Аспектно-ориентированное программирование

Анализ вариантов применения аспектно-ориентированного подхода при разработке программных систем.

Валентин Павлов

Код к статье.

1. Аннотация

В статье рассматриваются вопросы применения новой парадигмы программирования — аспектно-ориентированного подхода, основным свойством которой является модуляризация сквозных требований на разных уровнях абстракции и их локализация в отдельных программных модулях — аспектах.

В процессе исследования проведен анализ существующих подходов к разработке, выделены проблемы, возникающие при наличии сквозных общесистемных требований, а также рассмотрена проблема роста сложности проектируемых программных систем. В статье дается представление об аспектно-ориентированном программировании, аспектной декомпозиции и отличии этого подхода от традиционного программирования, использующего только функциональную декомпозицию. Кроме того, показаны преимущества и недостатки новой методологии.

В работе приведены примеры применения аспектного подхода на разных этапах жизненного цикла программных систем. Приведенные примеры позволяют отметить основное достоинство применения аспектно-ориентированного подхода: улучшение модульности, что выражается в локализации сквозных требований в специальных программных единицах — аспектах, в упрощении сопровождения программной

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

системы и внесении изменений, а также появлении новых возможностей повторного использования кода.

Возможность применения аспектного подхода показана для задач самого разного характера. Оценка применимости АОП сделана в рамках выбранных критериев сравнения ОО и АО реализаций предлагаемых вариантов применения новой методологии.

2. Благодарности

Данная статья является мастер-тезисом который защищиался в июне 2003 года в Санкт-Петербургском электротехническом университете (ЛЭТИ). В написании этого труда неоценимую помощь мне оказали мои руководители Журавлев Е.А. и Кирьянчиков В.А. Отдельное спасибо Изьюрову А.Л. за "жизненные примеры" применения новой парадигмы программирования.

3. Введение

Данная работа посвящена исследованию вариантов применения новой методологии — аспектно-ориентированного подхода при разработке программных систем.

3.1. Актуальность

Аспектно-ориентированное программирование (АОП) представляет собой одну из концепций программирования, которая является дальнейшим развитием процедурного и объектно-ориентированного программирования (ООП). Данная методология призвана снизить время, стоимость и сложность разработки современного ПО, в котором, как привило, можно выделить определенные части, или аспекты, отвечающие за ту или иную функциональность, реализация которой рассредоточена по коду программы, но состоит из схожих кусков кода. По оценкам специалистов [24], около 70% времени в проектах тратится на сопровождение и внесение изменений в готовый программный код. Поэтому достаточно важной в ближайшей перспективе становится роль АОП и подобных трансформационных подходов. Сравнительно новая технология уже получила довольно широкое распространение показав свою эффективность на тестовых приложениях, однако место этого подхода в индустрии ПО по ряду

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

объективных причин все еще не определено.

3.2. Объект исследования

Существенная черта программной системы — уровень сложности: один разработчик практически не в состоянии охватить все детали системы, причем сложность присуща большинству современных программных систем. Данная сложность неизбежна: с ней можно справиться, но избавиться от нее нельзя. Сложность программных систем обусловлена четырьмя основными причинами: сложностью реальной предметной области, из которой исходит заказ на разработку; трудностью управления процессом разработки; необходимостью обеспечить достаточную гибкость программы; неудовлетворительными способами описания поведения больших дискретных систем [9].

Объектом исследования данной работы является аспектно-ориентированный подход при разработке программных систем. АОП предлагает языковые средства, позволяющие выделять сквозную функциональность в отдельные модули, и таким образом упрощать работу (отладку, модифицирование, документирование и т.д.) с компонентами программной системы, и снижать сложность системы в целом. Здесь и далее под "модулем" понимается некоторая четко выраженная структурная единица программы — процедура, функция, метод, класс или пакет. Программный модуль в терминах некоторой рассматриваемой парадигмы программирования (например, ООП) назовем компонентом.

Цель исследования заключается в анализе возможных вариантов применения нового подхода при решении проблемы растущей сложности программных продуктов на разных этапах жизненного цикла, сравнение реализаций предложенных вариантов с традиционной объектно-ориентированной реализацией, а также просмотре дальнейших перспектив в развитии новой методологии.

Для решения поставленной задачи, необходимо осуществить анализ существующих подходов при разработке программных систем, описать проблемы которые появляются при использовании функциональной декомпозиции при наличии сквозных системных требований, сделать обзор аспектной методологии и инструментов поддерживающих ее. Ввести критерии сравнения ОО и АО реализаций, а также представить варианты применения аспектно-ориентированного подхода на примерах. Для каждого примера

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

приведена реализация на языке Java, ее аспектно-ориентированный аналог и их сравнительный анализ.

3.3. Научная новизна и практическая значимость

Появление новых парадигм программирования всегда являлось интересной темой, поскольку каждая новая методология разработки программного обеспечения позволяла решать проблемы, являющиеся посылками для ее появления, и значительно продвигала науку Computer Science и индустрию ПО в целом. Усложнение программных систем — как глобальная проблема требует постоянного внимания и изучения, поэтому появление АОП представляется широким полем для исследований с последующим их практическим применением.

В работе предлагаются пути использования аспектного подхода на разных этапах жизненного цикла ПО. Кроме того, в работе приводится описание АОП, дается обзор существующих подходов при разработке программных систем и предпринимается попытка локализации преимуществ и, главное, недостатков новой методологии.

Данная работа позиционируется как одна из первых работ на русском языке посвященных обзору принципов аспектного подхода и тематике применения аспектного программирования.

4. Работы в области аспектно-ориентированного програмирования

Основные идеи АОП были сформулированы наиболее явной форме в одной из ранних статей [1] идеологом методологии Грегором Кикжалесом (Gregor Kiczales). На сервере поддержки и развития АОП [22] можно получить актуальную информацию о развитии данной методологии, а также узнать о событиях и конференция проходящих по этой тематике. Хороший обзор по АОП представлен в диссертации [18].

АОП — методология, в основе которой лежат идеи, встречающиеся в области технологии программирования уже достаточно давно. Это субъектно-ориентированное программирование (subject-oriented programming) [25], композиционные фильтры (composition filters) [26], [14], адаптивное программирование (adaptive programming)

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

[15]. АОП тесно связано с ментальным программированием (intentional programming), концепции которого изложены в работе Чарльза Саймони [23]. Другой близкой идеологией является так называемое порождающее, или трансформационное, программирование (generative programming, transformational programming) [16].

В [2] авторы предлагают краткое введение в проблематику АОП, представлены ключевые понятия АОП и рассмотрены способы интеграции аспектов, в том числе и на этапе выполнения программы. В [3] отмечаются преимущества АОП-подхода, недостатки существующих реализаций описания точек связывания аспектов и функциональных модулей. Предложена модификация модели AspectJ для задания точек связывания аспектов.

В статье [21] автор дает основные понятия АОП, вводит термин *сквозная функциональность*, а также предоставляет краткое описание языка AspectJ.

Поскольку все примеры разрабатывались с использованием наиболее популярной реализации AspectJ, то основная литература, изучаемая по реализациям АОП, — это литература касающаяся этого языка. Руководство пользователя [17] является основным документом, описывающим данное расширение языка Java. В [4] автор детально рассматривает язык AspectJ, эта книга использовалась в качестве альтернативного источника информации. Кроме того, из источника [22] можно получить информацию обо всех поддерживаемых на данный момент расширениях языков и других инструментах поддерживающих АОП. Например, в статье [19] описана связь между АОП и .NET

В классическом труде [6] описываются простые и изящные решения типичных задач, возникающих в объектно-ориентированном проектировании. Авторы излагают принципы использования шаблонов проектирования и приводят их каталог. В [7] описаны эффективные методы применения всех типов шаблонов проектирования применительно к платформе Java. В [5] и [13] авторы показывают применимость аспектного подхода для реализации протокола взаимодействия объектов при реализации шаблонов проектирования. В этих работах они рассматривают улучшения в терминах модульности, повышения степени повторного использования и улучшенного восприятия исходного кода классов участников шаблонов из каталога [6].

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

В [11] дается понятие "контрактного проектирования" (Design by Contract) — техники способствующей созданию надежного объектно-ориентированного программного обеспечения. В [12] рассматриваются общие подходы при обработке ошибок на этапе разработки ПО, даются советы по использованию АОП и делаются выводы о применимости нового подхода на данном этапе жизненного цикла.

5. Существующие подходы к разработке программных систем

5.1. Эволюция методологий разработки ПО

В начале эры вычислительной техники программы разрабатывались посредством прямого кодирования на машинном языке. Программисты тратили большое количество времени, раздумывая над особенностями использования того или иного набора машинных инструкций, чем непосредственно над стоящей перед ними задачей. Затем перешли к языкам более высокого уровня, которые позволяли абстрагироваться от машинного уровня. Потом пришла эра структурированных языков, теперь программисты могли проводить структурную декомпозицию проблем в терминах процедур, необходимых для выполнения той или иной задачи. Тем не менее, с ростом сложности программного обеспечения возникла потребность в другой технологии. Объектно-ориентированное программирование позволило представить систему как множество взаимодействующих объектов. Классы позволили скрыть детали реализации за интерфейсами. Механизм полиморфизма обеспечил общее поведение и интерфейс связанных концепций и позволил управлять поведением компонентов без доступа к реализации базовых концепций.

Методологии разработки ПО и языки программирования определили путь взаимодействия человека с компьютером. Каждая новая методология представляет новые пути декомпозиции проблем: машинный код, машинно-независимый код, процедуры, классы и т.д. Каждая новая методология позволяла все более естественно отобразить требования к продукту на программные конструкции. Эволюция программных методологий позволила создавать более сложные системы.

В настоящее время объектно-ориентированное программирование (ООП) является

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

методологией, в пользу которой делается выбор большинством новых проектов разработки программных продуктов. Несомненно, методология ООП продемонстрировала свою силу при моделировании общего поведения разрабатываемой системы. Однако, как можно убедиться из опыта, ООП не в достаточной мере позволяет справляться с растущей сложностью программных систем.

5.2. Система как набор функциональных требований

Как правило, любая программная система состоит из нескольких частей: основной (предметно-ориентированной) и системной части, которые несут в себе требуемую функциональность. Например, ядро системы обработки кредитных карт предназначено для работы с платежами, тогда как функциональность системного уровня предназначена для ведения журнала событий, целостности транзакций, авторизации, безопасности, производительности и т.д. Большинство подобных частей системы, известны как *сквозная функциональность* [2], затрагивают множество основных предметно-ориентированных модулей. Можно рассматривать сложную программную систему как комбинацию модулей, каждый из которых включает в себя кроме бизнес-логики часть сквозной функциональности из набора требований к системе. Рисунок 1 показывает систему как набор таких требований разбитых на разные модули.

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

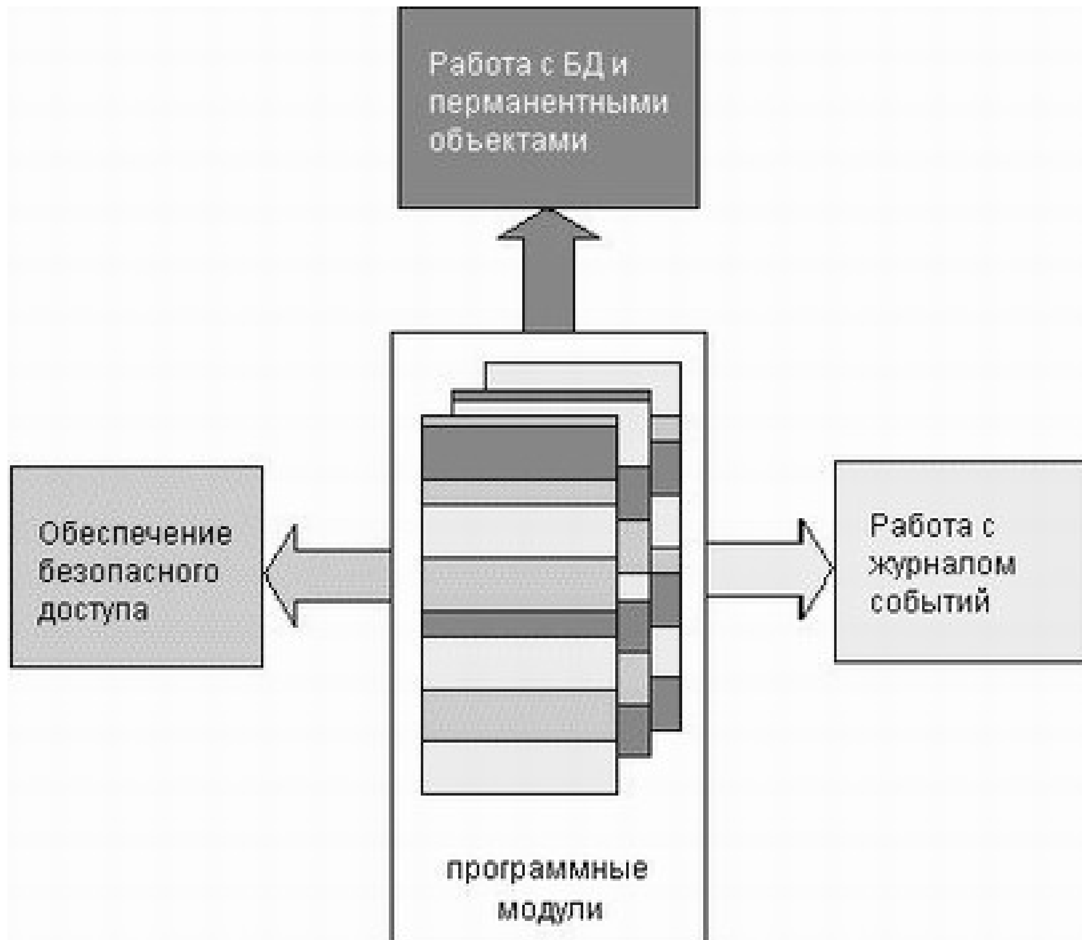


Рисунок 1. Система как набор функциональных модулей.

При разработке программной системы с использованием существующих методологий программирования сквозная функциональность будет включена во все модули, в результате система будет сложной при проектировании, понимании, реализации и поддержке.

Рисунок 2 представляет набор требований как световой луч, проходящий через призму. После прохождения луча через призму способную разделять требования, требования раскладываются в спектр по функциональности.

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

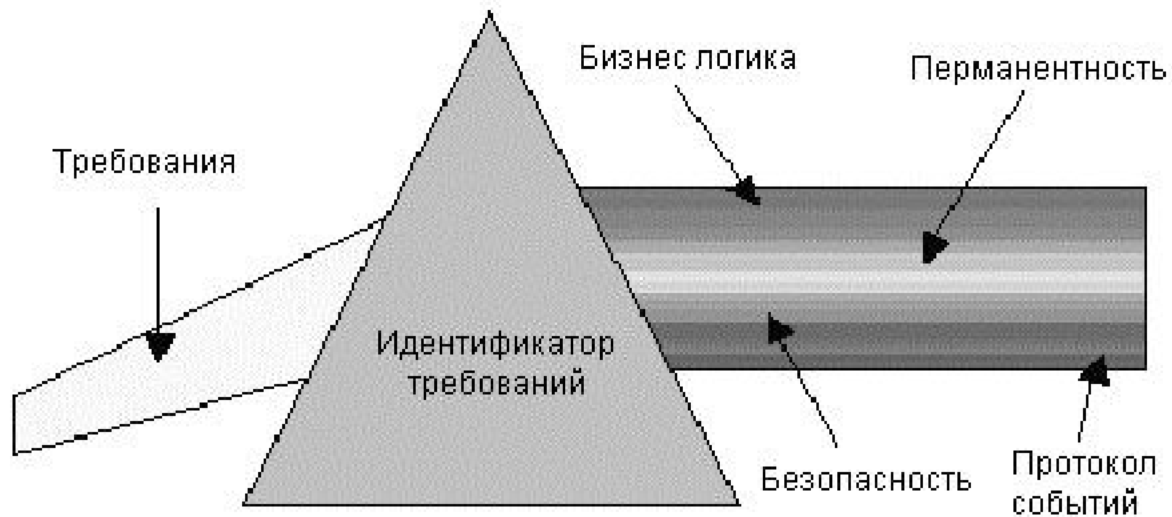


Рисунок 2. Декомпозиция требований: выделение функциональности.

Разработчик создает программную систему как результат обработки множества требований. Можно явно выделить из этого множества требования к логике конкретного модуля и общесистемные требования. Многие из системных требований могут быть ортогональными друг другу и требованиям конкретного модуля. Требования системного уровня имеют тенденцию пересекаться с множеством основных требований. Например, типичная система уровня предприятия включает в себя такие виды сквозной функциональности как аутентификация, ведение журнала событий, управление ресурсным пулом, администрирование, анализ производительности и управление носителями информации. Каждое из этих требований к системе затрагивает множество подсистем, например, требование по управлению носителями информации затрагивает каждый сохраняемый бизнес-объект.

Современные технологии разработки ПО на уровне языков программирования предоставляют удобные средства для выделения логики функционирования программы в отдельные модули, но не одна из них не предлагает удобного способа локализации в отдельные модули функциональности, которая должна распространяться на всю систему.

Из-за того, что реализация сквозной функциональности не может быть обособлена средствами языка программирования в отдельном программном модуле, элементы этой реализации присутствуют в том или ином виде в большинстве модулей,

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

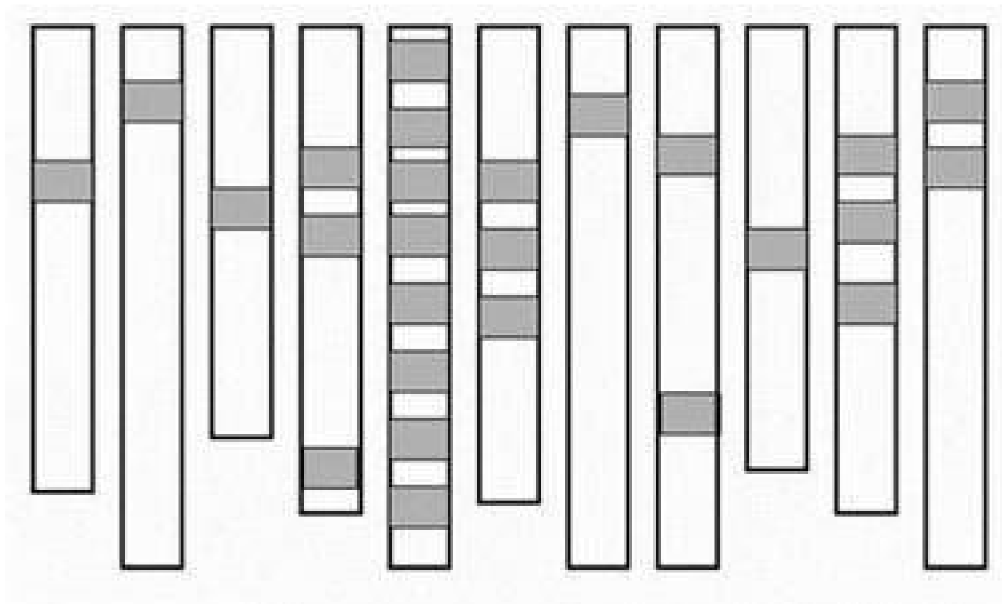
образующих программную систему. Рассмотрим пример java-класса, основной задачей которого является реализация некоторой логики:

```
public class SomeBusinessClass {
        // Бизнес-данные
        // вспомогательные данные
     public void performSomeOperation(OperationInformation info) {
                // проверить уровень доступа к данным
                // проверить соответствие входных данных контракту
                // запретить доступ к данным другим потокам выполнения
                // проверить состояние кэша данных
                // занести в журнал отметку о начале операции
                // ==== Реализация логики данного класса ====
         // занести в журнал отметку о конце операции
         // разрешить доступ к данным другим потокам выполнения
     }
}
```

В этом примере можно выделить две проблемы. Во-первых, определяемые вспомогательные данные не относятся к требованиям, накладываемым на данный модуль, а требуются для работы сквозной функциональности. Во-вторых, реализация performSomeOperation(..) выглядит более нагруженной, чем просто "Реализация логики данного класса". Для правильной работы данного метода по требованиям необходимо выполнить ряд действий, не относящихся конкретно к данному модулю — проверить уровень доступа к данным, проверить соответствие входных данных контракту, запретить доступ к данным другим потокам выполнения, проверить состояние кэша данных, занести в журнал отметку — это требования системного уровня. К тому же, многие из этих общих требований должны быть реализованы в других модулях.

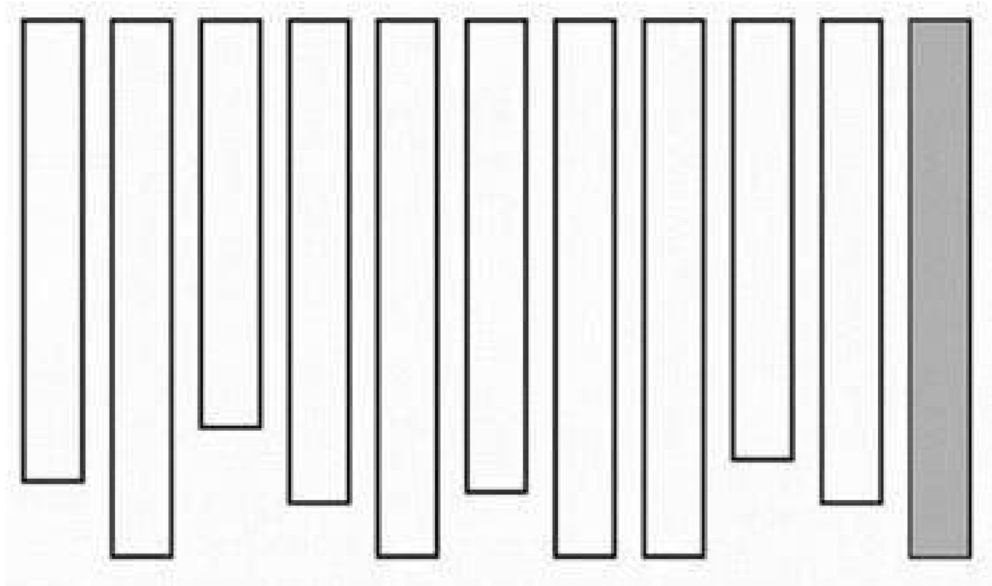
На рисунке 3 схематично изображено распределение функциональности "ведение журнала событий" по модулям некоторой программной системы. В каждом модуле отмечен код, реализующий требование "ведение журнала событий".

Copyright © 1999 - 2007, IT-archiv. All rights reserved.



Ведение журнала событий — типичный пример сквозной функциональности, аспекта функционирования системы, реализация которого присутствует во многих программных модулях. При анализе требований к системе можно выделить и другие аспекты функционирования подобного типа. Данные аспекты функционирования системы объединяет то, что их реализация не может быть локализована имеющимися средствами языка программирования в отдельном программном модуле (или ряде модулей). Схематичное распределение функциональности "ведение журнала событий" в программной системе в некотором "идеальном" варианте представлено на рисунке 4.

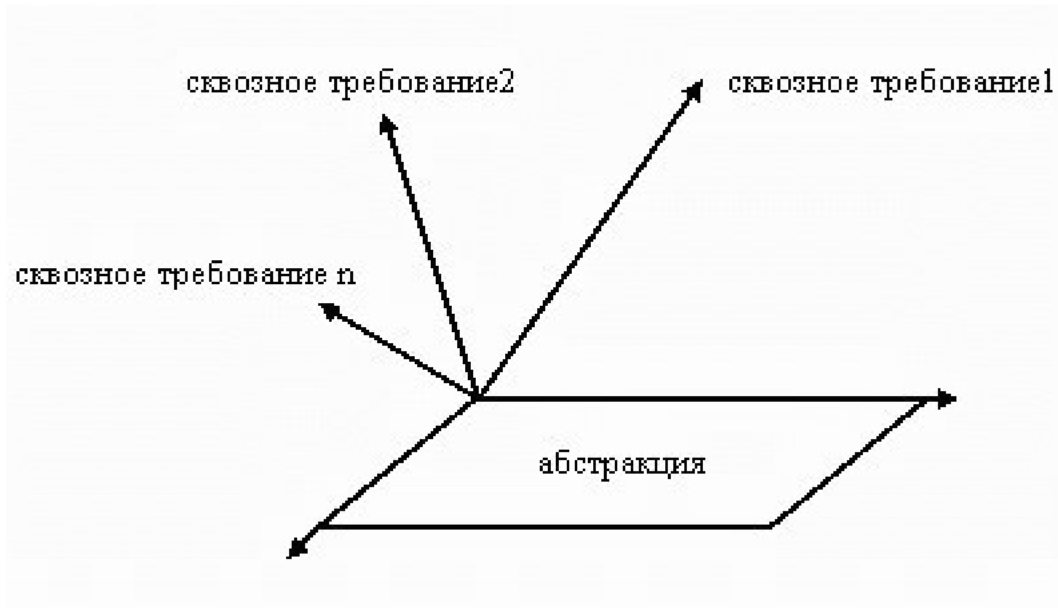
Copyright © 1999 - 2007, IT-archiv. All rights reserved.



5.3. Сквозная функциональность в системе

Несмотря на то, что сквозная функциональность может пронизывать большое количество модулей, существующие технологии имеют тенденцию реализовать эти требования, используя одномерные подходы, направляя все усилия по реализации требований вдоль одного измерения. Это единственное измерение как раз является реализацией требований относящихся к модулям. Оставшиеся требования ложатся вдоль этого главного измерения. Другими словами, все пространство требований является n-мерным, тогда как пространство реализации представляется одномерным пространством. В результате получаем трудно поддерживаемое отображение требований в реализацию.

Copyright © 1999 - 2007, IT-archiv. All rights reserved.



Признаки.

Несколько признаков, которые могут указывать на проблемную реализацию сквозной функциональности при использовании существующих подходов. Разобьем эти признаки на две категории:

- Запутанный код: В модуле может быть реализовано несколько требований. Например, часто разработчики одновременно решают проблемы связанные с бизнес логикой, производительностью, синхронизацией потоков и безопасностью. В результате множество элементов из разных требований присутствует в разрабатываемом модуле, что приводит к запутанному коду.
- Рассредоточенный код: Так как сквозная функциональность по определению распространяется на множество модулей, то вызовы этой функциональности будут рассредоточены по всей системе. Например, если в системе используется требование по слежению за производительностью работы базы данных, то такая сквозная функциональность затронет все модули, работающие с базой данных.

Следствия.

Наличие запутанного и рассредоточенного кода влияют на проектирование и реализацию во многих отношениях:

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

- Плохое прослеживание назначения модуля: Одновременная реализация нескольких требований в одной модульной единице делает неясным соответствие между отдельным требованием и его реализацией, в результате затруднительно понять, что реализует конкретный модуль.
- Непригодность кода для повторного использования: В связи с тем, что модуль может использовать в себе некоторую сквозную функциональность с жесткой привязкой к этой функциональности, другие части системы или другие проекты, где может потребоваться уже написанный модуль (но с другими требованиями к сквозной функциональности) не могут использовать уже написанный модуль.
- Большая вероятность ошибок: Запутанность кода влечет за собой код с множеством скрытых проблем. Более того, реализация нескольких ортогональных требований в одном модуле может привести к тому что ни одно из них не получит достаточного внимания разработчика.
- Трудность в сопровождении: Появление дополнительных требований в будущем потребует переработки текущей реализации, и это может затронуть большинство существующих модулей. Модификация каждой отдельной подсистемы в отдельности под новые требования может привести к несовместимости.

С тех пор как сложность программных систем возросла и появилась сквозная функциональность, находящаяся на срезе системы, не удивительно, что появились несколько подходов для решения этих проблем. Эти подходы включают в себя классы-примеси (mix-in) [20], шаблоны проектирования [6] и специфичные доменные решения.

При использовании классов-примесей реализацию сквозной функциональности можно поместить в них. Компоненты содержат экземпляр класса mix-In и позволяют другим частям системы устанавливать свой экземпляр.

Поведенческие шаблоны проектирования, такие как Visitor или Template Method позволяют поместить сквозную функциональность в главные классы, которые как в случае mix-In классов будут обходить компоненты, при этом вызывая логику специфичную для посещения данного компонента или вызывая специфичный шаблонный метод.

Специфичные доменные решения, такие как, например, каркасы (framework) и сервера приложений, позволяют разработчикам выносить некоторые сквозные требования на

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

уровень этих решений. Например, архитектура EJB позволяет вынести на уровень сервера приложения сквозную функциональность следующего вида — безопасность, администрирование, анализ производительности и управление поведением перманентных объектов, и сфокусироваться только на разработке компонентов уровня предприятия. Специфичные доменные решения предлагают специализированный механизм для решения специфичных проблем, однако при смене технологии приходится заново изучать новые подходы для решения тех же проблем.

Перед системным архитектором при проектировании системы возникает дилемма по выбору технологии при реализации сквозных требований — либо воспользоваться одним из существующих на данный момент решений конкретной проблемы либо воспользоваться новой технологией призванной решать подобные проблемы.

6. Введение в АОП

АСПЕКТ (от лат. aspectus — вид), точка зрения, с которой рассматривается какое-либо явление, понятие, перспектива. (Большой энциклопедический словарь)

Исследователи изучили различные пути выделения в отдельные модули сквозной функциональности в сложных программных системах. Аспектно-ориентированное программирование (АОП) является одним из этих решений. АОП предлагает средства выделения сквозной функциональности в отдельные программные модули — аспекты.

С точки зрения АОП в процессе разработки достаточно сложной системы программист решает две ортогональные задачи:

- Разработка компонентов, то есть выявление классов и объектов, составляющих словарь предметной области.
- Разработка сервисов, поддерживающих взаимодействие компонентов, то есть построение структур, обеспечивающих взаимодействие объектов, при котором выполняются требования задачи.

Современные языки программирования (такие как, например, C++, VB и т.п.) ориентированы, прежде всего, на решение первой задачи. Код компонента представляется в виде класса, т.е. он хорошо локализован и, следовательно, его легко просматривать, изучать, модифицировать, повторно использовать. С другой стороны, при программировании процессов, в которые вовлечены различные объекты, мы

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

получаем код, в котором элементы, связанные с поддержкой такого процесса, распределены по коду всей системы. Эти элементы встречаются в коде множества классов, их совокупность в целом не локализована в обозримом сегменте кода. В результате мы сталкиваемся с проблемой "запутанного" кода.

В рамках АОП утверждается, что никакая технология проектирования не поможет решить данную проблему, если только мы будем оставаться в рамках языка, ориентированного только на разработку компонентов. Для программирования сервисов, обеспечивающих взаимодействие объектов, нужны специальные средства, возможно специальные языки. После этапа кодирования компонентов и аспектов на соответствующих языках выполняется автоматическое построение оптимизированного для выполнения (но не для просмотра и модификации) кода. Этот финальный процесс называется слиянием или интеграцией (weaving).

6.1. Основные концепции АОП

Аспектно-ориентированный подход рассматривает программную систему как набор модулей, каждый из которых отражает определенный аспект — цель, особенность функционирования системы. Набор модулей, образующих программу, зависит от требований к программе, особенностей ее предметной области. Наряду с функциональными требованиями к программе предъявляются и общесистемные требования, например: целостности транзакций, авторизованного доступа к данным, ведения журнала событий и т. д. При проектировании программной системы разработчик выбирает модули так, чтобы каждый из них реализовывал определенное функциональное требование к системе. Однако реализация некоторых требований к программе зачастую не может быть локализована в отдельном модуле в рамках процедурного или объектно-ориентированного подхода. В результате код, отражающий такие аспекты функционирования системы, будет встречаться в нескольких различных модулях. Традиционные парадигмы программирования используют при проектировании программы функциональную декомпозицию и не позволяют локализовать сквозную функциональность в отдельных модулях. Необходимость реализации сквозной функциональности имеющимися средствами ведет к тому, что некоторый компонент содержит код, отражающий множество ортогональных требований к системе. Это делает такой модуль узкоспециализированным, ухудшает возможности его повторного использования и в

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

некоторых случаях приводит к дублированию кода. В свою очередь, это вызывает повышение вероятности внесения ошибок, увеличение времени отладки, снижает качество программы и в большой степени затрудняет ее сопровождение. Аспектно-ориентированный подход в некоторых случаях позволяет избежать описанных проблем и улучшить общий дизайн системы, обеспечивая возможность локализации сквозной функциональности в специальных модулях — аспектах.

АОП позволяет реализовывать отдельные концепции в слабосвязанном виде, и, комбинируя такие реализации, формирует конечную систему. АОП позволяет построить систему, используя слабосвязанные разбитые на отдельные модули (аспекты) реализации общесистемных требований.

Разработка в рамках АОП состоит из трех отдельных шагов:

- Аспектная декомпозиция: разбиение требований для выделения общей и сквозной функциональности. На этом шаге необходимо выделить функциональность для модульного уровня из сквозной функциональности системного уровня. Например, в примере с кредитными картами можно выделить три вещи: ядро обработки кредитных карт, журнал событий, аутентификация.
- Реализация функциональности: Реализовать каждое требование отдельно. В примере с кредитными картами необходимо отдельно реализовать модуль обработки кредитных карт, модуль журнала, модуль аутентификации.
- Компоновка аспектов: На этом шаге аспектный интегратор определяет правила для создания своих модулей — аспектов, составляя конечную систему. В примере с кредитными картами необходимо определить, в терминах языка реализующего АОП, при вызове каких операций необходимо вносить запись в журнал, и по завершению каких действий необходимо сообщать об успехе/неуспехе операции. Также можно определить правила, по которым будет вызываться модуль аутентификации перед доступом к бизнес-логике обработки кредитных карт.

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

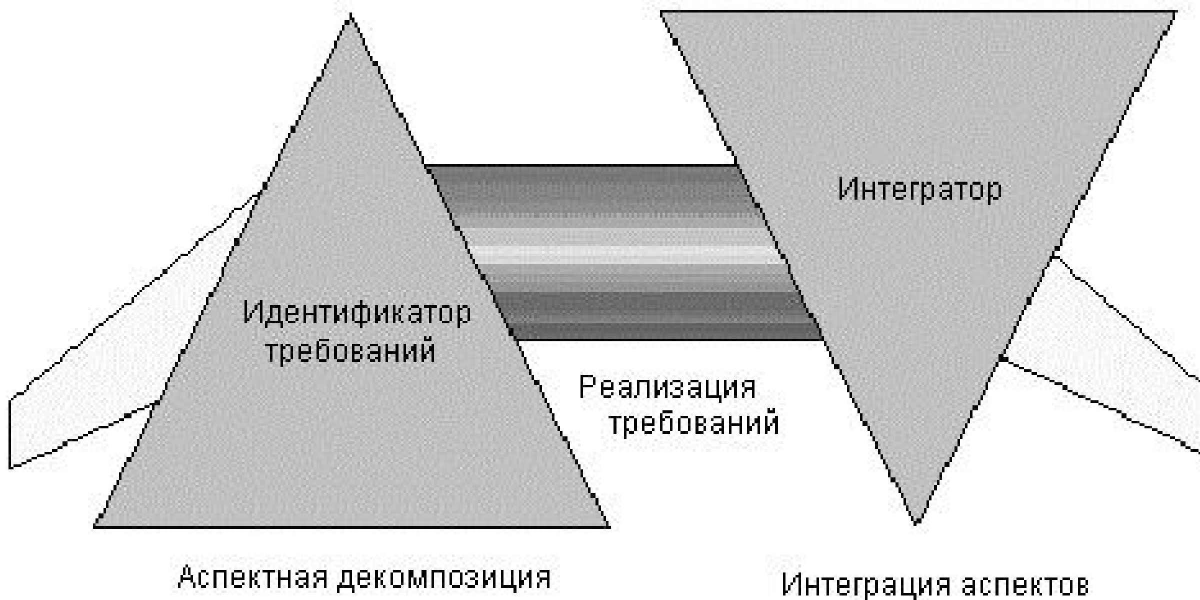


Рисунок 6. Фазы аспектно-ориентированной разработки ПО.

АОП отличает многое от традиционных подходов ООП при реализации сквозной функциональности: здесь нужно по-другому представлять себе процесс декомпозиции, а архитектура получающегося программного продукта в значительной степени выходит за рамки представлений, традиционных для объектного программирования. При разработке на АОП концепции реализуются абсолютно независимо друг от друга, так как все существующие между ними связи (сквозная функциональность) могут быть локализованы в аспектных модулях, описывающих протокол взаимодействия концепций. Например, в модуле обработки кредитных карт может отсутствовать запись в журнал или вызов модуля авторизации, однако при работе может вызываться подобная сквозная функциональность, если она описана в протоколе взаимодействия. Это серьезный шаг в развитии методологий от ООП.

Аспектом в АОП является системный модуль, в который вынесена сквозная функциональность. Аспектный модуль — это результат аспектной декомпозиции, на этапе которой выявляются какие-либо явления, понятия, события которые могут быть применены к группе компонентов, полученных после объектной декомпозиции. Аспект представляет собой языковую концепцию, схожую с классом, но только более высокого уровня абстракции.

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

Аспекты могут затрагивать многие компоненты и используют так называемые точки вставки для реализации регулярных действий, которые обычно рассредоточены по всему тексту программы. В аспектном модуле описываются срезы точек — точки выполнения программы, в которые встраиваются инструкции языка, которые должны выполняться до, после или вместо строго определенной точки выполнения программы. Подобные инструкции языка являются функциональностью, поддерживающей взаимодействие компонентов. Кроме того, в аспектном модуле могут описываться роли компонентов, на которые может воздействовать данный аспект. В отдельных реализациях АОП при помощи аспектных модулей можно влиять на существующую схему наследования. С точки зрения АОП аспект является сервисом, связывающим компоненты системы.

Пример интеграции аспектов.

Для иллюстрации работы интегратора аспектов вернемся к примеру обработки кредитных карт. Для краткости рассмотрим только 2 операции — кредит и дебет:

```
public classCreditCardProcessor {
    public void debit(CreditCard card, Currency amount)
        throwsInvalidCardException, NotEnoughAmountException,
              CardExpiredException {
        // логика по дебету
    }

    public void credit(CreditCard card, Currency amount)
         throws InvalidCardException {
        // логика по кредиту
    }
}
```

и интерфейс журнала событий:

```
public interface Logger {
        public void log(Stringmessage);
}
```

Для получения желаемой композиции требуется применение следующих правил, выраженных на обычном языке:

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

1. записать в журнал начало каждой операции
2. записать в журнал окончание каждой операции
3. записать в журнал каждую исключительную ситуацию, которая может возникнуть в процессе работы этого модуля.

Интегратор аспектов, применяя такие правила, получит код эквивалентный данному:

```
public class CreditCardProcessorWithLogging {
        Logger _logger;
         public void debit(CreditCard card, Money amount)
            throws InvalidCardException, NotEnoughAmountException,
            CardExpiredException {
            _logger.log("Starting CreditCardProcessor.debit(CreditCard,
            Money) "+ "Card: " + card + " Amount: " + amount);
            // Debiting logic
            _logger.log("Completing CreditCardProcessor.debit(CreditCard,
            Money) " + "Card: " + card + " Amount: " + amount);
        }
         public void credit(CreditCard card, Money amount)
            throws InvalidCardException {
            System.out.println("Debiting");
            _logger.log("Starting CreditCardProcessor.credit(CreditCard,
            Money) " + "Card: " + card + " Amount: " + amount);
            // Crediting logic
            _logger.log("Completing CreditCardProcessor.credit(CreditCard,
            Money) " + "Card: " + card + " Amount: " + amount);
         }
}
```

Автоматическая компоновка аспектов и традиционных модулей программы — компонентов является ключевым свойством АОП, которое определяет основное преимущество данной технологии: делает возможной инкапсуляцию сквозной функциональности в отдельных программных модулях.

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

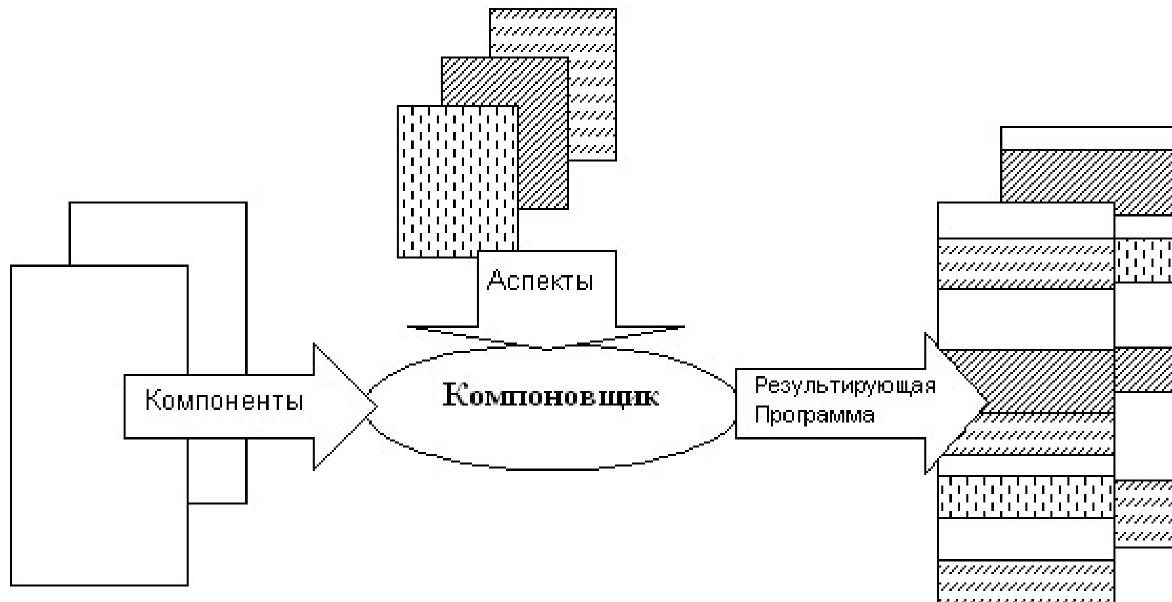


Рисунок 7. Процесс компоновки аспектных и традиционных модулей.

Автоматизированная компоновка аспектов и компонентов является мощным средством генерации кода и в общем случае гарантирует, что аспект будет применен ко всем модулям-компонентам, которые он затрагивает, чего сложно добиться, если вносить сквозную функциональность в модули (вручную). Реализация автоматической компоновки аспектов и компонентов во многом определяет возможности той или иной аспектно-ориентированной платформы. В настоящее время обсуждаются два подхода к интеграции аспектов:

- Статическая интеграция на этапе компиляции
- Динамическая интеграция на этапе выполнения программы

Подходы к интеграции аспектов определяются языком, поддерживающим АОП, и детально изложены в [2] .

6.2. Преимущества использования АОП

АОП помогает избежать вышеупомянутых проблем, вызванных запутанным и рассредоточенным кодом. Ниже представлены дополнительные преимущества предоставляемые АОП:

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

- Улучшение декомпозиции системы на отдельные модули: АОП позволяет инкапсулировать функциональность, которая не может быть представлена в виде отдельной процедуры или компонента. АОП позволяет реализовать каждое требование отдельно с минимальным связыванием, в результате получается модуль, содержащий данное требование к системе без внешних лишних зависимостей, даже если это требование — сквозная функциональность. При такой реализации модули содержат минимальное количество дублируемого кода. Поскольку каждое требование реализуется отдельно, это позволяет избежать запутанного кода. В результате получается система, которую легче понимать и поддерживать.
- Упрощение сопровождения программной системы и внесения в нее изменений: Так как могут существовать модули, на которые могут воздействовать аспекты, становится достаточно легко добавлять новую функциональность путем создания новых аспектов. Более того, если добавляется новый модуль в систему, то существующие аспекты начинают воздействовать и на него без дополнительных усилий. При использовании АОП системный архитектор может отложить решения, касающиеся потенциально возможных требований, поскольку впоследствии эти решения смогут быть реализованы как отдельные аспекты, что не затронет существующую функциональность.
- Появление возможностей повторного использования кода, реализующего сквозную функциональность: Следует из того, что при использовании АОП сквозная функциональность может быть реализована в виде аспектов и в виде слабосвязанных модулей.

Технология вынесения сквозной функциональности в отдельные аспектные модули стала важным эволюционным шагом в развитии таких концепций как абстракция и повторное использование программного кода. Идеи абстракции и повторного использования кода занимают одно из центральных мест в программировании.

Абстракция — это метод, с помощью которого разработчики решать сложные проблемы, последовательно разбивая их на более простые. Затем можно использовать имеющиеся готовые решения полученных типовых простых проблем в качестве строительных блоков, из которых разработчики получают решения, пригодные для реализации повседневных сложных проектов [7] .

Повторное использование не менее важно для разработки ПО, так как этот фактор

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

является целью, к которой устремлена технология разработки по своей природе [7]. В процессе эволюции методологий разработки ПО было придумано несколько методов повторного использования кода и концепций разработки ПО:

- Раньше других появился метод повторного использования, основанный на технологии "скопируй и вставь", или, проще говоря, на вставке в новые программы фрагментов ранее созданных программ. Данный подход является абсолютно неэффективным, кроме того, он не дает никаких сколько-нибудь заметных качественных преимуществ с точки зрения абстракции кода.
- Более гибкий метод повторного использования состоит в повторном использовании алгоритмов. В соответствии с этим методом, разработчик может использовать любой однажды разработанный алгоритм.
- Функциональное повторное использование программного кода и повторное использование структур данных позволяют обеспечить непосредственное повторное использование абстракции кода.

Двумя расширениями концепции повторного использования кода являются библиотеки функций и API. Они представляют разработчику получить полный пакет функциональности, доступный для всех последующих приложений без необходимости копирования программного кода из приложения в приложение [7].

В ходе развития объектно-ориентированных языков программирования был совершен огромный скачок вперед в области абстракции и повторного использования программного кода. С помощью ООП была создана целая плеяда высокопроизводительных методов его создания. Однако рост сложности программных систем привел к тому, что появилась необходимость в повторном использовании сквозной функциональности. Поэтому появилась потребность в новом подходе повторного использования кода, который бы охватил этот тип функциональности и решил бы проблемы, которые не могли решить предыдущие подходы.

Для иллюстрации важности повторного использования кода в таблице 1 приведена сравнительная характеристика из [7] различных подходов к повторному использованию.

Метод	Повторное использование	Абстракция	Универсальность подхода
Копирование и вставка	Очень плохо	Отсутствует	Очень плохо

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

Структуры данных	Хорошо	Тип данных	Средне — хорошо
Функциональность	Хорошо	Метод	Средне — хорошо
Типовые блоки кода	Хорошо	Типизируемая	Хорошо
Алгоритмы	Хорошо	Формула	Хорошо
Классы	Хорошо	Данные + Метод	Хорошо
Библиотеки	Хорошо	Функции	Хорошо — очень хорошо
API	Хорошо	Классы утилит	Хорошо — очень хорошо
Компоненты	Хорошо	Группы классов	Хорошо — очень хорошо
Шаблоны проектирования	Отлично	Решения проблем	Очень хорошо
Сквозная функциональность	Средне — хорошо	Отсутствует	Плохо
Сквозная функциональность	Хорошо	Аспект	Очень хорошо

В столбце "Абстракция" данной таблицы указаны сущности, для которых выполняется абстракция, а в столбце "Универсальность подхода" показано то, насколько легко применить существующий метод, не прибегая к изменениям или переработке кода. Показатель степени повторного использования очень сильно зависит от эффективности применения того или иного метода на практике.

Можно сказать что, слабосвязанная реализация сквозной функциональности — это ключ к реальному повторному использованию кода. АОП позволяет получить менее связанную реализацию, чем ООП. При использовании этой парадигмы программирования должно тратиться меньше времени на сопровождение и внесение изменений в готовый программный код, поэтому роль АОП становится все важнее.

АОП не является заменой существующих технологий. Наоборот, также как процедурное программирование используется в объектно-ориентированном

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

программировании (ООП) для реализации поведения объектов, АОП использует существующие подходы для реализации своих модулей — аспектов, то есть исполняет роль расширения, позволяющего обеспечить модуляризацию сквозной функциональности. В зависимости от технологии и языка программирования соответствующая реализация АОП будет обладать различными возможностями.

6.3. Недостатки аспектного подхода

В настоящий момент аспектно-ориентированный подход обладает рядом недостатков:

- **Не полностью проработана методология АОП-разработки программ.** В данный момент законченный и оттестированный компилятор имеется только для нескольких языков, что ограничивает применение данной технологии. С другой стороны, реализация концепций АОП только на уровне языковых расширений представляется неполной и неэффективной. Более перспективным стоит считать проникновение самих базовых идей АОП и использование их на различных уровнях средств разработки — языков (C++, Pascal), платформ (.NET, Java), и технологий (COM, CORBA и т.д.), по аналогии с уже традиционным использованием ООП.
- **Недостаточно качественная реализация расширений языков.** В настоящее время существующие реализации АОП-расширений для различных языков и платформ различаются по своим возможностям, но можно выделить общие черты: каждая реализация АОП должна предоставить механизм описания логики сквозной функциональности и механизм описания точек программы, в которых данная логика будет применяться. Однако каждая реализация по-своему определяет виды точек связывания, в которых можно применить аспект, а также решает вопрос их описания, что затрудняет понимание и использование общих принципов АОП.
- **Недостаточно проработан механизм привязки аспектов к компонентам.** В распространенных в настоящее время АОП-реализациях точки связывания описываются в терминах программных конструкций — классов, методов, полей класса. При этом получается тесная связь между аспектом и компонентом, к которому он применяется. Логика привязки аспекта к данной точке кода выражена неявно и целиком определяется программной конструкцией, в терминах которой построено описание. Тесная связь между аспектом и компонентом делает аспект зависимым от компонента и при этом нарушается одна из основных идей АОП —

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

независимость компонентов от применяемых к нему аспектов.
- **Не полностью исследованы случаи, когда аспекты удобно и целесообразно было бы применить.** Эту проблему частично решает данная статья.

6.4. AspectJ как одна из реализаций АОП

АОП можно поддерживать в рамках уже существующих языков. Так, в частности, исследовательский центр Xerox PARC разработал систему AspectJ, поддерживающую АОП в рамках языка Java. Этот пакет встраивается в такие системы разработки, как Eclipse, Sun ONE Studio, Forte 4J и Borland JBuilder. В данной работе AspectJ был выбран из-за того, что данная реализация АОП обладает наиболее широкими возможностями.

AspectJ — это простое и практическое расширение языка Java, которое добавляет к Java возможности предоставляемые АОП. Пакет AspectJ состоит из компилятора (ajc), отладчика (ajdb), и генератора документации (ajdoc). Поскольку AspectJ является расширением Java, то любая программа, написанная на Java, будет правильной с точки зрения семантики AspectJ. Компилятор AspectJ выдает байт-код совместимый с виртуальной машиной Java. Поскольку в качестве базового языка для AspectJ был выбран язык Java, то он унаследовал от Java все преимущества и спроектирован таким образом, что будет легко понятен разработчикам Java. Добавленные расширения касаются в основном способов задания правил интеграции аспектов и java-объектов. Данные правила выражаются в ключевых понятиях AspectJ:

- JoinPoint — строго определенная точка выполнения программы, ассоциированная с контекстом выполнения (вызов метода, конструктора, доступ к полю класса, обработчик исключения, и т.д.)
- Pointcut — набор (срез) точек JoinPoint удовлетворяющих заданному условию.
- Advice — набор инструкций языка java, выполняемых до, после или вместо каждой из точек выполнения (JoinPoint), входящих в заданный срез (Pointcut)
- Aspect — основная единица модульности AspectJ. В аспектах задаются срезы точек выполнения (Pointcut) и инструкции, которые выполняются в точках выполнения (Advice)
- Introduction — способность аспекта изменять структуру Java-класса путем добавления новых полей и методов, так и иерархию класса.

Pointcut и Advice определяют правила интеграции. Аспект — единица, напоминающая

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

класс в ООП, соединяет элементы pointcut и элементы advice вместе, и формирует модуль на срезе системы.

Рассмотрим пример, на котором можно понять, как язык AspectJ реализует принципы АОП. В качестве примера возьмем модель простого графического редактора. Данный графический редактор может работать с двумя типами графических элементов — точкой и линией. Диаграмма классов графического редактора представлена на рисунке 8. Классы Point и Line реализуют интерфейс FigureElement содержащий метод перемещения фигуры. Операциями, влияющими на обновление экрана, являются операции перемещения фигур. После перемещения фигуры необходимо обновить изображение (Display). Обновление изображения в данном случае является сквозной функциональностью, которая должна вызываться при некоторых условиях (изменении положения фигур).

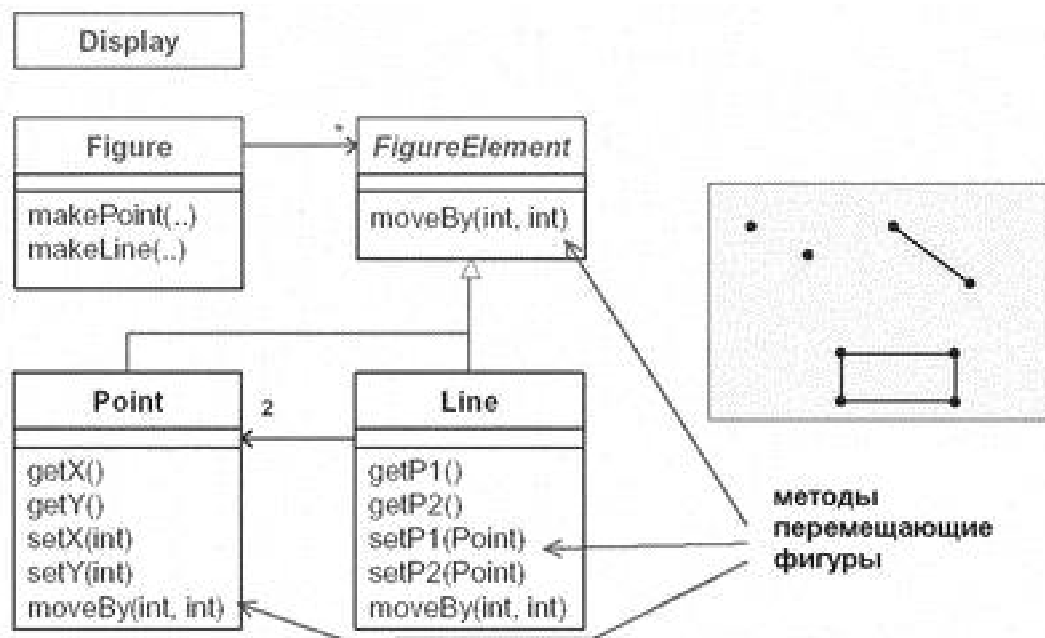


Рисунок 8. Модель графического редактора.

Класс Line содержит в себе 2 экземпляра класса Point. Классы, реализующие точку и линию, могут быть представлены в следующем виде:

```
class Line implements FigureElement{
```


Copyright © 1999 - 2007, IT-archiv. All rights reserved.

```
     private Point p1, p2;
     Point getP1() { return p1; }
     Point getP2() { return p2; }
     void setP1(Point p1) { this.p1 = p1; }
     void setP2(Point p2) { this.p2 = p2; }
     void moveBy(int dx, int dy) { ... }
   }

class Point implements FigureElement {
     private int x = 0, y = 0;
     int getX() { return x; }
     int getY() { return y; }
     void setX(int x) { this.x = x; }
     void setY(int y) { this.y = y; }
     void moveBy(int dx, int dy) { ... }
}
```

Предположим, клиент переместил фигуру на 2 пункта, при этом отработал метод lineInst.moveBy(2,2). На схеме вызываемых функций (рисунок 9) можно увидеть, как происходит работа этого метода. В теле вызываемого метода moveBy()содержится логика по перемещению фигуры; метод имеет входные параметры, возвращаемое значение и набор инструкций языка, среди которых находится вызов аналогичного метода у двух точек — элементов класса Line. Метод moveBy() у класса Point, также имеет входные параметры, возвращаемое значение и набор инструкций языка. Каждая строго определенная точка выполнения программы, ассоциированная с контекстом выполнения, с учетом ограничений языка AspectJ является JoinPoint-ом. Выполнение метода moveBy() у класса Line, вызов метода moveBy у Point — являются точками JoinPoint.

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

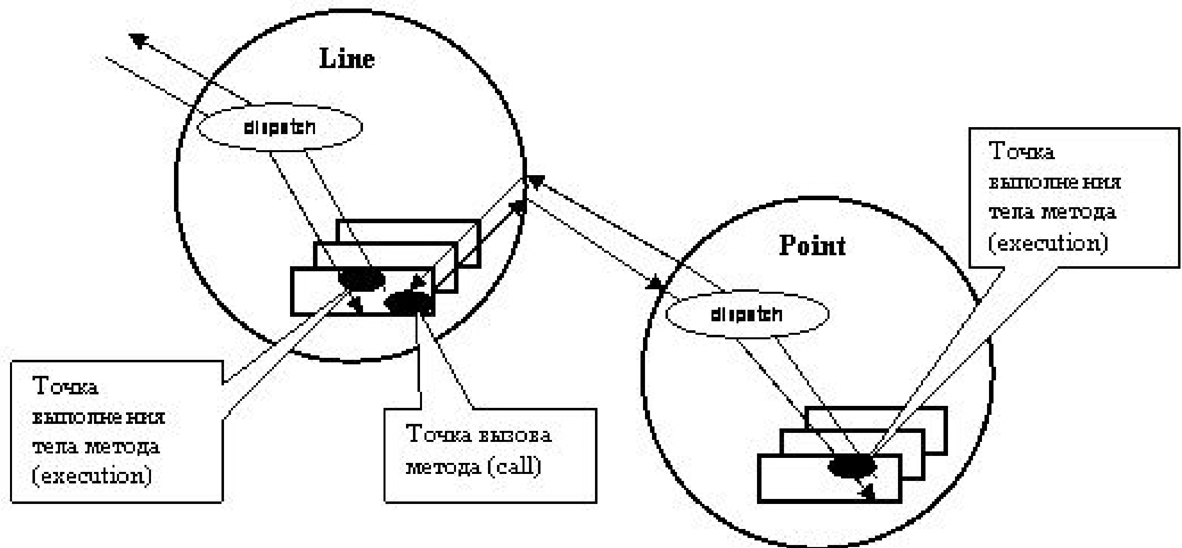


Рисунок 9. Схема вызываемых функций.

Здесь необходимо ввести различие в терминологию точек выполнения программы в рамках AspectJ. Точки выполения бывают двух типов — точка выполнение метода, которая представляет собой выполнение всего тела метода, и точка вызова метода, которая представляет собой точку непосредственного вызова метода. Другими словами тело метода может быть представлено точкой со стороны клиента либо только вызов метода может быть представлен как точка.

Язык AspectJ позволяет описывать несколько типов подобных точек выполнения программы.

- Вызов методов и конструкторов
- Выполнение методов и конструкторов
- Доступ к полям класса
- Обработка исключительных ситуаций
- Статическая и динамическая инициализация классов

При вызове клиентом метода moveBy все точки выполнения (joinPoint) находятся внутри потока управления, начиная с точки удовлетворяющей условию. Средствами языка AspectJ можно влиять на поток управления, описав соответствующий поток управления как набор точек pointcut.

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

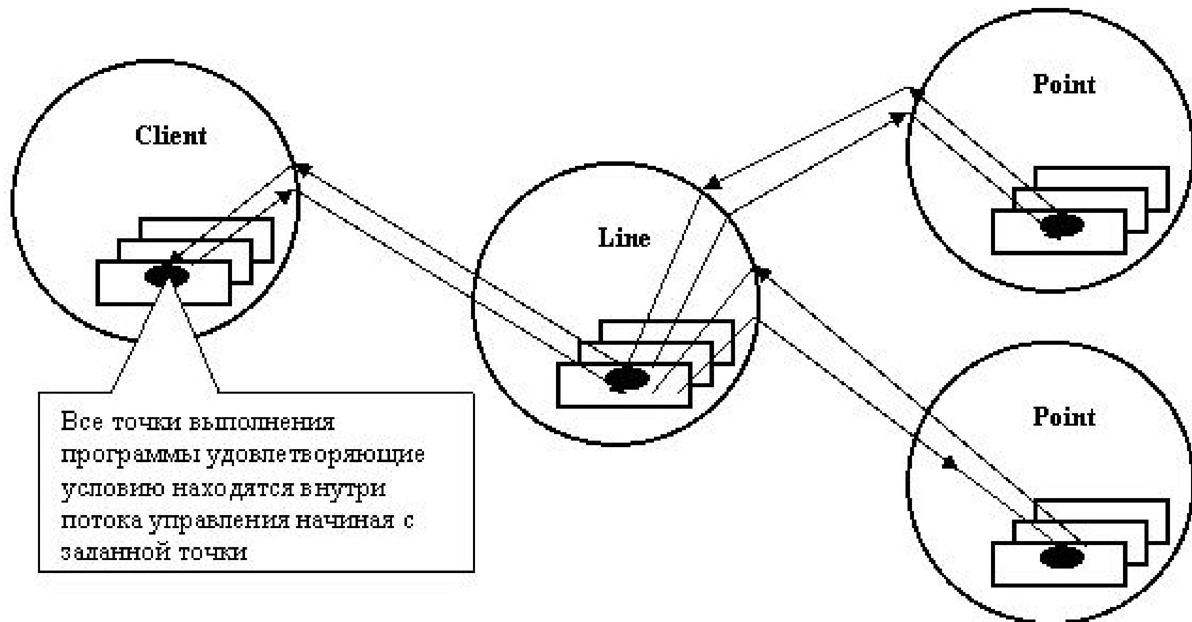


Рисунок 10. Поток выполнения метода moveBy.

Конструкцией языка позволяющей описывать набор точек JoinPoint удовлетворяющих заданному условию является pointcut. Например, **call(void Line.setP1(Point))** соответствует каждой точке выполнения программы, если она является вызовом метода с сигнатурой "void Line.setP1(Point)". При конструировании подобных срезов можно использовать логические операции &&, || и !, что дает большую гибкость при описании наборов точек выполнения программы. Срезы точек могут иметь имя, что позволяет их повторно использовать при конструировании других срезов и описании набора инструкций Advice.

Advice — набор инструкций языка java, выполняемых до, после или вместо каждой из точек выполнения, входящих в заданный срез, набор инструкций, выполняемый в некоторой точке выполнения по определенным правилам. Язык AspectJ позволяет описывать подобные инструкции по следующим правилам:

- **before.** Набор инструкций выполняется перед выполнением инструкций входящих в описываемую точку выполнения.
- **afterreturning.** Набор инструкций выполняется после возвращения значения из описываемой точки выполнения.
- **afterthrowing.** Набор инструкций выполняется после возникновения

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

исключительной ситуации в описываемой точке выполнения.

- **after.** Набор инструкций выполняется после возвращения из описываемой точки выполнения в любом случае.
- **around.** Набор инструкций выполняется вместо описываемой точки выполнения.

Приведенные правила формирования набора выполняемых инструкций в точке выполнения программы дают много возможностей при встраивании сквозной функциональности. При этом сложности, которые возникают при наличии в системе сквозной функциональности описанные выше, исчезают. При этом в системе нет запутанного и рассредоточенного кода сквозной функциональности мешающей пониманию сущности каждого конкретного компонента.

Аспект — единица модульности AspectJ. В аспектах задаются срезы точек выполнения и инструкции, которые выполняются в точках выполнения. Аспект имеет сходство с классом — аспект может содержать методы и поля, расширять другие классы или аспекты или реализовывать интерфейсы.

Далее представлено два варианта кода рассматриваемого графического редактора — с использованием языка AspectJ и без него. В варианте без использования AspectJ видно, что при изменении координат фигур в методы, отвечающие за эту функциональность, встроен код реализующий обновление дисплея. В примере с использованием аспектного подхода классы содержат код, отвечающий только за свою логику. Набор инструкций, реализующий сквозную функциональность "обновление дисплея" находится в аспектном модуле — aspect DisplayUpdating. На этом примере наглядно показано основное преимущество аспектного подхода — локализация сквозной функциональности в отдельных модулях и встраивание подобной функциональности в требуемые участки системы.

Пример кода графического редактора без использования языка AspectJ:

```
class Line implements FigureElement {
     private Point p1, p2;
  Point getP1() { return p1; }
  Point getP2() { return p2; }
  void setP1(Point p1) {
    this.p1 = p1;
    Display.update(this);
```


Copyright © 1999 - 2007, IT-archiv. All rights reserved.

```
  }
  void setP2(Point p2) {
    this.p2 = p2;
    Display.update(this);
  }
}
class Point implements FigureElement {
  private int x = 0, y = 0;
  int getX() { return x; }
  int getY() { return y; }
  void setX(int x) {
    this.x = x;
    Display.update(this);
  }
  void setY(int y) {
    this.y = y;
    Display.update(this);
  }
}
```

Пример кода графического редактора с использованием языка AspectJ:

```

class Line implements FigureElement {
       private Point p1, p2;
       Point getP1() { return p1; }
       Point getP2() { return p2; }
       void setP1(Point p1) {
         this.p1 = p1;
       }
       void setP2(Point p2) {
         this.p2 = p2;
       }
     }
class Point implements FigureElement {
       private int x = 0, y = 0;
       int getX() { return x; }
       int getY() { return y; }
       void setX(int x) {this.x = x; }
       void setY(int y) {this.y = y; }
     }
```


Copyright © 1999 - 2007, IT-archiv. All rights reserved.

```
aspect DisplayUpdating {
       pointcut move(FigureElement figElt):
         target(figElt) &&
         (call(void FigureElement.moveBy(int, int) ||
          call(void Line.setP1(Point))            ||
          call(void Line.setP2(Point))            ||
          call(void Point.setX(int))              ||
          call(void Point.setY(int)));
       after(FigureElement fe) returning: move(fe) {
        Display.update(fe);
       }
}
```

При разработке программных систем с использованием средств языка AspectJ можно полностью следовать трем принципам разработки аспектного подхода: выделять в отдельные модули сквозную функциональность — провести аспектную декомпозицию; реализовать каждое требование отдельно; интегрировать аспекты в программный код. В примере с графическим редактором на этапе аспектной декомпозиции была выявлена сквозная функциональность — обновление дисплея. Данное требование было реализовано в аспектном модуле DisplayUpdating. В этом аспекте определяется срез точек move(..), которые включают в себя точки выполнения программы, после которых будет встроена требуемая сквозная функциональность. На рисунке 11 схематично изображен аспект, находящийся на срезе модели графического редактора.

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

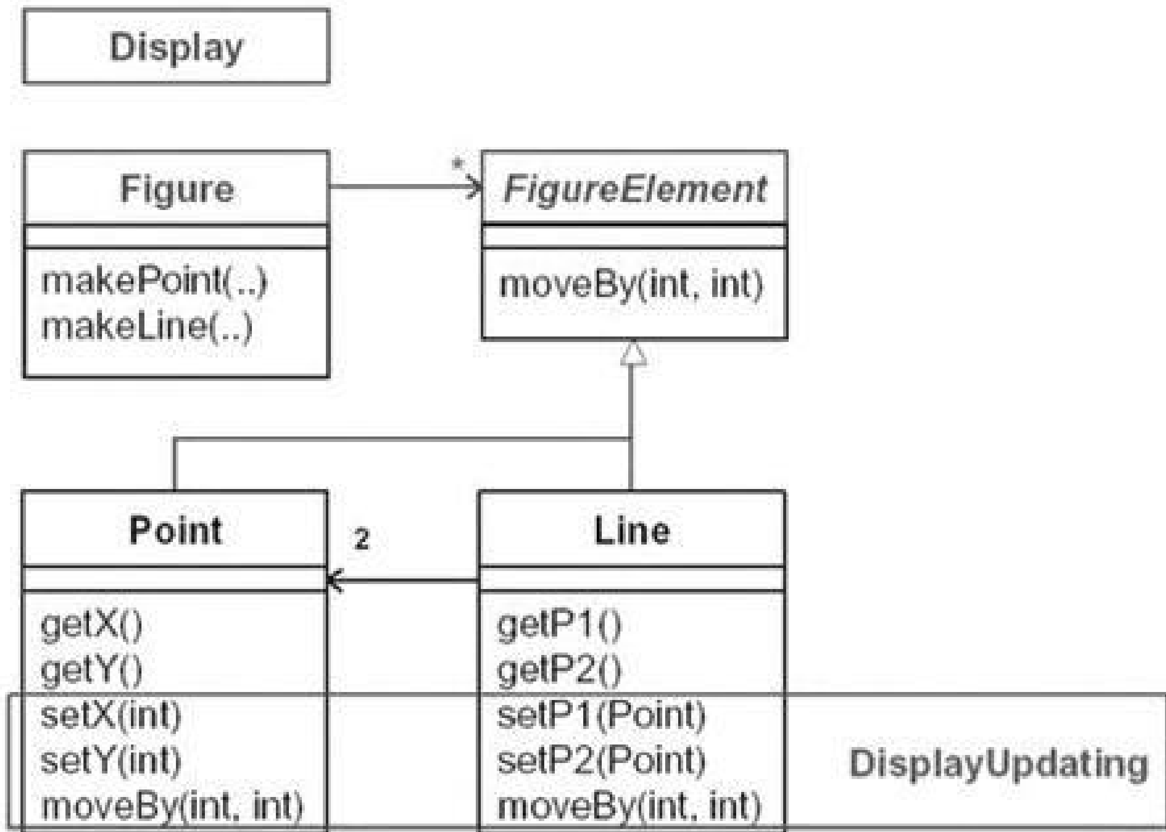


Рисунок 11. Аспект DisplayUpdating на срезе системы.

Интеграции аспектов (weaving) происходит в момент компиляции. Модель построения готовой программной системы при использовании компилятора ajc изображена на рисунке 12.

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

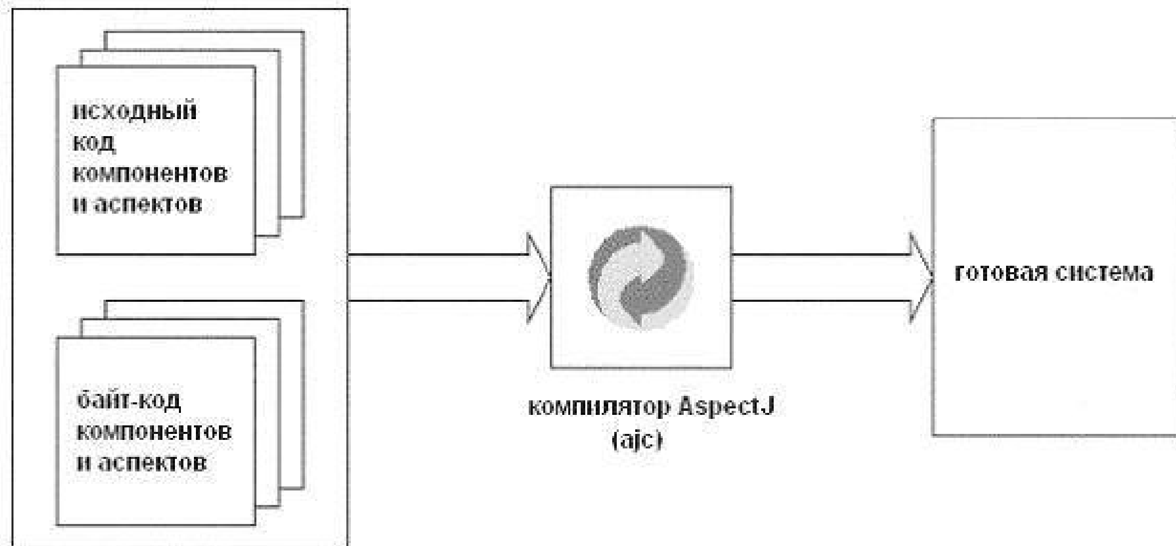


Рисунок 12. Интеграция аспектов.

После компиляции получаем готовую систему с интегрированной сквозной функциональностью по правилам, описанным в аспектных модулях.

6.5. Другие реализации АОП

В данный момент, кроме представленного выше языка AspectJ, известно несколько систем реализующих принципы АОП:

- **ANGIE Generation Now!** Предоставляет новый структурированный язык для реализации различных генераторов, в том числе интеграторов аспектов.
- **AspectC++.** Это аспектно-ориентированное расширение языка C++.
- **AspectR.** Система аспектно-ориентированного программирования на основе языка Ruby, которая позволяет в классах оборачивать код вокруг существующих методов.
- **AspectS.** Это один из первых прототипов, который дает возможность аспектно-ориентированного программирования в среде Squeak/Smalltalk.
- **Apostle** система близкая к AspectJ, которая является АОП расширением языка Smalltalk.
- **AspectC.** Простое АОП расширение языка С, похожее на AspectJ.
- **AspectC#** реализация, добавляющая поддержку аспектов к языку C#.
- **Caesar.** Аспетно-ориентированный язык программирования, который

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

фокусируется на многостороннем подходе к декомпозиции и повторном использовании аспектного кода.

- **DemeterJи DJ.** Делаются шаги в сторону структурной инкапсуляции сквозной функциональности.
- **Hyper/J.** Предоставляет поддержку "многомерного" разделения и интеграции концепций для стандартной java-платформы.
- **JAC**. Представляет собой основу, написанную на языке Java, для создания распределенных аспектно-ориентированных программных систем.
- **JMangler.** Представляет собой каркас на языке Java для преобразования программ на этапе загрузки, с поддержкой бесконфликтной композиции из независимой друг от друга разработанных аспектов (реализуемых в виде преобразующих компонентов системы JMangler). Также есть возможность интеграции аспектов в базовые классы Java .
- **MixJuice** это расширение языка Java, базирующегося на механизме поиска различий в модулях.
- **MozartProgramming System** это расширенная платформа для разработки "разумных и интеллектуальных" распределенных приложений.
- **ObjectEverywhere** — статья, описывающая АОП каркас, базирующийся на COM и языке Delphi.
- **PROSE** — аспектно-ориентированная платформа, базирующаяся на виртуальной машине Java, и позволяющая динамическое встраивание и вычленение аспектного кода.
- **Pythius** проект с открытым исходным кодом, добавляющий принципы АОП к языку Python.
- **SmartTools** — аспектно и XML ориентированный генератор семантических каркасов.
- **UMLAUT** представляет собой систему, которая позволяет встраивать многомерные высокоуровневые UML проектные модели в модели подходящие для каждого конкретного случая реализации.
- **Weave.NET** проект, направленный на исследование механизма поддержки АОП без привязки к конкретному языку программирования внутри компонентной модели .NET Framework.

7. Критерии сравнения аспектной реализации с

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

объектно-ориентированной

Для объективного анализа представленных вариантов использования аспектных реализаций по сравнению с объектно-ориентированными реализациями необходимо ввести критерии сравнения этих реализаций.

В процессе развития информационных технологий и программирования в частности были предприняты попытки дать количественную оценку характеристикам качества программного обеспечения путем разработки численных показателей, для чего осуществляется их формализация вводом метрик. Применение метрик позволяет упорядочить разработку, испытания, эксплуатацию и сопровождение программного продукта. В зависимости от характеристик и особенностей показателя качества применяются различные виды метрик и шкал для измерения показателей.

Первый вид метрик — это метрики, которым соответствует интервальная шкала, характеризуется относительными величинами или реально измеряемыми физическими показателями. Например, используя этот вид метрик можно сказать, что одна программа труднее или эффективнее другой программы на 10 единиц.

Второй вид метрик — это метрики, которым соответствует порядковая шкала. Она позволяет ранжировать некоторые характеристики путем сравнения с опорными значениями, т.е. измерение по этой шкале фактически определяет взаимное положение конкретных модулей или программных систем. Для объекта измерения устанавливается приоритетность признаков.

Третий вид метрик — это метрики, которым соответствует номинальная или категорированная шкала. Данный вид метрик характеризует наличие рассматриваемого свойства или признака у объекта, в частности у программного модуля, без учета градаций по данному признаку. Фиксируется, есть или нет данное качество в зависимости от наличия рассматриваемого показателя у комплекса программ (например, наличие структурирования, гибкости, простоты освоения). Например, такую характеристику как сложность модуля можно количественно оценить значениями этой метрики: [нетрудная для понимания], [умеренно трудная для понимания], [трудная для понимания], [очень трудная для понимания].

Существующие качественные оценки программных систем можно сгруппировать по

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

нескольким интересующим в данной работе направлениям:

- Оценки топологической и информационной сложности программ.
- Оценки уровня языковых средств и их применения.
- Оценки трудности восприятия и понимания программных текстов, ориентированные на психологические факторы, существенные для сопровождения и модификации программ.

7.1. Топологическая и информационная сложность программного модуля

Традиционной характеристикой размера программ является количество строк исходного текста. Оценка размера программ есть оценка по номинальной шкале, на основе которой определяются только категории программ без уточнения оценки для каждой категории. К данной группе оценок можно отнести метрику Холстеда. Основу этой метрики составляют четыре измеряемые характеристики программы:
NUOprtr (Number of Unique Operators) — число уникальных операторов программы, включая символы — разделители, имена процедур и знаки операций (словарь операторов);
NUOprnd (Number of Unique Operands)- число уникальных операндов программы (словарь операндов);
Noprtr (Number of Operators) — общее число операторов в программе;
Noprnd (Number of Operands) — общее число операндов в программе.

Опираясь на эти характеристики, получаемые непосредственно при анализе исходных текстов программ, М. Холстед вводит следующие оценки:

- словарь программы (Halstead Program Vocabulary) HPVoc = NUOprtr + NUOprnd;
- длина программы (Halstead Program Length) HPLen = Noprtr + Noprnd;
- объем программы (Halstead Program Volume) HPVol = HPLen log2 HPVoc.

Далее Холстед вводит сложность программы (Halstead Difficulty), которая вычисляется как HDiff = NUOprtr/2* (NOprnd / NUOprnd)
Используя HDiff Холстед вводит оценку HEff (Halstead Effort) Heff = HDiff* HPVol, с помощью, которой описывается усилия программиста при при разработке.
Вторая наиболее представительная группа оценок сложности программ — метрики сложности потока управления программ. Как правило, с помощью этих оценок

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

оперируют либо плотностью управляющих переходов внутри программ, либо взаимосвязями этих переходов. И в том и в другом случае программа представляется в виде управляющего графа. Впервые графическое представление программ было предложено Маккейбом. Основной метрикой сложности он предполагает считать цикломатическую сложность графа программы, или, как ее еще называют, цикломатическое число Маккейба, характеризующее трудоемкость тестирования программы.

Для вычисления цикломатического числа Маккейба CC (Cyclomatic Complexity) применяется формула:

CC = L — N + 2P,

где L — число дуг ориентированного графа;

N — число вершин;

P — число компонентов связности.

К метрикам сложности также относится:

- NORM (Number Of Remote Methods) — количество вызываемых удаленных методов. При формировании значения этой метрики просматриваются все конструкторы и методы класса, и подсчитывается количество вызываемых удаленных методов. Удаленным методом является метод, который не определен в классе и его родителях.
- RFC (Response For Class) Отклик на класс — количество методов, которые могут вызываться экземплярами класса. Эта метрика вычисляется как сумма количества локальных методов и количества удаленных методов.
- WMPC1 (Weighted Methods Per Class 1) — взвешенная насыщенность класса — дает относительную меру его сложности; если считать что все методы имеют одинаковую сложность, то это будет просто число методов в классе. Эта метрика определяется суммой сложности всех методов класса, где каждый метод взвешивается подсчетом его цикломатического числа. Количество методов и их сложность связаны для определения времени и усилий, которые потребуются для разработки и поддержки класса. Вообще, класс, который имеет большее количество методов среди классов одного с ним уровня, является более сложным; скорее всего, он специфичен для данного приложения и содержит наибольшее количество ошибок. Для расчета данной метрики используются только методы определенные в конкретном классе, все методы, наследуемые от родительского класса, не

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

включаются.

- WMPC2 (Weighted Methods Per Class 2) Эта метрика является мерой сложности класса, полагающая что класс с большим количеством методов чем другой является более сложным, и что метод с большим количеством параметров также является более сложным. Для расчета данной метрики используются только методы определенные в конкретном классе, все методы, наследуемые от родительского класса, не включаются.
- LOCOM1 (Lack Of Cohesion Of Methods 1) — недостаток связности методов. Связность — это степень взаимодействия между элементами отдельного модуля, характеристика его насыщенности. Наименее желательной является связность по случайному принципу, когда в одном модуле собираются совершенно независимые абстракции. Связанность объектов — мера их взаимозависимости. Разработчики стремятся спроектировать не зацепленные (то есть слабо связанные) объекты, поскольку они имеют больше шансов на повторное использование. Для каждой пары методов класса определяется набор полей, к которым они имеют доступ. Если множество полей имеет доступ к непересекающемуся множеству полей класса, то число P увеличивается на 1. Если оба метода используют хотя бы одно общее поле, то параметр Q увеличивается на 1. После рассмотрения каждой пары методов результат вычисляется как RESULT = (P > Q) ? (P — Q) : 0. Высокое значение этой метрики говорит о высокой связности методов — это означает, что потребуются большие усилия при тестировании этих методов, так как методы могут воздействовать на одни и те же атрибуты класса. Это также говорит о низкой готовности к повторному использованию. Метрика была определена Чидамбером и Кемерером (Chidamber & Kemerer) в 1993.
- LOCOM2 (Lack Of Cohesion Of Methods 2) Связанность методов — мера насыщенности абстракции. Класс, который может вызывать существенно больше методов, чем равные ему по уровню классы, является более сложным. У класса с низкой связанностью можно подозревать случайную или неподходящую абстракцию: такой класс должен быть переабстрагирован на несколько классов или его обязанности должны быть переданы другим существующим классам. Метрика подсчитывает процентное отношение методов, не имеющих доступа к специфичным атрибутам класса ко всем атрибутам данного класса. Высокое значение связности означает, что класс хорошо спроектирован. Хорошо связный класс имеет тенденцию к высокой степени инкапсуляции, тогда как отсутствие

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

связности уменьшает инкапсуляцию и увеличивает сложность.

- LOCOM3 (Lack Of Cohesion Of Methods 3) Измеряет степень различия методов в классе по атрибутам. Рассмотрим множество методов M1, M2, ... , Mm. Эти методы имеют доступ к набору атрибутов данных класса A1, A2, ... , Aa. Положим a(Mk) = число атрибутов с которым работает метод Mk , а m(Ak) = число методов которые имеют доступ к атрибуту Ak. Тогда LOCOM3 = (1/a * SUM1a(m(Ai)-m))/(1-m)*100. Данная метрика была предложена Хендерсоном-Шеллером в 1995. Низкое значение этой меры говорит о хорошей декомпозиции в классе выражаемой в его простоте и понятности и готовности к повторному использованию. Высокое значение отсутствия связности увеличивает сложность, повышает вероятность ошибок в процессе разработки.

В эту группу метрик также попадают базовые метрики:

- LOC (Lines Of Code) количество строк кода
- NOC (Number Of Classes) количество классов

Улучшение значений метрических характеристик из этой группы говорит о положительном эффекте от применения оптимизирующего метода/подхода. Например, уменьшение значений метрик Холстеда, цикломатической сложности говорит о том, что полученная реализация в целом лучше по соображениям топологической сложности. Уменьшение уровня связности, взвешенности метода на класс или отклика класса говорит об улучшенной функциональной декомпозиции. Уменьшение количества строчек кода также говорит о положительном эффекте, если ту же функциональность можно изложить при помощи нового улучшающего подхода за меньшее количество строк.

Данная группа метрик будет собрана при помощи автоматизированного программного средства — TogetherJ. Настоящая (6 версия) продукта TogetherJ не поддерживает сбор метрик для аспектных модулей языка AspectJ, следовательно, в случае аспектной реализации собранные метрики будут подсчитываться только для компонентов реализованных на языке Java, то есть для "чистых" компонентов без учета сквозной функциональности. Можно утверждать, что данная грубая оценка будет верна в случае не больших тестовых проектов, где отношение разницы кода объектной реализации и кода компонента к коду сквозной функциональности (например, в терминах LOC) не будет значительно превышать 1, что будет говорить о небольшом количестве вынесенной сквозной функциональности, но достаточном для того, чтобы оценить

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

модульность полученных компонент в терминах топологической и информационной сложности. Сравнение метрик компонента до и после вынесения сквозной функциональности, позволит оценить, в какую сторону изменилась реализация данного компонента. Для учета топологической сложности вынесенной сквозной функциональности и для анализа больших реальных проектов необходимо вырабатывать другие метрики анализа.

7.2. Уровень языковых средств и их применения

Для оценки степени применимости аспектного подхода к конкретной программной системе введем следующую метрику:

$$Adaptability = \frac{Q-P}{Z}$$

где P — размер кода программной системы без применения аспектного подхода;
Q — размер кода компонентов реализующих основную логику системы;
Z — размер кода аспектных модулей реализующих сквозную функциональность.

В результате любое число больше чем 1 говорит о положительном эффекте от применения аспектного подхода к данной программной системе.

7.3. Трудность восприятия и понимания программных текстов

Данные метрики не имеют количественной оценки и характеризуют наличие рассматриваемого свойства или признака у объекта, в частности у программного модуля, без учета градаций по данному признаку.

- Модульность. Сквозная функциональность может находиться в аспектных модулях, не затрагивая при этом модули, реализующие основные функциональные требования, то есть все зависимости между кодом реализованном в аспекте и компонентами-участниками локализованы в коде аспекта. Участвующие компоненты абсолютно свободны от контекста и как следствие эти компоненты полностью готовы к повторному использованию. В [9] модульность описывается

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

как "свойство системы, которая была разложена на внутренне связные, но слабо связанные между собой модули". Правильное разделение программы на модули является почти такой же сложной задачей, как выбор правильного набора абстракций.

- Готовность к повторному использованию. Компоненты, которые не связаны между собой специфической для данной программной системы сквозной функциональностью, легко могут быть использованы повторно. Причем в разных частях системы один компонент может играть разные роли, в зависимости от аспектных модулей. Кроме того, код аспектов и их составных частей также можно повторно использовать посредством наследования и это дает еще более широкие возможности.
- Понятность кода. Поскольку сквозная функциональность вынесена в аспектные модули, композиция классов в системе выглядит более понятной, так как она не обременена кодом, не относящимся непосредственно к сущности, которую представляет класс.

Анализируя данный набор метрик для каждого конкретного случая можно сделать вывод о применимости или неприменимости аспектного подхода на базе приведенных далее примеров.

8. Варианты применения АОП на разных этапах ЖЦ

Приведем варианты использования аспектно-ориентированного подхода на различных этапах жизненного цикла программной системы.

8.1. Использование АОП на этапе проектирования

Как отмечает Дейкстра, "Способ управления сложными системами был известен еще в древности — divide et imperia (разделяй и властвуй)" [10]. При проектировании сложной программной системы необходимо разделять ее на все меньшие и меньшие подсистемы, каждую из которых можно совершенствовать независимо. Декомпозиция вызвана сложностью программирования системы, поскольку именно эта сложность вынуждает делить пространство состояний системы. При построении системы необходимо провести алгоритмическую декомпозицию, объектно-ориентированную декомпозицию и аспектно-ориентированную. Объектно-ориентированная

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

декомпозиция позволяет выделить из требований компоненты, описывающие проблемную область, алгоритмическая — описать взаимодействие этих компонентов и сконцентрировать внимание на порядке происходящих событий [9]. Аспектно-ориентированная декомпозиция на этапе анализа и проектирования позволяет выделить сквозную функциональность на разных уровнях абстракции и локализовать ее в отдельных модулях-аспектах. Такие аспекты являются неотъемлемой частью результирующей программной системы.

8.1.1. Реализация протокола взаимодействия объектов для шаблонов проектирования

Шаблоны проектирования — это весьма ценное инструментальное средство в арсенале разработчика ПО, позволяющее существенно повысить эффективность создаваемого кода. Под шаблоном проектирования в [6] понимается следующее: "Шаблон проектирования — это описание взаимодействия компонентов, адаптированных для решения общей задачи проектирования в конкретном контексте. Шаблон проектирования именует, абстрагирует и идентифицирует ключевые аспекты структуры общего решения, которые и позволяют применить его для создания повторно используемого дизайна. Он вычленяет участвующие компоненты, их роль и отношения, а также функции. При описании каждого шаблона внимание акцентируется на конкретной задаче объектно-ориентированного проектирования".

Однако, исходя из приведенного определения и личного опыта применения шаблонов проектирования, можно сделать следующий вывод: несмотря на то, что шаблоны проектирования являются средством, увеличивающим способность кода к повторному использованию, и абстрактная модель шаблона проектирования может быть неоднократно использована, конкретную реализацию шаблона повторно удается использовать очень редко, так как она сильно зависит от контекста применения. К тому же, участвующие в реализации шаблона компоненты использовать в другом контексте практически невозможно, так как они сильно привязаны друг к другу в контексте поведения реализуемого шаблона. Реализация приводит к "растворению в коде" абстрактной модели шаблона и потере модульности, при этом достаточно сложно из конкретной реализации выделить абстрактную модель шаблона. Добавление и удаление реализации шаблона проектирования из кода системы является достаточно сложной задачей, которая влечет за собой большой рефакторинг программного кода.

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

Каталог шаблонов проектирования [6] включает в себя 23 известных шаблона, их назначение, сценарии, иллюстрирующие задачи проектирования и то, как они решаются при помощи конкретного шаблона, описание ситуаций, в которых можно применить шаблон, и графическое представление компонентов в шаблоне. В работе [5] проведен сравнительный анализ аспектной реализации шаблонов проектирования и сравнение с "традиционной" объектно-ориентированной реализацией. В данной работе показаны улучшения для 17 из 23 шаблонов. Улучшения выражаются в терминах метрик с номинальной шкалой — модульности кода, готовности к повторному использованию, и понятности кода. Под модульностью кода в этой работе подразумевается локализация сквозной функциональности — в данном случае абстрактной модели шаблона, в коде аспекта. Под готовностью кода к повторному использованию — возможность использовать компоненты-участники и аспектные модули в других контекстах, так как для шаблона был разработан общий протокол взаимодействия компонентов-участников. Под понятностью кода подразумевается то, что код реализации не является запутанным. Это означает, что компоненты-участники могут быть заняты в нескольких шаблонах проектирования, шаблоны могут разделять между собой компоненты-участники. Также в этой работе введена еще одна метрика — способность к встраиванию, которая характеризует возможность удалить или добавить в систему аспектный код, реализующий данный шаблон, без усилий и модификации компонентов-участников шаблона, что говорит о готовности данных компонентов к повторному использованию в других контекстах. В таблице 2 приведены полученные в [5] результаты сравнения реализаций шаблонов проектирования.

Имя шаблона	Модульность	Готовность к повторному использованию	Понятность кода	Способность к встраиванию
Facade	аспектная и объектная реализации идентичны			
Abstract Factory	o	o	o	o
Bridge	o	o	o	o
Builder	o	o	o	o
Factory Method	o	o	o	o
Interpreter	o	o		o

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

Template Method	#	o	o	#
Adapter	x	o	x	x
State	#	o		#
Decorator	x	o	x	x
Proxy	#	o	#	#
Visitor	#	x	x	#
Command	#	x	x	x
Composite	x	x	x	#
Iterator	x	x	x	x
Flyweight	x	x	x	x
Memento	x	x	x	x
Strategy	x	x	x	x
Mediator	x	x	x	x
Chain of responsibility	x	x	x	x
Prototype	x	x	#	x
Singleton	x	x		x
Observer	x	x	x	x

Таблица 2. Свойства модульности у аспектной реализации шаблонов проектирования.

Символом o в таблице показано отсутствие свойства у реализации, символом x — наличие свойства. Символ # означает наличие свойства с некоторыми ограничениями.

Рассмотрим пример реализации шаблона Observer. Шаблон Observer определяет зависимость "один ко многим" между объектами так, чтобы при изменении состояния одного объекта все зависящие от него оповещаются об этом и автоматически обновляются. Этот шаблон применяется в тех случаях, когда система обладает следующими свойствами:

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

- Существует, как минимум, один объект, рассылающий сообщения
- Имеется не менее одного получателя сообщений, причем их количество и состав может изменяться во время работы приложения.

Диаграмма классов шаблона Observer представлена на рисунке 14.

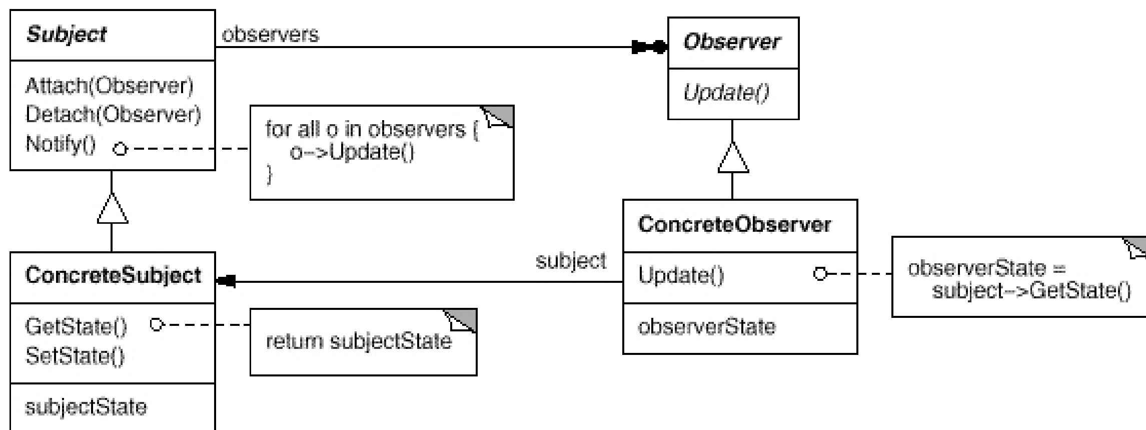


Рисунок 14. Диаграмма классов шаблона Observer.

Составными частями данного шаблона являются

- Subject(Объект) Знает своих наблюдателей. Любое количество объектов может наблюдать за объектом. Обеспечивает интерфейс для присоединения и отсоединения объектов Observer.
- Observer(Наблюдатель) Определяет интерфейс модифицирования объектов.
- ConcreteSubject (Конкретный объект) Состояние, представляющее интерес для объектов ConcreteObserver. Посылает уведомление своим наблюдателям, когда его состояние изменяется.
- ConcreteObserver (Конкретный наблюдатель) Поддерживает ссылку на объект ConcreteSubject. Сохраняет состояние, которое должно остаться совместимым с состоянием объекта. Реализует Observer-а, модифицирующего интерфейс.

Затруднения при использовании данного шаблона вытекают из реализуемой в нем модели построенной на обмене сообщениями. Можно использовать как специальную, так и универсальную стратегию рассылки сообщений, каждая из которых имеет свои недостатки.

При использовании универсальной стратегии сложнее определить, что конкретно

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

происходит с наблюдаемым объектом, поскольку сообщения носят универсальный характер. Кроме того, универсальная стратегия характеризуется избыточным потоком сообщений — некоторые события, пересылаемые наблюдателям, последними никак не обрабатываются, а лишь влекут за собой увеличение накладных расходов. В качестве примера применения данного шаблона и демонстрации увеличения накладных расходов при обработке лишних сообщений можно привести пример графического редактора, обсуждаемого ранее. Вызов обновления дисплея Display.update(this) может происходить, как и при вызове метода установки точки — setX, так и при вызове setXY или moveBy которые инкапсулируют в себе setX. Наконец универсальная стратегия требует дополнительных затрат на разработку классов наблюдателей, так как им нужно анализировать сообщения и вычленять из них необходимую информацию.

При специальной стратегии сообщения, направленные на решение узкой задачи обмена информацией наблюдаемого объекта с наблюдателями, выдвигают повышенные требования относительно программирования первого, поскольку он должен генерировать последовательности извещений при возникновении определенных условий. Это может повлечь за собой и увеличение сложности объектов-наблюдателей, так как им придется обрабатывать сообщения разных типов.

Для "традиционной" объектно-ориентированной реализации проведен расчет метрических характеристик при помощи автоматизированного программного средства. Значения представлены в таблице 3.

Рассмотрим аспектную реализацию данного шаблона. В структуре шаблона Observer можно выделить части, которые являются общими для всех экземпляров этого шаблона, также можно выделить специфичные для каждой конкретной реализации. На рисунке 15 представлена модель шаблона Observer реализованного средствами АОП. На диаграмме присутствует абстрактный аспект, который инкапсулирует в себе реализацию общих повторно используемых частей для всех реализаций шаблона и определяет поведение шаблона, конкретные расширения абстрактного аспекта содержат части специфичные для конкретной реализации.

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

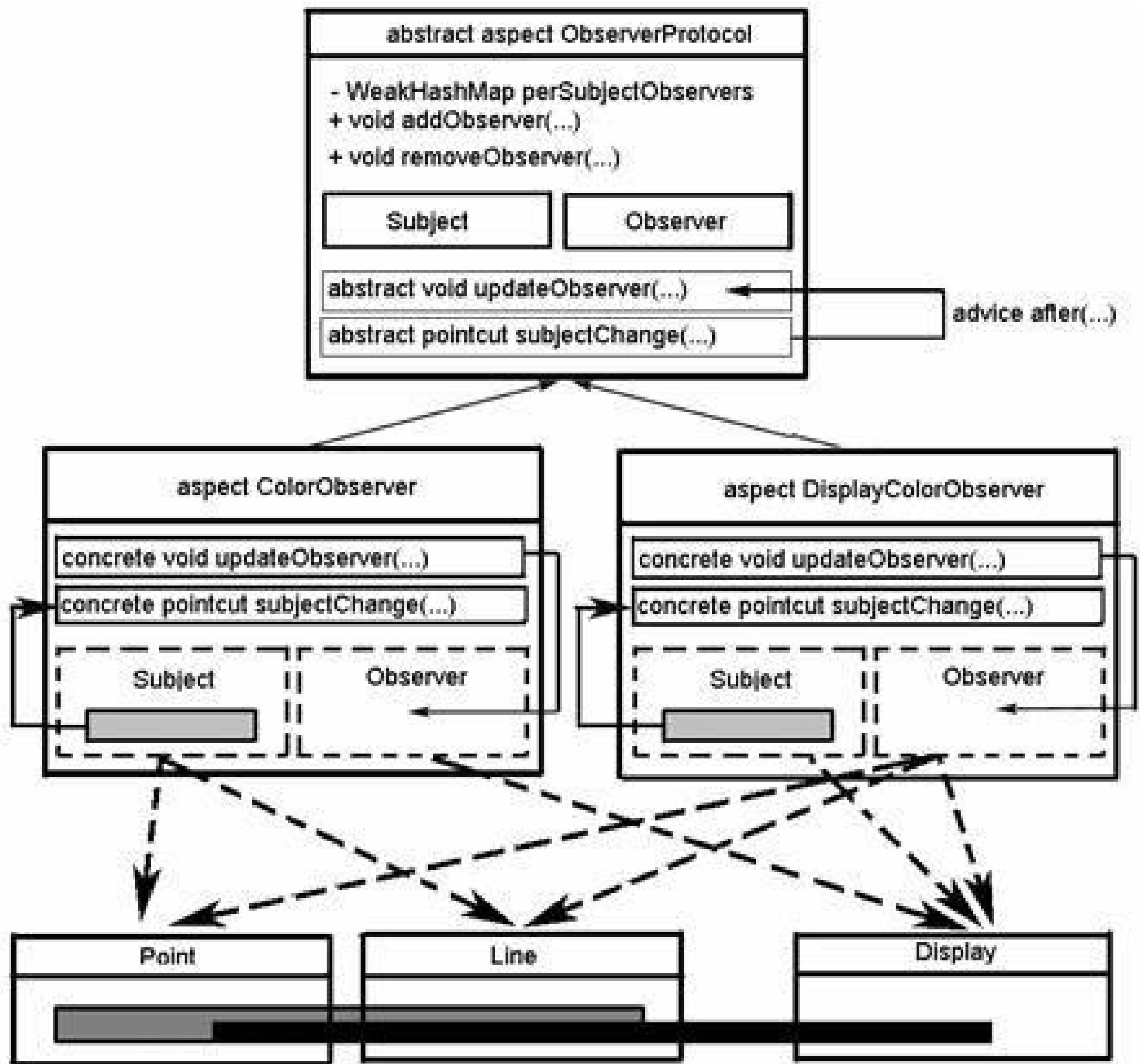


Рисунок 15. Модель шаблона Observer.

Общими частями в структуре шаблона Observer являются:

- Существование ролей Subject и Observer. Роли реализованы как вложенные интерфейсы Subject и Observer в абстрактный аспект. Их главное назначение разделять роли Observer и Subject, для которых в абстрактном аспекте определяется поведение в шаблоне. Конкретные реализации шаблона установят эти роли на конкретные классы.

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

- Отображение Subject на Observer. Реализовано в виде хеш-таблицы (perSubjectObservers) связанных списков для хранения связи объектов типа Observer для каждого объекта Subject. Каждое расширение этого абстрактного аспекта будет иметь свой собственный экземпляр отображения. Внесение изменений в отображение Observer-Subject реализовано посредством методов addObserver() и removeObserver(), которые могут быть вызваны для установления соответствия между объектами Observer и Subject.
- Логика оповещения. В повторно используемом абстрактном аспекте логика оповещения определяется общая концепция, по которой могут измениться объекты Subject (abstract pointcut subjectChange), что должно привести к обновлению всех его обозревателей (вызов абстрактного метода updateObserver для всех обозревателей измененного объекта Subject). Здесь не вводится понятие того, что может привести к обновлению объекта Subject или то каким образом должны быть обновлены объекты Observer. Здесь определяется набор абстрактных точек выполнения программы, которые могут привести к изменению состояния объекта Subject и после изменения состояния вызвать оповещение соответствующих объектов Observer. Определяется абстрактный механизм обновления объектов обозревателей — выделяется сквозная функциональность на этом уровне абстракции.

Каждая конкретная реализация шаблона определяет собственный вид отслеживания взаимосвязей между объектами. В конкретной реализации абстрактного аспекта определяется следующее:

- Определяются компоненты, которые играют роли Subject и Observer в этой конкретизации шаблона Observer.
- Переопределяются точки выполнения программы, объявленные в родительском аспекте как абстрактные, влияющие на обновление связанных с Subject объектов Observer посредством выделения желаемого поведения из объектов, играющих роль Subject
- Реализуется механизм уведомления и обновления объектов обозревателей посредством конкретизации метода updateObservers()

На рисунке 15 представлено две различных реализации шаблона Observer с использованием компонентов Point, Line и Screen. В первом случае компоненты Point и Line играют роль Subject, а компонент Screen играет роль Observer. Во втором Point и

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

Line играют роль Observer, а Display — Observer и Subject.

В аспектной реализации шаблона Observer весь код касающийся взаимосвязи между ролями Observer и Subject перемещен в аспекты и компоненты, играющие эти роли, не связаны друг с другом.

Для аспектно-ориентированной реализации также проведен расчет метрических характеристик, и результат расчета представлен в таблице 3.

Метрика	Объектно-ориентированная реализация	Аспектно-ориентированная реализация
Цикломатическая сложность (CC)	11	7
Сложность программы (HDiff)	16	9
Усилия разработчика (HEff)	7319	3563
Длина программы (HPLen)	283	190
Словарь программы (HPVoc)	37	35
Объем программы (HPVol)	1434	898
Количество строк кода (LOC)	109	56
Недостаток связности методов (LOCOM1)	15	3
Недостаток связности методов (LOCOM2)	68	57
Недостаток связности методов (LOCOM3)	75	66
Количество классов (NOC)	5	3
Общее число операндов в программе (Noprnd)	149	96
Общее число операторов в программе (Noprtr)	134	94
Количество вызываемых	9	5

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

удаленных методов (NORM)		
Число уникальных операндов программы (NUOprnd)	33	31
Число уникальных операторов программы (NUOprtr)	7	4
Отклик на класс (RFC)	17	7
Взвешенная насыщенность класса (WMPC1)	11	7
Взвешенная насыщенность класса (WMPC2)	18	13

Таблица 3. Метрические характеристики реализаций шаблона Observer.

Степень применимости аспектного подхода: Adaptability = 1,308

Исходя из полученных результатов и результатов оценки по номинальной шкале в [5] можно сказать о положительном эффекте от применения аспектного подхода для реализации данного шаблона проектирования.

8.1.2. Реализация авторизованного доступа к данным

Одним из важных показателей надежности системы является уровень защиты. Поскольку корпоративные системы играют серьезную роль в организации деятельности предприятий, потери вследствие нарушения защиты могут быть существенными. Проблемы безопасности возникают вследствие деятельности конкретных людей, и многообразие этих проблем ограничивается лишь человеческой изобретательностью, следовательно, обеспечение защиты системы представляет собой чрезвычайно важную задачу. Кроме того, решение проблемы защиты требует значительных усилий.

В системах безопасности термин принципал определяет пользователя или иного отправителя либо получателя сообщения. Идентификация принципала является одной из основных задач, выполняемых средствами защиты. Например, принимая решение о том, следует ли разрешить доступ к определенному ресурсу, система должна обязательно установить личность принципала, желающего получить этот доступ. Процесс установления личности принципала на доступ к ресурсам состоит из двух

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

частей — аутентификации (идентификации) и авторизации. Аутентификация только идентифицирует пользователя, проверяет тот ли он, за кого себя выдает. Авторизация — это проверка для уже идентифицированного пользователя, имеет ли он доступ к определенному ресурсу или право на выполнение определенной операции. Причем первая составляющая присутствует во всех системах, а вторая — только в системах, где нужно предоставлять разный доступ разным пользователям.

В качестве демонстрационного примера авторизации и аутентификации используем расширение JAAS (Java Authentication and Authorization service) платформы Java которое разрабатывалось для того, чтобы обеспечить стандартный способ ограничения доступа к ресурсам, основанный на аутентификации пользователей. API для регистрации и завершения работы в системе, предоставляемые JAAS, также обеспечивают стандартные средства аутентификации пользователей и передачу информации о контексте защиты и полномочиях. Модель поставщика службы JAAS позволяет работать с различными базовыми средствами аутентификации и авторизации.

Рассмотрим простую банковскую систему

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

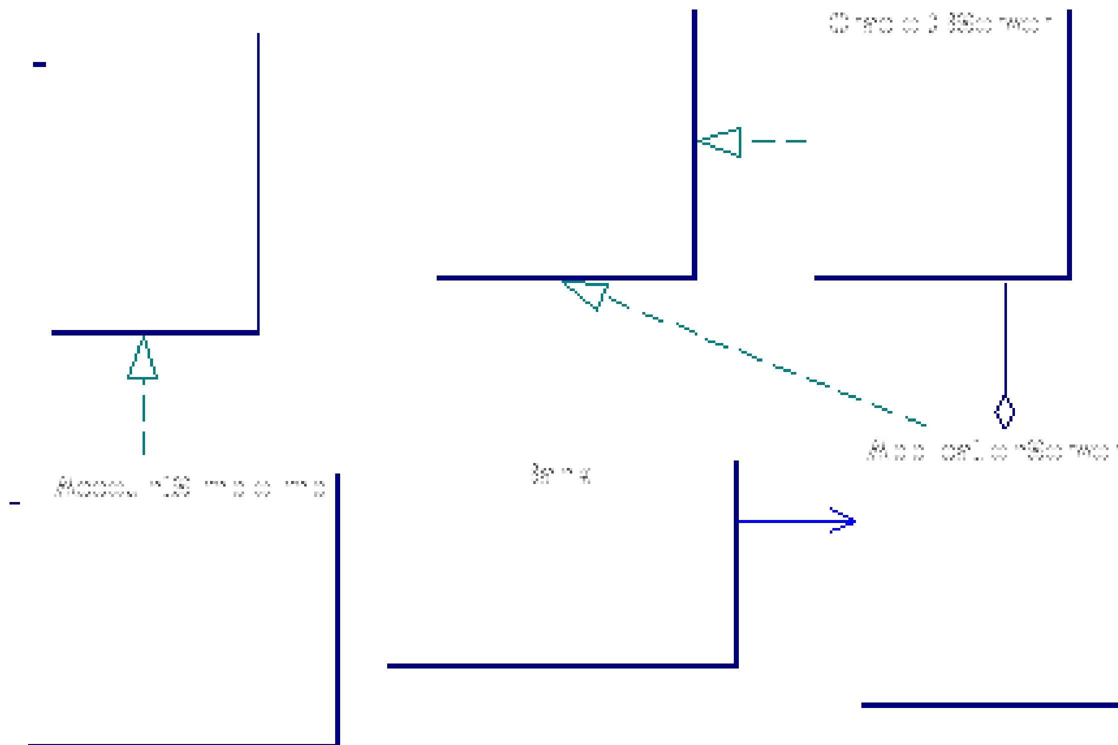


Рисунок 16. Модель банковской системы.

Ресурсами, которые нуждаются в защите, являются сервера счета пользователей. Следовательно, перед вызовом методов каждого из этих объектов необходимо удостоверится в том, что пользователь имеет право работать с данной системой. Для этого перед каждым вызовом метода защищаемого объекта необходимо проверить права пользователя, если этого еще не было сделано. Объектом, инкапсулирующим API JAAS будет объект Autentificator:

```
public class Autentificator {
  private static Subject _authenticatedSubject=null;
  public static void perform(){
         if(_authenticatedSubject != null) {
            return;
        }
        try {
```


Copyright © 1999 - 2007, IT-archiv. All rights reserved.

```
            authenticate();
        } catch (LoginException ex) {
            throw new AuthenticationException(ex);
        }
  }
  private static void authenticate() throws LoginException {
        LoginContext lc = new LoginContext("Sample",
                                           new TextCallbackHandler());
        lc.login();
        _authenticatedSubject = lc.getSubject();
  }
}
```

метод perform() выполняет аутентификацию принципала, и если он не проходит возбуждается исключительная ситуация. Метод perform должен быть встроен в начало каждого защищаемого метода.

Для некоторых систем выдвигаются более сильные требования по безопасности, когда кроме аутентификации необходимо проверять права пользователя на вызов конкретного метода — авторизацию. Вызов, например, метода кредитования объекта Account может выглядеть следующим образом:

```
Subject.doAsPrivileged(authenticatedSubject,
                    new PrivilegedAction() {
                        public Object run() {
                            account1.credit(300);
                            return null;
                        }}, null);
            try {
                Subject
                    .doAsPrivileged(authenticatedSubject,
                        new PrivilegedExceptionAction() {
                            public Object run() throws Exception {
                                account1.debit(200);
                                return null;
                            }}, null);
            } catch (PrivilegedActionException ex) {
                Throwable cause = ex.getCause();
                if (cause instanceof InsufficientBalanceException) {
                    throw (InsufficientBalanceException)ex.getCause();
```


Copyright © 1999 - 2007, IT-archiv. All rights reserved.

```
                }
}
```

Как видно из приведенного исходного кода даже при использовании специально спроектированного API могут возникнуть проблемы с восприятием трудно читаемого кода. Кроме того, функциональность авторизации и аутентификации должна быть встроена в методы нуждающиеся в защите, что однозначно приведет к перемешиванию требований в коде и потере модульности компонентов бизнес логики, которые будет невозможно использовать в другом контексте. Функциональность проверки подлинности пользователя, которая должна быть встроена в бизнес логику, является ортогональной по отношению к основным требованиям системы.

При использовании АОП подобную сквозную функциональность можно вынести в аспектный модуль на этапа проектирования системы, который будет инкапсулировать в себе эту функциональность.

На рисунке 17 представлена диаграмма аспектов модели безопасности. В абстрактный аспект AbstractAuthAspect выносится логика аутентификации и авторизации. Определяется набор инструкций языка, который должен быть вставлен до защищаемого кода — адвайзер аутентификации before():authOperations(), в котором будет выполняться закрытый метод данного аспекта — authenticate(). Этот встраиваемый код будет выполнен перед точками выполнения программы из набора authOperations.

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

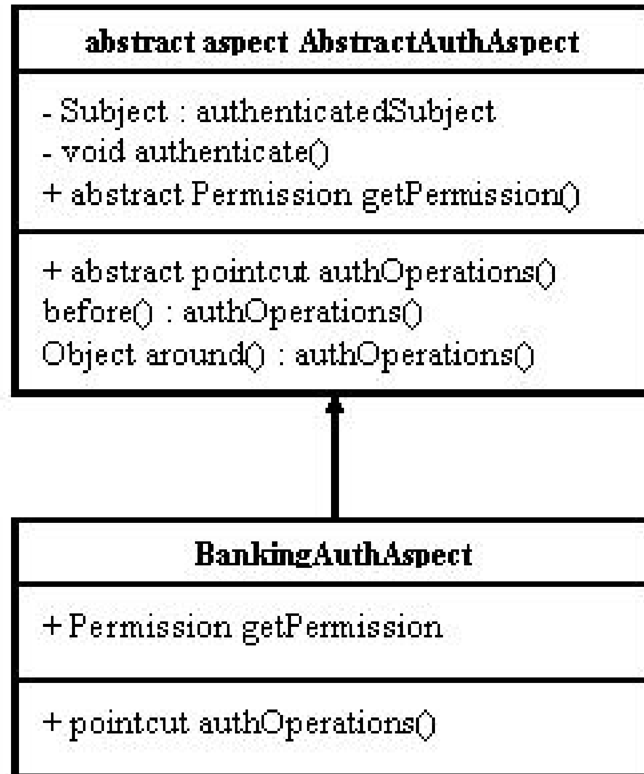


Рисунок 17. Модель безопасности в виде аспектов.

В аспекте определяется набор инструкций, который будет непосредственно вызван вместо защищаемого кода — адвайзер авторизации Object around(): authOperations(), который инкапсулирует в себе вызываемый код, что позволяет перед его выполнением выполнить код службы авторизации и в случае успеха авторизации позволить выполнить защищаемый код. Данный код будет выполнен вместо точек выполнения программы из набора authOperations. В абстрактном аспекте определен принципал — authenticatedSubject для которого в конкретном аспекте будут получаться права путем вызова переопределенного метода getPermission(). В конкретном аспекте должен быть определен набор точек связывания — точек интеграции аспектного кода (authOperations). В случае BankingAuthAspect такими точками являются вызовы публичных методов компонентов Account, Server и Bank. Во все эти методы интегрируется функциональность описанная в аспектном модуле, при этом сами

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

компоненты останутся не тронутыми с точки зрения исходного кода, то есть код этих компонентов останется не затронутым сквозной функциональностью и они будут представлять только свою сущность без кода, не относящегося к ним по смыслу, что значительно улучшает модульность системы и повышает способность кода к повторному использованию. Вся сквозная функциональность будет вынесена в аспектный модуль. Логика аутентификации и авторизации описанная в абстрактном аспекте может быть также повторно использована в других контекстах путем наследования аспектов.

Для аспектно-ориентированной и объектно-ориентированных реализаций требований по авторизации к банковской системе проведен расчет метрических характеристик, и результат расчета представлен в таблице 4.

Метрика	Объектно-ориентированная реализация	Аспектно-ориентированная реализация
Цикломатическая сложность (CC)	8	5
Сложность программы (HDiff)	22	11
Усилия разработчика (HEff)	7314	1255
Длина программы (HPLen)	212	90
Словарь программы (HPVoc)	32	16
Объем программы (HPVol)	934	328
Количество строк кода (LOC)	148	77
Недостаток связности методов (LOCOM1)	8	8
Недостаток связности методов (LOCOM2)	60	60
Недостаток связности методов (LOCOM3)	0	0
Количество классов (NOC)	6	8
Общее число операндов в	99	50

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

программе (Noprnd)		
Общее число операторов в программе (Noprtr)	113	40
Количество вызываемых удаленных методов (NORM)	10	8
Число уникальных операндов программы (NUOprnd)	23	12
Число уникальных операторов программы (NUOprtr)	9	5
Отклик на класс (RFC)	20	19
Взвешенная насыщенность класса (WMPC1)	8	5
Взвешенная насыщенность класса (WMPC2)	8	8

Таблица 4. Метрические характеристики реализаций Авторизации.

Степень применимости аспектного подхода: Adaptability = 1,73

Исходя из полученных результатов и видимых улучшений модульности компонентов можно сказать о положительном эффекте от применения аспектного подхода для реализации модели безопасности в виде аспектного модуля. Такая модель безопасности применяется на этапе проектирования системы и аспектные модули реализующие такую функциональность будут частью разрабатываемой системы. Изменения требований по безопасности в системе легко могут быть внесены без модификации отлаженной логики системы путем модификации только аспектного модуля.

8.1.3. Ведение журнала событий

Типичная проблема, которая может возникнуть перед системным архитектором при анализе требований и проектировании системы — это необходимость предусмотреть потенциально возможные требования заказчика. Однако, как получается на практике, принимать решения по такой проблеме заранее очень сложно и обеспечить поддержку возможных еще не существующих требований со стороны архитектуры системы

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

является трудновыполнимой задачей. Перед архитектором возникает дилемма — либо уделить больше внимание архитектуре системы при проектировании и попытаться заложить в нее возможность расширения, либо отложить расширение архитектуры до момента реального появления подобных требований. И первый и второй варианты не являются идеальными. Поддержка со стороны архитектуры чего-то неявного и пока не существующего практически не возможна, даже если предусмотреть какую-то область требований и сделать поддержку расширения системной архитектуры под эту область — всегда может возникнуть требование заказчика, которое не ложится на заранее принятое решение.

Примером подобных требований может служить требование по ведению журнала событий. Если не обеспечить поддержку данного требования на этапе проектирования, то впоследствии потребуется значительные усилия чтобы обновить все события, происходящие в системе на предмет записи в журнал. При этом модифицированный код компонентов становится непригодным в другом контексте, так как он будет привязан к логике "встраиваемого" в него журнала событий. Вызова же методов журнала событий будут рассредоточены по всей системе, что мешает восприятию кода компонентов. В результате в коде компонентов можно увидеть симптомы запутанного и рассредоточенного кода описанные выше.

```
public class Main {
       public Main() {
       }
       public void foo() {
           Logger.entry("foo()");
           System.out.println("foo1");
           Logger.exit("foo()");
       }
       public void foo(int i) {
           Logger.entry("foo(int)");
           System.out.println(i++);
           Logger.exit("foo(int)");
       }
       public double bar(double x, double y) {
           Logger.entry("bar(double, double)");
           double result  = x * y;
           Logger.exit("bar(double, double)");
```


Copyright © 1999 - 2007, IT-archiv. All rights reserved.

```
            return result;
        }
        public static void main(String[] args) {
            Logger.entry("main(String[])");
            Main main1 = new Main();
            main1.foo();
            System.out.println(main1.bar(1.2, 1.3));
            main1.foo(10);
            Logger.exit("main(String[])");
        }
}
```

При использовании аспектного подхода такой ситуации можно избежать. На этапе проектирования системы можно легко отложить подобные сквозные требования, если они уже существуют или находятся на стадии обсуждения и сосредоточится на проектировании основной логики системы. После окончания проектирования системы можно вернуться к сквозным требованиям и описать их в терминах аспектов — отдельных модулей, в которых будет описана логика этих требований и правила их интеграции в основную логику компонентов.

```
public aspect  AutoLog {
    pointcut publicMethods(): execution(public * *..*(..));
    pointcut logObjectCalls(): execution(* Logger.*(..));
    pointcut loggableCalls(): publicMethods() && (!logObjectCalls());
    before(): loggableCalls() {
        Logger.entry(thisJoinPoint.getSignature().toString());
    }
    after(): loggableCalls() {
        Logger.exit(thisJoinPoint.getSignature().toString());
    }
}
```

При этом стоит заметить, что компоненты системы останутся свободными от сквозной функциональности и их легко можно использовать в другом контексте и их код будет более понятен, чем в случае жесткой привязки к сквозным требованиям.

В таблице 5 приведены значения метрических характеристик для двух типов реализаций.

Метрика	Объектно-ориентированная	Аспектно-ориентированная

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

	реализация	реализация
Цикломатическая сложность (CC)	7	5
Сложность программы (HDiff)	15	5
Усилия разработчика (HEff)	6258	1001
Длина программы (HPLen)	172	44
Словарь программы (HPVoc)	31	23
Объем программы (HPVol)	830	199
Количество строк кода (LOC)	67	20
Недостаток связности методов (LOCOM1)	1	-
Недостаток связности методов (LOCOM2)	33	-
Недостаток связности методов (LOCOM3)	0	-
Количество классов (NOC)	2	1
Общее число операндов в программе (Noprnd)	92	23
Общее число операторов в программе (Noprtr)	80	21
Количество вызываемых удаленных методов (NORM)	7	2
Число уникальных операндов программы (NUOprnd)	24	16
Число уникальных операторов программы (NUOprtr)	7	7
Отклик на класс (RFC)	10	7
Взвешенная насыщенность класса (WMPC1)	7	5

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

Взвешенная насыщенность класса (WMPC2)	9	9

Таблица 5. Метрические характеристики реализаций журнала событий.

Степень применимости аспектного подхода: Adaptability = 2.9

Исходя из полученных результатов и видимых улучшений модульности компонентов, избавления от растворения в коде сквозной функциональности, повышения готовности к повторному использованию кода компонентов и возможности принимать решения о добавлении функциональности в систему, после того как была спроектирована архитектура, можно сделать вывод о положительном эффекте от применения аспектного подхода. Изменения требований для подобной функциональности не затронут отлаженную логику компонентов, причем подобную функциональность достаточно легко изымать и добавлять, что, в общем, повышает гибкость системы в целом и позволяет проще управлять подобными требованиями и их возникновением после окончания разработки архитектуры.

8.2. Использование АОП на этапе разработки системы

Разработка программного продукта — итеративный и последовательный процесс. За время разработки происходит многократное возвращение к каждой стадии процесса разработки, и каждый раз конечный результат улучшается на каждой стадии. В процессе разработки всегда решаются такие задачи как профилирование, трассировка, соблюдение проектных соглашений, слежение за корректностью входных и выходных данных на разных уровнях абстракции, отслеживается поведение объектов в многопоточной среде, применяются различные подходы к разработке повторно используемых компонент и стратегии их повторного использования. На этапе разработки системы существенную помощь разработчику могут оказать вспомогательные аспекты

8.2.1. Профилирование

В процессе разработки программной системы часто возникает необходимость проведения измерительных экспериментов как непосредственно для оценки характеристик системы, так и для проверки результатов, полученных на основе аналитического и имитационного моделирования. Для этих целей используются

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

специальные инструменты-профилировщики — программные средства, позволяющие получить ряд количественных данных о процессе выполнения программы и на основании этих данных выявить в ней "узкие места", отрицательно сказывающиеся на эффективности ее работы.

Профилировщик позволяет получить следующую информацию о процессе выполнения программы:

- как и на что расходуется время работы программы;
- сколько раз выполняется данная строка программы;
- сколько раз и какими модулями вызывается данный модуль программы;
- к каким файлам обращается программа и сколько времени она на это тратит.

К сожалению, стандартные инструментальные средства, предназначенные для этих целей, могут либо не обеспечивать заданных параметров оценок, либо быть недоступными, либо вообще отсутствовать. Для внутренних проектных целей при возникновении потребности профилирования и отсутствии подходящего профилировщика можно воспользоваться классом-профилировщиком, интеграция которого в проектный код позволит решить проблему выявления "узких мест".

```
public class Profiler {
    public static boolean DEBUG_MODE = false;
    private static Map profiles = new Hashtable();
    public static void TIMING(String pointName){
        if (!DEBUG_MODE) return;
        long ts = System.currentTimeMillis();
        Object profile = profiles.get(pointName);
        if (profile==null){
            profiles.put(pointName,new Long(ts));
        }else{
            profiles.remove(pointName);
            ts = ts-((Long)profile).longValue();
            System.out.println("Profile of [" + pointName + "] is
"+ts+" ms");
        }
    }
}
```

Сквозной функциональностью в данном примере является снятие профиля в

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

конкретной точке выполнения программы. Интеграция подобного класса в проектный код хоть и решит проблему сбора профилей, но при этом код компонентов будет нагружен "лишним" вызовом профилировочного класса, который не требуется в релизе системы. Кроме того код разрабатываемых компонентов становится привязанным к коду профилировщика, следовательно, такое решение проблемы выявления "узких" мест не всегда хорошее решение и не всегда применимое решение.

При использовании аспектного подхода достаточно легко логику, реализующую снятие профиля с блока программы, поместить в аспектный модуль, в котором будут описаны правила интеграции подобной сквозной функциональности в код компонентов. В данном случае очень помогает метод позволяющий окружить требуемую точку выполнения программы кодом сквозной функциональности — around().

```
public aspect ProfilerAspect {
    public pointcut appMethod() :
        call(public * Application.*(..)) ;
    void around() : appMethod() {
        long ts = System.currentTimeMillis();
        proceed();
        ts = System.currentTimeMillis()-ts;
        Signature sig = thisJoinPointStaticPart.getSignature();
           System.out.println("Profile of [" + sig.getName() + "] is
        "+ts+" ms");
    }
}
```

При использовании набора инструкций окружающих требуемую точку выполнения программы достаточно легко встроить логику по вычислению расхода времени на выполнение данного участка программы, количества выполнения данного участка кода, определение источников вызова данного блока и многое другое. Причем основные компоненты системы ничего не будут знать о том, что с их кодом работает проектный профилировщик, который в свою очередь может быть легко удален из системы в финальной версии без модификации кода основных компонентов.

В таблице 6 приведены значения метрических характеристик для двух типов реализаций.

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

Метрика	Объектно-ориентированная реализация	Аспектно-ориентированная реализация
Цикломатическая сложность (CC)	63	62
Сложность программы (HDiff)	93	79
Усилия разработчика (HEff)	587453	580420
Длина программы (HPLen)	1413	1291
Словарь программы (HPVoc)	166	166
Объем программы (HPVol)	9857	9210
Количество строк кода (LOC)	362	329
Недостаток связности методов (LOCOM1)	66	66
Недостаток связности методов (LOCOM2)	92	92
Недостаток связности методов (LOCOM3)	0	0
Количество классов (NOC)	4	3
Общее число операндов в программе (Noprnd)	797	732
Общее число операторов в программе (Noprtr)	616	559
Количество вызываемых удаленных методов (NORM)	7	5
Число уникальных операндов программы (NUOprnd)	139	139
Число уникальных операторов программы (NUOprtr)	27	27
Отклик на класс (RFC)	14	14
Взвешенная насыщенность	63	63

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

класса (WMPC1)		
Взвешенная насыщенность класса (WMPC2)	58	58

Таблица 6. Метрические характеристики реализаций профилирования.

Степень применимости аспектного подхода: Adaptability = 2.8

Аспектный подход максимально упрощает процесс профилирования и увеличивает эффективность его применения. При использовании аспектного подхода можно легко все средства необходимые для снятия профилей поместить в аспектные модули, которые благодаря принципам данного подхода можно легко изменять и удалять без модификации кода компонентов. По трудности восприятия исходного кода можно сказать, что аспектный подход увеличивает модульность, повышается способность компонентов к повторному использованию и повышается понятность кода, который не "обременен" логикой профилирования.

8.2.2. Трассировка

Трассировка позволяет разработчику получать больший объем информации о внутренних операциях компонент программной системы, нежели предоставляется в журнальном файле, если он существует. Трассировка любой операции генерирует детальную последовательность предложений, которые описывают события по мере их возникновения. Вывод трассировки можно направить либо в выходные файлы трассировки, которые можно анализировать после сбоя, чтобы выяснить, какие события привели к этому сбою, либо на консоль при отладке приложения. Обычно потребность в трассировке возникает после появления ненормальной ситуации, когда журнальный файл не дает ясного указания на причину при отладке приложения. В чем разница между журнализацией и трассировкой? В журнал нет смысла включать детальную информацию о состоянии всех объектах системы, поэтому журнал регистрирует состояние программных компонентов, находящихся ближе к пользователю; трассировка предоставляет описание всех программных событий по мере их возникновения, и поэтому позволяет получить дополнительную информацию о тех событиях, которые предшествовали ошибке. Может существовать несколько уровней диагностики, каждый из которых предоставляет больше информации, чем предыдущий. Трассировка важна при отладке и при разработке специальных

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

приложений, когда нет специально предназначенного для этих целей автоматического отладчика.

Используя принципы АОП легко можно добавить логику по трассировке компонентов системы в программный код.

Например, при работе с базой данных при поиске ошибки можно создать аспект, который будет складывать в трассировочный файл выполняемые системой SQL запросы.

```
aspect DatabaseDebugging{
   private interface TypesDebugged{}
   declare parents : DataCollection1 ||
                            DataCollection2 ||
                                 ...
  DataCollectionN implements TypesDebugged;
  pointcut queryExecution(String sql):
          call(* Statement.*(String))
          this(TypesDebugged)
          args(sql);
  before(String sql): queryExecution(sql){
          System.out.println(sql);
  }
}
```

Для полного анализа событий проходящих в системе можно создать несколько аспектов инкапсулирующих в себе логику необходимую для сбора трассы. На рисунке 18 представлена иерархия аспектов. Абстрактный аспект Trace содержит в себе логику сбора трассы в зависимости от уровня диагностики. Аспект TraceMyClasses определяет классы, для которых будет собираться трасса; Данный аспект имеет точку входа, при запуске приложения начиная с которой запустится приложение Application, и для классов, описанных в TraceMyClasses, будет собираться трасса.

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

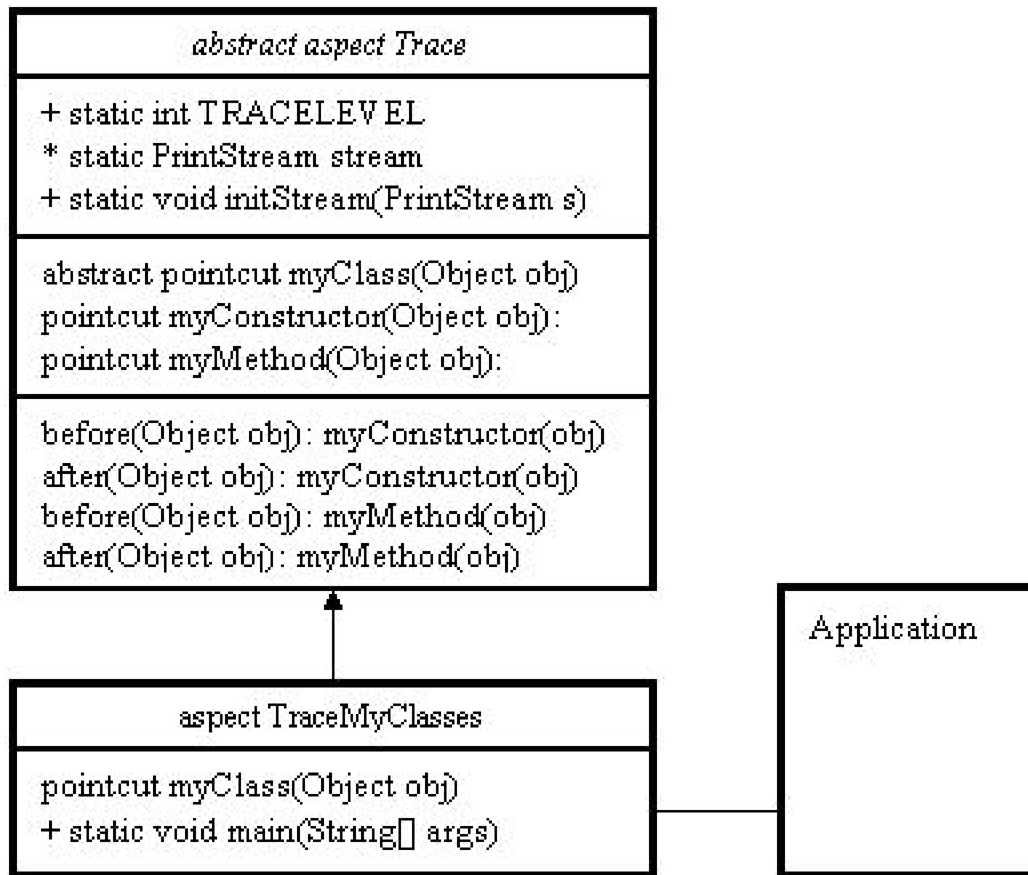


Рисунок 18. Трассировочные аспекты.

Использование АОП позволяет не интегрировать код сбора трассы в программные компоненты. Аспектные модули, инкапсулирующие такую функциональность, могут быть без усилий добавлены в программную систему при отладке и также легко могут быть удалены из нее. В таблице 7 приведены значения метрических характеристик для двух типов реализаций.

Метрика	Объектно-ориентированная реализация	Аспектно-ориентированная реализация
Цикломатическая сложность (CC)	16	7
Сложность программы (HDiff)	47	21

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

Усилия разработчика (HEff)	33410	6697
Длина программы (HPLen)	693	264
Словарь программы (HPVoc)	46	31
Объем программы (HPVol)	3730	1247
Количество строк кода (LOC)	289	85
Недостаток связности методов (LOCOM1)	37	9
Недостаток связности методов (LOCOM2)	73	43
Недостаток связности методов (LOCOM3)	60	25
Количество классов (NOC)	6	4
Общее число операндов в программе (Noprnd)	392	142
Общее число операторов в программе (Noprtr)	301	122
Количество вызываемых удаленных методов (NORM)	10	10
Число уникальных операндов программы (NUOprnd)	38	26
Число уникальных операторов программы (NUOprtr)	11	7
Отклик на класс (RFC)	19	11
Взвешенная насыщенность класса (WMPC1)	16	7
Взвешенная насыщенность класса (WMPC2)	26	13

Таблица 7. Метрические характеристики реализаций трассировки.

Степень применимости аспектного подхода: Adaptability = 2.25

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

8.2.3. Соблюдение контрактов

При оценке новых методов и средств разработки ПО обычно ориентируются на их производительность. Объектные технологии действительно могут существенно повысить производительность, при этом, однако, нельзя упускать из виду качество создаваемого ПО. Качественное ПО — это, прежде всего надежное ПО. Надежность — это способность системы функционировать в соответствии со спецификацией ("корректность") и при этом успешно справляться с возникающими ненормальными ситуациями ("устойчивость" — robustness). Другими словами, надежная программа не содержит ошибок.

Безусловно, надежность — это желательное качество ПО безотносительно к методу его разработки. Объектно-ориентированный подход предполагает повышенные требования к надежности — прежде всего из-за той особой роли, которую здесь играет повторное использование программных компонентов, в корректности которых не должно быть никаких сомнений.

Если разработчик хочет быть уверенным в надлежащей работе объектно-ориентированного ПО, тогда ему требуется систематический подход к специфицированию и реализации объектно-ориентированных программных сущностей и их взаимосвязей в программной системе. Такой подход существует и называется "Контрактное Проектирование" ("Design by Contract") и в его рамках программная система рассматривается в виде множества взаимодействующих компонентов, чьи отношения строятся на основе точно определенной спецификации взаимных обязательств — контрактов, которые являются сквозной функциональностью.

При использовании аспектного подхода достаточно просто придерживаться принципов контрактного программирования. Можно выделить два подхода к соблюдению контрактов — на этапе выполнения и на этапе компиляции. На рисунке 19 представлена схема применение системных соглашений при использовании AspectJ. Соглашения описываются в аспектных модулях и воздействуют на компоненты системы посредством аспектов. В момент компиляции происходит интеграция аспектов времени выполнения в компоненты и выдается информация о нарушениях соглашений времени компиляции.

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

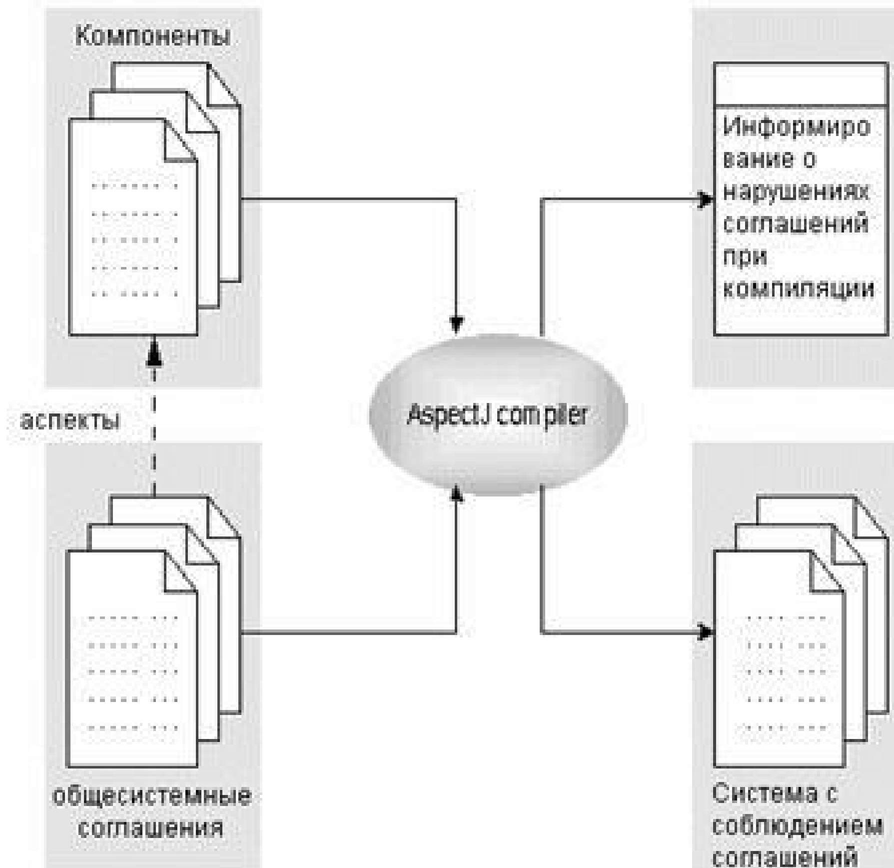


Рисунок 19. Схема применения системных соглашений при использовании AspectJ.

8.2.3.1. Контракты на этапе выполнения

На этапе выполнения подходом для проверки надлежащей работы участка кода разработчик должен встраивать в код проверку предусловий, постусловий и инвариантов.

Предусловие, это условие, которое налагается на входные данные для некоторого логически выделенного блока в программе. Например, для входных данных метода. Постусловие аналогично предусловию, но проверяется при возврате значения из блока "наружу" для проверки целостности результата. Инвариант же это условие, которое должно выполняется всегда в течение какой-либо итераций (например, внутри цикла). Инварианты класса — это ограничения, характеризующие его семантику с точки

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

зрения непротиворечивости. Это понятие является особенно важным в контексте управления конфигурацией и регрессионного тестирования — оно описывает более глубокие свойства класса: здесь не просто устанавливается характеристика, верная для определенного момента эволюции класса, но и налагаются ограничения, на все последующие изменения.

С точки зрения контрактной теории, инвариант является обобщенным утверждением, применимым ко всему множеству контрактов, определяющих класс.

Рассмотрим аспект, который реализует проверку инвариантов. Абстрактный аспект InvariantProtocolопределяет общее поведение при проверке инварианта — то есть определяет абстрактную точку выполнения программы (которая в конкретных условиях будет определена), абстрактный метод для проверки инварианта (которые будет определен в конкретном случае для классов нуждающихся в этом), и определяет моменты вызова метода проверки инварианта — до и после абстрактной точки выполнения программы.

```
public abstract aspect InvariantProtocol{
     protected interface InvariantCheck{}
     abstract protected pointcut methodCall(InvariantCheck obj);
     abstract protected void assert(InvariantCheck obj);
     after (InvariantCheck obj): methodCall(obj){
          assert(obj);
     }
     before (InvariantCheck obj): methodCall(obj){
          assert(obj);
     }
}
aspect InvariantProtocolImp {
     declare parents TestClass implements InvariantCheck;
     protected pointcut methodCall(InvariantCheck obj):
          target(obj) && execution(* .* (. .));
     protected void assert(InvariantCheck obj){
          TestClass testClass = (TestClass) obj;
          // проверкаинвариантадляклассаTestClass
     }
}
```


Copyright © 1999 - 2007, IT-archiv. All rights reserved.

Конкретная реализация проверки инвариантов представлена в виде аспекта InvariantProtocolImpl, который реализует проверку инварианта для некого TestClass. Этот аспект определяет метод, в котором будет проверяться инвариант и точку выполнения программы, в которой будет вызываться проверка. В данном случае точка, в которой будет проверяться инвариант класса, — это вызов любого метода класса TestClass. Идеи проверки соответствия данных контракту достаточно просты, и каждый программист их так или иначе выполняет. Простейшая проверка внутри функции, которая обрабатывает текстовые строки, на то не "подсунули" ли ей указатель на строку равный null, является проверкой предусловия. Другое дело, что если это оформить в виде обычного условного оператора, то читающий вашу программу человек сразу не догадается, что к общему алгоритму работы эта проверка условия не имеет никакого отношения, а при оформлении такой проверки в виде аспектов основной код не будет обременен такими проверками вообще.

Кроме проверки инвариантов, предусловий и постусловий на этапе выполнения программы можно обнаружить несоответствие обрабатываемых данных потоку управления. Например, проверить правильность вызова библиотеки стороннего производителя, на которую накладывается ограничение на поток, в котором она может вызываться. Или, например, при разработке серверной части в трехуровневом распределенном приложении, где нет смысла вызывать и работать с библиотекой пользовательского интерфейса.

```
public aspect DetectUISingleThreadRuleViolationAspect {
    pointcut viewMethodCalls()
        : call(* javax..JComponent+.*(..));
    pointcut modelMethodCalls()
        : call(* javax..*Model+.*(..))
        || call(* javax.swing.text.Document+.*(..));
    pointcut uiMethodCalls()
        : viewMethodCalls() || modelMethodCalls();
    before() : uiMethodCalls() && if(!EventQueue.isDispatchThread()) {
        System.err.println(
            "Нарушение: Метод библиотеки Swing вызван из не-AWT нити"
            + "\nВызывающий метод: "
            + thisJoinPointStaticPart.getSignature()
            + "\nВызывающий объект: "
```


Copyright © 1999 - 2007, IT-archiv. All rights reserved.

```
            + thisEnclosingJoinPointStaticPart.getSignature()
            + "\nИсточник: "
            + thisJoinPointStaticPart.getSourceLocation()
            + "\nНить: " + Thread.currentThread() +"\n");
    }
}
```

В примере аспект DetectUISingleThreadRuleViolationAspect обнаруживаются любые вызовы компонентов или моделей библиотеки Swingязыка Java, при этом перед вызовом этого метода выдается сообщение об ошибке.

8.2.3.2. Контракты на этапе компиляции

На этапе компиляции некоторые реализации АОП предоставляют возможность соблюдения соглашений на этапе компиляции, чего невозможно сделать при помощи сборки проекта компилятором объектного языка. При использовании данного свойства повышается управляемость проектом, так как в этом случае у системного архитектора появляется дополнительная возможность уведомлять разработчиков большого проекта о проектных изменениях.

Например, при интеграции аспектного модуля DetectLogUsage при сборке проекта будет выдаваться предупреждающее сообщение о том что необходимо использовать новый метод журнала событий Logger.logp() вместо потерявшего актуальность Logger.log(..), который сохранен для совместимости.

```
public aspect DetectLogUsage {
    declare warning : call(* Logger.log(..))
        : "Consider Logger.logp() instead";
}
```

Или, определить вызов вывода информации на консоль вместо того чтобы пользоваться принятым в проекте соглашением пользоваться журналом событий при выполнении операций. Аспект DetectSystemOutErrorUsage демонстрирует эту возможность. При сборке проекта будет выдано соответствующее предупреждение.

```
aspect DetectSystemOutErrorUsage {
       declare warning : (get(* System.out) || get(* System.err))
         && within(compile_time..*)
       : "Consider Logger.logp() instead";
```


Copyright © 1999 - 2007, IT-archiv. All rights reserved.

```
}
```

При разработке серверных компонентов недопустимы вызовы методов библиотеки пользовательского интерфейса, что описано в спецификации. Но язык программирования не препятствует использованию подобных ограничений, и отследить нарушения в большом проекте очень сложно. Аспект DetectEJBViolations не позволяет компилировать код, если наследники класса EnterpriseBean — серверные компоненты, пытаются вызвать внутри себя методы библиотеки пользовательского интерфейса, или если есть попытка напрямую обратиться к статическому полю данного класса. Выдаваемая при компиляции ошибка отсылает разработчика к спецификации, в которой изложены эти ограничения.

```
public aspect DetectEJBViolations {
    pointcut uiCalls() : call(* java.awt.*+.*(..));
    declare error : uiCalls() && within(EnterpriseBean+)
          : "UI calls are not allowed from EJB beans.See EJB 2.0
                specification section 24.1.2";
    before() : uiCalls() && cflow(call(* EnterpriseBean+.*(..))) {
        System.out.println("Detected call to AWT from enterprise
                bean");
        System.out.println("See EJB 2.0 specification section 24.1.2");
        Thread.dumpStack();
    }
    pointcut staticMemberAccess() : set(static * EnterpriseBean+.*);
    declare error : staticMemberAccess()
        : "EJBs are not allowed to have non-final static variables.
                See EJB 2.0 specification section 24.1.2";
}
```

Управление доступом — это один из видов соглашений, которое ограничивает доступ к функциональности. В примере класса Productописан аспект, который не позволит собрать проект, если новый экземпляр данного класса будет создан не специально предназначенной для этого фабрикой. Если метод конфигурации продукта будет вызван не специально предназначенным для этого конфигуратором — также не будет возможности собрать проект, так как будет выдана ошибка при компиляции.

```
public class Product {
      public Product() {
```


Copyright © 1999 - 2007, IT-archiv. All rights reserved.

```
        // constructor implementation
       }
       public void configure() {
        // configuration implemntation
       }

       //методыклассаProduct

       static aspect FlagAccessViolation {
            pointcut factoryAccessViolation()
                : call(Product.new(..)) && !within(ProductFactory+);
            pointcut configuratorAccessViolation()
                : call(* Product.configure(..)) &&
                   !within(ProductConfigurator+);
            declare error
                :  factoryAccessViolation() ||
                   configuratorAccessViolation()
                   : "Нарушениедоступа.
                 ТолькоProductFactory.createProduct() можетсоздаватьProduct";
       }
}
```

Еще пример с ограничением на вызываемые методы — если дизайном системы предусмотрено что только определенные объекты могут взаимодействовать друг с другом, то подобные ограничения легко описываются с помощью аспектов. При добавлении аспекта ShoppingCartAccessAspect в систему при сборке проекта будет выдаваться предупреждение что методы класса ShoppingCart может вызвать только ShoppingCartOperator.

```
public aspect ShoppingCartAccessAspect {
    declare warning
        : (call(* ShoppingCart.add*(..))
           || call(* ShoppingCart.remove*(..))
           || call(* ShoppingCart.empty(..)))
        && !within(ShoppingCartOperator)
        : "НевернаяработасShoppingCart; толькоShoppingCartOperator
        может выполнять подобные действия";
}
```

На протяжении нескольких лет общество программистов выработало несколько

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

удачных подходов в программировании, использование которых решает потенциальные проблемы. Описав подобные подходы в виде аспектов можно отслеживать их выполнение во всем проекте. Например, после добавления в проект аспекта DetectPublicAccessMembers при сборке будет выдаваться сообщение о попытке обратится к не final полю класса или при попытке установить значение в любое publicполе без использования специально предназначенных для этого геттеров или сеттеров.

```
aspect DetectPublicAccessMembers {
    declare warning :
        get(public !final * *) || set(public * *) :
         "Please consider using non-public access";
}
```

Теория Контрактного Проектирования является основой для решения многих проблем, критических для объектно-ориентированного подхода: на какие понятия опираться на стадии анализа, как специфицировать компоненты, как документировать код, чем руководствоваться при выполнении тестирования. Все это вместе обеспечивает систематический подход к построению системы с меньшим количеством дефектов. И надежная система получается как итог надлежащим образом встроенных в процесс разработки действий.

В настоящий момент существуют следующие подходы к соблюдению контрактов в проекте:

- Документирование ограничений
- Встраивание проверки контрактов в код
- Использование инструментов обнаруживающих нарушения контрактов

При использовании существующих в настоящий момент подходов к соблюдению контрактов возникают следующие проблемы:

- Неспособность программных компонентов к повторному использованию. Соблюдение контрактов может быть верно для компонентов только в контексте конкретной системы. В другой системе вполне могут быть другие соглашения, что ведет к модификации кода компонентов
- Перемешивание кода проверки условий и соблюдения контрактов с основным кодом компонентов. Это снижает способность компонентов быть повторно

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

использованными и снижает "читабельность" их кода.

- Рассредотачивание кода проверки контрактов по всей системе. Если возникает потребность изменить какое-либо из условий, то необходимо изменить их во всех модулях, на которые оно распространяется.
- Громоздкая реализация проверки условий. Изъятие и вставка кода проверки условий в программные компоненты является весьма сложной задачей. Поддерживать проверку условий для всей системы достаточно тяжело.
- Невозможность соблюдения контрактов во время сборки проекта. В настоящее время реализации компиляторов объектных языков (Java, C++) не позволяют соблюдать проектные соглашения при сборке проекта.

Аспектный подход решает все эти проблемы, и преимущества от его использования очевидны. АОП предлагает простое и мощное решение для реализации проверки проектных контрактов.

8.2.4. Управление объектами в многопоточной среде

Объекты позволяют разбить программу на независимые секции. Часто также необходимо превратить программу в несколько независимо выполняющихся подзадач. В этом случае на помощь разработчику приходит механизм синхронизации. Синхронизацией называется обеспечение заданной очередности прохождения процессов через определенные состояния. Наиболее часто синхронизация требуется для координации доступа нескольких процессов к одному разделяемому ресурсу.

Используя АОП и возможность окружать код компонентов сквозными инструкциями (around advice), достаточно легко можно окружить объекты сквозной функциональностью — защитой в многопоточной среде.

```
public abstract aspect  AbstractThreadSafe{
    abstract pointcut myClass(Object obj);
    pointcut safeMethods(Object obj): myClass(obj)
                && execution(* *(..));
    Object around (Object obj): safeMethods(obj){
        synchronized(this){
            return proceed(obj);
        }
    }
```


Copyright © 1999 - 2007, IT-archiv. All rights reserved.

```
}
```

Аспекты, расширяющие данный абстрактный аспект должны определить набор классов, методы которого необходимо защитить от совместного доступа. Так как для примеров используется язык AspectJ, то в примере используется примитив синхронизации языка Java. Однако вполне возможно написание собственного примитива синхронизации, и благодаря аспектному подходу код такого механизма синхронизации не будет "перемешан" с кодом бизнес компонентов.

8.2.5. Визуализация алгоритмов

АОП предлагает мощный подход к интеграции кода в уже существующий код. Рассмотрим простую задачу: Пять философов сидят за круглым обеденным столом. Между каждыми двумя философами есть одна вилка, которая может быть общей для них. Каждый философ может либо думать, не требуя вилок, либо есть, используя две соседние вилки, расположенные по одну и по другую стороны от него. Время обеих фаз "думать" и "есть" — произвольная конечная величина. При отсутствии двух свободных вилок, необходимых философу для еды, последний переходит в состояние ожидания.

Предположим, что данная задача была решена сторонним поставщиком, алгоритм реализован, имеет точку входа и задокументированы события, происходящие во время выполнения алгоритма. При этом алгоритм никак не интерпретирует свою работу, то есть никакого вывода о состоянии алгоритма во время работы не происходит. Необходимо дополнить реализацию этого алгоритма двумя видами вывода — выводом на консоль и графическим выводом по требованиям для конечной системы. В данном случае сквозной функциональностью на этом уровне абстракции является интерпретация работы алгоритма.

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

События, происходящие во время работы алгоритма, могут быть следующими:

- Захват первой вилки
- Переход в состояние ожидания
- Захват второй вилки
- Мысленный процесс
- Начало поглощения пищи
- Конец поглощения пищи
- Создание объекта Философ
- Создание объекта Вилка
- Запуск алгоритма

Первый путь решения проблемы — это по описанным событиям, которые происходят во время работы алгоритма, реализовать прокси-объект который будет пропускать все события через себя и по необходимому событию выполнять соответствующие действия по интерпретации. Подходы к созданию прокси-объекта описаны в [6]. Затем такой объект необходимо интегрировать в код алгоритма.

Второй подход — на каждый вариант интерпретации создать аспект, в котором будет инкапсулирована логика по интерпретации работы алгоритма и описаны события, при которых эта логика должна работать. В данном случае это будет один аспект на текстовую интерпретацию с выводом состояния алгоритма и состояний всех объектов-философов и объектов-вилок по событиям, и второй аспект — создание

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

графического окна и интерпретация работы алгоритма в окне с отображением состояний всех объектов в графическом виде. При этом очевидны преимущества аспектного подхода — достигается модульность при реализации сквозной функциональности — интерпретации работы алгоритма.

В таблице 8 приведены значения метрических характеристик для двух типов реализаций.

Метрика	Объектно-ориентированная реализация	Аспектно-ориентированная реализация
Цикломатическая сложность (CC)	35	27
Сложность программы (HDiff)	85	26
Усилия разработчика (HEff)	164206	50201
Длина программы (HPLen)	1618	331
Словарь программы (HPVoc)	118	92
Объем программы (HPVol)	10110	2126
Количество строк кода (LOC)	563	119
Недостаток связности методов (LOCOM1)	248	49
Недостаток связности методов (LOCOM2)	94	74
Недостаток связности методов (LOCOM3)	82	74
Количество классов (NOC)	10	2
Общее число операндов в программе (Noprnd)	804	166
Общее число операторов в программе (Noprtr)	814	165
Количество вызываемых удаленных методов (NORM)	37	21

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

Число уникальных операндов программы (NUOprnd)	101	71
Число уникальных операторов программы (NUOprtr)	21	21
Отклик на класс (RFC)	355	36
Взвешенная насыщенность класса (WMPC1)	35	27
Взвешенная насыщенность класса (WMPC2)	52	21

Таблица 8. Метрические характеристики реализаций задачи обедающих философов.

Степень применимости аспектного подхода: Adaptability = 1.69

Аспектный подход упрощает дополнение существующего отлаженного кода без модификации последнего. При этом повышается способность повторного использования существующего кода, который в результате не привязан к контексту выполнения.

8.3. Использование АОП при поддержке существующей системы

Сопровождение — это деятельность по управлению эволюцией продукта в ходе его эксплуатации. На данном этапе вносится меньше архитектурных новшеств, но вместо этого делаются более локализованные изменения, возникающие по мере учета новых требований и исправления старых ошибок. Следует, однако, учитывать, что на данном этапе необходимо с осторожностью относится к модификациям отлаженного кода. АОП предоставляет новые возможности при сопровождении программных систем при добавлении новой функциональности и модификации уже существующей.

8.3.1. Кэширование

В высоко производительных приложениях время реакции системы на запрос пользователя является важной характеристикой приложения. Однако в том случае если приложение обрабатывает большое количество данных, то ничего с этим сделать нельзя кроме кэширования результата для вызова его из кэша во время повторного

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

запроса пользователем, что в среднем уменьшит время реакции системы.

Рассмотрим пример математической библиотеки вычисляющей значение факториала:

```
public class MathLibrary {
    public static long factorial(int n){
        if (n==0){
            return 1;
        }else{
            return n* factorial(n-1);
        }
    }
}
```

Если на этапе поддержки системы возникает потребность в увеличении среднего времени реакции системы при запросе результата этой функции, то необходимо закэшировать результат ее выполнения. Однако объектно-ориентированный подход не предоставляет способа модификации кода без его изменения. В случае работы с ОО подходом придется либо добавлять в код библиотеки функциональность по кэшированию результата либо при желании сохранить модульность необходимо создать кэширующий объект, инкапсулирующий в себе вызовы метода математической библиотеки, и заменить во всей системе вызовы кэшируемого метода на вызов метода кэширующего объекта. Оба варианта представляются достаточно трудоемкими.

В этом примере сквозной функциональностью является кэширование результата деятельности математической библиотеки. В случае работы с АОП достаточно легко можно создать аспект который будет инкапсулировать в себе логику по кэшированию результата и описаны точки интеграции аспекта — вызов метода вычисления факториала во всей системе. При этом код самой библиотеки останется не затронутым данными изменениями.

```
public aspect OptimizeFactorial {
    pointcut factorialOperation(int n) :
        call(long MathLibrary.factorial(int)) && args(n);
    pointcut topLevelFactorialOperation(int n) :
        factorialOperation(n)
        && !cflowbelow(factorialOperation(int));
```


Copyright © 1999 - 2007, IT-archiv. All rights reserved.

```
    private Map _factorialCache = new HashMap();

    before(int n) : topLevelFactorialOperation(n) {
        System.out.println("Seeking factorial for " + n);
    }
    long around(int n) : factorialOperation(n) {
        Object cachedValue = _factorialCache.get(new Integer(n));
        if (cachedValue != null) {
            System.out.println("Found cached value for " + n
                                   + ": " + cachedValue);
            return ((Long)cachedValue).longValue();
        }
        return proceed(n);
    }
    after(int n) returning(long result)
        : topLevelFactorialOperation(n) {
        _factorialCache.put(new Integer(n), new Long(result));
    }
}
```

В таблице 9 приведены значения метрических характеристик для двух типов реализаций.

Метрика	Объектно-ориентированная реализация	Аспектно-ориентированная реализация
Цикломатическая сложность (CC)	2	2
Сложность программы (HDiff)	18	11
Усилия разработчика (HEff)	3527	1909
Длина программы (HPLen)	141	89
Словарь программы (HPVoc)	26	13
Объем программы (HPVol)	569	325
Количество строк кода (LOC)	38	21
Недостаток связности методов (LOCOM1)	3	2

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

Недостаток связности методов (LOCOM2)	72	47
Недостаток связности методов (LOCOM3)	69	42
Количество классов (NOC)	7	2
Общее число операндов в программе (Noprnd)	17	10
Общее число операторов в программе (Noprtr)	9	6
Количество вызываемых удаленных методов (NORM)	8	3
Число уникальных операндов программы (NUOprnd)	2	2
Число уникальных операторов программы (NUOprtr)	2	2

Таблица 9. Метрические характеристики реализаций кэширования.

Степень применимости аспектного подхода: Adaptability = 1.23

Аспектный подход предоставляет новые возможности по интеграции новых требований в программную систему без модификации кода системы.

8.3.2. Управление ресурсами

Рассмотрим задачу оптимизации использования некоторого ограниченного ресурса. При поддержке существующей системы может возникнуть необходимость изменить поведение какой-нибудь части системы. Предположим, что у системы есть "узкое место" — создание какого-либо емкого по времени ресурса, и перед нами лежит задача по оптимизации и улучшению системы. В программах, интенсивно работающих с базами данных, применяется пул потоков управления. Возьмем для примера в качестве ресурса нить(threads) и попробуем организовать пул нитей в рамках существующей системы. Объектно-ориентированный подход при организации пула ресурсов требует решения на раннем этапе при проектировании системы. То есть, дизайном проекта должно быть предусмотрено, когда и как можно получить ресурс из пула и каким

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

образом его можно вернуть назад для последующего повторного использования. Проектирование управления ресурсным пулом на ранней стадии разработки является сложной задачей, однако добавление такой функциональности в уже готовую систему требует гораздо больших усилий и изменений исходного кода. В большом проекте такую задачу можно считать непосильной, если требуется обернуть вызовы системного API.

АОП предоставляет новые возможности при поддержке готовых систем. На рисунке 18 представлена модель организации пула потоков управления при помощи АОП. Аспект "перехватывает" момент создания нового потока управления и возвращает свободный поток управления, полученный из пула. По завершении выполнения некоторой задачи поток возвращают в пул.

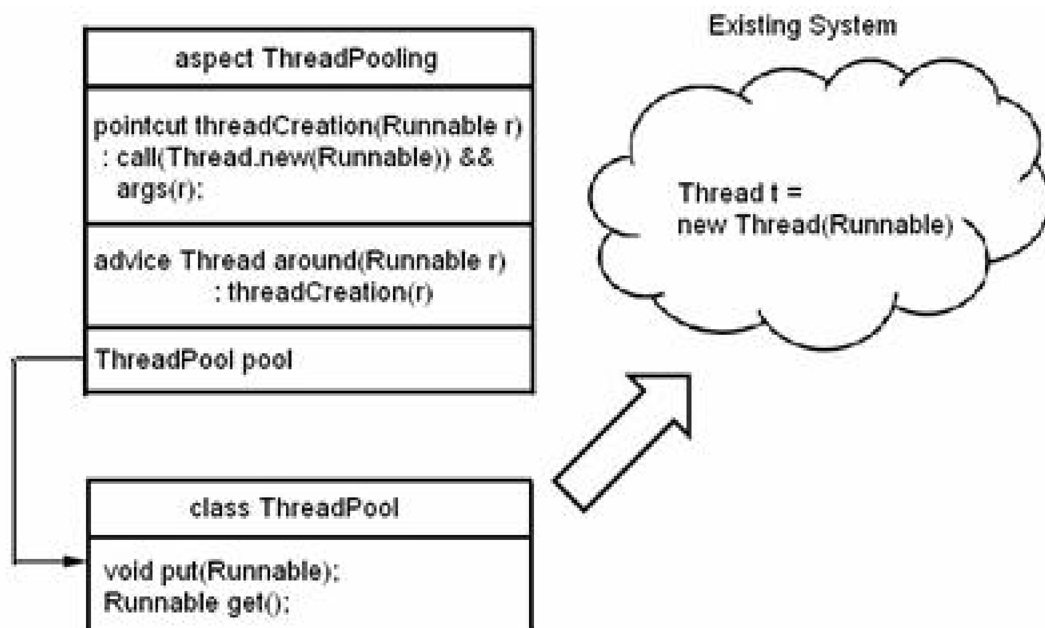


Рисунок 21. Организация ресурсного пула.

Рассмотрим детально:

- Срез точек pointcut threadCreation() — условие, выполняемое во время создания нового объекта Thread, принимающего объект Runnable в качестве аргумента.
- Advise threadCreation() — набор инструкций, выполняемых по созданию нового объекта Thread, где проверяется пул нитей на наличие доступных. Если на данный

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

момент нет доступных нитей, то создается новая, в противном случае делегат устанавливается на переданный аргумент — объект Runnable и возвращается существующий объект.

- Аспект ThreadPooling организует пул ресурсов при помощи класса ThreadPool, который имеет методы для извлечения и возвращения ресурса в пул, который может быть представлен в виде списка ресурсов. После того, как нить выполнила свою работу, она не уничтожается, а помещается назад в пул и переводится в состояние ожидания (wait).

Метрика	Объектно-ориентированная реализация	Аспектно-ориентированная реализация
Цикломатическая сложность (CC)	7	6
Сложность программы (HDiff)	33	27
Усилия разработчика (HEff)	9821	8745
Длина программы (HPLen)	356	311
Словарь программы (HPVoc)	38	37
Объем программы (HPVol)	1710	1503
Количество строк кода (LOC)	165	135
Недостаток связности методов (LOCOM1)	1	0
Недостаток связности методов (LOCOM2)	33	0
Недостаток связности методов (LOCOM3)	0	0
Количество классов (NOC)	7	6
Общее число операндов в программе (Noprnd)	177	154
Общее число операторов в программе (Noprtr)	179	157
Количество вызываемых	14	13

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

удаленных методов (NORM)		
Число уникальных операндов программы (NUOprnd)	31	31
Число уникальных операторов программы (NUOprtr)	9	9
Отклик на класс (RFC)	65	62
Взвешенная насыщенность класса (WMPC1)	7	6
Взвешенная насыщенность класса (WMPC2)	4	4

Таблица 10. Метрические характеристики реализаций ресурсного пула.

Степень применимости аспектного подхода: Adaptability = 1.11

Аспектный подход предоставляет новые возможности по интеграции новых требований в программную систему по сравнению с традиционным объектно-ориентированным подходом. Если возникают новые требования на этапе поддержки системы, то такие требования легко интегрируются без потери модульности и модификации кода компонентов.

8.3.3. Обработка ошибок

Ошибки при использовании объектно-ориентированного подхода обрабатываются с помощью механизма исключений

- Исключение это типизированный объект, содержащий информацию об ошибке
- Исключение создается и генерируется когда происходит ошибка
- Исключение передается через стек набору обработчиков исключений
- Обработчик исключения производит его обработку

При использовании АОП обработку исключений можно рассматривать как сквозную функциональность и добавить общее поведение в систему при обработке исключения определенного типа.

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

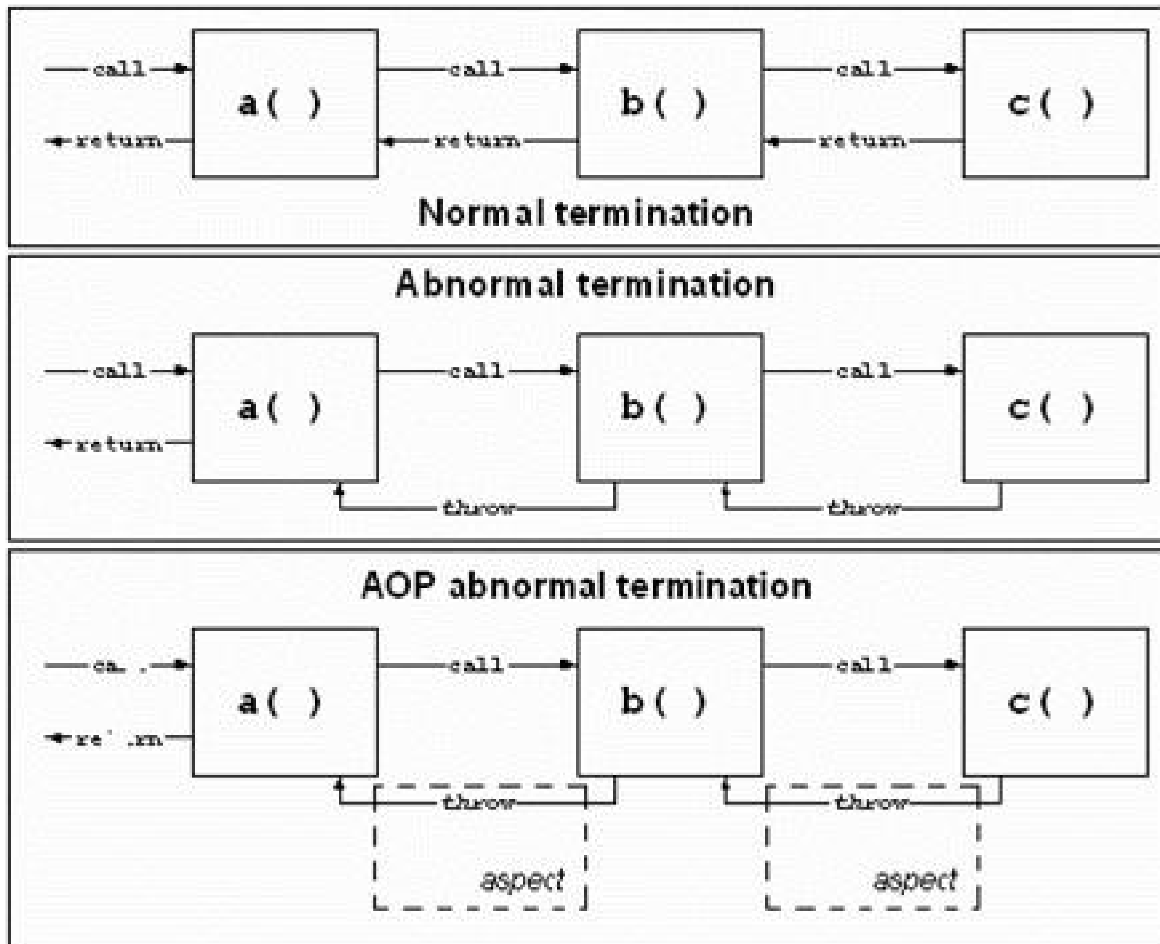


Рисунок 22. Структурированная обработка ошибок.

Аспектный код по обработке ошибок будет встроен в места обработки исключительных ситуаций по всей системе, что дает возможность однотипно обрабатывать одинаковые исключения.

Например, при определении абстрактного аспекта ExceptionHandlingAspect задается общая стратегия обработки ошибок в программной системе. Любой наследник этого аспекта должен определить срез точек на которые будет действовать этот аспект и метод обработки исключительной ситуации exceptionHandling(Throwable ex).

```
public aspect ExceptionHandlingAspect {
      public abstract pointcut exceptionJoinPoints();
```


Copyright © 1999 - 2007, IT-archiv. All rights reserved.

```
      protected abstract void exceptionHandling(Throwable ex);
      after() throwing (Throwable ex): exceptionJoinPoints()
          this.exceptionHandling(ex);
      }
  }
```

В распределенном приложении в части системы, которая относится к web, можно расширить этот аспект аспектом ServletsExceptionHandlingAspect, в котором можно задать общее поведение по обработке исключительных ситуаций в этой части системы

```
public aspect ServletsExceptionHandlingAspect extends ExceptionHandlingAspect{
   protected abstract void exceptionHandling(Throwable ex){
        // handling exceptions in servlets Java
   }
}
```

А этот аспект, в свою очередь, можно расширить аспектами, которые определяют конкретные срезы точек, в которые будут встроены данные аспекты при обработке ошибок.

```
public aspect DistributionExceptionHandlingAspect
            extends ServletsExceptionHandlingAspect{
    public pointcut exceptionJoinPoints():
        DistributionAspect.facadeMethodsCall();
}
```


Copyright © 1999 - 2007, IT-archiv. All rights reserved.

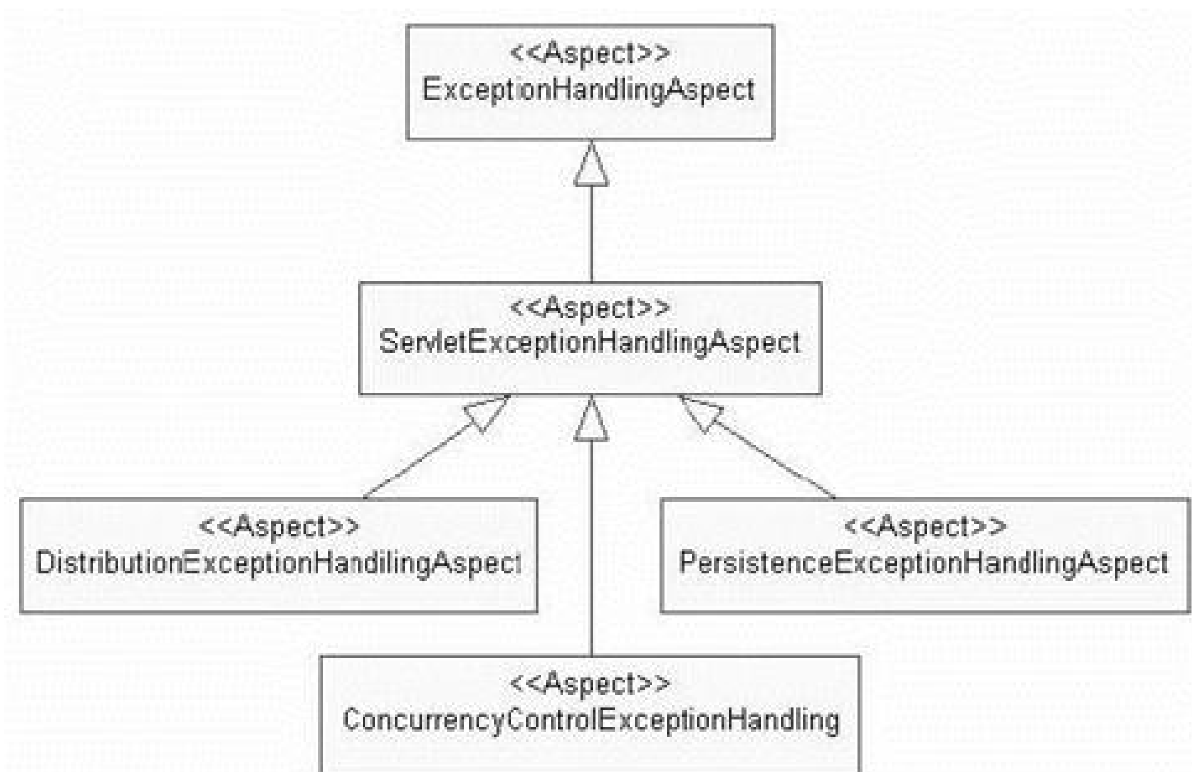


Рисунок 23. Диаграмма аспектов.

В работе [12] проводится изучение применимости аспектного подхода при обработке ошибок. В данной работе исследуются два варианта некоторой программной системы — оригинальный с нормальным следованием обработчиков ошибок и проверки предусловий и постусловий интегрированных в компоненты системы, и второй вариант системы переписанный на AspectJ. В таблице 11 приведены результаты улучшений использования аспектного подхода в терминах метрики LOC(количество строчек кода)

	без использования аспектов	с использованием аспектов
Определение ошибок	2120 проверки предусловий (2120 LOC)	620 проверок предусловий (660 LOC)
	666 проверок постусловий (666 LOC)	291 проверка предусловий (300 LOC)
Обработка ошибок	414 операторов catch (2070 LOC)	31 аспект обработки ошибок

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

		(200 LOC)
% от общего количества LOC	10.9%	2.9%

По полученным результатам можно сделать вывод о положительном эффекте от применения аспектного подхода при обработке ошибок. На этапе поддержки существующей системы возможности, предоставляемые АОП, очень актуальны, так как позволяют без усилий добавлять новые требования по обработке ошибок в уже существующий код.

8.3.4. Изменение ролей объектов и иерархии классов

Некоторые реализации АОП позволяют влиять на иерархию классов в какой-нибудь части системы. Позволяют расширять классы (наследовать), реализовывать интерфейсы, добавлять поля и методы в класс. Данное свойство может быть полезным в случае динамической интеграции аспектов во время выполнения программного кода и отсутствии исходного кода расширяемых/изменяемых классов либо при желании сохранить свойства модульности программных компонентов для повторного использования, если добавляемые в класс свойства или поведение отвечает только локальным требованиям конкретной части системы.

Рассмотрим пример иерархии классов

```
class A {
           void n() { print("A.n()");}
       }
       class B extends A {
           void m() { print("B.m()");}
       }
       class C extends B {
           public void x() {print("C.x()"); }
       }
       class D extends B {
           public void y() { print("D.y()");}
           public void x() {print("D.x()");}
       }
       class E extends C {
       }
```

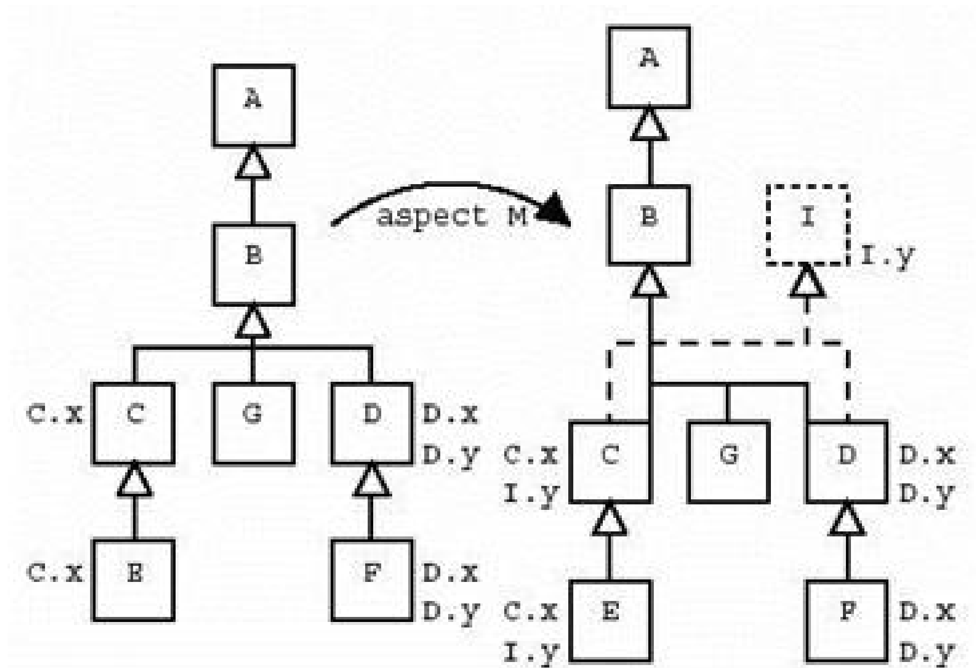

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

```
        class F extends D {
            void n() {print("F.n()");}
        }
        class G extends B {
            void n() {print("G.n()");}
        }
        interface I {
            void x();
            void y();
        }
}
```

применим к существующей иерархии классов аспект M

```
aspect M {
        declare parents: C implements I;
        declare parents: D implements I;
        public void I.y() { print("I.y()");}
}
```

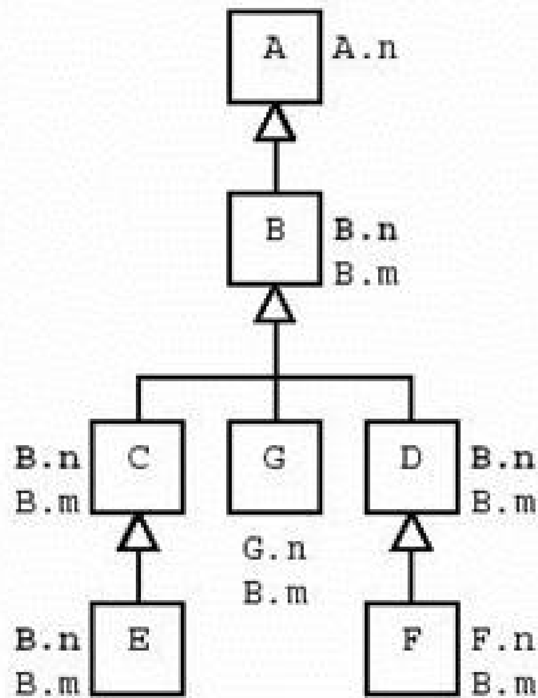
В результате получим



Copyright © 1999 - 2007, IT-archiv. All rights reserved.

Добавив аспект N, получаем, что все наследники класса B получают новый метод

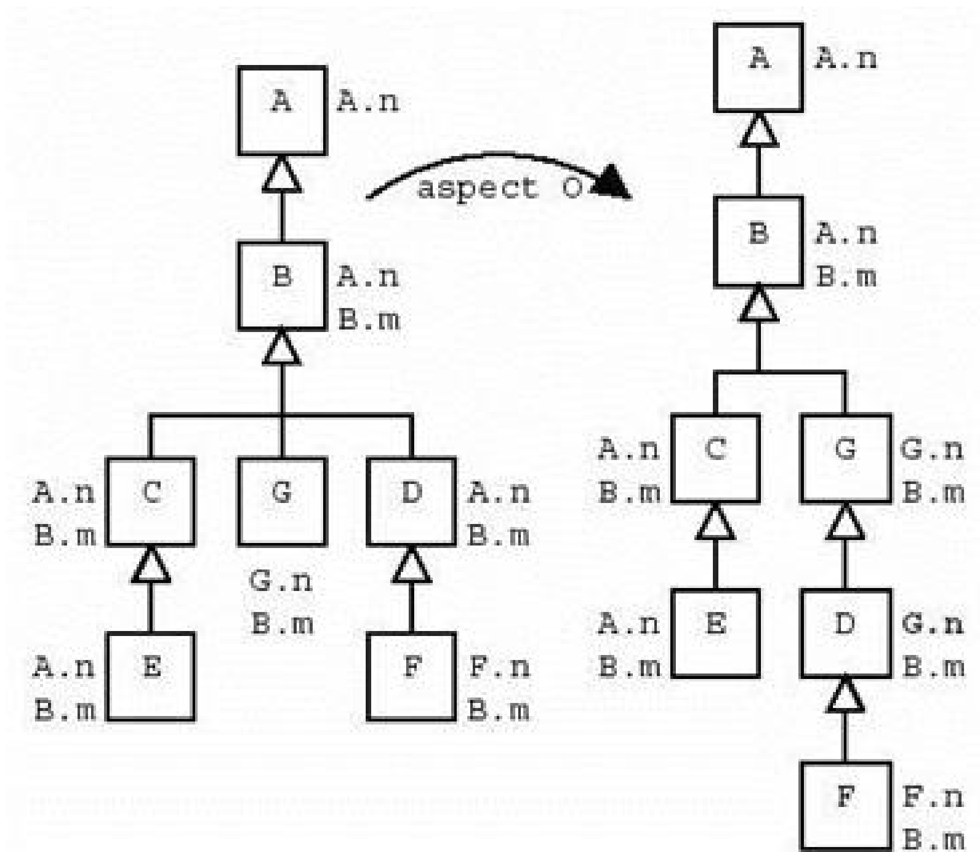
```
aspect N {
        void B.n(){
            print("B.n()");
        }
}
```



После добавления аспекта O класс D и его наследники приобретают свойства объекта G

```
aspect O {
        declare parents: D extends G;
}
```


Copyright © 1999 - 2007, IT-archiv. All rights reserved.



В рассмотренном выше примере реализации шаблона проектирования Observer при реализации протокола взаимодействия объектов описываются интерфейсы Subject и Observer, относительно которых строится взаимодействие этих сущностей.

```
public abstract aspect ObserverProtocol {
     protected interface Subject  { }
      protected interface Observer { }
        //реализация логики шаблона проектирования
}

public aspect CoordinateObserver extends ObserverProtocol{
  declare parents: Point  implements Subject;
  declare parents: Screen implements Observer;
  //:
}
```


Copyright © 1999 - 2007, IT-archiv. All rights reserved.

Затем в расширяющих аспектах данные интерфейсы реализуются существующими компонентами системы, которые ничего не знают о своей роли в шаблоне. Этим достигается повышенная модульность системы, и код существующих компонентов остается без изменений.

Свойство АОП влиять на иерархию классов — это достаточно мощное средство управления разработанными компонентами, а также адаптацией уже существующего кода под новые требования без модификации кода этих компонентов. Со стороны модульности, готовности существующего кода к повторному использованию, аспектный подход значительно выигрывает перед объектно-ориентированным подходом, так как предоставляет более гибкие средства для поддержки существующего кода.

9. Заключение

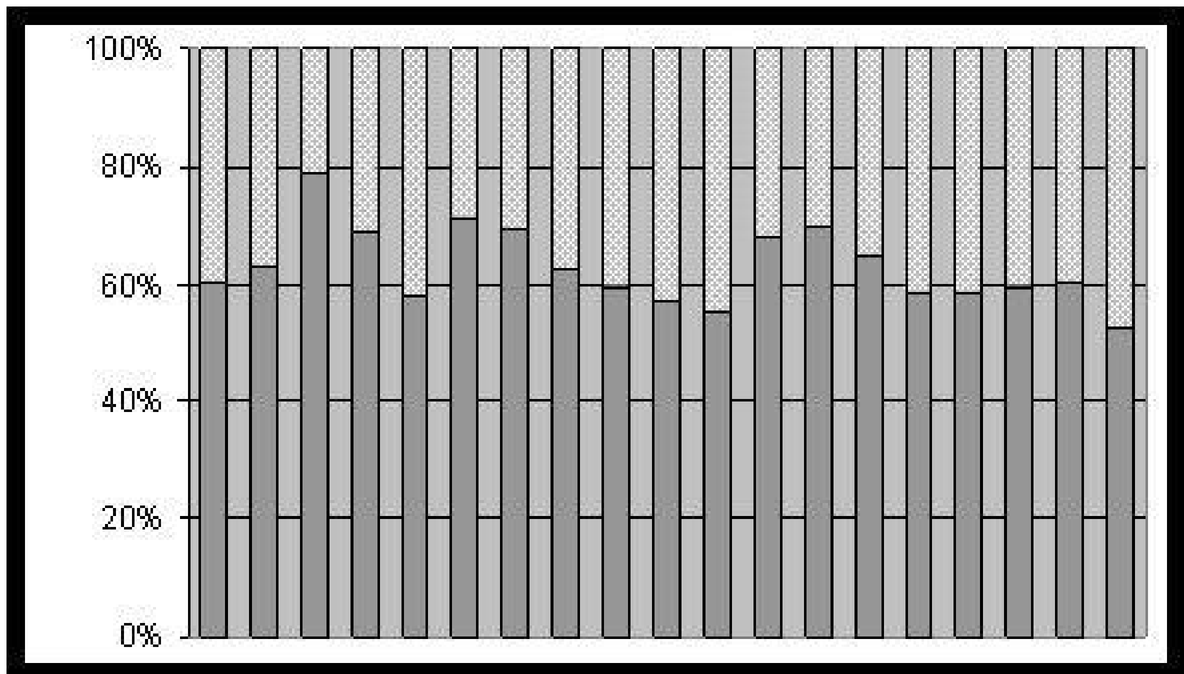
В рамках работы было проведено исследование проблемы применимости аспектно-ориентированного подхода при разработке программных систем. Были введены критерии сравнения традиционных объектно-ориентированных реализаций с аспектно-ориентированными. На основании результатов сравнения реализаций был сделан вывод о положительном эффекте при применении АОП на разных этапах жизненного цикла программных систем, а также рассмотрены новые варианты применения аспектов и решения типичных задач с использованием АОП, ранее не описанные в литературе. Все предложенные варианты оценены и проанализированы в соответствии с выбранными критериями.

Оценки проводились в соответствии с тремя выбранными направлениями

1. Оценки топологической и информационной сложности программ.
2. Оценки уровня языковых средств и их применения.
3. Оценки трудности восприятия и понимания программных текстов.

На этапе проектирования системы были приведены примеры реализации таких сквозных требований как реализация проколола взаимодействия объектов, реализация авторизации в системе и ведение журнала событий. По метрикам первой группы были получены следующие результаты:

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

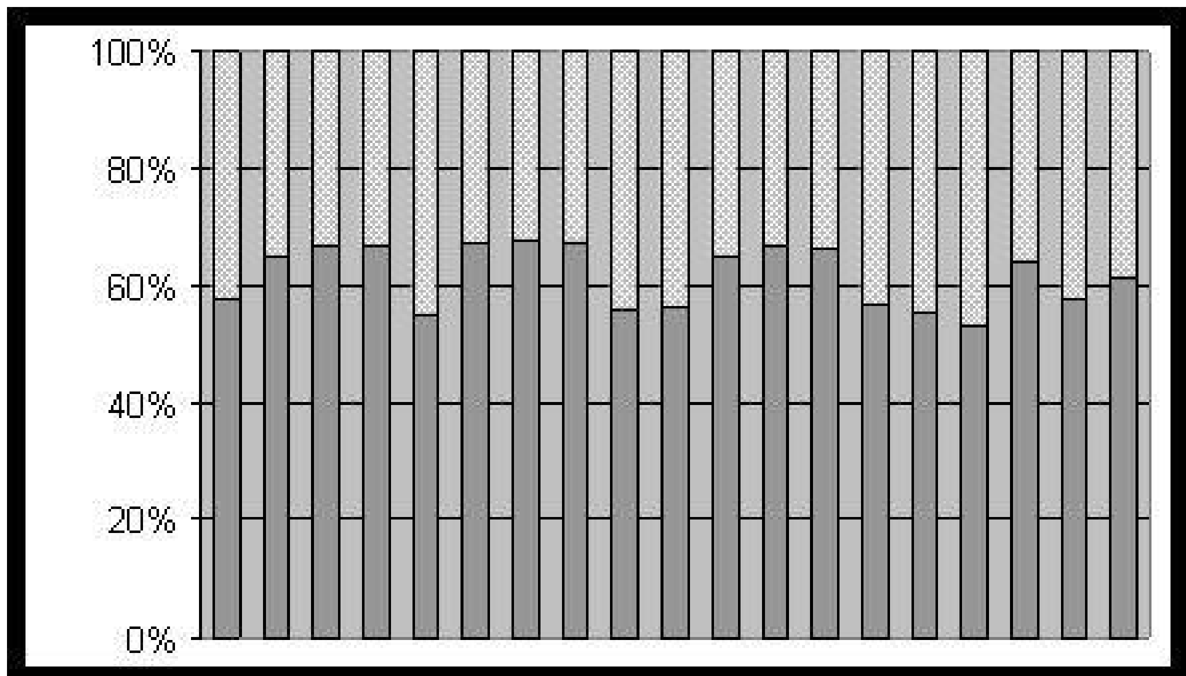


На приведенном графике видны уменьшения значений метрических характеристик на 20 — 45% при реализации программной системы с использованием АОП при принятии решений об аспектной реализации некоторых сквозных требований на данном этапе ЖЦ. На графике светлое значение представляет среднее значение метрики, подсчитанное для ОО реализации, темная — аспектной. Среднее значение метрики для ОО реализации принято за 100% значение метрики в случае аспектной реализации преобразовано в процентной отношение от метрики объектной реализации. Приведенный график можно рассматривать как функцию метрических характеристик (ресурсов) от времени, следовательно, уменьшение значений метрических характеристик в среднем положительным образом скажется на любом из этапов проекта, так как на каждый ресурс будет потрачено меньше времени, чем при подобной ОО реализации. По третьей группе метрик можно утверждать, что применение аспектно-ориентированной декомпозиция на этапе анализа и проектирования позволяет улучшить модульность разрабатываемой системы — выделить сквозную функциональность на разных уровнях абстракции и локализовать ее в отдельных модулях-аспектах. Такие аспекты являются неотъемлемой частью результирующей программной системы. При этом компоненты программной системы абсолютно свободны от контекста их применения и как следствие эти компоненты

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

полностью готовы к повторному использованию. Исходя из терминов модульности и компонентов, не обремененных кодом, не относящимся к представляемой ими абстракции, можно утверждать, что получаемая сквозная композиция классов выглядит более понятной, чем в случае ООП реализации.

На этапе разработки системы всегда решаются такие задачи как профилирование, трассировка, соблюдение проектных соглашений, слежение за корректностью входных и выходных данных на разных уровнях абстракции, отслеживается поведение объектов в многопоточной среде, применяются различные подходы к разработке повторно используемых компонент и стратегии их повторного использования. На этапе разработки системы существенную помощь разработчику могут оказать вспомогательные аспекты, а также аспекты, которые впоследствии станут частью программной системы. На данном этапе разработки по метрикам первой группы получены видимые результаты по уменьшению метрических характеристик этой группы на 35-45% по сравнению с объектной реализацией.

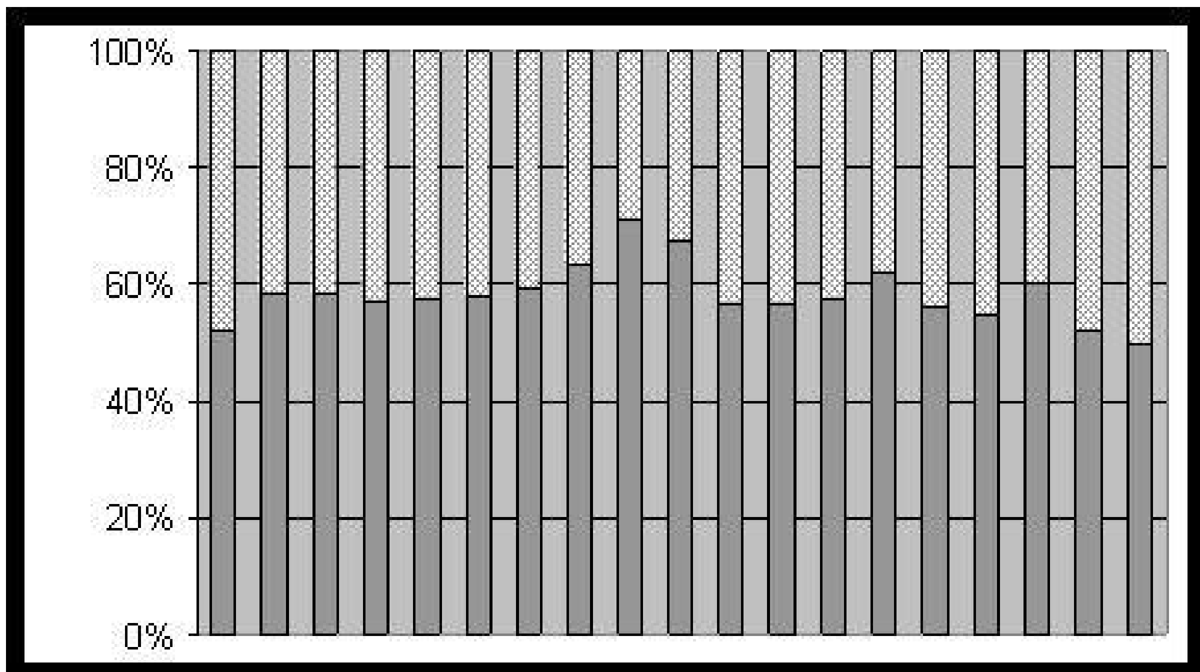


По трудности восприятия исходного кода (третья группа метрик) можно сказать, что аспектный подход увеличивает модульность, повышается способность компонентов к

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

повторному использованию и повышается *понятность* кода, который не "обременен" вспомогательной встраиваемой логикой, которая впоследствии может быть без усилий удалена.

На этапе поддержки существующей системы вносится меньше архитектурных новшеств, но вместо этого делаются более локализованные изменения, возникающие по мере учета новых требований и исправления старых ошибок. АОП предоставляет новые возможности при сопровождении программных систем при добавлении новой функциональности и модификации уже существующей. На данном этапе по метрикам первой группы получены видимые заметны уменьшения метрических характеристик этой группы на 30-50% по сравнению с объектной реализацией.



Аспектный подход предоставляет новые возможности по интеграции новых требований в программную систему по сравнению с традиционным объектно-ориентированным подходом. Если возникают новые требования на этапе поддержки системы, то такие требования легко интегрируются без потери модульности и модификации кода компонентов, что очень актуально на данном этапе разработки.

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

По метрикам второй группы (уровень языковых средств) на всех этапах ЖЦ можно утверждать о положительном эффекте от применения аспектной декомпозиции.

Итак, возможность применения аспектного подхода показана для задач самого разного характера. Приведенные примеры позволяют отметить основное достоинство аспектно-ориентированного подхода: улучшение модульности программной системы и вытекающие из него следствия:

- выражение в явной форме структуры "сквозной функциональности";
- упрощение сопровождения и внесения изменений;
- появление новых возможностей повторного использования кода.

Необходимо отметить, что АОП не рассматривается как замена сложившимся парадигмам программирования, а исполняет роль расширения, позволяющего обеспечить модуляризацию сквозной функциональности.

В настоящее время АОП — единственная методология, позволяющая справиться со сложностью, присущей очень большим системам.

10. Направления будущих исследований

В настоящий момент проводятся исследования в области АОП и аспектной декомпозиции, призванные оценить применимость этих технологий при создании информационных систем и подсистем различного типа, а также развитие теории АОП, находящейся на данный момент на начальной стадии развития, с использованием математических методов.

В качестве дальнейших исследований можно рассматривать работу по формализации подхода, а также работу по доказательству корректности программ, разработанных при помощи АОП. Также одним из перспективных исследований в этой области на данный момент считается изучение событийной модели АОП и применимость этой модели для построения операционных систем.

11. Список литературы

1. G. Kiczales, J. Lamping, A. Mendhekar, etc. *Aspect-oriented programming.* Published in proceedings of the European Conference on Object-Oriented Programming (ECOOP).

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

Finland, Springer-Verlag LNCS 1241. June 1997.
2. Е.А. Журавлев, В.А. Кирьянчиков. *О возможности динамической интеграции аспектов в аспектно-ориентированном программировании.*Изв. СПбГЭТУ (ЛЭТИ) Сер. Информатика, управление и компьютерные технологии. 2002. Вып. 3. С. 81 — 86.
3. Е. А. Журавлев, В. Н. Павлов. *Об одном подходе к реализации Аспектно-ориентированного программирования*. Изв. СПбГЭТУ (ЛЭТИ) Сер. Информатика, управление и компьютерные технологии. 2003
4. I. Kiselev. *Aspect-Oriented Programming with AspectJ.* Indianapolis, IN, USA: SAMS Publishing, 2002.
5. J. Hannemann, G. Kiczales. *Design pattern implementations in Java and AspectJ* OOPSLA 02, New York, USA, November 2002. P. 161 — 173.
6. Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. *Приемы объектно-ориентированного проектирования. Паттерны проектирования.* Издательство Питер, Санкт-Петербург, 2001.
7. S. Stelting, O. Maassen. *Применение шаблонов Java. Библиотека профессионала.*Издательство Вильямс, Санкт-Петербург, 2002
8. М. Холстед. *Начала науки о программах.* Москва, 1981
9. Г. Буч. *Объектно-ориентированный анализ и проектирование.* Издательство Бином, Невский диалект, Санкт-Петербург, 1999.
10. E. Dijkstra. *Programming Considered as a Human Activity. Classics in Software Engineering.* New York, Yourdon Press, 1979.
11. B. Meyer, *Applying Design by Contract*, Prentive Hall, 1992
12. M. Lippert, C Videira Lopes. *A Study on Exception Detection and Handling Using Aspect-Oriented Programming.*Xerox PARCTechnical Report P9910229 CSL-99-1, Dec. 99
13. O. Hachani, D. Bardou. *Using aspect-oriented programming for design patterns implementation*. Equipe SIGMA, LSR-IMAG, 38402 Saint Martin d'Heres Cedex, France.
14. M. Aksit, L. Bergmans, and S. Vural. *An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach*. In Proceedings of the ECOOP'92 Conference, LNCS 615, Springer-Verlag,1992
15. K.Leiberherr. Component Enhancement: *An Adaptive Reusability Mechanism for Groups of Collaborating Classes.* In Information Processing'92, 12th World Computer Congress,

Copyright © 1999 - 2007, IT-archiv. All rights reserved.

Madrid, Spain, J. van Leeuwen (Ed.), Elsevier, 1992, pp.179-185

16. Krzysztof Czarnecki, Ulrich Eisenecker *Generative Programming: Methods, Tools, and Applications,* Addison-Wesley, Paperback, Published June 2000.
17. *The AspectJ Programming Guide* 1998-2002, Xerox Corporation
18. K.Czarnecki. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models.* PhD thesis, Technische Universitat Ilmenau, Germany, 1998. (Глава Aspect-Oriented Decomposition and Composition)
19. Dharma Shkla, Simon Fell, Chris Sells. *Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse*. MSDN Magazine, March 2002.
20. Bruno Schaffer Design *and Implementation of Smalltalk Mixin Classes.* Ubilab Technical Report 98.11.1 Universitдtsstrasse 84 CH-8033 ZurichSwitzerland
21. Laddad, R. (2002). *I want my aop!, part 1. JavaWorld*. Avaliable at http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html.
22. Aspect-Oriented software development network www.aosd.net
23. Ch. Simony. *The Death of Computer Languages, The Birth of Intentional Programming*, Microsoft Research, 1995, *http://research.microsoft.com/pubs/view.aspx?tr_id=4*.
24. Rational Software Corporation www.rational.com
25. Homepage of the Subject-Oriented Programming Project, IBM Thomas J. Watson Research Center, YorktownHeights, New York, http://www.research.ibm.com/sop/
26. Homepage of the TRESE Project, University of Twente, The Netherlands, http://wwwtrese.cs.utwente.nl/; also see the online tutorial on Composition Filters at http://wwwtrese.cs.utwente.nl/sina/cfom/index.html
27. TogetherJ official web site. http://www.togethersoft.com

Copyright © 1999 - 2007, IT-archiv. All rights reserved.