Рефакторинг архитектуры программного обеспечения: выделение слоев

М. Ксензов (misha@ispras.ru)

1. Введение

В последнее время наблюдается тенденция к увеличению продолжительности жизненного цикла успешных программных проектов. Как следствие, растет объем унаследованного кода, поддерживаемого сообществом разработчиков [1]. Именно это объясняет исключительную важность задач, связанных с облегчением сопровождения и развития существующего программного кода. В то же время, этим задачам уделяется недостаточное внимание со стороны научного сообщества и разработчиков инструментальных средств. Как следствие, современные методики переоценивают значение начальной фазы жизненного цикла программной системы и практически игнорируют ее дальнейшую эволюцию. Таким образом, в настоящее время существует явный недостаток методик и эффективных инструментов поддержки работы с существующим кодом.

В последнее время наметился перелом ситуации: стали вызывать значительный интерес вопросы систематического использования трансформаций как центрального организующего принципа процесса развития и сопровождения существующего программного обеспечения. Однако большинство исследователей рассматривает трансформации достаточно узко – как трансформации на уровне исходного кода – рефакторинг [2]. Тем не менее, в настоящее время практически не существует исследований, посвященных трансформации 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 один из основных методов рефакторинга архитектуры ПО – выделение слоев, а также его место в контексте рефакторинга архитектуры как многошагового итеративного процесса.

2. Архитектура ПО и ее рефакторинг

В настоящее время не существует общепринятого определения термина “архитектура программного обеспечения”. В то же время, существует большое

количество различных определений этого понятия, имеющих во многом схожий смысл. В качестве примера можно привести следующее определение: *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* [3].

2.1. Зачем менять архитектуру?

Потребность в изменении существующего программного обеспечения может возникнуть в ходе решения широкого круга задач по его модернизации. В общем случае изменения существующего программного обеспечения способны затронуть не только его код, но и все остальные артефакты, связанные с трансформируемой программной системой. Одной из наиболее существенных разновидностей здесь является *изменение архитектуры программной системы*. В качестве примеров можно привести следующие сценарии, требующие изменения архитектуры существующего ПО:

Преобразования, обусловленные функциональными изменениями ПО. Желательно, чтобы внедрение новой функциональности не затронуло существующую логику системы. Также желательно, чтобы сложность внедрения новой функциональности в существующую систему не превышало существенным образом сложность реализации этой функциональности в рамках нового проекта. Хорошая архитектура позволяет достичь поставленных целей. Итак, изменение существующей архитектуры – хороший шаг на пути внедрения новой функциональности, к тому же облегчающий и дальнейшую эволюцию системы.

Смена платформы ПО. Крайне желательно, чтобы смена платформы ПО как можно меньше затронула существующий код, и чтобы можно было ограничиться изменениями только в узкой платформенно-зависимой прослойке системы. Выделение такой прослойки – архитектурная задача. Ее решение всегда сопряжено с необходимостью изменения архитектуры.

Преобразования, связанные с реорганизацией компании, ведущей разработку. Примером такой реорганизации может стать аутсорсинг. Решение об использовании аутсорсинга – типичный шаг по оптимизации производства. К сожалению, этот шаг зачастую затрудняется проблемой выделения и передачи компонентов для внешней разработки. Изменение архитектуры программной системы способно облегчить решение этой задачи.

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

2.2. Как представить архитектуру и ее изменения?

Специфика описания архитектуры и ее изменений 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. Возможность автоматического извлечения моделей из кода гарантирует их точность и позволяет своевременно их обновлять. Эта возможность становится ключевой при выборе инструментария, поскольку соответствие модели фактической структуре существующего кода при моделировании архитектуры и ее изменений представляется исключительно важным для обеспечения точного и управляемого процесса.

Для дальнейшего исследования архитектуры программных систем в данной статье используется нотация структурного моделирования, принятая в инструменте KLOCwork Architect, который предоставляет возможность автоматического извлечения моделей из программного кода и их редактирования. Приведем краткий обзор этой нотации.

Модели программных систем, используемые в KLOCwork Architect (в дальнейшем модели) [4], отдаленно напоминают модели типа сущность-связь (Entity-Relationship models). Говоря строго, они относятся к классу *контейнерных* моделей, подробно рассматриваемых в работе [5]. Основными элементами модели являются следующие элементы:

Архитектурный блок (Architecture Block). Модель KLOCwork Architect состоит из так называемых архитектурных блоков. Архитектурные блоки представляют в модели структурные элементы программной системы, вне зависимости от 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 происходит моделирование, и конкретной задачи, в рамках которой проводятся исследование архитектуры. Например, при моделировании систем, построенных в рамках каких-либо компонентных технологий, основным используемым типом архитектурных блоков являются "*компоненты*". При моделировании системы сборки ПО основными используемыми типами являются "*папки*" и "*файлы*".

Отношение (Relation). В модели KLOCwork Architect под отношением понимается односторонняя связь между парой архитектурных блоков. 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).

- **Доступ к данным**: A читает данные из B (блок A – функция, блок B – класс или атрибут класса).

Между любой парой блоков в модели может существовать произвольное количество разнонаправленных отношений, при этом их типы также могут различаться.

Пример модели. В качестве иллюстрации рассмотрим микроскопическую тестовую систему на языке C и модель, автоматически полученную из нее системой Architect. Система имеет следующую структуру:

Папка test, содержащая:

- Файл a.h, содержащий текст

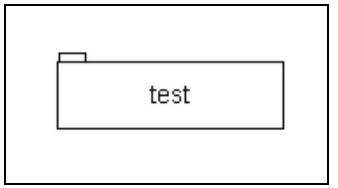
```
void a();
```

- Файл a.cpp, содержащий текст

```
#include "a.h"
void a() {
        int a = 0;
        a++;
    }
```

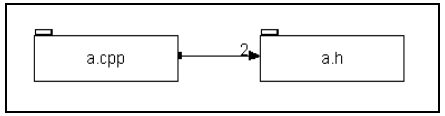
Для подобной системы извлеченная автоматически модель будет иметь следующую структуру:

Корневая диаграмма



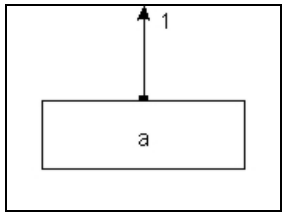
Содержит:
- Имя блока: test, тип: DIRECTORY

Диаграмма test



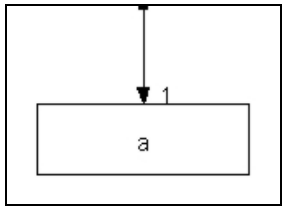
Содержит:
- Имя блока: a.cpp, тип: FILE
- Имя блока: b.cpp, тип: FILE
- Отношение между a.cpp и a.h, тип INCLUDES
- Отношение между a.cpp и a.h, тип DECLARED-BY

Диаграмма a.cpp



Содержит:
- Имя блока: a, тип: FUNCTION
- Исходящее отношение в блок a с диаграммы a.h, тип DECLARED-BY

Диаграмма a.h



Содержит:

- Имя блока: a, тип: FUNCTION-DECLARATION
- Входящее отношение из блока a с диаграммы a.cpp, тип DECLARED-BY

2.3. Рефакторинг 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.

Одним из наиболее успешных подходов к изменению существующего программного обеспечения является рефакторинг – подход, основанный на систематических трансформациях исходного кода. *Рефакторинг* – это изменение во внутренней структуре ПО, имеющее целью облегчить понимание его работы и упростить модификацию, не затрагивая наблюдаемого 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 или метода из одного класса в другой, расщепление класса и т.д. Однако суммарный эффект таких небольших изменений оказывается куму-лятивным и может радикально улучшить проект. Процесс рефакторинга явля-ется прямой противоположностью постепенной деградации кода системы [2].

При документации и каталогизации методов рефакторинга принято использовать полуформальную нотацию, в которой каждый из методов описан так называемым *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 [6]. Помимо прочего, паттерны формируют словарь решений для данной проблемной области и позволяют двум специалистам в этой области именовать типовые решения и понимать друг друга, не объясняя каждый раз суть самих решений.

Рефакторинг объектно-ориентированного кода зарекомендовал себя как эффективный способ решения задач эволюции и сопровождения программ. Однако в настоящее время практически не существует исследований, посвященных рефакторингу на более высоком уровне абстракции – уровне архитектуры ПО. Соответственно, вызывает значительный интерес перенос данной методологии на более высокий уровень абстракции. *Общей целью усилий по разработке и стандартизации методики изменения архитектуры, а также инструментальных средств, поддерживающих эту методику, является получение управляемого и предсказуемого процесса преобразования архитектуры*.

2.4. Отличия архитектурного и классического рефакторингов

При переносе методики рефакторинга на уровень архитектуры существует ряд особенностей, которые обуславливают изменение "облика" метода:

Объекты. При переходе от классического рефакторинга к архитектурному меняются объекты, с которыми идет работа. В классическом рефакторинге сущностями, с которыми идет работа, являются такие элементы, как класс, экземпляр класса. Архитектурный рефакторинг применяется к системам и компонентам. В разных типах рефакторинга различаются также и виды связей, возникающих между объектами.

Масштабы изменений. Классический рефакторинг применяется в существенно меньших масштабах – обычно последствия применения отдельного паттерна классического рефакторинга ограничивается несколькими файлами. Паттерны архитектурного рефакторинга применяются к компонентам архитектуры. При проецировании этих паттернов обратно с уровня структурной модели на программный код изменения могут затронуть существенно больший объем существующего кода (подробно о проецировании изменений модели на код – в разделе 2.3.2).

Описание изменений. Методы классического рефакторинга можно проиллюстрировать статическими моделями языка UML и фрагментами кода. Для описания архитектурного рефакторинга представляется более удобным использовать специализированные структурные модели и инструментальные средства поддержки обратной инженерии.

Тестирование. В соответствии с [2] трансформации можно считать корректными, если они не приводят к изменениям поведения программной системы в целом. Наличие достаточно полного набора модульных (unit) тестов позволяет убедиться в корректности трансформаций. Именно это делает модульное тестирование одним из ключевых аспектов классического рефакторинга. Для архитектурного рефакторинга также встает задача автоматической проверки корректности трансформаций. Конечно, наличие модульных тестов повышает уверенность разработчика в корректности проведенных изменений даже при применении архитектурного рефакторинга. Тем не менее, на настоящий момент неизвестно, насколько эффективно модульное тестирование работает с методами рефакторинга архитектуры. Это,

в частности, обуславливается разницей в масштабах изменений при классическом и архитектурном рефакторинге.

2.5. Фазы архитектурного рефакторинга

Необходимо отметить еще одну специфическую черту рефакторинга архитектуры: для достижения промежуточных целей, возникающих в ходе архитектурного рефакторинга, как правило, приходится выполнять более одного шага. Эти шаги относятся к различным фазам решения поставленных архитектурных задач. Можно условно выделить следующие фазы архитектурного рефакторинга: фаза "раскопки" архитектуры, фаза трансформации архитектуры, фаза семантического анализа подсистем и фаза проецирования изменений модели на программный код. Более подробное рассмотрение фаз архитектурного рефакторинга целесообразно начать именно с проецирования изменений модели на программный код.

Проецирование изменений модели на программный код. Как уже говорилось, для описания архитектурного рефакторинга представляется целесообразным использование структурных моделей. Модели извлекаются из программного кода автоматически, и каждому элементу модели соответствует некоторое множество символов исходного кода программной системы. Таким образом, редактирование модели, обусловленное применением шагов рефакторинга архитектуры, может (и зачастую должно) быть спроецировано на реальный программный код системы. Действительно, при проецировании удаления блоков из модели необходимо определить множество строк и файлов, которое соответствует удаленному блоку в программном коде. После этого необходимо удалить из программного проекта выявленные строки и файлы. При проецировании на код переноса блока в модели переносятся соответствующие строки и файлы в исходном коде программной системы и т.д.

Проецирование на программный код системы шагов, выполненных в ходе некоторого архитектурного рефакторинга, 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*.

"Раскопка" архитектуры. Шаги, относящиеся к этой фазе, характеризуются тем, что соответствующие действия, применяемые к модели, не ориентированы на последующее проецирование на программный код. Они нужны только для понимания и структуризации модели.

Трансформация архитектуры. Для шагов, относящихся к трансформации архитектуры, в отличие от шагов фазы раскопки, типично последующее проецирование их на реальный код программной системы. Шаги этой фазы четко связаны с реальной модификацией кода системы и, в конечном счете, ориентированы на его улучшение. Следует также отметить, что часть методов рефакторинга архитектуры 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 принимает разработчик, руководствуясь поставленной задачей.

Семантический анализ подсистем. Как правило, между шагами описанных выше фаз архитектурного рефакторинга предпринимаются шаги, которые можно отнести к фазе семантического анализа подсистем: по ходу трансформаций часто встает задача выявления смысловой нагрузки подсистем. Для решения подобных задач, 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, путем удаления вспомогательных блоков) и сделать его последовательным и направленным.

3. Выделение слоев

Имеющийся опыт рефакторинга архитектуры программных систем позволил выделить несколько групп приемов, используемых наиболее часто. Эти приемы переходят из одного процесса рефакторинга архитектуры в другой с небольшими модификациями или даже вовсе без них. Естественно, повторяющиеся решения заслуживают выделения в самостоятельные паттерны или группы паттернов. Пожалуй, одним из важнейших и наиболее употребимых паттернов рефакторинга архитектуры ПО является паттерн "выделение слоев", рассматриваемый в этой статье.

Значение паттерна "выделение слоев" легко объяснимо – слои обладают целым рядом замечательных свойств, делающих их незаменимым инструментом структуризации системы.

3.1. Слои в архитектуре ПО

Концепция слоев — одна из общеупотребительных моделей, используемых разработчиками программного обеспечения для разделения сложных систем на более простые части. В архитектурах компьютерных систем, например, различают слои кода на языке программирования, функций операционной системы, драйверов устройств, наборов инструкций центрального процессора и внутренней логики микросхем. В среде сетевого взаимодействия протокол FTP работает на основе протокола TCP, который, в свою очередь, функционирует "поверх" протокола IP, расположенного "над" протоколом Ethernet [7]. Итак, рассмотрим основные причины интереса к слоям архитектуры программных систем:

Слои легко формализуются. Интуитивно понятно, что если система разбита на ряд слоев, то слой n – это компонент или набор компонентов системы, которые используют только компоненты слоя $n-1$ и могут быть использованы только компонентами слоя $n+1$.

Слои обладают простой и наглядной семантикой. Как правило, в архитектуре программной системы слои представляют уровни абстракции. Слой *n+1* использует слой *n*, следовательно, абстракция понятий слоя *n+1*, по меньшей мере, не ниже, чем у слоя *n*, а в идеале – если архитектура системы эффективна, его уровень абстракции должен быть выше. Соответственно, слой *n* скрывает (инкапсулирует) логику работы с понятиями, определенными на этом слое, позволяя, таким образом, слою *n+1* реализовать работу с более сложными понятиями, организовать более сложную логику, используя выразительные средства нижележащего слоя.

Слои широко распространены. Действительно, достаточно большое количество программных систем, особенно если речь идет о программных системах масштаба предприятия (enterprise systems), имеют именно слоистую структуру. Конечно, достаточно часто встречается ситуация, когда строгая послойная структура системы нарушается – как правило, это является следствием эрозии архитектуры (архитектурным дефектом) и ее устранение в большинстве случаев способно принести ощутимые выгоды (эти аспекты рассматриваются далее).

Альтернативная реализация. Можно выбирать альтернативную реализацию базовых слоев – компоненты верхнего слоя способны работать без каких-либо изменений в нижележащих слоях [7], при условии сохранения интерфейсов.

Минимизация интерфейсов. Зависимость между слоями, то есть, фактически, интерфейсы, предоставляемые нижними слоями верхним, можно свести к минимуму [7]. Такая минимизация интерфейсов позволяет увеличивать гибкость системы.

Схема архитектурных слоев обладает и определенными недостатками [7]:

Каскадные изменения. Слои способны удачно инкапсулировать многое, но не все: модификация одного слоя подчас связана с необходимостью внесения каскадных изменений в остальные слои. Классический пример из области корпоративных программных приложений: поле, добавленное в таблицу базы данных, подлежит воспроизведению в графическом интерфейсе и должно найти соответствующее отображение в каждом промежуточном слое.

Падение производительности. Наличие избыточных слоев нередко снижает производительность системы. При переходе от слоя к слою данные обычно подвергаются преобразованиям из одного представления в другое. Несмотря на это, инкапсуляция нижележащих функций зачастую позволяет достичь весьма существенного преимущества. Например, оптимизация слоя транзакций обычно приводит к повышению производительности всех вышележащих слоев.

3.2. Паттерн

В этом разделе рассматривается основной паттерн выделения слоев – это всего лишь один из вариантов, представитель целого семейства паттернов выделения слоев. Подвидов у этого паттерна существует достаточно много, и каждый из

них обладает своей спецификой. Эти подвиды и их особенности будут рассматриваться далее.

Имя: выделение слоев.

Ситуация: на диаграмме представлены элементы, для которых верны следующие условия:

- Исходящие связи ведут только в последний выделенный слой (слой с номером *n*), или их нет, если ранее не был выделен ни один слой.
- "Кандидаты на объединение в новый слой" с номером *n+1* 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 "*n+1*". Отметим, что для двух произвольных слоев слой, обладающий большим порядковым номером, считается "вышележащим". Важно отметить, что если в результате применения паттерна было выделено *n* слоев и еще остались блоки, которые в силу ограничений не смогли быть отнесены ни к одному из выделенных слоев и формально не могут быть выделены в новый слой, то эти блоки по умолчанию считаются *n+1* слоем, который в дальнейшем именуется "чердаком".

3.3. Виды слоев

Строгие слои. Это слои, описанные в разделе 2.1. Канонический вариант слоев, не допускающий никаких отклонений в строгой структуре и потому встречающийся относительно редко. На рис. 2 А) могут быть выделены следующие строгие слои: слой 1 – блоки с номерами 5, 6. Слой 2 – блоки 3, 4. Слой 3 – блоки 1, 2.

Как уже упоминалось, систем с четкой послойной структурой, также известной как строгие слои, значительно меньше, чем систем с *псевдослоями* – то есть со структурой, близкой к послойной, но с допустимыми отклонениями. Далее рассматриваются некоторые виды послойных структур, используемых для рефакторинга архитектуры ПО.

Нестрогие слои (Non-strict layers). Смягчение условий: нестрогие слои допускают связи от вышележащего слоя к нескольким нижележащим слоям (потенциально – ко всем), а не только к непосредственному соседу снизу. Архитектура с нестрогими слоями может быть как результатом эрозии, так и осознанным решением.

Последствия: возможным дефектом архитектуры с нестрогими слоями является *нарушение абстракции*. Подобные дефекты затрудняют анализ системы. Кроме того, изменения слоя в такой архитектуре значительно сложнее локализовать – волна изменений прокатится по всем слоям, работающим с изменяемым слоем, то есть, в пессимистическом случае, по всем вышележащим слоям.

На рис. 2 Б) показаны шесть нестрогих слоев – по одному блоку в каждом слое: слой 1 – блок 6, слой 2 – блок 5, слой 3 – блок 4 и т.д. В то же время, выделение строгих слоев даст значительно более грубую структуризацию: слой 1 – блок 6, слой 2 – блок 5. Все остальные блоки попадут на "чердак", так как дальнейшее выделение строгих слоев невозможно – блок 4 нарушает строгую структуру и обращается к первому слою в обход второго слоя. Все оставшиеся блоки используют блок 4 и, соответственно, тоже попадают на "чердак".

Поглощение сильносвязанных компонентов (СК). Смягчение условий: уже рассмотренные виды слоев можно модифицировать, позволив включать в произвольный слой СК [8]. При таком подходе СК рассматривается, фактически, как атомарный элемент. Без подобного смягчения условий как сам СК, так и все блоки, которые могли бы попасть в вышележащие слои, будут отправ-лены на "чердак". Действительно, рассмотренные подходы не смогут расщепить СК на слои, и он автоматически попадает на чердак, так же, как и блоки, зависящие от него. Тем не менее, отнюдь не всегда СК на структурных диаграммах свидетельствуют о плохой архитектуре системы. Например, в системах с рассылкой сообщений достаточно часто встречается сильно связанный компонент, подобный представленному на рис 1: источник событий "знает" о зарегистрированных в нем слушателях (listeners), слушатели знают о событиях (events), отправляемых источником, а в событии содержится ссылка на его источник. Таким образом, представляется удобным выделять слои и на тех диаграммах, которые содержат СК, именно поэтому было введено *поглощение*.

Последствия: возможным дефектом архитектуры с поглощающими слоями может стать эффект "пропавшего слоя" – дефектная связь приводит к появлению СК, которые по смыслу должны находится на разных слоях.

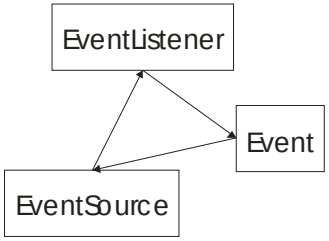


Рис 1. События, слушатели и источники событий.

На рис. 2 В) при использовании поглощения СК можно выделить те же слои, что и в случае применения строгих слоев к 2 А). Слой 1 – блоки 5, 6. Слой 2 – блоки 3, 4. Слой 3 – блоки 1, 2. Если применять обычный поиск строгих слоев, то все блоки окажутся на "чердаке".

На рис. 2 Г) при использовании поглощения СК можно выделить слой 1 – блок 6, слой 2 – блок 5, слой 3 – блоки 3, 4, слой 4 – блоки 1, 2. Если применять

обычный поиск нестрогих слоев, блоки 1, 2, 3, 4 окажутся на "чердаке", то есть структуризация в таком случае будет значительно хуже.

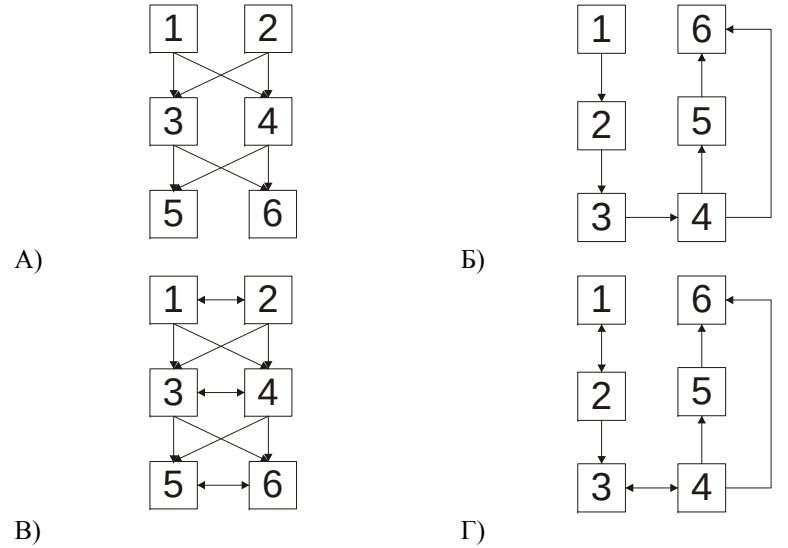


Рис.2. Примеры диаграмм.

3.4. Применение паттерна выделения слоев для анализа и улучшения архитектуры

3.4.1. Место паттерна выделения слоев в рефакторинге архитектуры

Как уже говорилось, рефакторинг архитектуры содержит шаги, которые можно сгруппировать по их назначению. Выделение слоев примечательно тем, что в разных контекстах может быть отнесено к разным фазам в рамках рефакторинга архитектуры.

Значение паттерна для анализа. Паттерн выделения слоев может быть применен для чистого анализа (раскопки) архитектуры. Для этого имеется несколько оснований. Выделение слоев – это прием, который в известной степени позволяет сократить и сделать более направленным семантический анализ системы за счет структурного анализа, что, несомненно, хорошо, поскольку семантический анализ – значительно более ресурсоемкий процесс. При этом нет никакой необходимости отражения слоев на программном коде и инфраструктуре его хранения. Например, при анализе архитектурного кода программной системы, написанной на Java, выделение слоев не предполагает обязательного выделения пакетов, соответствующих этим слоям. Если специалист, проводящий архитектурный рефакторинг, принимает решение все-

таки не отображать слои в пакеты языка Java, то можно считать, что выделение слоев было применено для чистого анализа архитектуры.

Значение паттерна выделения слоев для улучшения системы. Выделение слоев – хорошая основа для улучшения системы. Как уже упоминалось, найти строгие слои в произвольной программной системе значительно труднее, чем найти слои с некоторыми допустимыми отклонениями, как, например, такие, которые были описаны в предыдущем разделе. Тем не менее, для каждого из допустимых отклонений известны побочные эффекты, которые можно устранять, по возможности приводя слои к строгим.

3.4.2. Примеры применения паттерна выделения слоев

Улучшение архитектуры и нестрогие слои. В работе [9] описывается случай, когда найденные нестрогие слои позволили выявить архитектурный дефект в программной системе toolbus, существенно затруднявший реализацию задачи, поставленной перед разработчиками. Программная система toolbus представляет для своих клиентов механизмы коммуникации. В системе используется набор стандартных сообщений, которые были реализованы на основе сокетов операционной системы и специализированного текстового протокола для обмена данными по сокетам (на рис 3 – за это отвечает слой “Layer 2: Protocol Implementation”). Чтобы изолировать тонкости текстового протокола от разработчиков, был реализован API системы, скрывающий как ее сокеты, так и передаваемые через них коды команд за набором процедур языка программирования (Layer 3: Toolbus API). Однако по мере развития в систему были добавлены вспомогательные механизмы (“Layer 4: Applications”). Именно эти механизмы и нарушали строгость слоистой структуры, поскольку они обращались к уровню работы сокетов в обход специфицированного API системы (связь от “Layer 4” к “Layer 3”).

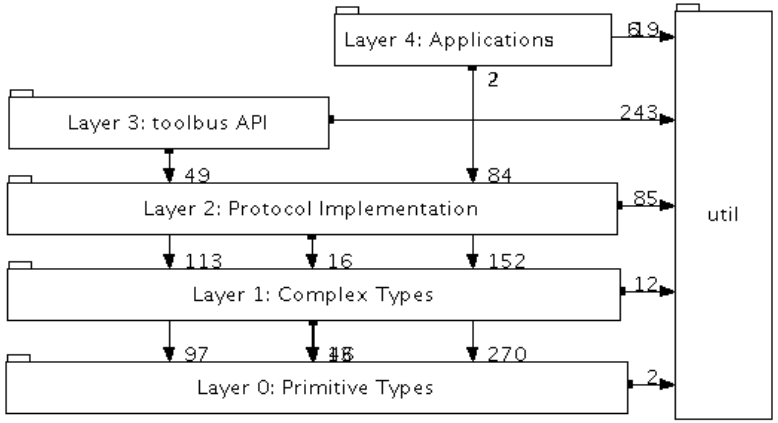


Рис.3 Архитектура toolbus.

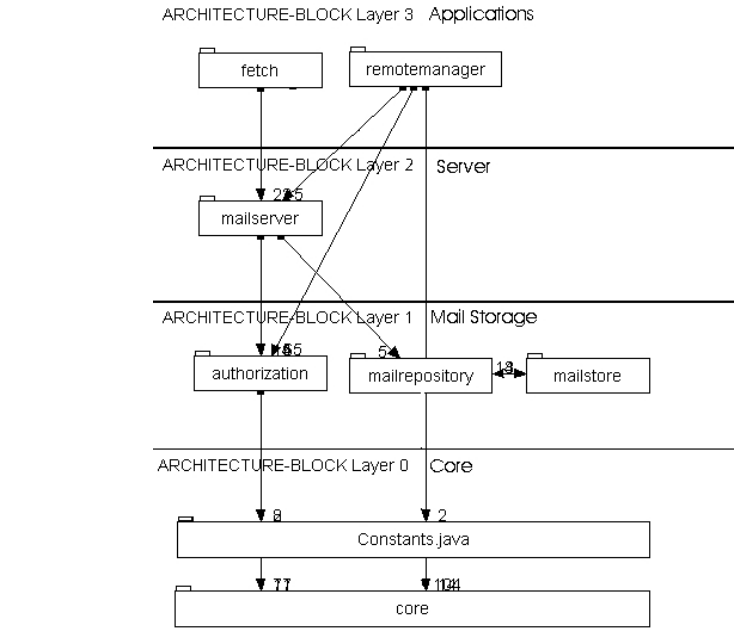
Поскольку перед разработчиками стояла задача смены текстового протокола, используемого в системе для обмена по сокетам, подобный дефект грозил переделкой всех стандартных механизмов системы. Архитектурный рефакторинг позволил избежать столь неприятной ситуации и впоследствии значительно сократить время, затрачиваемое непосредственно на кодирование.

Анализ архитектуры и поглощающие слои. Характерным примером, демонстрирующим пользу смягчения условий для поиска слоев с целью анализа архитектуры, является ситуация, возникшая при анализе архитектуры Apache James 2.2. Apache James – это реализованный на Java мощный почтовый сервер масштаба предприятия, поддерживающий все наиболее распространенные современные почтовые протоколы и предоставляющий ряд дополнительных возможностей. После того, как для сервера Apache James была построена структурная модель и в ней были выделены основные подсистемы, была предпринята попытка выделить в модели нестрогие слои.

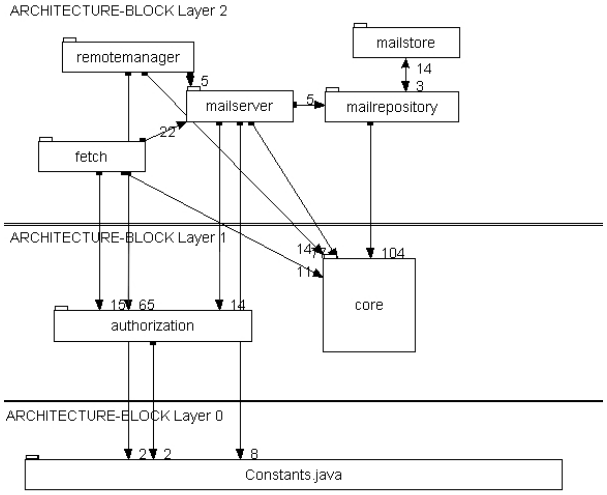
При выделении слоев было разрешено поглощение СК. В результате была получена диаграмма 4 А). Эта диаграмма вполне осмыслена: на слое 1 оказались блок Constants.java, содержащий константы, используемые во всей системе, и блок Core, содержащий основные типы данных системы, такие как Message – сообщение, MessageHeader – заголовок сообщения и пр. Таким образом, слой 1 содержит основные определения системы. Слой 2 содержит authorization – систему контроля доступа, mailrepository – репозиторий писем, mailstore – адаптер для mailrepository, созданный для унификации работы с различными системами хранения и управления сообщениями, в том числе он может служить для того, чтобы скрывать и mailrepository. Слой 3 – фактически, основной, поскольку содержит собственно ядро почтового сервера. На слое 4 располагаются вспомогательные механизмы для работы с почтовым сервером – система удаленного администрирования remotemanager и система fetch – облегчающая получение писем с сервера.

С другой стороны, попытка выделения нестрогих слоев, но без поглощения СК дала гораздо более слабый результат. Выделенные слои представлены на рис. 4 Б). Из-за того, что в системе существует сильносвязанный компонент, в который входят блоки mailrepository и mailstore, на "чердак" попало большинство слоев системы, а дать слоям какое-либо внятное определение оказалось невозможно.

Необходимо отметить, что СК, содержащий mailrepository и mailstore, свидетельствует о некоторой архитектурной проблеме. Как уже говорилось, mailstore – выступает как адаптер для mailrepository. Связь, идущая от mailrepository к адаптеру, свидетельствует о том, что изменение адаптера приведет к изменению адаптируемого объекта, что снижает гибкость архитектуры программной системы.



А)



Б)

Рис. 4. Анализ Apache James.

3.5. Тактика применения

Тактика применения паттерна выделения слоев имеет два основных аспекта: масштаб и время.

Масштаб. Выделение слоев можно применять на разных масштабах и уровнях абстракции. Важно понимать, что каков бы ни был уровень абстракции, на котором идет рефакторинг, всегда можно попытаться выделить слои и иметь достаточно хорошие шансы на успех; при этом не так важно, идет ли моделирование в терминах пакетов, компонентов или даже файлов.

Более того, слои облегчают изменение масштабов абстракции, используемые для моделирования. Если в системе найдены слои, то изменение масштабов можно производить и вручную – несколько соседних слоев достаточно легко объединяются в один слой, происходит укрупнение подсистем, с которыми идет работа. Понижение масштаба может быть достигнуто за счет разбиения одного слоя на несколько подслоев. Следует заметить, что это не всегда возможно, в отличие от объединения слоев, особенно без предварительной подготовки и вспомогательного рефакторинга систем разбиваемого уровня.

Время. Этот аспект касается того, когда нужно применять выделение слоев. Это достаточно хороший паттерн, чтобы с него начать анализ и рефакторинг архитектуры системы, особенно если эрозия архитектуры не катастрофична. Выделение слоев ведет к тому, что рефакторинг архитектуры приобретает "направленность", задача сводится к меньшим подзадачам; теперь детальный семантический анализ можно применять только к тем слоям, которые имеют непосредственное отношение к поставленной задаче. Тем не менее, как и всякий другой паттерн, выделение слоев особенно хорошо работает в связке с другими паттернами. Шансы на успех резко вырастут, если провести предварительную подготовку, применив как минимум "удаление мертвых блоков" и "инкапсуляцию приватных блоков", описанных в [9].

4. Заключение

Выделение слоев является одним из ключевых паттернов для анализа и улучшения архитектуры. Слои позволяют упорядочивать архитектурные модели и, следовательно, упрощать их анализ. Кроме того, существуют различные разновидности слоев и различные тактики применения этого паттерна. С архитектурной точки зрения, с каждой из разновидностей слоев связаны определенные плюсы и минусы, что позволяет быстро находить и устранять различные архитектурные дефекты.

В заключение необходимо отметить, что поиск блоков, которые могут быть выделены в слой на больших диаграммах, представляется затруднительной задачей. Тем не менее, решение такой задачи достаточно легко автоматизировать. Для автоматического поиска кандидатов на объединение в новый слой достаточно перебрать блоки исследуемой структурной модели и проанализировать направления их связей. В частности, при создании примеров

для этой статьи использовались подобные средства автоматического поиска слоев, реализованные как плагины к KLOCwork Architect.

Литература:

1. Research Issues in The Renovation of Legacy Systems, A. van Deursen, P. Klint, C. Verhoef, CWI research report P9902, April 1999.
2. Рефакторинг: Улучшение существующего кода, Мартин Фаулер, Символ, Санкт-Петербург, 2003.
3. Recommended Practice for Architectural Description of Software-Intensive Systems, ANSI/IEEE Std 1471-2000.
4. Insight: reverse engineer case tool, N. Rajala, D. Campara, N. Mansurov, IEEE Computer Society Press, 1999, Los Alamitos, CA, USA Pages: 630 – 633.
5. Managed Achitecture of Existing Codeas a Practical Transition towards MDA, N. Mansurov, D. Campara, Klocwork, 1 Chrysalis Wat, Ottawa, Canada, K2G6P9, mansurov@klocwork.com
6. A Pattern Language, C. Alexander, S. Ishikawa, M. Silverstain, M. Jakobson, I. Fiksdahl-King and S. Angel, Oxford University Press, 1977, New York.
7. Архитектура корпоративных программных приложений, Мартин Фаулер, Вильямс, Москва, 2004.
8. Алгоритмы: построение и анализ, Т. Кормен, Ч. Лейзерсон, Р. Ривест, МЦНМО, Москва, 2001.
9. Рефакторинг архитектуры программного обеспечения, М.В.Ксензов, ИСП РАН, Препринт 4, Москва, 2004.