Рефакторинг архитектуры программного обеспечения

Ксензов М.В.

1. Введение

В настоящее время вопросы сопровождения и развития существующего программного обеспечения получают недостаточное внимание со стороны научного сообщества, стандартизационных комитетов и разработчиков инструментальных средств. Как следствие этого, существует явный недостаток методик и эффективных инструментов поддержки работы с существующим кодом. Современные методики несколько переоценивают значение начальной фазы полного жизненного цикла программной системы. При этом полный жизненный цикл программной системы представляет собой эволюционный процесс производства многочисленных версий системы, и на всех фазах данного процесса, за исключением начальной, разработка системы ведется на основе существующего кода.

Если в 70-ые годы усиленное внимание к начальной фазе было оправдано, поскольку начальная фаза составляла большую часть времени жизни проекта, то в настоящее время ситуация изменилась коренным образом: начальная фаза занимает малую часть времени жизни проекта, а большую часть занимает сопровождение и развитие системы, то есть работа с существующим кодом [1].

Вопросы систематического использования трансформаций как центрального организующего принципа процесса развития и сопровождения существующего программного обеспечения вызывают значительный интерес. Однако, большинство исследователей рассматривает трансформации достаточно узко – как трансформации на уровне исходного кода – рефакторинг. Рефакторинг объектно-ориентированного кода зарекомендовал себя как эффективный способ решения задач эволюции и сопровождения программ.

Однако, на настоящий момент, практически не существует исследований, освещающих рефакторинг на более высоком уровне абстракции – уровне архитектуры ПО. В то же время, многие сценарии сопровождения и развития существующего кода подразумевают изменение архитектуры существующей системы. В связи с этим, большой интерес вызывает разработка методики и сопровождающих ее инструментальных средств, нацеленных на организацию предсказуемого и управляемого процесса изменения архитектуры ПО.

В данной работе рассматривается возможность и выгоды применения методов рефакторинга (процесса, традиционно ориентированного на изменение программного кода) на более высоком уровне абстракции: то есть на уровне архитектуры программных систем.

2. Архитектура программного обеспечения

В настоящее время не существует единого, принятого всеми сторонами определения термина “архитектура программного обеспечения”. С другой стороны, существует большое количество различных определений этого понятия, имеющих во многом схожий смысл. Большинство этих определений понимают под архитектурой высокоуровневое структурное описание программной системы, а также зачастую способы расширения функциональности этой системы.

В качестве примера можно привести следующее определение архитектуры программного обеспечения: *архитектура программного обеспечения - это первичная организация системы, сформированная ее компонентами, отношениями между компонентами и внешней средой системы, а также принципами, определяющими дизайн и эволюцию системы* [2].

2.1. Зачем менять архитектуру ПО?

Потребность в трансформации существующего программного обеспечения может возникнуть как результат широкого круга задач его модернизации. В качестве примера можно привести следующие сценарии, требующие изменения существующего ПО:

- Преобразования, обусловленные функциональными изменениями ПО.
- Смена платформы ПО.
- Обновление технологии разработки ПО.
- Преобразования, связанные с реорганизацией компании, ведущей разработку.

Список сценариев, приводящих к потребности в трансформациях существующего ПО, на этом не исчерпывается: приведенные выше примеры призваны лишь продемонстрировать широкий спектр задач, которые обусловливают необходимость подобных трансформаций.

В общем случае трансформации существующего программного обеспечения способны затронуть не только ее код: но и все остальные артефакты, связанные с трансформируемой программной системой. Пожалуй, одной из наиболее существенных разновидностей здесь является *трансформация архитектуры программной системы*.

2.2. Рефакторинг: систематические трансформации архитектуры.

Как уже говорилось, потребность в изменении архитектуры может возникнуть в различных сценариях. В силу большой актуальности задачи изменения

архитектуры, возникает интерес в организации методического и управляемого подхода к ее решению, а также сопровождающих ее инструментальных средств.

Вопросы систематического использования трансформаций как центрального организующего принципа процесса развития и сопровождения существующего программного обеспечения вызывают значительный интерес. Однако большинство исследователей рассматривает трансформацию достаточно узко – как трансформацию на уровне исходного кода (рефакторинг).

Рефакторинг (сущ.) – это изменение во внутренней структуре ПО, имеющее целью облегчить понимание его работы и упростить модификацию, не затрагивая наблюдаемого поведения. Производить рефакторинг – изменять структуру ПО, применяя ряд методов рефакторинга, не затрагивая его поведения.

В привычном понимании разработки ПО сначала создается дизайн системы, а потом пишется ее код. Со временем код модифицируется, и целостность системы, соответствие ее структуры изначально созданному дизайну постепенно ухудшается. Дальнейшее развитие системы постепенно сползает от направленной, проектируемой деятельности к хакерству.

Рефакторинг представляет собой противоположную практику. С его помощью можно взять плохой, хаотический проект и переделать его в хорошо спроектированный код. Каждый шаг этого процесса чрезвычайно прост. Например, шагом может стать:

- перемещение поля из одного класса в другой
- перемещение метода из класса в класс
- расщепление класса.

Однако суммарный эффект таких небольших изменений оказывается кумулятивным и может радикально улучшить проект. Процесс рефакторинга является прямой противоположностью постепенной деградации кода системы [3].

При описании методов рефакторинга принято использовать частично-формализованный формат: описание каждого шага рефакторинга называется *паттерном*. Любой паттерн описывает и именует типовую задачу, которая постоянно возникает в работе, а также принцип ее решения, причем таким образом, что это решение можно использовать потом снова и снова. Паттерн именует, абстрагирует и идентифицирует ключевые аспекты структуры общего решения [4]. Помимо прочего, паттерны формируют словарь в проблемной области и позволяют двум специалистам в этой области именовать типовые решения и понимать друг друга, не объясняя каждый раз суть самих решений.

Например, в [3] паттерн имеет следующую структуру:

- Название паттерна.
- Краткая сводка ситуаций, в которых требуется данный метод.
- Мотивировка применения.
- Пошаговое описание применения метода рефакторинга.

- Примеры.

В качестве примера можно привести паттерн из [3]:

- **Название паттерна**: перемещение метода.
- **Краткая сводка ситуаций, в которых требуется данный метод**: поле используется или будет использоваться другим классом чаще, чем классом, в котором оно определено.
- **Мотивировка применения**: Перемещение состояний и поведения между классами составляет саму суть рефакторинга. По мере разработки системы выясняется необходимость в новых классах и перемещении полей между ними. Разумное и правильное решение через некоторое время может оказаться неправильным. Это не проблема, если не оставлять это без внимания.
- **Пошаговое описание применения метода рефакторинга**:
    1. Если поле "открытое" – выполните его инкапсуляцию.
    2. Выполните компиляцию и тестирование.
    3. Создайте в целевом классе поле с методами для чтения и установки значений.
    4. Скомпилируйте целевой класс.
    5. Определите способ ссылки на целевой класс из исходного
    6. Удалите поле из исходного класса.
    7. Замените все ссылки на исходное поле обращениями к соответствующим методам в целевом классе.
    8. Выполните компиляцию и тестирование.

Необходимо отметить, что рефакторинг объектно-ориентированного кода зарекомендовал себя как эффективный способ решения задач эволюции и сопровождения программ. Однако на настоящий момент практически не существует исследований, освещающих рефакторинг на более высоком уровне абстракции – уровне архитектуры ПО.

Соответственно, вызывает значительный интерес следующий вопрос - возможен ли перенос данной методологии на более высокий уровень абстракции, с целью получения аналогичной методики систематического изменения архитектуры ПО? *В настоящей работе (в разделе 3) будет рассмотрен пример, иллюстрирующий то, как можно проводить рефакторинг архитектуры, а также выгоды, которые дает его использование.*

2.3. Как можно описать архитектуру и ее изменения?

Специфика исследования и трансформации архитектуры программного обеспечения заключается в том, что, в отличие от программного кода, архитектура не имеет явного представления, за исключением, может быть, тех случаев, когда она явно задокументирована. Однако даже в последнем, оптимистическом случае трудно гарантировать соответствие

задокументированной архитектуры той фактической высокоуровневой логической структуре, которая на самом деле существует в системе.

Способом описания архитектуры и ее изменений могут стать структурные модели. В настоящее время существует большое количество нотаций и инструментов, поддерживающих структурное моделирование ПО. В свете важности соответствия модели фактической структуре существующего кода при моделировании архитектуры, представляется исключительно важной возможность автоматического извлечения подобных моделей из кода программных систем, поскольку автоматическое извлечение моделей из кода гарантирует их актуальность и точность.

Для дальнейшего исследования архитектуры программных систем используется нотация структурного моделирования, принятая в инструменте KLOCwork Architect. KLOCwork Architect предоставляет возможность автоматического извлечения моделей из программного кода и редактирования их. Далее рассматривается эта нотация.

2.3.1. Модель ПО в KLOCwork Architect

Модели программных систем, используемые в KLOCwork Architect (в дальнейшем модель) [5], отдаленно напоминают модели типа сущность-отношение (Entity-Relation models). Основными единицами модели являются следующие элементы:

- **Архитектурный блок (Architecture Block)**. Архитектурные блоки – это основные элементы, составляющие модель. Архитектурные блоки отображают в модели структурные элементы программной системы вне зависимости от того уровня абстракции, на котором идет моделирование. Архитектурные блоки обладают, по меньшей мере, двумя основными атрибутами: имя и тип. Имена архитектурных блоков предопределяются именами тех структурных элементов системы, которые они представляют в модели. Типы архитектурных блоков существенно зависят от уровня абстракции, на котором проходит моделирование, и конкретной задачи, в рамках которой проводится исследование архитектуры. Например:
    - При моделировании взаимодействия клиент-сервер – основной используемый тип архитектурных блоков - “подсистема”.
    - При моделировании систем, построенных в рамках каких-либо компонентных технологий, – основной используемый тип - “компоненты”.
    - При моделировании системы сборки ПО – основные используемые типы - “папки” и “файлы”.
    - При объектно-ориентированном анализе – основной используемый тип - “класс”.

- **Отношение (Relation)**. В модели KLOCwork Architect под отношением понимается некоторая односторонняя связь между парой архитектурных блоков. Между любой парой блоков в модели может быть произвольное количество разнонаправленных отношений, при этом их типы так же могут различаться. Так же, как и архитектурные блоки, отношения могут быть различных типов. В качестве примера можно привести следующие типы отношений:
    - Инстанциация: A инстанциирует B (блок A – функция, блок B – класс).
    - Наследование: A наследует B (блоки A и B – классы).
    - Чтение данных: A читает данные из B (блок A – функция, блок B – класс, атрибут класса или функция).
    - Обращение: A вызывает B (блоки A и B – функции).
    - A использует B: (блок A – класс или функция, блок B – класс или атрибут класса).

Свойства модели:

- **Иерархичность**: каждый архитектурный блок может содержать другие архитектурные блоки. При этом связи между архитектурными блоками суммируются. Например, если в модели есть блок A, который содержит блок A1, и блок B, который содержит B1и B2, и между блоками A1 и B1 есть связь, а также между блоками A1 и B2 есть связь, то считается что между A и B есть две связи, поскольку они содержат два множества блоков, и количество связей между блоками из различных множеств равно двум.
- **Точность**: для каждого элемента модели можно указать «стоящий за ним» в моделируемой системе набор файлов и строк кода. Точность модели обеспечивается способностью инструмента KLOCwork Architect автоматически извлекать их из программного кода (более подробно об этом – в разделе 2.3.2).

Над моделями в KLOCwork Architect определены операции редактирования:

- **Добавление блока.** В модель допустимо добавлять новые блоки.
- **Удаление блока.** Из модели допустимо удалить произвольный блок.
- **Перенос блока.** Из модели допустимо вырезать блок и перенести его внутрь другого блок.
- **Переименование блока.** В модели допустимо переименовывать блоки.
- **Объединение блоков в группу.** Модель позволяет объединять блоки в группы. При этом создается новый блок, и группируемые элементы переносятся внутрь него.

Важным свойством этих операций является то, что они сохраняют основные свойства модели – то есть ее иерархичность и точность.

Следует отметить, что в настоящий момент группой Object Management Group (OMG) предпринимаются усилия по стандартизации "метамодели обнаружения знаний" (KDM – Knowledge Discovery Model) [6]. Эта модель призвана

обеспечить обмен информацией, связанной с трансформацией существующих ресурсов программного обеспечения и их операционных сред. Это позволит поставщикам решений, специализирующихся на определенных языках, платформах или типах трансформаций поставлять решения, сочетающиеся с решениями других производителей [7]. Кроме того, появление метамодели позволит формулировать и описывать архитектуру существующих систем, а также способы ее трансформации универсальным образом, вне зависимости от конкретных инструментальных средств и моделей.

2.3.2. Автоматически извлекаемые модели

KLOCwork Architect способен автоматически извлекать из программного кода базовые структурные модели, отражающие физическую структуру исследуемой программной системы. В состав таких моделей входят:

- Архитектурные блоки, представляющие папки, в которых находятся файлы с исходным кодом системы.
- Архитектурные блоки, представляющие собственно файлы. Такие архитектурные блоки находятся внутри соответствующих блоков, которые представляют папки, содержащие эти файлы.
- Архитектурные блоки классов, переменных, функций, находящихся внутри соответствующих блоков файлов.

Далее, в качестве примера, будет рассмотрена небольшая тестовая система на языке C++ и модель, автоматически полученная из него системой inSight. Система имеет следующую структуру:

Папка include, содержащая:
Файл a.h
Файл b.h
Папка src, содержащая:
Файл a.cpp
Файл b.cpp

Исходный код a.h:

```
class A {
   private :
      int a;
   public :
      void a();
         };
```

Исходный код b.h:

```
         #include "a.h"

class B : public A {
   private :
      int b;
   public :
      void b();
};
```

Исходный код a.cpp:

```
#include "a.h"

void A::a() {
           a++;
}
```

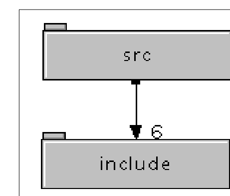
Исходный код b.cpp:

```
#include "b.h"

void B::a() {
           a--;
}
void B::b() {
           b++;
}
```

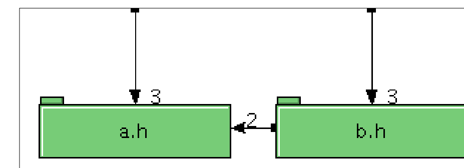

Для подобной системы извлеченная автоматически модель будет иметь следующую структуру:

Корневая диаграмма



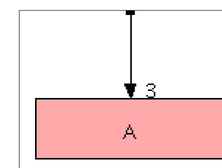
Содержит:
- Имя блока: src, тип: directory
- Имя блока: include, тип: directory

Диаграмма include



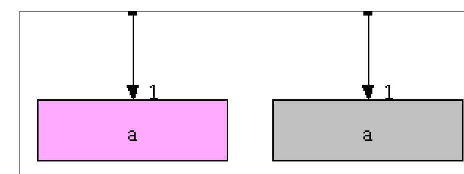
Содержит:
- Имя блока: a.h, тип: file
- Имя блока: b.h, тип: file

Диаграмма a.h



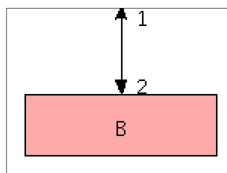
Содержит:
- Имя блока: A, тип: class

Диаграмма A



Содержит:
- Имя блока: a, тип: function-declaration
- Имя блока: a, тип: variable

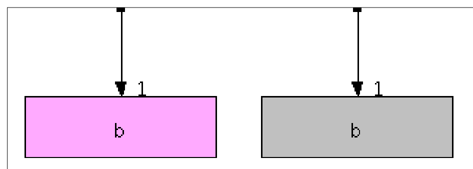
Диаграмма b.h



Содержит:

- Имя блока: B, тип: class

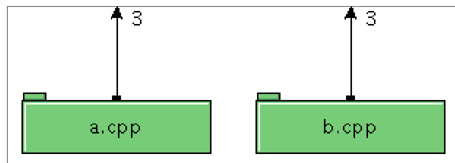
Диаграмма B



Содержит:

- Имя блока: b, тип: function-declaration
- Имя блока: b, тип: variable

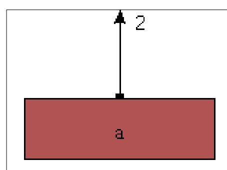
Диаграмма src



Содержит:

- Имя блока: a.cpp, тип: file
- Имя блока: b.cpp, тип: file

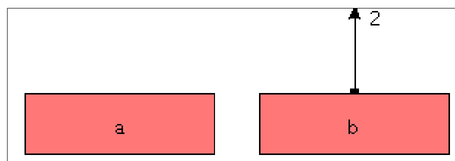
Диаграмма a.cpp



Содержит:

- Имя блока: a, тип: function-definition

Диаграмма b.cpp



Содержит:

- Имя блока: a, тип: function-definition
- Имя блока: b, тип: function-definition

Подобные модели извлекаются из программного кода автоматически. Хотя они и являются структурными, едва ли их можно назвать моделями архитектуры:

физическая структура хранения исходного кода программной системы, которую они фактически представляют, как правило, достаточно сильно отличается от логической структуры. Тем не менее, подобные модели являются хорошим стартовым пунктом для исследования архитектуры.

3. Пример

В данном разделе будет рассмотрен пример применения архитектурного рефакторинга. В рассматриваемом примере показывается, как организован архитектурный рефакторинг, а также то, какие выгоды могут быть извлечены из его применения.

Пример носит исключительно иллюстративный характер. Очевидно, что для подобного примера нужна система, не слишком большая, чтобы пример был обозримым, и в то же время он должен обладать достаточным объемом, чтобы продемонстрировать основные нюансы исследуемой методики. В качестве исходного материала для примера была выбрана небольшая система, именуемая *toolbus*, реализующая механизмы, которые обеспечивают взаимодействие и коммуникацию программных инструментальных средств. В основе функционирования данной системы лежит использование сокетов операционной системы. Для организации взаимодействия инструментальных средств в системе используется специальный протокол обмена. Протокол предполагает, что инструментальные средства обмениваются сообщениями, каждое из которых состоит из кода команды, набора возможных параметров и адреса назначения, если сообщение не является широковещательным. В наиболее общем виде сообщение можно представить как:

```
Message ::== MessageType [Parameters] [Destination]
```

Чтобы показать практическую значимость архитектурного рефакторинга, будем отталкиваться не от преобразований, нацеленных на улучшение качества вообще, а от конкретной задачи. Предположим, что в связи с возникшими задачами модернизации системы *toolbus* перед разработчиками встала задача по смене протокола: необходимо чтобы каждое сообщение теперь предоставлялось в формате XML. Например, предположим, что теперь требуется представить каждое сообщение в следующем виде:

```
<message>
<type>MessageType</type>
<parameters>
<parameter name="parameter_name_1">ParameterValue1
</parameter>
<parameter name="parameter_name_2">ParameterValue2
</parameter>
```

```
</parameters>
<destintaion id="destiantion_id"/>
</message>
```

В рассматриваемом примере демонстрируется, как проведение архитектурного рефакторинга существующей системы позволяет минимизировать затраты на подобную смену протокола.

3.1. Структурная модель

На Рис. 1 приведена основная диаграмма структурной модели, автоматически полученной из исследуемой системы. На этой диаграмме представлены все исходные файлы системы.

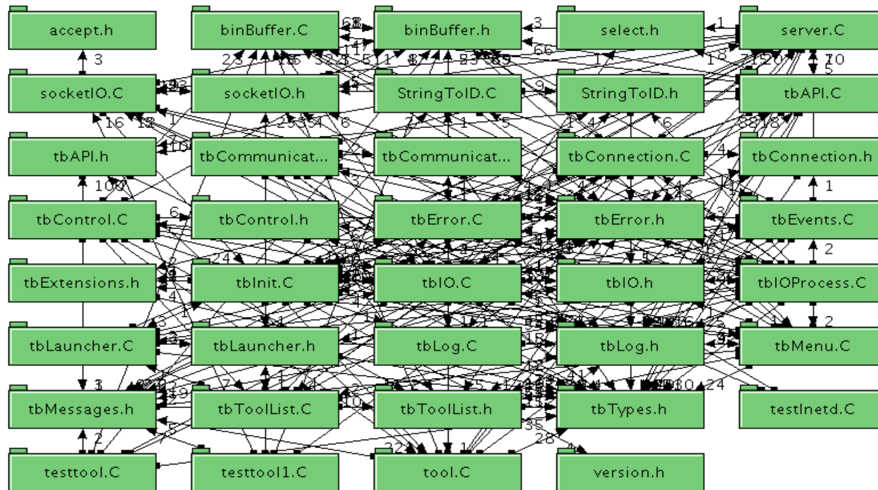


Рис. 1. Структурная модель системы, полученная автоматически.

Как видно, даже для небольшой системы, которая рассматривается в данном примере, модель получилась достаточно громоздкой и трудно обозримой. Фактически, она пока что никак не отражает архитектуру системы. Однако в ней представлено все, что необходимо для восстановления и рефакторинга архитектуры системы. Первым делом необходимо упростить представленную диаграмму настолько, насколько это возможно, при этом, по возможности, не вникая в значение каждого из представленных на диаграмме блоков.

3.2. Объединение интерфейса и реализации.

Первый паттерн, который будет применен к автоматически извлеченной структурной модели системы toolbus, будет “объединение интерфейса и реализации”:

- **Имя**: объединение интерфейса и реализации.
- **Ситуация**: на диаграмме представлен интерфейс с единственной реализацией.
- **Рецепт**: Объединить блоки, представляющие интерфейс и его реализацию.
- **Пример**:
  - В Java: объединить интерфейс “Manager.java” и его реализацию “ManagerImpl.java”, находящиеся на одной диаграмме (из одного пакета) в “Manager”.
  - В C/C++: объединить “count.h” и “count.c” в новый блок “count”.

Используя паттерн, объединим все заголовки модулей с соответствующими им реализациями. Результат применения паттерна представлен на Рис. 2.

Естественно, диаграмма стала значительно проще. При этом никакие существенные детали не были утеряны: из модели не были удалены какие-либо структурные элементы. Фактически, были скрыты лишь те различия между блоками, которые были обусловлены не логической структурой, а, скорее, способом представления модулей в конкретном языке программирования, который предполагает и поощряет описание интерфейсов и их реализаций в различных файлах. Как видно, диаграмма уже стала значительно проще, тем не менее, все еще возможно дальнейшее упрощение диаграммы, которое также не потребует никакого специфического анализа представленных блоков.

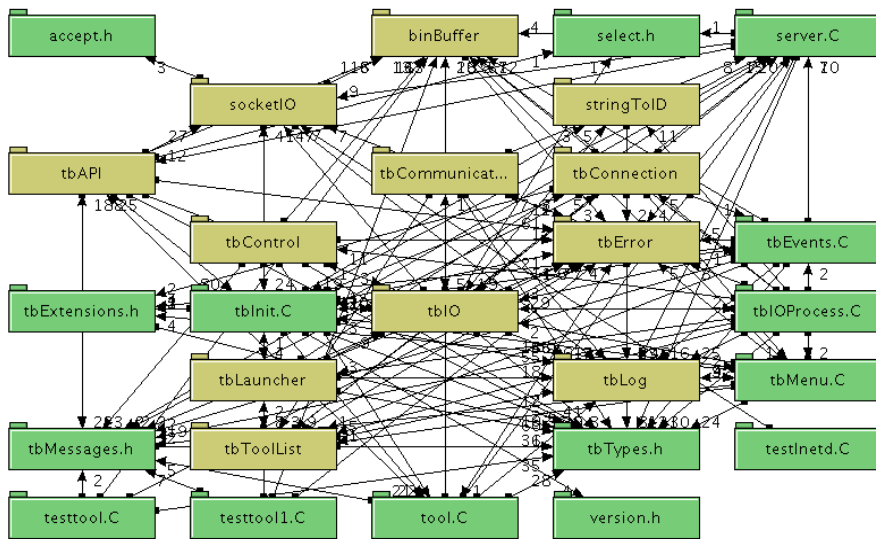


Рис. 2. Результат объединения интерфейсов с реализацией.

3.3. Удаление вспомогательных элементов

- **Имя**: Удаление вспомогательных элементов.
- **Ситуация**: на диаграмме представлены элементы, не имеющие непосредственного отношения к функционированию системы – например, тесты или элементы, используемые вне проекта, и пр.
- **Рецепт**: Удалить блоки представляющие на диаграмме артефакты, не используемые исследуемой системой.
- **Пример**:
  - В Java: удалить пакеты “test”, содержащие наборы unit-тестов.
- **Ограничения**: Вспомогательные элементы часто показывают связи, необходимые клиентской программе для работы с системой. Эта информация важна для некоторых задач рефакторинга и не должна быть утеряна.

В исследуемой модели также представлены тестовые элементы – маленькие кусочки кода, подключающиеся к системе и проверяющие ее способность передавать сообщения. Такие тестовые элементы в случае рассматриваемого примера можно опознать по наличию точки входа (функция main). Их следует удалить. Боле конкретно, удалению подлежат блоки: testtool.C, testtool1.C, tool.cC, testInetd.C и server.C. Результат применения паттерна представлен на Рис. 3.

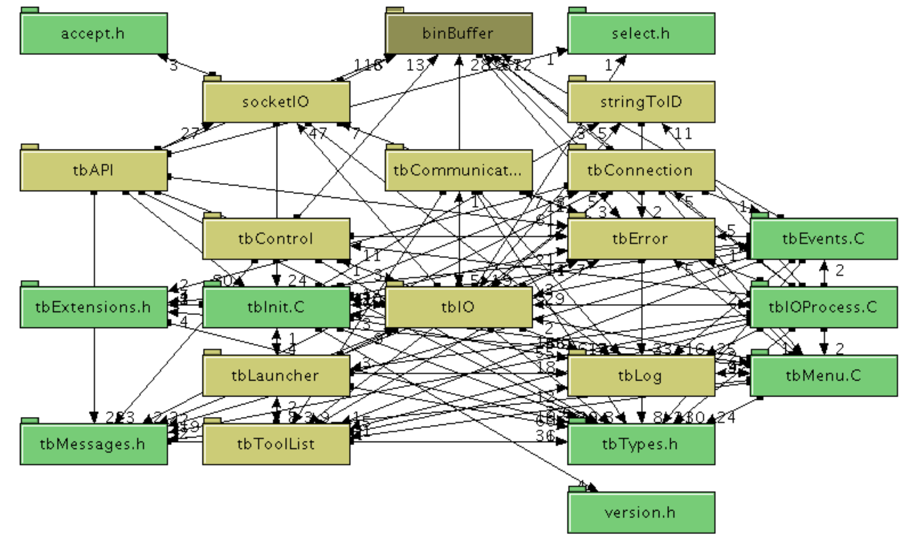


Рис. 3. После удаления вспомогательных элементов.

3.4. Инкапсуляция приватных блоков

Использование двух паттернов позволило существенно упростить исходную диаграмму. Дальнейшая работа с диаграммой обуславливается намерением поднять ее уровень абстракции от структуры хранения проекта во внешней памяти (именно на этом уровне абстракции находится исходная структурная модель) к архитектурному представлению.

Паттерн “инкапсуляция приватных блоков” позволяет скрыть те блоки, которые используются как некоторые частные вспомогательные элементы, не играющие роли в высокоуровневой, архитектурной структуре программы.

- **Имя**: инкапсуляция приватных блоков.
- **Ситуация**: на диаграмме представлен элемент, используемый одним и только одним архитектурным блоком (имеют только одно входящее отношение) – “приватный” блок.
- **Рецепты**: Можно:
  - Объединить “приватный” блок с его “хозяином”.
  - Переместить “приватный” блок в “хозяина”.
- **Пример**: В C++: заголовочный файл (header) используется только в одной и только одной папке с исходными файлами и, при этом, находится вне ее. Такой заголовочный файл можно перенести внутрь папки.

На исследуемой диаграмме имеется такой блок - “accept.h” - он используется только блоком socketIO. Это позволяет нам переместить “accept.h” в socketIO внутри нашей модели, убрав его, таким образом, с основной диаграммы. Кроме того, version.h используется только в tbInit.C, объединим эти два блока в новый архитектурный блок “tbInit”.

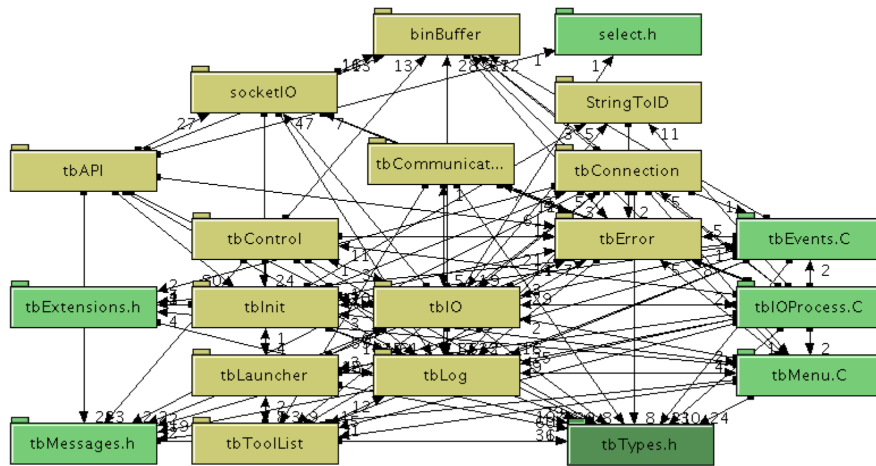


Рис. 4. Блоки accept.h и version.h теперь скрыты.

3.5. Выделение слоя

Для дальнейшего анализа диаграммы необходимо “погружение” в ее семантические свойства. По возможности, подобный семантический анализ надо упрощать, минимизировать и откладывать столь долго, сколь это возможно. Действительно, применение паттернов, описанных выше, позволило существенно сократить количество элементов диаграммы, значительно упростив дальнейший семантический анализ за счет сокращения объема этого анализа. Именно поэтому чем позже будет начат семантический анализ, тем (с большой вероятностью) проще он будет.

К счастью, хотя в дальнейшем полностью избежать рассмотрения семантики элементов диаграммы не удастся, есть паттерн, который позволит упорядочить и минимизировать этот анализ на данном этапе. Речь идет о паттерне “выделение слоев”.

- **Имя**: выделение слоев.
- **Ситуация**: на диаграмме представлены элементы, для которых верны условия, приведенных ниже:
  - исходящие связи ведут только в уже выделенные слои, или их нет, если ранее не был выделен ни один слой;

  - “кандидаты на объединение в новый слой” должны обладать общим смыслом и/или функциональностью. Простейшей проверкой на наличие общности является простой критерий: если для кандидатов можно подобрать "общее определение", то можно считать, что они обладают требуемой общностью.
- **Рецепт**: Объединить блоки в новый слой.

Послойная структура хороша тем, что она достаточно легко формализуется и, в то же время, она заключает в себе достаточно простую и емкую смысловую нагрузку: слои в системе, как правило, можно рассматривать как уровни абстракции системы.

Анализ входящих / исходящих связей позволил легко определить кандидатов на объединение в нулевой слой. В соответствие с приведенным паттерном, в кандидаты на объединение в нулевой слой входят те блоки, которые имеют входящие связи, но не имеют исходящих связей. Такие блоки могут быть найдены на исследуемой диаграмме – это блоки stringToID, tbTypes.h и tbError. Беглый просмотр содержимого блока tbTypes.h показал, что он содержит определение основных типов данных, используемых в исследуемой системе, таких как идентификатор инструментального средства, адрес назначения и пр. Блок stringToID позволяет вычислять/назначать внутренний идентификатор инструментального средства, использующего систему toolbus. Блок tbError определяет понятие ошибки в системе. Таким образом, все три блока определяют основные типы данных системы – идентификатор инструментального средства, адрес назначения, ошибка системы и пр. Результат объединения их в нулевой слой показан на Рис. 5.

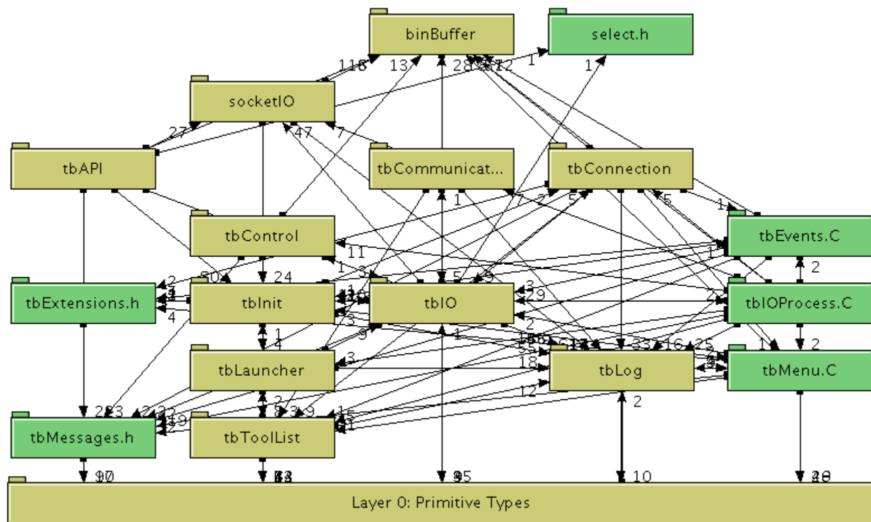


Рис. 5. Нулевой слой “Layer 0: Primitive types”.

Выделение слоев может быть продолжено. Анализ входящих/исходящих связей показал, что наиболее вероятные кандидаты на объединение в новый слой – это блоки binBuffer и tbToolList. Эти блоки используются (имеют входящие связи) практически всеми остальными элементами системы. Кроме того, для них можно подобрать общее определение: они представляют “составные типы данных”, используемых в системе. Действительно, tbToolList – реализует список идентификаторов инструментальных средств (которые определены в нулевом слое), используемый системой для организации обменов. Блок binBuffer – определяет двоичный буфер, используемый системой для обмена данными. Однако, в отличие от предыдущего опыта, на этот раз не удастся выделить новый слой так же гладко, как это удалось при первой попытке: tbToolList использует tbLauncher и tbLog, то есть блоки вне нулевого слоя. Потребуется дополнительный анализ для разрешения возникшей коллизии.

3.6. Выделение util-блока

Оправдано ли использование списком инструментов, определяемых в блоке tbToolList каких-либо блоков вне нулевого слоя, определяющего собственно понятие инструмента? Каков характер этих “проблемных связей”?

Первым делом, необходимо определить, что из себя представляют используемые блоки, находящиеся вне нулевого слоя. Анализ блока tbLog показал, что он используется практически всеми другими блоками на диаграмме и реализует подсистему журнализации событий. Блок tbLauncher реализует

подсистему запуска удаленных инструментальных средств. Использование блока tbLauncher блоком tbToolList обуславливается тем, что в нем содержатся статические параметры, определяющие поведение всей программы: статический булевский флаг ведения журнала (если он принимает значение “истина” - события журнализуются и наоборот), набор timeout-констант, используемых системой для прерывания обмена по времени. Очевидно, что эти параметры хранятся в "неправильном месте" и должны быть перенесены из блока tbLauncher. По смыслу, timeout-константы вполне можно перенести в нулевой слой. А вот для работы с блоком tbLog будет использован следующий паттерн:

- **Имя**: выделение util-блока.
- **Ситуация**: на диаграмме представлены элементы, которые реализуют вспомогательную функциональность, используемую всей системой и не меняющую ее состояния.
- **Рецепт**: Объединить такие блоки в util-блок, который:
  - может быть перенесен с текущей диаграммы на диаграмму верхнего уровня.
  - игнорируется при выделение слоев.
- **Пример**:
  - В Java: классы, содержащие только статические методы, могут быть перемещены в util-блок.
  - В C/C++ и Java: классы, содержащие только метод "main" и используемые для вывода конфигурации/диагностики, также могут быть перемещены в util-блок.

В результате применения этого паттерна на диаграмме появляется util-блок, и становится возможным выделение нового слоя, что отражено на Рис. 6.

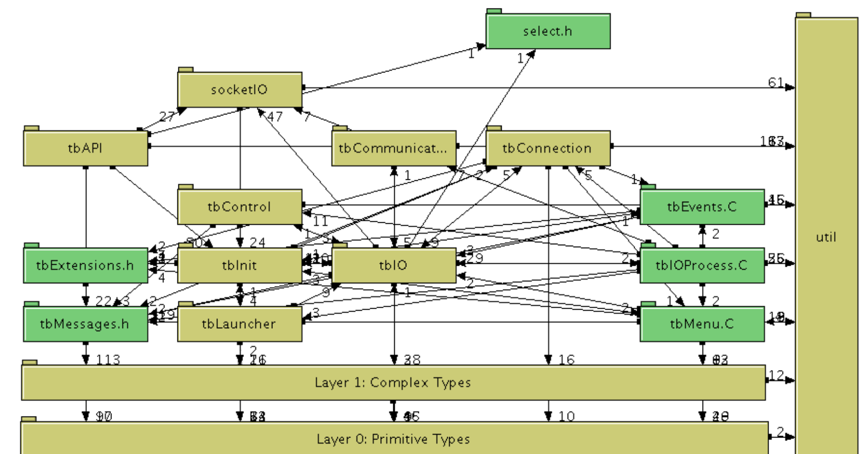


Рис. 6. util-блок и новый слой.

3.7. Обнаружение “драйвера”

Зачастую достаточно полезным для анализа модели является обнаружение блока, содержащего точку входа программной системы. Такой блок именуется “драйвером системы”. Интерес к “драйверу” обуславливается тем, что:

- Во-первых, такой блок может стать хорошей точкой отсчета для дальнейших исследований. Если такой блок найден, можно организовать изучение блоков системы “сверху-вниз”.
- Во-вторых, как правило, такой блок имеет смысл обособить (например, вынести его на диаграмму верхнего уровня) или даже удалить, упростив, таким образом, диаграмму. Объясняется это тем, что “драйвер”, как правило, не несет никакой смысловой нагрузки – его задача – просто запустить систему.

Вследствие единственности и специфичности задачи, стоящей перед “драйвером”, у него появляется любопытное свойство: у драйвера не должно быть исходящих связей. Действительно: драйвер может и должен использовать другие блоки системы – он ведь именно к ним обращается при запуске. Но другим блокам системы нет никакой необходимости знать о драйвере – он им не нужен для функционирования, поскольку сам по себе не предоставляет функциональности, иной, кроме запуска системы. Итак:

- **Имя**: обнаружение "драйвера".
- **Ситуация**: блок отвечает за запуск системы.
- **Рецепт**: Такой блок можно:
  - Обособить;
  - Выкинуть, если он не играет роли в рамках поставленной задачи.
- **Ограничения**: У блока не должно быть входящих связей.

В рассматриваемом примере “драйвером” является блок “tbInit” - это единственный оставшийся в системе блок, содержащий функцию “main”. Заметим, что для него не выполняется ограничение об отсутствии входящих связей. С чем это связано? Небольшая инспекция выявила, что в tbInit есть статический параметр, используемый всей системой – currentProgramName. Нужен этот параметр для журнализации событий и поэтому он должен быть перемещен в блок tbLog. Используемая системой и определенная в tbInit функция allocAndCopy, которая выделяет место и копирует туда заданную строку, перемещена в util.

Теперь блок tbInit является драйвером в полном смысле этого слова и, поскольку он не дает ничего для анализа системы и решения поставленной задачи, может быть удален.

Теперь выделение слоев может быть продолжено. Поступаем аналогично тем действиям, которые были описаны в 3.6. Кандидаты на объединение в очередной слой – это tbMessages.h и socketIO. tbMessages.h – содержат

определение кодов команд, используемых в коммуникационном протоколе системы, описанном в начале примера. Блок socketIO определяет оболочку для сокетов операционной системы, упрощающую передачу команд, определенных протоколом по сокетам. Таким образом, обнаружена пара кандидатов на новый слой “реализации протокола”. Поскольку этому не препятствуют существующие связи, выделяем новый слой. Результат показан на Рис. 7.

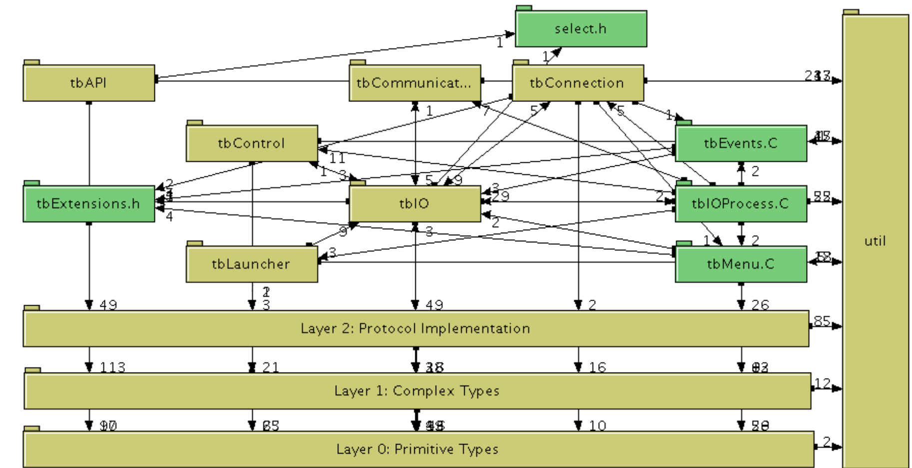


Рис. 7. Слой “реализации протокола”.

3.8. Расщепление API

В результате рефакторинга архитектуры и сопряженного с ним анализа постепенно начинает прорисовываться и восстанавливаться от эрозии архитектура исследуемой системы. В связи с этим особый интерес начинают представлять те блоки на исследуемой диаграмме, которые по-прежнему представляют файлы, поскольку блок, представляющий файл, скорее имеет отношение к первоначальной структурной модели системы, чем к модели архитектуры, которую мы стремимся получить. Первые вопросы, которые возникают при анализе блоков-файлов это:

- Почему эти файлы до сих пор не были объединены в какие-либо из существующих архитектурных блоков?
- Почему к оставшимся блокам-файлам не были применимы описанные ранее паттерны?

Одной из причин, по которой мы не смогли применить ни один из описанных паттернов рефакторинга, является внутренняя структура оставшихся блоков, представляющих файлы.

- **Имя**: расщепление API.

- **Ситуация**: блок описывает API, представляющий не связанную функциональность/поведение.
- **Рецепт**: Разбить такой блок на несколько, так чтобы API каждого из них был "целостным".
- **Ограничения**: "Фасадные" блоки, т.е. блоки, специально созданные для объединения нескольких разных API в одном из соображения удобства использования.

Блоком, к которому можно применить описанный паттерн, является блок-файл tbExtensions.h. Просмотр кода, стоящего за этим блоком, показал, что в нем содержатся две обособленные группы объявлений функций: группа функций, которая связана с действиями над меню инструментальных средств, работающих через систему toolbus (система позволяет добавлять некоторые элементы меню, предоставляемые удаленными инструментами) и группа функций, связанная с рассылкой сообщений через toolbus.

Таким образом, блок tbExtensions.h должен быть расщеплен на tbMenu.h и tbEvents.h. Образованные в результате расщепления файлы можно объединить соответственно с tbMenu.C и tbEvent.C, применяя паттерн “объединение интерфейса и реализации”.

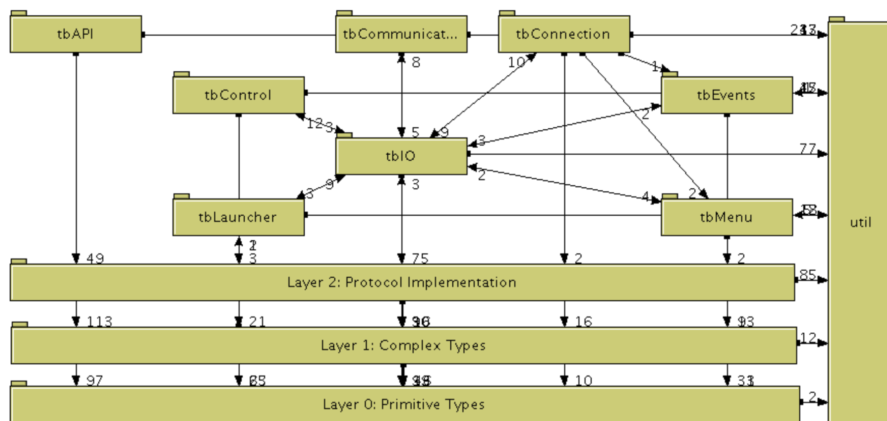


Рис. 8. После расщепления, удаления вспомогательного и инкапсуляции приватного блоков.

В результате на диаграмме осталось только два блока-файла: select.h и tbIOProcess.C. Анализ использования блока select.h показал, что он подключается как заголовочный файл (поэтому у него есть входящие связи), однако объявленные в нем сущности не используются. Следовательно, нет никакой необходимости подключать его как заголовочный файл, а, следовательно, нет необходимости в его использовании вообще. Похоже, что

он просто не был удален в то время, когда в результате модификаций он перестал использоваться. Используя паттерн “удаление вспомогательных элементов” выкинем этот блок. Оставшийся блок tbIOProcess.C используется только блоком tbIO, и, следовательно, является приватным. В соответствии с паттерном “инкапсуляция приватных блоков” он может быть перемещен в tbIO. Получившаяся в результате описанных трансформаций диаграмма представлена на Рис. 8.

3.9. Последний рывок

Попробуем дальше выделять слои: анализ связей показал, что на роль очередного слоя подходит блок tbAPI. tbAPI предоставляет программный интерфейс к функциональности, обеспечиваемой протоколом toolbus. Для каждой команды, определенной протоколом, он предоставляет функцию, которая передает соответствующий код команды и параметры по сокетам toolbus. Таким образом, tbAPI представляет очередной уровень абстракции системы toolbus, поскольку скрывает как сами сокеты, так и коды команд.

Все оставшиеся блоки, кроме рассмотренного tbAPI, предоставляют реализацию различных прикладных технологий поверх протокола toolbus – таких, как рассылка сообщений или описанная выше поддержка работы с меню удаленных инструментальных средств. Поэтому все эти блоки можно объединить в “прикладной уровень”. Восстановленная архитектурная модель представлена на Рис. 9.

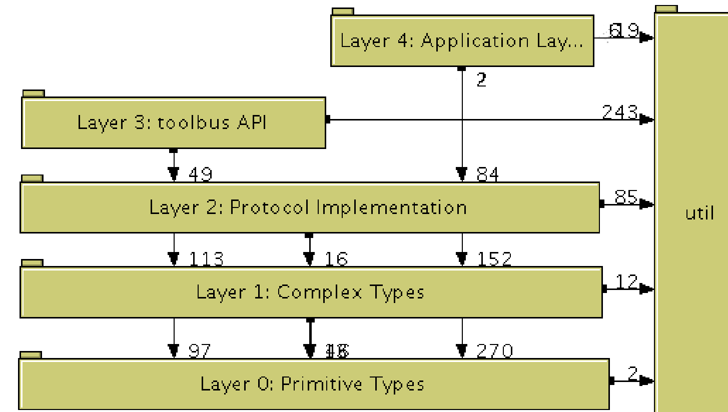


Рис. 9. Архитектура системы toolbus

При рассмотрении связей между слоями нельзя не заметить одной странности: слой 4 использует слой 2 напрямую, в обход слоя 3! Выявлен архитектурный дефект, который в условиях поставленной задачи затруднит ее реализацию.

Действительно, все прикладные механизмы системы toolbus игнорируют прикладной интерфейс системы и напрямую работают с сокетами, что чрезвычайно затрудняет смену протокола – ведь после смены протокола придется править все эти прикладные механизмы. Для устранения дефекта достаточно заменить все вызовы функций сокетов toolbus на соответствующие им функции из API системы toolbus. Для этого применяется паттерн “восстановление связей между слоями”.

- **Имя**: восстановление связей между слоями.
- **Ситуация**: Между выделенными в системе архитектурными слоями существуют связи, “нарушающие субординацию подчинения” - то есть связи между слоями, не являющимися соседними в иерархии слоев.
- **Рецепт**: Восстановить “субординацию”: если слой n+1 хочет использовать слой n-1, он должен делать это посредством косвенных обращений, то есть через слой n.

Проведенный эксперимент по трансформации системы toolbus был призван продемонстрировать возможность применения процесса рефакторинга на высоком уровне абстракции - уровне архитектуры системы. Подводя итоги, следует отметить, что в результате проведенного рефакторинга удалось:

- выявить архитектурную модель системы
- восстановить архитектуру от последствий эрозии (все изменения, проводимые на модели, в силу точности последней, могут быть перенесены на программный код)
- выявить архитектурный дефект, устранение которого значительно упростит решение поставленной задачи.

4. Замечания

4.1. Проецирование

Редактирование модели, обусловленное применением шагов рефакторинга в рассмотренном примере, могут быть спроецированы на реальный программный код системы. Действительно, при проецировании удаления блоков из модели, необходимо определить множество строк и файлов, которые в программном коде стояли за удаленным блоком. После этого необходимо удалить из программного проекта выявленное множество. При проецировании переноса блока в модели аналогично переносятся соответствующие строки и файлы в исходном коде программной системы и т.д.

Проецирование шагов, примененных в ходе некоторого архитектурного рефакторинга к программному коду системы, хотя и является чисто механическим действием, тем не менее, позволяет извлечь практическую выгоду из проведенного анализа. Производимые таким образом

трансформации можно рассматривать как *архитектурно-управляемый рефакторинг программного кода*.

4.2. Процессы

Необходимо отметить, что в ходе описанных выше трансформаций модели был выполнен целый ряд шагов, которые могут быть классифицированы по их назначению. Эти шаги относятся к различным процессам, которые выполняются поочередно в ходе рефакторинга архитектуры. Можно условно выделить следующие процессы: процесс "раскопки" архитектуры, процесс трансформации архитектуры и процесс семантического анализа подсистем.

- **"Раскопка" архитектуры.** Шаги, относящиеся к этому процессу, характеризуются тем, что действия, применяемые к модели в ходе этих шагов, не ориентированы на последующее проецирование на программный код. Они нужны только для понимания и структуризации модели. К таким шагам в рассмотренном примере можно отнести: удаление вспомогательных элементов, объединение интерфейса и реализации, удаление “драйвера”. Очевидно, что в реальном исходном коде системы нет смысла удалять тестовые наборы, так же, как и объединять интерфейс и реализацию: это нужно только для понимания и сокращения количества сущностей, с которыми идет работа.
- **Трансформация**. Для шагов, относящихся к трансформации архитектуры, в отличие от шагов процесса раскопки, типично последующее проецирование их на реальный код программной системы. Шаги этого процесса четко связаны с реальной модификацией кода системы и, в конечном счете, ориентированы на его улучшение. К этой группе можно отнести расщепление API, перенос из драйвера системы статических параметров системы в util-блок. Следует также отметить, что часть рассмотренных в примере шагов (например, выделение слоев) не может быть строго отнесена к одной из названных категорий (раскопка и трансформация). На практике это означает, что решение о проецировании этих шагов на код остается на совести разработчика и определяется поставленной задачей.
- **Семантический анализ подсистем**. Параллельно с описанными процессами постоянно идет процесс семантического анализа подсистем: при объединении и разбиении блоков, выделении слоев и т.д. постоянно всплывает задача выявления смысловой нагрузки подсистем. Для такого выявления, даже в первом приближении, зачастую приходится исследовать реальный программный код (здесь опять-таки помогает точность модели), анализировать сигнатуры функций и комментарии, а при отсутствии последних и сам код функций. Задача специалиста, вовлеченного в процесс архитектурного рефакторинга, – по возможности минимизировать объем семантического анализа (например, удалением вспомогательных блоков) и сделать его последовательным и направленным. Такую направленность

дает в рассмотренном примере пошаговое выделение слоев. Действительно, сначала на основе анализа входящих/исходящих связей выделяется группа блоков-кандидатов на объединение в слой, а после этого семантический анализ применяется только к этим кандидатам.

4.3. Автоматизация

Вызывает значительный интерес потенциальная возможность автоматизации применения описанных выше паттернов. Для большинства паттернов может быть реализована компьютерная поддержка, пусть и неполная. В большинстве случаев эта неполнота объясняется необходимостью семантического анализа подсистем.

В качестве примера можно привести поиск блоков-кандидатов на объединение в слой. При большом количестве блоков на диаграмме обнаружение таких блоков становится затруднительным, однако написать алгоритм перебора входящих/исходящих связи не так сложно: при решении задачи, описанной в примере, применялся специальный плагин к используемой для моделирования системе Architect, который обнаруживал кандидатов.

Подобным образом можно в той или иной степени облегчить использование практически всех названных паттернов.

5. Заключение

В заключении хочется упомянуть о направлениях дальнейшего развития рефакторинга архитектуры:

- **Каталогизация**. Это направление связано с дальнейшим сбором, обобщением и классификацией паттернов рефакторинга.
- **Автоматизация**. Как для существующих паттернов, так и для тех, которые еще не были выделены, представляет большой интерес возможность облегчения и автоматизации их применения.
- **Верификация**. Важным элементом технологии рефакторинга является декларируемое сохранение поведения системы при трансформации структуры. Вызывает интерес возможности верификации сохранности поведения при архитектурном рефакторинге.
- **Направленность**. Желательно вооружить человека, знающего паттерны архитектурного рефакторинга и обладающего соответствующими инструментами моделирования, также еще и некой процедурой, определяющей (подсказывающей) последовательность применения патернов, чтобы сделать процесс направленным и, по возможности, конечным. Выделение подобных процедур также представляет значительный исследовательский интерес.

Литература

1. Research Issues in The Renovation of Legacy Systems, A. van Deursen, P. Klint, C. Verhoef, CWI research report P9902, April 1999
2. Recommended Practice for Architectural Description of Software-Intensive Systems, ANSI/IEEE Std 1471-2000
3. Рефакторинг: Улучшение существующего кода, Мартин Фаулер, Символ, Санкт-Петербург, 2003
4. A Pattern Language, C. Alexander, S. Ishikawa, M. Silverstain, M. Jakobson, I. Fiksdahl-King and S. Angel, Oxford University Press, 1977, New York.
5. Insight: reverse engineer case tool, N. Rajala, D. Campara, N. Mansurov, IEEE Computer Society Press, 1999, Los Alamitos, CA, USA Pages: 630 – 633
6. OMG Architecture-Driven Modernization: Knowledge Discovery Meta-model RFP, 2003
7. Архитектурно-управляемая модернизация существующего программного обеспечения, Н. Мансуров, Труды Института Системного Программирования, том 5, Москва 2004