интерфейса *AddNImp* имеет форму записи

$$NewComp = AddNImp(OldComp, NewCImp^s, NewCIntO^s), \quad (6.11)$$

и семантику

$$NewCInt = OldCInt \cup NewCIntO^s,$$
$$NewCImp = OldCImp \cup \{NewCImp^s\}.$$

где $NewCImp^s$ – реализация, которая добавляется.

Как и для предыдущей операции, целостность компонента выполняется. Также можно сказать, что эта операция ассоциативна и коммутативна.

Операция замещения существующей реализации новой реализацией *ReplImp*:

$$NewComp=ReplImp(OldComp, NewCImp^s, NewCIntO^s, OldCImp^r, OldCIntO^r), \quad (6.12)$$

с такой семантикой. Если справедливо, что

$$(\forall OldCInt^t \in OldCInt) \ \& \ (Provide(OldCInt^t) \subseteq OldCImp^r) \Rightarrow (Provide(OldCInt^t) \subseteq NewCImp^s) \ V \ ((\exists OldCImp^j \in (OldCImp \setminus \{OldCImp^r\})) \ \& \ Provide(OldCInt^t) \subseteq OldCImp^j),$$

то

$$NewCInt=OldCInt \cup NewCIntO^s \setminus OldCIntO^r, \quad (6.13)$$
$$NewCImp = OldCImp \cup \{NewCImp^s\} \setminus \{OldCImp^r\},$$

где $NewCImp^s$ – реализация, которая прибавляется; $NewCIntO^s$ – множество дополнительных исходных интерфейсов для реализации, которая прибавляется; $OldCImp^r$ – реализация, которая заменяется; $OldCIntO^r$ – множество исходных интерфейсов, связанных с реализацией, которая замещается.

Лемма 6.1. *Операция замещения существующей реализации (6.12) новой реализацией с семантикой (6.13) сохраняет целостность компонента.*

Для любого входного интерфейса созданного компонента, кроме интерфейсов, отвечающих за реализацию $OldCImp^r$, выполняется замещение и условие целостности. Для интерфейсов, отвечающих за реализацию $OldCImp^r$, справедливы операции предусловий, если $Provide(OldCInt^t) \subseteq NewCImp^s$ или если существует другая реализация базового компонента. Объединяя эти два случая, получаем, что для любого входного интерфейса нового компонента выполняется условие целостности.

Операция расширения существующей реализации семантически эквивалентна операции замещения, соответствует удалению старой реализации и добавлению новой, которая является расширением старой. Поэтому нет необходимости вводить отдельную операцию.

Операция добавления нового интерфейса. Как уже отмечалось, исходный интерфейс может добавляться, если замещается существующая или добавляется новая реализация. В противном случае понятие добавления нового исходного интерфейса не имеет смысла (нет той реализации, в которой содержится обращение новых дополнительных компонентов). Операция добавления исходного интерфейса всегда является составной операцией добавления реализации и как самостоятельная операция не определяется.

Операция добавления нового входного интерфейса *AddInt* имеет вид

$$NewComp = AddInt(OldComp, NewCIntI^q), \quad (6.14)$$

и если справедливо условие с семантикой $(\exists\, OldCImp^s \in OldCImp)$ & $(Provide(NewCIntI^q) \subseteq OldCImp^s)$, то

$$NewCInt = OldCInt \cup \{NewCIntI^q\},\ NewCImp = OldCImp, \quad (6.15)$$

где $NewCIntI^q$ – новый интерфейс.

Как видно из семантики, операция имеет условный характер, т.е. она является частичной операцией. Докажем следующую лемму.

Лемма 6.2. *Операция добавления нового интерфейса для существующей реализации (6.14) и семантикой (6.15) сохраняет условие целостности компонента.*

Пусть выполняется условие (6.6) для базового компонента т.е. для каждого из входных интерфейсов существует соответствующая реализация. Предположение (6.15) требует существования реализации и для нового интерфейса. Поэтому в результате расширения множества входных интерфейсов выражение (6.5) тоже истинно для нового компонента и целостность сохраняется.

*Операция расширения существующего интерфейса***.** В отличие от операции расширения существующей реализации, для которой не существует обязательного требования ее сохранения в структуре компонента, все существующие входные интерфейсы должны сохраняться. Это требование вытекает из условий допустимости методов рефакторинга, рассмотренного выше. Поэтому суть операции сводится к добавлению расширенного интерфейса для новой семантики, которая аналогична предыдущей операции.

6.2.5. ЭВОЛЮЦИЯ КОМПОНЕНТОВ В КОМПОНЕНТНОЙ АЛГЕБРЕ

Операция расширения интерфейса *CFact*. Операция рефакторинга носит комплексный характер, так как дополнительные методы, которые входят в этот интерфейс требуют реализации со стороны контейнера. Поэтому реализация этой операции связана с существованием нового типа контейнера. Соответственно классификации методов рефакторинга, приведенной выше, операция расширения интерфейса управляет экземплярами компонентов и относится к расширенной классификации [68, 196].

Аналогичные суждения справедливы и для анализа операции расширения интерфейса системных сервисов. В этом случае необходимым условием является реализация дополнительных методов со стороны компонентной среды, а сама операция относится к расширенной классификации методов рефакторинга.

Таким образом, приведенный выше анализ позволяет определить *внутреннюю компонентную алгебру*.

Пусть $CSet = \{Comp_n\}$ – множество компонентов, каждый из которых описывается моделью (6.3).

Определим множество *операций рефакторинга* так

$$Refac = \{ AddOImp, AddNImp, ReplImp, AddInt\},$$

которые определяются операциями добавления и замещения. Тогда пара *(CSet, Refac)* входит во внутреннюю компонентную алгебру, которая включает допустимые методы рефакторинга.

Теорема 6.10. *Алгебра рефакторинга компонентов* $\Sigma^{rf} = (CSet, Refac)$ *является полной и неопровержимой*.

Связь внешней и внутренней компонентных алгебр. В соответствии с теоремой 6.10 результатом операции рефакторинга или операции суперпозиции выступает определенный компонент Это свидетельствует о связи внешней компонентной алгебры с алгеброй рефакторинга. Множество *CSet* состоит из компонентов репозитария и разных модификаций компонентов, как результатов выполнения операций рефакторинга, т.е. в операциях внешней компонентной алгебры вместо компонентов могут применяться их модификации. Например, для определенной компонентной модели компонентная конфигурация состоит из двух компонентов, и для второго компонента добавляется реализация и входной интерфейс с помощью такого выражения:

$$CE=Comp_1 \oplus AddInt(AddNImp(Comp_2, NewCImp^s, NewCIntO^s), NewCIntI^q) \oplus FW.$$

Модели реинжиниринга и реверсной инженерии. Внутренняя компонентная алгебра реинжиниринга компонентов $\Sigma^{re} = (CSet, Reeng)$ может быть построена лишь при условии нарушения целостности представления компонентов. Кроме операций рефакторинга *(Refac),* во множество *Reeng* входят операции, которые удаляют из компонента существующий интерфейс или изменяют его сигнатуру. Это усиливает условие целостности, так как другие компоненты, которые обращаются к нему, не могут иметь доступа к необходимой функциональности. Исходя из таких условий, вместо алгебры реинжиниринга в состав формальных методов компонентного программирования целесообразно включить модель реинжиниринга.

Модель реинжиниринга компонентов имеет вид

$$M_{Reeng} = (CSet, Reeng).$$

Семантика операций *Reeng* может состоять в одновременной трансформации не только целевого компонента, но и других. Например, при изменении сигнатуры входного интерфейса необходимо одновременное изменение других компонентов, в которых существуют обращения к методам этого интерфейса.

Аналогично, формируется соответствующая модель реверсной инженерии $M_{Revers} = (CSet, Revers),$ при этом $Reeng \subset Revers$. Особенность множества *Revers* состоит в том, что оно не определено полностью (количество операций неограниченно), а разрешает лишь классификацию операций. Например, качественные характеристики компонентов могут изменяться произвольно, а соответствующие операции образуют совокупность операций для изменения определенного показателя качества (сам показатель – классификационный признак в системе классификации множества операций *Revers*).

Таким образом, алгебра Σ^{f} и модели M_{Reeng} *и* M_{Revers} – формальный аппарат методов эволюции компонентов.

Обобщенная компонентная алгебра. Эта алгебра объединяет внешнюю и внутреннюю компонентные алгебры, а также модель системных сервисов на единой формальной платформе. Кроме непосредственного объединения возможностей указанных алгебр, обобщенная алгебра разрешает формализовать такие дополнительные задачи:

1) для выбранного множества компонентов определить последовательность операций рефакторинга, что обеспечит построение компонентного приложения (или доказать, что для такого множества эта задача не разрешима);

2) указать необходимые условия, чтобы выбранная компонентная конфигурация имела свойства транзакционности (или доказательство отсутствия такого свойства);

3) обеспечить реализацию комбинированных операций, например, переименование компонентов и/или их интерфейсов (которые по сути рефакторинг) с образованием нового компонента, который дополняет существующую компонентную среду.

6.3. ФОРМАЛИЗОВАННОЕ ПРЕДСТАВЛЕНИЕ КОМПОНЕНТНЫХ СИСТЕМ

Базовую задачу компонентного программирования формулируется следующим образом. Пусть есть множество компонентов *CSet* и приведены компонентные алгебры. Цель – построить компонентную программу, которая удовлетворяет функциональным и нефункциональным требованиям на заданном множестве компонентов [64–65].

Задача состоит в представлении результатов проектирования программы в компонентном виде $CP=\{M_v\}$ таким образом, чтобы для любого M_v существовал компонентов из *Cset* или он мог быть получен из этого множества в результате конечного числа допустимых операций из соответствующих компонентных алгебр.

На практике такие условия могут выполняться не всегда (например, разработка программы с принципиально новой функциональностью). В этом случае реализуются дополнительные компоненты и расширяется множество *CSet*.

Таким образом, основная задача компонентного программирования состоит в построении копонентных программ и систем на основе компонентных моделей, методов и средств. Переопределим и сформулируем эту задачу в терминах, понятиях и моделях, рассмотренных выше.

Компонентное представление ПС. Под *компонентным представлением программы* и ПС понимается такое абстрактное их определение, которое состоит из программных элементов, сопоставленных реальным компонентам, обеспечивающим реализацию функциональности задач ПрО и выполнение других, не функциональных требований. Для одной и той же программы существует множества ее компонентных представлений в зависимости от концепций проектирования и используемого компонентного подхода.

Рассмотрим обобщенный уровень такой модели.

Основная задача проектирования компонентной программы состоит в представлении ее функциональности и других требований, как совокупности контрактов между отдельными объектами – программными компонентами, из которых она будет собираться.

Пусть $\{A_i\}$ – множество контрактов для компонентной программы, определяющих ее функциональность. Каждому A_i можно сопоставить интерфейс I_i, описывающий контракт как клиент-серверное взаимодействие с соответствующими методами и структурами данных. Согласно этому, каждому интерфейсу можно сопоставить пары In_i и Out_i, которые будем называть соответственно реализующим представлением для I_i и определяющим представлением для I_i. Эти термины используются вместо понятий входной и выходной интерфейс потому что реально они не являются такими интерфейсами (для них могут и не существовать интерфейсы в существующих компонентах). В результате перепроектирования они могут быть переопределены и изменены в целях адаптации их к реальным интерфейсам. Out_i определяет условия и цель контракта со стороны клиента, а In_i задает аспект реализации контракта со стороны

сервера. После того, как все In_i и Out_i описаны можно их сгруппировать.

Рассмотрим произвольную совокупность $M_v = \{In_i\} \cup \{Out_j\}$. Назовем каждое M_v шаблоном компонента. Фактически такой шаблон содержит некоторое множество определяющих и реализующих представлений интерфейсов. По этим представлениям в дальнейшем осуществляется сопоставления шаблонов и реальных интерфейсов существующих компонентов.

Если результаты сопоставления для всех M_v успешные, то в итоге подобные операции группирования образуют множество $CP=\{M_v\}$, которое и будет называться *компонентным представлением программы*. Для одной и той же программы существует множество представлений, которые определяются множествами контрактов (или интерфейсов) и способами построения шаблонов.

6.4. СУТЬ ОБЪЕКТНО-КОМПОНЕНТНОГО МЕТОДА

Между объектным и компонентным представлениями программ существует неоднозначность, которая порождается тем, что определенный компонентов может иметь реализации для нескольких интерфейсов Isyst (6.14). В частном случае, если каждый из интерфейсов реализуется отдельным компонентом, существует единое эквивалентное отображение между объектными и компонентными представлениями [64].

Пример объектного и компонентного представления сложной программы показан на рис.6.2 четырьмя классами объектов: O_1, O_2, O_3, O_4. Для классов O_2, O_3, O_4 совокупности public-методов и управляемые переменные обозначены как входные объектные интерфейсы IO_2, IO_3, IO_4 соответственно.

В интерфейсном представлении *ISyst* соответствуют компонентные интерфейсы IC_2, IC_3, IC_4 (штриховые линии). При этом $OC_{11}, OC_{12}, OC_2, OC_3$ – исходные интерфейсы в компонентной модели. Между объектными и компонентными интерфейсами установлено однозначное отображение, а для объектной и компонентной модели такого отображения не существует, так как компонент Comp3 имеет два входных интерфейса, т.е. функциональность классов O_3 и O_4 реализована в одном компоненте.

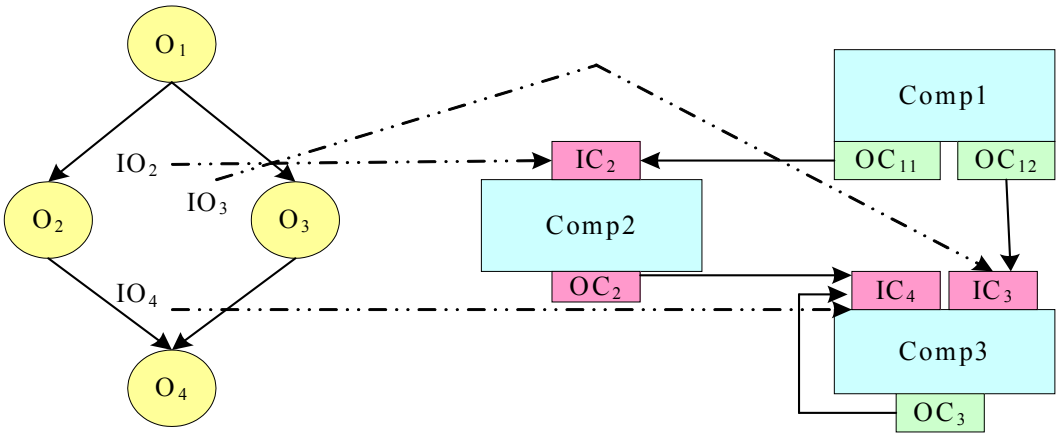


Рис.6.2. Пример структуры программы, созданной ОКМ

Выводы. Объектный анализ – первая фаза объектно-компонентного метода

проектирования ПС. На ней проводится анализ ПрО в целях выявления объектов и построения ОМ, которая адекватно отображает ее структуру, объекты и отношения между ними и т.п. Главная задача второй фазы – проектирование конкретных компонентов и ПС по результатам анализа ПрО. Метод ОКМ обобщает понятия объектов как элементов действительной реальности путем концептуального моделирования и объектно-ориентированного анализа ПрО с применением математических формализмов на разных уровнях представления объектов и ОМ. В ОКМ даны процессы определения объектов, начиная с отдельных сущностей ПрО и заканчивая заданием компонентов с учетом их поведения в среде. Объектная модель отображается в компонентную модель путем формирования функциональных интерфейсов и распределения их между конкретными компонентами на основе внешней и внутренней компонентной алгебры.

ОКМ метод включает следующие новые теоретические положения:

– концепции, терминологию и методы композиции/сборки, которые объединены в единую схему понятий метода ОКМ со строгим определением причинно-следственных связей между ними;

– базовые элементы теории – модели компонента и компонентной среды, внешняя и внутренняя компонентные алгебры, отношения наследования, экземпляризации, контракта и связывания;

– задачу компонентного программирования в заданном множестве программных объектов, операций внешней и внутренней компонентных алгебр;

– парадигму композиционной сборки компонентов в распределенной среде, которые обмениваются между собой через механизм вызова и взаимодействия.

Предложенная теория, включая компонентную алгебру, оригинальная, она не имеет прототипа. Методология компонентного программирования применялась при создании ИС для НАН Украины.

Глава 7

МЕТОДЫ СБОРОЧНОГО ПОСТРОЕНИЯ ИНТЕГРИРОВАННЫХ КОМПЛЕКСОВ

7.1. ОСОБЕННОСТИ ЗАДАНИЯ ИНТЕРФЕЙСОВ В ИНТЕГРИРОВАННЫХ КОМПЛЕКСАХ (ИК)

Ранее введенное понятие программного интерфейса рассматривалось для пары взаимодействующих компонентов. Это полностью справедливо при комплексировании модулей, где требуется обеспечить интерфейс между вызывающим и вызываемым модулями. При переходе к ИК понятие программного интерфейса требуется расширить. Покажем это на примере схемы ИК (рис. 7.1), содержащую программ P_1, P_2 , P_3 и файлы данных F_1, F_2 .

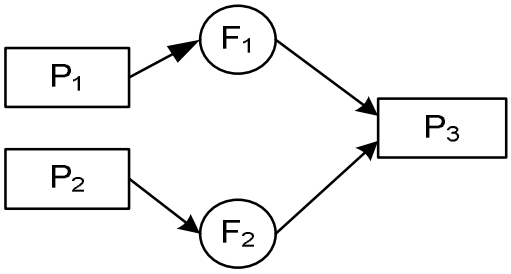


Рис.7.1. Схема интегрированного комплекса

Результатом работы программы P_1 служит файл F_1 а программы P_2 – файл F_2. Оба файла являются входными данными дли программы P_2. Возможны следующие последовательности выполняемых программ: $P_г – P_2 – P_3$ и $P_2 – P_1 – P_3$. Если необходимо изменить пред-ставление данных в файлах F_1 и F_2 для программы P_3 , то для этого требуется преобразовать эти данные с помощью соответствующих интерфейсов.

Рассмотрим интерфейс, требуемый для работы программы P_3. Очевидно, что прежнее понятие интерфейса модулей в данном случае не совсем применимо, поскольку нельзя выделить конкретную пару взаимодействующих модулей, связь между которыми охватывала бы все данные для программы P_3. Этот пример показывает, что для ИК требуется иное понятие интерфейса.

Множество данных ИК определено во второй главе формулой (2.6), а также:

$$D = (\bigcup_{i=1}^{s} D^i) \bigcup D^c. \qquad (7.1)$$

Дополним это множество для i-й компоненты D^i данными, характеризующими среды ЯП, систем программирования и т.д. для модулей, входящих в состав этого компонента. Примером таких данных могут служить данные среды ЯП, о которых говорилось в гл.3. Они обычно недоступны прикладному программисту и необходимы для реализации внутренних механизмов управления и функционирования отдельных модулей и всей компоненты в целом. Перечислим наиболее важные типы таких данных:

списки областей используемой и свободной памяти;

списки областей сохранения регистров;

стек, содержащий внутренние данные, адреса передачи и возврата управления для программ с модульной или блочной структурой;

блоки описания задач;

блоки описания файлов и устройств ввода-вывода;

признаки, определяющие необходимые действия при возникновении специальных условий, программных прерываний и аварийных ситуаций;

используемые шрифты для ввода информации с клавиатуры и вывода на терминал и печатающее устройство;

внутренние структуры для управления динамическими объектами, буферами ввода-вывода и т.д.

Внутренние структуры этих данных, их методы обработки зависят от конкретных операционных систем, трансляторов с ЯП, систем программирования и других инструментальных средств. Поэтому особенности их использования в настоящей книге не рассматриваются. Предполагается, что сами данные могут быть описаны в рамках теории структурной организации данных. Обычно на практике это условие выполняется.

Переопределив множества D^i, мы для упрощения изложения оставляем те же обозначения. Как было отмечено в гл. 2, каждое $d_j^i \in D^i$ характеризуется именем N_j^i, типом данных T_j^i и текущим значением V_j^i. Некоторые переменные, имеющие различные имена и принадлежащие различным компонентам, имеют одинаковое семантическое содержание, т.е. представляют различные реализации общей для нескольких компонентов информационной структуры. Приводимые ниже примеры показывают наиболее типичные случаи данных с одинаковым семантическим содержанием.

1. На вход программы сортировки SORT поступает файл F_1 содержащий неупорядоченные записи. Результатом работы SORT служит упорядоченный файл F_2. Программа FORM формирует отчет на основании данных из файла F_3 , содержащего упорядоченные записи. Если программа SORT будет выполнять сортировку записей со структурой, аналогичной структуре записи файла F_s , то обе программы могут быть объединены в единый программный комплекс. При этом файл F_2 и F_3 представляют собой данные с одинаковой семантикой. На практике соответствие между этими файлами может быть установлено следующими способами:

создание копии файла F_2 с именем F_3;

переименование имени F_2 *в* F_3 средствами ОС;

настройка программы FORM на обработку файла F_2, если имя определяется параметрически.

2. Программа FORM формирует отчет и выводит его в файл F_4.

Программа REGISTR регистрирует полученные документы. Для регистрации необходимы наименование отчета, дата и время его формирования. Эти данные программа REGISTER получает из командной строки (строки, содержащей имя вызываемой программы и передаваемые параметры). Если существуют универсальные средства интерфейса, позволяющие выбрать необходимую информацию из файла F_4 и сформулировать командную строку, то программы FORM и REGISTR можно объединить в единый программный

комплекс. При этом поля из файла F_4, содержащие наименование отчета, дату, время, и соответствующие поля из командной строки будут представлять данные с одинаковым семантическим содержанием.

3. Некоторые операционные системы (например, MS DOS) позволяют в общесистемной области определять строковые константы с указанием символического имени для каждой строки. Строка может представлять имя файла, каталоги и т. д. Программы могут обращаться к этой области и по соответствующему символическому имени выбирать необходимую информацию. Может возникнуть ситуация, когда несколько программ должны получать одинаковую информацию из общесистемной области, но каждая программа содержит свое собственное символическое имя, определяющее данные с одинаковым семантическим содержанием. Реализация интерфейса для рассматриваемых программ может включать:

копирование информации в общесистемной области;

перед вызовом конкретной программы производить замену соответствующего символического имени.

Приведенные выше примеры не исчерпывают всех случаев данных с одинаковым семантическим содержанием. Их характерность состоит в различных комбинациях расположения данных: файл–файл, файл– оперативная память (ОП), ОП – ОП.

Рассмотрим множество D, определяемое в (7.1). Ему соответствует множество имен переменных N:

$$N = (\bigcup_{i=1}^{s} N^i) \bigcup N^c .$$

При этом положим, что все имена в $N^1,...,N^s$, N^c разные. Это условие не существенно, так как можно перейти к рассмотрению составных имен вида $P^i.N^i_j$, где P^i соответствует имени i-го программного компонента, а $N^i_j \in N^i$. Эти имена явно отличаются.

Через E обозначим отношение на N, определяющее данные с одинаковой семантикой. Нетрудно заметить, что E – отношение эквивалентности. Оно определяет разбиение множества на классы эквивалентности N_p:

$$N=\bigcup_{p=1}^{r} \tilde{N}_p, \qquad (7.3)$$

где $N_p \bigcap N_q = \emptyset$ при $p \neq q$. Через $\tilde{N}$ обозначим фактор-множество N/E по отношению эквивалентности E. Определим множество G $=\{G_1 \ldots, G_r\}$, где G_p – имя для класса эквивалентности $\tilde{N}_p$.

В качестве имени можно выбрать любое имя, являющееся представителем класса $\tilde{N}_p$. Однако для упрощения будем рассматривать имена G_p, отличные от имен из N.

Для каждой программы P^i множество данных D^i состоит из входных и выходных переменных. Перед выполнением P^i выходные переменные не определены. Для рассмотрения этой ситуации в нашей терминологии введем понятие «пустого значения» V_0^0, которое принимает любая неопределенная переменная независимо от ее типа. Соответственно введем в рассмотрение и понятие «пустого типа данных» T_0^0. Отметим, что переменная, имеющая тип T_0^0,

всегда имеет значение K_0^0. Обратное выполняется не всегда – тип переменной может быть известен, но она сама не определена.

Рассмотрим множество $B=\{b_1, ..., b_r\}$ определяемое следующим образом. Каждый элемент b_p представляется тройкой (G_p, T_p, V_p). Для V_p имя G_p постоянно, а тип T_p и значение V_p меняются в процессе выполнения программ P^i, составляющих интегрированный комплекс. Перед началом выполнения $T_0 = T_0^0$, $V_0 = V_0^0$, при $p=\overline{1,r}$ (r – число программ в комплексе). В дальнейшем множество D будет называться базой данных, а B – информационной средой ИК. В каждый момент времени все T_p и V_p имеют конкретные значения (возможно, пустые). Набор конкретных значений для T_p и V_p будет определять состояние информационной среды ИК. В процессе выполнения программ состояние информационной среды $B = \{B^t\}$ меняется, т. е. существует функциональная зависимость вида $B^{t+1}=f(B^t, P^i)$, где B^{t+1} соответствует новому состоянию после выполнения программы P^i.

Рассмотрим более подробно переход от B^t к B^{t+1}. Программе P^i соответствует множество данных D^i. Пусть $d_j^i \in D$ и $d_j^i = (N_j^i, T_j^i, V_j^i)$. Имени N_j^i соответствует некоторый класс эквивалентности N_p, и, следовательно, $b_p = (G_p, T_p, V_p)$. Для входных переменных T_p и V_p должны быть определены, т. е. $T_P \neq T_0^0$, $V_P \neq V_0^0$. Выполнение этих условий рассматривалось в модели управления программными объектами (см. гл. 2). Пусть $T_p = T_i^k$ и $V_p = V_i^k$. Для выходных переменных из D^i соответствующие элементы b_p рассматриваются с $T_p = T_0^0$ и $V_p = V_0^0$.

Перед вызовом программы P^i данные $b_p=(G_p, T^k{}_i, V^k{}_i)$ должны быть преобразованы к представлению $d_j^i = (N_j^i, T_j^i, V_j^i)$. Эти преобразования являются частичными задачами в модели информационного сопряжения (см. гл. 2). Если существуют алгоритмы преобразований FT_{ij}^{ki} и FV_{ij}^{ki}, то происходит переход к промежуточному состоянию информационной среды $B^{t\prime}$. После выполнения программы P^i элементы b_P, соответствующие выходным данным из D^i, принимают вид $b_p = (G_p, T^i_j, V^i_j)$. Этим определяется новое состояние информационной среды b^{t+1}. При вызове следующей программы выполняются аналогичные действия.

Для завершения анализа необходимо рассмотреть начальное состояние информационной среды B^0. Ранее было отмечено, что к началу выполнения для всех $b_p = (G_p, T_p, V_p)$, $T_p = T^0_0$ $V_p = V^0_0$. На основании модели управления программными объектами выбирается начальная программа P^k. Для входных данных из D^k в соответствующих элементах b_p выполняется присваивание $T_P = T^k_j$, $V_P = V^k_i$. Этим и определяется начальное состояние информационной среды B^0.

Используя предыдущий анализ, введем понятие программного интерфейса ИК.

Программным интерфейсом для P^i в ИК называется комплекс средств, обеспечивающий сопряжение программ комплекса исходя из текущего состояния информационной среды B и множества данных D^i.

Это определение согласуется с моделью управления программными объектами и является обобщением интерфейса прикладных программ, работающих с БД. В отличие от БД, которая имеет постоянную структуру, в информационной среде связи между ее элементами могут динамически меняться, что зависит от конкретно выполняемых программ. Механизмы реализации БД и информационной среды различные. В то же время организация интерфейса с прикладными программами аналогично – через набор специальных подпрограмм.

7.2. МЕТОДЫ И СРЕДСТВА ИНТЕГРАЦИИ ИК

Вопросы интеграции программных средств неявно присутствуют во всех методах, подходах, системах программирования, где происходит регламентация объектов и правил их взаимодействия. В операционных системах прикладным программам предоставляются стандартные механизмы управления вводом-выводом, памятью, задачами и т. д. Благодаря использованию этих механизмов осуществляется совместное функционирование программ в мультизадачной и мультипрограммной средах.

Языки программирования предоставляют множество типов данных, набор операций над данными, механизмы взаимодействия программных объектов. Это является необходимым условием комплексирования объектов.

Различные системы программирования предоставляют разработчику стандартные методы, процедуры, алгоритмы для создания программного обеспечения. Используемые методы, процедуры, алгоритмы обычно содержат правила, необходимые для описания, разработки и объединения различных программных объектов в единый интегрированный продукт.

Несмотря на многообразие рассмотренных аспектов интеграции, их объединяет то, что большинство методов и средств используется на процессе разработки ПС. Методы построения интегрированных комплексов из готовых программ недостаточно исследованы и инструментальные средства их построения менее разнообразны, чем средства разработки ПС.

Наибольшее развитие интегрированные комплексы получили в связи с широким распространением персональных ЭВМ. К ним относятся интегрированные пакеты для деловых приложений – задачи организационного управления, обработки данных, коммерческих расчетов и т. д. В их состав входят: текстовый редактор, средства управления электронными таблицами, СУБД, графика, средства коммуникации ПЭВМ в сеть и др.

В настоящей книге ИК рассматривается как комплекс разных программ. Авторы не ставили перед собой цель – описать как можно больше известных интегрированных средств. Те пакеты или системы, которые будут приводиться, являются иллюстрациями методов интеграции, которые разделим по двум основным признакам.

1. Различие в процессах разработки ИК Выделяются методы, применяемые на процессе проектирования и комплексирования готовых программ.

2. Различие в применяемых инструментальных средствах. Выделяется использование готовых интерфейсов и средств построения ИК.

Рассмотрим некоторые интегрированные средства.

Методы процесса проектирования программ. Отличительной особенностью этих методов является предопределенность состава программ и совместно используемых типов данных. Главное достоинство – возможность построения интерфейсов, предоставляющих оптимальные механизмы передачи управления и преобразования данных. Недостаток – относительная сложность использования дополнительных программных компонентов. Рассмотрим несколько примеров.

Пример 1. Пакет Lotus 1-2-3 [167]. В его состав входят: средства обработки ЭТ; СУБД; пакет деловой графики. Это один из первых интегрированных пакетов.

Пример 2. Система Framework [32]. В ее состав входят: текстовый редактор; средства обработки ЭТ; СУБД; графический пакет.

Для работы с системой используется развитый человеко-машинный интерфейс.

Пример 3. Система Knowledge Man (KMan) [189]. В состав системы входят: текстовый редактор; средства обработки информации на экране, СУБД, пакет деловой графики, средства коммуникации и др.

Система обеспечивает различные уровни взаимодействия с пользователями в зависимости от их квалификации.

Все эти пакеты характеризуют единое представление данных (в рамках своего пакета) и процедуры управления, связанные с эффективным использованием оперативной памяти и минимизации обращения к дискам. Примером аналогичной системы, разработанной в нашей стране, служит интегрированный пакет Мастер [182]. В его состав входят: текстовый редактор; средства обработки экранов, СУБД, средства обработки графиков и рисунков, связь с удаленными абонентами и др.

К интегрированным комплексам следует также отнести Турбо-Системы программирования (Турбо-Паскаль, Турбо-Бейсик, Турбо-Пролог, Турбо-Си). В состав этих систем обычно входят: текстовый редактор, средства обработки файловой системы, транслятор и редактор связей, отладчик и др.

Использование готовых интерфейсов. Некоторые стандартные пакеты имеют готовые интерфейсы для связи с другими системами. Рассмотрим некоторые примеры.

Пример 1. Пакет Lotus 1-2-3 имеет специальную программу Translate, позволяющую переводить файлы в собственном формате в форматы других систем и обратно. К этим системам относятся также Visicalcud-Base II.

Пример 2. Система Supercalc [125] имеет специальную программу-утилиту SDI для преобразования файлов, записанных в собственном формате, в файлы других систем. К ним относятся: СУБД dBase-II и Data star, система VisiCalc, пакет Lotus 1-2-3; прикладные программы, написанные на ЯП Бейсик, Паскаль, Кобол.

Пример 3. Система KMan имеет стандартный интерфейс для подключения Си-программ.

Приведенные пакеты и системы реализованы на ПЭВМ. Однако стандартные интерфейсы реализованы и на других классах ЭВМ. Так, на ЕС ЭВМ имеются интерфейсы между СУБД ДИСОД и ППП Кама, между ППП Кама и СУБД Ока и т. д.

Средства интеграции программ. Под инструментальными средствами интеграции понимаются средства программирования и комплексирования программ ИК, а также средства, обеспечивающие функционирование в интегрированных средах.

Некоторые интегрированные системы имеют средства программирования для разработки прикладных программ, использующих возможности самих систем. К ним, в частности, относятся языки пакетов KMan и Framework.

Пакеты типа ППП Кама предоставляют набор средств для разработки прикладных программ на ЯП высокого уровня. Эти средства включают возможность единого описания данных и стандартные механизмы связи между программами.

Методы комплексирования готовых программ. Как было отмечено ранее, данные методы недостаточно исследованы. Многообразие типов и структур данных, типов внешних устройств, механизмов управления и связи между

программами не позволяет создать универсальную систему интеграции. Существующие методы и средства охватывают только отдельные вопросы интеграции.

Для комплексирования готовых программ применяются стандартные и разработанные самим пользователем интерфейсы.

На персональных ЭВМ реализованы специальные средства объединения прикладных программ и пакетов – интеграторы. К ним, в частности, относятся WINDOWS, CONCURRENT DOS, GEM, TOPVIEW. Обычно интеграторы работают на уровне ОС – заменяют ее или являются надстройкой над ней. Поэтому прикладные программы должны использовать только стандартные средства ОС ввода-вывода, управления памятью и т. д. Использование интеграторов позволяет:

– организовывать многооконный интерфейс с пользователем;

– закреплять за отдельными окнами конкретные прикладные программы и выполнять псевдопараллельную обработку;

– обмен информацией между прикладными программами через окна;

– сделать прикладные программы независимыми от типов используемых на ПЭВМ устройств ввода-вывода – адаптеров клавиатур и терминалов, принтеров.

Выполняя большое число функций по организации управления прикладными программами, интеграторы сравнительно мало обеспечивают возможности информационного обмена в объеме задач, рассматриваемых в настоящей книге. Поэтому универсальные инструментальные средства интегрирования программ дополняются средствами информационного обмена и выбора программ для ИК. Эти средства условно можно определить как средства логического проектирования ИК.

7.3. ЛОГИЧЕСКОЕ ПРОЕКТИРОВАНИЕ КОМПЛЕКСОВ

В предыдущем изложении методы и средства построения ИК рассматривались в виде отдельных элементов, затрагивающих отдельные частные проблемы. Реальное создание ИК требует применения этих методов и средств в комплексе с учетом их взаимосвязей. Ниже приведено общее описание методов и средств логического проектирования ИК, выполненных авторами.

В логическом проектировании ИК выделены следующие процессы:

1. Формулировка задач, решаемых создаваемым ИК, и выбор готовых компонентов.
2. Разработка программ для отсутствующих компонентов ИК.
3. Описание базы данных / знаний ИК.
4. Разработка моделей сопряжения и управления программными объектами.
5. Реализация среды функционирования ИК.

Ниже рассматриваются эти процессы более подробно.

7.3.1. АНАЛИЗ И ВЫБОР КОМПОНЕНТОВ ДЛЯ СОЗДАНИЯ ИК

Задачи, решаемые па данном процессе, относятся к проблеме повторного использования программных объектов. Однако, задачи эти сложнее, чем при выборе отдельных модулей. Сложность обусловлена тем, что выбор компонентов ИК должен происходить комплексно. В отличие от модулей, которые можно

модифицировать в процессе разработки, программы изменять нельзя, и особенности одной из них могут сказываться на функционировании всего ИК. Сложность этой проблемы показывает, что должны существовать специальные инструментальные средства, помогающие выбрать необходимые программные компоненты. Наиболее подходящей для этого является экспертная система.

Главное в создании такой системы состоит в определении характеристик программных объектов, знаний об этих объектах, на основании которых осуществляется их выбор. Выделим три класса характеристик – функциональные, программно-технические и технологические.

Функциональные характеристики. К ним относятся назначение и возможности данного программного объекта. При создании экспертных систем важным моментом является иерархическая упорядоченность возможностей по схеме: класс решаемых задач – задача – функции для решения данной задачи – операции, поддерживающие выполнение каждой функции. Наиболее типичный пример для иллюстрации иерархической упорядоченности возможностей представляет СУБД (рис.7.2). Степень подробности, с которой будут описаны аналогичные схемы программных объектов, влияет на их объективный выбор.

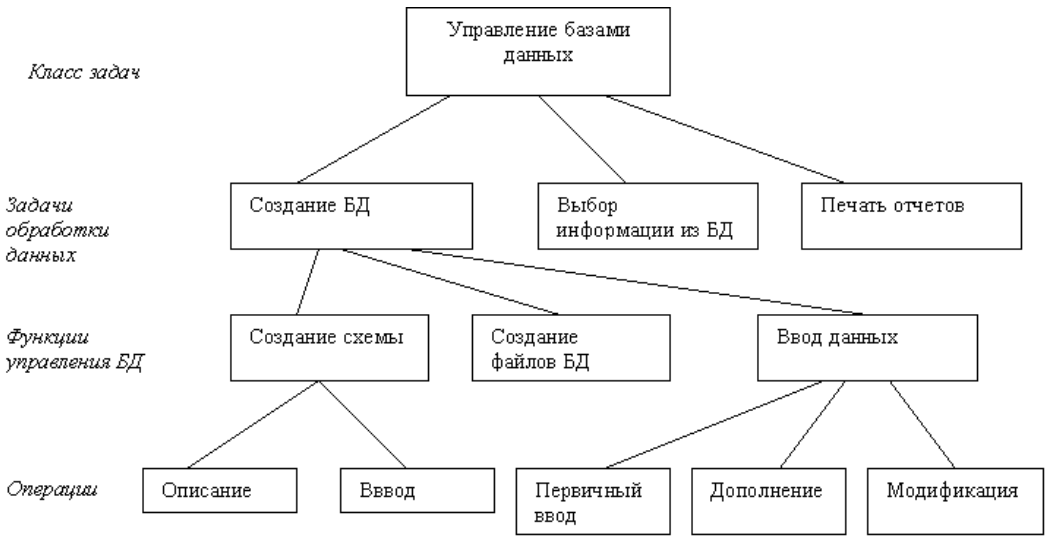


Рис. 7.2. Схема возможностей СУБД для представления знаний

Программно-технические характеристики. К ним относятся характеристики среды функционирования и особенности взаимодействия с другими компонентами. Среда функционирования включает наличие необходимых технических средств, тип операционной системы, особенности выполнения. Взаимодействие с другими компонентами характеризуется наличием стандартных интерфейсов, общим использованием данных, а также связями с системами программирования.

Технологические характеристики систем образуют неформализуемый класс, что связано с наличием человеческого фактора. К ним, в частности, относятся:

– уровень взаимодействия с системой (программист, конечный пользователь и т. д.);

– режимы взаимодействия (пакетный, диалоговый, удаленный и т. д.);

– наличие или отсутствие собственных средств программирования;

– наличие дополнительных сервисных средств;

– квалификация пользователя для работы с системой (оператор, системный или прикладной программист и т. д.);

– наличие и состав документации;

– наличие обучающих курсов;

– время освоения данных средств с учетом категории и квалификации пользователя;

– временные и надежностные характеристики, показатели качества.

Исходя из разнообразия и большого количества характеристик, можно сделать вывод, что экспертная система не может быть сделана окончательно. В этом ее отличие от обычных автоматизированных информационных систем (АИС), которые могут использовать в качестве отправной точки опыт создания экспертных систем. Их развитие связано с формальным определением связей между различными ПС, описанием различных альтернативных путей поиска, формированием численных значений и весовых коэффициентов для отдельных характеристик компонентов и построением оценочных функций выбора.

Важный процесс построения экспертной системы – выбор модели представления знаний. Наилучшим средством их преставления и сведений о характеристиках ПС являются фреймы [159]. Одной из задач процесса построении экспертной системы является выбор модели представления знаний. Информация о программных компонентах может быть описана как сеть фреймов.

Сформулировав запрос на выбор компонента (например, по функциональным признакам) и постепенно уточняя дополнительные характеристики, продвигаемся по фреймовой сети. С помощью механизмов наследования атрибутов и сопоставления выбирается следующий шаг поиска. Поскольку информация обо всех программных объектах входит в единую сеть, тем самым учитываются не только характеристики отдельных объектов, но и совместимость между ними, их связи и т. д. Например, механизм наследования атрибутов позволит контролировать выполнение всех компонентов в единой операционной системе и на однотипных ЭВМ. Более подробно фреймовые модели рассматриваются в [203].

Несмотря на очевидную привлекательность фреймовой модели, она не может применяться на начальном процессе создания ЭС. Это связано со значительной неопределенностью в выборе характеристик программных объектов, их значений, связей и т. д. Поэтому наиболее оптимальным вариантом может служить продукционная модель, т.е. представление знаний о ПС в виде правил типа «ЕСЛИ – ТО». Приведем примеры нескольких возможных правил для выбора СУБД:

ЕСЛИ (требуется СУБД) И (тип IBM PC) ТО (СУБД для IBM PC)

ЕСЛИ (СУБД для ЕС ЭВМ) И (ОС = UNIX) ТО (СУБД для UNIX)

ЕСЛИ (СУБД для IBM PC) И (ОС = MS DOS) ТО (СУБД для MS DOS)

ЕСЛИ (СУБД для ОС ЕС) И (SQL–модель) ТО (СУБД IMS)

ЕСЛИ (СУБД для UNIX) И (SQL–модель) ТО (СУБД INGRES)

ЕСЛИ (СУБД для IBM PC) И (SQL–модель) ТО (СУБД DBASE III)

ЕСЛИ (СУБД для IBM PC) И (интегрированная среда) ТО (Framework).

Эти правила очень простые. Правила для реальной экспертной системы будут значительно сложнее и содержать больше характеристик в условной части, которые потребуют механизмов управления применяемыми правилами. Продукционная модель позволяет добавлять и изменять значения соответствующих правил и

учитывать неопределенность в представлении знаний.

Форма записи продукционных правил зависит от применяемых инструментальных средств построения экспертных систем – оболочек ЭС, языков представления знаний, ЯП. Рассматриваемая ЭС имеет особенность, связанную с тем, что формировать. базу знаний может не специалист по инженерии знаний, а конечный пользователь. Для них подходят пакеты создания прикладных экспертных систем (например, Rule Master [182]. Он обеспечивает несколько уровней представления знаний. Один из них – уровень конечного пользователя. Знания представляются в табличной форме и для рассмотренных выше правил связи с СУБД будут иметь вид, приведенный в табл. 7.1. С помощью специальных средств эта таблица описывается, а также действий для ввода данных по отдельным ее столбцам.

Таблица 7.1.

Тип ЭВМ	**ОС**	**Характер истика**	**…**	**Характерис тика *N***	**Дальнейшие действия (выбор СУБД)**
ЕС ЭВМ	ОС ЕС	**...**		**...**	IMS
ЭВМ СМ	СВМ ЕС	**...**		**...**	...
ВМ СМ	ОС РВ	**...**		**...**	...
ЭВМ	UNIX	**...**		**...**	INGRES dBASE-III
IBMPC	MSDOS	**...**		**...**	

Средства Rule Master обеспечивают проверку правильности представления знаний в таблице, а также на полноту и непротиворечивость и на их основе строят дерево вывода. Экспертная система осуществляет ввод данных (с терминала пользователя, из базы данных), необходимых прикладной программе, и отыскивает подходящую строку в таблице. Если строка найдена, то выполняются дальнейшие действия (вывод результата или переход к анализу следующей строки таблицы). Необходимо отметить, что при таком подходе изменение и дополнение знаний связано с изменением или дополнением строк и столбцов таблицы.

7.3.2. ОПИСАНИЕ БАЗЫ ДАННЫХ ИК

Разработка программных компонентов. При отсутствии готовых компонентов или если они не удовлетворяют требуемым условиям, разрабатываются отдельные программы ИК по принципу модульности, суть которого рассмотрена в предыдущих главах. При этом вопросы создания межмодульных интерфейсов (гл.3), методы построения и управления модульными структурами программ (гл.4) и технологические аспекты применения модульного принципа будут приведены в гл.9.

Отметим некоторые особенности метода интеграции компонентов, связанными с автономной разработкой программ.

1. При проектировании учитываются взаимосвязи не только внутренних объектов (модулей, подпрограмм), но и связи с другими компонентами ИК.

2. Поскольку компоненты ИК известны, то структуры внешних данных вновь разрабатываемого ИК максимально согласовывается со структурами данных других компонентов.

3. Средства реализации интерфейсов могут входить в состав модульной структуры программы.

4. Функции монитора управления компонентами упорядочиваются.

При создании новых программ сложность ИК возрастает, что сказывается на технологии управления разработкой.

Данные, входящие в БД ИК, включают множества данных отдельных компонентов D^i и управляющих данных D^c. Множества D^i, в свою очередь, состоят из внешних и внутренних данных P^i, влияющих на функционирование других компонентов (например, данные сред ЯП описания программ P^i).

БД ИК включает описание всех переменных, входящих в состав P^i. Описание переменной $d^i_j \in D$ состоит в задании ее имени N^i_j, типа данных T^i_j и в определении соответствия между рассматриваемой переменной и областью памяти, где располагается ее значение.

Для описания переменных используется специальный язык описания типов данных (ЯОТ). Не останавливаясь на синтаксических конструкциях этого языка, рассмотрим сущность его семантики, которая должна удовлетворять следующим требованиям:

– допускать описание всех типов данных ИК на единой методологической основе независимо от ЯП, на которых разрабатываются программ ИК;

– отражать современные тенденции в теории типов данных;

– позволять описывать переменные и их типы без привлечения дополнительных языковых средств;

– содержать информацию для размещения данных в БД;

– обладать простотой в изучении и гибкостью в использовании.

Анализируя эти требования, можно отметить, что ЯОТ близок к средствам описания данных в современных ЯП высокого уровня – Паскаль и Модула-2. Методологическую его основу составляет теория структурной организации данных. Средства конструирования типов аналогичны соответствующим средствам указанных ЯП. Однако, ЯОТ имеет ряд особенностей, связанных с использованием его при проектировании ИК. В частности, отметим следующие:

1. Для описания управляющих переменных (множество D^c) введены специальные типы данных. Например, к ним относится тип PROGVAR, рассмотренный в гл.2 при описании модели управления программными объектами.

2. Расширены средства описания файлов, которые могут содержать фиксированное и переменное число записей. Записи могут быть постоянной или переменной длины, иметь регулярную или произвольную структуру. В файл могут явно входить разделители записей и отдельных полей записи. Средства языка должны обеспечивать описание указанных характеристик.

3. Введен специальный тип данных для описания командных строк, которые могут содержать неопределенное число параметров (возможно, пустое) с произвольным порядком их следования. Эта особенность должна быть отражена в описании рассматриваемого типа.

4. Введена возможность указания начальных значений переменных. Входные переменные ИК, отличные от файлов, должны быть инициализированы и переменным заданы начальные значения.

Средства данного языка включают следующие разделы описаний.

Имя описания. В качестве имени выступает имя соответствующей программы.

Раздел констант. В этом разделе константам присваиваются символические

имена, которые используются при обработке конструкций языка.

Раздел описания типов данных. В этом разделе описываются типы данных и их иерархия аналогично соответствующим описаниям в ЯП Паскаль, Модула-2. В описание типа может включаться признак разделителя.

Раздел описания переменных. Приводятся списки переменных с указанием их типов. Переменным могут быть присвоены начальные значения. В частности, могут указываться имена файлов и признаки разделителей переменных.

Конец описания. Это – оператор завершения описания данных для программы.

Транслятор с ЯОТ переводит описание в набор внутренних таблиц, которые служат основой для формирования БД, доступа и обработки данных в рамках реализации моделей информационного сопряжения и управления программными объектами. В процессе совершенствования методов построения ИК структура ЯОТ и средства его обработки будут изменяться в новых условиях применения.

7.3.3. РАЗРАБОТКА МОДЕЛИ СОПРЯЖЕНИЯ И УПРАВЛЕНИЯ ОБЪЕКТАМИ

Модель информационного сопряжения разработана исходя из понятия интерфейса, введенного раннее в гл.1. Алгоритм определения интерфейса состоит в следующем:

1. Все данные, входящие в БД ИК, делятся на классы эквивалентности. Каждый класс определяет совокупность данных, принадлежащих разным множествам D^i и имеющим одинаковое семантическое содержание.

2. Перед вызовом очередной программы происходит преобразование данных, входящих в текущее состояние информационной среды, в представление согласно описанию данных для этой программы. При этом используется набор стандартных функций преобразования типов данных. Типы данных описываются средствами ЯОТ. Внутреннее представление описания служит схемой доступа к данным для процедур, реализующих функции преобразования типов данных.

3. Действия, описанные во втором пункте, повторяются для каждой вызываемой программы.

Алгоритм основывается на описаниях данных и правильном разбиении их на классы эквивалентности. Средства описания данных приведены выше.

Рассмотрим средства выделения классов эквивалентности. Для этой цели используется язык описания классов эквивалентности. Он оперирует только именами переменных и их семантическим содержанием. Как и раньше, синтаксические конструкции мы не рассматриваем. Средства этого языка включает описание следующих разделов.

Имя описания. В качестве имени используется символическое наименование проектируемого ИК.

Раздел описания программ. В этом разделе указывается список имен программ, входящих в ИК. Если необходимо, для некоторых программ приводится список переменных, определяющих командную строку.

Раздел описания переменных. Состоит из нескольких подразделов согласно числу программ, входящих в ИК. Каждому подразделу присваивается имя соответствующей программы. В него входит список имен переменных с указанием их видов и семантического содержания. Переменные делятся на два вида – входные и выходные. Семантическое содержание представляет собой предложение на

обычном языке, определяющее назначение данной переменной для выбранной программы.

Раздел описания классов эквивалентности. Состоит из нескольких подразделов согласно числу построенных классов, каждому присваивается имя соответствующего класса эквивалентности. Подраздел состоит из списка пар – имя переменной, имя программы, в которой она используется. Данный раздел составляется и описывается разработчиком ИК.

Раздел описания ключевых слов. Этот раздел – альтернатива описания классов эквивалентности. Он содержит список слов, которые объявляются ключевыми. По этим словам осуществляется автоматический поиск в разделе описания переменных. Анализируется семантическое содержание переменных и строятся из них списки, связанные с каждым ключевым словом. Затем в процессе диалога происходит корректировка полученных списков и выделение классов эквивалентности. Данный раздел целесообразно включать при большом числе переменных, когда ручная обработка затруднительна. Необходимо отметить, что под ключевым словом понимается строка символов, которая в общем может содержать несколько слов.

Конец описания. Для этой цели используется оператор конца описания.

Транслятор с данного языка формирует набор таблиц, являющихся результатом его работы. Благодаря их формируется внутренняя модель представления данных БД ИК, согласно которой функционирует интерфейс, обеспечивающий информационное сопряжение отдельных компонентов ИК.

Разработка модели управления программными объектами. В гл. 2 подробно рассмотрено назначение, содержание и использование модели управления программными объектами. Здесь мы приведем общее описание языка для обработки данной модели. Как и прежде, синтаксические конструкции языка не рассматриваются. Язык описания модели управления программными объектами имеет следующие разделы описания.

Имя описания модели. В качестве имени описания используется символическое имя, присваиваемое создаваемому ИК.

Описание состоит из нескольких разделов описания условий выполнения программы согласно количеству компонентов, входящих в ИК.

Раздел описания условий выполнения программы. В качестве имени раздела используется имя программы. В раздел включаются описания действий, которые необходимо выполнить до вызова программы и после ее завершения. Выполняемые действия делятся на проверку условий и изменение базы данных. Проверка условий выполняется с помощью набора стандартных предикатов. В качестве примера приведем предикаты проверки:

– готовности (наличия) данных или отсутствия данных;

– выполнения определенных программ, входящих в ИК;

– вычисления логических выражений, заданных над данными из базы данных ИК;

– правильности функционирования (без зацикливания, аварийного завершения и т.п.);

– завершения работы ИК.

В качестве функций изменения БД могут быть:

– установка или сброс признака готовности данных;

– изменение значений некоторых элементов;

– перемещение данных (копирование файлов и т. д.);

– формирование признака завершения работы.

Программа готова для выполнения, если все предшествующие предикаты истинны и все предшествующие функции успешно выполнены.

Конец описания. Завершает описание специальный оператор окончания.

Транслятор с этого языка обрабатывает описание модели и переводит его в систему правил-продукций (пример таких правил приведен в гл. 2). Затем в зависимости от режима работы система продукций оформляется в виде текста программы на ЯП Пролог или кодируется в виде допустимых правил одного из пакетов для создания экспертных систем.

Особенности предикатов и функций ЯОУ состоят в том, что они связаны с данными, входящими в БД ИК. Поэтому условие типа $A < B$ будет записываться в виде *fget (A) < fget (B),* где функция *fget(x)* выбирает значение переменной с именем *x* из БД ИК. В ЯОУ входит небольшое количество базовых предикатов и функций, а остальные могут быть запрограммированы разработчиком ИК в виде их комбинации.

Одним из уровней представления модели управления программными объектами может быть непосредственное составление разработчиком ИК программы на одном из ЯП. В этом случае необходимо использовать функции доступа к данным из БД ИК.

7.3.4. СОЗДАНИЕ СРЕДЫ ФУНКЦИОНИРОВАНИЯ ИК

Предыдущие процессы логического проектирования ИК обеспечивают подготовку отдельных элементов интегрированной среды функционирования.

База данных ИК формируется на основе описаний, составленных на языке ЯОТ. Реальное расположение объектов базы данных определяется требованием на их обработку. Так, файлы, обрабатываемые программами, не копируются, а во внутренних таблицах отмечаются ссылки на них. Командные строки формируются непосредственно перед вызовом соответствующих программ. Для управляющих переменных выделяется область памяти, формируемая на диске, и т. д. Для доступа к базе данных ИК используется набор специально разработанных процедур, включающий функции занесения, выбора, изменения данных различных типов.

База знаний ИК. Ее основой является описание модели управления программными объектами. Результатом обработки описания служит текст программы на ЯП Пролог или набор продукционных правил для пакета программ построения экспертных систем. В первом случае происходит формирование загрузочной программы, которая включается в состав компонентов ИК. Во втором случае в качестве компонента ИК принимается экспертная система, построенная на основе набора продукционных правил. Основными функциями данного компонента являются выбор имени очередной выполняемой программы ИК; изменение в случае необходимости информации в БД ИК; определение условий завершения задач, решаемых в рамках ИК.

Для доступа к данным используется тот же набор процедур, как и при обработке БД ИК. Предикаты и функции реализованы с применением этих процедур.

Интерфейсы. Основой реализации интерфейсов служит описание классов

эквивалентности. Перед вызовом выбранной программы выполняются необходимые преобразования данных текущего состояния информационной среды согласно описанию типов данных соответствующего компонента. Для выполнения этих операций используется набор стандартных функций преобразования типов данных. Функции реализованы с учетом реального представления информации и ее расположения в файлах или оперативной памяти.

Средства управления ИК. К ним относится монитор ИК, выполняющий: организацию диалога с пользователем; вызов программных компонентов ИК для выполнения; контроль состояния среды функционирования и завершение работы.

Монитор служит надстройкой над компонентами ИК и является программой универсального типа.

Созданный на процессе логического проектирования, ИК функционирует согласно следующей схеме алгоритма:

1) Вызов монитора ИК. Монитор вызывается как обычная программа в рамках используемой ОС.

2) Вводится символическое имя ИК. Согласно этому имени активизируются соответствующие описания и выполняются действия по подготовке БД ИК к использованию.

3) Вызов компонента определения имени текущей выполняемой программы. Данный компонент (Пролог-программа или экспертная система) выполняет действия согласно модели управления программными объектами и определяет имя программы или формирует признак завершения работы.

4) Анализ признака завершения работы монитором. Если он установлен, то работа ИК прекращается. В другом случае для выбранной программы выполняются необходимые преобразования данных.

5) Из монитора формируется вызов выбранной программы. После окончания программы повторяются действия третьего процесса. Необходимо отметить, что на том же процессе выполняются операции, определенные в модели управления программными объектами и описанные после вызова программы.

Рассмотренный метод построения ИК ориентирован на вопросы логического проектирования комплекса. Поэтому описанные средства могут использоваться совместно с одной из систем-интеграторов, описанных ранее. В этом случае монитор ИК упростится и часть его функций будет возложена на выбранную систему. Алгоритм функционирования комплекса будет зависеть от применения конкретного интегратора.

7.4. СОВРЕМЕННЫЕ ПОДХОДЫ К СБОРКЕ РАЗНОЯЗЫКОВЫХ КОМПОНЕНТОВ

Приведенная концепция сборочного программирования модулей получила дальнейшее развитие за последние 20 лет за счет включения в проблематику сборки новых объектов – компонентов, КПИ, сервисов и др., а также появления новых общесистемных систем и сред, в которых решаются в той или иной степени вопросы взаимодействия разнородных объектов. В связи с этим глава посвящена рассмотрению новых объектов и их интерфейсов, а также формализованному представлению технологии сборки разноязыковых компонентов в современных средах и системах общего назначения [42-44, 89, 90].

*Парадигма сборки (*композиции*)* включает базовые артефакты программной

инженерии (объекты, компоненты, паттерны, каркасы и формализмы), используемые для описания задач предметной области, разнообразные композиции разработанных компонентов и готовых КПИ в каркасы, контейнеры, а также средства их трансформации и сборки их в конфигурационную структуру для последующего выполнения в среде современных систем.

Объекты сборки – это компонентные элементы и их *интерфейсы*, которые описываются средствами современных ЯП, языков интерфейсов (API, IDL) и визуального языка моделирования (UML – сценарии, патерны, окна и т.п.) архитектуры предметных областей.

Формализмы сборки систематизированы и усовершенствованы с целью подключения к сборке новых объектов:

– алгебраические системы, отображающие типы данных новых ЯП и обеспечивающие взаимодействие разноязыковых объектов;

– стандартизованные механизмы обмена данными, средства их кодирования и декодирования при смене платформы или среды;

– модели представления интегрированных компонентов, отображающие разные структуры ПС (связанной, гибкой в виде каркаса, конфигурации, интегрированной среды и т.п.).

К *артефактам* композиции, кроме указанных элементов, принадлежат также элементы реализованной деятельности в процессе разработки ПС, а именно: описания требований, модели, паттерны и т.п., включая КПИ и готовые подсистемы. Артефакты олицетворяют результат действий в плане реализации ПС и решения вопросов взаимодействия компонентов (на аппаратном и программном уровнях), а также представление их данных в необходимом для исполнения виде. Базовыми действиями над компонентами для получения сложных ПС есть операции сборки: комплексирование, взаимосвязь и взаимодействие. Эти базовые операции предмет дальнейшего рассмотрения формальных моделей и средств обеспечения взаимосвязей и взаимодействия разноязыковых компонентов в современных средах.

7.4.1. МОДЕЛИ СБОРКИ КОМПОНЕНТОВ В СОВРЕМЕННЫХ СРЕДАХ

В настоящее время широко применяются мощные распределенные системы (ONC SUN, OSF DCE, COM, SOM, CORBA, JAVA и др.) [209], предоставляющие разные возможности собирать, взаимодействовать программным объектам и компонентам вместе в структуре ПС, на основе стандарта взаимодействия открытых систем OSI (Open Systems Interconnection).

Модель OSI определяет стандарт для взаимодействия различных систем в процессе обмена данными, унифицированный метод и средства взаимосвязи аппаратного и программного обеспечения, механизмы информационного преобразования данных, а также средства взаимодействия между сервисными службами сетевой среды и компонентами-объектами на уровне сети. Связи между службами и объектами осуществляются посредством протоколов передачи данных по сети.

Особенности взаимодействия компонентов в ONC SUN и OSF DSE. Системы обеспечения взаимодействий компонентов в сетевых средах OSF DC, ONC SUN [5, 89, 128, 133, 209] основаны на механизмах удаленного вызова RPC, задаваемого языками высокого и низкого уровня в виде описания интерфейса

взаимодействующих удаленных компонентов. Интерфейс – это посредник stub, операторы которого (тип протокола, размер буфера данных и др.) обеспечивают передачу данных по сети.

Формальные средства интеграции в этих системах такие:

– оператор передачи сообщений (RPC-вызовы удаленных объектов сети);

– сетевые сообщения между компонентами по передачи данных;

– средства преобразования типов данных с ЯП высокого уровня к типам данных ЯП низкого уровня, а также кодирование и декодирование данных подобно базовых операций (put и get).

Преобразования данных в основном связаны с различиями в архитектуре машин и в транслированных кодах различных компиляторов, выполняются путем отображения релевантных типов данных к двоичному коду компонента и устранения неадекватного перевода программ в ЯП разными компиляторами в выходной или промежуточный код распределенной среды. В случае сложных структур данных (например, деревья, сеть) проводится их линеаризация [41– 42].

Связь компонентов в DCOM и CORBA. Объектная модель DCOM устанавливает связь распределенных объектов и документов, а архитектура OMA системы CORBA - взаимосвязь объектов выполняет брокером ORB через запросы и наличия описания stub-клиента и stub /skeleton сервера. Объекты описываются в современных ЯП, в том числе C, C++. Формализмы сборки компонентов и объектов в системе CORBA такие:

– механизмы передачи запросов удаленным объектам через stub и skeleton;

– обмен данными через сеть и их преобразование в случае отличий в архитектуре и платформе компьютеров среды;

– системы преобразования типов данных для каждой пары ЯП (C↔Смолток, Смолток ↔ Ада, Ада ↔ Кобол, Кобол ↔ Java, Кобол ↔ Ада и др.), которые строятся аналогично ранее описанным в главе 4.

Формализмы в системе DCOM такие:

– механизмы передачи данных (типа RPC-вызов);

– сетевой обмен данными;

– системы передачи данных и преобразования нерелевантных типов данных (C↔C++), а также кодирование и декодирование данных, передаваемых с разных архитектур компьютеров. Процедуры преобразования данных для сред ONC, DCE и CORBA реализованы на языке C++. Интерфейс описывается в языке IDL.

Типы данных ЯП разделены на две группы: базовые и сложные. К базовым типам относятся: целый тип (signed и unsigned integer), 32 и 64-разрядное число с плавающей точкой IEEE, символы ISO Latin/1, логический тип boolean и некоторые другие.

Сложные типы – это interface, struct, union, sequence и array. Тип struct аналогичен описанию в языке C++; sequence и array содержат элементы одинакового типа переменной и фиксированной длины соответственно; тип union семантически соответствует типу union в языке C++ с дополнительными дескрипторами.

Типы данных языка IDL для спецификации параметров компонентов в системе CORBA такие: IN – входные, OUT – выходные, INOUT – результат.

Общая схема передачи параметров разделена на простые типы данных – simple (short, long, unsigned short, unsigned long, float, double, boolean, char, octet, enum).

При изменении типа поля у структуры, также изменяется тип «fixed или variable». Это приводит к переписыванию во многих местах программы этих полей (для клиента и сервера). Определение типа переменной длины – рекурсивное, оно влечет за собой изменение «fixed или variable» для составных типов. Аналогично для клиента и сервера, где используются OUT и RESULT параметры любых типов, необходимо переписывать типы в тексте программы.

Для любого сложного типа данных T вводится специальный тип указателей на данные этого типа - T_var. Схема работы с параметрами на стороне клиента в интерфейсном посреднике одинакова для всех таких типов. Все параметры для объекта сервера передаются через T_var.

Во всех таких типах конструктор типа T_var (T) и оператор присвоения T_var & operator = (T) параметр T использует память для динамического выделения памяти, которая будет отключена после присвоения этому объекту другого значения. Данные копируются перед заполнением их в T_var. Например, заполнение типа String_var var: CORBA::String_var var = «some string», где копирование строк выполняется с помощью функции string_dup: CORBA::String_var var = CORBA::string_dup («some string»).

Для всех других типов данных функция string_dup отображения не предусматривает. Все вопросы решает пользователь.

Заполненный по умолчанию T_var не может использоваться для доступа к данным типа T, поскольку в нем до первого явного присваивания значения T сохраняется нулевая ссылка. Поэтому не заполненный T_var не может использоваться для параметров OUT.

Средства преобразования типов данных в среде JAVA. Язык JAVA и операторы вызова удаленных методов RMI позволяют проектировать распределенные приложения и обеспечивать их взаимодействие. Виртуальная машина работает с byte-кодами компонентов в других ЯП и, таким образом, обеспечивает взаимодействие компонентов в ЯП JAVA и C++.

Формализмы в системе Java:

– оператор вызова удаленных методов RMI;

– сетевой обмен данными между удаленными компонентами;

– виртуальная машина для интерпретации битовых кодов компонентов компилятора C++ в среде Java.

Подходы к преобразованию форматов данных. На каждой платформе компьютера используются соглашения о кодировании символов (например, ASCII, EBCDIC) о форматах целых чисел и чисел с плавающей точкой (например IEEE, VA, IBM и др.). Для представления целых типов (short, long) используется дополнительный код, для типов float и double – стандарт ANSI / IEEE, а для char - множество значений ISO Latin/1 [5, 133].

Порядок расположения байтов зависит от структуры процессора платформы (Big Endian или Little Endian), в частности от старшего к младшему байту и от младшего к старшему (например, в сетях Ethernet байты кодируются с младшего по значению бита). При этом существуют процессоры, которые поддерживают обе возможности (UltraSPARC, PowerPC) в зависимости от требований ПО. При передаче данных с одной платформы на другую учитывается расхождение в задании порядка байтов.

Для форматирования данных есть XDR–стандарт, который обеспечивает

преобразование данных, передаваемых на другие платформы (SUN, VAX, IBM). На одной платформе программы, написанные на разных ЯП, могут использовать одни и те же данные в XDR–формате, хотя компиляторы их выравнивают по-разному. Кодирование (code) или декодирование (decode) данных выполняется с помощью XDR-процедур к виду представления в C++.

Кодирование – это преобразование из локального представления в XDR–представление и запись полученного результата в XDR–блок. Декодирование – это чтение из XDR-блока и превращение данных в локальное представление платформы, которая его принимает. Библиотека содержит процедуры форматирования для простых и сложных типов данных XDR–стандарта. Они используются для создания новых процедур, поддерживающих специфические локальные типы данных и сложные структуры. Для выравнивания данных в памяти используется стратегия, основанная на выборке из адреса данного, кратного действительному размеру в байтах (2, 4, 8, 16). Данные выравниваются по наибольшей длине, необходимой для размещения значений базовых типов. В результате в памяти могут появляться «дырки», которые должны устраняться. При обработке сложных структур данных компиляторы оптимизируют эти поля с помощью специальных процедур и функций из состава системных средств ОС.

Эти процедуры обеспечивают анализ форматов представления данных и необходимое их преобразование к формату платформы, которая их принимает. Соответствующий компилятор генерирует к ним обращение, базируясь на стратегиях выравнивания и размещении данных для платформ клиента и сервера. Принимающая платформа обрабатывает полученные данные, а потом декодирует их обратно к виду формата платформы, которая отправила эти данные.

Наиболее общим подходом к оптимизации процедур преобразования форматов данных – это методы интерпретации и компиляции. При интерпретации переданный код формата данного более компактный, а процедура преобразования работает медленнее. В процессе компиляции форматы и структуры данных существенным образом увеличиваются, хотя соответствующие процедуры работают намного быстрее.

Таким образом, преобразование форматов данных для передачи по сети включает взаимно обратные процедуры: кодирование и декодирование.

Преобразование типов данных в TypeCode. Это средство обеспечивает преобразование значений базовых типов с помощью набора базовых примитивов в процедурах преобразования сложных типов и форматов представления данных:

– дополнительный код для представления целых чисел;

– числа с плавающей точкой (стандарт ANSI / IEEE);

– символы ISO Latin / 1;

– значения базовых типов, выравниваемых независимо от типов или данных, которые реализуется современными компиляторами;

– дополнительные базовые типы языка IDL (64-разрядный целый тип – signed и unsigned, как тип двойной точности и др.)

Процедуры преобразования типов реализуются путем интерпретации TypeCode для каждого типа языка IDL и интерфейса с методами доступа к сохраненной информации. , он содержит поле со значением типа TCKind и дополнительные параметры, которые отвечают этому значению. Тип TCKind определяет множество видов TypeCode базовых типов (tk_long, tk_float) и вид конструирования составных

типов (tk_struct, tk_sequence).

Для каждого типа данных, определенного пользователем, константы TypeCode порождаются компилятором по спецификации в языке IDL.

Преобразование данных осуществляется с помощью процедур encoder ( ) и decoder ( ), интерпретатора TypeCode.

Параметрами процедуры encoder( ) есть данные для преобразования, которое отвечает TypeCode, и указатель на буфер.

Таким образом, достоинством метода преобразования на основе TypeCode, есть компактность выполненного кода, унифицированность использования и единообразие при работе с данными любого типа и произвольной сложности. Интерпретатор позволяет превратить данные, типы которых не известны при компиляции или во время выполнения.

Типизированные функции. Преобразование сложных данных осуществляется с помощью функций отображения типов, описанных спецификациями IDL. Трансформация данных типа struct, например, включает последовательное преобразование всех ее полей, в порядке, указанном в спецификации функции в языке IDL, в язык C++. Эту функцию выполняет компилятор IDL, порождая файлы отображения в соответствующие конструкции C++, набор вспомогательных процедур, необходимых для обращения к брокеру ORB. Для каждого типа IDL имеются соответствующие процедуры их преобразования в C ++.

Функции преобразования базовых типов учитывают информацию о границе выравнивания и размере данных, совпадающими с методами класса CDR, а также реализуют преобразования составных типов, имеющих вложенную структуру, с помощью inline–подстановок. Данные типа array преобразуются специальными функциями и процедурами для простых типов данных.

Типизированные функции и процедуры такие, как кодирование и декодирование, имеют симметричные структуры с точностью до базовых процедур и других специфических действий, используемых для отображения типов данных языка IDL в тип данных языка C++.

7.4.2. ВЗАИМОДЕЙСТВИЕ КОМПОНЕНТОВ В СРЕДЕ JAVA

Средства интеграции компонентов в JAVA. Основные типы компонентов в языке JAVA – это проекты, формы (AWT компоненты), beans компоненты, COBRA компоненты, RMI компоненты, стандартные классы–оболочки, JSP компоненты, сервлеты, XML– документы, DTD документы и файлы разных типов и др. [25]. Интерфейс является частью спецификации названных компонентов и способствует проведению интеграции компонентов в среде системы JAVA.

Шаблон развертывания представляет собою скрытую часть и необязательную часть абстракции компонента, который может быть повторно использован в одной или многих средах и для этого он имеет несколько шаблонов отладки. К спецификации компонента могут добавляться новые шаблоны интеграции или изменяться старые шаблоны. В некоторых классах ПИК параметры интегрирования в новую среду включаются в интерфейс компонента, что ограничивает способность компонента адаптироваться к этим средам и тем самым уменьшается круг задач, в которых он может повторно использоваться.

Проекты как средство композиции компонентов. Создание нового проекта состоит в задании конфигурации системы с помощью компонентов JAVA и

обеспечении их взаимодействия следующими шагами:

– скомпилировать разные файлы с разными JAVA компонентами одной командой;

– установить основной компонент (класс) в проекте, который задает шаблон кооперации других компонентов в проекте;

– установить уникальную конфигурацию для каждого отдельного проекта,

– поддержать соответствующую файловую систему,

– установить уникальные типы компилирования, выполнения и отладки;

– подключить к работе иерархию окон.

Базовые операция проекта – это создание нового проекта, импорт компонентов из другого проекта, создание новых компонентов с помощью "Мастера шаблонов", компиляция, выполнение и отладка группы подключенных к проекту компонентов как единой композиции. Проект обеспечивает разработку, сохранение и корректировку шаблона для поддержки взаимодействия разных типов компонентов при решении одной задачи. К шаблонам повторного использования относятся:

– BlankAntProject, определяющий первоначальный бланк проекта, в котором не содержится ни одного класса или пакета, но разрешается подключение классов и пакетов схему проекта;

– SampleAntProject предназначен для конфигурирования общей схемы проекта в виде иерархии системы файлов, ее корневого узла с последующим добавлением в нее компонентов, выполнять или доработки отдельных компонентов.

– CastomTask обеспечивает создание нового проекта, начиная с формирования первоначального класса в нем.

Классы – основа JAVA, порождаются с помощью ключевого слова Extends, после которого указывается тип компонента (например, JApplet). В проектах используются основной класс, с которого начинается выполнение проекта, и вторичный класс. К основному классу относится Class, Main, Empty (пустой класс), как шаблон типа:

– exception для создания класса, его исключений и сообщений об ошибках, которые могут быть обнаружены в программе;

– persistence Capable для отображения реляционной схемы БД без подключения к MySQL;

– interface – шаблон для создания нового JAVA интерфейса, который можно использовать любом классе через ключевое слово implements.

Для построения классов с помощью шаблонов используются стандартные классы–оболочки (Boolean, Character, BigInteger, BigDecimal, Class), а также класс строчных переменных, класс–коллекция (Vector, Stack, Hashtable, Collection, List, Set, Map, Iterator) и класс–утилита (Calendar – работа с массивами и со случайными числами).

Формы. Интерфейсы компонентов содержат методы работы с графическими объектами и классы, реализующие эти методы. Они подключаются к AWT библиотеке классов, каждый из которых описывает отдельный графический компонент, применяемый независимо от других элементов. В AWT имеется класс Component, в котором графический компонент – это экземпляр этого класса. При выводе графического элемента на экран он размещается в окне дисплея, как потомок класса Container. Библиотека AWT содержит формы, каждая из которой представляет собою контейнер для размещения графических элементов интерфейса

пользователя, а также систему классов Abstract Window Toolkit для построения абстрактного окна.

AWT форма построена на базе “тяжелых” интерфейсов (peer–интерфейс), а Swing формы – на базе “легких” интерфейсов. В разных средах AWT компоненты имеют вид, соответствующий данной среде, а Swing компоненты сохраняют этот вид (“plaf” – Pluggable Look and Feel) за счет того, что они разработаны средствами языка JAVA независимо от платформы. Все упомянутые окна применяются как контейнеры, к которым можно добавлять более простые графические элементы интерфейса (кнопки, меню и т.п.). Интеграция простых компонентов осуществляется на панели графики, изменения которых выполняются автоматически.

Аплет – это небольшая программа, доступная на Интернет сервере, автоматически устанавливается и выполняется веб–браузером или Appletviewer пакета JDK (Java developer Kit). Аплеты не выполняются JAVA интерпретатором, а работают в консольном режиме. После компиляции аплет подключается к HTML файлу, использующий тег <applet>. Компонент в языке JAVA Applet поддерживается набором стандартных методов инициализации, запуска, подключения аплета в требуемый веб контекст для работы с URL адресами и объектами типа Image и др.

Взаимодействие компонентов в системе JAVA. Для обеспечения взаимодействия разных типов компонентов используется механизм вызова удаленного метода RMI, который дополняет язык JAVA стандартной моделью EJB (Enterprise Java Beans) компании SUN. К ней подключены классы языка JAVA, определения их атрибутов, параметров среды и свойств группирования компонентов в прикладную программу для выполнения на виртуальной машине JVM. Механизм развертывания JAVA–компонентов типа beans на сервере базируется на программах в исходном языке, а сервер создает для них оптимальную среду для выполнения задач EJB.

Для реализации и повторного использования КПИ типа beans в системе разработаны следующие шаблоны в Java for Forte:

– Beans для создания нового компонента и формирования каркаса компонента с простыми свойствами и возможностью автоматического изменения этих свойств;

– BeanInfo для интеграции beans компонентов и обеспечения взаимодействия;

– Customizer для создания панели, на которой размещаются элементы, которые со временем можно использовать для управления конфигурацией beans компонентов;

– Property Editor для создания класса, который используется во время проектирования и редактирования свойств beans компонентов.

Таким образом, в среде системы JAVA содержится богатый набор средств поддержки компонентного подхода и решения проблем интеграции и взаимодействия их в разных средах, совместимых с системой JAVA.

В настоящее время имеются механизмы обеспечения связей языков Java и C++. Они основываются на преобразовании Java классов, библиотеке классов C++ и языке описания интерфейса Java Native Interface. Основа взаимодействия компонентов в этих языках – аппарат установления соответствия класса интерфейса в C++ с Java-классом интерфейса. Транслятор C++ продуцирует Java-прокси в C++класс, результат которого – код Java–C++ является

интероперабельным. Реализация системы конвертирования релевантных типов данных между разноязыковыми компонентами (Java↔C++, C++↔Java и др.) и при условии, что они расположены на разных платформах или гетерогенных средах, основывается на дополнительных описаниях компонентов и соответствующих доработках в компиляторах с этих ЯП для настройки к особенностям платформ гетерогенной среды (Подробнее в Приложении 3).

7.4.3. ОБЕСПЕЧЕНИЕ ВЗАИМОДЕЙСТВИЯ КОМПОНЕНТОВ В MS.NET

Общая характеристика платформы Microsoft.NET. Это интегрированная система средств разработки, развертки и выполнения сложных распределенных ПС. Основа системы .Net – MS.NET Framework, являющийся своеобразным каркасом средств и технологий разработки и выполнения ПС [40, 176].

Ключевые характеристики MS.NET такие:

– 1. Платформа MS.NET содержит готовые компоненты и интегрированную среду разработки для поддержки *многоязыковой разработки* ПС в разных ЯП (C#, C++, VBasic.NET, Java#).

– 2. MS.NET имеет средства для снижения сложности ПС за счет компонентного их представления.

1. Технологическая платформа MS.NET обеспечивает проектирование и реализацию ПС на разных ЯП с использованием библиотеки CLR (Common Language Runtime).

2. Любой программный код на новой платформе является *управляемым (*managed code) и компилируется в бинарный вид .NET runtime, что обеспечивает интеграцию кодов, написанных на разных ЯП платформы Windows.

Платформа .NET состоит из нескольких основных компонентов:

– ОС Microsoft (Windows 2000/XP/ME/), как базовый уровень платформы MS.Net;

– серверы MS.Net (.Net Enterprise Servers) позволяют уменьшить сложность разработки сложных ПС (Например, Application Server, Exchange Server, SQL Server);

– сервисы MS.Net (.Net Building Block Services) – это готовые "строительные блоки" для сложных ПС;

– интегрированная среда Visual Studio.NET (VS.NET) – верхний уровень MS.NET, что обеспечивает создание сложных ПС на этой платформе.

Подсистема MS.NET Framework – ядро платформы MS.NET – обеспечивает построение и выполнение приложений. В ее состав входят: общеязыковая среда CLR и библиотека классов FCL (Framework Class Library), состоящая из наборов классов:

– для работы со строками, вводом-выводом данных и параллельного вычисления и т.п.;

– для работы с данными через SQL-запросы, ADO.Net и XML;

– Windows Forms для создания обычных Windows-приложений с использованием стандартных элементов управления Windows;

– Web Forms для быстрой разработки Web-приложений в среде стандартного графического интерфейса пользователя;

– Web Services для создания распределенных компонентов-сервисов, доступ к которым может быть организован через Интернет.

Значительное количество классов библиотеки FCL можно классифицировать в *пространстве имен* (Namespace). Например, пространство в Windows.Forms позволяет задавать формы окон, на которых размещаются элементы управления.

MS.NET Framework подсистема включает общеязыковую среду с библиотекой mscorlib.dll, как головная в .NET CLR. Среда CLR реализует управление памятью, типами данных, межъязыковым взаимодействием, развертыванием (deployment) приложений. Эта среда активизирует выполняемый код, проверяет безопасность его выполнения и управляет памятью, а именно, автоматическим ее освобождением с уборкой мусора.

Любой компонент на ЯП трансформируется к обобщенной спецификации типов CTS (Common Type System) в среде .NET, которая содержит все типы данных, определяет их взаимосвязи и сохраняет их отображение в системе типов .NET. Компилятор программ в ЯП создает файл на языке CIL, который ассемблируется (assembly) в сборный и переносной (Portable Executable) код.

Файлы с расширениями exe или dll – это программный код на языке CIL с дополнительными служебными метаданными о типе сборке, версии, ссылках на внешние компоненты и т.п. Такие файлы перед своим выполнением проходят определенную настройку для обеспечения работы в условиях конкретной платформы с помощью JIT-компиляторов (Just-In-Time compilers) среды CLR. CIL-код используется для перевода на промежуточный язык и машинный (native) платформы выполнения.

Таким образом, компиляцию модулей на ЯП осуществляют два процесса. На первом – создается независимый от платформы PE-файл (управляемый модуль) на промежуточном языке, метаданные и другая необходимую информацию. На втором – компиляция и верификация модулей с помощью JIT (Just In Time Compiler), а также формирование исходной сборки для конкретной платформы.

Универсальная система типизации данных. Система типов в Microsoft .NET представлена в виде иерархии с увеличением общности снизу вверх (рис 7.3).

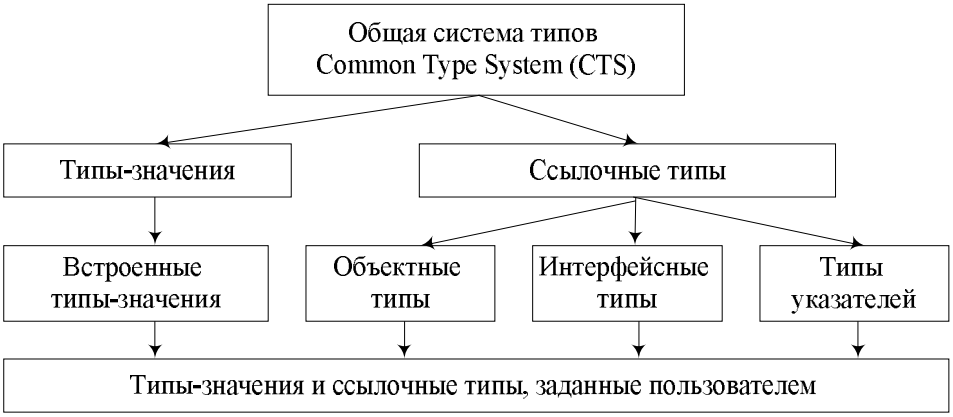


Рис. 7.3. Структура системы типов MS.NET

В этой системе выделены две группы типов: типы-*значений (*value type) *и типы-ссылок (*reference type). Она – общая для всех ЯП, которые выполняются в .NET и задаются пользователем в своей программе на ЯП. В системе существует механизм отображения типов CTS в типы конкретных ЯП и наоборот.

Типы-значения – это статические типы, память для которых выделяется в стеке

и освобождается после завершения работы программы. Эти типы встроены в CTS, их значения могут занимать память от 8 до 128 байтов, не принимают участия в наследовании и копируются при присвоении им значений. Перечень этих типов и память, которую они занимают, приведены в таблице 7.2.

Типы-ссылки или ссылочные типы используют указатели на объекты, которые они типизируют, а также механизмы централизованного хранения и освобождения памяти. Объекты этого типа – динамические, память под них выделяется из «кучи» и освобождается после уничтожения, «уборки мусора».

Ссылочные типы включают: объектные типы (object type); интерфейсные типы (interface type); типы-указатели (pointer type).

Та б л и ц а 7.2.

Соответствует FCL-типу	Память под тип данных
System.Sbyte	Целый.8-разрядное со знаком. Диапазон значений: –128 ... 127
System.Byte	Целый.8-разрядное без знака. Диапазон значений: 0 ... 255
System.Int16	Целый.16-разрядное со знаком. Диапазон значений: –32768 ... 32767
System.UInt16	Целый.16-разрядное без знака. Диапазон значений: 0 ... 65535
System.Int32	Целый.32-разрядное со знаком. Диапазон значений: –2147483648... 2147483647
System.UInt32	Целый.32-разрядное без знака. Диапазон значений: 0 ... 4294967295
System.Int64	Целый.64-разрядное со знаком. Диапазон значений: –9223372036854775808 ... 9223372036854775807
System.UInt64	Целый.64-разрядное без знака. Диапазон значений: 0 ... 18446744073709551615
System.Char	16 (!) разрядный символ UNICODE
System.Single	Плавающий. 32 разряда. Стандарт IEEE
System.Double	Плавающий. 64 разряда. Стандарт IEEE
System.Decimal	128-разрядное значение повышенной точности с плавающей точкой
System.Boolean	Значение true или false

Эти типы выступают в роли спецификации формальных параметров, а интерфейсный тип обеспечивает связь разноязыковых компонентов.

Типы CTS могут использоваться после инициализации (с учетом метода вызова, операций get и set и т.д.), преобразовываться и объединяться в пространства имен System.

Компонентная модель MS.Net. Эта модель реализует *компонентный подход* к проектированию приложений путем сборки объектов на основе интерфейсов (или фрагментов программ), представляющих собой независимые компоненты. Для инсталляции программ создаются инсталляционные комплекты в форме *сборок*.

Каждый тип сборки имеет уникальный идентификатор – номер версии сборки, а каждый программный проект формируется в виде сборки, как самодостаточный компонент для развертывания, тиражирования и повторного использования.

Между сборками и пространствами имен существует следующее соотношение. Сборка может включать несколько пространств имен и в то же время, пространство имен может занимать несколько сборок. Сборка может иметь в своем составе как один, так и несколько файлов, которые объединяются в манифест сборки, аналогично содержания книги. Манифест содержит метаданные о компонентах сборки, идентификатор автора и версии, сведения о типах и зависимости, а также режим и политику использования сборки. Метаданные типа манифест описывают все типы, которые представлены в сборке.

В результате компиляции программного кода в среде вычислений .NET создается сборка, или, так называемый, модуль. При этом сборка существует в форме выполняемого файла (с расширением EXE), или файла динамической библиотеки (с расширением DLL). В состав сборки входит и манифест.

Соответствие типов C# и FCL. Между именами *простых* типов в C# и именами FCL-*типов* существует взаимно однозначное соответствие (табл.7.3). В ней установлено соответствие типов данных языков C# и FCL. Запись в графе "Отвечает FCL-типу" указывается пространство имен, которое содержит объявление соответствующего типа. Данная таблица используется при компиляции и сборке компонентов, которые описаны в приведенных ЯП.

Таблица 7.3.

Тип C#	Отвечает FCL-типу	Значение типа данных
Sbyte	System.SByte	Целый.8-разрядное со знаком. Диапазон значений: –128 ... 127
Byte	System.Byte	Целый.8-разрядное без знака. Диапазон значений: 0 ... 255
Short	System.Int16	Целый.16-разрядное со знаком. Диапазон значений: –32768 ... 32767
Ushort	System.UInt16	Целый.16-разрядное без знака. Диапазон значений: 0 ... 65535
Int	System.Int32	Целый.32-разрядное со знаком. Диапазон значений: –2147483648 ... 2147483647
Uint	System.UInt32	Целый.32-разрядное без знака. Диапазон значений: 0 ... 4294967295
Long	System.Int64	Целый.64-разрядное со знаком. Диапазон значений: 9223372036854775808 ... 9223372036854775807
Ulong	System.UInt64	Целый.64-разрядное без знака. Диапазон значений: 0 ... 18446744073709551615
Char	System.Char	16 (!) разрядный символ UNICODE
Float	System.Single	Плавающий. 32 разряда. Стандарт IEEE
Double	System.Double	Плавающий. 64 разряда. Стандарт IEEE
Decimal	System.Decimal	128-разрядное значение повышенной точности с плавающей точкой
Bool	System.Boolean	Значение true или false

7.4.4. АНАЛИЗ ВЗАИМОДЕЙСТВИЯ РАЗНОЯЗЫКОВЫХ КОМПОНЕНТОВ ПО БЕЮ

Проблема взаимодействия разноязыковых программ на множестве современных ЯП (C/C++, Visual C++, Visual Basic, Matlab, Smalltalk, Lava, LabView, Perl) реализована в работа И. Бея [28] и представлена разными вариантами и конкретными примерами связей пары ЯП из этого множества с помощью практически реализованных функций преобразования и методов обращения к ним из программ на одном языке к программе на другом языке (табл. 7.4.).

В этой таблице представлены варианты взаимосвязи разноязыковых программ в указанных ЯП и виды интерфейсов между парами современных ЯП. Дадим им краткую характеристику.

Интерфейс между Visual Basic и другими ЯП осуществляется с помощью оператора обращения, параметрами которого могут быть текстовые строки, значения, массивы и другие типы данных. Их обработка выполняется функциями Windows API, API DLL и операциями преобразования типов данных. В качестве примера приведена схема обработки Интернет-приложений, заданная HTML-страницами Basic Visual, которые размещаются в веб-браузере и базах данных.

Matlab содержит средства для решения задач линейной и нелинейной алгебры, операций над матрицами и обеспечивает математические вычисления с помощью Matlab Compiler, Matlab C++, Matlab Libriary, Matlab Graphic Library. Схема независимого применения в среде Matlab с интерфейсом VC и Matlab, создает MatlabCompiler путем преобразования программы в формате Matlab (M-файлы или M-функции) в формат языка C. Сформированный файл вызывается из программы в C++ и преобразуется к виду архитектуры компьютера, куда отправляется результат.

Базовые средства Smalltalk обеспечивают создание приложений в среде VisualWorks и содержат модель приложений, методы объектов и сообщения для передачи значений внешним объектам. Модель включает в себя функции DLL из класса внешнего интерфейса, которые взаимодействуют с библиотекой C++.

Таблица 7.4.

Средство описания программы	**Язык взаимодействия**	**Вид интерфейса**
Visual Basic	- ANCI C - C, C++ - Windows API - DLL - VisualBasic 6.0 - Win 32 -API Viewer	Платформо-ориентированные функции Программный интерфейс Динамическая библиотека функций Интерфейс между Visual Basic Функции обработки событий Интерфейс в API
Matlab	- C C++ - Matlab Engine - Matlab в JNI - Visual Basic 6.0 - Java	Вызов приложения из среды Встраивание функций в VC++ Использование интерфейса JNI Функции из Matlab Функции в Java
Smalltalk	- C++ - Matlab - Start V1	Модель приложения в Visual Works Функции графической библиотеки Библиотеки C, C++ и процедуры Visual Works
Lab View	- ANCI C - Visual C++ - Visual Basic 6.0 - C C++	Интерфейс VI и API Связь Visual C, DLL, Obj Lib C, C++ Интерфейсные функции драйвера
JAVA	- C, C++ - Visual C++ - Matlab	Платформо-ориентированные функции Библиотеки функций в C++, C Функции в JNI
Perl	- C, C++ - API - Visual C++	Платформо-ориентированные функции Программный интерфейс Интерфейсные функции в C++

Система LabView предназначена для автоматизации производственных процессов, сборки данных, проведения измерений и управления созданием программ, которые взаимодействуют с аппаратурой. В ее состав входят прикладные средства, тестирование программ и драйверы взаимодействия с аппаратурой, которые запускаются с пульта. Система взаимодействует с ANS C, Visual Basic, Visual C++ Lab Windows/CV. Эти средства расширяют возможности создания систем реального времени и позволяют выполнять с помощью функций

связи измерения аппаратуры типа термометров, переключателей и т.п.. Результаты измерений могут передаваться в сеть.

Среда Java содержит в себе инструменты взаимодействия со всеми ЯП, приведенными во втором столбце таблицы.

Язык Perl – это язык описания сценариев для взаимодействия с Интернетом, управления задачами и созданием CGI-сценариев на сервере в системе Unix. Этот язык имеет интерфейс с языками C, C++, Visual Basic и Java. Интерпретатор языка Perl реализован на языке C, и каждый интерфейс с другим ЯП рассматривается как расширение процедурами динамической библиотеки. Оператор вызова программы в C или C++ обеспечивает преобразование ее в специальный код, который размещается в библиотеке интерпретатора Perl. Сам интерпретатор может быть включен в Win32 для программы на C/C++.

Таким образом, рассмотренные примеры взаимодействия практически проверены и ими можно пользоваться на практике в виде шаблонов взаимодействия разноязыковых компонентов.

7.4.5. СТАНДАРТ ISO/IEC 11404 - 2007. ТИПЫ ДАННЫХ ОБЩЕГО НАЗНАЧЕНИЯ

Для установки взаимосвязей между разнородными компонентами, которые реализованы разными ЯП для современных сред, разработан стандарт, который предоставляет возможность независимо от ЯП специфицировать разные сущности и типы данных для их передачи между компонентами. Этот стандарт содержит описание всех типов данных, механизмы их агрегации, упорядочение и преобразование на внешнем и внутреннем уровнях представления компонентов. Это альтернатива известным языкам описания интерфейсов – IDL, API, RPC, и в будущем он постепенно может занять доминирующее положение в программировании.

Основа стандарта – язык LI (Language Independent), метод генерации новых типов данных, механизмы преобразования типов данных ЯП в LI-язык, и наоборот. Стандарт предлагает специальные правила и операции генерации примитивных типов данных и типа объединение LI-языка в более простые структуры данных ЯП. Параметры интерфейса определяются средствами языков IDL, RPC и API.

Независимые от ЯП типы данных стандарта разделены на примитивные, агрегатные и те, которые сгенерированы. В этот язык включено семейство и генератор типов данных в форме XML-документов. Все существующие типы ЯП и общие типы данных ориентированы на генерацию других типов данных. Описание типов данных задается в разделе «declaration» обших типов данных, объявленных типов данных, объявленных генератором.

Раздел объявления типов данных включает описание, переименование существующих типов. Каждый тип данных имеет шаблон, который содержит в себе описание и спецификатор типа данных, значение в пространстве значений, синтаксическое описание и операции над типами данных. Для объявленного типа данных задается шаблон, который содержит синтаксическое описание, спецификатор типа данных, значение в пространстве значений и операции над типами данных.

Средствами языка LI описываются параметры вызова и интерфейсы,

необходимые при обращении к стандартным сервисам и готовым программным компонентам. LI-язык предполагает такие виды преобразования данных:

– внешнее преобразование типов данных ЯП в LI-тип данных;

– внутреннее преобразование с LI-типа данных в тип данных ЯП;

– обратное внутреннее преобразование.

Внешнее преобразование типов данных и генераторов типов данных это:

а) обеспечение связи каждого примитивного типа данных с одним из LI-типом данных;

в) определение связи путем преобразования допустимого значения внутреннего типа данных в эквивалентное значение соответствующего LI-типа данных;

с) определение существующего значения для любого внутреннего типа данных с целью преобразования в LI-тип данных и извлечение этого значения.

Внешнее преобразование определяет аномалии при идентификации внутренних типов и дает гарантию того, что интерфейс между программными компонентами адекватно задаются сервисным средством.

Внутреннее преобразование связывает внутренний примитивный или сгенерированный тип данных с LI-типом данных с конкретным внутренним типом данных ЯП. Это преобразование имеет такие свойства:

а) для каждого LI-типа данных (примитивного или сгенерированного) преобразование определяет наличие этого типа данных в ЯП;

в) для каждого LI-типа данных преобразование определяет отношение между допустимым значением этого типа и эквивалентным значением соответствующего внутреннего типа ЯП;

с) для каждого значения внутреннего типа данных преобразование определяет, является ли это значение образом (после преобразования) какого-то значения LI-типа данных.

Обратное внутреннее преобразование LI-типа данных заключается в преобразовании значений внутреннего типа данных в соответствующее значение LI-типа ЯП при наличии соответствия и отсутствия двусмысленности. Это преобразование – коллекция обратных преобразований LI-типа данных.

В Приложении D приведено описание интерфейса в LI-языке на примере типов данных языка Паскаль (ISO/IEC 7185-90) и соответствующего преобразования типов данных LI-языка (логический, исчислимый, символьный, целый рациональный и др.) в типы данных языка Паскаль.

Предложенные в стандарте рекомендации, а также средства описания типов данных и методов их преобразования – общие. Новый вариант стандарта ISO/IEC 14044 (General Purpose Datatypes) дает общее описание типов данных, которые могут использоваться в новых ЯП, и иметь программную поддержку для всех типов данных.

7.5. КОРРЕКТНОСТЬ СБОРКИ РАЗНОЯЗЫКОВЫХ КОМПОНЕНТОВ

Определение 7.1. Для корректности метода сборки компонентов необходимо и достаточно, чтобы:

1) существовали операции интеграции такие, что для заданного распределенного графа $G\{K, I\}$ ПС, процесс построения композиционной структуры был конечным;

2) были компоненты во множестве К (при условии принадлежности $k_i \in K$),

которые реализуют необходимые функции, заданные исходным текстом и интерфейсным посредником;

3) содержались средства преобразования k_i в интегрированной среде.

Доказательство данного утверждения сводится к доказательству следующих утверждений.

Лемма 7.1. Существует конечный алгоритм построения программной системы из компонентов.

Схема алгоритма метода интеграции задается графом $G = \{K, I\}$ и используется для создания ПС из компонентов множества K с помощью следующих шагов:

П1. Проверка правильности задания графа $G\{K, I\}$, результат которой

$$\Delta = \begin{cases} 0, \text{ если граф задан правильно,} \\ 1, \text{ если нарушается порядок расположения параметров или их количество.} \end{cases}$$

П2. Проверка согласованности входных параметров вызовов путем проверки принадлежности типов данных в параметрах X, У каждой пары взаимодействующих компонентов (k_i, k_j) в интерфейсах I_{ij}, заданных на графе так:

$$I_{ij} = \begin{cases} 1, \text{ если переменные параметров согласованы,} \\ 0, \text{ иначе.} \end{cases}$$

П3. Выполняется операция интеграции (при $\Delta=0$) для формирования команд трансформации нерелевантных типов данных входных и выходных параметров и обратно.

П4. Обработка обнаруженной ошибки и формирование диагностического сообщения (о несогласованности типов данных, о несопоставимости количества параметров и др.).

Лемма 7.2. Для любой пары компонентов $(k_i, k_j) \in K$ созданная программа $P=\{k_i, I_i, k_j, I_j\}$ будет корректной.

Справедливость этого утверждения вытекает из определения интерфейсного посредника, согласно которому для каждой пары компонентов $\{k_i, k_j\}$ существуют посредники I_i, I_j, из которых собирается программа $P = \{\{k_i, I_i\}, \{k_j, I_j\}\}$ с проверкой корректности каждого параметра при входе в вызываемый компонент и при выходе из него.

Так как количество компонентов графа $G\{K, I\}$ конечно и каждая пара – корректна, то в результате получается и корректность программы P, собранной из этих компонентов.

Лемма 7.3. Создание любой программы Р из компонентов $k_i \in K$ включает:

1) конечное количество операций преобразования данных;

2) компоненты выдачи диагностических ошибок при анализе параметров вызова компонентов;

3) конечное множество компонентов K.

Конечность процесса построения P на основе графа $G\{K, I\}$ представлена леммой 8.3. При этом может возникнуть невозможность связи некоторой пары компонентов из-за неэквивалентности типов данных в интерфейсных посредниках, а именно:

$\exists X_i \in X$ такое, что если $t(X) \sim T$ (тип параметра эквивалентен множеству типов T), то $\rho = 0$, иначе $\rho = 1$.

Из выполнения этого условия следует, что процесс построения P по графу $G\{K,$

I} – конечный и будет завершен сборкою корректной *P* или диагностическим сообщением.

Композиции сборочных компонентов

Композиция (сборка) компонентов включает задание ассоциаций между каркасами, компонентами, функциями, а также утверждениями об элементах и их композициях.

Пусть $P_1, P_2, \ldots, P_n$ – компоненты; T – совокупность композиций;

2. и θ' - некоторые формальные теории о правильности композиции компонентов;

C – отображение композиции $C = P_1 \times P_2 \times P_3, \ldots, \times P_n \rightarrow T$ и отображение композиции θ в θ' .

Для каждого входного предложения S определим условие принадлежности к некоторой формальной теории

$$\text{if } S \in \theta \text{ then } C(S) \in \theta \qquad (7.4)$$

При этом добавим необходимость выполнения условие в производной теории без добавления новых фактов этим компонентам

$$\text{if } S \in \theta \text{ then } C(S) \in \theta' \qquad (7.5)$$

Если элемент композиции не является компонентом P_n , то отображение композиций θ' не приводит к композиционной системе T.

Будем допускать, что композиции со свойствами (7.4) и (7.5) – точные и правильные композиции, в их состав могут входить факты, не связанные с компонентами, или новые объекты и компоненты.

Композиция компонентов может образовывать паттерн, содержащий абстрактный класс – интерфейс. Сложный паттерн является иерархичным объектом, удовлетворяющий условию унификации и композиции объектов. Доступ к абстрактным элементам такого агрегатного объекта обеспечивается итерационным путем.

Проблему композиции выполняет также каркас, который обеспечивает правильное и надежное взаимодействие компонентов. Иными словами, каркас объединяет множество взаимодействующих между собой объектов в некоторую среду для решения определенной конечной цели.

В общем случае сложились четыре возможных класса формальных композиций, элементами которых являются компоненты и каркасы, взаимодействующие между собой.

Композиция компонент-компонент обеспечивает непосредственное взаимодействие компонентов через интерфейс на уровне приложения.

Композиция каркас-компонент обеспечивает взаимодействие каркаса с компонентами, при котором каркас управляет ресурсами компонентов и их интерфейсов. Такое взаимодействие осуществляется на системном уровне.

Композиция компонент-каркас обеспечивает взаимодействие компонента с каркасом типа «черного ящика», в видимой части которого находятся спецификации для его развертывания и выполнения определенной сервисной функции. Такое взаимодействие осуществляется на сервисном уровне.

Композиция каркас-каркас обеспечивает взаимодействие между каркасами, каждый из которых разворачивается в гетерогенной среде, компоненты которой взаимодействуют между собой через их интерфейсы на сетевом уровне.

Компоненты высокого уровня допускают агрегацию компонентов согласно рассмотренной композиции первого класса, а также взаимодействие каркасов и высокоуровневых компонентов.

Метод композиции – это средство или планомерный подход для обеспечения взаимосвязей компонентов в сложных структурах (комплексах, интегрированных, распределенных системах).

Главное в методе композиции компонентов – это понятие межкомпонентного (модульного) и межъязыкового интерфейсов, которые обеспечивают взаимодействие в современных сетевых и гетерогенных средах.

Межмодульный интерфейс остается посредником между двумя взаимодействующими компонентами. Сущность межъязыкового интерфейса остается прежней. Он включает совокупность средств и методов сопоставления структур и типов данных ЯП для организации преобразования переданных типов данных, аналогично тому, как это было рассмотрено в главе 4. Проблема взаимодействия компонентов в разных ЯП в гетерогенной среде решена по-разному, а именно, через отдельные функции преобразования типов данных в ЯП, декодирование (кодирование) данных при передачи с одной архитектурной платформы на другую и т.п. Преобразование передаваемых данных в значительной степени решается с помощью новых средств спецификации интерфейсов компонентов (языков API, IDL) и стандарта ISO/IEC 11404–2007. На основе этих спецификаций устанавливаются связи между взаимодействующими компонентами в современных средах.

Вывод. В главе представлено описание новых возможностей сборочного создания ПС из компонентов, связанных с экспертными системами и методами представления знаний для них. Рассмотрены вопросы создания интегрированных комплексов с применением баз данных и экспертных систем. Основу создания ИК составляют модели сопряжения и управления программ в среде функционирования.

Приведен краткий анализ современных подходов к решению задач взаимосвязей и взаимодействия программных компонентов в современных средах и системах (JAVA, MS.NET, Matlab и др.). Изложены основные положения стандарта о типах данных общего назначения, который позволяет решить вопросы согласования типов данных разнородных компонентов на верхнем уровне по отношению к ЯП.

Глава 8

ОСНОВЫ ПОСТРОЕНИЯ ТЕХНОЛОГИЧЕСКИХ ЛИНИЙ И СРЕДСТВ АВТОМАТИЗАЦИИ ПРОГРАММ

8.1. СУЩНОСТЬ ТЕХНОЛОГИИ СБОРОЧНОГО ПРОГРАММИРОВАНИЯ

В последние годы в программировании интенсивно проводились работы по совершенствованию, развитию и изменению. программного обеспечения больших и малых машин; созданию конкретных «уникальных» ПС реализации отдельных задач прикладных областей, а также по разработке систем поддержки проектирования ПО и технологических комплексов общего назначения. Однако по-прежнему остаются недостаточно решенные важные проблемы программирования:

– постоянный рост стоимости ПО (так, стоимость ПО в 1990г. составляла 30 млрд. дол, а к 2005г.– 100 млрд. дол.) [149];

– невысокая производительность процесса создания ПО (в США ежегодный прирост продуктивности ПО составляет 3 – 8 %, а ЭВМ – 50 %) [174];

– низкое качество ПО, несмотря на возросшие к нему в последнее время требования (так, ВВС США предъявляют жесткие требования к ПО – отсутствие сбоев и корректировок).

Эти данные говорят о том, что имеется отставание в программирования в плане технологии разработки и производства ЭВМ. Увеличение годового роста спроса ПО до 30 %, что соответственно влечет за собой увеличение на 12% персонала (в США оно фактически растет на 3– 4 % , еще более обостряет эти проблемы.

Указанное отставание можно объяснить, прежде всего, тем, что в программировании медленно развиваются методы инженерии технологических процессов разработки ПО и средств их автоматизации (как это сделано в ГОСТ машиностроения), а также недостаточно развитым инженерным уровнем проектирования и разработки, недостаточным качеством планирования и контроля процесса разработки ПО, недостаточным использованием стандартов в области программной инженерии, проведением работ по классификации и унификации КПИ.

В связи со сказанным можно считать, что наиболее актуальная задача современного программирования – создание ТЛ и инструментов поддержки процессов инженерного проектирования и разработки программ и ПС, которые будут использоваться как уникальные в части реализации специфических функций различных предметных областей.

Под *технологией программирования* понимается совокупность принципов, методов, подходов, упорядочивающих процесс разработки программного

обеспечения с заданными характеристиками на всех процессах создания ПС. В такой постановке эффективность процесса создания программного продукта зависит от обоснованности применения методов проектирования автоматизируемой предметной области, принятой организации и методов инженерного управления разработкой ПО (планирование, нормирование, учет и др.), а также от степени оснащенности процессов разработки средствами автоматизации и готовыми КПИ.

Таким образом, разработке ПС должно предшествовать первоначальное определение или выбор технологии, учитывающей специфику и функции реализуемой предметной области. Именно такой подход традиционно существует в промышленности при разработке новых технических изделий, о чем свидетельствует большой прогресс в области роста продуктивности современных ЭВМ (на 40%). Для лучшего понимания сущности технологии программирования дадим определение основных ее элементов и их назначение.

Несмотря на значительный прогресс в области технологии программирования выделим три основных уровня технологий (рис.8.1): системного проектирования, прикладного программирования, конечного пользователя.

Первый уровень – это технология подготовки разработки (ТПР), направленная на определение модели предметной области и реализуемого программного объекта. Для объекта вычленяется набор его состояний ЖЦ, методов, средств и инструментов преобразования этих состояний с целью получения программного продукта. Эту технологию определяют высококвалифицированные специалисты (аналитики), знающие предметную область, создаваемый объект разработки, а также пути его проектирования. Результат их деятельности – особый продукт – конкретная прикладная технология, по которой будет проводиться разработка множества программ из заданного класса. На этом уровне технологии определена последовательность действий и инструментов, которые надо произвести для получения соответствующего прикладного программного продукта для ПрО.

Второй уровень технологии – подходы к разработке прикладных программных элементов по заданной на уровне ТПР прикладного программирования. Этот уровень предусматривает применение методов планирования (составления план графиков), контроля и регулирования хода разработки и оценки результатов труда для достижения высокого качества продукта. Разработку проводят прикладные программисты, а тестировщики и верификаторы проверяют правильность реализации функций и требований к системе. Результат их деятельности – программный продукт, готовый к выполнению, и технология решения задач конкретным пользователем.

Третий уровень – технология конечного пользователя, в соответствии с которой он осуществляет постановки задач и получение результатов решения. Основное требование, предъявляемое к этому уровню технологии, состоит в том, чтобы пользователь работал с прикладной системой в терминологии предметной области.

Главные элементы технологии – объект разработки, инструментальные методы и средства, объединенные в технологические процессы и ТЛ, технологический интерфейс для взаимосвязи технологических средств (модулей) и инженерные методы управления разработкой, которые встраиваются в ТЛ и обеспечивают управляемое создание фунционально-ориентированных программ.

Рис. 8.1. Уровни технологии программирования

Объект разработки – это программный продукт, реализующий определенные функции (задачи) предметной области, процесс проектирования и разработки которого осуществляется соответствующими методами и программными средствами.

Объектом разработки может быть: модуль, программа, комплекс программ, пакет прикладных программ, система и др., т. е. объект либо сам является отдельной конструктивной единицей разработки, реализующей элементарную функцию ПрО, либо состоит из взаимосвязанного их набора.

Как показывает анализ современной литературы для объекта разработки первоначально формируется его модель, в которой специфицируются реализуемые функции и элементы данных, детализируются их связи и отношения, т. е. для объекта определяются его спецификация или прототип.

К объекту разработки в техническом задании предъявляются определенные требования к составу реализуемых функций, ограничениям на такие характеристики, как работоспособность, что означает отсутствие ошибок при выполнении функций (свойство надежности), быстродействие, память, удобство общения (свойство эргономичности), мобильность (свойство переносимости из одной среды в другую) и др.

В зависимости от условий для обеспечения функционирования ПС (например, работать с большим быстродействием в реальном времени или с большой

надежностью в космических, военных системах и т.п.) требования к отдельным свойствам программных объектов отличаются и влияют на организацию процесса разработки [124].

Любой объект имеет начальное (исходное), промежуточные и конечное состояния. Начальное состояние – это исходная модель объекта. Промежуточное – измененное состояние, отличное от начального и конечного состояний объекта, полученное на определенном процессе ЖЦ под воздействием соответствующих данному процессу программных методов и средств. Промежуточным состоянием объекта в соответствии с ГОСТ 36400 являются: эскизный, технический, рабочий проекты.

Конечное состояние объекта – программный продукт, готовый для выполнения требуемых от него функций. Конечным состоянием считается опытный образец, передающийся либо в опытную эксплуатацию, либо на тиражирование (производство).

Следует отметить, что еще не стабилизировался формальный аппарат задания начального состояния объекта, а именно его модели, чего нельзя сказать о возможностях описания промежуточных состояний, для задания которых используются языки проектирования и программирования различного уровня, в том числе языки спецификаций.

Метод разработки (программный метод) – это способ и средства достижения той цели, которая ставится перед объектом разработки. Метод определяет стратегию проектирования или разработки.

Наиболее распространенными методами проектирования и разработки являются: сверху вниз, снизу-вверх, модульный, компонентный, метод сборочного программирования и др.

При нисходящем (сверху вниз) проектировании объекта после определения требований к объекту формируется его функциональная и системная архитектуры. В них декомпозированы все задачи ПрО и построена модель объекта. Задачи, в свою очередь, развиваются до понятий и функций объекта, выражаемых в базовых терминах рассматриваемой ПрО. Каждая функция объекта разрабатывается последовательным уточнением до определения элементов системной архитектуры.

Восходящий метод (снизу-вверх) является обратным нисходящему и начинается с уровня определения элементарных базовых понятий ПрО и формирования из них более крупных понятий, приводящих в конечном итоге к заданию некоторой функции ПрО. Для нее подбирается или разрабатывается один из готовых программных элементов: модуль, модель программы, шаблон, паттерн, КПИ и т. п. В этом плане, данный метод можно считать методом от готового.

Развитием восходящего метода является метод прототипирования [195], при котором для объекта вначале создается «грубый» его образец, прототип из готовых близких по смыслу компонентов. При этом от программ требуется, чтобы они соответствовали функциям объекта без учета эксплуатационных характеристик, т. е. цель прототипирования состоит в том, чтобы отработать «каркас» объекта и его функциональные возможности, а затем постепенно улучшаются характеристики и свойства компонентов. Данный метод в настоящее время начинает широко применяться в практике программирования, так как он дает возможность быстро, опробовать и отработать требования заказчика к программному объекту вместе с разработчиками.

Метод расширения ядра характеризуется начальным выделением множества вспомогательных функций. Наиболее эффективный метод выделения – анализ используемых данных и определение модулей, обрабатывающих различные информационные структуры. Для этого может быть использован метод Джексона, в основе которого лежит принцип соответствия организации программы согласно процессам преобразования обрабатываемых данных.

Несмотря на различие этих методов в стратегиях проектирования, общее, что их объединяет – это модульное или блочное (программное [73]) представление объекта разработки. Суть метода модульного программирования состоит в декомпозиции исходной задачи ПрО на отдельные функции вплоть до элементарных, каждой из которых в общем случае сопоставляется программа или модуль. Каждый модуль должен обладать интерфейсом для его связи с другими модулями и компонентами. Применение данного метода по сравнению с другими дает значительные преимущества в плане организации и управления разработкой, но вместе с тем требует создания определенных механизмов языкового и программного характера для решения проблем интерфейса при сборке модулей и программ в более сложные программные структуры. Расширением данного метода является метод сборочного программирования.

Таким образом, рассмотренные наборы методов проектирования и программирования взаимосвязаны и используют друг друга. Так, в случае применения восходящего метода проектирования для систем обработки данных на разных процессах ЖЦ программного объекта использовались методы структурного и модульного программирования, расширения ядра и др.

Технологический процесс – это взаимосвязанная последовательность операций разработки объекта. Процесс предназначен для перевода объекта из одного состояния в другое соответствующими методами и инструментами.

Характерная особенность многих технологических комплексов – регламентация процессов ЖЦ (ПРОМЕТЕЙ, ТКП, РТК и др.) [151] и технологический интерфейс, способствующий взаимодействию процессов в заданной последовательности. В этих комплексах ТП могут быть не связаны со спецификой функций ПрО (например, таких, как функции ввода, вывода данных и т. п.). Они образуют класс программ общего назначения и используются в разных процессов разработки ПС, ориентированных на любые виды программ. Другая интерпретация процессов заключается в том, что они позволяют реализовать набор типовых функций автоматизируемой ПрО, относящейся к СОД, АСНИ, САПР и др. Такой процесс становится типовым и может использоваться как объект сборки в технологиях, где он является необходимой составной частью. Типовые процессы вместе со специализированными процессами образуют базис любой линии разработки функционально-ориентированного программ.

Технологическая линия – это совокупность технологических систем, технологических модулей (ТМ) и процессов для последовательного производства программ от заданных требований до готового продукта. Она задает технологию разработки функции объекта, представленную совокупностью автоматизированных процессов, последовательно и систематически преобразующих состояния объектов, включая заключительное состояние – готовый программный продукт. Особенностью линии является отражение определенных функций ПрО, реализуемых в виде программ с заданными показателями качества.

Связь ТМ, систем и процессов выполняет технологический интерфейс (ТИ). Он используется при сборке ТЛ из готовых ТМ, которые сделаны специально для отображения специфики реализации ПрО заданного класса. Создание ТЛ по методу сборки основывается на установленной номенклатуре функций, реализуемых ТМ из класса готовых технологических средств.

Для простых объектов количество процессов в ТЛ меньше, чем для сложных. Независимо от сложности объекта основным условием выполнения процессов является их автоматизация. Процессы имеют модельное представление, базирующееся на модели ЖЦ объекта разработки.

Динамика изменения может быть представлена в ТЛ последовательностью процессов, переводящих объект из одного состояния в другое путем использования средств автоматизации процесса. В этом плане каждая конкретно разработанная ТЛ определяет множество допустимых состояний объекта, а на средства автоматизации возлагается функция слежения за переходом в недопустимые состояния.

Инструмент – это программное, языковое или методическое средство, применяемое для задания состояния объекта в некотором законченном виде. В зависимости от процесса ЖЦ и состояния объекта изменяется разными инструментами (например, языки, трансляторы, генераторы и т.п.).

Для описания модели программного объекта используются имеющиеся формальные или полуформальные средства (графические, языковые) спецификации требований и описаний программ [1].

К графическим средствам относятся: таблицы для описания конечных функций; решающие таблицы и структуры (матрицы, векторы и т. п.); графовые структуры (диаграммы взаимодействий и состояний Петри и др.); схемы HIPO, SADT и др.

К языковым – языки спецификаций [4, 41, 58, 144, 155, 171] общего и специального назначения, ориентированные на специальные ПрО (языки описания моделей АСУ в системах АРИУС, ISDOS). К языкам спецификаций общего назначения относятся GDL, PDL, PSL, PSA и др. Они позволяют представлять «каркас» или схему объекта посредством структур управления (ветвления, цикла, вызовов процедур).

Процесс преобразования модели объекта из одного состояния в другое не является полностью автоматизированным, и имеется ряд систем, специально предназначенных для реализации конкретных предметных областей.

Наибольший набор инструментов создан для процесса программирования. К ним относятся ЯП и соответствующие им системы программирования. Более поздние ЯП, такие, как АДА, Паскаль, Си, содержат средства описания абстрактных структур данных и задания структур сложных программ из модулей.

Управление разработкой. Каждая инженерная дисциплина базируется на принципах и методах конструирования (разработки) и промышленного производства продуктов, которые затрагивают как организационные, так и технические аспекты производства. В этом случае программную технологию в последнее время рассматривают с позиций инженерной дисциплины [109, 140, 188 и др.]. Исходя из анализа этих работ, отметим, что основными вопросами

управления инженерией разработки программных объектов являются:

– организация коллектива разработчиков (состав, структура, квалификация и др.);

– планирование работ, трудозатрат и обеспечение роста производительности труда;

– контроль хода разработки и оценка проектных решений в ходе разработки программных продуктов;

– экономические вопросы (стоимость, ценообразование, стимулирование и др.).

8.2. КОНЦЕПЦИЯ ПОСТРОЕНИЯ ТЕХНОЛОГИЧЕСКИХ ЛИНИЙ (ТЛ) ИЗГОТОВЛЕНИЯ ПРОГРАММ

Выше рассмотрены основные понятия и элементы технологии программирования. Многообразие методов, подходов, инструментальных средств и т. д. порождает многообразие программных технологий. Однако есть общее, что объединяет их. Во-первых, общие процессы ЖЦ разработки ПО. Во-вторых, общие концепции разработки программных технологий. Первая особенность рассмотрена в гл. 1, а вторую рассмотрим здесь.

Разработка программной технологии традиционно осуществлялась двумя путями. На первом фиксируется изначально набор инструментальных средств, на базе которого строятся процессы разработки программных систем. Во втором случае, исходя из общих и специфических свойств технологических процессов реализации функций ПрО, разрабатывается или выбирается набор инструментов, необходимых для эффективной их реализации.

Первый путь является наиболее типичным. Вся история развития программирования характеризовалась именно тем, что пользователи (разработчики ПС) пытались применять вновь появляющиеся средства автоматизации (ЯП, системы программирования и СУБД, пакеты прикладных программ и др.) под программные нужды, формируя при этом свои стили и технологии их применения.

При этом зачастую оказывалось, что данные средства не полностью удовлетворяли требованиям разработок и разработчики занимались их развитием и совершенствованием в нужном направлении.

К системам, использующим данный метод, относятся и современные хорошо развитые и широко используемые универсальные технологические системы и комплексы раннего периода автоматизации (РТК, ПРИЗ, ПРОМЕТЕЙ, ТКП и др.) и современные (CORBA, COM, MSF, Microsoft Visual, .NET и др.).

Ранние системы имели направленность на специфику конкретной ПрО с общей регламентациею процессов разработки ПС и хранения истории ведения разработки. Задача пользователя этих систем состоит в том, чтобы использовать предлагаемую технологию и средства ее инструментальной поддержки при реализации конкретной ПрО.

Второй путь состоит в том, что технологические процессы создания программного продукта ориентированы на описание структуры и свойств этого продукта, а процесс получения собственно продукта осуществляется автоматически (например, CORBA, COM, MSF). Данный подход используется при автоматизации конкретных предметных областей: АСУ, СОД, АИС и др.

В результате имеем прямой выход к:

определению типовых технологий (называемых в дальнейшем функционально ориентированными) для класса однотипных функций в некоторой ПрО, имеющих сходные модели программного объекта;

возможности повторного использования ранее определенных элементов функций и программ (заготовки для отдельных элементарных функций ПрО, модели программ, готовые модели и др.);

более гибкой реализации программного объекта с обеспечением требований в задании на его разработку;

специализированной инструментально-технологической поддержке процессов разработки классов однотипных программ с контрольными операциями в ходе разработки и оценки результатов труда на разных процессах ЖЦ прикладного программного обеспечения.

Для иллюстрации данной концепции в качестве ПрО используется СОД, характерной особенностью которой являются большие объемы данных, требующие привлечения инструментария СУБД для их ведения и множества прикладных программ ППО, реализующих специализированные и типовые функции в функциональных различных подсистемах (МТО, ТОР и др.) и приводящих к дублированию работ и выполнению рутинных операций при сборке прикладных программ в целенаправленные комплексы и ППП.

В классе указанных функциональных подсистем СОД типовыми функциями обработки информации являются:

– ввод и контроль данных;

– ведение данных (актуализация и загрузка) в базах данных;

– обработка данных научного, исследовательского, планово-экономического и других видов;

– вывод результатов обработки (в форме значений проблемных переменных, документов и т. п.) и справок о текущем состоянии баз данных.

Отмеченные функции характерны и для таких классов систем, как автоматизированные системы организационного управления, информационно-справочные системы, автоматизированные системы управления и многие другие.

Для автоматизированного создания ППО систем, реализующих приведенный набор типовых функций, при котором разработчики освобождены от многих рутинных операций программирования множества сходных по функциям прикладных программ и их сборки в программные компоненты функциональных подсистем, и поставлена задача исследования и разработки функционально-ориентированных технологий по принципу сборочного программирования.

Исходя из опыта реализации конкретных СОД, наблюдается общность и повторяемость в значительном множестве представлений отдельных функций в каждой подсистеме в виде типовых функциональных элементов разного вида. Это повлияло на развитие метода сборочного программирования для организации связей элементов повторного использования по данным и по управлению через аппарат интерфейсных функций с целью получения программных функциональных компонентов СОД.

Таким образом, ставится задача создания технологий с функциональной ориентацией с помощью метода сборки, требующего определенного развития для удовлетворения потребностей и специфических особенностей реализации рассматриваемой предметной области СОД.

8.3. ОПРЕДЕЛЕНИЕ ПРОЦЕССОВ И ТЛ В КЛАССЕ ФУНКЦИОНАЛЬНЫХ ЗАДАЧ СОД

В силу большого разнообразия программных методов и средств основная задача руководства проектом состоит в том, чтобы процессы разработки удовлетворяли потребностям реализуемых функций и были по возможности автоматизированными. Поставленная задача создания функционально-ориентированной технологии (ФОТ) может быть решена посредством метода, в основе которого лежит принцип сборочного программирования с целью создания требуемой технологии из имеющихся методов и средств, максимально удовлетворяющих целям реализации проекта.

Здесь элементами сборки являются типовые программно-технологические средства и процессы на их основе, реализующие однотипные функции ПрО и оснащенные известными или оригинальными программными методами и средствами. Главный элемент разработки – модель ЖЦ, которая адаптируется к рассматриваемому классу ПрО и использует программные методы и средства для последовательного перехода одного состояния объекта в другое.

Сложность объекта – главный фактор, влияющий на состав технологических процессов в ТЛ. В качестве примера можно привести программный объект – пакет прикладных программ статистики, реализованный в рамках ППО СОД, ТЛ разработки которого включает 13 технологических процессов. Каждый процесс модели ЖЦ данного программного объекта оказался представленным более чем одним процессом. В целом совокупность процессов процесса предназначается для получения нового состояния объекта из отдельных составляющих каждого процесса.

Принципы построения ТЛ. Процесс построения технологии прикладного программирования, ориентированный на реализацию функций ПрО СОД, – трудоемкий и творческий. Проектирование ТЛ выполняют высококвалифицированные специалисты (аналитиками, системные программисты), а также специалисты, знающие реализуемую ПрО, технологические аспекты создания современных прикладных систем, вопросы экономии затрат, повышения производительности труда и улучшения качества программ продукта. Технологический интерфейс способствует выполнению процесса создания отдельных программ для класса и сборки программ (например, поиск и нахождение необходимого КПИ в библиотеке, оценка их функций и границ использования и т.п.), которые реализуют некоторые программные функции в проектируемой системе ПрО.

ФОТ отображается в ТЛ и задает среду прикладного программирования, которая методически, программно и организационно поддерживается специальными программно-технологическими компонентами и ТМ. В класс ТМ входят такие общие технологические средства, как формализация описания задач ПрО, моделирование действий некоторой функции в конкретной среде, формирование тестов, создание рабочей документации и др. К ТМ также относятся различные соглашения группы разработчиков ТЛ (например, о форме представления данных в системе, платформе и др.) и методики выполнения ТМ.

Описание ТЛ должно ориентироваться на эффективное управление разработкой, включающее планирование сроков и трудоемкости, постепенное повышение производительности труда исполнителей за счет создания фондов

технологии (как готовых повторно используемых программных элементов, так и типовых технологических процессов). Входящие в ТЛ процессы должны обеспечивать изменение состояния объекта и оценку показателей качества в процессе разработки объекта.

Принятие технологических решений по составу и структуре ТЛ основывается на результатах анализа ПрО, сопоставлении со свойствами имеющихся аналогов и всестороннего анализа требований заказчика к разрабатываемому программному объекту.

Выделение ТП в ТЛ представляет собой многошаговый итерационный процесс, протекающий в двух (встречных) направлениях; первоначально весь ЖЦ объекта делится на крупные процессы (см. гл.1), затем в рамках каждого процесса происходит выделение более низких уровней (с различной степенью детализации) и, наконец, выполняется группирование ранее выделенных работ по ТП, при этом возможны повторное расчленение некоторых работ и их перегруппирование по ТЛ.

После определения структурного и функционального назначения и способов взаимосвязи выделенных ТП осуществляется анализ имеющихся типовых ТП с целью их включения в разрабатываемую ТЛ. Если такие ТП имеются в технологическом фонде, то проводится их адаптация к условиям применения в данной ТЛ. Адаптация предполагает добавление новых технологических операций, незначительное измененииее существующих операций в ТП, уточнение состава и функций проектно-технологических документов (ПТД), применяемых для фиксации результатов разработки на ТП.

Для каждого ТП определяется организационная структура коллектива, выполняющего разработку по ТП, с учетом профессиональной специализации и квалификационного уровня исполнителей каждой операции. Рекомендуемая на ТЛ организационная структура (фактически модель коллектива разработчиков) уточняется при настройке ТЛ на конкретный коллектив и средства разработки.

Технологические операции и технологические процессы объединяются во взаимосвязанную совокупность с последовательным, параллельным, циклическим или условным режимом выполнения посредством технологического маршрута.

Формализация описания ТЛ. Для описания конкретной функционально-ориентированной ТЛ могут быть использованы язык спецификаций табличного вида и полуформальное описание семантики ТП и ТЛ в виде комплекта технологических документов.

Язык спецификаций включает табличные формы, в которые заносится информация об операциях ТП и процессах ТЛ. Каждая строка таблицы ТП, называемая картой ТП (КТП), специфицирует операцию процесса, входные и выходные данные, представляемые в ПТД, метод, средство и инструмент выполнения операции, категорию исполнителя, метод контроля и оценки результата процесса.

Основное назначение процесса – задание среды изменения модели программы (на рис.8.2) и модели отдельного процесса с входными и выходными данными, определяющими состояния S_i программного продукта на $ТП_i$. Эти состояния определяются начальной спецификацией требований S_0, функциональной архитектурой S_1 и приводят к конечному результату S_n , т.е. к конечному состоянию продукта.

Реализация модели процесса в технологической среде ТЛ включает в себя:

решение задач управления проектом (планирование, контроль процесса и полученных состояний); проверку и оценку показателей качества, которые заданы в требованиях к ПС. Технологический диспетчер среды обеспечивает ведение информации относительно операций контроля результатов выполнения операций ТП и занесение их в базу данных проекта.

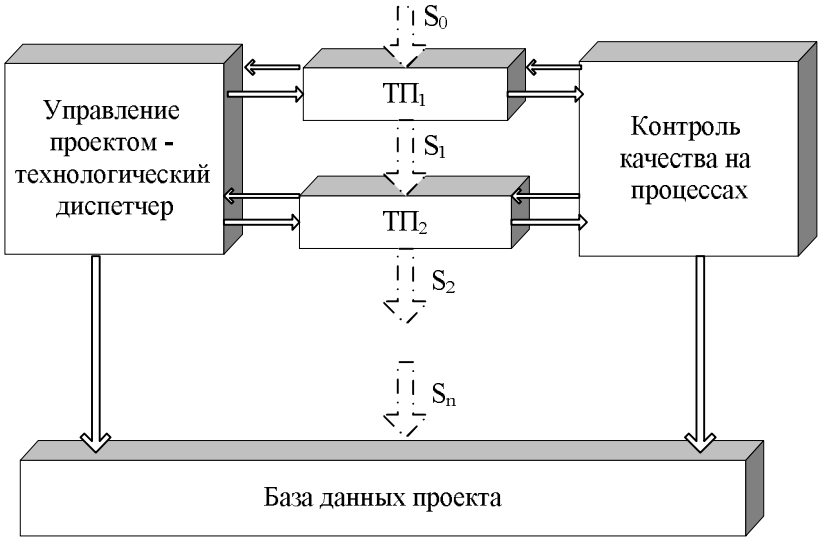


Рис. 8.2. Модель среды разработки по ТЛ

В конкретных условиях проводится настройка ТЛ с помощью параметров, заданных в картах линии или процесса и планов-графиков разработки программных объектов по заданной ТЛ.

8.4. БАЗОВЫЕ МОДЕЛИ ФУНКЦИОНАЛЬНО-ОРИЕНТИРОВАННЫХ ТЕХНОЛОГИЙ

Для проектирования и разработки методом сборки новых ФОТ используется класс взаимосвязанных моделей, позволяющих описать с разных позиций взгляд специалиста предметной области на способы представления ФОТ.

Класс моделей включает модели формализованного представления процессов и линий разработки, состояний программ, отображаемых в ходе разработки моделями следующего вида:

– модель качества ПС ($М_{кач}$);

– систему моделей (бланков) формализованного представления проектных решений в ходе разработки ($М_{пдт}$);

– модель эксплуатационно-программных документов ($М_{эпд}$);

– модели формализованного представления спецификаций технологических процессов и линий ($М_{ТП}$, $М_{ТЛ}$).

Ниже дается их определение.

8.4.1. МОДЕЛЬ ОЦЕНКИ КАЧЕСТВА ПРОГРАММ ПО ТЛ

Качество ПС характеризует его пригодность к использованию по назначению. Для разных типов ПС в классе задач ПрО предварительно разрабатывается

номенклатура показателей и устанавливаются их базовые значения, которые должны быть достигнуты при разработке отдельных прикладных программ по их моделям.

Модель качества $М_{кач}$ – это формализованное представление характеристик качества, их свойств и способов оценки на процессах ЖЦ, представленных в ТЛ.

Данная модель $М_{кач}$ – четырехуровневая. На первом находятся характеристики (показатели) качества, отображающие свойства, которыми должна обладать созданная ПС (табл.8.1).

Таблица 8.1

Характеристика (показатель)	**Атрибут показателя**	**Метрический анализ**	**Оценка продукта**
Функциональность	Точность, наличие функций, корректность защищенность, интероперабельность	Экспертиза структуры системы, полноты функций, непротиворечивости, защиты, совместимости со средой	Оценка сложности 1- функция реализована 0 – нет 0,9 | 1 $0,75 \le К_З \le 1$
Надежность	Безотказность, отказоустойчивость, завершенность	Экспертиза безошибочности функционирования, восстанавливаемости, помехоустойчивости	Оценочные методы надежности $0,75 \le Н \le 1$
Эффективность	Реактивность, Рациональность использования ресурсов	Экспертиза критериев эффективности, Работоспособности системы	0 или 1
Эргономичность	Понимаемость алгоритмов, обучаемость, адаптивность	Экспертиза простоты обучения, работы независимости от среды, возможности обучения	Оценка по формулам (гл.8) 0 или 1
Сопровождаемость	Стабильность документируемость, изменяемость, тестируемость	Экспертиза спецификаций, освоения, внесения изменений, верификации	0 или 1
Технологичность	Разработки Внедрения Сопровождения Стандартизации, унификации	Экспертиза структурности, интерфейсов, оценки трудоемкости внедрения, сопровождения КПИ (reuse), положений стандартов, унифицируемости	0 или 1 Оценка по формулам (гл.8) 0 или 1 0 или 1

Показатели второго уровня – это атрибуты показателей (корректность, безотказность и др.), необходимые для разрабатываемой ПС, чтобы достигнуть заданные характеристики качества первого уровня. Каждый атрибут представляется набором единичных свойств, уточняющих показатели качества первого уровня, т. е. каждый показатель определяется набором отдельных атрибутов, которые необходимо достигнуть в процессе разработки ПС на процессах ЖЦ и использовать их при комплексной оценке качества продукта.

На третьем уровне находятся так называемые метрики. Метрика – это

совокупность оценочных элементов, позволяющих определить степень достижения группы атрибутов по каждому показателю в отдельности. Иными словами, метрика, характеризующая показатель качества, может иметь один или несколько оценочных элементов.

На четвертом уровне находятся оценочные элементы, являющиеся элементарными характеристиками отдельных свойств создаваемого ПС. Набор оценочных элементов группируется и распределяется по соответствующим процессам технологии разработки ПС.

Текущая оценка отдельных элементов качества проводится на всех ТП. Контрольные точки проверки – специальные технологические операции, выполняемые службой инспекции результатов.

На процессах ЖЦ ТЛ проводится метрический анализ степени достижения соответствующего атрибута и соответственно показателя качества путем:

– экспертизы состояний программного объекта (структуры программы, правильности оформления документации и др.) и заполнения протокола экспертизы;

– аналитических вычислений с помощью информации, формируемой в картах регистрации ошибок о результатах тестирования объекта на соответствующих процессах ТЛ;

– оценки надежности соответствующими математическими моделями и методами.

– комплексной оценки качества созданной ПС с использованием отдельных промежуточных значений атрибутов оцененных элементов на ТП.

Стандартная модель качества и аналитическая оценка характеристик ПС

Для оценки качества программного продукта разработано несколько стандартов ISO/IEC 9126-2, ISO/IEC 9000 (1-4) и ядро знаний SWEBOK. В них модель качества стандартизована для всех видов и типов программных компонентов и ПС. Формально она имеет такой вид:

$$M_{кач} = \{Q, A, M, W\},$$

где $Q = \{q_1, q_2, \ldots, q_i\}_{i=1,\ldots,6}$, – множество характеристик качества *(Quality – Q)*;

$A = \{a_1, a_2, \ldots, a_j\}_{j=1,\ldots,J}$, – множество атрибутов (*Attributes – A*), каждый из которых фиксирует отдельное свойство q_i – характеристики качества;

$M = \{m_1, m_2, \ldots, m_k\}_{k=1,\ldots,K}$, – множество метрик (*Metrics – M*) для каждого a_j атрибута, используемого после измерения этого атрибута;

$W = \{w_1, w_2, \ldots, w_n\}_{n=1,\ldots,N}$, – множество весовых коэффициентов (*Weights – W)* для нивелирования метрик множества.

В стандарте ISO/IEC 9126-2 определено шесть базовых характеристик качества ПО:

q_1: функциональность (functionality),
q_2: надежность (realibility),
q_3: удобство применения (usability),
q_4: эффективность (efficiency),
q_5: сопровождаемость (maitainnability),
q_6: переносимость (portability).

Каждая из характеристик (показателей) обладает свойствами и атрибутами, определяющими разные ее свойства, которые необходимо знать пользователю при желании использовать соответствующее ПО.

Далее излагается сущность характеристик модели качества $M_{кач}$ и соответствующие формулы для их оценки с учетом m_i–метрик и веса каждой метрики w_i,.

Функциональность – совокупность свойств, определяющих способность системы предоставлять требуемое множество функций для решения задач в соответствии с требованиями. В модели качества эта характеристика задается набором атрибутов $q_1 = \{a_{11}, a_{12}, a_{13}, a_{14}, a_{15}, a_{16}\}$, семантика и оценка которых приведена ниже.

a_{11}: функциональная полнота – свойство компонента, которое определяет степень достаточности функций для решения задач в соответствии с его назначением. Данный атрибут можно представить в виде отношения всех реализованных функций F^c в компонентной системе (c), к функциям F^m (m – требование):

$$a_{11} = \sum_{i=1}^{N} F^c / \sum_{j=1}^{K} F^m.$$

a_{12}: корректность – атрибут, который означает степень достижения правильности каждой функции F^m, заданной в требовании, и каждой функции F^c, реализованной в компонентной системе. При этом система обладает свойством полной корректности, если $F^m = F^c$, и частичной корректности, если $F^m \subset F^c$. Для большинства ПС достаточно частичной корректности. Степень корректности – это степень функциональной корректности: $\sqrt{} = 1 – (card\ (F^m / F^c) / card\ F^m$.

a_{13}: точность – свойство, определяющее получение результатов с необходимой степенью точности. Данный атрибут оценивается отношением ∇ разности значений функций $F^c{}_i(D_i)$ и $F^m{}_i(D_i)$ на D_i входном наборе к значению функции в соответствии с выражением:

$$\nabla = \sum_{i=1}^{card\ F^m} ((F^c{}_i(D_i) – F^m{}_i(D_i)) / F^m{}_i(D_i)) / card\ F^m .$$

a_{14}: интероперабельность – свойство, обеспечивающее взаимодействие системы в другой операционной среде;

a_{15}: защищенность – атрибут, который показывает возможность компонента (системы) фиксировать дефекты, а также ошибки, связанные с данными. Оценку степени защищенности можно провести с помощью выражения $a_{15} = fal^r / fal$, где fal^r – количество дефектов, от которых компонент защищен; fal – общее количество дефектов в компоненте или системе;

a_{16}: согласованность – атрибут, который показывает степень соблюдения стандартов, правил и других соглашений.

Таким образом, характеристика функциональность q_1 вычисляется суммированием ее атрибутов с учетом метрик и их весовых коэффициентов:

$$q_1 = \sum_{j=1}^{6} a_{1j}\, m_{1j}\, W_{1j}.$$

Надежность ПО – определяется как вероятность того, что компоненты системы или сама система функционируют безотказно в течение фиксированного

периода времени в заданных условиях операционного окружения/среды. В модели качества надежность задается на множестве атрибутов $q_2 = \{a_{21}, a_{22}, a_{23}, a_{24}\}$, которые определяют способность системы преобразовывать исходные данные в результаты при условиях, зависящих от периода времени жизни системы (износ и старение не учитываются).

Снижение надежности компонентов происходит из-за ошибок проектирования. Отказы и ошибки в программных компонентах могут появляться на заданном промежутке времени функционирования компонента/системы. К атрибутам надежности относятся.

a_{21}: безотказность – свойство системы функционировать без отказов (программ или оборудования). Если компонент содержит дефект, то во множестве $D=\{De|e \in L\}$ всех дефектов, можно выделить подмножество $E \subseteq D$, для которых результаты не соответствуют функции F^m, заданной в требованиях на разработку. Вероятность p безотказного выполнения компонента на De, случайно выбранном из D среди равновероятных, равна:

$$p = 1 - card\ \{E\}/\ card\ \{D\}.$$

Отказ (failure) показывает отклонение поведения системы от предписанного выполнения предписанных ей функций. Появление отказа может быть причиной ошибки (fault/ error), вызывающей его. Если ошибка сделана человеком, то используется термин mistake. Когда различие между fault и failure не критично используется термин defect, означающий либо fault (причина), либо failure (действие). Связь между этими понятиями такая: fault → error → failure.

Существует большое разнообразие видов отказов ПО, типичные из них: внезапные, постепенные, перемещающиеся (сбои). Причины отказов могут быть физические, структурные, отказы взаимодействия и др. Они могут возникать естественным путём, вноситься человеком или внешней операционной средой в период создания или эксплуатации системы, а также быть постоянными или носить временный характер.

Наработка на отказ определяет среднее время между появлением угроз, нарушающих безопасность, и обеспечивает трудно измеримую оценку ущерба, которая наносится соответствующими угрозами.

Вычисление среднего времени T наработки на отказ реализуется формулой

$$T = \sum_{I=1}^{De} \nabla t_i^E \ / N,$$

где ∇t_i^E – интервал времени безотказной работы компонента i –го отказа; N – количество отказов в системе.

a_{22}: устойчивость к ошибкам показывает на способность программной системы выполнять функции при аномальных условиях (сбоях аппаратуры, ошибках в данных и интерфейсах, нарушениях в действиях операторов и др.). Оценку устойчивости можно получить по формуле

$$\Upsilon = N^v/N,$$

где N^v – количество разных типов отказов, для которых предусмотрены средства восстановления; N – общее количество всех отказов в системе.

a_{23}: восстанавливаемость – свойство, показывающее способность возобновлять функционирование системы после отказов для повторного исполнения. Среднее время восстановления компонента можно определить по формуле

$$T = \sum_{I=1}^{De} \nabla t_i^b \ / D,.$$

где ∇t_i^b – время восстановления работоспособности компонента после i – отказа; De – количество дефектов и отказов в системе.

a_{24}: согласованность – атрибут, который показывает степень соблюдения стандартов, правил и других соглашений.

Обнаруженные ошибки устраняются, а надежность системы возрастает. Чем интенсивнее проводится эксплуатация, тем интенсивнее выявляются ошибки, и обеспечивается рост надежности. Фактором, влияющим на оценку надежности ПО, относится – угроза, приводящая к снижению безопасности системы и ущербности всей системы.

В проблеме надежности ПО важное место занимает понятие устойчивости системы к отказам ПО и возможности восстанавливаться самопроизвольно после возникновения отказа. Количественная оценка надежности системы имеет вид

$$q_2 = \sum_{j=1}^{4} a_{2j}\, m_{2j\ W\,2j.}$$

Удобство применения – это множество свойств ПС, обеспечивающих условия использования ее пользователями для получения необходимых результатов. Эта характеристика определяется на множестве таких эргономичных атрибутов:

a_{31}: свойство понимания выражается усилием, затрачиваемым на распознавание логических концепций и условий применения программной системы;

a_{32}: свойство возможности изучения означает усилия отдельного пользователя понять, как применяется ПО, как проводиться контроль (например, ввода, вывода данных), а также процедур и документации;

a_{33}: оперативность – реакция системы при выполнении разных операций и их контроля;

a_{34}: согласованность – соответствие программной системы требованиям стандартов, соглашений, правил, законов и предписаний.

Все атрибуты оцениваются экспертами, которые в зависимости от их уровня знаний дают соответствующие качественные заключения. Поэтому оценка данной характеристики зависит от оценок экспертов и имеет вид:

$$q_3 = \sum_{j=1}^{4} a_{3j}\, m_{3j\,W\,3j.}$$

Эффективность – свойства системы, которые показывают взаимосвязь между уровнем ее выполнения, количеством используемых ресурсов (аппаратуры, расходных материалов и др.), услуг штатного обслуживающего персонала и др.

К свойствам относятся:

a_{41}: реактивность – время отклика, обработки и выполнения функций компонентов/системы;

a_{42}: эффективность – количество используемых ресурсов при выполнении функций ПО и продолжительность их вычислений;

a_{43}: согласованность – соответствие данного атрибута заданным стандартам, правилам и предписаниям.

Количественная оценка данной характеристики имеет вид

$$q_4 = \sum_{j=1}^{3} a_{4j}\, m_{4j\, W\, 4j}.$$

Сопровождаемость – множество свойств, которые отражают усилия, которые затрачиваются на проведение модификаций, включая корректировку, усовершенствование и адаптацию системы для новой среды выполнения, а также требований или функциональных спецификаций. Данная характеристика в модели качества состоит из следующих атрибутов:

a_{52}: анализируемость – свойство, необходимое для проведения диагностики отказов в ПО и идентификации частей системы с целью модификации;

a_{53}: изменяемость – свойство, которое определяет возможность проведения модификации компонента или системы с удалением ошибок при внесении изменений, а также дополнения новых возможностей в систему или среду функционирования;

a_{54}: стабильность – способность системы работать без ошибок и выполнять необходимые функции без риска проведения ее модификации;

a_{53}: тестируемость – свойство, обеспечивающее возможность верификации в целях проверки правильности и возможного обнаружения разного рода ошибок, а также валидации требований на соответствие их правильной реализации в системе;

a_{54}: согласованность – соответствие данного атрибута определенным стандартам, соглашениям, правилам и предписаниям.

Количественная оценка данной характеристики проводится экспертным и аналитическим способом по формуле

$$q_5 = \sum_{j=1}^{3} a_{5j}\, m_{5j\, W\, 5j}.$$

Переносимость **–** множество атрибутов, указывающих на возможность системы приспосабливаться к работе в новых условиях среды (организационной, аппаратной и программной). Перенос на другую платформу или среду связан с совокупностью действий по обеспечению возможности функционирования в новой среде, отличной от той, в которой система создавалось. К атрибутам данной характеристики относятся:

$a_{61:}$ адаптивность – способность системы адаптироваться к различным операционным средам, она оценивается по формуле: $a_{61} = Za / Zd,$ где

Za – затраты на адаптацию к новой операционной среде;

Zd – затраты на разработку новой системы для новой операционной среды;

a_{62}: настраиваемость – атрибут, определяющий необходимые затраты на запуск или инсталляцию программного продукта в другой среде;

a_{63}: сосуществование – возможность использования специального ПО или компонентов среды работать вместе в одной среде;

a_{64}: заменяемость – возможность заменять отдельные компоненты другими с условием обеспечения их взаимодействия (интероперабельности) с другими программами и не совместной работы в заданных условиях среды;

a_{65}: согласованность – соответствие стандартам или соглашениям и правилам переноса программной системы в другую среду.

Количественная оценка данной характеристики с учетом соответствующих метрик и их весов имеет вид

$$q_6 = \sum_{j=1}^{5} a_{6j}\, m_{6j\, W\, 6j}\,.$$

Комплексная оценка ПО. Если в требованиях к ПО было установлено несколько показателей качества, то просчитанный каждый показатель качества умножается метрику на соответствующий весовой коэффициент, а затем проводится суммирование по всем показателям. В результате получается интегральная оценка уровня качества ПО.

Приведенные формулы для оценки показателей качества с использованием метрик $a_i \subset A$ и весовых коэффициентов $wi \subset W$ для каждого атрибута могут использоваться отдельно для каждого компонента системы. Используя полученные оценки q_j характеристик качества применительно к отдельному компоненту (com), получаем интегральную оценку качества одного компонента в виде

$$Q_{com} = \sum_{\text{j=1}}^{6} q_j\,.$$

Если ПС система содержит N–компонентов и для них проведена оценка, то комплексная оценка качества системы (sys) имеет вид

$$Q_{sys} = \sum_{l=1}^{N} Q^{l}_{com}.$$

Заметим, имеются и другие подходы к аналитической оценки качества ПО, приведенные в [6, 140].

8.4.2. МОДЕЛИ ПРЕДСТАВЛЕНИЯ ПРОЕКТНЫХ РЕШЕНИЙ НА ТЛ

К этой категории моделей относятся модели проектных $\mathbf{M_{ПТД}}$, ТП, ТЛ, ЖЦ и эксплуатационных документов $\mathbf{M_{ЭПД}}$. Они задают структуру и содержание проектных решений и соответствующих технологических документов, которые создаются в ходе разработки ПС по ТЛ.

Модели $\mathbf{M_{ПТД}}$ формализованного представления проектных решений в ходе разработки образуют набор моделей, позволяющих осуществлять выполнение следующих функций разработчика в процессе разработки:

– описания результатов проектных решении в ПТД табличной формы, принятой в ТПР и в конкретных технологических линиях;

– проведения контроля процесса преобразования состояния программного объекта и/или фиксации ошибки;

– занесения фрагментов текстов для включения в соответствующие разделы документов программной документации согласно модели $M_{ЭТД}$.

Выбор формы представления моделей ПТД для конкретной ТЛ зависит от класса реализуемых объектов. Как показал опыт разработки, при проектировании ПС, работающих с БД, используются модели типа «сущность – связь», для которых применяются формы ПТД, более полно отображающие специфику программ их реализации. Наиболее часто используемая формы ПТД приведены ниже в таблицах.

Таблица 8.2

ТО	**Таблица данных**						**Лист**	
Порядковый номер	Уровень	Идентификатор	Тип	Шаблон	Диапазон	Точность	Начальное значение	Примечание

Таблица 8.3

ПМ	**Логика алгоритма**			**Лист**	**Листов**
1. Назначение 2. Среда					
ЯП	**ОС**	**Хранятся**	**Память**	**Время**	**Системное ПО**
3. Использует 4. Формирует 5. Тестирует					

Модели $М_{тд}$, $М_{тп}$ обеспечивают построение любых видов программ. Модель процесса (линии) – это типовая структура (карта) процесса (линии) для формализованного задания последовательности операций (процессов), связь между которыми задается маршрутом, отображая взаимозависимость операций.

Модели $М_{гл}$ и $М_{тп}$ служат способом спецификации формируемых ФОТ и моделирования процессов ЖЦ создаваемого программного объекта. $М_{тп}$ имеет вид:

$$М_{тп} = \{ КТП, ТМШ, М_{птд}, ТД_{ТП}\},$$

где КТП – структура карты процесса, принятая в ТПР (табл.8.4). для фиксации операций процесса ЖЦ; ТМШ – графовая форма представления технологического маршрута; Мптд – зафиксированные для данного ТП из множества ПТД модели табличных документов, используемые при описании состояния программного объекта на этом процессе; $ТД_{ТП}$ – структура технологических документов ТП для описания прагматики и семантики процесса разработки соответствующего объекта и его состояния.

Применяя структуру описания отдельного процесса, модель ТЛ можно представить следующим образом: $М_{ТД} = \{\{М^{i}_{ТП}\}\ ТМШ_{ТП})$, $i = \overline{1, N}$.

Таблица 8.4

КТП	**Карта технологического процесса**						**Лист**	**Листов**
Код ТП	Наименование ТП							
Код ТО	Наименование (краткое содержание)	ПТД				Исполнитель	Методы и средства выполнения ТО	
		Исходные		Выходные			Технологические	Инструментальные
		Код	Наименование	Код	Наименование			
1	2	3	4	5	6	7	8	9

. . .

Разработал	ФИО	Подпись	Дата
Принял			
Утвердил			

Здесь элементами модели являются наборы моделей ТП и модель технологического маршрута линии. Модель ТЛ – абстрактная модель управления процессом разработки ПС на инженерной основе с обратными связями и дискретным характером взаимодействия между каждым ТП. ТЛ содержит управляющую часть, представленную технологическим маршрутом, задающим связи между процессами, и операционную, задаваемую картами ТП в виде набора ТО.

Разделение ТЛ на управляющие и операционные элементы способствует ее настройке на реальные условия, заключающиеся в конкретизации круга специалистов, уточнении документов и инструментов и составлении сетевого графика работ в соответствии с технологическим маршрутом.

Данное представление ТЛ определяет характер развития современных гибких автоматизированных линий приема, передачи и обработки информации в среде распределенных систем обработки информации (рис. 8.3).

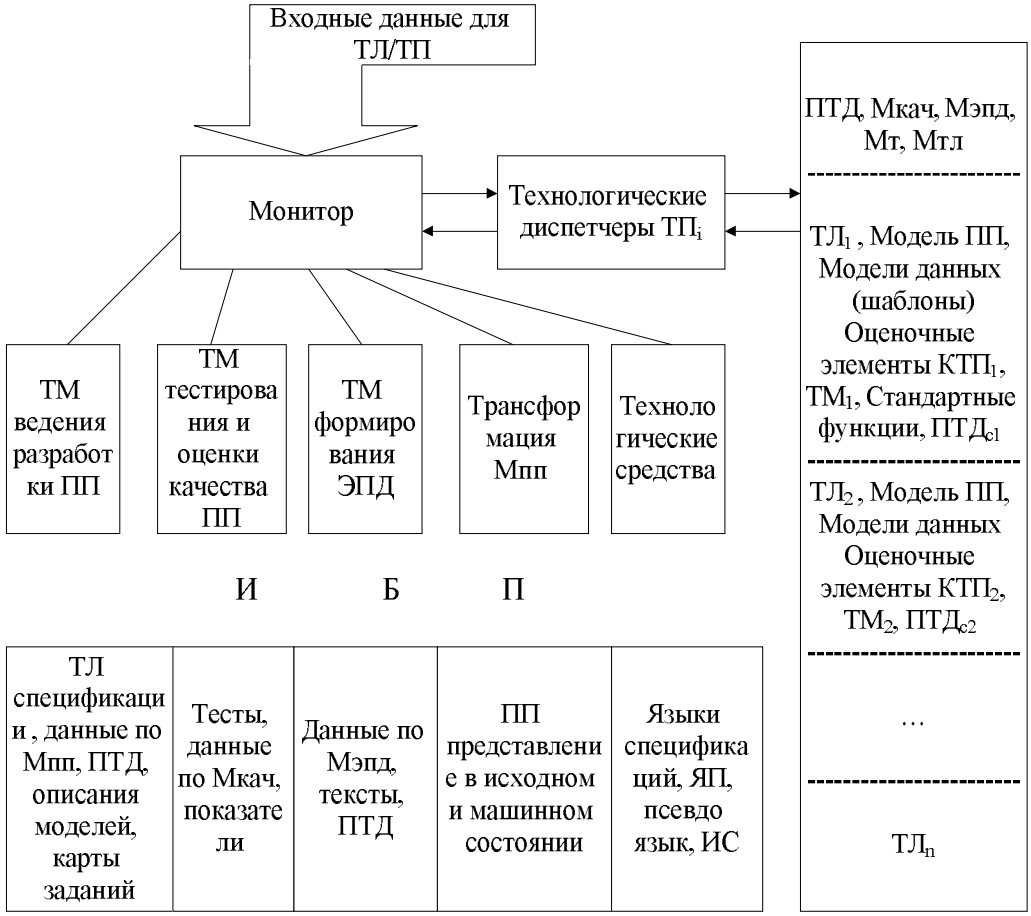


Рис.8.3 Структура программно-технологического комплекса

Модель ЭПД это формализованным представлением принятой в ЕСПД структуры каждого типа программной документации (ГОСТ 19.501 – 19.507–77), полученным исходя из анализа требований ГОСТов и специфики представления

отдельных функций ПрО набором функционально-ориентированных заготовок. Для них делается описание, которое затем используется при выводе программы из ее модели по заготовкам. Эти описания размещаются в соответствующих местах шаблонов документов.

Другим источником информации для формирования документации по $M_{ЭПД}$ являются ПТД, заполняемые в ходе разработки. Модель $M_{ЭПД}$ включает набор подмоделей (структур) конкретных документов ЭПД. и базовых компонентов ($M_{бк}$), содержащих формальное описание общих для всех ЭПД элементов, т. е.

$M_{эпд} = \{ M_{бк}. M_{оп}, M_{рпр}, M_{рсрп}, M_{ро} \}$,

где $M_{оп}$ – модель описания применения; $M_{рпр}$ – модель руководства программиста; $M_{рсрп}$ – модель руководства системного программиста;. $M_{ро}$ – модель руководства оператора и др.

Представленный класс технологических моделей может пополняться новыми моделями. Их состав и структура определяется на ТПР ПС соответствующего класса и служит основой реализации ТЛ.

Модель жизненного цикла ПС. Процесс разработки ПС определяется моделью ЖЦ, процессы которой будем отождествлять с технологическими процессами. Для каждого ТП фиксируются начальное (S_0), промежуточное (S_i) и конечное (S_k) состояния разрабатываемого программного объекта. Переход объекта из состояния S_i; в S_{i+1} на ТП связан с выполнением технологических операций этого ТП. При этом для каждого типа ПП из заданного класса программных систем может быть выбран свой набор ТП и состояний ТП, определяемых на множестве исходных данных.

Приведенная на рис.8.2 схема среды проектирования соответствует методу проектирования ПС сверху-вниз, при котором понятия ($i+1$)-го уровня $(ТП_{i+1})$ описываются через элементарные понятия i-го уровня и представляют собой данные, входящие в класс понятий состояния ПС. При этом не исключаются переходы с одного уровня абстракции на другой (сверху вниз и снизу вверх).

Модель ТП. Выделение состава ТП и содержания технологических операций зависит от класса ПС, используемой методологии и принятой организации проведения работ. При определении состава и содержания ТП решается задача обеспечения инженерной дисциплины разработки, основанной на современных программных принципах, методах и инструментах, способствующих полному и адекватному отображению понятий и функций реализуемой предметной области в элементы состояния S (рис.8.4).

Основная цель выделяемых $ТП_i \subset ТЛ$ состоит в получении некоторого полуфабриката ПП (S_i – состояние ПС), фрагментов ЭПД для проведения экспертизы уровня и качества в соответствии с моделью качества $M_{кач}$. Каждый ТП модели ЖЦ в общем виде определяет состояние элементов ПС, состав технологических операций, обеспечивающих преобразование исходного состояния ПС и получение его конечного состояния.

Таким образом, общая технологическая модель (схема) процесса разработки является отражением модели ЖЦ, способов преобразования состояний ПС и может быть представлена в виде

$$M_{пр} = (S, ТП_i, (ТО_j), ТМ_j),\ i = \overline{0, 7}\ ,\ j = \overline{1, k}.$$

Множество состояний S рассматриваемой модели включает в себя:

S_0 – исходное (начальное) состояние – описание требований заказчика,

предъявляемых к данному ПС;

S_1– состояние, включающее набор элементарных состояний, а именно, описаний функций *(S_1^1),* архитектуры *(S_1^2)*, структуры данных (S_1^3) и т. п. средствами языков спецификаций L_1;

S_2 – состояние соответствует техническому проекту и включает описание в классе языков L_2 алгоритмов функции *(S_1^2)*, данных *(S_2^2),* интерфейсов *(S_2^3),* гипертекстов документации *(S_2^4),* оценочных элементов модели качества *(S_2^5)* и др.;

S_3 – состояние соответствует рабочему проекту и включает описание в ЯП (класс L_3,) текстов программ (S_3^1), модулей (S_3^2), тестов (S_3^3) и т. п.;

S_4 –состояние соответствует отлаженным элементам программного продукта;

S_5 – состояние – это ПС после сборки;

S_6 – состояние ПС, соответствующее программному продукту, которое испытывается, проверяется на соответствие заданным функциям и значениям показателей качества;

S_7; – состояние ПС на процессе сопровождения.

Технологический процесс, входящий в состав $M_{пр}$, – промежуточный (частичный) процесс, осуществляет преобразование S_i - го состояния ПС с помощью набора $ТО_j \subset ТП_i$ данной $M_{пр}$, поддерживаемых TM_j (либо один ТМ реализует несколько ТО).

Набор операций $M_{пр}$ не является фиксированным, он уточняется для каждого типа ПС и описывается в технологических документах. Среди операций могут быть типовые, используемые готовыми при создании конкретной технологии разработки ПС определенного типа. Для обеспечения технологичности разработки в их состав входят операции управления качеством разработки и формирования документации на всех процессах ЖЦ в соответствии с системой моделей $М_{эпд}$, определенной выше.

Модель общего вида любого частичного $ТП_i$ представлена на рис.8.4.

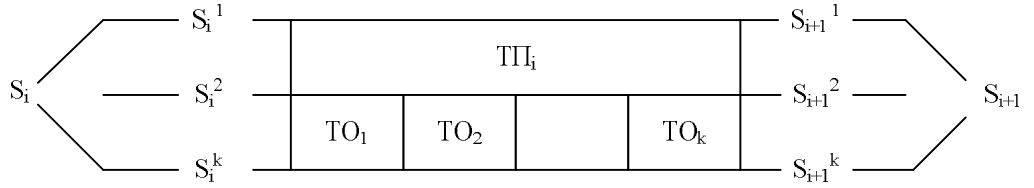


Рис.8.4. Модель частичного $ТП_i$ процесса

Состояние объекта S_i – определяется набором частичных его состояний S_i^1, ... , S_i^k, подаваемых на вход $ТП_i = \{ТО_k\}$. Данный набор для конкретного ПС конечен.

Управление процессом преобразования состояний объекта S_0 в S_k может быть задано в виде графовой модели (соответствует технологическому маршруту), в вершинах которой находятся процессы (или операции), а ребра задают всевозможные переходы из одного состояния объекта в другое. Графовая модель отображает способ ведения разработки с распараллеливанием работ и возвратами (при ошибках) в предыдущие процессы (рис. 8.5).

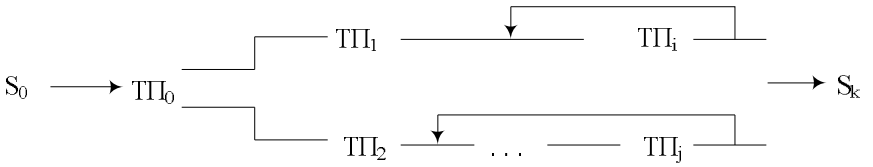


Рис.8.5. Графовая модель технологического процесса

Каждый частичный процесс ТП, является абстрактным конечным автоматом, граф которого совпадает с технологическим маршрутом процесса, множество состояний S_i которого определено на совокупности операций $\{TO_i\} \subset$ ТО данного процесса.

Переход объекта из состояния S_i в состояние S_{i+1} может быть только под действием технологического маршрута и осуществляется технологическим диспетчером посредством выбора из множества операций процесса, зафиксированных в карте *i*-го процесса, очередной операции при условии, что результат предыдущей операции проанализирован и выполнен.

Задача выполнения *i*-го процесса заключается в переводе автомата из некоторого промежуточного состояния объекта в другое S_{i+1} с выполнением операций преобразования, качественной или количественной оценки результата разработки объекта и формирования фрагментов ЭПД по $M_{ЭПД}$.

Таким образом**,** рассмотренные принципы и метод разработки ТЛ путем их сборки, предопределили специфику языка формального описания ТЛ, а также структуру модели качества, проектно–технологических и эксплуатационных документов. Приведен набор таблиц, которые заполняются необходимыми для моделей данными в среде программно-технологического комплекса. Предложена общая структура процессов и линий.

8.5. СРЕДСТВА РАЗРАБОТКИ ФУНКЦИОНАЛЬНЫХ ПРОГРАММ ПО ТЛ

Средства поддержки любой технологии программирования предполагают наличие инструментальных ПС и правил их применения. Конкретные инструментальные средства совместно с соответствующими процедурами их использования будем называть технологическим модулем (ТМ). В качестве примеров рассмотрены средства сборки программ ССПМ (прототип – система АПРОП [44, 125, 126]) из модулей и проектирования методо-ориентированных ППП (прототип–комплекс АПФОРС [118, 121]).

8.5.1. ТЕХНОЛОГИЧЕСКИЙ МОДУЛЬ СБОРКИ, ТЕСТИРОВАНИЯ МОДУЛЕЙ ПС

Данный ТМ предназначен для:

– сборки, тестирования и отладки отдельных программных компонентов (ПК) из модулей, написанных на различных ЯП и автономно отлаженных вне среды данного ТМ;

– тестирования и отладки программного продукта из собранных и отлаженных компонентов в среде данного ТМ (рис. 8.6);

– получения версии ПП (или его части) в хранилищах пользователя ТМ в виде готового к исполнению агрегата.

Инструментальной основой ТМ является ССПМ, предназначенная для автоматизированного изготовления ПП различного уровня сложности на основе методологии модульного программирования. Процесс конструирования ПП осуществляется в диалоговом режиме с использованием ЯМК.

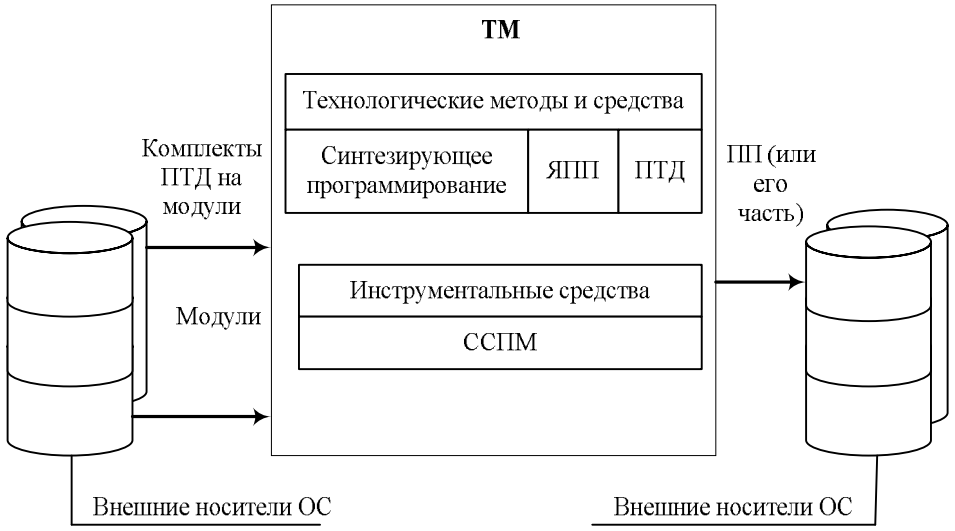


Рис. 8.6. Состав ТМ сборки, тестирования и отладки ПП

Средства ССПМ обеспечивают:

– ввод исходных текстов программных объектов в базу данных как с терминалов, так и с внешних носителей (вводимый объект сопровождается паспортом, содержащим сведения о его назначении, входных и выходных параметрах, вызываемых объектах, требуемых программно-информационных ресурсах и т. д.);

– ведение базы данных объектов (создание, уничтожение, занесение, корректировка и удаление из них различных объектов);

– обработку объектов с использованием средств ОС (трансляция, редактирование, счет и т.п.);

– сборку одноязыковых и разноязыковых модулей в программный агрегат;

– отладку и отторжение готового агрегата от системы.

Технологическую основу ТМ составляют методы и средства, поддерживающие стратегию синтезирующего программирования с соблюдением принципа модульности. В ЯМК описывается входная информация об объектах сборки (ее состав регламентируется формами ПТД), автоматически преобразуемая впоследствии в паспорт объекта для ССПМ. Эта информация подается на вход ССПМ с внешних файлов (табл.8.5).

Далее описываются основные операции процессов данной ТЛ.

Подготовка исходной информации. ПТД заполняются машинным способом на бланках ПТД средствами любого редактора текстов, которые объединяются в виде групп записей в последовательный файл.

В состав комплекта ПТД на модуль входят паспорт модуля, описание параметров и описание логики. На основе этих документов производится автоматическая генерация паспортов модулей в системном виде.

При описании паспорта модуля указываются имя модуля и список формальных параметров. Входные параметры этого списка отделяются от выходных точкой с запятой. Затем размещаются дополнительные разделы: ПАРАМЕТРЫ и ВЫЗЫВАЕМЫЕ МОДУЛИ, которым соответствует модули и их параметры.

Таблица 8.5

Номер операции	**Содержание операции**	**Оператор ЯМК**	**Ссылки на документы**
1.	Подготовка исходной информации		Технология разработки ПС
1.1	Подготовка проектно-технологических документов		То же Методика применения ТМ разработки ПС
1.2	Подготовка проектно-технологических документов		ГОСТ 19501
1.3	Подготовка ССПМ к запуску		Руководство программиста
1.4	Запуск ССПМ и начало работы	SUPER	То же
2	Погружение модулей и форм проектных документов среду ССПМ	SUBSYS	Технологическая инструкция
3	Сборка, тестирование и отладка ПП	PKD	Методики сборки, тестирования и отладки
3.1	Просмотр и редактирование паспортов и исходных текстов модулей	CORR TLIB	Руководство программиста
3.2.	Создание личной библиотеки - ЛБ	CREATE	То же
3.3	Запись исходного текста и модуля в ЛБ	LOAD	“ ”
3.4	Описание исходных модулей, находящихся в ЛБ	DSCR	“ ”
3.5	Трансляция модуля	TRANS	“ ”
3.6	Просмотр и редактирование паспортов и исходных текстов модулей из ЛБ	CORR PLIB	“ ”
3.7	Занесение объектного вида модуля в ЛБ	LOAD	“ ”
3.8	Описание объектного вида модуля из ЛБ	DSCR	“ ”
3.9	Построение графа ПП	LINK	“ ”
3.10	Отладка ПП с включением средств трассировки передач управления и параметров	LINK	“ ”
3.11	Сборка ПП без включения средств трассировки	LINK	“ ”
3.12	Подготовка тестовых данных	CORR	“ ”
3.13	Тестирование ПП на тестовых данных	EXEC	“ ”
3.14	Просмотр результатов вы-	LIST	“ ”

	полнения		
3.15	Распечатка результатов или их фрагментов	PRINT	“ ” _
3.16	Запись готового ПП и ЛБ	LOAD	“ ” _
4	Перенос ПП или его части в среду ОС и завершение работы		“ ” _
4.1	Запись готового ПП в библиотеку готовых модулей ОС	SUBSYS OS	Технологическая инструкция
4.2	Завершение работы системы, получение протокола и результатов тестирования ПП	SEND CANCEL	Руководство программиста
4.3	Анализ протокола работы системы и результатов тестирования ПП	_	_

Если формат параметра не указан, то используется принцип умолчания, по которому определяется значение, соответствующее его типу и коду ЯП. Задание фактических параметров, передаваемых вызываемому модулю, обязательно в случае сборки разноязыковых модулей. При использовании таблицы данных описание параметров выполняется на ЯМК. Обязательными являются поля «уровень», «имя», «размерность» и «тип». Если не заполнено поле «формат», принимается значение по умолчанию.

Подготовка исходных текстов модулей и их запуск. Кодирование модулей выполняется с помощью ПТД средствами ЯП. Тексты модулей оформляются в соответствии требованиям ЯП и ГОСТ на оформление документации. Правильные тексты (после трансляции) сохраняются в библиотеке исходных текстов для последующего их использования при сборке и тестировании в среде ССПМ.

Задание на запуск вводится из файла или с дисплея. Подготовка системы к запуску состоит в настройке задания на конкретную конфигурацию технических средств и в подключении требуемых наборов данных. Такими могут быть: наборы данных, содержащие ПТД; библиотечные наборы данных ОС, содержащие исходные тексты модулей; наборы данных, используемые при работе ПП (в том числе описывающие входные данные).

Запуск ССПМ осуществляется средствами ОС. При этом возможен ряд нерегулярных ситуаций, связанных с техническим состоянием терминалов или их занятостью, отсутствием свободной памяти под временные и динамически заказываемые системой наборы данных. После этого система выделяет терминалы, с которых вводятся: режим работы (диалоговый или пакетный); различные параметры функционирования; операторы ЯМК.

Таким образом, пользователь осуществляет сеанс работы в системе. В дальнейшем он может завершить сеанс и начать новый либо отключить терминал от системы.

Погружение модулей и ПТД в среду ССПМ. Эти действия выполняются с помощью оператора SUBSYS в форме диалога с пользователем. По запросам пользователя данный оператор осуществляет: подключение БИМ к системе, содержащей модули для сборки и тестовые данные, которые погружаются в ВБМ ССПМ.

Сведения о модулях, введенных в ВБМ, пользователь может получить, введя со

своего экрана оператор описания модулей. По этому оператору на экран выдается таблица, содержащая сведения об именах, видах, местонахождении и языке объектов, с которыми работают пользователи.

Исправления в текстах погруженных в среду ССПМ, исходных модулей и паспортов выполняются с помощью операторов ЯМК:

CORR PAS TLIB <имя модуля> (просмотр и редактирование паспорта модуля);

CORR TLIB <имя модуля> (просмотр и редактирование текста модуля) во время сеанса работы.

Создание и ведение ЛБ. Модули разрабатываемого ПП целесообразно хранить в постоянных архивах личных библиотек (ЛБ). Их создание выполняется оператором CREATE PLIB <имя ЛБ>.

При этом на соответствующем томе системы должен быть достаточный объем памяти под ЛБ, уточняемый системой в диалоге с пользователем.

Запись модуля в ЛБ выполняется с помощью оператора LOAD MOD <имя ЛБ>, <имя модуля> SOUR. Результатом работы является занесение модуля в ЛБ и сообщение системы о том, что модуль с заданным именем в исходном виде (чему соответствует SOUR) помещен в ЛБ. ССПМ информирует также о возможных нерегулярных ситуациях: модуль с заданным именем уже находится в ЛБ; в ЛБ нет свободного места для занесения модуля и др.

При переполнении ЛБ система автоматически ее уплотняет.

Занесение объектного вида модуля в ЛБ выполняется оператором

LOAD MOD <имя ЛБ> . <имя модуля> OBJ.

Сообщения ССПМ и возможные нерегулярные ситуации аналогичны описанным выше.

Для того чтобы объектный модуль, ранее занесенный в ЛБ ССПМ, стал доступен системе для дальнейшей обработки (например, при сборке агрегата), его необходимо описать оператором DSCR PLIB MOD <имя ЛБ>. <имя объекта> OBJ.

Если модули были записаны в ЛБ на предыдущих сеансах, то для их использования в данном сеансе задается оператор DSCR PLIB MOD <имя ЛБ> . <имя модуля> SOUR, по которому модуль становится доступным для обработки другими операторами ЯМК (трансляции, редактирования и т. д.).

Трансляция объектов выполняется оператором TRANS [<имя ЛБ>.] <имя объекта>. В списке может быть указан набор объектов. В качестве дополнительных параметров могут быть операции – вывод листинга, диагностического сообщения и т.п. По завершении трансляции система выдает запрос о выдаче и просмотре листинга на экран.

Объекты, находящиеся в ЛБ, корректируются оператором CORR PLIB <имя ЛБ> . <имя модуля>, а редактирование паспорта исходного текста модуля оператором CORR PAS PLIB <имя ЛБ> . <имя модуля>.

Программный агрегат создается из модулей, находящихся в библиотеках ССПМ. Его структура представляется в виде графа, в вершинах которого указываются имена объектов (модулей, макромодулей, модулей данных), а на дугах – типы отношений, указывающие на управление структурой ПП (оверлейной, динамической, простой, подзадачной). Граф описывается оператором

LINK {Prog /Seg }<имя агрегата> (<имя корневого модуля агрегата>) #.

На его основе система генерирует внутреннее представление в виде матрицы смежности, а также в форме, удобной для представления на экране дисплея.

Сборка и тестирование ПП. Сборка осуществляется с помощью оператора LINK, в котором указывается перечень модулей, входящих в собираемый агрегат. При этом модули, входящие в состав ПП (или отдельные компоненты), могут быть написаны на любом из ЯП. Связи между ними указывают на образование агрегата различной структуры.

Для тестирования ПП используется оператор TEST. Если корневому модулю ПП (или его части), подлежащему тестированию, не передаются никакие параметры, то для трассировки передач управления и данных создается отладочный вариант агрегата простой структуры

TEST PROG <имя отладочного агрегата> (<имя корневого модуля>) 0.

Для организации тестирования ПП или его цепочек, корневые модули которых имеют параметры, в среде ССПМ разрабатывается ТЕСТ-модуль, моделирующий вызов и передачу параметров корневому модулю ПК. Он должен иметь паспорт, оформленный согласно требованиям ССПМ и содержащий в разделе вызываемых модулей имя корневого модуля ПК.

В этом случае отладочный вариант агрегата простой структуры создается оператором

TEST PROG <имя отладочного агрегата> (<имя ТЕСТ – модуля>) 0.

В дальнейшем при выполнении собранного агрегата (ПП или его части) в среде ССПМ имена модулей, на которые передается управление, а также значения передаваемых им параметров будут выдаваться на экран терминала.

Оператором TEST могут быть созданы отладочные агрегаты простой, оверлейной, динамической и смешанной структур. Сборка ПП из отдельных исходных, объектных или загрузочных модулей с автоматической организацией межъязыкового интерфейса для разноязыковых модулей, но без включения средств трассировки выполняется оператором LINK.

Создание ПП простой структуры осуществляется оператором

LINK SEG <имя корневого модуля> (<имя корневого модуля>).

Полученный сегмент в загрузочном виде может быть помещен в ЛБ ССПМ для дальнейшего использования (например, при сборке ПП по частям).

Если необходимо выполнение собранного сегмента в среде ССПМ, то в случае, когда корневой модуль сегмента принимает (передает) параметры, необходимо, согласно требованиям системы, создать ТЕСТ-модуль, моделирующий вызов и передачу параметров корневому модулю сегмента. Оператор сборки в этом случае имеет вид LINK SEG <имя ТЕСТ-модуля> (<имя ТЕСТ-модуля>).

Подготовка тестовых данных. Задание для вызова ССПМ должно содержать описание данных, которые могут использоваться программами, выполненными в среде системы в данном сеансе работы. Входные данные могут быть реально существующими и будут использованы программой при ее выполнении. В то же время нет необходимости, чтобы все входные данные существовали заранее и сразу при запуске были описаны.

Входные (тестовые) данные могут готовиться в соответствии с требованиями ЯП того модуля, в котором производится их ввод, и оформляются в виде модуля данных. Он снабжается паспортом и записывается следующим образом:

МОДУЛЬ DATA
ПАСПОРТ DATA
ЯЗЫК <название ЯП>
ТЕКСТ {данные в ЯП}

МКОН.

Оформленные таким образом данные помещаются в ЛБ оператором LOAD, корректируются оператором CORR и т. п. Перезапись данных из модуля в файл, используемый в программе в качестве входного, производится оператором ЯМК.

Тестирование ПП. Выполняется согласно плану сборки и тестирования на тестовых данных. Если агрегат (ПП или его часть) в готовом виде находится в ЛБМ, то его тестирование осуществляется оператором EXEC (имя агрегата).

Если агрегат содержится в ЛБ, то применяется оператор

EXEC (имя ЛБ) . (имя агрегата).

Для трассировки передач управления и значений (предусмотренных оператором TEST) используется оператор

EXEC (имя агрегата) или EXEC # (имя ЛБ).(имя агрегата).

Результаты отображаются на экране терминала пользователя. Просмотр файлов за экраном выполняется с помощью оператора LIST (имя файла). Этот оператор рекомендуется применять при отладке для оперативного просмотра результатов счета. Для получения твердой копии используется оператор PRINT (имя файла).

Созданный ПП записывается в ЛБ оператором LOAD MOD (имя ЛБ).(имя ПП) LDR. Завершение работы системы осуществляется по оператору SEND или CANCEL. Оператор SEND завершает сеанс работы с возможностью начала нового сеанса за тем же терминалом, а CANSEL отключает терминал от системы.

Результатом работы системы может быть протокол, отражающий последовательность действий при выполнении операций, связанных со сборкой, тестированием и отладкой. Полученные результаты сравниваются с данными из таблиц тестов. Вся информация о ходе тестирования и отладки фиксируется в журнале регистрации ошибок. Обнаруженные в ПП ошибки могут быть исправлены средствами системы (оператор CORR) в новом сеансе, а сам ПП – повторно тестирован.

8.5.2. ТМ СОЗДАНИЯ ПРОБЛЕМНО–ОРИЕНТИРОВАННОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Рассматриваемый технологический модуль предназначен для выполнения–автоматизированного:

– построения функционального и системного наполнения методо-ориентированных ППП;

– получения программы решения задачи в проблемной области с помощью графа задачи;

– решения задачи средствами разработанного пакета.

Инструментальной основой ТМ является комплекс программных средств системы СППО, выполняющей автоматизированное построение программ решения задач на основе формализованных спецификаций модулей из функционального наполнения ППП.

Структурно комплекс включает в себя две функциональные подсистемы: проектирования и управления процессом создания ППП, создания и ведения элементов функционального наполнения (рис.8.7).

В функции *первой подсистемы* входят: описания ППП; ввод текстов модулей и данных, необходимых для решения задач с помощью пакетов; описание информационных файлов и задач; постановка и решение задач в терминах

предметной области; внесение изменений в исходные объекты. В функции *второй подсистемы* входят: ввод ПТД, описывающих элементы функционального наполнения; подготовка и ведение элементов в базе данных объектов; построение на основе ПТД спецификаций модулей и их отладка; сборка разноязыковых модулей и отладка агрегатов для задач ППП; информационное обслуживание.

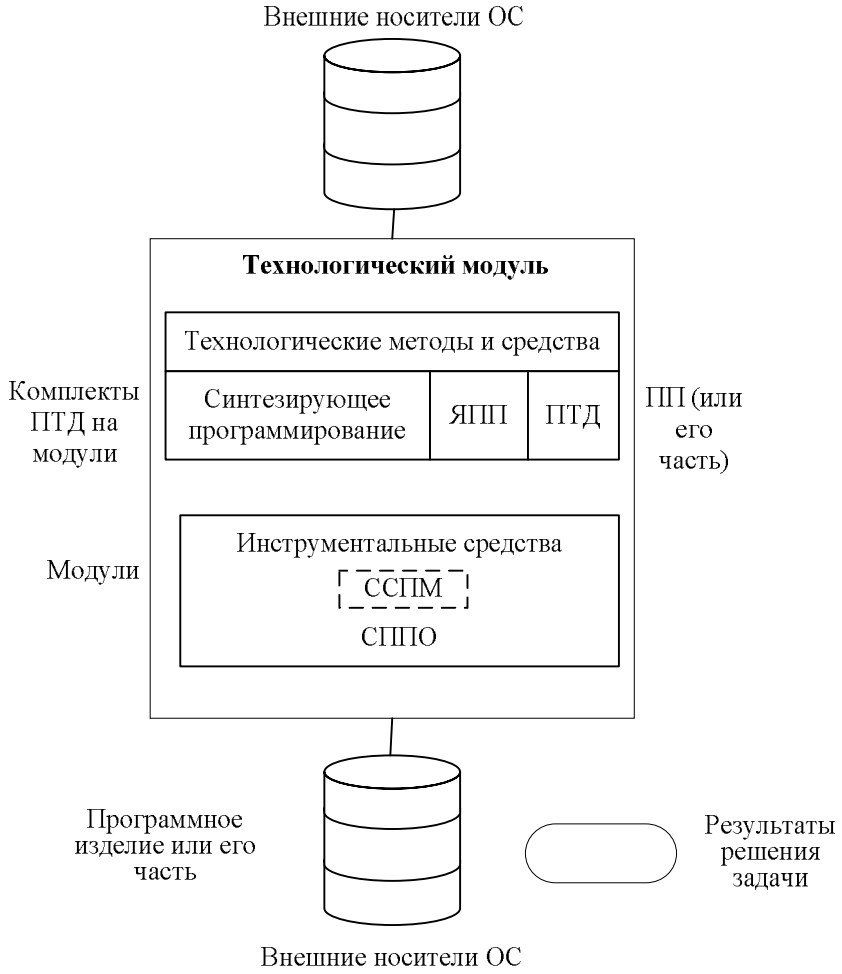


Рис. 8.7. Состав ТМ разработки методо-ориентированных ППП

При этом ТМ ССПМ представляет в рассматриваемом ТМ первый уровень автоматизации построения ППП. Создаваемый на его основе ПП – это набор программы решения задач ППП, разработанных по модульному принципу в ЯП, которые сохраняются в БИМ или ЛБ. Используя ТМ СППО пользователь на входном языке выполняет постановку задач и решает их.

Функцию управления работой СППО выполняет монитор, принимающий решение о подключении каждой из подсистем для обработки запросов пользователя. Работа осуществляется в диалоговом режиме. Технологическую основу рассматриваемого ТМ составляют методы и средства, поддерживающие стратегию синтезирующего программирования с соблюдением принципа модульности.

В качестве базовых технологических средств используются: ЯМК из ССПМ; модуль проектирования пакетов, представленных подсистемой проектирования и управления.

В дальнейшем описываются только те возможности данного ТМ, которые обеспечивают выполнение функций проектирования и управления созданием ППП.

Применение ТМ СППО для разработки ППП. Оно состоит в выполнении операций (полный набор которых приведен в табл. 8.6). Кроме этих операций могут выполняться сервисные операции инструментального комплекса.

Таблица 8.6

Номер операции	Операция	Оператор ЯМК (в реальной системе)
1.	Подготовка исходной информации	_
1.1	Подготовка исходных наборов данных, содержащих тексты используемых модулей и данные для решения задач	_
1.2	Подготовка комплекса к запуску	_
1.3	Запуск комплекса	SUPER
1.4	Начало работы	ПАКЕТ
2	Ввод модулей в среду комплекса	ВМ или ВМД
3	Ввод данных в среду комплекса	ВД или ВДД
4	Составление описаний задач ППП	ЗАДАЧА
5	Составление описаний файлов данных для задач	ФАЙЛ
6	Решение задач	СЧЕТ, ВЫДВ ВЫВП
7	Сервисные технологические операции	СТЕРЕТЬ, ВМД ВДД
8	Завершение работы комплекса ССПО	КП

Подготовка исходных наборов данных для запуска. Исходной информацией для СППО являются программные модули и их спецификации (паспорта), сформированные по требованиям ССПМ в виде порций ввода. Порции ввода располагаются во внешних файлах.

В случае, если предварительная разработка отдельных элементов функционального наполнения выполняется средствами ССПМ с использованием ЛБ, исходные тексты модулей и их спецификации должны переписываться в наборы данных ОС.

Запуск комплекса должен включать описание: данных с исходными текстами модулей задач и модулей данных; терминалов пользователей комплекса в данном сеансе работы; данных для задач, решаемых средствами ППП.

Запуск СППО осуществляется средствами ОС. В случае, если запуск произведен успешно, СППО вступает в диалог с пользователем по уточнению режима работы.

После завершения настройки системы на заданный режим работы вводится оператор SUPER (фактическое начало сеанса работы), что означает начало (либо продолжение) конструирования ППП.

Для этого используется оператор ПАКЕТ в одном из следующих форматов:

ПАКЕТ <имя пакета> НАЧАТЬ либо ПАКЕТ <имя пакета> ПРОДОЛЖИТЬ.

Для работы с СППО в режиме решения задач изготовленным пакетом используется оператор ПАКЕТ в формате

ПАКЕТ <имя пакета> ВЫПОЛНИТЬ.

Ввод модулей в среду СППО. Исходные тексты и спецификации модулей вводятся в среду СППО с помощью оператора языка проектирования пакетов. Для ввода модулей из файлов ОС используется оператор ВМ [<имя файла>].

В нем указывается имя одного из файлов с исходной информацией. Оператор следует вводить столько раз, сколько файлов будет обрабатываться в данном сеансе.

Для ввода текстов и паспортов модулей непосредственно с экрана дисплея используется оператор BMD [<имя модуля>] П [АСПОРТ] Т [ЕКСТ]

По умолчанию предполагается, что вводится текст модуля. Допускается раздельный ввод текстов модулей и паспортов.

После ввода исходной информации СППО автоматически выполняет трансляцию модулей и сохранение результатов трансляции в хранилищах системы.

Ввод данных в среду СППО. Входные данные (модули данных), как и исходные модули, оформляются в виде порций ввода в файлах ОС и вводятся в систему оператором BD [<имя файла>].

Введенные модули данных транзитом пересылаются в библиотеки данных СППО. В случае ввода данных непосредственно с экрана дисплея используется оператор BDD [<имя модуля данных>].

Имена модулей данных, а также имена функциональных модулей могут иметь синонимы, соответствующие понятиям предметной области разрабатываемого пакета. Синонимы задаются вслед за именем модуля в круглых скобках:

ТЕКСТ SKVIT (ВЕКТОР-ПЛОЩ).

Описание и решение задач. Все задачи, на решение которых ориентируется разрабатываемый ППП, должны быть описаны с помощью оператора

ЗАДАЧА <имя> [ВХ <список исходных данных>] [<описание процесса сборки>].

При описании процесса сборки указываются имя агрегата, отождествляемого с решаемой задачей, и список элементов функционального наполнения, составляющих агрегат. Первым в списке указывается корневой модуль. Порядок перечисления остальных элементов (модулей, программ) в списке не существен.

Связь задачи с исходными данными, оформленными как модули, осуществляется косвенно, т. е. с использованием логических файлов СППО, указанных в операторе ЗАДАЧА. Логический файл описывается оператором ФАЙЛ. Связь задачи с модулем данных осуществляется на процессе решения (счета) задачи. При этом информация из модуля данных автоматически переносится в набор данных, указанный в операторе ФАЙЛ.

Решение задач выполняется посредством оператора СЧЕТ <имя задачи> [<список исходных данных задачи>]. Каждому оператору СЧЕТ соответствует оператор создания задачи ЗАДАЧА. Если на шаге счета необходимо изменить некоторые исходные данные (т. е. входные логические файлы), указанные в операторе ЗАДАЧА, то они перечисляются в последовательности, установленной оператором ЗАДАЧА, а данные, не требующие изменения, отделяются запятой.

Результаты счета могут просматриваться за экраном терминала пользователя с помощью оператора ВЫВ D {<список имен файлов>}.

Вывод результатов на печать осуществляется оператором

ВЫВП {<список имен файлов>}.

Оператор ВЫВП используется в том случае, если выходной файл результатов для вывода на печать не является стандартным, а представляет собой набор данных

на МД или МЛ. В противном случае он выводится на АЦПУ автоматически.

Описание файлов данных и сервиса. Для описания логических файлов используется оператор ФАЙЛ <имя файла> <имя модуля данных>.

С его помощью описываются как входные, так и выходные логические файлы. Оператор ФАЙЛ для входных файлов выполняется столько раз, сколько модулей данных используется для наполнения одного функционального набора данных (и так для каждого функционального набора данных).

К сервисным ТО относятся:

– корректировка функциональных модулей и модулей данных;

– уничтожение модуля и уничтожение логических файлов, задач и пакета.

Завершение работы ССПО осуществляется оператором КП[ПАКЕТ], по которому монитор передает управление ССПМ и разрешает ввод операторов ЯМК. Возврат в среду проектирования пакета осуществляется оператором ПАКЕТ, а завершение работы ССПО – оператором SEND или CANCEL ССПМ.

8.5.3. СЕМАНТИКА ПРОЦЕССОВ РАЗРАБОТКИ

Выше описаны состав и функции ТМ разработки ППП. Для полноты изложения необходимо привести описания отдельных технологических процессов, входящих в конкретную ТЛ (технологический маршрут приведен на рис. 8.8 и схема разработки на рис.8.9).

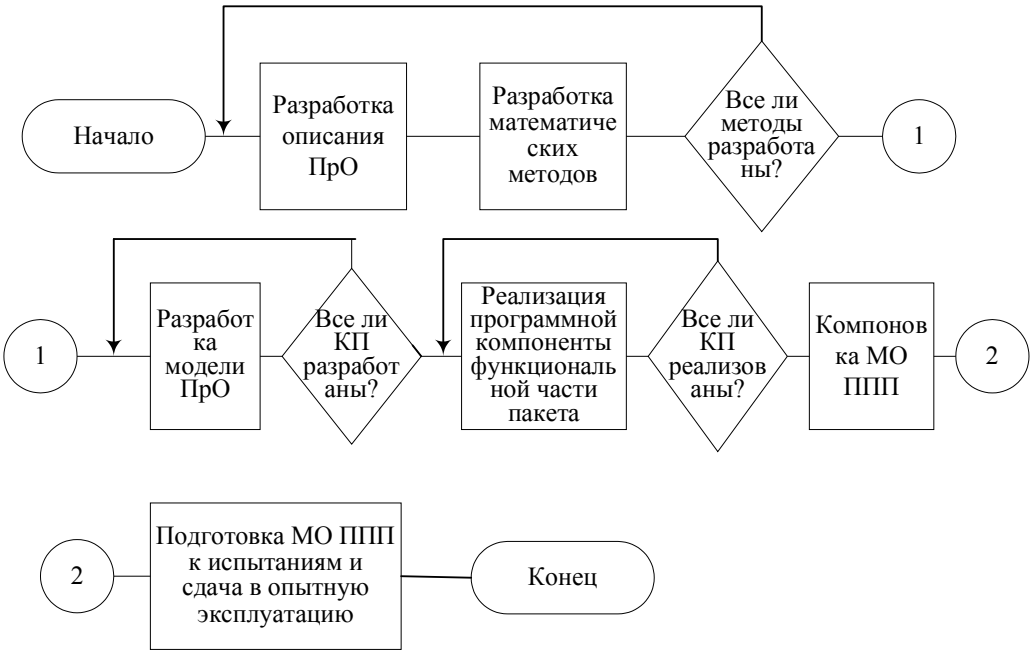


Рис.8.8. Технологический маршрут и карта ТЛ

Рис.8.9. Схема процесса разработки методо–ориентированных ППП

ТП «Разработка описания проблемной области». Исходные данные для данного ТП – информация, содержащаяся в документе «Исходные требования на разработку ППП», которая включает описание постановок задач ПрО, выполняемые проблемными программистами и постановщиками задач. Цель данного процесса состоит в формализации задач и установлении соответствия между понятиями в постановках задач, а также в определении характеристик и параметров математической модели, задаваемой на множестве входных и выходных данных ПрО.

В завершение процесса составляется документ «Описание проблемной области», содержащий спецификации задач, которые оформляется на бланке ПТД и

содержат следующую информацию: имя задачи, описание функции и математической модели задачи, описание понятий ПрО, включая их имена, статус и семантику. Информация о спецификациях задач накапливается при выполнения операций данного ТП.

Организационная структура. На рис. 8.10. приведена структура и состав исполнителей, связи между которыми определяют действия при выполнении работ.

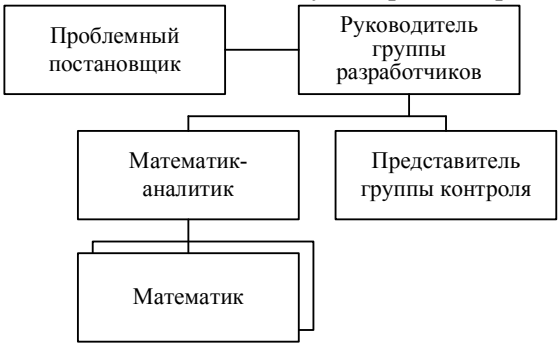


Рис.8.10. Организационная структура процесса

Руководитель группы разработчиков уточняет исходные требования на создание МО ППП, затем совместно с математиком-аналитиком определяет множество задач, входных и выходных понятий, а математики – математические модели задач. Работы контролируются с целью проверки полноты и непротиворечивости описания ПрО.

Состав процесса. Процесс состоит из четырех операций, описываемых ниже.

Операция 01 предназначена для анализа исходных требований на разработку МО ППП. В результате ее выполнения формируется информация об именах задач ПрО и их функциях, которая фиксируется на бланке ПТД.

Операция 02 предназначена для выделения множества понятий для каждой задачи и присвоения имен, возможных статусов (типа входной или выходной). Операция обладает цикличностью и может выполняться группой математиков для каждой задачи последовательно либо параллельно. В результате ее выполнения определяется вид информации, вносимой в выходной документ, связанный с описанием понятий.

Операция 03 – это последовательность действий по подбору и разработке математических моделей, адекватных задачам ПрО. При этом устанавливается соответствие между понятиями, определенными в постановке задачи, и параметрами математической модели. При выполнении операции не исключается дополнение или уточнение проведенных описаний понятий. Операция обладает цикличностью и может выполняться группой математиков для каждой задачи последовательно либо параллельно. Результатом выполнения является информация, содержащая описание математических моделей, которая заносится в выходной документ.

Операция 04 обеспечивает проверку требований, предъявляемых к разрабатываемым МО ППП, на:

– полноту множества задач, решенных в данной ПрО;

– адекватность математических модулей задач;

– непротиворечивость параметров математических моделей понятиям в постановках задач.

Результаты проверки фиксируются в протоколе контроля и используются при повторном выполнении ранее рассмотренных операций в соответствии с инструкцией.

ТП «Разработка математических моделей». Цель данного процесса – выбор

существующих и разработка новых методов решения задач ПрО. На основании анализа математических моделей задач и исходных требований к ним выбираются современные методы решения, наиболее полно удовлетворяющие исходным требованиям. Если их не удается найти, то предусматривается операция разработки новых методов решения задач. Данный процесс выполняется аналитиком (рис.8.9) с привлечением математиков. Результат – документ «Описание математических методов», содержащий описание методов и условий их применения для каждой задачи ПрО. Контроль осуществляет представитель специальной группы при участии математика-аналитика.

Данный процесс состоит из четырех операций, семантика которых приводится ниже.

Операция 01 заключается в анализе современных методов решения задач в данной ПрО, наиболее полно удовлетворяющих требованиям. Если такие методы найдены, то проводится обоснование выбора, заключающееся в характеристике метода и описании условий его применимости. В противном случае формулируется обоснование невозможности его применения, оформляется задание на разработку новых методов или доработку существующих с целью удовлетворения всем поставленным требованиям.

В результате выполнения данной операции составляется полное описание выбранных методов со ссылками на источники, а также описание условий их применения для каждой задачи.

Цель *операции 02* – выполнение заданий на разработку новых методов решения задач. Они составляются при выполнении предыдущей операции в тех случаях, если ни один из существующих современных методов не может быть применен для решения какой-либо задачи ПрО. Для вновь разрабатываемых методов проводится теоретическая разработка с полным описанием метода и условий его применения, а также строгое математическое доказательство корректности и адекватности в условиях, определяемых спецификой ПрО.

Если новые методы, полностью удовлетворяющие поставленным требованиям, разработать не удается, то по согласованию с математиком-аналитиком и руководителем разработки принимаются методы, наиболее полно удовлетворяющие требованиям.

Операция 03 выполняется с целью проверки непротиворечивости выбранных и разработанных методов поставленным требованиям. Выполняет эту операцию представитель группы контроля, имеющий соответствующую квалификацию, при участии математика-аналитика.

Операция 04 – завершающая в данном процессе и выполняется с целью оформления документа «Описание математических методов» на основе рабочих документов, разработанных при выполнении операций 01 и 02.

ТП «Разработка модели проблемной области». Цель данного процесса – создание функциональной и системной архитектуры ППП. Для разработки функциональной архитектуры ПрО проводится декомпозиция на отдельные компоненты для каждого из выбранного метода решения задач и их функциональная спецификация. Она отражает иерархию функций КП и содержит формализованное описание функций компонентов, их информационные и управляющие связи.

Разработка системной архитектуры состоит в создании графовой структуры КП

и оформлении спецификаций на комплексы программ и модули, входящие в них.

Графовая структура включает модули и типы взаимосвязей между ними. Спецификации на КП и входящие в пего модули содержат формализованное описание: функционального назначения модуля; входных и выходных данных; связей с другими модулями и т. п.

Для представления спецификаций используется язык проектирования программ. Спецификации оформляются в виде комплекта ПТД, При проектировании системной архитектуры КП используется принцип многоуровневой нисходящей методологии проектирования сложных программ. Процесс завершается подготовкой выходного документа «Системная архитектура модели ПрО», содержащего структурные схемы КП и спецификации на все модули, и КП в целом.

Организационная структура процесса аналогична приведенной на рис.8.9 и ориентирована на проектирование системной архитектуры модели ПрО и подготовку выходных документов по процессу. Основную функцию процесса выполняют аналитики или проблемные программисты под руководством главного программиста. Представитель группы контроля при участии главного проблемного программиста проводит контроль соответствия системной архитектуры ПрО и функциональной. Контроль включает проверку сохранения функций в графовой структуре и непротиворечивости использования данных модулями, входящими в структуру.

Состав операций процесса. Включает четыре операции, описываемые ниже.

Операция 01 выполняется с целью разработки функциональных спецификаций на КП. Суть ее состоит в проведении детального анализа задач и методов их решения с точки зрения их программной реализации в конкретной вычислительной среде. Анализ методов решения и детализация их на автономные компоненты обеспечивают реализацию конкретного метода. В результате формируется представление о модели КП как о некоторой его структуре. Функциональные спецификации являются результатом операции и должны содержать:

– иерархию функций (компонентов) КП и их описание;

– описание связей по управлению и информации между компонентами, включая входные и выходные данные, их типы и формы представления в среде;

– описание требований и ограничений к реализации КП.

Функциональные спецификации представляются в виде архитектур функциональных схем, выполненных с использованием диаграмм Джексона, HIPO-диаграмм, SA-диаграмм и т. п. Операция выполняется применительно ко всем методам, перечисленным в документе «Описание проблемной области».

Операция 02 состоит из анализа и детализации компонентов КП с целью выделения программных модулей и определения их взаимосвязей. Выделяемые модули должны удовлетворять следующим требованиям:

– выполнять одну достаточно простую функцию иди несколько. функций, работающих с одной и той же структурой данных; иметь один вход, один выход;

– возврат управления осуществлять в точку вызова вызвавшего модуля;

– передавать данные через стандартный вызов типа CALL).

Выполнение этих требований упрощает дисциплину взаимодействия программистов, закладывает основы распараллеливания работ по

программированию и документированию программ. Результатом выполнения такой операции является часть системной архитектуры модели ПрО, а именно графовая модель КП, разрабатывающаяся согласно методологии СППО, и ПТД. Эти документы представляют собой паспорт КП и модулей и содержат их общее описание.

Операция 03 предназначена для заполнения ПТД, которые оформляются в КП и отдельные модули, включая описание логики выполнения модулей. Созданные ПТД включаются в состав системной архитектуры модели ПрО.

Системная архитектура, разработанная на основе данной операции, уточняется и дополняется при выполнении следующих процессов. В частности, могут доопределяться некоторые разделы спецификаций и вводиться новые документы, сопровождающие процессы программирования, сборки и тестирования КП.

Операция 04 предназначена для экспертного контроля хода разработки системной архитектуры модели ПрО, заключающегося в проверке соответствия системной архитектуры функциональной архитектуре модели ПрО. Контроль выполняется представителем группы контроля с участием главного проблемного программиста.

ТП «Реализация функциональной части пакета». В данном процессе выполняются работы по программированию, отладке и тестированию КП. Эти работы выполняются в два процесса.

Первый процесс – реализация отдельных модулей. Для распараллеливания работ по программированию, отладке и тестированию модулей главный проблемный программист привлекает группу проблемных программистов.

На втором процессе производится сборка отлаженных модулей в комплекс в соответствии с его графовой структурой. Затем производится тестирование интерфейса модулей. Качество выполнения работ зависит от того, насколько тщательно были разработаны граф КП и спецификации на модули и КП. Процесс считается выполненным и оттестированным для всех КП, реализующих задачи ПрО.

Организационная структура данного процесса соответствует, приведенному на рис.8.9. В ней проблемные программисты выполняют программирование, отладку и тестирование модулей. Контроль за выполнением этих работ, сборку и тестирование КП выполняет главный проблемный программист с привлечением проблемных программистов, а также представитель группы контроля и главного проблемного программиста. Контроль заключается в установлении соответствия спецификации и графа КП результатам тестирования.

Состав операций данного процесса –три операции, описываемые ниже.

В *операции 01* объединены работы по программированию, отладке и тестированию модулей (компонентов) КП. Программирование выполняется на ЯП, способствующем получению наиболее эффективной программы.

Производительность труда программиста при этом предполагается высокой. Последний показатель зависит от того, насколько тщательно была разработана системная архитектура при выполнении процесса ТП03. Реализация программных модулей производится по принципу структурного проектирования, при котором программа представляется в виде графовой структуры и независимо от формы представления каждый из ее операторов заменяется конструкциями ЯП.

Отладка программ начинается после исправления синтаксических ошибок и

ошибок программирования, выявленных при тестировании программ. При этом используются средства отладки, имеющиеся в арсенале средств ОС. Тестирование проводится для выявления семантических ошибок и ошибок проектирования КП и выполняется на основании тестов, оформленных согласно применяемой методике. Критерием завершенности тестирования модуля является корректная обработка всех содержащихся в тесте ситуаций.

Контроль выполнения работ по данной операции проводит главный программист при участии проблемного программиста. При этом проверяются соответствие программы спецификациям модуля и полнота тестирования.

Операция 02 предназначена для сборки и тестирования КП. Процесс сборки и тестирования основан на общепринятой стратегии нисходящего тестирования с использованием драйверов и заглушек для моделирования взаимодействия модулей. Тестирование КП (или его частей) выполняется с использованием технологических методов и средств инструментальной системы ССПМ. Для его проведения составляются план и таблица тестов. План предусматривает тестирование на тестовых данных, моделирующих реальные ситуации, функций КП, связей по управлению и по данным.

Операция 03 предназначена для проверки соответствия текста КП его спецификациям и графовой структуре. Эту операцию выполняет представитель группы контроля при участии главного проблемного программиста.

ТП «Компоновка математического обеспечения ППП». Цель данного процесса – формирование модели функционального наполнения МО ППП. Операции автоматизированы и выполняются с использованием средств ИК. Модель функционального наполнения – это множество исходных и модулей данных, расположенных в библиотеках системы и их каталоги.

Форма представления исходных модулей соответствует требованиям СППО. Результатом выполнения данного процесса является готовый к эксплуатации МО ППП, общение с которым осуществляется на входном языке пакета, являющемся подмножеством языка проектирования пакетов.

Рассматриваемый процесс ТП05 выполняется главным проблемным программистом с привлечением других проблемных программистов. Контроль хода выполнения осуществляет представитель группы контроля при участии главного проблемного программиста. Проверяется соответствие задач графу КП.

Состав операций данного процесса – три операции, которые содержательно описываются ниже.

Операция 01 – первая в реализации модели функционального наполнения МО ППП и выполняется с целью организации библиотеки исходных модулей и библиотеки модулей данных. Выполнение поддерживается средствами инструментального комплекса. Форма представления вводимых модулей соответствует требованиям СППО.

Операция 02 предназначена для создания каталогов задач и модулей. Для этого формируются и подаются на вход инструментального комплекса описания задач, которые составляются согласно требованиям инструментального комплекса с использованием соответствующих средств. Данная операция является завершающей операцией процесса формирования модели функционального наполнения МО ППП.

Операция 03 выполняется с целью подготовки контрольных примеров для

апробации разработанного пакета. В качестве контрольных примеров выбираются тестовые примеры, использовавшиеся для тестирования КП в ТП04. При работе с МО ППП используются средства языка проектирования пакетов СППО. Апробация пакета заключается в проведении счета задач и получении точных результатов. Соответствие описаний задач сформированной модели функционального наполнения графу и исходным требованиям к МО ППП проверяется (операция 03) представителем группы контроля при участии главного проблемного программиста.

ТП «Подготовки пакета к испытаниям». Цель выполнения данного процесса – проведение комплексных испытаний разработанного ППП на соответствие исходным требованиям и сдача его в опытную эксплуатацию. Результат процесса – акт проведения испытаний МО ППП и оформление соответствующей эксплуатационной документации.

Организационная структура аналогична приведенной на рис. 8.9. Главный проблемный программист совместно с руководителем разработки составляет программу и методику испытаний МО ППП. Специалисты группы готовят рабочую документацию на пакет и на отдельные функциональные части МО ППП.

После проверки документов представителем нормоконтроля проводятся комплексные испытания пакета. В них участвуют руководитель группы разработки как отчитывающаяся сторона и проблемный постановщик как сторона, принимающая и оценивающая результат разработки.

Состав операций данного процесса. Процесс состоит из четырех операций, содержательный смысл которых приводится ниже.

Операция 01 выполняется с целью составления программы и методики испытаний разработанного МО ППП. Последняя разрабатывается в соответствии с требованиями, предъявляемыми к методике испытаний МО ППП. При составлении программы испытаний используется описание тестовых примеров, разработанных для отладки КП функциональной части пакета. При этом выбираются примеры, наиболее полно охватывающие проверку КП и использующиеся в качестве контрольных примеров. Общий перечень их составляется таким образом, чтобы охватить все КП разработанного МО ППП. Результатом выполнения операции является документ «Программа и методика испытаний».

Операция 02 выполняется с целью оформления всех документов рабочего проекта согласно требованиям стандартов ЕСПД в системе технической документации на АСУ. Разработка документов ведется на основании ранее подготовленных комплектов документации на стадии рабочего проекта функционального наполнения МО ППП и описаний архитектуры модели ПО.

Операция 03 выполняется для проведения нормоконтроля разработанных документов рабочего проекта. В результате будет получен готовый комплект документов рабочего проекта на МО ППП.

Операция 04 завершает ТП06 по сдаче в опытную эксплуатацию разработанного МО ППП. После проведения комплексных испытаний в соответствии с разработанной программой и методикой, а также подписания акта о проведении испытаний начинается опытная эксплуатация разработанного МО ППП.

8.6. СОВРЕМЕННЫЕ ПОДХОДЫ К СОЗДАНИЮ ПРОГРАММ НА ЛИНИИ

Концепция линии производства ПП. Институтом программной инженерии SEI

(Software Engineering Institute) США [236] разработана новая технология производства программ на линии (Product Line), производящая семейство продуктов. Она представляет собой множество программных систем, которые:

1) определяют общий набор характеристик качества;

2) удовлетворяют специфическим потребностям определенного рынка;

3) разработаны с использованием общего множества готовых ресурсов (КПИ, компонентов, приложений и т.п.).

Для их получения изучаются общие свойства ресурсов, а затем определятся индивидуальные свойства членов семейства программных продуктов.

Процесс создания линейки продуктов – типичный пример проблемно-ориентированного проектирования, когда на основе требований к продуктам линейки анализируются общие черты и индивидуальные отличия членов семейства и определяется модель ПрО (архитектура с членами семейства, которые имеют отдельные, общие и индивидуальные черты.

Понятие линии программных продуктов (Framework for Product Line Practice) сформировалось, как поддержка инженерии ПрО, в задачу которой входит применение подходов и методов для автоматизированного построения разных видов программных продуктов на ней.При этом в рамках домена исследуются рынок и потребности покупателей, строится производственный план, процессы и определяется организация их взаимодействия. На основе анализа потребностей рынка строится технология линии продукта, в которую включаются методы разработки, тестирования и оценки процессов и продуктов линии.

Построение конкретной линейки для разработки программного продукта для некоторого представителя (члена) семейства домена определяются:

– ограничения, свойственные продуктам линии;

– образцы и каркасы, которые могут использоваться на линии;

– производственные ограничения, стратегии и методы;

– набор средств и инструментов для разработки продукта на линии.

На основе этих данных определяются область действия линии и набор базовых средств, строится план создания продукта, который учитывает сроки, стоимость и требования к управлению производством продукта путем:

– контроля плана работ и отслеживания хода построения продукта;

– выявления рисков и управления ими в процессе исполнительской деятельности на процессе проектирования члена семейства;

– прогнозирования стоимостных и технических ресурсов проекта;

– управление конфигурацией, измерением и оценкой качества продукта.

Подход к организации разработки проекта ПП на основе ТПР. Для эффективной разработки программного проекта руководящий состав предприятия должен создать инфраструктуру управления проектом предприятия, включающую в себя ресурсы (технические, программные, людские и др.), оборудование (для создания рабочих мест сотрудников), необходимые средства коммуникации, а также финансовые инвестиции от заказчика на его выполнение. Кроме того должна быть разработана модель управления организацией (рис.8.11).

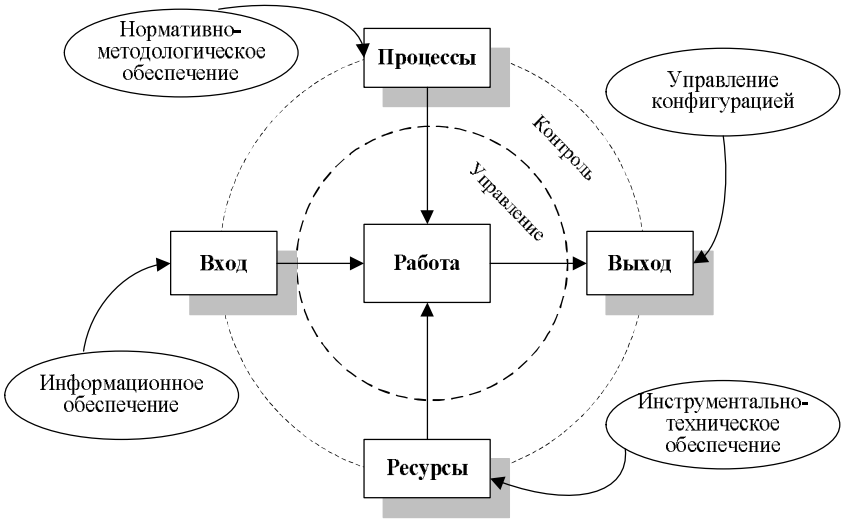


Рис.8.11. Модель процесса управления проектом.

Подбор кадров, т.е. исполнителей разных видов работ – одна из важнейших задач в сфере проектирования. Согласно международного стандарта ISO/IEC 12207-2006 среди исполнителей должны быть технологи, аналитики, программисты, планировщики, контролеры (верификаторы, тестировщики и др.), процессисты (подбор моделей ЖЦ подходящих для данного проекта) и другой дополнительный персонал. Новая структура коллектива разработчиков ПП приведена на рис.8.12.

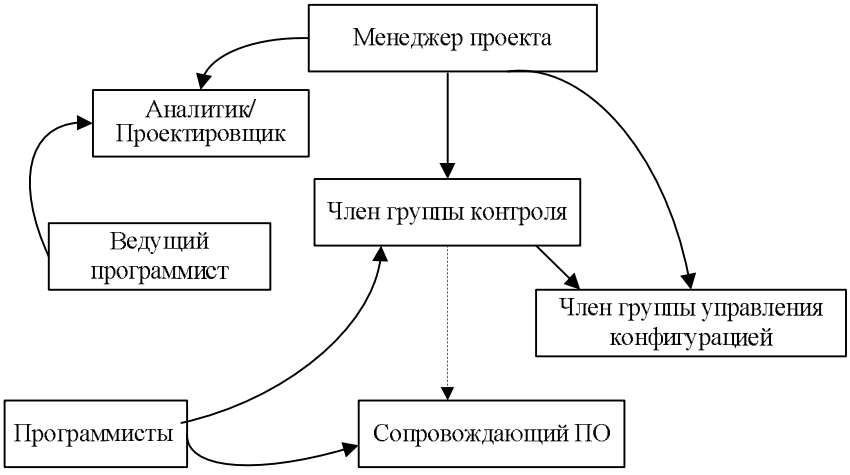


Рис.8.11. Современная структура коллектива разработчиков ПП

Вывод. Предложенные в данной главе принципы и концепции построения ТЛ апробированы на конкретном классе задач обработки данных СОД. Представлена структура, описание ТП и операции процессов, выполняемых в среде конкретной ТЛ (ППП, научно-исследовательских задач и др.) для построения наборов соответствующих программ. Данный метод построения ТЛ в настоящее время получил новое представление в линии продуктов (Product line), целью которых изготовление продуктов для некоторого сегмента рынка.

Глава 9

ВОПРОСЫ УПРАВЛЕНИЯ РАЗРАБОТКОЙ ПРОГРАММНЫХ СИСТЕМ ПО ТЕХНОЛОГИИ СБОРОЧНОГО ПРОГРАММИРОВАНИЯ

В данной главе рассматриваются различные аспекты применения технологии сборочного программирования к разработке ПС с использованием ТЛ. Рассматриваемые аспекты затрагивают организацию процесса разработки ПС, планирование, управление качеством, автоматизированную оценку надежности и качества программного продукта.

Организационный аспект управления разработкой. В разработке ПС принимают участие разные специалисты, связи между которыми в значительной степени влияют на результат разработки. В литературе предлагался и обсуждался ряд форм организации коллектива разработчиков [6, 7, 140 и др.] — бригада главного программиста, хирургическая бригада, бригада без специализации и др. Однако практически в каждом коллективе складываются свои организационные формы, которые, как правило, определяются видом объекта и квалификацией руководителя разработки.

При этом основная задача организации разработки состоит в расстановке специалистов по работам в соответствии с их квалификацией для получения качественного продукта и в срок. При расстановке необходимо иметь дополнительный ресурс (в случае болезни, увольнения специалистов и т. п.), способный продолжить разработку. Простое добавление специалистов в «пиковые» моменты разработки не только не приводит к желаемым результатам [31, 32], но и снижает производительность тех, кто работает.

На эффективность управления разработкой влияют:

– виды и свойства разрабатываемых объектов;

– набор процессов в ТЛ и их обеспеченность нормативно-методическими, регламентирующими и справочными материалами;

– принципы планирования трудозатрат, учета и регулирования хода разработки;

– нормирование труда, времени разработки и контроля работ и др.

Приведенные факторы характеризуют процессы разработки ПС в рамках применения любых технологий программирования. В этой главе рассматриваются вопросы управления разработкой, связанные с применением метода сборочного программирования.

9.1. ИНЖЕНЕРНЫЕ МЕТОДЫ И ПРИНЦИПЫ ПЛАНИРОВАНИЯ ВЫПОЛНЕНИЯ РАБОТ

Разработка ПС как любого нового продукта, с одной стороны носит характер творческой деятельности, а с другой — представляет собой регламентированную последовательность конкретных процессов в изготовлении изделия. Эта двойственность присуща практически всем процессам, сопровождающим создание новых программных средств. Используя традиционные подходы к программированию, разработчику ПО приходиться решать как вопросы собственно разработки (исследование предметной области, проектирование изделия, выделение модулей и т. д.), так и вопросы, связанные с управлением отдельными процессами и всей разработкой в целом. При этом, как правило, обе группы вопросов тесно связаны между собой.

Использование различных технологий программирования как раз и предназначено для максимального отделения творческих процессов от работ по управлению разработкой ПС. О качестве программной технологии можно судить по достижению этой цели. Регламентация процесса разработки программ по ТЛ должна сочетаться с экономико-производственными методами управления программированием: планирования трудозатрат в соответствии с ТЛ и наличием специалистов определенной квалификации, учет и контроль выполняемых работ, оценка результатов разработки и т.п.

Инженерные методы разработки. Совокупность методов, используемых для производства ПС, будем называть инженерными методами разработки ПС. Они относятся к промышленным методам разработки, поскольку основываются на организации и управлении разработкой ПС по ТЛ, а также на методах планирования по технологическому маршруту или сетевому графику, оценки результатов разработки и деятельности специалистов, изготавливающих программный продукт.

При использовании ТЛ инженер-исполнитель имеет в своем распоряжении карту процесса, где указаны операции для выполнения, требуемые им входные и выходные данные, применяемые методы, средства и инструменты разработки и контроля результатов труда. Одновременно с перечисленными выше факторами, рассматриваемые методы включают управляющие воздействия, направленные на соблюдение сроков разработки в соответствии с сетевым графиком и расходами на разработку и т. д.

По мнению многих зарубежных и советских авторов, созданное ПС отображает организационные формы и процессы человеческой деятельности в организации. Поэтому созданные на процессе ТПР ТЛ формализуют результаты деятельности высококвалифицированных специалистов, закладывающих свои знания в процессы регламентированного создания конкретных типов программ, и являются средством для обеспечения технологичности и качества программного продукта.

Инженерные методы основываются на комплекте технологических документов, ведении разработки по конкретной ТЛ, а также инструктивно-методических документах, регламентирующих организацию и управление разработкой ПС высокого качества с применением технологических модулей, реализующих методы разработки. Такой набор документов разрабатывается исходя из потребностей обеспечения инженеров-исполнителей программными и экономичными методами управления разработкой классов программ на основе ТЛ.

К основным принципам инженерного подхода относятся: принципы производственной организации, обеспечения технологичности и качества, планирования работ и разных затрат (трудовых, программных, технических и др.).

Принцип производственной организации. В отличие от творческого характера работ, на процессе ТПР при определении ТЛ (анализ ПрО, выбор подходящих средств описания ПрО, корректировка и выбор ограничений на проект и т. д.) инженерные методы, используемые на процессе прикладного программирования, отличаются строгой последовательностью ТП и ТО, оформленных в производственные процессы. В них имеется специализация операций, применяемых методов и инструментов, а также исполнителей, деятельность которых учитывается в соответствии с экономическими методами оценки труда. Необходимым условием успешного использования данного принципа является планирование программных работ, управление которыми осуществляет технологическая служба предприятия, выполняющая контроль соблюдения технологии и оценку объекта разработки, планирование работ с элементами нормирования, совершенствование технологических процессов и др.

Принцип планирования. Для соблюдения сроков разработки программного объекта во многих проектах проводится планирование. Известно около 20 различных моделей планирования, включающих расчет трудозатрат, сроков и требуемого числа специалистов. Каждая модель разрабатывалась индивидуальным способом на основе использования накопленных данных о проведенных разработках. В качестве исходных данных и критериев в моделях применялись разные характеристики продукта и среды разработки. Большинство методов основано на прогнозировании объема продукта, выражаемого в числе строк (операторов, команд).

Предполагаемый объем делился на среднестатистическое значение производительности (число строк) одного программиста в год. В результате получалась планируемая трудоемкость. Производительность разработчиков ПС колеблется в больших диапазонах в зависимости от применения ЯП высокого уровня (производительность повышается в 4—5 раз по сравнению с языком типа Ассемблер), форм организации работ в коллективе, режима работы программистов на ЭВМ, производительность которых повышается на 20 % при диалоговом режиме и возможности доступа к ЭВМ в любое время.

Эти данные получены в [6, 209] при проведении исследований 12 моделей затрат на одном гипотетическом проекте и состоят в следующем:

– отношения между самым низким и самым высоким значениями затрат выражаются как 1:7,6;

– отношения между самым коротким и самым длительным сроком разработки проекта выражаются как 1:1,9.

Все модели проверялись с одинаковыми данными для производительности, сложности и др. в разных коллективах. Для определения числа разработчиков планируемая трудоемкость делилась на время разработки, которое определялось в зависимости от заданных сроков разработки объекта. Естественно, что такая модель расчета является поверхностной и, кроме того, зависит от организации управления разработкой и требованиями создания высокого качества программного продукта. Данное обстоятельство приводит к увеличению сроков разработки и к снижению производительности труда, что в конечном итоге увеличивает общие

трудозатраты.

В производственных условиях проводится *текущее планирование работ*, при котором решается задача составления выполнимого (достижимого) плана работ Y по ТЛ, основными данными для составления которого являются:

– общий плановый срок разработки $[t_0, T]$, где t_0 и T – начальный и конечный сроки разработки;

– объем работ $W = \{W_i\}$ с учетом переделок;

– требуемые ресурсы $R = \{R_l, R_m\}$, где R_l – людские; R_m – материальные (технические и программные);

– нормы потребления людских ресурсов по всем ТП, $(i = \overline{1, N})\ NR_l$ и др.

Формально план работ записывается в следующем общем виде:

$$Y = Y(t_0, T, W, R_l, R_m, NR_l, f), \qquad (9.1)$$

где f – случайные факторы (ошибки при выполнении плановых работ на ТП, сбои технических средств и др.), а также факторы, связанные с появлением средств новой техники и программного обеспечения.

В качестве механизма управления разработкой может использоваться сетевой план-график работ и контроль его выполнения. В условиях производственной организации ведения разработки на основе ТЛ отсутствие его зачастую приводит к неуправляемости процессу.

Принцип обеспечения технологичности. Понятие технологичности целиком и полностью связано с наличием технологии и с полным соблюдением всех ее требований и правил. Технологичность — это понятие, включающее технологичность ПП и технологичность процесса разработки.

Технологичность ПП – это соответствие ПП потребительским свойствам и определенным функциям ПрО. Ее обеспечение основывается на заложенных в ТЛ элементах типизации, унификации и стандартизации конструктивных элементов ПП, применяемых моделях и заготовках, а также готовых повторно используемых программных объектах из фондов коллективного пользования. Технологичность определяет способность ПП к эксплуатации.

Технологичность разработки – это регламентированный (упорядоченный набор процессов, операций и процедур их выполнения), конструктивный (методом сборки из готового) и инструктивный (методическое и инструктивное обеспечение процессов ТЛ) порядок работы, а также организация управления разработкой ПП. Ее обеспечение основывается на применении эффективных методов ведения разработки ПП, воплощенных в ТЛ (или ТП) и направленных на повышение качества и производительности труда, минимизации затрат и времени на разработку ПП.

Принцип планирования трудозатрат. Исходя из результатов исследований [8] основное распределение трудозатрат на процессах разработки приходится на сопровождение и поддержку проекта.

Поэтому работы по планированию и нормированию в условиях производственной организации являются актуальными.

Взятый за основу подход разработки программ по ТЛ позволяет определить трудозатраты (в человеко-днях) на разработку программ исходя из следующих исходных данных;

5. N– количество процессов на ТЛ;

6. J_i– количество операций на $ТП_i$ ($i = \overline{1, N}$);

3) $\sum_{i=1}^{N} P_i + P$– директивная продолжительность разработки всей программной системы, где P_i– минимально требуемая продолжительность (в днях) для i-го процесса; P– резерв времени, который остается до планового срока и используется для варьирования исходными продолжительностями процессов;

4) x_{ij} ($i=\overline{1, N}$), $j=\overline{1, J_i}$) – объем ПС (в операторах), задаваемый экспертной оценкой. ПС должно разрабатываться на i-м процессе и j-й операции;

5) T_{ij} – производительность труда (в операторах в день) при выполнении j-й операции на процессе i.

За искомые величины примем: γ_i – оптимальная продолжительность i-го процесса, m_i – количество программистов, необходимых дли выполнения j-й операции на i-м процессе. Тогда $\sum_{j=1}^{j_i} m_{ij}$ задает количество исполнителей, необходимых для разработки ПС на i-м процессе, $F = \max \sum_{i=\overline{1,N}} \sum_{j=1}^{j_i} m_{ij}$ – максимальное количество специалистов, необходимых для реализации всей прикладной системы на основе ТЛ.

Исходя из введенных обозначений, математическую модель задачи планирования разработки ПС сформулируем следующим образом:

$$F^0 = \max \sum_{i=\overline{1,N}} \sum_{j=1}^{j_i} m_{ij} \Rightarrow \min,$$

$$x_{ij} \le \gamma_i m_{ij} T_{ij}, \gamma_i \ge P_i, \tag{9.2}$$

$$\sum_{i=1}^{N} \gamma_i \le \sum_{i=1}^{N} P_i + P \tag{9.3}$$

Величины x_{ij}, T_{ij}, P_i, P известны, а γ_i и m_{ij} – целевые переменные. Ограничения на m_{ij} выражаются в виде $m_{ij} > 1$.

Формула (9.2) означает, что x_{ij} – количество операторов в ПП должно быть создано m_{ij} специалистами за γ_i дней, а неравенство (9.3) означает, что разработка ПП должна быть выполнена в плановый срок.

Задача имеет много вариантов с большим перебором, требующих некомбинаторных методов решения, поэтому перейдем от этой математической модели к задаче линейного программирования путем изменения критериев и линеаризации ограничения (9.2). Для этого в F^o потребуем ппривеления всех сумм $\sum_{j=1}^{j_i} m_{ij}$ к минимально возможной величине. В этом случае

$$F^o = d_1 \sum_{i=1}^{N} \left(\sum_{j=1}^{j_i} m_{ij} - \frac{\sum_{i=1}^{N} \sum_{j=1}^{j_i} m_{ij}}{N} \right) + d_2 \sum_{i=1}^{N} \sum_{j=1}^{j_i} m_{ij} \Rightarrow \min, \tag{9.4}$$

где d_1, d_2– управляющие параметры.

Введем дополнительные переменные $y_i \geq 0$ и $z_i > 0$ такие, что

$$y_i - z_i = \sum_{j=1}^{j_i} m_{ij} - \frac{\sum_{i=1}^{N}\sum_{j=1}^{j_i} m_{ij}}{N}$$

Тогда после линеаризации (9.2), получим:

$$x_{ij} - \gamma_i^0 m_{ij} T_{ij} - m_{ij}^0 \gamma_i T_{ij} \leq - \gamma_i^0 m_{ij}^0 T_{ij}\ , \qquad (9.5)$$

Здесь γ^0, m_{ij}^0 соответствуют начальным приближениям, γ_i^0 – управляющим параметрам, которые выражены следующим соотношением:

$$m_{ij}^0 = \frac{x_{ij}}{\gamma_i^0 T_{ij}}$$

Исходя из этих предположений, получаем задачу линейного программирования

$$F^0 = d_1 \sum_{i=1}^{N}\left(\sum_{j=1}^{j_i} m_{ij} - \frac{\sum_{i=1}^{N}\sum_{j=1}^{j_i} m_{ij}}{N}\right) + d_2 \sum_{i=1}^{N}\sum_{j=1}^{j_i} m_{ij} \Rightarrow \min,$$

$$x_{ij} - \gamma_i^0 m_{ij} T_{ij} - m_{ij}^0 \gamma_j T_{ij} \leq - \gamma^0{}_i\ m_{ij} T_{ij}\ ,\ \gamma_i \geq P_i\ , \qquad (9.6)$$

$$\sum_{i=1}^{N} \gamma_i \leq \sum_{i=1}^{N} P_i + P\ ,$$

$$m_{ij}^0 = \frac{x_{ij}}{\gamma_i^0 T_{ij}}\ ,\ m_{ij} \geq 1,$$

$$\frac{\sum_{i=1}^{N}\sum_{j=1}^{j_i} m_{ij}}{N} - \sum_{j=1}^{j_k} m_{ij} - y_i + z_i = 0$$

При решении этой задачи будут получены γ_i, m_{ij} (нецелочисленные), с помощью которых определяются целочисленные значения базовой модели, т. е. будет получен пробный вариант решения задачи планирования трудозатрат:

Варьируя значениями управляющих параметров d_1, d_2 и $\gamma^0{}_i$, получим окончательное решение задачи планирования, т. е. искомое число специалистов, необходимых для разработки в срок программной системы по заданной технологической линии.

9.2. ВЛИЯНИЕ СЛОЖНОСТИ ПРОГРАММ НА ВЕДЕНИЕ ПРОЦЕССА РАЗРАБОТКИ СИСТЕМ

Сложность программного продукта – базовая характеристика и должна контролироваться и измеряться в процессе хода разработки по ТЛ.

Меры свойств процесса (трудоемкость, производительность и др.) являются измеряемыми, хотя полностью не отражают реального состояния объекта разработки. Меры свойств объектов разработки (количество спецификаций, данных, операторов и т. п.) характеризуют состояние развиваемого объекта в заданный момент времени. Они зависят от *сложности* создаваемого продукта, а,

следовательно, и от процесса его реализации. Кроме того, в тесной связи с этими факторами находятся квалификация и состав исполнителей, производящих разработку.

Рассмотрим понятие состояния разработки S в конкретный момент времени. Оно определяет уровне детализации и конкретизации продукта, которое осуществляется средствами для решения поставленной задачи. Элементами состояния программного объекта могут быть: спецификации; алгоритмы решения задачи; описание алгоритма в ЯП; программные средства как составные элементы обеспечения различной степени готовности продукта; техническая и технологическая документация; набор тестов; контрольные примеры и т.д.

Начальное состояние программного объекта состоит из общих спецификаций, выраженных в виде технического задания или других форм описания требований. Конечному состоянию соответствуют готовое программное средство с необходимым комплектом технической и технологической документации и контрольными примерами проверки его работоспособности. Переход из одного состояния в другое связан с постепенным уточнением начальной постановки задачи, выделением отдельных функций и соответствующих программных компонентов и информационных структур.

Технология ведения разработки объекта с учетом его состояний в ТП отражает процесс развития состояний S, приводящий к последовательному его усложнению. Сложность программного продукта – его внутреннее свойство, которое зависит от поставленной цели. Показатель сложности Δ характеризует взаимодействие между программным объектом и субъектом разработки, является характеристикой, оценка которой представляет значительные трудности [202, 240]. Поэтому рассматривается суммарная сложность, соответствующая элементам, входящим в состав ПС.

Объект разработки при переходе к новому состоянию может претерпевать следующие изменения:

– подвергаться делению на более мелкие элементы (деление систем на подсистемы, программ на модули и т. д.);

– переходить в качественно новое состояние или изменять свои количественные характеристики (переход от описания модуля на языке спецификаций к языку программирования, трансляция исходного текста с получением объектного модуля и т. д.);

– сопровождаться разработкой дополнительных сопутствующих элементов (документации, тестов, контрольных примеров и т. д.).

Такие изменения приводят к увеличению количества элементов разработки. В первом случае число элементов возрастает за счет разбиения, во втором – увеличения количества представлений отдельного элемента, в третьем – создания дополнительных элементов. Тогда процесс развития состояния S можно представить в виде дерева, приведенного на рис. 9.2.

Из каждой вершины, в общем, может исходить несколько ребер. В дальнейшем будем рассматривать только бинарные деревья. Покажем, что любое изменение элемента разработки может быть представлено в виде бинарного дерева.

1. **Деление объекта разработки**. Допустим, объект можно разделить на k более мелких элементов. Этот процесс может быть представлен в виде последовательности состояний $S_1, S_2, \ldots, S_{k-1}$, где S_i связан с выделением i-го

элемента. Соответствующее дерево представлено на рис. 9.3.

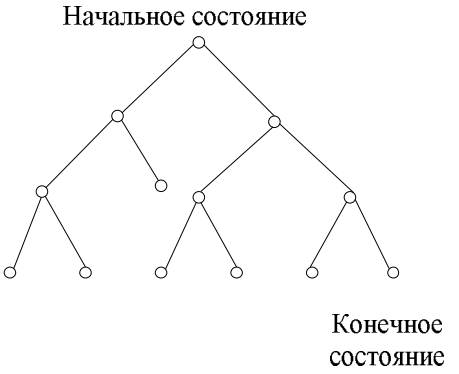


Рис. 9.2. Пример развития состояния

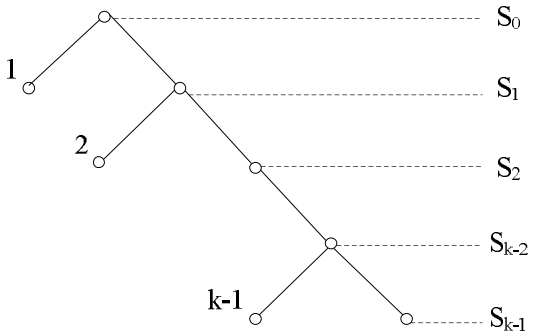


Рис. 9.3. Дерево состояний

2. Изменение качественного состояния или количественных характеристик. Данная ситуация представлена на рис. 9.4. Необходимость сохранять старый элемент объекта связана с возможными ошибками, возникающими в ходе разработки, и возвратом к предыдущему элементу объекта.

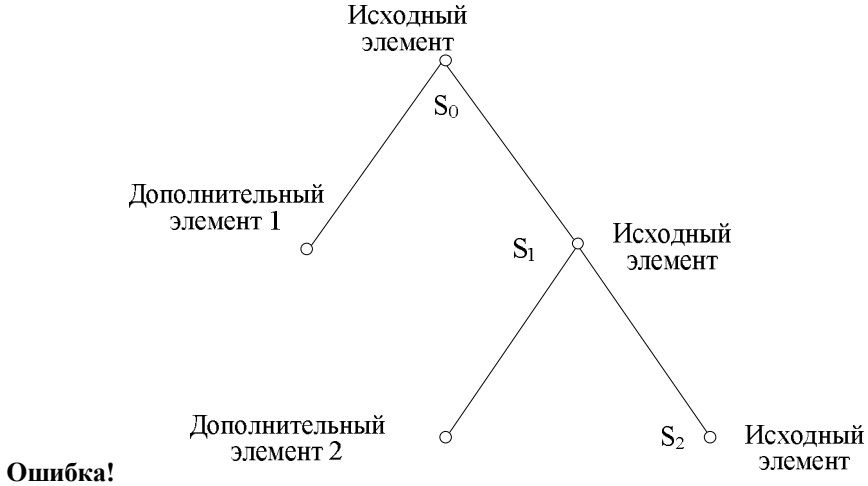


Рис. 9.4. Изменение состояния объекта

1. **Разработка дополнительных элементов**. Пусть для объекта требуется разработать k дополнительных элементов. Тогда процесс описывается последовательностью состояний $S_1, S_2, \ldots, S_k$. Соответствующее дерево представлено на рис. 9.5, которое – бинарное и для его анализа можно использовать соответствующий математический аппарат. В бинарном графе с N вершинами содержится N– 1 дуг. Если каждой дуге сопоставить технологическую операцию, то полное число операций не будет превышать N – 1 (некоторые операции представ-

лены формально и не требуют никаких действий: например, исходный и старый элементы на рис. 9.4 соответствуют одному и тому же объекту).

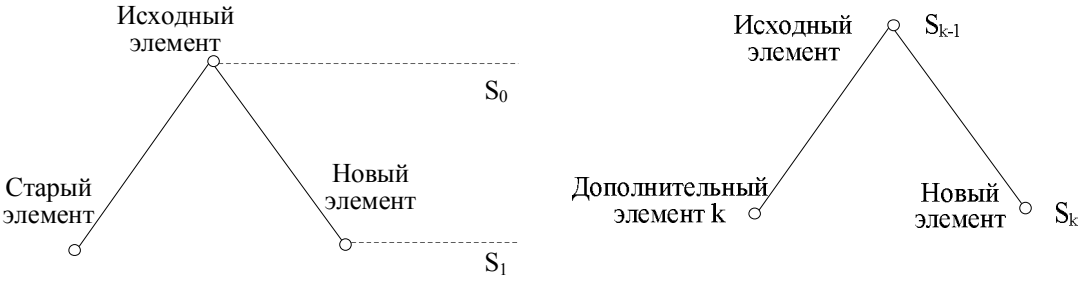


Рис. 9.5. Развитие состояния объекта новыми элементами

Пусть $\overline{\delta}$ – средняя сложность отдельного элемента объекта, а τ– средняя длительность одной технологической операции его преобразования. Общее время разработки определяется приблизительно выражением

$$\Delta = N\overline{\delta}, \qquad (9.7)$$

$$T = (N-1)\ \overline{\tau}. \qquad (9.8)$$

Из формул (9.7) и (9.8) получаем зависимость между сложностью разработки и ее общим временем:

$$\Delta = \frac{T + \overline{\tau}}{\overline{\tau}}\ \overline{\delta}. \qquad (9.9)$$

Отсюда средняя длительность одной технологической операции имеет вид:

$$\overline{\tau} = \frac{T\overline{\delta}}{\Delta - \overline{\delta}}.$$

Переменные $\overline{\delta}$ и $\overline{\tau}$ принимают средние значения. В соответствии с этим предполагается, что любая ТО для любого из соответствующего множества элементов объекта разработки может быть выполнена любым специалистом, входящим в коллектив разработчиков. В реальной ситуации необходимо учитывать общность объектов разработки и ТО, а также квалификацию исполнителей.

Пусть квалификация исполнителя оценивается некоторым коэффициентом q_r, а δ – сложность исходного элемента, δ_i– сложность элемента, полученного в результате применения $ТО_i$.

Тогда длительность операции $\tau_i = \tau_i(\delta_{i-1}, TO_i, q_r)$, $\delta_i = \delta_i(\delta_{i-1}, TO_i, q_r)$ и среднее время выполнения операции представляется так:

$$T = \sum_{i=1}^{N} \tau_i, \quad \Delta = \sum_{i=0}^{N} \delta_i.$$

По существу качество программного продукта будет функцией от τ_i, δ_i, и N. Предполагая, что общая сложность Δ – объективная характеристика и не зависит от конкретных исполнителей, следует, что показатель качества будет меняться в зависимости от значений от τ_i, δ_i, и N или от δ_0, TO_i, q_r, где δ_0 –сложность начального состояния объекта разработки.

Качество программного продукта по ТЛ можно повысить за счет следующих факторов.

1) Количество ТП и операций фиксировано, что позволяет $ТО_i$ разбить на классы однотипных операций.

2) Выделение классов операций позволяет упростить зависимость числа элементов разработки: на одной и той же ТЛ $N = N(\delta_0)$, в то время как в общем случае $N = N(\delta_0, TO_i)$

3) Каждая ТО определяется требованием к квалификации исполнителя q_r, поэтому подбирается состав исполнителей, распределятся между ними обязанности согласно их знаниям и квалификации.

4) При заданном времени разработки T и известном составе исполнителей можно так подобрать τ_i и δ_i, чтобы показатель качества был максимальным. В частности, это позволит дать ответ на вопрос: возможно ли решение поставленной задачи δ_0 за определенное время T при известном составе исполнителей с требуемым показателем качества.

9.3. ОРГАНИЗАЦИЯ И УПРАВЛЕНИЕ КОЛЛЕКТИВОМ РАЗРАБОТЧИКОВ

Управление разработкой программ по ТЛ включает решение вопросов планирования хода разработки, учета и контроля.

Метод планирования требуемого числа специалистов для проведения разработки, описан выше. Здесь рассмотрим средства, способствующие планированию конкретных работ и учету их выполнения. Введем в рассмотрение специальные информационные структуры – карты выполнения работ, разрабатываемые документы, сроки начала и окончания выполнения отдельных операций и квалификацию исполнителя. Такие карты ведутся ответственным исполнителем процесса разработки и могут контролироваться представителем технологической службы. Они создаются на основе технологического маршрута и сетевого плана-графика.

Один из сложных моментов планирования работ – определение норм на каждого специалиста с учетом особенностей процесса. Специалист должен иметь определенный квалификационный уровень (Q) и обладать навыками работ в соответствии со спецификацией процесса ЖЦ (проектирование баз данных, программирование модулей и др.).

На ТП, которые связаны с реализацией программ, трудоемкость на одного специалиста может быть вычислена по формуле

$$A = \left(\frac{1 + (I)^{0,8}}{c\sqrt{QT}} + \frac{D}{d} \right) \frac{R}{B},$$

где I – число операторов; D – число страниц документации; Q– коэффициент квалификации (не более 1); T – коэффициент технологической обеспеченности операции (не более 1); R– коэффициент трудоемкости применения процесса; B = 21,5– константа (средняя) числа дней в месяце; c = 30 – среднее число операторов в день; d = 2 – число страниц в день (усредненное).

Коэффициент квалификации в формуле (9.7) может быть вычислен по формуле

$$Q = \frac{1}{n} \sum_{m=1}^{n} \frac{1}{m} \sum_{i=1}^{m} q_i(i),$$

где q_i – коэффициент квалификации *i-го* специалиста в квартале; n – число кварталов; m – число участвующих в разработке ПС.

Коэффициент квалификации определяется экспертным путем и задает отношение производительности труда данного специалиста к средней производительности для данной категории разработчиков, к которой он относится.

Коэффициент T в формуле (9.7) предложено вычислять по формуле

$$T = \frac{T_i + T_M + T_n + T_c}{4},$$

*гд*е T_i – коэффициент применяемого языка *(*T_i = 0,6 для Ассемблера; 0,7 – для макроассемблера; 0,8 – для ЯП; 1,0 – современных ЯП типа Паскаль, АДА и др.; T_M – коэффициент применяемости программных методов на ТЛ (если метод не применяется, то $T_м$ = 0,5, в противном случае – $Tм$ = 0,8); T_n– коэффициент применяемости инструментальных средств *(*T_n*=0,8* – традиционные методы и T_n = 1,0 – новые); T_c – коэффициент ОС (определяется на основе экспертных оценок, учитывающих уровень технологичности этой ОС).

Учет выполнения работ по ТЛ проводится на основании графика в целях получения информации о состоянии работ за определенный промежуток времени и использовании этой информации для контроля и регулирования для достижения заданных показателей качества.

В современных условиях, когда требуется высокое качество выпускаемой продукции, учет информации и контроль хода разработки являются необходимым условием обеспечения промышленного способа разработки.

Контроль за ходом выполнения разработки ПП проводится для выявления отклонений фактических показателей от плановых и формирования определенной информации о характере и причинах отклонений. Результаты работ, подвергающиеся контролю, представляют собой карты выполнения работ, ПТД и программы, формируемые в процессе разработки. Результаты контроля оформляются протоколом, являющимся основанием для дальнейших доработки и разработки.

Операция контроля, предусматриваемая как заключительная на каждом ТП, регламентирует вид документов, подлежащих контролю, и результат. Контроль проводится специальной группой, входящей в состав технологической службы. Руководитель этой группы должен иметь высокую квалификацию и знание объекта контроля. В состав группы входят ответственный (старший) технолог и контролеры техпроцессов ТЛ. Кроме того, в эту группу могут входить и уполномоченные по стандартизации и нормоконтроля выпускаемой документации.

Содержательный контроль – это контроль соблюдения технологической дисциплины (полнота и корректность ПТД), правил и условий выполнения каждой операции (применения методов, средств и инструментов), а также нормативов, установленных в соответствующих технологических документах для применения.

Контроль технологичности и качества, проводимый контролерами ТП, а на заключительном процессе совместно с главным технологом осуществляется на основе методов, которые будут рассматриваться ниже.

Определение затрат на разработку программ по ТЛ. Каждая ТЛ включает процессы предпроектных и проектных исследований, являющихся творческой частью работ. Затраты на их ведение (методика и форма фиксации результатов исследований приводится в описаниях к ТЛ) не могут оцениваться количественно

из-за неопределенности единиц измерения творческой части программистской деятельности.

На остальных процессах ТЛ могут применяться оценки затрат на разработку, связанные, например, с такими единицами измерения, как количество строк, операторов и т.п., разрабатываемых в ходе ведения работ, учитываемых и контролируемых в базе проекта.

Процесс сборки на ТЛ основан на применении метода сборочного программирования. Основные затраты идут на создание новых компонентов, на анализ и комплектацию готовых КПИ и на определение «цены модульности» при их сборке. В связи с этим в расчет производительности труда входят затраты для вновь создаваемых объектов по ТЛ и затраты на определение интерфейса готовых компонентов.

Минимизации затрат на определение интерфейса модулей. С этой целью рассмотрим вначале задачу оценки вариантов компоновки структуры программы *P* по исходному графу. Применяя метод упорядоченного перебора вариантов связи модулей по данным (экспортируемым и импортируемым) на основе анализа операторов вызова, определим коэффициент $K^{p}_{ИН}$ информационной независимости модулей программы *P:*

$$K^{p}_{\text{ИН}} = \sum_{i=1}^{N} \frac{\xi_{nl}}{\xi_{p}}$$

где ξ_{nl} – мера связанности по данным *l*-модуля; ξ_{p}– мера связанности по всем данным программы *P* и определяется как суммарная сложность описания межмодульных интерфейсов, входящих в *P:*

$$\xi_{\text{p}} = \xi_{\text{n}} - \sum_{l=1}^{N} \xi_{nl} \ ,$$

«Цена модульности» создаваемой программы *P* из модулей , т.е. объем интерфейса $V_{ДОП}$ (в строках или командах) создаваемых интерфейсных модулей-связи для *N* разноязыковых модулей, можно вычислить по формуле

$$V_{\text{ДОП}} = \sum_{l=1}^{\frac{N}{2}} d(M'_{jk})\xi_{nl} \ ,$$

где $d(M'_{jk})$ – длина модуля–связи для M_j и M_k. Объем создаваемой программы

$$V_{p} = (\text{V}_{\text{доп}} + \sum_{l=1}^{\frac{N}{2}} d(M^{l}{}_{L}) K^{P}_{ИН} \Delta V),$$

где $\Delta\text{V} = \dfrac{V - V_{ДОП}}{N}$ – коэффициент отклонения проектного объема *V* от полученного; $d(M^{l}_{L})$ – длина *l*-модуля в одном языке *L*.

Зная объем $\text{V}_{\text{доп}}$ и число исполнителей на операции сборки, можно подсчитать трудоемкость выполнения операции создания интерфейса. В затраты на компоновку модулей входят также затраты для определения необходимого состава модулей, определения их функциональной пригодности и соответствия типов

входных данных типам и структурам данных реализуемой ПрО.

Стоимостные затраты сборочного создания ПС. Данные затраты определим следующим образом:

$$C = C_{аз} + C_{зм} + C_{сб},$$

где $C_{аз}$ – стоимостные затраты на анализ задачи ПрО; $C_{зм}$ – на анализ возможностей использования готовых объектов для реализации задач ПрО; $C_{сб}$ – на сборку комплекта модулей.

Затраты на анализ задач $C_{аз} = \sum_{r=1}^{N} b_r C_1(D_r)$,

где $C_1(D_r.)$ – затраты на анализ данных r-задачи ПрО ($r = \overline{1, N}$ число задач); $b_r = 1$, если задача выбрана для реализации функции ПрО, 0 – в противном случае.

Определим затраты на анализ возможностей использования имеющегося набора функциональных модулей для реализации требуемых задач СОД или АСУ:

$$C_{зм} = \sum_{i=1}^{N} \sum_{r=1}^{N_1} a_{rj} C_2(M_{jr}),$$

где $C_2(M_{jr})$ – затраты на анализ функциональных возможностей j-модуля для решения r-задачи ($j = \overline{1, N}$ —число модулей); $a_{rj} = 1$, если j-модуль используется для выполнения r-задачи, 0– в противном случае. Затраты на сборку модулей определяются следующим образом:

$$C_{сб} = \sum_{j=1}^{N} \sum_{r=1}^{N_1} \sum_{k=1}^{N_r} d_{jrk} C_3(V(M'_{jr})),$$

где C_3 $(V(M'_{jr}))$, в которой $V(M'_{jr}) = V_{ДОП}$ – затраты на сборку M_j и $M_г$ модулей; $d_{jrk} = 1$, если k – это параметр из набора $x=\{x_2,..., x_l\}$ и является входным для j-модуля r-задачи ($l = \overline{1, N_2}$ – число параметров), 0 – в противном случае. Таким образом, формула (9.20) приобретает вид

$$C = \sum_{r=1}^{N} b_r C_1(D_r) + \sum_{j=1}^{N} \sum_{r=1}^{N_1} a_{rj} C_2(M_{jr}) + \sum_{j=1}^{N} \sum_{r=1}^{N_1} \sum_{k=1}^{N_2} d_{jrk} C_3(V(M'_{jk})). \qquad (9.10)$$

Основными ограничениями данного выражения являются:

1) необходимость реализации требуемых функциональных задач ПрО

$$\sum_{r=1}^{N} b_r \geq N_0,$$

где N_0 – число реализуемых функциональных задач;

2) совместимость выбираемого набора модулей с языковыми средствами среды программирования a_{rj} $\rho_{jL} = \rho'_L$, в которой $\rho_{jL} = 1$, если j-модуль описан в ЯП высокого уровня и $\rho_{jL} = 0$ – в противном случае; $\rho'_L = 1$, если в среде программирования имеется система программирования в языке L и $\rho'_L = 0$ в противном случае;

3) решение r-задачи за время T_r

$$\sum_{j=1}^{N_j} a_{rj}\tau_r \le T_r$$

4) объем, необходимый для реализации агрегата:

$$\sum_{i=1}^{I} V_i \le V_{\max}$$

где $V_{\max}$ – максимальный объем допустимой памяти для загрузки в нее сформированного агрегата.

Расчет стоимостных затрат, выраженных формулой (9.12), является трудоемким процессом, поэтому необходимо минимизировать каждую ее составляющую.

Повышение производительности труда разработчиков. Повышение производительности и достижение требуемого качества ПС зависят от:

– правильности и полноты методов и средств, выбранных для реализации данного вида ПС и представленных упорядоченной последовательностью (исходное качество процесса) на ТЛ;

– контроля соблюдения технологии ТЛ и результатов изменения состояний объекта для определения степени отклонения от регламентных ТМ и средств, а также для определения погрешностей отклонения отдельных свойств текущего состояния объекта от установленных метрик и определения максимума погрешности на конечном процессе ЖЦ как степени несоответствия объекта разработки требованиям заказчика;

– унификации и стандартизации конструктивных элементов объекта разработки (функциональные заготовки и готовые модули) и их качества;

– спецификации и квалификации субъектов (умение программиста работать на инструментальных средствах и др.), автоматизирующих выполнение операций ТЛ;

– организации и управления разработкой (планирование норм, учет, контроль хода разработки и др.).

9.4. УПРАВЛЕНИЕ КАЧЕСТВОМ ПРОГРАММ ПО ТЕХНОЛОГИЧЕСКИМ ЛИНИЯМ

Под управлением качества ПС будем понимать действия (операции), осуществляемые в ходе разработки ПС по ТП в целях установления, обеспечения и поддержки необходимого уровня качества.

Одним из основных требований к ПС является обеспечение их высокого качества, что может быть достигнуто системой управления качеством.

Под *качеством программных продуктов* будем понимать совокупность свойств, обуславливающих его пригодность к использованию по назначению при регламентации совокупности условий функционирования. Каждое свойство является его качественной характеристикой и в процессе разработки может выражаться определенными признаками, которые могут оцениваться экспертным и аналитическим путями.

Для разных классов программных объектов на процессе разработки требований вырабатывается номенклатура показателей качества и устанавливаются их базовые значения в соответствии с имеющимся образцом. Показатели включаются в описание процессов, и на их основе проводится контроль качества.

Обеспечение качества представляет собой совокупность планируемых и

систематически проводимых мероприятий, направленных на удовлетворение заданным требованиям к качеству. Достижение заданных показателей качества ПС в значительной степени зависит от применяемых методов программирования и системы управления качеством.

Многие методы управления качеством программных продуктов [6-9 и др.] ориентированы либо на количественную оценку готового результата, либо на оценку надежности, получаемую на основе ошибок, фиксируемых при комплексных испытаниях объекта разработки. Методы оценки надежности позволяют установить пригодность или непригодность созданного программного продукта к употреблению.

Узким местом многих методов оценки качества и надежности как одного и свойств является их слабая ориентация на начальные процессы разработки ПС. В [162] предложены прогнозирующие модели надежности, применяемые на процессе программирования и сборки программного продукта. Однако для постепенного достижения требуемого качества требуется применять методы экспертной и количественной оценок промежуточных результатов (состояний) продукта.

Ряд работ посвящен методам экспертной оценки [29, 41, 42, 87 и др.], которые требуют проведения мероприятий по созданию экспертных групп и методик контроля получаемых на процессах признаков продукта, выражаемых качественно или количественно.

В работе [92], в частности, для систем, работающих с базами данных, предлагается уделять большое внимание не только процедурам экспертизы, распределенным по циклу создания проекта, но и психологическим аспектам работы экспертной комиссии, затрагивающей интересы коллективов разработчиков системы.

В результате исследований по управлению качеством ниже будет приведен ряд моделей надежности, которые получили развитие в связи со спецификой предметной области СОД и принятой организацией контроля, сбора статистической информации в ходе разработки программного продукта определенного вида по ТЛ. Предложено на каждом процессе иметь операцию контроля проектных решений, принимаемых при выполнении преобразования состояния объекта разработки.

Работы на следующем процессе начинаются после того, когда результаты предыдущего процесса удовлетворяют требованиям контроля качества.

Оценка качества – это количественное определение значений показателей свойств как создаваемого программного объекта, так и инструментов, поддерживающих процесс его разработки (трансляторов, редакторов, отладчиков и т. п.), на основе ТЛ.

Основная цель оценки качества в ходе разработки – проверка правильности выполнения операций ТП, выявление отклонений фактических показателей качества от плановых, а также сбор и формирование текущей информации о характере и причинах отклонений. В табл. 9.1 приведены задачи управления качеством, ранжируемые по процессам общего назначения.

Модель качества разрабатывается на процессе ТПР при создании ТЗ на проектируемое ПС. Модель учитывает структуру ТЛ, в которой указываются ТО контроля атрибута или показателя качества в ходе разработки и формула оценки. Рассмотрим показатели свойств качества программ на ТЛ более конкретно.

Табл и ц а 9.1

Наименование процесса	Решаемые задачи
Разработка ТЗ и спецификаций требований	Разработка моделей качества
Разработка ПС	Управление качеством. Сравнение фактических показателей качества с проектными
Испытание ПС	Оценка полученных и эксплуатационных показателей. Экспертиза технической документации. Оценка научно-технического уровня ПС
Изготовление ПС	Оценка качества изготовления ПС и документации
Эксплуатация ПС	Оценка эксплуатационных показателей качества. Заключение о соответствии исходным требованиям
Сопровождение ПС	Оценка удобства сопровождения ПС

Показатели свойств назначения. В этом показателе оценочный показатель *сложность* характеризуется объемом, выраженным количеством операторов исходного текста:

$$V = \sum_{i=1}^{N} (k \times O[i]),$$

где *0*[*i*] – количество операторов в *i*-модуле; *k*– уровень ЯП (*k=1)* для Ассемблера и *k*=10 для ЯП высокого уровня).

Полнота реализации функций характеризует степень удовлетворения требований пользователя к функциям и определяется по двум оценочным показателям:

1) коэффициент полноты реализованных функций

$$K_n = F / P \ ,$$

где P – число функций, включенных в требования к ПС; $F = \sum_{i=1}^{p} (F_i)$ – общее число реализованных функций; $F_i = 1$, если функция реализована, 0 – в противном случае;

2) средний взвешенный показатель полноты реализованных функций

$$\Phi = \sum_{i=1}^{P} (W_i * F_i) \ ,$$

где W_i - коэффициент весомости функции F_i , определяется из условия $\sum_{i=1}^{P} W_i = 1$.

Показатель *защищенности* ($П_З$) информации определяется из соотношения $П_З = V_З/V_d$, где $V_З$ – объем данных, фактически защищенных; V_d – требующих защиты.

Показатели свойств функционирования. Основной показатель этого свойства – *корректность.* Он определяется степенью соответствия ПС требованиям пользователя, и им можно управлять в ходе разработки. Методами определения корректности (контроль проектных решений, тестирование) достигается снижение числа ошибок путем применения на операциях процессов формальных методов проектирования.

Оценка качества программ, созданных по ТЛ, связана с понятием корректности и надежности. Проводится эта оценка на основании собранной статистики по результатам тестирования и испытаний, накапливаемых в журнале регистрации

ошибок (таб. 9.7–9.9) с помощью моделей.

Международный стандарт ANSI/IEEE–729–83 разделяет ошибки в разработке программ на следующие.

Ошибка (error) – состояние программы, при котором выдается неправильные результаты, причиной которых являются изъяны (flaw) в операторах программы или в технологическом процессе ее разработки, что приводит к неправильной интерпретации исходной информации, следовательно и к неверному решению.

Дефект (fault) в программе является следствием ошибок разработчика на любом из этапов разработки и может содержаться в исходных или проектных спецификациях, текстах кодов программ, эксплуатационной документация и т.п. В процессе выполнения программы может быть обнаружен дефект или сбой.

Отказ (failure) – это отклонение программы от функционирования или невозможность программы выполнять функции, определенные требованиями и ограничениями и рассматривается как событие, способствующее переходу программы в неработоспособное состояние из–за ошибок, скрытых в ней дефектов или сбоев в среде функционирования.

Отказ может быть результатом следующих причин:

– ошибочная спецификация или пропущенное требование, что означает, что спецификация точно не отражает того, что предполагал пользователь;

– спецификация может содержать требование, которое невозможно выполнить на данной аппаратуре и программном обеспечении;

– проект программы может содержать ошибки (например, база данных спроектирована без защиты от несанкционированного доступа пользователя, а требуется защита);

– программа может быть неправильной, т.е. она выполняет несвойственный алгоритм или он сделан не полностью.

Иными словами, отказы, как правило, являются результатами одной или более ошибок в программе или наличия разного рода дефектов.

Данные типы ошибок влияют на оценку надежности ПС, которая характеризуется такими основными атрибутными свойствами: безотказность и восстанавливаемость. Первое свойство характеризуется тем, что программа работает без сбоев и отказов. В случае их возникновения собирается статистика о среднем времени наработки на отказ и интенсивности отказов. Эти данные – входные для оценки надежности по моделям надежности, приведенным в п.7.6.

Восстанавливаемость определяется средним временем обнаружения и устранения ошибки. В качестве числовой характеристики можно использовать коэффициент готовности

$$K_{\text{г}} = T_{\text{ср}}(T_{\text{ср}} + T_{\text{в}}),$$

где $T_{\text{ср}}$ – среднее время наработки на отказ; $T_{\text{В}} = T_{\text{о}} + T_{\text{л}}$ – среднее время восстановления; $T_{\text{о}}$ – среднее время обнаружения ошибки; $T_{\text{л}}$ – время локализации ошибки.

Показатели эргономичности характеризуют удобство эксплуатации и оцениваются в процессе эксплуатации. Учет таких показателей в ходе разработки заключается в проверке наличия средств поддержки переноса программы на другой компьютер или на другую платформу. Упрощенную формулу переноса программы дадим такую $\text{П} = \text{З}_{\text{а}} / \text{З}_{\text{п}}$,

где $\text{З}_{\text{а}}$– затраты на адаптацию в новой операционной среде; $\text{З}_{\text{п}}$ – затраты на

полную разработку в новой операционной среде.

Простота подготовки к работе определяется количеством технологических операций при подготовке ПС к работе и ее запуску.

Документируемость определяется полнотой документации (по составу), простотой изложения, структурностью информации. Оценивается экспертным путем специалистами из группы нормоконтроля. Оценка носит качественный характер и оформляется в виде протокола, специалистами, которые проводили экспертизу. К настоящему времени разработаны новые подходы и методы экспертного оценивания программных продуктов (например, в [141] разработана математическая теория оценивания процессов и программных продуктов).

Показатели свойств технологичности характеризуются рациональным выделением и использованием типовых, унифицированных компонентов, способствующих сокращению трудовых затрат на создание ПС по ТЛ.

Уровень унификации определяется коэффициентами применяемости *($К_{ПР}$)* и повторяемости ($К_{ПОВ}$), которые вычисляются по формулам

$$К_{ПР} = (N - N_0) * N,$$

$$К_{ПОВ} = N_1 / N,$$

где N– общее количество составных элементов в формируемом ПС; N_0– количество оригинальных элементов; N_1– количество повторно используемых элементов (модулей).

При управлении качеством на основе ТЛ для рассмотренных свойств показателей основными факторами, влияющими на качество на процессах проектирования, являются методики, инструкции и контроль хода разработки. Факторами, влияющими на эксплуатационные характеристики, являются: корректность постановки задач и исходных требований; полнота тестирования; полнота и точность отображения результатов в ПТД.

Контроль качества проектирования ПС по ТЛ. В процессе разработки контролю на полноту подвергаются функциональная ФА и системная СА архитектуры. Контроль является качественным и провидится экспертным путем на основании требований к функциям. Результат контроля функций фиксируется в карте, структура которой приведена в табл. 9.2.

Таблица 9.2

Показатели функций			
Номер	Наименование	Оценка реализации	Вес

Структурный контроль СА проводится экспертом для отметки состояний элементов разрабатываемого ПС на ТП и фиксируется в опросной карте контроля СА (табл. 9.3). Результаты оформляются протоколом, передаются в группу контроля и используются для количественных оценок.

Структурный контроль программных компонентов до начала процесса программирования предназначен для выявления и устранения ошибок в ПТД. Основными контролируемыми показателями являются: полнота СА; технологичность СА; структурная сложность компонентов и/или модулей; трудоемкость кодирования. Результат контроля компонентов фиксируется в сводной карте (табл. 9.4).

Таблица 9.3

Номер вопроса	Содержание	Оценка эксперта
1	Используются ли повторно используемые компоненты (их доля)?	
2	Выделены ли компоненты по функциональному принципу ?	
3	Независимы ли компоненты?	
4	Стандартизован ли интерфейс между компонентами?	
5	Предусматриваются ли средства восстановления при ошибках и какие: – использование контрольных точек, – контроль данных, – другие .	
6	Предусматриваются ли средства восстановления при разрушении и какие: – использование страховой копии, – восстановление с помощью транзакций, – другие.	
7	Предусмотрены ли средства защиты данных?	
8	Унифицированы ли основные компоненты ?	

Таблица 9.4.

Контроль компонента	Название
Количество проверенных модулей – обнаруженных ошибок, – модулей с отклонениями от технологии требуемых повторных просмотров Время контроля	

Карта контроля СА используется при оценке показателя технологичности;

$$T_{ca} = \sum_{i=1}^{N} (W_i \times P_i),$$

где P_i = 1, если ответ положительный, 0 – отрицательный; W_i – весовой коэффициент вопроса в карте; N – количество вопросов.

Базовое значение T_{ca} находится в интервале $0{,}8 < T_{ca} \leq 1$.

Показатель сложности разрабатываемого ПК определяется с помощью оценочных элементов; количества модулей, их длины, объема и структурной сложностью модуля.

Они определяются по ПТД (паспортов модулей и описаний в СЯВ или ЯПП) и используются при прогнозировании трудоемкости кодирования. При оценке длины, объема, трудоемкости и сложности используются метрики Холстеда [202], в которых длина словаря операндов (K_2) определяется по следующей формуле:

$$K_2 = A \times K_1 + B,$$
$$A = M_2 / (M_2 \times 2) \times \log 2 \times M_2/2,$$
$$B = M_2 - 2 \times A,$$

где K_1 - длина словаря операторов модуля. Нижняя длина модуля определяется по формуле:

$$N = K_1 \times log\ (K_1) + K_2 \times log\ (K_2),$$

а верхняя считается равной *2N*. Запишем объем модуля в битах:

$$V = N \times log\ 2\ (K_1 \times K_2)\ ,$$

Время на программирование модуля можно определить по формуле

$$T = E / (Q \times 60),$$

где *Q* характеризует квалификацию программиста, определяемую по формуле (9.8). Эти и другие сведения заносятся в карту модуля (табл. 9.5).

Таблица 9.5.

Карта модуля (имя)	
Характеристики модуля (имя)	**Обозначения**
Длина словаря операторов	K_1
Длина модуля	N
Объем модуля	V
Уровень квалификации	Q
Трудоемкость кодирования	E
Технологичность текста	T_{ca}

В технологичность текста входит качественная оценка структуры модуля, оформления текста в требованиях технологии, а также наличие переменных без индексов и изменение параметра цикла в теле цикла и др.

9.5. ТЕСТИРОВАНИЕ ПРОГРАММ И СБОР ДАННЫХ ОБ ОШИБКАХ

Процесс тестирования прикладных программ СОД по соответствующим ТЛ включает следующие основные операции: составление плана тестирования; управление подготовкой и выполнением тестов; анализ результатов тестирования и отладки; повторное тестирование.

В зависимости от того, кто проводит тестирование, рассматриваются три режима:

1. проведение всех видов операций тестирования специальной группой;
2. совместное проведение работ по тестированию разработчиками и группой тестирования, которая планирует, контролирует и принимает решение о доработках;
3. выполнение работ по тестированию программ на процессах ТЛ.

Первые два режима не всегда выполнимы, как правило, из-за недостатка ресурсов для образования группы тестирования. Поэтому в основном каждый коллектив разработчиков тестирование выполняет сам.

Таблица 9.6

Тест	**Таблица тестов**	**Лист**
Имя теста	Тестовая ситуация	Ожидаемая реакция

Таблица 9.7

G_1	**Журнал регистрации ошибок (статистика)**											Лист
Вид объекта												
Тестируемый элемент объекта	Количество обнаруженных ошибок	В том числе										
		Источник							Тип			
		1	2	3	4	5	6	7	8	9	10	11	12

Таблица 9.8

G_2	Журнал регистрации ошибок (время выполнения)									**Лист**
Дата	Начало сеанса, час., мин.	Конец сеанса час., мин.	Номер ошибки	Время выявления	Имя теста	Тип	Лист	Содержательное описание ошибки	Дата устранения ошибки	Место ошибки

Таблица 9.9

G_3	**Журнал регистрации ошибок (сообщение об ошибке)**			**Лист**
Дата обнаружения	ФИО обнаружившего	Дата устранения	Подпись устранившего	Примечание
Описание ситуации:				
Реакция на ситуацию :				
Предположение об источнике ошибки:				
Анализ ошибки и корректирующие действия:				
Место ошибки:		Тип:		Источники:
Описание ошибки:				
Описание исправлений:				
Описание проверочного теста:				

В рамках технологий ТЛ в целях соблюдения единства технологической дисциплины может быть использован набор карт для отображения в них заданий на тестирование отдельных модулей (табл. 9.6) или компонентов, собираемых из модулей, и для регистрации ошибочных ситуаций (табл. 9.7 – 9.9) в ходе тестирования.

Тестирование модулей и компонентов состоит в обеспечении на тестах следующих критериев: прохождение каждого оператора не менее одного раза; выполнение программы не менее одного раза на совокупности тестов, включающих наборы входных и выходных данных; тестирование функций хотя бы один раз; тестирование межмодульных интерфейсов. Тестирование программ по ТЛ: вначале выполняется автономное тестирование модулей нижнего уровня без вызова других модулей, затем выбирается очередной модуль, непосредственно вызывающий уже проверенные. Выполняется сборка модуля с вызываемыми, а затем тестирование в комплексе. Процесс повторяется до тех пор, пока не будет достигнуто корневым модулем. При нисходящем подходе вначале выполняется тестирование корневого модуля совместно с заглушками, замещающими непосредственно вызываемые им модули, затем после подсоединения (замещение заглушки) в процессе тестирования выбирается очередной модуль. Процесс продолжается до тех пор, пока не будут подсоединены и протестированы все

модули.

Затем проводится тестирование интерфейсов, т.е. проверка правильности вызовов модулей и конкретности передачи экспортируемых и импортируемых значений параметров. Тестирование интерфейсов осуществляется для модулей верхних уровней, вместо заглушек для них используются реальные вызываемые модули. Многие тесты, которые использовались при автономном тестировании, могут использоваться при тестировании интерфейсов. Новые тесты должны проверять различные ситуации передачи параметров для границ областей, имитировать ошибочные ситуации и т.д.

Тестирование монолитных (объединенных модулей) программ является более трудоемким. Оно требует разработки модулей, как драйверов, так и заглушек. Основным недостатком является то, что ошибки в интерфейсах между модулями обнаруживаются позднее, чем при пошаговом тестировании, по которому последствия более серьезны, а стоимость их устранения выше.

После отладки обнаруженных в ходе тестирования ошибок ПС должна пройти процесс повторного тестирования для обнаружения ошибок, появившихся после выполнения корректирующих воздействий.

Завершающими технологическими процессами тестирования объекта являются тестирование функций и комплексные испытания. Оба они преследуют цель определить соответствие полученной программы исходным требованиям и внешним спецификациям. Объектом при тестировании функций выступают отдельные функции или их совокупности, а при комплексных испытаниях– в соответствии с ТЗ.

Основными критериями завершенности тестирования являются: экономические; вероятностные и статистические; структурные (критерии полноты тестирования).

Выбор определенного критерия зависит от классов ПС, реализуемых на ТЛ, от методов тестирования, а также от стадии тестирования. Структурные критерии полноты тестирования наиболее распространены на практике и применяются на разных стадиях тестирования.

Заключение о полноте тестирования производится на основании плана тестирования, таблиц тестов и определяет численную оценку полноты тестирования по формуле

$$P_T = T_n / T,$$

где T – общее количество тестовых ситуаций; T_n – количество проверенных ситуаций, соответствующих ожидаемым.

При приемке отлаженных программных модулей, как правило, применяется статистический критерий, основанный на метриках Холстеда [202]: n – словарь, включающий K_1 – число операторов и. K_2– операндов; N – длина модуля; V – объем модуля (программы).

Число ошибок B, оставшихся после отладки в программе, предположительно можно вычислить по формуле

$$B = \frac{V}{3000} \text{ или } B = \frac{N \log_2 n}{3000}$$

Вероятностные критерии завершенности тестирования применяются при испытаниях ПС для определения надежности и устойчивости функционирования

ПС. Применение существующих моделей надежности позволит достаточно точно прогнозировать надежность ПС в эксплуатации.

Экономические критерии устанавливают соответствие между необходимыми затратами на тестирование и реальными сроками его проведения.

Методологическая поддержка технологии сборки. Эффективность применения ТЛ зависит от степени их инструктивности. Последние определяются ТД общего и частного характера, присущими конкретному процессу. В них определены регламентация и порядок применения инструментальных средств, форм представления результатов выполнения процессов и документирования продукта, а также управления работами по получению ПП высокого качества.

Документы поддерживают единый подход к созданию ПС по ТЛ с использованием стандартных форм документов для фиксации проектных решений на всех процессах разработки. Для них принят стандарт в соответствии с требованиями ГОСТ ЕСКД. Дадим им краткую характеристику применительно к СОД.

1. ТД «Язык проектирования» представляет методику применения данного языка (типа PDL) для описания структур программ, способов улучшения качества проектирования и единообразного представления проектных решений; составления документации на ПС; применения методов контроля разработки ПС и т. д.

2. ТД «Методика применения ТМ» определяет способы использования инструментальных средств: различных СУБД, программ ввода, контроля и обработки данных. Эти документы способствуют освоению и практическому использованию методов и средств выполнения соответствующих ТО, ТП и ТЛ.

ТД «Организация и контроль разработки ПС» определяет задачи и функции группы контроля качества ПС по проведению контроля хода разработки. Содержит описание видов и методов контроля, порядок формирования номенклатуры показателей качества в классе реализуемой ПрО и методические указания по систематической, экспертной или количественной оценке показателей качества, достигнутых в ходе разработки ПС на процессах ТЛ. Сюда входят документы планированию и проверки сроков разработки ПП в соответствии с планом-графиком работ. Описаны принципы организации коллектива разработчиков для создания ПС по ТЛ исходя из оргструктуры, заложенной в каждой ТЛ.

4. ТД «Методика тестирования ПС» содержит рекомендации по организации и проведению тестирования на разных процессах процесса разработки ПС (автономное и комплексное тестирование), определяет порядок составления и содержание плана тестирования, фиксации обнаруженных ошибок в журналах регистрации ошибок и методы оценки полноты тестирования по разным критериям. Методика нацелена на тщательное планирование процесса тестирования, проведения его оценки и сбора данных об обнаруженных ошибках.

1. ТД «Формы проектно-технологических документов» представлен набором модулей табличного вида. Дано их назначение и порядок заполнения при выполнении ТО. Приведены комплекты ПТД, рекомендуемые для разных видов объектов (компонент, модуль) и поддерживающие разные процессы (программирование, тестирование на основе таблиц тестов и сбор статистики об ошибках в журналах регистрации ошибок и т. д.).

2. ТД «Порядок описания прикладных модулей. Стандарт модулей» предлагает стандартную структуру прикладного модуля, удобную для разработки и составления документации. В классе ППО СОД определены типы модульных элементов:

функциональные заготовки, прикладные и системные модули, КПИ. Для них предложена рекомендуемая форма представления внешних спецификаций и текстов модулей.

3. ТД «Описание текста модуля» определяет оформление текста модуля в любом ЯП в унифицированном виде.

Такой набор технологических методик и инструкций может быть использован при регламентации работ и в другом классе систем. Они могут уточняться и дополняться исходя из условий разработки и используемых инструментальных средств.

9.6. АНАЛИЗ МОДЕЛЕЙ НАДЕЖНОСТИ ДЛЯ ОЦЕНКИ ПРОГРАММНЫХ СИСТЕМ

На оценку надежности ПС, заключающуюся в определении вероятности безотказной работы ПС в заданных условиях в течение предопределенного периода времени, существенное влияние оказывают два фактора:

программные методы и средства технологии программирования, применяемые на процессах разработки и способствующие достижению требуемой надежности;

тестирование и проверка функционирования созданного ПС со сбором данных о результатах обнаружения ошибок и интенсивности отказов в интервалах времени функционирования.

Методы оценки надежности базируются на аналогичных методах и в теории надежности технических средств и отличаются тем, что ошибки в ПС устраняются, улучшая их качество. Отказы в технических средствах, как правило, являются следствием износа и меньше всего зависят от ошибок проектирования (аппаратные средства логически проще, чем программные). Отказы при функционировании технических средств могут привести к непригодности или физическому износу этих средств.

Исследования программных средств на надежность [2, 6, 80, 81, 102, 104, 150, 152, 154, 162, 235, 237-245, 251 и др.] интенсивно проводились по основным направлениям создания трех типов программных систем.

К первому типу систем относятся программы решения инженерных и научно-технических задач. Они характеризуются неполным жизненным циклом, небольшим объемом и при эксплуатации не требуют всех ресурсов ЭВМ, носят эпизодический и кратковременный характер.

Второй тип представляет собой класс программных систем для информационно-справочных и автоматизированных систем обработки информации, функционирующих вне реального времени.

К третьему типу относятся программы и комплексы, входящие в контур управления и функционирующие в реальном времени, используя все ресурсы ЭВМ (память, быстродействие и др.).

В соответствии с классификацией Хетча [251] в классе этих типов систем сформировались три типа моделей надежности (рис. 9.6).

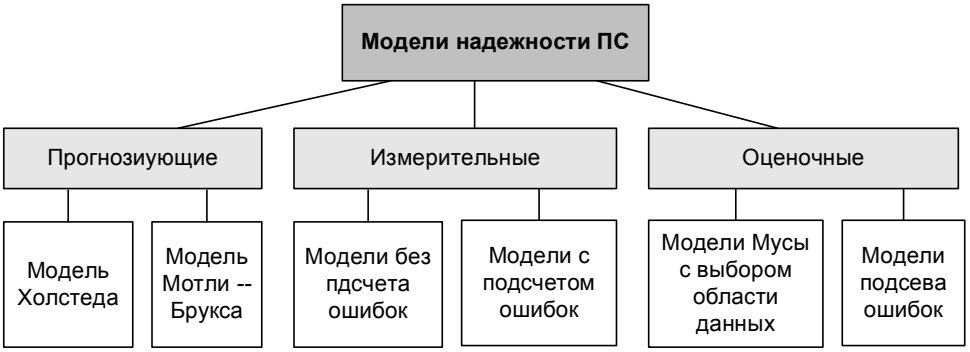


Рис. 9.6. Типы моделей надежности ПС

9.6.1. ТИПИЗАЦИЯ МОДЕЛЕЙ НАДЕЖНОСТИ ДЛЯ КЛАССА ПРОГРАММНЫХ СИСТЕМ

Прогнозирование надежности – это утверждение, основанное на количественных характеристиках создаваемых на ТП программ (длина программ, объем, число переменных и др.). Модель надежности Холстеда применяется для прогнозирования надежности на ранних процессах разработки (программирование и сборка) и рассмотрена ранее.

Оценочные модели надежности основаны на результатах тестирования, получаемых в тестовой среде при прогонах ПС на тестах и используемых при определении вероятности отказов ПС в реальной операционной среде функционирования. Данные модели исходят из частоты проявления и устранения причины, вызвавшей отказ ПС, поэтому их называют моделями интенсивности (сходство с моделями оценки надежности технических средств).

Измерительные модели предназначены для измерения надежности программного обеспечения, работающего с заданной внешней средой и имеющие следующие ограничения:

– программное обеспечение не модифицируется во время периода измерений свойств надежности;

– обнаруженные ошибки не исправляются;

– измерение надежности проводится для зафиксированной конфигурации программного обеспечения.

Типичным примером таких моделей является модель Нельсона и Рамамурти–Бастани и др.

Модель оценки надежности Нельсона основывается на выполнении k–прогонов программы при тестировании и позволяет определить надежность

$$R(k) = exp\,[-\sum t_j \lambda(t)\,],$$

где t_j – время выполнения j–прогона, $\lambda(t) = -[ln\,(1-q_i)\ \nabla j]$ и при $q_i \le 1$ она интерпретируется как интенсивность отказов.

В процессе испытаний программы на тестовых n_l прогонах оценка надежности вычисляется по формуле

$$R(l) = 1 - n_l / k,$$ где k – число прогонов программы.

Таким образом, данная модель рассматривает полученные количественные данные о проведенных прогонах.

Оценочные модели основываются на серии тестовых прогонов и проводятся на

процессах тестирования ПС. В тестовой среде определяется вероятность отказа программы при ее выполнении или тестировании.

Эти типы моделей могут применяться на процессах ЖЦ. Кроме того, результаты прогнозирующих моделей могут использоваться как входные данные для оценочной модели. Имеются модели (например, модель Мусы), которые можно рассматривать как оценочную и в тоже время, как измерительную модель [39, 233, 235].

Другой вид классификации моделей предложил Гоэл [32, 33, 219], согласно которой модели надежности базируются на отказах и разбиваются на четыре класса моделей:

– без подсчета ошибок,

– с подсчетом отказов,

– с подсевом ошибок,

– модели с выбором областей входных значений.

Модели без подсчета ошибок основаны на измерении интервала времени между отказами и позволяют спрогнозировать количество ошибок, оставшихся в программе. После каждого отказа оценивается надежность и определяется среднее время до следующего отказа. К такой модели относится модель Джелински и Моранды, Шика Вулвертона и Литвуда–Вералла [238, 240].

Модели с подсчетом отказов базируются на количестве ошибок, обнаруженных на заданных интервалах времени. Возникновение отказов в зависимости от времени является стохастическим процессом с непрерывной интенсивностью, а количество отказов является случайной величиной. Обнаруженные ошибки устраняются и поэтому количество ошибок в единицу времени уменьшается. К этому классу моделей относится модель Шумана, Шика–Вулвертона, Пуассоновская модель и др.[246, 249, 251].

Модели с подсевом ошибок основаны на количестве устраненных ошибок и подсеве внесенном в программу искусственных ошибок, тип и количество которых заранее известны. Затем строиться соотношение числа оставшихся прогнозируемых ошибок к числу искусственных ошибок, которое сравнивается с соотношением числа обнаруженных действительных ошибок к числу обнаруженных искусственных ошибок. Результат сравнения используется для оценки надежности и качества программы. При внесении изменений в программу проводится повторное тестирование и оценка надежности. Этот подход к организации тестирования отличается громоздкостью и редко используется из–за дополнительного объема работ, связанных с подбором, выполнением и устранением искусственных ошибок.

Модели с выбором области входных значений основываются на генерации множества тестовых выборок из входного распределения и оценка надежности, проводится по полученным отказам на основе тестовых выборок из входной области. К этому типу моделей относится модель Нельсона и др.

Таким образом, классификация моделей роста надежности относительно процесса выявления отказов, фактически разделена на две группы:

модели, рассматривающие количество отказов как Марковский процесс [24];

модели, которые рассматривают интенсивность отказов как Пуассоновский процесс.

Фактор распределения интенсивности отказов разделяет модели на

экспоненциальные, логарифмические, геометрические, байесовские и др.

Таким образом, классификация моделей роста надежности относительно процесса выявления отказов, фактически разделяет их на две группы:

– модели, которые рассматриваются с позиции марковского процесса;

– модели, в которых интенсивность появления отказов рассматривается с точки зрения пуассоновского процесса.

Фактор распределения интенсивности отказов разделяет модели на экспоненциальные, логарифмические, геометрические, байесовские и др.

Марковский процесс характеризуется дискретным временем и конечным множеством состояний. Временной параметр пробегает неотрицательные числовые значения, а процесс (цепочка) определяется набором вероятностей перехода $p_{ij}\ (n)$, т.е. вероятностью на n– шаге перейти из состояния i в состояние j. Процесс называется однородным, если он не зависит от n.

В моделях, базирующихся на процессе Маркова, предполагается, что количество дефектов, обнаруженных в ПС, в любой момент времени зависит от поведения системы и представляется в виде стационарной цепи Маркова [24, 133, 157]. При этом количество дефектов конечное, но является неизвестной величиной, которая задается для модели в виде константы. Интенсивность отказов в ПС или скорость прохода по цепи зависит *лишь от количества дефектов,* которые остались в ПС.

9.6.2. МОДЕЛИ НАДЕЖНОСТИ МАРКОВСКОГО ТИПА

К этой группе моделей относятся: Джелинського–Моранды, Шика–Вулвертона, Шантикумера [125, 133, 140].

Ниже рассматриваются некоторые модели надежности, которые обеспечивают рост надежности ПО (называются моделями роста надежности [162]), находят широкое применение на процессе тестирования и описывают процесс обнаружения отказов при следующих основных предположениях:

– все ошибки в ПС не зависят друг от друга с точки зрения локализации отказов;

– интенсивность отказов пропорциональна текущему числу ошибок в ПС (убывает при тестировании программного обеспечения);

– вероятность локализации отказов остается постоянной;

– локализованные ошибки устраняются до того, как тестирование будет продолжено;

– при устранении ошибок новые ошибки не вносятся.

Приведем основные обозначения используемых величин при описании моделей роста надежности:

m – число обнаруженных отказов ПО за время тестирования;

X_i – интервалы времени между отказами $i–1$ и i, при $i = 1,..., m$;

S_i – моменты времени отказов (длительность тестирования до i – отказа), $S_i = X_k$ при $i = 1,..., m$;

T – продолжительность тестирования программного обеспечения (время, для которого определяется надежность);

N – оценка числа ошибок в ПО в начале тестирования;

M – оценка числа прогнозированных ошибок;

MT – оценка среднего времени до следующего отказа;

$E\ (T_p)$ – оценка среднего времени до завершения тестирования;

$Var\ (T_p)$ – оценка дисперсии;

$R\ (t)$ – функция надежности ПО;

$Zi\ (t)$ – функция риска в момент времени t между i–1 и i отказами;

c – коэффициент пропорциональности;

b – частота обнаружения ошибок.

Далее рассматриваются несколько моделей роста надежности, основанные на этих предположениях и использовании результатов тестирования программ в части отказов, времени между ними и др.

Модель Джелински–Моранды. В этой модели используются исходные данные, приведенные выше, а также:

m – число обнаруженных отказов за время тестирования,

X_i – интервалы времени между отказами,

T – продолжительность тестирования.

Функция риска $Z_i\ (t)$ в момент времени t расположена между $i–1$ и i имеет вид:

$$Z_i\ (t) = c\ (N - n_{i-1})\ ,$$

где $i = 1\ ,..,\ m;\quad T_{i-1} < t < T_i$.

Эта функция считается ступенчатой кусочно–постоянной функцией с постоянным коэффициентом пропорциональности и величиной ступени – с.

Оценка параметров c и N производится с помощью системы уравнений:

$$\sum_{i=1}^{m} 1/(N - n_{i-1}) - \sum_{i=1}^{m} cX_i = 0$$

$$n/c - Nn/c - \sum_{i=1}^{m} NT - \sum_{i=1}^{m} X_i n_{i-1} = 0$$

При этом суммарное время тестирования вычисляется так: $T = \sum_{i=1}^{m} X_i$

Выходные показатели для оценки надежности относительно указанного времени T включают:

– число оставшихся ошибок $M_m = N–m;$

– среднее время до текущего отказа $MT_m = 1/(N–m)\ c$;

– среднее время до завершения тестирования и его дисперсия

$$E\ (T_p) = \sum_{i=1}^{N-n} (1/ic)\ ,$$

$$Var\ (T_p) = \sum_{i=1}^{N-n} 1/(ic)^2\ ,$$

При этом функция надежности вычисляется по формуле:

$$Rm\ (t) = exp\ (–\ (N–\ m)\ ct),$$

при $t>0$ и числе ошибок, найденных и исправленных на каждом интервале тестирования, равным единице.

Модель Шика–Волвертона. Модель используется тогда, когда интенсивность отказов пропорциональна не только текущему числу ошибок, но и времени, прошедшему с момента последнего отказа. Исходные данные для этой модели аналогичны выше рассмотренной модели Джелински–Моранды:

m – число обнаруженных отказов за время тестирования,

X_i – интервалы времени между отказами,

T – продолжительность тестирования.

Функции риска Z_i *(t)* в момент времени между *i–1* и *i–m* отказами определяются следующим образом:

$$Z_i\,(t) = c\,(N - n_{i-1}),$$

где $i = 1,..., m;\quad T_{i-1} < t < T_i\,,\quad T = \sum_{i=1}^{m} X_i\,.$

Эта функция является линейной внутри каждого интервала времени между отказами, возрастает с меньшим углом наклона.

Оценка с и N вычисляется из системы уравнений:

$$\sum_{i=1}^{m} 1/\,N - n_{i-1}\; - \sum_{i=1}^{m} X_i^{\,2}\,/\,2 \;= 0$$

$$n/c\; - \sum_{i=1}^{m} (N - n_{i-1}) X_i^{\,2}\,/\,2 \;= 0$$

К выходным показателям надежности относительно продолжительности T относятся:

– число оставшихся ошибок $M_m = N–m$;

– среднее время до следующего отказа $MT_m = (\pi\,/\,(2\,(\,N–m)\,c)\,)^{\,1/2}$;

– среднее время до завершения тестирования и его дисперсия

$$E\,(T_p) = \sum_{i=1}^{N-m} (\pi\,/(2ic)^{1/2}\,,$$

$$Var\,(T_p) = \sum_{i=1}^{N-m} ((2 - \pi\,/\,2)\,/\,ic)\,,$$

Функция надежности вычисляется по формуле:

$$R_T\,(\,t) = exp\,(–(N–m)\,ct^2\,/2)\,,\; t \geq 0.$$

9.6.3. МОДЕЛИ НАДЕЖНОСТИ ПУАССОНОВСКОГО ТИПА

Эти модели базируются на выявлении отказов и моделируется неоднородным процессом, который задает *{M(t), t≥0}* – неоднородный пуассоновский процесс с функцией интенсивности $\lambda(t)$, что соответствует общему количеству отказов ПС за время его использования t.

Модель Гоело–Окумото. В основе этой модели лежит описание процесса обнаружения ошибок с помощью неоднородного Пуассоновского процесса, ее можно рассматривать как модель экспоненциального роста. В этой модели интенсивность отказов также зависит от времени. Кроме того, в ней количество выявленных ошибок трактуется как случайная величина, значение которой зависит от теста и других условных факторов.

Исходные данные этой модели:

m – число обнаруженных отказов за время тестирования,

X_i – интервалы времени между отказами,

T – продолжительность тестирования.

Функция среднего числа отказов, обнаруженных к моменту t имеет вид:

$$m(t) = N(1 - e^{-bt}),$$

где b – интенсивность обнаружения отказов и показатель роста надежности $q(t) = b$.

Функция интенсивности $\lambda(t)$ в зависимости от времени работы до отказа равна

$$\lambda(t) = Nb^{-bt}, \quad t \geq 0.$$

Оценка b и N получаются из решения уравнений:

$$m/N - 1 + exp(-bT) = 0$$

$$m/b - \sum_{i-1}^{m} t_i - N_m \exp(-bT) = 0$$

Выходные показатели надежности относительно времени T определяют:

1) среднее число ошибок, которые были обнаружены в интервале [0, T]

$$E(N_T) = N\,exp(-bT),$$

2) функцию надежности

$$R_T(t) = exp(-N(e^{-bt} - e^{-bt(t+m)})), \quad t \geq 0.$$

В этой модели обнаружение ошибки, трактуется как случайная величина, значение которой зависит от теста и операционной среды. В других моделях количество обнаруженных ошибок рассматривается как константа.

В моделях роста надежности исходной информацией для расчета надежности являются интервалы времени между отказами тестируемой программы, число отказов и время, для которого определяется надежность программы при отказе. На основании этой информации по моделям определяются следующие показатели надежности:

– вероятность безотказной работы;

– среднее время до следующего отказа;

– число необнаруженных отказов (ошибок);

– среднее время для дополнительного тестирования программы.

Модель анализа результатов прогона тестов использует в своих расчетах общее число экспериментов тестирования и число отказов. Эта модель определяет только вероятность безотказной работы программы и выбрана для случаев, когда предыдущие модели нельзя использовать (мало данных, некорректность вычислений). Формула определения вероятности безотказной работы по числу проведенных экспериментов имеет вид: $P = 1 - Nex/N$,

где Nex – число ошибочных экспериментов, N – число проведенных экспериментов для проверки работы ПС.

Таким образом, можно сделать вывод о том, что модели надежности ПС основаны на времени функционирования и/или количестве отказов (ошибок), полученных в программах в процессе их тестирования или эксплуатации. Модели надежности учитывает случайный Марковский и пуассоновский характер соответственно процессов обнаружения ошибок в программах, а также характер и интенсивность отказов.

9.6.4. ПРАКТИКА ОЦЕНКИ НАДЕЖНОСТИ ПРОГРАММ

Оценочные модели охватывают более широкий класс программных систем и основаны на истории отказов, зафиксированной в картах регистрации ошибок (см.

табл. 9.7 - 9.9). В картах отмечаются интервалы времени между ошибками либо число ошибок на заданных интервалах.

В процессе эксплуатации оценочные методы надежности применительно к рассматриваемому классу ППО СОД обеспечивают прогнозирование поведения программ. Они используют результаты тестирования и историю отказов, фиксированных в журналах регистрации ошибок $G_1 \div G_3$. К основным используемым на ТЛ моделям оценки надежности ППО относятся модели Мусы и S-образная [102, 104, 125, 133, 140].

Оценочная модель Мусы основана на следующих положениях:

тексты адекватно представляют среду функционирования;

происходящие отказы учитываются (оценивается их количество);

интервалы между отказами независимы;

время между отказами распределено по экспоненциальному закону;

интенсивность отказов пропорциональна числу ошибок;

скорость исправления ошибок (относительно времени функционирования) пропорциональна интенсивности их появления.

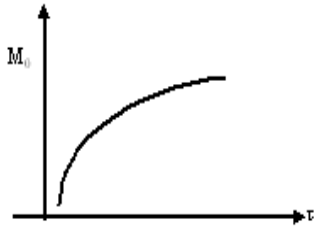
На основе этих ошибок устанавливаются:

1) зависимость среднего числа отказов от времени функционирования τ

$$m=M_0[1-\exp(\frac{c\tau}{M_0T_0})], \qquad (9.11)$$

где M_0– общее число ошибок; T_0– начальная наработка на отказ; c– коэффициент сжатия тестов (равен времени испытаний).

На графике зависимость выглядит следующим образом:



2) зависимость текущей средней наработки на отказ T от времени функционирования τ

$$T = T_0 \exp(\frac{c\tau}{M_0T_0}), \qquad (9.12)$$

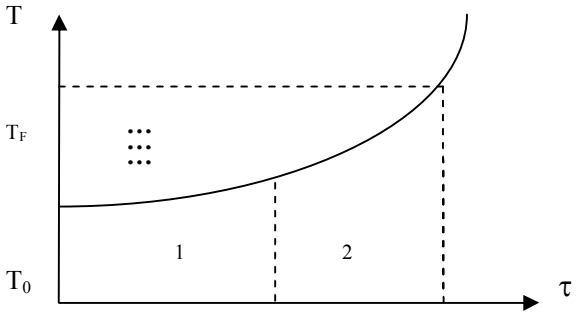
где M_0 , T_0 , c – те же величины, зависящие от наработки на отказ.

На графике эта зависимость выглядит так:

Из уравнений (9.11) и (9.12) следует

$$T = \frac{T_0M_0}{M_0 - m} T_0 \qquad (9.13)$$

График этой зависимости представлен областью 1, для которой M_1 = 1, 2, ...– номера наблюдений, а $\tau_1, \tau_2 \ldots \tau_{M1}$ - время между отказами. Область 2 соответствует достижению средней наработки T_p на отказ за время $\Delta\tau$.



Далее по собранным в карте G_2 (см. табл. 9.6) данным об ошибках оцениваются параметры T_0 и M_0. Дополнительное число ошибок, которые необходимо обнаружить, можно определить по формуле

$$\Delta m = M_0 T_0 \left[\frac{1}{T} - \frac{1}{T_0} \right].$$

В Байесовских моделях интенсивность отказов рассматривается как случайный процесс, зависящий от происшедших отказов, и является суммой случайных процессов:

$$\lambda_i = V_1 + V_2 + , \ldots , V_{N-i+1} \qquad (9.14)$$

где i– 1– количество, обнаруженных ошибок к текущему моменту; N– число ошибок.

Оценочная S-образная модель. В отличие от модели Мусы здесь зависимость числа отказов от времени функционирования задается в виде

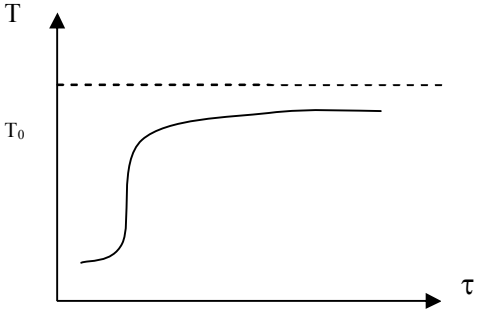
$$m = M_0 \left[1 - \left(1 + \frac{c\tau}{M_0 T_0}\right) \exp\left(-\frac{c\tau}{M_0 T_0}\right) \right]. \qquad (9.15)$$

Зависимость текущей средней наработки на отказ от времени функционирования определяется по формуле и имеет следующий вид на графике:

$$T = T_0 \exp\left(\frac{c\tau}{M_0 T_0}\right). \qquad (9.16)$$

Из соотношений (9.15) и (9.16) получаем

$$m = M_0 \left[1 - \left(1 - \ln\frac{T}{T_0}\right) \frac{T_0}{T} \right] \qquad (9.17)$$



На основании приведенных формул и собранных в G_2 данных результатов тестирования программного объекта можно получить оценку параметров M_0 и T_0.

Модель, основанная на природе ошибок. Описывается неоднородным Пуассоновским процессом

$$P_i = \{ M(t) - n \} = \frac{\{M(t)\}^n}{n!} \exp[-\mathrm{H(t)}], \quad t \geq 0, \tag{9.18}$$

где $M(t)$, $t \geq 0$– общее число ошибок, обнаруженных во временном интервале $[0, t]$; $H(t)$– функция математического ожидания вида

$$H(t) = m_p(t) = a \sum_{i=1}^{k} P_i [1 - \exp(-b_i t)] \ , \tag{9.19}$$

при условиях $a > 0,\ 0 < b_k < b_{k-1} < \ldots < b_1 < 1,$

$$\sum P_i = 1,\ 0 < P_i < 1,\ (i = \overline{1,k}),$$

где a– число ожидаемых ошибок, существующих в ПС на момент начала тестирования; b_i– интенсивность обнаружения ошибок типа i; k – количество типов ошибок; P_i– доля ошибок типа i.

Ожидаемое число необнаруженных ошибок r_p в момент времени t из формулы (9.19) будет

$$r_p(t) = a \sum_{i=1}^{k} p_i e^{-b_i t_i} \ . \tag{9.20}$$

Используя эту модель, можно по полученным фактическим данным об ошибках на процессе тестирования (типы ошибок и интенсивность их обнаружения) рассчитать ожидаемое число необнаруженных ошибок.

9.7. ПОДХОД К АВТОМАТИЗАЦИИ ОЦЕНКИ КАЧЕСТВА ПРОГРАММНЫХ СИСТЕМ

Исходя из рассмотренных методов оценки надежности ПС соответствующие программные средства представлены в виде отдельного технологического модуля – инструментального комплекса. Реализация модуля базируется на регистрации ошибок в процессе тестирования и сохранении истории отказов (времени и числа ошибок) в картах регистрации ошибок (см. табл. 9.6— 9.9). Эти данные являются исходными для работы ТМ [102, 104].

9.7.1. КОНЦЕПЦИЯ МОДЕЛИРОВАНИЯ НАДЕЖНОСТИ

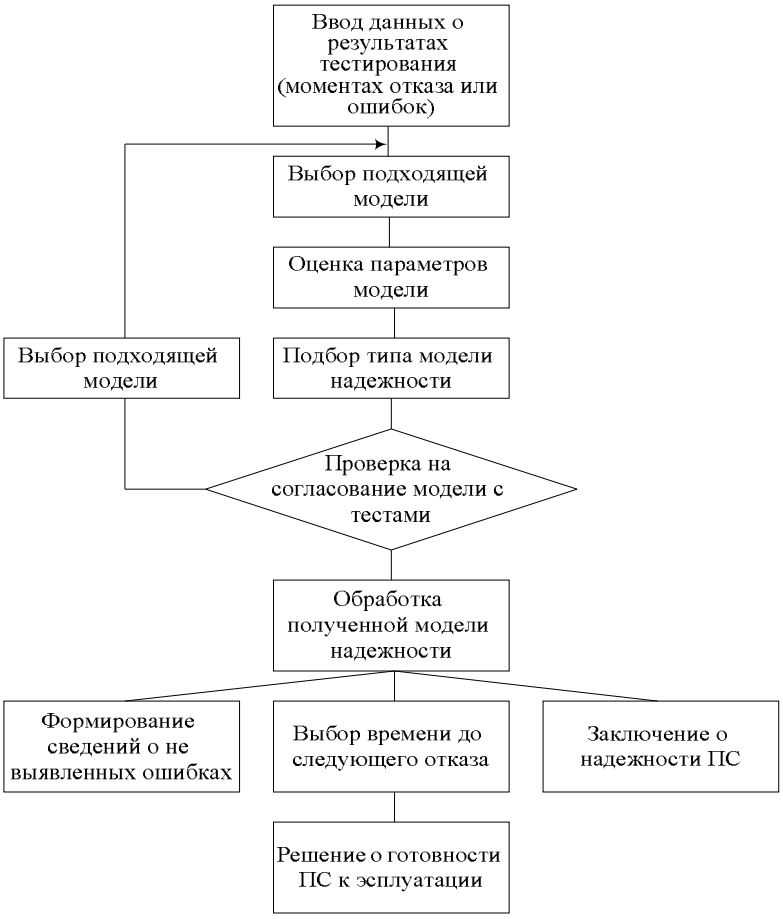
Процесс выбора модели и получения оценок измерений надежности по выбранной модели заключается в выполнении действий по определению типа модели: на основе моментов между отказами или на подсчете ошибок.

При выборе типа модели, основанной на моментах между отказами, используются предположения, характеризующие специфику модели: независимость моментов между отказами; вероятность появления каждой ошибки;

вложенные ошибки не зависят друг от друга; ошибки устраняются после появления; при коррекции не вносятся новые ошибки.

Модели, основанные на подсчете ошибок, имеют следующие предположения: интервалы тестирования не зависят один от другого; тестирование на интервалах однородно; количество выявленных ошибок на пересекающихся интервалах времени независимо.

Процесс выбора модели описывается следующими шагами по алгоритму представленному на рис.9.7.



Ошибка!

Рис. 9.7. Схема моделирования надежности ПС

Данная схема легла в основу создания ТМ оценки качества ПС, названного «Менеджер качества», описываемого ниже. В нем отражены модели надежности, технология проведения тестирования и всевозможных оценок по результатам тестирования отдельных компонентов и ПС в целом.

9.7.2. ТЕХНОЛОГИЧЕСКИЙ МОДУЛЬ ОЦЕНКИ КАЧЕСТВА

ТМ «Менеджер качества» реализован на основе многолетних научных исследований и разработок, выполненных в том числе и в рамках диссертационных работ [102, 104] и предназначен для оценки качества программ в классе СОД. Он состоит из

двух технологических модулей: ТМ «Тест инженер» и ТМ «Менеджер качества»

Оба технологических модуля разработаны по правилам ТПР ПС [120]. Они представлены совокупностью операций технологии, процессы которой объединены технологическим маршрутом операций (O_1, …, O_6) с входами, выходами, которые выполняются с помощью инструментальных и методических средств (рис. 9.8).

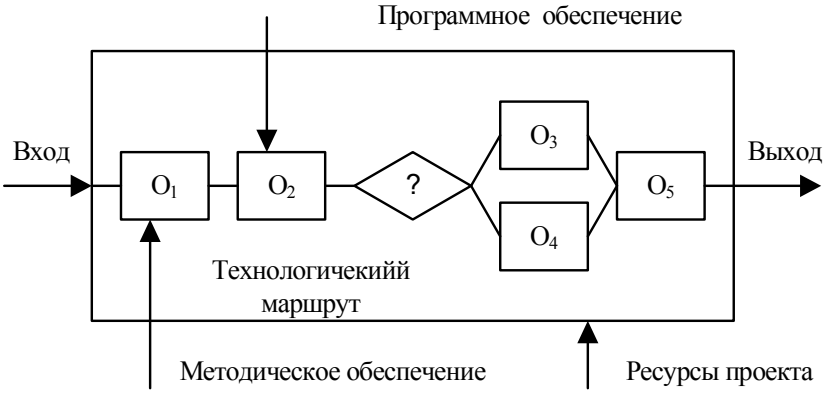


Рис.9.8. Структура ТМ

Рассмотрим назначение каждого их них.

ТМ «Тест инженер» предназначен для реализация базового процесса тестирования, отслеживания времени и частоты выполнения компонентов ПС во время тестирования, фиксации отказов и дефектов, оценки риска отказов модулей и определения оптимального времени их тестирования. В его состав входят компоненты:

– оценки трудозатрат по модели COCOMO II [30] и размера ПС по методологии FPA (Function Points Analysis) [248, 257];

– оценки времени, стоимости тестирования и риска отказов отдельных модулей;

– анализа данных об отказах и дефектах ПС, ведения базы данных об отказах, размере и плотности дефектов в модулях;

– ведения нормативных и классификационных данных (стоимостных и временных характеристик, состав группы тестирования, серьезности дефектов;

– оценки риска отказов модулей и определения оптимального времени тестирования модулей;

– предоставление отчетов по результатам тестирования;

Эти компоненты распределены по двум функциональным модулям (ФМ): ФМ «Тестирования» и ФМ «Оценка» в среде Oracle и MS Excel.

ФМ «Тестирования» реализует базовый процесс тестирования, регестрирует время (t_e) выполнения каждого модуля посредством использования счетчиков времени, встроенных в модули, рассчитывает общее время, частоту использования компонентов, а также дефектов отказов.

ФМ «Оценка» предназначен для оценки результатов тестирования (риска отказов модулей) в среде VB (Visual Basic) и Oracle с участием руководителя группы тестирования. Определение параметров моделей надежности пуассоновского типа проводится методом максимального правдоподобия. По этим параметрам оцениваются показатели надежности, риск отказов модулей и интенсивность отказов. Расчет параметров выполняются для одной избранной

модели (среди выше приведенных моделей надежности) или сразу для нескольких моделей. По каждой из моделей надежности рассчитываются значения оптимального времени тестирования модулей ПС.

Испытания данного ТМ выполнялись в системе информационно-аналитической поддержки принятия управленческих решений, которая включала семь функциональных модулей, объединенных деловым процессом обработки данных, которые функционировали в распределенной среде Oracle RunTime, Oracle Express и MS Office (Word, Excel).

Для регистрации времени выполнения и моментов отказов в модулях (М1-М5) на время тестирования были встроены фрагменты кода, которые определяли момент начала и завершения (нормального или аварийного) работы модуля. Ожидаемое время использования каждого модуля t_0 при эксплуатации ПС и их взносы $C_м$ определялись с учетом таких факторов: регламента представления документов (недельные, месячному и т.п.); количества пользователей; частоты обращения к модулям в каждом сеансе работы.

Стоимость тестирования и устранение отказов рассчитывалось с учетом таких факторов: стоимости времени работы специалистов; реального времени выполнения каждого модуля во время тестирования и времени, израсходованного на устранение дефектов в заданных модулях.

Испытания метода оценивания риска и оптимального времени тестирования модулей позволило:

– интегрировать базовый процесс тестирования в ЖЦ проекта системы;

– упорядочить деятельность членов группы тестирования с помощью шаблонов форм документов и инструментальной поддержки базового процесса тестирования;

– улучшить планирование и управления процессом тестирования;

– собрать данные и минимизировать количество дефектов в ПС и по ним провести оценку надежности.

ТМ «Менеджер качества» обеспечивает: прогнозирование и распределение надежности по отдельным компонентам; проведение процесса верификации или тестирования с анализом влияния дефектов на надежность; размещение полученных данных о дефектах в специальной базе данных; оценка плотности дефектов с применением одной их приведенных выше моделей надежности и представление результатов экспериментальной оценки надежности графически.

В среду ТМ входят:

– HUGIN Lite 6.5 API для реализации графической модели прогнозирования дефектов в ПС;

– MATLAB 6 для решения задач матричной алгебры, а также нелинейной оптимизации при реализации модели распределения надежности по модулям ПС;

– Oracle – ADO и OLE Object для организации обмена данными в среде БД Oracle (таблицы Excel)) и верификации в среде Oracle Designer'2000;

Информационная среда ТМ – это таблицы БД Oracle, в которых накапливаются общие данные (дефекты, отказы, ошибки) в оцениваемой ПС. При этом система Excel предоставляет шаблоны форм, в которых собираются данные о дефектах, которые классифицируются и проводится парное сравнение функций модулей в целях определения вероятности появления дефектов в модулях ПС, интерпретации распределения надежности и спрогнозированных значений надежности.

Данный ТМ состоит из ряда программно-технологических модулей (ПТМ).

Дадим им краткую характеристику.

1. ПТМ «Распределения надежности» реализует метод распределения надежности по компонентам ПС путем парного сравнения модулей и компонентов уровней иерархической структуры ПС и отображения их с помощью шкалы сравнения. По результатам парного сравнения строится квадратная матрица А размером n× n из элементов вида

$$a_{11} = a_{22} = ... = a_{nn} = 1, \quad a_{ij} = \frac{1}{a_{ji}} \ , \ u, j = 1,...,n; \ i \neq j; \ n=k, l, m,$$

где n – количество сравниваемых компонентов, k, l, m – количество функций и модулей соответственно. Матрица включает относительный вес i-го компонента и вычисляется по формуле

$$w_i = \frac{\sum_{j=1}^{n} a_{ij}}{\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}}, \quad \sum_{i=1}^{n} w_i = 1.$$

В случае больших размеров матрицы в целях получения более точных относительных оценок компонентов иерархии, вычисляются, так называемые, собственный вектор и собственные значения матрицы согласно известным уравнениям [25]. В них используются следующие данные: λ_{max} – максимальное собственное значение матрицы A n–порядка, w_i – коэффициент относительного веса элементов матрицы A, W = ($w_1, w_2, ... w_n$) – собственный вектор, которому соответствует λ_{max}.

Общность решения задачи сравнения устанавливается соотношением $\alpha = \sum_{i=1}^{n} w_i$ и значением $\sum_{i=1}^{n} w_i = 1$. Если матрица A имеет $n-1$ собственных значений λ, равных нулю и $\lambda_{max}= n$, то она является согласованной.

Оценки уровня согласованности суждений специалистов согласно многочисленным экспериментальным данным [25] проводится определение индекса согласованности CI и коэффициента согласованности CR по формулам

$$CI = \frac{\lambda_{\max} - n}{n-1}, \quad CR = \frac{CI}{E(CI)},$$

где $E(CI)$ – математическое ожидание для матрицы парных сравнений $A\ (n \times n)$.

Критерий приемлемости парного сравнения элементов в матрицах размером $n \geq 3$ получен такой: $CR \leq 0.05$ и $CR < 0.1$ для $n>5$.

По результатам сравнения формируется квадратная матрица $F(k \times k)$.

Аналогично проводится сравнение приложений ПС. В результате сравнения получают k матриц. Возможный порядок каждой матрицы – l, а максимальный порядок каждой из них – m.

Инструмент для поддержки метода сравнения – ExpertChoice, который для входной матрицы A автоматически получает собственный вектор W, собственное значение λ_{max} и коэффициент согласованности CR. Для вычисления λ_{max} и W используются соответствующие функции пакета MATLAB [1].

Результаты сравнений заносятся в форму, содержащую перечень весовых

коэффициентов программ, критерии, индексы и коэффициенты согласованности. Они предоставляются пользователю в виде готовых результатов обработки матриц. Полученные весовые коэффициенты синтезируются с помощью пакета MATLAB 6.5. Результаты отображаются в виде отчета о распределении надежности по объектам ПС.

2. ПТМ «Прогнозирование надежности» реализует метод прогнозирования распределенного количественного значения надежности по каждому модулю ПС согласно следующей модели надежности:

$$R_i = \exp[-D_i I_i \cdot (1 - \exp(\frac{\rho_i \cdot K}{I_i \cdot \varphi_i} \cdot t))],$$

где ρ_i – параметр среды эксплуатации i-го модуля, φ_i – характеристика среды ее разработки, I_i – оцененный размер начального кода, а D_i – прогнозируемая плотность дефектов в ПС. Коэффициент дефектов K – константа, предвиденная для всех объектов ПС, а значения ρ_i и φ_i – известны на момент первоначального прогнозирования надежности, они не изменяются во время разработки компонентов ПС.

3. ПТМ «Прогнозирования плотности дефектов» реализует набор моделей надежности для данного класса программ СОД. Прогнозирование надежности состоит в оценке влияния на плотность дефектов ряда факторов. Графическая модель отображает вероятностные значения полученных переменных в ее вершинах. Значения всех параметров модели прогнозирования надежности сохраняются в БД.

4. ПТМ «Оценка надежности ПС» построен согласно классификации дефектов (Orthogonal Defects classification), в соответствии с которой для каждого выявленного дефекта определяются параметры: тип дефекта, триггер дефекта, влияние дефекта. Эти параметры используется одной или двумя подходящими моделями надежности из выше приведенного в целях проведения оценки прогнозного значения надежности отдельных модулей и системы в целом. Результаты оценки сравниваются и выбирается из них наиболее правдоподобная.

На основе полученных данных о надежности и других показателях качества (функциональность, надежность, удобство применения и др.) рассчитывается *комплексный показатель качества*, который характеризует эксплуатационное качество ПС, как меру качества программного продукта.

Вывод. Рассмотрены вопросы управления разработкой ПП по ТЛ, к которым относятся: принцип производственной организации, методы поддержки технологии, модель планирования разработки ПС, определение трудозатрат, сложности и оценка показателей качества готового программного продукта для выбранного класса ППО СОД. Приведены модели надежности, дана их классификация, определены оценочные модели и предложена схема технологического модуля оценки качества программных продуктов. Аспекты тестирования, сбора данных на процессах ТЛ и технологический модуль оценки качества образуют необходимые инструменты индустриального производства программных продуктов.

Глава 10

ОСНОВНЫЕ ПОЛОЖЕНИЯ И ДИСЦИПЛИНЫ СБОРОЧНОГО ПРОИЗВОДСТВА ПРОГРАММНЫХ ПРОДУКТОВ

Основная цель *программной инженерия*, появившейся в 1968г. – обеспечить производство ПП на инженерной основе при достижении их качества и продуктивности. Программная инженерия – это раздел компьютерной науки (Computer Science), в задачу которой входит: изучение методов и средств построения компьютерных программ; отображение закономерностей развития программирования; обобщение накопленного опыта программирования простых и сложных объектов (ПО, ПС, приложений, семейств систем, программных проектов и т.п.); систематизация видов деятельности разных специалистов, участвующих в процессах построения сложных систем из более простых готовых ресурсов (модулей, КПИ, сервисов, агентов и др.).

По существу ПИ развивает программирование в направлении формирования основ инженерной деятельности коллективной разработки сложных программных систем, выполняемой с помощью ЖЦ, методов планирования, управления работами разных специалистов (программистов, верификаторов, тестировщиков, контролеров и др.), оценки их трудозатрат, стоимости и качества изготовления конечного программного продукта. В ПИ определен порядок решения основных задач построения ПП: формулировка требований, разработка, тестирование и сопровождение построенного продукта, а также проверка правильности и адекватности реализации проекта согласно требованиям и срокам.

При систематизации сорокалетнего периода развития программирования в направлении обозначения основ индустриального производства ПП, нами были определены новые дисциплины, которые соответствуют основным видам деятельности коллектива программистов, занимающихся изготовлением ПП (рис.10.1). Любое производство промышленных изделий (например, автомобильная, авиационная и др.) проходит цикл работ, включающий научное обоснование задач производства, их представление технологическими процессами, средствами их автоматизированного выполнения и сборки из готовых деталей отдельных частей. Работы на процессах планируются, управляются и контролируются относительно качества изготовления, как отдельных элементов изделия, так и конечного продукта в целом.

Производство программных продуктов по своей сути имеет специфический цикл работ, включающий в себя научную, инженерную, управленческую и экономическую деятельности, обеспеченную соответствующей теорией, практикой и инструментальными средствами.

Рис.10.1. Дисциплины программной инженерии

Иными словами, в рамках программной инженерии сформировались базовые основы производства ПП. Основные из них и наиболее усовершенствованные, проверенные на практике следующие:

1) стандарт жизненного цикла (ISO/IEC 12207: 2006) и модели ЖЦ;

2) технологические линии, Product lines, обеспечивающие изготовление ПП из готовых ресурсов (КПИ, модулей, сервисов и т.п.);

3) система управления качеством выпускаемой продукции, основанная на модели качества ПП, методах измерения и оценивания процессов ЖЦ и ПП;

4) менеджмент программных проектов;

5) система программно-технических методов и средств создания ПП в ядре знаний SWEBOK;

6) автоматизированные инструментальные средства, среды и методологии промышленного производства ПП (Microsoft Visual Studio Teams Systems, MSF, IBM Rational Rose, COM, CORBA и др.);

7) дисциплины ПИ поддержки всех видов деятельности при производстве ПП.

Среди названных основ производства (1-7), вторая основа изложена в предыдущих главах работы, а содержательная характеристика и обоснование важности остальных базовых основ индустрии ПП, как системы дисциплин организации производства приведены ниже.

10.1. РОЛЬ СТАНДАРТОВ И МОДЕЛЕЙ ЖИЗНЕННОГО ЦИКЛА (ЖЦ) ПРИ ИЗГОТОВЛЕНИИ ПРОГРАММНЫХ ПРОДУКТОВ

Каждая ПС на протяжении своего существования проходит определенную последовательность периодов «жизни» от замысла до воплощения в код программы, эксплуатации и изъятия. Такую последовательность действий называют *жизненным циклом*. Первоначально эти схемы действий были представлены в моделях жизненного цикла, а затем был создан стандарт ISO/IEC 12207 (версии 1992, 1996, 2006). Процессы ЖЦ задают определенную совокупность действий и задач реализации отдельных аспектов создания программного продукта [133, 140].

Промежуточные продукты процессов ЖЦ представляют собой определенные описания – тексты требований к разработке, спецификации, тексты программ, инструкции по эксплуатации и т.п.

10.1.1. СТАНДАРТНАЯ РЕГЛАМЕНТАЦИЯ ПРОЦЕССОВ ЖЦ

Разновидности действий, составляющих процессы ЖЦ ПС, зафиксированы в международном стандарте ISO/IEC 12207 (табл. 10.1).

Согласно этому стандарту все процессы регламентируют деятельность разных специалистов в производстве ПП и соответственно разделены на три категории:

– главные процессы;
– вспомогательные процессы;
– организационные процессы.

Главные процессы регламентируют деятельность покупателей, поставщиков и разработчиков ПП:

– приобретение (acquisition),
– поставка (supply),
– разработка (development),
– эксплуатация (operation),
– сопровождение (maintenance).

Процесс приобретения определяет действия организации покупателя (или заказчика), которая приобретает автоматизированную систему, программный продукт или сервис.

Процесс поставки регламентирует действия предприятия поставщика ПП, которое снабжает покупателя программным продуктом или сервисом.

Процесс разработки задает действия предприятия–разработчика по изготовлению ПП на процессах ЖЦ: выявление и анализ требований, проектирование, кодирование, тестирование, интеграция, эксплуатация и сопровождение.

Процесс сопровождения определяет действия организации, выполняющей сопровождение программного продукта (управление и поддержку изменений ПП).

Таблица 10.1

Номер п/п	Процесс (подпроцесс)
1. Категория “Основные процессы”	
1.1	Заказ (договор)
1.1.1	Подготовка заказа, выбор поставщика
1.1.2	Мониторинг деятельности поставщика, прием потребителем
1.2	Поставка (приобретение)
1.3	Разработка
1.3.1	Выявление требований
1.3.2	Анализ требований к системе
1.3.3	Проектирование архитектуры системы
1.3.4	Анализ требований к ПО системы
1.3.5	Проектирование ПО
1.3.6	Конструирование (кодирование) ПО
1.3.7	Интеграция ПО
1.3.8	Тестирование ПО
1.3.9	Системная интеграция
1.3.10	Системное тестирование
1.3.11	Инсталляция ПО
1.4	Эксплуатация
1.4.1	Функциональное использование
1.4.2	Поддержка потребителя
1.5	Сопровождение
2. Категория “Процессы поддержки”	
2.1	Документирование
2.2	Управление конфигурацией
2.3	Обеспечение гарантии качества
2.4	Верификация
2.5	Валидация
2.6	Общий просмотр
2.7	Аудит
2.8	Решение проблем
2.9	Обеспечение применяемости продукта
2.10	Оценивание продукта
3. Категория “Организационные процессы”	
3.1	Категория управления
3.1.1	Управление на уровне организации
3.1.2	Управление проектом
3.1.3	Управление качеством
3.1.4	Управление риском
3.1.5	Организационное обеспечение
3.1.6	Измерение
3.1.7	Управления знаниями

3.2	Усовершенствование	
	3.2.1	Внедрение процессов
	3.2.2	Оценивание процессов
	3.2.3	Усовершенствование процессов

Процесс эксплуатации регламентирует действия предприятия-оператора, обслуживающего ПП в процессе эксплуатации пользователями (консультации, изучение их потребностей и т.д.).

Процессы поддержки определяют действия, направленные на получение правильного и качественного ПП, включая документацию и действующую версию (конфигурацию) системы:

– документирования (documentation),
– управления конфигурацией (configuration management),
– обеспечения качества (quality assurance),
– верификации (verification),
– валидации (validation),
– совместного анализа (оценки) (joint review),
– аудита (audit),
– решения проблем (problem resolution).

Эти процессы выполняются верификаторами, валидаторами и другими контроллерами параллельно с основными процессами. Главная цель – достижение необходимого качества программной продукции.

Организационные процессы призваны управлять инфраструктурой разработки ПП, усовершенствовать процессы, продукты и подготавливать будущего пользователя к применению ПП*:*

– управления (management),
– создания инфраструктуры (infrastructure),
– усовершенствования (improvement),
– обучения (training).

Для каждого процесса стандарта определены виды деятельности и задачи, которые в него входят, совокупность результатов деятельности, а также некоторые специфические требования.

За выполнение каждого процесса стандарта должен отвечать определенный участник разработки или руководитель, руководствуясь заданной методологией изготовления ПП. Данный стандарт не задает способы и методы выполнения действий или задач, входящих в процесс. Любой стандарт формулирует определенные законные действия и не дает рекомендаций относительно их выполнения и содержания документов, выпускаемых на разных процессах.

Процессы, действия и задачи приведены в стандарте в наиболее естественной последовательности. Это не означает, что в такой же последовательности они должны быть применены в конкретной модели ЖЦ. В зависимости от проекта процессы, действия и задачи стандарта выбираются, упорядочиваются и включаются в модель ЖЦ. При применении они могут перекрывать, прерывать друг друга, выполняться итерационно или рекурсивно.

Поэтому организации, которая намерена применить этот стандарт в работе по изготовлению ПП, могут понадобиться дополнительные стандарты или процедуры, детализирующие отдельные детали выбранных элементов ЖЦ.

Данный стандарт имеет первостепенное значение для принятия ряда других связанных с ним стандартов, таких как стандарты по управлению ПО, обеспечению качества, верификации и валидации, управлению конфигурацией, измерению и оцениванию ПС и т.д.

Организация-разработчик программного проекта может выбрать из данного стандарта те процессы, которые более всего подходят для реализации программной системы ПС, и создать ведомственный стандарт предприятия. Обязательными среди процессов являются базовые процессы, которые в основном присутствуют во всех известных фундаментальных моделях ЖЦ. В зависимости от целей и задач предметной области в состав ведомственного стандарта могут войти отдельные процессы поддержки и организационные процессы стандарта. Руководитель проекта принимает решение о включении в новую создаваемую модель ЖЦ процесса обеспечения качества компонентов и системы управления обеспечением правильности продукта, а также соответствия (валидация) его заданным требованиям.

Процессы, включенные в модель ЖЦ предприятия, должны обеспечивать реализацию основных задач процессов ЖЦ и при необходимости привлекать процессы решения специальных задач (например, защита данных).

Процессы модели ЖЦ предприятия могут выполняться одним или несколькими разработчиками, при этом один из них должен быть ответственным за процесс или за все процессы данной модели.

Создаваемая ведомственная модель ЖЦ должна утверждаться и подкрепляться конкретными методиками, другими стандартами ПИ. Методика обеспечивается выбранными средствами, методами программирования и службами управления качеством.

10.1.2. ФУНДАМЕНТАЛЬНЫЕ МОДЕЛИ ЖЦ

В практике программирования сформировались фундаментальные модели ЖЦ, которые получили практическое применение как порядок разработки некоторых видов ПС. По своему содержанию они близки стандартному ЖЦ. Такими модели являются: каскадная, спиральная, эволюционная и др.

Каскадная модель – это модель, в которой каждая работа выполняется последовательно в указанном порядке (рис.10.2).

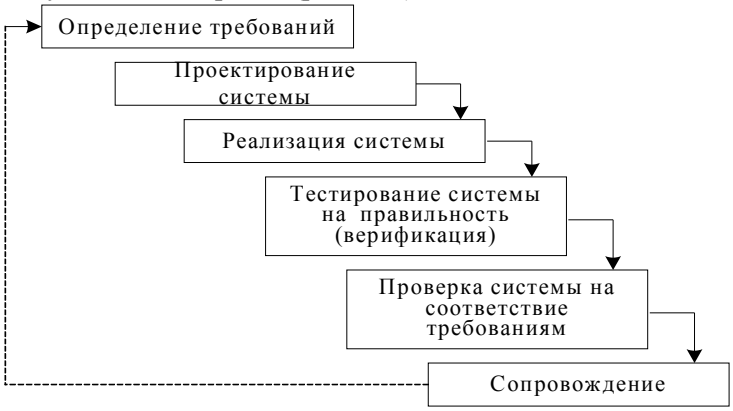


Рис. 10.2. Каскадная модель ЖЦ ПС

Каждая работа на процессе должна выполняться тщательно и правильно, поскольку после окончания работы в одном процессе осуществляется переход к следующему, не возвращаясь к предыдущему. Вспомогательные и организационные процессы (контроль требований, управления качеством и др.), как правило, выполняются с процессами разработки ПО. Результат процесса разработки верифицируется и подается на вход следующего процесса. Возврат к начальному процессу предполагается после сопровождения и исправления ошибок.

Основу данной модели составляет модель фабрики программ, в которой продукт проходит стадии от замысла к производству и поступлению к заказчику в виде готового изделия. В нем замена не предусмотрена. Каскадную модель можно рассматривать как модель ЖЦ, пригодную для создания первой версии ПО в целях проверки реализованных в ней функций. Обнаруженные ошибки при сопровождении и эксплуатации устраняются путем возврата к самому начальному процессу.

Спиральная модель. Она аналогична каскадной, с той разницей, что на каждом витке разработки, промежуточный продукт проверяется, изменяется и может возвращаться в предыдущий процесс (рис.10.3.).

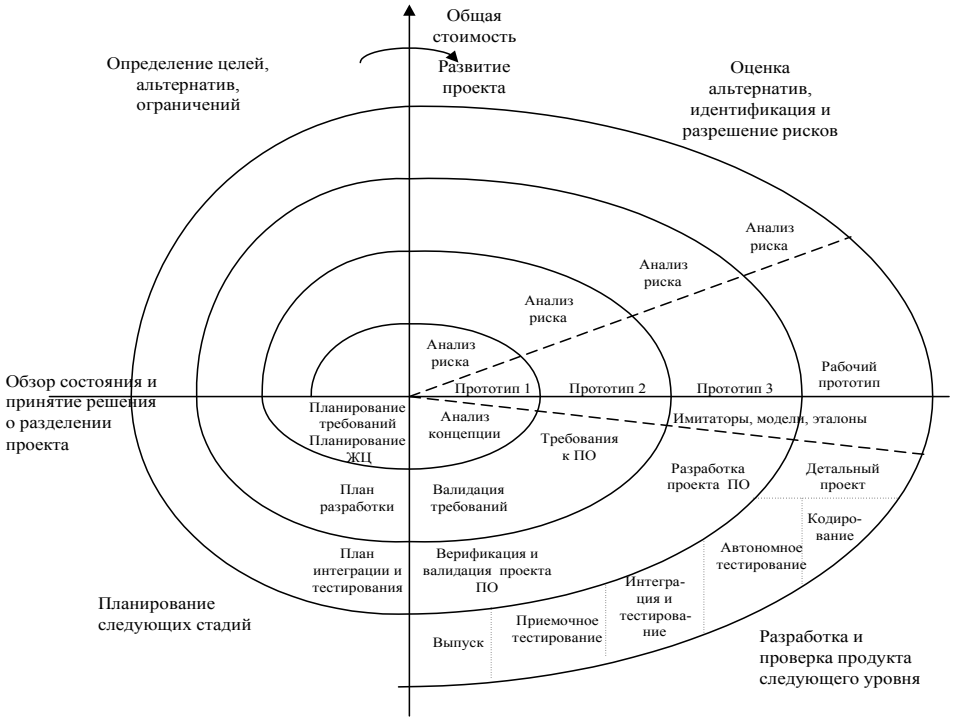


Рис. 10.3. Спиральная модель ЖЦ разработки ПО

На каждом витке спирали (стадии) выполняются следующие действия:

1. Определение целей стадии и рассмотрение альтернативных решений для достижения этих целей.

2. Оценка решений, идентификация рисков завершения данной стадии и принятие решения о продолжении или завершении стадии.

3. Разработка рабочего ПП стадии и составление плана для следующей стадии.

4. Последний виток спирали может выполняться по типу каскадной модели.

Спиральная модель применяется для сложных проектов или в тех случаях, когда проблемы проекта недостаточно понятны.

Эволюционная модель. Система последовательно разрабатывается из блоковых конструкций и для нее устанавливаются требования, которые уточняются в каждом следующем промежуточном блоке системы. Согласно данной модели изготовление системы начинается с исследования предметной области и анализа потребностей заказчика относительно возможности реализации функций системы (рис.10.4).

Модель применяется при разработке несложных и некритических систем. Цель разработки – получение некоторого прототипа системы итерационным и эволюционным путем. На нем проверяется реализация требований и функций системы. Эту модель часто называют еще моделью быстрой разработки программ RAD (Rapid Application Development).

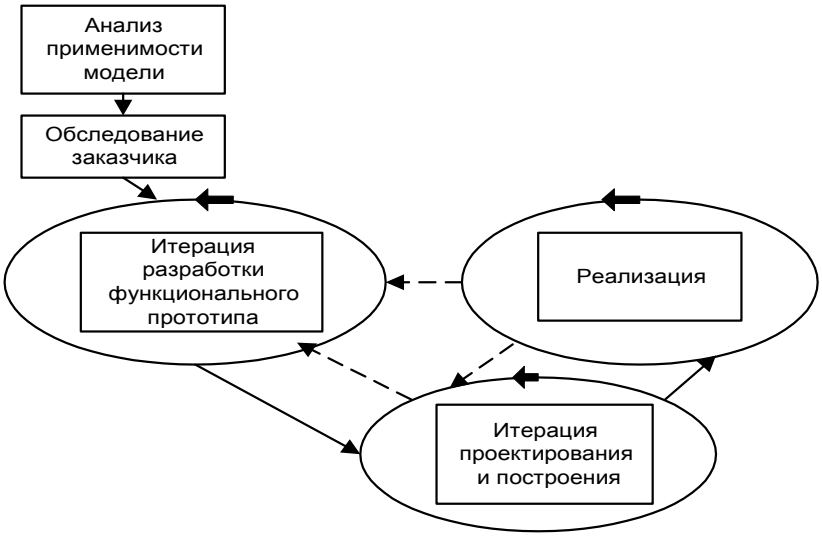


Рис.10.4. Модель эволюционного прототипа

В ней выполняются две главные итерации – формирование функционального прототипа и проверка реализации функциональных и нефункциональных требования. Основной идеей такой модели является моделирование отдельных функций системы в прототипе и постепенная эволюционная доработка перед выполнением всех заданных функциональных требований. Поскольку промежуточные прототипы системы соответствуют реализации некоторых функциональных требований, их можно проверять и во время сопровождения и во время эксплуатации, одновременно с процессом разработки очередных прототипов системы.

Суть итерационной эволюции – приближение к заданным функциям путем их наращивания. Последняя итерация дает окончательный прототип, соответствующий целям системы.

Таким образом, приведенные модели ЖЦ детализируют разные процессы ЖЦ с учетом структуры и сложности системы, и они могут использоваться при изготовлении разных ПП.

10.2. СИСТЕМА ЗНАНИЙ ЯДРА SWEBOK ДЛЯ ОРГАНИЗАЦИИ ИЗГОТОВЛЕНИЯ ПС

Специальный международный комитет, организованный ACM (Association for Computing Machinery) и компьютерным союзом института инженеров по электронике и электротехнике (IEEE Computer Society), проведя систематизацию знаний в области программирования и ПИ, предложил (2001, 2004г.) ядро знаний SWEBOK (Software Engineering Body Knowledge) ПИ, в которое вошло 10 разделов (areas knowledge) и было сформулировано следующее определение ПИ (www.swebok.com).

Определение. Программная инженерия – это система методов, способов и дисциплины планирования, разработки, эксплуатации и сопровождения программного обеспечения, способная к массовому воспроизводству.

В ядре было зафиксировано, что ПИ является инженерной дисциплиной, охватывающей все аспекты создания программного обеспечения от разработки требований до его использования, включает в себя планирование и сопровождение. Суть планирования – анализ задач разработки, их распределение согласно выделенным ресурсам, а сопровождение – устранение найденных недостатков в системе и внесение необходимых изменений. Отметим, что в этом определении речь идет только о *программном обеспечении* (ПО).

Разработка ПО должна проводиться с учетом ядра знаний, а именно, соответствующих разделов, дающих толкование всем направлениям и задачам проектирования и управления работами по изготовлению ПП.

Разделы ядра SWEBOK разделим, как в стандарте ISO/IEC 12207, на две группы. К первой из них (первых пять) отнести основные разделы, обеспечивающие изготовление ПП (рис.10.5), а ко второй – разделы организационного управления разработкой ПО (рис.10.6).

Дадим краткую характеристику каждого раздела названных групп и проведем сравнение с положениями стандарта ISO/IEC 12207, регламентирующего изготовление любых ПП.

10.2.1. РАЗДЕЛЫ ПРОЕКТИРОВАНИЯ SWEBOK

Согласно приведенной структуре первой группы разделов (рис.10.5) описываются задачи и методы каждого раздела, поддерживающего отдельные этапы проектирования и разработки программного продукта, начиная с анализа требований и заканчивая сопровождением изготовленного ПО.

Разработка требований к ПО

Раздел знаний SWEBOK «Разработка требований к ПО (Software Requirements)» включает в себя:

– инженерию требований (Requirement Engineering),
– выявление требований (Requirement Elicitation),
– анализ требований (Requirement Analysis),

– спецификацию требований (Requirement Specification),
– проверку требований (Requirement validation),
– управление требованиями (Requirement Management).

Рис.10.5. Основные разделы ядра SWEBOK по разработке ПС

В этом разделе представлено понятие требование, как свойства, которыми должно обладать ПО для адекватного выполнения предписанных функций, а также условия и ограничения на ПО, данные, ОС и техническое обеспечение. В нем определены подходы в формированию требований заказчиков, пользователей и разработчиков, заинтересованных в создании некоторого вида ПО.

Инженерия требований к ПО рассматривается как дисциплина анализа и документирования требований, которая заключается в преобразовании предложенных заказчиком требований к системе в описании требований к ПО и в их верификации. В основе инженерии лежат модели качества ПО, модели процессов и актеров – действующих лиц, обеспечивающих формирование, планирование и управление требованиями.

Модель процессов – это схема последовательности процессов, которые выполняются от начала проекта до определения и согласования требований. Процессом является также маркетинг и проверка осуществимости требований.

Управление требованиями к ПО заключается в планировании и контроле выполнения требований в процессе разработки программной системы на этапах ЖЦ.

Модель качества определяет подход к улучшению задания требований к ПО, обеспечивающий получение характеристик качества (надежность, реактивность и др.) на этапах ЖЦ.

Выявление требований – это процесс формирования требований из первоисточников, включающий в себя техники выявления (собеседование, сценарии, прототипы, собрания и др.) и мероприятия по идентификации интересов заказчика и разработчика в виде требований к разработке ПО.

Анализ требований – процесс изучения потребностей и целей пользователей, классификация их по требованиям к системе, аппаратуре и ПО,

разрешение конфликтов между требованиями, определение границ системы и принципов взаимодействия ПО со средой окружения. Требования классифицируются по функциональному и нефункциональному принципам. *Функциональные требования* отражают функции, которые будет выполнять система или ее ПО, а также поведение ПО при преобразовании входных данных в результаты. *Нефункциональные требования* – это требования, которые определяют условия и среду выполнения функций (защита, доступ к БД, секретность, и др.). Разработка требований и их локализация завершается на этапе проектирования архитектуры и отражается в специальном документе, по которому проводится согласование зафиксированных требований между заказчиком и разработчиком.

Спецификация требований – процесс формализованного описания функциональных и нефункциональных требований, требований к характеристикам качества в соответствии со стандартами качества (ISO 9126–98, ISO/IEC 12119–94), которые должны быть получены на этапах ЖЦ процесса разработки ПО. В спецификации отражаются требования к функциям, качеству и к документации, а также задается в общих чертах архитектура системы и ПО, алгоритмы, логика управления и структура данных. Специфицируются также системные требования, нефункциональные требования и требования к взаимодействию с другими компонентами (БД, СУБД, передача данных, сетевое взаимодействие и др.).

Валидация (аттестация) требований – это проверка требований, изложенных в спецификации, и реализованных в системе и ПО. Заказчик и разработчик ПО проводят экспертизу варианта требований для того, чтобы продолжить разработку ПО. *Верификация требований* – это процесс проверки правильности требований на их соответствие, непротиворечивость, полноту и выполнимость, а также на соответствие стандартам. В результате проверки требований создается согласованный выходной документ, устанавливающий полноту и корректность требований к ПО, и обеспечивается продолжение проектирования ПО. Одним из методов аттестации является прототипирование, т.е. быстрая отработка отдельных требований на конкретном инструменте.

Управление требованиями – это руководство процессами формирования требований на всех этапах ЖЦ, которое включает в себя управление изменениями требований и проведение мониторинга – восстановление источника требований. *Трассирование требований* состоит в отслеживании реализации отдельных требований к системе на этапах ЖЦ, и наоборот от продукта к требованиям.

Проектирование ПО

Раздел знаний SWEBOK «Проектирование ПО (Software Design)» включает в себя подразделы:

– базовые концепции проектирования ПО (Software Design Basic Concepts),

– ключевые вопросы проектирования ПО (Key Issue in Software Design),

– структура и архитектура ПО (Software Structure and Architecture),

– анализ качества проектирования ПО (Software Design Quality Analysis and Evaluation),

– нотации проектирования ПО (Software Design Notations),

– стратегия и методы проектирования ПО (Software Design Strategies and Methods).

В этом разделе проектирование ПО определено, как процесс определения

архитектуры, компонентов, интерфейсов, других характеристик системы и конечного результата.

К базовым концепциям проектирования ПО относятся методы проектирования архитектуры с использованием принципов (структурного, объектного, компонентного и др.) и техник: абстракции, декомпозиции, инкапсуляции и др. Проектируемая система декомпозируется на отдельные компоненты, осуществляется выбор готовых артефактов (нотации, методы и др.), компонентов и на их основе создается архитектура ПО.

Ключевые вопросы проектирования ПО – это декомпозиция на функциональные компоненты для независимого и параллельного их выполнения; принципы распределения компонентов и способов их взаимодействия между собой в операционной среде, а также механизмы обеспечения качества, живучести системы и др.

Проектирование структуры и архитектуры ПО выполняется архитектурным стилем путем определения основных элементов архитектуры – подсистемы, паттерны, компоненты и связи между ними.

Архитектура проекта – высокоуровневое представление структуры, задаваемое с помощью паттернов, компонентов и их идентификации. В описание архитектуры входит описание логики отдельных компонентов системы и связей между ними. Существуют и другие виды структур ПО, основанные на проектировании образцов, семейств программ и их каркасов.

Паттерн – это элемент структуры, в котором задается взаимодействие совокупности элементов проектируемой системы с определением ролей и ответственности актеров с помощью средств языка UML. *Структурный паттерн* включает в себя типовые композиции структур объектов и классов, задаваемых с помощью диаграмм классов, объектов, связей и др. *Поведенческий* паттерн состоит из схем взаимодействия классов объектов и их поведений, задаваемых диаграммами активностей, взаимодействия, потоков управления и др.

Анализ и оценка качества проектирования ПО – это системные мероприятия по анализу атрибутов качества, сформулированных в требованиях, оценка различных характеристик ПО (размер, число функций и др.) с применением функционально-ориентированных, структурных и объектно-ориентированных метрик, а также методы статического анализа, моделирования и прототипирования архитектуры ПО.

Нотации проектирования – средства представления артефактов ПО, его структуры и поведения. Существует два типа нотаций: структурные и поведенческие, а также множество различных их представлений.

Структурные нотации – графические способы представления аспектов проектирования, компонентов и их взаимосвязей, элементов архитектуры и их интерфейсов. К нотациям относятся языки спецификаций и проектирования: ADL (Architecture Description Language), UML (Unified Modeling Language), ERD (Entity Relation Diagrams), IDL (Interface Description Language), классы и объекты, компоненты и классы (CRC Cards), Use Case Driven и др.

Поведенческие нотации отражают динамический аспект поведения систем и их компонентов и задаются с помощью диаграмм: Data Flow, Decision Tables, Activity, Collaboration, Pre-Post Conditions, Sequence и др.

К стратегиям и методам проектирования ПО относятся: методы снизу–

вверх, сверху–вниз, абстракции, паттерны и др. Методы проектирования включают в себя структурный анализ, структурные карты, Dataflow-диаграммы, абстрактные структуры данных (диаграммы Джексона) и др.

Конструирование ПО

Раздел знаний SWEBOK «Конструирование ПО (Software Construction)» включает в себя следующие подразделы:

– снижение сложности (Reduction in Complexity),
– предупреждение отклонений от стиля (Anticipation of Diversity),
– структуризация для проверок (Structuring for Validation),
– использование внешних стандартов (Use of External Standards).

В этом разделе описан процесс конструирования работающего ПО с привлечением методов кодирования, верификации и тестирования компонентов. К средствам конструирования ПО отнесены ЯП, а также методы и инструментальные системы (компиляторы, СУБД, генераторы отчетов, системы управления версиями, конфигурацией, тестированием и др.). К формальным средствам описания ПО, взаимосвязей между человеком, компьютером и операционной средой отнесены языки конструирования ПО.

Снижение сложности конструирования ПО и предупреждение отклонений от стиля (лингвистического, формального, визуального и др.) обеспечивается с помощью подходящих стилей конструирования, структуризации ПО и внешних стандартов.

Лингвистический стиль основан на использовании словесных инструкций и выражений для представлений отдельных элементов (конструкций) программ. Он применяется при конструировании несложных конструкций и приводится к виду традиционных функций и процедур, их логическому и функциональному программированию и др.

Формальный стиль используется для точного, однозначного и формального определения компонентов системы. При его применении обеспечивается конструирование сложных систем с минимальным количеством ошибок, которые могут возникнуть из-за неоднозначности определений или обобщений при неформальном конструировании ПО.

Визуальный стиль является наиболее наглядным стилем конструирования ПО. Например, графический интерфейс освобождает разработчика от подбора необходимых координат и свойств объектов интерфейса. Визуальный язык проектирования UML предоставляет набор удобных диаграмм для задания статической и динамической структуры ПО, текстовое и диаграммное описание структуры ПО с выводом на экран дисплея. В процессе конструирования должны использоваться внешние стандарты ЯП (Ада 95, C++, Паскаль и др.), языков описания данных (XML, SQL и др.), средств коммуникации (COM, CORBA и др.), интерфейсов компонентов (POSIX, IDL, APL), сценариев UML и др.

Тестирование ПО

Раздел знаний SWEBOK «Тестирование ПО (Software Testing)» состоит из следующих подразделов:

– основные концепции и определение тестирования (Testing Basic Concepts and Definitions),

– уровни тестирования (Test Levels),
– техники тестирования (Test Techniques),
– метрики тестирования (Test Related Measures),
– управление процессом тестирования (Managing the Test Process).

Данный раздел рассматривается как процесс проверки работы программ в ОС, основанный на выполнении набора тестовых данных и сравнении полученных результатов с ожидаемыми.

К основным концепциям относятся терминология, теории, инструменты и принципы организации подготовки и проведения процесса тестирования ПО, а также подходы к аналитической оценке данных об ошибках, собранных в процессе тестирования.

Уровни тестирования:

– *тестирование отдельных элементов,* которое заключается в проверке отдельных, изолированных и независимых частей ПО;

– *интеграционное тестирование* ориентировано на проверку связей и способов взаимодействия (интерфейсов) отдельных компонентов, в том числе расположенных в разных компьютерах распределенной среды;

– *тестирование системы* состоит в проверке функционирования системы, обнаружении отказов и дефектов в ней и их устранении. При этом проводится контроль выполнения нефункциональных требований (безопасность, надежность и др.) к системе, правильность задания и выполнения внешних интерфейсов в операционной среде.

Видами тестирования являются:

– *функциональное тестирование*, которое заключается в проверке соответствия реализованных функций и заданий в требованиях;

– *регрессионное тестирование* – тестирование системы или ее компонентов после внесения в них изменений;

– *тестирование эффективности* – проверка производительности, пропускной способности, максимального объема данных и системных ограничений в соответствии с требованиями;

– *стресс тестирование* – проверка поведения системы при максимально допустимой нагрузке или при повышенной;

– *альфа и бета-тестирование* – внутреннее и внешнее тестирование системы Альфа – без плана, бета с планом тестирования;

– *тестирование конфигурации* – проверка структуры системы с различными наборами конфигурационных данных.

К техникам тестирования относится тестирование:

– типа *белый ящик,* основанное на задании информации о структуре системы;

– типа *черный ящик,* основанное на задании тестовых наборов данных для проверки правильности работы компонентов и системы в целом, и не требующее знания их структуры;

– на основе спецификаций, таблиц решений, потоков данных, статистики отказов и др.

Метрики тестирования предназначены для измерения процесса планирования, тестирования и оценки результатов тестирования на основе статистики об отказах и дефектах, покрытия границ тестирования, потоков данных и др.

Управление тестированием заключается в планировании процесса тестирования и измерения показателей качества ПО; в проведении тестирования reuse-компонентов и паттернов; в генерации необходимых тестовых сценариев и среды выполнения ПО; в верификации и валидации реализованных функций и требований к ПО; в сборе данных об отказах, ошибках и др. непредвиденных ситуациях; в подготовке отчетов по результатам тестирования.

Сопровождение ПО

Раздел знаний SWEBOK «Сопровождение ПО (Software maintenance)» состоит из подразделов:

– основные концепции (Basic Concepts),

– процесс сопровождения (Process Maintenance),

– ключевые вопросы сопровождения ПО (key Issue in Software Maintenance) ,

– техника сопровождения (Techniques for Maintenance).

В этом разделе описаны подходы к обеспечению жизнедеятельности ПО после поставки его заказчику, внесения в него изменений, связанных с устранением обнаруженных ошибок при эксплуатации и адаптации ПО к новой среде функционирования, а также здесь приведены соображения о необходимости повышения производительности или улучшения отдельных характеристик ПО,.

Сопровождение рассматривается с точки зрения удовлетворения требованиям в ПО, корректности его выполнения, обучения и оперативного учета процесса сопровождения.

Основные концепции включают в себя базовые определения и терминологию, подходы к эволюции и сопровождению ПО, а также к оценке стоимости сопровождения и др.

К основным определениям относятся ЖЦ ПО и документация. Сопровождение – это процесс выполнения ПО, анализ необходимости модификации и оценки стоимости работ по внесению изменений. Рассматриваются проблемы, связанные с увеличением сложности продукта при большом количестве изменений в ПО.

Процесс сопровождения включает в себя модели процесса сопровождения, планирование деятельности людей, которые проводят запуск ПО, проверку правильности его выполнения и внесения в него изменений. Процесс сопровождения состоит из:

– корректировки ПО, т.е. изменения продукта в следствие обнаруженных ошибок и нереализованных задач;

– адаптации, т.е. настройки продукта к новым условиям эксплуатации;

– улучшения, т.е. изменения продукта в целях наращивания функций или повышения производительности системы;

– системной проверки продукта для поиска и исправления скрытых ошибок.

К техникам сопровождения относятся реинженерия, реверсная инженерия и рефакторинг.

Реинженерия – это повторная реализация наследуемой системы в целях повышения удобства ее эксплуатации и сопровождения путем ее реорганизации и реструктуризации, перепрограммирования или настройки на другую платформу или среду.

Реверсная инженерия состоит в восстановлении спецификации (графов вызовов, потоков данных и др.) по полученному коду системы (особенно, когда в

нее внесено много изменений).

Рефакторинг – это процесс изменения объектов и их интерфейсов для улучшения структурных и качественных показателей объектных программ. Изменения вносятся в отдельные операции над текстами, интерфейсы, среду проектирования и в инструментальные средства поддержки ПО.

Таким образом, рассмотренные разделы регламентируют основные методы и средства для выполнения соответствующих процессов ЖЦ, которые представлены в стандарта ISO/IEC 12207.

10.2.2. ОРГАНИЗАЦИОННЫЕ РАЗДЕЛЫ В SWEBOK ДЛЯ УПРАВЛЕНИЯ

Проектирование ПО базируется на ключевых принципах и методах управления разными аспектами деятельности специалистов по созданию версии (конфигурации) системы, применению необходимых методов и средств, оценивания показателей качества, которые приведены в требованиях и др. В частности, раздел управления программный проектом (рис.10.6), содержит основной круг задач и действий, которые возникают при производстве программного проекта коллективом разнородных специалистов.

Рис. 10.6. Организационные разделы ядра SWEBOK

Управление конфигурацией ПО

Раздел знаний SWEBOK «Управление конфигурацией ПО» (Software Configuration Management–SCM) состоит из следующих подразделов:

– управление процессом конфигурации (Management of SMC Process),
– идентификация конфигурации ПО (Software Configuration Identification),
– контроль конфигурации ПО (Software Configuration Control),
– учет статуса конфигурации ПО (Software Configuration Status Accounting),
– аудит конфигурации ПО (Software Configuration Auditing),
– управление версиями ПО и доставкой (Software Release Management and Delivery).

В этот раздел включено описание способов идентификации компонентов

системы для обеспечения их системного контроля при внесении изменений, трассирования конфигурации и управления выполнением решений проекта, документированием функциональных и физических характеристик конфигурации, контроля изменений этих характеристик; отчетности о внесении изменений и верификации ПО после этих изменений.

Конфигурация системы – состав функций, физических характеристик системного и аппаратного обеспечения, а также готового ПО или комбинаций.

Конфигурация ПО включает в себя набор функций и характеристик ПО, заданных в технической документации и достигнутых в готовом продукте. Элемент конфигурации – график разработки, проектная документация, исходный и исполняемый код, библиотека, инструкции по установке системы и др.

Управление процессом конфигурации включает в себя:

– систематическое отслеживание вносимых изменений в отдельные составные части конфигурации, проведение аудита изменений и автоматизированного контроля за внесением изменений в конфигурацию;

– поддержку целостности конфигурации, ее аудит и обеспечение внесения изменений в один объект конфигурации, а также в связанный с ним другой объект;

– ревизию конфигурации для проверки разработки необходимых программных или аппаратных элементов и согласованности версии конфигурации с заданными требованиями;

– трассировку изменений конфигурации на этапах сопровождения и эксплуатации ПО.

Идентификация конфигурации ПО проводится путем выбора требуемого элемента конфигурации ПО и документирования его функциональных и физических характеристик, а также оформления технической документация на элементы конфигурации.

Контроль конфигурации ПО состоит в проведении работ по оценке, координации и утверждению реализованных изменений в элементы конфигурации с последующей их идентификацией.

Учет статуса конфигурации ПО – это комплекс мероприятий для определения уровня внесения изменений в конфигурацию, а также аудит проверки правильности внесения изменений в конфигурацию ПО.

Управление версиями ПО заключается в отслеживании версии конфигурации; создании новой версии системы на базе существующей с внесением изменений в исходную конфигурацию; согласование версии с требованиями и изменениями на этапах ЖЦ; обеспечение оперативного доступа к информации об объектах конфигурации.

Управление проектом (инженерией ПО)

Раздел знаний «Управление инженерией ПО (Software Engineering Management)» состоит из следующих подразделов:

– организационное управление (Organizational Management),

– управление процессами и проектом (Process/Project Management),

– инженерия измерения ПО (Software Engineering Measurement).

Данный раздел ориентирован на менеджера, содержит методы руководства работами команды разработчиков в процессе выполнения проекта, критериев

оценки и измерения процессов и продуктов проекта с использованием общих методов управления, планирования и контроля работ.

Менеджмент проекта – планирование, координация, измерение, мониторинг, контроль и отчет. Управление гарантирует системную, дисциплинированную и измеряемую разработку ПО. Ответственность за координацию человеческих, финансовых и технических ресурсов при реализации целей проекта несет менеджер проекта. В его задачу входит также выполнение целей проекта с соблюдением бюджетных и временных ограничений, стандартов и требований.

Более подробная информация о проблеме управления проектом содержится в ядре знаний – PMBOK (Project Management Body of Knowledge (http://petukhov.zaklad.rupmbok2004_rus.pdf), а также в стандарте ISO/IEC 12207 – Software life cycle processes [224].

Организационное управление включает в себя процессы управления, планирования и составления графика работ, оценивания стоимости работ, подбор кадров и контроль выполнения работ. Главные задачи организационного управления – это подготовка персонала (тренировка и мотивация, обучение разработчиков), управление коммуникациями (каналами, встречами, презентациями) и риском (техника управления, селекция проекта и др.). Создается организационная структура коллектива, специалисты которой распределяются по ответственным участкам и все вместе подчиняются менеджеру проекта.

Управление процессами проекта включает в себя исследование, составление и уточнение плана проекта, построение графика работ (сетевых и временных диаграмм) с учетом ресурсов, распределение персонала по этапам работ и длительности их выполнения; анализ финансовой, технической, операционной и социальной политики; контроль процессов управления.

Управление продуктом включает в себя уточнение требований и проверку (валидацию) соответствия создаваемого продукта заданным требованиям и верификацию правильности реализованных функций в проекте.

Управление рисками представляет собой процесс определения рисков и разработки мероприятий по уменьшению их влияния на ход выполнения проекта. *Риск* – вероятность проявления неблагоприятных условий, которые могут негативно повлиять на реализацию качества проекта (например, увольнение сотрудника, отсутствие его замены и др.) Предотвращение риска – это действия, которые снимают риск (например, увеличение времени разработки и др.) или уменьшают вероятность появления нового риска.

Измерение процессов инженерии ПО – это определение категорий рисков и отслеживание факторов для регулярного пересчета вероятностей их возникновения; совершенствование процессов управления проектом; оценки временных затрат и стоимости ПО в целях их регулирования и др.; выбор метрик для процессов и продукта и их оценка.

Процесс инженерии ПО

Раздел знаний SWEBOK «Процессы инженерии ПО (Software Engineering Process)» состоит из описаний следующих подразделов:

– концепции процесса инженерии ПО (Software Engineering Process Concepts),

– инфраструктура процесса (Process Infrastructure),

– определение процесса (Process Definition),

– процесс измерения (Process Measurement),
– качественный анализ проекта (Qualitative Process Analysis),
– выполнение процесса и изменение (Process Implementation and Change).

Данный раздел включает в себя концепции, инфраструктуру, методы определения и измерения процессов ЖЦ, поиска изменений и их реализаций, а также методы анализа и оценки качества ПП.

Инфраструктура процесса – это виды ресурсов (группы разработчиков, технические средства, программные продуктов и т.п.), а также процесс инженерии ПО (групповой или по типу экспериментальной фабрики – Experience Factory), базирующиеся на моделях проекта и продукта, моделях качества и риска, методах управления коллективом и организации разработки проекта.

Определение процесса основывается на типах процессов и моделей ЖЦ (водопадная, спиральная, итерационная и др.), стандартах ЖЦ ПО ISO/IEC 12207 и 15504, IEEE std 1074–91 и 1219–92, а также на методах спецификации процессов и средствах их поддержки.

Процесс измерения предполагает стадию наблюдения за выполнением процесса для сбора информации, моделирования, классификации ошибок и дефектов, а также для статического контроля и измерения отдельных показателей процесса.

Качественный анализ процесса заключается в идентификации и поиске слабых мест в ПО до начала его выполнения. Рассматривается две техники анализа: обзор данных и сравнение данного процесса со стандартным ISO/IEC–12207; сбор данных об атрибутах качества продукта и процессов; анализ причин отказов в функционировании ПО, откат назад от точки возникновения отклонения до точки нормальной работы ПО и выяснения причин.

Выполнение процесса и изменение – это развертывание ПО, инспекция проекта, принятие решений о необходимости изменения процесса, организационной структуры проекта и некоторых инструментов управления проектированием.

Методы и средства инженерии ПО

Раздел знаний ядра SWEBOK «Методы и средства инженерии ПО (Software Engineering Tools and Methods)» состоит из подразделов:
– инструменты (Software Tools),
– методы (Software Methods).

Данный раздел содержит сведения, необходимые для определения среды разработки, используемой на процессах ЖЦ, а также приведены методы и средства поддержки процессов: спецификации требований, конструирования и сопровождения ПО. Методы обеспечивают проектирование, реализацию и выполнение ПО в процессах, а также оценку качества процессов и продуктов.

Инструменты инженерии ПО разделены по следующим типам: сбор требований, проектирование ПО (редакторы схем и диаграмм), конструирование ПО, тестирование, автоматизация процесса инженерии ПО, контроль качества, управление конфигурацией ПО и программным проектом.

Методы инженерии ПО включают в себя эвристические (неформальные) – структурные, объектно-ориентированные методы, ориентированные на данные и на прикладную область, а также формальные методы проектирования и

прототипирования.

Качество ПО

Раздел знаний SWEBOK «Качество ПО (Software Quality)» состоит из описания следующих подразделов:

– концепция качества ПО (Software Quality Concepts),

– определение и планирование качества (Definition & Planning for Quality),

– деятельность и техника гарантии качества и V&V(Activities and Techniques for Software Quality Assurance, Validation–V & Verification – V),

– измерение в анализе качества ПО (Measurement in Software Quality Analysis).

Данный раздел содержит описание сформировавшегося набора характеристик продукта или сервиса, которые определяют его способность удовлетворять установленным или предполагаемым потребностям заказчика (пользователя) ПО. Это описание включает в себя обширный материал по проблеме качества ПО и путей его достижения в процессе проектной деятельности групп разработчиков.

Концепция качества ПО – это современные методы и стандарты определения внешних и внутренних характеристик качества и их метрик, а также модели качества, заданные на множестве внешних характеристик и представленные в стандартах качества. Установлено шесть базовых характеристик качества и для каждой из них по 4-5 атрибутов. К ним относятся: функциональность, надежность, удобство использования, эффективность, сопровождаемость, переносность. Стандартная модель качества определяет любой тип программного продукта, а конкретная модель качества включает только те характеристики, которые указаны в требовании к конкретному ПО.

Определение и планирование качества ПО основывается на стандартах, планах графиков работ и процедурах проверки и др. План обеспечения качества – набор действий для проверки процессов обеспечения качества (верификация, валидация и др.) и формирование документа по гарантии качества.

Техника гарантии качества и V&V включают в себя инспекцию, верификацию и валидацию ПО.

Инспекция ПО – анализ и проверка различных представлений системы и ПО (спецификаций, архитектурных схем, диаграмм и др.) и выполняется на всех этапах ЖЦ разработки ПО.

Верификация ПО – процесс проверки и анализа правильности выполнения функций системы в соответствии со спецификаций. Верификация дает ответ на вопрос правильно ли создана система. *Валидация* – процесс проверки соответствия ПО функциональным и нефункциональным требованиям и ожидаемым требованиям заказчика. По результатам проверки делается оценка полноты реализации функций и требований.

Измерения в анализе качества ПО – это измерение характеристик процессов, ресурсов и продуктов; оценки модели процесса измерения и документации. Для измерения фактических характеристик качества продукта проводится *тестирование ПО* – проверка продукта на тестовых данных и анализ выходных результатов. Тесты разрабатываются так, чтобы имитировать работу системы с реальными исходными данными и фиксировать возникающие отказы, дефекты и ошибки. Тестирование заканчивается измерением полученных характеристик качества, атрибуты которых заданы в требованиях.

Эти разделы недостаточно полно отображают вопросы индустрии ПП, например, отсутствуют технологии подготовки и управления разработкой по технологическим линиям, теория экспертного анализа проектных решений на ТЛ, инженерия управления и экономики производства ПП. Приведенные методы в ядре SWEBOK не охватывают сути индустриальных принципов производства новых видов целевых объектов (доменов, прикладных систем, семейств систем) и инструментов их изготовления [6–14]. Поэтому далее предлагается набор дисциплин, которые определяют суть отдельных аспектов производства ПП,

Таким образом, организационные разделы SWEBOK и которые приведены в стандарте ISO/IEC 12207 имеют много общего (название и задачи). Они наполняют эти процессы содержательным смыслом в части применения на них необходимых методов и средств при выполнении задач процессов.

Назначение и применение SWEBOK в производстве и обучении

Ядро знаний SWEBOK предоставляет всю необходимую информацию для выбора наиболее подходящих методов, средств и инструментов для реализации в процессах ЖЦ задач проектирования и управления изготовлением ПС на инженерной основе, означающей, что вся деятельность с технологической точки зрения управляется, контролируется, оценивается и измеряется. Таким образом, ПИ обеспечивает множество задач по созданию ПО, дополняя процессы разработки (анализе, проектирование и реализация) такими процессами, как управление проектом, конфигурацией, качеством ПО, а также оценка результатов труда и затраченных ресурсов. К инженерной деятельности относятся планирование и сопровождение. Планирование предполагает анализ целей и задач проекта, возможностей его реализации и необходимых ресурсов, а сопровождение обеспечивает проверку реализованных задач и устранение найденных ошибок в системе, среде и деятельности изготовителей.

Инженерный аспект деятельности в ПИ близок к инженерной деятельности, определенной в толковом словаре так:

– инженерия – это применение научных результатов для получения пользы от свойств материалов и источников энергии;

– инженерия – это деятельность по созданию новых машин для предоставления полезных услуг.

Согласно содержанию науки ПИ, являющейся разделом Computer Science, деятельность по разработке ПС выполняют инженеры. Они применяют теорию, методы и средства компьютерной науки, которая предоставляет теорию и методы *построения* вычислительных и программных систем. Отметим, что индустриальный аспект производства ПП в ядре знаний SWEBOK не рассмотрен, а созданы лишь предпосылки. Знание компьютерной науки необходимо специалистам в области ПС и проектов так же, как знание физики – инженерам-электронщикам. При этом решение отдельных задач проектирования и разработки ПС инженер может выполнить, применяя практические знания, накопленные им в процессе создания конкретных разработок и специальных инструментально-программных средств в рамках заключенных контрактов и с учетом разных (финансовых и технических) ограничений.

В отличие от других наук, целью которых является получение знаний, в ПИ *знание* – это способ получения некоторой общественной пользы. Специалист в

области ПИ, Ф.Брукс считает, что «ученый строит, чтобы научиться, инженер учится, чтобы строить».

Признав разработку ПО инженерной деятельностью, отметим, что она имеет отличия от традиционной «технической» инженерии:

– традиционные «ветви» инженерии имеют высокую степень специализации, а у программной инженерии специализация заметна только в довольно узких применениях (например, операционные системы, трансляторы, редакторы и др.);

– объекты традиционной инженерии хорошо определены и действия над ними происходят в узком контексте общих технических проектных решений, которые отвечают типовым требованиям заказчиков и касаются отдельных деталей, а не общих вопросов разработки, тогда как у программной инженерии подобная типизация отсутствует;

– отдельные готовые решения и изделия в традиционной инженерии классифицированы и каталогизированы, а в ПИ каждое новое решение и разработка некоторого элемента программной системы – это новая проблема, для которой довольно трудно установить аналогию с ранее выполненными разработками таких продуктов, как программа, компонент, система, семейство систем и т.п.

Приведенные отличия требуют значительных усилий для их нивелирования и превращения ПИ в инженерную и научную специальность.

Мировая компьютерная общественность провела работы по преобразованию ПИ в специальность. Она преподается во многих университетах и студенты получают степени бакалавра и магистра. Этим и другими аспектами подтверждается, что специализация профессиональной деятельности в ПИ постепенно становится «зрелой». Для нее существуют:

– системы начального обучения специальности;

– механизмы развития умений и навыков персонала, которые необходимы для его практической деятельности;

– лицензирование специалистов, организованное под управлением соответствующих государственных органов;

– системы профессионального повышения квалификации персонала и отслеживания современного уровня знаний и технологий по данной специальности;

– этический кодекс специалистов;

– профессиональные объединения.

В последнее десятилетие инженерная деятельность в ПИ приблизилась к энциклопедическому определению, она стала специальностью в области информатики. Сформированы профессиональные организации и объединения, которые создали ядро знаний SWEBOK (www.swebok.com), приняли этический Кодекс специалистов по программной инженерии, разработали руководства для обучения программной инженерии, а также создали программу обучения Computing Curricula – 2001, 2004 [181]. В США работает комитет по сертификации учебных заведений Computing Accreditation Commission of the Accreditation Board for Engineering and Technology.

10.3. НАУЧНО–ТЕХНОЛОГИЧЕСКИЕ ДИСЦИПЛИНЫ ДЛЯ РЕШЕНИЯ ЗАДАЧ ИНДУСТРИАЛЬНОГО ПРОИЗВОДСТВА ПРОГРАММ

ПИ имеет фундаментальный базис, связь с другими науками, интеграцию принципов математики и основных положений теории алгоритмов, математической

логики, управления, множества, доказательства и др.[132–140].

К фундаментальным основам построения ПП относятся следующие:

– правила логических исчислений, высказываний и умозаключений математической логики;

– математический вывод теорем и программ с помощью аксиом и утверждений, верификация программ (VDM, RAISE, Z, методы Хоара, Дейкстры и др.);

– кванторы всеобщности, существования и операции над множествами;

– принципы, методы планирования и управления теории управления.

Кроме того, систему знаний ПИ образуют:

– формальные методы программирования для спецификации программ, их верификации и тестирования, а также оценки надежности ПП и т.п.;

– прикладные методы, а именно: средства, принципы, правила и процессы ЖЦ производства прикладных систем, как инструментов коллективной разработки больших программных проектов;

– методы управления коллективами, включая планирование сетевых графиков, контроль работ на процессах ЖЦ, измерение и оценивание качества созданного промежуточного и конечного ПП, регулирование сроков и стоимости изготовления и сертификации.

10.3.1. СОДЕРЖАНИЕ НАУЧНОЙ ДИСЦИПЛИНЫ ПРОИЗВОДСТВА ПРОГРАММНЫХ ПРОДУКТОВ

В отличие от математической или другой фундаментальной науки, цель которых – получение новых знаний для решения соответствующих задач, в ПИ знания – это общая теория построения программ для компьютеров, изготовление продукта для получения посредством него результата [132].

Научная дисциплина – это теоретические, формальные методы и средства построения сложных программных объектов. Построение включает анализ предметной области, проектирование и образование выходного кода для его выполнения на компьютере. Данная наука предоставляет базовые понятия, объекты и формальные механизмы, необходимые при создании общих и специфических особенностей ПП в соответствии с заданными к нему требованиями. Иными словами, наука ПИ включает в себя:

1) основные понятия и объекты (исходные и проектируемые);

2) теорию программирования и методы управления изготовлением ПП;

3) средства и инструменты процессов производства продукта.

1. Основные понятия – это: данные и их структуры, функции и композиции, базовые объекты (модуль, компонент, каркас, КПИ и др.) и целевые объекты (ПО, ПС, семейство систем, программный проект, сложные программные системы).

Разработка простых объектов осуществляется с помощью элементарных операций формального их описания, а изготовление из них целевых объектов – это применение инженерных методов управления коллективом, их работами, сроками и стоимостью.

Приведем общее определение целевых объектов ПИ.

Программная система (Application) – комплекс интегрированных программ и средств, которые реализуют набор функций некоторой ПрО в заданной среде. В комплекс могут входить: ПО, прикладные системы (зарплата, учет и др.), общесистемные компоненты (транслятор, редактор, СУБД и т.п.), системы защиты

и безопасности функционирования. *Способ изготовления* – инженерия ПС или application engineering, которая включает процессы ЖЦ, методы разработки, управления, оценивания продуктов и процессов.

Программное обеспечение – совокупность программных средств, которые реализуют функции компьютерной системы (или функции аппаратно-программной системы), включая общесистемные средства (например, ОС, СКБД, системы защиты и т.п.), подсистемы контроля (показателей процессов, обработки сигналов) и прикладные ПС. Так например, основные функции ОС – это управление задачами, программами, данными и т.п. ПО может входит в состав ПС или быть идентичным функциям ПС. *Способ изготовления* – инженерия реализации задач.

Семейство систем (systems family) – совокупность ПС с общим (неизменным для всех членов семейства) и управляемым изменяемым набором ее характеристик, удовлетворяющих требованиям прикладной области (домену). *Способ изготовления* – инженерия домена (Domain Engineering) или конвейерное производство однотипных ПП с единой схемою (каркасом) и специально разработанными отдельными членами семейства и других готовых ПС с помощью базового процесса или линейки продукта (Product line).

Программный проект – уникальный и интегрированный программный продукт, в котором отображены совокупности реализованных целей, задач и результатов деятельности, удовлетворяющих требованиям заказчика проекта. *Способ изготовления* – инженерия производства и менеджмента проекта.

Сложные программные объекты – совокупность взаимосвязанных целевых объектов разных типов, которые выполняют необходимые функции в такой системе, представленные вновь разработанными простыми объектами или выбранными с репозитария готовых ресурсов ПИ.

2. Теория программирования – это множество методов, языков и средств описания (спецификации) и проектирования целевых объектов, а также методов их доказательства, верификации и тестирования. Кроме того, в программную инженерию привлечены формальные методы управления проектом (персоналом, материальными и финансовыми ресурсами) и отдельными его показателями.

Методы программирования в ПИ описаны ниже и представлены на рис.10.7:

– теоретические методы (алгебраический, алгоритмический, VDM, RAISE и др.) и прикладные (объектный, компонентный, аспектный и др.), предназначенные для проектирования целевых объектов;

– методы проверки правильности программ формальными процедурами (утверждения, вывода и др.);

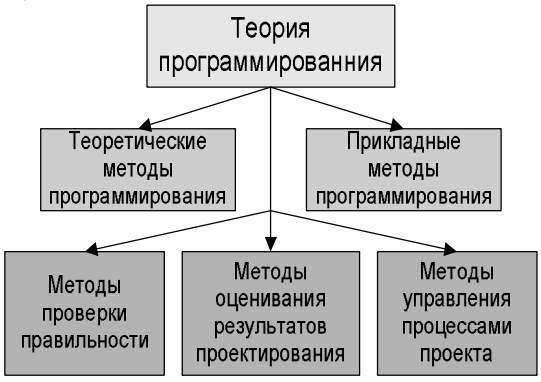


Рис. 10.7. Методы ПИ

– методы экспертизы промежуточных и конечных результатов проектирования на процессах проекта и оценки показателей качества (надежность, точность, производительность и т.п.);

– методы оценки результатов последовательного проектирования (промежуточных рабочих продуктов) на процессах проекта и оценки конечного продукта на показатели качества (надежность, точность, производительность и т.п.);

– методы управления планами и контроля промежуточных результатов на процессах проекта, а также дополнительные расчетные методы (трудозатрат на каждую работу, ее стоимость и т.п.).

3. Средства и инструменты ПИ. Все объекты как артефакты ПП продукта включают в себя разные описания: требования к разработке, согласованные с заказчиком, архитектуру, структуры данных, спецификации программ и т.п.

Проектирование целевых объектов осуществляется с помощью современных языков, в том числе визуальных (например, C++, Java, Pascal, UML, и др.) и использования соответствующих инструментальных сред, которые содержат необходимые языковые преобразователи и инструменты поддержки артефактов процесса разработки. К таким средствам также относятся шаблоны, каркасы, диаграммы потоков данных, классов, поведения и т.п.

Проверка правильности целевых объектов осуществляется методами программирования и соответствующими инструментами, приспособленных для разработки разных задач проекта в соответствующей среде проектирования. Готовый продукт проверяется на соответствие реализованных функций заданным требованиям, тестируется с помощью специальных методик, а также подвергается измерению и оцениванию в целях получения показателей качества ПП.

В качестве среды проектирования целевых объектов используют передовые, современные технологии и соответствующие инструментально-технологические системы (например, Microsoft Visual Studio, MSF, RUP, Rational Rose и др.). Они поддерживают проектирование разных типов целевых объектов, управление менеджментом проекта (персоналом, планами) и качеством продуктов. Эти средства и инструменты удобно использовать в коллективной разработке проекта.

10.3.2. ОСНОВЫ ИНЖЕНЕРИИ ПРОИЗВОДСТВА ПРОГРАММНЫХ ПРОДУКТОВ

Инженерная дисциплина (или инженерия) в ПИ – это способы выполнения работ, связанных с изготовлением программного продукта для разных видов целевых объектов с применением методов, средств и инструментов научной дисциплины программной инженерии [132], основу которой составляют следующие базовые элементы (рис. 10.8):

1) ядро знаний SWEBOK – набор теоретических концепций и формальных правил, которые могут применяться в изготовлении ПП;

2) базовый процесс ПИ – стержень процессной деятельности в организации-разработчике ПП;

3) стандарты ПИ – набор регламентированных правил конструирования артефактов на процессах ЖЦ;

4) инфраструктура – условия среды и

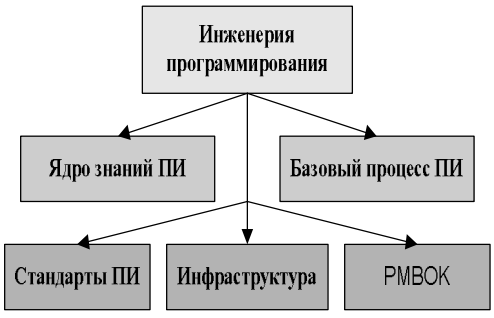


Рис. 10.8. Базовые элементы инженерной дисциплины

методического обеспечения базового процесса ПИ и исполнителей, которые занимаются изготовлением ПП;

5) менеджмент проекта (PMBOK) – стандартные положения и процессы, а также научные принципы и методы планирования и контроля работами проекта.

С инженерной точки зрения в программной инженерии реализуются задачи изготовления ПП, как технологические процессы формирования требований, проектирования и сопровождения продукта, а также проверки операций базового процесса на правильность реализации разных функциональных задач проекта и завершения его работ в заданный заказчиком срок.

Программную инженерию можно рассматривать с двух точек зрения:

– как инженерную деятельность, в которой инженеры разных категорий осуществляют работы по проекту, учитывают соответствующие теоретические методы и средства ПИ, рекомендованные в ядре знаний SWEBOK, а также положения процессов ЖЦ стандартов и выбранных для процессов методов;

– как систему управления проектом, качеством и рисками с привлечением правил и положений стандартов ЖЦ, качества и менеджмента проекта [5, 6, 129].

Инженерная деятельность планируется, в ней проводится распределение проекта на отдельные работы, которые будут выполнять исполнители проекта. Менеджер проекта – это главное действующее лицо проекта, ответственное за проектирование и контроль выполнения этих работ разными службами инфраструктуры проекта в организации (верификации, тестирования, оценки качества и др.). Продукт коллективного изготовления передается заказчику для сопровождения. Найденные в нем разные ошибки и недостатки устраняются разработчиками.

Эта деятельность практически уже отработана и по своей сути она близка к инженерной деятельности в промышленности, где *инженерия* – это способ применения научных результатов изготовления некоторых технических изделий с помощью технологических правил и процедур, методик измерения, оценки и сертификации изготовленного продукта.

Ниже приведена общая характеристика базовых элементов пп.1–5 инженерной дисциплины, представленных на рис 10.8.

1. Ядро знаний SWEBOK – это главная содержательная трактовка и перечень концептуальных основ разделов программной инженерии. Структурно ядро включает в себя 10 разделов (knowledge areas). Их условно поделим на две категории: проектная, организационная и инженерная деятельность. Первая категория – это методы и средства разработки (формирование требований, проектирование, конструирование, тестирование, сопровождение). Вторая категория – это методы управления проектом, конфигурацией, качеством и базовым процессом организации-разработчика.

Методы ядра знаний SWEBOK менеджер проекта сопоставляет с задачами стандартных процессов ЖЦ и обеспечивает наполнение базового процесса этими методами. Таким образом, создается базис инженерной дисциплины изготовления продукта, включающий регламентированную последовательность процессов разработки и сопровождения программного продукта. На начальных процессах определяются требования к продукту, проектные решения и каркас (абстрактная архитектура) будущего продукта. На основе требований и каркаса разрабатываются новые или подбираются готовые объекты для „заполнения”

каркаса конкретным содержанием и для последующей его доработки до конечного программного продукта.

2. Базовый процесс – это метауровень для процессирования изготовлением продукта. Он содержит основные понятия, которые относятся к оснастке, организационной структуре коллектива разработчиков и методологии оценки, измерения, управления изменениями и совершенствованием самого процесса. В целом базовый процесс – это множество логически связанных с ним видов инженерной деятельности организации-разработчика и набор средств и инструментов, предназначенных для использования при изготовлении программного продукта.

3. Стандарты ПИ – это технологически отработанный набор разных процессов со строго определенным и регламентированным порядком проведения работ в программной инженерии, связанных с разработкой и оцениванием продукта на качество, риски и т.п. Стандарты в области программной инженерии регламентируют разные направления деятельности. Они стандартизируют термины и понятия, ЖЦ, качество, измерение, оценку продуктов и процессов. Наиболее важными среди них – стандарт ISO/IEC 12207, стандарты серии ISO/IEC 9000 – 1, 2, 3, ДСТУ 2844–94 и 2850-94, регламентирующие аспекты системы управления и обеспечения качества ПП.

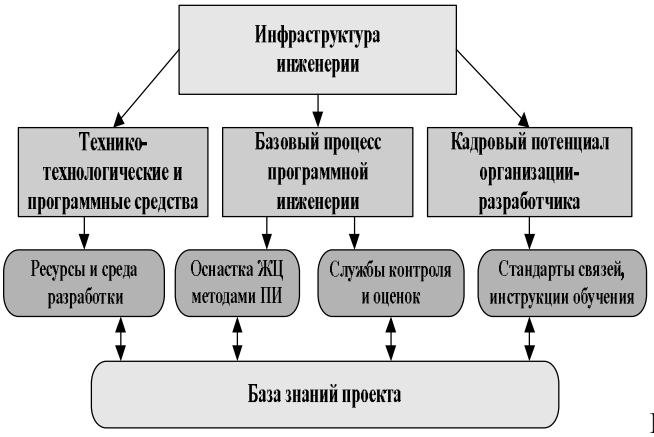
Стандарт ISO/IEC 12207 содержит организационные процессы планирования, управления и сопровождения. Процесс планирования включает в себя действия по составлению планов и графиков работ для разработки проекта, распределения работ между разными категориями специалистов, а также действия по контролю планов и выполненных работ. Процесс управления проектом определяет процессы управления работами проекта, которые должны выполнять специалисты, владеющие теорией управления, способные планировать и следить за сроками выполнения проекта. Третий процесс – это сопровождение готового продукта проекта, выявление и устранение найденных в нем недостатков, а также дополнения новых или удаление некоторых функций в продукте.

Ядро знаний SWEBOK и стандарты ЖЦ взаимосвязаны. Они используются при определении структуры базового процесса ПИ и разработки необходимых методик и ограничений при изготовлении продукта. Здесь могут использоваться фундаментальные модели ЖЦ (водопадная, спиральная и т.п.), которые реализуют заложенный в них стиль проектирования и реализации для некоторых видов продуктов.

4. Инфраструктура – это набор технических, технологических, программных (методических) и человеческих ресурсов *организации-разработчика*, необходимых для выполнения базового процесса, ориентированного на выполнение договора с заказчиком программного проекта (рис.10.9.).

К техническим ресурсам проекта относятся: компьютеры, устройства (принтеры, сканеры и т.п.), серверы и т.п.; к программным – ПО, информационное и системное обеспечение. Технологические и методические ресурсы – это методики, процедуры, правила, рекомендации стандартов для процесса и управления персоналом.

Человеческие ресурсы – это группы разработчиков и служб управления проектом, планами, качеством, риском, конфигурацией и проверки правильности выполнения проекта. Вместе – это базовый процесс и комплект документов по регламентации, выполнению и регулированию процессов ЖЦ.



Р

Рис.10.9. Элементы инфраструктуры проекта

Средства, промежуточные результаты разработки на процессах ЖЦ, а также методики руководства ресурсами БП и результаты применения соответствующих методов программирования, сохраняются в базе знаний проекта.

После выполнения проекта и получения опыта построения конкретного проекта базовый процесс и его элементы могут совершенствоваться (доработка, замена, добавление новых средств) соответственно стандарту ISO 15504 (Software Process Improvement and Capability).

Готовность всех видов элементов процесса и персонала организации-разработчика производить качественный продукт – основа оценки зрелости как самой организации, так и выпущенного продукта. Для этого применяется модель зрелости CMM (Capability Maturity Models) Института программной инженерии SEI США [132, 133]. Такая модель рекомендует подход к рассмотрению разной степени готовности организации при создании программных продуктов (например, удовлетворительно, средне, хорошо и очень хорошо). Уровень степени готовности определяется наличием в организации базового процесса, всех необходимых для него ресурсов, соответствующих стандартов и методик, а также профессиональных способностей (зрелости) членов коллектива изготавливать программные продукты в заданный срок и укладываться в стоимость.

Модель CMM – пятиуровневая. Первый и второй уровень определяют недостаточную подготовку организации зрело производить продукт. Третий – пятый уровни характеризуют выше ступень готовности, зрелости и профессионализма специалистов организации изготавливать, соответственно, средний, хороший и отличный продукт. Пятый уровень зрелости свидетельствует о способности команды разработчиков создавать качественный программный продукт. Кроме базовой модели, используются CMM – SW для оценки зрелости ПО, CMMI для учета потребностей государственных структур в ПО, Bootstrap – для оценки зрелости маленьких и средних коммерческих компаний [133, 179].

5. Менеджмент проекта – это управление разработкой проекта с использованием адаптированной теории управления и процессов ядра знаний PMBOK (Project Management body of knowledge) к типу проекта. В PMBOK представлены положения и правила руководства в соответствии с заданным временем производственного цикла построения уникального продукта в рамках некоторого проекта [8]. PMBOK – это стандарт IEEE Std.1490:2003, который

разработан американским Институтом управления проектами (www.pmi.org) для создания ПО.

Ядро знаний PMBOK построено аналогично ядру знаний SWEBOK и содержит описание лексики, структуры процессов и областей знаний, которые отображают современную практику управления проектами в некоторых областях промышленности и в частности ПО. Ядро содержит процессы и определяет принятие решений относительно ресурсов проекта в каждый момент времени выполнения и управления задачами проекта. Раздел знаний – управление содержанием проекта – это процессы, регламентирующие планирование, разбивку работ на более простые в целях упрощения процесса управления. Раздел управления качеством содержит процессы и операции достижения целей проекта, а также правила и процедуры для улучшения процесса обеспечения качества в соответствии с заданными требованиями заказчика и положениями стандарта качества (ISO/IEC 9000 –1, 2, 3). Раздел управления человеческими ресурсами включает в себя распределение работ между исполнителями в соответствии с их квалификацией и профессионализмом.

Таким образом, стандарты PMBOK и ISO/IEC 12207 имеют много общих процессов по организации управления проектом и его ресурсами.

Современные технологии инженерного производства ПП

Главная особенность производства ПП – это использование разработанных готовых программ, КПИ и информационных ресурсов Интернет (MatLab, Greenstone, Grid-системы и др.). Доступ к ним может осуществить любой пользователь и получить бесплатно или на коммерческой основе готовый ресурс, как сервис. Он может быть одноразово использован для решения соответствующей задачи, либо как некоторая программа постоянного и многоразового применения в некотором домене.

Практика изготовления отдельных ПП привела к формированию трех инженерных технологий: Reusing Engineering (инженерия КПИ), Application Engineering (инженерия приложений), Domain Engineering (инженерия доменов), Family System Engineering (инженерия семейства систем). Их основа – готовые программные ресурсы, а именно, КПИ, компоненты, приложения, ПС и системы. Применение готовых ресурсов дает значительную экономию при производстве на их основе новых ПС и семейств систем. Все виды КПИ сохраняются в современных хранилищах – репозитариях и библиотеках [20, 22].

Технология производства ПП базируется на готовых ресурсах, средствах и инструментах их построения. Основу изготовления новых ПП составляет каркас, заполняемый КПИ, вновь реализованными компонентами, а также прикладные системы (например, из бизнеса, коммерции, экономики и т.п.) и системы общего назначения (трансляторы, редакторы, СУБД, системы интеграции, генерации и т.п.).

Инженерия приложений и инженерия доменов основана на многократном использовании разных КПИ и других программных элементов. Основная задача этих видов инженерно-производственной деятельности – построение прикладных систем или семейств систем для реализации задач приложения или домена с учетом общих и изменяемых характеристик ее членов. Технология изготовления доменов вплотную подошла к современным принципам конвейерного производства

продуктов из готовых «деталей» типа КПИ по модели домена в DSL (Domain Specific Language) и спецификациям каждого члена семейства [133, 140, 204]. Основная суть этой технологии – управление работами по изготовлению ПП, базирующееся на планах–графиках работ, контроле и экспертизе результатов работ, оценивании степени применимости готовых ресурсов в процессе реализации специфических задач домена.

Базовые компоненты данной инженерной дисциплины должны непрерывно совершенствоваться и адаптироваться к новому типу целевых объектов и условиям производственной среды (что в духе концепций совершенствования, заложенных в моделях CMM, SPICE, Trillium и др.).

В производстве ПП первостепенное значение имеют инженерные методы. Без них не мыслится ни один промышленный продукт. Здесь нужно тщательно исследовать все наработки (как научные, так и инженерные) в области компьютерных наук, и на их основе создать фундамент инженерной дисциплины, который будет включать в себя описание стандартных принципов инженерии и базовых компонентов, а также современных языков спецификации доменов, членов семейств и процессов производства ПП средствами и инструментами инженерных технологий.

Инженерия КПИ – это систематическая и целенаправленная деятельность для подбора реализованных и представленных в репозитарии КПИ [20, 22]. Система проектируется из них снизу–вверх. Сначала создается общая структура – каркас продукта, дается его описание и по этому описанию готовые компоненты и КПИ интегрируются в систему.

Инженерия приложений – это деятельность по изготовлению более сложного продукта из КПИ, готовых программ и прикладных систем. Проектирование одиночных уникальных приложений – это инженерия программирования из готовых КПИ. Изготовление начинается с анализа предметной области, определения концептуальной модели, разработки проектных решений и сборки компонентов с использованием шаблонов или каркасов.

Инженерия ПрО – это деятельность по изготовлению систем семейства (доменов) на основе модели, содержащей общие и изменяемые характеристики представителей семейства и набора средств, инструментов производства из готовых ресурсов. Выбранные КПИ или одиночные приложения встраиваются в модель домена. Технология разработки семейства включает процессы анализа, проектирования и встраивания КПИ в отдельные представители семейства. По своему характеру данная инженерия приближена к конвейерному производству продуктов из готовых ресурсов. Управление этой инженерией состоит в планировании и распределении работ за каждым участником процесса создания домена и контроле их выполнения в заданные сроки.

Линии продуктов. Технологические приемы производства программных продуктов из готовых компонентов и ПС воплощены в так называемые *линии продуктов,* которые соответствуют конвейерному производству программных продуктов на основе спроса рынка [217, 224].

Практические инструменты производства – ядро знаний SWEBOK, PMBOK и стандарты, в которых регламентированы процессы инженерии доменов (Domain engineering process). Согласно этому стандарту процесс инженерии доменов включает в себя: анализ домена (выявление связей, постоянных и изменяемых

требований, понятий и моделей); проектирование домена (Domain design) – каркаса для КПИ, активов (asset) и интерфейсов, согласованных с моделью домена); технологию домена с подпроцессами формирования ресурсов (asset provision), разработкой базы ресурсов (asset-based development) и сопровождения ресурсов. В результате применения инженерной технологии домена в софтверной организации создается архитектурный базис производства из многоразовых КПИ репозитария и языков спецификации специфических особенностей разных доменов.

Таким образом, наука и инженерия, SWEBOK, СТАНДАРТЫ, PMBOK – главные составные элементы производства ПП. При изготовлении используются инструментально-технологичные системы и среды с набором необходимых инструментов. Пример среды с реализацией в ней инженерного подхода к проектированию ПП – система Visual Studio Teams Systems фирмы Microsoft. В ней реализованы все этапы технологии проектирования ПП [136].

В этой среде реализована *технология* подбора, распределения и выполнения работ между группами специалистов, имеющими разные уровни знаний и навыков в области программирования. Они разделены на четыре категории специалистов, в которой каждый получает работу по своим способностям. Переход в более квалифицированную группу зависит от качества выполненных работ предыдущей задачи и от полученного опыта. Т.е. четвертая группа – это специалисты высокой квалификации, они несут ответственность за правильность разработки проекта в целом.

10.3.3. УПРАВЛЕНИЕ ИЗГОТОВЛЕНИЕМ ПРОГРАММНЫХ ПРОДУКТОВ

Базис дисциплины управления ПП – классическая теория управления, менеджмент производства проектов и стандарт IEEE Std.1490 PMBOK (Project Management Body of Knowledge). Теория управления, а также теория организационного управления разработанная акад. В.М.Глушковым в 70-х годах прошлого столетия, проверена практикой при построении технологических процессов в металлургической, судостроительной и химической промышленностях, а также при внедрении в целях массового производства телевизоров, автомобилей и др. (например, АСУ “Львов”) [52, 57].

Таким образом, теория управления сложными системами развивалась заграницей, особенно в части теории планирования производством. Так, на фирме «Dupon» для планирования и создания планов-графиков больших комплексов работ при модернизации заводов был разработан метод CRM (Critical Path Method), базис которого – графическое представление работ, соответствующих видов операций и времени их выполнения. Другой метод – сетевое планирование PERT (Program Evaluation and Review Technique), был апробирован при реализации проекта разработки ракетной системы «Polaris», которая объединяла около 3800 подрядчиков (число операций свыше 60 тыс.). Применение метода было успешным, и в результате проект был завершен на два года раньше запланированного срока. Каждый из этих методов возник в недрах промышленного производства. В настоящее время они адаптированы к среде программирования и стали использоваться в индустрии программных продуктов [140].

Менеджмент программного проекта. Элементы теории управления и планирования отображены в стандарте PMBOK, содержащем структуру

взаимосвязанных процессов и областей знаний по управлению проектами в любых отраслях, в том числе и программными продуктами. PMBOK определяет процессы ЖЦ проекта (их 39), объединенных в группы процессов и ключевые области знаний, которые типичны для любых проектов (рис.10.10). Группы процессов таковы: инициация, планирование, исполнение, мониторинг и управление, завершение [6–9].

На каждой группе процессов проекта выполняются соответствующие виды деятельности, обозначенные в конкретной области знаний, относящейся к разработке и внедрению изготовленного программного продукта.

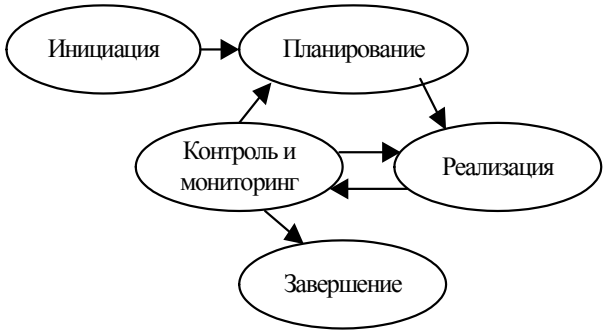


Рис.10.10. Схема взаимосвязей групп процессов в PMBOK

Например, процесс планирования, связаный с организацией управления, базируется на методах принятия решений по ресурсам, задачам проектирования, рискам системы и ресурсов, службам контроля и др. Области знаний PMBOK предлагают принципы управления интеграцией объектов, сроками, стоимостью, качеством, ресурсами, рисками и др.

Области знаний PMBOK определяют управление интеграцией, сроками, стоимостью, качеством, человеческими ресурсами, рисками и др. Дадим им краткую характеристику.

Управление интеграцией проекта – это процессы выявления, определения, комбинирования, унификации и координации всех других процессов и операций по управлению проектами. Интеграция ориентирована на принятие решений о концентрации ресурсов в каждый момент выполнения проекта, о потенциальных проблемах проекта и их решении, поиск компромиссов между пересекающимися целями и альтернативами в проекте. Она включает в себя процессы управления выполнением проекта, мониторинг и управление работами проекта, управление изменениями и др.

Управление содержанием проекта включает в себя процессы, обеспечивающие включение в проект всех тех и только тех работ, которые необходимы для успешного выполнения проекта и соответствующих его содержанию (целям, требованиям, ограничениям и др.). К таким процессам относятся: планирование содержания (верификация и контроль содержания проекта); формирование иерархической структуры работ и разбивки проектных работ проекта на более мелкие, удобные для реализации; управление изменениями в части содержания проекта.

Управление сроками проекта – это разработка графика проекта с учетом последовательностей операций, их длительности, требований к ресурсам и ограничений в сроках, оценка необходимых ресурсов для выполнения операций, оценка длительности операций по количеству периодов, необходимых для выполнения отдельных операций и управления изменениями графика.

Управление стоимостью проекта базируется на процессах, выполняемых в ходе планирования, разработки сметы и контроля затрат, обеспечивающих

завершение проекта в рамках утвержденного бюджета. Управление относится к оценкам стоимости отдельных операций, факторов, вызывающих отклонения от стоимости, изменений бюджета проекта для получения фактического плана по стоимости проекта.

Управление качеством проекта – это процессы и операции, выполняемые организацией-исполнителем, которая определяет цели и степень ответственности сторон по изготовлению качественного проекта. Данное управление осуществляется на основе системы менеджмента качества с помощью правил и процедур, способствующих улучшению процессов выполнения проекта. Процессы управления качеством таковы: планирование качества (выбор стандартов качества и способов их соблюдения); обеспечение качества путем выполнения систематических операций и проверки соответствия установленным требованиям; контроль качества посредством анализа результатов проекта на соответствие принятым стандартам качества и принятия решений по устранению недостатков.

Управление человеческими ресурсами проекта охватывает процессы по организации и управлении командой исполнителей проекта, включая персонал проекта, каждому из которых дана определенная роль и ответственность за выполнение проекта. Процессы управления человеческими ресурсами такие: планирование этих ресурсов и составление плана управления обеспечения проекта персоналом; привлечение ресурсов, в основном людских, для выполнения проекта; повышение квалификации членов команды проекта и поддержка взаимодействия между ними; контроль за эффективностью работы команды проекта посредством решения проблем и задач координации изменений.

Управление рисками проекта – это деятельность, которая заключается в планировании и управления рисками, в их идентификации и анализе, реагировании на риски, мониторинге и управлении рисками проекта. Цель управления рисками проекта – повышение вероятности возникновения и степени влияния позитивных событий и снижение вероятности возникновения и степени влияния негативных для проекта событий. Данная деятельность определена соответствующими процессами.

Управление поставками проекта состоит в выполнении процессов закупки или приобретения необходимых продуктов, услуг или результатов, которые производятся вне команды проекта. Согласно условиям контракта организация может выступать в качестве продавца или покупателя продукта, услуги или результатов. Данное управление основывается на планировании закупок и приобретений, контрактов на продукты, услуг и результатов, а также на контроле хода выполнения контрактов и при необходимости на внесение своевременных изменений.

Для применения основных положений стандарта PMBOK необходимы методики и руководства, среди которых имеется стандартное руководство (ISO/IEC TR 16326:1999 «Guide for the application of ISO/IEC 12207 to project management») по применению стандарта ISO/IEC 12207 при управлении программными проектами. Этим руководством обеспечивается применимость положений PMBOK в ПИ.

Методы управления проектами. В настоящее время сформировалось несколько методов, эффективно применяемых в практике реализации программных проектов [140]. Рассмотрим их.

Метод критического пути CPM (Critical Path Method) разработан Уолкером – Келли для управления проектом. Критический путь – наиболее длинный и полный путь в сети. Работы, которые лежат на этом пути, также называются критическими. Именно продолжительность критического пути определяет наименьшую общую продолжительность работ в проекте в целом.

Метод основан на графическом представлении задач (работ) и видов действий на проекте и заданием ориентировочного времени их выполнения. Это представление задается в виде графа (рис 10.11), в вершинах которого располагаются работы и время выполнения каждой работы под вершинами либо на дугах графа.

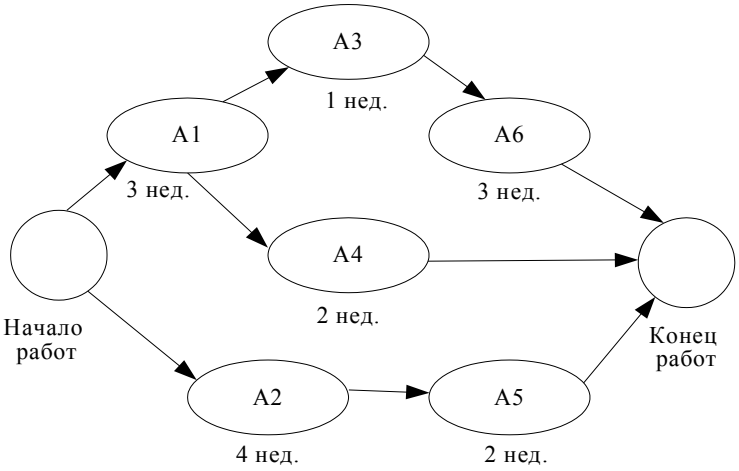


Рис.10.11. Граф со сроками выполнения работ

Граф строиться тогда, когда работы и время их выполнения являются заданными (определенными).

Критический путь в графе указывает максимальную продолжительность работ на графе (от начальной работы до последней). При выполнении проекта выбираются и выполняются работы, которые не влияют на время выполнения других (независимых) работ проекта или на их продолжительность. Работы на критическом пути могут сокращаться за счет изменения времени выполнения.

Время выполнения всего проекта может быть сокращено за счет уменьшения времени выполнения задач, которые лежат на критическом пути. Таким образом, любая задержка выполнения задач критического пути приводит к увеличению времени выполнения проекта. Эта концепция концентрирует внимание менеджера на критических работах.

Основным преимуществом метода критического пути является возможность управления сроками выполнения задач, которые не лежат на критическом пути. Этот метод разрешает рассчитать возможные календарные графики выполнения комплекса работ на основе описанной логической структуры сети и оценок времени выполнения каждой работы.

Представление работ в таком виде называется *сетевой диаграммой* и служит для графического отображение работ проекта, их взаимосвязей, последовательностей и времени выполнения. В графе вершины отображают работы, а

линии взаимные связи между работами. Такой граф является наиболее распространенным представлением сети на сегодняшний день.

Метод анализа и оценки PERT основан на сетевых диаграммах в виде вершин–событий, а работы – на линии между двумя событиями, отображающими начало и конец данной работы (рис.10.12).

Дуге, выходящей из начальной вершины и входящей в заключительную вершину, соответствует временная метка 0. С помощью этих меток задается время выполнения процесса. В графе могут присутствовать также и циклические пути.

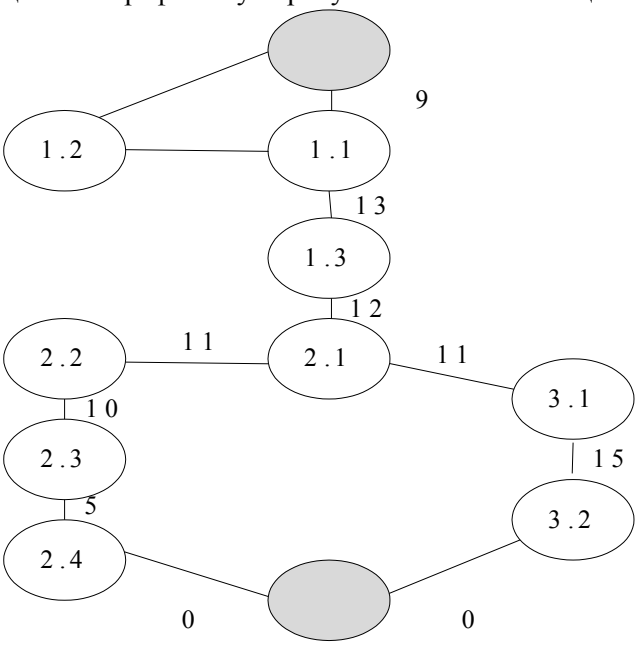


Рис.10.12. Граф номеров работ и сроков (на дугах)

На основе графа проводится анализ критических путей в выполнении работ, т.е. определяется продолжительность каждого процесса. План проекта создается в сроках этапов: планирование, проектирование, кодирование, тестирование и сопровождение. Для первых двух методов планирование затрагивает: определение спецификаций, бюджета и расписания, а также развития плана проекта в целом. Представление более сложных связей между работами для задания узлами графа в виде вершина–событие является более сложным и потому этот метод реже используется на практике.

Между этими двумя методами сетевого представления графа работ имеются незначительные расхождения. Данный метод, например, в отличие от CMP, учитывает возникающие неопределенности во времени выполнения каждой операции.

Современные средства управления проектами обеспечивают визуализацию структуры графа проекта и процессов выполнения работ. Это дает возможность наблюдать виды деятельности, которые могут выполняться последовательно или параллельно, или находиться на критическом пути (например, в системе Project Management).

Таким образом, рассмотрены базовые стандартные положения PMBOK, серия

стандартов ISO 9001–9003, регламентирующие управление качеством проекта, теория управления и планирования, а также методы управления проектами – базис дисциплины управления в программной инженерии. Применение теории управления обеспечит стратегическое планирование работ по производству ПС проектов и эффективную подготовку высококвалифицированных менеджеров проектов, а также других специалистов в области организационного управления по изготовлению и выпуску ПП на производственной основе.

10.3.4. ЭКОНОМИКА ИЗМЕРЕНИЯ И ОЦЕНКИ ПРОГРАММНЫХ ПРОДУКТОВ

Экономика ПИ – это самостоятельная дисциплина, определяющая экономические аспекты индустрии ПП. Она включает в себя методы прогнозирования общего объема работ и требуемых ресурсов для выполнения работ, составления планов и распределение специалистов по этим работам, экспертизы результатов выполнения работ, а также определение стоимости и производимых затрат на разработку и сопровождение изготавливаемого продукта. Сюда относятся методы анализа ПрО для построения ее модели, проектирования архитектуры будущей системы и программирования отдельных компонентов.

Иными словами, в ее основе лежат экономические расчеты разных сторон деятельности исполнителей проекта с учетом знаний всех экономических факторов и текущих затрат в проекте. Эта дисциплина имеет свою теорию и практику решения задач по проведению экспертизы проекта, оценке стоимости, сроков и экономических показателей, устанавливаемых в требованиях к ПП при заключении контракта на его разработку. В рамках этой дисциплины проводится оценка требований, проектных решений, архитектуры, рисков разработки, связанных с имеющимися материальными и человеческими ресурсами, показателями качества ПП, а также оценка финансовых расчетов на всех процессах выполняемых договоров с каждым исполнителем [6–9, 52, 54, 133, 140, 141].

Техника измерения ПП изложена в ряде стандартов (ISO/IEC 14143, 19761, 20926 и др.), а оценивания ПП отражена в ряде методов, некоторые из которых стандартизованы. К методом экономических расчетов относятся: прогнозирование размера ПП (Function Points Analyses, Feature Points, Mark-H Function Points, 3D Function Points и др.), оценки трудозатрат на разработку ПП с помощью семейства моделей COCOMO, набор математических моделей оценки трудозатрат на разработку ПП (Angel, Slim, Seer-SEM и др.), а также интегрированная модель, объединяющая экономические показатели жизнеспособности и качества ПС [6, 190, 141]. Рассмотрим некоторые из них.

Метод Function Points (FP) предназначен для определения элементарных функций ПС и их внутренних и внешних информационных объектов. Экспертная оценка сложности функций и объектов данных с помощью шкалы (низкая, средняя, высокая) зависит от количества структурных элементов каждого объекта данных и ссылок функций на эти объекты.. Каждому типа элемента объекта и уровню сложности ставят в соответствие весовой коэффициент. Рассчитаем показатель функциональности объектов – УОФ (отдельно для данных и функций), результаты просуммируем по формуле:

$$УОФ = УОФ_а + УОФ_ф = \sum_{i=1}^{m}\sum_{j=1}^{3} So_i \cdot Wo_j + \sum_{i=1}^{n}\sum_{j=1}^{3} Sф_i \cdot Wф_j ,$$

где *So* – количество внешних и внутренних информационных объектов, *Wo* – весовые коэффициенты сложности; *Sф* – количество функций ввода, вывода;, *Wф* – соответствующие весовые коэффициенты сложности.

Метод Mark-II Function Points основан на выделении одного (а не пяти, как в FP-методе) базового логического элемента – логической транзакции [231]. Функциональный размер ПС – это сумма размеров всех логических транзакций. Каждая из них включает в себя: исходные данные и функцию обработки сохраненных в ПС данных (например, в реляционных таблицах). Сложность транзакций устанавливается экспертным путем. Метод применяется в системах обработки данных, работающих с большими объемами данных.

Метод 3D Function Points (3D FP) основан на предположении, что сложность любой ПС рассматривается в трех измерениях – сложность *данных*, *функций* и *управления* [241]. Для получения общего размера ПС экспертным путем оценивают уровень сложности одних-двух характеристик по каждому измерению. Для оценки сложности данных (*СкД*) непосредственно используют FP-метод. Для оценки сложности функций (*СкФ*) устанавливают число *трансформаций* (*Nтранс*) – совокупность шагов процесса обработка данных (т.е. число операций, которые преобразуют вход в выход) и семантических утверждений, которые руководят выполнением операций. Каждая трансформация «решается» на трех уровнях сложности (низкая, средняя и высокая). Для оценки сложности управления (*СкУ*) определяют множество уникальных состояний (*Nc*), в которых может находиться процесс, а также множество переходов (*Nn*) между состояниями. Значения *СкУ* определяется по формуле: $СкУ = Nc + Nn$.

Расчетная формула для функционального размера имеет вид:

$$УОФ = СкД + СкФ + СкУ$$

Метод измерения функционального размера представляет собой развитие FP-метода в плане улучшения точности оценок функционального размера сложных ПС, который использует базовые расчетные элементы FP–метода (внутренние и внешние объекты данных, функции ввода, вывода, запроса) и функциональный элемент – алгоритм. Сложность объектов, функций и алгоритмов устанавливается экспертным путем, как и в методе FP. Метод применяется в системах реального времени, ОС, системах управления процессами и др. Для учета сложности метод предлагает оценку трех общесистемных характеристик на уровнях сложности от первого до пятого и по суммарному уровню сложности характеристик определение общего функционального размера ПС: $FeP = Ксложс \cdot ПФР$.

Анализ методов оценивания функционального размера ПС показал, что унифицированную оценку объема функциональных возможностей ПС в так называемых Functional Points (FP) – условных единицах функциональности – на протяжении всего ЖЦ ПС предоставляет методология Function Points Analysis (FPA) [190, 229] на основе требований к ПС и сложности реализации функций обработки данных.

Метод Object Points for ICASE использует для оценки объекты типа – *экранные формы*, *формы отчетов* и *модули.* Метод применяют для расчета функционального

размера [6]. Для этого определяют число объектов каждого вида и экспертным путем для каждого из них устанавливают уровень сложности (простой, средний и сложный) в зависимости от количества полей в формах и количества и типа источников данных (на сервере / клиенте). Затем определяют относительный вес элементов в зависимости от уровня сложности. Функциональный размер вычисляют как сумму полученных значений по всем объектам.

Оценка затрат. Среди основных категорий методов оценки затрат наиболее широкое применение нашли методы экспертных оценок и методы параметрических уравнений [141, 257]. Основным преимуществом методов первой категории является учет опыта предыдущих разработок, а недостатком большая зависимость от компетентности экспертов. Преимущества методов второй категории – возможность повторения оценок (выполнения расчетов по моделям), простота модификации входных данных и корректировки формул с учетом условий разработки проектов.

Наиболее известные модели – *композиционные модели семейства COCOMO* (табл.10.2).

Таблица 10.2

Название модели	Назначение модели
COCOMO II – Constructive Cost Model	Полная трехуровневая модель оценки затрат для новых проектов ПС
COINCOMO – Constructive Incremental COCOMO	Учет и распределение затрат при итеративной разработке с интеграцией оценок для проекта
COPROMO – Constructive Productivity-Improvement Model	Оценка эффективности вложения ресурсов в новые технологии для улучшения производительности.
COPLIMO – Constructive Product Line Investment Model	Поддержка оценивания стоимости линии продуктов и анализ отдачи от инвестиций
COPSEMO – Constructive Phased Schedule & Effort Model	Распределение оценок затрат на процессах разработки. Применяется совместно с CORADMO
CORADMO – Constructive Rapid Application Development Model	Оценивание и распределение затрат для небольших проектов, которые разрабатываются согласно модели быстрой разработки (RAD)
COCOTS – Constructive Commercial-Off-The-Shelf Cost Model	Оценка затрат, связанных с выбором и интеграцией готовых (COTS) программных продуктов в ПС
COSOSIMO – Constructive System Systems Integration Cost Model	Оценка затрат, связанных с определением и интеграцией крупномасштабных интегрированных распределенных ПС
WEBMO – Cost Model for Web Application	Оценка трудозатрат и продолжительности разработки веб-приложений

В данных моделях используются экспертные и алгоритмические методы оценки [6–9, 29]. Их основные отличия – это учет разных видов работ, факторов, которые влияют на затраты, а также распределение затрат по процессам и видам работ в зависимости от модели ЖЦ.

Для расчета трудоемкости разработки ПС по моделям семейства COCOMO сначала вычисляется функциональный размер ПС в FP, который конвертируется в KSLOC.

Оценки трудозатрат Tn определяются по формуле

$$T_n = \prod_{i=1}^{7}(K_i) \cdot A \cdot V^B \ ,$$

где $\prod_{i=1}^{7}(K_i)$ – произведение коэффициентов стоимостных атрибутов, оценки которых могут быть получены экспертным путем на ранних процессах разработки ПС; $A = 2.45$ константа, полученная по результатам статистического анализа фактических данных для 80 реальных проектов; V – предвиденный размер ПС или его компонента в KSLOC

$$B = 0.91 + 0.01 \cdot \sum_{j=1}^{5} \Phi_j \ ;$$

Фj – значение соответствующих коэффициентов масштаба, которые характеризуют сложность и условия разработки ПС. Равенство их рейтингов устанавливаются экспертным путем.

Оценка рисков. Экспертным путем может оцениваться риск проекта разработка ПС – SRE (Software Risk Evaluation), рекомендованный специалистами SEI [6– 9, 141, 230]. Он регламентирует выполнение процесса идентификации, анализа и разработки стратегий ослабления рисков создания ПС. Процесс SRE включает пять шагов:

– идентификация целей проекта, достижения договоренностей относительно SRE и координация ресурсов на проведение оценивания по SRE;

– оценка по методу SRE путем опроса участников проекта в целях формулирования утверждений о рисках, их анализ, упорядочение по приоритетам в контексте последствий для проекта и группировка рисков по областям рисков;

– повторный анализ области рисков, описание рекомендаций по тем рискам, которые должны быть охвачены при планировании ослабления рисков и согласование их с менеджером проекта;

– определение целей, стратегий и действий, связанных с рисками, создание планов верхнего уровня по ослаблению рисков в выбранном подмножестве областей рисков, а также обсуждение с участниками проекта, руководства и командой оценщиков;

– формирование заключительного отчета по информации о рисках в проекте.

Для получения экспертных оценок участников проекта относительно рисков используется вопросник, основанный на таксономии риска TBQ (Taxonomy Based Questionnaire), которая охватывает области риска в категориях, связанных с продуктом, процессом и источниками ограничений проекта.

Одной из сфер применения экспертных методологий в программной инженерии является оценка зрелости. В настоящее время используется много моделей зрелости организации-разработчика ПС [6, 7], среди которых наиболее популярные модели семейства CMMs (Capability Maturity Models).

Семейство моделей CMM включает: SW-CMM (CMM for Software) – модель зрелости процесса разработки программных продуктов; SE-CMM (System Engineering CMM) – модель зрелости процесса системной инженерии; SSE-CMM (System Security Engineering CMM) – модель зрелости процесса обеспечения безопасности системы; SA-CMM (Software Acquisition CMM) – модель зрелости

процесса закупок ПО; People CMM – модель зрелости процесса управления кадрами.

Модель SW-CMM [5-10] (CMM for Software) идентифицирует 5 уровней зрелости организации и распределенных 18 обязательных ключевых направлений деятельности KPA (Key Process Area). В структуре описания каждого KPA выделен пять разделов, каждый раздел определяет перечень рекомендованных практических действий. Один из разделов KPA – предназначен для описания требований к проведению измерений и анализа их результатов в деятельности организации. Таким образом, алгоритм установления уровня зрелости (от 2 до 5) на основе анализа результатов измерений CMM пока отсутствует. Он находится на стадии разработки.

Метод оценки стоимости. Оценка стоимости затрат на разработку ПС можно осуществить, используя семейство моделей COCOMO [6–8, 29], отличающихся друг от друга элементами, определяющими количество дефектов, отдачу от инвестиций, затраты на стадиях разработки, интеграции составляющих системы и др. Эта модель объединила три техники оценки проекта. В первой модели COCOMO1 используются показатели цены, персонал и свойства проекта, продукта и среды. Эти показатели определяются на трех стадиях ведения проекта. На первой стадии строится прототип системы для задач повышенного риска и проводится оценка затрат (например, число таблиц в БД, экраны и отчетные формы др.). На второй стадии оцениваются затраты на проектирование и реализацию функциональных точек проекта согласно требованиям к проекту. На третьей стадии проводится оценка размера системы в готовых строках программ и с учетом других факторов.

Для этой оценки применяют следующее уравнение: $E=bS^c\ m(X)$, где первичная оценка bS^c корректируется с помощью вектора стоимости m(X) и учета числа старых и новых объектов. Параметр c изменяется от 0 до 1.0 для первой стадии и от 1.01 до 1.26 для остальных стадий.

Стоимость проекта можно определить по формуле

$$E = (a+bS^c)\ m\ (X),$$

где S – оценка размера системы, $a, в, c$ – эмпирические константы, $X[n]$ – вектор факторов стоимости, m – регулирующий множитель, основанный на затратных факторах.

В работе [29] предложена модель оценки, полученная экспериментальным путем: $E = 5.25S^{0.91}$. Эта модель применялась при оценке проекта, в котором программы имели размер от 4000 до 467000 строк кода и были написаны на 28 различных ЯП для 66 компьютеров. На разработку было затрачено от 12 до 11758 человеко–месяцев. В [8] предложено такое уравнение затрат организации–разработчика: $E = 5.5+0.73S^{1.16}$.

Таким образом, сущность экономической дисциплины определяется фундаментальными методами экспертной и количественной оценок экономических показателей продукта и процессов. Оценка зависит от принципов распределения и проведения экспертиз в сложных системах, а также от методов расчета стоимости отдельных частей систем с учетом их размера, существующих стандартов по измерению и оценке готового продукта.

Иными словами, систематическое использование методов экономических расчетов и экспертиз продуктов и процессов – основополагающие элементы

экономики в ПИ. Такие методы должны использоваться в индустриальном цикле производства продуктов и процессов.

10.3.5. ОСНОВНЫЕ АСПЕКТЫ ИНДУСТРИИ ПРОГРАММНЫХ ПРОДУКТОВ

Главным вопросом индустрии, как таковой, является не только изготовление и выпуск программной продукции для пользователей, а и получение прибыли. В области ПИ продукты массового производства, создаваемые известными фирмами Microsoft, IBM, Intel и др., а также результаты аутсорсинга (обновление устаревшего, унаследованного ПО), приносят владельцам большие прибыли. Этим подтверждается (в соответствии с толкованием понятия «производство»), что виды ПП этих фирм выпускаются на индустриальной основе. Производство ПП базируется на технологических процессах изготовления определенных видов продуктов с применением теории проектирования и инструментальных сред поддержки выпуска ПП.

Первые образцы индустриального производства – технологическая подготовка разработки ПП (ТПР) [119–121], линия продукта (Product line) Института программной инженерии (SEI) США [224], обеспечивающая удовлетворение рыночных потребностей пользователей на некоторые виды программной продукции. К передовым инженерным технологиям производства ПП относятся инженерия приложений, доменов, семейств систем, средства поддержки их производства (ОС, общесистемные средства, новые языки, интегральные среды и т.п.), а также современные инструментальные системы и среды (Microsoft Visual Studio System Teams, MSF и др.), которые поддерживают весь производственный цикл работ по изготовлению ПП.

ТПР применялась при разработке АІС ”Юпитер” для производства программ обработки данных на нескольких объектах этой системы. В последние годы ТПР не развивалась из-за отсутствия такого рода систем. Производство ПП на упомянутой линии продуктов осуществляется из готовых программ, информационных ресурсов, ПИК, средств и инструментов по ТЛ, в которую включаются необходимые методы разработки, тестирования и оценивания конечного результата. Технология конструирования ПП на ТЛ выполняется на основе каркаса ПП и применения готовых КПИ. Инструментальная среда их разработки должна содержать необходимые средства и инструменты производства, а также механизмы отслеживания процесса изготовления продукта в соответствии с установленным заказчиком требованиями и планом.

Для производства программных продуктов по ТЛ требуется инженерная, технологическая и экономическая дисциплины для нормативного и регламентированного проведения работ как это происходит в любой производственной деятельности. Эту организационно-управленческую работу выполняют такие специалисты, как менеджеры, эксперты, оценщики и др. В их функции входит управление, проведение экспертиз, измерение и оценивание разных видов объектов на ТЛ.

Формальное описание дисциплин производства ПП. В предметной области ПИ выделены дисциплины (*Di*), которые обеспечивают основные аспекты организационно-управленческой деятельности организации-разработчика ПП:

$$ПИ = \{DiSc, DiEn, DiEc, DsMa\},$$

где *DiSi* – научная, *DiEn* – инженерная, *DiEc* – экономичная, *DiMa* – управленческая дисциплины, которыми должны владеть участники процессов производства ПП (аналитики – Sciences, инженеры – Engineers, экономисты – Economics, управленцы – Managers).

Производственную деятельность (*PA* – product activity) участников процесса производства ПП зададим кортежем:

$$PA = < TЛ, Ap, Rp >,$$

где $TЛ = \{Tp \cup CP\}$, *Tp* – учебный процесс и процесс управления *CP* (Control process); $Ap = \{ DiSi \cup DiEn \cup DiEc \cup DiMa\}$ – подобласти (дисциплин) ПрО; *Rp* – квалификационные требования к участникам ТЛ при производстве ПП;

Сопоставление нормативно-методического обеспечения процессов ТЛ и принципов стратегического управления [179] позволяет их рассматривать единообразно как процесс управления изготовлением ПП согласно целевым требованиям и подходов к усовершенствованию. При этом предполагается, что на ТЛ будут многократно выполняться оценочные действия для установления значений актуальных критериев достижения заданных целей (или целевых характеристик) для управляемых (контролируемых) объектов и элементов деятельности по управлению ими (называемых оцениваемыми объектами ПрО).

Инженерия индустриального производства программных продуктов

К настоящему времени разработано много подходов и методов производства ПП [46, 105, 109, 205, 209]. Они сыграли важную роль в формировании производственного характера изготовления ПП. На смену пришли усовершенствованные или новые методы и технологии, которые соответствуют производству ПП. Например, ряд зарубежных фирм занимаются выпуском программной продукции, которая основывается на технологических и инструментальных инструментах и средствах автоматизированного производства программных продуктов. К условиям относят средства и инструменты, процессы и технологические линии изготовления продуктов. Фирмы–изготовители проводят мониторинг для определения запроса на выпуск некоторого вида программной продукции.

Для организации производства готовятся несколько команд разработчиков. Первая из них готовит базовый процесс производства, настройку информационного, технологического и технического состояний среды для изготовления продукта. Вторая команда – это разработчики ПС. Третья команда – это служба контроля хода разработки и оценки объектов производства: простых, готовых КПИ и сложных, создаваемые из них продуктов.

Известными средами с такой организацией изготовления ПП являются линии производства (Product Line Practice), система Microsoft Visual Studio Teams Systems, Microsoft MSF, CORBA и т.п. [82, 133, 140, 241]. Рассмотрим их детальнее.

Технология изготовления ПП в системе Microsoft

Основное назначение данной технологии – управление проектом предприятия EPM (Enterprise Project Management). Основные инструменты технологии такие:

– пакет инструментов VSTS (Visual Studio Teams Systems) для разработки

больших проектов с участием разных видов специалистов (аналитиков, менеджеров, тестировщиков, программистов, кодировщиков и др.), которые могут выполнять работы, находясь в разных географических точках;

– методология MSF (Microsoft Solution Arhitecture) для построения производственной архитектуры предприятия с помощью ЖЦ (Software Development Life Cycle), стандарта PMBOK, моделей перспектив и процессов [133, 140, 219] ;

– система Professional Studio и Foundation Server для поддержки процессов проектирования, кодирования, тестирования, формирования версий ПП и т.п.;

– система CMMI Process Impovement для регулирования сроками разработки с помощью технологии Agile.

Пакет VSTS – это семейство продуктов для проектирования ПС, в котором разные члены команды распределены на разных работах проекта в зависимости от их квалификации и категории (рис.10.13).

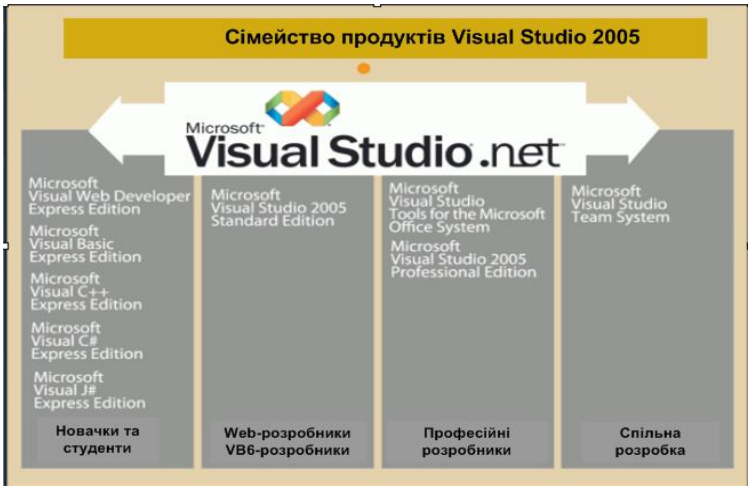


Рис.10.13. Категории разработчиков в пакете VSTS

Согласно приведенной схемы переход исполнителей в более высшую категорию возможен по мере роста его знаний и выполненных работ.

Инструментами разработки разных программных продуктов являются: IDE (Integrated Development Environment), Microsoft Office Project и Microsoft Office Excel, а также:

– *Microsoft SQL Server 2005* для ведения результатов, исходного кода и данных через инструменты *Analysis и Reporting Services* и *SharePoint Portal Services*;

– *Microsoft Project 2003* и *Microsoft Excel 2003* для управления и планирования работ в проекте.

Таким образом, среда VSTS служит примером коллективного изготовления ПП. В ней имеются инструменты для поддержки процессов ЖЦ с учетом плановых работ согласно графику, отслеживания хода разработки, оценки качества и сроков выполнения проекта.

Средства и инструменты методологии MSF

Методология MSF – это система стратегий, принципов и методов управления проектом производственной архитектуры предприятия. с учетом заданных

объемов, времени и стоимости, а также наличия необходимого персонала, контрактов и т.п. [205]. В ее состав входят модели:

– производственной архитектуры;
– проектной группы;
– процесса разработки ПС;
– управления рисками;
– процесса проектирования;
– применения.

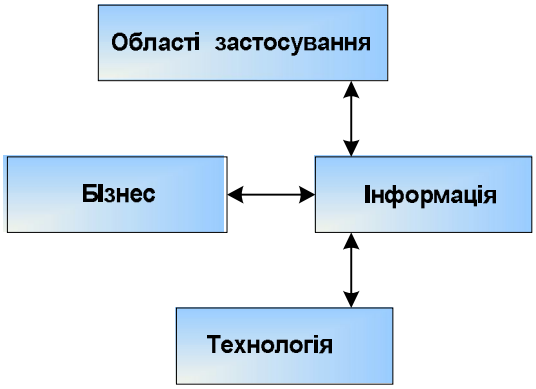


Рис.10.13.Перспективы производственной архитектуры MSF

Модель производственной архитектуры предприятия разрабатывается исходя из следующих аспектов: область применения, бизнес, информация и технология (рис.10.13). Создается план реализации архитектуры, который учитывает приоритет отдельных аспектов модели, предоставляемые ресурсы (технические, человеческие, программные и информационные) для реализации архитектуры, баланс целей в соответствии с заданными требованиями.

Модель производственной архитектуры – это набор принципов для создания версии производственной архитектуры предприятия.

Бизнес-перспектива определяет стратегии и планы перехода к улучшенному состоянию предприятия, учитывающего цели и задачи организации; виды продуктов и услуг; бизнес-процессы реализации основных функций.

Информационная перспектива базируется на возможностях автоматизировать бизнеса-задачи с использованием общесистемных средств, сетей, документов и таблиц БД.

Технологическая перспектива основана на регламентации деятельности разработчиков, участвующих в реализации архитектуры.

Модель проектной группы определяет роли, обязанности каждого участника проекта и распределение между ними ответственности с учетом целей проекта и их квалификации.

Модель процесса разработки ПС – это набор процессов, видов деятельности и результатов процесса разработки ПС. Эта модель связана с проектной группой в целях проведения контроля хода разработки проекта, минимизации рисков, повышения качества и сокращения сроков выполнения проекта.

Члены проектной команды в процессе разработки создают: код системы, конфигурацию, функциональную спецификацию и сценарии тестирования. Они также создают инфраструктуру и документ на конфигурацию.

Модель процесса проектирования определяет цель и задачи процесса разработки производственной архитектуры. *Модель применения* – это трехуровневая структура, отображающая выполнение работ в проекте в наглядном виде.

Таким образом, методология MSF дает возможность проектировать

программное и информационное обеспечения предприятия, сокращать время их разработки, укладываться в заданную стоимость и повышать качество ПП при использовании модели *MSF for CMMI Process Improvement*.

Суть производства продукции

Производство – процесс создания материальных благ, необходимых для удовлетворения индивидуальных и общественных потребностей существования и развития общества.

Экономика и организация производства вместе образуют дисциплину, ориентированную на обеспечение деятельности организации, в том числе и программостроительного типа, по изготовлению разных видов продукции, необходимой определенным слоям общества.

Условием выполнения основной задачи производства любого вида продукции есть соблюдение научно обоснованных принципов организации производства и управления, а именно:

– специализация (спецификация) номенклатуры изготовляемой продукции в целях повышения производства;

– согласованность пропускной способности компьютеров и целей разных производственных подразделений (служб контроля, тестирования, оценивания результатов труда и др.) для обеспечения процессов производства продукции из выбранного класса;

– наличие необходимых человеческих, технических и программных (готовых компонентов, КПИ и т.п.) ресурсов для изготовления программных продуктов.

Основные принципы организации производственного процесса:

– экономичность, т.е. достижение возможных результатов производства с минимальными затратами;

– системность, т.е. взаимодействие всех служб организации–изготовителя программной продукции как частей технико-экономической (комплексной) системы с отрегулированным порядком выполнения работ согласно их планам и срокам;

– пропорциональность, как согласованность и заинтересованность каждого исполнителя выполнять работы ответственно и в срок с необходимым контролем отдельных результатов труда.

Процесс производства программной продукции начинает набирать обороты. Действуют отдельные организации и фирмы, выпускающие такую продукцию по собственной технологии. В перспективе предстоит еще много сделать, чтобы организации-изготовители создавали программную продукцию массового употребления, используя современные наработки в организации технологического производства.

В рамках данной дисциплины ПИ еще не решены проблемы снижения сложности продуктов и процессов для их изготовления на промышленной основе. Отсутствуют публикации, в которых предлагаются эффективные способы преодоления сложности, особенно при интеграции больших программных проектов из разных простых, КПИ и готовых программных ресурсов. Существуют проблемы и в организации работ в оффшорных фирмах по эволюции унаследованных программных продуктов, имеющих спрос на рынке.

Данная дисциплина сложится как самостоятельная с регламентированными

процессами производства программной продукции разного назначения. Она будет базироваться на классических методах управления, технологии производства разных видов технических продуктов, методах формализованного описания специфических особенностей целевых объектов и их реализации, методах оценки готовых ресурсов, процессов и инструментальных сред, а также новых стандартах, регламентирующих производственную деятельность по изготовлению программной продукции.

10.4. РОЛЬ И МЕСТО ДИСЦИПЛИН ПРОГРАММНОЙ ИНЖЕНЕРИИ В ИНФОРМАТИКЕ

Сегодня вопросы создания информационных систем и технологий решаются методами и средствами программной инженерии. Она занимает ведущую роль и в других областях разработки компьютерных систем.

Программная инженерия – составная часть CS (Computer science) компьютерной науки. CS – наука о компьютерах разного назначения (общего, специального, проблемного и пр.), теории их построения, управления компьютерными системами обработки информации, методах проектирования Hardware (HW) и Software (SW) этих систем. CS объединяет научные дисциплины, включая теорию дискретных систем, алгоритмов, автоматов, моделирования, управления, математику, логику и пр. В ней по сравнению с кибернетикой, важное место занимают новая теория микросхем различных установок в пределах EE (Electronic Engineering), компьютерной инженерии CE (Computer Engineering) и программной инженерии SE (Software Engineering). С учетом этих теорий и практических применений компьютерных систем возникли информационные системы (ИС) обработки больших объемов информации, технологии информационной поддержки потоков данных и их обработки на современных компьютерах [205]. Появление их (с 1960г.) инициировано также потребностями бизнеса (например, системы бухгалтерского учета, заработной платы, складского аудита и пр.). ИС и информационные технологии (ИТ) с конца 1990 г.г. стали составной частью CS, как средства поддержки продуктивности и эффективности организаций, работающих с компьютерными системами и информационными магистральными потоками информации.

Связь между всеми порожденными CS дисциплинами показаны на рис.10.14.

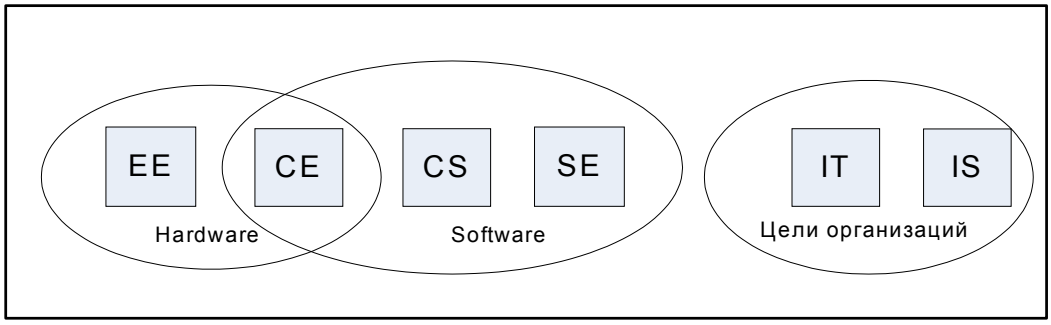


Рис.10.14. Структура связей языков дисциплин CS

Здесь основа CS – это теория построения Hardware компьютеров (фреймворки,

кластеры, макро- и микрокомпьютеры и пр.), Software (ОС, преобразователи форматов данных и т.п.), а также компьютерных прикладных систем, автоматизирующих различные отрасли науки (вычислительной математики, математической физики, биологии, медицины), предприятий (АСУ, АСУТП, СОД и пр.), ИС и интеллектуальных (искусственного интеллекта) систем.

Иными словами, компьютерная наука является теорией построения цифровых компьютеров (Digital computer) для поддержки информационных процессов и систем обработки информации и компьютерных систем (Computer systems) для решения различных классов задач с помощью современных компьютеров.

Главные направления CS отображены в соответствующей рабочей программе подготовки специалистов образовательно-квалификационного уровня бакалавра (Постановление Кабинета Министров Украины от 27 января 2007 г. № 58) Министерства науки и образования Украины в области таких знаний, как информатика, вычислительная математика и автоматизированные системы управления:

– компьютерная наука как специальность предназначена для подготовки специалистов по эксплуатации систем обработки информации и принятия решений (500406);

– компьютерная инженерия как специальность предназначена для подготовки специалистов по обслуживанию компьютеров и интеллектуальных сетей (500404);

– системная инженерия как специальность предназначена для подготовки специалистов по обслуживанию систем управления и автоматики (5091404);

– программная инженерия как специальность предназначена для подготовки специалистов для вычислительной техники и автоматизированных систем (5091405).

Ниже представлена структура компьютерной науки (рис.10.15) и дана общая характеристика её дисциплин, основное содержание и особенности каждой из них.

Компьютерная инженерия – это дисциплина по теории и принципам построения компьютеров (frameworks, микросхем, микропроцессов, кластеров, суперкомпьютеров и пр.), системного обеспечения (ОС, трансляторов, компиляторов и т.п.), информационных процессов систем и связей между их объектами. Она связана с оптимизацией наборов операций для вычислительных моделей компьютеров и механизмов их контроля при построении Hardware и Software. Эта инженерия базируется на исследованиях и разработках принципов, теории и методов построения концептуальных Фреймворков с использованием математики и логики, а также на теории сложности, анализа систем, теории автоматов, компьютерной генетики, лингвистики и пр.

Принципы построения Software включают общие возможности и свойства ОС, СУБД, трансляторы, интерпретаторы и т.п. Компьютерные архитектуры включают в себя: процессоры, многопроцессорные (Pentium, Intel, Скит2 и др.), рекурсивные процессоры, различные форматы данных и соответствующие преобразователи, а также интеграцию приборов, блоков, карт, кабелей и пр. Основа теории этого направления – теория автоматов, алгоритмов, машины Тьюринга, Неймана и т.п.

Системная инженерия – это теория, методы и принципы построения систем обработки информации, информационных систем и автоматизированного управления ими, которые базируются на структурах компьютерных систем

(Computer Systems), принципах их использования, методах управления и обработки различных классов задач с теоретическим обоснованием их свойств и ограничений относительно объемов обработки информации. Теория архитектуры, процессы и память компьютеров – базис для моделирования различных типов средств обработки информации, компьютерных приложений (Computer Applications) и новых систем управления информационными системами. Мост между компьютерной наукой и различными компьютерными применениями – это спецификации архитектуры компьютеров и систем ПО (ОС, БД, СУБД, трансляторы и пр.), определяющие действия информационных процессов.

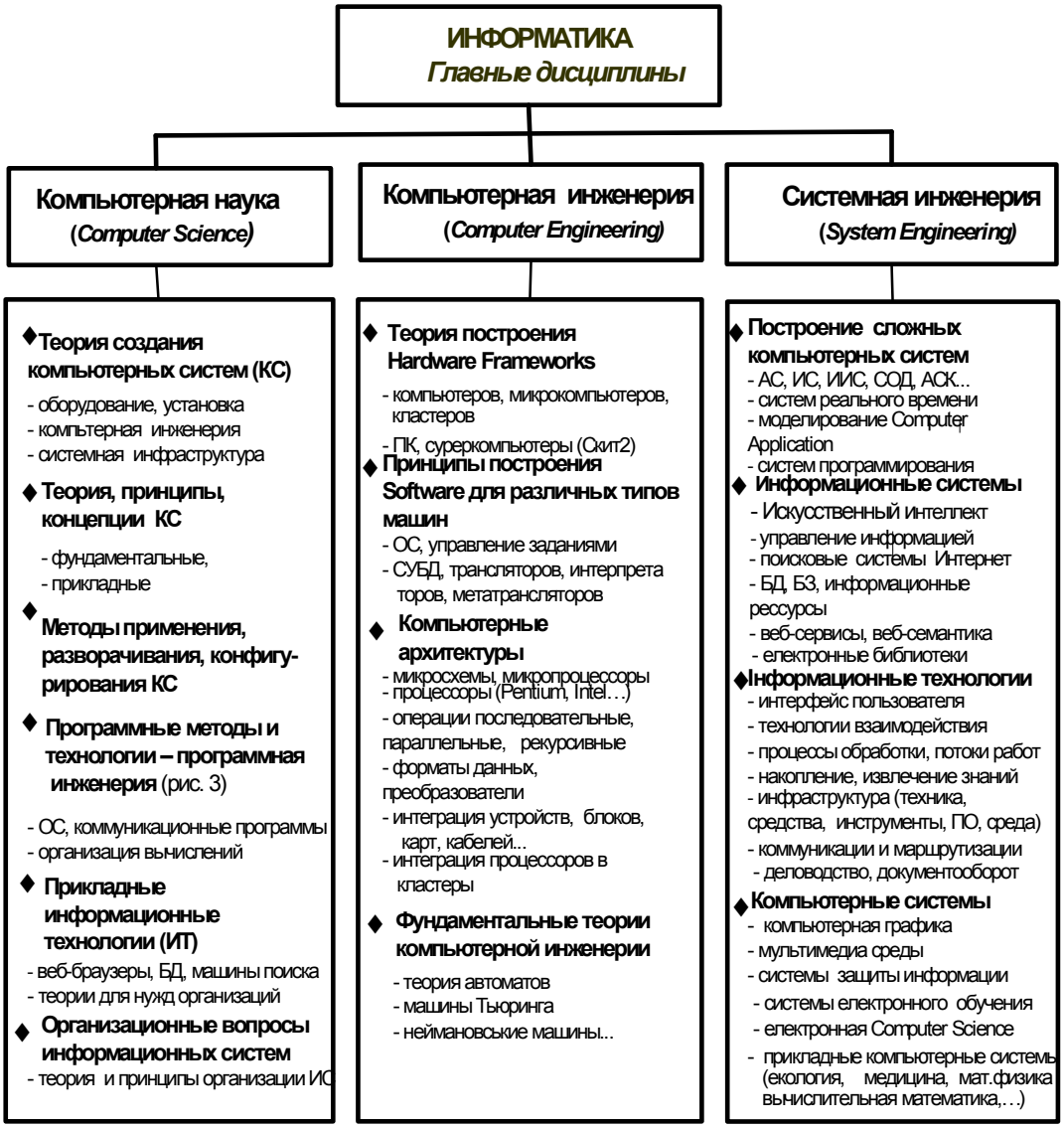


Рис.10.15. Состав дисциплин информатики и их подразделы

К средствам их поддержки относятся логика, математика, электронная и программная инженерия, а также интеллектуальные дисциплины (комбинаторика, графика, искусственный интеллект и пр.). В новых комплексных компьютерных и

информационных технологиях нашли отображения новые методы в экономике, финансовой, банковской деятельности, а также в различных сферах общественной жизни. Новые компьютеры и кластеры способствуют улучшению и упрощению сложных задач систем обработки информации.

Программная инженерия – это наука построения качественных компьютерных программных систем конвейерным способом производства с использованием накопленных готовых ресурсов и инженерных методов их сборки на технологических линиях. Особенность производства новых систем – анализ предметной области, описание требований и её специфики на специальном языке DSL (Domain Specific Language), задание исходя из этой специфики необходимых моделей из следующих: модель управления архитектурой MDA (Model Driven Arhitecture), модель генерации домена GDM (Generative Domain Model), платформо-независимая модель PIM (Platform Independed Model), модель специфичной платформы PSM (Platform Specific Model) и пр..[138, 204, 215]. Каждая из приведенных моделей трансформируется в другие промежуточную или конечную модель исходного кода для последующего адаптивного исполнения в разных компьютерах и средах и оценки результата изготовления продукта на заданные показатели качества. Основа этой науки – теория алгоритмов и программирования, теория вычислений и распределенной обработки, теория вычислительных сетей и пр. ПИ занимает центральное место среди названных дисциплин CS (рис.10.16), без нее не мыслимы никакие теоретические и практические достижения в этих дисциплинах.

Иными словами, ПИ обеспечивает их и сама развивается, используя новые достижения в теории и практике построения самих компьютеров.

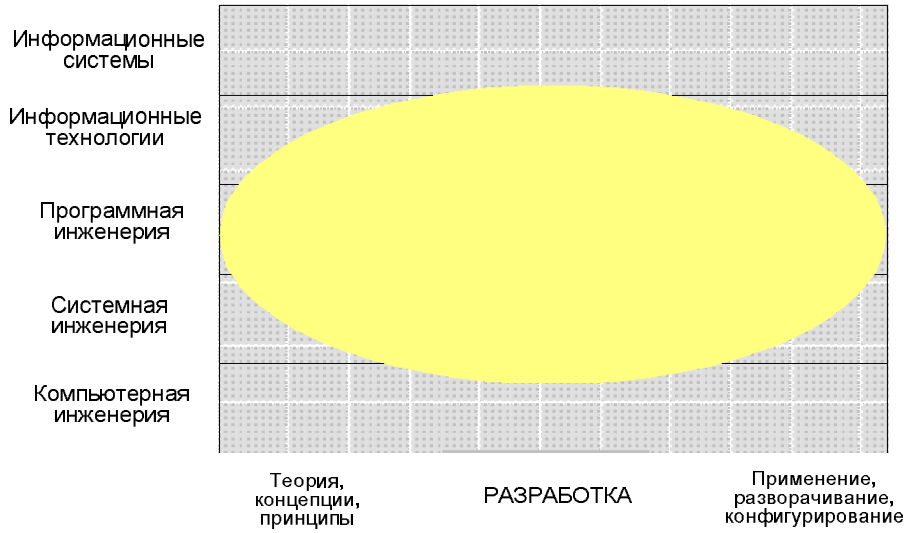


Рис.10.16. Область ПИ в пространстве информатики

На приведенном рисунке показано, что ПИ занимает центральную область в проблемном пространстве информатики [205], начиная частично с области компьютерной инженерии, проходя системную инженерию, информационные технологии, направляясь в сторону информационных систем. ПИ обеспечивает

систематические разработки крупномасштабных программных проектов, которые автоматизируют значительную часть различных сфер пространства информатики. Эта область достигает организационных и промышленных вопросов проектирования различных компьютерных систем.

Стимулом развития ПИ является массовое производство ПП разными программастроительными фирмами и широкое использование ПП в различных сферах жизни общества.

При создании ПП разработчики проектов ИС, АСУ, СОД и др. используют из ПИ новейшие технологии и средства управления программными проектами (планирование работ, регулирование ресурсов, контроль), измерения результатов процессов ЖЦ, оценки риска и качества. Руководством для разработки ПП являются стандарты ЖЦ, качества (ISO/IEC 12207, ISO/IEC 9126) и др. Стандарты регламентируют все виды деятельности, начиная с разработки требований, необходимых моделей, тестирования и оценки качества экспертными и количественными методами. И, кроме того, на основе стандартов создаются ТЛ и процессы, которые соответствуют выпускаемой продукции и после завершения работ процессы и продукты усовершенствуются.

Вывод. В данной главе представлен системный анализ аспектов индустриального производства ПП, включая ЖЦ, разделы ядра знаний SWEBOK, PMBOK, стандарты качества и оценивания ПП. Предложена основополагающая система (научной, инженерной, управленческой, экономической, производственной) дисциплин. Сформулировано их содержание и назначение в производственном цикле. Приведены аргументы использования этих дисциплин на практике. Исходя из основ этих дисциплин, рассмотрены аспекты построения целевых программных объектов (программного обеспечения, приложений и доменов) из готовых КПИ. Показана роль базового процесса ПИ, как процессной основы производства компьютерных программ с использованием ядра знаний, стандартов ЖЦ, инфраструктуры и менеджмента в организации–разработчике. Предложены и описаны основы индустриального производства программных продуктов, включающие в себя систему дисциплин, методы, принципы и средства каждой из них поддерживают соответствующие аспекты производственной деятельности при изготовлении программной продукции. Дано определение роли и назначения сформулированных основ ПИ в создании информационных систем и технологий в пространстве информатики.

ЗАКЛЮЧЕНИЕ

Подведя итоги, рассмотренным в данной монографии основным понятиям сборочного программирования, отметим широкий спектр вопросов, имеющих отношение к дальнейшему развитию данной проблематики. В частности, это относится к вопросам мобильности, интероперабельности, совместимости изготовленного ПС на разных платформах и средах функционирования. Вместе с тем идея сборки входит явно или неявно в состав большинства новых видов программирования ПС (например, компонентного, сервисного и др.). В рамках сборочного программирования получены результаты, которые являются перспективными и в настоящее время:

– метод объектно-компонентного проектирования ПрО, объединяющий объектно-ориентированный анализ предметной области, построение объектной модели с отображением взаимоотношений объектов и переход от этой модели к компонентной путем добавления формального аппарата спецификации функций этих объектов в виде отдельных компонентов. Между объектными и компонентными моделями устанавливаются эквивалентные отображения, которые обеспечиваются формальными преобразованиями объектного представления ПрО в компонентное;

– систематизация дисциплин программной инженерии, дополнение сборочного программирования новыми аспектами управления, экономического и инженерного изготовления программных продуктов. Сформулированное содержание каждой из дисциплин образуют базовые основы производства ПП из готовых программных ресурсов, собираемых согласно формализации их интерфейсов, в семейство систем с использованием инструментов, реализованных в современных средах. Процессы разработки ПС из готовых ресурсов включают в себя теоретическую постановку задач предметной области, инженерное их описание средствами современных ЯП, верификацию и оценивание результатов деятельности специалистов и заданных в требованиях характеристик качества программного продукта.

Рассмотренные технологические аспекты сборочного программирования по своей сути соответствуют следующим основополагающим принципам современных технологий и основ производства программных продуктов:

– регламентация процессов анализа, проектирования и разработки ПС на единой методологической основе в рамках единой технологии;

– смещение поздних процессов разработки ПС (программирование, тестирование) на более ранние (постановка задач, формирование требований, проектирование, верификация и др.) процессы, на которых активно используют готовые артефакты, КПИ и новые виды формальных моделей ПрО (DSL, GDM, UML, PIM и др.) с описанием специфических функций системы (или семейств систем) и их трансформации к выходному коду систем семейства;

– управление качеством, экспертизой, измерением и оценкой промежуточных и конечных результатов проектирования на всех процессах ЖЦ;

– переход к промышленным методам и средствам изготовления продуктов массового применения.

– подготовка специалистов Вузов по новым программам квалификационного образования компьютерных дисциплин, в частности, программной инженерии и информатики (Curricula-2004) с учетом современных достижений в них, позволит получить широкий профиль специальностей (менеджеров, инженеров, аналитиков, верификаторов и др.), способных выполнять разнообразные задачи автоматизации систем типа: системы семейства, информационные системы, СОД, АСУ и др. Это дает основание считать наличие обоснованных закономерностей и достаточных условий для постепенного перехода к гибким технологиям индустриального производства разных типов ПС.

При изложении вопросов формирования и описания ТЛ и ТП предложены современные подходы к их созданию. Рассмотрена новая версия линии – Product line, используемая для выпуска продукции для рынка. Это один из путей автоматизированного производства ПП и пока не достаточно распространенный в практической жизни.

Теоретические и практические достижения такого типа, определение фундаментальных основ производства – системы дисциплин ПИ в перспективе обеспечат значительный прогресс сборочной индустрии программной продукции разного назначения из накопленных КПИ и других готовых программных ресурсов.

СПИСОК ЛИТЕРАТУРЫ

1. *Ануфриев И.Е., Смирнов А.Б., Смирнова Е.Н.* MATLAB 7.–СПб.: БХВ-Петербург, 2005.– 1104 с.
2. *Авен О.И., Гурин Н.Н., Коган Я.А*. Оценка качества и оптимизации вычислительных систем. – М.: Наука, 1982,–464 с.
3. *Агафонов В.Н.* Типы и абстракция данных в языках программирования // Данные в языках программирования.–М.: Мир, 1982.–С. 267—327.
4. *Агафонов В.Н.* Спецификация программ: понятийные средства и их организация. – Новосибирск : Наука, 1987.–290 с.
5. *Андон Ф.И., Лаврищева Е.М.* Методы инженерии распределенных компьютерных приложений.–К.: Наук. думка, 1998.– 228с.
6. *Андон Ф.И., Коваль Г.И, Коротун Т.М., Лаврищева Е.М., Суслов В.Ю.* Основы инженерии качества программных систем // 2-е изд. – Киев: Академпериодика, 2007. – 680 с.
7. *Андон Ф.И., Суслов В.Ю., Коротун Т.М., Коваль Г.И, Слабоспицкая О.А.* Модель оценки технологической зрелости организаций-разработчиков ПО // Проблемы программирования.– 1998. – №4. – С. 46–57.
8. *Андон П.І., Суслов В.Ю., Коротун Т.М., Коваль Г.І., Слабоспицька О.О.* Визначення витрат на створення ПЗ автоматизованих систем // Проблемы программирования, 1998.– №3.– С.23 –34.
9. *АндонФ.И., СусловВ.Ю., Коротун Т.М., Коваль Г.И., Слабоспицкая О.А.*. Управление риском проектов ПО // Проблемы программирования.–1999.– №1. – С. 53 – 62.
10. *Андон Ф.И., Захарова Э.Г., Резниченко В.А., Яшунин А.Е.* Интеллектуализация информационных систем // УсиМ.–1987.–№ 1.–С. 84—92.
11. *Андон Ф.И., Яшунин А.Е., Резниченко В.А.* Логические модели интеллектуальных информационных систем.–Киев, Наук.думка, 1999.–396с.
12. *Айлиф Дж.* Принципы построения базовой машины.–М.: Мир, 1973.—120 с.
13. *Александров П.С.* Введение в теорию множеств и общую топологию.—М.: Наука, 1977.–367 с.
14. *Андерсон Р.*Доказательство правильности программ.–М.: Мир, 1982.—164 с.
15. *Ахо А., Хопкрафт Дж., Ульман Дж.* Построение и анализ вычислительных алгоритмов.–М.: Мир, 1979.– 536 с.
16. *Бабаян Б.А.* Основные принципы программного обеспечения МВК «Эльбрус».–М., 1977.–11 с.–(Препр./АН СССР. Ин-т точной механики и вычисл. техники им. С. А. Лебедева; № 7).
17. *Бабенко Л.П.* Проблемно-ориентированные средства языка Кобол.–М.: Статистика, 1979.– 292с.
18. *Бабенко Л.П., Лавріщева К.М.* Основи програмної інженерії. Посібник.–К.: Знання, 2001.– 269с.
19. *Бабенко Л.П.* Онтологический подход к спецификации свойств программных систем и их компонент // Кибернетика и системный анализ. – 2009. – № 1. – с.30–37.
20. *Бабенко Л.П., Поляничко С.Л.* Онтологические модели описания готовых ресурсів в разработке программ//Пробл. программирования.– Киев, 2004.– № 2-3.-с.173-179.
21. *Бабенко Л.П.* Проблемы повторного использования в программной инженерии// Кибернетика и системный анализ, 1999.– №2– с.155–166.
22. *Барзтисс А.Т.* Структуры данных.–М.: Статистика, 1974.–408 с.
23. *Барлао Р., Прошан Ф.* Математическая теория надежности. Пер.с англ. М.: 1969.– 483с.
24. *Барлет Н, Лесли А., Симкин С.* Программирование на Java, Путеводитель.–К.:1996.– 736с.
25. *Бассакер Р., Саати Т.* Конечные графы и сети.—М.: Наука, 1974.—236 с.
26. *Баховец О.Б., Грінченко Т.О., Гуляєв К.Д. та ін.*. Передумови становлення інформаційного суспільства в Україні.– К.: Азимут України, 2008.–287с.
27. *Бежанова Н.М.* Анализ и систематизация встроенных проблемно-ориентированных систем //УСиМ.–1981.—№ 4.—С. 113—118.

28. *Бей И.* Взаимодействие разноязыковых программ.– М.–С.–Петербург.–Киев: Изд. дом «Вильямс», 2005. – 868 с.
29. *Боэм Б.У.* Инженерное проектирование программного обеспечения.–М.: Радио и связь, 1985.—511 с.
30. *Боэм Б., Браун Дж., Каспар Х. и др*. Характеристика качества программного обеспечения : Пер. с англ. Е. К. Масловского,—М.: Мир, 1981.—208 с.
31. *Брукс Ф.П.* Мифический человеко-месяц или как создаются программные системы – СпБ.: Символ-плюс, 2005.–304с.
32. *Брябрин В. Н.* Программное обеспечение персональных ЭВМ.–М. : Наука, 1988.–272 с.
33. *Бутаков В А*. Методы создания качественного программного обеспечения ЭВМ.—М : Энергоатомиздат, 1984.—232 с.
34. *Бухштаб Ю.А., Горлан А.И., Камынин С.С. и др.* Интеллектуальный пакет, использующийся при планировании знания о предметной области и функциональных модулях //Изв. АН СССР, Техн. Кибернетика,—1981—№ 5.–С. 113—124.
35. *Буч Г., Рамбо Д., Джекобсон А.* Язык UML. Руководство пользователя: Пер. с англ. – М.: ДМК, 2000. – 432 с.
36. *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами приложений на C++. - М.: "Издательство Бином", СПб.: "Невский диалект", 2001. – 560 с.
37. *Вдовкин С.В., Кубенский А.А., Сафонов В.О.* Реализация языка CLU // Прикладная информатика.–1984.—Вып. 2.—С. 127—130.
38. *Вегнер П.* Программирование на языке Ада.–М. : Мир, 1983.–240 с.
39. *Вельбицкий И.В.* Технология программирования.–Киев : Техника, 1984.–279 с.
40. *Вилдермьюс Ш. , Нортрап Т. , Райан Б.* Основы разработки приложений на платформе MS.NET Framework 2.0 – "Питер", "Русская Редакция", 2007, 864 с.
41. *Вирт Н.*Систематическое программирование: Введение.–М.: Мир, 1977.–188 с.
42. *Вирт Н.*Алгоритм+структуры данных=программы: Пер. с англ.–М.: Мир, 1985.–406 с.
43. *Вирт Н.* Модула-2 // Алгоритмы и алгоритмические языки. Языки программирования.–М.: Наука, 1985.—С. 3—46.
44. *Вишня А.Т., Лаврищева Е.М., Грищенко В.Н. и др*. Система автоматизации программ АПРОП.–Киев, 1980.–570 с.–Деп. в РФАП АН УССР 21.09.80, № 5707.
45. *Волин В.Г., Грузман В.А., Синичкин В.И., Эпштейн В.Л.* Автоматизация проектирования систем управления.–М. : Финансы и статистика, 1981,—С. 101 –111.
46. *Волховер В.Г., Иванов Л.А.* Производственные методы разработки программ.–М. : Финансы и статистика, 1983.—208 с.
47. *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб: Питер, 2001. – 368 с.
48. *Гантер В.* Методы управления проектированием программного обеспечения.—М. : Мир, 1981.–392 с.
49. *Гласс Р.* Руководство по надежному программированию.–М. : Финансы и статистика, 1982.–256 с.
50. *Глушков В.М.* Основы безбумажной информатики.—М. : Наука, 1982. –552 с.
51. *Глушков В.М., Цейтлин Г.Е., Ющенко Е.Л.* Алгебра. Языки. Программирование.–Киев : Наук. думка, 1974.—318 с.
52. *Глушков В.М.* Введение в АСУ.–Киев: Техніка, 1974.–320 с.
53. *Глушков В.М., Вельбицкий И.В.* Технология программирования и проблемы ее автоматизации // УСиМ.–1976.–№ 6.–С. 75—93.
54. *Глушков В.М.* Фундаментальные исследования и технология программирования // Программирование.–1980.–№ 1.—С. 3—13.
55. *Глушков В.М., Капитонова Ю.В., Летичевский А.А.* О применении метода формализованных технических заданий к проектированию программ обработки структур данных // Там же.–1978.–№ 6.–С. 31—43.
56. *Глушков В.М., Лаврищева Е.М., Стогний А.А. и др.* Система автоматизации производства программ. (АПРОП) –Киев : Ин-т кибернетики АН УССР.–1976.–134 с.
57. *Глушков В.М.* Кибернетика, вычислительная техника, информатика.–Избран. труды в трех томах.–К.: Наук. думка, 1990.–768с..
58. *Грис Д.* Наука программирования / Пер. с англ. под ред. А. П. Ершова.–М.: Мир, 1984.—416 с.

59. *Громов Г.Р.* Национальные информационные ресурсы: Проблемы. пром. эксплуатации.—М.: Наука, 1984.–291с.
60. *Грищенко В.Н.* Вопросы комплексирования программных средств // УСиМ.–1987.—№ 1.—С. 68—71.
61. *Грищенко В.Н., Лаврищева Е.М.* О создании межъязыкового интерфейса для ОС ЕС // УСиМ.–1978.–№ 1—С. 34—41.
62. *Грищенко В.Н., Лаврищева Е.М.* О стандартизированной сборке сложных программ // Технологическое и программное обеспечение АСУ.–Киев: Ин-т кибернетики АН УССР, 1978.—С. 36—42.
63. *Грищенко В.Н., Лаврищева Е.М.* Методы и средства компонентного программирования // Кибернетика и системный анализ.– 2003.– № 1.– С. 39–55.
64. *Грищенко В.Н.* Метод объектно-компонентного проектирования программных систем.–К.: Проблеми програмування, 2007.– № 2.–с.113–125.
65. *Грищенко В.Н.* Подход к формализации объектно-ориентированной методологии // Проблемы программирования. – 1997. – № 1.– С. 33–39.
66. *Грищенко В.М.* Підхід до аналізу поведінки об`єктних систем при моделюванні предметних областей // Проблеми програмування. – 1998. – № 4. – С. 67–75.
67. *Грищенко В.Н.* Формальные модели компонентного программирования // Проблемы программирования. – 2003. – № 2.– С. 42–57.
68. *Грищенко В.Н.* Методи еволюції програмних компонентів для їх повторного застосування// Проблеми програмування. – 2004. – № 2–3. – С. 215–222.
69. *Грогоно П.* Программирование на языке Паскаль.–М.: Мир, 1982.–382 с.
70. *Гультяев А.К.* Microsoft Project 2002. Управление проектами. – СПб.: Корона принт, 2003. – 589
71. *Дал У., Дейкстра Э., Хоор К.* Структурное программирование.–М.: Мир, 1975.–247 с.
72. Данные в языках программирования.–М.: Мир, 1982.–328 с.
73. *Дворцин В.Н., Прокудин Г.С.* Изготовление программного обеспечения агрегированием из готовых компонент // УСиМ.–1984.–№ 6.–С. 62—64.
74. *Джалота П.* Управление программным проектом на практике. М.: Лори, 2005.–223с.
75. *Диковский А.Я.* ФОРМАТ: язык для автоматического синтеза и тестирования программ обработки данных // Синтез, тестирование, верификация и отладка программ.–Рига : Изд-во Латв. ун-та, 1981.–С. 87—88.
76. *Добров А.Д. и др*. Особенности комплексирования программ в системе программирования МВК «Эльбрус» / А. Д. Добров, В. М. Пенковский, М. С. Приказчиков, В. С. Чижова // УСиМ.–1980.–№ 4.–С. 92—96.
77. *ДСТУ 2850–1994.* Программные средства ЭВМ. Обеспечение качества. Показатели и методы оценки качества программного обеспечения.
78. *ДСТУ 3230–1995.* Управление качества и обеспечение качества. Термины и определения.
79. *Ершов А.П.* Введение в теоретическое программирование.–М.: Наука, 1977.—288 с.
80. *Ершов А.П.* Два облика программирования // Кибернетика.–1982.–№ 6. С. 122 –123.
81. *Ершов А.П.* Опыт интегрального подхода к актуальной проблематике программного обеспечения // Там же.–1984.–№ 3.—С. 11—21.
82. *Жоголев Е.А.* Принципы построения многоязыковой системы модульного программирования // Там же.–1974.–№ 4.—С. 78—83.
83. *Жоголев Е А.* Технологические основы модульного программирования // Там же.–1980.—№ 2.—С. 44—49.
84. *Задорожная Н.Т., Лаврищева Е.М.* Менеджмент документооборота в информационных системах образования. – Киев: Педагогічна думка, 2007. – 220 с.
85. *Замулин А.В., Скопин И.Н.* Конструирование базы данных на основе концепции абстрактных типов данных // Программирование.– 1981.– № 5.– С. 38–43.
86. *Замулин А.В.* Типы данных в языках программирования и базах данных.– М.: Наука, 1987.– 152 с.
87. *Зелковиц М., Шоу А., Гэннон Дж.* Принципы разработки программного обеспечения.—М.: Мир, 1982.—368 с.
88. *Землянский А.А.* Метод композиции программ иерархической структуры// Прикл. информатика.– 1984.– Вып. 2.– С. 135–142.
89. *Зинглер К.* Методы проектирования программных систем.–М.: Мир, 1985.–328 с.
90. *Иванников В.П.* Соглашения о связях и ядро операционной системы суперЭВМ // Кибернетика

и вычисл. техника. – 1985. – Вып. 1.– С. 49–52.
91. *Иванова Э.Б., Вершинин М.М.* Java 2, Enterprise Edition. Технологии проектирования и разработки. Спб.: Бхв-Петербург, 2003.-1088 с.
92. *Иванов А.С.* Язык Си. Предварительное описание // Прикл. информатика.–1985.—Вып.1.—С. 68—113.
93. *Йенсен К., Вирт Н.* Паскаль. Руководство для пользователей и описание языка.– М.: Финансы и статистика, 1982.– 152 с.
94. *Камынин С.С., Любимский Э.З.* Алгоритмический машинно-ориентированный язык –АЛМО // Алгоритмы и алгоритм. яз.–1967.–Вып. 1.–С. 5—58.
95. *Канторович Л.В.* Перспективы работы в области автоматизации программирования на базе крупноблочной системы // Тр. Ин-та математики им. Стеклова. – 1968.—196.—С. 5—15.
96. *Капитонова Ю.В., Летичевский А.А.* О конструировании математических описаний предметных областей // Кибернетика.– 1988.– № 4.– С. 17–25.
97. *Капитонова Ю.В., Летичевский А.А.* Парадигмы и идеи академика В.М. Глушкова.–К.: Наук.думка, 2003.–454с.
98. *Катков В.Л., Казан В.Н.* Система тестовой обработки «Стрела» //УСиМ.– 1984.–№ 1.–С.106–111.
99. *Кахро М.И., Калья А.П., Тыугу Э.Х.* Инструментальная система программирования ЕС ЭВМ (ПРИЗ).–М.: Финансы и статистика, 1981.–157 с.
100. *Кендел С.*Унифицированный процесс. Основные концепции.–М.:Спб.–Киев.–2002.–157с.
101. *Коваль Г.И., Коротун Т.М., Лаврищева Е.М.* Об одном подходе к решению проблемы межмодульного и технологического интерфейсов // Диалоговые системы: Межотрасл. сб. АН СССР и Минвуза СССР.– 1988.
102. *Коваль Г.И.* Модели и методы инженерии качества программных систем на ранних стадиях жизненного цикла.– Автореф. дис.канд..-физ.-мат. наук. – Киев: Ин-т кибернетики им. В.М. Глушкова НАН Украины, 2005. – 19 с.
103. *Коллинз Г., Блей Дж.* Структурные методы разработки систем от стратегического планирования до тестирования.– М.: Финансы и статистика, 1986.–264 с.
104. *Коротун Т.М.* Модели и методы инженерии тестирования программных систем в условиях ограниченных ресурсов. – Автореф. дис.канд.-физ.-мат. наук. – Киев: Ин-т кибернетики им. В.М. Глушкова НАН Украины, 2005. – 21 с.
105. *Королев Л.И.* Структуры ЭВМ и их математическое обеспечение.–М.: Наука, 1978.–362 с.
106. *Корягин Д.А.* Об одном подходе к проблеме разработки системного обеспечения пакетов программ для задач вычислительной физики // Программирование.– 1982.– № 2.– С. 44–50.
107. *Краснощеков П.С., Петров А.А.* Принципы построения моделей.- М.: Изд-во МГУ, 1983. -264с.
108. *Кульба В.В., Маминоков А.Г.* Методы анализа ч синтеза оптимальных модульных систем обработки данных // Автоматика и телемеханика.–1980.–№ П.—С. 81—92.
109. *Куприянов В.П. и др*. Технологическая система ТКП-3 / В. П. Куприянов, А. К. Мочалов, Н. А. Маслобойщиков н др.//Приборы и системы упр.– 1983.–№ 4.–С. 21–23.
110. *Лаврищева Е.М.* Методика построения программных комплексов на основе банка модулей // Разработка математических и технических средств АСУ.–Киев: Ин-т кибернетики АН УССР, 1975.–С. 3—12.
111. *Лаврищева Е.М.* Модульный принцип конструирования больших программ // Там же.—С. 12—20.
112. *Лаврищева Е.М.* Вопросы объединения разноязыковых модулей в ОС ЕС// Программирование.—1978.—№ 1.—С. 22—27.
113. *Лаврищева Е.М.* Подход к промышленной технологии изготовления больших программ // Перспективы развития в системном и теоретическом программировании: Тр. Всесоюз. симп. Новосибирск, 20—22 марта 1978 г.–Новосибирск: Изд-во СО АН СССР, 1978.—С. 122—127.
114. *Лаврищева Е.М.* Об автоматизированном изготовлении программных агрегатов из равноязыковых модулей//УСиМ.–1979.—№ 5.–С. 54—60.
115. *Лаврищева Е.М.* Методика изготовления программных агрегатов // Кибернетика.–1980.–№2.– С. 77—82.
116. *Лаврищева Е.М., Грищенко В.Н.* Связь равноязыковых модулей в ОС ЕС –М.: Финансы и статистика, 1982.–127 с.
117. *Лаврищева Е.М.* Технология создания прикладного программного обеспечения в СОД.–Киев: О-во «Знание» УССР, 1983.–15 с.
118. *Лаврищева Е.М., Хоролец Д.С.* Комплекс АПФОРС –средство автоматизации проектирования

ППП // Средства информационного обеспечения АН УССР.—Киев: Ин-т кибернетики им. В. М. Глушкова АН УССР, 1984.–С. 67—74.
119. *Лавришева Е.М.* Принципы технологической подготовки разработки ППО СОД//Проектирование и разработка пакетов программ.–Киев: Ин-т кибернетики им. В. М. Глушкова АН УССР, 1987.—С. 34—40.
120. *Лавришева Е.М.* Основы технологической подготовки разработки прикладных программ. СОД.—Киев, 1987.—29 с.–(Препр./АН УССР. Ин-т кибернетики им. В. М. Глушкова; № 87—5).
121. *Лавришева Е.М., Хоролец Д.С., Куцаченко Л.И. и др.* Комплекс программных средств, обеспечивающих автоматизированное построение пакетов прикладных программ на основе формализованных спецификаций модулей – АПФОРС.—Ереван, 1985.—220 с.—Деп. в ЕрНУЦ СНПО «Алгоритм» 18.05.85, № 104; Рег. в РФАП АН УССР 25.09.87, № АД0002.
122. *Лавришева Е.М. и др.* Программно-технологический комплекс ведения разработки прикладного программного обеспечения. Технологические документы / Е.М. Лаврищева, Г.И. Коваль, Т.М. Коротун, Е.И. Моренцов,–Киев, 1988.—571 с—Рег. в РФАП АН УССР 30.09.88, № АП0218—И.
123. *Лавришева Е.М.* Технологическая подготовка и программная инженерия// УСиМ.–1988.—№ 1.—С. 48—52.
124. *Лавришева Е.М.* Модель процесса разработки программных средств // УСиМ.–1988.–№ 5.–С. 43—46.
125. *Лавришева Е.М., Грищенко В.М.* Сборочное программирование.– Киев: Наук. думка, 1991.– 213С.
126. *Лавришева Е.М.* Сборочное программирование. Некоторые итоги и перспективы // Проблемы программирования, 1999.–№2.–с. 20–32.
127. *Лавришева Е.М.* Парадигма интеграции в программной инженерии //Проблемы программирования.–2000.–№1–2.–С.351-360.
128. *Лавріщева К.М.* Основні напрями досліджень у програмній інженерії і шляхи їх розвитку // Проблеми програмування. – 2002.– №3–4.– С.44 –58.
129. *Лавришева Е.М., Рожнов А.М.* Концепция аналитической оценки характеристик качества программных компонентов // Проблеми програмування. – 2004.– №3–4.– С.180–187.
130. *Лавришева Е.М.* Современные методы программирования. Возможности и инструменты //Проблеми програмування.–2006.–№2–3.–С.60–74.
131. *Лавришева Е.М., Коваль Г.И., Коротун Т.М.* Подход к управлению качеством программных систем обработки данных // Кибернетика и системный анализ.– 2006.–№ 5.–С.174–185.
132. *Лавришева Е.М.* Интерфейс в программировании // Пробеми програмування.– 2007.–№2.– С.126–139.
133. *Лавришева Е.М.* Методы программирования. Теория, инженерия, практика.–К.: Наук.думка, 2006.–451с.
134. *Лавришева Е.М., Петрухин В.А.* Методы и средства инженерии программного обеспечения.– Москва, Учебник .–МФТИ.– 2007.– 415 с.
135. *Лавришева Е.М.* Становление и развитие модульно-компонентной инженерии программирования в Украине //Препринт 2008–1.–Ин-т кибернетики им. В.М.Глушкова, 33 с.
136. *Лавришева Е.М.* Программная инженерия – научная и инженерная дисциплина //Кибернетика и системный анализ. – 2008. – №3. – С. 19 – 28.
137. *Лавріщева К..М.* Визначення предмету – програмна інженерія. //Проблеми програмування.– 2008.–№ 2–3.–С.191–204.
138. *Лавріщева К.М., Коваль Г.І., Коротун Т.М.* Подход к управлению качеством программных систем обработки данных // Кибернетика и системный анализ.–2006.– №5.–С.174–185.
139. *Лавришева Е.М.* Классификация дисциплин программной инженерии // Кибернетика и системный анализ.–2008.– №6.–С.3–9.
140. *Лавріщева К.М.* Програмна інженерія. Підручник.–К.: Академперіодика, 2008.– 319 с.
141. *Лавришева Е.М., Слабоспицкая О.А.* Подход к єкспертному оцениванию в программной инженерии. //Кибернетика и системный анализ. – 2009. – №4. – С. 151 – 168.
142. *Лавришева Е.М.* Генерувальне програмування програмних систем і сімейств //Проблеми програмування. – 2009.– № 1.– С. 3-16.
143. *Лавров С.С.* Синтез программ // Кибернетика.–1982.—№ 6.–С. 11—16.
144. *Лавров С.С.* Основные понятия и конструкции языков программирования.– М.: Финансы и

статистика, 1982.–80 с.
145. *Леман Д., Смит М.* Типы данных // Данные в языках программирования.– М.: Мир, 1982.–С. 196–213.
146. *Леонов И.В.* Введение в методологию разработки программного обеспечения с помощью Rational Rose,– Эскейп, 2004.–301с.
147. *Летичевский А.А , Маринченко В.Г.* Объекты в системе алгебраического программирования // Кибернетика и системный анализ. –1997.– №2. – С.160–180.
148. *Летичевский А.А, Капитонова Ю.В., Волков В.А., Вішемирский В.В., Летичевский А.А (мл)* Инсерционное программирование // Кибернетика и системный анализ.–2003.– №1.–С12–32.
149. *Липаев В.В.* Оценка затрат на разработку программных средств,–М.: Финансы и статистика, 1988.–225 с.
150. *Липаев В.В.* Надежность программного обеспечения АСУ.–М.: Энергоиздат, 1981.—240 с.
151. *Липаев В.В,. Позин Б.А., Штрик* А.А. Технология сборочного программирования..–М.: Радио и связь, 1993.–272с.
152. *Липаев В.В.* Методы обеспечения качества крупномасштабных программных систем.-М.: Синтег, 2003.–510с.
153. *Литвинов В.В.* Математическое обеспечение проектирования вычислительных систем и сетей.– Киев: Техника, 182.– 176 с.
154. *Майерс Г.* Надежность программного обеспечения.–М.: Мир, 1980.—360с.
155. *Массер Д.* Спецификации абстрактных типов данных в системе AFFIRM// Требования и спецификация в разработке программ.—М.: Мир, 1984.–С. 199—222.
156. *Марчук Г.И., Котов В.Е.* Модульная развиваемая система (концепция).– Новосибирск, 1978.–2 ч.– (Препр./ВЦ АН СССР; 86–87).
157. *Маурер У.* Введение в программирование на языке Лисп.– М.: Мир, 1976.– 104 с.
158. *Мейер Б.* Объектно-ориентированное конструирование программных систем. – "Русская Редакция", 2005. – 1232 с.
159. *Минский М.* Фреймы для представления знаний.–М.: Энергия, 1979.–191 с.
160. *Михалевич В.С. и др.* Организация вычислений в многопроцессорных вычислительных системах /В. С. Михалевич, Ю. В. Капитонова, А. А. Мышевский, И. Н. Молчанов//Кибернетика.–1984.–№ 3.–С. 1–10.
161. *Морган М.* Java2. Руководство разработчика: Пер.с англ. – М.: Издательский дом "Вильямс", 2000. – 720 с.
162. *Мороз Г.Б., Лаврищева Е.М.* Модели роста надежности программного обеспечения.//Препр. 1992–38– Ин-т кибернетики им. В.М.Глушкова, Киев. – 23 с.
163. *Нікітченко Н.С. , Шкільняк С.С.* Математична логіка та теорія алгоритмів.– ВПЦ "Київський національний університет", 2008.–528с.
164. *Опарин Г.А.* САТУРН – метасистема для построения ППП//Разработка ППП.–Новосибирск: Наука, 1982.–С. 130–360.
165. *Орлов В.И.* Комплексирование программ в ОС ЕС.—М.: Финансы и статистика, 1986.–136 с.
166. *Орфали Р., Харки Д., Эдвардс Дж.* Основы CORBA. –М.: Из.–во "Малип", 1999.–317с.
167. *Оши К., Хьюгз П.* Бухгалтерский учет на микроЭВМ: использование прикладного пакета LOTUS 1-2-3.–М.: Финансы и статистика, 1988.—255 с.
168. *Павлов В.* Аспектно–ориентированное программирование //Технология клиент-сервер, №3-4.–с.3-45.
169. *Парасюк И.Н., Сергиенко И.В.* О некоторых задачах модульного анализа при проектировании пакетов программ //УсиМ,–1982.–№ 4.–С. 73–80.
170. *Парасюк И.Н., Сергиенко И.В.* Инструментально-базовая технология сборочного программирования одного класса интеллектуальных пакетов прикладных программ на ЕС ЭВМ.–Труды межд. научно-технической конференции.–Калинин, 1987.–секция 3.–с.43-47.
171. *Парнас Д.* Метод спецификации модулей программного обеспечения // Данные в языках программирования.–М.: Мир, 1982.–С. 9–24.
172. *Пелипенко Н.И.* Организация связей между разноязыковыми программами //УсиМ.–1981.–№ 1.–С. 65–68.
173. *Перевозчикова О.Л., Ющенко Е.Л.* Система диалогового решения задач на ЭВМ.–Киев: Наук. Думка, 1986.–264 с.
174. *Пратт Т.* Языки программирования: разработка и реализация.–М.: Мир, 1979.–576 с.
175. Представление и использование знаний / Под ред. Х.Уэно, М.Исидзука.–М.: Мир, 1989.–220 с.
176. *Просиз Дж.*Программирование для Microsoft.Net /Пер. с англ. - М.: Издательско-торговый дом

"Русская Редакция*, 2003. - 704 с.
177. *Рамбо Дж., Джекобсон А , Буч Г.* UML: специальный справочник.– СПб.: Питер, 2002.– 656с.
178. *Редько В.Н.* Основания композиционного программирования // Программирование.–1979.– №3.–С. 3–13.
179. *Редько В.Н.* Семантические структуры программ // Там же.–1981.–№ 3.—С. 3—19.
180. *Редько В.Н.* Прогматические основания дескрипторных сред // Программирование.– 2005.– №3.– С. 3–25.
181. Рекомендации по преподаванию программной инженерии и информатики в университетах.– Computing Curricula-2001: Computer Scie..–Перев. с англ.–М.: Университет Информ. технологий, 2007.–462 с
182. *Риз Ч.Е., Стюарт Дж Д.* Rule Master: система конструирования знаний для построения научных экспертных систем / Искусственный интеллект: применение в химии.—М.: Мир, 1988.—С. 33—48.
183. *Римский Г.В.* Структура и функционирование системы автоматизации модульного программирования // Там. Же.–1987.–№ 5.–С. 36—44.
184. *Родионов С.Т.* Основные концепции модульности и анализа некоторых направлений развития общей технологии программирования // Там же.– 1980,– №2.–С. 31–38.
185. *Д.М. Рябко.* Подход к реализации среды разработки для DSL//Проблеми програмування. – 2007.– № 4.– С. 3-12.
186. *Ройс Уокер* . Управление проектами по созданию программного обеспечения. – М.: Лори, 2002.– 424 с.
187. *Сергиенко И.В., Парасюк И. П., Тукалевская Н.И.* Автоматизированные системы обработки данных.– Киев: Наук. думка, 1976.– 256 с.
188. *Сергієнко І.В.* Інформатика і комп'ютерні системи. Киев: Наук. думка, 2004.– 430с.
189. Стогний А.А., Ананьевский С.А., Барсук Я. И. и др. Программное обеспечение персональных ЭВМ.– Киев : Наук. думка, 1989.— 368 с.
190. *Слабоспицыка О.О.* Моделі і методи експертного оцінювання у життєвому циклі програмних систем. Автореф. дис канд. физ.-мат. наук. – Ін-т кібернетики ім. В.М. Глушкова НАН України, Київ: 2008. – 21 с.
191. *Сомервилл И.* Инженерия программного обеспечения.– Изд. дом „Вильямс", Москва+ Санкт–Петербург+ Киев.– 2002.– 623с.
192. *Трахтенгерц Э.А.* Программное обеспечение автоматизированных систем управления.– М.: Статистика, 1974.– 288 с.
193. Требования и спецификации в разработке программ.–М.: Мир, 1984.–344 с.
194. *Турский В.* Методология программирования.–М.: Мир, 1981.–265 с.
195. *Тыугу Э.Х.* Концептуальное программирование.–М.: Наука, 1984.–256 с.
196. *Фаулер М.* Рефакторинг: улучшение соответствующего кода. – СПб.:Символ
197. *Фреге Г.* Логика и логическая семантика. – М.: Аспект
198. *Филина Л.Н.* Вопросы связи модулей, транслированных с языков ФОРТРАН, ПЛ-1, Ассемблер в ОС ЕС ЭВМ // Программирование.–1980.–№ 3.–С. 39—43.
199. *Фомичев В., Пютчлер У.* Промежуточные языки систем программирования // ЭВМ в проектировании и производстве.–Л., 1987.–Вып. 3.—С. 251—270.
200. *Хаббибуллин И.Ш.* Разработка –служб средствами. СПб.: БХВ-Петербург, 2003.- 400 с.
201. *Хоар Ч.* О структурной организации данных// Структурное программирование.—М.: Мир, 1975.—С. 92—197.
202. *Холстед М.Х.* Начало науки о программах / Пер. с англ. В. М. Юфы.– М.: Финансы и статистика, 1981.– 201 с.
203. *Хорн Э., Винклер Ф.* Проектирование модульных программных структур/ Вычисл. техника, соц. стран.– 1987.– Вып. 21,–С. 64–72.
204. *Чернецки К., Айзенекер У.* Порождающее программирование. Методы, инструменты, применение.– Издательский дом Питер. – М. – СПб. – Харьков. – Минск. – 2005. – 730 с.
205. *Черников А.* Теория и практика управления проектами // Компьютерное обозрение.– 2003.– №10.–с.24–39.
206. *Цейтлин Г.Е.* Введение в алгоритмику.–Изд. Фара, 1999.–310с.
207. *Цимбал А.А, Аншина М.Л.* Технологии создания распределенных систем. Для профессионалов. – Спб.: Питер, 2003. – 576 с.
208. *Шлеер С., Меллор С.* Объектно-ориентированный анализ: моделирование мира в состояниях. – Киев: Диалектика, 1993.– 238 с.

209. *Эммерих В.* Конструирование распределенных объектов. Методы и средства программирования интероперабельных объектов в архитектурах OMG/CORBA, Microsoft/ COM и Java/RMI. – М.: Мир, 2002. – 510 с.
210. *Beech D.* Modularity of computer languages // Software – practice and experience.– 1982.–12, № 10.–P. 929–958.
211. *Beynon-Davies P.* Software Engineering and Knowledge Engineering: Unhappy Bedfellows // Polytechn. of Wales.– Pontypridd. UK.– 1987.– 3, PT. 4.–P. 9–11.
212. *Bril R. J.* Safe Use of Partial Operations // Informatie (Netherlands), Philips, Eindhoven, Netherlands.– 1987.– 29, № 7/8.–P. 700–703, 706-709.
213. *Burstall R., Lampson B.* Kernel language for Abstract Data Types and Modules // Lecture Notes in Comput. Sci.– 1984.– 173. – P. 1–50.
214. *Campbell F.* The portable UCSD p. system // Microprocess and Microsyst– 1983.–7, № 8.–P. 394–398.
215. *Chen P.P.* The entity relationship model: toward a unified view of data // ACM Trans. Database System.– 1976.– P. 9–36.
216. *Chiang J.C.* Software in the Large // Afips Conf. Proc. Vol. 56. Nation. Comput. Conf. (Chicago. IL. USA, June 15–18 1987).–Chicago, 1987.– P. 473–490.
217. *Consel C.* From a program family to a domain-specific language // Symposium on Principles of Programming Languages, Charleston, SC, USA, ACM Press, 1993.– P 19–29.
218. *Danahue P.* On the semantics of data types // SIAM J. Comput.– 1979.– 8, N 4.– P. 546–560.
219. *Goel A.L.* Software reliability models: assumptions limitations and applicability // IEEE Trans. Software Engrg.–1985.–Vol. SE, № 11/12.–P. 23—45.
220. *Harders N.* Software Engineering as a Methodology // Electronic (West Germany).–1987.–36, № 21.–P. 160–163.
221. *Horowitz E., Munson B.* An Expensive View of reusable Software // IEEE Trans. Software,–1984.–2, № 5.—P. 477—487.
222. http://www.acm.org/education/curricula.html та http://computer.org/curriculum.
223. http://www.cwi.nl/ftp/CWIreports/SEN/SEN-E0309.pdf
224. *ISO/IEC 12207: 2002*.– Information technology – Software life cycle processes) Информационные технологии – Процессы жизненного цикла программного обеспечения.
225. *ISO/IEC 9126–2.* Infofmation Technology. - Software Quality Characteristics and metrics. – 1997.
226. *Jacobson I.* Object-oriented Software Engineering. A use case Driven Approach, Revised Printing.– New York: Addison Wesley, Publ. Co.– 1994.– 529c.
227. *Jacobson I., Griss M., Johnson P.* Software Reuse: Architecture, Process and organization for Business Success – Addison Wesley, Reading , MA, May 1997.– 501 p.
228. *Johnsson R.K., Wick J.D.* An Overview of the MESA Processor Architecture // Computer Architecture News.–1982.–№ 2. –P. 20-29.
229. *Jones C.* A short history of Function Points and Feature Points // Cambridge, Massachusetts: Software Productivity Research, Inc.– 1988.– 45 p.
230. *Lubars M.D., Harandi M.J.* Intelligent support for software application and design // IEEE expert.– 1986.–№ 4.–P. 33–41.
231. *Macleish K.S., Vennergrund D.A.* An expert system development life cycle model and its relevance to traditional software systems // 5 Ann. Int. Phoenix. Conf. Comput. And Comm. (Scottsdale, Ariz. March 26—28 1986: Conf. Proc.)—Washington (D. C), 1986,—P. 592—596.
232. *McCabe J.L.* A complexity measure // IEEE Trans. Software Engrg.– 1976. 2, № 4.–P. 308–320.
233. *Mclauchlan M.R.* Specification of VLSI Designs: A Software Engineering Approach // IEE Colloquium on'VLSI System Design: Specification and Synthesis' (London, UK, Oct. 29 1987).– London, IEE, 1987.–P. 32.
234. *Mernik M., Heering J., Sloane A*: When and how to develop domain-specific languages. Technical Report SEN-E0309, CWI, Amsterdam, 2003, Available from
235. *Misra P.N.* Software reliability analysis // IBM systems.– 1983.–22, № 9.–P. 262–270.
236. Model Driven Development and Software Product Lines. BigLever Software Inc. – 2007. http://www.biglever.com/technotes/mdd_spl.html?source=mdd
237. *Mohanty S.N.* Software cost Estimation: Present and Future, P. E. 1981.–l. –p. 27—31.
238. *Moranda P.B.* Predictions of software reliability during debugging // Proc. Reliability and Maint Symp. (Washington, June 1975).–Washington (D.C.).1975. –327—332.
239. *Muller K.* Fundamental Principle of Software Engineering: The Phases Model // Electronic (West Germany).–1987.–36, № 21.–P. 169–172.

240. *Musa S.D.* Validity of execution-time theory of software reliability / /IEEE Trans. Reliability.–1979.– 28, № 3.–P. 181 – 191.
241. *Northrop L.M.* SEI`s Software Product Line Tenets // IEEE Software.–2002.– v.19, № 4.– p.32–39.
242. *Okumoto K.* A statistical method for Software Quality Control // IEEE Trans. On Software Engrg.– 11, № 12.–1984.–P. 1424–1430.
243. *Parkinson R.* Emerging Trends in Software Engineering Tools // Tektronix UK LTD, Marlow, UK, Electron. Prod, des.–1987.–7.–P. 47–50.
244. *Penedo M.H., Berry D.M.* The use of a module interconnection specification capability in the Sara System design methodology // Comput. Sci. dent. UCLA.–1978.
245. *Rine D.C.* A method for increasing software productivity called object-oriented design with applications for AL // George Mason Univ. Zairfax. VA. USA,–Afips Conf. Proc. Nation. Comput. Conf. (Chicago JL USA, June 15–18 1987).–Chicago, 1987,– 56.– P. 432–441.
246. *Shooman M.L.* Software Engineering: Reliability, Development and Management // Prentice Hall. N. Y.–1983.–672 p.
247. *Stoneburner G., Goguen A., Feringa A.* Risk Management Guide for Information Technology Systems. Recommendations of the National Institute of Standards and Technology. – http://csrc.nist.gov/publications/nistpubs/800-30/sp800-30.pdf
248. *Symons C.* Software Sizing and Estimating: Mk II FPA // Chichester: John Willey &Sons Ltd.- 1991.-57 p.
249. *Szentes J.* Qualigraph – a software tool for Quality metrics and Graphic documentation // Proc. ESA Estec. software seminar.–NOORDWISN.– 1983.– P. 73–81.
250. *Tichy N.* Software development Control Based on Module interconnection // Proc. 4-th Int. Conf. on Software Engrg.–Munchen, 1979.–P. 565–577.
251. *Troy R., Miawad*. Assessment of Software Reliability Models // Ibid.–1986.– № 9.–P. 25–31.
252. *Ueno H.* Expert systems // Dept. system, engrg,– Tokyo: Denki Univ., Japan.–1987.–28, № 2.– P.147–157.
253. *Urban S.* Building intelligence into software tools // IEEE expert.–1986.– № 1.–P. 1–21.
254. *Wegner P.* Capital – intensive software technology // IEEE Trans. Software, Engrg.– 1984.– 1, №4.– P. 7–45.
255. *Wegner P.* Capital Intensive Technology // Ibid.–1984.–P. 21–30.
256. *Wegner P.* Interaction Foundation of Object-oriented Programming// ECOOP-97. – Finland, 1997. – С. 123 – 139.
257. *Whitmire S.* An Introduction to 3D Function Points // Sottware Development –1995.- V.3, №.4.– P. 43-53.

ПРИЛОЖЕНИЕ 1

ОНТОЛОГИЧЕСКАЯ СИСТЕМА «ОНТОАСПЕКТ» РЕПОЗИТАРИЯ КПИ

Система “Онтоаспект” реализована аспиранткой отдела С.Л.Поляничко. Она обеспечивает информационную поддержку многоаспектного представления специфицированных пошуковых образов КПИ в онтологической модели репозитария в целях их аттестации, ведения, поиска и принятия решения об использовании КПИ в новых ПС, а также практического конструирования онтологий различных доменов (например, графика, деловодство и др.). Здесь дано краткое описание основных положений, структуры и примеров меню данной системы.

Функции системы. Система имеет три уровня:

1-уровень определяет представление готовых КПИ и артефактов, полученных на соответствующих процессах ЖЦ разработки доменов в HTML формате.

2-уровень отображает спецификации артефактов и КПИ первого уровня и задаются в виде поисковых образов (ПОБ) в фасетном виде;

3-рівень – база знаний, заданная совокупностью предметных онтологий доменов средствами объектно-ориентированных средств языка Java.

Основными функциями системы «Онтоаспект» являются следующие:

– построение онтологий в виде графов с объектами домена в вершинах;

– аннотирование артефактов в терминах онтологий последовательностью ключевых слов ПОБ, поисковых запросов к КПИ и артефактам репозитария;

– служба синонимии лексики домена, совместимой с лексическими понятиями других доменов;

– ведение словаря синонимов для концептов онтологии системы;

– изменение или дополнение списка словаря онтологии с учетом синонимии;

– построение ПОБ и ПОЗ в соответствии с выбранной классификацией объектов онтологии (компоненты, КПИ, аспекты, фреймы и пр.).

Структура системы «Онтоаспект» представлена на рис.П1.1 и включает следующие компоненты:

– репозитарий КПИ,

– классификатор компонентов,

– базу знаний фасет аспектов по каждому КПИ домена,

– словарь домена,

– хранилище кодового представления артефактов,

– панель управления и набор панелей для текущего выполнения операций системы, соответствующих функциям системы.

Взаимодействие пользователя с системой осуществляется через следующие операции:

– создание графического эскиза аспекта;

– сохранение файла аспекта в GML нотации;

– отображение GML (Graph Modelling Language) нотации в графическом виде;

– сохранение нормированной лексики аспекта;

– редактирование аспекта в графическом режиме;

– удаление аспекта;

– копирование аспекта КПИ в другой домен.

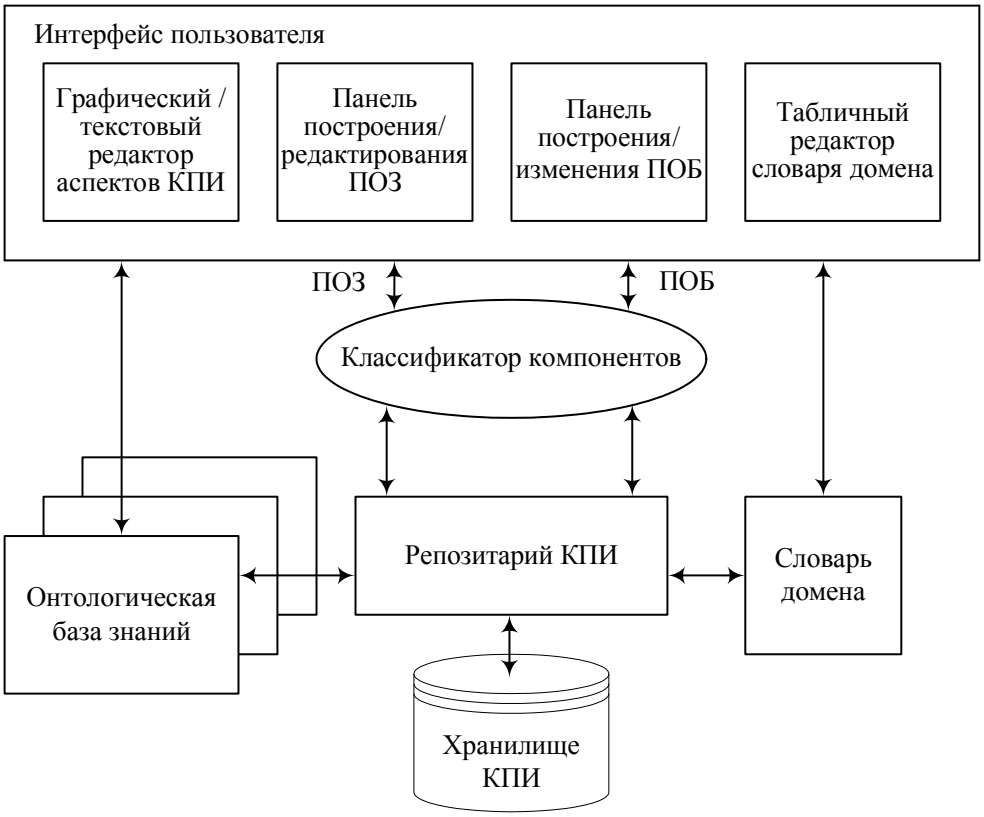


Рис.П1.1. Архитектура системы „Онтоаспект”

Среда реализации системы “Онтоаспект”. Алгоритмы реализации функций системы выполнены в языке JAVA for Forte, Release 3.0 с использованием библиотеки визуализации графов Graphlet. Для моделирования онтологий используется язык GML, с помощью которого задается формат описания графов доменов в виде GML-файла, состоящего из списков пар „ключ – значение”.

Общение пользователя с системой осуществляется через набор экранных форм – панелей, содержащих набор необходимых операций для выполнения заданной на панели функции. Главная панель содержит перечень функций, каждая из которых инициирует соответствующую панель функций системы. Пример онтологии (рис.П1.3) и базовых панелей и их назначение приведено на рис.П1.2, П1.4 и П1.5.

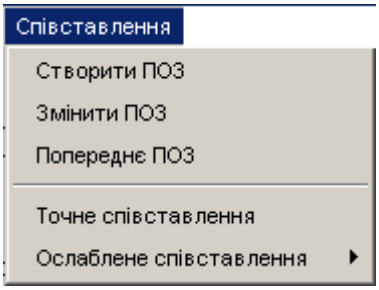


Рис.П1.2 Меню данной функции

Функция “Сравнение” позволяет выполнить запросы, создать ПОБ, осуществить упрощенное сравнение с образом, который имеется в репозитарии на релевантность с помощью операций панели меню этой функции.

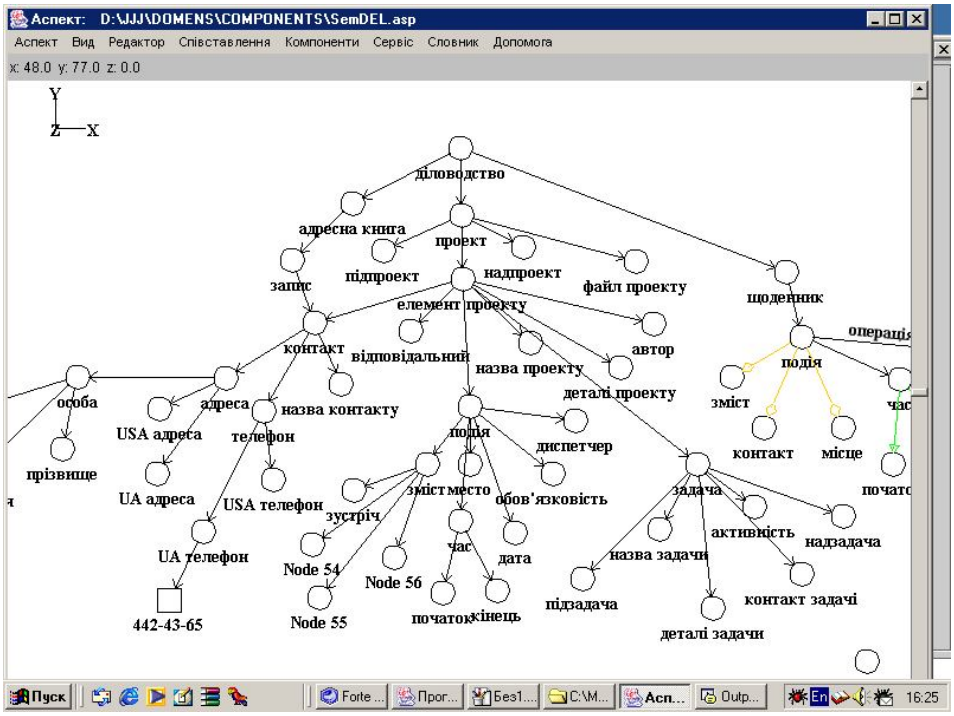


Рис П1.2. Фрагмент онтологии аспекта “Semantic” домена “Деловодство”

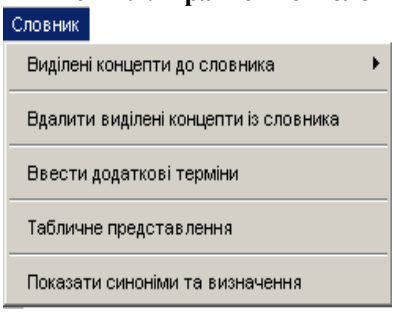
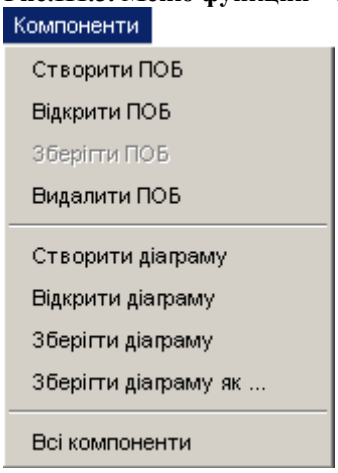


Рис.П1.3. Меню функции "Словарь"

Функция “Словарь” представлена несколькими строками панели и позволяет осуществить нормализацию лексики в домене с помощью операций “Выделить концепты словаря” или “Концепты с синонимами и определениями, которые можно изменить, удалить концепты или внести новые.



Меню функции “Компоненты” содержит несколько строк (рис.11.4) и предназначена для создания и накопления многоаспектных образов компонентов ПОБ для их занесения в директорий репозитария системы. Допускается модификация старого ПОБ, создание, хранение диаграмм классов и сценариев для аттестованных компонентов, которые могут использоваться в разработке новых ПС.

Рис.П1.4. Меню функции "Компоненты"

ПРИЛОЖЕНИЕ 2

АЛГОРИТМ РЕАЛИЗАЦИИ ВЕБ-ПРИЛОЖЕНИЯ НА ОСНОВЕ МЕНЕДЖЕРОВ ИНФОРМАЦИОННЫХ РЕСУРСОВ

Веб-приложение строится методом интеграции менеджеров ИР. Менеджер ресурсов описан (В.Н.Грищенко) как Java-компонент соответственно общей его структуры (см. рис. 5.4). При этом использованы следующие условия для ресурсов.

Дескрипторы ресурсов подаются как XML-файлы, которые определяют основные характеристики для создания соответствующих соединений с ресурсами, и их функциональность. Компонент, который обеспечивает обработку данных по дескриптору, использует DOM-модель при описании XML-файлов.

Перечни всех ресурсов составляющих общую систему ИР также представлены в виде XML-файлов, распределенных между несколькими веб-серверами, на каждом из которых развернут отдельный экземпляр веб-приложения для навигации и доступа к ИР. Каждый перечень определяет совокупность ресурсов, связанных с конкретным сервером.

Унифицированный запрос определяется стандартным для веб-среды CGI-механизмом. Формат ответа на унифицированный запрос – HTML-страница.

Структуры типизированных запросов к ИР и ответам на них зависят от типа избранного менеджера ресурсов.

Менеджер ресурсов для веб-сайтов на основе унифицированного запроса отыскивает дескриптор ресурса, по данным которого формирует адрес целевого сайта и выполняет переадресацию согласно сформированному адресу. Типовой дескриптор для этого типа менеджеров ресурсов имеет следующий вид:

```
<?xml version="1.0" encoding="WINDOWS-
  1251"?>
<ir_manager_description>
<example_name> Web-Сайт ІПС
</example_ name>
  <example_id>ir_manager_description_for_iss_site
  </example_id>
  <description> Web-Сайт Института программных
   систем НАНУ</description>
  <manager_type> web-site</manager_type>
  <location>
    <url>www.isofts.kiev.ua:80</url>
    <protocol>http</protocol>
  </location>
</ir_manager_description>
```

В этом дескрипторе приведено наиболее важные и наиболее употребляемые характеристики описания ресурса:

– имя;

– уникальный идентификатор в системе представления ресурсов;

– назначение;

– тип менеджера;

– адрес и протокол доступа к менеджеру.

Менеджер ресурсов для веб-приложений на основе унифицированного запроса отыскивает соответствующий дескриптор ресурса и описание метода доступа к нему. Метод доступа определяется совокупностью параметров и их значений. После этого устанавливается адрес целевого сайта, где находится веб-приложение, формируется множество CGI-параметров и их значений, выполняется переадресация согласно сформированному адресу. Пример типичного дескриптора для этого типа менеджеров ресурсов:

```
<?xml version="1.0" encoding="WINDOWS-
  1251"?>
<ir_manager_description>
<example_name>Работа с Web-
  приложением</example_name>
<example_id>ir_manager_description_for_app
  </example_id>
<description>Моделирует работу с определенным
  Web-Приложением</description>
<manager_type> web-application
</manager_type>
<location>
   <url>www.server.isofts.kiev.ua:
     8080/app/test</url>
   <protocol>http</protocol>
</location>
<request_list>
<request>
     <description>Построение дерева информационныхресурсов</description>
     <parameter_list>
       <parameter>
          <parameter_name>name
          </parameter_name>
       <parameter_value>*
       </parameter_value>
       </parameter>
       <parameter>
          <parameter_name>object
          </parameter_name>
          <parameter_value>tree
          </parameter_value>
       </parameter>
       <parameter>
          <parameter_name>type
          </parameter_name>
          <parameter_value>tree
          </parameter_value>
       </parameter>
```

```
    </parameter_list>
  </request>
  <request>
    <description>Поиск ресурсов соответственно
     поисковому слову</description>
    <parameter_list>
      <parameter>
        <parameter_name>type
        </parameter_name>
        <parameter_value>search
        </parameter_value>
      </parameter>
      <parameter>
        <parameter_name>object
        </parameter_name>
        <parameter_value>*</parameter_value>
      </parameter>
    </parameter_list>
  </request>
</request_list>
</ir_manager_description>
```

Это веб-приложение находится на веб-сервере и реализует две прикладные функции: построение дерева ИР и поиск ресурсов словом, которое входит в их описание. Для каждой функции существует дескриптор запроса, который определяет CGI-параметры и их значения.

Менеджер ресурсов для отдаленного доступа к БД на основе унифицированного запроса отыскивает соответствующий дескриптор ресурса и описание метода доступа к ресурсу. Как и для веб-приложения, метод доступа определяется совокупностью параметров и их значений. На основе этих параметров формируется SQL-запрос. Дескриптор ресурса содержит данные об JDBC-драйвер, который обеспечивает связь с ресурсом. Драйвер загружается и инициируется для доступа к соответствующей БД. Сформированный SQL-запрос передается JDBC-драйверу, который организует его выполнение. Результатом запроса является структура данных, которая описывает таблицу с результатами обработки SQL-запроса. На основе этой структуры данных генерируется фрагмент HTML-страницы, которая пересылается клиенту. Пример типового дескриптора для менеджера ресурсов имеет следующий вид:

```
<?xml version="1.0" encoding="WINDOWS-
  1251"?>
<ir_manager_description>
<example_name>Доступ к БД
</example_name>
<example_id >ir_manager_description_for_app_db
</example_id>
<description>Моделирует работу по доступу к базам данных</description>
```

```
<manager_type> web-application-db
</manager_type>
<location>
  <url>localhost:8080/app/test</url>
  <protocol>http</protocol>
</location>
<db_connection_parameters>
  <driver>com.mysql.jdbc.Driver</driver>
  <url>jdbc:mysql://localhost:3306/test</url>
  <username>root</username>
  <password></password>
</db_connection_parameters>
<request_list>
  <request>
    <description>Запрос к БД</description>
    <parameter_list>
      <parameter>
        <parameter_name>type
        </parameter_name>
        <parameter_value>db
        </parameter_value>
      </parameter>
      <parameter>
        <parameter_name>name
        </parameter_name>
        <parameter_value>ir_manager_
          description_for_app_db
        </parameter_value>
      </parameter>
      <parameter>
        <parameter_name>statement
</parameter_name>
        <parameter_value></parameter_value>
      </parameter>
    </parameter_list>
  </request>
</request_list>
</ir_manager_description>
```

Структура дескриптора в целом аналогична приведенному для предыдущего случая. Дополнительно он описывает параметры JDBC-драйвера для MySQL, т.е. целевая БД как ИР создана и поддерживается этой СУБД. При необходимости может быть применена другая СУБД, например Oracle. В этом случае нужно изменить данные из дескриптора относительно JDBC-драйвера, который поддерживает связь с Oracle. Другие компоненты менеджера доступа к ресурсам изменений не требуют.

ИНТЕРФЕЙС ВЗАИМОДЕЙСТВИЯ ПРОГРАММ В ЯЗЫКАХ ПРОГРАММИРОВАНИЯ ДЕЛЬФИ И ПАСКАЛЬ

Дельфи (Delphi) – ЯП, производный от языка Турбо-Паскаль, он используется в одноименной среде разработки. ЯП Дельфи и Паскаль были расширены объектно-ориентированными свойствами и способами доступа к метаданным классов. Все классы наследуют функции базового класса, а объекты могут располагаться в динамической памяти.

Для обеспечения взаимодействия программ в этих ЯП, как правило, используют язык Ассемблера, который осуществляет:

- оптимизацию критичных к скорости участков программ в ЯП;
- создание ОС (например, версия Unix);
- создание драйверов, антивирусов и программ защиты;
- написание трансляторов ЯП.

Программа в этом языке является интерфейсной программой и в ней самостоятельно решаются вопросы преобразования типов данных.

Рассматриваемые языки содержат простые, строковые, структурные, указательные, процедурные и вариантные (хранения данных различных типов) типы данных. Остановимся на сопоставлении типов данных этих ЯП.

Простые типы данных

К ним относятся: порядковые, целые, символьные, булевы перечислимые, поддиапазонные, действительные.

В Object Pascal простые типы данных разбиты на две группы: порядковые для представления данных разных объемов и действительные, обеспечивающие приближенные математические вычисления.

Порядковые типы. Из простых типов данных порядковые определяет естественные отношения порядка между элементами. В Object Pascal определены три группы порядковых типов и два типа, определяемых пользователем. К группам относятся целые, символьные и булевы типы. Первый элемент любого порядкового типа имеет номер 0, второй элемент – номер 1 и т.д.

Целые типы. Эти типы в Object Pascal подразделяются на физические (фундаментальные) и логические (общие). В Object Pascal определены следующие целые типы: integer, shortint, smallint, longint, byte, word, cardinal. Их список, диапазон и формат представлены в следующей таблице.

Тип	Диапазон значении	Физический формат
Shortint	-128-127	8 бит, со знаком
Smallint	-32 768-32 767	16 бит, со знаком
Longint	-2 147 483 648-2 147 483 647	32 бит, со знаком
Byte	0-255	8 бит, без знака
Word	0-65 535	16 бит, без знака

Диапазоны значений и форматы физических целых типов не зависят от микропроцессора и операционной системы, в которых выполняется программа. Они не меняются (или, по крайней мере, не должны меняться) с изменением реализации или версии Object Pascal.

Диапазоны значений логических целых типов (Integer и Cardinal) определяются совершенно иным образом. Как видно из следующей таблицы, они не связаны с диапазонами соответствующих физических типов.

Тип	Диапазон значений	Физический формат
Integer	-32 768-32 767	16 бит, со знаком (SmallInt)
Integer	-2 147 483 648-2 147 483 647	32 бит, со знаком (Longint)
Cardinal	0-65 535	16 бит, без знака (Word)
Cardinal	0-2 147483647	32 бит, без знака (Longint)

Символьные типы. Смысл символьных данных понятен, когда они выводятся на экран или принтер. Тем не менее, определение символьного типа может зависеть от того, что подразумевать под словом символ. Для совместимости со всеми этими представлениями в Object Pascal определены два физических символьных типа и один логический. Физические типы перечислены ниже.

AnsiChar — Однобайтовые символы, упорядоченные в соответствии с расширенным набором символов ANSI

WideChar — Символы объемом в слово, упорядоченные в соответствии с международным набором символов UNICODE. Первые 256 символов совпадают с символами ANSI

Символьные типы объемом в двойное слово (32 бит) отсутствуют.

Булевы типы. По аналогии с целыми и символьными типами, подразделяющимися на физические и логические, естественно предположить, что ByteBool, WordBool и LongBool – физические типы, Boolean – логический. Они представлены в следующей таблице.

Тип	Размер
Boolean	1 байт
ByteBool	1 байт
WordBool	2 байт (объем Word)
LongBool	4 байт (объем Longint)

Все четыре типа различны. Для Object Pascal предпочтителен тип Boolean, остальные определены для обеспечения совместимости с другими ЯП и операционными системами.

Переменным типа Boolean можно присваивать только значения True (истина) и False (ложь). Переменные ByteBool, WordBool и LongBool могут принимать и другие значения, интерпретируемые обычно как False в случае нуля и True - при любом ненулевом значении.

Перечислимые типы. Тип последовательность состоит из набора имен**,** задающих дискретные значения. Синтаксис определения имеет вид: Type enum type = (first value, value2, value3, last value). Тип Boolean – простейший перечислимый тип в Object Pascal. Булевы переменные могут принимать два значения, выражаемые именами True и False, а сам тип определен в Object Pascal следующим образом:

Type Boolean = (False, True).

С помощью типа Boolean в Object Pascal выполняют операции сравнения. Большинство перечислимых типов - это списки уникальных имен или идентификаторов, зарезервированных в конкретных целях.

Поддиапазонные типы. Переменные поддиапазонного типа содержат информацию, соответствующую некоторому заданному диапазону значений исходного типа, представляющего любой порядковый тип. Синтаксис определения поддиапазонного типа имеет следующий вид:

Type subrange type = low value...high value;

Поддиапазонные переменные сохраняют особенности исходного типа. Единственное отличие состоит в том, что переменной поддиапазонного типа можно присваивать только значения, входящие в заданный поддиапазон.

Действительные типы. Данные типы, их пороги и значащие цифры в байтах приведены в следующей таблице.

Тип	Порог	Максимальное значение	Количество значащих цифр	Объем (байт)
Real	2.9E-39	1.7E38	11-12	6
Single	1.5E-45	3.4E38	7-8	4
Double	5.0E-324	1.7E308	15-16	8
Extended	3.4E-4932	1.IE4932	19-20	10
Comp	1.0	9.2E18	19-20	8
Currency	0.0001	9.2E14	19-20	8

Строковые типы

В языке Delphi поддерживается три физических строковых формата: короткий (ShortString), длинный (LongString) и широкий (WideString). Их можно комбинировать в операторах присваивания и выражениях. Все необходимые преобразования выполняются автоматически. Они представляют динамически распределяемые массивы символов. Обычно вполне достаточно одного типа AnsiString. При работе с международными наборами символов, такими как UNICODE, удобнее использовать WideString.

Тип ShortString – массив Array [0..255] of char. Первый его элемент задает динамическую длину строки, которая может принимать значения от 0 до 255 символов. Символы, составляющие строку, занимают места от 1 до 255. Логический строковый тип – это тип String. Отнесение его к типу AnsiString или ShortString задается командой $H. По умолчанию задается { $H+}, и String совпадает с AnsiString. Если задать команду {$H}, то String будет совпадать с ShortString и иметь максимальную длину, равную 255 символам.

Структурные типы

К структурным типам относятся: записи, массивы, множества, файлы, классы, указатели на классы.

При этом перечисленные типы являются не типами, а структурными типы выступают в роли методов, дополняющих существующие типы.

Записи. С помощью зарезервированного слова record (запись) в одном типе можно объединять данные разных типов. Общий синтаксис объявления этого типа выглядит следующим образом:

```
record
fieldnamel: fieldtypel;
fieldname2, fieldname3: fieldtype2;
case optional tagfield: required ordinal type of
1: variantnamel: varianttype3;
2, 3: variantname2: varianttype4;
end;
```

Данное описание состоит из фиксированной и вариантной частей. Однако не обязательно вставлять в одно описание записи обе эти части. Обычно удобнее работать с каждой из этих частей отдельно.

Фиксированные записи определяют одно или несколько независимых полей. Каждому полю присваивается имя и тип:

```
record
fieldnamel: fieldtypel;
fieldname2, fieldname3: fieldtype2;
end;
```

Имея доступ к информации в записи, можно обрабатывать всю запись целиком (все поля одновременно) или только отдельное поле. Для обращения к отдельному полю наберите имя записи, точку и идентификатор поля, например

```
MyRec.Fieldnamel.
```

Для доступа ко всей записи указывается ее имя.

Вариантная запись – это вариантная часть типа record, которая дает возможность по-разному трактовать область памяти, занимаемую совместно вариантами поля:

```
record
case optional tagfield: required ordinal type of
1: variantnamel: varianttype3;
2, 3: variantname2: varianttype4;
end;
```

Массивы. Массивы могут быть одно- или многомерными, как в приведенном примере:

```
array [ordinal_type] of type_definition;
array [ordinal typel, ordinal type2] of type definition;
```

Каждый массив содержит определенное количество элементов информации одного типа. Для обращения к элементу массива указывается имя массива и индекс элемента, заключенный в квадратные скобки. Число элементов массива в каждом измерении задается порядковым типом (ordinal_type). Для этого используется идентификатор некоторого типа (например, Boolean или AnsiChar). На практике явно задается поддиапазон целых.

Количество элементов массива равно произведению количеств элементов во всех измерениях. Для обращения к элементу массива указывается имя этого массива и индекс элемента в квадратных скобках. Пусть, например, массив определен следующим образом: var MyArray: Array [1..10] of Integer;

Тогда обращение к его третьему элементу будет выглядеть, как MyArray [3], и выполняться, как к переменной Integer.

Множество. Слово set (множество) определяет множество не более, чем из 256 порядковых значений: Set of ordinal type.

Минимальный и максимальный порядковые номера исходного типа (на основе которого определяется множественный тип) должны быть в пределах между 0 и 255. Переменная множественного типа содержит (или не содержит) любое значение исходного порядкового типа. Каждое значение из заданного диапазона может принадлежать или не принадлежать множеству. Рассмотрим следующий пример.

```
Type CharSet = set of AnsiChar; // Тип множества символов. ANSI.
var MyAlphaSet: CharSet; // Переменная типа CharSet.
```

Переменная set может содержать все элементы множества или не содержать ни одного. При присвоении значения переменной множественного типа элементы множества (порядковые значения) указываются в квадратных скобках:

```
MyAlphaSet := ['A', 'E', 'Г, 'O', 'U', 'Y']; // Все прописные гласные.
```

Пустые квадратные скобки задают пустое множество, не содержащее ни одного элемента. Это относится ко всем множественным типам.

Файловый тип. Тип file представляет линейную последовательность элементов, которые могут быть данными любого типа, кроме типов file и class. Описание файлового типа аналогично объявлению массива, только без указания числа элементов.

```
file of Typel  // Файл определенного типа, содержащий
               // записи фиксированной длины.
file   // Файл без типа или "блочный".
textfile  // Файл с записями переменной длины, разделенными символами CR
          // и LF ("возврат каретки" и "новая строка").
```

Механизм ввода-вывода информации зависит от языка и реализации. В большинстве случаев предполагается, что программисту незачем вникать во внутреннюю структуру переменных, управляющих вводом-выводом, и при передаче информации следует полностью полагаться на специальные процедуры. Их реализация подобна «черному ящику». В basic файлы обозначаются числовыми значениями – дескрипторами. В C/C++ программисты манипулируют указателями на структуру FILE, а в Delphi файловая структура является переменной.

Указательные типы

Переменная указательного типа содержит значение, указывающее на переменную типа –адрес этой переменной.

```
pointer   // Указатель без типа.
^typel    // Указатель с типом.
```

Если исходный тип (тип переменной, на которую должен ссылаться указатель) еще не объявлен, его надо объявить в том же разделе описания, что и тип указателя.

Процедурный тип

Данный тип включает: тип Variant, вариантные значения, процедуры обработки вариантных массивов, OLE Automation для изменения свойств объектов.

Тип Variant. Данный тип предназначен для представления значений, которые могут динамически изменять свой тип. Если любой другой тип переменной фиксированы, то в переменные типа Variant можно вносить переменные других типов. Этот тип применяется в тех случаях, когда фактический тип данных изменяется или неизвестен в момент компиляции.

Вариантные значения. Ранее рассмотренный тип Record имеет вариантную часть записи, где в одном фрагменте памяти хранится информация нескольких типов. Возникает необходимость иметь в памяти действительное значение с фиксированной запятой, которое может интерпретироваться, как целое. Переменным типа Variant можно присваивать значения типов – целое, действительное, строковое и булевое. Для совместимости с другими ЯП предусмотрена возможность присвоения их переменным значения даты/времени и объектов типа OLE Automation. Кроме того, вариантные переменные могут содержать массивы переменной длины и размерности с элементами указанных типов.

Все целые, действительные, строковые, символьные и булевы типы совместимы с типом Variant в случае операции присваивания. Вариантные переменные могут сочетаться в выражениях с целыми, действительными, строковыми, символьными и булевыми переменными; при этом все необходимые преобразования в системе Delphi выполняет автоматически. Можно произвольно задавать выражение типа Variant в форме Variant (X).

В следующей таблице перечислены типы значении, которые можно присваивать вариантным переменным, и вариантные типы результата.

Тип выражения	Вариантный тип
Целый	varlnteger
Действительный, кроме Currency	varDouble
Currency	varCurrency
Строковый и символьный	varString
Булев	varBoolean

Вариантные переменные в отношении операции присвоения совместимы с элементарными типами данных Object Pascal (Integer, Real, String и Boolean). Все необходимые преобразования в системе Delphi выполняются автоматически. Если необходимо указать, что вариантное значение интерпретируется как целое, действительное, строковое или булево, следует задать тип в форме TypeName (V), где TypeName – идентификатор соответствующего типа, V– выражение Variant. Задание типа изменяет способ считывания значения из вариантной переменной, а не само значение. Внутреннее представление изменяется с помощью процедур VarAsType и VarCast.

OLE Automation. Вариантные переменные удобно применять для изменения свойств объектов OLE Automation и вызова методов этого объекта. Чтобы инициировать эту возможность, подключают модуль OleAuto.

Синтаксис вызова метода или обращения к свойству объекта OLE Automation такой же, как вызов из созданного класса. Имеется несколько отличий. Во-первых, вызов метода объекта OLE Automation происходит по схеме позднего связывания, т.е. компилятор не проверяет, существует ли данный метод и правильно ли определены типы параметров. Для компилятора приемлем любой идентификатор метода и любое число параметров разных типов. При выполнении вызванного таким образом метода может произойти ошибка.

Пример описания фактариала числа в Программе1 в языке Pascal, в Программе2 в языке Delphi и модуля интерфейса Unit1 демонстрирует принципы взаимодействия разноязыковых программ**.** В этих программах используются типы

данных, свойственные каждому из ЯП. Программа **Unit1 осуществляет роль интерфейсного посредника между Программа1 и Программа2.**

Программа 1 в языке Pascal	**Программа Unit1**	**Программа 2 на Delphi**
program pr1; *uses Crt;* *var fact, i, N : longint;* *begin* *clrscr;* *writeln ('Vvedit N: ');* *readln (N);* *fact:=1;* *for i:=1 to N do* *begin* *fact:=fact*i;* *end;* *writeln('Factorial 4isla ',* *N, ' = ', fact);* *readln;* *end.*	*unit Unit1*; **interface** uses *Windows, Messages, SysUtils,* *Variants, Classes, Graphics, Controls,* *Forms, Dialogs, StdCtrls;* *type* *TForm1 = class(TForm)* *Label1: TLabel;* *Edit1: TEdit;* *Label2: TLabel;* *Edit2: TEdit;* *Button1: Tbutton* *procedure Button1Click(Sender:* *TObject);* *private {Private declarations}* *public {Public declaration}* *var Form1: TForm1;* *implementation {$R *.dfm}* *procedure* *TForm1.Button1Click(Sender:* *TObject);* *var i, fact : Integer;* *begin* *fact:=1;* *for i:=1 to StrToInt(Edit1.Text) do* *begin* *fact:=fact*i;* *end;* *Edit2.Text:=IntToStr(fact);* *end; end.*	*program Project1;* *uses Forms,* *Unit1 in 'Unit1.pas'* *{Form1};* *{$R *.res}* *begin* *Application.Initialize;* *Application.CreateForm(TFor* *m1, Form1);* *Application.Run;* *end.*

Исследование концепции взаимодействия компонентов в языках Дельфи и Паскаль провели студенты 4 курса (С.Балаклеенко и Л.Зинченко) кафедры информационных технологий Киевского Национального университета им. Тараса Шевченко.

Наукове видання

Національна Академія наук України
Інститут програмних систем

ЛАВРІЩЕВА Катерина Михайлівна
ГРИЩЕНКО Володимир Миколайович

ЗБІРКОВЕ ПРОГРАМУВАННЯ. Основи індустрії програмних продуктів

Підп. до друку 5.10.2009р. Формат 70×100 1/16
Папір офс.№1. Друк.різограф. Обл.–вид.арк.30,72
Умв. друк.арк. 26. Тираж 300прим. Замовлення № 2282.

Друкарня «Видавничого дому «Академперіодика» НАН України
01004, Київ–4, вул. Терещенківська, 4
Свідоцтво про внесення до Державного реєстру суб'єкта
Видавничий справи серії ДК №544 від 27.01.2001.