Е. М. Лаврищева

SOFTWARE ENGINEERING КОМПЬЮТЕРНЫХ СИСТЕМ

ПАРАДИГМЫ, ТЕХНОЛОГИИ, CASE-СРЕДСТВА ПРОГРАММИРОВАНИЯ

НАЦИОНАЛЬНАЯ АКАДЕМИЯ НАУК УКРАИНЫ

Институт программных систем

Е. М. ЛАВРИЩЕВА

SOFTWARE ENGINEERING КОМПЬЮТЕРНЫХ СИСТЕМ

ПАРАДИГМЫ, ТЕХНОЛОГИИ, CASE-СРЕДСТВА ПРОГРАММИРОВАНИЯ

Киев
Наукова думка
2013

УДК 004.41
Software Engineering компьютерных систем. Парадигмы, технологии и CASE-средства программирования/ Е. М. Лаврищева. –К.:Наук. думка, 2013. – 283 с.

В монографии рассмотрены парадигмы программирования и Case-средства для разработки сложных компьютерных систем из программных ресурсов данных парадигм. В *первом разделе* даны базовые понятия программной инженерии, технологии программирования и метода сборки разноязычных модулей в сложные системы, а также компьютерные средства их автоматизации и реинженерии ресурсов и систем. Во *втором разделе* приведены новые формализмы парадигм программирования (модульной, объектной, компонентной, аспектной и сервисной) в программной инженерии. Каждая парадигма представлена теоретическим аппаратом моделирования и проектирования соответствующего ресурса. Дано формальное описание метода сборки ресурсов этих парадигм в сложные системы с инструментами их поддержки. *В третьем разделе* описаны разработанные технологии, линии и CASE-средства поддержки парадигм средствами процессов жизненного цикла и инженерии качества. Представлен оригинальный набор CASE-инструментов – линии изготовления компонентов, сборки их в конфигурационные структуры, а также линии обучения языкам C#, JAVA, VBasic описания ресурсов и аспектам предмета программной инженерии в среде веб-сайтов ИТК и фабрики программ КНУ.

Для разработчиков и специалистов, занимающихся теоретическими и прикладными вопросами проектирования и реализации сложных компьютерных систем, а также для студентов высших учебных заведений по специальности программная инженерия, компьютерные науки и информатика.

Рецензенты:
д-р физ.-мат. наук, проф. Н. С.Никитченко,
(Киевский национальный университет имени Тараса Шевченко),
д-р физ.-мат. наук, проф. М. М. Глибовец
(Национальный университет "Киево-Могилянская академия")
д-р техн. наук, проф. С. А. Лукьяненко
(Национальный технический университет Украины
"Киевский политехнический университет")

Утверждено к печати ученым советом Института программных систем НАН Украины (Протокол № 12 от 19.12.2013)

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

ISBN 978-966-00-1416-9

ОГЛАВЛЕНИЕ

СПИСОК СОКРАЩЕНИЙ 7

ОТ АВТОРА 8

ПРЕДИСЛОВИЕ 11

Раздел 1
ПРОГРАММНАЯ ИНЖЕНЕРИЯ. БАЗОВЫЕ ПОНЯТИЯ 14

Глава 1. СТАНОВЛЕНИЕ ПРОГРАММНОЙ ИНЖЕНЕРИИ 14

1.1. Определение программной инженерии с 70–90-х годов XX столетия 15
1.2. Основные понятия программной инженерии 18
1.3. Принципы программной инженерии 24
1.4. Управление разработкой и качеством систем 28
1.5. Реинженерия, реверсная инженерия, рефакторинг 30
1.6. CASE-средства программной инженерии 33

Глава 2. СТАНОВЛЕНИЕ ОТЕЧЕСТВЕННОЙ ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ 34

2.1. Технологии компьютерных систем и программ 35
2.2. Формирование сборочной технологии программирования в бывшем СССР 37
2.3. Развитие индустриальных технологий в программной инженерии 38

Глава 3. КОМПЬЮТЕРНЫЕ ТЕХНОЛОГИИ ФАБРИК ПРОГРАММ 39

3.1. Зарубежные компьютерные технологии 39
3.2. Индустриальные основы программной инженерии 41
3.3. Дисциплины программной инженерии 43
3.4. Современные фабрики программ. Типы, ресурсы, платформы 49

Глава 4. ТЕХНОЛОГИЯ КОНВЕЙЕРНОЙ СБОРКИ СИСТЕМ 63

4.1. Сущность сборочного конвейера 66
4.2. Линии программ и Product Lines 66
4.3. Метод сборки специализированных технологий 68

Раздел 2
ПАРАДИГМЫ ПРОГРАММИРОВАНИЯ 70

Глава 1. МОДУЛЬНОЕ ПРОГРАММИРОВАНИЕ. БАЗОВЫЕ ПОНЯТИЯ 70

1.1. Понятие модуля и интерфейса. Метод их сборки 71
1.2. Теория сборки разнородных модулей 76
1.3. Фундаментальные типы данных (ТД). Простые и сложные ТД 86
1.4. Общие типы данных. Неструктурные и генерированные ТД 91
1.5. Стили сборочного программирования 98
1.6. CASE-средства интеграции модулей и интерфейсов 102

Глава 2. ПАРАДИГМА ОБЪЕКТНОГО ПРОГРАММИРОВАНИЯ 107
2.1. Математическое моделирование объектной модели 108
2.2. Алгебра объектного анализа предметной области (ПрО) 118
2.3. Методы объектов и их интерфейсы 119
2.4. ЖЦ объектного моделирования ПрО 124
2.5. CASE-средства объектного подхода в современных средах 125

Глава 3. ПАРАДИГМА КОМПОНЕНТНОГО ПРОГРАММИРОВАНИЯ 127

3.1. Теория компонентного программирования. Базовые понятия 129
3.2. Модели разработки систем из компонентов 135
3.3. Операции внешней, внутренней и эволюционной алгебры 137
3.3. Объектно-компонентный метод 146
3.4. Типизация компонентов. Корректность сборки компонентов 149
3.5. ЖЦ компонентной разработки ПС 152
3.6. CASE-средства поддержки компонентов и систем 155

Глава 4. ГЕНЕРИРУЮЩЕЕ ПРОГРАММИРОВАНИЕ. МОДЕЛИ И МЕТОДЫ 158

4.1. Элементы программных систем и семейство систем 161
4.2. Трансформация и конфигурация программных систем 162
4.3. Аспектно-ориентированное программирование 164
4.4. Модели взаимодействия систем. Теория и реализация 171
4.5. Модель конструирования вариантных систем и семейств 179
4.6. Модели сложных и распределенных систем 181
4.7. CASE-системы поддержки мультипрограммирования 184

Глава 5. СЕРВИСНОЕ ПРОГРАММИРОВАНИЕ 185

5.1. Сервис. Базовые понятия 185
5.2. Сервисно-ориентированная архитектура 189

5.3. Сервисы контрактов WCF 191
5.4. CASE-средства JAVA EE 193

Раздел 3
ТЕХНОЛОГИЯ СИСТЕМ, ЛИНИЙ И CASE-СРЕДСТВ 197

Глава 1. ТЕХНОЛОГИЯ СЛОЖНЫХ СИСТЕМ ИЗ ГОТОВЫХ РЕСУРСОВ 197

1.1. Базовые подходы к проектированию сложных систем 199
1.2. Модели систем для разных платформ 201
1.3. Генерация и сборка сложных систем 203
1.4. Методология проектировании систем с помощью ЖЦ 204

Глава 2. МОДЕЛИРОВАНИЕ ДОМЕНОВ СРЕДСТВАМИ ОНТОЛОГИИ 207

2.1. Онтологическое моделирование проблемной области 208
2.2. Описание доменов средствами онтологии 210
2.3. Основные понятия онтологии представления ПрО 211
2.4. Формализация онтологической модели ЖЦ 213
2.5. Онтологии процесса тестирования ЖЦ 216

Глава 3. ОБЕСПЕЧЕНИЕ КАЧЕСТВА ПС 218

3.1. Основные задачи проблемы управления качеством 218
3.2. Моделирование характеристик качества ПС 221
3.3. Задачи управления качеством ПС 222
3.4. Модель требований с ориентацией на обеспечение качества ПС 223
3.5. Система прогнозирования безотказной работы ПС 225
3.6. Анализ достижения уровня качества 228
3.6. Задачи оценки качества сложных систем 229
3.7. Эталонная модель качества оценки показателей ПС 231

Глава 4. ТЕСТИРОВАНИЕ И ЭКСПЕРТИРОВАНИЕ ПС 236

4.1 Модель тестирования и определение оптимального времени 236
4.2. Экспертирование компонентов и систем 239
4.3. Методы управления программным проектом 243

Глава 5. CASE-СРЕДСТВА РАЗРАБОТКИ СЛОЖНЫХ СИСТЕМ 245

5.1. Классификация средств производства ПП 246
5.2. Ресурсы фабрики программ. Их виды и использование 247
5.3. Базовые основы средств индустрии программ 249
5.4. Разработка ТЛ для фабрик программ 251

Глава 6. CASE ИТК. ТЕХНОЛОГИИ, ЭЛЕКТРОННОЕ ОБУЧЕНИЕ 255

6.1. Основные задачи ИТК 256
6.2. Функции и структура веб-сайта ИТК 259
6.3. Описание раздела сайта "Технологии" 262
6.4. Веб-сервисы в ИТК 266
6.5. Раздел сайта "Взаимодействие" 267
6.6. Разделы сайта: Презентации, Инструменты 268
6.7. Электронное обучение предмету "Программная инженерия" 268

Глава 7. ПЕРСПЕКТИВА ПЕРЕХОДА ИТ-ТЕХНОЛОГИЙ К НАНОТЕХНОЛОГИЯМ 270

7.1. Оценка достижений компьютерных технологий 271
7.2. На пути к нанотехнологии 272

ЗАКЛЮЧЕНИЕ 275

СПИСОК ЛИТЕРАТУРЫ 276

СПИСОК СОКРАЩЕНИЙ

АИС	–	автоматизированная информационная система
АОП	–	аспектно-ориентированное программирование
АСУ ТП	–	автоматизированная система технологических процессов
АСУ	–	автоматизированная система управления
ГОР	–	Готовые ресурсы
ГКНТ	–	Государственный комитет по науке и технике
ЖЦ	–	жизненный цикл
ИВ	–	информационный вектор
ИС	–	информационная система
КП	–	комплекс программ
КТП	–	карта технологического процесса
МТЛ	–	модель технологической линии
МЯИ	–	межъязыковый интерфейс
ОКМ	–	объектно-компонентный метод
ОМ	–	объектная модель
ООП	–	объектно-ориентированный подход
ПИ	–	Программная инженерия
ПО	–	Программное обеспечение
ПП	–	программный продукт
ППП	–	пакет прикладных программ
ПС	–	Программная система
ПрО	–	предметная область
СА	–	системная архитектура
САА	–	система алгоритмических алгебр
АПРОП	–	система автоматизации программирования
СОД	–	система обработки данных
ССПМ	–	система сборки программ из модулей
ТЛ	–	технологическая линия
ТМ	–	технологический модуль
ТО	–	технологическая операция
ТП	–	технологический процесс
ТПР	–	технология подготовки разработки
ЯМК	–	язык модульного конструирования
ЯОТ	–	язык описания типов
ЯП	–	язык программирования
ACM	–	Association for Computing Machinery
CBD	–	Component-Based Development
CMM	–	Capability Maturity Models
SOA	–	Servise oriented architecture
CS	–	Computer science
GDM	–	Generative Domain Model
DSL	–	Domain specific language
SWEBOK	–	Software Engineering Body Knowledge
V&V	–	Верификация и валидация
PSM	–	Platform System Model
PIM	–	Platform Independed Model

ОТ АВТОРА

Главным научным достижением XX века являются ИТ-технологии Всемирной паутины и системы Skype, проникшие во все сферы жизни людей мирового сообщества, живущих в разных точках планеты. Пользователи Интернета .могут получать различные сервисы, не отходя от компьютеров, начиная от заказа товаров и продуктов и кончая решением простых бизнес-задач банковского и государственного уровня. Такой сервис пользователям предоставляют высокоразвитые компьютерные технологии, включающие в себя новые уникальные методы, средства и инструменты информационной поддержки разных систем Интернета.

Создание ИТ-технологий осуществляется средствами компьютерной науки программной инженерии (Software Engineering – SE), которая в 1968 году в начале компьютеризации применяться на первых электронно-вычис-лительных машинах (ЭВМ) для создания программ. и продолжает применяться для реализации различных глобальных научных и информационных задач исследования природных явлений в атмосфере, космосе и на земле.

В данной книге представлены результаты исследований и разработок автора, связанные с теорией, технологией и CASE-инструментами, обеспечивающими постановку, реализацию и решение сложных научно-технических задач с помощью компьютеров. В ней представлены новые подходы и методы моделирования современных компьютерных технических и программных систем с использованием накопленных и вновь создаваемых программных ресурсов, которые отражают функции отдельных глобальных задач, научных экспериментов в разных областях (физики, математики, биологии, медицины и др.). Главной, стержневой концепцией реализации такого класса задач является тезис конвейерной сборки академика В.М.Глушкова разных видов научно-технических ресурсов в сложные структуры, легко настраиваемые под решение конкретных задач. Первыми такими ресурсами стали математические и физические методы решения народно-хозяйственных задач и задач военно-промышленного комплекса страны. Они создавались в виде стандартных библиотечных программ на ЭВМ и программ общего характера, которые накапливались в неавтоматизированных Фондах алгоритмов и программ, а потом и в электронных библиотеках программ и данных.

Развитие технических ресурсов ЭВМ шло быстрыми темпами по пути их совершенствования и стандартизации микроэлементов (диоды, триоды, транзисторы и др.). Методом конвейерной сборки из них изготавливают большие и малые компьютеры, а также очень малые настольные и карманные компьютеры, мобильные телефоны, компьютеризованные приборы для применения в медицине, космосе, авиации и т.п.

Такого прогресса еще не достигнуто в области изготовления сложных программных и информационных систем. Программная инженерия находиться на стадии формирования готовых многоразовых ресурсов типа модуль, объект, компонент для повторного сборочного их использования в сложных системах.

Наметились и совершенствуются пути формализации процессов и линий реализации отдельных элементарных ресурсов, стандартизованных для сборки, как

в компьютерных технологиях. Здесь только определились языки стандартного описания ресурсов и процессов их сборки в сложные структуры.

Тезис повторного использования возник интуитивно в системах программирования Автокод, Algol-60 и Cobol для УВК "Днепр-2" (1965–1975) при создании общих модулей синтаксического анализа и арифметического блока для этих языков. Затем он стал во главу угла при разработке системы автоматизации производства программ АПРОП (1975–1985) по проекту ГКНТ СССР. Цель АПРОП – обеспечить автоматизированную сборку модулей в разных языках программирования (Algol-60, ПЛ/1, Fortran, Cobol и др.) в широко используемой системе АПРОП в ОС ЕС ЭВМ в СССР. Для модулей был определен *интерфейс*, как посредник для связи и обмена данными между двумя разнородными модулями. Это привело к созданию нового стиля программирования – сборочного, основанного на модулях, интерфейсах и методе сборки их в более сложные программные образования.

Дальнейшее развитие этого стиля программирования осуществлялось в отделе автора "Программная инженерия" (с 1980 г.). Сотрудники отдела, а также аспиранты и студенты Киевского национального университета имени Тараса Шевченко (КНУ, 1965) и филиала МФТИ (с 2000), в которых автор постоянно читал курсы лекций по технологии программирования и SE, стали основным человеческим ресурсом создания сложных систем из более простых.

Работы в области программной инженерии проводись отделом в рамках 14 фундаментальных тем ГКНТ СССР, ГКНТ и НАН Украины (1965–2013). В бывшем СССР была разработана теория и технология сборочного программирования, включающая в себя метод сборки, технологические процессы и линии изготовления сложных систем из модулей. В годы независимой Украины отдел провел разработку формальных основ парадигм программирования (модульной, объектной, компонентной и сервисной) как стилей сборочного программирования с использованием конвейерной сборки и обеспечения качества готовых элементов в этих парадигмах, а также создаваемых из них сложных систем на основе соответствующих CASE-инструментов.

Под руководством автора по данной проблематике разработаны и защищены учениками школы программной инженерии 11 кандидатских и 3 докторские диссертации. Результаты исследований по фундаментальным проектам проблематики технологии программирования и проведенных реализаций прикладных систем и CASE-средств их поддержки представлены в сотнях научных статей, в 10 совместных монографиях и в 3 учебниках.

Главные научно-технические концепции, теории, парадигмы программирования и изготовленные под руководством автора CASE-инструменты в составе Инструментально-технологичного комплекса (ИТК) и экспериментальной фабрики программ Киевского национального университета (КНУ) нашли отражение в данной книге.

Преподавая курсы дисциплин "Технология программирования" и "Программная инженерия" в КНУ с 1965 г. и в МФТИ с 2001 г., автор обучал студентов не только программированию, но и всем аспектам разработки, реализации качественных программ. В результате сформировалось новое видение решения задач SE и ТП, которые нашли отражение в новых формализмах парадигм про-

граммирования отдельных ресурсов. Их использование снижает сложность изготовления программных систем и их качество. С участием студентов удалось разработать стратегию проектирования вариантных систем в семействе систем и продуктов (Product Line). Получено теоретическое и практическое обобщение новых проблем технологии программирования, которые отображены в десятках дипломных, магистерских работ и в первых научных статьях студентов, а также в последней диссертационной работе А.Л.Колесника (2013).

Новые идеи и концепции по программной инженерии докладывались автором и ее учениками на конференциях по технологии программирования (1982–1992), УКРПрог (2007–2012), 8 – 10 международной конференции "The International Conf. ICTERI 2011–2013" (http://senldogo0039.springer-sbm.com/ocs/ home/ и http://ceur-ws.org/Vol-1000/) и Международном научном конгрессе агентства по информатизации "Информационное общество в Украине" (2011–2013) http://www.ict-congress.com.ua/attachments/, а также в научных журналах "Проблемы программирования", "Кибернетика и системный анализ", Весник КНУ и НАНУ и др.

Отдельные аспекты разработанных парадигм, технологий и инструментов представлены в ИТК на сайте http://sestudy.edu-ua.net, на фабрики Киевского национального университета имени Тараса Шевченко http://programsfactory. univ.kiev.ua, а также на российском сайте электронного обучения в МФТИ http://www.intuit.ru.

Автор благодарит всех сотрудников отдела, особенно тех, кто на протяжении 33 лет (1980–2013) работал по проблематике программной инженерии. Это В.М.Грищенко, Г.И.Коваль, Л.П.Бабенко, Е.И.Моренцов, В.М.Зинькович, Л.И.Куцаченко, Е.Л.Карпусь, О.А.Слабоспицкая, П.П.Игнатенко и др.), а также аспиранты и студенты (А.Колесник, А.Островский, И.Радецкий, А.Аронов, А.Дзюбенко, А.Стеняшин, Д.Буряков и др.). Последние принимали непосредственное участие в апробации новых аспектов теории и практики технологии программирования, особенно при изготовлении веб-сайтов.

От имени коллектива отдела "Программной инженерии" аврор благодарит директора Института программных систем НАНУ академика Ф.И.Андона за поддержку фундаментальных проектов по проблематике программной инженерии и CASE-средствам их поддержки.

ПРЕДИСЛОВИЕ

В данной работе изложены оригинальные теоретические и прикладные аспекты технологии программирования, программной инженерии (ПИ), ориентированные на проектирования сложных программных систем (ПС) из готовых программных элементов (ресурсов). Их базисом является идеи сборочного конвейера академика В.М.Глушкова (1975). Она стала на долгие годы основным принципом индустриального производства компьютерных систем, АСУ, АСУТП, программных и информационных систем.

По концепции конвейерной сборки фактически развивались технологии компьютеров, систем и программ, начиная с появления первых ЭВМ и программ для них, ставших программными "заготовками" для последующего использования в сложных структурах АСУ и АСУТП. Первоначально это был *модуль* – самостоятельный программный элемент, задаваемый в любом языке программирования (Algol-60, Fortran, Cobol, PL/1, Modula2 и др.), накапливаемый в библиотеках программ и модулей для повторного использования в других сложных системах. Постановлением Кабинета Министров СССР (1969) было утверждено, что "программы являются продукцией производственно-технического назначения", используемой при индустриальном изготовлении программных продуктов (ПП). Были разработаны методы, средства и CASE-инструменты построения и объединения модулей в более сложные ПП на базе ОС ЕС ЭВМ. Одновременно с этим в стране массово использовалось структурное программирование функций больших систем (более 100 тысяч команд), постепенно приведшее их к кризису сложности, поскольку внесение изменений в системы и адаптации их функций к новым условиям операционных сред является проблематичным.

Технология программирования обсуждалась на всесоюзных конференциях (1965–1992), включая вопросы построения средств автоматизации больших и сложных программ традиционными, структурными методами и разработки технологий сборки готовых модулей и компонентов повторного использования (КПИ), теперь более известных как reuses. Модули стандартизировались и служили строительным материалом для создания больших программ методом сборки, тем самым снижая их сложность и обеспечивая возможность внесения изменений в существующие модули и заменять их новыми.

Существенный вклад в преодоление кризиса сложности программ внес Г.Буч, предложив новый, *объектный* стиль программирования, ставший одним из альтернативных путей создания больших ПП и способствующий снижению их сложности. Широко используемые языки программирования (ЯП) стали пополняться новыми механизмами объектно-ориентированного программирования (такими как наследование, полиморфизм, классы и др.) и системами поддержки ООП (RUP, UML, JAVA и др.). Таким образом, объекты стали элементами сборки наряду с модулями, КПИ, reuses и др.

Отдел автоматизации программирования (с 1967) и программной инженерии (с 1980) все эти годы развивал технологию программирования из готовых эле-

ментов (модулей, объектов, компонентов и сервисов) на теоретическом и прикладном уровнях.

Базисом исследований отдела стал сборочный стиль программирования, в котором роль готовых деталей играли первоначально программные модули, обладающие определенной структурной и функциональной целостностью. Их описание стандартизировалось для обеспечения четко определяемого и контролируемого информационно-логического их взаимодействия для обмена данными между используемыми модулями по схеме сборки.

Сборочное и объектное программирование стало более распространенным и эффективным по сравнению с традиционным структурным программированием. С помощью метода сборки выполнялось комбинирование и конфигурирование готовых модулей, как на сборочном конвейере, для быстрого решения задач определенного класса. Ориентация на класс задач сделало сборочное программирование актуальным, особенно, когда готовыми КПИ, накопленными в информационном сообществе в виде программных модулей и компонентов, смогли пользоваться все работающие в Интернете, решая отдельные задачи из подходящих для этого модулей и сервисов. Схема сборки готовых модулей составляется по концептуальной модели ПС для класса задач и автоматизируется на конвейере, давая разные варианты продуктов для решения отдельных задач из некоторой прикладной проблемы.

Современное сборочное производство ПП из готовых программных ресурсов базируется на элементах, которые создаются в разных парадигмах программирования (модуль, объект, компонент, сервис). Для этих парадигм определены концепции и сформулирован свой формальный стиль программирования соответствующего ресурса, который после сертификации может использоваться в сборочной и другой технологии.

В книге предложены средства моделирования объектов в составе ПС и вариантов ПС в семействах ПС (СПС) с помощью предметно-ориентированных языков DSL (Domain Specific Language) и новые модели (взаимодействия, вариабельности) ПС для обеспечения интероперабельности, изменчивости заново конструируемых отдельных ПС в современных гетерогенных средах. Фактически главное в данной книге – парадигмы программирования и сборки ПС из готовых сертифицированных ресурсов данных парадигм. .Парадигмы представлены новыми формализмами и теорией построения из элементов этих парадигм ПС.

В данной книге представлены CASE-средства реализации доменов: жизненного цикла стандарта ISO/IEC 12207; общих типов данных с помощью фундаментальных типов стандарта ISO/IEC 11404; вычислительной геометрии, оценки качества и др. Технология создания ПС из готовых ресурсов методом сборки КПИ в сложные структуры программ представлена десятью линиями в рамках комплекса ИТК и студенческой фабрики программ КНУ.

Книга представлена следующими разделами.

В разделе 1 "Программная инженерия. Базовые понятия" дано описание становления программной инженерии и технологии программирования компьютерных систем по типу сборочного конвейера В.М.Глушкова. Предложены термины и определения программной инженерии, включая основные факторы –

качество и индустрию изготовления программных продуктов на фабриках программ с помощью новых дисциплины SE, включающих в себя теорию, инженерию, экономику и менеджмент ПП на фабриках программ. Рассмотрены пути становления технологии программирования в СССР, послужившей главной движущейся силой развития технологий компьютерных систем, АСУ, АСУТП, информационных и прикладных систем. Отмечается, что технология компьютерных систем, включая ее элементную базу и сборку из этих элементов новых компьютеров для применения в разных областях (авиации, медицине, космосе и др.) в десятки раз опередила технику изготовления ПП.

В разделе 2. "Парадигмы программирования ПИ" представлена формальная трактовка парадигм программирования – модульного, объектного, компонентного, сервисного и мультипрограммирования. Каждая парадигма содержит в себе теорию моделирования соответствующего программного ресурса и операций их преобразования к промежуточному или выходному коду. Предложена компонентная алгебра (внешняя, внутренняя и эволюционная), с помощью которой ресурсы обрабатываются, изменяются и собираются на формальной основе Описаны модели ресурсов, систем и сред, в которых конфигурируются готовые ресурсы (модули, объекты, компоненты и сервисы) в семейство вариантных программных систем, способных к взаимодействию друг с другом через интерфейс.

В разделе 3. "Технология систем, линий и CASE-средств" дано описание отечественной технологии сборочного программирования сложных систем из готовых программных ресурсов; онтологии стандартного жизненного цикла ПС, включая тестирование КПИ и ПС; задачи управления качеством КПИ и ПС, а также методы достижения надежности и завершенности разработки систем для конкретного применения. Предложены оригинальные CASE-средства в рамках двух веб-сайтов – ИТК и фабрики программ КНУ. В ИТК реализованы 10 технологических линий программирования отдельных элементов, их сборки, проектирования онтологии вычислительной геометрии, ЖЦ стандарта ISO/IEC 12207 и процесса тестирования, преобразования общих типов данных стандарта ISO/IEC 11404 к фундаментальным типам данных. На фабрики программ реализована технология электронного обучения студентов вузов предмету программной инженерии по учебнику, а также языкам программирования C#, Basic и JAVA для написания артефактов и КПИ в стандарте WSDL и системе Grid.

Заключение. Здесь представлены общие аспекты ТП, позволившие конструировать сложные системы на высоком научно-техническом уровне, используя парадигмы программирования и новые CASE-инструменты**. С**формулирована точка зрения автора на постепенный переход от ИТ-технологий к нанотехнологиям в e -sciense (биология, физика, медицина и др.).

Текст книги сопровождается рисунками и таблицами со сквозной нумерацией каждого раздела. Теоремы, аксиомы, определение понятий и формулы последовательно нумеруются по главам в разделах.

Раздел 1
ПРОГРАММНАЯ ИНЖЕНЕРИЯ. БАЗОВЫЕ ПОНЯТИЯ

Глава 1. СТАНОВЛЕНИЕ ПРОГРАММНОЙ ИНЖЕНЕРИИ

Программная инженерия (ПИ) с момента своего появления в 1968 г. заняла центральное место среди компьютерных наук, информатики, информационных систем и технологий. ПИ ориентирована на разработку *программного обеспечения* (ПО) прикладных и информационных систем разного назначения. ПИ – *это система методов, средств и дисциплин планирования, разработки, эксплуатации и сопровождения программного обеспечения (ПО), готового к внедрению*. Основные базовые критерии ПИ – **продуктивность, индустрия и качество**.

Основу ПИ составляет ядро знаний SWEBOK (Software Engineering body of Knowledge, www.swebok.com, 2001 г.), созданное международным комитетом IEEE и ACM, которое включает в себя 10 разделов знаний (area knowledge). Первые пять разделов – это инженерия требований, проектирование, конструирование, тестирование и сопровождение ПО. Следующие пять разделов являются организационными. К ним относится управление проектом, конфигурацией, качеством, методы и средства инженерии (технологии) ПО. Этим разделам соответствуют процессы жизненного цикла (ЖЦ) стандарта ISO/IEC 12207, которые так же ориентированы на реализацию ПО.

Разделы знаний ПИ SWEBOK – 2001 ориентированы на ПО. Они не охватывают целевые объекты ПИ (ПС, СПС, домены, семейства систем, распределенные системы и т. п.), а дают толкование вопросов только теории и практики индустрии программных продуктов (ПП), подходов к определению требований к создаваемому ПО сложных систем и доменов. Кроме того, в SWEBOK-2001 не были включены проблемы защиты данных и обеспечения безопасности работы изготовленных систем. Для объявленной индустрии не было раздела, касающегося экономики затрат на разработку, стоимости ПП и вопросов внедрения ПП в другие организации. Хотя в кругах программистов, вне SWEBOK-2001, в рамках ПИ такие вопросы прорабатывались и получили сое развитие.

Создатели ПИ считали, что главное назначение ПИ – это развитие индустрии ПП высокого качества. Под *индустрией* понимается производство разных видов продуктов массового применения и средств их изготовления. Другими словами, основная задача любой индустрии – массовый выпуск продукции и получение прибыли [1–3]. В. М. Глушков в 1975г. выдвинул идею сборки программных систем из готовых программных элементов, общих и необходимых для многих автоматизируемых систем, подобно сборочному конвейеру в автомобильной промышленности Форда. И, как показала практика, эта идея оказалась наиболее плодотворной. Сборочный принцип производства ПП является наиболее развитый и используемый во многих фирмах производителей ПП.

Сегодня индустрия *программной продукции* мировых фирм-производителей базового программного обеспечения (Microsoft, IBM, CORBA, JAVA, Intel, Apple и др.), а также индийской фирмы по обновлению и доработке наследуемых Legacy систем дает огромные прибыли. Главным и важным вопросам индустрии является обеспечение качества продуктов, в рамках как бывшего СССР для военно-промышленного комплекса, так и НАТО, где возник Swebok. Для развития идей по обеспечению производительности и качества ПП были созданы новые стандарты ГОСТ 9126, международные стандарты ISO/IEC 9000 (1-4) и др., кторые регламентировали процессы достижения и обеспечения качества при создании программ.

В нашей стране успешно развивались технологии компьютеров, систем и программ в рамках создания новых вычислительных, управляющих и инженерных ЭВМ. Эти технологии соответствовали сформировавшейся за рубежом компьютерной науки (Computer Sciences – CS), включающей в себя теорию и эксперименты создания аппаратных и прикладных вычислительных систем. Основные компоненты этой науки: Computer Engineering (компьютерная инженерия или технология), System Engineering (системная технология), Software Engineering (SE – программная инженерия, соответствующая отечественной технологии программирования). Ссылаясь на энциклопедию CS (1992) приведем краткое определение технологических дисциплин CS.

Компьютерная технология – это теории, принципы и методы построения компьютеров (frameworks, суперкомпьютеров и т. п.), а также системного обеспечения ЭВМ (ОС, трансляторы, загрузчики и т. д.). *Системная технология* – это теория, методы и принципы построения автоматизированных информационных систем, систем управления и Computer Systems. *Программная технология* – это система методов, способов и дисциплин планирования, разработки, эксплуатации и сопровождения ПО, обеспечивающих промышленное производство ПП (www.swebok.com).

Со временем начали формироваться информационные системы (ИС) и технологи обработки данных на ЭВМ. Принципам и методам построения ИС В. М. Глушков посвятил свой последний научный труд "Безбумажная информатика" (1982) [3]. В нем дается определение *информационной системы,* как компьютерной системы обработки информации на предприятиях, в органах управления и в производственной сфере. Базис таких систем – документы, не бумажные, а электронные и система документооборота на всех уровнях управления государственных предприятий и органов. *Информационные технологии* – базис компьютерной инфраструктуры современных корпораций, предприятий и государственных органов управления, на которых решаются различные задачи обработки информации локального и глобального характера.

1.1. Определение программной инженерии с 70–90-х годов XX столетия

Вопросами технологии программирования и CS автор занимается системно с 60-х годов прошлого столетия. Было опубликовано много статей и сделаны доклады на Всесоюзных конференциях [4–16] по проблематике, связанной с SE. В

результате сложилось общее понимание научно-технической дисциплины ПИ у многих специалистов институтов бывшего СССР. Автор неоднократно представляла свою точку зрения на курсах лекций студентам КНУ (1980–1991) и лекций в Доме научно-технической информации в Киеве (1990–1992), после этого был опубликован препринт под названием "Проблематика программной инженерии" (1991) [4]. В нем дано определение программной инженерии, основные базовые понятия, процессы, линии и методы разработки ПП. Это изложение отражало согласованную точку зрения многих отечественных и зарубежные специалистов того времени в области ПИ. Такие определения имеют смысл и в настоящее время. Далее приведен текст этой работы с некоторыми корректировками.

Термин "программная инженерия" ("Software Engineering", 1968, www.swebok.com) означает "систематический подход к разработке, эксплуатации, сопровождению и прекращению использования программных средств. С ПИ связаны аналогичные разработки в области технологии программирования, а также работ в SE, которые опубликованы в ряде научных журналов: LEEE Transaction on Software engineering, Software Engineering Journal, Computer Technology, Computer Systems и др.

ПИ называют инженерно-научной дисциплиной, методы, средства и инструменты которой обеспечивают качественный и производительней труд лиц, занимающихся различными видами деятельности по созданию и использованию ПС на инженерной основе.

Эта дисциплина находилась на стадии становления, ее появление знаменовало переход от отдельного "мануфактурного" производства программ к промышленному, при котором для заданного предмета труда (программного объекта) определены: спецификация объекта (эскиз), действия (операции и процедуры), выполняемые исполнителями С целью создания по эскизу опытного образца ПС с заданными функциями и требуемым качеством. На основе опытного образца осуществляется его тиражирование для массового применения.

Термину ПИ в бывшем СССР соответствовал термин "технология программирования", который обозначает "методы, средства и инструменты, обеспечивающие процесс создания ПС". Эти термины настолько близки, что фактически их можно трактовать как дальнейшее развитие ТП в плане обеспечения программистского труда инженерными методами организации (планирование, учёт, контроль) выполнения работ. Такое развитие по существу означало переход от одиночного создания программ отдельными лицами к промышленному их производству со всеми вытекающими проблемами организации труда.

На первом этапе создания ПС главную роль в процессе программирования играли интуиция и знания отдельных программистов. Они же вкладывали свои знания и опыт, творческие способности при создании компьютерных методов и средств для реализации ПС. Тем самым налагались некоторые ограничения на свободу действий рядовых программистов в виде определенных регламентаций и правил. В результате, как и в промышленности, технологические приемы, при которых не только определяется смысл труда, выполняемый при операциях и процессах, но и проводится оценка этого труда по трудоемкости и стоимости, с учетом показателей качества (надёжность, эффективность, понимаемость и др.).

В становлении ПИ сложилось три основных подхода [4]:

– промышленный;
– административный (управленческий);
– инструментальный.

Промышленный подход базируется на технологии производства, состоящей из регламентированной последовательности технологических процессов и операций, призванных гарантировать получение продукции с заданными свойствами. Эти операции обусловлены индивидуальными особенностями исполнителей, которые не учитывались, так как не отражали их творческие способности. Поэтому некоторые специалисты в области ПИ считали, что такой подход не приведет к большому прогрессу.

Однако в сфере ПИ существуют виды работ, относящиеся к этому подходу: правильное копирование программ и документации; настройка и генерация программ; ввод и контроль данных и др. Такие виды работ, как правило, задействованы в производстве. Здесь также выполняются специализированные работы по жестко отработанной, многократно проверенной технологии.

Вместе с тем, творческие способности специалистов могут быть использованы при более трудных процессах создания ПС, сделав труд разработчика программ более инженерным и производительным. При этом процесс разработки может включать в себя операции, требующие элементов творчества от разработчика программ, которые не ограничены определенными рамками процесса создания ПС, а именно: заданной последовательностью работ и результатом, зависящим от заданных в техническом задании требований к готовому программному продукту.

Административный подход ставит во главу угла четкие организационные структуры групп разработчиков программ и обязательное выполнение нормативных документов и стандартов. Данный подход не учитывает основ ПИ, ориентированных на оптимизацию отношений руководителя с подчиненным и производства с потребителями в зависимости от специфики ПС.

Вместо оптимизации отношений предлагаются инструкции, как надо делать, что считать качеством ПО.

Особенно это касается нового стандарта ГОСТ 28915–1989 "Оценка качества программных средств". Он предусматривает огромный объем рутинных работ по экспертированию отдельных свойств и характеристик ПС на этапах жизненного цикла (ЖЦ). Согласно стандарта ГОСТ 9126 –1989 и проекта стандарта СЭВ оценка качества должна проводиться по 250 оценочным элементам с 50 критериев и 6 факторов. Так как экспертная оценка недостаточно точно отражает реальное качество ПО, то требуются более точные расчеты отдельных количественных показателей. Это доказывает, что административными, организационными методами разных служб проводится контроль, экспертизы и проверки элементов процессов жизненного цикла.

Инструментальный подход основывается на автоматизации средств создания ПО в виде CASE-инструментов, а также методов программирования.

Известны попытки свести решение различных проблем повышения производительности труда и качества разработки к созданию или освоению технологических ПС простейшего, частного или общего характера, автоматизирующих все программистские работы с помощью программных инструментов (**Software**

tools). При этом также возникает опасность замены профессионального мастерства готовыми инструментами и средствами, инструменты могут оказаться ненадёжными, трудоёмкими в использовании, с достаточной документацией. Все это зачастую ведет к игнорированию и положительных сторон рассматриваемого подхода, связанных с методологическим характером программирования, включающим в себя:

1) разработку программной документации;
2) комплексное управление качеством ПО;
3) технику модульного и структурного программирования и т. д.

В практике программирования каждый из указанных подходов самостоятельно не используется. Как правило, программист выбирает подход, исходя из особенностей среды, в которой он работает.

Главное заключается в том, какой продукт он производит.

Программный объект является средством ПИ в том случае, если он обладает следующими признаками:

1) многократной применяемостью в некоторых определенных условиях;
2) надёжной работоспособностью;
3) отличительной особенностью, выгодно выделяющей данное средство из всех других при рассмотрении его как объекта применения, внедрения, обмена и т. д.

В связи с изменчивостью условий и среды функционирования объекта на ЭВМ первый признак недостижим без обеспечения соответствующего качества продукта, гарантирующего передаваемость его другим лицам и организациям.

Продуктом деятельности программиста может быть не только программа, но и технология ее изготовления, предоставленная методически или инструментально.

Далее дается определение основных элементов и процессов ПИ.

1.2. Основные понятия программной инженерии

Основные понятия ПИ: программа во всех ее проявлениях (состояниях) на этапах ЖЦ, а также методы, средства и инструменты ее изготовления.

Программа – это объект разработки, который не является осязаемым (нельзя пощупать, взвесить и т. п.) человеком, а доступен пониманию ЭВМ, для которой написан. Готовая программа – это программный продукт, реализующий определённые функции (задачи) ПрО, процесс проектирования и разработка которого осуществляется соответствующими программными методами, средствами и инструментами ТП.

Объектами разработки могут 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 их взаимосвязанного набора.

Для объекта разработки первоначально формируется его модель, которой специфицируются реализуемые функции и элементы данных, устанавливаются их связи и отношения. Иными словами, для объекта определяется его спецификация или прототип.

В техническом задании (ТЗ) к разработке программы предъявляются определенные требования к составу реализуемых функций и к характеристикам – рабо-

тоспособность (свойство надежности), быстродействие, удобство общения (свойство эргономичности), мобильность (свойство переносимости из одной среды в другую) и др.

В зависимости от требований к условиям функционирования объекта (работать с большим быстродействием в условиях реального времени или с большой точностью в условиях космических, военных систем и т. п.) требования к его свойствам отличаются и влияют на организацию процесса разработки.

Любой объект имеет начальное (исходное), промежуточные и конечное состояния. Начальное состояние – это исходная модель объекта (эскиз опытного образца). Промежуточное состояния – это изменённое состояние, отличное от начального и конечного состояний объекта, полученное на определенном этапе жизненного цикла под воздействием соответствующих данному этапу программных методов и средств. Промежуточным состоянием объекта, например, является эскизный, технический и рабочий проекты.

Конечное состояние объекта – это программный продукт, готовый для исполнения требуемых от него функций. Конечным состоянием считается опытный образец, который передаётся в опытную эксплуатацию либо на тиражирование (производство).

В качестве аппарата задания начального состояния объекта используются языки спецификации (SADT, PSA, PSI, и т. д.), а для задания промежуточных состояний применяются ЯП.

Метод разработки (программный метод) – это способ или планомерный подход к достижению той цели, которая ставится перед объектом разработки.

Наиболее распространенные методы проектирования и разработки следующие: нисходящий, восходящий, модульный, метод расширения ядра, метод сборочного программирования. Метод определяет стратегию проектирования и разработки.

При нисходящем (сверху-вниз) проектировании после определения требований к объекту формируется его функциональная и системная архитектуры, при которых 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 определения элементарных базовых понятий ПрО и формирования из них более крупных понятий, приводящих в конечном итоге к определению некоторой функции ПрО. Для сформированной функции подбирается, или вновь разрабатывается программный элемент (модуль, модель программы, макрос, заготовка и т. п.). В этом плане данный метод является методом от готового.

Развитием восходящего метода является метод прототипирования, при котором для объекта вначале создаётся "грубый" его прототип из готовых компонентов, от которых требуется, чтобы они соответствовали функциям объекта без учета эксплуатационных характеристик. Цель прототипирования состоит в том, чтобы отработать "каркас" объекта и его функциональные возможности, а затем улучшать характеристики свойств программных элементов.

Метод расширения ядра характеризуется начальным выделением множества вспомогательных функций. Наиболее эффективным методом выделения являет-

ся метод, использующий анализ данных и определение модулей, обрабатывающих различные информационные структуры. Для этого может применяться метод Джексона, в основе которого лежит принцип соответствия организации программы этапам преобразования обрабатываемых данных.

Несмотря на различие этих методов в стратегиях проектирования, общее, что их объединяет – это модульное или блочное (программное) представление объекта разработки. Суть метода модульного программирования состоит в декомпозиции находкой 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 предоставляет по сравнению с другими 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 в более сложные программные структуры. Расширением данного метода является метод сборочного программирования, обеспечивающий сборку программных компонентов различной степени сложности [5–7].

Таким образом, указанные методы проектирования и программирования взаимосвязаны, они используют друг друга. Так, при применении восходящего метода проектирования для систем обработки данных на ранних этапах ЖЦ программного объекта применяются методы модульного программирования, расширения ядра и др.

Технологический процесс – это взаимосвязанная последовательность операций, выполняемых при разработке объекта. Процесс предназначен для перевода объекта из одного состояния в другой соответствующими программными методами и средствами.

Для реализации набора типовых функций автоматизируемой ПрО, относящейся к СОД, АСНИ, САПР и др., используются типовые технологические процессы (ТП), которые вместе со специализированными образуют линию программ в технологии функционально-ориентированного типа [8–11].

Технологическая линия (ТЛ) задает набор процессов разработки функций объекта, представленных совокупностью автоматизированных операций, которые последовательно и систематически преобразуют состояния объектов, включая заключительное его состояние – готовый программный продукт. Особенность линии – отражение определенных функций ПрО, реализуемых в виде программ с заданными показателями качества.

Для простых программ количество процессов в ТЛ меньше, чем для сложных. Независимо от сложности программного объекта основным условием выполнения процессов является их автоматизация.

Процессы имеют модельное представление, базирующееся на модели жизненного цикла программного объекта разработки.

Динамика изменения объекта может быть представлена в ТЛ последовательностью процессов, переводящих объект из одного состояния в другой путем использования средств автоматизации операций процесса. В этом плане каждая конкретно разработанная ТЛ определяет множество допустимых состояний объектов, а на средства автоматизации возлагается функция перевода и слежения за переходом в недопустимые состояния.

Инструмент – это программное, язычное или методическое средство, применяемое для получения состояния объекта в некотором законченном виде. В зависимости от этапа жизненного цикла и состояния объекта для работы с ним могут быть языки, трансляторы, генераторы и т. п.

Процесс преобразования модели объекта из одного состояния в другое не является полностью автоматизированным, и имеется ряд систем, специально предназначенных для реализации конкретных предметных областей.

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. Основными вопросами управления разработкой программных объектов как инженерии ведения разработки являются [9–10]:

1) организация коллектива разработчиков (состав, структура, квалификация и др.);

2) планирование работ и трудозатрат и обеспечение роста производительности труда;

3) контроль хода разработки и оценка проектных решений в ходе разработки программных продуктов;

4) экономические вопросы (стоимость, ценообразование, стимулирование и др.);

5) управление качеством.

Жизненный цикл – это совокупность последовательных состояний ПС и всех действий, связанных с превращением ПС в продукцию производственно-технического назначения, начиная с анализа потребностей в автоматизации функций ПрО до создания ПС и кончая его моральным износом.

С ЖЦ связаны сроки и период разработки и эксплуатации ПС.

По срокам эксплуатации ПС разделяются на два класса:

1) с краткосрочной эксплуатацией;

2) с долгосрочной эксплуатацией.

К краткосрочным ПС относятся продукты научного творчества студентов, аспирантов, а также стажёров, занимающихся их созданием для проверки научной идеи. ЖЦ таких продуктов включает в себя этапы проектирования и разработки. Как такового этапа эксплуатации нет, он соответствует процессу апробации созданного продукта.

Программным объектам второго класса соответствуют регламентированные процессы проектирования, разработки и эксплуатации. Они создаются в проектных и промышленных организациях, занимающихся реализацией на ЭВМ крупных народнохозяйственных задач. Объекты данного класса изменятся в диапазоне 100–110 команд (операций Ассемблера) со свойством изменяемости и модификация при сопряжении и использовании. Такие объекты тиражируются и вместе с документацией передаются потребителю, отчуждаясь тем самым от разработчика. Срок жизни таких программ 10–20 лет, 70–80% этого срока приходится на эксплуатацию и сопровождение.

Программы с долгосрочным характером эксплуатации требуют применять в процессе разработки индустриальные методы организации и планирования разработки. На первое место выступают задачи экономического характера, способствующие повышению производительности, качества ПС и уменьшению сроков

разработки. Главной задачей стаяло оценивание затрат на разработку ПС и установление конечной стоимости.

ТЛ ориентированы на производства ПП, требующее технико-экономическое обоснование процессов разработки, достижения высокого качества ПС, методов оценивания стоимости ПС и учета человеческого фактора в процессе разработки ПС [11].

Предложенная в [12] Боемом модель стоимости (COCOMO) позволяет провести оценку на этапах ЖЦ с учётом ряда стоимостных атрибутов, квалификации исполнителей, степени использования современных методов программирования.

Основные элементы этой модели:

1) соотношение текущих и будущих расходов на изготовление ПП;
2) эффективность ПС;
3) методы расчета доходов и чистой стоимости;
4) способы оценивания стоимости, основанные на планировании ресурсов, уточнения требований, учета плановых сроков и др.;
5) методы аналитических в экспертных оценок свойств программного продукта, рассматриваемых как факторы (надежность, сложность, быстродействие и др.), влияющие на стоимость;
6) средства создания ПС (стили и методы программирования, инструменты и др.).

Модель жизненного цикла ПС. Процесс разработки ПС определяется моделью ЖЦ, этапы которой будем отождествлять с ТП. Для каждого ТП рассматриваемого ПС фиксируются его начальное (S_0), и промежуточные (S_j) и конечное (S_R) состояния. Переход объекта из состояния S_j в S_{j+1} из $ТП_j$ обеспечивается выполнением ТО, входящих в этот ТП. При этом для каждого типа ПС может быть свой набор ТП и состояний *S,* определяемых на соответствующем множестве исходных данных, характеризующих эти состояния [11]. Фактически формировался процессный подход к разработке ПП.

Приведена схема проектирования, соответствующая методу сверху–вниз, при которой понятия i+1-го уровня ($ТП_{i+1}$) описывается через элементарные понятия этого уровня и представляют собой данные, входящие в класс понятий состояния ПС. При этом для каждого типа ПС не исключаются переходы с одного уровня абстракции на другой (сверху–вниз и снизу–вверх).

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 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 ПС; S_1 – состояние, включающее в себя 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 техниче-

скому проекту и включает в себя 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 в себя 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 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 в процессе сопровождения.

Технологический процесс, входящий в состав $M_{пр}$ – промежуточный (частичный) процесс, осуществляет преобразование S_i-го состояния ПС с помощью набора $ТО_i \subset ТП_i$ данной $M_{пр}$, поддерживаемых TM_i (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 $M_{эпо}$, определенной выше.

Модель общего вида любого частичного $ТП_i$ представлена на рис. 1.1.

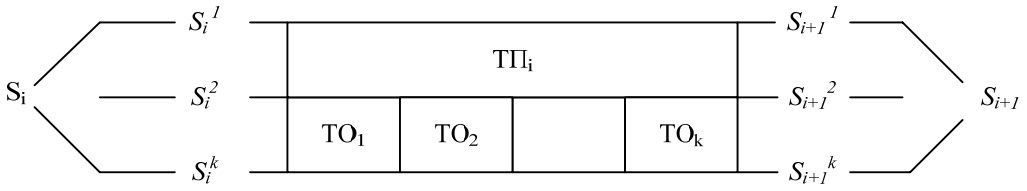


Рис. 1.1. Модель $ТП_i$ процесса

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. 1.2).

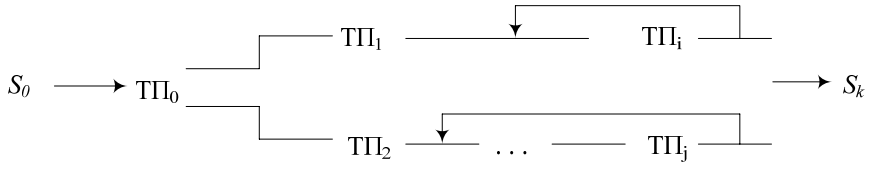


Рис. 1.2. Графовая модель технологического процесса

Каждый частичный процесс ТП, является абстрактным конечным автоматом, граф которого совпадает с технологическим маршрутом процесса, множество состояний S_i которого определено на совокупности операций $\{\text{ТО}_i\} \subset$ 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 для сборки элементов процесса, предопределили специфику языка формального описания ТЛ, а также структуру модели качества, формата проектно-технологических и эксплуатационных документов.

1.3. Принципы программной инженерии

К принципам программной инженерии отнесены:

принцип производственной организации, принцип обеспечения технологичности, принцип планирования трудозатрат [4, 13–15].

Принцип производственной организации. В отличие от творческого характера работ в процессе ТПР при определении ТЛ (анализ ПрО, выбор подходящих средств описания ПрО, корректировка и выбор ограничений на проект и т. д.) инженерные методы, используемые в 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 в себя 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.

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 [12, 14].

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:

1) общий плановый срок разработки $[t_0, T]$, где t_0 и T – начальный и конечный сроки разработки;

2) объем работ $W = \{W_i\}$ с учетом переделок;

3) требуемые ресурсы $R = \{R_l, R_m\}$, где R_l – человедческие; R_m – материальные (технические и программные);

4) нормы потребления человеческих ресурсов по всем ТП, $(i = \overline{1, \mathrm{N}})$, NR_l и др.

Формально план работ записывается в следующем общем виде:

$$Y = Y(t_0, T, W, R_l, R_m, NR_l, f),$$

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 процесса.

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 в себя технологичность ПП и технологичность процесса разработки [10–14].

Технологичность ПП – это соответствие ПП потребительским возможностям 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 подготовку 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 [4] можно заключить, что основное распределение трудозатрат в 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: N – количество процессов на ТЛ; J_i– количество операций на $ТП_i$ $(i = \overline{1, N})$; $\sum_{i=1}^{N} P_i + P$ – 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; 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; T_{ij} – производительность труда (в операторах в день) при выполнении j-й операции в 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 определим следующим образом:

$$F^0 = \max \sum_{i=\overline{1,N}} \sum_{j=1}^{j_i} m_{ij} \Rightarrow \min,$$

$$x_{ij} \le \gamma_i m_{ij} T_{ij}, \gamma_i \ge P_i, \tag{1.1}$$

$$\sum_{i=1}^{N} \gamma_i \le \sum_{i=1}^{N} P_i + P. \tag{1.2}$$

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 (1.1) означает, что x_{ij} – количество операторов в ПП должно быть создано m_{ij} специалистами за γ_i дней, а неравенство (1.2) 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 (1.1). Для этого в F^o потребуем приведения всех сумм $\sum_{j=1}^{j_i} m_{ij}$ к минимально возможной величине. В этом случае

$$F^o = d_1 \sum_{i=1}^{N} \left(\sum_{j=1}^{j_i} m_{ij} - \frac{\sum_{i=1}^{N} \sum_{j=1}^{j_i} m_{ij}}{N} \right) + d_2 \sum_{i=1}^{N} \sum_{j=1}^{j_i} m_{ij} \Rightarrow \min, \qquad (1.3)$$

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 (1.1) получим

$$x_{ij} - \gamma_i^0 m_{ij} T_{ij} - m_{ij}^0 \gamma_i T_{ij} \leq - \gamma_i^0 m_{ij}^0 T_{ij}, \qquad (1.4)$$

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:

$$F^0 = d_1 \sum_{i=1}^{N} \left(\sum_{j=1}^{j_i} m_{ij} - \frac{\sum_{i=1}^{N} \sum_{j=1}^{j_i} m_{ij}}{N} \right) + d_2 \sum_{i=1}^{N} \sum_{j=1}^{j_i} m_{ij} \Rightarrow \min,$$

$$x_{ij} - \gamma_i^0 m_{ij} T_{ij} - m_{ij}^0 \gamma_j T_{ij} \leq - \gamma_i{}^0 \, m_{ij} T_{ij}, \ \gamma_i \geq P_i, \qquad (1.5)$$

$$\sum_{i=1}^{N} \gamma_i \leq \sum_{i=1}^{N} P_i + P,$$

$$m_{ij}^0 = \frac{x_{ij}}{\gamma_i^0 T_{ij}}, \ m_{ij} \geq 1,$$

$$\frac{\sum_{i=1}^{N}\sum_{j=1}^{j_i} m_{ij}}{N} - \sum_{j=1}^{j_k} m_{ij} - y_i + z_i = 0.$$

Решение этой задачи состоит в получении γ_i, m_{ij} (нецелочисленных), с помощью которых определяются целочисленные значения базовой модели, как пробного варианта решения задачи планирования трудозатрат:

Варьируя значениями управляющих параметров d_1, d_2 и γ_i, можно получить окончательное решение задачи планирования, т. е. искомое число специалистов, необходимых для разработки ПС по заданной технологической линии и в срок.

1.4. Управление разработкой и качеством систем

Управление разработкой программ по ТЛ включает в себя решение вопросов планирования хода разработки, учета и контроля [4, 14] с целью создания качественного продукта.

Метод планирования обеспечивает реализацию плана выполнения конкретных работ и учет их выполнения. Для этого вводятся специальные информационные структуры – карты выполнения работ, структура документа, сроки начала и окончания выполнения отдельных операций и квалификация 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 разработки ПП. 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 трудоемкость на одного специалиста вычислялась по формуле

$$A = \left(\frac{1+(I)^{0,8}}{\mathrm{c}\sqrt{QT}} + \frac{D}{d}\right)\frac{R}{B},$$

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 может быть вычислен по формуле

$$\mathrm{Q} = \frac{1}{n}\sum_{m=1}^{n}\frac{1}{m}\sum_{i=1}^{m} q_i(i),$$

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 специалиста к средней производительности для данной категории разработчиков, к которой он относится.

Учет выполнения работ по ТЛ проводится на основании графика в целях получения информации о состоянии работ за определенный промежуток времени и использовании этой информации при контроле и регулировании заданных показателей качества.

В современных условиях, когда требуется высокое качество выпускаемой продукции, учет информации и контроль хода разработки являются важным условием обеспечения промышленного способа разработки.

Контроль хода выполнения работ по созданию ПП проводился 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 разработок.

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 нормоконтролю выпускаемой документации.

Контроль качества проводится контролерами ТП, а на заключительном процессе совместно с главным технологом осуществляется оценка качества ПП на основе методов, которые будут рассматриваться ниже.

Определение затрат на разработку программ по ТЛ. Каждая ТЛ включает процессы предпроектных и проектных исследований, являющиеся 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 базах 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.

Управление качеством ПС – целенаправленная систематическая деятельность по планированию, учету, контролю и регулированию качества ПС в ходе его разработки. Под управлением качества понимается управление требованиями, конфигурацией, планированием проекта для достижения качества и мониторинга (учет и контроль) проекта. В связи с применением *инженерного* подхода к разработке ПП появляется возможность управлять процессом *регулирования* и *измерения* как процессов, так и продуктов ПС. Это управление является залогом

создания конкурентоспособных ПП, качество которых может непрерывно улучшаться. Основополагающие стандарты в области качества ПП – стандарты ISO серии 9000. Такие стандарты рекомендуют каждой разрабатывающей или поставляющей ПП для организации оздания системы качества, которая по определению стандарта ГОСТ 2844 включает в себя совокупность организационной структуры, процедур, процессов и ресурсов, направленных на реализацию управления качеством ПС. Эта система качества должна обеспечивать контроль, инженерию и измерение показателей качества с целью достижения требуемого качества продукта.

1.5. Реинженерия, реверсная инженерия, рефакторинг

Методы систематического изменения отдельных элементов программ (модулей, компонентов, КПИ и др.) и систем программного обеспечения образуют концепцию реинженерии и реверсной инженерии, сформулированных в рамках программной инженерии. Когда сформировалось компонентное программирование, новым явлением стал рефакторинг М. Фаулера, по которому изменение программ и систем определяется как эволюционное развитие систем в процессе их использования [4, 16, 17].

Сущность изменений в программных элементах и системам лежит в плоскости улучшения функциональности и повышения их качества. С помощью изменений в некотором смысле продлевается время жизни действующих, стареющих и наследуемых программ и ресурсов многоразового использования .

Так, для изменения даты 2000 года в действующие в мире компьютерные системы прошлого столетия привлекался широкий круг их разработчиков, а также пользователей в оффшорных зонах (Индия, Россия, Украина, Европа и др.). В системах находилось место расположения этой даты и потом осуществлялась ее замена новой датой. Назначение ракого типа работ состоит в обеспечении жизнеспособности таких систем и адаптации к новым возможностям ОС и платформ современных компьютеров. На решение проблемы изменения даты , как свидетельствую иностранные источники, потрачены большие человческие и финансовые ресурсы. Кроме того, это свидетельствовало о необходимости развития методов реинженерии и реверсной инженерии в проблематике разработки больших программных систем.

С общей точки зрения эволюционное развитие систем обеспечивается внешними и внутренними методами изменения компонентов, интерфейсов и/или систем. К внешним методам относятся: добавление, объединение, удаление отдельных элементов ПС. К внутренним методам – методы реинженерии, реверсной инженерии и рефакторинга. С их помощью проводится целенаправленное изменение программ или систем разного назначения.

Реинженерия (Reengineering) ПС

Реинженерия – это совокупность моделей, методов и процессов изменения структуры, функций элементов ПС для достижения качества функциональности и добавления новых функций на основе требований пользователей.

Метод реинженерии используется при частичной или полной переделке отдельных функций, компонентов, некоторых фрагментов старых программ на

новые при переходе к новым условиям ОС, платформ и требований к качеству продукта или процессов ЖЦ. При этом решается задача обеспечения мобильности ПС, т.е. способности ПС адаптироваться к новой платформе. Такие изменения могут выполняться путем:

1) реорганизации спецификаций и характеристик отдельных ресурсов;

2) модификации или модернизации функций ресурсов и данных;

3) определения способов взаимодействия ресурсов между собой, которые могут располагаться в разных средах и др.

При этом структура ПС может оставаться неизменной, если модификация касается только отдельной функции системы или замены ее другой.

Если в системе изменяется много функций, то процесс изменений является дорогостоящим и связан с большим риском достижения требуемого результата.

С помощью реинженерии совершенствуется системная архитектура, создается новая документация и конфигурация и тем самым обеспечивается продолжение времени жизни наследуемой, изменяемой системы.

Процессы реинженерии применяются и при изменении деловых процессов в целях уменьшения лишних видов деятельности или повышения их эффективности. Зависимость процесса использования наследуемой системы будет минимальной, если заранее будут спланированы возможные изменения в системе.

К основным процессам реинженерии относятся:

1) перевод отдельных компонентов в старом ЯП в новую версию в этом или в другом языке;

2) анализ документов описания структуры и функциональных возможностей системы для изменения проектных решений;

3) декомпозиция системы на более мелкие компоненты для проведения модификации и снижения сложности большой по размеру системы.

4) изменение данных, с которыми работают компоненты системы.

Преобразование исходного кода программ – это наиболее простой способ реинженерии, особенно, если оно может проводиться автоматически или автоматизировано.

Если конструкции ЯП этой системы имеют различия со структурами программ и типами данных нового языка описания ПС, то может потребоваться ручное преобразование элементов ПС. На что затрачиваются значительные технические, человеческие и финансовые ресурсы.

В процессе разработки системы создается комплексная модель, по которой можно выбрать варианты реализации и модификации с учетом требований к защите, безопасности и др.

Таким образом, результатом реинженерия ПС является измененная система с новыми компонентами, усовершенствованными компонентами и соответствующими связями для среды функционирования.

Метод реверсной инженерии (Revers Engineering)

Реверсная инженерия – анализ и рассмотрение структуры системы и ее элементов для построения нового ПП путем обновления и восстановления системы, полученной *прямой инженерией*. Другими словами, изучение элементов структуры системы в целях построения нового варианта системы сводится к восстановлению исходной спецификации, ее структуры и логики компонентов, а также выбору метода перепрограммирования системы.

Сложилось два типа задач реверсной инженерии.

Задачи первого типа включают в себя:

1) анализ системы для проведения изменений в структуре кода;

2) расширение функциональности ПС;

3) замену платформы, ЯП. и т.п.;

4) изменение логической структуры системы и применение шаблонов проектирования для перепрограммирования;

5) изменение моделей и, структур данных.

Задачи второго типа состоят в восстановлении:

1) структуры системы и компонентов, а также в выборе подходящего ЯП;

2) расширения видов интерфейсов и форматов данных для организации вычислений.

Развитию реверсной инженерии послужили переход к ООП и новые способы визуализации, измерения метрик (metric) компонентов ПС и выполнения следующих задач:

1) обеспечение высокого качества системы, переоценка сроков, объемов, сложности и структуры разрабатываемой системы;

2) определение иерархии классов и атрибутов программных объектов в целях наследования их в новой системе;

3) идентификация классов объектов, поиск и выбор паттернов для их идентификации и фиксации их места в структуре системы.

Термин Software Revers Engineering используется сегодня как процесс обратной инженерии над двоичными кодами в направлении рассмотрения структуры электронных книг фирмы ABOVE. Эта инженерия применяется для поиска особенностей в ПС, а также для reassembling программ и обратного проектирования информационных и бизнес систем.

Рефакторинг (Refactoring)

Рефакторорингг – перестройка, улучшение внутренней структуры программного кода с сохранением функциональности.

Метод рефакторинга ориентирован на изменение (замену, замещение, расширение) реализаций компонентов и их интерфейсов с учетом требований и плану конфигурации компонентов.

Каждая операция рефакторинга является базовой, атомарной функцией преобразования, которая сохраняет целостность компонента, т.е. совокупность правил, ограничений и зависимостей между составными элементами компонента, задающих компонент как единую и цельную структуру. Цель рефакторинга – сохранить функции и идентичность компонента. Добавление новых функций, характеристик и свойств проводится с помощью реинженерии.

К операциям рефакторинга отнесены: добавление компонента и измененного интерфейса; замена существующей реализации компонента новой эквивалентной функциональностью и новым интерфейсом; добавление нового интерфейса к старому компоненту и др. Используя базовый набор операций, получаем измененная ПС.

Основным условием рефакторинга является обеспечение целостности компонентов и системы в целом с сохранением свойств и характеристик объектов.

1.6. CASE-средства программной инженерии

CASE – Computer Aided Software Engineering (CASE) – это система автоматизированной разработки программного обеспечения [4, 16 – 18].

CASE-средства – это совокупность методов и способов проектирования и разработки программ с помощью автоматизированных инструментов, которые ориентированы на реализацию отдельных процессов ЖЦ для разных предметных областей. На первых порах их появления сложились специальные CASE-пакеты для военных, коммерческих и др. применений. Среди таких средств были такие:

1) ассемблеры, анализаторы, компиляторы, интерпретаторы;
2) символические отладчики, пакеты программ;
3) анализа требований и систем;
4) редактирование, интеграция текстов и генерация отдельных программ поддержки операций ЖЦ.

Такими средствами осуществлялось проектирование моделей, спецификация элементов систем, формирование словарей данных, логическое проектирование БД и др.

Первые CASE-средства представляли собой автоматизированные системы разработки ПС по схеме: метод–нотация–средство. Метод – это системные процедуры генерации компонентов на основе потоков и структур данных. Нотация – это описание системы, элементов данных и функций с помощью графов, диаграмм, таблиц, блок-схем и др. Средство – инструмент для поддержки метода. К ним относятся инструменты поддержки уровней абстракции и реализации компонентов.

В конце 90-х годов прошлого столетия были разработаны для применения CASE-средства в классе широко используемых в программировании методов структурного анализа (CADT, SSADM, IDEF, OOP, UML и др.), а также системы поддержки разработки систем разного назначения (РТК, АПРОП, ПРОМЕТЕЙ, ППП ДИСПРО, МАЯК и др.).

CASE-системы структурного и объектного проектирования представлены в [Методы]. К ним относятся следующие:

1) метод структурного проектирования SD (Structured Design by Yourdan and Constantine), позволяющий специфицировать требования, данные и программы преобразования входных данных в выходные с помощью карт Джексона;
2) методология объектно-ориентированного анализа и проектирования OOAD (Object-oriented analysis and design by Martin and Odell) для ER-моделирования, определения функций и данных типа "сущность–связь" диаграмм и др.;
3) технология объектного моделирования OMT (Object modeling Technique by Rumbaugh) для проведения анализа, проектирования и реализации моделей ПС (объектной, динамической, функциональной и взаимодействия);
4) метод объектно-ориентированного анализа систем OOAS-SM (Object-Oriented system analysis by Shlaer and Mellor) обеспечивает выделение сущностей, объектов ПрО, их свойств и отношений, а также создание информационной модели, модели состояний и dataflow;

5) метод объектно-ориентированного анализа ООА (Object-Oriented analysis by Coad and Yourdan) обеспечивает моделирование ОМ с помощью "сущность–связь", спецификации потоков данных и процессов в виде диаграмм;

6) метод объектно-ориентированного программирования Буча ориентирован на анализ и выделение объектов ПрО, объединение объектов в классы, суперклассы на основе аппарата наследования и полиморфизма;

7) метод вариантов использования (Use case) применяется для задания сценариев работы системы и набора правил их трансформации и взаимодействия;

8) метод моделирования системы CORBA созданы на основе эталонной объектной модели ПрО, строящейся из предметов реального мира со своими характеристиками, типами и операциями, группируемыми в классы или суперклассы;

9) метод моделирования UML обеспечивает анализ и моделирование объектов ПрО на основе требований с помощью диаграмм и др.

Отдел программной инженерии (с 1980) работал над CASE-средствами: СУБД "Пальма" [19], ПТК для разработки ПС, ППП "АПФОРС" [19, 20, 27].

Глава 2. СТАНОВЛЕНИЕ ОТЕЧЕСТВЕННОЙ ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

Технология программирования (ТП) в СССР сформировалась в рамках промышленного производства ЭВМ многими школами (В. М. Глушков, С. С. Лавров, А. П. Ершов, М. Р. Шура-Бура, Э. Х. Тыугу, Е. Л. Ющенко и др.). Академик В. М. Глушков в 60-х годах прошлого столетия определил ТП, как метод повышения уровня создания ЭВМ, вычислительных систем и способ конвейерной сборки изготовления программных продуктов (ПП) из готовых программных элементов [21]. Технология – это постепенный переход от ремесленного создания программ к промышленному производству компьютеров, систем и программ. Она всегда была двигателем прогрессивного развития любой науки, в том числе и теории создания компьютеров, их программного обеспечения и сложных автоматизированных компьютерных систем – АСУ, АСУТП, САПР и др. Академик В. М. Глушков стал фундатором основных идей технологии разработки отечественных универсальных, управляющих ЭВМ, машин инженерных расчетов (серия "Мир") и со схемной интерпретацией языков высокого уровня (проект "Украина"), а также технологий сложных программных систем. Все это шло по пути развития следующих направлений:

1. Теория построения ЭВМ, вычислительных машин для инженерных расчетов "Мир" и для управления технологическими процессами предприятий (Днепр-1, Днепр-2), машин со схемной интерпретацией языков программирования (ЯП) "Украина", мозгоподобных, макроконвейерных и многопроцессорных машин.

2. Методы автоматизации процессов создания системного программного обеспечения новых ЭВМ (операционные системы, трансляторы, редакторы и подобные им), больших программных систем, пакетов прикладных программ математического, экономического, статистического типа и другие.

3. Теория и практика АС управления для массового создания АСУ на предприятиях бывшего СССР и проекта общегосударственной автоматизированной системы (ОГАС) для управления предприятиями страны, оборудованными новыми системами сбора и обработки данных с автоматизированных рабочих мест (АРМ) предприятий и заводов страны.

Одна из первых работ В.М.Глушкова в области программирования (Об одном методе автоматизации программирования, ж. Программирование.–№1, 1957) посвящена автоматизации программирующих программ (ПП). Алгоритм ПП промоделировал его аспирант А. А. Стогний на примере задачи Коши для систем обыкновенных дифференциальных уравнений первого порядка с помощью библиотек ПП и схемы счета (см. Программирование.–№1, 1957). Первая конконцепция автоматизации ПП с адресного языка и других ЯП для ЭВМ была предложена Е. Л. Ющенко, а подход для автоматизации предприятий и организации АСУ, АСУТП, САПР сформулировал В. М. Глушков [1–3].

Автор данной работы начала развивать идеи ПП с проекта новой УВК "Днепр-2" (1965) и реализации АСУТП. Для УВК были разработаны трансляторы с ЯП (Автокод, Алгамс, Кобол) [22, 23] и ОС управления технологическими процессами промышленных объектов АСУ. УВК "Днепр-2" был внедрен в АСУ металлургического комбината ГДР (Берлин–Лейпциг) по межправительственному соглашению ГДР и СССР (1971–1973). Группа разработчиков (10 специалистов) от Института кибернетики Украины, включая автора, принимала участие во внедрении УВК "Днепр-2" и в разработке АСУТП на площадке ВЦ комбината BMHW ГДР. После завершения работ участники были награждены орденом "Дружбы ГДР". Данная АСУТП работала в ГДР на УВК "Днепр-2" до 1992 г.

2.1. Технологии компьютерных систем и программ

Технология – это совокупность методов, способов, приемов, средств автоматизации и порядка их использования при производстве некоторого продукта.

Технология ЭВМ. При изготовлении первой ЭВМ под руководством академика С. А. Лебедева была сформирована технология проектирования и изготовления *универсальных* ЭВМ. Она совершенствовалась в плане унификации элементной базы и методов сборки в отдельные структурные компоненты ЭВМ. Известны работы В.М.Глушкова, В.А.Мясникова и др., в которых предлагались новые виды рекурсивных, микро- и макроконвейерных ЭВМ с новой элементной базой для организации высокоэффективных вычислений сложных математических и народнохозяйственных задач, а так же для управления оборудованием с приоритетным прерыванием объектов в системах АСУ и АСУ ТП [2, 24–25]. В. М. Глушков считал, что структурные элементы ЭВМ и систем будут постоянно модернизироваться и стандартизироваться. Процесс изготовления компьютерных систем приближается к автоматизированной сборке, как на конвейере Форда [26, 27]. Сборочный конвейер оборудован технологическими линиями сборки отдельных частей компьютерных систем и систем в целом. Это предвидение сбылось. И сегодня мы видим, что компьютеры разных вариантов и типов массово собираются по принципу сборочного конвейера, как на конвейере Форда.

Технология АС, АСУ и АСУТП. Теория построения АСУ была изложена Глушковым в работах [1–3], которыми пользуются многие специалисты и сегодня. По его методологии создавались АСУ и ИС (например, на Львовском телевизионном заводе, металлургическом комбинате ГДР и др.). В.М.Глушков предложил систему ОГАС для объединения разных АС и АСУ в единую систему, доступную всем организациям и предприятиям страны. Этот проект не был поддержан ЦК КПСС, так как требовал огромных финансовые ресурсов. Теория АСУ Глушкова постоянно развивается учеными и практиками. В качестве примера можно привести работу аспирантки ИК Украины Н. Т.Задорожной (2004), которая не только использовала идеи и принципы Глушкова, но и практически реализовала ИС документооборота в Академии педагогических наук для сферы образования Украины. Был создан первый учебник по менеджменту документооборота ИС (http://lib.iitta.gov.ua/view/creators), который пользуется спросом для управления научными проектами в образовательной сфере Украине.

Технология программирования. Методы автоматизации процессов создания системного программного обеспечения новых ЭВМ позволили разработать системы программирования с ЯП (Algol, PL/1, Cobol, Fortran, Modula-2 и др.). Их разработкой, включая адресный язык и ЯП для отечественных ЭВМ, руководила Е. Л. Ющенко. Был создан СМ-метод анализа ЯП [23] в рамках реализации ЯП АЛГОЛ и Кобол для комплекса УВК "Днепр- 2".

В. М. Глушков 5 марта 1975г. на научном семинаре специалистов Института кибернетики выдвинул идею сборки разнородных модулей и программ средствами фабрик, работающих по принципу сборочного конвейера в автомобильной промышленности. Эта идея многие годы плодотворно развивалась в Институте кибернетики по разным направлениям автоматизации программных систем и пакетов прикладных программ (ППП).

Развитие направлений ТП сформулированы В. М. Глушковым в 1980 [28]:

1) модульная система автоматизации производства программ (АПРОП) из стандартизированных программных заготовок в сложные системы [26, 27];

2) метод формализованных технических заданий для проектирования сложных программных комплексов с использованием нескольких алгоритмических языков путем последовательной детализации компьютерного проекта [29];

3) Р-технология программирования систем средствами графического Р-языка, близкого структурному проектированию программ и данных в АСУ ВПК [30].

В результате выполнения этих направлений были разработаны: система на основе формализованных технических заданий (Ю. В. Капитонова, А. А. Летичевский); система автоматизации программ – АПРОП (Е. М. Лаврищева); комплексирование пакетов прикладных программ (ППП) для методов численного анализа (И. Н. Молчанов); блочно-сборочное создание ППП статистики и оптимизации (И. Н.Парасюк, И. В.Сергиенко); генерация систем обработки данных из макросов Макробол (Бабенко Л. П.), расширяемая система общих компонентов трансляторов (Н. М. Мищенко), система композиции функций ДЕФИПС (В. Н. Редько); технологический комплекс РТК (И. В. Вельбицкий), система Терем (Н. М. Мищенко) и др. Данные системы относятся к классу CASE-систем. Они внесли существенный вклад в развитие идеи сборки сложных программ, ППП из готовых модулей и послужили полигоном формирования сборочного программирования программных продуктов на ЕС ЭВМ.

2.2. Формирование сборочной технологии программирования в бывшем СССР

Актуальной задачей развития ТП В. М. Глушков считал "технологию комплексного проектирования вычислительных систем, когда проектирование технических средств системы объединено в единый процесс проектирования базисного математического обеспечения". Эта идея реализована в системе ПРОЕКТ с использованием формализованных технических зданий и средств языка Аналитик "Мир-2". Перспективной тенденцией развития таких систем Глушков считал "переход от однопроцессорных фоннеймановских машин до мозгоподобных машин.

В этот период был взят курс на развитие индустрии программ. Были созданы фонды алгоритмов и программ во всех республиках СССР. С участием В. М.Глушкова сформировалась концепция сборки программ (система АПРОП), в том числе из библиотек и фондов, как путь к индустрии программ из уже разработанных элементов или готовых "деталей" [31].

После изучения стандартов сборки в автомобильной промышленности впервые был определен *интерфейс*, как механизм связывания разнородных объектов и обмена данных между ними. В результате сформировалось сборочное программирование, объектами которого были разнородные модули, интерфейсы и функции преобразования нерелевантных типов данных, передаваемых между связываемыми модулями в ЯП на платформе ЕС ЭВМ. Операциями в этом программировании стали процессы сборочного конвейера

Интерфейс апробирован на разных типах программных модулей в ЯП (Algol, Fortran, PL/1, Cobol и др.). Вопросам интерфейса была посвящена международная конференция "Интерфейс СЭВ" (1987) [32]. Авторская концепция интерфейса, включая межъязычный, межмодульный и технологический, была принята и ученые Г. И.Коваль, Т. М.Коротун, Е. М.Лаврищева были награждены Почетной грамотой. Аналогичная концепция интерфейса, как связывающего звена разноязычных модулей, отобразилась в языке MIL (Module Interface Language, 1984) и в современных языках API (Application programs Interface), IDL (Interface Definition Language), SIDL (Scientifical IDL) и др.

Интерфейс стал главным элементом в процессе создания новых ПС из готовых модулей и затем компонентов, КПИ в современных глобальных средах. Он стал базисом сборочного конвейера Глушкова и сборочного программирования. Данное программирование развивали академик А. П. Ершова, профессор В. В. Липаев, академик Э. Х. Тыугу и многие другие. Система АПРОП [24], как CASE-инструмент автоматизации процесса сборки, финансировалась министерством радиопромышленности более 10 лет. Данная система стала составной частью в проекте "ПРОТВА – технология создания бортовых систем" и коллектив разработчиков этого проекта был награжден премией Кабинета Министров СССР (1987). Системой сборочного программирования АПРОП официально использовали 52 организации СССР.

Академик А. П. Ершов считал "сборочное программирование эффективным, поскольку готовые запрограммированные модули позволяют быстро решить

любые задачи из определенной проблемной области для ЕС ЭВМ и мини-, микро- и макро ЭВМ" [33].

Сборка стала важным технологическим решением в индустрии программных продуктов в бывшем СССР. Автором по этой тематике была защищена докторская диссертация "Методы, средства и инструменты сборочного программирования" (1988), которую оппонировали специалисты – Э. Х. Тыугу, Э. З. Любимский и И. В. Вельбицкий. Результаты исследований в области сборочного программирования были опубликованы в монографиях "Сборочное программирование" (Е. М. Лаврищева, В. Н. Грищенко, 1991), "Технология сборочного программирования" (В. В. Липаев, Б. А. Позин, А. А. Штрик, 1992) и "Прикладные программные системы" (В. Н. Редько, И. В. Сергиенко, В. С. Стукало, 1992).

Как результат определился главный элемент сборочного конвейера Глушкова, а именно технологические линии (ТЛ), которые апробированы в проекте Института кибернетики АИС "Юпитер-470" для военно-морского флота СССР (1983–1991). В нем было создано шесть ТЛ и сгенерировано около 500 программ обработки данных для разных объектов этой АИС.

2.3. Развитие индустриальных технологий в программной инженерии

Курс на индустрию ПП поддерживается известными фабриками программ – AppFabric (VS.Net, VSphere IBM, CORBA, Intel, Grid и др.). Разработаны научные проекты фабрик программ: мультитехнология К. Чернецки с лейтмотивом "от ручного труда к конвейерной сборке"; технология взаимодействия разноязычных программ И. Бея; потоковая сборка – use case UML Дж. Гринфильда и Г. Ленца; сборочный конвейер ЕПАМ (БГУ), Авдошина и многие другие. О таких фабриках мечтал академик В. М.Глушков и для нас эта идея стала стратегической. В ИПС НАНУ был разработан комплекс ИТК ИПС (http://sestudy.edu-ua.net) с десятью ТЛ для электронного изготовления ПП из готовых компонентов повторного использования (КПИ).

Концепция сборочного конвейера Глушкова реализована студентами 4 курса факультета кибернетики КНУ имени Тараса Шевченко в виде фабрики программ (веб-сайт http://programsfactory.univ.kiev.ua) под руководством автора. Фактически сделан подарок к 90-летию В. М.Глушкова. Основу фабрики составляют артефакты студентов и программные компоненты.

К ней обратилось более 15 000 пользователей (Goole статистика) из разных стран. ИТК пополняется новыми перспективными линиями: DSL, GDT, Services и др. Веб-сайты ориентированы на обучение студентов современным фундаментальным основам ТП, SE и получения знаний в области индустрии ПП. Эти сайты используют ряд вузов Украины, стран СНГ, а также других зарубежных стран в целях обучения ТП, SE, Computer Sciences.

В связи с развитием новых стилей программирования класс элементов сборки расширился новыми стандартизированными КПИ (объектами, компонентами, сервисами и др.), как главные ресурсы конвейерной сборки на современных фабриках программ.

Глава 3. КОМПЬЮТЕРНЫЕ ТЕХНОЛОГИИ ФАБРИК ПРОГРАММ

Одновременно с развитием ТП в бывшем СССР, за рубежом сформировалась Computer Science (CS).

Она была ориентирована на инженерное конструирование компьютеров (Computer Engineering), систем (Systems Engineering), программного обеспечения (Software Engineering).

Впервые термин Software Engineering (SE) прозвучал на научно-технической конференции НАТО в 1968г. Официальное определение SE было сформировано международным комитетом специалистов ACM и IEEE в SWEBOK (Software Engineering Body Knowledge) в 2001г. (www.swebok.com). Ядро знаний SWEBOK содержит описание десяти разделов знаний (knowledge area), соответствующих сформировавшимся процессам ЖЦ, методам и средствам изготовления качественных ПП, их эксплуатации и сопровождения.

Технология программирования в бывшем СССР по смыслу имела более широкое содержание (теория, методы и средства), чем это было представлено разделами знаний SWEBOK. SE дало регламентированное создание ПС, но в нем еще отсутствовала теория и технология качественного изготовления ПП.

3.1. Зарубежные компьютерные технологии

В рамках новой дисциплины Computer Science за рубежом были определены следующие технологии: компьютерная инженерия (Computer Engineering); системная инженерия (System Engineering) и программная инженерия (Software Engineering), информационные систем и информационные технологий (рис. 1.3).

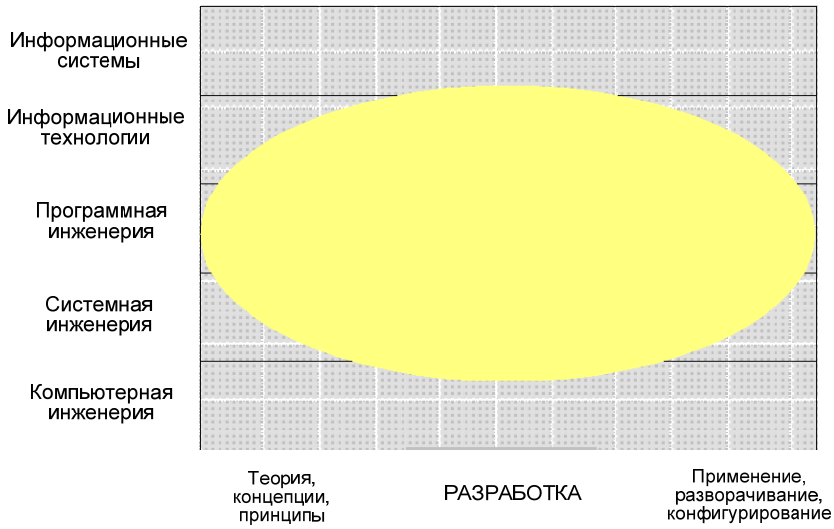


Рис. 1.3. Место программной инженерии в пространстве информатики

Среди этих технологий программная инженерия занимает центральное место. Она задает парадигмы, теории и концепции для каждой технологии Computer Science, определяет процессы разработки каждой из них, а также применение, конфигурирование и развертывание созданного ПО. Суть этих инженерных технологий приведена исходя из изучения материалов американской энциклопедии CS (1992).

Компьютерная инженерия – это теория и принципы построения компьютеров, Фреймворков, суперкомпьютеров, вычислительных кластеров и их системного программного обеспечения. Основами дисциплины являются теории Тьюринга, фон Неймана, автоматов, алгоритмов и положения кибернетики, разработаны В. М. Глушковым [2–6]. Она использует математику, логику, теории анализа и систем. С помощью этих средств строятся компьютеры, Фреймворки, многопроцессорные, макроконвейерные машины, устройства, микросхем и т. п.

Системная инженерия – это теория, методы и принципы построения информационных, компьютерных, АС и систем управления ими. Она представляет собой междисциплинарный подход, объединяющий теоретические положения, методы и средства, направленные на создание и объединение системных решений для поддержки сложно организованных объектов. Данный подход базируется на технологиях компьютерных систем для разных областей применения и новых средствах управления информационными системами (ОС, БД, СУБД и другие). Системная технология включает в себя теорию АСУ и информатики Глушкова [2, 3], а также математику, компьютерные наук и методы, которые применяют в экономике, финансовой деятельности и других. Системна инженерия или технология получили развитие при создании АС различного назначения разными международными организациями.

Программная инженерия – это система методов, способов и дисциплин по планированию, разработке, эксплуатации и сопровождения ПО, предназначенного для его промышленного производства (www.swebok.com). Она охватывает все аспекты создания ПО от начала формулирования требований и до разработки, сопровождения и окончательного списания его. Основы программной инженерии – теории алгоритмов, программирования, методы вычислений и распределенных коммуникаций. Массовое сопровождение готовых ПП основывается на теориях менеджмента, планирования процессов и необходимых для этого ресурсов, верификации и тестирования, оценивания рисков и качества [33]. Технология создания программных продуктов развивается в направлении создания программных и информационных технологий.

Информационные технологии с 1990-х годов получили широкое развитие в современных корпорациях, предприятиях и государственных органах управления. Они обеспечивают решение задачи, связанных с обработкой информации, в том числе, размещенной в глобальных сетях. Для разработки таких технологий подготавливают высококвалифицированных ИТ-специалистов в университетах. Цели и задачи информационных технологий были сформулированы В. М. Глушковым в работе [3], но они и по сей день остаются актуальными и имеют большое значение. Сегодня ИТ-технологии обслуживающего типа широко представлены в Интернете для массового применения.

Информационные системы – это компьютерные системы обработки разнообразной информации, относящейся к различным сферам деятельности человека (в частности, обмен информацией, документооборот на всех уровнях управления [34, 35] и др.). Информационные системы и технологии стали основным инструментом жизнедеятельности современного общества.

Значительный вклад в развитие информатики внесли советские ученые. В сборнике «Кибернетика. Становление информатики» (М.: Наука, 1986) представлены основополагающие идеи в области информатики ученых страны: А.А Александрова, О..М.Белоцерковского, В.М.Гушкова, А.АДородницина, А.П.Ершова, Д.А.По-спелова и др. В их статьях определены цели и задачи информатики, а также актуальные направления научно-технического ее развития, включая элементную базу ЭВМ и суперЭВМ, компьютерные и информационные технологии, интеллектуальное и индустриальное аспекты кибернетики и социально-эконо-мические вопросы управления мировым сообществом в будущем. Отдельные вопросы этих направлений развивались в рамках программной инженерии в технологии программирования.

3.2. Индустриальные основы программной инженерии

Термин индустрия отражает производство разных видов продуктов массового применения и средств их изготовления промышленными предприятиями, фирмами и корпорациями, в том числе программно-технологического типа. Главным вопросом любой индустрии является не только выпуск соответствующей продукции, но и получение прибыли от этого. Сейчас большие прибыли от выпуска ПП получают такие мировые фирмы, как Microsoft, IBM, CORBA, Intel и др., а также индийские фирмы по адаптации устаревших (legacy) систем программ. У нас за последние десятилетия развитие индустрии ПП фактически отсутствовало, не было государственных и научных программ ее поддержки. Однако наряду с крупными корпорациями действуют предприятия коммерческого типа, которые разрабатывают ПП по заказ у разных организаций [36 – 38].

Под ***производством*** **ПП** понимаются процессы ТЛ, посредством которых исполнители используют соответствующую теорию и инструментальные средства для выработки ПП массового применения. Эти линии из процессов, ориентированы на разработку ПП. Примером являются развитые технологии – Engineering application, family, domain, а также сформированные средства обновления устаревших систем в оффшорных организациях (например, в Индии). В настоящее время для поддержки процессов изготовления ПП используются системные сервисные средства (ОС, общесистемные сервисы горизонтального и вертикального типа, новые языки, трансляторы, редакторы, компоузеры и .т. п.), готовые программные ресурсы, КПИ и методики проведения робот, к которым относятся предложенные нами *дисциплин ПИ,* которые определяют разные аспекты деятельности по производству ПП, а именно научные, инженерные, управленческие, экономические, производственные. Они нужны при производстве ПП и участии профессиональных специалистов для изготовления ПП и разработки новых средств производства для фабрики программ [7–9].

Сущность производства продукции. Производство продукции – это процесс создания материальных благ, необходимых для удовлетворения индивидуальных и социальных потребностей общества. Продукция производится из готовых элементов, ресурсов.

От объема используемых ресурсов зависит уровень производства некоторой продукции, которая вычисляется с помощью функцией вида:

$$v = F(z, u),$$

где $v = (v_i)$ – вектор выпуска продукции, $z = (z_i)$ – вектор расходов ресурсов, $u = (u_i)$ – матрица параметров зависимости функции расходов $z = F(w_j, u)$, $j = 1, 2 .., n$. Показатели функции w_j отвечают объему производства, структуре производственных фондов и уровню специализации фабрики, их значения для уровня специализации фабрики формируются, как правило, статистически.

Общая производственная функция Кобба–Дугласа выпуска продукции имеет вид

$$v = ne^{rt} L^{\alpha} K^{\beta},$$

где v – обобщенный выпуск, n – нормативный множитель, e – основа натурального алгоритма, t – показатель уровня научно-технического прогресса, L – расходы человеческого труда K – величина капитала, α, β – коэффициенты эластичности [7].

Расчеты этих функций позволяют определить соответствующие оценки доходной части, в частности на фабрике программ. Экономичность, системность и пропорциональность выпуска ПП зависят от принципов организации и управления ресурсами, номенклатуры продукции, применения компьютеров и служб фабрики (контроля, тестирования, оценивания ПП и др.).

Первой индустриальной пробой стала *студенческая фабрика программ и артефактов* (http://programsfactory.univ.kiev.ua) на факультете кибернетики в КНУ имени Тараса Шевченко из готовых артефактов соответствующих кафедр, их апробация на технологических или продуктовых линиях (Product Lines), и распространение их не только на отечественном, но и на зарубежном рынке.

Фабрика базируется на линиях, оборудованных набором средств, комплектующих "деталей" – готовых ресурсов, инструментов и сервисов для автоматизированного выполнения процессов на этих линиях в конкретной операционной среде. Другие информационные технологии предоставляют набор инструментов для перехода от разработки отдельных продуктов, к индустрии сложных программных и компьютерных систем с повышением производительности создания отдельных элементов продукта на процессах жизненного цикла. Линии разработки для простых продуктов реализованы в среде MS.Net с использованием, каркасов, языка DSL (Domain Specific Language), WorkFlow и др. Элементы фабрики накапливаются в разных библиотеках и репозиториях и используются при сборке из них сложных ПС.

Методика производства ПП. Ядро SWEBOK и соответствующая программа обучения Curricula-2004 не содержит дисциплин, связанных с производством программной продукции на фабриках программ, принципами комплектации фабрики техническими и человеческими ресурсами, методами создания необходимых *продуктовых линий,* служб поддержки средств и управления соот-

ветствующими специалистами. Иными словами в индустрии ПП не решены проблемы оценки сложности объектов и процессов изготовления ПП на *линиях* работ, в которых предлагаются пути преодоления сложности, особенно при сборке больших систем из разных готовых КПИ.

Потому требуется методическое обеспечение для разработки и выполнения процесса производства ПП, а именно, описание набора новых дисциплин SE, которые могут выступать в качестве нормативного и регламентированного выполнения разных видов работ по разработке, сборке, управлению экспертизами, измерениями и оценками артефактов. Таким образом, . методиками являются научно-технические дисциплины (*Di*) программной инженерии (ПИ), предложенные в [39-40]:

$$ПИ = \{DiSc, DiEn, DiEc, DsMa, DiDe\},$$

где *DiSi* – научная дисциплина, *DiEn* – инженерная, *DiEc* – экономическая, *DiMa* – управленческая (менеджерская) и *DiDe* – производственная.

3.3. Дисциплины программной инженерии

Проведена систематизация и классификация программной инженерии.

Автором предложены новые дисциплины (ДПИ), с помощью которых разные специалисты, участвующие в производстве ПП, выполняют виды работы, соответствующие новым дисциплинам. Сформулированы их задачи и цели (рис. 1.4), а именно, методы и принципы ведения разработки ПП, учитывая научные и инженерные вопросы проектирования и разработки, а также организационные – сроки, стоимость, производительность и способы достижения качества продукта.

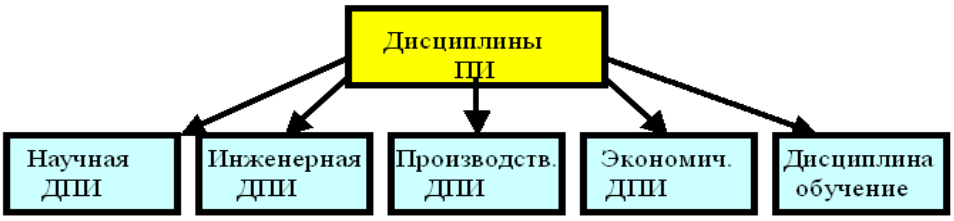


Рис. 1.4. Набор дисциплин ДПИ программной инженерии

Научная дисциплина

Основу этой дисциплины ПИ составляют классические науки (теория алгоритмов, теория множеств, теория доказательств, математическая логика и др..), теория программирования, теория абстрактных моделей и архитектур программных объектов, теория управления и др. Эта дисциплина включает в себя основные формализованные базовые понятия и объекты, формальные подходы, методы программирования, CASE-системы, теорию данных, методы управления изготовлением ПП [17–18] и др.

К *основным* понятиям дисциплины относятся типы и структуры данных, функции и композиции, простые и сложные целевые объекты. Разработка простых объектов выполняется посредством их формального описания, спецификации, а сложных целевых – посредством применения инженерных методов проектирования и организации управления процессов их изготовления.

Теория программирования – основа дисциплины. Она включает в себя методы (рис. 1.5):

1) языки, средства спецификации и проектирования целевых объектов, методы доказательства их правильности (верификация, тестирование);

2) формальные методы управления (персоналом, материальными и финансовыми ресурсами) проектом и отдельными его характеристиками;

3) методы оценивания промежуточных и конечных результатов проектирования для достижения заданных показателей качества ПП (надежность, корректность и т. п.).

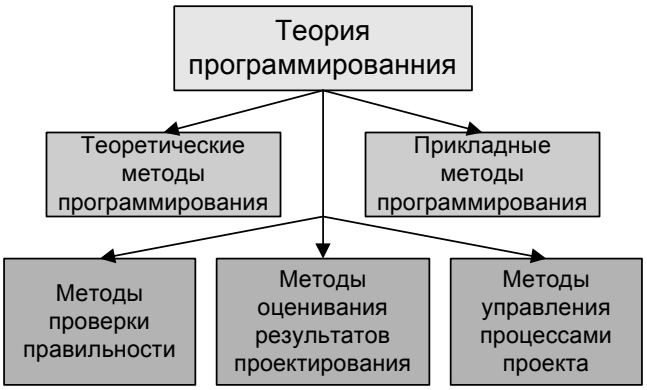


Рис. 1.5. Структура теории программирования

Таким образом, научная дисциплина – это теоретический фундамент ПИ и его необходимо изучать будущим компьютерщикам не только для повышения уровня квалификации, но и для поддержки и развития теорий, возможностей и средств программирования, которые усовершенствуют соответствующие направления индустрии ПИ. Одна из важных научных проблем индустриального производства ПП – это интеграция (композиция, сборка) составных элементов будущего продукта, основанная на совместимости их интерфейсов при сборке этих элементов в сложные программные структуры. Интеграция решается, опираясь на фундаментальную теорию синтеза, как одну из ветвей научной дисциплины ПИ.

Главным курсом обучения в вузах должна стать научная дисциплина, которую необходимо, на наш взгляд, представить общими дисциплинами теоретического курса, курсами систематического программирования (объектного, компонентного, агентного и т. п.), отдельными классическими курсами программы Curricula-2004 в области программной инженерии [39], а также CASE-средствами и инструментами поддержки парадигм программирования [18] .

Инженерная дисциплина

Инженерия ПИ – это способы применения теории программирования, технологических правил и процедур, процессов ЖЦ, методик измерения и оценки выполнения деятельности по изготовлению программного продукта (рис. 1.6).

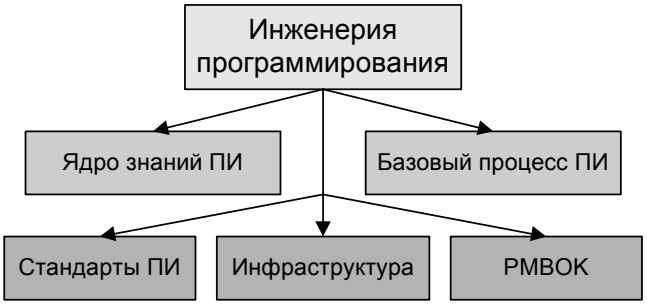


Рис. 1.6. Структура инженерии ДПИ

Ядро – набор областей знаний; БПО – схема деятельности; стандарты – набор правил и положений; модель ресурсов проекта; PMBOK – управление проектом.

Данная дисциплина определяет совокупность инженерных приемов, средств и стандартов, ориентированных на изготовление целевых объектов ПП с применением научной дисциплины ПИ [9–13]. Базовыми компонентами этой дисциплины являются:

1) ядро знаний SWEBOK, как набор методов и средств разработки ПП и управления проектами;

2) базовый процесс ПИ, как стержень процессной деятельности разработчиков ПП;

3) стандарты, как набор регламентированных правил конструирования промежуточных артефактов на процессах ЖЦ;

4) инфраструктура – условия среды, методическое и организационное обеспечение базового процесса ПИ и поддержки деятельности исполнителей ПП;

5) общие системные средства и инструментальные среды поддержки процессов изготовления ПП.

Технология инженерии ПП базируется на КПИ, готовых сервисах, ресурсах и инструментах их построения. К таким технологиям относятся: инженерия КПИ (Reuse Engineering), инженерия приложений (Application Engineering), доменов (Domain Engineering) и семейство систем (Family of systems Engineering- Product line SEI www.sei.com) [16, 17, 43].

Инженерия КПИ сформировалась как систематическая и целенаправленная деятельность по поиску и подбору готовых КПИ, которые размещаются в современных хранилищах (репозиториях или библиотеках). Базисом изготовления из них ПП является каркас, набор вновь реализованных компонентов и готовых, функциональных КПИ. В новых ПП могут использоваться готовые прикладные системы (например, в бизнесе, коммерции, экономике и т. п.), а также системы общего назначения (трансляторы, редакторы, ОС, СУБД, системы интеграции, генерации и т. п.).

Инженерия приложений и *инженерия доменов* также основываются на многократном использовании разных КПИ и других программных элементов. Основная задача этих видов инженерной деятельности – это построение прикладных систем или семейств систем, которые реализуют задачи приложения или

домена с учетом общих и изменяемых характеристик составляющих их элементов (членов семейства). В инженерии доменов сформировалась технология, основанная на принципах конвейерного производства продуктов из готовых "деталей" типа КПИ по модели домена в DSL (Domain Specific Language) и спецификациях каждого члена семейства [43]. Основная суть этой технологии – управление изготовлением ПП, основанное на планах-графиках работ, контроле результатов работ и оценивании степени применимости готовых ресурсов в процессе реализации специфических задач домена. Компоненты данной инженерной дисциплины совершенствуются и адаптируются к условиям производственной среды (что близко моделям CMM, SPICE, Trillium и др.).

На основе исследований научных и инженерных аспектов компьютерных наук создан фундамент инженерной дисциплины, который включает в себя описание стандартных принципов инженерии, современных языков спецификации доменов, членов семейств и процессов производства ПП средствами и инструментами инженерных технологий.

Дисциплина управления в ПИ

Базис этой дисциплины – классическая теория менеджмента производства проектов и стандарт IEEE Std.1490 PMBOK (Project Management Body of Knowledge).

Теория управления, а также теория организационного управления разработана академиком В. М. Глушковым (1970г.). Эта теория проверена на практике при построении технологических процессов в металлургической, судостроительной и химической промышленностях, а также при внедрении для целей массового производства (например, АСУ "Львов"). После смерти В.М.Глушкова (1982г.) теория управления и практика не получили должного развития [2, 3].

Однако за границей теория управления сложными системами успешно развивалась, особенно в части теории планирования производством. Так, на фирме "Dupon" с целью планирования и создания планов-графиков больших комплексов работ для 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 с числом операций более чем 60 тыс. Применение этого метода оказалось настолько успешным, что проект был завершен на два года раньше запланированного срока. Каждый из этих методов возник в недрах промышленного производства. Они адаптированы к среде программирования и стали базовыми в индустрии программных продуктов.

Элементы теории управления и планирования нашли отражение в стандарте PMBOK. В нем определены процессы жизненного цикла проекта и главные области знаний, сгруппированные по таким задачам, как инициация, планирование, мониторинг, управление и завершение. Основная область знаний этого ядра – интеграция включает в себя концепцию управления организационной деятельностью коллектива исполнителей проекта. Она базируется на методах принятия решений, касающихся ресурсов, общих задач проектирования, служб контроля правильно-

сти выполнения проекта и проверки выполнения финансовой стоимости, заданной заказчиком.

Эти базовые теории управления и планирования, стандартные положения PMBOK, серии стандартов ISO 9001, регламентирующих управление качеством, и соответствующее методическое обеспечение проекта должны стать базисом дисциплины управления в ПИ. Сформированный курс обучения этой дисциплины будет обеспечивать подготовку в вузах будущих высококвалифицированных менеджеров проектов и других специалистов в области организационного управления выпуском программной продукции.

Экономическая дисциплина в ПИ

Экономика ПИ является самостоятельной дисциплиной предмета ПИ и связана с экономическими аспектами индустрии ПП. В ее основу положено проведения экономических расчетов разных сторон деятельности на проекте с учетом знаний специалистов всех факторов и 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 финансовые расчеты с каждым исполнителем на всех этапах выполняемых договоров.

Эта дисциплина наиболее развита с точки зрения методов экономических расчетов в ПИ, а именно, наличие методологий прогнозирования размера ПП (FPA – Function Points Analyses, Feature Points, Mark-II Function Points, 3D Function Points и др.); оценки трудозатрат на разработку ПП с помощью семейства моделей COCOMO, ряда других математических моделей оценки трудозатрат на разработку ПП (Angel, Slim, Seer-SEM и др.), а также моделей, связывающих экономические показатели ПП с характеристиками качества [12, 33]. В рамках дисциплины выполняются следующие расчеты:

1) измерение показателей продукта и производительности;
2) анализ риска создания проектов и процесса разработки;
3) сбор данных для оценка стоимости ПП;
4) определение размера ПП и др.

На основе этих показателей проводится оценки трудозатрат на разработку проекта и оценка качества продукта для проведения его сертификации.

При формировании задач дисциплины используются фундаментальные экономические методы, связанные с принципами распределения и экспертизы работ в сложных системах, а также современные методы расчета стоимости отдельных частей систем с помощью данных, полученных при определении размера КПИ и системы в целом. В этом плане привлекаются стандарты (ISO/IEC 9000 (1–4), ISO/IEC 9126 и др.), обеспечивающие оценку качества и сертификацию готового продукта. Систематизированный и научно обоснованный курс экономической дисциплины ПИ закроет тот пробел, который имеется при выпуске ПП в индустриальном цикле его производства.

Производственная дисциплина ПИ

Главный вопрос индустрии – выпуск ПП 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 таких 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.

Первыми попытками индустриального производства являются технологическая подготовка разработки ПП (ТПР) [8, 9], линия продукта (Product line) института SEI США, обеспечивающая удовлетворение рыночных потребностей пользователей некоторыми видами программной продукции. Более продвинутые инженерные технологии производства ПП – это инженерия приложений, доменов, семейств систем, а также средства поддержки их производства (ОС, общесистемные средства, новые языки, интегральные среды и т. п.).

ТПР было применено при разработке АІС "Юпитер" для производства программ обработки данных на нескольких объектах этой системы. ТПР не развивалась последние годы из-за отсутствия такого рода систем. Производство ПП на упомянутой *линии продуктов* осуществляется из готовых программ, информационных ресурсов, КПИ, средств и инструментов по технологической линии, в которую включаются необходимые методы разработки, тестирования и оценивания конечного результата. Технология конструирования на такой линии выполняется с помощью каркаса ПП и применения подобранных КПИ. Инструментальная среда их разработки содержит необходимые средства и инструменты производства, а также механизмы отслеживания хода построения продукта в соответствии с установленным заказчиком планом.

В последние годы в Украине в основном не разрабатываются такие линии и инструментальные среды. Появился новый стиль работ – внедрение, аутсортинг готовых зарубежных систем и инструментов, который составляет больше 35 % общего объема работ по программированию. При сопровождении таких систем возникают трудности, поскольку они не всегда получены по лицензии и, как правило, не имеют соответствующей документации по принципам построения и использования готового продукта.

На сегодня в индустрии ПП не решены проблемы, касающиеся сложности объектов и процессов изготовления ПП. Мало работ, в которых предлагаются пути преодоления сложности, особенно при интеграции больших проектов из разных простых и готовых ресурсов и продуктов, а также при их эволюции. Среди основных научных направлений по вопросу сложности ПП отмечается теория вариабельности К.Чернецки и У.Азенакер. В рамках проекта разработана новая концепция вариабельности в виде модели вариантности для создания ПС в семействе систем из готовых программных ресурсов.

Данная дисциплина включает в себя методы и технологии производства разных видов продуктов, методы анализа сложности структуры ПС, средства описания специфических особенностей целевых объектов, оценки готовых ресурсов и осо-

бенностей инструментальных сред, а также языков спецификации этих объектов, стандартных положений по производству и документированию готового продукта.

Каждая из приведенных дисциплин – это теоретические основы, понятия, объекты и метод их выполнения при изготовлении ПП в фабричных условиях. Особенность идеи производства ПП – применение готовых ПП (reuses, services, assets и др.), используемых как готовые "детали" на складе выпуска продукции в автомобильной промышленности и т. п.

Связующее звено сборки ПП из готовых продуктов (или КПИ) – это интерфейс. Он определяет типы и виды данных, передаваемых между объектами КПИ сборки в ПС или семейства ПС. Для их автоматизации стали разрабатываться ТЛ, включающие в себя последовательность действий, ТО и форм документов, ориентированных на регламентированное выполнения определенной деятельности при изготовлении ПП и методов оценки разных сторон ПП.

Таким образом, предложенные дисциплины ДПИ детализируются, в будущем они станут предметом подготовки студентов для их участия в индустриальном производстве качественных ПП с помощью таких специалистов: аналитики – DiSi Sciences, инженеры – *DiEn* Engineers, экономисты – *DiEc* Economics, руководители – *DiMa* Managers, разработчики – *DiDe* Developer и т. д.

3.4. Современные фабрики программ. Типы, ресурсы, платформы

В работе [44] приведен анализ фабрик программ и дана их характеристика:

1) система АПРОП *(ИК)* [24];
2) система Sun Microsystems *(IBM)* [45]*;*
3) *OMA-архитектура* или система CORBA *(OMG)* [46]*;*
4) фабрика "ручной" сборки разноязычных программ Инга Бейя [47];
5) фабрики программ по методу UML Дж. Гринфильда [48];
6) среда для групповой разработки ПП – MS.VSTS [49];
7) инфраструктура вычисления Grid [50];
8) фабрика Г. Ленца на основе use case [51];
9) каркас фабрики программ Авдошина и др. [52].

Анализ сред систем – АПРОП, IBM Sun Microsystems, CORBA. Это основные системы на начальном этапе конвейерного производства ПП, проложившие путь к созданию современных фабрик программ.

АПРОП – это фабрика, которая работала в среде ОС ЕС и обеспечивала сборку разноязычных модулей в монолитную структуру на ЕС ЭВМ через интерфейсных посредников, сгенерированных с помощью специальной библиотеки функций преобразования нерелевантных FTD типов данных (например, символьного к целому и т. п.), которые передаются операторами CALL в ЯП 4GPL модулях и реализуют методы численного анализа и статистики, которые расположены в специальном банке модулей.

IBM – среда со сборкой разноязычных программ в 4GPL (1980-е годы) с помощью внешних интерфейсных данных, которые трансформируются функциями XDR-библиотеки, Sun Workshop, Toolbox и т. п. к соответствующей платформе Дальнейшим развитием новых направлений производства ПП является модель

архитектуры SOA, веb-сервисы, языки C, C++, JAVA, RUBY, SCRIPT, которые обеспечивают связь программ в ЯП и передачу данных через порт системы AIX. Интеграция разнородных программ выполняется на общей платформе IBM в средах – ONC (Sun Microsystems), MVS, VM, OS/2, AIX, Open source, а также на сервере (WebSphere Application Server Compunity Edition).

CORBA или OMA-архитектура (Apple, IBM, Win-NT, x-Open, Dec), ее можно считать фабрикой программ с обеспечением связи разноязычных объектов в ЯП (C, C++, Smalltalk, JAVA, Cobol, ADA-96 и др.) через интерфейсные посредники (stub, skeleton, dill), которые описываются в языке IDL для клиент-серверной архитектуры (Client-interface, Server-Interface) с использованием сред (COSS, DCE/RPC, PCTE, ToolTalk, JAVA2SDK, NetPilot CCS и т. п.). Данные между клиентом и сервером передаются протоколами TCP/IP, IIOP через брокер ORB, который обеспечивает их разным сервисом, в том числе по преобразованию несовместимых типов данных разноязычных объектов, устранения неоднородности платформенных данных взаимодействующих объектов клиента и сервера. Эта среда поддерживает связи с другими средами CORBA, OLE/DCOM, SOM/DSOM, IBM OS и т. п.

Фабрика сборки И. Бея. Базисом этой фабрики производства разноязычных программ есть интерфейсный посредник, командные строки, конфигурационные файлы, проверенные в операционных средах (VC++, VBasic, Matlab, JAVA, Visual Works Smalltalk и т. п.) для разных платформ (Microsoft.Net, HP, Apple, IBM и т. п.). Автор разработал разные варианты связей пар разноязычных программ в названных ЯП. Они передавали данные между собой для некоторых из них проводилось преобразование данных, типы которых нерелевантные или и их форматы зависят от особенностей платформы, механизмов передачи данных (через протоколы, вызовы, интерфейсные карты MIO-16E-2 и др.) и средств RMI, Java Native Interface и его реализации в виде exe-файла. Он апробировал Domain and Application Model, Model Interconnection, Microsoft Foundation, а также современные визуальные средства (панели, сценарии, иконки и т. п.) для внесения изменений типов данных вызывающей и вызываемой программ.

Фабрика программ Дж. Гринфильда. Для будущей фабрики сформулированы методологические и технологические аспекты производства ПП по методу UML с использованием моделей архитектур систем и компьютеров, механизмов интеграции разнородных компонентов с типами данных FDT и описанием интерфейса в языках (IDL, XML, RDF и т. п.). Главные концептуальные объекты производства: *reuse*, которые накоплены в общепринятых хранилищах (библиотеках, репозиториях и т. п.) Интернет и имеют сертификаты качества; *активы* (assets), программы, приложения и системы; *модели,* шаблоны и инструменты UML при реализации ПП на линии производства; *веб-сервисы,* процессы линий; методики измерения и контроля качества ПП и т. п. Анализ моделирования UML на данной фабрике показал следующее: язык UML является языком эскиза ПП, который не допускает использования компонентов reuse и интерфейса на разных ЯП; проблема модификации ПП не решена в визуальном языке UML и др. Фактически автор разработал меморандум современной фабрики ПП, которая базируется на reuses, применяемых на производственных линиях, моделях систем, каркасах процессов и продуктов. Конкретной фабрики построить не удалось.

Фабрики программ фирмы Microsoft. Основа данной фабрики – среда MS.NET. Она содержит большое количество средств и инструментов, а именно: готовые ресурсы (компоненты, сервисы) Интернета, языки – JAVA, C++, Basic, JAVA, Pascal, C#, библиотеки CLR и FCL, сборка кодов (exe, dill) в готовый ПП, веб-обслуживание разного назначения, управление PM-2007 группами разработчиков ПП в виртуальной среде VSTS, MSF, которые могут работать по проекту, находясь в разных уголков мира. В состав средств автоматизации входят: пакет инструментов VSTS-2005 (Visual Studio Teams Systems); MSF (Microsoft Solution Arhitecture) для построения производственной архитектуры предприятия, стандарт PMBOK для управления группами разработчиков; модели процессов и систем; Professional Studio и Foundation Server для поддержки процессов проектирования, кодирования тестирования, формирования версий ПП (SDLC, IDE, MS Office, MS SQL server, MS Visual Studio 2005); модель усовершенствования процессов (CMMI Process Impovement), в частности процесса сборки, который использует CLR-библиотеки (Common Language Routine), классы, FCL-типы, трансляторы, General code (exe), Portable Executable code и т. п. В среду включены средства определения сроков работ, трудозатрат, оценки показателей качества изготовлении разных видов ПП и др.

Фабрики для вычислений в Grid. Европейский проект Grid ориентирован на организацию распределенных вычислений задач в разных научных направлениях (физика, математика, медицина, биология и др.) [14, 15]. Он включает ряд подпроектов: Gcube-операционная среда, ETICS – как сборочная среда и т. п. Она предназначена для производства распределенных систем разного назначения методом интеграции (сборки) существующих готовых КПИ и ПП с применением веб-порталов и многоплатформенных ресурсов.

Технология создания пакетов из исходных или комбинаций перекомпилированных программ в двоичном представлении обеспечивается процессом доступа к репозиторию для выбора компонентов системы в альтернативной сетевой среде Grid. Задача представления типов данных объектов среды: проект, подсистема и компонент реализуется с использованием типового формата CIM как механизма связи между разными компонентами с помощью системы MySQL на основе аннотации ряда свойств компонентов (имя, лицензия, URL в репозиторий и т. д.), глобального уникального идентификатора – ID (GUID) и скомпилированного компонента с отображением его в конфигурационном файле ПП.

Сервисы – главные ресурсы инфраструктуры вычислений. Они обеспечивают процесс вычислений научных задач глобального масштаба. Ресурсы задаются в протоколе для передачи данных по распределенной сети. Сервисы предоставляются стандартным протоколом, интерфейсом API и инструментом SDK (Software Development Kits). Сборка программ, компонентов, подсистем и систем научного назначения реализуется протоколом (Global Protocol). Подсистема ETICS содержит средства разработки, тестирования пакетов и услуг, а также сборку и конфигурирование программных элементов на основе механизмов плагинов [46, 50] и открытых интерфейсов для обслуживания потребителей или поставщиков, что и на фабрике программ.

Главные особенности рассмотренных фабрик программ – это методы производства, ТЛ и инструментальная поддержка операционной среды для автоматизации процессов ЖЦ или ТЛ.

Фабрика ПО концепции Ленца включает в себя 4 базовых блока (рис. 1.7).

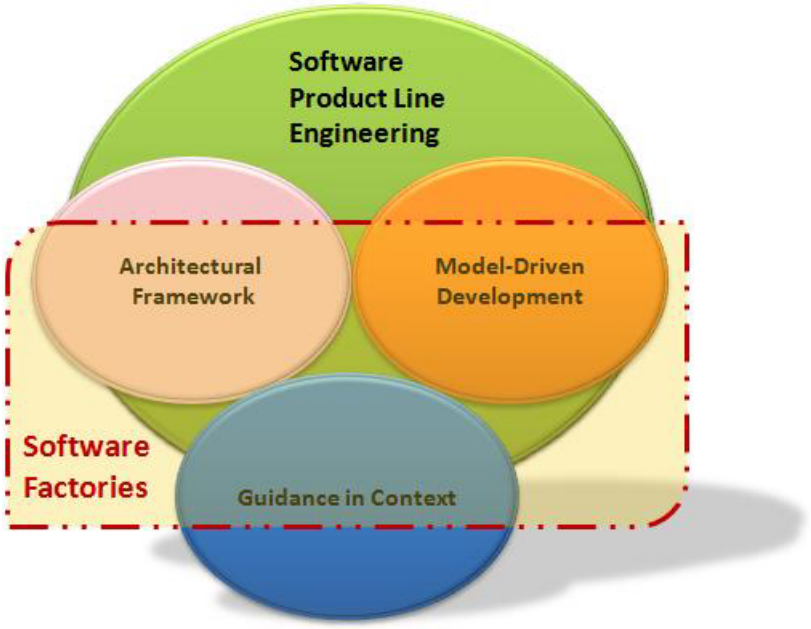


Рис. 1.7. Блоки фабрики Ленца

Ключевые моменты Software Factory – схемы и шаблон Software Factory. Схема Software Factory – это описание того, как реализовать продукты, которые могут быть произведены на этой фабрики. Типичный пример шаблона – завод ПО, а именно, инструмент Visual Studio, обеспечивающий разработку проектов из многократно используемых компонентов при реализации проектных решений и требований.

Эти требования и решения описываются на языке DSL в виде задания архитектуры, блоков и документов, которыми заказчики приложения будут пользоваться при реализации соответствующей линии продукта или ее экземпляра.

Архитектурный каркас фабрики Авдошина. Основным ресурсом и активом данной фабрики является архитектурный каркас, как отправная точка, в разработке любого продукта на линии. Каркас покупается или разрабатывается организацией-разработчиком ПП в среде фабрики. В нем аккумулируются ресурсы: классы, компоненты, конфигурации, образцы и т. п. Активы выбираются на стадии разработки линии фабрики [53]. База знаний фабрики, включает в себя разные пособия, справочники, статьи, примеры программ и программы-образцы. Модельно-ориентированная разработка по абстрактной модели оперирует набором понятий ПрО, заданных в языке DSL.

Моделирование понятий и основных концепций основывается на генерации кода и конфигурации структуры продукта. Для реализации фабрики задается схема фабрики, которая охватывает все активы, точки зрения и связи между ни-

ми. Из схемы берется шаблон фабрики и уточняется набор необходимых ресурсов для автоматизации работы среды разработки.

Разработчик на фабрике задает схему продукта или модель описания рабочих продуктов в заданной ПрО, а также процессы и активы в виде коллекции избранных и тщательно подобранных активов и рекомендаций для реализации применения на линии. Пользователь фабрики работает с шаблоном фабрики, которая специфицирует процесс разработки. Фабрика рассматривается с позиций разработчика и пользователя фабрики.

Любая схема фабрики состоит из набора взаимозависимых точек зрения, каждая из которых разрешает посмотреть на систему с разных сторон. Точки зрения – это стандартные блоки в схеме фабрики, которые определяют действия по созданию и изменению рабочих продуктов. Они отражают логическую и физическую стороны системы, набор используемых компонентов и документирование архитектуры семейства ПП. Каждая точка зрения имеет имя и описание, т. е. она указывает разработчику, что строить, как строить и из чего (моделей, средств, шаблонов и т. п.).

Рабочие продукты, которые вырабатываются или потребляются исходя из данной точки зрения, составляют вид – экземпляр точки зрения, который описывает разрабатываемый продукт с какой-нибудь стороны. Одна точка зрения может иметь несколько видов.

Виды могут использовать разные механизмы (языки, модели), которые разрешают документировать разные аспекты производства ПП, и ими могут быть элементы:

1) идентификатор вида и его описание;
2) представление системы соответственно заданной точке зрения;
3) конфигурация продукта.

При определении отношений между классом, объектом задаются вид и точки зрения, которая соответствует механизмам фабрики. Схема фабрики – это набор взаимозависимых точек зрения, необходимых для построения продуктов. Она подобна схеме базы данных, которая помогает определять и вести данные. Схемой задается организация фабрики из точек зрения, связанных с конкретными ресурсами. Например, точки зрения фабрики на создание программ управления предприятием средствами сервис-ориентированной архитектуры.

Шаблон фабрики – это пакет с ресурсами в составе фабрики с экземпляром схемы фабрики, которые включает в себя совокупность всех активов для точек зрения схемы фабрики следующих категорий:

1) библиотеки и каркасы с многократно используемыми программными компонентами, разработанными при проектирования линии (.NET Enterprise Library);

2) рекомендации, технологии, распоряжения и руководства по автоматизации рутинного процесса построения ПП;

3) языки предметной области и дизайнеры, которые задают необходимый уровень абстракции при разработке приложения и генерации кода по модели.

Построенный шаблон фабрики может быть загружен в интегрированную среду разработки для автоматизации ПП.

Сконфигурируем типовые средства создания ПС на фабриках программ.

Типовые фабрики программ на современных платформах

В каждой операционной распределенной системе массового применения созданы фабрики программ, выполняющие изготовление сложных программных продуктов, работающие практически на всех современных компьютерных и кластерных платформах:

1) AppFab в системе коллективной разработки VS.Net;
2) AppFab в системе Grid рамочного Европейского проекта;
3) в системе IBM для создания доменов бизнес-систем;
4) AppFab в системе CORBA для сборки разнородных программных ресурсов;
5) AppFab в системе Intel;
6) Product Line SEI USA;
7) фабрики потоковой сборки программ Дж. Гринфильда, Г. Ленца,
8) фабрика continious integration М. Фаулера ;
8) коммерческие фабрики программ типа ЕПАМ;
9) другие системные фабрики.

К *системным фабрикам* интеграции (сборки) разнородных компонентов в сложные ПС, семейства систем, распределенные приложения (РПС), работающих с данными из разных хранилищ данных, получивших название "Big Data", а также библиотек глобальных данных (хранилища данных, big data), которые необходимы при организации Cloud Computing вычислений важных научно-технических задач.

К ним относятся следующие: IBM WebSphere, Microsoft Biz Talk 2004, BEA WebLogic Oracle 10g, SAP NetWeaver, ИВК "Юпитер (см. табл. 1.1). В этой таблице отражены стедства и инструменты передовых фирм разработчиков CASE-систем интеграции, их платформы и содержание серверов приложений, каждый из которых включает в себя специфические компоненты поддержки процессов интеграции программных ресурсов в этих системах. Эти программные средства предлагаются для изготовления новых программных и компьютерных систем. Они располагаются на специальных серверах рассматриваемых CASE-систем.

Таблица 1.1. **CASE-системы интеграции программных ресурсов**

Платформа	Разработчик	Содержание платформы
IBM WebSphere	Корпорация IBM	Сервер приложений J2EE, брокеры обмена данными, ГОР, портал, workflow/BPM, средства EII
Microsoft Biz Talk 2004 и компоненты .Net	Корпорация "Майкрософт"	Сервер приложений COM, брокеры обмена данными, ГОР доставки, портал, workflow/BPM
BEA WebLogic	Корпорация "BEA Systems" (в 2008 г. присоединенная к корпорации "Oracle")	Сервер приложений J2EE, брокеры обмена данными, ГОР, сервер прикладных приложений, портал, workflow/BPM
Oracle 10g	Корпорация "Oracle" http://www.oracle.com	Сервер приложений J2EE, брокеры обмена данными, ГОР, портал, workflow/BPM, средства EII

Окончание табл. 1.1

Платформа	Разработчик	Содержание платформы
SAP NetWeaver	Корпорация SAP http://www1.sap.com/ www.sap.ru	Сервер приложений J2EE/ABAP, брокеры обмена данными, портал, инструменты BPM
ИВК "Юпитер	Компания ИВК (Россия) http://www.ivk.ru/	Брокеры обмена данными, ГОР, среда выполнения, сертификация, защита данных

CASE корпорации IBM WebSphere. Эта платформа позволяет работать на основе веб-технологий. Ядро WebSphere включает в себя WebSphere Application Server (WAS) и использует открытые стандарты J2EE, XML и веб-сервисы.

Платформа WebSphere – это многофункциональный набор инструментов интеграции приложений в рамках предприятия (EAI) на уровнях: данных, обмена сообщениями, сквозных бизнес-процессов, B2B business to business; выполнения бизнес-логики программ на JAVA. В этот набор входят следующие:

1) система обработки очередей сообщений (Message Oriented Middleware, MOM), Business Integration Interchange Server (ICS) и MQ Business Integration Message Broker (WSMB);
2) сервер приложений WAS;
3) Portal Server средства, функционирующего на WAS;
4) система проектирования Workflow, совместимая с WSMB.

В состав WebSphere входит Business Integration Workbench для проектирования бизнес-процессов и управления ими. Продуктами платформы являются образцы в своих классах, брокер сообщений WSMB, WAS и встроенные возможности для задания бизнес-правил и сценариев workflow, а также связывания компонентов Enterprise JAVA beans (EJB).

Сервер WAS позволяет использовать ресурсы вуб-сервисов с компоновкой у единый процесс. IBM WebSphere объединяет следующие средства:

WebSphere Business Integration *for Automotive* для поддержки автоматизированного создания сервисно-ориентированных приложений РПС деловых процессов. WebSphere Business Integration *for Banking* предоставляет для банков средства обслуживания мелких и корпоративных клиентов, а также финансовые услуги. WebSphere Business Integration *for Financial Networks* позволяет руководить обработкой платежей путем консолидации разнородных сетей обмена сообщениями. WebSphere Business Integration *for Electronics* ориентирован на компании, которые производят электронику. Поддерживает интеграцию и оптимизацию операций проектирования и производства, обеспечивая ускорение выпуска на рынок новых продуктов, сокращения расходов на сохранение запасных элементов, улучшения выполнения заказов и обслуживания заказчиков. Кроме того, этот продукт позволяет соединять между собой "унаследованные" системы и разнородные приложения. WebSphere Business Integration *for Energy* and Utilities обеспечивает оптимальную интеграцию процессов эксплуатации (в частности, бесперебойного электроснабжения), управления активами и их обслуживания. WebSphere Business Integration Express *for Item Synchronization* помогает компаниям среднего размера связать информацию из их цепочек услуг.

Таким образом, IBM WebSphere предоставляет набор средств по созданию приложений разного типа методом их интеграции.

CASE Microsoft .NET Framework. Эта платформа предоставляет разработчику ту же функциональность, что и J2EE, но в среде ОС Windows. Инструменты, необходимые для реализации разных интеграционных подходов, сгруппированы в ней в виде нескольких продуктов, а отдельные функции возложены непосредственно на ОС (например, компонент управления транзакциями MTS, веб-сервер Internet Information Server, библиотеки и среда выполнения "управляемого кода" .Net).

Основная функциональность BizTalk Server 2004 – сервер интеграции на базе XML, как брокер сообщений, осуществляет преобразование данных и коммутацию сообщений. Сервер приложений является средством выполнения бизнес-логики – низкого (компоненты EJB) высокого (через механизмы workflow) уровней, BizTalk Server поддерживает только высокоуровневую бизнес-логику и интеграцию систем, а выполнение логики низкого уровня реализуется моделью COM+ или .Net.

Модель Microsoft позволяет размежевать работу программиста и аналитика бизнес-процессов. Бизнес-аналитик может графически "рисовать" бизнес-процесс, задавая схемы обмена документами и передачи управления через workflow. Для интеграции определяются точки вызова внешней функциональности через Сом-объекты, веб-сервисы и т. п**.**

CASE WebLogic Integration. Данная платформа – инструмент интеграции, сконструированной по принципу "все включено" и позволяет осуществлять интеграцию приложений и информационное взаимодействие с бизнес-партнерами (B2B), а также создавать бизнес-логику программ на языке JAVA. Платформа включает в себя пять основных компонентов: виртуальная машина JAVA, сервер приложений, средство построения порталов, пакет инструментов интеграции, среда разработки.

Ключевым преимуществом платформы считается возможность снижения требований к группе разработки за счет использования трехуровневого подхода к созданию приложений, подобно подходу корпорации Microsoft. Программы создаются на языках JAVA, Visual Basic или COBOL. Платформа BEA имеет поддержку новейших стандартов XML (XSLT, XQuery и т. п.), веб-сервисов и средств гарантированной доставки, совместимых со стандартом JMS, и брокер сообщений. WebLogic предоставляет один из самых полных наборов интерфейсов для интеграции приложений, файлов и баз данных разной природы у приложения. Для платформы создано много готовых конвекторов, а также системы документооборота, синтаксических анализаторов форматов файлов, средств для обращения ко всем выполняемым модулям программ Windows и JAVA для взаимодействия с интеграционными платформами других разработчиков.

CASE Oracle Integration. Данная платформа предоставляет полный набор средств корпорации "Oracle" для интеграции приложений из разнородных приложений. Платформа соединяет технологии нескольких классов со стилями интеграции: данных (технология Transparent Gateways и конвекторы базы данных), интерфейса пользователя, сервера и системы MOM.

В Oracle Application реализованы новейшие возможности СОА и веб-сервисов управления их координацией и композицией приложений для любых сред разработки приложений. В ней выполняется конструктивное развитие двух современных парадигм конструирования приложений с корпоративными вычислениями и использованием сервисов (Service-Oriented Computing) и сетевых вычислений (Grid Computing). В ней предлагается аспектная инфраструктура реализации SOA и объединения (federate) приложения с необходимой функциональностью обеспечения доступа к ним как к сервисам. В свою очередь, корпоративные вычисления на базе сервисов обеспечивают гибкую инфраструктуру корпоративных приложений.

Сетевые вычисления предусматривают координированное использование значительного количества собираемых из модулей серверов и блоков памяти, которые имеют низкую стоимость при эксплуатации стратегически важных приложений поддержки деловых процессов. Предложены три базовых технологических решения:

1) Business Intelligence – средства построения РПС анализа бизнес-информации предприятия, которые позволяют собирать, анализировать и распределять разноуровневую информацию согласно с компетенциями и полномочиями ее адресатов;

2) Business Integration – средства интеграции разнородных приложений, которые делают возможным объединение отдельных подсистем и автоматизацию деловых процессов;

3) Identity Management (управление идентификационными параметрами личности), что позволяет консолидировать средства администрирования защиты для снижения полной стоимости владения ими и сокращения количества точек уязвимости.

Для поддержки конструирования приложений предлагается основной пакет – Oracle Integration Interconnect.

Этот пакет обеспечивает многоаспектные функциональные возможности композиции и координации сервисов (служб) предприятия через соответствующую шину ESB для быстрого развертывания интеграционных приложений на предприятии. Платформа SAP предлагает два репозитория метаданных для интеграционных связей: для разработки (Repository SAP), и для развертывания (Directory). Это позволяет максимально приблизить условия разработки и тестирования создаваемой ПС к условиям ее эксплуатации.

Особенность платформы SAP – поддержка интеграции не только на уровне конектора к шине обмена данными, но и на высших уровнях, совместимых с системой управления контентом, и порталом среды разработки. SAP позволяет использовать среду разработки Eclipse, IBM WebSphere поддерживает среду выполнения SAP Web Application Server через SAP JAVA Connector, а MS.NET поддерживает эту среду через SAP .NET Connector.

Платформа ИВК "Юпитер". Эта платформа – российский продукт, который обеспечивает функции интеграции на уровне данных и обмена сообщениями. Он разрабатывался для потребностей государственных структур, которые выдвигают повышенные требования к защите информации и возможностям интеграции унаследованных и устаревших ПС.

ИВК "Юпитер" поддерживает и современные вычислительные платформы. Он представляет собой интегрированное средство, соединяющее характеристики виртуальной машины, транспортной магистрали, отдельные свойства систем документооборота и средства защиты информации. Состоит из двух высокоуровневых логических блоков:

1) первый обеспечивает общую функциональность на главном компьютере;
2) второй – связь между разными компьютерами;

На каждом компьютере имеется реализация так называемой унифицированной модели вычислительного процесса. Она соединяет среду выполнения ИВК "Юпитер" и набор библиотек. Объем функциональности достаточный для создания приложений в современных вычислительных средах типа Cloud Computing при использовании модулем API ИВК "Юпитер". Продукт обеспечивает контроль целостности вычислительного процесса: в начале загрузки среды выполнения ИВК "Юпитер".

Важное свойство – это встроенная возможность эмуляции IP поверх многих унаследованных транспортных протоколов, что позволяет гарантированную доставку сообщений в гетерогенной сети. В принципе, ИВК "Юпитер" предлагает весь традиционный функционал средств МОМ (расширенный возможностями документооборота).

Таким образом, из приведенного описания известных современных платформ индустриального изготовления приложений следует, что это типичные фабрики с базовыми средствами создания, компоновки и координации специальных ГОР и сервисов. Такие платформы позволяют выбирать необходимые готовые ресурсы, сервисы и инструментальные средства для реализации и конструирования новых приложений, способных функционировать в этих же или в других средах организации вычислений типа Drid и Cloud Computing.

Общие виды ресурсов фабрик программ

Исходя из теоретических достижений, закономерностей развития технологии программирования и анализа зарубежных вариантов фабрик программ на современных платформах компьютеров, новых идей взаимодействия разнородных программ, основанной на теории организации фундаментальных и общих типов данных (Fundamental Data Types – FDT, GDT (General Data Types – ISO/IEEC 11404), установлены общие черты, свойственные разным фабрикам [26–39]. Это прежде всего операционная среда (типа SUN ONC, MS.Net, Oberon, Babel, Grid, Eclipse и др.), метод программирования (компонентный, структурный, сервисный и др.), средства (ЯП, Rational Rose, CLR, и др.) и инструменты. Они поддерживают процессы линий изготовления ПП отдельных компонентов и систем, используя библиотеки готовых продуктов, сервис разных аспектов производства (данные, интерфейс, качество, управление, контроль, планирование, расчет разных затрат и др.). Главный ресурс фабрики – специалисты по производству программ (аналитики, программисты, инженеры, менеджеры, тестировщики, контролеры и т. п.).

Фабрика программ – это интегрированная инфраструктура сборочного производства ПП (компонентов, подсистем, систем, семейств систем, ИС, АСУ, АСУТП и др.), предназначенная для выполнения государственных, коммерче-

ских и других заказов на ПП [49]. Фабрика включает в себя комплекс средств, инструментов и сервисов для производства ПП на процессах ТЛ. Ядро фабрики – операционная среда и метод изготовления ПП (UML, компонентный, структурный, модульный, сервисный и др.). Обязательное условие сборочного конвейера – средства обеспечения связи разноязычных программ, аналогично реализации в MS.Net. При использовании сервисов в среду вводятся современные средства (веб-сервисы, веб-семантики и др.) для управления выбором и сборкой необходимых ресурсов при производстве ПП.

В инфраструктуру фабрики входят следующее: ТЛ или продуктовые линии, набор КПИ и набор методических средств, а также инструментов и сервисов для автоматизированного выполнения процессов линий изготовления продукта в операционной среде.

Фабрика софтвера – это согласованный набор процессов, средств и разных видов ресурсов для всего цикла создания тех или иных программных компонентов, приложений и систем.

Линии фабрики обеспечивают переход от ремесла к индустрии ПП с целью повышения производительности разработки продукта на процессах линии с заданными функциями, архитектурой и качеством. Эти линии содержат соответствующий набор средств разработки простых и сложных ПП из простых элементов продуктов. Им соответствует ЖЦ, например, реализованный в среде MS.Net с использованием каркасов (framework), DSL-языков и др. Линии сложных ТЛ или продуктовые линии, набор КПИ и методических средств, а также инструментов и сервисов автоматизированного их выполнения реализованы в ряде современных фабрик программ крупых фирм (IBM, Microsoft, Grid и др.).

Характеристика ресурсов фабрики

На рис. 1.8 приведены все виды ресурсов фабрики: технические, технологические , общесистемные, человеческие и др. Далее дается их характеристика.

Технические ресурсы: платформы, процессоры (Intell, IBM, Apple, MS; коммуникации (OSI, TCP/IP; компьютеры пользователей; файлы и серверы; локальные и глобальные сети; электронная почта (e-mail); тестеры и т. п.

Технологические ресурсы: библиотеки, репозитарии готовых ПП (КПИ, Reuses, Assets, Applications, Domains, Systems); методики методов программирования сборочного типа; руководства, методики по языкам интерфейсов объектов (IDL, API, DII, SIDL, XML, RDF и др.); стандарты (каркасы, шаблоны, контейнеры, процессы, проекты, системы и др.).

Общесистемные ресурсы: ОС, инструменты: клиент/серверные технологии; офисные системы (ридери / райтери форматов PDF, PS, HTML и т. п.); системы документооборота; утилиты (архиваторы, программы записи на носитель, конфигураторы и т. п.); средства защиты информации (антивирусные, парольные и др.); CASE-инструменты, трансляторы; графические инструменты; СКБД.

Человеческие ресурсы. Включают в себя группы разработчиков, служб управления и выполнения проектных работ (по планам, сетевым графикам), добивающиеся необходимого качества, выявления рисков, формирования конфигурации, проверки правильности реализации проекта и т. п.

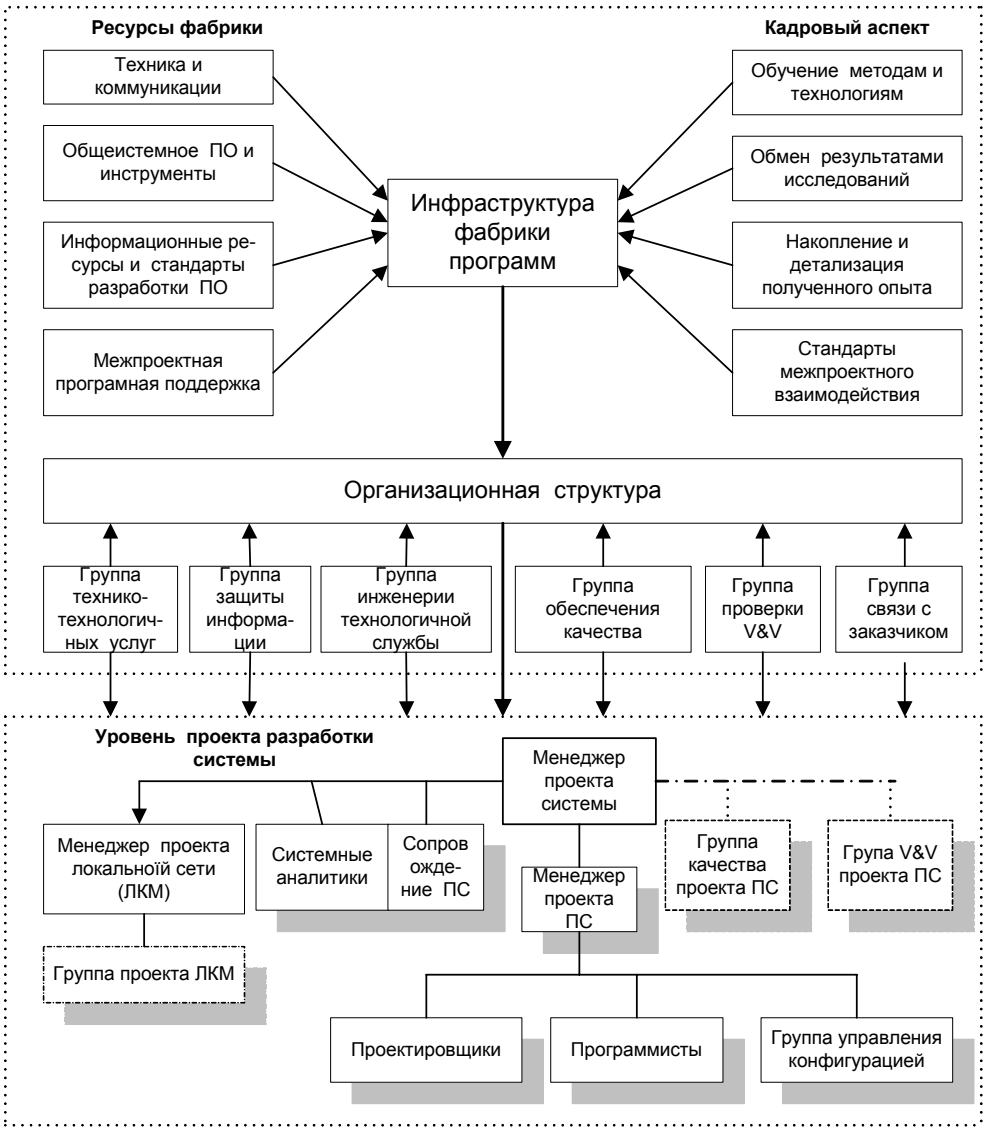


Рис. 1.8. Общая инфраструктура фабрики программ

Стандарт ISO/IEC 12207 определяет следующие группы служб поддержки фабрики:

1) технико-технологической поддержки (изучение рынка, приобретение CASE, ПП, консультации и т. п.);

2) технологической службы (сопровождения, поддержки ЖЦ, контроля и т. п.);

3) качества (SQA-группа) с функциями планирования и выполнения ЖЦ, проверка работ, контроль качества рабочих продуктов, документов ПП и т. п.;

4) верификации, валидации и тестирования компонентов или ПП на правильность задания требований, координации планов работ с менеджером, проверки правильности ПП в тестовой среде системы и др.;

5) руководителя проекта, который отвечает за финансовые и технические ресурсы, а также за выполнение проектных соглашений заказчика и управление разработкой ПП;

6) менеджера проекта, ответственного за разработку программного проекта фабрики, согласующего требования, решения и планы работ и реализации по всем группам по срокам и стоимости;

7) проектировщиков и программистов, которые отвечают за разработку проектных решений и программирование, разработку документов и разных выходных результатов;

8) руководителя конфигурации (ответственные за версию) ПП, который регистрирует версии ПП, сохраняет твердые копии и версии с размежеванием доступа к ним.

Эти группы необходимы при индустриальном и коллективном производстве ПП, как это есть в VS.Net.

Стандарты в области ПИ

Комитет по стандартизации ISO разработал ряд стандартов программной инженерии, которые регламентируют порядок разработки ПП, управления методами программирования сложных программ [33]. К ним относятся следующие.

Базовый процесс (БП) предназначен для "процессного продуцирования" ПП как вид инженерной деятельности по изготовлению ПП, включающий в себя операции оценки, измерения, управления изменениями и усовершенствованием ПП и БП, согласно стандарту ISO/IEC 15504–2007 ("Оценивания процессов ЖЦ ПЗ. Установки по усовершенствованию процесса"). Оценка зрелости организации или фабрики программ осуществляется с помощью модели зрелости CMM (Capability Maturity Models) Института SEI США, модели Bootstrap, Trillium и т. п. Согласно этим стандартам уровень зрелости организации зависит от наличия в ней ресурсов, стандартов, методик и способностей (зрелости) членов коллектива фабрики изготавливать ПП в заданные сроки и от стоимости.

Стандарт ISO/IEC 12207 "Жизненный цикл ПО" регламентирует разные направления деятельности специалистов по разработке, проектированию и управлению ПП, организации процессов (планирования, управления и сопровождения), измерения, оценивания продуктов и процессов. Наиболее важные из них – серия стандартов: ДСТУ ISO/IEC 14598 "Оценивания программного продукта", стандарт ДСТУ ISO 15939 "Процесс измерения", серия стандартов ISO/IEC 15504 "Оценивания процессов ЖЦ ПО", базовые стандарты качества – ISO 9001 "Системы управления качеством. Требования", ГОСТ 2844–1994, ГОСТ 2850–1994 и 9126 регламентируют разные аспекты обеспечения качества ПП.

Стандарт "The Software Engineering Body of Knowledge" (www.swebok.com), названный нами, "Ядро знаний SWEBOK" включает в себя десять разделов ПИ, распределенных по двум категориям. Первая – это методы и средства разработки (формирование требований, проектирование, конструирование, тестирование, сопровождение), вторая – методы управления проектом, конфигурацией, качеством и БП [44]. Методы ядра знаний соответствуют стандартным процессам ЖЦ с учетом потребностей конкретной фабрики программ и регламентированной последовательностью процессов, начиная с требований к разработке проектных

решений, определения каркасов ПП и выбора готовых компонентов для "наполнения" его соответствующим содержанием.

Стандарт "The Project Management Body of Knowledge" – PMBOK (IEEE Std.1490 "IEEE Guide adoption of PMI Standard. A Guide to the Project Management Body of Knowledge), разработан Институтом PMI [44, 45]. Он содержит описание лексики, структуры процессов и областей знаний: *управление содержанием проекта (*планирования с распределением работ); *управление качеством* и контроль результатов на соответствие стандартам качества; *управление человеческими ресурсами* согласно их квалификации и профессионализму.

Базовые компоненты фабрик программ

Рассмотренные виды операционных сред позволил определить необходимые атрибуты, свойственные любой фабрике программ. Принятие решения о их полноте и функциональности для производства ПП зависит от наличия финансов и знаний менеджеров, которые намерены заниматься изготовлением ПП определенного назначения. Экспериментальный вариант фабрики программ в ИПС НАНУ – бесплатная система Eclipse [53], содержащая базовые инструментальные средства для производства ПП из готовых компонентов повторного использования КПИ, а именно:

1) механизм плагинов в формате XML в средстве Plug Development Environment, которая обеспечивает автоматизированное подключение плагинов и новых инструментов (например, Protege, JAVA, RMI), репозиториев и готовых программ;

2) автоматизированное подключение новых меню к интерфейсу пользователя, иконок, сценариев и т. п.;

3) использование языка JAVA и механизма вызова RMI для описания разных программ и объединения их в выходном коде и т. п.

Эта система дополнена нами алгеброй операций компонентного программирования, средствами сборки, генерации и конфигурирования компонентов повторного использования в семействе систем [32, 37]. Метод порождения и генерации моделируется на процессах создания репозиториев компонентов, КПИ (сертификация, накопление, выбор, интеграция и др.) и сборки разнородных программных объектов применительно к сгенерированным членам семейства СПС в среде Eclipse. После исследования среды Grid, нами сделан вывод о необходимости дополения ее средствами генерации программных ресурсов парадигм программирования и сборки ПП, апробированных в среде Eclipse ИТК.

Исходя из полученной практики автоматизированной сборки разнородных программ в ЯП в среде ОС ЕС и анализа современных и зарубежных фабрик программ в системах (IBM, OMG, Microsoft, Oberon и тому подобное) [3–8], нами сформированы общие составные элементы фабрики индустрии (производства) программ, а именно:

1) *готовые программные ресурсы* (артефакты, программы, системы, reuses, assets, компоненты повторного использования – КПИ) и т. п.;

2) *интерфейс,* как спецификатор готовых ресурсов, независимо от языков программирования, в языке интерфейса (IDL, API, SIDL, WSDL, RAS и т. п.) [6, 7];

3) *технологические линии (ТЛ), продуктовые линии* (Product Lines) [1] из производства ПП;

4) с*борочный конвейєр* фабрики программ;

5) *методики и приемы* проведения работ на фабрике программ;

6) *среда* разработки программ в условиях фабрики.

Эти составные элементы были сформулированы автором и развиты в рамках фундаментальных проектов Института кибернетики (1980–1991) и ИПС НАНУ (1992–2010). В них впервые определена концепция *интерфейса* (1982г.), метод сборки разноязычных программ, ТЛ, 1987–1991 и средства автоматизации выпуска ПП.

Глава 4. ТЕХНОЛОГИЯ КОНВЕЙЕРНОЙ СБОРКИ СИСТЕМ

Методика создания ТЛ предложена автором в 1987 г. [8] и апробирована на 6 линиях автоматизированной информационной системы "Юпитер-470" Института кибернетики АН УССР для военно-морского флота СССР (1983–1991). Эти ТЛ стали первой работой по формализации и применению ТЛ в проектах разработки больших информационных систем. Концепция технологических линий была частично автоматизирована с помощью модулей посредников, которые генерировала система АПРОП для готовых разнородных объектов. Такой подход способствовал сокращению объема работ при сборке таких объектов и тестировании их интерфейсов. Построенные ТЛ в АИС – это первый вариант сборочного конвейера Глушкова. С их помощью было создано более 500 программ обработки данных для разных объектов АИС.

Одна из главных задач современного программирования – создание теоретических и прикладных основ построения сложных программ из более простых программных элементов, которые записаны на современных ЯП. Фактически решение этой задачи осуществляется путем объединения или интеграции разнородных программных ресурсов, включая модули, простые программы реализации функций некоторой предметной области.

Цель метода сборки – интеграция готовых ресурсов, накопленных в Интернете, в новые программные структуры программ или систем. Объекты сборочного программирования – готовые модули (макромодули, подпрограммы, функции, алгоритмы, программы и т. п.), описанные в разных ЯП четвертого поколения (Алгол, Фортран, ПЛ1, Кобол и др.) и объектных ЯП следующего поколения. Готовые модули и другие ресурсы накапливались во входном или выходном коде в библиотеках, в фондах алгоритмов и программ, а также в архивах самих разработчиков этих ресурсов.

Метод сборки – способ соединения разноязычных объектов в ЯП, основанный на теории спецификации и отображения типов и структур данных ЯП, представленных алгебраическими системами.

Основу формализмов этой алгебры составляют типы данных, операции над ними и функции эквивалентного преобразования одних типов в другие. Соединение пар объектов в ЯП осуществляется с помощью оператора вызова, в списке параметров которого задаются значения формальным параметрам. Они прове-

ряются на соответствие типов данных утверждениями алгебры, доказывающих необходимые и достаточные условия отображения данных в классе ЯП. Результат отображения – операторы эквивалентного преобразования типов данных в посреднике соединенных объектов.

Сборочное программирование характеризуется построением программ из готовых "деталей" – программных ресурсов разной степени сложности. Элементы процесса сборки есть во многих методах программирования: сверху-вниз, снизу-вверх и т. д.

Программисты, разрабатывая программы традиционными методами, определяют повторно использованные операторы и оформляют их в виде отдельных, самостоятельных фрагментов или подпрограмм для последующего применения.

Метод сборочного программирования является одним из методов программирования, имеет общие закономерности повторного использования программных средств и обеспечивает качественный процесс сборки КПИ. Процесс сборки ресурсов характеризуется: комплектующими деталями и узлами, схемой сборки (взаимосвязями отдельных компонентов и правилами взаимодействия), технологией сборочного конвейера. Комплектующие метода – это простые программные элементы (модули, объекты, компоненты, сервисы, аспекты и др.), описаны в табл. 1.2.

Таблица 1.2. **Схема эволюции программных ресурсов**

Элемент сборки	**Описание элемента**	**Схема взаимодействия**	**Представление, хранение**	**Результат сборки**
Процедура, подпрограмма, функция	Идентификатор	Прямой вызов, оператор вызова	Библиотеки подпрограмм и функций	Программа
Модуль	Паспорт модуля, Интерфейс связи	Вызов модулей, интеграция модулей	Банк, библиотеки модулей	Программа с модульной Структурой
Объект	Описание класса	Схема экземпляров классов, вызов методов	Библиотеки классов	Объектно-ориентированная программа
Компонент	Описание логики (бизнес), интерфейсов (APL, IDL), схемы развертки	Удаленный вызов в моделях (COM, CORBA, OSF, …)	Репозитарий КПИ, 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

Дадим определение основным прогрраммным элементам

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

Объект – базовое понятие в объектно-ориентированном программировании, которое имеет свойства наследования, инкапсуляции и полиморфизма. Он объединяет данные и операции (методы) над ними. Объекты взаимодействуют между собой через сообщения или запросы. Объекты с общими свойствами и методами образуют класс. В нем каждый объект является экземпляром этого класса объектов. В языках С ++, С# создано несколько библиотек классов объектов общего применения.

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

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

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

Из приведенных программных ресурсов собираются программные структуры разной сложности и степени готовности: программы, комплексы, пакеты прикладных программ, программные системы и т. д.

Среди этих элементов для практического применения выбираются базовые программные объекты, как готовые компоненты метода сборочного программирования.

Для технологичности сборки все объекты сборки должны иметь паспорта, которые содержат данные, необходимые для информационного соединения и организации общего функционирования в рамках программной системы более сложной структуры. Важное условие сборки заключается в наличии большого числа разнообразных комплектующих, которые обеспечивают решение широкого спектра задач из разных предметных областей.

Схема сборки. Под схемой сборки понимается схема взаимодействия программных ресурсов, которая определяется непосредственными обращениями к отдельным из них или последовательности их выполнения. При этом взаимодействие каждой пары объектов зависит от использования общих данных. В общем случае схеме сборки соответствует совокупность моделей, которые отображают разные типы связей между компонентами: передача управления, обмен информацией, условия общего функционирования и т. д.

Операции сборки выполняются согласно паспортам объектов и правилам соединения. Информация в паспортах должна быть систематизирована и выделена

в такие отдельные группы: переданные данные и их типы, вызывающие объекты, совместно используемые файлы и т. д. Правила соединения определяют совместимость объединяемых элементов и содержат описание функций, необходимых для согласования их разных характеристик, представленных в паспортах.

4.1. Сущность сборочного конвейера

Процесс сборки может проводиться ручным, автоматизированным и автоматическим способами. Как правило, последний способ невозможен, это связано с недостаточно формальным определением программных объектов и их интерфейсов. Ручной способ нецелесообразен, поскольку сборка готовых ресурсов представляет собой большой объем действий, которые носят скорее рутинный характер, чем творческий . Наиболее приемлемый – автоматизированный способ сборки в среде системы, которая по заданным спецификациям (моделям) программ осуществляет сборку посредством стандартных правил их соединения под управлением человека. Средства, которые поддерживают данный способ сборки, называются инструментальными средствами сборочного программирования.

К ним относятся средства комплексации (объединения) ресурсов в более сложный объект; средства описания интерфейсов и использования моделей сборочного программирования (совокупность моделей составления разных программных объектов).

Из рассмотренной схемы сборки выделяются операции и условия применения:

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

2) проектирование модели создаваемого объекта, элементами которого являются готовые программные элементы, определенные на множестве данных выбранной предметной области;

Необходимые условия применения метода такие:

1) большое количество разнообразных программных элементов, как объектов сборки;

2) паспортизация программных элементов сборочного конвейера;

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

4) создание сборочного конвейера технологии применения разработанных объектов для использования их в более сложных системах;

5) реализация интерфейсов между отдельными модулями и/или ресурсами, которые обеспечивают их "стыковку" или связь.

Последние условия означают, что должны существовать определенные формы представления ПС как знаний о предметных областях с точки зрения проектирования и разработки этих систем.

4.2. *Линии программ и* Product Lines

Позднее, в 2004 г. появилась альтернатива ТЛ – линии продуктов (Product Lines) Института SE США http://sei.cmu.edu/productlines/frame_report/) Этот подход основан на интеграции семейств продуктов из готовых, ранее разработан-

ных ПП. Его используют в коммерческой реализации ПП. ТЛ сложно модифицировать и эксплуатировать. Сегодня специалисты пытаются создать методы реинженерии ПП, которые решали бы некоторые проблемы кризиса сложности больших ПП, созданных традиционными методами программирования.

Способом уменьшения сложности больших программ является сборка из готовых стандартизованных модулей, компонентов и объектов. Однако объектно-ориентированные языки программирования по сравнению с традиционными еще не получили промышленной реализации. Идея автора сборки сложных ПС из готовых программных ресурсов (модулей, объектов, компонентов) реализована на фабрике программ студентами Киевского национального университета (КНУ) на веб-сайте http://programsfactory.univ.kiev.ua, как инструмент создания ПС и снижения их сложности. Именно идея сборки средствами фабрики программ отображает смысл создания сложных программных продуктов в работах автора (Compositional programming: theory and practice, K. M. Lavrischeva, 2009, Volume 45, Number 6, Pages 845–853, Theory and practice of software factories K. M. Lavrischeva, 2011, Volume 47, Number 6, Pages 961 – 972.).

Эта идея получила наибольшее распространение и на системных фабриках AppFab во многих общесистемных операционных средах (VS.Net, IBM, Intel, Unix, JAVA и др.), а также в коммерческих конвейерах Дж. Гринфильда и Г. Ленца (потоковая сборка, 2005), М. Фаулера (Continuos integration, 2007), ЕПАМ, Авдошина (сборка use case, 2008) и др. [44 – 54].

Если для коммерческих продуктов, созданных традиционными методами, необходимо применять методы реинженерии и реверсной инженерии при построении моделей вариабельности для обеспечения задач изменения и адаптации продуктов к новым условиям функционирования, то при решении проблемы индустрии программ и кризиса сложности конкретным ответом являются варианты ПС, создаваемые из готовых ресурсов в сборочном программирования. Один из аспектов индустрии – вариантность ПС через точки вариантов в интерфейсе, защищена в диссертации А.Л. Колесника на Ученом совете Киевского национального университета имени Тараса Шевченко 26.12.2013.

Базис сборочного конвейера – готовые модули (детали), которые выполняют программные модули, компоненты, обладающие определенной структурной и функциональной целостностью. Их описание стандартизировано для обеспечения четко определяемого и контролируемого информационно-логического взаимодействия, путем обмена информацией с другими модулями, задаваемых соподчиненной схемой сборки конвейера. Сборочный конвейер более эффективен по сравнению с традиционными методами программирования. В нем выполняется комбинирование (конфигурирование) готовых модулей для быстрого решения задачи из некоторого класса проблем. Ориентация на класс задач делает его особенно актуальным, когда готовыми накопленными в мире программными модулями могут пользоваться все работающие в Интернете, решая некоторые задачи, для которых есть много подходящих готовых программных модулей и компонентов.

Технология программирования сложных систем базируется на методах проектирования, основанных на моделях ООП, UML, ОКМ, а также на концептуальных моделях ПрО, создаваемых онтологическими методами и средствами.

4.3. Метод сборки специализированных технологий

Разработанный метод сборки программ оказался адекватным и к процессам их создания. Усовершенствованный метод сборки применительно к процессам программирования отдельных элементов ПС направлен на сборку специализированных технологий программирования из процессов и способов их нотаций для реализации конкретной предметной области. Этот подход был сформулирован автором в 1987 г., как технологическая подготовка разработки (ТПР) новых технологий программирования. Такой подход является оригинальным, он не имеет аналогов. Сформулированы задачи ТПР, дано формализованное определение специализированных технологий программирования для решения разных задач (АСУ, СОД, АСНИ и др.) и определен язык спецификации ТЛ. К настоящему времени создан стандарт спецификации процессов – язык BPMN и система реализации описаний процессов в этом языке [8, 55 – 57].

Таким образом, к основным элементам ТПР отнесены:

1) объект разработки (его начальное, промежуточные и конечное состояния);
2) методы программирования, средства и инструменты, обеспечивающие изменение состояний объекта;
3) модели технологических процессов (ТП) и ТЛ;
4) инженерные методы управления процессами разработки программ по ТЛ.

Для новых ТП и ТЛ в рамках ТПР определены классы объектов, которые разделяются на понятийные, технологические и инструментальные. В класс понятийных объектов входят модели данных, задач и программ. Задачи разбиваются на классы функций, каждой из которых определяется подмножество схем данных и их программ элементов (заготовок). В классе типовых задач определяются модели типовых и специализированных программ реализации функций ПрО. К классу технологических объектов отнесены модели формализованного представления состояний объектов и процессов их разработки. Это система моделей для фиксации проектных решений в ходе разработки; модели процессов и линий для формализации деятельности исполнителей; модель качества программных объектов для управления качеством разработки на всех этапах ЖЦ; модель эксплуатационных документов для формирования рабочей документации в ходе разработки ПП по заданной технологии.

Определены этапы жизненного цикла ПС и дано определение восьми основным этапам жизни отдельных объектов ПС, каждый из которых отображается в одном или нескольких ТП линии. Каждый процесс ТП трактуется как вероятностный автомат, переводящий объект разработки из одного состояния в другое. Заключительным состоянием является готовый программный продукт.

К классу инструментальных объектов относятся методы программирования, средства и инструменты, из которых подбираются более подходящие для создаваемой ТЛ. С их помощью осуществляется целенаправленное преобразование состояний объекта разработки на каждом ТП. Для формального определения входящих в ТЛ процессов разработан язык спецификации ТЛ, являющийся первой попыткой формализованного задания технологий программирования. Основные

элементы этого языка – технологические процессы и составляющие их технологические операции, в состав которых входят операции разработки объекта в соответствии с моделью жизненного цикла, документирования и контроля.

Основными атрибутами каждой операции разработки являются: вид состояния объекта, его входные и выходные данные, метод и средство разработки. Описание контрольной операции включает в себя: наименование показателя качества, контролируемый оценочный элемент, метрика и метод оценки. Операция документирования способствует формированию определенного вида модели документа и набору точек для внесения соответствующих текстов в формируемую документацию.

Данный метод апробирован при создании конкретных функционально-ориентированных технологий программирования прикладных программ, работающих с базами данных.

В настоящее время проведено усовершенствование методологии проектирования ТЛ в рамках созданной экспериментальной фабрики программ КНУ [55 – 60], результаты которой будут приведены детально в разделе 3.

Реализация интерфейсов КПИ на линиях

При разработке КПИ на основе ТЛ используется теория сборочного программирования по обеспечению интеграции разноязычных модулей с помощью интерфейсов [5, 32, 44].

Реализация интерфейса базируется на механизмах преобразования нерелевантных типов данных объектов в ЯП, передаваемых через аппарат формальных и фактических параметров между ними. Параметрам могут соответствовать данным, которые относятся к общим ТД (GDT) и фундаментальным ТД (FDT). Они могут быть не совместимыми между собой и платформой и поэтому требуют соответствующих видов преобразования передаваемых параметров.

При появлении ЯП сформировались фундаментальные FDT и позднее общие GDT в стандарте ISO/IEC 11404. Их описание дано выше, а здесь подводится итог и подход к реализации.

К фундаментальным типам данных FDT относятся:

- простые ТД (real, integer, char и др.);
- структурные ТД (array, record, vector и др.);
- сложные ТД (set, table, vector, sequence и др.).

Они используются во всех ЯП для описания программ и реализуются трансляторами.

К общим типам данных GDT относятся :

- примитивные ТД (character, integer, real, complex , enumerated и др.);
- агрегатные ТД (choice, pointer, set, bag, sequence и т. п.);
- сгенерированный ТД, который получаются в результате генерации из одного и нескольких других ТД;

Эти типы данных используются в компонентах повторного использования КПИ (reuse, artifact, object, component, service). Они описываются в ЯП в форме стандарта WSDL, Grid и интерфейсов КПИ в IDL. Интерфейсы сохраняются в репозитории интерфейсов, а готовые компоненты в репозитории КПИ.

Раздел 2
ПАРАДИГМЫ ПРОГРАММИРОВАНИЯ

Глава 1. МОДУЛЬНОЕ ПРОГРАММИРОВАНИЕ. БАЗОВЫЕ ПОНЯТИЯ

Одна из ключевых проблем современного программирования – повторное использование модулей и компонентов (КПИ). Ими могли быть программы, подпрограммы, алгоритмы, спецификации и т. п., пригодные для использования при разработке новых более сложных ПС. В них материализуется многолетний опыт компьютеризации разных знаний в деятельности человека, который может их применять непосредственно либо путем настройки и адаптации КПИ к новым условиям среды обработки. После модулей, используемых долгое время в практике программирования, появились КПИ, готовые к применению. Использование КПИ в программировании дало возможность усовершенствовать метод снизу–вверх проектирования сложных программ из простых программных элементов, находящихся в библиотеках. Программирование прошло длинный путь своего развития – от элементов библиотек стандартных машинных подпрограмм общего назначения на конкретной ЭВМ до современных библиотек программ и КПИ нового поколения в современных языках С, С++, Basic, JAVA и др. Каждый из этих путей развития и совершенствования готовых КПИ определялся возможностями вычислительной техники, операционных сред и достижений в области теории и практики программирования сложных программ, комплексов и систем.

В этот период широко использовались большие ЭВМ (ЕС ЭВМ, БЭСМ-6 и др.), снабженные компиляторами с подмножества ЯП: Ассемблер, Алгол, ПЛ-1, Фортран и Кобол. Разнообразие ЯП и большой объем памяти ЕС явились базисом реализации идеи сборки программ на машинах ЕС ЭВМ. Исследование проблем сборки модулей проводилось в отделе автора по трем направлениям.

1. Анализ средств создания крупных программных комплексов с монолитной и динамической структурой, а также особенностей объединения программ, отличающихся представлением типов и структур данных ЯП для их формализованного описания и определения стандартизованного задания модулей сборки, как это принято в промышленности. Исследовался и оформился базис теории абстрактных типов данных и подходов к формальному их преобразованию. На основе этой теории и теории алгоритмических алгебр Глушкова были исследованы все типы и структуры данных подмножества ЯП ЕС и построены алгебраические системы формального преобразования типов и структур данных ЯП, позволяющие установить взаимно однозначное соответствие данных каждой пары ЯП указанного подмножества. В результате были разработаны 65 функций преобразования одного типа данных к другому в подмножестве ЯП и определен специальный класс интерфейсных модулей, как механизмов связывания разноязычных модулей.

2. Сформированное в практике программирования понятие – повторная подпрограмма библиотеки машинных программ (В.Б.Курочкин, Москва) было изменено до уровня исходного представления на любом из ЯП в специальном хранилище - Банке модулей с функциональными разделами (вычислительная математика, экономика, АСУ и др.). Решение проблемы сборки разных модулей в исходном представлении на ЯП из Банка модулей основывалось на разработке стандарта описания исходных модулей, который кроме описания тела модуля, включал описание его паспорта – информационного раздела, содержащего задание типов данных входных и выходных параметров и операторов вызова других модулей из Банка модулей. Определены новые принципы и механизмы стыковки исходных разнородных модулей, основанные на интерфейсных модулях.

3. Разработан метод сборки прикладных модулей и программ, реализация которого позволила сформировать не только новый вид программирования – сборки модулей, интерфейсов, схем линий сборки модулей. Он реализует функции автоматизации предметной области в рамках системы АПРОП [6, 8, 16, 60 – 70].

1.1. Понятие модуля и интерфейса. Метод их сборки

Модуль – это логически законченная часть программы, выполняющая определенную функцию, обладающая свойствами завершенности, повторного использования и др.

Модуль возник как обобщение понятия стандартных подпрограмм и процедур, которые описывались в ЯП. В их заголовке задавались внешние входные данные, а в теле модуля – операторы вычислений и вызовов подпрограмм по их имени и списку фактических параметров. Последовательность и число формальных параметров соответствовало фактическим параметрам. Вызов заготовок на одном ЯП не является проблемным, так как типы данных параметров совпадают с типами данных ЯП.

Идею сборки разнородных модулей в системе АПРОП по принципу сборочного конвейера (как в автомобильной промышленности) высказал В. М. Глушков 05.03.1974 г. на семинаре ведущих специалистов Института кибернетики АН УССР. Она послужила развитию автоматизации больших программ разными методами: система "Проект" на основе формализованных технических заданий (Ю. В. Капитонова, А. А. Летичевский); система автоматизации программ – АПРОП (Е. М. Лаврищева) из модулей; пакеты прикладных программ (ППП) численных методов (И. Н. Молчанов), статистики (И. В. Сергиенко, И. Н. Парасюк); генератор систем обработки данных Макробол (Л. П. Бабенко), технологический комплекс программиста (И. В. Вельбицкий); система "Мультипроцессист" САА (Г. Е. Цейтлин, Е. Л. Ющенко) и др. Эти системы создавались на ЭВМ (ЕС, БЭСМ-6 и др.), снабженные компиляторами с ЯП: Ассемблер, Алгол-60, Фортран, ПЛ-1, Кобол и др. Разнообразие ЯП, набор библиотечных программ в этих ЯП и большой объем памяти таких машин стали базисом реализации тезиса сборки сложных ПП Глушкова [1].

Для связи разноязычных модулей и передачи данных между ними нами впервые было определено понятие **интерфейса**, как связника модулей. С общей точки зрения *интерфейс* – это модуль-связник, посредник двух отдельных про-

граммных элементов, в котором заданы внешние переменные и структуры данных для обменной связи информацией между модулями в ЯП. В виду отличий типов передаваемых данных в системе АПРОП были разработаны библиотека интерфейса (65 функций) по преобразованию нерелевантных типов данных ЯП и генератор интерфейсных модулей-посредников (новый тип модуля) для связи и преобразования типов данных объединяемых разноязычных объектов в ЯП.

Система АПРОП стала основным сборочным инструментом, связывающим разные модули в сложные структуры ПП. Каждый модуль имел паспорт с информацией о типах данных, параметрах, вызовах других модулей из Банка готовых модулей, сгруппированных по разным разделам (вычислительная математика, экономика, АСУ и др.).

Интерфейс – новый элемент сборки

Главным нововведением в концепции сборочного программирования является интерфейс (межмодульный и межъязычный). Его задачи сформулированы в 1976 г. и намного опередили зарубежные разработки. В настоящее время интерфейс сохранил свою актуальность и выступает в качестве главной доминанты взаимодействующих компонентов и объектов в современных глобальных и сетевых средах [60 – 68].

Межмодульный интерфейс – это интерфейсный модуль-посредник между двумя взаимодействующими программными объектами, выполняя функции передачи и приема данных между ними. Впервые разработан язык определения интерфейсов (ЯОИ) для описания операторов вызова модулей, параметров и их типов, а также операций проверки правильности обмена данными.

Межъязычный интерфейс – совокупность средств и методов преобразования структур и типов данных между ЯП с помощью алгебраических систем и функций (и макроопределений) библиотеки интерфейса для взаимно однозначного обмена данными между разноязычными модулями через механизмы интерфейса. Библиотека интерфейса в момент широкого использования ЕС была передана в 50 организаций СССР для самостоятельного применения при работе с разными языками ОС ЕС [16].

Созданная нами концепция интерфейса модулей, как средства связи разных типов объектов в ЯП путем автоматически генерируемого интерфейсного модуля-посредника, является первой отечественной парадигмой интерфейса в программировании, реализованной в 1975–1982 г. в системе АПРОП. Гораздо позже в 1985–1990 годах появились средства описания интерфейсов объектов за рубежом – языки API (Application Progpam Interface) и IDL (Interface Definition Language).

В основе реализованного нами сборочного программирования в системе АПРОП лежит метод сборки (интеграции) сложных программ и специализированных технологий программирования для классов задач обработки данных.

Метод сборки разнородных модулей

Данный метод базируется на теории абстрактных типов данных ЯП и подходах к формальному преобразованию типов данных ЯП связываемых программ, различающихся функциями, данными, их значениями и форматами представления их трансляторами и формой представления в памяти ЭВМ.

Модули, интерфейсный связник, библиотека интерфейсных функций и метод сборки – основа нового сборочного программирования, реализованные в автоматизированной системе АПРОП [20 – 24]. В первой публикации по системе АПРОП и методу сборки модулей приведена концепция В.М.Глушкова на представление фабрик программ на машинах ЕС, автоматизирующих связи разноязычных модулей (1976).

Метод сборки задавался операторами Link <имя модуля>&<имя модуля>. На основе такого оператора генерировался модуль-связник, в функции которого входило отображение фактических параметров модуля в типы другого модуля, проверка соответствия параметров (количество и порядок), форматов данных в ЯП и в памяти машины и др. Сгенерированный модуль-связник как посредник содержит обращения к элементам библиотеки интерфейса, которые выполняются в момент перехода связника от одного модуля к другому и обратно.

Процесс сборки реализован компонентами системы:

1) обработка паспортных данных модулей;
2) анализ операторов Call и Link и составление задания на их обработку;
3) генерация модуля-посредника, составление таблицы матрицы соответствия пар компонентов и преобразование ТД связанных модулей (b-boolean, c-character, i-integer, r-real, a-array, z-record и др.) через обращение к функциям библиотеки интерфейса;
4) интеграция пар модулей и их размещение в базе данных системы для сборки всех пар в сложную структуру ПС;
5) трансляция и компиляция модулей агрегата к виду готовой программной структуры ПП;
6) трассировка интерфейсов и отладка функций модулей в каждой паре БП;
7) тестирование БП в целом;
8) формирование готового ПП для запуска БП и руководства инсталляцией и выполнением.

Разработка системы выполнялась по договору с Министерством радиопромышленности СССР, как составная часть программных проектов "РУЗА" и "Прометей" (В. В.Липаев). Система АПРОП стала первой апробацией автоматизации взаимосвязи разноязычных модулей с помощью модулей-посредников для языков четвертого поколения.

Разработанный метод сборки и программные средства его поддержки стали базисом автоматизированной инженерии программирования модулями, получившей название АПРОП. В первой публикации по методу сборки модулей были приведены основные пути реализации индустрии программ на машинах ЕС (1976) [4].

Главные объекты системы АПРОП: модуль, модуль-посредник, межмодульный интерфейс, Банк модулей и метод проектирования систем снизу–вверх с выбором готовых модулей из Банка модулей и их сборки в новые программные структуры. Базовая концепция системы – интерфейс как аппарат связи ЯП и модулей, записанных в разных ЯП (Фортран, ПЛ/1, Алгол-60, Ассемблер, Кобол). Каждый исходный интегрируемый ("погружаемый") в АПРОП прикладной модуль имел паспорт, в котором описывались сведения о назначении, объеме, параметрах и др. Среда была одна для всех – ОС ЕС.

Множество троек из вызываемого и вызывающего модулей в разных ЯП и модуля-посредника объединялись в системе АПРОП в агрегат – монолитный продукт на ЕС ЭВМ, предназначенный для решения класса прикладных задач. В функции посредника входило отображение формальных и фактических параметров, проверка соответствия передаваемых параметров (количества и порядка расположения), а также их типов данных. Типовая схема связи разноязычных объектов показана на рис. 2.1.

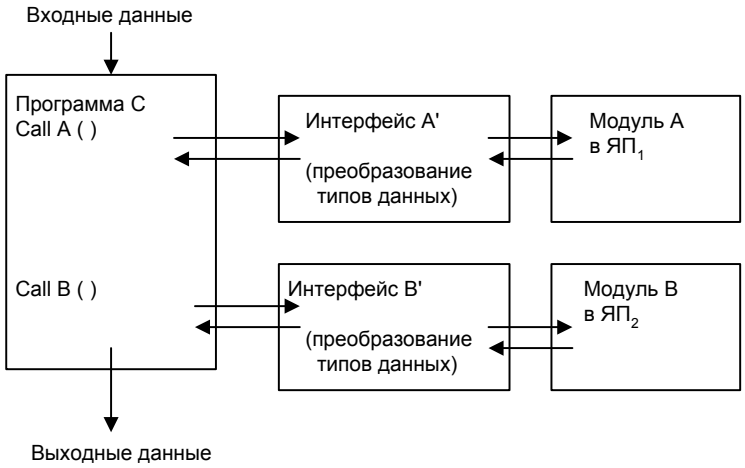


Рис. 2.1. Схема вызовов модулей A и B через интерфейсы A' и B'

На схеме приведена программа С, в которой содержатся два вызова – Call A ( ) и Call B ( ) с параметрами. Эти вызовы "проходят через" интерфейсные модули-посредники A' и B', которые осуществляют функции преобразования данных и их передачу модулям A и B. После выполнения модулей A и B результаты преобразуются обратно к виду программы С. Если типы данных параметров – не релевантные (например, передается целое, а результат – вещественное или наоборот), то в функции посредника входит прямое обратное их преобразование. Сгенерированный модуль-посредник включал операции обращения к вызываемому модулю, передаваемые параметры и функции их преобразования из библиотеки интерфейса.

В общую структуру системы АПРОП входят следующие компоненты:

1) обработка паспортных данных модулей в ЯП;

2) анализ описания параметров модулей и составление задания на обработку несоответствующих типов данных, проверка правильности передачи параметров как по их количеству, так и по соответствию типов данных в классе ЯП;

3) преобразование типов данных в ЯП (b-boolean, c-character, i-integer, r-real, a-array, z-record и др.) в виде обращения к функциям библиотеки интерфейса;

4) генерация модулей-посредников и составление таблицы связи пар компонентов;

5) интеграция пар модулей и их размещение в структуре программного агрегата;

6) трансляция и компиляция модулей в ЯП агрегата в виде готовой программной структуры;

7) трассировка интерфейсов и отладка функций модулей в каждой паре агрегата;

8) тестирование программного агрегата в целом;

9) формирование программ запуска программного агрегата и документации.

В системе реализовано два типа интерфейса – интерфейс модулей и пары ЯП, т. е. межмодульный и межъязычный интерфейсы.

Межмодульный интерфейс – компонент системы для генерации интерфейсных модулей-связок взаимодействующих между собой модулей.

Межъязычный интерфейс – компонент системы, содержащий набор функций и макроопределений в классе типов данных и структур множества ЯП, а также метод их релевантного преобразования.

Теоретическое обобщение концепции и метода в системе АПРОП – *теория сборочного программирования*, защищена в докторской диссертации Лаврищевой Е. М. ("Модели, методы и средства сборочного программирования", 1989), кандидатской диссертации В.Н. Грищенко ("Реализация межмодульного интерфейса в системе АПРОП", 1991) и отображена в монографиях [5, 6].

Таким образом, *интерфейс модулей* как средство связи разных типов объектов в ЯП – первая отечественная парадигма интерфейса в программировании практически реализована в системе АПРОП (1974–1985 гг.). Интерфейс развивался за рубежом в проектах MIL, SAA, OBERON для комплексирования модулей на разных вычислительных машинах. В настоящее время идея связи программ через интерфейс для класса современных языков описана в [16].

Система АПРОП передана в ЕрНУВЦ с документацией объемом более 1000 стр., внедрена в 52 организациях бывшего СССР и отмечена премией Совета министров СССР (1987 г.), включая автора (Е. М. Лаврищева). Теория сборочного программирования, защищена в докторской диссертации автора ("Модели, методы и средства сборочного программирования", 1989) и отображена в монографиях [5, 6].

Сборка готовых модулей. В системе АПРОП сборка базируется на совокупности модулей, их паспортах и операторах сборки частей программ и сложных систем:

1) оператор сборки Link, задающий сборку двух разноязычных объектов или модулей графа;

2) Link seg A (A2, A3, *A4) – связать модули A, A3 и A4 в сегментную структуру A, где A4 вызывается динамически;

3) Link Prog B ((B1, B2), C= X(C1), D=(Y, D1=Y1)) – объединить модули B1, B2, к ним присоединить C и D с параметрами C1. Y, D1.

4) оператор //EXEC *модуль* A_1 //PL *Trans* A_1 .

5) генерировать интерфейсный связник mod-interface generation for $A_1 \cap A_2$ и др.

Правила сборки определяют совместимость объединяемых объектов, которые содержат описание функций для согласования разных характеристик, представлены в их паспортах.

Процесс сборки объектов может проводиться ручным, автоматизированным и автоматическим способами. Как правило, последний способ невозможен, связан с недостаточно формальным определением программных КПИ и их интерфейсов. Ручной способ нецелесообразен, так как сборка готовых КПИ представляет собой большой объем действий, которые носят скорее рутинный, чем

творческий характер. Наиболее приемлемый способ – это автоматизированная сборка, когда по заданным спецификациям программ осуществляется сборка с помощью стандартных правил сборки разнородных объектов.

Средства, которые поддерживают данный способ сборки, называют инструментальным средствами сборочного программирования. К ним относятся средства комплексирования (объединения компонентов в более сложный объект); интерфейсные средства описания и использования моделей программирования (совокупность моделей интеграции разных программных объектов).

Необходимыми условиями применения этого метода программирования является:

1) наличие большого количества разнообразных КПИ, как объектов сборки;
2) паспортизация объектов сборки;
3) наличие довольно полного набора стандартных правил сопряжения объектов, алгоритмов их реализации и средств автоматизации процесса сборки;
4) технологии линией с последовательностью операций постепенного изготовления и установления связей между КПИ при образовании системы или семейства ПП.

Последнее условие означает, что должны существовать определенные формы представления ПС как знаний о предметных областях, универсальные с точки зрения проектирования и разработки ПС. Основное задание сборки – выявление типов связи, описание их в интерфейсе и реализация в виде посредника интерфейсов между отдельными модулями и/или компонентами, который обеспечивают их "стыковку" или связь в процессе выполнении в некоторой среде.

Развитие идеи сборки. Идея сборки с помощью интерфейса развивалась за рубежом в проектах MIL, SAA, IBM, Sun, Oberon, CORBA и др. В настоящее время идея сборки стала типовой в классе традиционных и современных ЯП. Она реализована аналогично в названных системах и базируется на теории преобразования нерелевантных типов данных в ЯП и описана в руководствах по применению. Новые подходы к сборке опубликованы в ряде работ, в том числе у И. Бея («Взаимодействие разноязычных программ», 2005) и др. Система CORBA для объектных элементов реализовала универсальный управляющий связник – брокер объектных запросов, который связывает готовые объекты-методы и компоненты в любых ЯП через посредники – stub (туда) и skeleton (обратно). Интерфейсный посредник объектов описывается в новом языке IDL (Interface Definition Language). В нем параметры передачи данных помечаются *in* и *out* между разнородными объектами в любых ЯП. Данные готовых объектов и КПИ содержат описание *типов данных* в соответствующем ЯП и при их передачи от одного объекта к другому они проверяются на соответствие описания в посреднике stub, их релевантность параметров из skeleton.

1.2. Теория сборки разнородных модулей

Метод сборки модулей в новые программные системы сложной структуры разработан Е. М. Лаврищевой и В. Н. Грищенко [60 – 69]. Он включает в себя математические формализмы определения связей (по данным и по управлению) между объектами сборки и генерации интерфейсных модулей для каждой

пары объединяемых модулей. Связь или сопряжение модулей задается оператором вызова CALL, в котором представлены фактические параметры вызываемому модулю.

Сущность задачи сборки пары разноязычных модулей состоит в определении взаимно однозначного соответствия между задаваемым множеством фактических параметров $V = \{v_1, v_2, ..., v_к\}$ вызывающего модуля и соответствующим множеством формальных параметров $F = \{f_1, f_2, ..., f_к\}$ вызываемого модуля, а также в отображении типов данных одних параметров в другие. Если отображение не удается выполнить, то задача автоматизированной связи для данной пары модулей считается неразрешимой.

Преобразование типов данных осуществляется путем построения алгебраических систем, содержащих для каждого типа множество значений и операций над ними. Каждой операции преобразования типов данных соответствует изоморфное отображение одной алгебраической системы в другую.

Алгебраические системы построены в классе простых типов данных ЯП (t = *b* (*bool*), *c* (*char*), *i* (*int*), *r* (*real*)) сложных типов данных (t = *a* (*array*), *z* (*record*), *u* (*union*), *e* (*enum*)) и допустимых видов их преобразования. Преобразования между типами массивов и записей сводятся к определению изоморфизма между основными множествами соответствующих алгебраических систем с помощью операций изменения уровня структурирования данных – селектора и конструирования. Для массива операция селектора сводится к отображению множества индексов на множество значений элементов массива. Аналогично такая операция определяется для записи как отображение между селекторами компонентов и самими компонентами.

Формально преобразование P неэквивалентных типов данных в ЯП выполняется следующими этапами.

Этап 1. Построение операций преобразования типов данных $T = \{T_\alpha^t\}$ для множества языков программирования $L = \{ l_\alpha \}_{\alpha=1, n}$.

Этап 2. Построение отображения простых типов данных для каждой пары взаимодействующих компонентов в $l_{\alpha 1}$ и $l_{\alpha 2}$ и применение операций селектора S и конструктора C для отображения сложных структур данных в этих языках.

Формализованное преобразование типов данных осуществляется с помощью алгебраических систем для каждого типа данных T_α^t:

$$G_\alpha^t = <X_\alpha^t, \Omega_\alpha^t>,$$

где t – тип данных; X_α^t – множество значений, которые могут принимать переменные этого типа; Ω_α^t – множество операций над этими типами данных.

Простым и сложным типам данных современных ЯП соответствуют классы алгебраических систем: $\Sigma_1 = \{G_\alpha^b, G_\alpha^c, G_\alpha^i, G_\alpha^r\}$,

$$\Sigma_2 = \{G_\alpha^a, G_\alpha^z, G_\alpha^u, G_\alpha^e\}. \quad (1.1)$$

Каждый элемент класса простых и сложных типов данных определяется на множестве их значений и операций над ними:

$$G_\alpha^t = <X_\alpha^t, \Omega_\alpha^t>,$$

где $t = b, c, i, r, a, z, u, e$.

Операциям преобразования каждого t типа данных соответствует изоморфное отображение двух алгебраических систем с совместимыми типами данных двух

разных ЯП. В классе систем Σ_1 и Σ_2 преобразование типов данных $t \to q$ для пары языков l_t и l_q обладает такими свойствами отображений:

1) $G_\alpha^{\,t}$ и $G_\beta^{\,q}$ – изоморфны (q определенный на том множестве, что и t);

2) между $X_\alpha^{\,t}$ и $X_\beta^{\,q}$ существует изоморфизм, для которых множества $\Omega_\alpha^{\,t}$ и $\Omega_\beta^{\,q}$ разные. Если $\Omega = \Omega_\alpha^{\,t} \cap \Omega_\beta^{\,q}$ не пусто, то рассматриваем изоморфизм между $G_\alpha^{\,t'} = < X_\alpha^{\,t}, \Omega >$ и $G_\beta^{\,q'} = < X_\beta^{\,q}, \Omega >$. Такое преобразование сводится к случаю 1;

Между множествами $X_\alpha^{\,t}$ и $X_\beta^{\,q}$ может не существовать изоморфного соответствия. В этом случае необходимо построить такое отображение между $X_\alpha^{\,t}$ и $X_\beta^{\,q}$, чтобы оно было изоморфным. Если такое отображение существует (в каждом конкретном случае оно может быть разным), то имеем условие случая 1) с соответствующими изменениями в определении алгебраических систем;

3) мощности алгебраических систем должны быть равны $|G_\alpha^{\,t}| = |G_\beta^{\,q}|$.

Любое отображение 1, 2 сохраняет линейный порядок, так как алгебраические системы (1.1) линейно упорядочены.

Лемма 1. Для любого изоморфного отображения φ между алгебраическими системами $G_\alpha^{\,t}$ и $G_\beta^{\,q}$ выполняются равенства $\varphi(X_\alpha^{\,t}{}_{.\,\min}) = X_\beta^{\,q}{}_{.\,\min}$, $\varphi(X_\alpha^{\,t}{}_{.\,\max}) = X_\beta^{\,q}{}_{\max.}$

Доказательство леммы тривиальное и простое. При условии, когда одно или два вышеприведенных равенства не выполняются, тогда для основных множеств алгебраических систем изменяется линейный порядок, что противоречит определению.

Формальные условия преобразования типов данных $t = c, b, r, a, z$ определяются теоремами 1 – 5 [6, 7].

Теорема 1. Пусть φ – отображение алгебраической системы $G_\alpha^{\,c}$ в систему $G_\beta^{\,c}$. Для того, чтобы φ было изоморфизмом, необходимо и достаточно, чтобы φ изоморфно отображало $X_\alpha^{\,c}$ на $X_\beta^{\,c}$ с сохранением линейного порядка.

Необходимость. Пусть φ – изоморфизм. Тогда при отображении сохраняются все операции множества $\Omega = \Omega_\alpha^{\,c} = \Omega_\beta^{\,c}$, в том числе и операция отношения, которая определяет линейный порядок $X_\alpha^{\,c}$ и $X_\beta^{\,c.}$

Достаточность. Пусть φ изоморфно отображает $X_\alpha^{\,c}$ на $X_\beta^{\,c}$ с сохранением линейного порядка. Операция отношения выполняется соответственно принципу упорядоченности. Операцию *succ* докажем с помощью леммы, согласно которой выполняется равенство $\varphi(X_\alpha^{\,c.}{}_{.\,\min}) = X_\beta^{\,c}{}_{.\,\min}$.

Последовательно применяя операцию *succ* к этому равенству и учитывая линейную упорядоченность $X_\alpha^{\,c}$ и $X^c\beta$ ($x < succ(x)$), получаем, что для любого $x^c\alpha \in X^c\alpha$ и $x^c\alpha \neq X_\alpha^{\,c}{}_{\min}$ из равенства $\varphi(X_\alpha^{\,c}) = x^c\beta$, где $x^c\beta \in X^c\beta$, выполняется равенство

$$\varphi(succ(x^c\alpha)) = succ(x^c\beta). \qquad (1.2)$$

Операция pred доказывается аналогично с помощью $\varphi(X_\alpha^{\,c.}{}_{\max}) = X_\beta^{\,c}{}_{\max}$.

Теорема 2. Любой изоморфизм φ между алгебраическими системами $G_\alpha^{\,b}$ и $G_\beta^{\,b}$ является тождественным изоморфизмом:

$$\varphi(X_{\alpha.\,\text{false}}^{\,b}) = X_{\beta.\,\text{false}}^{\,b},$$
$$\varphi(X_{\alpha.\,\text{true}}^{\,b}) = X_{\beta.\,\text{true}}^{\,b}. \qquad (1.3)$$

Доказательство. При отображении $G_\alpha^{\,b}$ и $G_\beta^{\,b}$ всегда справедливо $X_{\alpha.\,\text{false}}^{\,b} < X_{\beta.\,\text{true}}^{\,b}$. Поэтому, учитывая сохранение линейного порядка, единственно возможным изоморфизмом является (3).

Теорема 3. Любой изоморфизм между алгебраическими системами с соответствующими числовыми типами является тождественным автоморфизмом.

Доказательство этой теоремы тривиальное и является следствием свойств элементов числовых множеств.

Теорема 4. Пусть $G_\alpha{}^a$ и $G_\beta{}^a$ – алгебраические системы, которые отвечают типам данных массива (a); φ_i и φ_v – изоморфные отображения множеств индексов (i) и значений элементов (Y) массивов, которые сохраняют линейный порядок. Тогда изоморфизм φ между алгебраическими системами целиком определяется изоморфными отображениями:

$$\varphi_i : X_\alpha{}^a \to X_\beta{}^a,$$
$$\varphi_v : Y(X_\alpha{}^a) \to Y(X_\beta{}^a).$$

Изоморфизм φ между алгебраическими системами $G_\alpha{}^a$ и $G_\beta{}^a$ определяется отображениями φ_i и φ_v, которые сохраняют линейный порядок и упорядоченность элементов массива.

Теорема 5. Пусть $G_\alpha{}^z$ и $G_\beta{}^z$ – две алгебраические системы, которые отвечают типам данных "запись" или структура и $x_\alpha{}^z \in X_\alpha{}^z$, $x_\beta{}^z \in X_\beta{}^z$. Тогда, если между последовательностями компонентов записей $x_\alpha{}^z$ и $x_\beta{}^z$ существует взаимно однозначное соответствие, то изоморфизм φ между $G_\alpha{}^z$ и $G_\beta{}^z$ определяется изоморфными отображениями алгебраических систем, которым соответствуют компоненты записи или структуры.

Преобразования между массивами и записями сводятся к преобразованию простых типов данных их элементов. Преобразования между действительными типами и другими числовыми значениями предполагают использование эмпирических случаев, так как отсутствует изоморфизм основных множеств этих алгебраических систем.

При преобразовании простых и структурных типов используются операции селектора S и конструирования C для изменения уровня структурирования данных. Операция селектора S для массива определяется в виде ограничения отображения:

$$M : I \to Y \text{ на } I', E \text{ M}\in M : I \to Y,$$

где E – вложения $I' \in I$. Тогда $M \mid \{k\}$ соответствует k - элемент массива при $I' = \{k\}$. Аналогично эта операция определяется и для записи $M \mid \{S_{vm}\}$, где M – отображение между селекторами компонентов и самими компонентами, а S_{vm} определяет соответствующий компонент записи.

Операция конструирования C массива состоит в формальном приведении в порядок компонентов и определении соответствия между множеством индексов и множеством элементов массива. Аналогично эта операция определяется для записи.

Таким образом, множество операций P, S и C определяет элементарные правила для конструирования сложных типов данных из более простых для взаимодействующих компонентов на разных МП.

Модель сопряжения модулей

Под сборочным *сопряжением* двух модулей понимается процесс преобразования их общих данных к форме, согласующейся с представлением каждого из

них. Модель сопряжения – это совокупность формальных описаний общих данных двух модулей и функций их преобразования к релевантному виду.

Процесс сборки разнородных объектов – это создание сложного программного объекта путем объединения готовых более простых программных элементов.

Пусть $P = \{p^i\}_{i=1,s}$ – множество компонентов, входящих в состав сложных систем. С каждым p^i связано множество данных D^i, с помощью которых осуществляется взаимодействие путем обмена между компонентами.

Множество $D^i = \{d^i_j\}_{j=1,t^i}$ состоит из переменных d^i_j, каждая из которых характеризуется тройкой: именем (идентификатором переменной) N^i_j, типом T^i_j и текущим значением V^i_j).

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

Введем следующие обозначения: $N^i = \{ N^i_j \}_{j=1,t^i}$, $T^i = \{T^i_j\}_{j=1,t^i}$, $V^i = \{V^i_j\}_{j=1,t^i}$. 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 $\tilde{V}^i$. Имеют место следующие два случая.

1. Каждой переменной $d^i_j \in \tilde{D}^i$ соответствует только одна переменная $d^i_j \in \tilde{D}^k$. Тогда преобразование $F^{ik}: \tilde{D}^i \rightarrow \tilde{D}^k$ состоит из множества преобразований для отдельных переменных: $F^{ik}=\{F^{ik}_{ji}\}$. Вводя обозначения $FN^{ik} = \{FN^{ik}\}$, $FT^{ik} = \{FT^{ik}_{ji}\}$, $FV^{ik} = \{FV^{ik}_{ji}\}$, определяем преобразования:

$$FN^{ik} : \tilde{N}^i \rightarrow \tilde{N}^k$$
$$FT^{ik} : \tilde{T}^i \rightarrow \tilde{T}^k$$
$$FV^{ik} : \tilde{V}^i \rightarrow \tilde{V}^k$$

соответственно для множеств идентификаторов, типов данных и значений.

2. Между переменными d^i_j и d^k_j 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 $\tilde{d}^k_{j1, \ldots\ldots}\tilde{d}^k_{jr}$, а S – функция селектора, снижающую уровень структурирования данных: $S(\tilde{d}^i_{j1}) = (\tilde{d}^i_{j1, \ldots\ldots}\tilde{d}^i_{jr})$, 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 $\tilde{\tilde{D}}^i$. Построение отображения $F^{ik}: \tilde{\tilde{D}}^i \rightarrow \tilde{D}^k$ производится аналогично случая 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 $C(\tilde{d}^i_{j1, \ldots\ldots}\tilde{d}^i_{jr}) = \tilde{d}^i_j$, 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, это 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 $T_j^i = (X_j^i, \Omega_i^i)$ в тип $T_i^k = (X_i^k, \Omega_j^k)$ 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 изоморфное отображение между двумя алгебраическими системами.

При практической реализации модель сопряжения – это совокупность моделей для пар программных компонентов $P = \{P^{ik}\}$ в создаваемой программе.

Преобразования типов данных при сборке систем

При сборке сложных систем возникают условия преобразования, которые включают в себя допущения, противоречащие условиям формальной сборки из-за нарушения свойств модулей с учетом рассмотренного изоморфизма алгебраических систем и т. д. Условия сборки приведены ниже.

1. Типы данных, описанные выше, являются наиболее общими. При разработке модулей часто приходиться 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 не могут использоваться. 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 характеризуется множеством Ω^i, включающим в себя все возможные операции для целого типа. Исключение из этого множества некоторых операций дает нам некоторую другую 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 различиями в описаниях модулей.

2. 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 мультиотображениями. 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 преобразованию, данных, при которых отсутствует изоморфное соответствие между основными множествами алгебраических систем, описывающих преобразуемые типы. Анализ подобных преобразований должен проводиться в частном порядке.

4. Среди формальных преобразований не рассмотрен случай, когда параметр содержит строку символов из последовательности цифр, и он должен преобразовываться в целый тип 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 межъязычного 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 межъязычного интерфейса. Однако рассматриваемый подход с соответствующими дополнениями может использоваться для построения любых интерфейсов, предназначенных для комплексирования модулей.

Формальный подход к построению межъязычных интерфейсов, потребовал создания функций преобразования, которые позволяют упорядочить разработку процессов реализации сложных систем из модулей с использованием модулей посредников.

Процессы практической реализации сборки разнородных модулей

Выделено два основных процесса:

1) разработка множества интерфейсных функций преобразования типов данных при обмене ими разнородных объектов;

2) реализация сопряжения каждой пары модулей с помощью интерфейсных модулей-посредников, преобразующих данные, передаваемые между парой сопрягаемых объектов.

Данные процессы в значительной мере независимы. Поэтому они реализуются 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 преобразования является связующим звеном для создания сложных систем из готовых программных объектов.

1. Формальный подход к интерфейсным функциям позволяет в процессе проектирования выделить множество допустимых операций, исключая операции, которые не могут быть реализованы или не относятся к функциям межъязычного интерфейса.

2. 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 в процессе сопряжения можно определять допустимость преобразования типов для соответствующих параметров.

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

а) несоответствия числа параметров в списках фактических и формальных параметров;

б) несогласованности типов передаваемых параметров в вызывающем и вызываемом модулях;

в) несоответствия во множествах значений типов фактических и формальных параметров;

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

д) новые операции, которые не включены в описания существующих типов данных ;

е) различий в уровнях структурирования фактических и формальных параметров;

ж) неверное описании типов данных при передачи параметров между вызывающими и вызываемыми модулями;

з) отсутствие обратных преобразований типов данных после работы вызываемого модуля.

Все эти классы ошибок выделены на основании приведенного анализа создания сложных систем.

4. Реализация множества интерфейсных функций и алгоритмов сборки позволяет автоматизировать процесс сопряжения модулей. Такой процесс автоматизированного сопряжения реализован в системе АПРОП, рассмотренной ниже.

Приведем 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 каждым компонентом 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 межъязычного 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 процесса построения межъязычного интерфейса сложных систем из разнородных модулей, которые обмениваются между собой разными типами данных. Отличающиеся переданные типы данных преобразуются к соответствующим типам и форматам.

Пример сборки разнородных модулей в системе АПРОП

Основной механизм системы АПРОП – библиотека интерфейса была пополнена новыми функциями преобразования типов данных для двух ЯП – Дельфи и Паскаль. В [7] приведено описание типов данных этих ЯП и реализация имеющихся различий. Согласно данной концепции преобразование различающихся типов данных в языках Delphi и Pascal провели студенты 4 курса (С. Балаклеенко и Л. Зинченко) кафедры информационных технологий Киевского национального университета им. Тараса Шевченко на примере программы факториал числа в табл. 2.1.

Таблица 2.1. **Схема сборки программ P1 и P2**

Программа P1 в языке Pascal	**Программа Unit1**	**Программа P2 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(TForm1, Form1);
		Application.Run;
		end.

На этой программе показана схема сборки модулей P1 и P2 с использованием библиотечных функций преобразования в программе факториал числа, алгоритм которой описан в *модуле P1* в языке Delphi и в *модуле P2* – в языке Pascal и модуль-посредник *Unit1* для преобразования неэквивалентных типов данных этих двух языков взаимосвязи.

Модуль-посредник (*Unit1*) содержит сгенерированные операторы обращения к программе в языке Pascal из библиотеки функций преобразования P1 через *Unit1* После выполнения модуля P2 в языке Delphi через модуль посредник *Unit1,* который проводит обратные преобразования данных, полученных результатов после возврата из модуля P1.

1.3. Фундаментальные типы данных (ТД). Простые и сложные ТД

Тип данных – это фундаментальное понятие в теории программирования. Он определяет множество значений и операций, которые применяются к этим значениям. Данные, которыми оперируют современные программные элементы, относятся к определенным ТД. Типизацию данных в комбинаторной логике и в языках, основанных на теории лямбда-исчисления, исследовал Р. Хиндли (1960). Затем, в 80-х прошлого столетия он предложил полиморфную систему типов, в которой полиморфный тип – это представление набора типов как единственного типа. Тип (сорт) – это независимая совокупность элементов, выделенная для предметной области. Математически тип задается:

1) множеством всех значений, принадлежащих типу;

2) предикатной функцией, определяющей принадлежность элемента к данному типу.

В математике тип "целое число" не имеет ограничений, а в программировании ограничен диапазоном значений и объемом занимаемой памяти в ЭВМ.

ЯП по способу определения ТД разделяются на:

1) языки с полиморфными ТД (например, Vbasic - тип вариант, Prolog, Lisp - списки). Переменные принимают значение любого типа и возвращают обратно те же значения. При выполнении *а + в* трактуется как сложение чисел, если они оба имеют числовые значения, или как конкатенация символов, если они строковые и если типы несовместимы, то они недопустимы. Это называется динамической типизацией, а в ООП и в теории чисел – полиморфным типом.

2) языки с неявным определением типа (например, в языке Basic различают строковые типы с добавлением 8 и массивы с добавлением скобок).

3) языки с типом, задаваемым пользователем. Их применяют в любых выражениях, а транслятор обеспечивает необходимое преобразование. В языке Ада ТД строго типизированы, и каждая операция требует описания типов.

Классификация фундаментальных типов данных

Для анализа основных фундаментальных ТД (FDT) используется теория структурной организации данных [7, 70 – 75]. FDT базируется на аксиоматике ТД и правилах выполнения операций над ними. Система аксиом определяет структуру множества значений типа, принадлежность его отдельным элементам, их свойства и отношения с другими ТД.

Для каждой операции, выполняемой над типизированными переменными, определяются типы операндов и результатов. Среди существующих ЯП данная теория воплощена в языках Паскале, Модула-2, Ада и др.

В FDT существуют базовые ТД: целый (integer), вещественный (real), булев (boolean), символьный (character) и др. Они характеризуются тем, что практически во всех ЯП и на уровне архитектуры ЭВМ имеются аппаратные средства для их представления и обработки. Структурные ТД – производные от них и образуются путем генерации или конструирования их с помощью базовых типов. Все FDT делятся на простые, структурные и сложные. К *простым* относятся перечислимые и числовые, к *структурным* – массивы, записи, множества, списки, последовательности и т. д. Перечислимые ТД рассматриваются на основе булева и символьного типов; числовые на основе целого и вещественного типов; массивы и записи как объекты структурных типов.

Простые типы данных. К простым типам относятся перечислимые и числовые. Общее обозначение перечислимого типа имеет вид type $T = (x_1, x_2, ... x_n)$, где T – имя типа, $x_1, x_2, ..., x_n$ – имена значений типа T во множестве значений X. Операции над перечислимыми типами включают бинарные операции 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 значений X.

Перечислимые типы – булевы и символьные. Значение булева типа – false и true. Множество операций Ω, кроме перечисленных выше, включают в себя операции булевой алгебры &, V, ¬. Алгебраическая система булева типа имеет вид

$$\Sigma^b = <X^b, \Omega^b>,$$
$$X^b = \{\text{false, true}\},$$
$$\Omega = \{\&, \text{V}, \neg, \text{pred, succ}, \leq\},$$
$$T^b = (\text{false, true}).$$

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

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 для символьного типа имеет вид

$$\Sigma^c = <X^c, \Omega^c>,$$
$$X^c = \{... , 'A' ... , 'X' \ldots , '0', '1', . .. , '9'\},$$
$$\Omega^c = \{\text{pred, succ}, \leq\},$$
$$T^c = (..., \text{A}, ..., \text{X} ...., 0, ..., 9).$$

Алгебраическая система Σ^c имеет тип <1,1; 2> согласно арности соответствующих операций и предикатов. Описание символьного типа char в модулях может быть опущено, если он стандартный. Операция ord присваивает каждому символу порядковый номер X^c, и succ определяет по порядковому номеру его значение.

Для перечисленных типов характерны следующие аксиомы:

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

При практическом использовании числовых типов имеются ограничения, определяемые архитектурой ЭВМ (конечное значение количества разрядов слова памяти для представления чисел) или явным описанием в модулях (для ограничения диапазона значений отдельных элементов). Числовые типы рассматриваются как отрезки (X.min,…, X.max). Для любого $x \in X$ выполняется условие $x.\min < x < x.\max$. Для стандартных числовых типов (integer и real) приведенное описание может быть опущено. Элементы X.min и X.max не определяются и зависят от реализации транслятора с ЯП на ЭВМ.

Над переменными целого типа и типов, связанных с ним, выполняются те же операции, что и в перечисленных типах. Добавляются операции целочисленной арифметики: унарный минус, +, –, ×, div (целочисленное деление) и mod (получение остатка от деления). Алгебраическая система, соответствующая целому типу, имеет вид

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

Во множестве Ω^i операция " – " соответствует унарному минусу. Остальные операции выражаются через операции Ω. Алгебраическая система имеет тип (2, 2, 2, 1; 2) согласно арности операций и предикатов.

Над переменными вещественного типа и связанных с ним типов, выполняются операции отношения и арифметические операции для действительных чисел (унарный минус, +, –, ×, /). Алгебраическую систему Σ^r, соответствующую вещественному типу, имеет вид

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

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

Порядок выполнения операций над любыми типами следующий:

1) все операнды приводятся к базовому типу;
2) операция выполняется, как над объектами базового типа;
3) обратный перевод (от базового к исходному типу) для полученного результата.

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

Аксиомы для числовых типов следующие:

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

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 аксиома выполнения операций для числовых типов определяется двухместной арифметической операцией $\oplus$:

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

Структурные типы данных. Структурные ТД содержат набор упорядоченных элементов, обработка которых проводится как над отдельными объектами, так и на уровне отдельных элементов. Эти ТД строятся из базовых типов и различаются функциями конструирования и механизмами обработки. В качестве основных структурных типов в работе рассматриваются массивы и записи.

Массивы. Функция конструирования *массива* осуществляется с помощью базовых типов путем отображения множества индексов и значений:

$$M : I \rightarrow Y,$$

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

Над массивами выполняются операции:

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

2) 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);

3) умножение двухмерных массивов по правилам умножения матриц.

Операции сложения и вычитания выполняются только на числовых массивах. Поэтому они не входят в состав множества общих операций над массивами. Операция умножения накладывает ограничения на область значений индексов массивов, связанных с правилами умножения матриц. Алгебраическая система, соответствующая ТД *массив,* имеет следующий вид:

$$\Sigma^{\alpha} = < X^{\alpha}, \Omega^{\alpha} >,$$

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

$$I(x_2)) \& (Y(x_1) \cup Y(x_2) \subset \overline{Y}(X^{\alpha}))\}, \Omega^{\alpha} = \{ \leq \},$$

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

где $I(x)$ – множество индексов для массива x, $Y(x)$ – множество значений элементов массива x, $\overline{Y}(X^{\alpha})$ – множество значений элементов массива рассматриваемого типа, $T(I)$ – ТД индексов массива; $T(\overline{Y})$ – ТД множество значений элементов массивов типа T^{a}. К данному типу принадлежат только те массивы, у которых множества индексов совпадают, а множество значений их элементов принадлежат одному и тому же множеству рассматриваемого типа. Для всех x и постоянное I отображение имеет вид

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

Множество Ω^a состоит только из одного предиката. Операции выполняются над массивами, как над единым структурным значением. Кроме того, над элементами множеств I и Y могут выполняться операции, соответствующие их ТД.

Многомерные массивы определяются рекурсивно. ТД $T(Y)$ в описании массива T^a могут быть массивами: type T^a = array $T(I^1)$ of $T(Y^1)$, type $T(Y^1)$ = array $T(I^2)$ of $T(Y^2)$. Эти описания типов эквивалентны описанию: type T^a = array $T\,(I^1 \times I^2)$ *of* $T(Y^2)$. В нем множество индексов массивов, принадлежащих типу T^a, представлено в виде прямого произведения множеств значений для типов $T(I^1)$ и $T(I^2)$.

Запись. Структурный тип *запись,* как и *массив,* состоит из нескольких компонентов, которые могут быть разнородными, т. е. принадлежат простым или структурным типам. Функция конструирования записей представляет конкатенацию отдельных компонентов. Множество значений типа *запись* – прямое произведение множества значений ее компонентов. Над записями выполняются только операции отношения.

Пусть запись состоит из n компонентов. Каждый m - компонент (m = 1, 2, ... , n) имеет тип T^{vm} и ей соответствует алгебраическая система $\Sigma^{vm} = <X^{vm}, \Omega^{vm}>$. Индекс v_m – один из индексов ТД.

Алгебраическая система для *записи* имеет вид:

$$\Sigma^z = <X^z, \Omega^z>,\ \Omega^z = \{ \le \}$$

$$X^z = \{ x \mid (x = x^{v1} \times \ldots \times x^{vn}) \,\&\, (x^{v1} \in X^{v1}) \,\&\, \ldots \,\&\, (x^{vn} \in X^{vn}) \},$$

$$T^z = (S_{v1}: T^{v1}; \ldots S_{vn}: T^{vn}),$$

где $S_{v_1}, \ldots, S_{v_n}$ – селекторы, а $T^{v_1}, \ldots, T^{v_n}$ – ТД для компонентов записи.

Сложные типы данных. К сложным ТД относятся: множества, объединения, динамические объекты, списки, последовательности, стеки, деревья и др. Некоторые из этих типов – стандартные в конкретных ЯП, а другие реализуются путем моделирования соответствующих структур и операций над ними. При этом некоторые ТД имеются в ЯП и могут быть сведены к базовым типам, а другие ТД отсутствуют в этих ЯП.

Множества. В ЯП Паскаль реализован ТД множество. Общая форма записи типа данных *множество* следующая: type T = powerset T^0, где T – тип множества, перечислимый или целый тип; T^0 – базовый тип для элементов множества.

Для типа T реализованы все основные операции как над математическими объектами – объединение, пересечение, разность, включение и др. Операции селектора – это 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 *объединение* имеет вид type T = union ($T^{v_1}, \ldots, T^{v_n}$), где T – тип объединения; $T^{v_1}, \ldots, T^{v_n}$ – базовые типы.

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 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 ТД в различных вариантах реализован в ЯП: ПАСКАЛЬ, АДА, ПЛ/1, СИ и др. Общая форма записи для этого типа имеет вид: type T = pointer to T^0, где T – определяет ссылочный тип; T^0 – базовый тип.

Объект типа T представляет собой 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. Использование ссылочного типа отличается от использования простых типов. Операции над ссылочными типами не формализованы. 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 арифметики.

Списки. Списки – это конструкции Лисп, они могут реализовываться программным моделированием. Элемент списка описывается как запись, содержащая одну или несколько компонентов ссылочного типа, которые обеспечивают связь между элементами списка. К ним могут быть применены операции, аналогичные операциям над фиксированными записями. Существуют операции, применяемые к целому списку: выбор начального элемента, получение остатка списка, соединение списков, сравнение, конвертирование, поиск элементов в списке и др. Для языков, не имеющих стандартных средств обработки списков, операции над списками реализуются отдельными процедурами.

Последовательности. Общая форма имеет вид type T = sequence T^0, где T - тип последовательности; T^0 – базовый тип.

Последовательность – это один из вариантов списка, у которого каждый элемент содержит только одну ссылочную переменную. Операции над последовательностями аналогичны операциям над списками. Одной из разновидностей последовательности является строка, каждый элемент которой содержит компонент символьного типа. Обработка символьных строк допускается в языке Снобол.

Стеки. Стек – это специально организованная память с дисциплиной обработки LIFO (последним прошел – первым обработан). Отдельный элемент стека принадлежит простому типу. Используя средства программного моделирования, можно реализовать обработку стеков, содержащих элементы любых типов. На практике стеки реализуются в виде массивов или списков. Множество операций над стеками фиксировано и включает в себя: инициализацию стека, занесение элемента в стек, выбор из стека, анализ элемента, находящегося на вершине стека. Операции над стеками могут быть реализованы на аппаратном уровне стандартными средствами ЯП или программным моделированием.

Деревья. Деревья – это списковые структуры для представления графов или других аналогичных объектов. Множество операций над деревьями аналогично множеству операций над списками. Реализация этих операций зависит от конкретных приложений.

1.4. Общие типы данных. Неструктурные и генерированные ТД

В связи с развитием технологии программирования в традиционных ЯП и появлением новых объектных ЯП фундаментальные типы данных получили развитие в виде общих типов данных, представленных в стандарте GDT (ISO/IEC 11404 General Data Type, 2007) [75 – 77]. В нем даны следующие понятия ТД:

1) концептуальное или абстрактное понятие, которое характеризует ТД и его номинальное значение или свойство;

2) структурное понятие, которое определяет ТД согласно концепции интерфейса и стандартных сервисных средств;

3) реализационное понятие, которое определяется правилами представления ТД в этой среде.

Общие типы данных это:

1) независимые от языка ТД для формального описания концептуальных и фундаментальных типов данных как формализация метаданных для элементов данных, понятий элемента данных и значений областей;

2) ТД для текущих языков программирования C#, JAVA, Express и XML, языка интерфейса IDL, APL, SIDL, XML и др.;

3) полуструктурированные и неструктурированные совокупности данных, где ТД неизвестные или неопределенные предварительно путем поддержки типов данных, перспективных, устаревших и сохраненных особенностей, таких как элементы данных и допустимые значения.

Модель типов данных GDT является вычислительной абстрактной моделью**,** как средство манипулирования информацией в компьютерных системах. Это абстрактная модель, поскольку она оперирует с принятыми свойствами единиц информации для представления их в компьютерных системах.

Основные понятия GDT

Пространство значений – это совокупность (коллекция) значений типа данных, которая определяется одним из следующих способов:

1) перечислением;

2) аксиоматичным определением согласно основным положениям;

3) как подмножество уже определенного пространства значений, которое имеет тот же набор свойств;

4) как комбинация любых значений некоторого, уже определенного пространства значений посредством специфицированной процедуры конструирования новых значений.

Каждое отдельное значение принадлежит только одному типу данных, хотя оно может принадлежать и нескольким подтипам этого типа данных.

Равенство. Для каждого пространства значений существует понятие равенства (equality) по таким правилам-аксиомам.

Аксиома 1. Для любых двух значений (*a, b*) из пространства значений выполняется условие равенства *b*, специфицированое как $a=b$, или неравенство *b*, специфицированное как $a \neq b$;

Аксиома 2. Не существует пары таких значений (*a, b*) из пространства значений, для которых одновременно выполняются условия $a=b$ и $a \neq b$;

Аксиома 3. Для каждого значения *a* из пространства значений выполняется условие $a=a$;

Аксиома 4. Для любых двух элементов значений (*a*, *b*) из пространства значений $a=b$ тогда и только тогда, когда $b=a$;

Аксиома 5. Если для произвольных трех элементов значений (*a, b, c*) из пространства значений выполняются условия $a=b$ и $b=c$, то выполняется условие $a=c$.

Для каждого типа данных операция равенства *Equal* определяется как свойство равенства пространства значений. Для любых значений *a* и *b* из пространства значений *Equal* (*a*, *b*) есть *true* (истина), если *a=b,* и *false* (ошибочность) в противном случае.

Порядок. Пространство значений упорядоченное, если для него установлено отношение порядка (order), которое отражается как меньше или равно (≤) и удовлетворяет следующим правилам:

1) для каждой пары значений (*a, b*) из пространства значений выполняется условие $a \leq b$ или $b \leq a$ или оба этих условия;

2) для любых двух значений (*a*, *b*), если $a \leq b$ и $b \leq$, *a* то *a=b*;

3) для любых трех значений (*a, b, c*), если $a \leq b$ и $b \leq c$, то, $a \leq c$.

Запись $a < b$ используется для нотации следующих отношений: $a \leq b$.

Тип данных упорядоченный, если отношение порядка определяется на его пространстве значений. Тогда соответствующая операция, которая называется InOrder, определяется : для произвольных двух значений *a* и *b* из пространства значений InOrder(*a*, *b*) есть true, если $b \leq a$, и false в противном случае.

Ограниченность. Т*Д* ограниченный сверху, если он упорядоченный и существует такое значение U из его пространства значений, при котором для всех значений *s* этого пространства выполняется условие $s \leq U$. Значение *U* образует верхнюю границу пространства значений. Аналогично, ТД ограничен снизу, если он упорядоченный и существует такое значение *L* из его пространства значений, что для всех *s* этого пространства выполняется условие $L \leq s$. Значение *L* образует нижнюю границу пространства значений. ТД называется ограниченным, если его пространство значений имеет верхнюю и нижнюю границу.

Для каждого ограниченного снизу типа данных определяется операция нулевой арности Lower bound для построения значения, которое является нижней границею пространства значений. Для каждого ограниченного сверху типу данных определяется операция нулевой арности Upper bound для получения верхней границы этого пространства значений.

Кардинальность. Пространство значений основывается на математической концепции кардинальности (cardinality): и может быть конечным или бесконечным. ТД должен иметь кардинальность (мощность) своего пространства значений. Моделью предусмотрены важные категории ТД, пространство значений которых может быть: конечное; точное (exact) и бесконечное; приближеное, имеющее конечную или бесконечную модель, концептуальное пространство значений которой может быть бесконечным.

Каждый концептуальный ТД обязательно точный. Невычисляемый ТД является бесконечным.

Точный и приближенный Т*Д*. Модель типов данных устанавливает границу различных значений типов данных. Если каждое значение в пространстве значений концептуального типа данных можно отличить от другого значения в пространстве этой модели, то ТД считается точным (exact).

Математические ТД, которые имеют значения, которые не имеют определенного представления, называются **приближенными** (approximate) и формиру-

ются следующим образом. Пусть M – математический ТД, а С – соответствующий вычисляемый ТД, P – преобразует пространство значений M в пространство значений C. Тогда для каждого значения v' с C существует соответствующее значение типа данных v с M и такое действительное значение h, что $P(x)=v'$ для всех x с M и $|v\text{-}x|<h$. Таким образом, v' – это приближение в C для всех значений с M, которые находятся в h-области значений v''. Кроме того, по крайней мере для одного значения v' с C существует более, чем одно такое значение $в$ с M, что $P(y) = v'$. Таким образом, C – это неточная модель M.

Приближенные ТД имеют вычислительные модели, которые через параметрические значения определяют степень приближения**,** т.е. определенный минимальный набор значений математического типа данных для обеспечения различий в вычисляемом типе данных.

Числовой. ТД называется числовым (numeric), если концептуально его значения определяются количественно (в системе нумерации). ТД, значение которого не имеет этого свойства, называется нечисловым (non numeric).

Примитивные типы данных

Логический (boolean) – это математический ТД, связанный с использованием двузначной логики.

Состояние (state) – это ТД семьи, каждый из которых имеет законченное число разных, неупорядоченных ситуационных значений.

Перечисленный (enumerated) – это семья типов данных, каждый из которых допускает законченное число разных значений со свойственным им порядком.

Символьный (character) – это тип данных, пространство значений которого есть набор символов.

Порядковый (ordinal) – это ТД порядковых номеров, который отличается от значимых чисел (ТД, целый (integer)).

Целый (integer) – математический ТД, который описывает только целые числа.

Рациональный (rational) – это математический ТД, который соответствует рациональным (действительным) числам.

Масштабированный (scaled) – это тип данных, пространство значений которого – подмножество пространства рациональных чисел и каждый отдельный ТД имеет фиксированный знаменатель; как ТД с аппроксимацией значения.

Действительный (real) – это ТД, которые являются вычислительными аппроксимациями относительно отношения к математическому типу данных, соответствующему вещественному числу.

Комплексный (complex) – это ТД семейства, каждый из которых задает числовую аппроксимацию математического типа данных, который задает комплексные числа. Каждый ТД составляет коллекцию математических комплексных величин, которые при применениях известны с некоторой конечной точностью и при этом должны различаться с этой же точностью.

Сгенерированные типы данных

Сгенерированные ТД (generated datatypes) – это типы данных, полученные в результате применения генератора типов данных. **Генератор типов данных** – это концептуальная операция на одном или нескольких типах данных, которая создает новый ТД и оперирует типами данных для создания более нового типа данных, а не значениями для генерации значений. ТД, с которыми работает генератор, имеют название **параметрический** или **компонентный ТД**. Сгенерированный ТД семантически зависит от параметрических типов данных, но имеет собственные характеристические операции. Важной характеристикой всех генераторов типов данных является то, что генератор может применяться ко многим параметрическим ТД. Генераторы указателя и процедуры генерируют ТД, значения которых атомарные, тогда как генератор выбора и агрегатных типов данных генерирует ТД, значения которых позволяют производить их декомпозицию.

Выбор (choice) генерирует ТД. Каждое значения образуется из любого набора альтернативных типов данных. Этот ТД логически учитывает их соответствие значению другого типа данных с признаком (tag).

Указатель (pointer) генерирует ТД каждое значение которого устанавливает средства ссылки на значение другого типа данных, специфицированного типом данных *element-type*. Эти значения типа данных указателя являются атомарными.

Процедура (procedure) генерирует ТД, каждое значением которого является значением других типов данных, которые называют **параметр**. Такой ТД включает в себя набор всех операций над значениями конкретной коллекции типов данных, концептуально атомарных.

Запись (record) генерирует ТД, значения которого составляют совокупность значений компонентов типов данных и каждая совокупность имеет значение для каждого компонента типа данных, специфицированного фиксированным идентификатором поля field-identifier.

Набор (set) генерирует ТД из пространства значений из поднаборов пространства значений типа данных элемент с операциями, свойственными математическому множеству *set.*

Портфель (bag) генерирует ТД, значения которого составляют коллекции образцов значений типа данных элемент. Многочисленные образцы того же значения могут подаваться в этой коллекции и порядок их в коллекции несущественный.

Последовательность (sequence) генерирует ТД, значениями которого являются упорядоченные последовательности значений типов данных из значений, несвойственных этому типу данных; одно и тоже значение может встречаться многократно в этой последовательности.

Массив (array) генерирует ТД, значения которого ассоциируются с произведением пространств одного или нескольких конечных типов данных, которые называются индексными **ТД**. Пространство значений этого типа данных, такое, что каждому значению из пространства индексного типа данных соответствует только одно значение элемента .

Таблица (table) генерирует ТД, значения которого составляют коллекции значений из пространства одного или нескольких типов данных поле, такое что каждое значение из пространства задает ассоциации между значениями его полей.

Объявленный ТД (defined) – это ТД, определенный посредством объявления типа type-declaration.

Type-identifier – идентификатор типа некоторого объявления типа и ссылается на ТД или генератор типов данных, определенный таким образом. Actual-type-parameters, если он существует, соответствует номеру и типу объявления type-declaration. Таким образом, каждый актуальный параметр отвечает формальным в соответствующей позиции. Если formal-parameter-type составляет type-specifier, то actual-type-parameters будет value-expression, определяя значение типа данных, специфицированных, как formal-parameter-type. Если formal-parameter-type есть "type", то actual-type-parameter это type-specifier и имеет свойства этого параметрического типа данных в объявлении генератора.

Type-declaration идентифицирует type-identifier в type-reference с одним типом данных, семейством типов данных или генератором типов данных. Если идентификатор типа type-identifier задает семью типов данных, то type-reference ссылается на тот член семьи, пространство значений которого определяется посредством type-definition после замены каждого значения actual-type-parameters для всех входов formal-parametric-value. Если type-identifier задает генератор типов данных, то type-reference означает ТД, который получается применениям генератора типов данных к реальным параметрическим типам данных. Во всех случаях объявленный ТД имеет прямо или косвенно заданные значения, свойства и характеристические операции посредством объявления type-declaration.

Характеристические операции. Набор таких операций охватывает операции, которые создают значение любого типа с помощью генератора ТД, который создает пространство значений из параметрических ТД. Такие операции необходимы для выделения ТД по их названиям и генерации агрегатных ТД как композиции следующих операций:

1) с нулевой арностью для генерации значений этого типа данных;
2) с унарной операциею (арности 1), которая превращают значение этого ТД в новое значение этого же ТД или в значение boolean;
3) с арностью 2, которые преобразуют пары значений этого ТД в значение этого же ТД или в значение boolean;
4) с n-арностью, которые преобразуют упорядоченные n-елементные группы значений, каждая из которых относится к определенному ТД, который может быть параметрическим со значением этого же ТД или агрегатным ТД.

Не существует уникальной коллекции характеристических операций для заданного ТД. Одна коллекция операций для ТД (или генератора типов), достаточная для выделения этого ТД среди других из пространства значений той же мощности.

Таким образом, существует посимвольная замена, которая преобразует все пространство значений одного ТД (domain) во подмножество значений пространства другого ТД (диапазон, range**)** так, чтобы значение отношений и характеристических операций домену сохранялись в соответствующих значениях отношений и характеристических операций диапазона ТД.

Агрегатный ТД (aggregate datatype) – это сгенерированный ТД, каждое значение которого получено из значений параметрических ТД. Параметрические ТД агрегатного ТД или его генератор включают в себя имена компонентов ТД. Генератор агрегатного типа данных генерирует ТД с помощью алгоритмической процедуры пространства значений агрегатного ТД.

В отличие от других сгенерированных ТД агрегатный ТД обеспечивает доступ к компонентам значений через характеристические операции. Агрегатные значения разных типов различаются между собой свойствами, которые характеризуют отношение между компонентами ТД и отношение между каждым компонентом и агрегатным значением.

Генератор ТД (datatype generator) – это концептуальная операция над одним или несколькими ТД, которая создает новый ТД. Генератор ТД оперирует с ТД, а не с его значениями, представляет собой:

1) набор критериев для характеристик ТД, над которыми будут выполнены операции;

2) процедуры конструирования, которые допускают набор ТД с данным критерием для создания нового пространства значений из пространств значений этих ТД;

3) набор характеристических операций, которые применяются в конечном пространстве значений для завершения определения нового ТД.

Стандарт включает генераторы ТД: выбор (choice), указатель (pointer), процедура (procedure), запись (record), набор (set), портфель (bag), последовательность (sequence), массив (array), таблица (table) и т. п.

Сгенерированный ТД (generated datatypes) – это ТД, которые получаются в результате применения генератора ТД, как операции для создания нового типа из одного или нескольких ТД. Этот ТД зависит от семантически разных параметрических ТД и имеет собственные характеристические операции. Он является, агрегатным ТД, каждое значение которого получено из значений параметрических ТД и имеют имена компонентов ТД) Их значения различаются между собой свойствами и отношениями между каждым компонентом и агрегатным значением.

Подход к реализации GDT<=>FDT

С практической точки зрения общие ТД GDT можно генерировать к фундаментальным ТД с помощью специального набора процедур (функций), специфических для разных компьютерных систем. Нами предложена схема генерации GDT<=>FDT [44] (рис. 2.2).

Основные функции для генерации ТД GDT. Соответственно спроектированной нами схеме генерации необходимо разработать набор библиотек функций (процедур) в общепринятом языке JAVA, XML для применения их при отображении разных ТД в программах в современных или будущих ЯП.

Это такие функции:

1) преобразование типов данных $ЯП_1$, …, $ЯП_n$;

2) представление типов данных FDT;

3) представление GDT для обработки из апробированной схемы FDT;

4) отображение GDT<=>FDT.

Теория представления FDT была разработана нами и реализована в виде библиотеки функций преобразования между собой ТД для класса ЯП 4GL [4, 5].

Для реализации данного набора функций для FDT<=>GDT и GDT<=>FDT необходимо:

1) создать библиотеки функций для преобразования ТД GDT (примитивных, агрегатных и генерированных) к FDT ТД (простым, структурным и сложным) ЯП, подобно элементам среды взаимодействия разноязычных компонентов, подсистем и системы Grid;

2) специфицировать внешние ТД компонентов, подсистем и систем в ЯП средствами языка GDT с накоплением их в одном из репозитариєв среды разработки программных продуктов на некоторой фабрике программ;

3) разработать формат новых интерфейсных посредников типа stub с операциями обращения к соответствующим функциям GDT<=>FDT для организации передачи данных взаимодействующему компоненту и обратно.

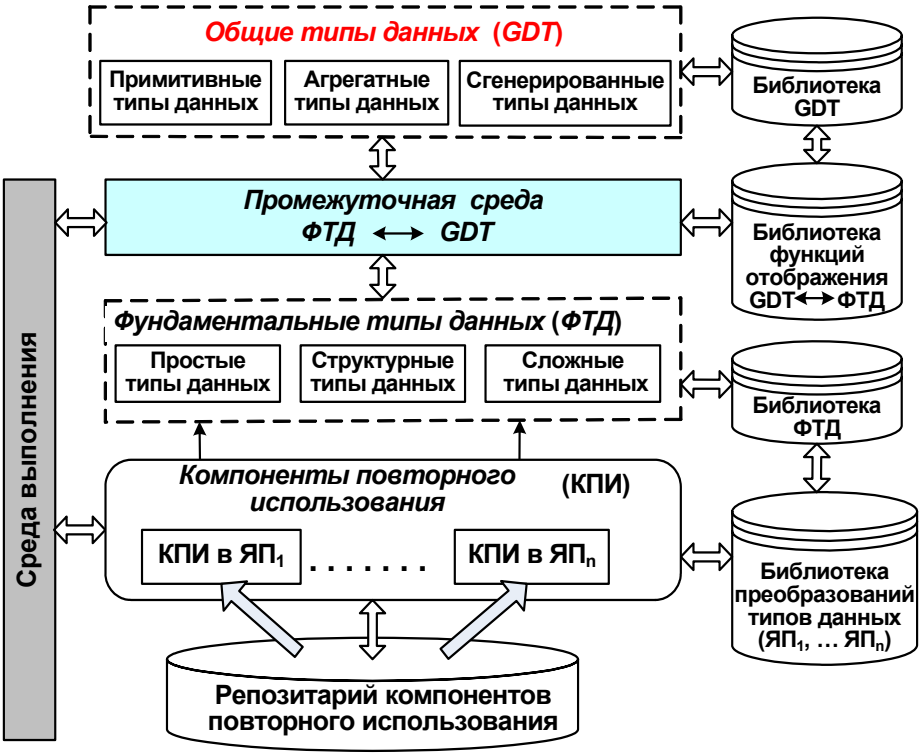


Рис. 2.2. Схема трансформации типов данных FDT <=> GDT

Таким образом, проблема сборки разнородных компонентов в новых ЯП с учетом архитектур платформ и современных сред преобразования ТД разноязычных программ частично решена в инструментально-технологическом комплексе ИТК для класса ТД (таблица, вектор, стек, очередь и др.). Это преобразование представлено линией на веб-сайте (http://sestudy.edu-ua.net) и реализовано в студенческой фабрике программ КНУ (http://programsfactory.univ.kiev.ua).

1.5. Стили сборочного программирования

Идея сборочного изготовления ПС и членов семейств ПС из готовых ресурсов и КПИ исследовалась и разрабатывалась в рамках ряда фундаментальных проектов ИПС НАН Украины, в частности в последние годы (2007–2012). В результате были отработаны не только модули, но и готовые программные элементы, как объекты, компоненты и сервисы.

Таким образом, был расширен набор операций алгебры преобразования готовых разнородных программных элементов в современных средах.

Сборочное программирование предполагает, что программа собирается из уже известных готовых элементов, включая отдельные фрагменты программ. Так решается задача многократного и быстрого их применения в процессе создания программы из заранее изготовленных "деталей".

Сборочное программирование обозначает:

1) динамическое расширение функциональности сложной системы за счет поиска, выбора и привлечения в сферу обработки новой сложной ПС готовых ресурсов;

2) повышение масштаба и улучшение возможностей коммуникации между отдельными элементами сложных ПС за счет стандартного интерфейсного механизма подключения ресурсов в систему;

3) повышение язычного уровня коммуникации через протоколы доступа к другим системам и конечным пользователям.

Сборка ПС из готовых программных и информационных ресурсов выполняется исходя из следующих принципов:

1) композиционность сложных систем из компонентов, интерфейсов и сервисов с их характеристиками и механизмами агрегации в более сложные структуры для выполнения в интегрированной среде;

2) компонентность создания ПС из готовых "деталей", базированная на реально существующих положениях инженерии продуктов, включая стандарты ЖЦ, языки описания КПИ и их интерфейсов (IDL, API, WSDL и др.), операции добавления, замены и уничтожения КПВ, а также унификации, стандартизации и классификации элементов сборки;

3) интероперабельность ресурсов и элементов для взаимодействия ресурсов между собой и функционирования их в разных гетерогенных средах;

4) вариантность ресурсов и компонентных ПС, ориентированная на замену некоторых незавершенных систем или добавления новых элементов в конфигурационную структуру ПС или СПС.

Данные принципы определяют технологию построения больших и сложных систем из готовых программных элементов.

Стили парадигм сборочного типа

К настоящему времени сформировались:

1) стили программирования с соответствующим принципам модульности и стандартом элементов, которые собираются в более сложную структуру;

2) метод повышения эффективности межмодульных интерфейсов при передачи данных на компьютеры с разными форматами данных;

3) базы программных КПИ (библиотеки, репозитории) для их идентификации, выбора и проверки пригодности применения исходя из стандартизованного описания элементов и их интерфейсов.

Готовые модули и КПИ стали "программными кирпичиками", из которых "строится" программа, как дом.

Программные элементы могут быть в исходном и бинарном видах. Методология сборки позволяет подключать к сборочному программированию новые

элементы, ресурсы, которые определяются в модульном, объектном, компонентном, генерирующем и сервисном программировании. Эти ресурсы могут разрабатываться стандартизировано в любых операционных средах IBM, MS, Microsystems, CORBA, Com, Oberon и др.

Модульное сборочное программирование. Этот подход был исторически первым и базировался на процедурах и функциях, разноязычных модулях и методологии императивного программирования в среде ЕС ЭВМ прототипа IBM-360.

Объектно-ориентированное сборочное программирование. Подход базируется на методологии объектно-ориентированного программирования и предполагает использование библиотек методов и классов (исходные коды или упаковки классов) в динамическую библиотеку. Существуют конкретные технологические подходы, поддерживающие это программирование, например CORBA.

Компонентное сборочное программирование. Основные идеи этого подхода – распространение классов в бинарном виде и предоставлении доступа к методам класса через строго определенные интерфейсы, которые позволяют снять проблему несовместимости компиляторов и обеспечивать смену версий классов без перекомпиляции. Существуют конкретные технологические подходы, поддерживающие компонентное сборочное программирование: COM (DCOM, COM+, .NET).

Аспектно-ориентированное сборочное программирование. Оно дополняет компонентное программирование концепцией аспекта для изменения варианта реализации критичных по эффективности процедур и программ. Это программирование заключается в сборке полнофункциональных приложений из многоаспектных компонентов, инкапсулирующих различные варианты реализации (безопасность, синхронизация, надежность и др.). Существуют конкретные технологические подходы, поддерживающие данное программирование

Сервисно-ориентированное сборочное программирование. Это новая концепция интеграции сервисов и обслуживания ПС. К сервисам относятся: общие системные сервисы для поддержки реализации и выполнения ПС (связь, управление, каталогизация и др.); объектные сервисы, которые поддерживают объекты и классы, операции их выполнения и др.; веб-сервисы Интернет для быстрого решения поставленных задач. Существуют конкретные системы, поддерживающие данное программирование.

Поддержка данных стилей программирования

Сборка первоначально представлена нами как способ объединения разноязычных объектов в ЯП и преобразования ТД с помощью теории спецификации и отображения типов и структур данных ЯП средствами алгебраической системы с операциями и функциями релевантного преобразования одних ТД в другие.

Сборка это:

1) один из методов программирования и подчиняется общим закономерностям и функциям; – одна из форм поддержки повторного использования готовых программных элементов, объектов, КПИ;

2) экономический и качественный способ за счет стандартизации КПИ и их интерфейсов. Этим он отличается от процессов синтеза, композиции и интеграции в других методах программирования.

Элементы сборки обладают свойствами (наследования, полиморфизма и инкапсуляции). Они включают в себя данные и операции (методы) для обеспечения связи между собою разных КПИ.

Для технологичности сборки все модули и другие объекты должны иметь паспорта, которые содержат данные, необходимые для информационной их связи в более сложной структуры и для выполнения в операционной среде.

Важное условие сборки – наличие большого количества разнообразных комплектующих КПИ, которые обеспечивают решение широкого спектра задач из разных предметных областей. Для их сборки задается схема сборки и операторы вызова (CALL, RPC, RMI и т. п.) в модуле, связанным отношением связи с другим модулем. В вызове задается список параметров и значений, которые при их передаче другому проверяются на соответствие ТД, исходя из аксиом и утверждений системы преобразования одних ТД к другим в классе ЯП. Результат отображения – сгенерированные интерфейсные модули-посредники для эквивалентных преобразований ТД в процессе выполнения.

Процесс сборки любых КПИ как готовых ПП на основе схемы сборки – это линия сборочного конвейера, в котором роль "деталей" выполняют КПИ разной степени сложности, а роль "стыковщика" – интерфейсы-посредники. Последние присутствуют во многих методах и стилях программирования. На фабрики программ разработчики работают с КПИ как с деталями и подбирают те из них, которые могут быть комплектующими, т. е. повторно используемыми.

Взаимодействие каждой пары объектов схемы зависит от использования данных, их ТД и значений, которые передаются через параметры, а также от наличия библиотек классов и функций преобразования ТД, таких например, как в системе MS.Net (библиотеки CRL, CTS, FCL, CIL и др.).

Структуры сложных систем для сборки

Фактически термин программные системы появился в 80-х годах XX ст. и используется для определения сложных систем из модулей или других самостоятельных программных элементов (процедур, функций, подпрограмм и др.) статического и динамического типа. Наиболее распространенные термины обозначения сложных систем в индустрии – ПС, семейство ПС, семейство продуктов, программные комплексы и др.

Программная система – совокупность отдельных программных ресурсов, которые реализуют взаимоувязанные функции некоторой предметной области в заданной среде выполнения.

Семейство программных систем – это совокупность ПС, которые определяются общим множеством понятий для членов семейства и множеством специальных понятий, которые присущи каждому отдельному члену СПС.

Семейство программных продуктов – это product family (семейство продуктов, СПП) или продуктовая линия (Product Lines) SEI (www.sei.com/product line). Эти термины определены в словаре ISO/IEC FDIS 24765:2009(E) – Systems and Software Engineering Vocabulary как "группа продуктов или услуг, которые имеют общее управляемое множество свойств, которые удовлетворяющих потребностям определенного вида деятельности".

В технологии программирования сложных систем из множества готовых базовых элементов сборка стала эффективным средством для обеспечения их эволюции, адаптивности и интероперабельности. Промышленные фирмы производства ПП развивают теорию и практику производства вариантов продуктов, которое получило название вариабельности и взаимодействия ПП (см. глава 4 данного раздела).

1.6. CASE-средства интеграции модулей и интерфейсов

С годами проблема связи разноязычных, разнородных (по коду и среде) программ обострилась в связи с быстрым изменением архитектуры компьютеров, появлением распределенных, клиент-серверных сред и т. п. Проявилась неоднородность ЯП в смысле как представления в них типов данных, так и платформ компьютеров, на которых реализованы соответствующие системы программирования, а также в различных способах передачи параметров между объектами в разных средах – маршаллинг данных через разные виды операторов удаленного вызова. Единого подхода к решению проблемы интерфейса не существовало. Стандарт ISO / IEC 11404–1996 определил подход к решению вопросов интерфейса всех видов ЯП с помощью универсального языка LI (Language Independent), независимого от ЯП. Однако до настоящего времени инструментальной его поддержки не существует. Пользователям разных ЯП приходится выбирать подходящую реализацию интерфейса из множества имеющихся в разных средах [10].

Отметим особенности сред, влияющих на реализацию интерфейса.

Вначале рассмотрим некоторые особенности систем программирования для ЯП:

1) разные двоичные представления результатов компиляторов для одного и того же ЯП, реализованные на разных архитектурах компьютеров;

2) двухнаправленность связей между ЯП и их зависимость от среды и платформы;

3) параметры вызовов объектов отображаются в операции методов;

4) связь с разными ЯП через ссылки на указатели в компиляторах;

5) связь модулей в ЯП осуществляется через интерфейсы каждой пары из множества языков (L_1, …, L_n) промежуточной среды.

Современные наиболее распространенные среды – CORBA, Com, JAVA, каждая по своему решает проблему связи разноязычных компонентов с помощью интерфейса.

Связь компонентов в среде CORBA

В системе CORBA механизм связи разнородных объектов напоминает проектные решения в системе АПРОП с помощью модуля-посредника (stub, skeleton). Модуль-посредник stub выполняет аналогичные функции, связанные с преобразованием типов данных клиентских компонентов в ТД серверных компонентов посредством [46].

1) отображения запросов клиента в операции языка IDL (Interface Definition Language), RMI (Remote Invocation Interface) или API (Application Program Interface);

2) преобразования операций IDL в конструкции ЯП и передачи их серверу средствами брокера ORB, реализующего stub в ТД клиента.

Так как ЯП (C++, JAVA, Smalltalk, Visual C++, Cobol, Ada-95) реализованы на разных платформах и в разных средах, и двоичное представление объектов зависит от конкретной аппаратной платформы, в системе CORBA реализован *общий механизм связи* разнородных готовых объектов – брокер ORB

В эту среду может входить модель COM, в которой ТД определяются статически, а конструирование сложных типов данных осуществляется для массивов и записей. В системе CORBA методы объектов используются в двоичном коде, т. е. допускается двоичная совместимость машинных кодов объектов, созданных в разных средах, а также в разных ЯП за счет отделения интерфейсов объектов от их 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. 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.

В среде клиент-сервера Corba реализуется два способа связи:

1) на уровне ЯП через интерфейсы прикладного программирования;

2) на уровне компиляторов IDL, генерирующих клиентские и серверные интерфейсные посредники – stub, skeleton.

Интерфейсы определяются в IDL или APL для объекта-клиента и объекта-сервера, имеют отдельную реализацию и доступны разноязычным программам. Интерфейсы включают в себя описание формальных и фактических параметров программ, их типов и порядок задания операций передачи параметров, результатов при их взаимодействии. Другими словами, такое описание есть не что иное, как спецификация интерфейсного посредника двух разноязычных программ, которые взаимодействуют друг с другом через механизм вызова, который реализован на разных процессах. В функции интерфейсного посредника (stub) клиента входит:

1) подготовка внешних параметров клиента для обращения к сервису сервера;

2) посылка параметров серверу и его запуск для получения результата или сведений об ошибках.

Общие функции интерфейсного посредника (skeleton) сервера:

1) получение сообщения от клиента, запуск удаленной процедуры, вычисление результата и подготовка (кодирование или перекодирование) данных в формате клиента;

2) возврат результата клиенту через параметры сообщения и уничтожение удаленной процедуры и др.

Таким образом, интерфейсные посредники задают связь между клиентом и сервером (*stub* – для клиента и *skeleton* – для сервера).

Современные средства JAVA

Кроме перечисленных подходов реализации интерфейса, имеются:

1) связь готовых, разнородных элементов с помощью интерфейса в IDL, в котором определены входные и выходные данные взаимодействующих элементов;

2) специальный программный интерфейс – JNI (JAVA Native Interface), допускающий обращение из JAVA-классов к функциям и библиотекам на других ЯП путем поиска прототипов обращений к функциям на C/C++, генерации заглавных файлов компилятором C/C++ и обращения из JAVA-классов к COM-компонентам;

3) технология Bridge2JAVA, по которой генерируется оболочка для COM-компонента в виде прокси-класса и обеспечивается необходимое преобразование данных для разных ЯП средствами стандартной библиотеки преобразований типов;

4) связь с помощью языка CLR (Common Language Runtime) платформы .Net для любых ЯП, в который транслируются объекты в ЯП (C#, Visual Basic, C++, Jscript) с использованием библиотеки стандартных классов и средств генерации в представление .Net-компонентов;

5) стандартное решение ISO/IEC 11404–1996 описания ТД независимо от ЯП с помощью языка LI (Language Independent) для разноязычных компонентов, содержащего все существующих ТД ЯП либо средства их конструирования. В языке LI описываются параметры вызова, как элементы интерфейса, они преобразуются в ТД конкретных ЯП специальными правилами и операциями агрегации, представленными в стандарте;

6) XDL-стандарт описания структур данных произвольной сложности и преобразования форматов данных, передаваемых с одной платформы компьютера на другую с помощью специальных процедур;

7) XML-стандарт обеспечения взаимосвязей с преобразованием типов данных ЯП к единому формату XML, понятному многим распределенным средам.

Решения по конкретному преобразованию данных этим не исчерпываются, они еще будут появляться при внедрении новых платформ компьютеров и сред.

Новое толкование интерфейса объектов дал известный специалист по информатике П. Вегнер [17], сформулировав парадигму перехода от алгоритмов вычислений к *взаимодействию объектов*. Суть этой парадигмы – вычисление и взаимодействие объектов – две ортогональные концепции. Взаимодействие – это некоторое действие (action), но не вычисление. Сообщение – не алгоритм, а действие, ответ на которое зависит от операций, влияющих на состояние разделенной памяти (shared state) локальной программы. Он считал операции интерфейса неалгоритмическими, а сообщение взаимодействием объектов в операционной среде.

Дальнейшее развитие идеи взаимодействия, основанной на действиях, – язык AL (Action language), разработан А. А. Летичевским и Гильбертом [18]. Этот язык определяет вызовы процедур (локальных или распределенных) и их развертку в новую программу в виде операторов действий. Программа из вызовов процедур, т. е. действий рассматривается как ограниченное множество конечных программ, взаимодействующих со средой, в которую они погружаются.

Несмотря на приведенные различные подходы к проблеме интерфейса разноязычных объектов, она по-прежнему остается острой.

Система АПРОП может рассматриваться как прототип для построения новой подобной системы в классе ЯП, новых компьютеров и сред. Новая особенность такой системы – наличие трех языков описания интерфейсов API, IDL и RMI для представления разных видов посредников и сред их выполнения.

Прикладным базисом взаимодействия программ может стать руководство И. Бея [16], где автор представил более 100 вручную составленных вариантов модулей-посредников в классе современных языков: C/C++, Visual C++, Visual Basic, Matlab, Smalltalk, Lava, LabView, Perl. Эти варианты практически проверены автором в современных средах функционирования.

Однако в этих вариантах не учтены проблемы, связанные с появлением новых современных архитектур компьютеров и сред. Нужно доработать варианты связей и реализовать новый проект современной системы автоматизации взаимодействия разноязычных программ.

Таким образом, концепция интерфейса модулей, представленная в парадигме сборочного программирования, внесла значительный вклад в развитие теории программирования сложных систем из программ, разработанных средствами современных ЯП, инженерных подходов к конструированию и управлению компонентами повторного использования (КПИ). Роль этой концепции возрастает благодаря накопленному огромному запасу КПИ, т. е. reuse-компонентов в разных предметных областях, использовать которые без определения интерфейса не представляется возможным.

CASE-инструменты конфигурационной сборки в Grid Etics

В системе Grid ETICS реализуются такие сложные структуры: проекты, пакеты, подсистемы, системы и модули данных, в которых задаются ТД простой и сложной структуры (рис. 2.3) [50].

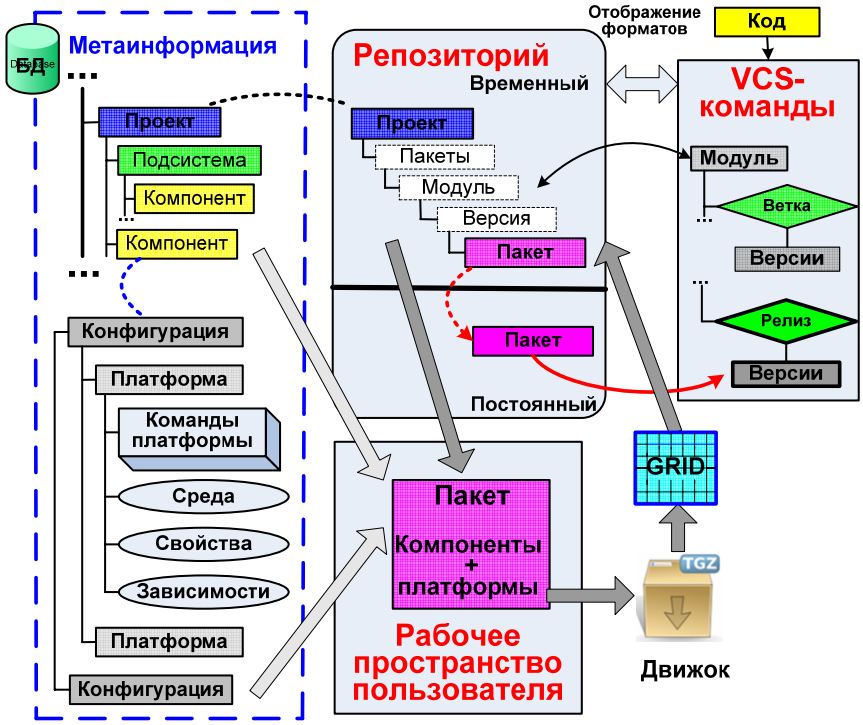


Рис. 2.3. Структура технологии проектов в системе Etics

Программные объекты в Etics разрабатываются по модели CIM (Common Information Model), предназначенной для описания сложных программ и систем объектной структуры. Сценарии работы системы задаются use case UML. В Etics содержится набор характеристик и процедур для построения и тестирования новых пакетов и систем. Этот набор расширяется через плагины. Каждый плагин включает в себя публичный интерфейс с описанием услуг.

В набор характеристик входит механизмы спецификации зависимостей между разными пакетами и их тестами. Функциональные плагины обепечивают проверку договоров, тестов для выполнения разных элементов систем, генерацию документации и ведения готовых объектов в оперативном или постоянном репозитории Etics.

Технология создания больших наборов пакетов из входных или комбинаций перекомпилированных двоичных элементов поддерживается процессом доступа к репозиториям для формирования распределенной версии. Скомпилированная структура автоматически находит и загружает из репозитория необходимые элементы в двоичном представлении. Проблема отображения элементов системы на альтернативную платформу решается путем построения перекрестных ссылок (cross platform) к необходимой платформе и генерации модулей, которые расставляют необходимые флажки в скомпилированный код (например, при переходе от 32-разрядных к 64-разрядным платформам) сетевой среды Grid.

В системе Etics стандартизовано описание ТД для главных объектов: *Проект, Подсистема и Компонент. Проект.* Подсистема может содержать только Компоненты. В системе предложена модель данных CIM для связей между разными объектами.

Модель данных, как и модель CIM, позволяет вводить формальные сущности в структуре проектов, описывать объекты и связи между ними, а также предоставлять результаты выполнения. Внутреннее сохранение данных базируется на модели данных реляционного типа, реализованной средствами MySQL.

Описание модели данных основано на следующих базовых положениях:

1) каждый компонент содержит описание сведений (имя, лицензия, URL репозитория и т. д.), глобального уникального идентификатора – ID (GUID);

2) объект конфигурации содержит информацию о версиях, связи с репозиторием, GUID, вид платформы и связь с конфигурацией;

3) объект содержит команды проверки (checkout) скомпилированного элемента, тестовых команд и GUIDs, а также связи с каждой конфигурацией;

4) при определении конфигурации и платформы в каждом объекте появляется GUID, его свойства, среда выполнения и зависимости, которые могут быть объявлены статически или динамически. Статическая зависимость – это взаимоотношение между двумя конфигурациями, динамическая зависимость – взаимоотношение между конфигурацией и модулем.

ETICS по функциям близок концепции современной фабрике программ. Она базируется на наборах характеристик, услуг и процедур изготовления пакетов. Последние могут объединяться плагинами с описанием услуг для потребителей или поставщиков, средствами управления заданиями из рабочих мест, а также доступа к OCAM, архитектуре CPU, компиляторам с ЯП и средствами спецификации зависимостей между разными пакетами и их тестами при сборке программ

и их развертывании. Множество функциональных плагинов обеспечивает проверку контрактов, тестов выполнения разных элементов систем, генерацию документации, ведения готовых КПИ в оперативном или статическом репозитории ETICS.

Главная проблема в ETICS – преобразование некоторых компонентов систем для альтернативной платформы гетерогенной среды компьютеров путем ссылок с 16-, 32-разрядной платформы на 64-разрядную платформу среды Grid.

Глава 2. ПАРАДИГМА ОБЪЕКТНОГО ПРОГРАММИРОВАНИЯ

На рубеже 80-х XX столетия Г. Буч предложил объектный подход, который изменил сложившийся процесс структурного, функционального программирования. В то время индустрию программного обеспечения постиг *кризис сложности*. Нужна была не только новая *теория* Буча, но и *технология* выхода из кризиса [78, 79].

Объектно-ориентированная технология не только изменила традиционный структурный подход к разработки ПС средствами процедурных ЯП, но и сформировала новый стиль программирования разных компьютерных систем путем моделирования ПрО объектами и методами. Появились объектно-ориентированные ЯП, библиотеки классов объектов, routines и ТД. Они стали ресурсом общего назначения для разработки многих сложных систем, автоматизованными системами ООП (COM, CORBA, DCE RPC и т. п.). В системе CORBA формально определена объектная модель (ОМ) и брокер объектных запросов. Созданные из объектов и элементов сложные системы в априори менее сложные. Они обеспечивают пополнение и удаление элементов без особых трудностей, а это способствует снижению кризиса сложности ПС.

В период с 1992 – 2012 гг. отдел "Программная инженерия" ИПС НАНУ выполнял фундаментальные проекты в ГКНТ и НАНУ по развитию объектного, компонентного и сервисного программирования в направлении совершенствования процессов моделирования ПрО функциональными и компонентными объектами, а также сборки готовых ресурсов в более сложные системы. Был разработан объектно-компонентный метод ОКМ (Е. М. Лаврищева и В. Н. Грищенко [7]), описан в докторской диссертации (2007) [80] и получившей развитие в направлении создания теории взаимодействия и вариабельности ПС с учетом условий функционирования современных сред. При участии аспирантов и студентов в ОКМ были реализованы операции конфигурации КПИ в вариантные ПС и взаимодействия систем между собой в современных средах [81 – 84].

Была разработана технология построения ПС, в которой объединен на одной концептуальной основе теоретический аппарат объектного анализа и представления объектов-методов современными системами ООП, трансформации их программных компонентов и сборки в разные сложные структуры систем и семейств ПС, способных к формальному конфигурированию вариантов ПС и их семейств. Реализация отдельных аспектов этой технологии выполнена в комплексе ИТС на веб-сайте http://sestudy.edu-ua.net. В нем представлен спектр простых линий

по разработке компонентов и КПИ, сертификации, сохранения их в репозитории, а также выбора необходимых для сборки компонентов в сложные структуры с обеспечением их взаимодействия и преобразования типов данных общего назначения GDT к необходимым форматам готовых и изменяемых объектов ПС [39].

Линии сформулированы с участием сотрудников отдела и студентов кафедры ТТП, ИС факультета кибернетики и филиала МФТИ [55–58]. Среди линий есть линия электронного обучения дисциплинам программной инженерии, современным языкам C#, JAVA и работы прикладных и общих CASE-систем (Eclipse, Protégé, DSL Tool, JAVA и др.).

Ниже дано описание теоретических и прикладных аспектов ОКМ, базовых понятий и положений парадигмы компонентного программирования, включающей теории моделирования ПрО из объектов, доказательства изоморфизма отображения методов объектов в компоненты и их адаптация в разные среды функционирования.

2.1. Математическое моделирование объектной модели

Объектная теория построена с использованием базовых понятий объектного подхода Г. Буча и треугольника Фреге, исходя из принципов [7, 83, 84]:

1) всеобщности объектного определения, все сущности – суть объекты;

2) уникальности, каждый объект – уникальный элемент;

3) объектной упорядоченности, все объекты упорядочены в соответствии с их отношениями;

4) целостности объектной модели, объекты и отношения между ними однозначно определяются в ней на определенном уровне абстракции и описания.

5) интероперабельности объектов, объекты связываются операциями вызовов на множестве входных и исходных интерфейсов.

Г. Буч ввел в программирование понятия объекта, класса, наследования, инкапсуляции, полиморфизма. Эти понятия вошли в новые стили программирования объектного типа и сейчас широко используются системно и практически.

Принципы, понятия ООП и формализм треугольника Фреге позволили обобщить понятие объекта в виде "денотат, концепт, знак" и отношений между ними.

Объект выделяется на уровнях объектного анализа с привлечением логико-математических формализмов для описания и уточнения функций объектов в ОМ и отображения понятий, их сущностей, взаимоотношений и поведения объектов.

Построение модели ПрО начинается с декомпозиции объектов, определения их функций и свойств средствами четырехуровневого проектирования (обобщенного, структурного, характеристического и поведенческого) с помощью соответствующего логико-математического моделирования элементов объектов на этих уровнях.

Объект – именуемая часть действительной реальности с определенным уровнем абстракции имеет согласно понятийной структуре Фреге (денотат, знак, концепт). Каждый объект *OM* как сущность множества объектов $O=(O_0, O_1, ..., O_n)$, где $O_1 = Oi$ (*Nai, Deni, Coni*), а *Nai, Deni, Coni* соответственно означают – знак (имя), денотат и концепт объекта. $Coni = (P_{i1}, P_{i2}, ..., P_{is})$ определяется на множестве предикатов $P=(P_1, P_2, ..., Pr.)$ [1, 2].

Аксиома 1. Предметная область, которая моделируется из объектов, сама является объектом.

Аксиома 2. Предметная область, которая моделируется, может быть отдельным объектом в составе другой предметной области.

При моделировании объект ПрО имеет хотя бы одно свойство или характеристику и уникальную идентификацию в множестве объектов ПрО и множестве предикатов свойств и отношений между объектами.

Свойство объекта определяется на множестве объектов ПрО унарным предикатом, который принимает значение истины с помощью внешних и внутренних свойств или характеристик. Множество предикатов для модели ПрО может быть произвольным и определяться на отношениях объектов.

Характеристика – это совокупность свойств (унарных предикатов) с условием получения значений истины не более чем одним предикатом из совокупности внешних и внутренних характеристик из области значений свойств, которое имеет истину.

Отношение определяется бинарным предикатом на множестве объектов ПрО, принимает значение истины на заданной паре объектов. Отношению соответствуют операции обобщения, специализации, агрегации, ассоциации, детализации, классификации и др. Некоторые отношения имеют *роде-видовое* отношение (*IS–A*, либо *часть–целое* (*PART–OF*).

Уровни логико-математического моделирования ПрО

Проектирование модели ПрО выполняется на четырех уровнях логико-математического определения объектов:

I. Обобщающий уровень определяет базовые понятия ПрО без учета их сущности и свойств.

II. Структурный уровень определяет расположение объектов в структуре модели ПрО и установления отношений между ними.

III. Характеристический уровень задает общие и специфические особенности концептов объектов.

IV. Поведенческий уровень определяет поведение и изменение объектов в зависимости от событий, которые они создают при их взаимодействии.

Каждому из четырех уровней проектирования ОМ соответствуют концепции проектирования модели предметной области (рис. 2.4):

1) обобщенная теория формального определения понятия объекта с помощью денотатов теории Фреге и классов теории Геделя–Бернайса;

2) теоретико-множественного упорядочения объектов множества операциями объединения, пересечение, разницы, добавления, симметричной разницы);

3) логико-алгебраические операции на множестве объектов и их характеристик для установления отношений между объектами;

4) формальное определение поведения и состояния объектов с учетом времени их существования в ОМ.

Четырехуровневое проектирование ОМ состоит в последовательном применении этих концепций, начиная с обобщенной и заканчивая поведенческой. Это проектирование завершается нормализированным описанием сущностей ОМ с учетом соответствующего математического аппарата каждой абстрактной концепции.

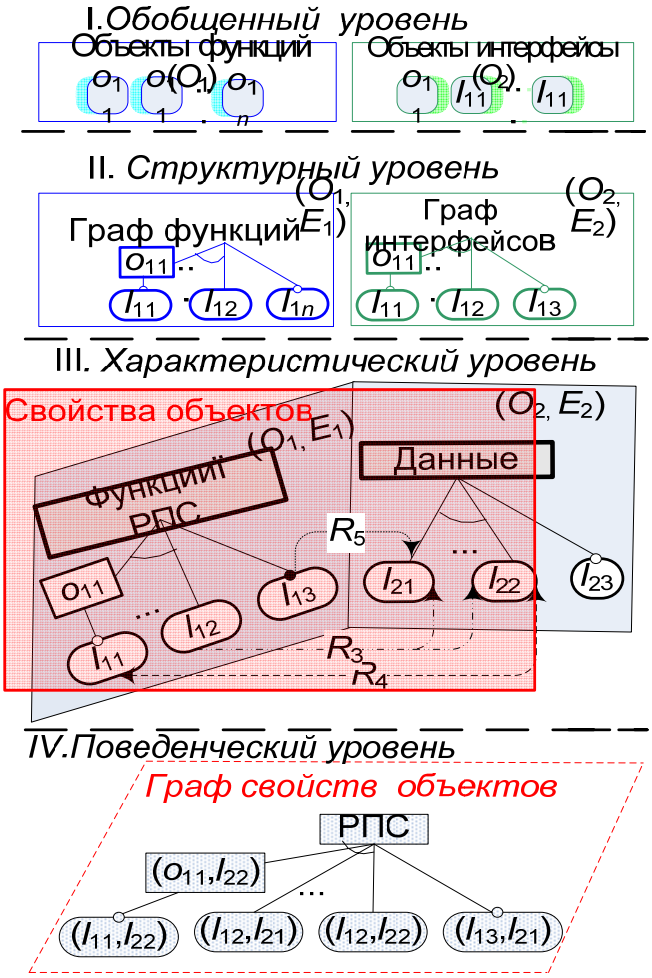


Рис. 2.4. Схема логико-математического проектирования ПрО

На базе определенных понятий формулируется определение принципа изменения объектов, обусловленного описанием и трансформацией объектов, полученных в рамках использования данных концепций. Определение объектов, исходя из использования этих четырех концепций, проводится по единственной схеме: объект, характеристика и операция.

Система формирования изменения объектов является основой концептуального моделирования, которая обеспечивает определение множества объектов ПрО, необходимых их характеристик и описания места каждого объекта в ОМ с заданием связей и поведения относительно других объектов.

Сущность логико-математической концепции

Концепция обобщения. Эта концепция предназначена для выделения и описания сущностей ОМ в процессе анализа ПрО при использовании соответствующих понятийных структур – треугольник Фреге с полным снятием неодно-

значности и неопределенности объекта, а по теории Геделя–Бернайса – объект рассматривается как класс.

После выполнения концепции обобщения выявленные объекты ПрО структурно упорядочиваются посредством бинарного отношения принадлежности.

Понятие класса заменяется понятием множества. Если объект является элементом другого объекта, то он определяется как множество. При этом не каждый объект – элемент какого-нибудь другого объекта (класса). Другими словами, объект, который соответствует всей ПрО, может не являться элементом другого объекта. При этом основное условие определения объекта такое: каждый объект является множеством или элементом другого множества.

Изменение – это упорядочение объектов с учетом отношения принадлежности и выделения элемента из множества. Порядок может быть определен через построение отображения определенного счетного множества (множеством природных чисел). Объект выделяется из множества всех элементов через построение отображения.

Операции над объектами по данной концепции выполняются с помощью теоретико-множественных операций – объединение, разница, добавление, симметричная разница, декартово произведение.

По данной теоретико-множественной концепции ниже дана конкретизация понятия объекта:

1) *класс* – это объект, который представляет собой множество;

2) *экземпляр класса* – это объект, который является элементом множества, которое само является классом;

3) *объединенный класс* – это множество, которое является прямой суммой других множеств;

4) *класс-пересечение* – это множество, которое является общей частью других множеств;

5) *агрегированный класс* – это множество, которое является подмножеством декартова произведения нескольких других множеств.

Логико-алгебраическая концепция

В рамках этой концепции множество объектов ПрО рассматривается как система алгебры, в которой, кроме основного множества, определена совокупность предикатов определенной сигнатуры, позволяющая проанализировать сущность объектов и выявить их особенности в рамках модели ПрО без учета внутренних свойств.

Подмножеству предикатов соответствуют следующие требования:

1) при определении объектов назначения предикатов и их количества является достаточными для удовлетворения условий концептуального моделирования ПрО, т.е. выделения любого объекта из множества объектов;

2) каждый конкретный предикат, его тип и сигнатура должны быть отображением субъективного восприятия сущности объекта исполнителем, который проводит анализ определенной ПрО.

Предикаты классифицируются с помощью операций:

1) *0-арные*, которые соответствуют константам и определяют установившиеся характеристики объектов ПрО;

2) *унарные*, которые соответствуют свойствам отдельных объектов;

3) *бинарные*, которые соответствуют взаимосвязям между отдельными парами объектов.

Свойство объекта. Это унарный предикат, который определяется на множестве объектов ПрО и принимает значение истины на данном объекте. Объект имеет следующие свойства. Внешнее свойство объекта это какой-либо аспект проблемной ориентации объекта в рамках данной модели ПрО. Внутреннее свойство объекта определяет статус объекта как множества и, в общем случае, совместно с другими внутренними свойствами объекта, принимающего участие в формировании его вида.

Объекту, который имеет одновременно статус множества, являясь элементом какого-либо множества, присуще два вида свойств – внешняя и внутренняя, как характеристик, способствующих формированию множеств.

Характеристика объекта – это совокупность свойств (унарных предикатов), которая является подмножеством множества, выделенных в системе предикатов и удовлетворяющих условию, за которым на каждом объекте принимается значение истины одновременно не более, чем одним предикатом из совокупности, которая определяет характеристику.

Дадим конкретизацию типов характеристик:

1) *внешняя характеристика объекта* – это характеристика, каждое свойство которой есть внешнее свойство объекта, и которая задает все аспекты проблемной ориентации объекта в рамках ПрО;

2) *внутренняя характеристика объекта* – это характеристика объекта, которая отображает внутреннее свойство данного объекта и определяет вид объекта как множество.

Характеристики имеют область определения и их значений. Областью определения характеристик объекта является упорядоченное множество, у которого существует взаимно однозначное соответствие между ним и совокупностью свойств, которые определяют заданную характеристику объекта. Значением характеристики объекта является элемент области ее определения объекта, который имеет свойство объекта и входит в состав данной характеристики со значением истины этого объекта.

Отношение. Взаимоотношение между объектами задается бинарным предикатом, который определяется на множестве объектов ПрО и принимает значение истины на заданной паре объектов.

При рассмотрении типов взаимоотношений используются все возможные комбинации двух элементов совокупности (множество, элемент множества). Кроме того, учитывается упорядоченность пары, первое и второе понятие. Взаимоотношение между объектами – это упорядоченная взаимосвязь объектов, которая возникает в результате определения отношений между ними по типу ООП.

К основным типам взаимоотношений множеств относятся:

1) множество–множество;
2) элемент множества–элемент множества;
3) элемент множества–множество;
4) множество–элемент множества;

Некоторые варианты типов взаимоотношений приведены ниже.

1. Тип взаимоотношений множество–множество включает в себя следующее.

Обобщение – это взаимоотношение, по которому каждое внутреннее свойство первого множества является внешним свойством второго множества.

Специализация – это взаимоотношение, обратное предыдущему.

Агрегация – взаимоотношение, по которому второе множество, рассматриваемой пары является агрегированным множеством, а первая является одним из сомножителей декартова произведения.

Детализация – это вариант взаимоотношения, где первое множество пары, является агрегированным множеством, а другое – сомножитель декартового произведения.

2. Тип взаимоотношения элемент множества–множество включает в себя следующие понятия.

Классификация – это вариант взаимоотношения, которое соответствует случаю, когда внешнее свойство соответствующего элементу множества является внутренним свойством множества.

Экземпляризация – вариант взаимоотношения, при котором внутреннее свойство множества является внешним свойством элемента множества.

Такой тип имеет два вида отношений.

Роде - видовое отношение (*IS–A*) – это отношение, которое определяет обобщение или специализацию.

Отношение часть - целое (*PART–OF*) – это отношение, которое определяет агрегацию или классификацию.

Приведенное высшее определение отношений и взаимоотношений базируются на структурной упорядоченности между парой объектов. Для полноты картины вводится понятие ассоциации, которое достаточно часто используется в практических ОМ.

Ассоциация – это произвольное взаимоотношение типа множество–множество, по которой для соответствующих объектов отсутствует структурная упорядоченность .

Изменение. В данной концепции отмена объектов определяется благодаря формированию необходимой системы предикатов над множеством объектов.

Если существуют несколько объектов, которые не удовлетворяют этому условию, то система предикатов не полная, ее нужно дополнить новыми предикатами либо путем упорядочения этих объектов.

Концепция описания поведения объектов

В наиболее общем случае эта концепция ориентирована на рассмотрение зависимостей характеристик объектов ПрО от времени, т. е. она зависит от фиксированного значения на отрезке оси, которая соответствует времени существования данного объекта.

Понятие времени – это конкретный параметр системы, значения которого упорядочено и каждому из них соответствует определенное состояние отдельных объектов и системы в целом. Математическим аппаратом формализованного представления объектов в отношении данной концепции является модель событий и сообщений.

Концепция поведения характеризуется часовым аспектом, а изменение объектов определяется разными значениями времени, как параметра. Если объект и его характеристики не изменяются, то он не зависит от времени.

Основным понятием данной концепции является понятие *состояния* объекта, который включает в себя:

1) *атрибут состояния,* который соответствует характеристике объекта и принимает значение истины на конкретном объекте с состоянием; *статический атрибут состояния* – это атрибут состояния с одним свойством;

2) *динамический атрибут состояния* – это атрибут состояния из нескольких свойств;

3) *значение атрибута состояния* – это значение характеристики объекта, при посредничестве которой определяется данный атрибут.

Для статического атрибута состояния это значение постоянно и соответствует существующему свойству. Для динамического атрибута состояние зависит от параметра, моделирующего время и свойства объекта при определенном значении этого параметра;

1) *состояние объекта* – это совокупность статических и динамических атрибутов состояния конкретного объекта, которые моделируют время;

2) *пространство состояний* – это множество всех возможных состояний, в которых может находиться определенный объект модели;

3) *диаграмма перехода состояний* – это совокупность всех возможных переходов между парами элементов пространства состояний, которые определяются всеми возможными вариантами жизненного цикла объекта;

4) *метод* – это операция, которая обеспечивает переход между состояниями объекта соответственно диаграмме перехода состояний;

5) *состояние объектной модели* – это совокупность состояний всех объектов модели из одного и того же значения параметра времени;

6) *событие* – реакция модели объектов и связана с необходимостью изменения этого состояния.

Событию соответствует отправление и получение сообщения, определенному значением параметра времени.

Приведенный комплекс понятий используется при построении ОМ с учетом заданных четырех концепций определения объектов ПрО.

Четырехуровневое проектирование модели ПрО

Логико-математическое проектирование модели ПрО проводиться согласно концепциям, которые описаны выше.

Структура процесса четырех уровневого проектирования приведена на рис. 2.1.

В соответствии с **обобщающим** уровнем объект рассматривается как математическое понятие или класс, который можно трактовать с точки зрения аксиоматической теории множества Геделя–Бернайса: множество $O=(O_0, O_1, ..., O_n)$, в котором O_0 – объект ПрО.

На этом уровне формируется множество базовых функций ПрО путем декомпозиции или композиции денотатов и концептов объектов. Над множествами объектов могут выполняться базовые операции (объединения, пересечения, разницы, дополнения, принадлежности и др.).

Для множества объектов $O=(O_0, O_1, ..., O_n)$ выполняется отношение:

$$\forall\, i[(i>0)\ \&\ (O_i \in O_0)]. \qquad (2.1)$$

При **структурном** уровни определяются такие понятия, как класс, экземпляр класса, абстрактный класс и др. Определение свойств объектов и их отношений (агрегация, детализация, классификация и др.) выполняется на характеристическом уровне, когда определяются: атрибут состояния, состояние, пространство состояний и др. Множество объектов упорядочивается и каждый из объектов может задаваться как множество или элемент множества. Тогда выражение (1.1) трансформируется в такой вид:

$$\forall\, i\ \exists j[(i>0)\&(j>0)\&\ (i\neq j)\&\ (O_i \in O_j)], \qquad (2.2)$$

который определяет отношение часть–целое, ассоциация, экземпляризация и агрегация.

В соответствии с **характеристическим** уровнем для каждого из объектов формируется соответствующий концепт. $O'=(O_1, O_2\ .\ O_n)$ – множество объектов ПрО, а $P'=(P_1, P_2\ .\ P_r)$ – множество унарных предикатов, связанных со свойствами объектов ПрО, и концепт объекта O_i определяется множеством утверждений, построенных на основе предикатов с P', которые принимают значение истины. Другими словами, концепт Coni = $\{P_{ik}\}$ при условии $P_k(O_i)$ = true, где P_{ik} является утверждением для объекта O_i в соответствии с предикатом P_k. Согласно этим правилам определяются свойства объектов в рамках отношения род–вид.

Выражение $A = (O', P')$ определяет систему концептов объектов O' алгебры и предикатов P' с помощью операций: *0-арных, унарных* и *бинарных*.

Аксиома 1. Каждый объект ПрО по крайней мере имеет хотя бы одну характеристику, которая определяет семантику и уникальную идентификацию в множестве объектов ПрО.

Исходя из **поведенческого** уровня, определяется последовательность состояний объектов и их процессы с отображением переходов состояний. Взаимосвязи между объектами формируются на основе бинарных предикатов, которые связаны со свойствами объектов ПрО, и детализируют взаимосвязи между состояниями объектов.

Понятие класса объектов заменяется понятием множества. Если объект – элемент другого объекта, то он определяется классом. При этом не каждый объект является элементом любого другого объекта (класса). Конкретизация понятия объекта – это *класс*, который задает *экземпляры класса* – объект; *объединенный класс* – множество, которое является прямым произведением нескольких других экземпляров; *класс-пересечение* – это множество, которое является общей частью других множеств; *агрегированный класс* – это множество, которое является подмножеством определенного декартова произведения нескольких других множеств объектов.

Декомпозиция и композиция объектов ПрО

На множестве базовых логико-математических функций проектирования для определения денотат и концептов объектов построена алгебра, включающая множество объектов и множество операций над ними. Каждая из операций имеет определенный приоритет и арность, связанные с соответствующими допус-

тимыми изменениями денотатов и концептов соответственно их детализации, экземпляризации, агрегации и др.

Объектный анализ как процесс декомпозиции ПрО базируется на алгебре анализа объекта, включающей в себя определение денотата ПрО и формальных их изменений в зависимости от полученных о них знаний. Алгебра это множество объектов $O'=(O_1, O_2, ..., O_n)$ и каждый объект $O_i = O_i(Name_i, Den_i, Con_i)$ *задается тройкой* имя, денотат и концепт $Name_i$, Den_i, Con_i соответственно; множество интерфейсов $I= (I_1, I_2, ..., I_n)$; множество действий (Action) $A' =(A_1, A_2, ..., A_n)$ как операций над элементами множества O; множество предикатов $P=(P_1, P_2, ..., P_r)$, на основе которых определяются концепты объектов $Con_i = (P_{i1}, P_{i2}, ... P_{is})$ на основе денотат.

К базовым операциям объектного анализа и построения тройки "имя, денотат и концепт" объекта отнесены следующие:

1) ***декомпозиционное изменение денотата*** для однородных объектов:

$$decds(O_i): O_i \rightarrow (O_{i1}, ... O_{ik}),$$

где $O_{ij} = O_{ij}(Name_{ij}, Den_{ij}, Con_{ij})$, $\forall j\ Con_{ij} = Con_i, \cup Den_{ij} = Den_i$ и для неоднородных объектов:

$$decdn(O_i): O_i \rightarrow (O_{i1}, ... O_{ik}),$$

где $O_{ij} = O_{ij}(Name_{ij}, Den_{ij}, Con_{ij})$, $\forall j\ Con_{ij} = \varnothing, \cup Den_{ij} = Den_{\text{i}}$.

2) ***композиционное изменение денотата*** для однородных объектов:

$$comds(O_{i1}, ... O_{ik}): (O_{i1}, ... O_{ik}) \rightarrow O_i,$$

где $O_i = O_i(Name_i, Den_i, Con_i)$, $\forall j\ Con_i = Con_{ij}, Den_i = \cup Den_{ij}$ и для неоднородных объектов:

$$comdn(O_{i1}, ... O_{ik}): (O_{i1}, ... O_{ik}) \rightarrow O_i,$$

где $O_i = O_i(Name_i, Den_i, Con_i)$, $Con_i = \varnothing, Den_i = \cup Den_{ij}$.

3) ***расширение концепта***. Если $(P_t \in P)$, $(P_t \notin Con_i)$ и $P_t(O_i)$ принимает значения истины, то

$$conexp(O_i, P_t): O_i \rightarrow O_i',$$

где $O_i' = O_i'(Name_i, Den_i, Con_i')$, $Con_i \cup \{P_t\} = Con_i'$

4) ***сужения концепта***. Если $P_t \in Con_{\text{i}}$, то

$$connar(O_i, P_t): O_i \rightarrow O_i',$$

где $O_i' = O_i'(Name_i, Den_i, Con_i')$, $Con_i' = Con_i \setminus P_t$.

Утверждение 1. Множество операций Ψ алгебры Σ является полной системой операций относительно функций объектного анализа.

Это утверждение обеспечивает адекватность перехода от функций к операциям алгебры объектного анализа:

1) изменение денотата путем декомпозиции однородных объектов decds (O_{i}): $O_i \rightarrow (O_{i1}, \dots O_{ik})$, где $O_{ij} = O_{ij}(Name_{ij}, Den_{ij}, Con_{ij})$; $\forall j\ Con_{ij} = Con_i$; $Den_i = Den_{i1} \cup, \dots, \cup\ Den_{ik}$. и неоднородных объектов decdn (O_i): $O_i \rightarrow (O_{i1}, \dots, O_{ik})$, где $O_{ij} = O_{ij}$ $(Name_{ij}, Den_{ij}, Con_{ij})$; $\forall j\ Con_{ij} = \varnothing$; $Den_i = Den_{i1} \cup ... \cup Den_{ik}.$;

2) изменение денотатов путем композиции однородных объектов comds ($O_{i1}, \dots, O_{ik}$): $(O_{i1}, \dots, O_{ik}) \rightarrow O_i$, где $O_i = O_i(Name_i, Den_i, Con_i)$; $\forall\ j\ Con_i = Con_{ij}$; $Den_{i1} \cup, \dots, \cup\ Den_{ik} = Den_i$ и неоднородных объектов *comdn* $(O_{i1}, ..., O_{ik})$: $(O_{i1}, ..., O_{ik}) \rightarrow O_i$, где $O_i = O_i(Name_i, Den_i, Con_i)$; $Con_i = \varnothing$; $Den_{i1} \cup \dots \cup Den_{ik} = Den_i$.

Аксиома 1. Если предикат $P_t \in P$, $P_t \notin Con_i$ и $P_t(O_i)$ принимает значение истины, то $conexp\ (O_i, P_t)$: $O_i \rightarrow O_i'$, где $O_i' = O_i'\ (Name_i, Den_i, Con_i')$; $Con_i \cup \{P_t\} = Con_i'$ является расширением концепта существующего объекта.

Аксиома 2. Если предикат $P_t \in Con_i$, то $connar\ (O_i, P_t)$: $O_i \rightarrow O_i'$, где $O_i' = O_i'\ (Name_i, Den_i, Con_i')$; $Con_i' = Con_i \setminus P_t$. является сужением концепта существующего объекта.

Каждая из операций имеет определенный приоритет и арность, а также связана с соответствующими изменениями денотатов и концептов.

Утверждение 2. Множество операций Ψ алгебры Σ является полной системой операций для функций объектного анализа, обеспечивает адекватность перехода от функций к операциям алгебры объектного анализа.

Правила объектного моделирования. Концептуальное моделирование 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, которая поддерживает целостность представления ОМ. Процесс завершается формализованным описанием сущностей и модели ПрО с учетом каждого аспекта абстрагирования и применения соответствующего математического аппарата.

Основные правила объектного моделирования:

1) объектный анализ выполняется при условии минимизации потери информации из описания действительной реальности для выбранной ПрО;

2) все изменения, которые происходят с ОМ, отвечают процессам детализации описания ПрО и определяются в рамках представления множества объектов, как совокупности треугольников Фреге;

3) каждый шаг объектного моделирования определяется изменениями денотата или концепта одного или нескольких объектов ОМ;

4) новые объекты на определенном шаге моделирования определяют соответственно изменения денотатов объектов;

5) все изменения, которые происходят с ОМ, отвечают условиям существования и определения формальных уровней абстракции представления объектов;

6) функции объектного анализа определяют преобразования в соответствии с допустимыми изменениями ОМ и ее отдельных элементов;

7) каждый шаг объектного моделирования обеспечивает целостность ОМ.

Приведенные формализмы способствуют последовательной дефиниции терминов и построению отношений между понятиями. Понятия объектов выполняются на уровнях с постоянным уточнением и развитием характеристик и свойств объектов. На обобщающем уровне определяется объект. Структурно-упорядоченный уровень обеспечивает детализацию таких понятий, как класс, экземпляр класса и др. На характеристическим уровне дается определение таких понятий: свойство объекта, отношения между объектами, агрегация; детализация, экземпляризация, ассоциация и др. На поведенческом уровне определяются понятия, которые задают состояние объекта, статические, динамические атрибуты состояния и метод.

2.2. Алгебра объектного анализа предметной области (ПрО)

Алгебра включает объекты и операции над ними:

$$\Sigma = (O', I', A') \qquad (2.3)$$

где $O'=(O_1, O_2 \ldots O_n)$ – множество объектов, $I = (I_1, I_2 \ldots, In)$ – множество интерфейсов; $A' =(A_1, A_2 \ldots, A_n)$ – множество (Action – A') операций над элементами множества O объектов. Каждая из операций A' имеет определенный приоритет и арность, а также связанные с соответствующими допустимыми описаниями концептов объектов и операций множества $A' = \{decds, decdn, comds, comdn, conexp, connar\}$, где *decds, decdn* – декомпозиции, *comds, comdn* – композиции и *conexp, connar* – сужение [84].

Утверждение 2. Множество операций A' алгебры объектов является системой действий для функций объектного четырехуровневого представления модели ПрО и операций детализации, экземпляризации и агрегации.

Модель ПрО задается объектным графом G (рис. 2.5, 2.6). Граф рис. 2.5 соответствует обобщенному и структурному уровням логико-математического проектирования ОМ. Граф на рис. 2.6 соответствует ОМ, расширенной интерфейсными объектами для связи их между собой.

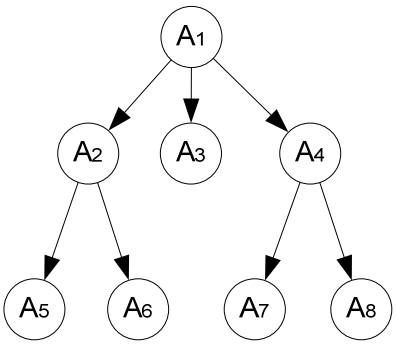


Рис. 2.5. Граф ОМ

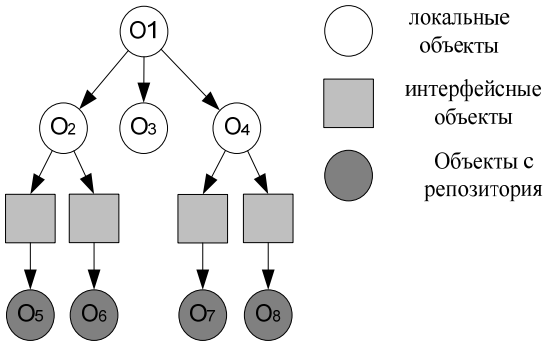


Рис. 2.6. Граф ОМ с интерфейсними объектами

Общий граф $G =\{O, I, R\}$ определен на множестве объектов O, интерфейсов I и отношений R (relations) между объектами.

Вершины графа G задают объекты, которые находятся в репозитории, а дуги соответствуют отношениям между ними, Отношения могут задаваться интерфейсными объектами (рис. 2.6). Его элементы описываются в ЯП, а интерфейсные объекты – в специальном языке. Параметры внешних характеристик интерфейсных объектов передаются между объектами и специфицируются *in, out, inout* в языке IDL системы Corba. Объекты помещаются в библиотеку или репозиторий.

В интерфейсном объекте описываются входные и выходные параметры, которыми обмениваются связанные между собой функциональные объекты. Для объектов O_2 и O_5 интерфейсным объектом является $O_{2.5}$. который обеспечивает обмен данными для проведения вычислений. Если типы передаваемых параметров в интерфейсном объекте нерелевантные (*in* integer, *out*- real), то в нем генерируются функции преобразования integer ↔ real [36, 61].

Таким образом, модель ПрО задается объектным графом $G = \{O, I, R\}$, определенным на множестве объектов ПрО, интерфейсов I и отношений R (relations) между объектами. Графу присущи такие правила:

1) существует хотя бы одна вершина, которая имеет статус множество-объект и отображает ПрО в целом;

2) множество вершин *графа* задает взаимно однозначное отображение объектов ПрО;

3) каждой вершине соответствует хотя бы один интерфейс $Ik \in I$ и отношения, которые принадлежат множеству R соответственно правилам.

Множество объектов-функций связано с набором методов реализации удаленных объектов ПрО. При конкретизации объекты графа G имеют связи через интерфейсные объекты из множества I. Другими словами, вершины графа G – объекты двух типов – функциональные O та интерфейсные I.

Интерфейсным объектам графа соответствует описание данных, методы их передачи через запросы RPC или RMI данных и возможные операции преобразования этих данных к соответствующим форматам среды выполнения.

Результатом связи двух объектов графа (например, O25, O47) есть интерфейсный объект, в которого множество входных интерфейсов совпадает с множеством интерфейсов объекта-приемника, а множество исходных интерфейсов – это множество интерфейсов объекта-передатчика.

Аксиома 2. Построенный граф G дополненный интерфейсными объектами структурно упорядочивается (наверх) с контролем полноты и избыточности устранения дублирующих элементов.

Объекты могут иметь несколько интерфейсов, которые наследуют интерфейсы других объектов, тогда последние предоставляют сервис всего множества исходных интерфейсов.

Множество объектов и интерфейсов графа задается общими и индивидуальными характеристиками ОМ. Проверка характеристик объектов проводится операциями (экземпляризации, классификации, специализации, агрегации и др.). Каждая из них это попарное сравнение внутренних характеристик объекта с их внешними характеристиками. Они считаются достоверными, если выполняются такие условия: каждое внутреннее свойство эквивалентно внешнему свойству объекта. Если это условие не выполняется, то такой элемент удаляется из списка элементов множества и из графа соответственно.

2.3. Методы объектов и их интерфейсы

Принципы проектирования ПрО из объектов и интерфейсов:

1) классификация, полиморфизм и наследование объектов;
2) типизация (typing, subtyping) – абстракция объектов;
3) UML-диаграммы представления схем объектов и их отношений;
4) интероперабельность объектов для взаимодействия через интерфейс и др.

Подход к формальному проектированию распределенных ПС из объектов-функций и интерфейсов состоит в построении модели ПрО, как совокупности понятий, концептов, объектов и их атрибутов и классифицированных связей через интерфейсы, задаваемыми между объектами.

Модель предметной области имеет вид

$$M_{ПС} = \{M_f, M_i, M_d\},$$

где M_f – множество функций модели ПрО, которым соответствуют объекты множества $O = \{O_1, O_2., Or\}$; M_i – множество интерфейсов *in*, *out* и *inout* для пары объектов. С практической точки зрения все общие ТД специфицируются как внешние данные в паспортах объектов и как внутренние – самих объектов множества O; M_d – множество данных и метаданных предметной области ПС, которые специфицированы примитивными или сложными типами данных, которые имеются в каждом ЯП описания функций объектов.

Определение базовых понятий распределенной ПС в терминах объектов приведено ниже:

F – множество функций объектов

I – множество интерфейсов объектов;

O – множество объектов $O=O_1, O_2..., O_k$;

In – множество входных (*In*) интерфейсных объектов объектов от клиента)

$$O_k \in O, In(O_k)$$

Out – множество выходных интерфейсов (объектов от сервера)

$$O_k \in O, Out\ (O_k)$$

Inout – промежуточные интерфейсы

Модель объединения объектов включает в себя свойства и характеристики объектов модели $M_{ПС}$, которые отображаются в описание интерфейса компонентов в IDL (параметры *In*, *Out*) и операций принадлежности:

Результатом объединения двух объектов будет компонентный объект, у которого множество входных интерфейсов ($O_k \leftarrow O_l$) совпадают с множеством выходных интерфейсов объекта от клиента, а множество выходных интерфейсов совпадает с множеством входных интерфейсов объекта от сервера:

Аксиома 3. Операция взаимодействия O_k, O_l дает объект, в котором множество интерфейсов *In* совпадает с множеством интерфейсов *Out*, а множество *Out* интерфейсов – с множеством *In* интерфейсов:

$$O_k = (Out(O_k), In(O_k)),$$
$$O_l = (Out(O_l), In(O_l)),$$
$$O_k \cdot O_l = (Out(O_k), In(O_l)).$$

Аксиома 4. Композиция объектов $O_k \cdot O_l$ является корректной, если объект-клиент полностью обеспечивает сервис, необходимый объекту-серверу, т.е. имеем

$$\forall I_m \in In(O_k) \Rightarrow \exists I_n \in Out(O_l) \wedge I_m = I_n.$$

Компонентные объекты могут иметь несколько интерфейсов, которые могут наследовать интерфейсы других объектов ($O_k \leftarrow O_l$), тогда последние предоставляют сервис для всего множества входных интерфейсов:

$$O_k \leftarrow O_l \Rightarrow Out(O_k) \subseteq Out(O_l).$$

В случае, когда объект наследует другой объект, у которого множество входных интерфейсов содержит все его интерфейсы, а множество выходных интерфейсов содержит только интерфейсы, которые необходимы для задания сервиса.

Формальные операции над объектами и интерфейсами

Можно построить нескольких видов проекции объектов:

1) проекция объекта на *интерфейс* , как объекта, в котором *In* содержит один интерфейс, а множество интерфейсов *Out* содержит только те интерфейсы, которые необходимы для предоставления сервиса;

2) проекция объекта на *множество интерфейсов;*

3) проекция *объекта на объект* – это проекция объекта на множество входных интерфейсов объекта;

4) проекции *объекта на объект и взаимодействие* дает равенство.

$$(O_k \cdot O_l)[I_m] = O_k[I_m] \cdot O_l .$$

Операция *параллельного выполнения* РПС имеет вид Po || . || Pr.

Операция *взаимодействия объектов и среды* задает объект, у которого множество интерфейсов *In* совпадает с множеством интерфейсов *In* объекта-сервера, а множество интерфейсов *Out* является объединением множеств интерфейсов *Out* для объектов среды.

Аксиома 5. Взаимодействие объекта и среды корректное, если выполняется условие: среда полностью обеспечивает сервис, необходимый объекту клиента:

$$O_k \cdot \left(O_{l_1} \| \ldots \| O_{l_n}\right) = \left(Out(O_k), \bigcup_{j=1}^{n} In\left(O_{l_j}\right) \right),$$

$$\forall I_m \in In(O_k) \Rightarrow \exists I_n \in \bigcup_{j=1}^{n} Out\left(O_{l_j}\right) \wedge I_m = I_n .$$

Операция наследования объектом интерфейса из РПС дает объект, который унаследовал интерфейсы всех объектов среды. Объект, который наследуется, передает все интерфейсы и имеет следующие свойства:

1) *транзитивности* $\forall O_{1,2,3} \in O : O_1 \leftarrow O_2, O_2 \leftarrow O_3 \Rightarrow O_1 \leftarrow O_3$,

2) *симметричности* $\forall O_k \in O \Rightarrow O_k \leftarrow O_k$.

Операция параллельного выполнения программ РПС не является симметричной:

$$O_k \leftarrow \left(O_{l_1} \| O_{l_2}\right) \neq O_k \leftarrow \left(O_{l_2} \| O_{l_1}\right).$$

Результатом отображения объекта на объект является интерфейс из множества входных интерфейсов $O_k[O_l] = O_k[In(O_l)]$ и выходных интерфейсов $O_k\,[O_1] = O_k\,[Out(O_i)]$.

Описание интерфейсов

Формализм описания интерфейсов IDL представлен в OMG CORBA. В нем интерфейсные посредники (Stub, Skeleton) содержат описание передаваемых данных между объектами в языке ЯП для клиента и сервера (Stub для клиента и Skeleton для сервера) и операции передачи данных между ними.

Описание интерфейса в IDL начинается с ключевого слова **interface**, за которым следует: имя интерфейса, описание типов параметров и операций (op_dcl) вызова объектов:

interface A { ... }
interface B { ... }
interface C: B, A { ... }.

Параметры операций (op_dcl) в задании интерфейсов это:

1) тип данных (type_dcl);
2) константа (const_dcl);
3) название исключительной ситуация (except_dcl), которая может возникнуть в процессе выполнения метода объекта;
4) атрибуты параметров (attr_dcl).

Описание типов данных (ТД) начинается ключевым словом typedef, за которым следует базовый или конструируемый тип и его идентификатор. В качестве константы может быть некоторое значение типа данного или выражение, составленное из констант. ТД и константы описываются как фундаментальные типы данных: integer, boolean, string, float, char и др.

Описание операций op_dcl передачи данных включает в себя:

1) наименование операции интерфейса;
2) список параметров (от нуля и более);
3) типы аргументов и результатов, иначе – *void*;
4) управляющий параметр или описание исключительной ситуации и др.

Атрибуты передаваемых параметров начинаются служебными словами: **in** – при отсылке параметра от клиента к серверу; **out** – при отправке параметров-результатов от сервера к клиенту; **inout –** при передаче параметров в оба направления (от клиента к серверу и обратно).

Описание интерфейса для одного объекта может наследоваться другим объектом и тогда это описание становится базовым. Пример такого дан ниже:

```
const long l=2
interface A {
void f (in float s [l]); }
interface B {
const long l=3 )
interface C: B, A { }.
```

Интерфейс С использует интерфейс В и А. Это означает, что интерфейс С наследует описание типов данных А и В, которые по отношению к С являются внешними. Но при этом синтаксис и семантика остаются неизменными. Согласно приведенного примера - операция функции f в интерфейсе С наследуется из А.

Механизм наследования интерфейса состоит в сохранении имен объектов без их переопределения. Это касается описания операций, которые должны иметь уникальные обозначения.

Имена операций могут использоваться динамически во время выполнения интерфейса *skeleton.*

Общая структура описания модуля с интерфейсом в языке IDL имеет вид:

```
Regust Operations
module CORBA {
interface Reguest {
```

Status add-arg (
in Identifier name,
in Flags arg_flags
);
Status invoke (
in Flags invoke_flags // invocation flags
);
Status send(
Status get_respouse (
out Flags response_flags // response flags);););

Тип описывается в классе ТД, которые передаются через параметры операторов RPC, RMI, а также протоколами в WCF VS.Net и др. ТД описывается в ЯП ООП (C#, Vbasic, Pascal, и др.).

Входные и выходные интерфейсы для программ P_1, P_2 (рис.2.7) имеют разную семантику, но одинаковое синтаксическое описание в некотором ЯП. Передача данных от этих программ для P_3 осуществляется через функции F_1 (…) , F_2 (…) и интерфейсы *In, Out*., с помощью которых осуществляется преобразование ТД переданных между P_1, P_3 и P_2, P_3 туда и обратно.

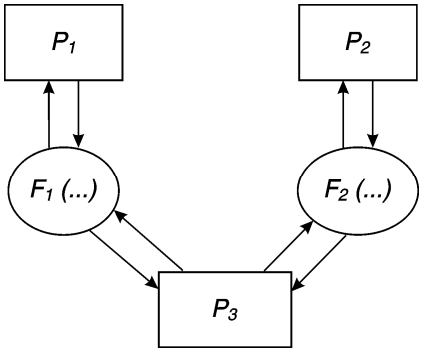


Рис. 2.7. Схема вызова функции объектов

Данный аппарат интерфейса реализован в системе CORBA, ОС IBM, Microsoft и др. Его основу составляют библиотеки преобразования ТД в этих ОС, которые применяется при интеграции разнородных программных объектов.

Классы объектов. Объекты объединяются в соответствии общим характеристик в классы. ОМ имеет такой формальный вид:

$$OM = (Oclass, GK),$$

где $Oclass = \{Oclassi\}$ – множество классов объектов функций или методов с общими свойствами; GK – объектный граф, что задает связи и отношения между классами и ее экземплярами.

Каждый класс представляется в виде:

$$Oclassi = (ClassNamei, Methi, Fieldi\},$$

где *ClassNamei* – имя класса; $Methi = \{Methji\}$ – множество методов; $Fieldi = \{Fieldni\}$ – множество переменных, определяющих состояние экземпляров класса.

Пусть *Pfieldi* $\subset$ *Fieldi* – множество внешних переменных (public), которые доступны.. Каждому *Pfieldni* $\in$ *Pfieldi* отвечает метод *get* <$Pfield_n^i$> и *set* <*Pfieldni*> для задания и выборки значений соответствующих переменных, как атрибутов объектов и интерфейсов ОМ и компонентных моделей, в которых методы объектов отображаются программными компонентами.

Множество методов имеет вид:

$$Imethi = Methi \cup \{get <Pfield_n^i>\} \cup \{se <Pfield_n^i>\}.$$

Ему сопоставляется интерфейс *Ifunci*, который состоит из методов, входящих в *Imethi.* 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 их представлению в виде элементов множества интерфейсов в интерфейсном графе.

***Определение* 2.** 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 происходит через public-методы и управляемые переменные, то для него реализуется интерфейсный принцип доступа.

***Теорема* 2.** Если для каждой вершины ОМ внешнее взаимодействие с классами происходит с помощью public-методов и управляемых переменных, то образуется единый интерфейс *ISyst* с эквивалентной функциональностью.

Эта теорема определяет условия существования единого эквивалентного отображения между объектным и интерфейсным представлением элементов ОМ модели.

2.4. ЖЦ объектного моделирования ПрО

Объектная технология возникла из опыта разработки реальных систем и стала по существу обобщением опыта разработчиков больших и сложных программ за последние десятки лет.

Технология базируется на принципах Г. Буча – классы, наследование, полиморфизм, которая ориентирована на новую структуризацию ПС и упрощает моделирование и проектирование ПрО. Если сложную программу представлять из объектов, то она легче создается, получается более качественной и надежной, ее легче исправлять путем добавления или удаления объектов, методов и интерфейсов.

Технология ООП включает в себя процессы ЖЦ:

1. Анализ ПрО.
2. Формулировка требований.
3. Создание модели ОМ из объектов.
4. Проектирование ОМ и описание объектов объектно-ориентированными языками (Basic, C#, JAVA и др.).
5. Описание интерфейсов объектов в языке IDL, API и др.
6. Обработка программных объектов в среде соответствующей ООП системы программирования с объектно-ориентированного языка.
7. Отладка и тестирование отдельных программ и ПС с применением объектных инструментов: JAVA, JAVA Beans, JAVAScript, C#, Vbasic и др.

2.5. CASE-средства объектного подхода в современных средах

В настоящее время широко применяются системы (ONC SUN, OSF DCE, COM, SOM, CORBA, JAVA и др.), представляющие собой разные возможности собирать, взаимодействовать программным объектам и компонентам вместе в структуре ПС, на основе стандарта взаимодействия открытых систем OSI (Open Systems Interconnection) [45–50].

Особенности взаимодействия объектов в ONC SUN и OSF DSE. Системы обеспечения взаимодействия объектов основаны на механизмах удаленного вызова RPC, задаваемого языками высокого и низкого уровня в виде описания интерфейса взаимодействующих объектов. 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:

1) оператор передачи сообщений (RPC-вызовы удаленных объектов сети);
2) сетевые сообщения между компонентами по передаче данных;
3) средства преобразования типов данных с ЯП высокого уровня к типам данных ЯП низкого уровня, а также кодирование и декодирование данных подобно базовым операциям (put и get).

Преобразования данных, связанных в основном 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

Связь объектов 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:

1) механизмы передачи запросов удаленным объектам через stub и skeleton;
2) обмен данными через сеть и их преобразование в случае различий в архитектуре и платформе компьютеров среды;

3) системы преобразования типов данных для каждой пары ЯП (C↔Smaltalk, Smaltalk↔ADA, ADA↔COBOL, COBOL↔ JAVA, COBOL↔АДА и др.), которые строятся аналогично описанным в главе 1.

Формализмы в системе DCOM такие:

1) механизмы передачи данных (типа RPC-вызов);

2) сетевой обмен данными;

3) системы передачи данных и преобразования нерелевантных типов данных (C↔C++), 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. Спецификации ТД в языке IDL системы CORBA такие: *in* – входные, *out* – выходные, *inout* – результат. К передаваемым ТД относятся – 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 таких полей во многих местах программы. Определение типа переменной длины – рекурсивное, оно влечет за собой изменение "fixed или variable" для составных типов. Параметры OUT и RESULT для любых типов переписывают их 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 присваивания T_var & operator = (T) параметр T использует память для динамического выделения памяти. Данные копируются перед заполнением их в T_var.

Например, заполнение типа String_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 предусмотрена. 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 используется для параметров OUT.

Принципы взаимодействия объектов 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 ТД 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 (stub, sceleton) и параметров для методов объектов в промежуточном слое. Связь между объектными моделями каждого ЯП системы COM и JAVA выполняет брокер ORB.

Если в общую объектную модель CORBA входит OM COM, то в ней ТД определяются статически, а конструирование сложных ТД осуществляется для массивов и записей. Методы объектов используются в двоичном виде, и допускается совместимость машинного кода объекта из одной среды разработки к коду другой

среды, а также совместимость разных ЯП за счет отделения интерфейсов объектов от реализаций посредниками stub, sceleton в языке IDL.

В случае вхождения в состав модели CORBA объектной модели JAVA/RMI, вызов удаленного метода объекта осуществляется ссылками на объекты, задаваемые указателями на адреса памяти. Интерфейс как объектный тип реализуется классами и предоставляет удаленный доступ к нему сервера. Компилятор JAVA создает байт-код, который переносит с одной платформы на другую в среде CORBA.

Средства преобразования ТД 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:

1) оператор вызова удаленных методов RMI;
2) сетевой обмен данными между удаленными компонентами;
3) виртуальная машина для интерпретации битовых кодов объектов компилятора C++ в среде JAVA.

Преобразование сложных данных объектов осуществляется с помощью функций отображения типов, описанных в IDL. Например, преобразование ТД **struct** включает последовательное преобразование всех ее полей 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 библиотеке C++. Тип данных **array** преобразуются специальными функциями и процедурами для простых ТД**.**

Функции преобразования базовых типов учитывают информацию о границе выравнивания и о размере (формате) данных, совпадающими с методами класса CDR (Common Domain Routine), а также сложных ТД (запись, таблица и др.).

Глава 3. ПАРАДИГМА КОМПОНЕНТНОГО ПРОГРАММИРОВАНИЯ

По оценкам экспертов, 75 % работ по программированию используют КПИ.

И несмотря на это дублируются часто использованные задачи (например, программы складского учета, начисления зарплаты, расчета затрат на производство продукции и т. п.). Многие из готовых программ – типовые и образуют классы многоразовых КПИ.

Одно из ключевых направлений в компонентном программировании – это разработка компонентов и оформление из них КПИ. В КПИ отражается многолетний опыт компьютеризации разных сфер человеческой деятельности, который может применяться путем их настройки и адаптации к новым условиям функционирования.

КПИ начли использовать в программировании при проектировании из простых элементов сложных программ по методу снизу–вверх [84 – 88].

Главным преимуществом этого метода является использование готовых и реализованных функций в виде подпрограмм и программ. Этот метод уменьшал затраты на разработку новых и сложных систем. Требовались поиск и выбор КПИ с типовыми функциями для данной системы и настройки их на новые условия применения, на что тратилось гораздо меньше усилий, чем на аналогичную новую разработку.

Для поиска готовых компонентов в ООП сложились новые методы классификации и каталогизации КПИ. Метод классификации состоял в спецификации программы в виде паспортной информации о компонентах с указанием вида и типа функции программы, метод каталогизации – в физическом размещении готовых программ в библиотеках и репозиториях.

Компонентное проектирование систем дополняет и расширяет ООП. В нем объекты рассматриваются на логическом уровне проектирования ПС, а переход к компонентам происходит при физической реализации функции объекта.

Компонент конструируется как некоторая абстракция, включающая в себя информационный раздел (назначение, дата изготовление, условия применения и т. п.). КПИ – это артефакт, включающий в себя реализацию (implementation), интерфейс (interface) и схему развертывания (deployment) компонента. Он может иметь несколько реализаций в зависимости от операционной среды, модели данных, СУБД и др. Интерфейс хранит данные для связи одного компонента с другими. Компонент может иметь несколько интерфейсов. Развертывание представляет собой выполнение физического файла в соответствии с конфигурацией.

Компоненты наследуются в виде классов и используются в модели компонента и каркаса (Фреймворка) в интегрированной среде. Компонент описывается в любом ЯП и не зависит от операционной среды и реальной платформы. Они могут размещаться в компонентной среде, как внутри одного сервера, так и в разных серверах и платформах. Существуют следующие типы компонентов:

1) серверы компонентов;
2) контейнеры компонентов;
3) экземпляры компонентов внутри контейнеров;
4) клиентские компоненты, веб-клиенты и др.;
5) компонентные программы и системы.

Каждый из этих типов имеет спецификации, требования и правила взаимодействия с другими объектами среды.

Другими словами, в компонентном программировании сформировались понятия и положения о компонентах (их свойствах, параметрах и атрибутах качества), подходы к спецификации компонентов и их интерфейсов, о способах сборки, а также отдельные теоретические положения парадигмы компонентного программирования. Это программирование дополняет и расширяет существующие методы программирования со своей концептуальной базой, теорией, методологией и инструментами, а также своими средствами обеспечения качества и расходов на разработку комплексной системы. Компонентный стиль программирования сложных систем из отдельных КПИ характеризуется:

1) большей формализацией и упорядоченностью процесса разработки отдельных компонентов и систем с КПИ;

2) механизмами описания интерфейсов компонентов и их выполнения в интегрированной среде с использованием общесистемных сервисов и др.;

3) средствами эволюционного изменения систем из КПИ.

В рамках компонентного программирования с участием аспирантов разработан оригинальный метод ОКМ комплексного анализа и компонентного моделирования сложных программных систем (В. Н. Грищенко и Е. М. Лаврищева). В нем дано обобщение понятия объектов, как элементов реального мира с соответствующими свойствами и характеристиками, которые последовательно определяются и уточняются средствами логико-математического формализма описания объектов, с присвоением им необходимых и достаточных свойств и характеристик, которыми они отличаются друг от друга. Доказательство принадлежности свойств объектов каждому из них и множеству выполняется с помощью формализма треугольника Фреге и классов Геделя–Бернайса. Разработана компонентная алгебра – внешняя, внутренняя и эволюционная. Она устанавливает теоретическую и практическую связь между объектным анализом и компонентным методом создания ПС. Метод ОКМ – метод последовательного перехода от объектов к компонентам и их интерфейсам.

3.1. Теория компонентного программирования. Базовые понятия

Переход к компонентам происходил эволюционно, начиная от подпрограмм, модулей и функций. При этом усовершенствовались сами элементы, методы их композиции и накопления для последующего использования [7].

К базовым элементам относятся: объект, компонент и сервис. Каждый из них включает: в себя описание имени (идентификатор): описание интерфейса в виде операторов вызова и параметров.

По сути компонент является самостоятельным продуктом, который реализует функцию предметной области и может взаимодействовать с другими компонентами через интерфейсы. Каждый компонент конструируется самостоятельно, как некоторая абстракция, которая содержит в себе информационную часть и артефакт.

Информационная часть имеет паспорт, который содержит назначение, дату изготовления, условия применения (ОС, среда, платформа и т. п.) и др.

Артефакт – это реализация (implementation), интерфейс (interface) и схема развертывания (deployment) компонента**.**

Реализация – это код, который будет выполняться при обращении к операциям, определенным в интерфейсах компонента. Компонент может иметь несколько реализаций в зависимости от операционной среды, модели данных, СКБД и др.

Интерфейс – это операции обращения к другим компонентам в языках IDL или APL для передачи аргументов и результатов при взаимодействии компонентов между собой. Каждый компонент может иметь множество интерфейсов.

Развертывание – это выполнение физического (кода) конфигурационного файла компонента путем запуска.

Определение 3.1. *Программный компонент* или просто компонент – это независимый от ЯП, самостоятельно реализованный программный элемент, который обеспечивает выполнение определенной совокупности прикладных сервисов,

доступ к которым возможен только с помощью интерфейсов, которые определяют функциональные возможности и порядок обращения к ним.

Компонент отображает некоторое типовое решение, имеет типовую архитектуру, характеристики, и атрибуты в его интерфейсной части для обмена данными и взаимодействия их в разных средах. По существу компонент становится неделимым и инкапсулированным объектом, удовлетворяющим функциональным требованиям к компонентной архитектуре ПС и среде взаимодействия.

Определение 3.2. *Компонент повторного использования* (КПИ) – это готовый программный компонент (проектное решение, функция, шаблоны и др.), который используется в ходе разработки не только самими разработчиками, но и другими пользователями. КПИ упрощает и сокращает сроки разработки ПС. В библиотеках, репозиториях содержится огромное количество готовых КПИ, которые можно применять в новых проектах.

С формальной точки зрения КПИ – это

$$\text{КПИ} = (T, I, F, R, S),$$

где T – тип компонента, I – интерфейс компонента; F – функциональность, R – реализация, S – сервис взаимосвязи с компонентами и средой.

Определение 3.3. *Компонентная программа* – это совокупность компонентов, которые необходимы для обеспечения функциональных и нефункциональных требований, которая построена и функционирует в соответствии с правилами создания компонентных конфигураций и взаимодействия КПИ.

Модели компонента, среды и интерфейса

Технология построения ПС из компонентов и КПИ базируется на механизмах поиска и отбора компонентов, аннотирования и размещения готовых компонентов в репозитории.

Сущность компонентного программирования заключается в создании сложных ПС из готовых компонентов и КПИ. Главная цель этого программирования – представление компонентной ПС как совокупности взаимодействующих КПИ с помощью метода сборки компонентов в сложную систему для последующего ее выполнения в компонентной среде.

Разработка формального аппарата построения сложных систем из КПИ и новых разработанных компонентов провел отдел в рамках фундаментальных проектов (1998–2001 и 2002–2006) и диссертационных исследований. Этот аппарат включает в себя модели компонентов, компонентной среды и программы, компонентную алгебру (внешнюю, внутреннюю и эволюции), а также интерфейсы и систему преобразования ТД взаимодействующих компонентов ПС.

Принципами построения ПС из компонентов являются следующие:

1) композиционность сложных систем из компонентов;

2) конструктивность построения доменов из новых компонентов и унифицированных КПИ, каталогизированных в хранилищах и библиотеках;

3) интероперабельность КПИ и ПС через интерфейсы и правила взаимодействия компонентов между собой адекватно в других средах функционирования;

4) вариантность – способность КПИ в компонентных ПС к изменениям – удаление, добавления новых КПИ в конфигурационную структуру ПС;

5) производительность вычисления ПС в гетерогенных средах с использованием данных, которые накоплены в виртуальных хранилищах данных [7].

Модель компонентной системы – это типовая архитектура ПС из компонентов и интерфейсов, моделей и их реализаций на основе каркасов (рис. 2.8).

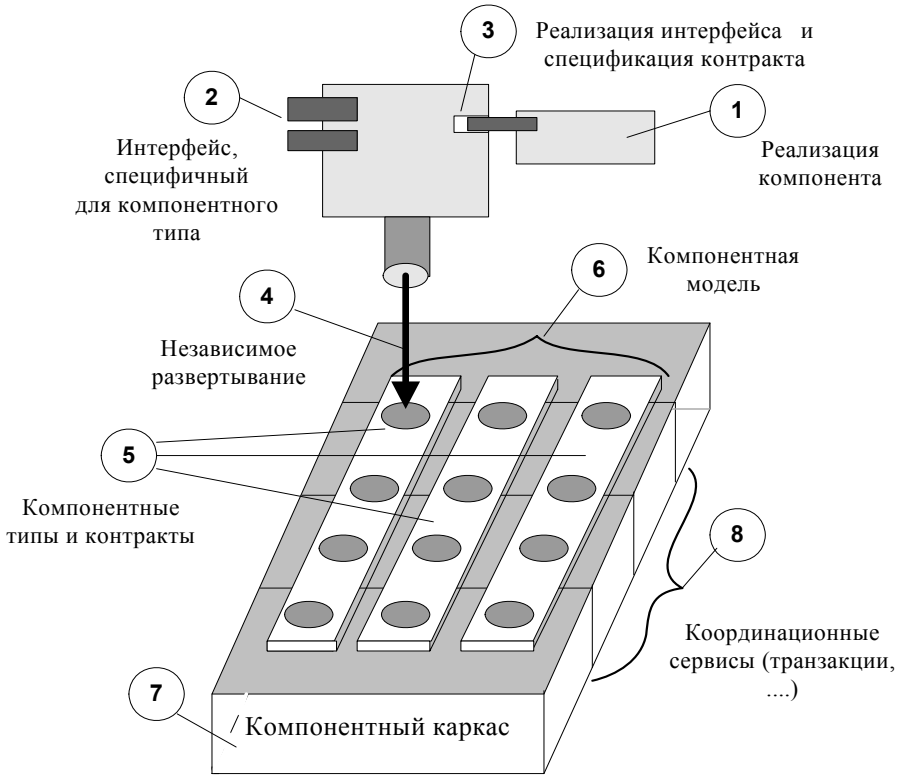


Рис. 2.8. Модель компонентной системы

Модель компонента обобщает типовые решения, касающиеся функциональной сущности компонента, его архитектуры, структуры, свойств и характеристик [84, 85].

Формально модель компонента имеет вид

$$Comp = (CName, CInt, CFact, CImp, CServ), \quad (3.1)$$

где *CName* – уникальное имя компонента; $CInt = \{CInt^i\}$ – множество интерфейсов, связанных с компонентом; *CFact* – интерфейс управления экземплярами компонента; $CImp = \{CImp^j\}$ – множество реализаций компонента; $CServ = \{CServ^r\}$ – множество системных сервисов.

Модель компонентной среды:

$$CE = (NameSpace, IntRep, ImpRep, CServ, CServImp),$$

где $NameSpace = \{CName^m\}$ – множество имен компонентов среды; $IntRep = \{IntRep^i\}$ – репозиторий интерфейсов компонентов среды; $ImpRep = \{ImpRep^j\}$ –

репозиторий реализаций; $CServ = \{CServ^r\}$ – интерфейс множества системных сервисов; $CServImp = \{CServImp^r\}$ – множество реализаций для системных сервисов.

Модель интерфейса. Каждый *интерфейс* компонента имеет вид

$$CInt^i = (\ IntName^i,\ IntFunc^i,\ IntSpec^i\ ) \qquad (3.2)$$

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:

$$\forall CInt^i \in CInt\ \exists CImp^j \in CImp\ [Provider(CInt^i) \subseteq CImp^j], \qquad (3.3)$$

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

$$Sign\ (CInt^i{}_1) = Sign\ (CInt^k{}_2)\ \&\ Provider(CInt^i{}_1) \subseteq CImp^j{}_2, \qquad (3.4)$$

где $Sign(\ldots)$ означает сигнатуру соответствующего интерфейса.

При замене компонентов возникает задача их функционального сопоставления Пара компонентов $Comp_1$ и $Comp_2$ при сопоставлении могут иметь следующие свойства.

Определение 3.4. 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 3.5. 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 3.6. 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 (3.3) 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:

$$CImp^j = (ImpName^j, ImpFunc^j, ImpSpec^j), \quad (3.5)$$

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 ТД для параметров, которые передаются, или для параметров, которые возвращаются. По этим сигнатурам и типам данных сопоставляются реализации и интерфейсы в виде описания собственных методов, обеспечивающих процесс связывания. В отличие от объектно-ориентированного и других подходов связывание в компонентном программировании происходит на заключительных процессах построения компонентной программы, а иногда и во время выполнения программы (что характерно для динамических интерфейсов в системе CORBA).

Связь компонентной и объектной моделей

Связь компонентной модели с объектной ОМ прослеживается с помощью следующих процессов.

В процессе 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 $ImpFunc^j$; $Iins_k^{ij}$ – уникальный идентификатор экземпляра компонента.

Пусть существует некоторая объектная система, представленная диаграммой классов:

$$Osyst = (Oclass, G), \quad (3.6)$$

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.

Переход от объектной к компонентной структуре и связями компонентов и интерфейсов представлен на рис. 2.9. Здесь показаны внутренняя структура компонентов Comp1 и Comp2, соответствующие интерфейсы. Int1, IntO1 и IntI2, IntIO2 в классе компонентов *Oclass1, Oclass2* и *Oclass3*.

Каждый *Oclass* представляется как модель класса

$$Oclass^i = \{ClassName^i, Method, Field\},$$

где $Oclass = \{Oclass^i\}$ – множество классов; $ClassName^i$ – имя класса; $Method^i = \{Method_j^i\}$ – множество методов; $Field^i = \{Field_n^i\}$ – множество переменных, определяющих состояние экземпляров класса.

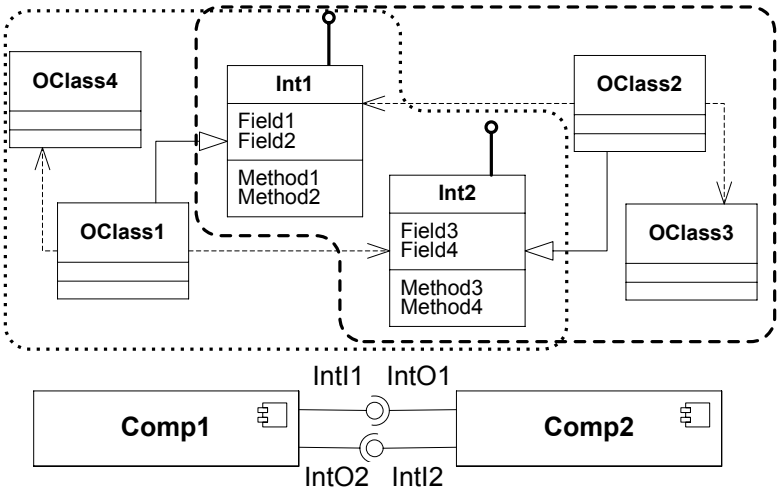


Рис. 2.9. Внутренняя структура связываемых компонентов

Пусть $Pfield^i \subset Field^i$ – множество внешних переменных (public), доступных 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 переменной. Другими словами, эти переменные становятся атрибутами в современных компонентных моделях.

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 $Imethod^i$.

Вместе с *Osyst* рассматрим систему

$$Isyst = (Ifunc, IG),$$

где $Ifunc = \{Ifunc^i\}$ – множество интерфейсов; *IG* – интерфейсный граф, идентичный графу *G*.

Класс $Oclass^i$ порождает свои экземпляры (объекты)

$$Obj_k^i = \{ObjName_k^i, Method^i, Field^i\},$$

которым в системе *Isyst* отвечают 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), сформируем 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 в процессе развертывания, а не на ранних стадиях, как это нужно для ООП. Во-вторых, экземпляр объектного класса порождается его описанием и не может содержать элементов больше, чем сущест-

вует в самом классе или 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.

3.2. Модели разработки систем из компонентов

Компонентная среда *CE* (Component Enviroment System) для разработки ПС имеет вид:

$$CE = (NameSpace, IntRep, ImpRep, CServ, CServImp), \qquad (3.7)$$

где $NameSpace = \{CName^m\}$ – пространство имен, представляющее собой множество имен компонентов в среде; $IntRep = \{IntRep^i\}$ – репозиторий интерфейсов, который содержит интерфейсы компонентов среды; $ImpRep = \{ImpRep^j\}$ – репозиторий 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.

Элемент репозитория интерфейсов определяется как пара:

$$IntRep^i = (CInt^i, CName^m),$$

где $CInt^i$ – интерфейс компонента; $CName^m$ – имя компонента, который реализует этот интерфейс.

Элементом среды может быть каркас. Под ним понимается высокоуровневая абстракция проекта ПС, в которой функции компонентов отделены от задач управления ими. Каркас соединяет множество взаимодействующих между собой объектов в некоторую интегрированную среду для достижения определенной конечной цели. В зависимости от специализации каркас называют белым ящиком или черным ящиком.

Взаимодействие компонента с сервером может осуществляться через стандартизированные интерфейсы контейнера, который руководит порождаемыми компонентами и их экземплярами с соответствующей функциональностью. В общем случае внутри контейнера может существовать произвольное количество экземпляров-реализаций, каждая из которых имеет уникальный идентификатор.

Введем понятие нейтрального (нулевого) элемента компонентной среды, называемой каркасом (framework):

$$FW = (\varnothing, \varnothing, \varnothing, CServ, CServImp),$$

где пространство имен, репозитории 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 $CServ = \{CServ^r\}$ относятся сервисы, которые необходимы при организации построения и функционирования компонентных сред, а также управления компонентными конфигурациями. Набор сервисов может быть таким.

1. Сервис наименования – Naming обеспечивает возможность поиска компонентов в распределенной среде с учетом пространств имен.
2. Сервис связывания – Binding, предназначается 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 моделе асинхронных транзакций.

В реальных средах могут быть и другие системные сервисы, например, сервис управления событиями, каталогами и т. д. Такие сервисы 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 являются обязательными для любой компонентной модели и ее реализации. Они отражают реализацию базовых функций управления компонентными средами:

1) поиск компонентов;
2) доступ к их ресурсам;
3) организация обмена информацией между компонентами;
4) динамическое управление функционированием совокупности компонентов или каркасов.

Определение 3.7. 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 3.8. Каркас FW_1 совместимый с каркасом FW_2, если существует отображение $SMap: CServ_1 \text{ –> } CServ_2$, 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.

Таким образом, 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 – это именование объектов среды и наоборот. Аналогичная ситуация должна существовать и для других системных сервисов, так как любая компонентная среда определенного типа использует те же сервисы, а их связи определяют 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.

3.3. Операции внешней, внутренней и эволюционной алгебры

Рассмотрим множество компонентов *Cset,* операции внутренней компонентной алгебры и операции алгебры компонентной среды для каждого элемента множества $CSet^*$. Получим, что множество *CSet* и все допустимые операции над всеми элементами *Cset* являются замыканием. Для множества $CSet^*$ можно построить новую компонентную среду CE^{*c}, которая будет является расширением предыдущей. Подобный процесс расширения может неоднократно повторяться с получением новых компонентов и базовых компонентных сред. Условия распределенной среды и проблемы возникновения , а также 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.

Операция добавления компонента к компонентой среде обозначается $\oplus$. Добавление компонента к среде выполняется согласно правилам

$$Comp \oplus CE_1 = CE_2,$$
$$CE_2.NameSpace = \{Comp.CName\} \cup CE_1.NameSpace,$$
$$CE_2.IntRep = \{Comp.(CInt_i, CName)\} \cup CE_1.IntRep,$$
$$CE_2.ImpRep = \{Comp.(CImp_j, CName)\} \cup CE_1.ImpRep.$$

Аксиома 3.1. Операция добавления компонента к компонентной среде коммутативна: $Comp \oplus CE = CE \oplus Comp$.

Аксиома 3.2. Операция добавления компонента к компонентной среде ассоциативна:

$$Comp_1 \oplus (CE \oplus Comp_2) = (Comp_1 \oplus CE) \oplus Comp_2.$$

Утверждение 3.1. Каждая непустая компонентная среда CE может быть представка так:

$$CE = Comp_1 \oplus Comp_2 \oplus ,..., \oplus Comp_n \oplus FW .$$

Операция удаления компонента из компонентной среды обозначается знаком $\Diamond$ и подчиняется следующим правилам:

$$CE_1 \Diamond Comp = CE_2,$$
$$\exists CName_m: CName_m \in CE_1.NameSpace \wedge (CName_m = Comp.CName) \Rightarrow$$

$$CE_2.NameSpace = CE_1.NameSpace \setminus \{Comp.CName\} \wedge CE_2.IntRep = CE_1.IntRep \setminus \{(\forall i: IntRep^i.CName = Comp.CName)\ IntRep^i\} \wedge CE_2.ImpRep = CE_1.ImpRep \setminus \{(\forall j\ \&ImtRep^j.CName = Comp.CName)\ ImtRep^j\}.$$

Теорема 3.2. Для любого компонента *Comp* и компонентной среды *CE* выполняется равенство

$$(Comp_2 \oplus CE) \lozenge Comp = CE.$$

Доказательство. Представим операцию добавления компонента к среде так:

$$Comp \oplus CE = (\{Comp.CName\} \cup CE.NameSpace\},\ \{Comp.(CInt_i,\ CName)\} \cup CE.IntRep,\ \{Comp.(CImp_j,\ CName)\} \cup CE.ImpRep,\ CSe,\ CSeIm) = CE.$$

Тогда . операция удаления компонента из среды имеет вид

$$CE' \setminus Comp = (CE'.NameSpace \setminus \{Comp.CName\},\ \{CE'.IntRep \setminus Comp.(CInt_i,\ CName)\},\ CE'.ImpRep \setminus \{Comp.(CImp_j,\ CName)\},\ CSe,\ CSeIm) = CE.$$

Операция замены одного компонента для другой среды обозначим знаком "–" представим в виде

$$CE.NameSpace(Comp_1) - Comp_2 = (CE \lozenge Comp_1) \oplus Comp_2.$$

Внешняя компонентная – двуосновная алгебра, которая включает в себя операции над компонентами и компонентной средой, имеет вид

$$\varphi = \langle CSet,\ CEnv, \Omega \rangle,$$

где $CSet = \{Comp_1,\ldots,Comp_n\}$ – множество существующих компонентов, построены на модели M_{Comp}, $CEnv = \{CE_1,\ldots,CE_n\}$, на множестве компонентных сред, $\Omega = \{\oplus, \lozenge, -\}$ – множество операций над компонентами.

Операция объединения компонентных сред (обозначается $\cup$) и выполняется согласно следующих правил:

$$CE_1 \cup CE_2 = CE_3\ ,$$
$$CE_3.NameSpace = CE_1.NameSpace \cup CE_2.NameSpace,$$
$$CE_3.IntRep = CE_1.IntRep \cup CE_2.IntRep,$$
$$CE_3.ImpRep = CE_1,\ ImpRep \cup CE_2.ImpRep.$$

Теорема 3.3. Операция объединения компонентных сред ассоциативна*:*

$$(CE_1 \cup CE_2) \cup CE_3 = CE_1 \cup (CE_2 \cup CE_3).$$

Доказательство

$$\forall Comp \in (CE_1 \cup CE_2) \cup CE_3 \Rightarrow Comp \in CE_1 \vee Comp \in CE_2 \vee Comp \in CE_3;$$
$$\forall Comp \in CE_1 \cup (CE_2 \cup CE_3) \Rightarrow Comp \in CE_1 \vee Comp \in CE_2 \vee Comp \in CE_3.$$

Аксиома 3.3. Операция объединения компонентных сред коммутативна:

$$CE_1 \cup CE_2 = CE_2 \cup CE_1.$$

Утверждение 3.4. Для любых компонентных сред выполняется:

$$CE \cup FW = FW \cup CE = CE.$$

Утверждение 3.5. Для любой компонентной среды $CE \cup FW = FW \cup CE = CE$.

Утверждение 3.6. Для двух произвольных компонентных сред CE_1, CE_2 и компонента Comp всегда выполняется:

$$Comp \oplus (CE_1 \cup CE_2) = (Comp \oplus CE_1) \cup CE_2 = (Comp \oplus CE_2) \cup CE_1.$$

Утверждение 3.7. Для любого компонента *отр* и компонентной среды всегда выполняется соотношения: $Comp = CE$.

Определение 3.5. Модель компонентной программы M_{nc} состоит из множества КПИ, функций (объектов), интерфейсов и данных.

Условие целостности компонентной программы заключаются 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 $Cont_{12}^{im} = (CInt_1^u, CInt_2^m, IMap_{12}^{im})$, входящих 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.

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:

$$\varphi_I = \{ CSet, CESet, \Omega_1 \}, \qquad (3.8)$$

где $CSet = \{Comp_n\}$ – множество компонентов, каждый из которых представлен моделью (3.3); $CESet = \{CE_n\}$ – множество компонентных сред, каждое из которых описывается выражением (3.4); Ω_1 – множество операций внешней алгебры.

В состав элементов множеств входят: $Comp$ – компонент, CE_1, CE_2, CE_3 – компонентные среды. К множеству операций Ω относятся операции $\{\oplus, \setminus, \cup\}$ обработки элементов множества компонентов. Формальное определение базовых операций над компонентами 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 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 ($Comp \oplus CE$) \ $Comp = CE$. 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 имеет вид*:* $CE_2 = Comp_2 \oplus (CE_1 \setminus Comp_1)$.

Условие целостности компонентной системы заключается в существовании для каждого компонента $Comp_1$ из CE, имеющего исходный интерфейс $CInt_1^u$, компонента $Comp_2$ с соответствующим входным интерфейсом $CInt_2^m$, а контракт $Cont_{12}^{im} = (CInt_1^u, CInt_2^m, IMap_{12}^{im})$ входит в состав множества $Cont$.

Процесс выполнения компонентной программы начинается с развертывания компонентов с помощью контрактов в соответствии с функциональными требованиями к ней.

Теоретический аспект внутренней алгебры компонентов

Компоненты внешней компонентной алгебры представляют собой целевые объекты, которые обрабатываются в компонентной среде. Однако модель компонента имеет собственную структуру и поэтому целесообразно рассмотреть операции над отдельными элементами, которые называются операциями изменения или эволюционного развития.

Суть этих операций – изменение имен, множества интерфейсов, реализаций, отношений и связей между этими множествами, а также преобразование структуры и функций. Особый интерес представляет собой множество операций эволюции, которые обеспечивают целостность понятия компонента, состоящее в условиях определения и существования компонента после изменения [89].

К таким операциям относятся:

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 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 для практического применения компонентного подхода (например, JAVA, CORBA, COM), как правило, требуют, чтобы каждая реализация компонента имела интерфейс. С целью получения реализации системы с большей функциональностью потребуются дополнительные интерфейсы. А это, в свою очередь, означает, что новая реализация компонента может иметь избыточный характер, когда добавляется новая

функциональность с обязательным определением интерфейса. И с другой стороны, в случае добавления нового интерфейса необходима реализация. В этом и состоит суть условия целостности.

Другими словами, среди множества компонентов существует множество интерфейсов и множество реализаций:

$$CInt = \varnothing \text{ и } CImp = \varnothing,$$

которые могут быть пустыми множествами. Назовем их нулевым компонентом или шаблоном компонента: *TComp* = (*Template* ∅, *CFact* ∅, *CServ*).

Условие целостности компонента выполняется для шаблона, в котором множество входных интерфейсов – пустое множество. И независимо от наличия или отсутствия реализаций компонентов это выражение имеет истинное значение.

Отметим, что такой компонент фактически является основой средств автоматизации создания компонентов, где разработчик может, в основном, сосредоточиться на интерфейсах и функциональности будущего компонента, на средства управления экземплярами – на взаимодействии с системными сервисами. Оформление компонента как целостной структуры обеспечиваются в инструментальной среде с использованием реализации функциональности компонента и интерфейса.

Внутренняя компонентная алгебра имеет вид:

$$\varphi_2 = \{CSet, CESet, \Omega_2\},$$

где *Cset = {OldComp, NewComp}* – множество старых *OldComp* компонентов в системе и множество новых компонентов *NewComp,* вновь разработанных или некоторого преобразованного старого компонента к новому *NewComp;*

OldComp = (*OldCName,OldInt, CFact, OldImp, CServ*), включающий интерфейсы, реализации в серверной среде;

NewComp = (*NewCName, NewInt, CFact, NewImp,CServ*), включающий интерфейсы, реализации этого компонента, как необходимые элементы любого компонента, в том числе и нового компонента в серверной среде;

$$\Omega_2 = \{addImp, addInt, replInt, replImp\},$$

где *add imp* – операции добавления реализации; *addInt* – добавления интерфейса; *replImp* – операция замещения реализации компонента, *replImp* – замещения интерфейса компонента.

Операция добавления реализации и интерфейса компонента. Особенность операции *addImp* заключается в том, что множество интерфейсов компонента расширяться за счет добавления новых входных интерфейсов, связанных с реализованной дополнительной функциональностью нового компонента реализации.

Пусть имеем дополнительное множество исходных интерфейсов:

$$NewIntOs = \{NewIntOsq\}.$$

В частном случае *NewIntOs* = ∅, если не требуется добавлять дополнительную реализованную функциональность.

Рассмотрим две разновидности этой операции: *добавление AddOImp* существующего интерфейса и нового интерфейса:

$$NewComp = AddImp(OldComp, NewCImps,NewCIntOs)$$

и представление семантики $NewInt = OldInt \cup NewIntOs$

$$NewCImp = OldCImp \cup \{NewImps\}$$
$$(\exists\ OldIntt \in\ OldCIntI)\ Provide(OldInt) \subseteq NewImps$$

где *NewCImps* – реализация, которая добавляется; *OldCInt* – множество существующих интерфейсов.

Условие целостности компонента выполняется автоматически, потому что множество входных интерфейсов остается прежним. Из целостности старого компонента вытекает целостность нового компонента.

Операция *AddImp* является ассоциационной и коммутативной над множествами компонентов. Доказательство этих фактов вытекает из анализа множеств интерфейсов и множества реализаций, которые входят в состав соответствующих множеств компонентов.

Вторую разновидность операции добавления реализации представим как *AddNImp* в форме

$$NewComp = AddNImp(OldComp,NewCImps,NewCIntOs)$$

и семантикой $NewCInt = OldCInt \cup NewCIntOs$

$$NewCImp = OldCImp \cup \{ NewCImps\}.$$

где *NewCImps* – реализация, которая добавляется к множеству реализаций.

Как и предыдущая, данная операция обеспечивает целостность компонента и обладает свойством ассоциативности и коммутативности.

Операция замены существующей реализации новой *ReplImp* имеет вид

$$NewComp=ReplImp(OldComp,NewCImps,NewCIntOs\ OldCImpr,OldCIntOr)$$

с семантикой.

Если справедливо, что

$$(\forall OldCIntt \in OldCInt)\&(Provide(OldCIntt) \subseteq OldCImpr) =>$$
$$(Provide(OldCIntt) \subseteq NewCImps) \vee ((\exists OldCImpj \in$$
$$(OldCImp \setminus \{OldCImpr\})) \ \&\ Provide(OldCIntt) \subseteq OldCImpj)$$

то $NewCInt = OldCInt \cup NewCIntOs \setminus OldCIntOr;$

$$NewImp = OldImpl \cup \{NewCImps\} \setminus \{OldCImpr\},$$

где *NewCImps* – реализация, которая добавляется; *NewCIntOs* – множество дополнительных исходных интерфейсов для реализации, которая добавляется; *OldCImpr* – реализация, которая замещается; *OldCIntOr* – множество исходных интерфейсов, связанных с реализацией, которая замещается.

Лемма 3.1. Операция замещения реализации компонента новой семантикой сохраняет условие целостности компонента.

Для любого входного интерфейса, создаваемого компонентом, кроме интерфейсов, соответствующей реализации, которая замещается *OldCImpr* – условие целостности выполняется. Для интерфейсов, отвечающих реализации *OldCImpr,* справедливо, что $Provide(OldCIntt) \subseteq NewCImps$ существует соответствующая реализация базового компонента. Объединяя эти два случая, получаем, что для любого входного интерфейса создаваемого компонента выполняется условие целостности.

Операция *расширения* существующей реализации семантической эквивалентна операции замещения, где исходная реализация замещается, а расширенная – добавляется. Потому нет необходимости вводить отдельную операцию.

Операция добавления интерфейса. Исходный интерфейс может добавляться путем замены или существующей реализации или новой реализации. Операция добавления исходного интерфейса – это составная операция добавления реализации.

Операция добавления нового входного интерфейса *AddInt* имеет вид:

$$NewComp = AddInt(OldComp, NewCIntIq)$$

с семантикой согласно следующему правилу.

Если справедливо, что

$$(\exists\ OldCImps \in OldCImp)\ \&\ (Provide(NewCIntIq) \subseteq OldCImps),$$

то $NewCInt = OldCInt \cup \{NewCIntIq\}$, $NewCImp = OldCImp$, где *NewCIntIq* – новый интерфейс.

Лемма 3.2. Операция добавления интерфейса с заданной семантикой сохраняет условие целостности компонента.

Пусть выполняется условие целостности для базового компонента, т.е. для каждого из входных интерфейсов существует соответствующая реализация. Эта предпосылка требует наличия реализации и для нового интерфейса. Потому расширенное множество входных интерфейсов целостности компонента истинно для созданного компонента с сохранением целостности.

Для операции расширения существующего интерфейса в отличие от операции расширения существующей реализации, не требующей своей сохранности для структуры компонента, все существующие входные интерфейсы должны сохраняться. Операция *CFact* носит комплексный характер, потому что дополнительные методы, которые входят в состав интерфейса, требуют реализации со стороны контейнера.

Выше была отмечена важность построения модели сервиса для определения совместимости компонентных сред. Эта модель детализирует *CServ* с указанием сервисов, которые необходимы для поддержки функционирования компонентов и компонентных сред в рамках парадигмы компонентного программирования, которая обеспечивает формальное определение дополнительных статических и динамических свойств компонентных сред.

Эволюционная компонентная алгебра

К внешней и внутренней компонентной алгебре добавим эволюционную алгебру. В результате получим общую компонентную алгебру компонентного программирования**:**

$$\sum = \{\varphi_1, \varphi_2, \varphi_3\},$$

где $\varphi_1 = \{CSet, CESet, \Omega_1\}$ – внешняя алгебра, $\varphi_2 = \{CSet, CESet, \Omega_2\}$ – внутренняя алгебра, $\varphi_3 = \{Set, CESet, \Omega_3\}$ – алгебра эволюции компонентов.

Алгебра эволюции φ_3 включает операции $\Omega_3 = \{O_{refac}, O_{Reing}, O_{Rever}\}$,
где O_{refac} – операции рефакторинга, O_{Reing} – операции реинженерии и O_{Rever} – операции реверсной инженерии компонентов.

Данная алгебра содержит множество компонентов и специальные операции из Ω_3. Эта алгебра объединяет внешнюю и внутреннюю компонентные алгебры, а также модель системных сервисов. Кроме непосредственного объединения возможностей отмеченных алгебр, она позволяет формализовать дополнительные задачи для реализации сложных систем:

1) рефакторинг, обеспечивающий построение компонентного приложения и изменений в нем;

2) реинженерия и реверсная инженерия, обеспечивающие конфигурацию систем с принципом транзакционности;

3) замена и переименование компонентов и/или их интерфейсов и добавление нового компонента, который дополняет существующую компонентную среду.

Операции данной алгебры обеспечивают изменение компонента и интерфейса. Они входят в состав следующих моделей эволюции компонентов.

Модель рефакторинга компонентов*:*

$$M_{refac} = \{O_{кefac}\ ,\ \{CSet = \{NewComp^n\}\},$$

где $O_{refac} = \{AddOImp,\ AddNImp,\ ReplImp,\ AddInt\}$ – операции рефакторинга*,*
пара $(CSet,\ O_{refac})$ – элемент компонентной алгебры эволюции.

Операция рефакторинга носит комплексный характер, так как методы, входящие в этот интерфейс, требуют реализации компонента в составе контейнера. Поэтому реализация этой операции связана с существованием нового типа контейнера. Согласно классификации методов рефакторинга, операция расширения интерфейса управляет экземплярами компонентов и относится к расширенной классификации [68, 196].

Аналогичные суждения справедливы и при анализе 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.

Множество операций рефакторинга *AddOImp, AddNImp, ReplImp, AddInt* включают в себя операции добавления и замещения реализаций компонентов и интерфейсов. Другими словами, пара (*CSet, Refac*) входит в компонентную алгебру и включает в себя методы обеспечения рефакторинга.

Утверждение 3.2. Алгебра рефакторинга компонентов $\sum^{refac} = (CSet,\ Refac)$ является полной и достаточной для изменений.

В соответствии с теоремой 3.2 результатом операции рефакторинга или операции суперпозиции выступает компонент. Это свидетельствует о связи внешней компонентной алгебры с алгеброй рефакторинга. Множество *CSet* состоит из компонентов репозитория 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:

$$CE=Comp_1 \oplus AddInt(AddNImp(Comp_2, NewCImp^s,\ NewCIntO^s),\ NewCIntI^q) \oplus FW.$$

Модель реинженерии компонентов*:*

$$M_{Reing} = \{O_{Reing}\ ,\ \{CSet = \{NewComp^n\}\},$$

где $O_{reing} = \{rewrite,\ restruc,\ adop,\ supp,\ conver\}$ – операции реинженерии, пара $(CSet,\ O_{Reing})$ – элемент компонентной алгебры эволюции.

Алгебра реинжиниринга $\sum^{Reing} = (CSet,\ Reing)$ используется при условии нарушения целостности компонентов или изменения функциональности.

Компонентная алгебра реинжиниринга компонентов Σ^{reeng} = (*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 реверсной инженерии компонентов*:*

$$M_{Rever} = \{O_{Reve}, r, \{CSet = \{NewComp^n\}\},$$

где O_{Rever}= { restruc, design, restruct}– операции реверсной инженерии; пара (*CSet,* O_{Reing}) – элемент компонентной алгебры эволюции.

Алгебра реверсной инженерии Σ^{Rever} = (*CSet, Rever*) используется при условии нарушения целостности компонентов или изменения функциональности.
В реверсной инженерии могут использоваться операции реинженерии $Reeng \subset Revers$. Особенность множества *Revers* состоит в том, что ее операции не определены полностью, а лишь частично. Это связано с тем, что реверсная инженерия обозначает полную перестройку программной системы из компонентов в некоторой среде, включая изменение отдельных показателей качества компонентов, которые могут изменяться в зависимости от содержания смысла функции и применяемых операций изменения компонента (классификационный признак в системе классификации множества операций *Revers*).

Семантика операций *Reeng* может определять трансформацию не только целевого компонента, но и других связанных компонентов. Например, при изменении сигнатуры входного интерфейса необходимо одновременно изменить другие компоненты, которые содержат обращения к методам этого интерфейса.

Таким образом, операции эволюционной алгебры обеспечивают изменения конкретного компонента и относятся к классу соответствующих операций в других методах программирования, которые являются операциями типа *эволюции*. Базовая суть этих операций – изменение имен, интерфейсов, реализаций и связей между элементами множества алгебр, а также преобразования структуры и функций компонентов.

Среди множества операций эволюции особый интерес представляют те, которые обеспечивают *целостность* компонента.

К операциям изменения интерфейса относятся:

1) добавление новой реализации для существующего интерфейса;
2) добавление нового интерфейса и новой реализации;
3) объединение существующих интерфейсов и реализаций в структуру.

Множество элементов модели компонентов и множество операций рефакторинга, реинженерии и реверсной инженерии определяют сущность изменения эволюции этих элементов.

Ниже приведены операции компонентной алгебры, которые реализованы комплексе ИТК ИПС:

link PS ($Comp_A$, $Comp_B$, $Comp_C$) – сборка компонентов $Comp_A$, $Comp_B$, $Comp_C$;

config SPS ($Comp_A^{l1}$, $Comp_B^{l1}$, $Comp_A^{l1}$ ($CInt_A^{idl}$, $CInt_B^{idl}$, $CInt_C^{idl}$) – конфигурированием $Comp_A$, $Comp_B$, $Comp_C$ в языке l_i и интерфейсов в IDL;

redoing $x, y \Rightarrow BD$ – передача данных x, y BD с принятым форматом;

redo TD (x,y) – передача данных с преобразованием значений данных;

interconect PS ($Comp_A$, $Comp_B$, $Comp_C$, $CInt_A$, $CInt_B$, $CInt_C$) – связь A, B, C с интерфейсом;

redevelop PS ($CInt_A$, $CInt_B$) – модификация правил связи $Comp_A$, $Comp_B$.

Операции эволюции компонентов в ИТК включают следующие:

makeaway PS ($Comp_A$) – удалить из PS компонент $Comp_A$;

add PS ($Comp_A$, $Comp_C$) – добавить $Comp_A$, $Comp_C$ к PS;

insert $F \Rightarrow PS$ – вставить F в PS;

redact $Comp_A$ (PS) – редактировать компонент $Comp_A$ в PS, а также .заменить имя компонента $Comp_A$ на другое имя – $CName_C$;

3.3. Объектно-компонентный метод

Типы отношений между компонентами. Пусть выражение $Comp_n$=($Cname_n$,Ci_n , $Cfact_n$, $Cimp_n$,$Cserv$) определяет компоненты, которые имеют следующие типы отношений.

Отношение наследования двух компонентов определяется установлением отношения наследования для входных интерфейсов (как 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 через контракт $Cont_{12}^{im}$, который описывается выражением $Bind$ ($IIns_{1k}^{ij}$, $IIns_{2p}^{mq}$, $Cont_{12}^{im}$).

Объектно-компонентный метод

Между объектным и компонентным представлениями программ существует неоднозначность, которая порождается тем, что определенный компонент может

иметь реализации для нескольких интерфейсов *Isyst*. В случае если каждый из интерфейсов реализуется отдельным компонентом, существует единое эквивалентное отображение между объектными и компонентными представлениями.

Пример объектного и компонентного описания сложной программы показан на рис. 2.10 четырьмя классами объектов: O_1, O_2, O_3, O_4.

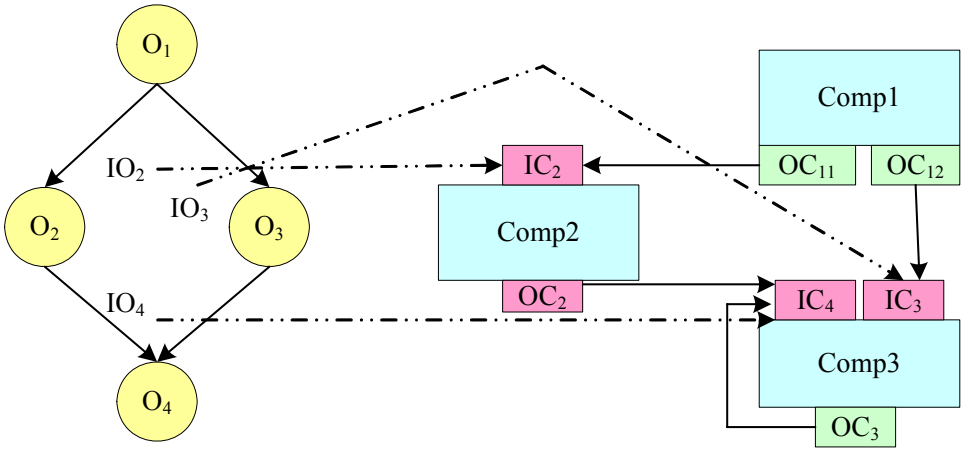


Рис. 2.10. Структуры программы в ОКМ

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* существуют компонентные интерфейсы IC_2, IC_3, IC_4 (штрихпунктирные 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.

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 процессы определения объектов начинаются с отдельных сущностей ПрО и заканчиваются заданием компонентов с учетом их поведения в компонентной среде. Объектная модель отображается в компонентную модель путем формирования функциональных интерфейсов и распределения их между конкретными компонентами на основе внешней и внутренней компонентной алгебры.

Основные теоретические и прикладные положения ОКМ приведены на рис. 2.11.

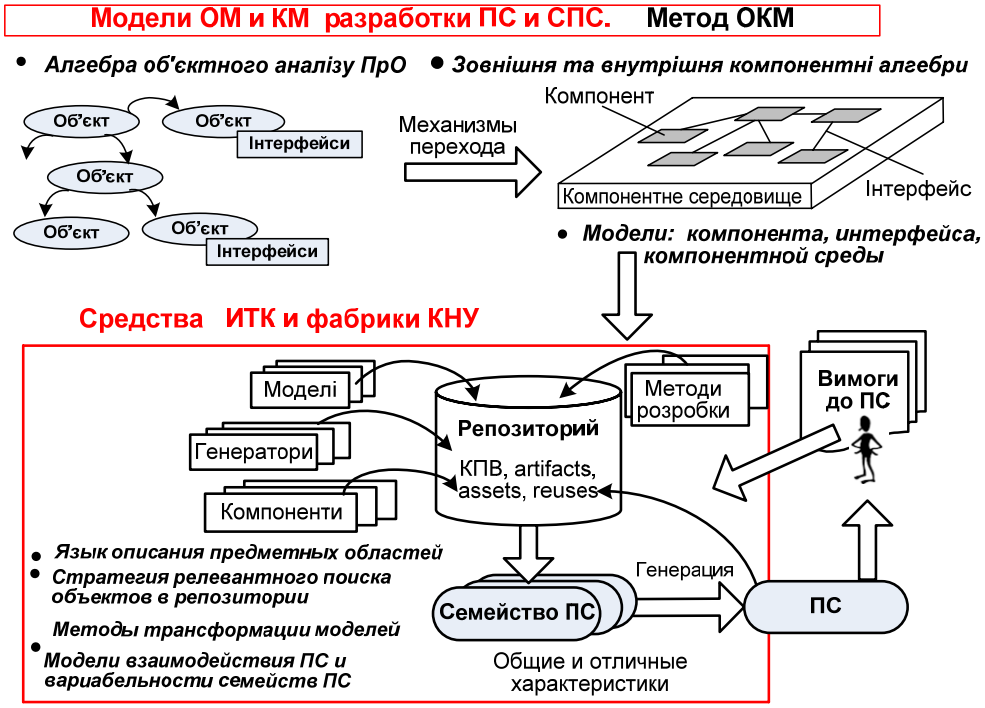


Рис. 2.11. Проектная среда разработки сложных ПС по ОКМ

На рисунке представлены:

1) концепции, терминологии и методы композиции/сборки, которые объединены в единую схему понятий метода ОКМ со строгим определением причинно-следственных связей между ними;

2) базовые элементы теории – модели компонента и компонентной среды, внешняя и внутренняя компонентные алгебры и отношения;

3) задачи реализации программных объектов, операциями внешней и внутренней компонентных алгебр;

4) определение компонентной среды, в которой находятся репозитории компонентов, интерфейсов, где размещаются готовые ресурсы (объекты и компоненты), элементы базовой теории, а также методы и средства разработки;

5) метод сборки компонентов для взаимодействия в распределенной среде;

6) механизмы перехода от объектной модели к компонентной;

7) модели интерфейса, компонента, среды;

8) репозиторий готовых компонентов и интерфейсов;

9) языки описания компонентов и сборки из них ПС и семейств систем.

Предложенная парадигма включает в себя компонентную алгебру, которая является оригинальной и не имеет прототипа. С помощью операций этой алгебры выполняется сертификация КПИ, их хранение и поиск в репозитории для ис-

пользования в сборочной технологии, а также сборка КПИ в сложные программные структуры. Отдельные аспекты парадигмы комонентного программирования реализованы при создании прикладных систем для Национальной академии наук Украины (например, система обслуживания зарубежных командировок – Зак). Эта система входит в состав ИТК в разделе "Прикладные системы".

3.4. Типизация компонентов. Корректность сборки компонентов

Компоненты из множества компонентов С являются типовыми:

1) простые компоненты, которые реализуют некоторую функцию ПрО;

2) готовые компоненты (КПИ, beans-компоненты в языке JAVA и др.);

3) сложные компоненты (каркас, паттерны и др.) со схемой конфигурирования компонентов;

4) интерфейсные посредники, которые содержат описание параметров по обеспечению связей между разными функциональными компонентами;

5) сервисные элементы, которые поддерживают разные системные действия по организации передачи параметров и данных, представленных в интерфейсах или контрактах, а также управления выполнением компонентов;

6) средства развертывания компонентов через конфигурационный файл в соответствующей гетерогенной среде.

Таким образом, компонентное программирование – это самостоятельный стиль программирования со своей концептуальной базой, теорией, методологией и инструментальными средствами. Оно дополняет объектную парадигму средствами сборки или композиции готовых КПИ, упрощает процессы разработки и сопровождения ПС за счет:

1) формализации и упорядоченности отдельных компонентов и систем с КПИ, уменьшает время разработки и улучшает качество и надежность ПС из компонентов;

2) механизмов стандартного описания интерфейсов компонентов для передачи данных компонентам, которые будут выполняться в интегрированной среде посредством соответствующих общесистемных сервисов и др.;

3) упрощение процессов построения ПС с помощью формализованных моделей, интерфейсов и использования метода конфигурирования для сбора компонентов из множества *С* в разные структуры ПС с вариантами их выполнения.

Компонентная модель ПС конструируется из компонентов, которым сопоставлены реальные функции или методы объектов ПрО для реализации функциональности ПС с соблюдением не функциональных требований. Для одной и той же ПС может существовать множество компонентных моделей в зависимости от концепций проектирования и использования КПИ. Модель компонентной ПС–СМ состоит из совокупности моделей $M_{ПС}$ или одной из них с той разницей, что здесь используются готовые КПИ, как ресурсы разного назначения.

На общем уровне представления компонентная модель СМ состоит из совокупности компонентов и их интерфейсов или контрактов для обеспечения взаимодействия между собой отдельных программных компонентов, из которых складывается модель.

Пусть $\prod_{i=1} A_i$ – множество интерфейсов, контрактов, которые определяют функциональность интерфейсов. Каждый *Ai* описывает контракт как клиент-серверное взаимодействие с соответствующими методами и структурами данных. Каждый интерфейс имеет описание интерфейсных данных, *In*, *Out* и *Inout* - (входных, выходных и промежуточных). *In* определяет условия и цель контракта со стороны клиента, *Out* задает реализацию компонента на сервере, а *Inout* - общие свойства интерфейсов *In* и *Out.*

Определение интерфейсов *Ai* для компонентов модели СМ могут группироваться в разные сочетания с учетом семантики характеристик общего и внутреннего типов, что определено для модели системы и компонентов КС.

Произвольная совокупность интерфейсов *Ini, Outj, Inoutij*, где $i \cong$ J, может не входят в исходную совокупность и одновременно определять пары программных компонентов при их сборке в сложную структуру системы.

Для каждой полученной совокупности пар $Ci \cup In = \{Ci, In, Out, Inout\}$ *создается* модель компонента или шаблон, который содержит некоторое множество параметров для задания интерфейсов. По ним осуществляется сопоставление шаблонов и интерфейсов реальных компонентов.

В результате операций объединения имеем множество пар $Ci \cup In$, которое и будет называться множеством контрактов компонентной модели ПС. Для одной и той же программы существует множество таких моделей, которые определяются множествами контрактов (или интерфейсов) со способами разбивки на шаблоны компонентов для последовательного их выполнения.

Общее определение алгебры компонентов ПС на X – множестве объектов, где $x_i \in X$ – произвольный объект некоторой ПрО, которой определяет некоторую функцию ПрО, связанную с другими объектами отношением связи. Этому множеству соответствует множество компонентов C ($C \in X$). Каждый компонент реализует функцию и характеризуется некоторым, присущим ему свойством (например, вариантность, изменяемость и т. п.), которые размечаются в интерфейсах.

Применив к множеству реальных компонентов $c_i \in C$ множество операций $R_1, R_2, ..., R_m$, получим множество, которое будет замыканием множества C', т. е. максимально возможным множеством компонентов, которые можно получить на основе выбранных компонентов из множества *C.*

В реальных процессах конструирования ПС из КПИ рассматривается множество компонентов *C* (или *C'*), для которого справедливы все положения компонентной алгебры при построении компонентных ПС (например, в комплексе ИТК [7]).

Корректность сборки разноязычных компонентов

Лемма 3.3. Для корректности метода сборки компонентов необходимо и достаточно, чтобы:

1) существовали операции интеграции такие, что для заданного графа $G = \{O, I\}$, процесс построения сборочной структуры был конечным;

2) были компоненты во множестве *O* (при условии принадлежности $O_i \in O$), которые реализуют необходимые функции, заданные исходным текстом и интерфейсным посредником;

3) содержались средства преобразования O_i в интегрированной среде.

Доказательство данной аксиомы сводится к доказательству следующих утверждений.

Утверждение 3.3. Существует конечный алгоритм построения программной системы из разнородных компонентов.

Схема алгоритма метода сборки задается графом $G = \{O, I\}$ и используется для создания ПС из компонентов множества O с помощью следующих шагов:

П1. Проверка правильности задания графа $G=\{O, I\}$, результат которой

$$\begin{cases} 0, \text{ если граф задан правильно,} \\ \Delta = 1, \text{ если нарушается порядок параметров или их количество.} \end{cases}$$

П2. Проверка согласованности входных параметров вызовов из интерфейсного посредника осуществляется проверкой принадлежности типов данных в параметрах X, Y пары взаимодействующих компонентов (O_i, Ok_j) в интерфейсах I_{ij}, заданных на графе так:

$$\begin{cases} I_{ij}=1, \text{ если переменные параметров согласованы,} \\ 0, \text{ иначе.} \end{cases}$$

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.).

Утверждение 3.4. Для любой пары компонентов $(O_i, O_j) \in O$ созданная программа из пар $P=\{O_i, I_i, Ok_j, I_j\}$ будет корректной.

Справедливость этого утверждения вытекает из определения интерфейсного посредника, согласно которому для каждой пары компонентов $\{O_i, O_j\}$ существуют посредники I_i, I_j, из которых собирается программа $P = \{\{O_i, I_i\}, \{O_j, I_j\}\}$ с проверкой корректности каждого параметра при входе в вызываемый компонент и при выходе из него.

Так как количество компонентов графа $G\{O, I\}$ конечно и каждая пара является корректной, то получаем корректность программы P, собранной из этих компонентов.

Лемма 3.4. Создание любой программы P из компонентов $C_i \in C$ включает:

1) конечное количество операций преобразования данных;
2) компоненты выдачи диагностических ошибок при анализе параметров вызова компонентов;
3) конечное множество компонентов C.

Конечность процесса построения P на основе графа $G\{O, I\}$ представлена утверждением 3.3. 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 $G=\{O, I\}$ – конечный и будет завершен сборкой корректной P или диагностическим сообщением.

Сборка компонентов может образовывать паттерн, содержащий абстрактный класс – интерфейс. Сложный паттерн – иерархичный объект, удовлетворяющий условию унификации и композиции объектов. Доступ к абстрактным элементам такого агрегатного объекта обеспечивается итерационным путем.

Проблему композиции выполняет также каркас путем обеспечения правильного и надежного взаимодействия 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.

Сборка компонент–компонент обеспечивает непосредственное взаимодействие компонентов через интерфейс на уровне приложения.

Сборка каркас–компонент обеспечивает взаимодействие каркаса с компонентами, при котором каркас управляет ресурсами компонентов и их интерфейсов. Такое взаимодействие осуществляется на системном уровне.

Сборка 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.

Сборка 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 межъязычного интерфейсов, которые обеспечивают взаимодействие в современных сетевых и гетерогенных средах.

Межмодульный интерфейс остается посредником между двумя взаимодействующими компонентами. Он включает в себя совокупность средств и методов сопоставления структур и типов данных ЯП для организации преобразования переданных типов данных, аналогично тому, как это было рассмотрено в разделе 1. 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 проводится сборка компонентов и их взаимодействие в современных средах.

3.5. ЖЦ компонентной разработки ПС

Главные задания ЖЦ – определение, поиск и выбор всех необходимых компонентов, их адаптация к потребностям объединения, обеспечение условий для необходимого преобразования ТД взаимодействующих компонентов, интегри-

рованной среды для ПС. Все это выполняется в следующих процессах: поиск компонентов соответственно описанию интерфейсов; выбор совокупности компонентов, которые обеспечивают необходимую функциональность; адаптация существующих компонентов для требований интегрированной среды; создание новых компонентов, для которых результаты поиска, выбора и адаптации привели к негативным результатам; инсталляция компонентов для потребностей композиции или сборки; определение полной совокупности правил и условий сборки компонентов; конфигурация КПИ в распределенную ПС; тестирование интегрированной среды ПС.

Компонентная алгебра включает в себя операции сборки, которым соответствуют выполнение процессов ЖЦ.

Поиск компонентов. Этот процесс предусматривает существование информационных хранилищ (репозиториев) с описаниями компонентов. Для реализации более качественного и оптимального поиска создается система классификации программных компонентов. В частности, для представления и поиска информации о компонентах может использоваться поисковая машина (типа AltaVista) сети Интернет. Для выполнения этих задач строится информационное обеспечение КПИ на основе их поисковых образов и т. п.

Выбор компонентов. Результаты предыдущего процесса являются четко определенными, если существует хотя бы один компонент для реализации нужного интерфейса или не существует ни одного. В первом случае компонент применяется, во втором – выполняется разработка нового компонента. Но часто результаты поиска могут быть неопределенными. Иными словами, могут существовать несколько компонентов для определенного интерфейса, или некоторый компонент "почти" подходит. Для таких случаев выполняются процедуры выбора и оценки компонентов.

Адаптация компонентов. Если компоненты, которые используются для сборки, не полностью удовлетворяют условиям взаимодействия и функционирования в интегрированной среде, то над компонентами осуществляются операции их эволюции с сохранением их свойств. Посредством этих операций выполняется трансформация возможностей компонентов с одновременным сохранением их основных функциональных, технологических характеристик и соответствующих интерфейсов, условий взаимодействия и особенностей интегрированной среды. Основа механизма такой трансформации – реализация свойств компонентов для совершенствования и расширения их функциональности и других возможностей.

Создание компонентов. Этот процесс выполняется, когда предыдущие процессы поиска, выбора, адаптации для определенных компонентов дают негативные результаты. В этом случае необходимо создавать новые компоненты с заранее определенными интерфейсами и функциональностью. Разработка таких компонентов осуществляется современными языками и средствами. Важно, чтоб все требования к компонентам, их свойства и типичная структура соответствовали требованиям компонентной модели, которая применяется для сборки.

Инсталляция компонентов. По содержанию этот процесс не отличается от инсталляции отдельных компонентов. Единственное различие у этих процес-

сов состоит в том, что определенные компоненты специально созданы самим разработчиком ПС для ее потребностей, а, следовательно, условия и процедуры развертывания он может оптимизировать именно для этой системы. Инсталляция происходит согласно соответствующей документации.

Построение ПС. Главная цель – выполнить все действия по подготовке интегрированной среды к функционированию и создания плана конфигурации компонентов ПС. Для этого выполняются следующие шаги.

1. Компилирование описания интерфейсов и генерации вспомогательных компонентов для потребностей определения модели и генерации компонентов на языке программирования.

2. Компилирование дополнительных компонентов и связь их с другими компонентами. Эти объекты играют роль разъемов, с помощью которых компоненты подсоединяются к интегрированной среде.

3. Фиксация плана компонентной конфигурации в виде перечня компонентов, их связей и зависимостей между ними для среды функционирования.

4. Тестирование созданной конфигурации путем дополнения реальной информацией о расположения компонентов, сервисах и видах протоколов.

Развертывание ПС. Этому процессу соответствует такие подпроцессы:

1) инсталляция отдельных компонентов на определенных компьютерах для целей сборки;

2) управление конфигурационной схемой для сопровождения системы.

Все процессы развертывания выполняются как отдельный процесс.

Оптимизация плана компонентной конфигурации. План компонентной конфигурации определяет абстрактную модель интегрированной среды ПС. Все взаимосвязи и взаимодействия компонентов задаются без учета фактического расположения компонентов. Оптимизация происходит при использовании начальных данных, имеющейся компьютерной мощности, средств коммуникации с пользователем и др. В качестве условий выступают функциональные, технологические, нефункциональные требования к ПС.

Сопровождение ПС. Этот процесс включает в себя модификацию компонентной конфигурации; адаптацию новых компонентов к требованиям и условиям интегрированной среды; анализ отказов функционирования, поиск и исправление ошибок в ПС; тестирование РПС.

Необходимыми условиями для этого процесса является возможность манипуляций с компонентами как отдельными объектами с сохранением свойств и характеристик других частей программной системы. Это достигается благодаря применению систем управления конфигурациями, посредством которых отслеживаются и выполняются все изменения в конфигурации системы.

Адаптация компонентов к условиям среды. Этот процесс по существу и содержанию не отличается от соответствующего процесса сборки. Различия носят непринципиальный характер. К ним, в частности, нужно отнести то, что в случае неудовлетворительной адаптации всегда есть возможность возвратиться к существующему компоненту и ПС без изменений. Кроме этого, сам процесс адаптации может выполняться персоналом пользователя, а не разработчиком ПС.

Каждый из типов объектов может реализовываться отдельно, потому что для каждого из них существуют свои спецификации и требования, а также правила взаимодействия с другими объектами компонентного программирования.

Теория и компонентная технология изготовления ПС из КПИ прошла апробацию в проекте информатизации НАН Украины на примере создания прикладной системы "Обслуживания командировок в НАН Украины". Эта система представлена в ИТК ИПС.

3.6. CASE-средства поддержки компонентов и систем

Компонентный подход реализован в системе MS.NET, JAVA, IBM и др. [40 – 45].

Платформа MS.NET состоит из нескольких основных компонентов:

1) ОС Microsoft (Windows 2007/XP/ME/), как базовый уровень;

2) серверы MS.Net (.Net Enterprise Servers) уменьшают сложность разработки сложных ПС (например, Application Server, Exchange Server, SQL Server);

3) сервисы .Net Building Block Services – это готовые "строительные блоки" для сложных ПС;

4) интегрированная среда Visual Studio.NET (VS.NET) – верхний уровень MS.NET, которая обеспечивает создание сложных ПС.

Подсистема MS.NET Framework обеспечивает построение и выполнение любых приложений. В ее состав входят: общеязычная среда CLR и библиотека классов FCL (Framework Class Library), состоящая из набора классов для работы со строками, числовыми данными и для параллельного вычисления, а также для создания Windows-приложений в среде графического интерфейса и Web Services и с доступом через Интернет. Эта подсистема включает в себя средства управления памятью, ТД, взаимодействием и развертыванием (deployment) приложений в ЯП.

Любой компонент на ЯП трансформируется к общей спецификации типов CTS (Common Type System) для всех ТД ЯП, определяет их взаимосвязи и сохраняет их отображение. Компилятор в ЯП создает файл на языке CIL, который ассемблируется (assembly) в сборный и переносный (Portable Executable) код и используется при переводе на промежуточный язык и машинный (native).

В системе типов .NET выделены две группы типов: типы-значения (value type) и ссылочные типы (reference type), а также механизм отображения типов CTS в типы ЯП и наоборот.

Типы-значения – это статические типы, встроены в CTS, их значения могут занимать память от 8 до 128 байтов. Они не принимают участия в наследовании и копируются при присвоении им значений.

Ссылочные типы задают ссылки на объекты, которые они специфицируют (объектные, интерфейсные типы и указатели), а также механизмы хранения и освобождения памяти.

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. Программы создаются как 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 в пространстве имен имеет место следующее соотношение. Сборка может включать несколько пространств имен и занимать несколько сборок. Сборка может иметь в своем составе как один, так и несколько файлов, которые объединяются в манифест сборки. Манифест содержит метаданные о компонентах сборки, идентификатор автора и версии, сведения о типах и зависимости, а также режим и политику сборки. Метаданные описывают все типы.

Под объекты ссылочного типа память выделяется из "кучи" и освобождается после уничтожения борки мусора .Ссылочные типы включают в себя: типы: объектные (object type); интерфейсные (interface type); указатели (pointer type).

CTS включает в себя все ТД, которые поддерживаются средой выполнения, представляются в форме метаданных .NET. При сборке программы в ЯП .NET используются правила, принятые в CLS. В ней применяются другие библиотеки: CLR (Common Language Runtime), CLS (Common Language Specification), CIIL. Функция CLR состоит в выявлении и загрузке ТД в .NET и выполнении.

Все ЯП оперируют целочисленными данными или данными с плавающей точкой в формате и единственной длиною. Представление строк тоже будет единственным для всех ЯП.

За счет этой системы типов достигается более простая интеграция компонентов и кодов, написанных на разных ЯП. В отличие от COM-технологии, основанной на наборе стандартных типов, представленных в бинарном виде, CLR позволяет выполнять интеграцию любых кодов. Сервисы в CLR предоставлены библиотекой классов (более 1000) и моделью ASP.NET.

Двоичные или бинарные файлы представляются в платформо-независимом "промежуточном языке" Microsoft Intermediate Language (MIIL или IL), который задает промежуточный уровень работы с любыми языками в системе .NET компонент MIIL конвертирует данные в код CPU во время выполнения (just-in-time-JIT). Если программа написана на языке системы Effil, то она будет конвертирована в промежуточный язык. Программы, расположенные на разных компьютерах сети, передают друг другу данные через интерфейсные протоколы. Эти данные преобразуются к формату данных .NET.

Между именами *простых* типов в C# и именами FCL-*типов* существует взаимно однозначное соответствие. FCL-тип указывает на пространство имен, которое содержит объявление соответствующих типов.

Средства сборки 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 [90].

Проекты как средство композиции компонентов. Создание нового проекта состоит в задании конфигурации системы с помощью компонентов JAVA и обеспечения их взаимодействия следующими шагами:

1) скомпилировать разные файлы с разными JAVA-компонентами одной командой;

2) установить основной компонент (класс) в проекте и уникальную конфигурацию для каждого отдельного проекта;

3) установить уникальные типы компиляции, выполнения, отладки и др.

Базовые операция проекта – это создание нового проекта, импорт компонентов из другого проекта, создание новых компонентов с помощью мастера шаблонов, компиляция, выполнение и отладка группы компонентов как единой композиции. Проект обеспечивает разработку, сохранение и корректировку шаблона для поддержки взаимодействия разных типов компонентов в одной задачи.

К шаблонам повторного использования относятся:

1) BlankAntProject, определяющий первоначальный бланк с возможностью подключения классов и пакетов проекта;

2) SampleAntProject предназначен для конфигурации общей схемы проекта в иерархию файлов и корневого узла с добавлением в нее новых компонентов;

3) CastomTask обеспечивает создание нового проекта.

Классы – основа JAVA, порождаются с помощью ключевого слова Extends, после которого указывается тип компонента (например, JApplet). В проектах используют основной класс, с которого начинается выполнение проекта, и вторичный класс. К основному классу относится Class, Main, Empty (пустой класс), как шаблон типа:

1) exception для создания класса, его исключений и сообщений об ошибках, которые могут быть обнаружены в программе;

2) persistence Capable для отображения реляционной схемы БД без подключения к MySQL;

3) interface – шаблон для создания нового JAVA интерфейса, который можно использовать в 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-коллекций (Vector, Stack, Hashtable, Collection, List, Set, Map, Iterator) и класс-утилита (Calendar – работа с массивами и со случайными числами).

Шаблон развертывания представляет собою скрытую часть и необязательную часть абстракции компонента повторного использования в одной или многих средах. Для этого он имеет несколько шаблонов отладки, к которым могут добавляться новые шаблоны интеграции. В некоторых классах КПИ параметры интегрирования в новую среду включаются в интерфейс компонента, который ограничивает способность компонента адаптироваться к этим средам.

Интерфейсы компонентов содержат методы работы с графическими объектами и классы, реализующими 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. В библиотеке AWT содержатся формы контейнеров для раз-

мещения графических элементов интерфейса-пользователя, а также системы классов Abstract Window Toolkit для построения абстрактного окна.

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, использующему 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.

Удаленный вызов 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 применения КПИ типа beans в системе разработаны следующие элементы в JAVA for FORTE:

1) Beans для создания нового компонента и формирования каркаса компонента с простыми свойствами и возможностью автоматического изменения этих свойств;
2) BeanInfo для интеграции beans-компонентов и обеспечения взаимодействия;
3) Customizer для создания панели, на которой размещаются элементы, которые со временем можно использовать для управления конфигурацией beans-компонентов;
4) Property Editor для создания класса, который используется во время проектирования и редактирования свойств beans-компонентов.

Таким образом, в среде системы JAVA содержится большой набор средств поддержки компонентного подхода и решения проблем сборки и взаимодействия их для разных сред, совместимых с системой JAVA, в том числе и MS.Net.

В ИТК реализован принцип взаимодействия компонентов, в языке VS.Net ↔ Eclipse, созданных в среде JAVA и MS.Net с помощью промежуточного модуля и плагина Eclipse (см. Раздел 3).

Глава 4. ГЕНЕРИРУЮЩЕЕ ПРОГРАММИРОВАНИЕ. МОДЕЛИ И МЕТОДЫ

В начале 70-х годов XX ст. Г.Дейкстра ввел понятие семейства программ с общей "семейной" постановкой задач, по которой они порождаются, как программы. Некоторые из них менялись в связи с недостаточно заданными функциями или уточнением постановки отдельных задач семейства. Проблема порождения программ берет свои истоки в синтезе программ, выполняемым по формальной постановке задач ПрО и их спецификации специальными языками.

Работы в области синтеза впервые сформулированы Э.Х.Тыугу [18], развиты С.С. Лавровым (Синтез программ, ж.Ктбернетика, 1982), А.П.Ершовым [33], В.Д Ильиным (Система пророждения программ, М.: Наука, 1989) и др. В результате сформировалось синтезирующее и сборочное программирование, которое стало использоваться в программистских кругах первоначального периода развития технологии программирования. Несколько позднее появился термин генерация программ, который В.Д,Ильин рассматривал как процесс порождения программ по математической постановке задач ПрО. В своей системе порождения программ ГЕНПАК он разработал механизмы порождения целевых программных систем из других формально описанных постановок задач. Эти механизмы он называл преобразованием спецификации программ к некоторому промежуточному или к конечному результату. При изменении или корректировки отдельных постановок функциональных задач процесс преобразования сводился к формированию новых вариантов программ и систем.

Это привело к появлению понятия семейство систем (family system в методологии Product Lines SEI, 2004) и семейство программных систем (СПС) в рамках фундаментального проекта по генеририрующему программированию (2006-2011) [ ]. Под *семейством* ПС понимается совокупность ПС (членов семейства), связанных между собой общим множеством понятий ПрО, их общими характеристиками в модели характеристик (МХ), готовыми КПИ, предикатами их отбора для уточнения вариантов членов семейства, а также операции конфигурирования и изменения членов СПС и семейства программных продуктов (СПП).

Основу семейства составляют готовые reuses и КПИ, которые способствовали развитию идеи сборки модулей, рассмотренной в главе 1 данного раздела. Кроме того, рассмотрены задачи обеспечения качества всех элементов СПС, которые разрабатываются в разными стилями программирования (ментального, компонентного, аспектного, сервисного и др.), стандартизируются и накапливаются как готовые КПИ в различных библиотеках и репозиториях [89–92].

К. Чернецки в работе "Генерирующее программирование" (Generate programming) (ГП) [51] рассматривает СПП как целевую функцию технологии разработки ПП под девизом "от ручного труда к конвейерной сборке". Generate programming – это парадигма разработки ПС, основанная на моделировании групп или отдельных элементов ПС, таких, которые удовлетворяют требованиям к системе, создаваемой из элементов, аспектов, КПИ средствами генерации спецификации программы в промежуточный или конечный продукт. Главный элемент данного программирования – не уникальный программный продукт, созданный из КПИ для конкретных применений, а семейство ПС или конкретные его члены. Для генерации элементов и членов семейства некоторого домена автор предлагает общую генерационную модель GDM (Generative Domain Model).

Модель GDM задается тремя элементами: 1) член семейства; 2) компоненты реализации функций, из которых собираются члены семейства; 3) конфигурационные знания (configuration knowledge) о спецификациях элементов членов семейства, их общих характеристиках, используемых для получения корректного и качественного конечного ПП.

Член семейства отражает знания об изготовлении ПС, его конфигурации, инструментах измерения и оценки элементов, а также о методах тестирования и планирования и пр. Эти аспекты отображают специфику ПрО, многократно используемых КПИ, представленных в активных библиотеках, которые содержат не только базовый код реализации понятий ПрО, но и целевой код, реализованный разными CASE-инструментами.. Фактически компоненты таких библиотек выполняют роль интеллектуальных агентов, ориентированных на решение конкретных задач ПрО.

Используя эти библиотеки, можно конструировать ПС из ее компонентов, а также из специальных метапрограмм, которые осуществляют редактирование, отладку, визуализацию, взаимодействие и др. Кроме того, есть возможность пополнять ее новыми сгенерированными компонентами в рамках отдельных ПС семейства, которые относятся к компонентам многоразового применения. Вместе с тем ЯП компонентов дополняются новыми аспектами, которые расширяют ПрО новыми возможностями [91 – 94].

Цель генерирующего программирования – разработка разных программных ресурсов для их применения в сборке целевого семейства. На его основе сформировались направления:

1) прикладная инженерия – процесс производства конкретных ПС из КПИ и отдельных элементов процесса создания некоторой ПрО;

2) инженерия ПрО – построение семейства ПС путем сбора, классификации, фиксации КПИ и отдельных членов систем с четким выделением их задач.

Основные этапы инженерии ПрО: анализ ПрО и выявление общих и изменяемых характеристик в соответствующей МХ модели. Она устанавливает зависимость между различными членами семейства, а также создает базис для изготовленияч конкретных членов семейства с учетом механизмов изменчивости отдельных элементов в семействе. На основе модели МХ, повторных компонентов знания о конфигурациях генерируется доменная модель семейства.

Функциональные компоненты ПрО представляются объектами, процедурами, модулями, которым необходимы такие свойства, как безопасность, синхронизация и др. Эти свойства, как правило, выражаются фрагментами кода для нескольких функциональных компонентов. Они представляют собой отдельные аспекты в терминологии АОП. Компоненты и аспекты являются элементами реализации отдельных ПС.

В инженерии ПрО используются следующие процессы:

1) корректировка процессов для разработки решений на основе КПИ;

2) моделирование изменчивости и зависимости согласно словарю описания различных понятий, а также правил фиксации их в МХ модели и в справочной информации об объектных, Use Case, взаимодействии и др.

С учетом зависимостей между характеристиками МХ проводится разработка инфраструктуры КПИ – описание, хранение, поиск, оценивание и объединение готовых ресурсов.

При определении членов семейства ПрО используются новые понятия – пространство задач, а в технологии реализации компонентов на основе каркаса конфигураций – пространство решений.

В рамках инженерии ПрО разрабатывается XML-модель, которая обобщает характеристики системы и изменяемые параметры разных частей семейства, а также решения, связанные с группами ПС.

Инструменты. Примером поддержки инженерии ПрО и реализации горизонтальных методов является система DEMRAL [51], предназначенная для разработки библиотек: численного анализа, контейнеров, распознавания речи, графовых вычислений и т. д. Основные виды абстракций этих библиотек ПрО – абстрактные ТД (abstract data types – ADT) и алгоритмы. DEMRAL, которые позволяют моделировать характеристики ПрО в виде высокоуровневой характеристической модели и предметно-ориентированных языков конфигурирования.

Система конструирования RSEB базируется на вертикальных методах, КПИ и ориентирована на использование элементов Use Case при проектировании крупных ПС. Эффект достигается, когда вертикальные методы инженерии ПрО "вызывают" различные горизонтальные методы, относящиеся к разным прикладным подсистемам. При работе над отдельной частью семейства системы могут быть задействованы такие основные аспекты: взаимодействие, структуры, потоки данных и др.

Модель GDM и модель характеристик MX отображают общие понятия и характеристики ПрО, их взаимосвязи, а также знания о конфигурации используемых КПИ в СПС. Для реализации КПИ в мультипарадигме ГП могут применяться разные стили программирования: ООП, компонентное, сервисное, аспектное и др. За основу методологии производства СПС из готовых компонентов в ГП взята концепция продуктовых линий Института SE США (www.sei.com), по которым создаются коммерческие ПП для массового применения.

Аналогичные разработки в технологии индустриального изготовления ПС, членов семейства и СПС из готовых КПИ на технологических линиях (ТЛ) системы сборочного программировании [7]. Эта технология активно развивается в направлении создания линий разработки и сборки КПИ на фабриках программ, в том числе на фабрике КНУ (http://programsfactory. univ.kiev.ua) и за рубежом [57–59]. Она отображают развитие индустрии ПП в современном информационном мире. Более подробно методология построения линий изложена ниже.

4.1. Элементы программных систем и семейство систем

Семейство продуктов (*Product family*) – "группа продуктов или услуг, которые имеют общее управляемое множество свойств, удовлетворяющие потребностям определенного сегмента рынка или вида деятельности" ("*product family /product line* – group of products or services sharing a common, managed set of features that satisfy specific needs of a selected market or mission")..

Программирование СПП и СПС из готовых КПИ – наиболее продуктивный путь разработки сложных ПС. Здесь объекты рассматриваются на логическом уровне проектирования модели ПрО и ПС с переходом к компонентам при физической реализации методов объектов [91 – 94].

Процесс проектирования ПрО в этом программировании – многоуровневый. Он включает в себя анализ ПрО для установления *единой терминологии*, употребляемой разработчиками семейства и его членов, которые реализуют задачи ПрО с общими архитектурными решениями, основанные на КПИ. Их реализация выполняется средствами частичной или полной автоматизированной сборки компонентов методом генерации (конфигурации) или трансформации.

Некоторые члены семейства могут описываться общими языками GPL и DSL. Преимущества DSL над GPL в представлении DSL текстовым, графическим или фреймовым видам с заданием терминологии для всех членов СПС и правил их применения [13]. Процесс проектирования ПС и членов выполняется на базе терминологии, понятия которой размещаются в специальных базах знаний – пространстве проблем и решений.

Для определения модели GDM и членов семейства К. Чернецки вводит пространство задач (problem space), пространство решений (solution space) и базу конфигурации (configuration base) СПС. *Пространство задач* отображает понятия СПС, членов СПС и их общие характеристики в модели МХ или Feature Model, а также функции и задачи, которые описываются GPL (General-Purpose Language) или предметно-ориентированными языками DSL, UML2.

Пространство решений – это компоненты, каркасы, шаблоны и КПИ реализации задач членов семейства СПС. Механизмы, правила описания, генерации компонентов и подбора КПИ для отдельных задач СПС – основные элементы *базы конфигурации.* При этом каркас оснащается изменяемыми параметрами модели, что может привести к излишней его фрагментации и появлению "множества мелких методов и классов". Каркас обеспечивает динамическое связывание аспектов и компонентов в процессе реализации изменчивости между разными приложениями. Образцы проектирования обеспечивают создание многократно используемых решений в различных ПС. Для задания таких аспектов, как синхронизация, удаленное взаимодействие, защита данных и т. д., применяются компонентные технологии ActiveX и JAVABeans, а также новые механизмы композиции, мета программирования и др.

Результаты описания ПрО в этих пространствах объединяет конфигурационная база знаний, в которой сохраняются связи и характеристики (функциональные или не функциональные), заданные в соответствующей модели МХ членов семейства и выполняются операциями конструирования и объединения компонентов в общую ПС или СПС. Иными словами, в ней отображены знания о конфигурации системы в виде абстракций общего и специального назначения, элементах КПИ и знаний о новых компонентах, результатах их тестирования, измерении и оценивании ПС.

4.2. Трансформация и конфигурация программных систем

В ГП предложены методы трансформации и конфигурации ПС из пространства моделей, которые разрабатываются при проектировании приложений предметно-ориентированными языками [91].

Трансформационная модель. Модель приложения или домена, описанная языком DSL (из пространства проблем может превращаться в пространство ре-

шений путем трансформации DSL-спецификации модели в реализацию более простых ЯП (рис. 2.12).

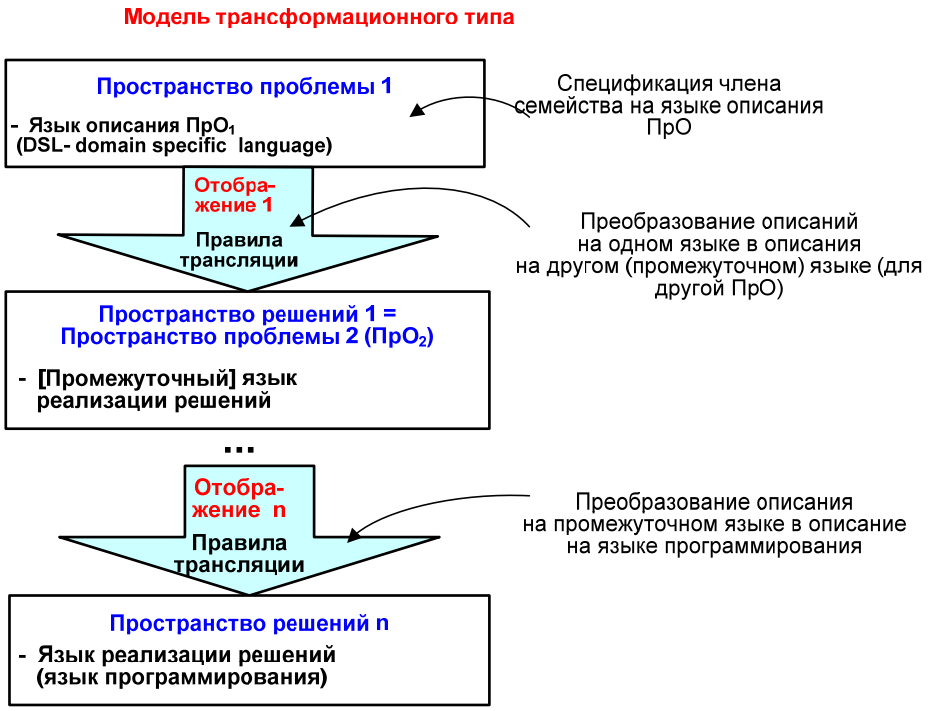


Рис. 2.12. Модель трансформации ПрО

Главный механизм перехода от описания приведенных моделей к исходному результату – трансформация описаний понятий домена в промежуточный язык DSL-пространства решений, а дальше в язык реализации компонентов с учетом платформы, где расположены готовые компоненты и/или новые задачи.

Пространство проблемы состоять из отдельных аспектов проблем. В зависимости от них трансформация может происходить в язык реализации, а в другой DSL-язык (фактически, язык другой ПрО, например, качественных ПС).

Конфигурационная модель. Модель базируется на конструкторских правилах, которые оптимизируют абстракции и характерные черты домена, входящих в GDM (рис. 2.13).

Результатом является конфигурация членов семейства в виде конфигурационного файла. Таким образом, описание специфики домена трансформируется в ЯП компонентов пространства задач для последующей их генерации. При конфигурационном способе отображения пространства проблемы с общими характеристиками и ограничениями фактически соответствует понятиям проблемно-ориентированного языка и множеству компонентов из пространства решений в ЯП.

Данными для конфигурационного способа преобразования пространства задач являются понятия ПрО, их характеристики (свойства) и недопустимые сочетания характеристик, согласованные по умолчанию параметров и зависимостей между

этими элементами. Созданная из них конфугурационная модель отображается по правилам конфигурирования и их оптимизации в пространстве решений.

После отображения это пространство содержит множество компонентов (КПИ, reuse), схемы сборки готовых компонентов, варианты архитектур некоторых членов семейства**.**

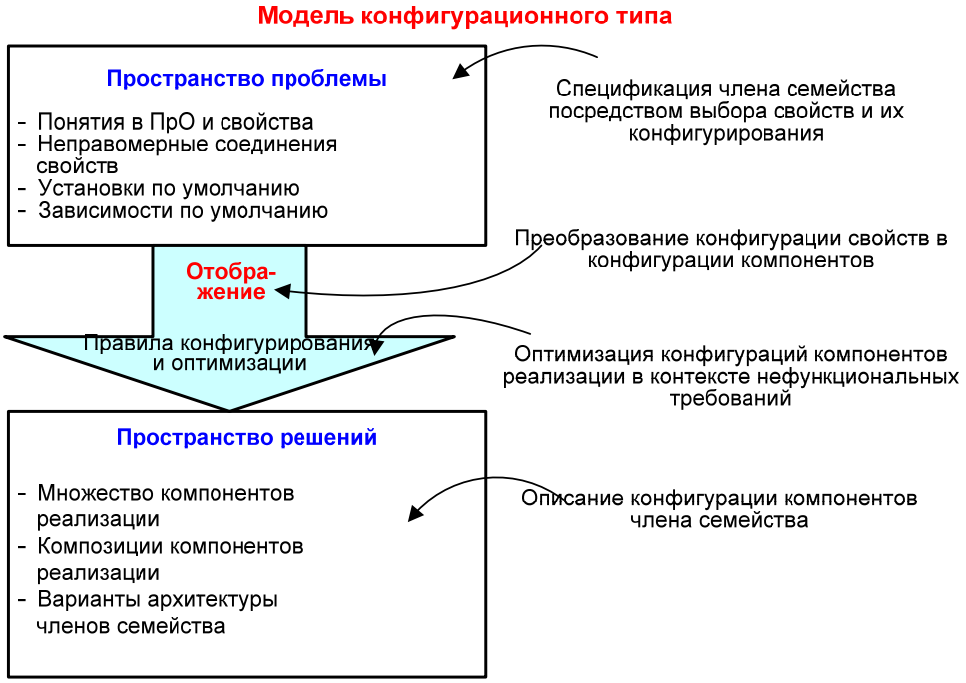


Рис. 2.13. Конфигурационная модель ПрО

Все эти элементы входят в конфигурационный файл пространства решений, а также в конфигурационную базу знаний.

При конфигурационном подходе может применяться другой язык описания архитектуры ADL (Arhitecture Description Languages).

По правилам описания архитектуры, трансформация описаний характеристик и ограничений МХ производится в описание обобщенной архитектуры семейства ПС в языке ADL. Следующая трансформация описаний компонентов выполняется средствами ЯП. Конфигурационный файл содержит все элементы реализации компонентов ПС, а также для выполнения и внесения изменений в структуру ПС и СПС.

4.3. Аспектно-ориентированное программирование

АОП является новым подходом в программировании в части добавления новых аспектов к ПС [95, 96].

В рамках ГП при моделировании ПрО используется аспектно-ориентированный подход (рис. 2.14) АОП (*Aspect-Oriented Development)*.

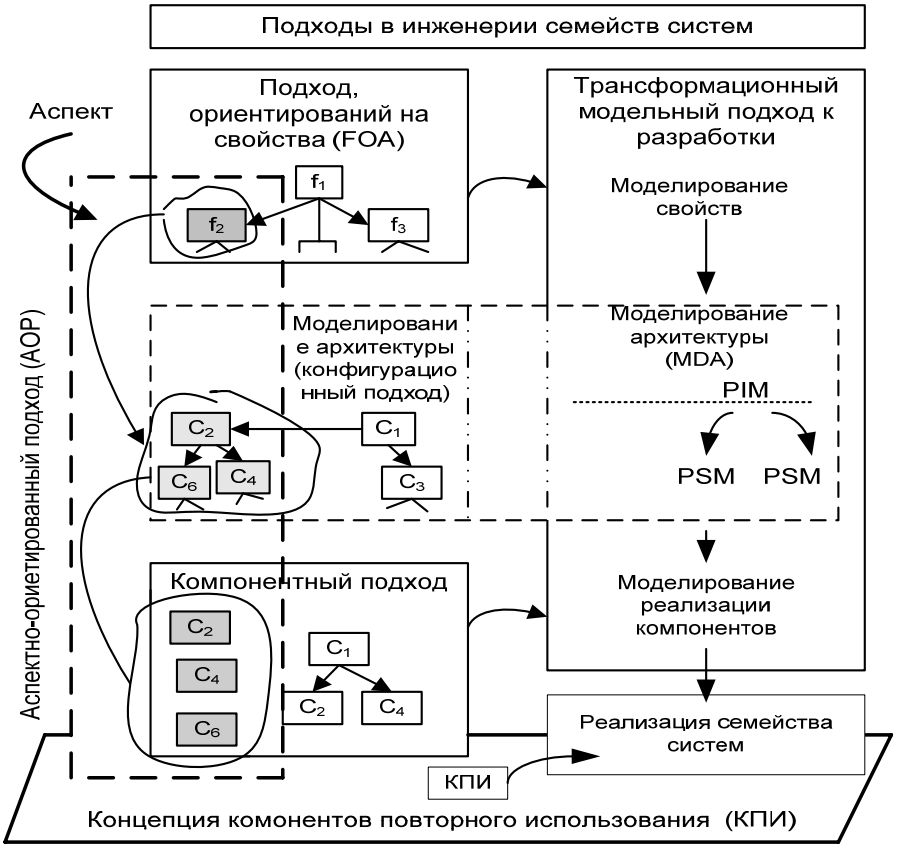


Рис. 2.14. Подходы к конструированию ПС

АОП способствует решению задач декомпозиции в пространстве проблем и пространстве решений с новыми аспектами (безопасность, синхронизация, надежность и т. п.), которые не учитывались при моделировании характеристик ПС.

Моделирование характеристики ПрО средствами FOD возможно также с помощью онтологического подхода (Ontology-Driven Development) и онтологии (Ontology-Based Feature Modelling) представления характеристик.

Этот подход к разработке программ, идейно близкий к идеям современного АОП, который был предложен в 1970 г. в бывшем СССР А. Л. Фуксманом (Ростовский университет) под названием технология рассредоточенных действий или технология вертикального слоения [1]. Соответственно этой технологии вертикальный слой (срез) содержит совокупность рассредоточенных действий, фрагментов кода, которые реализуют определенную *расширяющую функцию*, а процесс разработки и модификации программы представляет собой последовательность операций добавления или изменения этих функций.

Концепцию вертикальных слоев А. Л. Фуксман положил в основу новой стратегии поэтапной разработки программ. Согласно этой концепции на первом этапе создается "основа", т. е. максимально упрощенная версия программы после удаления из нее всех вертикальных слоев. Затем на последующих этапах

(транзакциях) реализуются и добавляются к программе все новые и новые вертикальные слои. С технологической точки зрения заметное преимущество данной стратегии заключается в том, что при наладке любой промежуточной (без некоторых вертикальных слоев) версии программы ненужные "заглушки" – имитаторы отсутствующих частей, без которых не могут обойтись известные стратегии разработки сверху–вниз или снизу–вверх.

Любая транзакция, направленная на эволюционное развитие программы, оформляется как добавление нового компонента (вертикального слоя), который распадается на модули, предназначенные для нескольких горизонтальных слоев. Любое "расширение" программы имеет определенную структуру в виде серии расширений имеющихся горизонтальных слоев. Исследование, анализ и постановка основных задач в аспектном программировании выполнена Г. И.Коваль в рамках мультипрограммирования фундаментального проекте ГП.

Основные элементы парадигмы АОП

Идея Фуксмана нашла отображение в концепции АОП (1997) группой разработчиков с Xerox PARC во главе с Г. Кикзалесом как инструмент реализации свойств ПС, которые отображают тот или другой не *функциональный аспект* их работы, например:

1) реакция на ошибки (их обработка и выдача диагностических сообщений),
2) обеспечение доступа к памяти (динамический заказ-освобождение),
3) синхронизация параллельно действующих объектов и действий, которые обычно "рассеиваются" по всем элементам системы, "пересекая" (cross-cutting) структуру системы, вплетаясь в код и запутывая его.

Аспект – это точка зрения, которая лежит в основе рассмотрения какого либо понятия, процесса, перспективы. Задача АОП заключается в том, чтоб выделить *сквозную функциональность* и вынести ее за пределы бизнес-логики программных приложения. В результате получают бизнес-логику в основных классах ПС и аккуратно расфасованную сквозную функциональность снаружи базовой модели. Компоненты связываются с аспектами в ходе компиляции.

Таким образом, АОП предлагает разные методы и методики *разбивки задач* (concern crosscutting) на ряд *функциональных компонентов*, а также *аспектов*, которые "пересекают" функциональные компоненты, и предусматривает их *композицию* с целью получения реализации систем. Причем, механизмы композиции обеспечивают не просто вызов процедур, а свободное и декларативное сцепление между частичными описаниями компонентов и аспектов.

Парадигма АОП фиксирует *модель встраивания аспекта в систему* (JPM, от Join Point Model), которая включает в себя элементы парадигмы:

1. *Точка присоединения* (Join point) – однозначно определяет место ("пересечение" компонент и аспектов в ПС), в котором выполняется вызов метода, инициация класса, доступ к полю класса и др.

2. *Срез* (PointCuts) – набор точек присоединения, который удовлетворяет заданному правилу (задания объектов, которые представляют интерес за определенным аспектом определенного события. Вместо правила может быть задан

простой перечень точек). В срезе определяется, где и когда может быть применена функциональность данного аспекта.

3. *Фрагмент вставки* (Advice) – набор "рекомендуемых" инструкций ЯП., которые нужно интегрировать во все точки присоединения, указанные в заданном срезе. Это код функциональности объектов, который будет срабатывать при наступлении определенного события. Наиболее популярное стандартные событие в языках АОП – события before (перед), after (после), around (вокруг, до и после вызова метода).

4. *Аспект* (aspect) – пара: правило, которое задает срез, и фрагмент вставки, которая подлежит вставке в точки этого среза. Аспект представляет собой языковую абстракцию типа класс, но более высокого уровня. Аспекты могут охватывать много классов и использовать точки присоединения для реализации регулярных действий, связанных с безопасностью, обработкой ошибок и т. п.

5. *Объявление* (Inter-type declaration) – нормализированное правило изменения структуры класса (открытые классы или смеси (mixins)). Декларации обеспечивают объявление дополнительных членов существующих типов.

Эти элементы (pointcuts, advice и inter-type) позволяют выразить процессы, которые пересекаются. Объявление аспекта может содержать эти три типа членов в добавление к обычным полям и методам языка программирования. Аспект представляет собой модуль хорошо скомпонованной рядовой структуры. Реализация этих механизмов различается в разных АОП-инструментах, но в основе каждого подхода лежит механизм доступа, создание, именование и абстрагирование точек присоединения.

Модуляризация сквозной функциональности, ее автоматизация, а также поиска (локализации) и модификации целевой программы обеспечивается следующим:

1) *компонентным языком*, посредством которого создаются компоненты;
2) одной или несколькими *аспектными языками* реализаций аспектов;
3) *компоновщиком* аспектов для комбинирования и интеграции компонентов и аспектов приложения.. Компоновка (сплетение аспектов) в систему может осуществляться во время выполнения (runtime weaving) или во время компиляции (compile time weaving) и загрузки классов.

Парадигма АОП связана с ООП и компонентным программированием и содействует повышению качества программ и сопровождения. Она входит в мультипарадигменную концепцию программирования, сущность которой заключается в том, что разные аспекты проектируемой ПС могут быть реализованы в других парадигмах программирования.

Для эффективной реализации аспектов в разных контекстах их применения разработаны библиотеки расширений языка программирования для определенных ПрО. Они включают в себя отдельные функции компиляторов, средств оптимизации, редактирования, визуализации понятий, перестройки компонентов компиляторов под новое языковое расширение, средства программирования на основе шаблонов и т. п. Пример – библиотека матриц, которая предоставляет средства для вычисления выражений с элементами массивов. Программирование с использованием таких библиотек принадлежит к разряду стилей родового программирования (generic programming).

Так как современные языки программирования не позволяют инкапсулировать аспекты в проектные решения системы, то в некоторые инструментальные компонентные системы введен механизм фильтрации входных сообщений, с помощью которых выполняется изменение параметров и имен текстов аспектов в конкретном компоненте. "Нечистый" код компонента (код с пересекаемыми его аспектами) требует разработки новых подходов к композиции компонентов, ориентированных на ПрО и на выполнение ее функций.

Одним из механизмов композиции является фильтр композиции, суть которого состоит в обновлении заданных аспектов синхронизации или взаимодействия без изменения функциональных возможностей компонента с помощью входных и выходных параметров сообщений, которые проходят фильтрацию и изменения, связанные с переопределением имен или функций самих объектов. Фильтры делегируют внутренним компонентам параметры, переадресовывая установленные ссылки, проверяют и размещают в буфере сообщения, локализуют ограничения на синхронизацию компонентов для выполнения.

Поскольку в ОО-программах может содержаться много мелких методов, которые не выполняют расчеты самостоятельно, и обращаются к другим методам, расположенным в областях внешнего уровня, Деметер сформулировал закон [53, 54], согласно которому не разрешаются длинные последовательности методов, связанные с передачами параметров через внутренние объекты. В результате создается код алгоритма, который содержит имена классов, не задействованных в выполнении расчетных операций. При необходимости внесения изменений в структуру классов, создается новый дополнительный класс, который расширяет ранее созданный код и не вносит качественных изменений в расчетные программы.

С точки зрения моделирования аспекты можно рассматривать как каркасы декомпозиции системы, в которых отдельные аспекты синхронизации, взаимодействия и др. пересекают ряд многократно используемых КПИ. Разным аспектам проектируемой системы могут отвечать и разные парадигмы программирования: объектно-ориентированные, структурные и др. Они по отношению к проектируемой ПрО образуют мультипарадигменную концепцию аспектов, такую, как синхронизация, взаимодействие, обработка ошибок и др., и требуют значительных доработок процессов их реализации. Кроме того, можно устанавливать связи с другими предметными областями для описания аспектов приложения в терминах родственных областей.

Существенной чертой любых аспектов является модель, которая пересекает структуру другой модели, для которой первая является аспектом. Так как аспект связан с моделью, то ее можно перестроить так, чтобы аспект стал, например, модулем и выполнял функцию посредника, беря на себя все образцы взаимодействия. Таким образом, проблема пересечения может привести к усложнению и понижению эффективности выполнения созданного модуля или компонента. Переплетение может проявиться на последующих этапах процесса разработки, когда реализуются аспекты, и они делают запутанным выходной код. Одним из путей оптимальной реализации аспектов является минимизация сцепления между аспектами и компонентами, которая реализуется ссылками на языковые конструкции, варианты использования, сопоставление с образцом.

Реализация аспектов в различных блоках кода позволяет устанавливать перекрестные ссылки между ними. Декларируется связь с соответствующей моделью, точками соединения сообщений, обработкой ошибок и т. п.

Связь между характеристиками и аспектами может быть выявлена в ходе анализа ПрО. Создается динамическое связывание через косвенный код или таблицы виртуальных таблиц для повторного связывания или статическое ("жесткое") связывание в период компиляции.

Для целей АОП хорошо подходит модель модульных расширений, создаваемая в рамках метамодельного программирования. Она предлагает оперативное распространение новых механизмов композиции в отдельные части ПС или их семейств с учетом предметно-ориентированных возможностей языков (например, SQL) и каркасов, которые поддерживают разного рода аспекты [51].

АОП можно рассматривать как самостоятельный стиль программирования в рамках сборочного конвейера, поставляющего некоторые специальные элементы, которые в функциональном плане не предусматривались. Зададим ему трактовку развития.

Современная трактовка АОП. АОП представляет собой одну из парадигм программирования, которая является дальнейшим развитием процедурного и ООП. В отличие от ООП, парадигма АОП выступает в роли логического дополнения в рассмотренных подходов, расширяя возможности разработчика в решении вопросов сквозной функциональности. В ней не определены формы абстракций и методики разработки компонентов. Главное в АОП это решение проблемы сквозной функциональности и смежных задач. Данная методология призвана снизить время, стоимость и сложность разработки современного ПО. В нем можно выделить определенные части, так называемые аспекты, отвечающие за ту или иную функциональность, реализация которой рассредоточена в коде программы.

Программа в АОП разделяются на основную и аспектную части. Основная часть состоит из компонентов, которые могут быть созданы в ООП и реализуют главную функциональность. Аспектная часть содержит реализации сквозной функциональности, которая была отделена от компонентов основной части. Такая организация программы способствует улучшению ее структуры, поскольку сквозная функциональность инкапсулирована в отдельные компоненты. АОП позволяет проводить тестирование и профилирование ПС без внесения каких-либо изменений в код.

Данное программирование позволяет автоматически добавлять необходимую функциональность во все компоненты при условии соответствующей настройки аспектов. Оно поддерживает мультипарадигменное программирование. Сущность АОП заключается в том, что разные аспекты проектируемой ПС могут быть реализованы в других стилях программирования. В нем сборка разноязычных модулей и компонентов базируется на библиотеках расширений ЯП, которые включают в себя отдельные функции компиляторов, средств оптимизации, редактирования, визуализации понятий, перестройки компонентов компиляторов под новое язычное расширение.

Технология и инструменты АОП

Технология построения ПП в АОП включает в себя следующие шаги:

1. Декомпозиция функциональных задач с условием многократного применения модулей и выделенных аспектов выполнения (параллельно, синхронно и т. д.).
2. Анализ языков спецификации аспектов для описания выделенных аспектов и других задач ПрО.
3. Определение точек встраивания аспектов в компоненты и формирование ссылок и связей с другими элементами ПрО.
4. Разработка фильтров для их представления на стороне сервера в целях управления соответственно заданными аспектами.
5. Определение механизмов композиции (вызовов процедур, методов, сцеплений) функциональных модулей, КПВ и аспектов в точках их соединения, как фрагментов управления выполнением или обращением из этих точек к другим модулям.
6. Создание объектной или компонентной модели, дополненной входными и исходными фильтрами сообщений.
7. Компиляция, общая отладка модулей и аспектов, а затем сборка их в ПП.

Для эффективной реализации аспектов разработаны системы Aspect, IP-библиотека (Intensional Programming – интенсивное программирование) расширений, активные библиотеки, а также проведенное расширение ЯП Smalltalk средствами описания аспектов (AspectJ, Aspect++, Aspect, AspectC#, JAC).

AspectJ (http://eclipse.org/aspectj/), инструмент, который поддерживает АОП в рамках языка JAVA, предоставляет *расширительные конструкции* для этого языка и встраивает такие системы разработки, как Eclipse (http://eclipse.org/aspectj/), Sun ONE Studio и Borland JBuilder.

Функционально аналогичные инструменты: для языка C++ это *AspectC++* (http://www.aspectc.org), для языка JAVAScript – AspectJS (http://www.aspectjs.com), для языка PHP 5 – phpaspect (http://phpaspect.org/), для C# - Eos (http://www.cs. virginia.edu) для Фреймворка .NET. Для платформы .NET разработан Aspect.NET в Санкт Петербургском университете [96].

Инструментами "переплетения" кода (weaving) являются такие: Xweaver (http://www.xweaver.org/Xweaver/), AspectXML (http://www.aspectxml.org), – DotSpect (.SPECT) (http://dotspect.tigris.org) – это двигатель переплетения независимых от языка аспектов во время компиляции в среде .NET. Предоставляет язык, подобный AspectJ с дополнительными синтаксическими элементами. Поддерживается для C# и VB.NET.

Специально для среды Eclipse разработан AJDT (AspectJ Development Tools for Eclipse) (http://www.ibm.com/developerworks/JAVA/library/j-ajdt/index.html) – инструмент (открытый проект), предназначенный для разработки и выполнения приложения на AspectJ.

Методические средства АОП. В АОП используется механизм фильтрации входных сообщений, посредством которых проводится изменение параметров и имен текстов аспектов в конкретно заданном компоненте системы. Код компонента становится "нечистым", когда его пересекают аспекты, и при композиции с другими компонентами общие средства (вызов процедур, RPC, RMI, IDL и др.)

становятся недостаточными. Механизм композиции компонентов и аспектов – *фильтр композиции* для обновления аспектов и их функциональных возможностей. Фактически фильтрация касается входных и выходных параметров.

Через аспектные механизмы устанавливаются связи с другими предметными областями в семействе программ или систем ПрО. Язык АОП позволяет описывать аспекты для разных систем семейства. После компиляции описаний пересекаемых аспектов, они генерируются, соединяются, оптимизируются и ориентируются на выполнение в динамике. При этом каждый аспект может стать модулем-посредником, который реализует шаблон взаимодействия отдельных программ или систем, а также самостоятельным компонентом, неся дополнительную функцию в среду инфраструктуры разработки ПС или семейства систем.

4.4. Модели взаимодействия систем. Теория и реализация

Интерфейс базис взаимодействия. В связи с быстрым изменением архитектуры компьютеров, появлением распределенных, клиент-серверных сред выявилась неоднородность ЯП как в смысле представления в них типов данных, так и платформ компьютеров, на которых реализованы соответствующие системы программирования, а также в различных способах передачи параметров между объектами в разных средах (маршаллинг данных через разные виды операторов удаленного вызова). Единого подхода к решению проблемы интерфейса пока не существует [36, 37, 97, 98].

Стандарт ISO/IEC 11404–1996 определяет подход к решению вопросов интерфейса всех видов ЯП с помощью универсального, независимого от ЯП языка LI (Language Independent). Однако до настоящего времени не существует егоинструментальной поддержки. Пользователям ЯП приходится выбирать подходящую реализацию интерфейса из множества имеющихся в разных средах [10].

Отметим особенности сред, влияющих на реализацию интерфейса:

1) разные двоичные представления результатов компиляторов для одного и того же ЯП, реализованные на разных архитектурах компьютеров;

2) двухнаправленность связей между ЯП и их зависимость от среды и платформы;

3) параметры вызовов объектов отображаются в операции методов;

4) связь с разными ЯП через ссылки на указатели в компиляторах;

5) связь модулей в ЯП осуществляется через интерфейсы каждой пары из множества языков (L_1, …, L_n) промежуточной среды.

Современные наиболее распространенные среды CORBA, COM, JAVA, каждая по-своему решает проблему связи разноязычных компонентов с помощью интерфейса.

Реализация связи компонентов в среде CORBA. В системе CORBA механизм связи разнородных объектов напоминает проектные решения в системе АПРОП с помощью модуля-посредника (stub, skeleton). Модуль-посредник stub выполняет аналогичные функции, связанные с преобразованием типов данных клиентских компонентов в ТД серверных компонентов посредством [46]:

1) отображения запросов клиента в ЯП в операции языка IDL (Interface Definition Language), RMI (Remote Invocation Interface) или API (Application Program Interface);

2) преобразования операций IDL в конструкции ЯП и передачи их серверу средствами брокера ORB, реализующего stub в ТД клиента.

Так как ЯП (C++, JAVA, Smalltalk, Visual C++, COBOL, ADA-95) реализованы на разных платформах и в разных средах и двоичное представление объектов зависит от конкретной аппаратной платформы, в системе CORBA реализован *общий механизм связи* разнородных готовых объектов – брокер ORB.

В эту среду может входить модель COM, в которой ТД определяются статически, а конструирование сложных типов данных осуществляется для массивов и записей. В системе CORBA методы объектов используются в двоичном коде, т. е. допускается двоичная совместимость машинных кодов объектов, созданных в разных средах, а также разных ЯП за счет отделения интерфейсов объектов от их 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. 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.

В среде клиент-сервера CORBA реализуется два способа связи:

1) на уровне ЯП через интерфейсы прикладного программирования;

2) на уровне компиляторов IDL, генерирующих клиентские и серверные интерфейсные посредники – stub, skeleton.

Интерфейсы определяются в IDL или APL для объекта-клиента и объекта-сервера, имеют отдельную реализацию и доступны разноязычных программам. Интерфейсы включают в себя описание формальных и фактических параметров программ, их типов и порядок задания операций передачи параметров и результатов при их взаимодействии. Другими словами, такое описание есть не что иное, как спецификация интерфейсного посредника двух разноязычных программ, которые взаимодействуют друг с другом через механизм вызова, который реализован на разных процессах. В функции интерфейсного посредника (stub) клиента входит:

1) подготовка внешних параметров клиента для обращения к сервису сервера,

2) посылка параметров серверу и запуск получения результата или сведений об ошибках.

Общие функции интерфейсного посредника (skeleton) сервера:

1) получение сообщения от клиента, запуск удаленной процедуры, вычисление результата и подготовка (кодирование или перекодирование) данных в формате клиента;

2) возврат результата клиенту через параметры сообщения и уничтожение удаленной процедуры и др.

Таким образом, интерфейсные посредники задают связь между клиентом и сервером (*stub* для клиента и *skeleton* для сервера).

Теория взаимодействия программ и систем

Проблема взаимодействия систем реализована в стандартной модели OSI (Open System Interconnection) открытых систем. Она определяет различные уровни взаимодействия систем в компьютерных сетях. Средства взаимосвязей определены на семи уровнях: прикладной, представительный, сеансовый, транспортный, сетевой, канальный и физический. Они обеспечивают *системные средства* взаимодействия, реализуемые операционной системой и аппаратными средствами. Модель OSI не включает в себя средства взаимодействия приложений в разных средах.

Для взаимодействия приложений используются механизмы взаимодействия (типа *RPC/RMI*), которые реализуют связь между компонентами приложения и могут обращаться с запросом к прикладному уровню модели OSI. Такие механизмы взаимодействия реализованы в современных операционных средах (VS.Net, CORBA, Eclipse, JAVA и др.) и поэтому приложения, изготовленные в этих средах, не могут переноситься в другую среду [9–11, 30–32]. В связи с этим нами решена задача определения взаимодействия между членами семейства для разных сред.

Модель взаимодействия программ и систем

В настоящее время программы, системы и другие артефакты нужно строить так, чтобы они могли быть перенесены в другие среды и функционировать там. При этом хотелось бы, чтобы на адаптацию продукта не требовалось значительных усилий разработчиков, связанных с доработками механизмов функционирования программ в новой среде. Долгое время в отделе проводились исследования проблем, связанных с действующими средствами поддержки интероперабельности, вариантности сложных систем (например, для обеспечения версий ОС IBM-360 в связи с введением новых функций или необходимостью внесения исправлений при обнаружении ошибок.

Под *взаимодействием* понимают совместимость двух и больше объектов (включая и людей). Данный термин имеет специальный спектр применения в программистской деятельности (например, взаимодействие программ и сред между собой). Способность к взаимодействию двух и больше программ или систем обусловлена обменом информацией и использованием ее для организации вычислений. Для обеспечения взаимодействия программ и систем применяют такие средства – RPC, RMI, ORB (stub, skeleton), IContract и т. п. Благодаря этим средствам, связи разноязычных и разноплатформенных программ осуществляет интерфейс, который специфицируется языком IDL (Interface Definition Language, APL, SIDL и т. п.). На общем уровне интерфейс используется, как механизм обеспечения взаимодействия (interconnection) разнородных программ и систем.

Сейчас возникает проблема обеспечения взаимодействия разнородных компонентов, которые строятся в разных интегрированных средах Eclipse, MS.Net, CORBA, COM и др. и могут адаптироваться в другой среде, например, Grid для проведения вычислений по программам в области e-science. При этом взаимодействие должно обеспечить обмен разнородной информацией между разными системами при вычислении в разных средах.

Модель взаимодействия. Под *моделью взаимодействия* понимается набор параметров относительно интероперабельности программ и процессов для установления зависимостей между ними и выполнение. Другими словами, модель должна отображать систему отношений, которая существует между элементами ПС. Эти отношения могут задаваться математическими средствами – абстрактной алгеброй, теорией множеств, логико-математическими операциями и др. Основными параметрами модели взаимодействия является программа (компонент), интерфейс и сообщение [37].

Модель взаимодействия M_{inter}, имеет такой вид:

$$M_{inter} = <M_{pro}, M_{sys}, M_{env}>,$$

где M_{pro} = (*Com, Int, Pr, Pro*) – модель программы или ПС*; Com* – компонент*, Int* – интерфейс*, Pr* – программа, *Pro* – тип запроса (*RPC, RMI, Icontract*)*;* M_{env} = {*Env, Int, Pro*} – модель среды, в которой *Int, Pro, Icontract/IP* задают совокупность внешних интерфейсов, программ и протокола *Icontract/IP*, через который передаются данные между распределенными программами и средами.

Модель среды M_{inter} по отношению к стандартной семиуровневой модели OSI моделью верхнего уровня, она включает в себя программные элементы, интерфейс и среду.

Программный элемент специфицируется ЯП, DSL, IDL, API и сетевыми языками типа XDL, RDF и т. п. Элементы сохраняются в библиотеках и репозиториях интегрированной среды ИТС ГП и используются при выполнении разных функций технологии объединения и взаимодействия КПИ, программ и систем.

Интерфейс [26, 27] как объект модели взаимодействия, включает параметры, набор операций и предикатов, которые определяют необходимые действия по обработки данных, переданных от одного программного элемента к другому. Он играет роль посредника между программами, которые обмениваются между собой данными. Если ТД оказывались не релевантными, то выполняется их прямое и обратное преобразование. Проблему взаимодействия между клиентными и серверными программами выполняют сервисные средства Service consumer и Service provider через протокол (*Icontract*) в системе WCF.

Среды разработки программ и систем. Для поддержки процессов разработки программ в ЯП (MS.Net, CORBA, JAVA) и сборки их в члены СПС в ИТС ГП включена система Eclipse, на базе которой разработаны специальные механизмы взаимодействия для следующих пар системных сред: Visual Studio↔Eclipse, CORBA ↔ VS.Net , IBM VSphere ↔ Eclipse.

Среда разработки программ. Рассматривается операционная среда с системными программными средствами и инструментами для поддержки процессов разработки отдельных программ в ЯП и сборки их в ПП. Базисом инструментальной среды выбран Eclipse, как средство управления репозиторием КПВ и использования при выработке новых ПС методом сборки. Жизненный цикл сред MS.Net, CORBA, JAVA допускает разработку программ в допустимых ЯП. Каждая из этих сред содержит свой специфический набор средств и подходов к трансформации описаний программ в ЯП к выходному коду с преобразованием типов данных , которые различаются между собой по типу. Общая модель распределенных систем сетевой среды Интернет приведена на рис. 2.15.

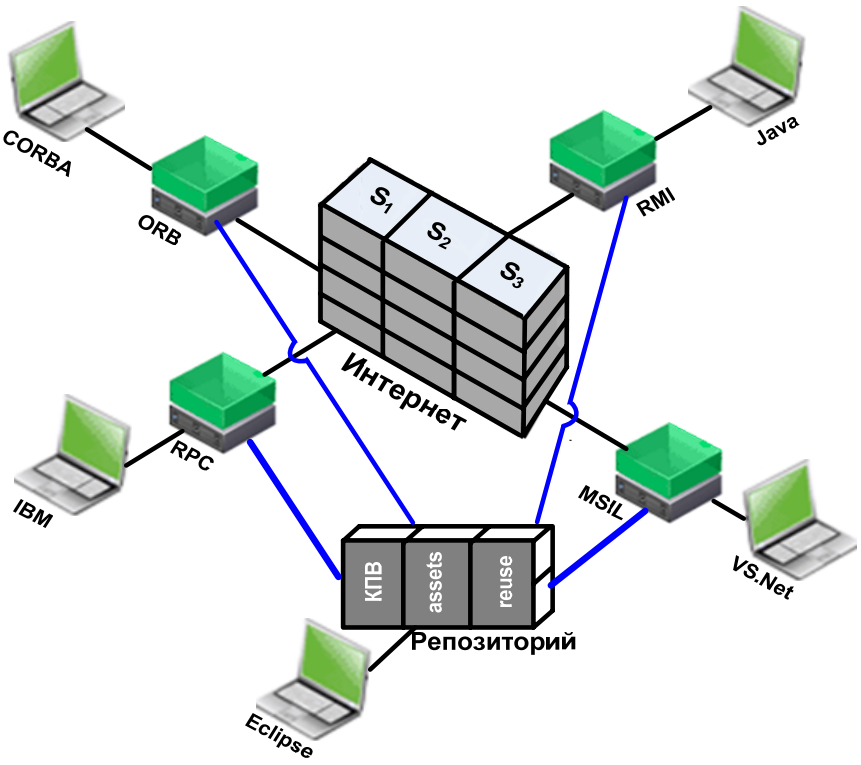


Рис. 2.15. Схемы разработки и взаимодействия программ в Интернет

Каждая из приведенных сред CORBA, IBM, VS.Net, JAVA, базируется на своих интерфейсах взаимодействия и включает в себя общие методы и средства доступа к данным сетевой среде. Программы, изготовленные в одной из сред, могут быть перенесены из одной среды в другую через репозиторий Eclipse. В пределах ГП проведено расширение среды Eclipse путем интеграции новых программ, произведенных в одной из сред, интегрированных в репозиторий. Эти операционные среды обеспечивают соответствующие процессы ЖЦ разработки разнородных программ и методы их объединения в разные структуры ПП через механизмы взаимодействия, реализованные в этой среде.

Проведена экспериментальная реализация принципов интеграции и взаимодействия программ для пар системных сред (см. раздел 4):

1) Visual Studio.Net↔Eclipse – это среда для технологии разработки отдельных программ в языке C# и спецификацией интерфейса с паспортными данными и возможности переноса готового продукта в репозиторий системы Eclipse. Такой инструмент отображает связь с данной средой разработки программ с помощью плагинов и конфигурационного файла с параметрами и операциями обработки данных в той или другой среде;

2) CORBA, JAVA, MS.Net обеспечивают разработку программ на ЯП этой среды и установление связей между этими программными средами в целях обеспечения размещения разработанных программ в репозитории и предоставления доступа другим разработчикам этих программ;

3) IBM VSphere, Eclipse предоставляет средства для разработки новых программ с использованием ЯП, которые допустимы в этой среде или в VSphere виртуального варианта этой системы.

Интерфейс является главным параметром моделей взаимодействия программ и систем в операционной среде Интернет.

Характеристика моделей взаимодействия программ, систем и сред

Модель взаимодействия программ – это схема связей отдельных частей программ и компонентов между собой. В качестве связей выступают операторы обращения (типа CALL) к процедурам и функциям этих программ с формальными параметрами. Операторы обращения содержат имена объектов (процедур и функций), которые вызываются, список фактических параметров, с заданием их значений, для получения результатов. Последовательность и число формальных параметров должны отвечать фактическим параметрам. Выполнение функции в программе на одном ЯП не является проблемным, когда типы данных параметров совпадают.

Модели взаимодействия систем наиболее связаны с использованием готовых программ в процессе разработки новых ПС методом сборки. Если все готовые программы заданы в разных ЯП и расположены они на разных компьютерах сети, то при их сборки могут возникать проблемы неоднородности типов данных в этих программах, структурах памяти платформ компьютеров и операционных сред, где они выполняются. Собранные ПП учитывают форматы данных готовых программ на платформах современных компьютеров, особенности структуры исходного кода программ после компиляторов с ЯП, который зависит от использования специальных библиотек *data types* и *routines* или без них. Реально существующие расхождения в аппаратной части платформ и в исходном коде программ учитываются в конфигурационном файле ПП, который используется при организации выполнения ПП в современных вычислительных или гетерогенных средах.

Некоторым вопросом преобразования общих типов данных (GDT) к фундаментальным FDT типам данных посвящен стандарт ISO/IEC 11404–2007 General Data Types. Он предлагает механизмы генерации сложных типов данных (контракт, портфель и др.) к более простым, которые есть в ЯП. Но фундаментальные типы данных ЯП поддерживаются специальными средствами современных операционных сред (Sun IBM, Microsofts.Net, CORBA, COM, JAVA и т. п.). К ним относятся промежуточный посредник типа stub, skeleton брокерного типа, который выполняет преобразование типов данных от клиента для передачи серверу и наоборот, а также промежуточный язык MISL (Microsoft Intermediate Language) или EXE для выполнения кода в таких системах, как Linux, Windows Server, MS.Net, IBM Web Sphere и др. [37, 38].

Модель взаимодействия операционных сред между собой рассматривается нами как последовательность действий перенесения программы, созданной в одной среде, в другую для их выполнения. Общая схема разработки распределенных программ в среде ГП на основе интерфейсов программ и моделей взаимодействия для указанных сред IBM, VS.Net, JAVA, CORBA приведено на рис. 2.16.

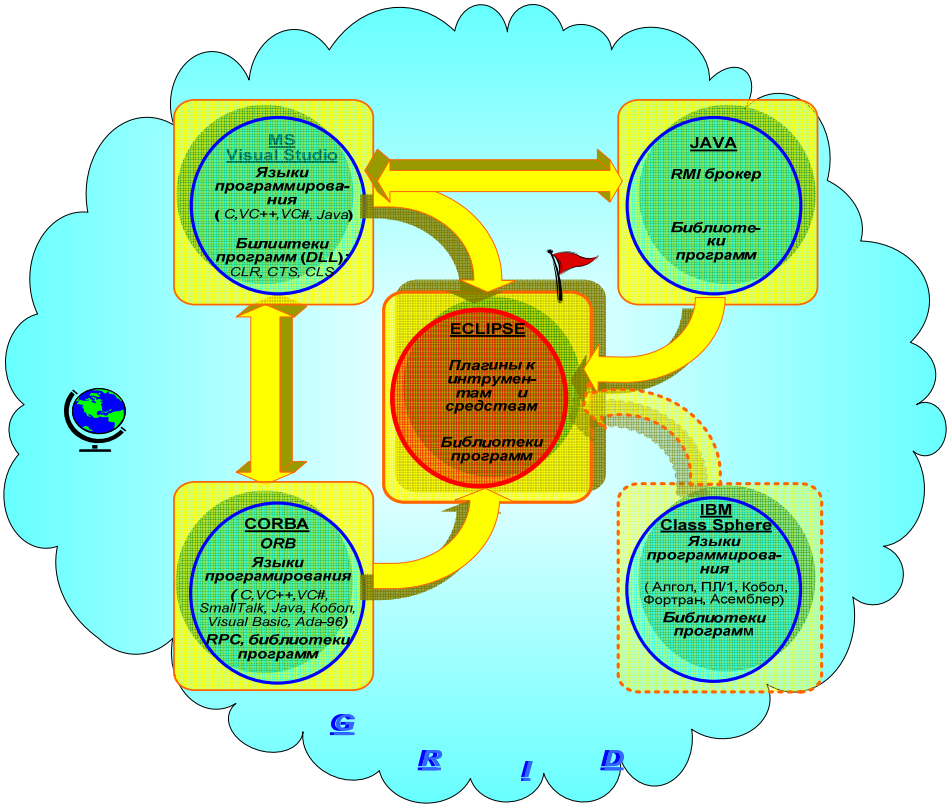


Рис. 2.16. Схема взаимодействия сред

В центре этой схемы размещена система Eclipse, которая выполняет, по существу, функцию интегратора программ в репозитории из других сред Интернета и администратора этого репозитория по сохранению, отбору и применению готовых программных ресурсов типа КПИ для выполнения ПП, изготовленных в средах. Некоторые среды, например, VS.Net и JAVA уже имеют непосредственные связи и создают одну интегрированную среду.

В рамках фундаментального проекта проведена реализация моделей взаимодействия пар следующих операционных сред: Visual Studio ↔ Eclipse и CORBA ↔ Visual Studio при участии студентов КНУ имени Тараса Шевченко (А.Аронова , А.Дзубенко) и МФТИ (А.Островского) [97– 98].

Реализация моделей взаимодействия в ИТС

Модели взаимодействия программ, систем и сред практически реализованы в средах Visual Studio, CORBA, VSphere, Eclipse. Проведена апробация механизмов взаимодействия программ и систем, а также их перенос в другую среду.

Модель взаимодействия систем *Visual Studio↔Eclipse* реализована на примере построения программ в языке C# Visual Studio, размещения его в репозитории Eclipse и выполнения средствами плагіна *Emonic* и компонента *NAnt* платформы VS.*Net*. Для них создавался соответствующий архив, аналогичный

среды Eclipse. Для выполнения программ в среде Eclipse создается пустой проект, для которого из контекстного меню выбирается пункт *Import→ File System* и из файловой системы импортируется выбранный проект. В конфигурационном файле *.build* изменялись имена исходных файлов, библиотек и файлов ресурсов, а также использовались исходные файлы *dll*-бібліотки VS.*Net* и файлы ресурсов (*.resx*). При переходе из этой среды в Visual Studio применялся импорт проекта в среду Eclipse.

Модель взаимодействия сред *CORBA ↔ VS.Net* реализована на примере программы в языке JAVA пакета *JDK* и системы CORBA, которая после компиляции IDL-интерфейса передается брокеру ORB. Перенос этой программы в MS.Net проведен с помощью пакета IIOPNet. Реализация задачи взаимодействия прикладных компонентов, разработанных в средах MS.Net (клиентская часть) и JAVA (серверная часть) выполнена CORBA. на платформе MS.Net Вычисленные значения данных из MS.Net путем маршалинга MarshalByRefObject возвращдись исходному объекту. Взаимодействие программ между двумя средами JAVA и MS.Net происходит посредством системных процедур серверной части приложения; включая Skeleton и інтерфейсного объекта в языке IDL утилитами *rmic,* средствами *CLS*-библиотеки и сервисным пакетом IIOPNet.

Модель взаимодействия *VSphere ↔ Eclipse* изучается для дальнейшей ее реализации в заданной среде VSphere. В виде эксперимента разрабатывается программа в ЯП и интерфейса в API для выполнения в виртуальной среде Vsphere и Eclipse↔VS.Net.

Практика обеспечения взаимодействия в современных средах

Кроме проведенных экспериментов по взаимодействию программ и сред с помощью C# и CORBA, JAVA и Eclipse, исследованы новые системные средства взаимодействия в рамках WCF (Windows Communication Foundation) [97]. В нем рализован новый тип интерфейса для сервисов – *IContract*, как составная часть Consumer и Provider .

Интерфейс *Icontract* содержит описание атрибутов и операций передачи данных от одного сервисного объекта клиента (*Service consumer*) к другому (*Service provider*). Их описание задается в языке XML. Передача интерфейсов между ними выполняет протокол, в котором задаются атрибуты и операции интерфейса. В системе WCF реализован новый тип интерфейса взаимодействия сервисов через контракт. С помощью четырех видов интерфейсов-контракта:

1) службы операций, вызываемых клиентом;
2) фундаментальных данных (int, float, string и др.) для передачи их через сервисные службы;
3) ошибок, которые могут содержаться для передачи контрактов клиенту;
4) сообщений прямого взаимодействия объектов между собой.

Сообщение в языке XML задается протоколом SOAP в следующем виде:

```
<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://www.cbsystematics.com">
<!-Конверт протоколу SOAP->
<env:Header>
```

<!- Заголовок протоколу SOAP->
</env:Header> <env:Body>
<!-Тело протоколу SOAP->
</env:Body> </env:Envelope>

При взаимодействии между пользователем и провайдером могут возникнуть разного конфликты (рис. 2.17) следующего вида.

1. Несовместимость ТД, которые передаются в контрактных интерфейсах (например, тип "целое", а параметр описан как "символьный").
2. Различие конфигурационных файлов по структуре и содержанию информации.
3. Разница описаний некоторых ТД в ЯП.
4. Разтличие порядка задания параметров в протоколе передачи информации между сервисными объектами;
5. Различие в архитектуре платформ объектов клиента и сервера.

Снятие таких конфликтов решается системными средствами.

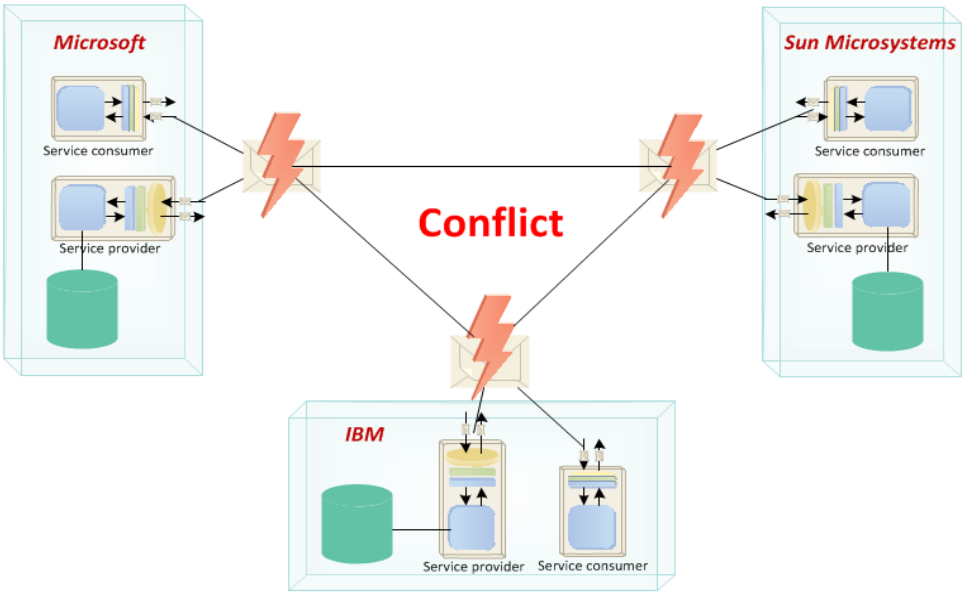


Рис. 2.17. Схема взаимодействия программ с конфликтами

4.5. Модель конструирования вариантных систем и семейств ПС

Проблема изменения функциональности ПС связана с изменением требований заказчика, адаптации ПС на другие среды функционирования, а также замены отдельных КПИ в составе некоторого члена семейства другим. В связи с этим в рамках диссертационного исследования рассмотрена проблемы вариабельности систем и семейств СПС, ориентированной создание вариантов членов семейства и процессов адаптации их в современных средах. Создана теория моделирования разных вариантов членов ПС. Результаты теории и практики опубликованы в [88, 89] и в электронной монографии [91]. Отдельные ас-

пекты обеспечения вариантности реализованы в ИТС с помощью компонента - конфигуратора.

Определение вариабельных, вариантных систем. Вариабельность – способность семейства продуктов или систем к расширению, замене и конфигурации отдельных компонентов в виде ПС для решения задач конкретной ПрО. Впервые идея вариабельности СПП сформулирована Product Lines SEI, как механизм поддержки вариантов семейства продуктов, изготовленных или взятых готовых по требованию заказчика. Варианты СПП обеспечиваются "извлечением" из него отдельных КПИ и заменой их новыми функциональными КПИ для удовлетворения требований заказчика.

С общей точки, зрения семейство СПС (SPS) это кортеж из совокупности моделей вариабельных и вариантных элементов членов семейства:

$$M_{\mathrm{SPS}} = <M_{\mathrm{PrO}}, (KPV, PRG, RPC), M_{\mathrm{FM}}, M_{\mathrm{var}}, M_{\mathrm{Config}}, M_{\mathrm{ref}}>,$$

где M_{PrO} – модель ПрО; M_{FM} – модель характеристик (базовых и специальных) членов семейства; *KPV* – множество КПИ; *PRG* – предикат принадлежности *KPV*; к члену ПС; *RPC* – интерфейсный предикат, определяющий операции передачи данных КПИ в другие члены семейства; M_{var} – модель вариантности ПС из КПИ и членов СПС; M_{Config} – модель процесса сборки (конфигурации); M_{ref} – модель рефакторинга.

Модель M_{var} определяется двумя связанными между собой моделями – объектной ОМ и компонентной КМ. Их представление в среде ГП это усовершенствование модельной среды и компонентной алгебры ПС, описанных в п 3. данного раздела, а также в [4].

Составляющие элементы модельной среды КП ИТК – объектная, компонентная вариантная модель СПС, задаваемая при четырехуровневом проектировании ПС.

Объектная модель имеет такой вид:

$$OM = \langle G^t_1; G^t_2, G^t_3, G^t_4\rangle,$$

где G^t_1 – граф объектов ПрО, создаваемый на обобщающем уровне проектирования (t=1); G^t_2 – feature model характеристического уровня (t=2); G^t_3 – архитектурно–компонентная модель структурного уровня (t=3); G^t_4 – интерфейсная модель взаимодействия компонентов СПС на поведенческом уровне (t=4).

Объектам функций G^t_1 и их характеристикам соответствуют методы и данные (уровня t = 2, 3), которые необходимы при реализации их в СПС и обеспечении взаимодействия.

Компонентная модель СПС – развитие ОМ, методы объектов которой реализуются программными КПИ для одного, и только одного ее объекта и интерфейса между ними. Модель имеет следующий вид:

$$KM = \langle RC, In, ImC, Fim, B_{KM}\rangle,$$

где *RC* – базовые компоненты множества готовых компонентов С, которые соответствуют базовым объектам модели ОМ; *In* – интерфейс компонентов, среди параметров которого задается имя точки вариантности; *ImC* – реализация базового компонента в заданной среде; *Fim* (·) – функции преобразования входных и выходных параметров интерфейса и множество данных в сигнатуре интерфейса; B_{KM} – множество данных КМ.

Подход к управлению вариантами КПИ и систем СПС. Компонентная модель ориентирована на обеспечение вариантности ПС в семействе СПС и оценки уровня вариабельности для потребностей заказчика. Конструирование вариабельных СПС по моделям ОМ и КМ выполняется с помощью процессов управления вариантами одной ПС, вариантами других членов СПС путем конфигурирования и рефакторинга КПИ и полученных членов СПС.

Основную роль в этих процессах выполняет конфигуратор среды ИТК ГП. Он обеспечивает объединение КПИ и их интерфейсов с вариантами отдельных рабочих продуктов ПС, которые находятся в репозитории, и реализуют логическую вариантность.

Применение конфигуратора КПИ необходимо в инженерии семейств систем, поскольку варианты каждого КПИ отвечают контекстам его использования в разных системах СПС и находятся под контролем конфигуратора ИТК ГП. Готовая система ПС, как член СПС, также является некоторым вариантом и может применяться в другой сложной системе. СПС с вариантами способно к расширению, изменению и настройки на новую среду реализации и выполнения.

Для управления конфигурацией СПС репозиторий ИТК дополнен механизмами хранения, введения (изъятия) КПИ и их интерфейсов. Они используются при сборки, объединении КПИ в вариантные структуры ПС.

Управление конфигурацией ПС в среде ИТК состоит в обслуживании КПИ (подбор, поиск, выбор) из репозитория, сборки из них по конфигурационному файлу нового варианта ПС в семействе СПС.

4.6. Модели сложных и распределенных систем

Первый аспект моделирования систем всех известных подходов – понятийная база, включающая в себя систему понятий, посредством которой формулируются все модели и процессы сложной системы. Понятийная база определяет терминологию, которой должны пользоваться участники задания требований к системе и ее разработчики.

Концептуальная модель формируется путем обобщения понятий в классы, конкретизации отдельных понятий, агрегации понятий в новое понятие и ассоциации, установления связей и отношений между понятиями. При этом совокупность терминологии, понятий, характерных для них отношений и парадигмы их интерпретации в рамках создаваемой модели называют онтологией доменных знаний.

Такая модель включает в себя объектную, компонентную модели, принципы построения которых рассмотрены в п. 2, 3. Здесь предлагаются новые модели сложных систем, основанные на объектном и компонентном моделировании, а также на перспективных моделях взаимодействия и вариантности сложных систем для их функционирования в гетерогенных средах.

Модель предметной области

Предметная область – это совокупность точных определений понятий, концептов объектов ее реального пространства с характеристиками, а также множества синонимов и классификационных логических взаимосвязей между этими

понятиями. Объекты как абстракции реального мира и понятийная структура отображают методы (функции) объектов и отношения между ними. Выделенные в ПрО объекты структурно упорядочиваются посредством теоретически множественных операций (селекции, композиции, пересечения и тому подобное) и объединяются по общим признакам и характеристикам в классы и суперклассы. Представим модель ПрО в виде

$$M_{ПрО} = <Mo, Minc, Mcx, M_{ПС,} P, D>,$$

где *Mo* – множество объектов, отношений между ними и сведений о содержании функций и данных объектов, с которыми они взаимодействуют; *Minc – модель* взаимосвязи между собой объектов через интерфейс; *Mcx* – множество общих характеристик связанных объектов через данные и характеристики внутреннего типа, что присущие каждому объекту и входят в схему композиций объектов ПрО; $M_{ПС}$ – модель программной системы или $M_{СПС}$, которая реализуют задание и функции ПрО; *P* – множество предикатов порядка и условий выполнения объектов по их функциональным и нефункциональным характеристикам модели свойств (featute) и взаимосвязи объектов модели *Mo*, методы которых обеспечивают их программную реализацию в ПС; *D* – множество данных ПрО, которые необходимы для выполнения отдельных компонентов и ПС в целом и которые могут сохраняться в базах данных СУБД.

На основе ОКМ обеспечивается перестроение объектной модели и преобразование ее в компонентную. В ней методы объектов реализуются множеством компонентов, а также их интерфейсами и характеристиками, обеспечивающими изменение некоторых элементов и их взаимодействие между собой и средой [3].

Другими словами, практически формируется множество компонентов, адекватно множеству функций (методов) об'єктов ОМ. Создается множество компонентов С, из которых необходимо сконструировать ПС по определенным функциональным и нефункциональным требованиям, которые были сформулированы в объектной модели. Цель заключается в том, чтобы представить объектную модель, потом ее представить в виде компонентной модели СМ при условии, что для любого элемента объектной модели существует программный компонент из множества $C= (C_1, C_2, C_N)$ или он может быть получен из другого множества, как готовый ресурс.

На практике такие условия выполняются не всегда, например, разработана ПС, имеющая принципиально новую функциональность или ее функции необходимо усовершенствовать. В таких случаях реализуются дополнительные компоненты, множество *C* расширяется или уменьшается за счет не нужных. Это в свою очередь инициирует конфигурирование нового варианта ПС.

Таким образом, ПС создаются, как правило, из компонентов, которые реализуют функций ПрО, или отбираются из готовых КПИ, накопленных в современных библиотеках или репозиториях.

Далее рассмотрим ряд формальных моделей представления разных моделей ПС и СПС, которые сформированы и усовершенствованы в рамках фундаментальных проектов ИПС НАНУ(2001–2012) на основе метода ОКМ [4, 5]. Здесь предложен метод проектирования модели ПрО из объектов по теоретико-множественным операциям для отображения объектов ПрО, метод трансформации абстрактных описаний объектов в форму программных компонентов и по-

гружения в операционную среду с целью накопления некоторых готовых КПИ для их выполнения.

На практике ОКМ представлен совокупностью простых технологий для реализации отдельных частей элементов моделей систем и их сборки методом конфигурации в новые ПС из КПИ, сервисов и ПП [7–9].

Модель программной системы

Программная система – совокупность отдельных компонентов, которые реализованы из объектов ПрО, с установленными интерфейсами, взаимоувязанными с функциями некоторой ПрО в заданной среде. Программный компонент, исходя из общей точки зрения – это самостоятельный продукт (код) метода или функция, которая поддерживает объектную модель, реализует часть или всю отдельную предметную область и умеет взаимодействовать с другими компонентами системы через интерфейсы.

Модель ПС определим так:

$$M_{ПС} = <L, M_f, M_s, M_i, M_d>,$$

где $L = (L_1, L_2 ., L_k)$ – язык описания функций ПрО и каждый язык включает в себя набор операций алгебры действий (actions) по реализации и выполнению соответствующих компонентов; M_f – множество функций модели ПрО, которая является по сущности моделью объектов ПрО $M_f = (O_1, O_2., O_r,)$, каждый объект которой трансформируется к компонентному коду; $M_s = (M_{sin}, M_{sout}, M_{sinout})$ – множество сервисов, которое базируется на модели связи и включает в себя модели передачи входных данных M_{sin}, выходных данных M_{sout} и серверных данных M_{sinout} ПС на сервере Application, базированных на множестве системных сервисах (Common Facility Services) операционной среды, которая реализуется специальным брокером или монитором сервисов и др.; M_i – множество интерфейсов в языке IDL с описанием входных *in* и выходных параметров *out* и *inout* КПВ объединенного типа для пары связанных компонентов в структуре ПС. С практической точки зрения все общие ТД специфицируются как внешние данные в паспорте модели ПС из объектов ПрО; M_d – множество данных и метаданных предметной области ПС, которые специфицированы в КПИ, как примитивные или сложные фундаментальные ТД языков программирования, а также в модулях-посредниках (stub, skeleton) в языке IDL.

Множества M_{inc}, M_s и M_i связаны с интерфейсными и общим данными и имеют пересечение по данным, которые отнесены к *in*, *out*, *inout* и входят в состав внешних типов данных, которыми обмениваются по сети один компонен с другим на сервере Application. По переданным данным выполняются компоненты и отправляют результаты компонента, который обратился с запросом или протоколом к серверному компоненту.

Модель семейства систем

Семейство систем СПС – это совокупность ПС, которые определяются общим множеством понятий и множеством специальных понятий и характеристик, которые присущи каждому члену этого семейства при реализации некоторой ПрО.

Понятие семейства программ ввел Дейкстра (1970). Оно основывается на том, что программы имеют общую родственную связь, из которой могут поро-

ждаться разные варианты других программ, нуждающихся в необходимой корректировке или замене в случае некорректных либо недостаточно определенных функций или в связи с уточнением общей постановки задач семейства ПС.

Модель СПС включает в себя модели ПС и имеет вид

$$M_{СПС} = < (M_{ПС1}, M_{ПС2}, M_{ПСk}), (M_{gf}, M_{sf} = (M_{sf1}, M_{sf2}, M_{sfk}), M_i, M_d) >,$$

где $M_{ПС1}$, $M_{ПС2}$, $M_{ПСk}$ – множество членов семейства ПС; M_{gf} – множество внешних характеристик ПрО, которые определяют общую терминологию и характеристики всех ПС семейства; $M_{sf} = (M_{sf1}, M_{sf2}, ..., M_{sfr})$ – множество внутренних характеристик каждого отдельного члена семейства, заданного моделью $M_{ПС}$, т. е. M_{sf1} для $M_{ПС1}$, M_{sfr} для $M_{ПСk}$ и т. д.; M_i – множество общих интерфейсов членов ПС, которые описываются языком IDL; M_{inc} – модель взаимосвязи ПС на множестве элементов Mi; $M_d = (M_{d1}, M_{d2}, ., M_{dk})$ – множество данных каждого члена ПС.

Монитор или брокер запросов руководит функциональными компонентами через RPC, RMI, Icontract, передает данные, заданные в M_{inc}, специфицированных языком интерфейса IDL, API модели клиент-сервер интерфейсного посредника (stub, skeleton).

4.7. CASE-системы поддержки мультипрограммирования

В ГП как в мультипарадигменной системе используется ряд систем, которые поддерживают разработку разных программных элементов и сложных ПС:

1. *Систему Aspect*
2. *Component Object Model* (COM) – стандарт Microsoft для компонентов включает в себя протокол конкретизации и применения компонентов внутри процесса, между процессами или между компьютерами. ActiveX, OLE обеспечивают взаемосвязь в языках Visual Basic, C++, C#, .NET и др.
3. *JAVA Beans* – стандарт Sun Microsystems для компонентов (зависящий от языка).
4. *CORBA* (IDL-интерфейс для связи КПВ в языках программирования (ЯП), сложность отображения одного ЯП реализации в другой, проблемы преобразования типов данных).
5. *Microsoft Visual Studio.NET* – среда разработки, среда выполнения – Common Language Runtime (CLR), среда выполнения как CLR реализация ТД, межъязычного взаимодействия и разворачивания (deployment) компонентов.
6. *IBM, VSphere IBM* – сервисная компонентная архитектура (SCA) с интерфейсом в WSDL для импорта и экспорта данных (EJBs), интеграция и упаковка компонентов в сервисный модуль типа файла J2EE.
7. *OBERON* компонетная обработка на многих ЯП, подобно CORBA на коммерческой основе и др.

Каждая системная среда производит КПИ в тех ЯП, которые предсталены в их среде. Они накапливаются в разных библиотеках и используются на многих фабриках программ. Одна из систем поддержки КПИ и их замена новыми аспектами защиты и безопасности данных – система Aspect.

Case-инструменты АОП. *Система Aspect* разработана на базе языка JAVA. Система имеет компилятор, отладчик и генератор документации.

Компилятор выдает байт-код, совместимый с виртуальной машиной JAVA. Расширения языка JAVA касаются способов описания правил интеграции аспектов и JAVA-объектов и базируются на таких ключевых понятиях, заданных выше.

После компиляции получается готовая система с функциональностью, интегрированной по правилам, описанным в аспектных модулях.

Существуют другие реализации АОП: AspectC++, AspectC, AspectC#, расширение словно C++, C, C# аспектами; JAC - система, написанная языком JAVA, для создания распределенных ПС; Weave.NET - проект реализации механизма поддержки АОП без привязки к конкретной ЯП внутри компонентной модели .NET Framework и др.

В процессе создания ПС с применением аспектов могут использоваться: IP-библиотека расширений, которая включают в себя их коды, а также активные библиотеки, МП SmallTalk, расширенные средства описания аспектов [17].

Использование таких библиотек в средах программирования называют *родовым программированиям с* решением проблем экономии, перестройки компиляторов под каждое новое язычное расширение, использования шаблонов и результатов предыдущей обработки относят к области *ментального программирования* [17]. Библиотека IP включает в себя: отдельные функции компиляторов, средства оптимизации, редактирования, отображения понятий, перестройки отдельных компонентов компиляторов под новое язычное расширение, а также средства программирования на основе шаблонов и т. п. Библиотеки с такими возможностями получили название *библиотек генерации*.

Другой вид библиотек АОП – *активные библиотеки*, которые включают в себя не только базовый код реализации понятий ПрО, но и целевой код обеспечения оптимизации, адаптации, визуализации и редактирования. Активные библиотеки пополняются средствами и инструментами интеллектуализации агентов, посредством которых обеспечивается разработка специализированных агентов для реализации конкретных задач ПрО.

Глава 5. СЕРВИСНОЕ ПРОГРАММИРОВАНИЕ

Эта парадигма программирования появилась как следствие рассмотрения программных компонентов, которые могут использоваться в качестве сервиса. Для них определены интерфейсы взаимодействия с разными программами и распределенными системами (CORBA, DCOM и EJB, MS.Net, IBM и тому подобное) и с веб-сервисами Интернета. Сейчас действуют сервисно-ориентированная архитектура SOA (Service-Oriented Architecture), средства их поддержки (XML, SOAP, WSDL и др.) и механизмы взаимодействия обычных сервисов распределенных приложений и веб-сервисов Интернет [18].

5.1. Сервис. Базовые понятия

Веб-сервис – программа, которая идентифицируется строкой URI, свойства и методы которой описаны с помощью специального языка WSDL. Доступ к ресурсам такой системы осуществляется через протокол SOAP, который представ-

ляется XML-запросами, которые передаются через HTTP Интернет-протокола. Веб-сервисы близкие классам объектно-ориентированных ЯП (JAVA EE). Ключевые понятие веб-сервиса – это сообщение из одной или нескольких переменных. Методы классов задаются операциями с входным и выходным значениями сообщений. После вызова операции переменной входного сообщения протокола SOAP, интерпретируются как параметры соответствующего метода класса, который лежит в основе сервиса.

Сервис определяется, с одной стороны, как открытый компонент, который может быть элементом быстрой композиции в прикладные приложения. С другой стороны, сервис предлагается как готовый ресурс, который реализует некоторые дополнительные возможности, необходимые всем разнородным программам для технической поддержки, нужной потенциальным пользователям. Как правило, описания сервисов содержат в себе информацию об их возможностях, интерфейсах, поведении и характеристиках. Благодаря такому описанию пользователь может найти сервисы, выбрать нужные и интегрировать их в свою композиционную структуру, как готовый ресурс.

Рассматривается три вида сервисов:

1) общие системные сервисы, которые есть в каждой общесистемной среде для поддержки процессов проектирования и реализации РПС на основе сформулированных моделей ПС, ПС и РПС;

2) объектные сервисы, которые поддерживают объекты и классы, операции ЖЦ, услуги необходимы для разработки РПС в объектно-ориентированной среде;

3) веб-сервисы, которые базируются на информационных ресурсах Интернет и обеспечивают создание элементов РПС путем композиции или интеграции КПИ и сервисов, способных к функционированию в вебе Всемерной паутины.

Системные сервисы создают некоторое множество сервисов CServ = {CServr}, которое необходимо для организации построения и функционирования функций компонентов и сервисов РПС, а также для управления их конфигурационными структурами. В частности, набор сервисов включает в себя сервис:

1. Наименование Naming, которое обеспечивает возможность поиска компонентов в распределенной среде с учетом пространства имен.

2. Связи Binding предназначены для определения (связи) соответствия имя-объект и применяемая к выбранным компонентам.

3. Транзакций Transaction, который обеспечивает организацию и управление функционированием совокупности компонентов и функций сервисов.

4. Сообщения Messaging, необходимые для организации общения между компонентами, являются составным элементом в модели асинхронных транзакций.

В реальных гетерогенных средах могут быть реализованы и другие системные сервисы. Например, сервис управления событиями, служба каталогов и др.. Но много таких сервисов определяются, в основном, условиями упрощения организации функционирования сред и могут быть созданы на базе других сервисов. В дальнейшем будем считать, что перечисленные выше четыре вида сервиса обязательны для любой модели РПС и ее реализации. Они отражают реализацию базовых функций управления компонентами в среде:

1) поиск компонентов;

2) доступ к их ресурсам;

3) организация обмена информацией между компонентами;

4) динамическое управление функционированием, обусловленным совокупностью КПИ в РПС.

Модель сервисов РПС базируется на унификации и совместимости, что позволяет рассматривать РПС как набор сервисов, их функциональность и взаимодействие.

Унификация сервисов достигается путем:

1) типизации функциональности сервисов и других характеристик;

2) применения унифицированных языков для описания сервисов и их взаимодействия;

3) использование стандартных базовых технологий.

С общей точки зрения эта модель создается на основе объектной модели, каждый объект которой имеет типизированную структуру и интерфейс. Типизация переносится и на реализацию объектов в виде сервисов. Для представления разных аспектов описания и функционирования сервисов используются унифицированный язык XML. К этим аспектам относятся описание структур и семантики данных, механизма взаимодействия с сервисами, функциональных услуг и механизм поиска необходимых сервисов.

Средством описания механизма взаимодействия с сервисами является SOAP, а для описания функциональности сервисов – язык WSDL. Описание функциональности сервисов выполняется унифицированными языками XML, WSDL; описание структуры и семантики данных выполняется языком RDF; описание процессов представления и обработки сервисов языком BPMN, а взаимодействие с сервисами и поиска необходимых сервисов – SOAP.

Отдельный класс средств унификации составляет онтология – модели и словари, которые обеспечивают согласование терминов и понятий согласно описанию сервисов на уровне семантики. В качестве стандартных базовых технологий при реализации сервисов используются: клиєнт-серверна архитектура, унифицированные коммуникационные протоколы, компонентные модели и т. д.

Модель сервиса обеспечивает:

1) динамическое расширение функциональности системы за счет поиска и привлечения в сферу обработки новых сервисов;

2) повышение масштаба и улучшение возможностей коммуникации между отдельными элементами системы за счет стандартного механизма подключения сервисов через их интерфейсы;

3) повышение языкового уровня коммуникации системы с конечными пользователями.

Принципы разработки систем из готовых программных и информационных ресурсов (компонентов, сервисов, интерфейсов, данных, артефактов и т. п.) такие:

1) композиционность сложных систем типа РПС из компонентов, интерфейсов и сервисов с их свойствами, характеристиками и механизмами агрегации в более сложные структуры и правилами взаимодействия в интегрированных средах;

2) компонентность инженерии (CBSE), как деятельности создания РПС из готовых "деталей", базированная на реально существующих положениях инженерии продуктов, а именно, стандартизирован ЖЦ требований, эксплуатации и уничтожения КПИ или СПС с использованием систем классификации и катало-

гизации КПИ, средств унификации, стандартизации описания КПИ и интеграции их в ПС и СПС;

3) интероперабельность КП и элементов ПС, которая базирована на интерфейсах и правилах взаимодействия компонентов между собой для обеспечения интеграции и функционирования в разных гетерогенных средах;

4) вариантность как способность КПИ и компонентных ПС к изменениям путем уничтожения некоторых функциональных, незавершенных или добавления новых функциональных КПИ в конфигурационную структуру СПС, или ПС и т. п.

Сервис Интернета – это ресурс, который реализует некоторые функции (в том числе бизнес-функции), является КПИ и содержит в себе технологически независимый интерфейс с другими ресурсами. Например, сервисы транзакций, именования, безопасности в модели CORBA. Они образуют службу сервисов для создания ПС.

Основная форма реализации сервисов – это *веб-сервисы,* которые сохраняются и идентифицируются URL-адресами и взаимодействуют между собой посредством сети Интернета, например, RPC (Remote Procedure Call). Стремительное использование Интернета привело к тому, что традиционное интегрированное предприятие прошлых поколений все чаще замещается сетью бизнеса, которые совместно выполняют определенные функции при том, что и собственность, и менеджмент распределены между партнерами. Именно информационные потребности распределенных бизнесов вызвали к жизни веб-сервисы как адекватную форму компонентов типа КПИ.

Веб-сервис имеет URL-адрес, интерфейс и механизм взаимодействия с другим сервисом через протоколы Интернет или связки с другими программами, БД и деловыми бизнес-операциями. Обмен данными между веб-сервисом и программой осуществляется с помощью XML-документов, оформленных в виде сообщений. Веб-сервисы обеспечивают решение задачи *интеграции приложений* разной природы, являясь при этом инструментом построения распределенных систем. Веб-сервис предоставляется провайдером сети Интернет, который имеет стандартный способ взаимодействия с распределенными (.NET, J2EE, CORBA и др.) и прикладными системами для получения некоторых услуг.

Основные средства описания и разработки новых систем средствами веб-сервисов:

1) язык XML для описания и построения SOA-архитектуры;

2) язык WSDL (Web Services Description Language) для описания веб-сервисов и их интерфейсов в XML, а также типов данных и сообщений, моделей взаимодействия и протоколов связи сервисов между собой;

3) SOAP (Simple Object Access Protocol) для определения форматов запросов к веб-сервисам;

4) SCA (Servise-Component Architecture) для создания более сложной системы на основе компонентов и сервисов;

5) UDDI (Universal Description, Discovery and Integration) для универсального описания, выявления и интеграции сервисов, обеспечения их хранения, упорядочения деловой сервисной информации в специальном реестре с указателями на конкретные интерфейсы веб-сервисов.

К средствам моделирования сложных систем и СПС относится система IBM ® WebSphere ® Integration Developer. Она предоставляет сервис-ориентированную архитектуру SOA (Servise Oriented Architecture), компонентну архитектуру сервисов SCA (Servise-Component Architecture) на основе вариантов использования UML. Эта система предлагает интеграцию сервисов SCA через модель интерфейсов JAVA, которая задается в языке WSDL, а реализация – классы JAVA ™.

SCA – модель программирования, посредством которой можно получить доступ к сервисным компонентам и определить зависимости между ними через аппарат ссылок. Компоненты SCA вместе со связанными с ними зависимостями могут быть упакованы в модуль для выполнения сервисного модуля с WebSphere Process Server – эквивалентного EAR-файлу J2EE и некоторым другим. Подмодули J2EE и артефакты упаковываются с модулем SCA Это позволяет запустить сервис SCA через *модель программирования,* передавать данные при их интеграции. Механизмы, которые используются для вызова внешнего сервиса, названного *импортом* и *экспортом*, они связаны с другими технологиями, такими как JMS, Enterprise JAVABeans или веб-сервисы. SCA модуль позволяет обратиться к существующему Enterprise JAVABean, используя модель программирования SCA с целью релевантного представления данных по *универсальной модели данных*. Элементы SCA могут компоноваться и обмениваться данными друг с другом, пересылая SDO в необходимом виде. В этой модели объекты данных представлены в JAVA интерфейсы common.sdo.DataObject, который включает в себя метод, который позволяет пользователям получать свойства данных. WebSphere Integration Developer для разработки на платформе Eclipse 3.0 и их интеграции путем задания SCA модулей, сервисного интерфейса и реализации.

5.2. Сервисно-ориентированная архитектура

Сервисно-ориентированная архитектура (SOA) – это совокупность взаимодействующих между собой системных сервисов и веб-сервисов.

Любой из компонентов SOA создается сервисами безотносительно к конкретным технологиям, за которые можно принять готовые приложения типа "черный ящик". Интеграция компонентов и сервисов в архитектуру SOA включает в себя следующие виды:

1) *пользовательскую интеграцию* (user integration) для взаимодействия информационной системы с конкретным пользователем;
2) *связь приложений* (application connectivity) для задания взаимодействия;
3) *интеграцию процессов* (process integration) в бизнес-приложениях;
4) *информационную интеграцию* (information integration) для обеспечения доступа к интегрированной информации и данным.

При этом к создаваемой архитектуре SOA выдвигаются следующие требования:

1) наличие существующих информационных систем и появление новых;
2) поэтапное внедрение новых и миграция существующих ПС и ИС;

3) стандартизация технологии и реализация инструментов для поддержки сервисных архитектур с повторным использованием приложений и компонентов;

4) использование разных моделей и систем (порталы, grid-системы и др.).

Если объектом сервисно-ориентированной архитектуры является веб-сервис, то применяется две технологии, что обеспечивают функциональность (Functions) и качество сервисов (Quality service). Эти технологии вынесены на уровень IT-стандартов комитета W3C и имеют следующие уровни.

Технология обеспечения функциональности веб-сервисов включает в себя:

1) транспортный уровень (transport layer) для обмена данными;

2) коммуникационный уровень (service communication layer) для определения протоколов;

3) уровень описания сервиса (service description layer) и связанных с ним интерфейсов;

4) уровень бизнес-процессов (business process layer) для реализации бизнес-процессов и потоков работ через механизмы веб-сервисов;

5) уровень реестра сервисов (service registry layer), который обеспечивает организацию библиотек веб-сервисов для их публикации, поиска и вызова по их WSDL-описанию интерфейсов. Технология обеспечения качества веб-сервисов имеет следующие уровни:

6) политики (policy layer) для описания правил и условий применения веб-сервисов;

7) безопасности (security layer) для описания вопросов безопасности веб-сервисов и функционирования (авторизация, аутентификация и распределение доступа);

8) транзакций (transaction layer) для установления параметров обращения к веб-сервисам и обеспечению надежности их функционирования;

9) управление (management layer) веб-сервисами.

Технологический фундамент веб-сервисов составляют: XML, SOAP, UDDI, WSDL. С их помощью осуществляется реализация базовых свойств веб-сервиса и механизма взаимодействия между собой веб-сервисов в среде SOA;

1) провайдер сервиса осуществляет реализацию сервиса, прием и выполнение запросов пользователей, публикацию сервиса, от из реестра сервисов;

2) реестр сервисов содержится библиотеку сервисов для поиска и вызова сервиса по запросам от поставщика или провайдера сервисов, предоставляющих сервисы;

3) потребитель или пользователь сервиса осуществляет поиск и вызов необходимого сервиса из реестра описания сервисов, а также использует сервис, предоставленный поставщиком в соответствии с его интерфейсом.

Связь между поставщиком и потребителем (рис. 2.18) осуществляется через HTTP и XML-сообщения сетевой среды, которая использует интерфейсы веб-сервисов. Посредником между этими службами и приложениям является *провайдер*, который обеспечивает взаимодействие меэжу поставщиками и провайдерами с помощью средств описания и передачи сервисов WSDL, SOAP, XML.

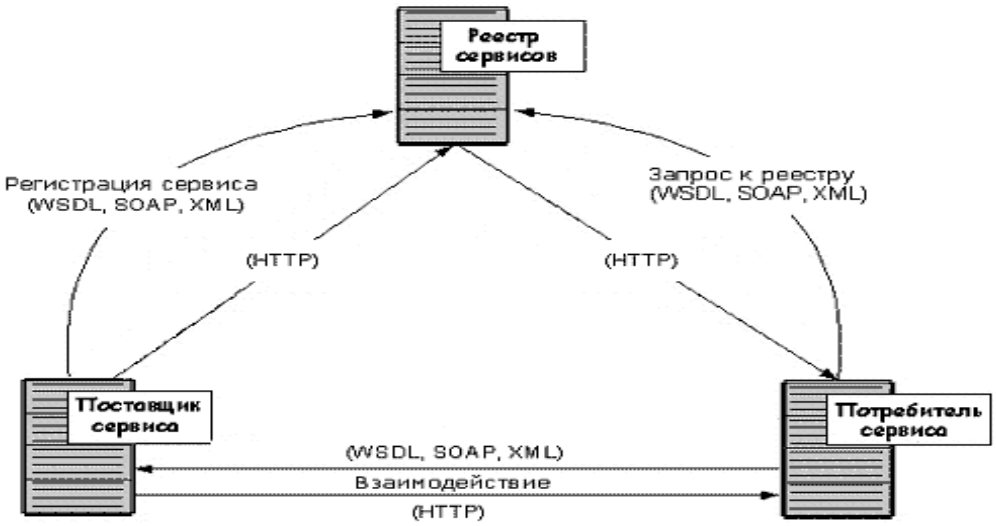


Рис. 2.18. Поставщики и потребители сервисов для взаимодействия

Операции SOA. Для получения сервиса в архитектуре SOA выполняются следующие операции:

1) публикация сервиса WSDL с целью обеспечения доступности (через вызов) пользователю сервиса и его интерфейса;

2) поиск по протоколу SOAP осуществляет пользователь сервиса в реестре сервисов по заданным критериям;

3) связь UDDI через описание пользователем необходимого сервиса, который может предоставляться в таких моделях как COM, CORBA, DBMS, .JNET и т. п.

При этом предусматривается, что в реестре архитектуры SOA содержится описание сервиса с форматом запросов пользователя к провайдеру, содержащему в себе перечень описаний сервисов, которые могут быть вызваны соответственно с опубликованным интерфейсом сервиса.

5.3. Сервисы контрактов WCF

Windows Communication Foundation (WCF) – программный фреймворк, предназначенный для обмена данными между приложениями входными данными, которые входят в состав .NET Framework. Он является логическим развитием предыдущих технологий компании Майкрософт, в частности веб-сервисов, .Net Remoting и DCOM.

В основе WCF лежит SOA, которая заключается в том, что на сервере работает некоторое количество сервисов, которые представляют собой группу операций, определенных в некотором интерфейсе, и которые получают абстрактные входные/исходные параметры. Все это описывается в WSDL (Web Service Description Language) и может быть выставлено вверх через, так называемые mex-endpoints (Metadata Exchange Endpoints). Это позволяет получить "метаданные" сервиса. Подключаясь к этому интерфейсу; можно получить описание сер-

виса и всех его операций, а также сгенерировать соответствующий прокси-класс для заданного языка или платформы. Сервис описывается в языке WCF, а использовать его можно с Java/Python/Ruby и и т. п.. Клиенты в свою очередь имеют на стороне прокси-классы, которые содержат ссылку на соответствующие операции на стороне сервиса.

В пределах WCF MS.Net действует технология и фабрики программ для нескольких веб-служб, а также фабрика сервисов. Основу технологии составляют схемы, "рецепты ", методы и средства построения фабрик разного назначения.

Фабрика программ сервисного типа включает в себя набор потребных ресурсов, блоков кода, документации, образцов приложений, автоматизированных инструментов и патернов VSIP для создания на их основе разных пакетов. Группа p&p Microsofts создает рекомендации, схемы и методы, а также стандарты их выполнения разработчиками готовой продукции. Фабрика сервисов дает рекомендации для их использования при проектировании и конструировании, которые накапливаются в Global Bank. К ним относятся ASP.Net для применения в WCF.

Фабрики программ и сервисов базируются на готовом наборе программных элементов и разного рода сервисов в MS.Net. Функционирование WCF зависит от так называемых конечных точек (endpoint), что составляет связь "Address - Binding - Contract", "ABC". Каждая составляющая играет важную роль:

"Address" содержит в себе место расположения конечной точки по абсолютному или относительному адресу;

"Binding" задает привязку и определят транспорт, на основе которого будет происходить взаимодействие. В объектной модели WCF есть ряд классов-привязок (BasicHttpBinding за HTTP, NetTcpBinding за транспортом TCP и т. д.;

"Contract" задает контракт, на основе которого будет происходить взаимодействие клиента и сервиса и определяет контракт с соответствующими операциями сервиса, который строит клас-проксі на стороне клиента.

Как правило, на сервисной стороне задается множество конечных точек. Кроме того, можно разместить немного сервисов на сервисной стороне. В результате, сервисная сторона распределенного приложения будет выглядеть как совокупность конечных точек. В этом случае клиент осуществляет соединение с той конечной точкой, которая ему необходимо.

Для построения распределенных средств как WCF или .Net Remoting используется принцип пирога из слоев, каждый слой которого отвечает за свой конкретный уровень абстракции и не знает ничего о нижележащих уровнях. То есть инфраструктура WCF состоит из двух главных уровней: Service Model Layer и Channel Layer. Первый уровень ближе к самому сервису клиента и обеспечивает превращение метода и его параметров в сообщение для передачи более низкому канальному уровню. Канальный уровень (Channel Layer) икапсулирует в себе канал передачи данных, которых может быть множество: каналов, которые используют как транспорт TCP, Http, Named Pipes и т. д. Каждый из этих уровней содержит подуровни, и может включится в каждый из них.

Контракты представляют собой описание сообщений, переданных конечными службами с возвратом. Конечная точка должна специфицироваться и может выполняться в формате ожидаемых данных. Совокупность этих спецификаций и есть контракт.

WCF содержат три вида контрактов:

1) сервисов для описания функциональных операций, реализованных сервисом. Внутри контракта сервиса имеются контракты об операциях, как отдельные операции сервиса, оторые реализуют функции;

2) данных, определяющих формат данных, которыми будут обмениваться сервисы. Это относится как к запросу на сервис, так и к октету сервиса. Если используются примитивные типы – int, string и др., то контракт не нужен, потому что .Net понимается как сериализация и дисериализация типов. В случае применения комплексных типов – Customers, Order и др., н еобходимо указать принцип сериалиализации и дисериализаии этих объектов;

3) сообщений, как тип контракта, который используется для того, чтобы получить контроль над заглавком SOAP пакета.

Пример описания сообщения в языке XML.

<?xml version="1.0" ?>

<env:Envelope xmlns:env="

http://www.cbsystematics.com">

<!-Конверт протокола SOAP-> <env:Header>

<!-Заглавие протокола SOAP-> </env:Header> <env:Body>

<!-Тело протокола SOAP-> </env:Body> </env:Envelope>

Для обеспечения интероперабельности контрактов с широким диапазоном систем, используются языки WSDL и XSD, в этом случае программа работает с типами CLR и необходимо отобразить одну систему типов на другую.Такая задача решается в три этапа.

Сначала при написании кода службы поставляется класс с определенными в WCF атрибутами [ServiceContract], [OperationContract], [FaultContract], [MessageContract] и [DataContract].

При написании клиентского кода в сервисе приводятся детали контракта посредством Visual Studio или утилиты svcutil.exe, которая вызывает инфраструктурную конечную точку сервиса для возврата данных, необходимых для генерации WSDL документа при получении их атрибутов.

На этапе выполнения клиента вызывается метод, определенный в интерфейсе сервиса, WCF сериализует типы CLR и вызов метода в формат XML и посылает сообщение в сеть для привязки и схемы кодировки, согласованной с WSDL. При этом участвуют четыре конструкции: две со стороны .NET и две со стороны XML. Со стороны .NET имеется тип CLR, который определяет структуры данных и функциональные возможности, но это делается лишь после того, как создан объект такого типа. Со стороны XML задается XSD-описание структуры данных, но сообщение осуществляется лишь после того, как будет создан экземпляр XML (XML Instance). Пример описания WCF сервиса подготовленный и апробированный студентом КНУ И.Радецьким в рамках магистровской диссертации (2012).

5.4. CASE-средства JAVA EE

JAVA Entreprise Edition (EE) содержит набор спецификаций, документаций, которые необходимы при работе с сетевыми программами и сервисными средствами для автоматизации некоторого предприятия.

Веб-сервис (веб-служба) идентифицируется URI (Unified Resource Identifier, универсальный идентификатор ресурсов), ресурсы которой (свойства и методы) описаны посредством специального языка. Доступ к ресурсам осуществляется через протокол SOAP (Simple Object Access Protocol), который представляет собой XML-запросы, передающиеся через Интернет-протоколы высокого уровня HTTP. По своей архитектурой веб-сервисы напоминают классы объектно-ориентированных ЯП (и в пределах JAVA EE генерируются на основе классов), но есть и определенные разтличия.

Ключевым понятием этого веб-сервиса является сообщение (message), которое состоит из одной или нескольких переменных. Вместо методов классов в веб-сервисах используются операции, которые определяются входным и выходным сообщениями. После вызова операции переменные, которые входят во входное сообщение, полученное по протоколу SOAP, интерпретируются как параметры соответствующего метода класса, который лежит в основе сервиса. После завершения работы метода формируется исходное сообщение, которое содержит в себе возврат значения метода, которое после этого через SOAP отправляется клиенту. Такая архитектура сервиса позволяет вызвать асинхронный метод.

Общение с веб-сервисом осуществляется с помощью протокола SOAP. Для вызова операции веб-сервиса, определенной в его WSDL-описании, используется XML-запрос, который называется SOAP-envelope и состоит в общем случае из заголовка (SOAP-header; может быть пустым) и тела запроса (SOAP-body). Заголовок может содержать дополнительную информацию для запроса, такую как его приоритетность, срок обработки и т. п. В теле содержится одна или несколько операций веб-сервиса с соответствующими параметрами.

Пример запроса, который вызывает операцию сервиса MyService.someMethod (32.5, true), приведено ниже, там же приведенно и SOAP-сообщение, возвращенное сервисом.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:q0="http://webservice.demo"xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Body>
<q0:someMethod>
<q0:arg0>32.5</q0:arg0>
<q0:arg1>true</q0:arg1>
</q0:someMethod>
</soapenv:Body>
</soapenv:Envelope>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Body>
<someMethodResponse xmlns="http://webservice.demo">
<someMethodReturn>Nothing to see here</someMethodReturn>
</someMethodResponse>
</soapenv:Body>
</soapenv:Envelope>
```

Сервлеты и язык WSDL. Для доступа к веб-сервисам используются сервлеты – JAVA-программы, которые выполняются на стороне сервера и расширяют его функциональность. Сервлети широко применяются в технологии JAVA EE, в частности, в них превращаются страницы на языке JSP. Базовая функциональность сервлетов, заложенная в их возможности обработки HTTP-запросов GET и/или POST. Согласно спецификации веб-программ, существует возможность задавать (через конфигурационные XML-файлы) границы действия отдельных сервлетов (servlet mapping). Решение применять сервелеты, принимается на основе анализа URL-адреса запроса. Например, веб-сервисы, расположенные в директории /services проекта, обрабатываются стандартным сервлетом Apache Axis. С его помощью можно получить WSDL-описание сервиса, расположенное по адресу /services/[название сервиса]?wsdl. Этот же сервлет выполняет парсинг, отправленных на сервер SOAP-сообщений, их передачу соответствующему JAVA-классу и составление SOAP-сообщений с результатами работы. Таким образом, Axis обеспечивает полную функциональность веб-сервиса [17].

Язык WSDL. Для описания общедоступных ресурсов веб-сервиса используется язык WSDL (Web Service Definition Language) на основе XML; среда программирования Eclipse позволяет автоматически создавать такие описаний на основе классов JAVA. В языке определены следующие основные типы данных:

1) строки (xsd:string);
2) целые числа (xsd:int, xsd:long, xsd:short, xsd:integer, xsd:decimal), числа с плавающей запятой (xsd:float, xsd:double);
3) логический тип (xsd:boolean);
4) последовательности байт (xsd:base64Binary, xsd:hexBinary);
5) дата и время (xsd:time, xsd:date, xsd:g);
6) объекты (xsd:anySimpleType).

В качестве переменных для сообщений также можно использовать последовательности, созданные из фиксированного количества переменных простых типов (это отвечает методам с несколькими входными параметрами).

Типичный WSDL-файл имеет следующую структуру.

```
<wsdl:definitions [.]>
<!-- Декларация типов, которые используются в сервисе —>
<wsdl:types>
<element name="someMethod">
<complexType>
<sequence>
<element name="arg0" type="xsd:double"/>
<element name="arg1" type="xsd:boolean"/>
</sequence>
</complexType>
</element>
<element name="someMethodResponse">
<complexType>
<sequence>
<element name="someMethodReturn" type="xsd: string"/>
</sequence>
```

</complexType>
</element>
</wsdl: types> …

Сначала декларируются типы, которые будут использоваться в сервисе. Приведенное ниже WSDL-описание принадлежит веб-сервису MyService с единственным методом String someMethod(double arg0, boolean arg1), на основе этого были згенерованы два типа данных, которые отвечают входным и исходным аргументам метода. Эти типы применяются в описаниях someMethodRequest и someMethodResponse – входного и выходного сообщений для операции someMethod.

Операции декларируются в описании интерфейса сервиса (декларация wsdl:portType) и дальше в описании привязки сервиса к SOAP (декларация wsdl:binding), причем во втором случае также оговаривается способ вызова (<wsdlsoap:body use="literal"/>). За счет этого позволяется при вызове операции использовать те же названия параметров, что и в методе класса. В конце WSDL-файла находится собственно декларация веб-сервиса (<wsdl:service>), в которой содержится информация относительно его расположения (параметр location).

Механизмы взаимодействия с веб-сервисами. Среда программирования Eclipse предназначена для работы с сетевыми прикладными программами, в MS.NET и обеспечивает возможность создания клиентов для веб-сервисов на основе WSDL-описания. При этом скрыта большая часть черновой работы по обеспечению функционирования клиента, например, составление корректных SOAP-запросов и парсинг возврата сервиса SOAP-сообщений. Эти действия выполняются клиентской частью технологии Apache Axis. Для обеспечения работы клиента создаются JAVA-файлы:

1) локатор сервиса (service locator) выполняет нахождение веб-сервиса;
2) интерфейс локатора;
3) стаб для SOAP-привязки (SOAP binding stub) – клиентский стаб, предназначенный для составления и парсинга SOAP-сообщений;
4) интерфейс сервиса;
5) прокси-класс, который реализует этот интерфейс; использует клиентский стаб и локатор для доступа к операциям веб-сервиса.

Например, для вышеупомянутого сервиса MyService автоматически созданные классы и интерфейсы называются MyServiceServiceLocator, MyServiceService, MyServiceSoapBindingStub, MyService, MyServiceProxy. Обращение к операции MyService.someMethod имеет следующий вид:

```
My Service svc = new MyServiceProxy ();
Try {System.out.println (svc.someMethod(32.5, true));
} catch (Remote Exception e) {
    System.err.println("Error occurred while accessing web service");
        e.printStackTrace(); }.
```

Таким образом, среда MS.NET дает возможность клиентам создавать веб-сервисы.

Раздел 3
ТЕХНОЛОГИЯ СИСТЕМ, ЛИНИЙ И CASE-СРЕДСТВ

Глава 1. ТЕХНОЛОГИЯ СЛОЖНЫХ СИСТЕМ ИЗ ГОТОВЫХ РЕСУРСОВ

В настоящее время сформировались теоретические, методические и практические подходы к производства прикладных ПС. Построены автоматизированные инструментальные средства и среды (Microsoft Visual Studio, MSF, Rational Rose, COM, CORBA и т. п.) для разработки и производства сложных ПС из готовых ресурсов (компонентов, reuses, assets, сервисов и т. п.) и унаследованых программ (Legacy system). В названных системах эти ресурсы реализуются с использованием сборочной технологии., которая базируется на ЯП и предметных языках типа DSL, моделях стандартного ЖЦ, методах генерации (трансформации или конфигурации), методике тестирования готовых ресурсов и сложных ПС, оценке рисков и отдельных показателей качества компонентов, ПС и СПС.

Вопросы изготовления ПС из готовых ресурсов. Методология проектирования и реализации ПС и СПС методом сборочного программирования в ГП с использованием языка DSL для задания модели домена, *готовых программных ресурсов*, которые будут использоваться при проектувании структуры домена и применения CASE-инструментов (трансляторы с ЯП, тестирование, трасформеры, генераторы и т. п.) для выполнения задач производства членов семейства и СПС в целом. Для разработки компонентов и СПС создана интегрированная среда инструментальных средств, включая Eclipse, Protege, VS.Net, CORBA, JAVA и т. п. Главным регламентом при разработке избран ЖЦ стандарта 12207, как с технологической точки зрения, так и организационной.

Под *готовыми ресурсами* понимается разная форма представления КПИ (reuse, assets, artifacts, services и т. п.), которые отображают накопленный опыт построения некоторых системных функции ПрО для современных проблемных областей. Каждый КПИ специфицируется соответствующими стандартами путем описания данных в паспорте специального вида.

Паспорт – это информационная часть некоторого программного компонента и по существу является описанием интерфейсных параметров для передачи данных другому компоненту и получения от него результата. Это обеспечивается разными технологическими линиями разработки отдельных частей ПС и СПС средствами инструментально-технологического комплекса (ИТК) (рис. 3.1).

Основные направления методологии проектирования ПС и СПС такие:

1) проектирование ПС с использованием процессов стандартного ЖЦ и моделей MDD, MDA и др.;

2) онтология проектирование доменов с заданием модели характеристик сущностей модели доменов и создания архитектуры системы из готовых компонентов, ресурсов средствами сборочного конвейера;

3) спецификация разнородных программных ресурсов в ЯП, их реализация, тестирование с целью проверки правильности и накопления верифицированного компонента в репозитории системы вместе с паспортом;

4) отбор готовых компонентов в репозитории средствами созданной фабрики программ и сбор разнородных КПИ в новых ПС для реализации некоторых задач автоматизуемой ПрО;

5) генерация из некоторых асетов из артефактов исходного кода и адаптации их конкретных целей ранее созданного программного решения или программы;

6) трансформация с описанием специфики ПрО графическим языком DSL и использованием DSL TooLs VS.Net для получения исходного кода созданной объектно-компонентной модели;

7) тестирование КПИ, ПС и сбор необходимых данных для оценки качества ПС;

8) инженерия качества ПС, включая экспертный и метрический анализ показателей качества, а также оценки достигнутых показателей качества ПП;

9) сохранение результатов проектирования в репозитории компонентов и интерфейсов;

10) документирование КПИ, новых компонентов в составе ПС.

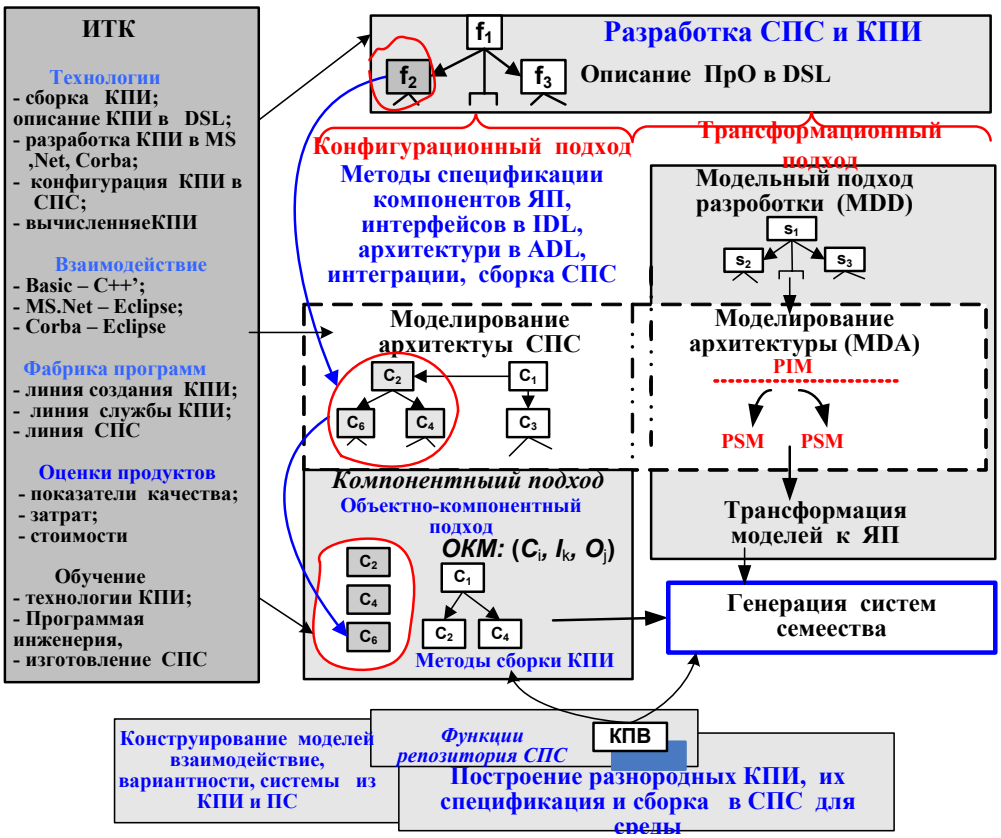


Рис. 3.1. Общая схема информационной среды ИТК

1.1. Базовые подходы к проектированию сложных систем

Модели MDD, MDA, MGD разработки программных систем

Моделирование ПС и их семейств выполняется средствами компонентного программирования в части построения отдельных КПИ и их сборки в конфигурационный вариант ПС или СПС на основе спецификаций компонентов в пространстве проблем. Это описание базируется на подходах FOD (Feature-Oriented Development) построения МХ ПрО, MDD (Model Driven Development) и MDA (Model Driven Architecture) и др. Архитектура системы может выполняться с помощью платформо-независимой модели PIM (Platform Independent Model) и платформо-зависимой модели PSM (Platform Specific Models). Концепции двухуровневого моделирования архитектуры MDA (Model Driven Architecture) и отображения PIM↔PSM соответствует концепции отображения пространства проблемы в пространстве решений [91 – 94] .

Модель MDD. Базируется на унифицированном языке моделирования UML с графической формой представления семантики. В MDD определяется система с четко определенной семантикой, и ее артефакты могут быть преобразованы, трансформированы к коду реализации (аналогично тому, как код JAVA компилируется в байт-код). При MDD определяется модель семейства ПС (Application Model), члены которой имеют общие функции, но зависят от платформы реализации компонентов и данных. Преимущество применения MDD для построения членов семейств заключается в том, что "управление" точкой вариантности для платформы выполняется автоматически путем трансформации моделей PIM - PSM. Члены семейства различаются не только на уровне платформы реализации, но и на уровне функций ПС, требований, а также достижения качества ПС.

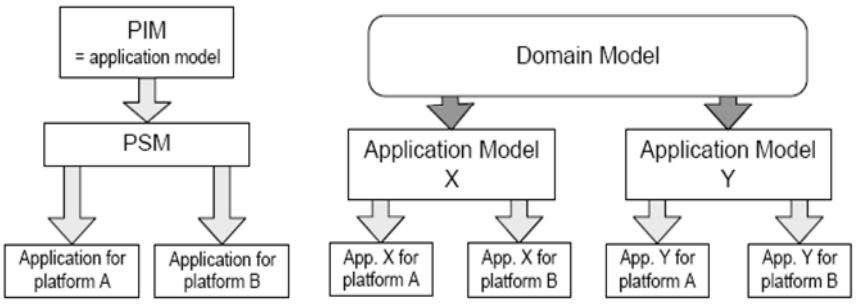
Приспособление подхода MDD к потребностям приложения может быть выполнено путем добавления к модели ПрО вариантов *точек* для управления альтернативными концепциями вариантной конфигурационной структуры. Это особенно необходимо при их уточнении или изменении некоторых КПИ в ПС. Выбор разных концепций представления модели ПрО будет предопределять появление разных прикладных моделей ПС, которые, при генерации, автоматически трансформируются к модели MDD.

Модель MDA. В основе этой архитектуры лежит идея разделении этапов моделирования и последующей реализации приложения на конкретной платформе. Сначала с помощью специальных средств создается общая и независимая от способов реализации модель приложения, а затем осуществляется ее реализация в некоторой среде разработки.

Архитектура MDA возникла на основе наличия ряда стандартов и технологий. Концептуальная основа MDA – спецификации OMA, ORB, CORBA, ООП, стандарт CWM, языки UML, XML, MOF.

MDA – это новый виток эволюции технологий программирования в плане описания процесса разработки в целом и с использованием современных средств представления, что позволяет автоматизировать создание приложений. MDA станет промышленным стандартом в ТП.

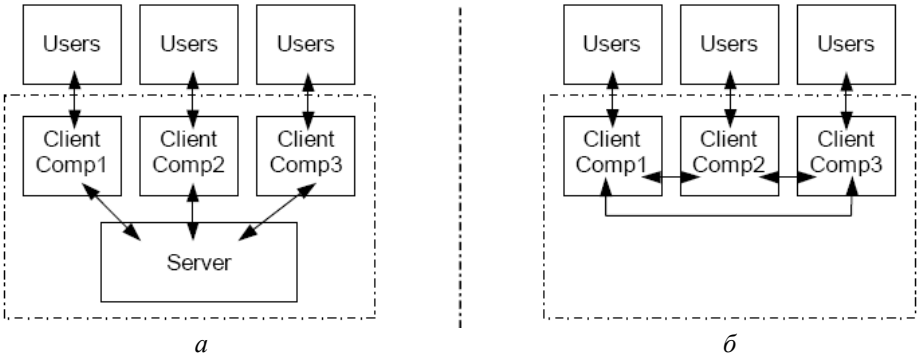
В классической модели MDA действует схема (рис. 3.2,*а*). С учетом потребностей доменов эта схема модифицируется (рис. 3.2,*б*).



а *б*

Рис. 3.2. MDA -два варианта (*а*, *б*) семейств систем

Фактически спецификация ПС соответствует подходу MDD (рис. 3.3, *а*) определяет модель семейства ПС (Application model), члены которого имеют общие функции, но выделяются платформами реализации. Как это видно на рис. 3.3, где показаны два способа реализации информационного взаимодействия прикладных применений. Выбор альтернативных платформ связан с "точкой вариантности" в семействе, которая находится над моделью ПС и она невидимая на уровне конкретной ПС из семейства СПС.



а *б*

Рис. 3.3. Два подхода к реализации взаимодействия приложений:
а- централизованный доступ, *б*- взаимодействие парами

Преимущество применения MDD для построения члена семейств заключается в том, что "управление" точкой вариантности платформ выполняется через трансформацию PIM↔PSM.

Члены семейства различаются не только на уровне платформы реализации, но и на уровне функций ПС, требований к качеству и инфраструктуре, а также активов, которые реализуют альтернативные концепции. Приспособление подхода MDD к потребностям ГП может быть выполнено путем добавления *модели ПрО*, которая будет определять точки, где могут быть выбраны альтернативные концепции.

Выбор разных концепций из модели ПрО предусматривает появление разных прикладных моделей ПС, которые, при правильной реализации трансформации автоматически в рамках традиционного подхода MDD (Vodel Domain Development).

Таким образом, преимущества этого подхода следующие: независимость модели от средств разработки; возможность переноса из одной операционной системы в другую и экономия ресурсов приложения одновременно для нескольких платформ; создание частей приложения.

1.2. Модели систем для разных платформ

Проектирование ПС проводится, как правило, с помощью стандартных моделей и моделей ЖЦ (спиральной, водопадной, итерационной и др.). К стандартным моделям относятся MDA, MDD, GDM, SOA и др. Эти модели рассмотрены выше. Здесь дается характеристика моделей, которые могут адаптироваться к разным платформам компьютеров.

Платформенно-зависимая модель PSM (Platform Specific Model**)** задает состав, структуру, функциональность системы применительно к конкретной платформе. Модель платформы задает технические характеристики, интерфейсы, функции платформы. Она используется при преобразовании модели PIM в модель PSM.

В зависимости от уровня детализации платформы, модели могут содержать сведения о различных функциональных частях системы. Уровень бизнес-логики содержит описание основного функционала приложения, обеспечивающего исполнение его назначения. Уровень данных описывает структуру данных приложения, используемые источники, форматы данных, технологии и механизмы доступа к данным. Для приложений .NET чаще всего применяются возможности ADO.NET. Уровень пользовательского интерфейса содержит состав форм приложения, функциональные элементы управления. Легкость автоматизации этого уровня зависит от того, насколько унифицированы пользовательские операции. Если удается создать типовые шаблоны элементов управления для основных операций, то появляется возможность автоматической генерации форм и их содержания при создании приложений.

Вычислительно-независимая модель CIM **(**Computation Independent Model**)**, структурная схема которой приведена на рис. 3.4, базируется на модели PIM **(**Platform Independent Model) для ее преобразования к платформенно-зависимой модели с использованием языка моделирования UML.

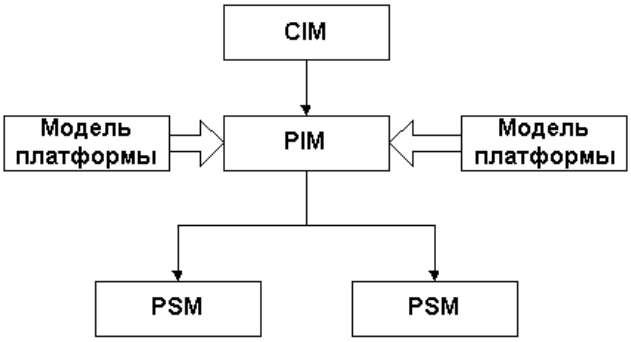


Рис. 3.4. Структурная схема CIM

Эта модель включает в себя разработку общих требований к системе, создание общего словаря понятий и описание окружения, в котором система будет функционировать. Сущности и понятия, описываемые в модели CIM, должны тщательно анализироваться и отрабатываться. Модель CIM должна отображать общую концепцию системы, используемую для программирования элементов приложения. Преобразование CIM в платформенно-независимую модель (PIM) осуществляется средствами языка UML. Она включает в себя элементы, описывающие бизнес-логику, общую структуру системы, состав и взаимодействие подсистем, распределение функциональности по элементам и требования к пользовательскому интерфейсу. Модель PIM включается во все автоматизированные среды разработки приложений, использующих модель MDA.

Переход к платформенно-зависимой модели определяется числом программных платформ, на которых будут функционировать элементы приложения. Рассматриваются случаи, когда приложение (или его составные элементы) должны работать на нескольких платформах одновременно. Модель PSM создается путем преобразования модели PIM к модели другой платформы. На этапе создания модели PSM разработка приложения осуществляется с помощью модели MDA, которая преобразуется к коду другой платформы.

Преобразование моделей PIM→PSM. На этом этапе общее описание системы на языке UML включает выполнение следующих действий:

1) разработка схемы преобразования (mapping),
2) маркирование (marking),
3) трансформация (transformation).

Для каждой платформы создается собственная схема преобразования, зависимая от возможностей платформы. (формата, языков, средств обработки и др.). Схема преобразования затрагивает как содержание модели (совокупность элементов и их свойств), так и механизмы ее представления (метамодель, используемые ТД, форматы данных платформы). В схеме преобразования свойствам метамодели и элементам модели PIM ставятся в соответствие свойства метамодели и элементы платформо-зависимой модели PSM.

При преобразовании PIM→ PSM моделей может использоваться несколько схем преобразования. Для их связывания применяется марка (mark), как самостоятельная структуры данных, принадлежащая не моделям, а схемам преобразования, содержащая информацию о созданных связях (рис. 3.5). Процесс задания марок называется маркированием. Простейшим случаем является элемент модели PIM, который соединяется маркой с одним элементом модели PSM. В более сложных случаях один элемент модели PIM может иметь несколько марок из разных схем преобразования этой модели. В процессе маркирования используются сведения о платформах, которые содержатся в модели платформы.

Процесс преобразования моделей заключается в переносе маркированных элементов модели и метамодели PIM в модель и метамодель PSM. Процесс преобразования документируется картой переноса элементов модели и метамодели.

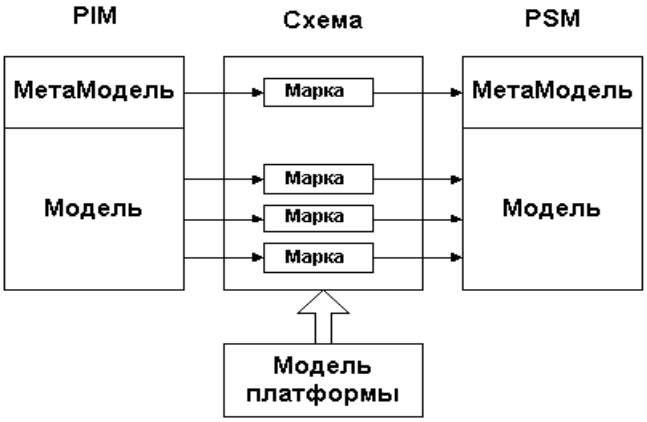


Рис. 3.5. Преобразование моделей по схеме

1.3. Генерация и сборка сложных систем

Система генерации базируется на специальных инструментах DSL TooLs VS.Net, которые трансформируют описание модели домена GDM к необходимому исходному коду, а также на Protege, как онтологический механизм построения моделей некоторых проблемных областей.

Для ПрО могут разрабатываться:

1) специализированные домены, которые отражают производственные интересы специалистов определенного профиля в рамках проекта или некоторой отрасли бизнеса;

2) домены общего использования, которые характеризуют независимые от специализированных ПрО общие процессы ЖЦ (определение требований, проектирование функций и архитектуры, кодировка, тестирование, документирование и т. п.) и организационные процессы управления проектом, обеспечения защиты данных и авторизованного доступа к ним и др.

Готовые ресурсы и КПИ при занесении в репозиторий специфицируются, как правило, по таким категориям:

1) прикладное приложение или домен;
2) процессы ЖЦ разработки и генерации;
3) типизованные КПИ.

Процесс генерации компонентов начинается с поиска готовых КПВ по требованиям заказчика, которые отображают цели построенного ПС и полученного ПП. После поиска необходимого КПИ принимается решение о достаточности или недостаточности его выполнения в СПС и объединения программных ресурсов в члены СПС.

CASE-сборки

Сейчас известны средства поддержки интеграции или сборки разных программ. Среди них системы – **Make, Apache Ant, Apache Maven** и др.

Make – это кросплатформенная система автоматизации сборки ПС из исходного кода. Она генерирует файлы управления сборкою, например Makefile в системах Unix для сборки посредством make.

Apache Ant – JAVA-утилита для автоматизации процесса сборки программного продукта. Ant – платформонезависимый аналог UNIX-утилиты Make, но с использованием языка JAVA приспособленный для JAVA-проектов. Самая важная непосредственная разница между Ant и Make состоит в том, что Ant использует XML для описания процесса сборки и его зависимостей, тогда как Make имеет свой собственный формат Makefile. По умалчиванию XML-файл, называемый build.xml, осуществляет сборку.

Apache Maven – программный инструмент для управления (management) JAVA проектами и сборщика (build) разных программ. По принципам функционирования он подобен Apache Ant, но имеет более простую build-модель конфигурации, которая базируется на формате XML. Двигатель ядра может динамически загружать плагины с репозиторию, который обеспечивает доступ до многих версий разных JAVA-проектов с открытым кодом от Apache и других организаций и отдельных разработчиков.

Gradle – система автоматической сборки, построенная на принципах Apache Ant и Apache Maven, но в отличие от них представляет DSL на языке Groovy вместо традиционной XML-подобной формы представления конфигурации проекта.

Способом описания процесса сборки, тестирования и других процессов ЖЦ является новый язык BPMN.

1.4. Методология проектировании систем с помощью ЖЦ

Методологии сборочного проектирования базируется на основных процессах ЖЦ стандарта ISO/IEC 12207 (требования, проектирование, конструирование, тестирование и сопровождение), а также на процессах поддержки и организации изготовления производства программ (экспертиза, верификация, тестирование), сбора данных об ошибках и отказах, которые используются при оценки разных показателей качества разработанного ПП, в частности надежности. Кроме того, в методологию входят методы проектирования моделей доменов, а также методы достижения качества, оценки стоимости выполненных работ и расходов на разработку [16, 17, 39, 100 – 102].

Построение формальной модели домена выполняется методами онтологии путем аккумуляции знаний о моделях, ресурсах, доменах, которые созданы в некоторой операционной среде.

В парадигме генерирующего прогаммирования главным аспектом производства программ является генерация, которая базируется на представлении знаний о специфике ПрО и накопленных знаний, а также на методах, средствах и инструментах ПИ, которые необходимы для линий автоматизированного изготовления отдельных ресурсов (компонентов) и ПС.

Процесс генерации рассматривается как последовательная трансформация промежуточных продуктов одного процесса ЖЦ в продукты следующего процесса путем использования систем программирования и других систем трансформации предметно-ориентированных описаний членов ПС с применением КПИ.

Идея применения КПИ на разных процессах ЖЦ от формулировки постановки задачи к исходному коду методом сбора их в ПС, который реализуется некоторую задачу, является особенностью метода.

При проектировании систем используются модели ЖЦ и стандарт ISO/IEC 12207-ЖЦ. Стандартный ЖЦ Іявляется общим механизмом регламентированного построения разных ПС. Последний вариант этого стандарта (2007) (табл. 3.1) включает в себя 17 процессов, 74 подпроцессов и 232 технологических операционных задач (действий). Их необходимо и достаточно для проектирования систем с помощью процессного подхода. Некоторые системные фирмы поддержки реализуют отдельные фрагменты или варианты этого стандарта.

Таблица 3.1. **Процессы, подпроцессы и задачи ЖЦ**

Классы	**Процесс**	**Действие**	**Задача**
Основные процессы	5	35	135
Процессы поддержки	8	25	70
Организационные процессы	4	14	27
Всего	17	74	232

Каждый коллектив разработчиков может использовать этот стандарт в качестве базиса и создать на его основе некоторое необходимое подмножества процессов как стандарт предприятия. Тогда потребуется сделать для него автоматизированную поддержку. Нами предлагается концепция автоматизации стандарта ЖЦ с применением концепции онтологии и соответствующих инструментальных средств. Это позволит поднять уровень реализации систем на индустриальной основе в фирмах разработчиков ИТ-технологий и ПП разного назначения.

Концепция автоматизации стандарта ЖЦ средствами онтологии является новой. В основе реализации лежит структура процессов ЖЦ и их взаимосвязи, а также подход к генерации отдельных вариантов рабочих ЖЦ для конкретных применений [3].

Средствами представления процессов ЖЦ могут быть: языки OWL (Web Ontology Language), ODSD (Ontology-Driven Software Developmen), XML (Extensible Markup Language); действующие системы моделирования доменов – ODM (Organizational Domain Modeling), FODA (Feature-Oriented Domain Analysis), DSSA (Domain-Specific Software Architectures), DSL (Domain Specific Language), Eclipse-DSL Tools VS.Net, Protege и т. п. Иными словами, имеются разнообразные языковые и технологические средства формального описания процессов ЖЦ для последующего автоматизированного моделирования разных программных продуктов.

В настоящее время имеются новые средства – язык MBPN для описания процессов ЖЦ и язык DSL для описания семантики доменов. В качестве примера реализации процессов ЖЦ избран онтологический подход. В нем ЖЦ представляется с помощью словарей понятий, концептов и отношений между ними в среде Protégé, DSL Tool VS.Net и др. В них онтологическое описание трансформируется к языку XML, который является языком реализации размеченных данных домена ЖЦ, установленных связей и обменов данными между процессами.

Концепция онтологизации ЖЦ обсуждалась в КНУ на научных семинарах кафедр ТТП и ИС, а также в группах студентов, которые читаются лекции по предмету "Программная инженерия".

Домен ЖЦ занимает центральное место в программной инженерия, основным назначением которой есть методы и средства изготовления сложных ПС. Студенты изучают эти методы и средства, а также современные стандарты ЖЦ ISO/IEC 12207–2007 и ISO/IEC 11404–2006. GDT (General Data Types). С участием студентов был разработан экспериментальный вариант онтологии ЖЦ средствами открытых инструментов – DSL Tools VS.Net и Protege [4 –6].

Структура ЖЦ стандарта ISO/IEC 12207–2007

ЖЦ в данном стандарте представлен тремя категориями [5]:

1) основные процессы;
2) процессы поддержки;
3) организационные процессы.

Для каждого из процессов определены виды деятельности (действия – activity), задачи, совокупность результатов (выходов) деятельности и решения задач и др. В стандарте приведен перечень работ для этих процессов, но не задан способ их выполнения и форм представления результатов. Основные процессы – разработки, эксплуатации и сопровождения ПС (рис. 3.6).

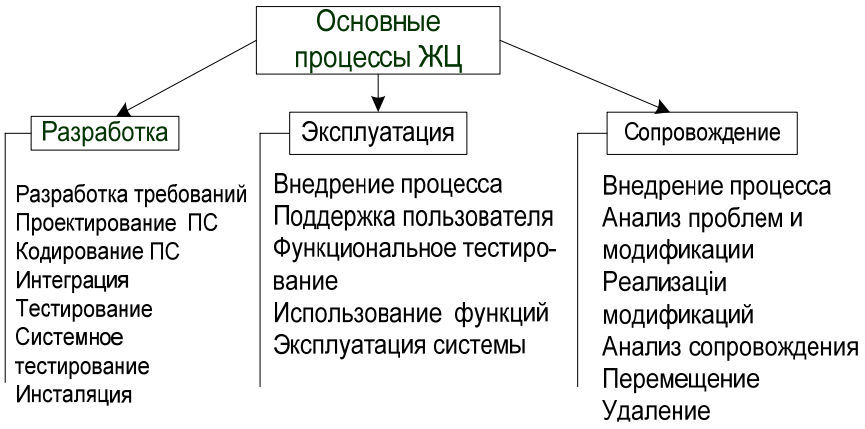


Рис. 3.6. Схема основных процессов ЖЦ ПС

Стандарт ЖЦ содержит в себе также вспомогательные процессы, которые регламентируют дополнительные действия по проверки продукта, управления проектом и качеством (рис. 3.7).

Как правило, в зависимости от целей конкретного проекта на ПП главный разработчик и менеджер выбирают процессы, действия и задачи, выстраивают определенную схему ЖЦ для применения в конкретном программном проекте.

Описание семантики процессов, парадигм и методов их выполнения (объектные, компонентные, сервисные и др.) приведены в ядре знаний SWEBOK (www.swebok.com) [2]. В каждой технологии программирования сложных ПС с использованием стандарта ЖЦ применяются теоретические, прикладные методы, стандарты качества, общие и фундаментальные ТД (ISO/IEC 15404, ISO/IEC 9126, ISO/IEC 11404 GDT и др.), а также методики этих стандартов.

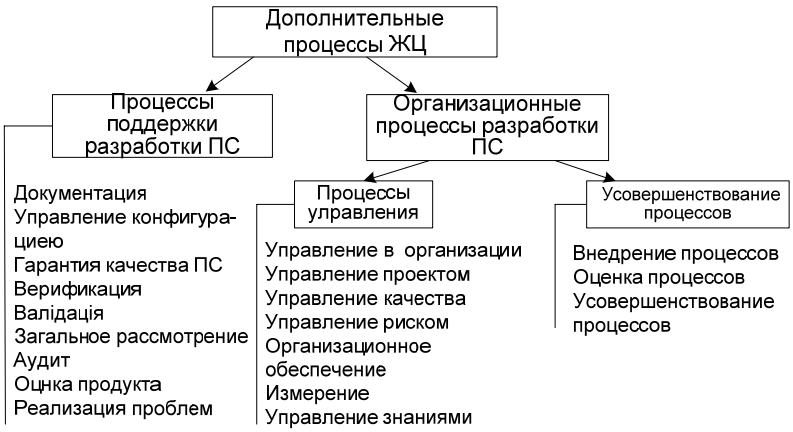


Рис. 3.7. Схема вспомогательных процессов ЖЦ ПС

Задача автоматизации стандартного ЖЦ возникла при выполнении фундаментального проекта по ТП (2007–2011) и разработке ИТК. Была постелена цель – автоматизировать ЖЦ и обеспечить генерацию разных его вариантов при изготовлении отдельных ПС из готовых ресурсов. Первый эксперимент по реализации ЖЦ проведен с участием студентов КНУ 4 курса кафедр ИС, ТТП, МФТИ и 2 аспирантов. Участники разработки изучили современные онтологичные средства и средства визуального представления процессов ЖЦ – WWF (Windows Workflow Foundation), DSL Tools VS.Net, Protégé и др. На основе этих средств было реализовано описание онтологии ЖЦ в графическом и XML видах в рамках систем DSL Tools VS.Net и Protégé.

Глава 2. МОДЕЛИРОВАНИЕ ДОМЕНОВ СРЕДСТВАМИ ОНТОЛОГИИ

В основе представления любой системы знаний лежит понятийная база, совокупность концептов (понятий) и отношений между ними. Следующим шагом нормализации знаний стало появление зафиксированной классификации понятий в виде тезауруса [91].

Процесс построения модели проблемы, ориентированной на ее понимание человеком, называют *концептуальным моделированиям.*

Каждая проблемная область или *домен* имеет присущую ему систему понятий, систему "умалчивания" того, что должен считаться "общеизвестным" в рамках своего домена, собственные характерные свойства (атрибуты), отношения и правила поведения. Фактически концептуальная модель это посредник между заказчиком и разработчиком. Степень ее формализации должна быть достаточной, чтоб обеспечивать точность и однозначность толкования разными носителями интересов в ее разработке и доступность для понимания ими.

Описание онтологии домена. Модель домена включает в себя его понятийную базу, т. е. систему понятий, посредством которой формулируются все

аспекты проблемного пространства. Она определяет не только терминологию, но и существенные отношения между понятиями и их интерпретацию.

В самом общем случае, *онтология* – это соглашение об общем использовании понятий, средств представления предметных знаний и договоренности о методах рассуждений. Онтология подается семантическими сетями, узлы которых являются понятиями, а дуги – связями или отношениями, или ассоциациями между ними.

Содержание понятия может быть охарактеризовано совокупностью общих существенных признаков тех явлений и предметов, отражаются в названии понятий. Совокупность явлений, охваченных понятиям, называется *его объемом*. Отношения обеспечивают:

1) обобщение существенных признаков понятия, как механизма расширения круга охваченных понятиями объектов и объема;

2) *конкретизацию* понятий путем добавления существенных признаков, благодаря чему содержание понятия расширяется, а объем понятия сужается;

3) *агрегацию,* т.е. объединение ряда понятий в новое понятие, существенные признаки которого при этом могут быть или суммой признаков компонентов или существенно новых;

4) *ассоциацию* как наиболее общее отношение, которое утверждает наличие связи между понятиями, не уточняя зависимости от содержания и объемов.

Для отдельных доменов могут использоваться специфические для них отношения. Онтология позволяет держать пользователя в возможном пространстве предопределенных понятий, содержание которых зафиксировано и понятно как разработчику, так и заказчику.

2.1. Онтологическое моделирование проблемной области

Концепты онтологии служат для отображения знаний, соответственно с интересами определенной проблемной области. Источниками таких знаний могут быть терминологические или толковые словари в близких проблемных областях, задокументированные требования на разработку ПС, другие документы. Один из методов рассмотрения модели проблемной области является ООП.

1) реальный мир состоит из объектов, которые взаимодействуют между собой;

2) каждому объекту присущ определенный набор свойств или атрибутов (аналог существенных признаков понятия);

3) атрибут определяется своим именем и значениями, которые он может принимать;

4) объекты могут иметь отношение друг с другом;

5) значения атрибутов и отношения могут со временем обмениваться;

6) совокупность значений атрибутов конкретного объекта в определенный момент времени определяет его состояние, а совокупность состояний объектов определяет состояние мира, которые могут находиться в разных состояниях;

7) в определенные интервалы времени могут возникать события, которые могут вызывать другие события или изменять состояния;

8) в течение времени каждый объект может принимать участие в определенных процессах, которые сводятся к выполнению последовательности действий,

разновидностями которых являются переходы из одного состояния к другому под воздействием соответствующих событий, возбуждения определенных событий или ссылки определенных сообщений к другим объектам;

9) действия, которые могут выполнять объекты, называют операциями объектов (как синонимы употребляют также термины "методы объекта" и "функции объекта");

10) возможная совокупность действий объекта называется его поведением;

11) объекты могут состоять из частей и взаимодействать путем обмена сообщениями;

Объект – это определенная абстракция данных и поведения; множество экземпляров с общим составом атрибутов и поведением составляет класс объектов. Определение объектов включает в себя видимую и невидимую части. Первая из них вмещает в себя вся сведения о взаимодействии объектов, и называется интерфейсом объекта, а другая содержит сведения о его внутренней структуре, которая инкапсулирована от пользователя. Еще одним средством определения объектов является наследование (эквивалент отношения обобщения). Один класс объектов унаследовал другой, если он полностью содержит все атрибуты и поведение унаследованного класса, но имеет еще и дополнительные атрибуты, и (или) поведение. Класс, который унаследован, называется *суперклассом*, а класс, который унаследовал – *подклассом*. Наследственность явным способом фиксирует общие и отличные черты объектов и позволяет явно выделить составные компоненты проблемы, которые можно использовать в нескольких случаях при построении нескольких классов-наследников.

Выявление объектов. Начальным шагом построения онтологии есть выявление существенных объектов, установление отношений между ними и предоставление им уникальных наименований. Для классов объектов выбираются имена, уникальные в границах домена.

Особенным источником для поиска объектов может быть выявление тех работ в границах проблемной области, составляющих отдельные задачи по достижению определенных профессиональных целей, которые желательно компъютеризовать и они могут быть реализованы за одно обращение к системе. Таким образом, происходит последовательная декомпозиция сложности каждой проблемы, которую можно выявить в домене предметной области:

1) сложная проблема трансформируется в совокупность целей ее достижения или работ, необходимых для ее достижения;

2) каждая из целей (или работ) трансформируется в совокупность возможных примеров использования системы, т.е. примеров достижения целей (выполнение работ), которые отражаются в сценарии;

3) сценарии трансформируются в процессе их анализа в совокупность взаимодействующих объектов.

Определенная таким образом цепочка *проблема–цели* или *работы–сценарии –объекты* отображает степени концептуализации для достижения понимания проблемы, последовательного снижения сложности ее частей и повышения уровня формализации моделей последних. Для объектов устанавливаются атрибуты, связи и состояния.

2.2. Описание доменов средствами онтологии

Анализ ПрО моделирование ее характеристик – первая задача, которая решается при построении семейств ПС, предназначенных для функционирования в данной ПрО. Традиционным подходом к ее решению является построение характеристических диаграмм (feature diagram) и DSL языка описания ПрО. Распространение онтологического подхода в разных отраслях науки обусловило проведение исследований применительно к моделированию ПрО и конкретизации подхода FDD.

Онтология применяется для описания понятий и связей удобному для целей разработки члена СПС в языке XML.

Одной из новых сфер применения онтологического подхода является речь описания ПрО–DSL и трансформация этих моделей в программный код.

Проектирование языка DSL включает в себя следующие шаги анализа ПрО:

1) идентификация ПрО;
2) сбор знаний об ПрО;
3) кластеризация этих знаний в терминах семантических понятий и операций над ними;
4) проектирование DSL, который хорошо описывает применение в ПрО.

Реализация ПрО это

1) создание библиотек, которые реализуют семантические понятия;
2) проектирование и реализация компиляторов для трансляции DSL программ в последовательность библиотечных вызовов;
3) описание в DSL программ для программ ПрО и их компиляция.

Цель анализа заключается в том, чтобы понять ПрО (выявить сущности и связи). Анализ обычно проводится путем моделирования объектов ПрО и выполняется аналитиком ПрО.

Систематическое моделирование ПрО как вид деятельности, называется доменной инженерией (domain engineering).

Существуют известные методологии анализа ПрО [103]: ODM (Organizational Domain Modeling, FODA (Feature-Oriented Domain Analysis, DSSA (Domain Specific Software Architectures). Предложен ряд систематических подходов к разработке семейств: Lucent, Family–Oriented Abstraction, Specification and Translation (FAST). Семейства программ непосредственно связанные (и часто отождествляются) с линиями программных продуктов.

При реализации DSL может использоваться известный метод: компиляции или интерпретация. При этом могут применяться стандартные компиляторы или специальные инструменты. Альтернативным подходом к реализации DSL является расширение существующих языков. Преимущество такого подхода заключается в том, что базовые средства МП не нужно повторно реализовывать.

Разработка некоторой онтологии предусматривает глубокий структурный анализ ПрО и включает в себя следующие действия:

1) выделение концептов – базовых понятий данной предметной области;
2) определение "высоты дерева онтологии" – количество уровней абстракции;
3) распределение концептов по уровням;

4) построение связей между концептами – определение отношений и связей базовых понятий;

5) консультации с разными специалистами для исключения противоречий и неточностей.

Процесс построения онтологии всегда итеративный.

Методология построения онтологии допускает вопросы обозначения целей и области применения создаваемой онтологий, а также построение онтологии, которая включает такие действия:

1) фиксация знаний о ПрО, определение основных понятий и их отношений в выбранной предметной области; создание точных непротиворечивых определений для каждого основного понятия и отношения; определение терминов, которые связаны и отношениями;

2) кодировка, т.е. разделение совокупности основных сроков, используемых в онтологии, на отдельные классы понятий;

3) выбор или разработку специального языка для представления онтологии;

4) непосредственное задание фиксированной концептуализации на выбранном языке представления знаний;

5) совместимое использование людьми или программными агентами общего видения структуры информации;

6) обеспечение возможности использования знаний ПрО;

7) создание явных предположений в ПрО, которые лежат в основе реализации;

8) отделение знаний об ПрО от оперативных знаний – это еще один вариант общего применения онтологии;

9) анализ знаний в ПрО.

2.3. Основные понятия онтологии представления ПрО

Существуют традиционные языки спецификации онтологии (Ontolingua, CycL, языки, основанные на дескриптивной логике, такие как LOOM, и языки, основанные на фреймах – OKBC, OCML, Flogic). Более современные языки, основанные на веб-стандартах, такие как XOL, SHOE или UPML, RDF(S), DAML, OIL, OWL созданные специально для обмена онтологией через веб.

В целом, различие между традиционными и веб-языками спецификации заключается в выразительных возможностях описания ПрО и в некоторых механизмах логического вывода понятий.

Важное направление использования языков онтологии связано с построением (генерацией) ПС, что оперируют знаниями, содержащимся в онтологии.

На сегодняшний день язык XML (Extensible Markup Language) фактически стал стандартом разметки данных для их сохранения и обмена информацией между разными пользователями. XML активно используется в самых разных областях научной и коммерческой деятельности, и в ближайшие времена его популярность будет только расти. Однако XML и его реализации представляют низкий уровень описания ресурсов и необходимые средства автоматической трансформации моделей ПрО (и онтологии) в XML-схемы, пригодные для работы приложений.

Связь языков моделирования с онтологией ПрО. Основной речью моделирования ПрО, которая используется сейчас для разработки архитектуры ПС, есть UML (и его расширение). Путем последовательного применения моделей можно сгенерировать описание классов и заготовки программного кода для выбранных языков программирования и платформ (см. подход MDA, раздел 1).

Альтернативный подход с подобными целями – онтологический, который называется *Ontology-Driven Software Development.* Он позволяет получать описания классов, которые отображают понятие ПрО. В отличие от предыдущего, модели ПрО могут не только использоваться для генерации кода, но и являются "выполняемыми" артефактами.

Средства описания онтологий ПрО

Классы описывают понятие ПрО, а *слоты* – свойства (атрибуты) классов.

Фасеты описывают свойства слотов (конкретные типы и возможные диапазоны значений).

Аксиомы определяют дополнительные ограничения (правила).

Абстрактные классы являются контейнерами конкретных классов и могут содержать *абстрактные* атрибуты (которые не содержат конкретных значений).

Атрибуты понятий ПрО в Protege называются *слотами*.

Конкретные классы содержат конкретные слоты, которым могут быть назначены значения (экземпляры атрибутов).

Слоты в Protege описывают свойства классов и экземпляров (возможные атрибуты). Согласно фреймовой модели представления знаний, которая используется в Protege, слот – это фрейм. Слоты определяются независимо от любого класса и один и тот же слот может принадлежать разным классам.

Фасеты позволяют вводить ограничение на типы и диапазоны значений экземпляров (значений атрибутов), подобные соответствующим понятиям XML-схемы. Кардинальность слота определяется количеством значений слота, ограничениями для типов значений (например, целое, символьное и т.п.) экземпляров класса и граничных значений (min, max). Фасеты определяют ограничение на присоединение слота к фрейму класса.

Слоты-образцы (*template slot*) *и собственные* (*own*) *слоты*. Слот можно присоединить к фрейму (класса) одним из двух способов: как *слот-образец* или как собственный слот. Собственный слот, присоединенный к фрейму, описывает свойства объекта, представленного фреймом. Классы также могут иметь собственные слоты. Например, документация класса является собственным слотом, присоединенным к классу, поскольку описывает сам класс, а не экземпляры класса.

Жизненный цикл онтологии включает в себя такие процессы: проектирование, валидация, поддержка, развертывание, оценивание, обновление, визуализация, интеграция, повторное использование.

Один из источников домена – требование к системам, которые предоставляют концентрируемые сведения о совокупных потребительских свойствах домена, которые должны быть трансформированы в базу знаний домена и систем КПИ, нацеленных на решение его составных задач. При этом успех онтологии домену определяется степенью адекватности избранной модели видения требований задачам домена.

Сложилось несколько подходов к конструированию онтологии:

1) *интуитивный,* который опирается на интуицию;

2) *индуктивный*, который базируется на рассмотрении, проверке и анализе специфических случаев в домене для использования в других доменах;

3) *дедуктивный*, который базируется на принятии нескольких общих принципов, используемых для конструирования онтологии как сочетания в них общих и специфических случаев;

4) *синтезированный,* как базовый набор онтологии, в которую пересекаются общие сущности, создающие единственную онтологическую структуру.

2.4. Формализация онтологической модели ЖЦ

Формально модель домена ЖЦ включает в себя процессы P (process), действия (Action) и задачи T (Task) и имеет вид

$$M_{жц} = (P_к, A_м, T_п),$$

где $P_к = (P_{1к}, P_{2к1}, P_{3к2})$, $P_{1к\ к\ =1\text{-}5}$ (основные процессы ЖЦ), $P_{2к1,\ к1=1\text{-}8}$ – дополнительные процессы ЖЦ, $P_{3к2,\ к2=4}$ – организационные процессы; $A_м = (A_{кr}, A_{к1l}, A_{к2j})$ – действия или задачи процесса.

В них задачи означают:

$Ак,r,r = 1\text{-}35$ – действия на основных процессах ЖЦ,

$A_{к1l}$, $l=1\text{-}25$ – действия на процессах поддержки ЖЦ,

$A_{к2}$, $j= 1\text{-}14$ – действия в организационных процессах ЖЦ;

$Tn = (T_{пк}, T_{nl}, T_{nj})$ - $T_{пк}$, $к= 1\text{-}135$ – задачи основных процессов ЖЦ,

$T_{nl,\ l\ =1\text{-}70}$ – задачи процессов поддержки ЖЦ

$T_{nj,\ j=1\text{-}27}$ – задачи организационных процессов ЖЦ.

Описание содержания задач в стандарте не приведены. Их семантика задается при реализации формального описания.

Для представления структуры ЖЦ используется графический язык DSL, а для отображения процессной специфики ЖЦ могут использоваться языки общего назначения (JAVA, C++, C# и др.), ориентированные на реализацию вычислительных действий программ. Язык DSL содержит общие абстракции для отображения классов объектов ПрО, типов процессов и действий, а также отношений между ними [101–102]. Описание в этом языке сводится к языкам HTML, XML, WSDL и др.

Модель ПрО ЖЦ описана одним из языков DSL, которая может быть трансформирована к другой модели с более низким DSL. Это позволяет интегрировать между собой разные части процессов системы, написанные на разных языках DSL. Иными словами, ПрО ЖЦ может быть описана на одном уровне абстракции, а затем преобразована к языку более низкого уровня абстракции. Модель ПрО дополняется повторными компонентами и объектами, и соответственно уточняется характеристиками и автоматизируется с использованием высококачественных доменно-специфических языков, настроенных специально на процессы и действия, которые есть в классе языков онтологии. Модели могут содержать информацию об объединении процессов и действий, включая артефак-

ты и их зависимости между собой. На их основе формируется информация для конфигурационной структуры ПС с учетом аппаратных и программных ресурсов, необходимых при выполнении ЖЦ процессов.

Описание модели характеристик процессов ЖЦ. Среди существующих методологий доменного анализа наиболее известны такие: ODM (*Organization Domain Modeling*), FODA (*Feature-Oriented Domain Analysis*). Для анализа некоторых доменов используется модель DSSA (*Domain-Specific Software Architectures*), в которой задается *модели характеристик MX*, содержащая общие характеристики программных компонентов и процессов ПС. Отличительной особенностью представления процессов ЖЦ является *диаграмма* зависимостей между характеристиками. Концепция таких диаграмм унаследована из метода FODA. В ней описываются все возможные конфигурации процессов из разных категорий стандарта ЖЦ. Каждый из них рассматривается как *экземпляр* или *образец* (instances). Нотация диаграмм характеристик процессов выполнятся языком FDL (Feature Definition Language), позволяющим описывать:

1) атомарные и композиционные характеристики, имена которых определяются и используются в разных процессах;

2) необязательные (*optional*) и обязательные характеристики (*mandatory*) выражений, которые относятся к замкнутым конструкциям *all*();

3) альтернативные характеристики (*exclusive* – выбор) через *one-of* ();

4) характеристики по умолчанию (default).

Результат трансляции характеристик в FDL может быть выполнен XMI для обмена информацией Metadata (XML Metadata Information Exchange format) и может быть импортирован в систему моделирования UML при генерации классов. Подход к описанию модели характеристик ПрО использован при разработке вариантов ЖЦ ПС и конфигурировании разных процессов с помощью данной модели генерируются необходимые варианты ЖЦ для реализации определенного класса ПС.

Описание процессов ЖЦ средствами. DSL, Protege

Для описания онтологии домена ЖЦ взят Eclipse DSL. В нем имеются средства разработки графических моделей ЖЦ. Описание основных процессов домена ЖЦ с помощью инструментария DSL Tools VS приведен на рис. 3.8.

Типы отношений в данном графическом представлении задают основную логику процессов домена ЖЦ. В каждом классе заданы методы и поля, необходимые для функционирования.

Процессы поддержки заключают в себе все процессы, которые выполняются после построения системы и поддержки его работоспособности. Их онтологическая структура отражает структуру основных процессов ЖЦ. Затем осуществляется генерация имеющихся графических моделей в текстовое представление XML-языка. При этом процесс тестирования ЖЦ рассмотрен отдельно с помощью инструментального средства Protege. В нем он представления соответствующими правилами и отношениями.

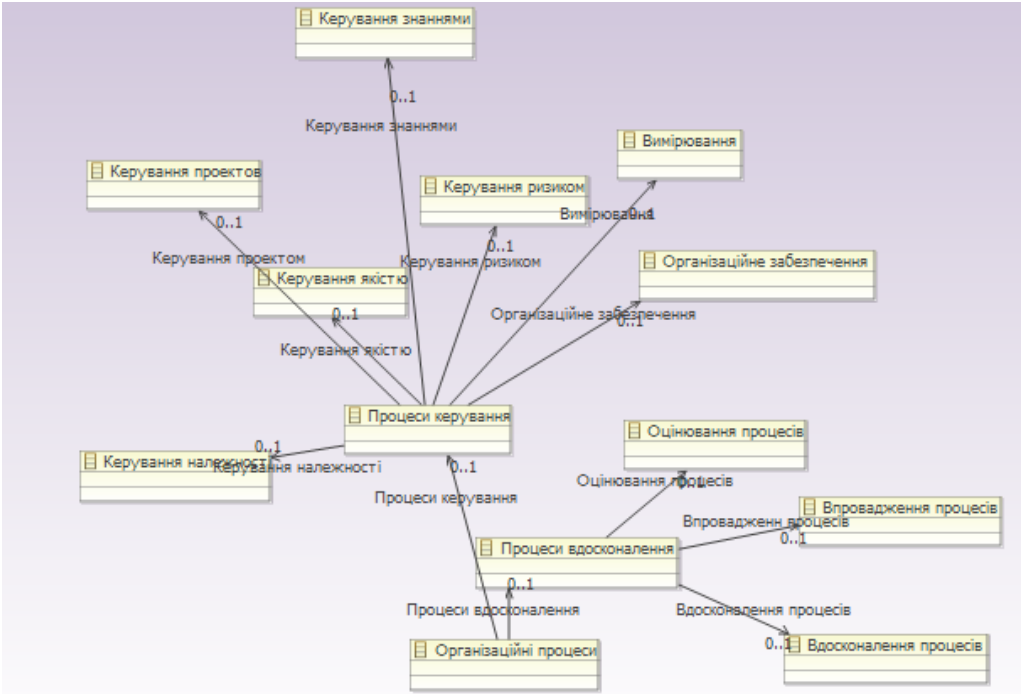


Рис. 3.8. Онтология основных процессов ЖЦ

Текстовое описание процессов ЖЦ в языке XML

Данное графическое представление процессов ЖЦ было использовано для получения текста в языке XML.

Ошибки в графическом описании были найдены дизайнером и исправлены с помощью соответствующего редактора. Результат каждого процесса дается в XML. Пример фрагмента описания основных процессов ЖЦ (рис.3.6.) в языке XML приведен ниже.

<?xml version="1.0" encoding="utf-8"?>

<AssociationLine Name ="Визначення вимог" Type="Main.Визначення_вимог" ManuallyRouted

= "true" FixedFromPoint = "true" </AssociationLine>

<AssociationLine Name = "Интеграция_ПС" Type = "Main.Інтеграція_ПС"

<AssociationLine Name = "Инсталляция" Type = "Main.Інсталяція"

<AssociationLine Name = "Эксплуатация" Type = "Main.Експлуатація"

<Property Name = "Интеграция_ПС" />

<Property Name = "Инсталляция" />

<Property Name = "Анализ_вимог" />

<Property Name = "Эксплуатация" />…

Для процесса тестирования ЖЦ выполнено аннотирование его средствами Protege. Такое описание отсутствует в практике программирования и ориентировано на проведения тестирования простых программ [1–3].

2.5. Онтологии процесса тестирования ЖЦ

Концептуальная модель процесса тестирования ПС имеет вид [4]:

$$SFT = \langle TM, TD, TA, Env \rangle,$$

где *TM* – подпроцесс управления тестированием; *TD* и *TA* – подпроцессы тестирования систем и приложений; *Env* – концептуальная и информационная среда процесса тестирования ПС.

Все подпроцессы имеют унифицированное формальное представление:

$$TM = \langle Task(TM, TD, TA), En(TM, TD, TA), CM(TM, TD, TA) \rangle,$$

$$Env(TM) \cup En(TD) \cup En(TA) = Env$$

где *Task* – задачи соответствующего подпроцесса; *Env* – концептуальная и информационная среда соответствующего подпроцесса; *CM* – подмодель координации операций подпроцесса.

В состав среды *Env* входят следующие элементы:

$$Env = TG \cup SG \cup T \cup P \cup RG \cup RP,$$

где *TG* и *SG* – тесты и готовые программы; *T* и *P* – тесты и протестированное приложение; *RG* и *RP* – отчеты о выполнении тестовых программ и тестов.

Онтологическое описание процесса тесирования. В системе Protege знания о модели процесса задаются *классами, слотами, фасетами* и *аксиомами*. Подобную возможность предоставляют также и другие инструменты. Например, диаграммы классов UML системы Rational Rose отображаются в программный код на нескольких ЯП.

Для представления онтологии тестирования выделяются две группы понятий: *простые* и *сложные.*

Простые понятия это такие: Тестер (*Tester*), Контекст (*Context*), Действие (*Activity*), Метод (*Method*), Артефакт (*Artefact*) *и* Среда (*Environment*). Они имеют атрибуты, входящие в базовое (родительское) понятие, которое принимает конкретные значения.

Tester – определяет субъект или объект, который выполняет тестирование. Группа тестирования имеет лидера, который является атрибутом понятия (имя, *тип, обязанности*), где имя – значение атрибута, задающие роль и обязанности тестера в процессе тестирования.

Контекст определяет соответствующие уровни, методы тестирования, входы и выходы задач тестирования. В онтологии это понятие определяет один атрибут: Context type (Уровень_тестирования) по форме уровень тестирования = {модульное, интеграционное, системное, регрессионное}

Действие состоит из понятий, которые детализируют шаги процесса тестирования: планирование тестирования, разработку (генерацию) тестов, выполнение тестов, оценку результатов, измерение тестового покрытия, генерация отчетов и др. Для этого понятия определяется один атрибут – тип действия (Activity type) с возможными значениями: тип действия = {планирование, разработка тестов, выполнение тестов, проверка результатов, оценка покрытия, подготовка отчета}

Метод – это понятие, которому соответствует несколько способов тестирования – структурное и функциональное тестирования. Каждый метод по отношению к исходному коду может классифицироваться как "белый ящик", "черный ящик" со спецификацией (specification-based). Структурный метод – это подсев ошибок и

поток данных (control-flow methods), включающий в себя покрытие операторов, ветвей и критериев их покрытия. Аналогичным образом классифицируются методы "черного ящика", основанные на спецификации функций и предположениях об ошибках. Все методы можно разделить на систематические (поиск ошибок) и стохастические (статистические) для выявления отказов. Метод тестирования включает в себя следующие атрибуты: имя метода, тип метода и подход.

Артефакт. Каждое действие включает в себя несколько артефактов, таких как объект тестирования, промежуточные данные, результаты тестирования, планы, наборы тестов, скрипты и т. п.. Их называют "тестовыми активами". Объекты тестирования могут быть разных типов: начальный код, HTML файлы, XML файлы, встроенные изображения, звук, видео, документы и др. Все эти артефакты отображаются онтологией. Каждый артефакт ассоциируется с местом сохранения, данными, историей создания и просмотра.

Enviroment. Программные среды, где выполняется тестирование, как правило, описываются такими данными: имя, тип и версия продукта. Данное понятие разбивается на два подпонятия: аппаратное и программное обеспечение с атрибутами – *название устройства, модель, производитель и версия*.

Среда = {*ОС, БД, Компилятор, веб-браузер*}

Сложные понятия процесса тестирования это: обязанности тестера (capability) и задача (task). В распределенной системе взаимодействие между компонентами выполняется посредством интерфейсов (сообщений). После обработки сообщения, компонент, который его получил, возвращает ответ. С каждым сообщением можно связать атрибуты: тип и значение: *Тип* = {*директивное, декларативное, …*}. С каждым ответом можно связать состояние: *Состояние: ответ* = {*Успех, Отказ*}.

Используя эти понятия, студенты КНУ на практических занятиях создали вариант онтологии процесса тестирования и реализации программы тестирования в языке Ruby.

Эта программа встроена у ИТК сайта http://sestudy.edu-ua.net, обращение к которой осуществляется нажатием на слово "Онтология" в главной панели данного сайта.

Реализация ЖЦ в ИТК. На веб-сайте ИТК реализована комплексная технология, которая включает в себя спектр технологий, средств, инструментов проектирования и спецификации КПВ, ПС, членов семейства систем, а также стандартные системы (Eclipse, Protege, репозиторий, CORBA, MS.Net и др.), ЯП, системы поддержки взаимодействия программ, систем и сред между собой VS.Net↔Eclipse, VS.Net↔JAVA, VBasic↔Eclipse [104–105]. Базис технологии. – онтология процессов ЖЦ , типов данных, используемых для вычислений различных задач, вычислительной геометрии с применением онтологической системы Protégé и др.

На сайте ИТК содержится электронный учебник "Программная инженерия", который используется для обучения студентов аспектам этого предмета, а также моделям ЖЦ. Метод онтологизации ЖЦ стандарта ISO/IEC 12207 является оригинальным, он требует дальнейшего развитие и представление в ИТК. По этой тематике были сделаны доклады на международных университетских конференциях TAAPS и ICTERI (2011– 2013), а также в программу курса обучения "Программная инженерия" включены лабораторные работы по автоматизации процессов ЖЦ.

Глава 3. ОБЕСПЕЧЕНИЕ КАЧЕСТВА ПС

Разработка ПС достигла такого уровня развития, что возникла необходимость в инженерии качества, включая методы оценивания результатов проектирования компонентов. на процессах ЖЦ, риска и степени использования готовых компонентов для снижения стоимости разработки нового проекта и метрического анализа и контроля достигнутых показателей качества. Основой инженерных методов в программировании является повышение качества. Для достижения этого были сформулированы методы определения требований к качеству, подходы к выбору и усовершенствованию моделей метрического анализа показателей качества, методы количественного измерения рисков на процессах ЖЦ.

Главная составляющая качества – *надежность.* Ей уделяется большое внимание в сфере надежности технических средств и тех критических систем (реальное время, радарные системы, системы безопасности и др.), для которых надежность является главной целевой функциях оценки их реализации. Как следствие в проблематике надежности разработано свыше сотни математических моделей надежности, которые в основном основываются на функциях ошибок, которые остались в ПС, интенсивности отказов или частоты возникновения дефектов в ПС. На их основе осуществляется оценка надежности ПС.

Качество ПС – предмет стандартизации. В стандарте ДСТУ 2844 – 1994 определена модель *качества ПС* из совокупности свойств (показателей качества) ПС, которые характеризуют способность ПС удовлетворять потребностям заказчика соответственно назначению. Этот стандарт регламентирует базовую модель качества и показатели, главным среди которых есть надежность. Стандарт ISO/IEC 12207 определяет не только основные процессы ЖЦ разработки ПС, но и организационные и дополнительные процессы, которые регламентируют инженерию, планирование и управление качеством ПС.

3.1. Основные задачи проблемы управления качеством

В условиях быстрых темпов развития индустрии программ и роста конкуренции, разработанных ПС проблема выдачи сертификата на ПП, включающая в себя оценку характеристик качества по окончании его разработки. Мониторинг качества по *количественным* показателям в ходе всего периода разработки ПП начинается с ранних этапов ЖЦ. Последняя версия международного стандарта "Процессы ЖЦ программного обеспечения" (ISO/IEC 12207:2007) включает в себя процесс "*управление качеством"* наряду с процессами верификации, тестирования и обеспечения гарантий качества. Для эффективного выполнения разработаны модели и методы инженерии качества, обеспечивающие поддержку принятия обоснованных решений по управлению качеством на всех этапах ЖЦ ПС.

Существенный вклад в развитие этого направления программной инженерии внесли В.В.Липаев, А.Ф.Кулаков и американские специалисты.

В работах В. В. Липаева [13] предложена целостная *системы управления качеством*, основанная на эталонной модели качества [14], совокупности организационных методов управления процессом проектирования комплексов программ

(КП) на этапах ЖЦ и методике учета технических показателей в ходе ЖЦ для последующего их использования при окончательной оценки полученных показателей качества в соответствии с требованиями к ним. *Качество КП* определяется как совокупность технических, технологических и эксплуатационных характеристик КП, которые входят в состав эталонной модели качества, представленной на верхнем уровне шестью базовыми характеристиками, значения которых могут быть определены количественно или оценены качественно. Этот подход применяется на этапах ЖЦ КП с контролем качества службой качества и устранением возникающих угроз качеству в связи с обнаруженными дефектами в КП. В задачи службы качества входит планирование и слежение за процессом достижения качества на этапах ЖЦ, квалификационное тестирование, испытание пробной версии СКП и оценка базовых показателей качества этой версии на основе собранных технических данных КП.

В работах А.Ф.Кулакова [15] предложен комплекс средств оценки качества больших программ ЭВМ, включающий в себя набор свойств и показателей качества программных изделий, а также методы их оценки.. К ним относятся статистические методы анализа программ, основные функции тестирования и испытания программ. Основную роль сыграли введенные им функциональные и эксплуатационные показатели качества, определены задачи контроля и оценки качества при статистических испытаниях на основе моделирования, планирования и оценки уровня завершенности испытаний продукта или изделия. По его мнению, качество ПП находится в прямой зависимости от объема затрат на его разработку и сопровождение, т.е. 75% затрат на этапах ЖЦ идет на разработку продукта и его сопровождение. Под сопровождением понимается процесс модификации программ, обусловленных необходимостью устранения выявленных ошибок ПП и изменения его функциональности. Согласно положениям стандарта ГОСТ 15467–1989 уровень качества конкретной продукции включал в себя задачи выбора номенклатуры показателей качества, методов определения их значений и сопоставления полученных значений с базовыми. Он положил начало созданию научно-обоснованных показателей качества и методов их определения, ориентированных на степень завершенности ПП.

Вопросами инженерии качества отдел "Программная инженерия" занимался на протяжении многих лет. Автор входила в рабочую группу СЭВ по разработке стандарта качества. Проект стандарта был разработан под руководством А. Ф. Кулакова и В. В. Липаева и обсуждался на рабочей группе СЭВ в марте 1987 г. в Дрездене. После развала СССР А. Ф. Кулаков возглавлял эти работы в ГКНТ Украины. Отдел имел фундаментальный проект в ГКНТ по проблематике качества и технологии его реализации.

Наибольший вклад в развитие этой проблематики внесла с. н. с. Г. И. Коваль. Ею был проведен анализ зарубежных стандартов качества серии 9000, проекта СЭВ, разработана структура и основные положения к коллективной монографии "Основы инженерии качества программных систем" [106, 107]. По результатам многолетних исследований проблематики качества Г. И. Коваль сформулирован комплекс задач инженерии управления качеством. Отдельные задачи были теоретически определены и реализованы в кандидатских диссертациях, защищенных под руководством автора в 2004–2008 гг. [108 – 111]: Т. М. Коротун (тестирова-

ние), Г.И. Коваль (управление качеством), О. А. Слабоспицкая (экспертная оценка процессов и систем),. А.Л. Колесник (вариантность ПС).

В результате был разработан комплекс базовых задач и инструментов инженерии качества (рис. 3.9), включающий в себя следующие из них:

1) *Модели управления качеством,* начиная с ранних стадий ЖЦ ПС на основе модели процесса принятия решений по управлению качеством ПС, моделей задания количественных требований к качеству компонентов ПС, прогнозирования плотности дефектов с помощью байесовских сетей и систематического количественного контроля качества. ПС;

2) *Модель процесса тестирования* ПС, которая способствует улучшению планирования тестирования, выделению необходимых ресурсов тестирования и обеспечивает управление тестированием с учетом рисков отказов ПС и рисков срыва выполнения проектов систем ПС в условиях ограниченных ресурсов;

3) *Совершенствования моделей процессов* ЖЦ путем экспертно-аналитического оценивания процессов, выполняемых в каждом цикле управления, начиная с этапа прогнозирования целевой характеристики качества и ее достижения на процессах разработки ПС;

4) *Модель интегрированной технологии* управления рисками программных проектов с использованием аппарата экспертного оценивания деревьями ценности и байесовскими сетями;

5) Методический инструментарий оценки качества программ и процессов на процессах анализа требований, проектирования, кодирования и тестирования.

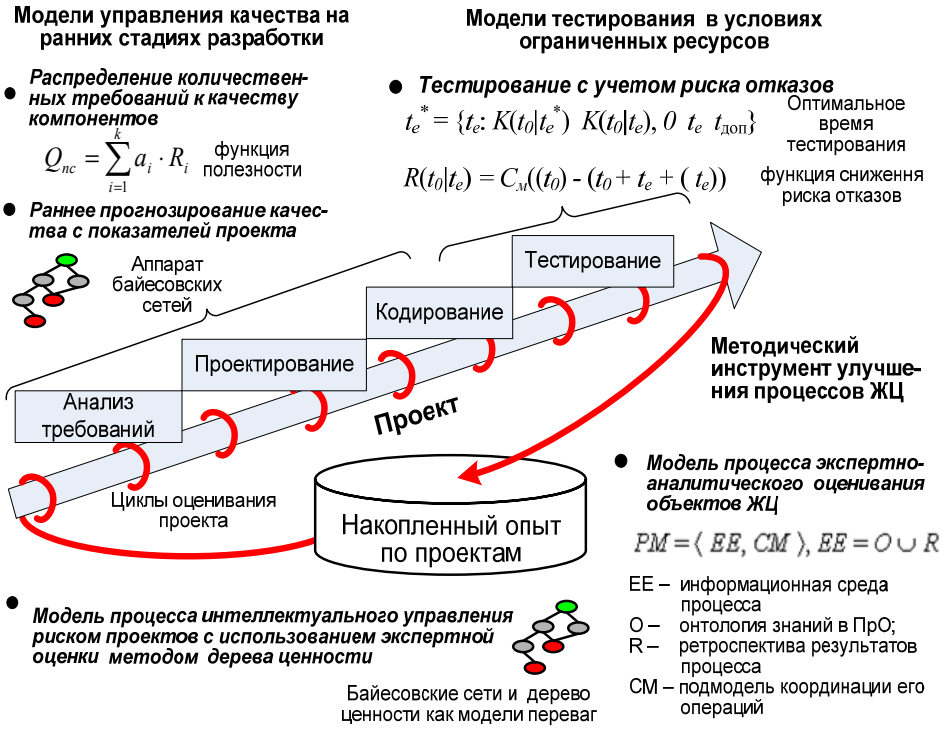


Рис. 3.9. Общая схема комплекса задач инженерии качества ПС

Новый подход к управлению и оценке качества базируется на процессах беспрерывного оценивания усовершенствованных процессов ЖЦ (парадигма процессо-ориентированной инженерии качества), ориентированный на создание моделей процессов ЖЦ, включающих задачи совершенствования с учетом требований к ПС, прогнозирования качества и принятия решений по управлению процессом тестирования для учета риска отказов ПС в процессе эксплуатации. В основу моделей прогнозирования качества ПЗ и диагностики недостатков в применяемых процессах ЖЦ ПС положен теоретический аппарат – *байесовская сеть.* Этот подход к управлению качеством ПП ориентирован на достижение *завершенности* ПС, т.е. получения характеристики качества и обеспечения надежной безотказной работы всех программных компонентов ПС. В нем акцент переносится на *процессы* ЖЦ, с помощью которых проводится *выявление и устранение дефектов* в ПС. В основе подхода лежит теория прогнозирования уровня завершенности ПС на процессах ЖЦ путем установления взаимосвязи внутренних и внешних метрик качества в эталонной модели и совершенствования методов прогнозирования дефектов с учетом возникающих угроз и рисков к качеству.

3.2. Моделирование характеристик качества ПС

Качество, по определению стандарта ГОСТ 2844–1994, это "совокупность свойств ПС, обеспечивающих способность удовлетворять *установленные* или *предполагаемым* потребностям в соответствии с назначением". Понятие свойства продукта ассоциироваться с качеством ПС согласно стандартной модели и особенностей ПС. В качестве ПС взята СОД, состоящая из нескольких компонентных приложений, работающих с большими объемами информации в базах данных, не критическими по отношению безопасности функционирования [112 – 115].

Ключевыми характеристиками качества ПС выбираются надежность и завершенность, как свойство системы исключать отказы в случае скрытых дефектов, которая моделируется путем улучшении *управляемости* ПС, исходя из этого критерия и *модели качества*, устанавливающей взаимосвязь мер и метрик внутреннего, внешнего и эксплуатационного типа.

На начальных этапах разработки ПС специфицируются значения (область значений) *внешних метрик*, которые служит критерием достижения установленного уровня качества при испытании ПС, определяют наиболее пригодные (с точки зрения прогноза значений внешних метрик) *внутренние метрики* и планируется поэтапное достижение внешних требований к качеству. Внешние требования к качеству ПС устанавливается с позиций *эксплуатационного качества* ПП.

С позиций *завершенности* ПС главным показателем ее внутреннего качества являются дефекты, внешнего – отказы, а эксплуатационного – обобщенный взгляд пользователей на работу ПС. Это соответствует трехуровневой модели качества, устанавливающая взаимосвязь внутренних, внешних и эксплуатационных мер качества (рис. 3.10):

1) внутренняя мера D_0 – количество (плотность) дефектов в каждом компоненте ПС;

2) внешняя мера $R(t)$ – это безотказность функционирования каждого компонента ПС в течение заданного времени t, т. е. вероятность того, что за время t

эксплуатации компонента не возникнет последовательность входных данных в сценарии его использования и которая приведет к отказу;

3) мера эксплуатационного качества Q_{nc} связана с безотказным функционированием ПС и соответствует характеристике качества при эксплуатации ПС и обозначает *удовлетворенность* ("satisfaction") ПС..

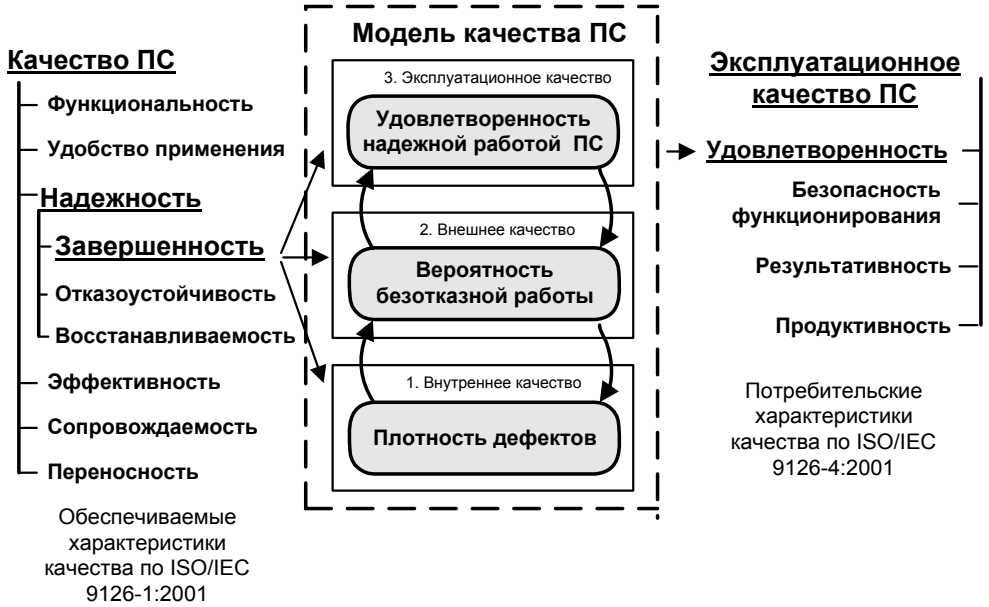


Рис. 3.10. Трехуровневая модель качества ПС

В данной трехуровневой модели качества ПС главной характеристикой мы считаем *завершенность*. Ей соответствует показатели модели качества, которые определяют надежность ПС. К ним относятся – удовлетворенность надежной работой ПС, вероятность безотказной ее работы и плотность дефектов, а также отказоустойчивость и восстанавливаемость ПС. Эти показатели .заданы взаимосвязанными *метриками качества* на разных уровнях модели качества стандарта ISO/IEC 9126 (1 – 4) 2001. Они определяют подходы к их использованию при оценке показателей качества программных продуктов.

3.3. Задачи управления качеством ПС

Процесс принятия решений по управлению качеством включает в себя решение комплекса таких задач:

1. Определение значения выбранной характеристики качества ПС (*целевого уровня качества*), как критерия удовлетворения потребностям пользователей.
2. Прогнозирование достижимости уровня качества на процессах ПС и условий устранения "слабых мест" в процессах ЖЦ.
3. Выбор и принятие решений для предметных показателей (ПрП) в соответствии с заданным уровнем качества и оценки их реализации.

4. Определение стратегии, методов и средств обеспечения качества компонентов ПС, а также методов проверки правильности рабочих продуктов ПС и сбора данных о ходе их выполнения.

5. Анализ тех или иных факторов успешного достижения уровня качества и коррекция хода выполнения процессов проекта ПС.

Подход к управлению качеством упорядочивает решение указанных задач и может быть схематически представлен в виде концептуальной схемы принятия решений по управлению качеством по выбранной характеристике (рис. 3.11). Она включает в себя описание моделей и методов, разработанных в рамках данного подходов к управлению качеством.

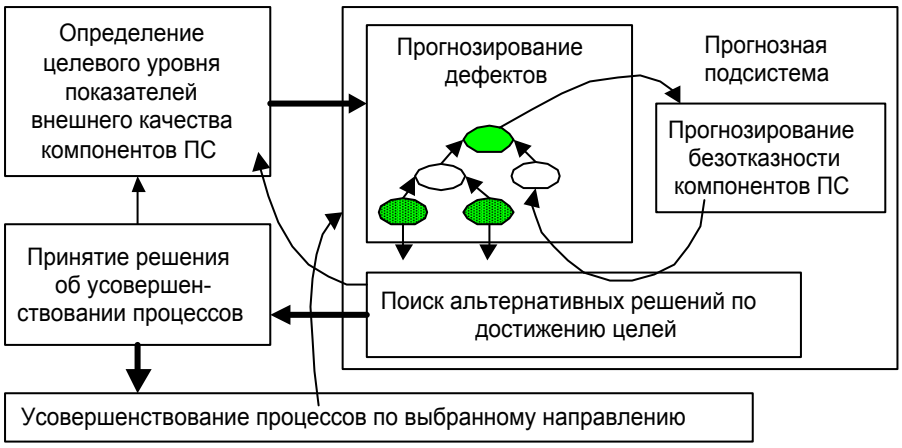


Рис. 3.11. Схема процесса управления качеством ПС

3.4. Модель требований с ориентацией на обеспечение качества ПС

Предложена модель требований к завершенности компонентов ПС, построенная с учетом дифференцированного подхода к ним и необходимости достижения установленных целевых значений завершенности компонентов ПС, адекватных потребностям заказчика ПС.

Мера эксплуатационного качества ПС определена как *функция полезности* вида

$$Q_{nc} = \sum_{i=1}^{k} a_i \cdot R_i ,$$

где a_i – мера важности i-й функции ПС для делового процесса, R_i – надежность (безотказность) выполнения функции в заданном периоде t эксплуатации системы.

Безотказность выполнения функций ПС оценивается во время системного тестирования и эксплуатации ПС. Поэтому проблема нахождения оптимального уровня завершенности декомпозируется по четырехуровневой иерархии (*ПС* $\rightarrow$ *функции ПС* $\rightarrow$ *программные приложения* $\rightarrow$ *программные модули*). Такая иерархия обеспечивает определение параметров модели требований для каждого компонента ПС.

Оценки важности отдельных функций в деловом процессе и оценки важности отдельных программных приложений для выполнения функций ПС выполняется в контексте риска отказов модулей при работе системы.

Задача нахождении оптимального целевого уровня *завершенности* каждого компонента системы, который обеспечивает максимизацию функции полезности Q_{nc} с учетом технических и ресурсных ограничений проекта ПС. Пусть:

u_i – коэффициент относительного веса функции F_i при достижении эксплуатационного качества Q_{nc}, $i = 1,..., k$$v_{ij}$ – коэффициент относительного веса j-го ПрП в обеспечении выполнения i–й функции, $u = 1,..., k; j = 1,..., l;$

w_{js} – коэффициент относительного веса s–го ПрП при выполнения j–го программного приложения, $s = 1,\ldots, m; j = 1,\ldots, l$;

r_s – безотказность модуля M_s в период эксплуатации t;

E_j – множество номеров всех модулей, необходимое для выполнения j-го компонента ПС;

α_s – нижняя граница безотказности модуля M_s;

β_s – верхняя граница безотказности модуля M_s;

G – общая цена ПС;

C – себестоимость создания ПС организацией–разработчиком;

c_s – накладные расходы, связанные с разработкой модуля M_s ;

d_s – расходы, необходимые для достижения единичного уровня безотказности модуля M_s;

δ – доля прибыли в цене ПС.

Целевые значения безотказности модулей $r_1 \ldots r_m$, определяются по функции максимума полезности Q_{nc} :

$$Q_{nc}(r_1,...,r_m) = \sum_{j=1}^{l}(\sum_{i=1}^{k} u_i v_{ij} \cdot \prod_{n \in E_j} r_n) \to \max \qquad (3.1)$$

при ограничениях $$0 < \alpha_s \le r_s \le \beta_s \le 1 \text{ s } = 1,..., m \qquad (3.2)$$

$$c_s + d_s \cdot r_s \le (1-\delta) \cdot G \cdot \sum_{j=1}^{l}\sum_{i=1}^{k} u_i v_{ij} w_{js} \qquad (3.3)$$

$$\sum_{s=1}^{m}(c_s + d_s \cdot r_s) \le C \qquad (3.4)$$

Данная задача нелинейной оптимизации с линейными ограничениями (3.2) – (3.4) практически решается с помощью пакета MATLAB.

Параметры u_i, v_{ij}, w_{js} ($u = 1,\ldots, k; j = 1,\ldots, l; s = 1,\ldots, m$) находятся методом анализа иерархий (МАИ) путем парного сравнения и последовательного определения локальных приоритетов компонентов ПС в пределах каждого уровня иерархии по отношению к компонентам предыдущего (высшего) уровня.

Ограничения (3.2) задают допустимые нижние α_s и верхние β_s границы безотказности модулей, исходя из оценок важности каждого модуля.

Ограничения (3.3) устанавливают взаимосвязь общих расходов на разработку модуля для установления линейной зависимости между стоимостью модуля и уровнем его безотказности.

Ограничение (3.4) устанавливает взаимосвязь суммарных расходов на разработку всех модулей и себестоимости создания ПС.

Таким образом, данная модель устанавливает требования к завершенности каждого модуля (r_i), а затем и каждого приложения (q_i) с учетом независимости модулей в структуре ПрП. Так определяются целевые уровни завершенности всех компонентов, модулей ПС.

3.5. Система прогнозирования безотказной работы ПС

Контроль достижимости уровня завершенности q_i, базируется на построении прогнозов двух видов: поискового прогноза с помощью методов и моделей прогнозной системы:

1) модель раннего *прогнозирования безотказности* ПрП (внешняя метрика качества ПС в контексте завершенности);

2) модель раннего *прогнозирования скрытых дефектов* в ПС (внутренняя метрика качества ПС в контексте завершенности);

3) метод *анализа альтернатив* достижения уровня качества ПС;

4) модель определения *оптимального времени тестирования* компонентов ПС с учетом риска их отказов.

Раннее прогнозирование безотказной работы ПС. Рассматриваемый подход включает "раннее" прогнозирование (до начала тестирования) и соответственно традиционное "позднее" прогнозирование [113].

Раннее прогнозирование безотказности ПС заключается в построении проекции значений мер безотказности, полученные по внутренним метрикам на определенном этапе ЖЦ, а также значений, вычисляемых по внешним метрикам на следующем этапе и в конце разработки ПС.

Цель раннего прогнозирования – определить, какие усовершенствования могут быть сделаны в применяемых методах и процессах разработки ПС с целью обеспечения минимальной плотности дефектов к моменту начала системного тестирования.

"*Позднее" прогнозирование* безотказности ПС связано с применением аналитических моделей роста надежности на этапе системного тестирования после сбора определенного количества данных об отказах ПС. При прогнозировании безотказности ПС процесс отказов целесообразно моделировать неоднородным пуассоновским процессом , в котором функция надежности $R(t \mid T)$ определяется по формуле

$$R(t \mid T) = \exp(-(m(T+t) - m(T))),$$

где $R(t \mid T)$ – условная вероятность того, что в течение заданного времени t эксплуатации ПС в определенных условиях среды функционирования не возникнет отказ при тестировании ПС в течение времени T; $m(t)$ – функция роста надежности, представляющей собой среднее количество дефектов в ПрП, выявленных во время его функционирования в течение времени t.

Для раннего прогнозирования безотказности целесообразно использовать экспоненциальную модель Дж. Мусы, которая не требует данных об отказах осо-

бенно пригодна для раннего прогнозирования эксплуатационной надежности ПрП в начале системного тестирования.

Функция роста надежности $m(t)$ для этой модели определяется формулой

$$m(t) = N_0(1 - \exp(-\frac{\lambda_0}{N_0} \cdot t)),$$

где N_0 – количество скрытых дефектов в ПрП в начале системного тестирования; λ_0 – интенсивность отказов ПрП в начале системного тестирования, определяемая по формуле

$$\lambda_0 = N_0 \cdot \frac{\rho \cdot K}{I \cdot \varphi},$$

где ρ – интенсивность выполнения кода (скорость процессора); $K = 4 \cdot 10^{-7}$ – коэффициент выявления дефектов (постоянный для модели Дж. Мусы); I – количество инструкций исходного кода; φ – коэффициент расширения кода (число инструкций выполняемого кода, которое приходится на одну инструкцию исходного).

Тогда условная функция надежности эксплуатации ПС имеет вид

$$R(t \mid T) = \exp(-(m(T+t) - m(T))) = \exp[-N_0 \exp(-\frac{\lambda_0}{N_0} T)(1 - \exp(\frac{\lambda_0}{N_0} \cdot t))]. \quad (3.5)$$

В частности, если *T=0,* то есть ПрП вообще не будет пребывать на этапе системного тестирования,

$$R(t) = R(t|0) = \exp[-D_0 \cdot I \cdot (1 - \exp(\frac{\lambda_0}{D_0 \cdot I} \cdot t))] \quad (3.6)$$

где $D_0 = N_0 / I$ – плотность скрытых дефектов в начале тестирования.

Формулы (3.5) и (3.6) – это метрики раннего прогнозирования безотказности ПрП с учетом (или без) времени его тестирования.

Размер (объем) ПС (I) можно определять одним из методов FPA (Function points analysis) в условных единицах функциональности и конвертировать полученное значение в единицы KSLOC .

Для прогнозирования *вероятной плотности дефектов* D_0 (или количества дефектов N_0) на момент начала системного тестирования используются существующие мультипликативные модели, например, модель Rome Laboratory [16]. Однако, большинство из этих моделей ориентировано на стандартный каскадный ЖЦ и с другой стороны , в коллективах накоплены собственные ретроспективные (исторические) данные для калибровки таких моделей. В этой ситуации для выполнения прогнозов дефектов предлагается использовать новый класс моделей – графические модели [112].

Прогнозирование плотности дефектов

Высокие темпы развития программной индустрии, изменчивость потребностей заказчиков и условий выполнения проектов способствуют широкому внедрению в проекты ПС новых адаптивных методологий разработки и моделей ЖЦ (так называемых agile-методологий, например, экстремальное программирование), в которых ЖЦ на основе статической каскадной модели заменяется дина-

мическим ЖЦ по модели "Обдумывание – Взаимодействие – Обучение". Для принятия решений в таких проектах используется, как правило, интуитивный подход и вероятностные рассуждения, основанные на собственном опыте менеджера проекта. Существует значительная неопределенность относительно влияния одних факторов качества на другие и на качество конечного программного продукта. Сказанное свидетельствует о потребности применения для управления качеством механизмов корректировки суждений по мере накопления опыта [94].

Средства для построения логически непротиворечивой схемы суждений с возможностью их пересмотра в свете новых данных предоставляет байесовский подход, который может быть положен в основу не только управления качеством, но и программными проектами в целом. Однако его непосредственное применение для разрешения задач программной инженерии долгое время усложнялось очень большим объемом расчетов условных вероятностей. Фактически, только с появлением *байесовских сетей доверия* он обрел практический смысл не только в этой, но и в других отраслях знаний [112].

С помощью графических моделей, в основе которых лежат байесовские сети, можно формулировать предположение о существовании зависимости между различными факторами качества, а потом последовательно "распространять" получаемые объективные данные наблюдений по сети. Семантическое описание дефектов с позиций байесовской сети приведено в таблице 3.2.

Таблица 3.2. **Плотности дефектов**

Название вершины	**Описание вершин сети и зависимостей переменных**
Остаток дефектов	Разница между плотностью внесенных дефектов в текущую версию ПрП и плотностью устраненных (откорректированных) дефектов
Внесенные дефекты	Сумма плотности новых внесенных дефектов и плотности остатка дефектов (от предыдущего этапа)
Внесенные дефекты (новые)	Плотность внесенных новых дефектов. Зависит от сложности решаемых задач в ПрО и совершенства (качества) процесса разработки
Остаток старых дефектов	Плотность дефектов, которые, согласно прогнозу, остались не выявленными в ПрП на прошлом этапе разработки (остаток минус фактически устраненные дефекты)
Устраненные дефекты	Плотность устраненных дефектов. Зависит от плотности выявленных дефектов в ПрП, а также точности их устранения (нормируемой на [0,1]). Биномиальный закон распределения значений переменной в вершине
Выявленные дефекты	Плотность выявленных дефектов. Зависит от плотности внесенных дефектов и качества проверки (нормируемой на [0,1]). Биномиальный закон распределения значений
Точность коррекции	Способность разработчика точно устранить дефект при коррекции. Чем больше значение, тем выше способность
Качество проверки	Способность верификатора найти дефекты в ПрП. Чем больше значение, тем выше способность
Качество разработки	Способность разработчиков предотвратить внесение дефектов при разработке. Чем больше значение, тем выше способность
Сложность проблемы	Связана с риском "сложности реализации требований" к ПрП. Чем выше оценка риска, тем больше сложность проблемы

Основные преимущества использования графических байесовских моделей для управления качеством – поддержка *прогнозирования* дефектов и *диагностики* наиболее вероятных причин (источников) их возникновения, а также простота модификации посредством действующих эффективных алгоритмов и доступных инструментов [19].

Одна из базовых моделей является модель прогнозирования дефектов верхнего уровня и может быть детализирована в исходных вершинах (без родителей). Модель дефектов является результатом определения множества факторов качества, анализа причинно-следственных связей между ними, комбинирования качественных (экспертных) и количественных оценок их влияния на плотность дефектов, а также учета ограничений выбранного *доступного* инструмента моделирования Hugin Lite 6.5 [91].

Модель позволяет на начальном этапе прогнозировать, какой будет плотность дефектов D_{0i} в i-м ПрП, если не изменятся условия его разработки. Прогнозное значение плотности дефектов используется для нахождения вероятного прогнозного значения безотказности ПрП $R_i(t)$.

3.6. Анализ достижения уровня качества

Анализ альтернатив достижения целевого уровня завершенности i-го программного приложения базируется на сравнении прогнозируемого значения безотказности $R_i(t)$ с установленным значением q_i.

Для проведения анализа должны быть предварительно определены такие *критерии* для принятия решений по управлению качеством i-го ПрП:

q_i – установленный *целевой уровень безотказности* i-го ПрП;

D_i^* – максимальное значение плотности дефектов в ПрП, которое не будет препятствием для достижения целевого уровня q_i (*целевой уровень дефектов* в ПрП), определяется из уравнения

$$q_i = \exp[-D_i^* I_i \cdot (1 - \exp(\frac{\lambda_{0_i}}{D_i^* I_i} \cdot t))];$$

L_0 – минимально допустимая вероятность прогнозного значения плотности дефектов (*приемлемый уровень уверенности* менеджера проекта);

максимально допустимые *отклонения*:

σ_r – прогнозного значения безотказности $R_i(t)$ от целевого значения q_i;

σ_d прогнозного значения плотности дефектов D_{0i} от значения D_i^*;

σ_d – полученной наибольшей вероятности прогнозного значения плотности дефектов $L(D_{0i})$ от приемлемой L_0.

В предположении, что на ранних стадиях ЖЦ $q_i \geq R_i(t)$ и $D_{0i} \geq D_i^*$, следуют такие альтернативы относительно последующих действий по обеспечению качества i-го ПрП:

1. Если прогнозирование выполняется с приемлемым уровнем уверенности $L(D_{0i}) - L_0 \geq \sigma_p$, тогда:

а) если $| q_i - R_i(t) | \leq \sigma_r$, то продолжать выполнение проекта "как есть",

б) если $|q_i - R_i(t)| > \sigma_r$, то принимать решение, учитывая оценки ресурсов проекта, в частности времени до завершения разработки, времени и трудоемкости системного тестирования, а также возможностей совершенствования процессов ЖЦ ПС.

2. Если уровень уверенности при прогнозировании $|L(D_{0i}) - L_0| < \sigma_p$, то это не позволяет принять однозначного решения о состоянии разработки, целесообразно вычислить $R_i(t)$ в диапазоне близких значений D_{0i} и оценить относительные отклонения $R_i(t)$ от q_i для каждого из них, имея в виду, что D_{0i} – это *плотность* дефектов в ПрП, размер которого вычислен в условных единицах функциональности, т.е. фактический диапазон значений $R_i(t)$ может быть достаточно широким.

3. Если уровень уверенности при прогнозировании $L_0 - L(D_{0i}) > \sigma_p$, т. е. вероятности распределения значения D_{0i} ниже приемлемого уровня уверенности, или с самого начала проекта нарушаются исходные предположения для ранних стадий ЖЦ ПрП, это свидетельствует о несовершенстве применимой графической модели (малое количество учтенных факторов дефектов, некорректно определены априорные распределения переменных в вершинах без родителей и т. п.) и необходимости ее уточнения.

Если в ходе выполнения проекта после стабилизации кода *i*–го ПрП (при приближении разработки к завершению) не наблюдается тенденция к снижению плотности дефектов (по результатам прогнозирований в контрольных точках проекта t_s, t_{s+1}... получаются значения D_{0i}, для которых $D_{0i} - D_i^* > \sigma_d$), то это свидетельствует о несовершенстве процессов разработки (проектирования, программирования, верификации). Одно из возможных решений – передача ПрП в группу тестирования для выполнения процесса тестирования [21].

Проблема тестирования состоит в определении ресурсов (в частности, времени и денежных средств), адекватных важности ПрП и его модулей для функционирования системы, и решении задачи оптимального выпуска ПС. Может быть применен, например, так называемый риско-операционный подход к решению этой задачи, базирующийся на дифференцированном распределении времени тестирования между модулями ПС с учетом риска их отказов во время использования ПС [22].

3.6. Задачи оценки качества сложных систем

Сущность подхода к оценки качества с позиций завершенности состоит в усовершенствовании процесса конструирования качественных членов семейств ПС и генерации новых ПС с регулированием показателей качества сложных ПС из готовых КПИ и обязательной оценки показателей качества .

К новым задачам процесса проектирования ПС относятся:

1) моделирование качества семейства ПС и оценивание качества сгенерированных артефактов на каждом процессе инженерного проектирования ПрО;

2) подбор компонентов из репозитория, которые удовлетворяют требованиям ПС для их включения в состав члена семейства с близкими функциям для семейства;

3) верификация КПИ, взятых из репозитория ИТК, которые включаются в состав ПС разрабатываемого СПС;

4) тестирование компонентов, интерфейсов связанных между собой КПИ в членах ПС или СПС;

5) проверка показателей качества членов ПС в СПС с точки зрения реализации функциональных требований к ПС или семейству.

Формулировка задач для их реализации:

1. Обеспечение качества семейства ПС в процессах инженерного проектирования предметной области;

2. Решение задачи принятия решений по управлению вариантами ПС из КПИ путем конфигурации конечных программных продуктов в среде ИТС ГП.

Задача 1. Сущность задачи состоит в моделировании набора совместных не функциональных требований к семейству ПС и их спецификации для рассмотрения тех из них, которые могут конфликтовать с показателями качества:

1) высокая надежность → низкая эффективность,

2) высокая эффективность → низкая модификация,

3) высокое качество → высокие затраты и др.

Спецификация конфликтных требований к качеству отмечается в "матрице конфликтов", называемой "крышей дома качества" с указанием влияния отдельных характеристик друг на друга – "положительный, отрицательный или никакого влияния". Предполагается метод анализа иерархий для приведения в порядок спецификаций, взвешивания характеристик и оптимизации их целевых значений.

Задача 2. Цель задачи – проведение верификации описания компонентов ПрО, сбор данных и экспертная оценка полноты, точности, наличия спецификации КПИ и т. п. Верификация выполняется методом формальной инспекции, прототипирования и сценариев тестов и т. п. Инспекция ПС проводится вручную опытным экспертом (или экспертами) с использованием опросных карт и форм данных для измерения показателей. Результаты анализа выполняются с помощью данных, размещенных в репозитории. К ним отнесены XML-шаблоны спецификации требований, описания артефактов домена, зависимостей между ними, заданными средствами XML-link.

Задача 3. В рамках научных проектов разработаны новые методики количественного измерения и оценки показателей качества программ. Были получены такие оригинальные результаты:

1) модель качества и оценка надежности ПП, включая КПИ, которые включены по своим функциям в состав ПС;

2) модель принятия решений в процессе управления качеством, основанная на байесовских методах и методах системного контроля надежности на ранних процессах ЖЦ, количественного измерения требований к надежности и прогнозирования дефектов в элементах ПС;

3) модель распределения надежности системы из компонентов на основе функции полезности продукта – $Q_{nc} = \sum_{s=1}^{m} w^*_s \cdot r_s$ в зависимости от приоритетов w^*_s и надежности $q_j = \prod_{n \in E_j} r_n$ отдельных компонентов;

Разработанные модели и методики их использования прошли испытания при создании систем ПС МО Украины. Далее они развиваются для класса семейств продуктов, создаваемых из объектов, КПИ и сервисов.

3.7. Эталонная модель качества оценки показателей ПС

Для оценки показателей качества сложных систем используется стандартная модель качества, которая для всех видов и типов ПС имеет вид [17, 39]

$$M_{кач} = \{Q, A, M, W\},$$

где $Q = \{q_1, q_2, ..., q_i\}$ $i = 1,..., 6$, – множество характеристик качества (*Quality – Q*); $A = \{a_1, a_2, ..., a_j\}$ $j = 1,..., J$, – множество атрибутов (*Attributes – A*), каждый из которых фиксирует отдельное свойство q_i – характеристики качества; $M = \{m_1, m_2, ..., m_k\}$ $k = 1,..., K$, – множество метрик (*Metrics – M*) каждого элемента a_j атрибута для проведения измерения этого атрибута; $W = \{w_1, w_2, ..., w_n\}$, $n = 1,..., N$, – множество весовых коэффициентов (*Weights – W*) для метрик множества *M*.

В стандартах качества и в ядре знаний SWEBOK (www.swebok.com) 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).

Далее описаны характеристики качества q_1 _ q_6 и формулы их оценки общего вида $q_1 = \sum_{j=1}^{6} a_{1j} m_{1j} w_{1j}$.

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 приведены ниже.

a_{11} : функциональная полнота – свойство компонента, которое показывает степень достаточности реализованных в нем 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, заданным в требованиях (*m*):

$$a_{11} = \sum_{i=1}^{N} F^c \Big/ \sum_{j=1}^{K} F^m ;$$

a_{12} : корректность – атрибут, который указывает на степень соответствия каждой функции F^m, заданной в требовании, и каждой функции F^c, реализованной в ПС. При этом система обладает свойством полной корректности, если $F^m = F^c$, и частичной корректности, если $F^m \subset F^c$. Для большинства систем достаточно частичной корректности. Степень корректности компонента определяется, как степень функциональной корректности: $\sqrt{} = 1 - (card(F^m / F^c) / card\, F^m$;

a_{13} : точность – свойство, определяющее получение системой правильных результатов. Она может оцениваться отношением ∇ разности значения функции $F^c_i(D_i)$ компонента и значения функции $F^m_i(D_i)$, заданной требованиями на D_i входного набора к значению функции:

$$\nabla = \sum_{i=1}^{card\ F^m} \left(F_i^c(D_i) - F_i^m(D_i)\right) / \left(F_i^m(D_i)\right) / card\ F^m ;$$

a_{14} : интероперабельность – свойство компонента взаимодействовать с другими компонентами и операционной средой;

a_{15} : защищенность – атрибут, который показывает возможность компонента (системы) фиксировать дефекты, как следствие субъективных ошибок, или вызванные программными или аппаратными средствами, а также ошибок, связанных с данными. Оценку степени защищенности можно представить с помощью выражения $a_{15} = fal^z / fal$, где fal^z – количество дефектов, от которых компонент защищен; fal – общее количество дефектов в компонентах или ПС;

a_{16}: согласованность – атрибут, который показывает степень соблюдения стандартов, правил и других соглашений процесса разработки, и оценивается экспертно.

Таким образом, характеристика функциональности 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 ПС будем определять как вероятность того, что компоненты системы или сама система функционируют безотказно в течение фиксированного периода времени в заданных условиях операционной 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 в требованиях, проектировании и выполнении. Отказы и ошибки в программных компонентах могут появляться на заданном промежутке времени функционирования компонента/системы [26–30].

Свойства каждого атрибута надежности:

a_{21}: безотказность определяет функционирование системы без отказов программных компонентов или оборудования. Если компонент содержит дефект, вызванный субъективными ошибками при разработке, то во множестве $D=\{De/e \in L\}$ всех дефектов, можно выделить подмножество $E \subseteq D$, для которого результаты не соответствуют функции F^m, заданной в требованиях на разработ-

ку. Вероятность p безотказного выполнения компонента на De, случайно выбранном из D среди равновероятных, имеет вид $p = 1 – (card\ (E) / card\ (D))$.

Отказ (failure) показывает отклонение поведения системы от предписанного и система перестает выполнять заданные ей функции. Кроме того, 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 является критическим, и используется термин defect, который означает либо fault (причина), либо failure (действие). Связь между ними такая: fault → error → failure. Отказы в ПС являются типичными: внезапные, постепенные, перемещающиеся (сбои). Они могут возникать естественным путём, вноситься человеком или внешней операционной средой в период создания или эксплуатации системы, быть постоянными или носить временный характер.

Наработка на отказ как атрибут надежности определяется как среднее время между появлением угроз, нарушающих безопасность, и обеспечивающих трудно измеряемую оценку ущерба от соответствующей угрозы.

Для вычисления среднего времени T наработки на отказ применяется формула

$$T = \sum_{i=1}^{De} \nabla t_i^E \big/ N ,$$

где ∇t_i^E – интервал времени безотказной работы компонента i-го отказа; N – количество отказов в системе.

a_{22} : устойчивость к ошибкам – свойство компонентов системы, которое указывает на способность ПС выполнять функции при аномальных условиях (сбоях аппаратуры, ошибках в данных и интерфейсах, нарушениях в действиях оператора 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, указывающее на способность возобновлять функционирование системы после отказов и восстанавливать в ней поврежденные компоненты и/или данные для повторного исполнения.

Среднее время восстановления компонента можно определить по формуле

$$T = \sum_{i=1}^{De} \nabla t_i^b \big/ D ,$$

где ∇t_i^b – время восстановления работоспособности компонента после i-го отказа; De – количество дефектов и отказов в системе.

a_{24}: согласованность – атрибут, который отражает степень соблюдения стандартов, технологии, правил и других соглашений на стадиях разработки и тестирования системы для поиска разного рода ошибок разработки.

Некоторые типы систем реального времени, безопасности и другие требуют высокой надежности (недопустимость ошибок, точность, достоверность и др.), которая в значительной степени зависит от количества оставшихся и не устраненных ошибок в процессе ее разработки на этапах ЖЦ. В ходе эксплуатации ошибки также могут обнаруживаться и устраняться. Если при их исправлении не вносятся новые либо вносится их меньше, чем устраняется, то в ходе экс-

плуатации надежность системы непрерывно возрастает. Чем интенсивнее проводится эксплуатация, тем интенсивнее выявляются ошибки и быстрее растет надежность. К факторам, влияющим на надежность ПО, относятся:

1) угроза нарушения безопасности системы;

2) совокупность угроз, приводящих к нарушению системы или среды и др.

Надежность постоянно изучается и развивается. Новое ее направление – инженерия надежности ПО (Software reliability engineering – SRE), которая ориентирована на решение следующих задач [20, 28]:

1) измерение надежности, т. е. проведение ее количественной оценки с помощью предсказаний, сбора данных о поведении системы в процессе эксплуатации и современных моделей надежности;

2) разработка стратегии, метрик конструирования КПИ и ПС в целом и среды функционирования, обеспечивающей надежность работы системы;

3) применение современных методов инспектирования, верификации, валидации и тестировании ПС в процессе разработки и эксплуатации.

Количественная оценка характеристики надежности системы по всем ее атрибутам и метрикам имеет вид:

$$q_2 = \sum_{j=1}^{4} a_{2j} m_{2j} w_{2j} .$$

Удобство применения – это множество свойств ПС, которые показывают на необходимые и пригодные условия ее использования. Эта характеристика определяется на множестве эргономичных атрибутов и включают:

a_{31} : понимаемость означает усилие, затрачиваемое на распознавание логических концепций и условий применения ПС;

a_{32} : изучаемость означает усилия пользователей при определении возможности применения ПС посредством анализа документации;

a_{33}: оперативность – реакция системы при выполнении операторов и операционного контроля;

a_{34}: согласованность – соответствие ПС требованиям стандартов, соглашений, правил, законов и предписаний.

Все атрибуты оцениваются экспертами, которые в зависимости от их уровня знаний, дают соответствующие качественные значения. Формула оценки данной характеристики имеет вид: $q_3 = \sum_{j=1}^{4} a_{3j} m_{3j} w_{3j} .$

Эффективность – множество атрибутов, которые показывают взаимосвязь между уровнем ее выполнения, количеством используемых ресурсов (аппаратуры, расходных материалов и др.), услуг штатного обслуживающего персонала и др.

К ним относятся:

a_{41} : реактивность – время отклика, обработки и выполнения функций компонента/системы;

a_{42} : эффективность – количество используемых ресурсов при выполнении функций ПС 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, затрачиваемые на проведение модификаций (корректировка, усовершенствование и адаптация) при изменении среды выполнения, требований или спецификаций. Данная характеристика в модели качества состоит из следующих атрибутов:

a_{52} : анализируемость – необходимые усилия для диагностики отказов в ПС или идентификации частей, которые будут модифицироваться;

a_{53} : изменяемость – усилия, которые затрачиваются на модификацию компонента, удаление ошибок или внесение изменений, дополнение новых возможностей в систему или среду функционирования;

a_{54} : стабильность – риск проведения модификации компонента /системы;

a_{53} : тестируемость – усилия при проведении верификации в целях обнаружения ошибок и несоответствий требованиям (валидация), а также на необходимость исправления обнаруженных ошибок и проведения сертификации системы;

a_{54} : согласованность – соответствие данного атрибута определенным стандартам, соглашениям, правилам и предписаниям.

Количественная оценка данной характеристики имеет вид

$$q_5 = \sum_{j=1}^{3} a_{5j} m_{5j} w_{5j} .$$

Переносность – множество атрибутов, указывающих на возможность компонентов системы приспосабливаться к работе в новых условиях среды выполнения. Перенос компонентов или ПС на другую платформу или среду связан с совокупностью действий, направленных на обеспечение возможности функционирования в новой среде, отличной от той, в которой система создавалось. К атрибутам данной характеристики относятся:

a_{61} *:* адаптивность – усилия, затрачиваемые на адаптацию системы к различным операционным средам. Этот атрибут можно представить в виде $a_{61} = Za / Zd,$ где Za – затраты на адаптацию к новой операционной среде; Zd – затраты на разработку новой системы для новой операционной среды;

a_{62} : настраиваемость для запуска или инсталляции ПП в другой среде;

a_{63} : сосуществование специального ПО в среде действующей системы;

a_{64} : заменяемость – возможность взаимодействия с другими программами при совместной их работе, инсталляции или адаптации системы;

a_{65} : согласованность со стандартами или соглашениям по правилам переноса программной системы в другую среду.

Количественная оценка данной характеристики имеет вид

$$q_6 = \sum_{j=1}^{5} a_{6j} m_{6j} w_{6j} .$$

Комплексная оценка. На основе полученных количественных характеристик вычисляется итоговая оценка путем суммирования значений отдельных показателей и сравнения их с эталонными показателями ПС. Если в требованиях к ПС

установлено несколько показателей, то просчитанный показатель умножается на соответствующий весовой коэффициент, а затем суммируются все показатели

$$Q_{com} = \sum_{j=1}^{6} q_j \cdot$$

Если ПС содержит *N*-сом, то комплексная оценка качества системы (sys):

$$Q_{sys} = \sum_{l=1}^{N} Q_{com}^{l} \cdot$$

Таким образом, данный подход к аналитической оценке атрибутов показателей качества ПС позволяет получить количественную оценку качества.

Глава 4. ТЕСТИРОВАНИЕ И ЭКСПЕРТИРОВАНИЕ ПС

Основным направлением работ отдела в течение многих лет была проблематика разработки ПС, включающая в себя аспекты проверки правильности путем тестирования отдельных компонентов, КПИ и ПС, а также экспртирования создаваемого ПП и его процессов [113 – 116].

4.1 Модель тестирования и определение оптимального времени

Тестирование модулей и компонентов состоит в обеспечении на тестах следующих критериев: выполнение программы не менее одного раза на совокупности тестов, включающих в себя наборы входных и выходных данных; тестирование функций хотя бы один раз; тестирование межмодульных интерфейсов.

Вначале выполняется автономное тестирование отдельных модулей ПС нижнего уровня без вызова других модулей, затем выбирается очередной модуль, непосредственно вызывающий уже проверенные. Выполняется сборка модулей, а затем их тестирование в комплексе. Затем проводится тестирование интерфейсов, т.е. проверка правильности вызовов модулей и передачи экспортируемых и импортируемых значений параметров. Многие тесты, которые использовались при тестировании отдельных компонентов, могут использоваться при тестировании интерфейсов.

При тестировании объединенных модулей могут быть найдены ошибки в интерфейсных посредниках. Завершающим процессом тестирования является тестирование функций и их комплексные испытания.

Концептуальная модель процесса тестирования ПС и семейства ПС из готовых ресурсов имеет вид

$$SFT = \langle TM; TD, TA, Env \rangle,$$

где *TM* – подпроцесс управления тестированием; *TD* и *TA* – подпроцессы тестирования артефактов предметной области и приложения; *Env* – концептуальная и информационная среда процесса тестирования ПС.

При этом все три подпроцесса имеют унифицированное формальное представление:

$$TM = \langle\ Task(TM, TD, TA), En(TM), CM(TM)\ \rangle,$$

$$En(TM) \cup En(TD) \cup En(TA) = Env$$

где *Task* – задачи, разрешимые при выполнении соответствующего подпроцесса; *Env* – концептуальная и информационная среда соответствующего подпроцесса; *CM* – подмодель координации операций соответствующего подпроцесса.

Схема концептуальной среды *Env* задается выражением

$$Env = TG \cup SG \cup T \cup P \cup RG \cup RP$$

где *TG* и *SG*– тесты активы КПИ и программные КПИ; *T* и *P* – тесты и тестированные ПС; *RG* и *RP* – отчеты о выполнении тестовых КПИ и тестов.

Согласно этой модели формируются данные об интенсивности ошибок для организации оценки надежности в модели качества ПС.

Таким образом, тестирование компонентов и ПП находилось в центре внимания исследований, направленных на обеспечение методов выявления ошибок, поиска дефектов, отказов и сбоев программ, которые были вызваны разными нерегулярными ситуациями в ПП или аварийным прекращением функционирования компонента или всей ПС. Эти исследования проведены специалистом отдела Т. М. Коротун. Результаты освещены в ряде работ [30, 32, 35] и защищены в ее в диссертации. К ним относятся:

1) модель процесса тестирования с учетом рисков отказов и их оценки в процессе эксплуатации ПП;

2) математическая модель определения оптимального времени тестирования компонентов t_e^* с максимальной прибылью в зависимости от функции возрастания надежности, интенсивности $\lambda(t)$ и риска $C_м$;

3) технология тестирования ПП и сбора информации о всех видах ошибок для определения оптимального времени тестирования отдельных программ ПС и их семейств.

Определение оптимального времени тестирования

Эта задача выполняется с учетом типичных областей рисков и уровней угроз при отказах ПС, экономического взноса отказов модулей в убытки пользователей и стратегий тестирования.

Для каждого модуля введено понятие *полезности* (прибыльности). Тестирование считается прибыльным, если его стоимость и стоимость устранения дефектов не превышают ожидаемых потерь.

Стратегия тестирования основывается на анализе: риска отказов ПС, определения взноса каждого модуля в угрозы ПС; оценивания ожидаемого количества отказов и рисков отказов модулей.

Задачи оптимального времени тестирования определяются согласно следующих данным:

$H = \{H_i, i=1,2,\ldots,r\}$ – множество всех потенциальных угроз ПС, обусловленных отказами и дефектами в модулях;

$S = \{S_i, i=1, 2.,\ldots, n\}$ – множество всех возможных сценариев функционирования ПС;

t_e – время тестирования;

t_0 – время выполнения модуля в период эксплуатации ПС;

$C_м$ – взнос модуля в риск отказов ПС;

$R(t_0) = C_м\mu(t_0)$ – величина риска отказа модуля за время t_0;

$\mu(t)$ – функция роста надежности;

$\lambda(t) = d\mu(t)/dt$ – интенсивность отказов модуля;

$P(S_i)$ – вероятность того, что при реализации сценария S_i $(i=1,2.,n)$ будет выполняться данный модуль;

$P(H_j|S_i)$ – условная вероятность того, что при реализации сценария S_i причиной возникновения угрозы Hj будет отказ именно данного модуля;

C_j – стоимость последствий реализации угрозы H_j ($j=1, 2, \ldots, r$);

$C(t_e)$ – полная стоимость тестирования в течение времени te;

c_1 – стоимость единицы времени тестирования;

c_2 – стоимость устранения дефекта, который привел к отказу в процессе тестирования.

$\Delta R(t_0|t_e)$ – функция снижения риска отказа модуля за время его выполнения t_0, при условии, что модуль тестировался время te;

$K(t_0|t_e)$ – прибыль за время эксплуатации ПС, при условии, что модуль тестировался время t_e;

Необходимо найти такое время тестирования t_e*, чтоб прибыль была max

$$te^* = \{t_e: \mathrm{K}(t_0|\mathrm{t_e}^*) \geq K(t_0|t_e),\ 0 \leq t_e \leq t_{don}\}$$

где $tdon$ – максимально допустимое время тестирования.

Оптимальній час тестирования определяется по формуле

$$\Delta R(t_0|te) = Cм\,(\mu(t_0) - \mu(t_0 + t_e) + \mu(t_e))$$

где $C_м = \sum_{i=1}^{n}\left[P(S_i)\sum_{j=1}^{r}P(H_j \mid S_i)\cdot C_j\right]$ x$_0$ – взнос модуля в риск ПС.

Прибыль от тестирования находится за формулой

$\mathrm{K}(t_0|t_e) = \Delta R\ t_0|t_e) - C(t_e) = C_м\,(\mu(t_0) - \mu(t_0 + te) + \mu(t_e)) - c1\ t_e - c_2\mu(t_e)$.

Походная функции $K(t_0|t_e)$ по t_e равняется

$\mathrm{K}'(t_0|t_e) = Cм\,(\lambda(t_e) - \lambda(t_0 + t_e)) - c1 - c2\lambda(t_e)$

Значение оптимального времени t_e* получается как решение равнения $K'(t0|t_e) = 0$ в зависимости от конкретной функции $\lambda(t)$.

Метод оценивания риска отказов модулей заключается в последовательном решении следующих задач.

1) идентификации возможных угроз ПС и оценки стоимости последствий угроз C_j, $j=1,2,..,\ .r$.

2) взноса каждого модуля в угрозы для ПС путем связи возможных отказов модулей с внешними угрозами.

3) оценки ожидаемого количества отказов модуля за время t_0, посредством функции $\mu(t_0)$ модели надежности.

4) вычисления риска отказов для каждого модуля $R(t_0) = C_м\mu(t_0)$ с учетом неравнозначности компонентов и величины риска их отказов в ПС.

Реализация задачи оптимального времени проводится с помощью *базового процесса* тестирования, включающего в себя подготовку, проведение и оценивание результатов тестирования. Он базируется на стандартах разных документов, метрик процесса, а также методах для решения задач тестирования. Результаты тестирования документируются в специальных шаблонах документов.

Данные модели и технология тестирования апробированы в прикладных проектах МО Украины, усовершенствованы для сложных ПС и СПС в рамках фундаментального проекта по генерирующему программированию (2007–2011) [92].

4.2. Экспертирование компонентов и систем

На основе всесторонней проверки разработки ПС сформировались методы экспортирования ПС и процессов их создания путем измерения и оценивания показателей качества компонентов и ПП в классе задач СОД:

1) оригинальная модель экспертизы ПС в процессах создания продукта;
2) тестирование ПС и сбор данных для метрического анализа;
3) модель качества с ориентацией на оценку надежности ПП;
4) модель распределения количественных требований по отдельным компонентам и с учетом приоритета показателей пользователя;
5) модель распределения надежности ПП из компонентов с функцией полезности ПП $Q_{nc} = \sum_{j=1}^{l} v_j^* \cdot q_j = \sum_{s=1}^{m} w_s^* \cdot r_s$ в зависимости от весомых коэффициентов w_s и надежности $q_j = \prod_{n \in E_j} r_n$ отдельных компонентов;
6) модель принятия решений по управлению качеством, включая методы контроля надежности, начиная с ранних стадий ЖЦ, измерения требований, тестирования ПС, оценки надежности с учетом интенсивности отказов и прогнозирования дефектов.

Модель экспертизы процессов

Названные модели экспертизы разработаны в рамках фундаментального проекта ГП и диссертационного исследования О. А. Слабоспицкой. Результаты исследований изложены в технических отчетах данного проекта и в защищенной диссертации [110] под руководством автора и в ряде монографий (twirpx,com), которая пользуется большим спросом в странах СНГ. Модели адаптированы для семейства систем ПС и описаны в коллективной е-монографии [33, 34, 35]. Модель качества и оценки стоимости ПП отражены и реализованы в инструментальном комплексе ИТК [117].

Идея экспертизы ПС, определение ее модели и процесса управления рисками, основанного на дерева ценности. Метод группового экспертного оценивания компонентов и ПП, базируется на общей экспертной теории применительно к процессов стандарта ЖЦ 12207.

Модель процесса экспертивно-аналитической оценки объектов ЖЦ имеет вид:

$$PM = < EE, O, R, OM >,$$

где *EE* – информационная среда процесса; *O* – онтология знаний о предметной области; *R* – ретроспектива результатов работы процесса; *OM* – модель операций процесса.

Подмодель *CM* неявно задает состав методов оценивания (*MF*) и актуализации (*MA*) для процедур технологии, обеспечивающей их полную поддержку, и модель интеграции методов.

Для обеспечения использования опыта оценивания (механизм M_5) предложен специальный формализм представления концептов онтологии *O*. Концепты он-

тологии *O* описываются в виде абстрактного класса фреймовой модели Н. Ной посредством идентификации его типизированных связей с другими элементами *O* (концептами и параметрами).

Экспертная оценка базируется на технологических нормативах, регламентах и результатах операций проектирования ПС в *базовом* процессе (*BP*), а также в процессе менеджмента *MP* (с участием заказчиков, менеджеров, оценщиков).

Производство ПС (Production Action – *PA*) можно представить кортежем:

$$PA=\langle P\in\{BP, MP\}; DA_{BP}=DC\cup DI\cup DE\cup DM; DA_{MP}=DC\cup DM \qquad (4.1)$$

где DA_{BP} – предметная область принятия решений по управлению *P*; *DC*, *DI*, *DE*, *DM* – подобласти предметной области программной инженерии (ПИ), соответствующие ее выделенным дисциплинам: экономической (*Economics*), производственной (*Industry*), инженерной (*Engineering*) и дисциплине управления (*Management*).

Задача достижение качества ПС при производстве *PA* предполагает многократное оценивание в среде процессов *P* из (4.1), т. е. делается установка текущих и прогнозных значений: критериев эффективности; уровней достижения для деловых процессов *P*.

Для критериев эффективности процесса *BP* сформированы следующие группы методов:

1) оценивания качества и трудоемкости проекта ПС;

2) специальные экспертно–аналитические для отдельных характеристик (функциональный объем и качество ПС, зрелость организации, мощность процесса ЖЦ и др.). Они оцениваются экспертно или экспертно-аналитически на процессах *P* в зависимости от ретроспективных данных;

3) специальные аналитические расчета показателей надежности и оптимальных параметров, полученных при тестирования ПС.

Кроме критериев эффективности, в процессах *P* возникает потребность оценки двух групп характеристик. К первой группе относятся факторы влияния на критерии эффективности процессов $P\in\{BP, MP\}$. Вторая группа – характеристики, имеющие статус критериев эффективности в подходах к повышению качества ПС в методологиях разработки ПС, которые определяются требованиями к оценке объектов и характеристик.

Оценка процессов

Деятельность по оцениванию в процессов *P* представлена системой унифицированных действий по оцениванию, информационно преемственных для объектов, сходных между собой, имеющих сходные характеристики и/или многократно контролируемых в ходе *P*. Каждое такое действие – установление значения характеристики, присущей типу управляемых (контролируемых) объектов Р и/или элементов деятельности по управлению ими. Эта характеристика служит актуальным критерием достижения целей усовершенствования и/или управления *P*. Она называется целевой, а объекты, для которых устанавливается ее значение, называются оцениваемыми объектами ПрП.

На основании нормативных методик процессов *P* зафиксированы потребности *P* в результатах действий по оцениванию:

1) получение достоверной информации для выработки и обоснования решений по управлению ходом *P* согласно его текущим целям;

2) усовершенствование процессов *P* единой средой производственной деятельности РА, обеспечивающей обоснованность управленческих решений и условий эффективной деятельности ее участников;

3) предоставление формального аппарата методологий проектирования архитектуры семейств ПС заданного качества.

Выявлена специфика процессов *P* и среды оценивания. Их методическому обеспечению свойственны альтернативность методов: и представление их многокритериальными древовидными моделями или известными зависимостями от древовидных подхарактеристик.

Концептуальные особенности ПрП включают: в себя потенциальные различия взглядов на характеристики и источники информации в *P* и различной ведомственной принадлежностью (структурные, функциональные, ролевые) и отношения классификации объектов в процессах реинжиниринга, разработки семейств и линий продуктов и дублирования функций объектов *P*.

Технологические особенности процессов *P* – это множественность и повторяемость действий с одними и теми же объектами и их характеристиками; информационная преемственность и общие ресурсные ограничения действий; множественность их субъектов и альтернативность способов выполнения; зависимость от информационного контекста – методических источников и документов сопровождения *P*; необходимость автономного и совместного оценивания объектов ПИ.

Результаты анализа использованы в составе математического аппарата экспертного оценивания.

Задачи экспертной оценки объектов

Математический аппарат экспертного оценивания в процессах $P \in \{BP; MP\}$ предназначен для решения формализованных задач оценивания с помощью экспертиз объектов, взаимосвязанных в единую информационную среду согласно потребностям *P*, их специфике и требованиям к действиям по оцениванию. Данный подход включает в себя:

1) определение задачи оценивания объектов ПрП, конструктивное для достижения целей действий по оцениванию путем адекватной интеграции специальных и автоматических методов в экспертизах;

2) формирование структуры математического аппарата решения таких задач в виде системы моделей и методов процесса оценивания;

3) математическую и программную реализацию элементов аппарата;

4) введение разработанных моделей, методов и программных средств в процессы *P* и их апробацию в соответствующих организациях.

В задачу методического аппарата входит сопоставление оценок аксиомы AP_1–AP_3 и конструктов, обеспечивающих удовлетворение потребностей процессов *P* после проведения экспертиз процесса оценивания. Они включают в себя: требования к функциям, агрегирующие потребности выбранных методов и требования методических источников; функции оценочных действий по решению задач в ходе *P;* механизмы реализации функций в экспертизах.

Средства организации процесса оценивания обеспечивают выполнение функций с помощью введенных механизмов. Модель процесса оценивания определяет информационную и технологическую среду реализации методов в его операциях. Она также задает направления развития выбранных многокритериальных методов для обеспечения механизмов и требования к их входным и выходным данным.

Средства интеграции оценивания в процессы P – типовые постановки задач оценивания и управления, а также методы их решения – предоставляют исходные для методов процесса оценивания и позволяют использовать его результаты для обеспечения потребностей.

Накопление опыта оценивания снизу–вверх обеспечивается иерархической структурой – за счет актуализации подчиняющего компонента на основании результатов, полученных на уровне подчиненного. Наоборот, использование опыта сверху–вниз осуществляется за счет целевых требований к элементам.

Технология экспертного оценивания объектов ПрО

Суть разработанной технологии (интегрирующего механизма M_1) – сопоставление задач оценивания множества унифицированных *подпроцессов оценивания*. Такой подпроцесс реализует отдельное действие по оцениванию для фиксированных альтернатив A, формируя обоснованные оценки $\{ch(a), a \in A\}$ при фиксации остальных элементов постановки как начальних данных.

Процесс оценивания представлен пополняемой системой подпроцессов. Они развертываются в процессах P на этапах фиксации концепции ПрП в информационной среде, соответствующей текущей концепции ПрП до ее очередного пересмотра. Подпроцессы формируют систему экспертиз, информационно взаимосвязанных для объектов действий по оцениванию: многократно контролируемых в процессах P; связанных аналогиями и/или отношениями классификации; обладающих целевыми характеристиками, для которых имеют место указанные отношения. Первоначально решается одна и та же задача оценивания при разных начальных условиях, а затем – разные задачи для разных (*сходных*) типов объектов и/или их целевых характеристик.

Для подпроцесса оценивания технология определяет четыре этапа последовательного решения задачи экспертного оценивания, методы их выполнения и результаты, представленные на рис. 3.12 (серым цветом обозначены их обязательные элементы). Для получения промежуточных результатов первых трех этапов (общей и детализированной постановки, а также оценки подхарактеристики) и окончательного результата четвертого этапа (для решения задачи предназначены процедуры (PG, PP, PE, PS) и методы *оценивания* ($MF=MG \cup MP \cup ME \cup MS$)). При этом концепция ПрП пополняется элементами этих результатов, интерактивно введенными как типы и объекты ПрП, с помощью процедур (PC, PE) и методов ($MA=CM \cup EM$) *актуализации*. Для обеспечения целостности среды оценивания процедура *пересмотра* концепции ПрП по результатам экспертиз предусмотрена вне подпроцессов на этапах ее пересмотра.

Описанная унифицированная структуризация результатов подпроцесса отображает все элементы постановки задачи экспертного оценивания. При этом ре-

зультаты очередного этапа служат входными данными для следующего. Вследствие этого образуется иерархия результатов этапов, названная *следом подпроцесса*. Именно она обеспечивает возможность повторного использования результатов процесса оценивания всеми его участниками на всех этапах подпроцессов.

Технология устанавливает требования к методам подпроцесса:

1) использование всех информационных источников (интерактивное формирование среды оценивания его агентами; ее формальный анализ и поиск в ней, агрегирование индивидуальных мнений экспертов);

2) формализация свойства качества экспертного решения с помощью показателя *обоснованности*;

3) максимизация *обоснованности* в одной экспертизе и ее последовательное повышение в экспертизах из подпроцессов.

Таким образом, метод оценивания проводится на процессе ЖЦ с общей информационной средой, адекватной потребностям и специфики производственной деятельности, связанной с изготовлением ПП.

4.3. Методы управления программным проектом

Менеджмент программного проекта является одной из проблем индустрии ПП, освещен в ряде зарубежных работ, в том числе в стандарте PMBOK (Project Managerment Body of Knowledge, 2005). Стандарт регламентирует управление работами команды исполнителей программного проекта с использованием общих методов управления, планирования и контроля работ (стартовые операции, планирование итераций, мониторинг и отчетность), управление рисками и конфигурацией ПП проекта. Исследования по менеджменту под руководством автора Н. Т. Задорожной привели к новым теоретическим и прикладным результатам [34, 35], а именно:

1) формальная модель управления документооборотом в информационной системе с учетом идей академика В.М.Глушкова, изложенных в монографии "Безбумажная информатика" (1982), и метода оценки всех видов ресурсов ПП;

2) метод формирования варианта плана X работ по сетевому графику, включающему последовательность ($l_i \in L$) работ, объем q_i , вид W_i, и ресурс $R =< R_L, R_S>$ *с* нормой использования ($NR_i \in NR$) и с учетом закона распределения случайных величин $F = \{F_1,..., F_r\}$, времени t планового периода $[t_0, T]$ и вероятности окончания работ исходя из оптимального плана $K(X^*)=\min K(X)$.

Рассмотрим основные положения проблемы управления документооборотом.

Предложены подходы и принципы моделирования документооборота в системе управления документооборота (СУД). Сформулирована концепция определения информационных характеристик документов и модели документооборота в СУД, разработана методика моделирования документооборота. Выделено два типа информационных характеристик документов – объем и временные характеристики. К объему отнесен размер документа (средний и максимальный) и количество документов, которые поступают за определенный промежуток времени на обработку в СУД. Временные характеристики документа определяют время его обработки в разных узлах нахождения документов, передачи их по сети и выполнения операций над документами и т.п.

Характеристики объема – это регулярная часть документа из последовательности повторяемых групп полей и нерегулярную часть документа, не содержащая повторяемых структур данных.

Выведены формулы расчетов характеристик объема документов:

средний объем $V=l_h+n_s k_s\, l_s^{\max}$;

максимальный объем $V_{\max}=l_h+ n_s^{\max} k_s\, l_s^{\max}$, где l_h – размер нерегулярной части документа; n_s – количество строк, которые заполняются для данного типа документов; k_s – коэффициент заполнения; $l_s^{\max}$ – максимальный размер регулярной части документа.

Временные характеристики документов включают в себя:

1) суммарные значения времен обработки разных типов документов в соответствии с их маршрутом;

2) время выполнения отдельных операций над документами в разных P_i и P_j узлах системы;

3) время передачи документов между разными узлами обработки в УИС.

Расчеты характеристик объема и временных характеристик выполняются в два этапа.

На первом этапе их значения вычисляются статически с предположением, что документы обрабатываются автономно и вычислительные ресурсы для других работ не используются. Второй этап допускает существование потоков документов разного типа и назначения. Расчеты времени прохождения и обработки документов выполняются формулами:

$T_i^c = V_i^g(1/R_i^g + 1/R_i^c + 1/R_{i+1}^g)$ – время, необходимое на перемещение документа из узла P_i в узел P_{i+1};

$T_i^d=t_i^{\,1}+t_i^{\,d}+t_i^{\,2}$ – время обработки документа в узле P_i;

$T = \sum_{i=1}^{r_1} T_i^c + \sum_{i=1}^{r} T_i^d$ – общее время обработки документа в соответствии с прохождением им разных маршрутов между узлами.

Моделирование документооборота в СУД осуществляется на базе модели информационных потоков документов в ИС распределенного типа. Главная цель этой модели состоит в определении аналитических зависимостей между значениями интенсивности потоков документов в СУБД, и общим временем их обработки и средств передачи данных. Эти зависимости используются для определения и оценки числовых результатов моделирования документооборота. При возникновении очередей увеличиваются значение временных характеристик, которые зависят от размера очередей и загрузки узлов обработки документов. Поэтому моделирование процессов прохождения документов и распределения их совокупностей для обработки отдельными АРМ в узлах системы соответствуют выполнению научно-аналитическому и контрольно-инспекционному управлению и т.п.

Данные результаты дисертационного исследования в области управления документоообороота получены Н. Т.Задорожной (2004), защищены в диссертации, опубликованы в учебнике ПИ (http://programsfactory.univ. kiev.ua) и менеджмента на сайте http://lib.iita.gov.ua/new/creators. а также прошли апробацию в системах автоматизации "Документооборот в образовании Украины", портале "Дети Украины", "Учитель новатор" (2004–2010) в Академии педагогических наук Украины.

Глава 5. CASE-СРЕДСТВА РАЗРАБОТКИ СЛОЖНЫХ СИСТЕМ

Сформированная автором концепция фабрики программ с конвейерным способом производства обсуждалась в отделе и на лекциях КНУ имени Тараса Шевченко и филиала МФТИ при Институте кибернетики НАН Украины. Были созданы разные средства поддержки разных аспектов сборочного производства ПП. Объекты сборки – модули повторного использования в ЯП (Алгол, ПЛ-1, Кобол, Фортран, Модула-2 и др.) и их интерфейсы в языках (MIL, MESA, IDL, API и др.), которые служили источником генерации модулей посредников (Р-код, stub, skeleton и др.) для каждой пары связываемых модулей, размещаемых в библиотеках ЭВМ или в республиканских фондах алгоритмов и программ. К системам автоматизации метода сборки модулей можно отнести те, которые сделаны в ИК АН УССР (Дельтастат, Проект, Маяк, Мультипроцесист, РТК, АПРОП), а также систем в других институтах бывшего СССР – СИМПР (МГУ), ПРИЗ (АН Эстонии), Альфа (Новосибирск) и в зарубежных – SUN ONC IBM, CORBA, Oberon, MS.net и др.. Метод конвейерной сборки программ был определен на основе анализа конвейера в автомобильной промышленности и включал в себя линии (ТЛ) процессов производства ПП на основе названных систем автоматизации. В это же время было определено понятие интерфейса ЯП и разноязычных модулей, как необходимых элементов сборочного производства ПП [117, 58, 56 – 60].

В настоящее время отделом программной инженерии были получены новые теоретические результаты, которые описаны в данной работе. Они внесли существенный вклад в технологию программирования и программной инженерии.

К 1990 г. с участием специалистов ИК сформулированы базовые понятия и атрибуты первых фабрик программ, а именно, линии производства, системы сборки ПП, язык спецификации модулей, методы интеграции (в частности, сборки), интерфейсы для передачи данных и преобразования нерелевантных типов данных ЯП, библиотеки модулей и операционные среды.

Концепция конвейерной сборки В.М.Глушкова была частично реализована в рамках АИС "Юпитер" (1982–1991). В ней впервые были созданы несколько видов ТЛ для изготовления прикладных программ (ввода и вывода данных, пакетов прикладных программ, задач статистики, численных методов и т. п.) для четырех технических объектов АИС. В связи с развалом СССР, работы по индустрии ПП выполнялись теоретически специалистами отдела "Программная инженерия" Института программных систем НАН Украины. Эта концепция постоянно развивалась и выразилась в формальзованном представлении парадигм программирования сборочного тиав, изложенных в разделе 2.

Отметим, что процесс разработки основ фабрик программ активно развивается за рубежом по многим направлениям: стандартизация качества (ISO/IEC 9000-1, 2, 3, 4, ISO/IEC 12598-1, 2, 3, 4, ГСТУ 9126, 9150 и др.), жизненного цикла (ЖЦ) ISO/IEC 12207 и типов данных общего назначения ISO/IEC 11404; систематизация разных видов деятельности (экономика, управление и др.) [41, 42]; индустриализация ПП (Product Lines, Fabric program); формализации языков

описания интерфейсов (IDL, API, XML, RDF, SIDL) и совершенствования сред – MS.VSTS, CORBA, JAVA, Grid и т. п.

Нами сформулирована концепция фабрики программ сборочного типа. Определена структура фабрики, принципы ее организации, управления и функционирования, а также методы индустриального производства ПП.

5.1. Классификация средств производства ПП

Концепция В.М.Глушкова последние десятилетия развивалась теоретически в фундаментальных и диссертационных исследованиях в рамках научных проектов ГКНТ (1992–1996) и НАН Украины (1997–2010). Сформулировано сборочное программирование, основанное на идеи ТЛ с процессами проектирования, сборки разноязычных программ со средствами преобразования несовместимых FDT типов данных. Разработана теория экспертиз ПП, тестирования и оценивания качества ПП. Специалисты института получили оригинальные научные результаты по многим аспектам производства ПП, которые опубликованы во многих статьях, монографиях и научных отчетах [89, 91]. Некоторые результаты приводятся в коротком изложении.

Структуризация программной инженерии (ПИ) на 10 knowledge areas of SWEBOK (Software Engineering of Body Knowledge, www.swebok.com) соответствует процессам стандарта ISO/IEC 12207, программе обучения Computing Curricula (2001, 2004) и направлена на получение знаний в этой области.

Рис. 3.12. Классификация дисциплин «Программной инженерии»

Производство разных видов сложных ПС и семейств систем требует новой структуризации программной инженерии, ориентированной не только на процессы ЖЦ и общие методы их поддержки, но и на теоретические и прикладные методы решения важных технологических задач производства ПП (инженерия, управление и т. п.).

Предложена новая классификация дисциплин, которая обеспечивает регламентацию разных видов работ в производстве ПП на фабриках программ. Это такие научные дисциплины (инженерная, управления, экономики, менеджмента и производства) ПП [40 – 42]. Сущность и назначение каждой из дисциплин представлена в разделе 1, а выше на рис. 3.12 приводится другая уточняющая схема.

Новые научные дисциплины положены в основу создания экспериментальной фабрики программ в новой среде генерации Eclipse.

5.2. Ресурсы фабрики программ. Их виды и использование

Исходя из теоретических исследований, закономерностей развития технологии программирования и анализа зарубежных видов фабрик программ на современных платформах компьютеров, новых идеях обеспечения взаимодействия разнородных программ с использованием теории организации фундаментальных FDT и общих типов данных, GDT (General Data Types – ISO/IEEC 11404), установлены общие черты, которые свойственны разным фабрикам. Это прежде всего операционная среда (типа SUN ONC, MS.Net, Oberon, Babel, Grid, Eclipse и др.), метод программирования (компонентный, структурный, сервисный и др.), средства (ЯП, Rational Rose, CLR, и др.) и инструменты, поддерживающие процессы линий изготовления ПП или разработки отдельных компонентов, а также библиотеки готовых продуктов, сервисное обслуживание разных аспектов производства (данных интерфейса, качества, управления, контроля, планирования, расчета разных затрат и др.). Главный ресурс фабрики – специалисты по производству программ (аналитики, программисты, инженеры, тестовики, контролеры и т. п.) [44, 58 – 60, 97, 98, 118 – 122].

Определение. Фабрика программ – это интегрированная инфраструктура сборочного производства ПП (компонентов, подсистем, систем, модулей (блоков) и семейств систем, АСУ, АСУТП и др.) из готовых элементов, предназначенная для выполнения государственных, коммерческих и других заказов на ПП [53].

Фабрика программ включает в себя комплекс средств, инструментов и сервисов для производства ПП на процессах ТЛ. Ядро фабрики – операционная среда и метод изготовления ПП (UML, компонентный, структурный, модульный, сервисный и др.). Обязательное условие работы сборочного конвейера – средства связи разноязычных программ, аналогично тому, как это реализовано в MS.Net.

Сервисный метод вводит в среду дополнительные возможности по предоставлению сервисов и обслуживанию, связанного с выбором и использованием необходимого сервиса.

Ресурсы для производства ПП на фабриках. К ним относятся следующие.

Технические ресурсы: платформы, процессоры (Intell, IBM, Apple, MS; коммуникации (OSI, TCP/IP; компьютеры пользователей; файлы и серверы; локальные и глобальные сети; электронная почта (e-mail); тестеры и т. п.

*Технологические ресурсы***:** библиотеки, репозитории готовых ПП (КПИ, Reuses, Assets, Applications, Domains, Systems); методики методов программирования сборочного типа; руководства, методики с языков интерфейсов объектов (IDL, API, DII, SIDL, XML, RDF и др,); стандарты (каркасы, шаблоны, контейнеры, процессы, проекты, системы и др.).

Общесистемные ресурсы: ОС, инструменты: клиент/серверные технологии; офисные системы (ридеры /райтеры форматов PDF, PS, HTML и т. п.); системы документооборота; утилиты (архиваторы, программы записи на носитель, конфигураторы и т. п.); средства защиты информации (антивирусные, парольные и др.); CASE-инструменты, трансляторы; графические инструменты; СКБД.

Человеческие ресурсы включают в себя группы разработчиков, служб управления и выполнения проектных работ (по планам, сетевым графикам), связанные с достижением качества, выявления рисков, формирования конфигурации, проверки правильности реализации проекта и т. п.

Стандарт ISO/IEC 12207 определяет следующие группы:

1) технико-технологической поддержки (изучения рынка, приобретения CASE, ПП, консультации сотрудникам и т. п.);

2) защиты информации (паролей, ключей защиты, проверки и т. п.);

3) технологической службы (сопровождения, поддержки ЖЦ, контроля и т. п.);

4) качества (SQA-группа) с функциями планирования и выполнения ЖЦ, проверки работ, контроля качества рабочих продуктов, документов ПП и т. п.;

5) верификации, валидации и тестирования компонентов или ПП на правильность задания требований, координации планов работ с менеджером, проверка правильности ПП в тестовой среде системы и др.;

6) руководителя проекта, который отвечает за финансовые и технические ресурсы, а также за выполнение проектных соглашений заказчика и управление разработкой ПП;

7) менеджера проекта, ответственного за разработку проекта, согласует требования, решения и планы работ и реализации по всем группам человеческих ресурсов по срокам и стоимости;

8) проектировщиков и программистов, которые отвечают за разработку проектных решений и программирование, разработку документов и разных выходных результатов;

9) руководителя конфигурации (ответственные за версию) ПП, который регистрирует версии ПП, сохраняет твердые копии и версии с размежеванием доступа к ним.

Эти группы необходимы при любом индустриальном коллективном производстве ПП.

Стандартные ресурсы. Комитет по стандартизации ISO разработал ряд стандартов программной инженерии, которые регламентируют порядок разработки ПП, управления методами программирования на фабриках программ [44]. Главные из них следующие.

Базовый процесс (БП) включает в себя операции изготовления ПП, оценки, измерения, управления изменениями и усовершенствованием БП, приведенному в стандарте ISO/IEC 15504-7 ("Оценивания процессов ЖЦ ПЗ. Установки на усо-

вершенствование процесса"). Наиболее важными среди стандартов: ГСТУ ISO/IEC 14598 "Оценивания программного продукта", стандарт ГСТУ ISO 15939 "Процесс измерения", серия стандартов ISO/IEC 15504 "Оценивания процессов ЖЦ ПО", базовые стандарты качества – ISO 9001 "Системы управления качеством. Требования", ГОСТ 2844–1994, ГОСТ 2850–1994 и 9126 регламентируют разные аспекты обеспечения качества ПП.

Ядро знаний SWEBOK – это стандарт из 10 разделов программной инженерии, распределенных по двум категориям. Первая – это методы и средства разработки (формирование требований, проектирование, конструирование, тестирование, сопровождение), вторая – методы управления проектом, конфигурацией и качеством и БП [44]. Методы ядра знаний соответствуют стандартным процессам ЖЦ с учетом потребностей конкретной фабрики программ и регламентированной последовательностью процессов, начиная от требований, разработки проектных решений, определения каркасов ПП и выбора готовых компонентов для "наполнения" его соответствующим содержанием.

Ядро знаний менеджмента проекта – это стандарт по управлению проектом – PMBOK (IEEE Std.1490 "IEEE Guide adoption of PMI Standard. A Guide to the Project Management Body of Knowledge), разработанный институтом PMI [44, 45] и содержит описание лексики, структуры процессов и областей знаний: *управления содержанием проекта* (планирования с распределением работ); *управление качеством* и контроль результатов на соответствие стандартам качества; *управление человеческими ресурсами* в соответствии с их квалификацией и профессионализмом.

5.3. Базовые основы средств индустрии программ

Программная инженерия (Software Engineering-SE), как дисциплина ТП разных программ и систем, впервые была определена в 1968 г. на научной конференции НАТО. За это время в SE сформировался широкий диапазон средств и методов, ориентированных на *качество, производительность и индустрию* ПП. Главные из них такие: стандарт SWEBOK (Software Engineering body Knowledge, www.swebok.com, 2001г.); международная программа обучения SE – Curricula–2004 (www.com.org/education/cc2004, www.intuit.ru); аппарат продуктовых линий (Product Lines) для производства ПС, семейств систем (www.sei.com.edu), набор стандартов, регламентирующих процессы построения и оценивания качества ПП; фабрики программ – AppFab Microsoft, IBM, Intel, Apple, Mac и др. с новейшими средствами и инструментами поддержки процессов разработки ПП с применением КПИ, reuses, assets, сервисы и т. п.

Большой вклад в развитие ТП был сделан в Украине. Академик В.М.Глушков инициировал производство серии вычислительных машин, технологий систем (АСУ, ОГАС, АСУ ТП), программ на уровне Кабмина СССР. Так, задачи изготовления типичных программ, их накопление в государственных и республиканских фондах алгоритмов и программ, как продукции производственно-технического назначения, и автоматизации их сборки в сложные системы, были утверждены постановлениями Кабинета Министров СССР еще в 70-годах про-

лого столетия. Как результат, в государстве было автоматизировано много областей военного и хозяйственного назначения и созданы новейшие теории, методы и средства технологии программирования.

В то же время предмет SE и технологии программирования изучались во многих университетах цивилизованных государств мира. Таким образом, более 30 лет готовятся специалисты для индустриализации и компьютеризации всех видов деятельности общества. Кабмин Украины утвердил Постановление №1719 от 21 декабря 2006 г. об обучении международной дисциплины SE во многих ВНЗ Украины и программу индустрии ПП до 2016 г.

В связи с этим в ИПС НАНУ созданы новейшие концепции и теории индустрии программ. Они представлены в контексте дисциплин SE. Студенты факультета кибернетики КНУ имени Тараса Шевченко и филиала МФТИ одни из первых начали изучать дисциплину SE, процессы ЖЦ, ТП и др. С их участием построена первая экспериментальная студенческая фабрика программ (http://programsfactory.univ.kiev.ua) по разработке и накоплению новых артефактов и программ при выполнении дипломных и магистерских работ, а также их занесению в репозиторий фабрики для доступа к ним других студентов и преподавателей ВНЗ Украины. Для поддержки индустриального производства ПП в ИПС НАНУ разработан ИТК ИПС на веб-сайти http://sestudy.edu-ua.net [1] для изготовления сложных ПС.

Индустрия – это производство разных видов ПП массового применения. Главным вопросом любой индустрии является не только выпуск соответствующей продукции, но и получение прибыли от этого. У нас за последние десятилетия отечественной индустрии ПП фактически нет. Хотя действуют предприятия (фабрики) коммерческого типа, которые выполняют работы по разработке ПП для разных фирм, которые финансируют их. Кроме некоторых научных исследований относительно индустрии ПП и обучения отдельным ее аспектам студентов в вузах по утвержденной программе Кабмина, у нас отсутствуют национальные фабрики программ, как это есть в России (их более 1600). Потому для уменьшения отставания этой отрасли от мирового уровня развития индустрии ПП необходимо, как утаверждал академик Б. Е. Патон на сессии НАНУ в 2006г., создавать свои теоретические и прикладные методы индустрии программ, применять их на практике получать доход, как это делают индусы, получая прибыль от программной продукции более чем 25 млрд долларов.

Для поддержки индустрии ПП нами разработаны теории экспертного анализа проектных решений с ПП, технология разработки, управления и экономического оценивания качества ПП, расходов и стоимости его создания. В рамках этих робот проведена классификация общих задач производства ПП в виде отдельных функциональных *дисциплин* SE*,* которые определяют разные аспекты деятельности по производству ПП на фабриках, включая научные, инженерные, управленческие, экономические, педагогические и т. п. [2, 3].

Предложенная классификация дисциплин SE, обеспечивает регламентацию разных видов работ по производству ПП на фабриках программ [16, 17, 35, 36].

5.4. Разработка ТЛ для фабрик программ

Под *фабрикой программ* понимается интегрированная инфраструктура сборки готовых программных элементов в ПС, необходимыми государственными, научными, коммерческими и другим организациями. Фабрика оборудуется продуктовыми линиями, набором средств, инструментов и сервисов для автоматизированного выполнения процессов в операционной среде [36, 37, 44]. Фабрика софтвера – это согласованный набор процессов, средств и других ресурсов для ускорения всего цикла создания тех или других ПП.

Фабрика базируется на некоторой операционной среде, ориентированной на автоматизацию производства ПП с помощью линий. С точки зрения информационных технологий фабрика представляет набор инструментов для перехода к индустрии ПП с целью увеличения производительности разработки продукта на каждом этапе ЖЦ с заданными функциями, архитектурой и качеством. Фабрики содержат в себе линии и соответствующий набор средств разработки простых и сложных ПП по ним. Линия разработки простых продуктов, как правило, соответствует ЖЦ, например, реализованный в среде MS.Net с использованием каркасов (framework), DSL-языков и др. Линия разработки сложных ПП может быть сборочного типа из готовых программных ресурсов, которые находятся в разных библиотеках и репозиториях.

Исходя из полученной практики автоматизированной сборки разнородных программ в ЯП и опыта современных фабрик программ, нами определены общие составные элементы, характеризующие любую фабрику софтвера [36]:

1) *готовые программные ресурсы* (артефакты, программы, reuses, assets, КПИ и т. п.);

2) *интерфейсные программы,* которые специфицируют паспортные данные готовых разнородных ресурсов в языке интерфейса (IDL, API, SIDL, WSDL и др.);

3) *метод разработки* программ на ТЛ (объектный, компонентный, аспектный и др.);

4) *операционная среда,* наполненная системными программными средствами и инструментами для поддержки индустриальной сборки разнородных ресурсов;

5) *технологические линии и продуктовые линии* для производства ПП.

Эти индустриальные элементы производства ПП, используются фирмами на фабриках программ (например, IBM, Microsoft, CORBA, и др.)

Остановимся на толковании линий, как составляющих индустрии ПП, дисциплин SE и подходов к обучению IT-специалистов в ВНЗ.

Аспект конфигурации ПС на множестве вариантных КПИ в ИТК является актуальным, поскольку варианты каждого КПИ содержат контексты его использования в разных ПС с контролем конфигуратора. ПС также рассматривается как готовый вариант продукта.

Принцип вариантности ПС – это способность продукта к расширению, изменению одних КПИ другими. Вариабельнисть обеспечивается механизмами идентификации и спецификации общих и отличительных черт КПИ на всех уровнях представления ПС – характеристик ПрО, а также механизмов конфигурации КПИ. Ообеспечение вариантных ПС из КПИ осуществляется с помощью

механизмов расширения ПС вариантными точками в интерфейсах КПИ. Они используются для принятия решений по созданию новых вариантов КПИ, накопленных в базе знаний проекта ПС. Процесс управления конфигурацией ПС состоит в управлении выбором КПИ и построении из них вариантных ПС.

Для этого среда ИТК дополнена информационными структурами вариантных КПИ, а также функциями введения (изъятия) отдельных КПИ, сопоставления между собой КПИ "по горизонтали" и "по вертикали" и их композиций с учетом требований к ПС. Для адекватного управления конфигурацией требуется согласованная интеграция с другими КПИ, включая нормативно-методическую и инструментальную поддержку. Для интеграции сервисов конфигуратор будет совершенствоваться.

Линии продуктов. Рассмотрим отечественную методологию построения ТЛ и методологию SEI USA из продуктовых линий (www.sei.com.edu).

Методология ТЛ определена в [8, 9], как этап технологической подготовки работ (ТПР) для создания специальных ТЛ или схем линий из процессов и операций, которые обеспечивают продуцирование программ будущей ПС с помощью ЯП и комплекса инструментов.

ТЛ комплектуется из процессов, которые соответствуют некоторой ПрО, стандартных инструментов и технологических модулей (ТМ) реализации специфики функций ПрО системными средствами современной операционной среды и комплекса нормативно-методического обеспечения (рис. 3.13). Набор процессов ТЛ создается с учетом требований международного стандарта ISO/IEC 12207 – 2007, ГОСТ 3918–1999. Процессы поддерживаются отобранными методами, средствами и инструментами, которые преобразуют состояния промежуточных элементов процессов, их выполнения в заданной среде и внесения изменений в них.

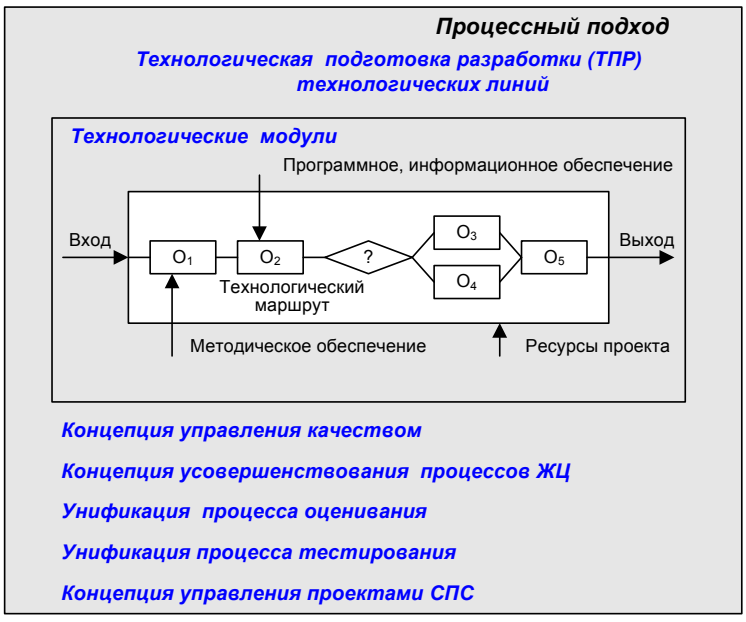


Рис. 3.13. Схема процессов ТПР

Технологический маршрут ТЛ описывается специальным языком с используемыми инструментами, ТМ и методиками управления последовательной целеустремленной деятельностью специалистов по выполнению процессов построения ПП и трансформации их данных, которые передаются между КПИ. Сегодня таким языком является BPMN.

Для ТЛ подбираются прикладные готовые ресурсы, КПИ, средства и инструменты порождения и реализации отдельных функций или элементов программ.

Все ресурсы связываются технологическим маршрутом, который упорядочивает процессы и операции ТЛ в структуру проектных решений по реализации и внесению изменений отдельных элементов ПП. К конечным операциям маршрута, как правило, относится операция оценки качества ПП с соответствующей методикой и инструментальными средствами [6].

Данный подход к определению конкретных технологий – оригинальный подход (Г. И. Коваль, Т. М. Коротун, Е. М. Лаврищева), не имеет аналогов. Впервые были сформулированы задачи ТПР, решение которых было направлено на определение процессов для ТЛ, которые реализуют классы программ в прикладных системах (АСУ, СОД, АСНИ и др.). Процессы и операции ТЛ определяются с помощью специального языка спецификаций PMDN [41].

Продуктовая линия (Product Lines) SEI включает в себя Product line (линию продуктов) и Product family (семейство продуктов, СПП, близко СПС). Они определены в словаре ISO/IEC FDIS 24765:2009(E) – Systems and Software Engineering Vocabulary как "группа продуктов или услуг, которые имеют общее управляемое множество свойств, удовлетворяющие потребностям определенного сегменту рынка или вида деятельности".

Согласно этому стандарту деятельность по разработке ПС проводится на линии с использованием инженерной и процессной моделей.

В *процессной модели* выделено множество процессов, выполняемых на двух уровнях инженерии ПрО, "для обеспечения повторного использования" (for reuse) и инженерии приложений "с использованием КПИ" (with reuse). Эти два уровня используют на линии сборки готовые КПИ, которые уменьшают время разработки СПП и улучшают уровень готовности ПС. Т.е., сборка СПС из КПИ является заключительной в цикле производственных работ данной модели в соответствии с требованиями и потребностями заказчиков рыночного ПП.

Модель инженерии включает в себя:

1) разработку КПИ, сборку КПИ в СПП и управление этими деятельностями [8].

2) планирование разработки КПИ и реализации каждого ПС из множества разработанных КПИ путем сбора их в СПП. Деятельность *по управлению СПП* из КПИ базируется на координации заданий организационного и технического управления каждым членом семейства.

При конструировании ТЛ и выполнении задач производства на линии предложено использовать новые дисциплины SE, которые изучают студенты КНУ факультета кибернетики для понимания информационных технологий и получения современных знаний по реализации разных видов работ (программирование, тестирование, измерение, оценивание и др.) при создании ПС и СПС на современных фабриках программ [45–51].

Каждая линия повышает производительность труда исполнителей, улучшает условия их работы, сокращает число сборщиков, повышает качество продукции и снижает себестоимость выпуска продукции на конвейере, о котором мечтал академик В.М.Глушков (1975). Он говорил "Пройдет 20-30 лет и сложные программы будут выпускаться по принципу сборочного конвейера, как в автомобильной промышленности". Студенты восприняли идею Глушкова и на практических занятиях по ТП и SE осуществляли под руководством автора реализацию спектра простых линий, создав экпериментальную фабрику программ КНУ для производства программ с КПИ и обеспечения их взаимодействия в среде ИТК через Eclipse, а также обучение студентов дисциплинам SE и программированию разным ЯП в основном в среде Майкрософт.

Структура соременной фабрики производства ПП. Идея ТПР построения технологических процессов и ТЛ постоянно обсуждается в публикациях и продолжает развиваться. Так, в последние годы появились новые технологические направления, в основе которых лежат процессы интеграции повторных компонентов (КПИ) и многоразовых компонентов и систем [123].

Инфраструктура разработки линии продуктов (рис. 3.14), кроме необходимых методов, средств построения и эксплуатации линий ПП, включают в себя руководящие материалы и методики. Хотя линии семейства продуктов зачастую отождествляют, линия продуктов для рынка может быть построена на базе определенного представителя семейства продуктов (разрабатываемого, например, по определенному заказу).

При построении линии для некоторого члена семейства определяются:

1) технологические ограничения, образцы и каркасы, которые могут использоваться на линии;

2) производственные ограничения, стратегии и методы;

3) набор средств и инструментов для разработки продукта на линии производства.

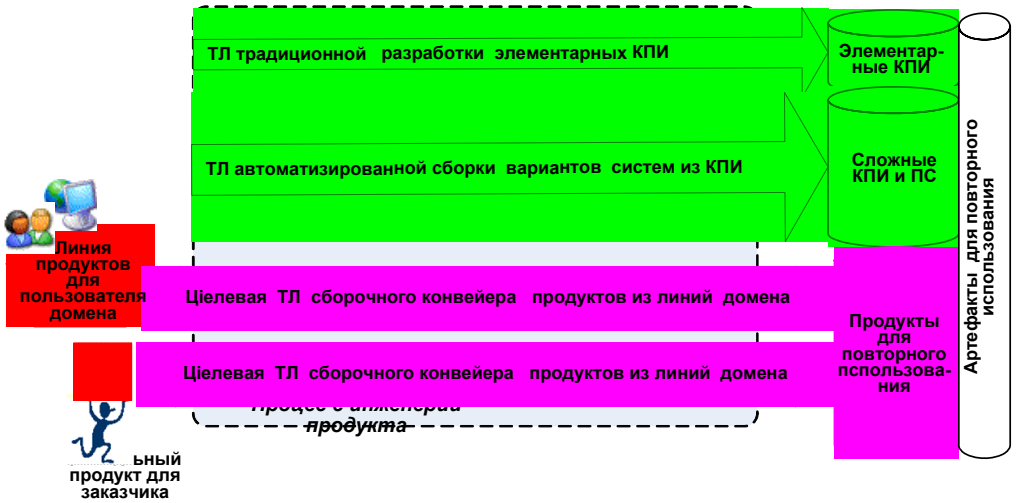


Рис. 3.14. Общий вид структуры фабрики программ

На основе этих данных уточняется структура линии и план создания ПП линии с учетом сроков, стоимости и требований к управлению производством ПП путем:

1) контроля плана работ и отслеживания хода построения продукта;
2) выявления рисков и управления ими при выполнении исполнительской деятельности в процессе проектирования линии производства;
3) прогнозирования стоимостных и технических ресурсов проекта;
4) применения технологии управления конфигурацией продукта;
5) измерения и оценки качества продукта.

Данное направление учитывает ориентацию на информационные технологии и КПИ, что улучшает процесс сборки готовых компонентов и КПИ в единый продукт на линии производства.

Сравнительный анализ двух технологических подходов – ТПР и инфраструктуры построения линий ТЛ показывает, что ТПР (1982–1999) не имел формализованного языка описания новых ТЛ. Продуктовые линии (2002–2005), способствующие созданию коммерческих продуктов из готовых программных продуктов также не формализованы. Оба направления имеют общие технологические цели, хотя и определяют разные пути построения линий производства программ. Сегодня комитет W3C создал новый язык описания процессов ЖЦ, из которых формируются ТЛ под создание определеннх типов продуктов (ППП, АСНИ и др.). Нами этот язык будет изучаться в КНУ и применяться при создании онтологии ЖЦ стандарта ISO/IEC 12207–2007.

Глава 6. CASE ИТК. ТЕХНОЛОГИИ, ЭЛЕКТРОННОЕ ОБУЧЕНИЕ

Методология изготовления СПС из готовых КПИ включает в себя: парадигмы сборочного программирования; модели взаимодействия и вариантности; комплекс технологических линий для построения разных элементов ПС; средства обучения фундаментальным основам программной инженерии. Предложенная нами методология отображает новые теоретические и прикладные аспекты технологии изготовления программных продуктов с учетом ядра SWEBOK (www.swebok.com) [39], ЖЦ стандарта ISO/IEC 12207, процессы которого адаптированы к отдельным ТЛ, включенным в ИТС, как инструментов производства СПС [19 – 123].

В данной методологии используется инженерия ПрО и доменов. В ней инженерная деятельность основывается на моделях проектирования (DDD, DSL, MDA, GDM и др.) СПС. В парадигме ГП главным аспектом производства программ является *генерация*, базирующаяся на представлении знаний о специфике ПрО и знаний, накопленных о методах, средствах и инструментах программной инженерии, которые необходимы для линий автоматизированного производства СПС из КПИ. В данной методологии КПИ, члены СПС описываются ЯП и предметно-ориентированными языками типа DSL. Процесс генерации этих описаний рассматривается как последовательная их *трансформация* от одного исходного ЯП до промежуточного и так до получения готового продукта.

Процесс сборки компонентов начинается с поиска готовых КПИ, согласно требованиям заказчика, и принятия решения о достаточности или нет реализованных свойств КПИ и сборки (конфигурации) их в системы или СПС.

С помощью набора линий ИТК (рис. 3.15) проводится: разработка КПИ, сервисное обслуживание компонентов и интерфейсов в репозитории сборка или конфигурация разноязычных КПИ в члены семейства или СПС; преобразование общих типов данных (GDT) стандарта ISO/IEC 11404 -2007 к фундаментальным (FDT); оценка качества КПИ и затрат на их разработку; использования веб-сервисов и обучения студентов аспектам производства ПП.

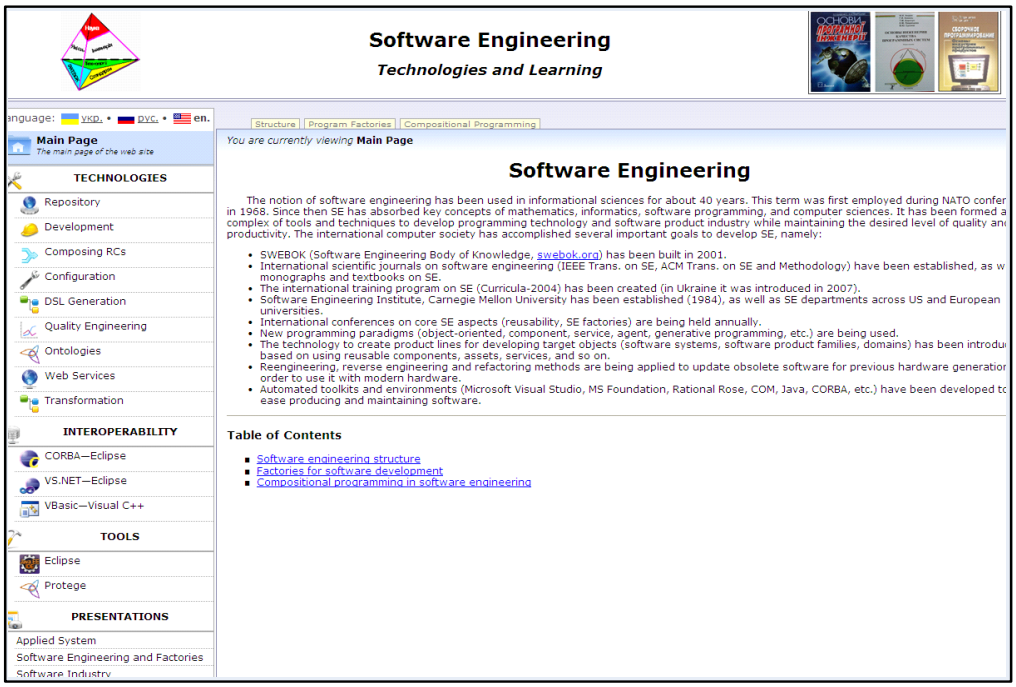


Рис. 3.15. Главная страница веб-сайта ИТК

ИТК построен как сайт http://sestudy.edu-ua.net.

6.1. Основные задачи ИТК

В данном комплексе реализованы специальные программные средства и CASE-интстументы поддержки разработанной ТП (рис. 3.16).

Теория формальной разработки ПС представлена в ИТК описанием методологии моделирования и проектирования ПС с помощью моделей MDA, MDD, GDM, предметно-ориентированных языков DSL, методов экспертиз, тестирования и оценки качества КПИ и ПС.

Технологические линии разработки КПИ и ПС ИТК включают в себя:

1) разработку КПИ и ПС из разноязычных КПИ;
2) взаимодействия программ, систем и сред;

3) сборки, интеграции и конфигурации КПИ в сложные ПС средствами Ant MS.Net и др.;

4) генерацию фундаментальных и общих данных и возможные преобразования данных для нерелевантных типов данных;

5) оценку качества отдельных КПИ и программного продукта;

6) онтологические средства поддержки ЖЦ, вычислительной геометрии и типов данных с помощью систем Protégé, Tools DSL MS.Net и др.;

7) электронное обучение программированию в языках C#, JAVA и аспектам программной инженерии по учебнику фабрики программ.

Рис. 3.16. Функциональная структура ИТК

Инструментальные средства общего назначения ИТК включают в себя

1) общесистемное обеспечение (CORBA, JAVA, VS.Net, IBM, Eclipse, MCF, Tools DSL MS.Net и др.);

2) специального назначения (Eclipse, Protégé) для моделирования онтологии моделей предметных областей и добавления КПИ в репозитории ИТК;

3) системы программирования с ЯП, Visual Basic, C++, JAVA , ЯП в рамках систем общего назначения VS.Net (C#, C++, Visual Basic), CORBA (C++, C, Lisp, Smalltalk, JAVA, Pascal, PL/1, Python и др.) и JAVA/RMI;

4) поддержки предметно-ориентированного проектирования – Tools DSL VS.Net, Eclipse-DSL, Work Flow и др.;

5) тестирования программ - Spec# и др.;

6) СУБД MS.Net для ведения базы данных репозиториев КПИ, интерфейсов и артефактов.

Многие проектные и технологические решения изготовления ИТК направлены на реализацию следующих задач ТП:

1) парадигм программирования (объектной, компонентной и сервисной) и средств обеспечения адаптивности сложных систем из готовых программных ресурсов;

2) линий для КПИ и ПС таким образом, чтобы они выполняли задачи путем сборки, конфигурации, тестирования и оценки качества как КПИ, так и ПС;

3) дистанционное электронное обучение современным языкам C#, JAVA, а также CASE-системам – Protégé, Eclipse, VS.Net, JAVA и др.

Данные задачи используют следующие базовые понятия ТП:

1) интерфейс, метод сборки, объектно-компонентный метод проектирования ПС из готовых КПИ;

2) предметно-ориентированные языки DSL и КПИ (Reuses, Assets, Services;

3) теория взаимодействия программ и систем, вариабельности (изменяемости и адаптации ПС к новым условиям современных сред), живучести (отказоустойчивости и восстанавливости систем) ПС;

4) концепция производства фабрик программ из ТЛ и продуктовых линий;

5) качество ПС и процессов ЖЦ;

6) концепции представления знаний КПИ в репозитории;

7) понятия CASE-инструментов – Protege, Eclipse, CORBA, Eclipse - DSL, Ant, C#, JAVA, Basic и др..

Реализованные в ИТК новые средства ориентированы на производство ПС и СПС из готовых КПИ по простым линиям обработки КПИ на разных ЯП, который фактически отображает идею сборочного создания современных программ, систем и их семейств.

В комплекс ИТК входит экспериментальная фабрика программ КНУ, реализованная студентами (А.Аронов А.Дзюбенко и И.Радецкий) под руководством автора, как и преподавателя курсов "Программная инженерия" и "Технологии программирования ИС". С 2006г. также работает сайт www.intuit.ru, в котором размещен учебник по методам и средствам программной инженерии, который преподавался автором. в филиале МФТИ при ИК имени В.М.Глушкова НАН Украины. Эта фабрика программ является самостоятельным веб-сайтом http://programsfactory.univ.kiev.ua. В ее состав входят линии, которые разработаны студентами для обучение разным вопросам технологий программирования, в том числе:

1. Линия проектирования программ в MS.NET.
2. Представление артефактов в языке WSDL.
3. Сертификация КПИ для репозитория.
4. Конвейерная сборка КПИ.

Фабрика сделана студентами к 90-летию академика В.М.Глушкова на факультете кибернетики, который он организовал в 1969г. К студенческому сайту обратилось более 15000 из разных стран мира. Он будет развиваться в направлении нанотехнологий для изготовления новых механизмов и веществ в рамках e-scitnse (биология, физика, химия и др.). Основные положения нанотехнологий рассматриваются ниже.

6.2. Функции и структура веб-сайта ИТК

Данный сайт разрабатывался как инструментарий ТП и одновременно демонстрировались отдельные моменты ПИ по новому курсу SE – Software Engineering (www.swebok.org) "Программная инженерия" на лекциях в КНУ. В результате появилась идея внедрить разработанный ИТК в систему обучения студентов и аспирантов аспектам ПИ.

В курс обучения были включены электронные технологии программирования в современных ЯП JAVA, C#, C++, Basic, а также обучения компьютерным предметам – программная инженерия, вычислительная геометрия и т. п.

Таким образом, была выбрана стратегия обучения многим аспектам технологии промышленного изготовления программ и систем. Для постепенной и последовательной реализации этой стратегии в ИТК взят современный аппарат дизайна веб-сайта, современные действующие системы, которые поддерживают аспекты технологии разработки программ.

Техника реализации сайта. Каждый раздел сайта содержит подразделы из ключевых слов, которые обозначают название линии разработки (их 11). Все разделы и подразделы построены по общему образцу. Разделы включают общее теоретическое описание, пример, иллюстрирующий смысл описания (в большинстве случаев разработан средствами одной из поддерживаемых сред ИТК), реализацию и выполнение примера на соответствующем инструменте этой среды.

В реализации данного сайта и отдельных линий технологии изготовления отдельных аспектов программ из готовых КПИ принимали участие сотрудники отдела (В.М.Зинькович, Л.И. Куцаченко в плане применения средств онтологии и веб-сайта научных командировок), магистранты МФТИ (А.Островский, И. Скотников), дипломанты КНУ имени Тараса Шевченко (И.Радецький и А.Аронов, А.Дзюбенко).

Для отображения структуры и содержания линий была выбрана архитектура, промежуточная между статическими веб-страницами и архитектурой Model-View-Controller (MVC).

Все страницы, отображающие статьи по тематикам сайта, строятся по единому шаблону, включающему в себя следующие основные детали:

1) заголовок, единый для всех страниц, содержащий баннер сайта и его название;
2) главное меню, включающее в себя панель выбора языка и ссылки для навигации по разделам сайта;
3) панель навигации, содержащая ссылки для перехода к различным подразделам текущей статьи;
4) строка текущего местонахождения;
5) содержание статьи;
6) "подвал", содержащий сведения об авторах сайта.

Все разделы сайта реализованы по одной схеме – **описание, пример, выполнение, выход (закрытие)**. Они задают:

1) описание смысла технологии на конкретной линии;
2) метод реализации на примерах программ;

3) загрузка exe-кода для выполнения на рабочем столе;

4) возврат в сайт.+

Формирование динамических компонентов страницы, кроме заголовка и "подвала", осуществляется с помощью языка программирования PHP. При этом используется древовидная структура представления разделов и соответствующих им статей, позволяющая без особых усилий формировать все вышеперечисленные детали страницы. Для долговременного хранения информации, наподобие заголовков статей и их содержимого, используется база данных SQLite, отличающаяся от аналогов отсутствием необходимости в выделенном сервере. В связи с частыми повторами в различных статьях однообразных элементов (например, нумерованных рисунков и таблиц с информацией о скачиваемых файлах), содержание статей, помимо стандартных HTML-тегов, может включать также XML-теги, которые перед отображением страницы преобразуются определенным образом в HTML при помощи препроцессора.

Первая страница веб-сайта ИТК содержит изложение сущности предмета "Программная инженерия" в виде описания ядра знаний SWEBOK (www.swebok.org), фабрики программ, технология программирования и др.

На каждой странице сайта в левой части дан перечень функций ИТК, в центре – их схемы реализации с учетом научных и проектные решений методологии производства программ в среде ГП, включая обслуживание репозитория, методику реализации линий сборочного производства в ИТК, взаимодействия программ, систем и технологии представления некоторых доменов средствами инструментальных систем среды ГП и КПИ в левой части (рис. 3.17, а).

Для выполнения раздела в окне сайта необходимо нажать на соответствующее семантическое название приведенных функций (рис. 3.17, б). Им соответствуют операции их выполнения технологий, взаимодействия и обучения.

Самый главный раздел "ТЕХНОЛОГИЯ" имеет следующий перечень реализованных технологий по следующим компонентам сайта:

1) Репозиторий КПИ со спецификациями и их паспортамив, а также их выбор и апробация. Это можно сделать и на студенческой фабрики программ http://programsfactory.univ.kiev.ua, являющейся составной частью ИТК;

2) сборочный конвейер разноязычных программ и компонентов в ПС с конвертированием несовместимых между собой типов данных;

3) конфигуратор КПИ в сложную структуру ПС с точками вариантности для внесения изменений в КПИ и ПС в среде Workflow MS.NET ;

4) реализация доменов прикладных областей в языке DSL (ЖЦ стандарта IO/IEC 12207-2007 с графическим и текстовым представлениями) на инструменте Eclipse-DSL и DSL Tool VS.Net;

5) CASE Softest для инженерия качества и иценки затрат и стоимости разработки ПС;

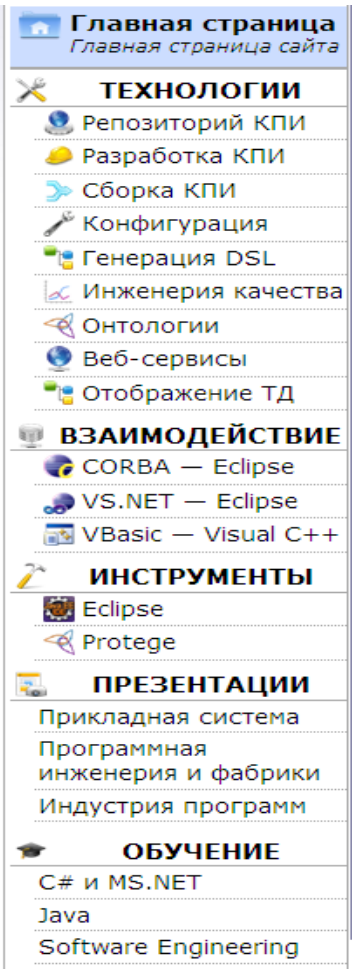
6) CASE Protege для создания онтологии предметной области на примере домена "Вычислительная геометрия";

7) веб-сервисы для установления взаимосвязей разных ПС в СПС, которые находятся на разных платформыах и средах;

8) отображение общих и фундаментальных (GDT и FDT) типов данных стандарта ISO/IEC 11404 в виде примитивов библиотеки и приближенной до системы GRID (на примере векторов, таблиц, и др.);

9) генерация ГОР и объединение их конфигуратором по модели вариабельности в структуру – программа, член семейства, СПС;

10) тестирование программы в среде ОС для получения правильного продукта и сбора сведений об отказах и ошибках, необходимых при оценке надежности.



а

Веб-сайт содержит разделы на 3.17, *а* и перечень технологий сайта на рис. 3.17, *б*.

ТЕХНОЛОГИИ ПОБУДОВИ ГОР і ПС:
Технология обслуживания репозиторию КПИ,
Технология разработки КПИ,
Технология сборки КПИ,
Технология конфигурирования КПИ,
Технология генерации описания КПИ в языке DSL,
Технология оценка затрат и качества,
Технология онтологии вычислит. геометрии, ЖЦ ISO/IEC12207
Технология веб-сервисов,
Технология генерация типов данных ISO/IEC 11404.

ВЗАИМОДЕЙСТВИЕ программ, систем и сред:
Модель CORBA - Eclipse-JAVA,
Модель VS.Net C# - Eclipse,
Модель Basic - C++.

ИНСТРУМЕНТЫ ИТК:
Система Eclipse,
Система Protégé

ПРЕЗЕНТАЦИИ ПС В ИТК:
Система ведения зарубежных командировок для НАНУ,
Слайды про ИТК, фабрики программ
Методологии построения ТЛ

ОБУЧЕНИЕ:
технологии программ в C# VS.Net и JAVA,
электронный учебник
"Программная инженерия" веб-сайту КНУ
http://programsfactory.univ.kiev.ua.

б

Рис. 3.17. Главная страница веб-сайта
а - название разделов веб- сайта, б - перечень технологий веб-сайта

Операционные среды (VS.Net, IBM, CORBA, JAVA, Eclipse), включенные в веб-сайт ИТК, реализуют процессы ЖЦ разработки разнородных программ и методы объединения в разные структуры ПП с помощью специальных механизмов связи в каждой среде.

6.3. Описание раздела сайта "Технологии"

Согласно рис. 3.17 далее дано описание подразделов, относящихся к технологии. Подразделы выделены самостоятельно, как на рисунке.

Репозиторий. Он работает в среде MS SQL Server 2005 и технологии ADO.NET Microsoft Framework.NET. Администратор БД проверяет нет ли дублирования в репозиторий новых КПИ, несовместимость паспорта данного объекта, а также право доступа к

КПИ. Главными функциями репозитория являются:

1) запись компонента и его паспорта в репозиторий;
2) выбор КПИ для его анализа и применения в новой ПС как заготовки;
3) проведение вычисления КПИ и вывод результата;
4) внесение изменений в КПИ и контроль версий.

С функциональной точки зрения репозиторий разделен на 3 панели: **описание, репозиторий, загрузка**.

Панель **Описание** – содержит информацию о средствах репозитория КПИ.

Панель **Репозиторий** содержит набор КПИ для пользователя;

Рабочая область этой панели разделен на 2 подобласти. В левой размещено дерево репозитория, в правой – описание компонента (его спецификация)

Панель **Загрузка** служит для спецификации паспортов КПИ по шаблону системы Etics Grid, занесения описания КПИ и паспорта в каталог репозитория, а исходный код в библиотеку.

По каждому КПИ формируется спецификатор (см. табл. 3.3 и 3.4)

Таблиця 3.3. **Формат спецификатора КПИ для репозитория**

Название	Описание	Согласие
Разработчик	Фамилия, имя, отчество, владельца компонента, который добавляется	Да
Дата создания	Дата создания КПИ автором (дата конечной, протестированной, специфицированной версии компонента)	Нет
Дата изменения	Дата внесения изменений в КПИ	Авто
Версия	Версия компонента КПИ	Нет
Платформа	Платформа, для которой создавался КПИ и на которой проверена его работоспособность	Да
Операционная система	Операционная система для создания КПИ и где проверена его работоспособность	Да
Размер	Общий размер КПИ (продукта, документации и др.)	Авто
Описание	Краткое описание КПИ (список внесенных изменений, системные требования, требования к пользователям, список ПО для корректной работы и др.)	Да
Правило использования	Особенности описания КПИ (пожелания автора, способ распространения и др.)	Нет

Таблиця 3.4. **Формат специфікатоа в репозитории для QIP 2010**

QIP 2010

Дата	01:01:1970, 03:00:00
Версія	Build 4983
Платформа	x86 (x32)
Підтримка ОС	
Розмір	n/a
Автор	Аронов Андрій Олексійович
Опис	QIP 2010 является прямым преемником QIP Infium и QIP 2005. Сочетает в себе простоту интерфейса QIP 2005 и возможности QIP Infium. Сегодня QIP 2010 поддерживает XMPP (Jabber), Gtalk (Jabber), Вконтакте, LiveJournal (Jabber), Mail.Ru Agent, IRC (требует установки дополнительного модуля) и XIMSS (SIP), Twitter, Facebook, OSCAR. QIP 2010 работает под всеми версиями операционных систем семейства Microsoft Windows NT от Windows 2000 и выше.
Правила використання	QIP Mobile — это версия QIP для устройств с ОС Windows Mobile 5 и 6. Программа поддерживает все режимы экрана. Программа поддерживает два языка: английский и русский. Также можно загрузить другие языки с нашего форума. Пользователь может модифицировать внешний вид программы под себя, загрузив понравившийся ему скин или набор смайлов
Надрукувати	
Завантажити	

Сборка разноязычных программ в среде Eclipse

С помощью механизма плагинов подключаются средства для создания программ в разных ЯП и фреймворках. Плагин обеспечивает работу с CORBA–объектами и Eclipse CORBA plugin (ecp). Новый IDL-интерфейс создается CORBA –> IDL file, как стаб для компилятора OpenORB. Средствами External Tools Configuration создается новая конфигурацию сборки КПИ, которые указаны в операторе Link.

Конфигурирование КПИ

Конфигуратор собирает КПИ в специальный файл.

Он может изменять его структуру при внесении изменений в какой-нибудь КПИ по заданным точкам вариантности (рис. 3.18). Они выделяют позиции в описания КПИ или интерфейсе ПС, в которые могут вносится изменения. ИТК приведена программа решения квадратного уравнения с тремя точками вариантов средством Work Flow MS.Net.

Для этой программы строится соответствующий конфигурационный файл, который обеспечивает выполнение этой программы.

На рисунке справа изображена элементы модели для конфигурирования в среде VS.Net.

Пример выполняется за несколько шагов.

1. Запускаем ярлычок: WorkflowDesignerExample.exe – Открывается соответствующая программа.
2. Нажимаем на окно "открыть" и открываем файл:

D:\Проект-2011\Проект сайта\Колесник\Workflow2.xml

3. Нажимаем меню Workflow → Compile Workflow

Конфигуратор генерирует программу, пишет в библиотеку и запускает ее на выполнение.

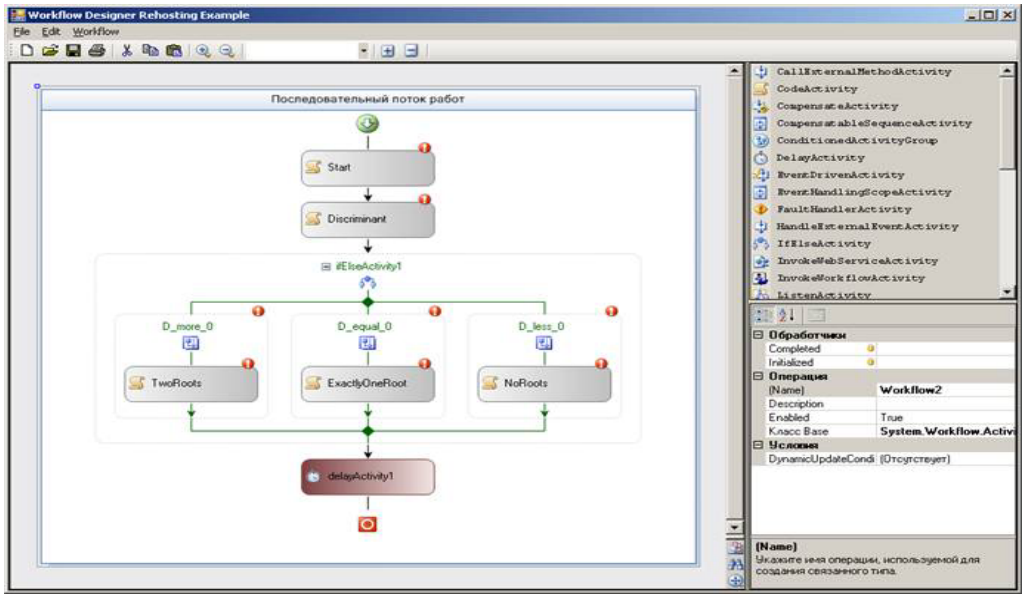


Рис. 3.18. Схема Work Flow для решения системы уравнения

Онтология доменов

Термин онтология, как метафизический, возник еще в 1613г. В Computer Science под онтологией понимается спецификация концептуальных знаний о некоторой предметной области, т.е. представляет описание множества объектов и связей между ними. Формально онтология содержит понятия, организованные в таксономию их описаний и правил вывода. Концептуальная модель онтологии $O = <X, R, F>$, где X – конечное множество понятий предметной области или домену; R – конечное множество отношений между понятиями; F – конечное множество функций интерпретации.

Операции онтологии:

1. Структура онтологии "Вычислительная геометрия" на сайте http://sestudy.edu-ua.net/ technologies /ontologies /descr.php?lang=ua.
2. Подача онтологии "Вычислительная геометрия" http://sestudy.edu-ua.net/ technologies /ontologies /example.php?lang=ua
3. Пример разработки онтологии в ИТК

Схема онтологии вычислительной геометрии имеет такой вид:

1. Классы задач
   а) Статические задачи.
   б) Динамические задачи.
   в) Задачи геометрического поиска .
2. Вариации задач.

Домен "Вычислительная геометрия" представлен в графическом виде средствами системы.

Protege поддерживает два основных способа моделирования онтологии посредством редакторов Protege-Frames и Protege-OWL. Онтология, построенная в

Protege, может быть экспортирована во множество форматов, включая RDF (RDF Schema), OWL и XML Schema. Protege имеет открытую, легко расширяемую архитектуру за счет поддержки модулей расширения функциональности.

Онтология ЖЦ 12207 и процесса тестирования

Для описания процесса тестирования использована онтологическая система Protege. В ней знания о модели процесса тестирования задается *классами, слотами, фасетами* и *аксиомами*. Подобную возможность предоставляют также и другие инструменты онтологии. Например, диаграммы классов в UML системы Rational Rose, которые могут отображаться в программный код на нескольких ЯП.

Трансформация ТД в ИТК

На сайте в разделе "Трансформация ТД" представлена апробация концепции преобразование типов данных ТД: вектор, очередь, стек, комплексное число в разделе сайта . Проводится работа по разработке и реализации генератора ТД. Приведена сущность трансформации избранных ТД, которая заключается в следующем:

Некоторый тип данных, например, вектор элементы которого int, необходимо перестроить к типу данных complex. Для этого разрабатывается сначала соответствующая примитивная функция, которая перестраивает эти типы данных с учетом форматов данных архитектуры компьютеров. Созданные функции (ниже приведен фрагмент описания этой функции в языке JAVA) тестируются в ИТК, а затем записываются с библиотеку примитивов GDT<=>FDT.

```
// create the zero vector of length N
public Vector(int N) {
this.N = N;
this.data = new double[N]; }
// create a vector from an array
public Vector(double[] data) {
N = data.length;
// defensive copy so that client can't alter our copy of data[]
this.data = new double[N];
for (int i = 0; i < N; i++)
this.data[i] = data[i]; }
public int length() {
return N; }
// return the inner product of this Vector a and b
public double dot(Vector that) {
if (this.N != that.N) throw new RuntimeException("Dimensions don't agree");
double sum = 0.0;
for (int i = 0; i < N; i++)
sum = sum + (this.data[i] * that.data[i]);
return sum;
}
```

Другие программы находятся на веб-сайте и выполняют свои функции.

6.4. Веб-сервисы в ИТК

Среда Eclipse поддерживает веб-сервис MyService после выполнения задания для обращения к веб-сервису.

Сайт выдает ответ в следующей форме (рис.3.19):

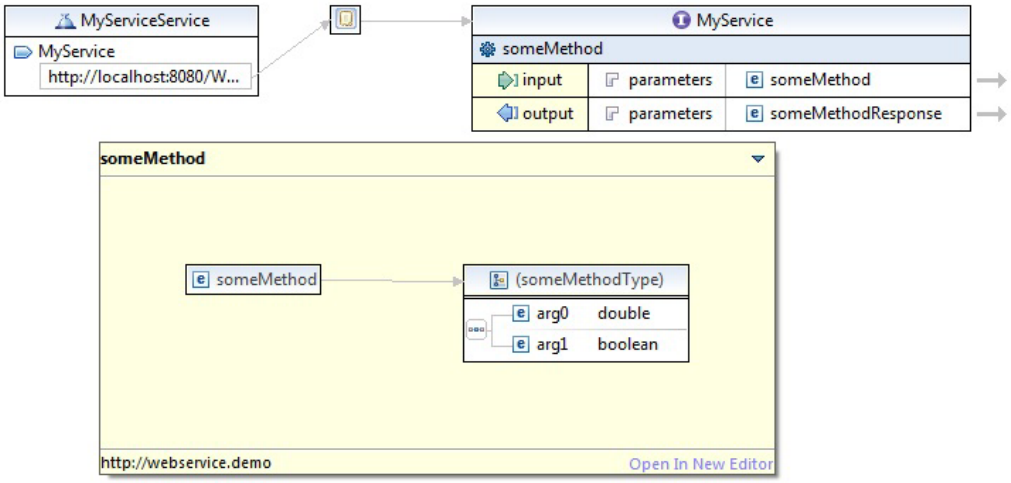


Рис. 3.19. Схема сервиса с помощью Eclipse

Вы сейчас находитесь в технологии "веб-сервисы"

Название файла webservices.zip Розмір:2243.88 Кб Дата модификации:Fri Nov 11 20:14:08 2011 Опис:

Программа с графическим интерфейсом использует простой веб-сервис на примере добавления и вычитания целых чисел. Для запуска распаковуется архив в любую директорию и запускается на выполнение файл WSLoader.jar. На машине должен быть установлен сервер программ GlassFish (его можно бесплатно загрузить на сайте Oracle). Для более детального описания программы *см. Инструкцию для выполнения.*

Загрузить.

Нажав на кнопку "загрузить", можно получить результат выполнения сервиса.

Содержание раздела веб-сайта

Описание принципов работы веб-сервисов http://sestudy.edu-ua.net/technologies/webservices /example.php?lang=ua.

Пример работы веб-сервисами http://sestudy.edu-ua.net/technologies/webservices/example_descr.php?lang=ua.

Инструкция по работе с примером http://sestudy.edu-ua.net/technologies/webservices /webservice.php?lang=ua.

Страница доступа к локальному веб-сервису.

"Вы сейчас находитесь в Технологии "Веб-сервисы ". Пример

Название файла: webservices.zip Размер: 2243.88 Кб. Дата модификации: Fri Nov 11 20:14:08 2011 Описание:

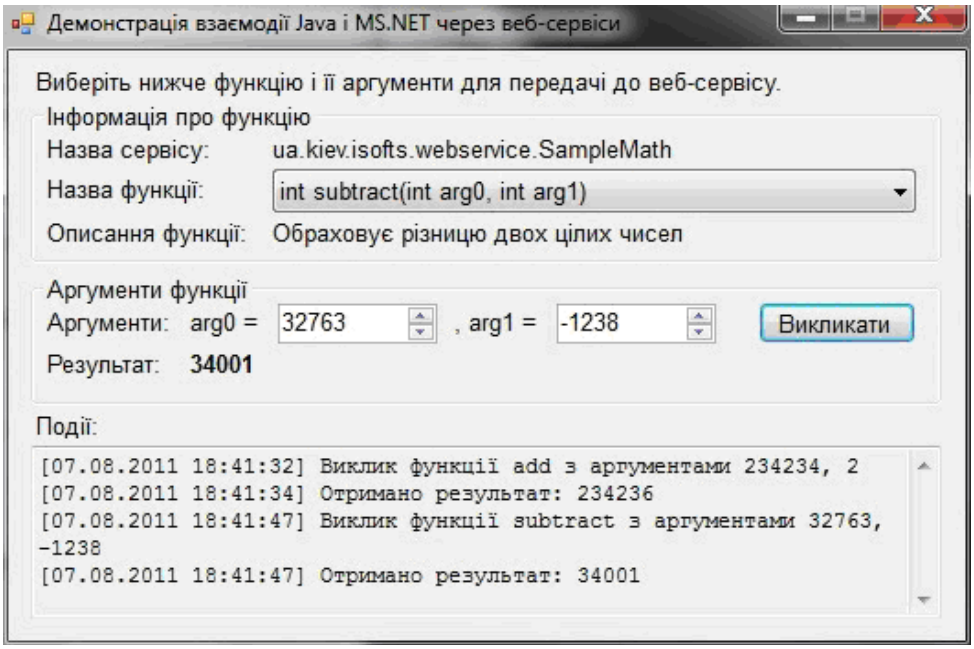


Рис. 3.20. Результаты счета

6.5. Раздел сайта "Взаимодействие"

Современный Интернет обеспечивает пользователя разными видами взаимодействия между распределенными системами, средами и их инструментами выполнения.

Программная поддержка взаимодействия программ, систем и сред разработана согласно теории взаимодействия [20, 23], сущность которой состоит в усовершенствовании общих методов и средств адаптации систем при переносе программ из одной программной среды в другую, используя механизмы репозитория Eclipse в ИТК.

Новые средства взаимодействия, реализованы на веб-сайте для решения проблемы миграции программ и систем из одной среды в другую через Eclipse, при создании "водораздела" между средами распределенных систем. прототипа. Эти средства реализованы, как механизмы взаимодействия систем между собой в ИТК, следующего вида:

1. Взаимодействие программ Visual Basic↔C++ выполняется через интерфейсный посредник, который передает данные от одной программы к другой и

при необходимости преобразовывает несовместимые типов данных. ЯП средствами динамической библиотеки (по методике И. Бея).

2. Взаимодействие систем и сред CORBA ↔ Eclipse платформ JAVA и MS.NET осуществляется брокером объектных заявок системы CORBA с использованием языка IDL для описания интерфейсов взаимосвязей этих систем.

3. Взаимодействие сред VS.Net↔Eclipse осуществляли на примере передачи данных программ, разработанных на платформе Visual Studio, в репозиторий Eclipse, используя механизм плагинов.

6.6. Разделы сайта: Презентации, Инструменты

Презентации. Данный раздел содержит по тематике программной инженерии три презентации:

1) системы автоматизации производственной деятельности отдела международных связей НАН Украины, она представлена и как самостоятельный веб-сайт в корпоративной среле Академии [116];

2) основные принципы создания фабрик программ, структуры, запаса КПИ и ГОР, технических программных и человеческих ресурсов, методов и средств средств поддержки процессов разработки на технологических линиях;

3) концепции и аспекты индустрии программ и систем, предложенных в упомянутом фундаментальном проекте.

Инструменты. В состав раздела входит описание средств разработки Eclipse для использования при агрегации инструментов ИТК за счет широких возможностей по расширению функциональности – механизм плагинов и методическое руководство проектирования моделей предметных областей и последующего их представления средствами нового предметно-ориентированного языка DSL средствами Protege.

В разделе дано описание технологии моделирования предметных областей или доменов средствами Protégé. В ИТК дана реализация (студенткой КНУ Т. Лихо) онтологической модели "Вычислительная геометрия" и онтология ЖЦ стандарта 12207 средствами DSL Tools VS.Net и DSL Eclipse.

6.7. Электронное обучение предмету "Программная инженерия"

Раздел "Обучение" в ИТК состоит из 3 линий следующего содержания:

1) электронное обучение современному языку – C# VS.Net,

2) обучение курсу ЯП JAVA по учебнику Хабибуллина (Санкт-Петербурга) со свободным доступом и включает последовательные действия по проведению процесса программирования, трансляции, выполнения на контрольных примерах автора.

3) обучение предмету "Программная инженерия" проводится по электронному учебнику Е.М.Лаврищевой (укр.) на сайте фабрики программ http://sestudy.edu-ua.net), а также на русском языке в Интернете (www.intuit.ru), занесенного в электронный университет России в 2007 году.

Студенты выходят на сайт фабрики http://programsfactory.univ.kiev.ua, открывают раздел сайта "Программная инженерия", переходят к учебнику и по его оглавлению выходят на соответствующий раздел учебника. Представленный в нем материал студенты изучают и отвечают на вопросы, которые поставлены в конце раздела. Они также могут задавать вопросы преподавателю и устно получать ответы.

Сайт этой фабрика разработан по методологии ТЛ с участием студентов кафедры ИС факультета кибернетики КНУ А.Аронова, А.Дзубенко, и И.Радецкого.

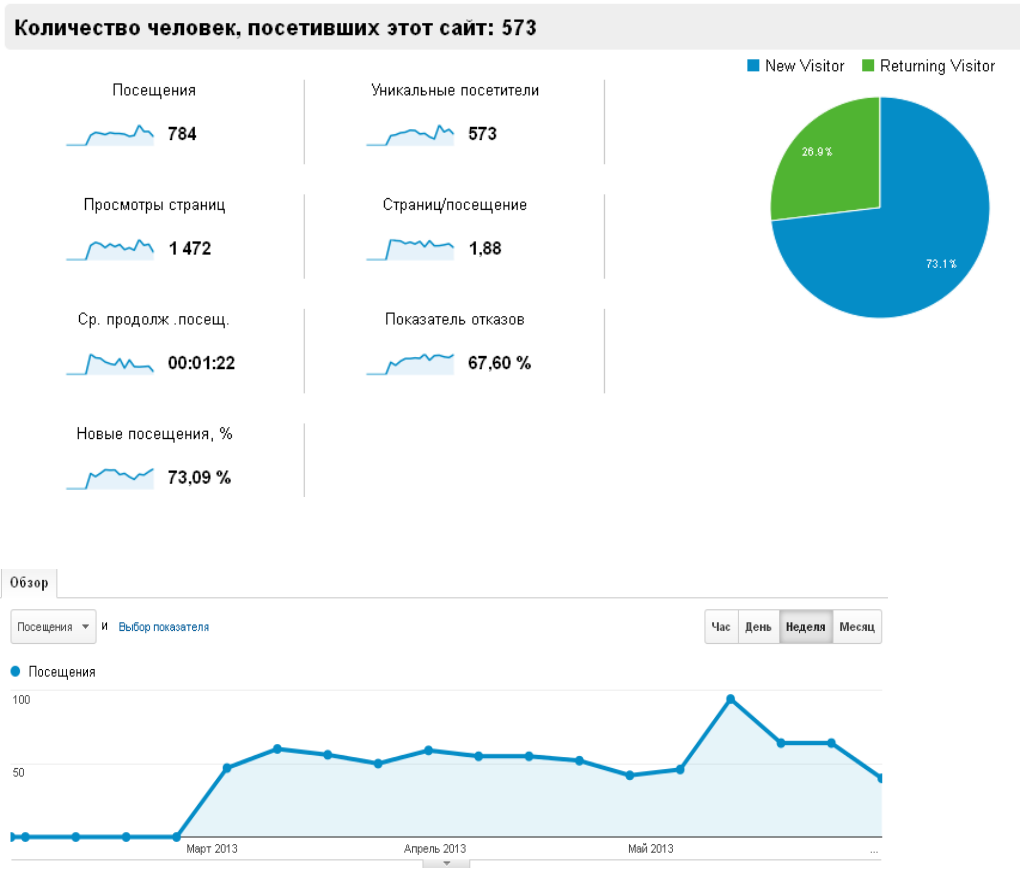
Фабрика поддерживает разработку КПИ и разных артефактов фундаментальных аспектов предметов обучения (математики, логики и др.), а также Computer Sciences, Software Engineering, Information Systems и Technologies. Эти объекты представляются сертифицированными КПИ и артефактами в стандарте и сохраняются в соответствующих репозиториях. Для выполнения этих работ предлагаются следующие ТЛ:

1. Линия проектирования программ в ЯП MS.NET (C#, Vbasic, JAVA)..
2. Оформление студенческих программ и артефактов, их тестирование.
3. Сертификация КПИ и артефактов для репозитория.
4. Конвейерная сборка (объединение нескольких готовых ресурсов в ПС).

Таким образом, сайт фабрики предназначен для поддержки технологии изготовления артефактов, которые можно использовать другим студентам, а также обучаться им разным технологиям, в том числе и таких направлений как математика, биология, физика и др. путем постепенного движения в плане создания малых компьютерных элементов, необходимых для применения новых приборов и механизмов.

Данный сайт входит в состав ИТК комплекса. Им и фабрикой пользуются многие специалисты из СНГ и зарубежных стран. Google статистика показывает (ниже один фрагмент) статистику специалистов, обращающихся сайту из разных стран.

Google Analytics (march-may 2013) on http://sestudy.edu-ua.net



В соответствии со статистикой самого сайта, к нему обратились более 15000 пользователей.

Глава 7. ПЕРСПЕКТИВА ПЕРЕХОДА ИТ-ТЕХНОЛОГИЙ К НАНОТЕХНОЛОГИЯМ

В. М. Глушков один из первых подхватил идею Норберта Винера (1948) "Кибернетика или управление, связь животных и машин". В это время уже появились первые ЭВМ. Термин употреблялся как управление живыми организмами, машинами и обществом с помощью информации, передаваемой между ними по каналам связи. Информация стала источником жизнедеятельности всех сфер жизни. Начиная с 80-х прошлого столетия кибернетика положила начало научной дисциплине информатики, которая изучает общие свойства, методы и системы создания, сохранения и использования информации во всех сферах жизни общества с помощью вычислительной техники и связей. Информатика фактически соответствует новой науки Computer Sciences.

С точки зрения этих наук началась эволюция научных идей, связанных с техникой разработки программных и информационных технологий и других форм концентрации знаний. В настоящее время мы видим развитие информатики от первых экспериментов создания информационных систем (ИС) до индустрии.

Кибернетика способствовала развитию теории автоматов, математической логики, теоретическое и прикладное программирование, теории компьютеров, информационных систем, баз данных и знаний и др.

Основным способом передачи знаний в производственной и управленческой сфере являются информационные технологий (ИТ). Они имеют свою перспективу развития в направлении нанотехнологий. Такие технологии реализуют научные идеи на потребности человека, включает метод (способ) и определенный порядок их применения в производстве разных видов продуктов (материальных и нематериальных). Элементы компьютерных ИТ–технологий включают ресурсы повторного использования (микро, макроэлементы, микротехника, материалы, данные и т.п.), а в нанотехнологии – это подобные микроресурсы, адаптивные специфики предметной области [124].

7.1. Оценка достижений компьютерных технологий

Начиная с 60-х годов при изготовлении первой ЭВМ сформировались технологии проектирования и изготовления *универсальных* ЭВМ. Они совершенствовалась в плане унификации *элементной базы и методов их сборки* в отдельные структурные компоненты ЭВМ. В процесс изготовления компьютерных систем создавалась технология автоматизированной сборки машин из малых отдельных элементов. И сегодня мы видим, что компьютеры разных вариантов, типов и размеров собираются массово по принципу *нанотехнологий* из элементов подобных атомам и молекул. Они стали настолько малыми, что их встраивают в мобильные телефоны, микроэлементы медицинских приборов и т. п.

По этому пути развивались и *информационные системы,* принципы которых изложены В. М. Глушкым в книге "Безбумажная информатика" (1982) [4]. В ее основу положены документы (не бумажные) управления государства, начиная от маленькой организации вплоть до электронного правительства. ИТ-*технологии* образуют базис компьютерной инфраструктуры современных корпораций, предприятий и государственных органов управления, на которых решаются различные задачи обработки информации локального и глобального характера. В них важное место занимали АСУ и АСУ ТП, конкретные из них реализованы на Лисичанском химкомбинате, Донецком горно-обогатительном комбината, Львовском телевизионном заводе и немецкого металлургического комбината и др.

Пути развития индустриальних технологий программирования

Правительство Украины уделяет большое внимание вопросу индустрии ПП. Агенство по информатизации провело международные научные конгрессы 17–18 ноября 2011, 25–27 октября 2012 и 28–29 октября 2013 по инфраструктуре электронного правительства и индустрии ПП. Их цель – создание высоких ИТ-

технологий и технологий производства ПП, открытие новых проектов по нанотехнологиям и улучшение системы подготовки ИТ-специалистов в ВНЗ Украины.

Каждая ИТ-технология по пути построения гибких и высоких технологий, приближающих нас к нанотехнологиям, когда элементы линии автоматически интегрируются в большие ИС.

7.2. На пути к нанотехнологии

Впервые идею синтеза атомов в макроатомы с помощью специального программного сборщика предложил Р.Фейнман (1959), как манипулятора атома, на который не действуют силы гравитации, а действуют межмолекулярные вандервальссовы силы. Он считал, что может быть произвольное число таких механизмов, представленных манипуляторами из элементов, уменьшенных в четыре и более раза копии "руки" оператора, который может закручивать маленькие болтики и гайки, сверлить очень маленькие отверстия, выполнять работы в масштабе 1:4,1:8, 1:16. К маленьким элементам относятся микроэлементы для хирургических макроприборов, микростимуляторов и т.п.

Р.Фейнман считал, что будут созданы миллионы миниатюрных заводиков, на которых "крошечные станки будут непрерывно сверлить отверстия, штамповать маленькие детальки" для маленьких приборов, собирать их в макромеханизмы, макровещества и т.п.

Эта идея привела к современной идеи миниатюризации и получения новых веществ из очень маленьких частиц других элементов со свойствами, необходимыми для конкретного применения. Такая маленькая частица названа нано или "карлик".

Нанотехнология – это технология производства, ориентированная на получение веществ и устройств с заранее заданной атомарной архитектурой (Э.Дрекслер).

Атом – это 10-10 =1 нанометра (нм), а бактерии это 10-9 нм.

Частицы от 1 до 100 нанометров называют наночастицами.

Наночастицы имеют одно свойство, слипание которых друг с другом приводит к образованию новых нанометратов (в медицине, керамике, металлургии и др.).

Один из важнейших вопросов, стоящих перед нанотехнологией — как заставить молекулы группироваться определённым способом, самоорганизовываться, чтобы в итоге получать новые материалы или устройства. Например, белки, которые могут образовывать комплексные структуры синтезом молекул белков ДНК с новыми специфическими свойствами.

Большие научные работы в области нанотехнологий ведутся в США, России, Франции и Украине. Вопросы нанотехнологии в компьютерных системх рассмотрены в работе В.П.Деркача [124] еще в период Советского союза. Он представил наработки в области элементной базы компьютерных систем, как наноэлементы, и показал пути их дальнейшего совершенствования и развития.

Рассмотрим современные тенденции направлений развития нанотехнологий.

Российская нанотехнология. Нанотехнология получения новых физико–химических свойств некоторых материалов для создания сверхчастотного инфракрасного материала в видимом диапазоне частот создана в Московском дворце творчества "Интеллект", в котором в течение нескольких лет изучались физико-химические особенности и свойства разных материалов. Этот новый материал предназначался для маскировки военной техники и инженерных сооружений от оптических и радиолокационных средств разведки на растительных, горных и пустынные участков земли. Полученный ими материал представляет собою волокна SiO_2. В материал внедрены ферримагнитные наночастицы, обеспечивающие коэффициент отражения в 15-80 раз больше по сравнению с металлической пластинкою в диапазоне частот А–37 ТЦ.

Эта нанотехнология была выставлена в Украине как передвижной учебный класс "Нанотехнологии и материалы" (www.intelltct-cit.ru) в районе ВДНХ летом 2013 недалеко от зданий факультетов кибернетики и радиофизики Киевского университета имени Тараса Шевченко.

На этой выставке был представлен системный комплекс, включающий:

1) сканирующий туннельный микроскоп "Умка" для изучения поверхности материалов с атомарной разрешающей острой иглой, скользящей по исследуемому материалу;
2) устройство заточки иглы;
3) смеситель материалов получения сверхчастотного материала;
4) образцы материалов (астробетон-ЛБ, радиопоглощающий материал РПМ, гидрофобные покрытия);
5) описание процесса получения наночастиц золота размером 15-20 нм. и др.

Опыты и лабораторные работы проводились под научным руководством специалистов МГУ, институтов стали и сплавов, химико-технологического, электронной техники, концерна наноиндустрии и др.

Украинская нанотехнология. Информация о нанотехнологии в украинской науке содержалась в статье газеты "Зеркале недели" №35(132) от 28.09.2013 под названием "Нанотехнологии в Украине – вдогонку за уходящим поездом". В ней представлен материал о создании электронно-лучевых веществ в Институте электросварки НАНУ им. Е.О.Патона. Эта работа проводилась с 70-х годов прошлого века. К тому времени физики многих стран методом испарения в вакууме получили тонкие нанометаллические пленки. В процессе сварки металлических материалов разных толщин использовался электронный луч для наноконцентрированного их нагрева в вакууме. Сварка в космосе выполнялась новыми материалами для покрытия металлических аэрокосмических конструкций по методу электронно-лучевой плавки и конденсации веществ в вакууме. Был создан новый технологический прием электронно-лучевой сварки, который был запатентован в ряде передовых стран в 1984 году под названием (EB-PVD).

Однако в статье отмечалось,, что нанонаправление почти не развивается и "научный поезд нанотехнологии уходит" со сцены. В медицине создаются новые приборы и лекарственные препараты. Однако торможение наноисследований связано с отсутствием средств, материалов и специалистов, способных создавать новые нанотехнологии в биологии, химии, генетике и др.

После изучения этих направлений нанотехнологий у автора возникла идея исследования свойств и особенностей высокоэффективных компьютерных, ИТ–технологий в плане создания разных мини и микроэлементов и наноэлементов применительно в областях e-sciense (биологии, химии, физики, медицины, генетики и др.).

ИТ–технологии достигли высокого развития, проникли во все сферы обслуживания мирового сообщества. Появившиеся Skype-технологии позволили визуально общаться людям и видеть друг друга, находясь в разных точках планеты. Требуется разработать такие технологии, чтобы "поезд нано постепенно набирал обороты" в ближайшее десятилетие. Такие технологии, нанотехнологии, которые будут способствовать поддержке жизни на земле, улучшать качество жизни и здоровье людей, а также исследованиям недр земли, океана и атмосферы, чтобы создать жизненно-важные новые вещества из природных и наукоемких частиц [124, 125].

Приоритетные направления нанотехнологий. В России научными институтами сформулированы на ближайшую перспективу 16 приоритетных научно-технических задач в биологии, медицине, генетике, энергетики, металлургии, компьютерной и сетевой проблематике, в изучении природных и космических явлений и др. Решение каждой задачи определяется необходимостью создания новых высокоэффективных технологий и нанотехнологий.

Исходя из приведенной статьи нанотехнологии стало ясно, что подготовлен проект закона на уровне правительства Украины "О государственном стимулировании отечественной наноиндустрии" для утверждения на Верховном Совете в 2014 г.

В перспективе готовые программные элементы будут развиваться в направлении нанотехнологий компьютеров путем "уменьшения" составных элементов к виду маленьких частиц с заранее заданной структурой и функциональностью. Стандартизация этих элементов и автоматизация связи, синтеза маленьких частиц в большие сложные. Это предполагается для производства новых продуктов в e-science (биология, медицина, генетика, физика, и др.), которые будут способствовать улучшению здоровья общества и улучшения жизни на земле.

В перспективе предстоит создать маленькие технические элементы в виде "чипов" и транзисторов и др. для разных предметных областей e-sciences. Они будут накапливаться в запас и средствами компьютерной нанотехнологии из них можно собирать новые продукты как конвейере.

В ближайшее десятилетие современные технологии будут развиваться в направлении новых нанотехнологий во многих областях науки и техники (биологии, химии, физике, медицине, космос и др.).

ЗАКЛЮЧЕНИЕ

В настоящее время отделом программной инженерии были получены новые теоретические результаты в парадигмах програмирования, которые детально описаны в данной работе. Эти научные результаты внесли существенный вклад в технологию программирования и программной инженерии. Кроме того, в отделе созданы ряд CASE:

1) система *взаимодействия* разнородных программ и систем между собой с возможностью их переноса в другую операционную среду для работы с данными, которые передаются через интерфейсы или выбираются из баз данных, онлайн хранилищ данных в Cloud Computing? Azure, Big-data и др.;

2) *технология реализации КПИ* и их описание в стандарте WSDL, их интерфейсных данных, которые накапливаются в репозитории КПИ и интерфейсов для применения при поиске и отборе готовых КПИ пользователями;

3) *сборка разноязычных программ* из готовых КПИ репозитория, имеющих паспортные данные, необходимые для объединения и трансформации несовместимых типов данных КПИ, которые передаются между ними и могут располагаться на разных платформах сред выполнения ПС;

4) *ОКМ метод* логико-математичекого моделирования предметных областей из объектов и компонентов, как элементов для построения из них сложных ПС путем конфигурации разнородных ресурсов;

5) *онтологичесое* представление доменов ЖЦ стандарта ISO/IEC 12207 и ISO/IEC 11404 GDT, вычислительная геометрия средствами DSL Tools VS.Net Eclipse-DSL и Protégé в среде ИТК;

6) набор примитивных функций преобразования новых типов данных (таблиц, массивов, последовательностей и др.) GDT<=>FDT стандарта ISO/IEC 11404 – общие типы данных;

7) *конфигуратор* КПИ средствами новой модели вариантности программ и ПС как членов семейства СПС;

6) *средства обучения* с помощью линий разработки программ в ЯП C#, JAVA, Basic, C++ в среде VS.Net, CORBA и Eclipse, а также электронного курса "Программная инженерия" по соответствующему учебнику автора;

7) *реализация* стандарта ISO/IEC 12207 ЖЦ программных систем на примере процесса тестирования.

Эти средства обеспечивают формальное представление программных элементов парадигм (модульной, объектной, компонентной, сервисной), их реализацию и сохранение в репозитории комплекса ИТК в стандартном виде, а также сборку этих элементов в более сложные программные структуры, способные адаптироваться в фабрики AppFab современных сред типа IBM, VS/Net и др.

В ближайшей перспективе предложенные и реализованные средства сборочного стиля программирования будут развиваться по пути инженерии к виду наноэлементов в таких областях науки – биология, химия, физика, генетика, медицина и др. Нанотехнологии займут ведущее место в информационном мировом сообществе.

СПИСОК ЛИТЕРАТУРЫ

1. *Капитонова Ю.В., Летичевский А.А.* Парадигмы и идеи академика В.М. Глушкова.– Киев, Наук. Думка.– 2003.–355 С.

2. *Глушков В.М.* Кибернетика, ВТ, информатика (АСУ).–Избран. труды в 3-х томах. –К.: Наук. думка, 1990, 262 С, 267 С., 281 С.

3. *Глушков В.М.* Основы безбумажной информатики.– М.: Наука, 1982. – 552 С.

4. *Лаврищева Е.М* .Проблематика программной инженерии, Изд."Знание", РДНТП, 1991. – 19 С.

5. *Лаврищева Е.М.* Методы и средства сборочного программирования, Тез. докл. междунар. конф. "Технология программирования 90-х", Киев, 1991. – с.30 – 32.

6. *Лаврищева Е.М., Грищенко В.Н.* Сборочное программирование. – Киев: Наук.думка, 1991. – 213 с.

7. *Лаврищева Е.М., Грищенко В.Н.* Сборочное программирование. Основы индустрии программных продуктов. – К.: Наук. думка, 2009. – 372 с.

8. *Лаврищева Е.М.* Основы технологической разработки прикладных программ СОД // Киев, 1987. – 29 с. – (Препр./Ин-т кибернетики им. В. М. Глушкова; 87 – 5).

9. *Лаврищева Е.М.* Технологическая подготовка и программная инженерия // УСиМ, №1, 1988. – с.35 – 43.

10. *Лаврищева Е.М.* Модель процесса разработки ПО // УСиМ, №5, 1988. – с.53 – 60.

11. *Лаврищева Е.М.* Программная инженерия. Основные понятия и определения. Сб. "Методы и средства программной инженерии", РИО ИК АН УССР, Киев, 1989. – с.30 – 32.

12. *Боем Б.У*. Инженерное проектирование программного обеспечения. – М.: Радио и связь, 1986. – 510 С.

13. *Липаев В.В.* Технология проектирования комплексов программ. –М.: Радио и связь, 1983. – 235 С.

14. *Липаев В.*В. Оценка затрат на разработку программных средств. – М.: Финансы и статистика, 1988. – 225 С.

15. *Кулаков А.*Ф. Оценка качества программ ЭВМ. – К.: Техника, 1984, – 166 С.

16. Лаврищева Е.М. Методы программирования. Теория, инженерия, практика. Наук. Думка, 2006.– 451 С. (www.twirpx.com)

17. *Лаврищева Е.М., Петрухин В.А.* Методы и средства инженерии программного обеспечения. – М.: МОН РФ, 2007. – 415 с.– Aviable (www.twirpx.com, www.intuit.ru).

18. *Тыугу Э.Х.* Концептуальное программирование. – М.: Наука, 1984, 256 с.

19. *Моренцов Е.И*. Реализация варианта CASE-системы на базе СУБД "Пальма". – Сб. Программная инженерия. – Киев1993. – с.15 – 19.

20. *Лаврищева Е.М., Хоролец Д.С.* Комплекс АПФОРС – средство автоматизированного проектирования ППП // Сб. Средства информационного обеспечения. – ИК АН УССР, 1984. – с.67 – 74.

21. *Лаврищева К.М.* Розвиток идей академіка В.М.Глушкова з питань технологии програмування.–Київ, Вісник НАН України, 2013, №9.– с. 66–83.

22. *Лаврищева Е.М., Л.Г.Борисенко, Л.Г*. Усенко и др. Транслятор "Д –АЛГАМС" для УВК "Днепр –2".– Киев: Ин – т кибернетики АН УССР, 1971.– 246 с.

23. *E. M. Lavrishcheva and E. L. Yushchenko*. A method of analyzing programs based on a machine language, 1972, Springer, Volume 8, Number 2, Pages 219–223.

24. *Глушков В.М., Лаврищева Е.М., Стогний А.А.* и др. Система автоматизации производства программ – АПРОП, Киев, ИК АН УССР, 1976.–136с.

25. *Андон Ф.И., Лаврищева Е.М., Моренцов Е.И.* Технология и средства автоматизированного проектирования и реализации высококачественного ПП для систем обработки данных, Сб."Инструментальные средства программирования", Киев.– РИО Ин–та Кибернетики им. В.М.Глушкова АН Украины, 1993.–с.3–12

26. *Лаврищева Е.М., Вишня А.Т., Грищенко В.Н.* и др. Система автоматизации производства программ с режимом мультидоступа (АПРОП–2).–ЕрНУЦ, Ереван, 1981.– №П005508, ЯЩ 15001–33–01, 09.12.81. –587с.

27. -*Лаврищева Е.М*. Объектно-ориентированое проектирование в отечественной CASE–системе, Тез.докл. 2 – междунар. конф."Технология программирования 90-х", Киев, 1992. – с.63 – 74

28. *Глушков В.М.* Фундаментальные основы и технология программирования // Программирование. – 1980. – № 2. – С. 13–24.

29. *Глушков В.М., Капитонова Ю.В., Летичевский А.А.* О применении метода формализованных технических заданий к проектированию программ обработки структур данных // Программирование. – 1978. – № 6. – С. 5–12.

30. *Вельбицкий И.В., Ходаковский В.Н., Шолмов Л.И.* Технологический комплекс автоматизации программ на машинах ЕС ЭВМ и БЭСМ-6.– М.: Статистика, 1980.– 263 с.

31. *E. M. Lavrishcheva.* Modular design of large programs,, 1980, Springer. – Volume 16, Number 2, Pages 244 – 249.

32. *Коваль Г.И., Коротун Т.М., Лаврищева Е.М.* Об одном подходе к решению проблемы межмодульного и технологического интерфейсов // Межотраслевой сборник АН СССР и Минвуза СССР , 1986. – с.38 – 49.

33. Ершов А.П. Научные основы доказательного программирования.– Научное сообщение в президиуме АН СССР, Наука, 1984.–с.1–11.

34. Задорожная Н.Т. Управляемое проектирование документооборота в управленческих информационных системах. – Автореферат диссертации. – ИК НАН Украины, 2004. – 23 С.

35. *Задорожная Н.Т., Лавріщева К.М*. Менеджмент документооборота в информационных системах образования. – К.: Педагог. думка, 2007. – 220 с (http://lib.iitta.gov.ua/view/creators)

36. *Лавріщева К. М.* Взаємодія програм, систем й операційних середовищ // Проблеми програмування, №3, 2011. – с. 13–24.

37. *Лаврищева Е. М.* Концепція індустрії наукового софтвера і підхід до обчислення наукових задач // Проблеми програмування, №1, 2011. – с.3–17.

38. *Анісімов А. В., Лавріщева К. М., Шевченко В. П.* Про індустрію наукового софтвера. – Conf. Theoretical and Applied Aspects of Cybernetics, Kiev, Febrary, 2011, Ukraine.

39. Лаврищева К.М. Програмна інженерія.–К.: Академперіодика, 2009, 371 С. (www.programsfactory.kiev.ua)

40. *E. M. Lavrishcheva*. Software engineering as a scientific and engineering discipline 2008, Springer, Volume 44, Number 3, Pages 324 – 332.

41. *Лавріщева К.М.* Програмна інженерія – напрями розвитку. – Праці міжнар. конференції "50 років Інституту кібернетики імені В.М.Глушкова НАН України", К.: 2008. – с.336 – 345.

42. *Лаврищева Е.М.* Классификация дисциплин программной инженерии // Кибернетика и системный анализ.–2008.– №6.–С.3–9.

43. *Коваль Г. І.*, *Колесник А. Л., Лавріщева К. М., Слабоспицька О. О.* Удосконалення процесу розроблення сімейств ПС елементами гнучких методологій // Проблеми програмування (Спецвипуск конференції УкрПРОГ–2010). – 2010. – № 2–3. – С. 261–270.

44. *Андон П. І., Лавріщева К. М.* Розвиток фабрик програм в інформаційному світі // Вісник НАН України. – 2010. – №10. – С. 15–41.

45. *Corbin J.* The art of Distributed Application. Programming Techn. For Remote Procedure Calls.- Berlin: Springer Verlag, 1992. – 305 p.

46. *Эммерих В*. Конструирование распределенных объектов. Методы и средства программирования интероперабельных объектов в архитектурах OMG/CORBA, Microsoft COM и JAVA RMI. – М.: Мир, 2002. – 510с.

47. *Бей И.* Взаимодействие разноязыковых программ.–Диалектика.–М.:С-Петербург–Киев.–Изд. дом. "Вильямс".– 2005.–868с.

48. *Гринфильд Дж.* Фабрики разработки программ.–Диалектика.–М–С–Перебург–Киев.–Изд.дом. "Вильямс".– 2007.–591с.

49. *Guckenheimer S.*, Perez J.I. Software Engineering with Microsoft Studio Team System.–Crawfordsville, USA: Adison–Wesley, 2006.–304 P.

50. *Meglio A., Bégin M.E., Couvares P., Ronchieri E., Takacs E*. ETICS: the International Software Engineering Service for the Grid – Jornal of Physics Conference Series 119. – 2008. – с.58 – 67

51. *Чернецки К., Айзенекер У.* Порождающее программирование. Методы, инструменты, применение.– Издательский дом Питер. – М. – СПб. – Харьков. – Минск. – 2005. – 730 с.

52. *G.Lenz, Wienands C.* Pactical software fabrics in Net. From theory in practice – a primer referense and case study. – Apress, 2007. – 205 p.

53. *Авдошин С.М*. Фабрики программного обеспечения. – ttp://www/ewwek.com./on

54. Duval P., Matyas, Grover A. Continuous integration Împroving Software Quality and Reducing Risk, Addison Wesley, 2009. – 691 p.

55. *Андон П. І., Лавріщева К. М.* Методологія побудови ліній виробництва програмних продуктів і їх застосування // Інформаційне суспільство в Україні: Матер. міжнар. наук. конгр. (25–26 жовтня 2012, Київ, Україна). – С. 19–26.

56. *Лаврищева Е.М.* Сборочній конвейер фабрик программ – идея академика В.М.Глушкова, "В.М.Глушков. Прошлое устремленное в будущее".–Академпериодика, 2013. – с.130 – 138.

57. *Лаврищева Е.М.*. От технологии программирования к компьютерным нанотехнологиям программ и систем. – Междунар. науковий конгрес "Інформаційне суспільство в Україні – 2013". – с.56 – 60,

58. *Аронов А. О. Дзюбенко А. І.* Підхід до створення студентської фабрики программ // Проблеми програмування, № 3, 2011. – с. 42–49.

59. *Lavrischeva E., Dzubenko A., Aronov A*. Conception of Programs factory for Representation and E-learning Disciplines of Software Engineering / 9-th International Conf. ICTERI–2013 "ICT in Education, Research and Industrial Applications; Integration and Knowledge Transfer", Ukraine, June 17–21, 2013, http://ceur-ws.org/Vol-1000//

60. *Lavrischeva E., Aronov A., Dzubenko A.* Programs factory – a conception of Knowledge Representation of Scientifical Standpoint of Software Engineering. Jornal of Computer Science, Canadian Senter of Science and Education, ISSN1913-8989, 2013, p.21 – 27.

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.* Вопросы комплексирования программных средств // УСиМ.–1987.—№ 1.—С. 68—71.

64. *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.

65. *Лаврищева Е.М.* Модульный принцип конструирования больших программ // Там же.—С. 12—20.

66. *Лаврищева Е.М.* Вопросы объединения разноязыковых модулей в ОС ЕС// Программирование.—1978.—№ 1.—С. 22—27.

67. *Лаврищева Е.М.* Парадигма интеграции в программной инженерии //Проблемы программирования.–2000.–№1–2.–С.351 – 360.

68. *Лаврищева Е.М.* Об автоматизированном изготовлении программных агрегатов из равноязыковых модулей//УСиМ.–1979.—№ 5.–С. 54—60.

69. *Лаврищева Е.М.* Методика изготовления программных агрегатов // Кибернетика.–1980.–№2.–С. 77–82.

70. *Леман Д., Смит М.* Типы данных // Данные в языках программирования.– М.: Мир, 1982.–С. 196–213.

71. *Вирт Н.*Алгоритм+структуры данных=программы: Пер. с англ.–М.: Мир, 1985.– 406 с.

72. *Агафонов В.Н.* Спецификация программ: понятийные средства и их организация. –Новосибирск : Наука, 1987.–290 с.

73. *Замулин А.В., Скопин И.Н.* Конструирование базы данных на основе концепции абстрактных типов данных // Программирование.– 1981.– № 5.– С. 38–43.

74. *Замулин А.В.* Типы данных в языках программирования и базах данных.– М.: Наука, 1987.– 152 с.

75. *Danahue P.* On the semantics of data types // SIAM J. Comput.– 1979.– 8, N 4.– P. 546–560.

76. *Лавріщева К., Стеняшин А.* Підхід щодо трансформації загальних типів даних стандарту ISO/IEC 11404 для використання в гетерогенних середовищах//2-nd International Conference on High Performance Computing, October 8– 10, 2012, Kyiv.– Ukraine.– с.227– 234.

77. *Стеняшин А.Ю.* Про формальний опис типів і структур даних в різнорідних програмах.– Проблеми програмування, №2, 2011.– с.50–61.

78. *Буч. Г.* Объектно-ориентированный анализ .–М.:"Бином", 1998. – 560 с.

79. *Буч Г., Рамбо Д., Джекобсон А.* Язык UML. Руководство пользователя: Пер. с англ. – М.: ДМК, 2000, – 432 с.

80. *Грищенко В.Н.* Теоретические и прикладные средства комппонетного программирования. – Автореаф. докт. диссертации. – ИК имени В.М.Глушкова, 2007. – 34 С.

81. *Грищенко В.Н.* Подход к формализации объектно-ориентированной методологии // Проблемы программирования, – 1997. – №1. – С.33 – 39.

82. *Грищенко В.М.* Підхід до аналізу поведінки об'єктних систем при моделюванні предметних областей // Проблемы программирования. – 1998. – №4. – С. 67 – 75.

83. *Грищенко В.М.* Метод об'єктно-компонентного проектування програмних систем // Проблеми програмування. – 2007. – №2. – С.113–125.

84. *Андон Ф. И., Лаврищева Е. М.* Методы инженерии распределенных компьютерных приложений. – К.: Наук. думка, 1997. – 229 с.

85. *Лавріщева К.М., Колесник А.Л., Стеняшин А.Ю.* Об'єктне-компонентне проектування програмних систем. Теоретичні і прикладні питання – Вісник КГУ, серія фіз.-мат.наук. – 2013. – №4. – С. 103 – 117.

86. *Бабенко Л. П., Лавріщева К. М.* Основи програмної інженерії. – К.: Знання, 2001. – 269 с.

87. *Лавріщева К.М.* Компонентне програмування. Теорія і практика // Проблеми програм..2012, №4. – с.3 – 12 (www.isofts.kiev.ua).

88. *Лавріщева К.М., Колесник А.Л.* Концептуальні моделі розподілених компонентних систем. –Ж. Проблеми програмування. – 2013, №2. – с13 – 22.

89. *Колесник А.Л.* Модели и методы разработки семейства вариантных программных систем. – Автореф. – КНУ, 2013. – 22 С.

90. *Морган М. JAVA2.* Руководство разработчика: Пер.с англ. – М.: : Издательский дом "Вильямс", 2000. – 720 с.

91. Лавріщева К.М., Коваль Г.І., Бабенко Л.П. і др. Нові теоретичні засади технології виробництва сімейств ПС у контексті ГП.– Електронна монографія.–ДНТІ України, ВИНИТИ Росії та ДНТБ, 2012.–277 С. Available at http://www.nbuv.gov.ua.

92. *Лавріщева К.М.* Генерувальне програмування ПС і сімейств // Проблеми програмування. – 2009.– № 1.– С. 3 – 16 (www.isofts.kiev.ua)

93. *Лавріщева К.М., Слабоспицька О.О., Коваль Г.І., Колесник А.О.* Теоретичні аспекти керування варіабельністю в сімействах програмних систем. – Вісник КГУ, серія фіз.–мат.наук. – 2011. – №1. – С. 151–158.

94. *Коваль Г. І.*, *Колесник А. Л., Лавріщева К. М., Слабоспицька О. О.* Удосконалення процесу розроблення сімейств програмних систем елементами гнучких методологій // Проблеми програмування (Спецвипуск конференції УкрПРОГ–2010). – 2010. – № 2–3. – С. 261–270.

95. *www.aspectjs.com,* http://casearjs.org/

96. *WeSevice.*SoftwareFactory,http://msdn2.microsoft.com/enus/library/aa480534.aspx

97. *Островський А. И.* Подход к обеспечению взаимодействия программных сред JAVA и Ms.Net. – Проблеми програмування, 2011.–№ 2.–с. 37–44.

98. *Радецький І. О.* Один з підходів до забезпечення взаємодії середовищ MS.Net і Eclipse // Проблеми програмування, № 2, 2011.– с. 45–52.

99. *www.edu.cbsystematics.com,* hppp://127.0.0.1^400/icontract

100. *Лавріщева К. М.* Онтологічне подання ЖЦ ПС для загальної лінії виробництва програмних продуктів. – Праці конференції TAAPSD'2013, "Теоретичні і приклані аспекти побудови програмних систем", 25 травня – 2 червня 2013. – с. 81–90.

101. *Лавріщева К.М.* Підхід до формального подання онтології життєвого циклу програмних систем. – Вісник КГУ, серія фіз.-мат.наук. – 2013. – №4. – С. 94 – 102.

100. *Бабенко Л.П.* Онтологический подход к спецификации свойств программных систем и их компонент//Кибернетика и системный анализ.–2009.–№1.–с.30–37.

101. *Зінькович В.М.* Онтологічне моделювання предметної області з проблематикою e–science' // Проблеми програмування. – 2011 – № 3. – С. 91–99

102. *Островський А. И.* Подход к обеспечению взаимодействия программных сред JAVA и Ms.Net. – Проблеми програмування, 2011.–№ 2.–с. 37–44.

103. *Радецький І. О.* Один з підходів до забезпечення взаємодії середовищ MS.Net і Eclipse // Проблеми програмування, № 2, 2011.– с. 45–52.

104. *Основы* инженерии качества программных систем / Ф.И.Андон, Г.И.Коваль, Т.М. Коротун, Е.М.Лаврищева, В.Ю. Суслов // 2–е изд. – К.: Академпериодика. – 2007. – 672 с.

105. *Лаврищева Е.М.* Становление и развитие модульно-компонентной инженерии программирования в Украине.–Препринт 2008–1.–Институт кибернетики им. В.М. Глушкова, 2008.–33 с.

106. Коротун Т.М. Модели и методы инженерии тестирования программных систем в условиях ограниченных ресурсів, – Автореф. – ИК НАНУ, 2004. – 21 С.

107. Коваль Г.И. Модели и методы инженерии качества программных систем на ранних стадиях жизненного цикла. – Автореф. – ИК НАНУ, 2005. – 20 С.

108. Слабоспицкая О.А. Модели и методы экспертного оценивания в жизненном цикле программных систем.. – Автореф. – ИК НАНУ, 2008. – 19 С.

109. Колесник А.Л. Модели и методы разработки семейства вариантных программных систем. – Автореф. – КНУ, 2013. – 22 С.

110. *Коваль Г.І.* Байєсівські мережі як засіб оцінювання та прогнозування якості програмного забезпечення // Проблеми програмування. – 2005. – № 2. – С. 15–23.

111. *Коваль Г.І.* Підхід до моделювання якості сімейств програмних систем // Проблеми програмування. – 2009. – С. 49–58.

112. *Лаврищева Е.М.*, *Слабоспицкая О.А.* Подход к экспертному оцениванию в программной инженерии // Кибернетика и системный анализ. – 2009. – № 4 С.151–168.

113. *Слабоспицкая О.А.* Формальный аппарат экспертного решения проблемы многокритериального оценивания при учёте ряда точек зрения на проблему // Проблемы программирования".– 2002, № 1– 2. –С. 430– 440.

114. *Коротун Т.М.* Моделі і методи тестування програмних систем – Проблеми програмування. – 2007. – № 2. –С. 76–84.

115. *Лаврищева Е.М., Зинькович В.М., Колесник А.Л.* и др. Инструментально–технологический комплекс разработки и обучения приемам производства программных систем, (укр.).– Гос. служба интеллектуальной собственности Украины.– Свид. о регистрации авторского права.– № 45292, от 27.08.2012.–103 с.

116. *Грищенко В. М., Куцаченко Л. І.* Автоматизова інформаційна система підтримки міжнародної діяльності НАНУ. – Державний департамент інтелектуальної власності. – Свідотство № 32304 від 23.12.2009.

117. *Лавріщева К. М.* Інструментально-технологічний комплекс для розробки й навчання прийомам виробництва програмних систем // Вісн. НАН України. – 2012. – № 3. – С. 17–26.

118. *Lavrischeva E., Ostrovski A.* General Disciplines and Tools for E–Learning Software Engineering. – http://senldogo0039.springer–sbm.com/ocs/.

119. *Аронов А. О., Дзюбенко А. І.* Підхід до створення студентської фабрики програм // Проблеми програмування. – 2011. – № 3. – С. 42–49.

120. *Ekaterina Lavrischeva1, Alexei Ostrovski.* New Theoretical Aspects of Software Engineering for Development Applications and E-Learning, *Journal of Software Engineering and Applications*, 2013, 6, 34-40 http://dx.doi.org/10.4236/jsea.2013.69A004 Published Online September 2013 (http://www.scirp.org/journal/jsea).

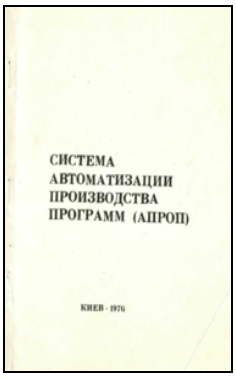
121. *Lavrischeva E. and A.Ostrovski*. General Disciplines and Tools for E-learning Software Engineering 8– th International Conf. ICTERI– 2013 "ICT in Education, Research and Industrial Applications", Ukraine, June, 2012, Springer.com, Communication in Computer and Information Sciences, ISSN 1865-0929, p.212 – 229.

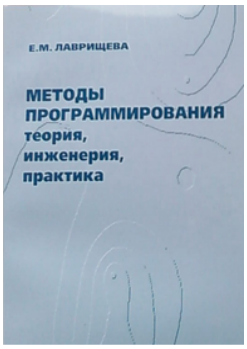
122. Лаврищева К.М. Розвиток идей академіка В.М.Глушкова з питань технологии програмування. – Київ, Вісник НАН України, 2013, №9. – с. 66 – 83.123. Андон П.І., Лавріщева К.М. Наукові і прикладні підходи до розвитку індустрії програмної продукції.– Теза.– Матер.праць "Міжнародного наукового конгресу з розвитку інформаційно-комунікаційних технологій та розбудови інформаційного суспільства в Україні при Кабміні Україні (17–18 листопада 2011 р.).– Київ.– с.6–8.

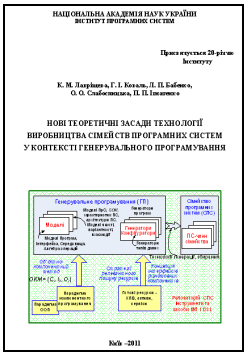
124. Деркач В.П. От электронных ламп до молекулярных схем и нанотехнологий. – Международный научный журнал "Наука ы науковедение", 4(58)б 2007. – с58 – 68.

125. *Лаврищева Е.М.* От технологии программирования к компьютерным нанотехнологиям программ и систем. – Международний науковий конгрес "Інформаційне суспільство в Україні – 2013". – с.56 – 60.

Обложки основных публикаций







Наукове видання

ЛАВРІЩЕВА Катерина Михайлівна

SOFTWARE ENGINEERING КОМП'ЮТЕРНИХ СИСТЕМ

ПАРАДИГМИ, ТЕХНОЛОГІЇ, CASE-СРЕДСТВА ПРОГРАМУВАННЯ

Російською мовою

Художний редактор І. Р. Сільман
Коректор М. К. Пуніна

Підп. до друку 24.01.2014 р. Формат 70x100$^{1/16}$
Папір офс. №1. Друк. різограф. Обл.–вид. арк. 30,72
Умв. друк. арк. 26. Тираж 300 прим. Замовлення № 2282.

Друкарня Видавничого дому "Академперіодика" НАН України
01004, Київ–4, вул. Терещенківська, 4
Свідоцтво про внесення суб'єкта видавничої справи до Державаного реєстру
Серія ДК № 3179 від 08.05.2008 р.

ЛАВРИЩЕВА
ЕКАТЕРИНА МИХАЙЛОВНА

доктор физико-математических наук (1989), заслуженный деятель науки и техники Украины (2008), профессор кафедры теории и технологии программирования Киевского национального университета (КНУ) имени Тараса Шевченко и кафедры теоретической кибернетики и методов оптимального управления филиала Московского физико-технического института при Институте кибернетики имени В.М. Глушкова НАН Украины.

Специалист в области программной инженерии. Под ее руководством научный отдел выполнил 14 фундаментальных проектов ГКНТ и НАНУ, разработав ряд парадигм программирования сборочного типа и соответствующих CASE-инструментов на веб-сайтах http://sestudy.edu-ua.net, http://programsfactory.univ.kiev.ua и www.intuit.ru для электронного обучения студентов предмету «Программная инженерия». Была научным консультантом 4 докторских диссертаций (1989–1996, 2007), руководителем 11 кандидатских диссертаций и оппонентом более двух десятков диссертаций по специальности 01.05.03 (1991–2013).

Основные научные результаты автора опубликованы в 20 зарубежных и в 120 отечественных статьях, а также в 3 самостоятельных монографиях и 6 в соавторстве с учениками. Разработала четыре учебника по программной инженерии и технологии программирования. Е.М.Лаврищева – лауреат премии кабинета Министров СССР (1987) и комитета по науке и техники Украины (1992, 2003). Награждена знаком НАНУ "За подготовку научной смены" (2007).