Технологии программирования. Компонентный подход

В. В. Кулямин

Лекция 8. Образцы проектирования (продолжение)

Аннотация

Рассматриваются дополнительные примеры образцов: архитектурный стиль «данные–представление–обработка», ряд образцов проектирования, идиом и образцов организации работ.

Ключевые слова

Образец проектирования, архитектурный стиль, идиома, образец организации, образец процесса, архитектурный стиль «данные-представление-обработка», образец проектирования «подписчик», идиома «шаблонный метод», инспекция программ по Фагану.

Текст лекции

В этой лекции мы продолжим разбирать примеры образцов — рассмотрим детально архитектурный стиль «Данные–представление–обработка», а также примеры тех видов образцов, которые не уместились в предыдущую лекцию: образцов проектирования в узком смысле, идиом и образцов организации.

Данные–представление–обработка

Название. Данные–представление–обработка (model–view–controller, MVC).

Назначение. Интерактивные приложения с гибким интерфейсом пользователя. Требования к пользовательскому интерфейсу в интерактивных приложениях меняются чаще всего. Разные пользователи имеют разные наборы требований. В несколько меньшей степени это касается методов обработки данных, лежащих в основе таких приложений, — визуальное представление управляющих элементов может меняться вместе с интерфейсом, а сами выполняемые действия зависят от бизнес-логики и предметной области, и поэтому более устойчивы. Наименее подвержена изменениям модель данных, с которыми работает приложение.

Поэтому для увеличения гибкости и удобства изменений в таких приложениях необходимо соответствующим образом разделить их компоненты. При этом нужно принимать во внимание следующие факторы.

Действующие силы.

- Одна и та же информация может быть представлена по-разному и в нескольких местах для удобства доступа к ней многих различных пользователей, имеющих разные привычки и разные навыки работы с информацией.
- Изменения в данных должны немедленно отображаться в различных представлениях этих данных.
- Внесение изменений в пользовательский интерфейс должно быть максимально простым, иногда оно даже должно быть возможно прямо во время работы приложения.
- Поддержка различных стандартов пользовательского интерфейса и его перенос между платформами не должны влиять на код, связанный с методами работы с данными и структурой данных приложения.

Решение. Выделяется три набора компонентов. Первый набор — *данные*, *модель данных* или просто *модель* (*model*) — соответствует структуре данных предметной области, в которой работает приложение. Обязанности этих компонентов: представлять в системе данные и базовые операции над ними. Компоненты второго набора — *представления* (*view*)

— соответствуют различным способам представления данных в пользовательском интерфейсе. Для одних и тех же данных может иметься несколько представлений. Каждому компоненту представления соответствует один компонент из третьего набора, *обработчик* (*controller*) — компонент, осуществляющий обработку действий пользователей. Такой компонент получает команды, чаще всего нажатия клавиш и нажатия кнопок мыши в областях, соответствующих визуальным элементам управления — кнопкам, элементам меню и пр. Эти команды он преобразует в действия над данными. В результате каждого действия требуется обновить все представления всех данных, которые подверглись изменениям.

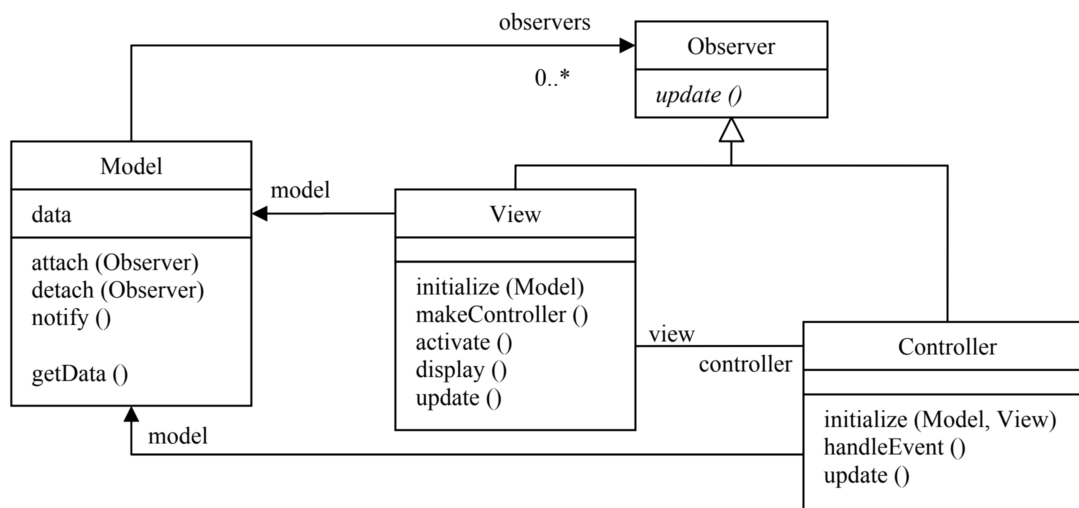

Рисунок 51. Структура классов модели, представления и обработчика.

Структура. Основными ролями компонентов в данном стиле являются *модель*, *представление* и *обработчик*.

Компонент-модель моделирует данные приложения, реализует основные операции над ними и возможность регистрировать зависимые от него обработчики и представления. При изменениях в данных модель оповещает о них все зарегистрированные компоненты.

Компонент-представление представляет данные в некотором виде для пользователей, читая их из модели при необходимости, т.е. при инициализации и после сообщений о произошедших изменениях. Кроме того, он инициализирует связанный с ним обработчик.

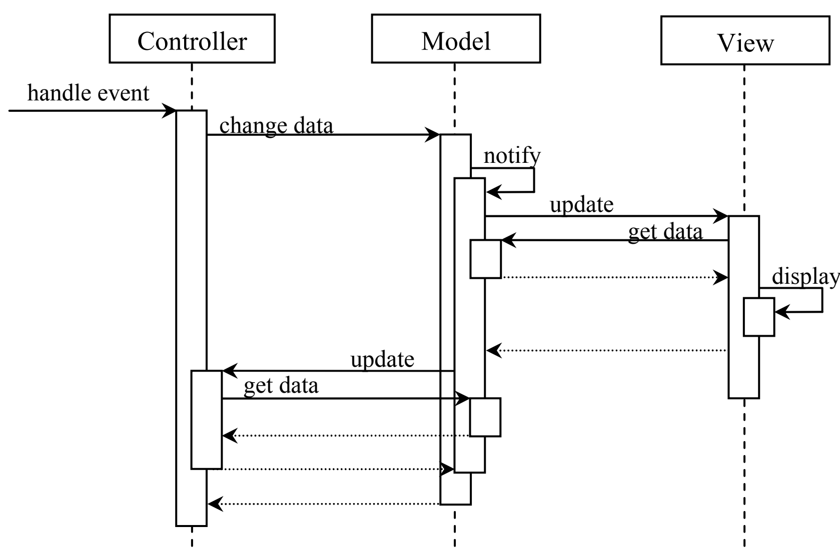

Рисунок 52. Сценарий обработки действия пользователя.

Компонент-обработчик обрабатывает действия пользователя, транслируя их в операции над моделью или запросы на показ некоторых элементов представлений. При оповещении об изменениях в модели он соответствующим образом изменяет собственное состояние, например, делая активными или отключая какие-нибудь кнопки и пункты меню.

Динамика. У системы два базовых сценария работы — инициализация всех компонентов и обработка некоторого действия пользователя с изменением данных и обновлением соответствующих им представлений и обработчиков.

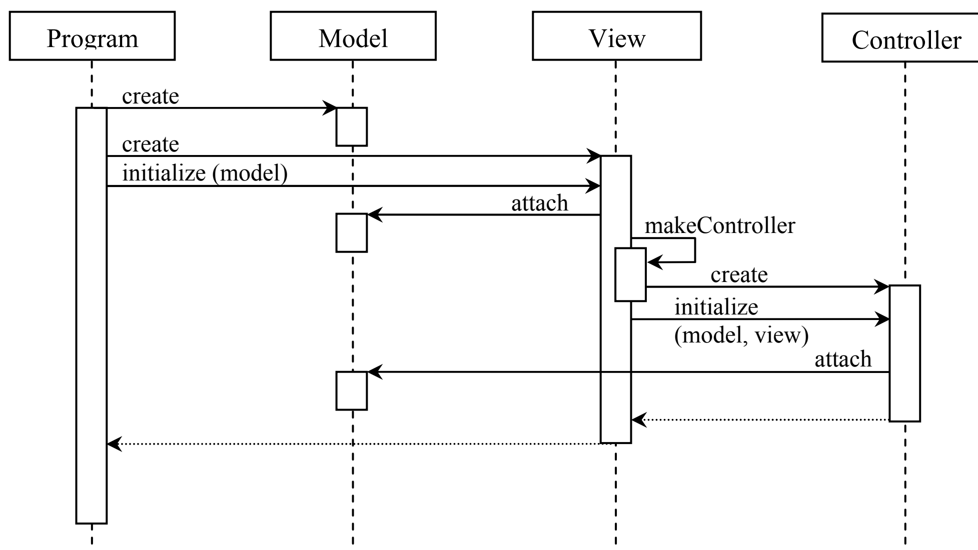

Рисунок 53. Сценарий инициализации системы.

Реализация. Основные шаги реализации следующие.

- Отделить взаимодействие человека с системой от базовых функций самой системы. Для этого необходимо выделить структуру данных, с которыми система работает, и набор необходимых для функционирования системы операций над ними.
- Реализовать механизм передачи изменений. Для этого можно воспользоваться образцом проектирования Подписчик (иначе называемым Наблюдатель, Observer).
- Спроектировать и реализовать необходимые представления.
- Спроектировать и реализовать необходимые обработчики действий пользователя.
- Спроектировать и реализовать связь между обработчиком и представлением. Обычно представление должно инициализировать соответствующий обработчик. Для этого можно, например, использовать образец проектирования Метод порождения (Factory Method).
- Реализовать построение системы из компонентов и инициализацию компонентов.

В качестве дополнительных аспектов реализации необходимо рассмотреть следующие.

- Динамические представления, создаваемые во время работы приложения.
- Подключаемые элементы управления, которые могут быть включены во время работы приложения. Например, переключение из режима «новичок» в режим «эксперт».
- Инфраструктура и иерархия представлений и обработчиков.
  Часто имеется готовая библиотека таких компонентов, на основе которых нужно строить собственные представления и обработчики. Эту задачу нужно решать с учетом семантики и возможностей библиотечных компонентов и связей между ними.
  Кроме того, одни представления могут визуально включать другие, а также элементы управления, с которыми связаны обработчики. Эта визуальная связь часто

должна быть поддержана, например, возможностью переключения фокуса пользовательского ввода между отдельными элементами.
Необходимо внимательно спроектировать (насколько это возможно, с учетом ограничений платформы и библиотек визуальных компонентов) стратегии обработки событий, особенно таких, в которых могут быть одновременно заинтересованы несколько компонентов, присутствующих на экране.

- Возможно, потребуется сделать систему еще более переносимой за счет отделения ее компонентов от конкретных библиотек и платформ. Для этого нужно разработать собственный набор абстрактных визуальных компонентов.

Следствия применения образца.

Достоинства.

- Возможность иметь несколько представлений одних данных, обновляемых по результатам воздействий пользователя на одно из них. Все такие представления синхронизированы, их показания соответствуют друг другу.
- Поддержка подключаемых и динамически изменяемых представлений и обработчиков.
- Возможность изменения стилей пользовательского интерфейса во время работы.
- Возможность построения каркаса (библиотек визуальных компонентов) для разработки многих интерактивных приложений.

Недостатки.

- Возрастание сложности разработки.
- Потери в производительности из-за необходимости обработки запросов пользователей сначала в обработчиках, затем в моделях, а затем во всех обновляемых компонентах.
- Если не оптимизировать производимые обновления аккуратно, чаще всего в ходе работы происходит много ненужных вызовов операций, обновляющих представления и обработчики.
- Представления и обработчики связаны очень тесно, из-за чего эти компоненты почти никогда нельзя переиспользовать по отдельности.
- И представления, и обработчики достаточно тяжело использовать без соответствующей им модели.
- Представления и обработчики наверняка потребуют изменений при их переносе на другую платформу или в другую библиотеку элементов графического интерфейса пользователя.
- Данный образец тяжело использовать в большинстве средств разработки GUI, поскольку они чаще всего определяют собственные стратегии обработки событий и стандартные обработчики для многих событий, например, для нажатия правой кнопки мыши.

Примеры. Впервые этот архитектурный стиль был использован при проектировании библиотеки разработки пользовательского интерфейса для языка Smalltalk [1]. С тех пор создатели множества подобных каркасов и библиотек используют те же принципы.

Еще один пример библиотеки для разработки пользовательского интерфейса, построенного на основе данного стиля — библиотека MFC (Microsoft Foundation Classes) от Microsoft. В ней используется более простой вариант стиля — с объединенными представлениями и обработчиками. Такая схема получила название документ-представление (Document-View): документы соответствуют моделям, представления объединяют функции представлений и обработчиков.

Такое объединение часто имеет смысл, поскольку представления и обработчики тесно связаны и практически не могут использоваться друг без друга. Их разделение на отдельные компоненты должно обосновываться серьезными причинами.

Последний пример использования этого стиля — архитектура современных Web-приложений, т.е. бизнес-приложений с пользовательским интерфейсом на основе HTML и связью между отдельными элементами, построенной на базе основных протоколов Интернет. В ней роль модели играют компоненты, реализующие бизнес-логику и хранение данных, а роль представлений и обработчиков исполняется HTML-страничками и HTML-формами, статичными или динамически генерируемыми. Далее в этом курсе построение таких приложений будет рассматриваться более детально, поэтому образец «данные-представление-обработка» имеет большое значение для дальнейшего изложения.

Образцы проектирования

Образцы проектирования в узком смысле являются типовыми проектными решениями, позволяющими удовлетворить часто встречающиеся требования к гибкости приложений и реализовать возможности их расширения за счет специальных форм организации классов и их связей.

Далее мы в деталях рассмотрим образец проектирования «подписчик». О другом примере, «адаптере», было рассказано в начале предыдущей лекции.

Подписчик

Название. Подписчик (subscriber) или подписчик-издатель (publisher-subscriber). Известен также под названиями «наблюдатель» (observer), «слушатель» (listener) или «подчиненные» (dependents).

Назначение. Реализация системы, в которой нужно поддерживать согласованными состояния большого числа объектов. Чаще всего при этом достаточно много компонентов зависит от небольшого набора данных. Можно было бы связать их явно, введя обращения ко всем компонентам, которые должны знать об изменениях, при каждом внесении изменений, но полученная система станет очень жесткой. Добавление нового компонента потребует сделать изменения во всех местах, от которых он зависит. Предлагаемое в рамках данного образца решение основано на гибкой связи между субъектом (от которого зависят другие компоненты) и этими зависимыми компонентами, называемыми *подписчиками*. Здесь нужно принимать во внимание следующие факторы.

Действующие силы.

- Об изменениях в состоянии некоторого компонента должны узнавать один или несколько других компонентов.
- Количество и конкретный вид компонентов, которые должны оповещаться об этих изменениях, заранее не известны или могут быть изменены во время работы приложения.
- Проведение зависимыми компонентами время от времени явных запросов о произошедших изменениях неэффективно.
- Источник информации (субъект или издатель) не должен быть тесно связан со своими подписчиками или зависим от них.

Решение. Компонент, от которого зависят другие, берет на себя функции *издателя*. Он предоставляет интерфейс для регистрации компонентов-*подписчиков*, заинтересованных в информации о его изменениях, и хранит множество зарегистрированных подписчиков. При изменениях он оповещает их с помощью вызова предназначенного для этого метода `update()`.

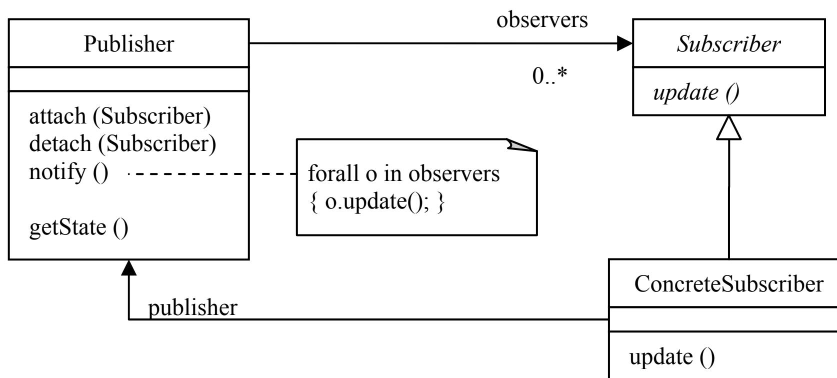

Рисунок 54. Структура классов подписчиков и издателя.

Структура. Основными компонентами являются *издатель* (или *субъект*) и *подписчики* (или *наблюдатели*). Подписчики реализуют общий интерфейс, у которого имеется метод `update()` для оповещения подписчика о том, что в состоянии издателя произошли изменения. Издатель хранит множество подписчиков, позволяет регистрировать или удалять их из этого набора. При возникновении изменений он оповещает все элементы этого множества при помощи метода `update()`.

Динамика. Можно использовать два вида обновления подписчиков: издатель сам может сообщать им о том, какие именно изменения произошли (схема проталкивания, push model), или после получения уведомления подписчик сам обращается к издателю, чтобы узнать, что именно изменилось (схема вытягивания, pull model). Вторая схема значительно более гибкая, она позволяет подписчикам получать только необходимую им информацию, в то время как согласно первой схеме каждый подписчик получает всю информацию о произошедших изменениях.

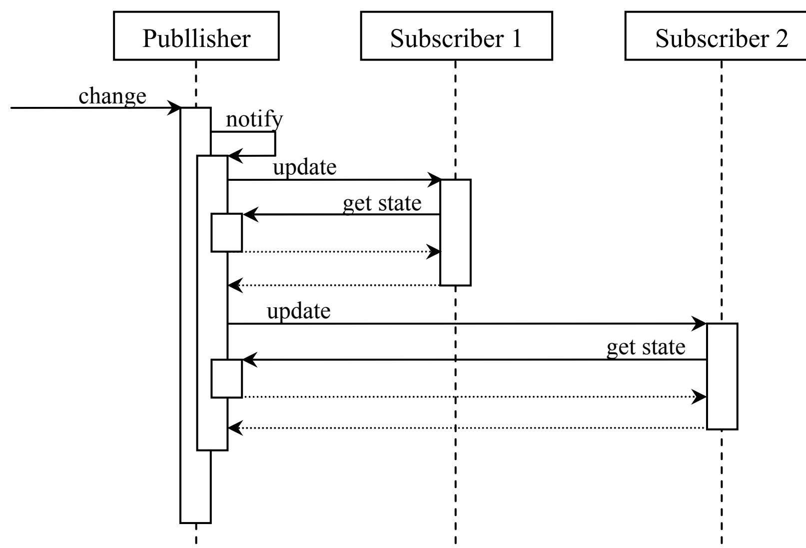

Рисунок 55. Сценарий оповещения об изменениях по схеме вытягивания.

Реализация. Основные шаги реализации следующие.

- Определить протокол обновления — будет ли использоваться простой метод `update()` или в качестве его параметров нужно будет передавать изменившийся объект-издатель и данные произошедшего изменения.
- Определить схему обновления одного подписчика: на основе проталкивания или на основе вытягивания информации.
- Определить отображение издателей на подписчиков. Если издателей много, а подписчиков мало, то хранение множеств подписчиков для каждого издателя может

быть неэффективным — для экономии памяти за счет времени поиска подписчиков можно использовать отдельную таблицу, отображающую издателей на подписчиков.

- Обеспечить гарантии целостности состояния издателя перед оповещением подписчиков.
- Обеспечить гарантии аккуратного удаления объекта-подписчика из системы — нужно, чтобы он был удален из всех списков оповещений.
- Если семантика обновлений сложна, например, если подписчик зависит от нескольких издателей, которые могут изменяться в рамках одной операции, то, возможно, потребуется выделить такие сложные связи в отдельный компонент, называемый *менеджером изменений (change manager)*. Такой компонент должен сам хранить отображение между издателями и подписчиками, определять стратегию проведения обновления и обновлять всех зависимых подписчиков по запросу от издателя. В качестве стратегии обновления может выступать механизм, гарантирующий подписчику получение только одного уведомления, если изменяются несколько издателей, от которых он зависит.

Следствия применения образца.

Достоинства

- Слабая связанность между издателем и подписчиками — издатель знает только, что у него есть несколько подписчиков с одинаковым интерфейсом.
- Удобным образом, не зависящим от числа участников, поддерживаются широковещательные оповещения.

Недостатки

- Низкая производительность в случае большого количества подписчиков — даже небольшое изменение требует оповестить их всех, и каждый из них будет пытаться провести обновление своего состояния.
- Неэффективное использование информации из-за необходимости оповещать всех подписчиков, даже тех, для которых выполненные изменения несущественны. Кроме того, подписчик может зависеть от нескольких издателей и не знать, какой именно издатель оповещает его об изменении. Чтобы исправить эту ситуацию, необходимо внесение изменений в протокол оповещения — предоставление дополнительной информации, как об изменившемся издателе, так и о виде изменения.

Примеры. Один из примеров мы уже видели — в рамках более широкого стиля «данные-представление-обработка» представления и обработчики являются подписчиками по отношению к модели-издателю.

Вариант этого образца с введением менеджеров изменений описан под названием «канал событий» (Event Channel) в спецификации службы оповещения о событиях в стандарте CORBA [2].

Идиомы

Идиома представляет собой типовое решение, определяющее специфическую структуризацию элементов кода на некотором языке программирования. Чаще всего это некоторый «трюк», с помощью которого можно придать программе на данном языке нужные свойства. При этом идиома может оказаться специфичной для языка и не иметь аналога в других языках. Кроме того, очень часто удачные идиомы при развитии языков программирования превращаются в новые синтаксические конструкции, делающие такую идиому ненужной.

Далее мы увидим примеры таких идиом в виде соглашений о построении кода компонентов JavaBeans, превратившихся в C# в элементы языка.

В этой лекции мы рассмотрим в качестве примера идиому «шаблонный метод» [3].

Шаблонный метод

Название. Шаблонный метод (template method).

Назначение. Фиксация общей схемы некоторого алгоритма, имеющего много вариантов, с предоставлением возможности реализовать эти варианты за счет переопределения отдельных его шагов, без изменения схемы в целом. При использовании этой идиомы нужно принимать во внимание следующие факторы.

Действующие силы.

- Инвариантные части алгоритма нужно записать один раз и переиспользовать, изменяя только варьирующиеся шаги и элементы.
- Иногда такие инвариантные части еще нужно выделить, чтобы сделать более понятным и более удобным для сопровождения код нескольких классов, реализующих близкие по назначению методы.
- Многие алгоритмы при их практическом использовании могут зависеть от большого числа факторов, изменяющихся гораздо чаще, чем общая схема такого алгоритма (вспомните принцип разделения политик и алгоритмов).

Решение. Общая схема алгоритма, которую нужно зафиксировать, помещается в абстрактный класс в виде метода, код которого реализует эту схему. В тех местах, где необходимо использовать какой-то варьирующийся элемент алгоритма, этот метод обращается к другому методу данного класса, который можно переопределить в классах-потомках.

Структура. Метод, реализующий основную схему алгоритма, называется *шаблонным методом*. Он пишется один раз в абстрактном базовом классе и не переопределяется в потомках. Методы, вызываемые им, делятся на следующие группы.

- *Конкретные операции*, реализация которых известна на момент написания метода и не должна изменяться.
- *Абстрактные операции*, которые представляют собой изменяемые части алгоритма. Они не имеют реализации и должны определяться в каждом классе-потомке в соответствии с теми вариациями, которые он вносит в базовый алгоритм.
- *Операции-перехватчики,* или *зацепки (hook operations),* которые также представляют собой изменяемые элементы алгоритма, но имеют некоторую реализацию по умолчанию, записанную в базовом классе. Эти операции могут переопределяться в классах-потомках, если представляемые ими элементы алгоритма нужно изменить по сравнению с имеющейся реализацией по умолчанию.
- *Фабричные методы (factory methods),* предназначенные для создания объектов, которые связаны с работой конкретного варианта алгоритма. Они реализуются по образцу, также называемому «фабричный метод». Суть такого метода в том, что при его вызове мы точно не знаем, объект какого конкретного класса будет создан — это зависит от текущей конфигурации системы.

Динамика. Типичный сценарий работы шаблонного метода показан на Рис. 56. Для наглядности на этой диаграмме операции, выполняемые в одном объекте, разделены между двумя виртуальными объектами: первый представляет все действия, выполняемые в рамках абстрактного класса, определяющего шаблонный метод, а второй — те действия, которые (пере)определяются в конкретном подклассе.

Реализация. Основные шаги реализации следующие.

- Определить основные шаги алгоритма. Определить набор данных, которыми он пользуется.
- Выделить среди шагов алгоритма те, которые зависят от конкретных политик. Определить для каждого такого шага абстрактный метод.

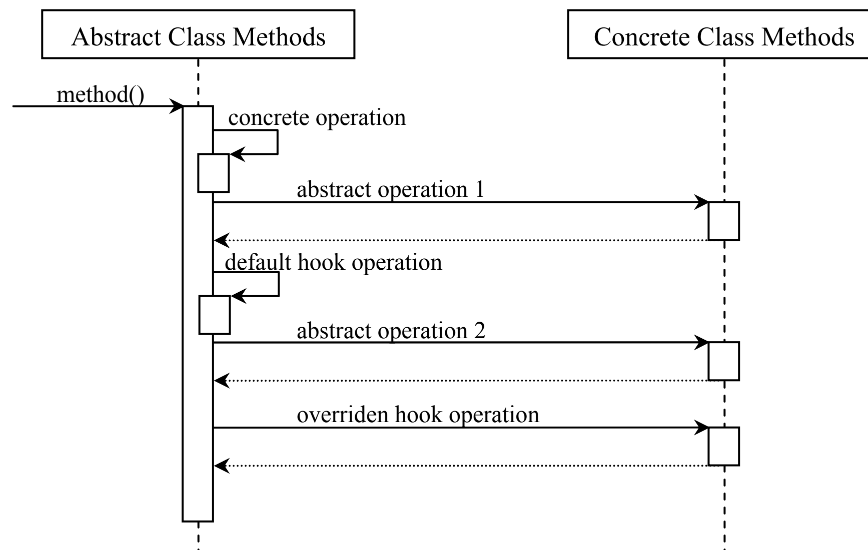

Рисунок 56. Сценарий работы шаблонного метода.

- Выделить среди данных, которыми пользуется алгоритм, те, чья структура зависит от политик или конкретного варианта алгоритма. Определить для порождения таких данных фабричные методы, а для операций над ними — абстрактные методы. Для их представления могут понадобиться дополнительные классы, но изменяемая часть данных должна, по возможности, находиться в абстрактном классе, определяющем шаблонный метод.
- Реализовать общую схему алгоритма в теле шаблонного метода. Выделить его элементы, используемые несколько раз или представляющие собой отдельные операции, в отдельные методы абстрактного класса.
- Определить, какие дополнительные возможности по вариации поведения алгоритма могут понадобиться. Определить для таких возможностей методы-перехватчики.
- Определить несколько наиболее часто используемых вариантов алгоритма. Реализовать их в виде подклассов определяющего шаблонный метод класса, определив в них методы, которые представляют абстрактные операции, и, если это необходимо, переопределив методы-перехватчики.

Следствия применения образца.

Достоинства

- Общая часть алгоритма реализуется явно и может быть легко переиспользована.
- Изменяемые части алгоритма могут варьироваться удобным образом, не влияя друг на друга и давая в результате различные его модификации.
- Алгоритм может быть параметризован большим набором политик, для каждой из которых возможна реализация по умолчанию.

Недостатки

- Снижение понятности кода за счет сложного потока управления.
- Снижение производительности в случае большого числа параметров, из которых в каждом конкретном варианте алгоритма используется лишь несколько.

Примеры. Шаблонные методы очень часто используются при построении библиотечных классов и каркасов приложений.

Жизненный цикл компонентов EJB реализован в виде шаблонного метода, в котором абстрактной операцией служит создание объектов данного компонента. Имеется также несколько операций-перехватчиков, позволяющих разработчику компонента специфическим образом обрабатывать переход компонента из одного состояния в другое.

Другой пример — реализация метода `start()`, запускающего отдельный поток в Java. Инструкции, выполняемые в рамках потока, помещаются в метод `run()` объекта класса `Thread` или класса, реализующего интерфейс `Runnable`. Этот метод служит операцией-перехватчиком для метода `start()` — реализация метода `run()` по умолчанию ничего не делает.

Образцы организации и образцы процессов

Образцы организации работ и образцы процессов существенно отличаются от остальных видов образцов, рассматриваемых здесь. Они фиксируют успешные практики по организации деятельности, связанной с разработкой ПО (или другими сложными видами деятельности). Такие образцы только поддерживают проектирование и разработку ПО, не давая вариантов самих проектных решений.

Образцы этого вида чаще всего извлекаются из форм организации работ и процессов, принятых в успешных компаниях-производителях ПО. Плодотворность их использования оценивается управленцами достаточно субъективно. При этом, однако, для признания некоторого вида организации работ образцом, необходимо успешное ее использование для решения одних и тех же задач в нескольких организациях.

Шаблон для описания таких образцов выглядит следующим образом.

- Название образца.
- Контекст использования, включающий основную решаемую задачу и начальные условия.
- Действующие силы — проблемы, ограничения, требования, рассуждения и идеи, под воздействием которых вырабатывается решение.
- Решение — описание используемой формы организации работ, выделяемых подзадач, выполняемых действий, используемых техник.
- Итоговый контекст — описание ожидаемых результатов использования образца, обоснование того, что его применение даст нужный эффект.

В качестве примера образца организации работ приведем процесс *инспекции программ (Fagan inspection process)*, определенный Майклом Фаганом (Michael Fagan) [4,5] (похожий процесс, называемый технической экспертизой, technical review, может быть найден в [6]).

Инспекция программ по Фагану

Название. Инспекция программ по Фагану (Fagan inspection process).

Контекст использования. Поиск ошибок на ранних этапах разработки программного обеспечения — при подготовке требований, проектировании, начальных этапах кодирования, планировании тестов.

Действующие силы.

- Усилия, необходимые для исправления ошибки, и, соответственно, ее стоимость возрастают в зависимости от этапа проекта, на котором она обнаружена. Из эмпирических данных известно, что каждый раз при переходе через границу между фазами (при использовании водопадной модели разработки) подготовка требований – проектирование – кодирование – тестирование – эксплуатация трудозатраты на исправление найденных на данном этапе ошибок возрастают в 3-5 раз. При использовании итеративных моделей затраты возрастают меньше, но не намного. Поэтому, чем раньше ошибки будут обнаруживаться, тем эффективней будет разработка в целом.
- Членам команды разработчиков надо понимать, над чем работает каждый из них и какие решения он использует. Это помогает значительно повысить эффективность собственной работы.

- Каждый артефакт — требования, проектные документы, код, тестовые планы — должен быть подготовлен на нужном уровне качества, прежде чем он будет использован для дальнейшей работы.
- Знания о найденных ошибках позволяют членам команды избегать их повторения, а также обращать больше внимания на компоненты, которые оказались наиболее подвержены ошибкам на предыдущих этапах.

Решение. Несколько членов команды разработчиков проводят тщательную инспекцию результатов работы одного из них. Такие инспекции основываются на *первичных документах*, чтобы проверить соответствие им *вторичных документов*. Первичные и вторичные документы для каждого вида деятельности в ходе разработки, для которых проведение инспекций эффективно, представлены в Таблице 8.

Выделяются следующие роли участвующих в процессе инспекции лиц.

- *Ведущий (moderator).* Он руководит проведением инспекции, руководит собраниями, фиксирует обнаруженные ошибки, назначает время проведения собраний, сроки подготовки отчетов, следит за исправлением найденных ошибок.
  В качестве ведущего должен использоваться компетентный разработчик или архитектор, не вовлеченный в проект, материалы которого инспектируются.
- *Автор (author).* Это автор первичного документа или человек, имеющий достаточно полное представление о нем. Его обязанности — подготовить рассказ об основных положениях первичного документа и отвечать на вопросы, возникающие у членов инспектирующей команды по его поводу.
- *Интерпретатор (reader).* Это автор вторичного документа, который разработан в соответствии с первичным. Его обязанности — объяснить участникам инспекции основные идеи, лежащие в основе его интерпретации первичного документа, и отвечать на их вопросы по поводу вторичного документа.
- *Инспектор (tester).* В ходе всей инспекции он анализирует вторичный документ, проверяя его на соответствие первичному.

Вид деятельности	**Первичные документы**	**Вторичные документы**
Анализ требований	Модели предметной области, составленные заказчиками и пользователями требования	Требования к ПО
Проектирование	Требования к ПО	Описание архитектуры, проектная документация
Кодирование	Проектная документация	Код, проектная документация на отдельные компоненты
Тестирование	Требования к ПО, проектная документация, код	Тестовые планы и наборы тестовых вариантов

Таблица 8. Первичные и вторичные документы на разных этапах разработки.

Обычно рекомендуется использовать не более 4-х человек в команде, проводящей инспекцию. Расширение ее возможно в особых случаях и только за счет разработчиков, которым непосредственно придется иметь дело с инспектируемыми вторичными документами.

Сам процесс инспекции состоит из следующих шагов.

1. *Планирование (planning).*
   На этом шаге ведущий должен убедиться в том, что первичный и вторичный документы готовы к проведению инспекции — они существуют, написаны достаточно понятно, с достаточной степенью детализации.

Кроме того, на этом шаге проводиться планирование всего хода инспекции — определяются участники, их роли, назначаются сроки проведения собраний и время, выделяемое на выполнение каждого шага.

2. *Обзор (review).*
   Проводится собрание, на котором автор представляет наиболее существенные положения первичного документа и отвечает на вопросы участников о нем.
   Первичный и вторичный документы выдаются на руки участникам инспекции для дальнейшей работы.
   Ведущий объясняет задачи данной инспекции, вопросы и моменты, на которые стоит обратить особое внимание, а также сообщает, какие ошибки были уже обнаружены в рассматриваемых документах, чтобы участники группы имели представление об их проблемных местах.
3. *Подготовка (preparation).*
   Каждый из участников тщательно изучает оба документа самостоятельно, пытаясь понять заложенные в них решения и проследить их реализацию.
   Часто на этом этапе обнаруживаются ошибки, но гораздо меньше, чем на следующем.
4. *Совместная инспекция (inspection meeting).*
   Проводится совместное собрание, на котором интерпретатор рассказывает об основных идеях и техниках, использованных во вторичном документе, а также объясняет, почему были приняты те или иные решения и почему они соответствуют первичному документу.
   Участники задают вопросы и акцентируют внимание на проблемных местах. Как только ведущий по ходу собрания замечает ошибку (или кто-то обращает его внимание на нее), он сообщает о ней и убеждается, что все участники согласны с тем, что это именно ошибка, т.е. несоответствие между первичным и вторичным документами. Каждая ошибка фиксируется, описывается ее положение, она классифицируется по некоторой схеме, например, критическая (приводящая к ошибке в работе системы) или некритическая (связанная с опечатками, излишней сложностью или неудобством интерфейса и пр.).
5. *Доработка (rework).*
   В ходе доработки интерпретатор исправляет обнаруженные ошибки.
6. *Контроль результатов (follow-up).*
   Результаты доработки проверяются ведущим. Он проверяет, что все найденные ошибки были исправлены и что не было внесено новых ошибок. Если по результатам инспекции было переработано более 5% вторичного документа, следует провести полную инспекцию вновь. Иначе ведущий сам определяет, насколько документ подготовлен к дальнейшему использованию.
   Кроме того, ведущий подготавливает отчет обо всех обнаруженных ошибках для последующего использования в других инспекциях и при оценке качества результатов разработки.

Итоговый контекст. В результате проведения инспекций повышается качество проектных документов и кода, разработчики знакомятся ближе с работой друг друга и с задачами проекта в целом, углубляют понимание проблем проекта и используемых в нем решений. Кроме того, руководитель проекта получает надежные данные о качестве результатов разработки.

Руководитель должен понимать, что *результаты инспекций не должны использоваться как показатель качества работы разработчиков*, иначе все положительные эффекты от их проведения пропадают — разработчики начинают неохотно участвовать в инспекциях, скрывают детали своей работы, снисходительнее относятся к ошибкам других в расчете на взаимность и пр.

При выполнении этого условия инспекции являются эффективным средством обнаружения ошибок на ранних этапах. Статистика показывает, что они находят до 80% ошибок, обнаруживаемых за весь период разработки ПО.

Литература к Лекции 8

[1] G. E. Krasner, S. T. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. Journal of Object-Oriented Programming, 1(3), pp. 26–49, August-September 1988. SIGS Publications, NY, USA, 1988.

[2] CORBA Event Service Specification, version 1.2. Object Management Group, October 2004. Доступен как http://www.omg.org/cgi-bin/apps/doc?formal/04-10-02.pdf.

[3] Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер-ДМК, 2001.

[4] M. E. Fagan. Design and Code inspections to reduce errors in program development. IBM Systems Journal, vol. 15, No. 3, pp. 258–287, 1976.

[5] M. E. Fagan. Advances in Software Inspections. IEEE Transactions on Software Engineering, vol. 12, No. 7, pp. 744–751, July 1986.

[6] S. Ambler. Process Patterns: Building Large-Scale Systems using Object Technology. Cambridge University Press, Cambridge, MA, 1998.

[7] М. Фаулер и др. Архитектура корпоративных программных приложений. М.: Вильямс, 2004.

[8] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. Pattern-Oriented Software Architecture. A System of Patterns. Wiley, 2002.

[9] Э. Дж. Брауде. Технология разработки программного обеспечения. СПб.: Питер, 2004.