ЧАСТЬ V

Обработка вариаций с применением шаблонов проектирования

В этой части

В этой части книги мы рассмотрим еще один практический пример. В данном случае требования к системе будут анализироваться поочередно, без их общего предварительного обсуждения. Будем считать, что существует некоторая уже готовая и функционирующая система, по отношению к которой выдвигаются новые требования. Появление каждого нового требования заставляет выполнить поиск наилучшего варианта изменения уже существующего программного кода. Такой подход позволит нам последовательно познакомиться с несколькими новыми шаблонами проектирования, по одному для каждого изменения требований.

Глава	Предмет обсуждения
14	Шаблон Strategy. Обработка изменений в алгоритме и бизнес-правилах
15	Шаблон Decorator. Динамическое добавление функциональности до или после уже существующих правил поведения объекта
16	Шаблон Singleton и шаблон Double-Checked Locking. Гарантированное создание не более одного экземпляра объекта заданного класса даже в многопоточном окружении
17	Шаблон Observer. Извещение одной части системы о событии, произошедшем в другой ее части
18	Шаблон Template Method. Решение проблемы существования нескольких различных прецедентов, использующих одну и ту же процедуру, но с отличающейся последовательностью выполнения отдельных ее этапов
19	Шаблон Factory Method. Передача функции выбора класса объекта, экземпляр которого будет создан, производным классам
20	Метод матрицы анализа. Выявление множества присутствующих в проблемной области вариаций и отображение их в шаблоны

Для каждого нового требования, предъявляемого к системе, мы рассмотрим такие возможные варианты реализации, как:

- использование переключателей в программном коде;
- специализация на основе механизма наследования;
- инкапсуляция изменений с последующим включением данного объекта или организацией ссылки на него.

Обсуждение этих альтернатив поможет нам уяснить, что среди многих шаблонов существует определенное сходство: обычно различные шаблоны предусматривают обработку вариаций и удовлетворение новых требований в одном и том же стиле.

Глава 14

Шаблон Strategy

Введение

В этой главе мы познакомимся с новым практическим примером, взятым из области электронной коммерции. В уже существующую систему поддержки розничных продаж через Internet будет предложено внести определенное изменение. Решая поставленную задачу, мы познакомимся с шаблоном Strategy (стратегия). В каждой последующей главе, вплоть до главы 20, *Матрица анализа*, наш новый пример будет последовательно усложняться.

Здесь мы выполним следующее.

- Обсудим оптимальный подход к обработке новых требований к существующей системе.
- Познакомимся с новым учебным примером.
- Опишем шаблон Strategy и рассмотрим, как его можно использовать для обработки новых требований в нашем примере.
- Проанализируем ключевые особенности шаблона Strategy.

Оптимальный подход к обработке новых требований

Как в обычной жизни, так и при проектировании программных приложений каждому из нас неоднократно приходилось искать оптимальный подход к решению задачи или устранению проблемы. Большинство из нас на собственном опыте знает, что решения, обеспечивающие моментальную выгоду, часто приводят к серьезным затруднениям в будущем. Например, каждый водитель знает, что после определенного пробега масло в автомобиле обязательно следует поменять. Конечно, нет необходимости менять масло через каждые 5 000 км, однако откладывать полную смену масла в двигателе до достижения пробега в 50 000 км нельзя (иначе необходимость замены масла отпадет сама собой — автомобиль просто выйдет из строя!). Другой пример — представьте себе письменный стол. Очень удобно хранить все нужные бумаги и канцелярские принадлежности под рукой, прямо на его поверхности. Однако это удобство будет сохраняться недолго, и через некоторое время стол окажется завален грудами бумаг, папок, книг и тому подобного, в которых найти что-либо будет просто невозможно. Кстати, различные катастрофы чаще всего являются отдаленным следствием псевдо-оптимальных решений, принятых на скорую руку, без учета длительной перспективы.

К большому сожалению, многие специалисты в области разработки программного обеспечения все еще не усвоили этот урок. Множество проектов разрабатывалось с ориентацией на решение только конкретных, сиюминутных задач без учета проблемы сопровождения системы в будущем. Существует несколько предпосылок, способствующих сохранению тенденции к игнорированию таких важнейших долгосрочных аспектов создаваемых проектов, как простота его сопровождения или удобство внесения изменений. Чаще всего указывают на следующие причины.

- Очень сложно предвидеть, как требования к системе будут изменяться в будущем.
- Если заняться выяснением, как требования к системе будут изменяться в будущем, этап анализа не закончится никогда.
- Если начать писать создаваемое программное обеспечение так, чтобы оно позволяло легко добавлять в него новую функциональность, проектирование не закончится никогда.
- У нас на это нет ни времени, ни денег.

Существуют две крайности.

- Чрезмерное углубление в анализ и разнообразные аспекты проектирования, что обычно называют "параличом от анализа".
- Принятие скороспелого решения с немедленным переходом к кодированию без какого-либо анализа долгосрочных аспектов проекта. В результате, очень скоро потребуется начать новый проект, так как подобная недальновидность порождает слишком много проблем.

Поскольку руководство обычно больше интересуется сроками сдачи системы, а не вопросами ее сопровождения, то такой результат не удивителен. После недолгого размышления становится совершенно очевидным, что именно удерживает разработчиков программного обеспечения от проведения анализа возможных альтернатив. Не вызывает сомнения, что большинство из них просто убеждены в том, что проектирование с поддержкой возможных будущих изменений непременно потребует дополнительных затрат.

Но это утверждение вовсе необязательно будет справедливо. В действительности, истинно как раз обратное утверждение. Если взглянуть на систему в целом и попытаться определить, как она будет изменяться в будущем, обычно удается найти лучшее проектное решение, реализация которого потребует фактически такого же количества времени, как и при обычной практике проектирования.

Именно такой подход к проектированию — с предварительным анализом и учетом возможных изменений — мы используем при обсуждении приведенного ниже учебного примера. Здесь очень важно еще раз обратить ваше внимание на то, что мы будем заранее ожидать появления изменений, и задача наша состоит в том, чтобы выяснить, *где* они будут происходить, а не в том, чтобы определить, какими именно они будут. Данный подход основан на принципах, предложенных в книге "банды четырех".

- "Проектируйте интерфейс, а не его реализацию."[1]

---

[1] *Gamma E., Helm R., Johnson R., Vlissides J.* Design Patterns: Elements of Reusable Object-Oriented Software, *Reading, MA: Addison-Wesley, 1995, с. 18.*

- "Отдавайте предпочтение композиции объектов, а не наследованию классов."[2]
- "Выясните, что может быть подвержено изменениям в вашем проекте." Этот подход — прямая противоположность фокусированию внимания на причинах, способных вызвать перепроектирование системы. Вместо того, чтобы искать то, что способно привести к изменению проекта, следует отобрать то, для чего нужно обеспечить возможность изменения в будущем без необходимости перепроектирования. Суть подхода заключается в инкапсуляции изменяемого на концептуальном уровне, что является лейтмотивом многих шаблонов проектирования."[3]

Я настоятельно вам рекомендую, столкнувшись с необходимостью изменения программного кода в связи с появлением новых требований, проанализировать возможность применения приведенных ниже стратегий. Использование этих стратегий не приведет к существенному удорожанию проекта или его реализации, но позволит получить значительные преимущества в будущем.

Однако я не предлагаю следовать этим стратегиями вслепую. Всегда можно оценить качество альтернативного варианта, проверив, насколько полно он отвечает показателям хорошего объектно-ориентированного проекта. Такой же, в сущности, подход был использован при выведении шаблона Bridge в главе 9, *Шаблон Bridge*, где мы сравнивали качество альтернативных вариантов на основании того, какой из них лучше соответствует принципам объектно-ориентированного программирования.

Исходные требования к учебному проекту

Предположим, что перед нами поставлена задача разработать систему поддержки электронной розничной торговли в пределах Соединенных Штатов Америки. Общая структура приложения включает объект-контроллер, предназначенный для обработки поступающих запросов на продажу. Он определяет момент поступления запроса на формирование заказа и передает поступивший запрос объекту выписки счета для дальнейшей обработки.

Общий вид исходного варианта системы представлен на рис. 14.1.

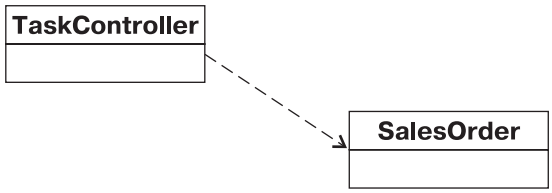

РИС. 14.1. ***Схема обработки заказа в системе розничной электронной торговли***

[2] *Там же, с. 20.*

[3] *Там же, с. 29.*

Класс **SalesOrder** имеет следующие функции.

- Предоставление графического интерфейса для заполнения заказа.
- Вычисление суммы налога.
- Обработка заказа и печать накладной на продажу.

Некоторые из этих функций, вероятно, следует реализовать с помощью других объектов. Например, класс **SalesOrder** не обязательно должен непосредственно заниматься выводом на печать, скорее его назначение состоит в хранении информации о поступившем заказе на продажу. Отдельные объекты класса **SalesOrder** для распечатки накладной на продажу могут обращаться к одному и тому же объекту класса **SalesTicket**.

Обработка новых требований

Предположим, что в процессе работы над этим приложением было выдвинуто новое требование об изменении способа начисления налога. Теперь система должна будет поддерживать начисление налога и для заказов тех клиентов, которые находятся за пределами Соединенных Штатов. Как минимум, потребуется поддержка новых правил начисления таких налогов.

Как организовать в системе поддержку этих новых правил? Можно попробовать еще раз воспользоваться уже существующим классом **SalesOrder** и обрабатывать данную ситуацию просто как новый вид коммерческого заказа с отличающимся набором правил налогообложения. Например, для обработки заказов из Канады можно создать новый класс с именем **CanadianSalesOrder**, производный от класса **SalesOrder**, в котором переопределяются правила налогообложения. Данный вариант решения представлен на рис. 14.2.

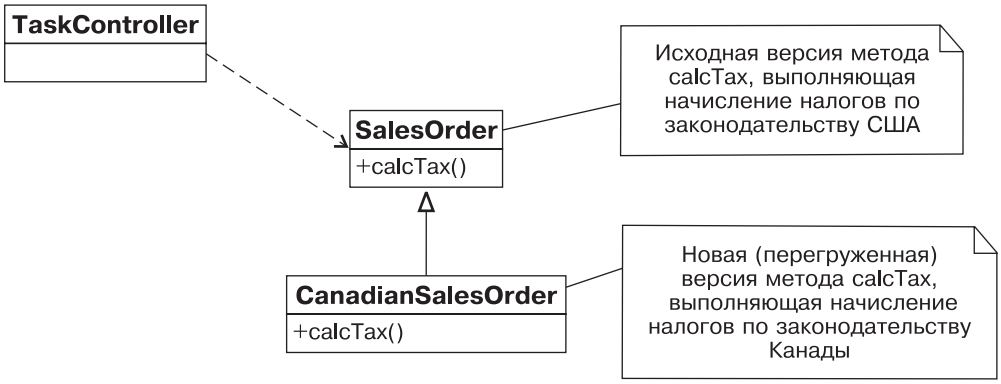

РИС. 14.2. ***Возможный вариант схемы обработки заказа в системе розничной электронной торговли***

Напомню, что шаблоны проектирования постоянно демонстрируют применение фундаментального правила идеологии шаблонов: "Отдавайте предпочтение композиции объектов, а не наследованию классов".[4] В решении на рис. 14.2 использован прямо противоположный подход! Другими словами, изменение в налоговых правилах обрабатывается здесь с использованием механизма наследования посредством определения производного класса, включающего поддержку новых правил.

Какое другое решение можно предложить? Следуя изложенному выше правилу, нужно попытаться найти то, что является в проекте изменяемым, и инкапсулировать эту концепцию.[5]

Требуемый результат достигается в два этапа.

1. Найти то, что изменяется, и инкапсулировать это в соответствующий специализированный класс.
2. Поместить этот класс в другой класс.

В нашем примере уже известно, что изменяемой концепцией являются правила налогообложения. Инкапсуляция в этом случае подразумевает создание абстрактного класса, определяющего решение поставленной задачи на концептуальном уровне, с последующим созданием конкретных производных классов для каждого из существующих вариантов. Другими словами, необходимо создать абстрактный класс ***`CalcTax`***, определяющий интерфейс, требуемый для решения поставленной задачи, а затем создать конкретные классы, производные от него, для каждой существующей версии. Схематично данное решение представлено на рис. 14.3.

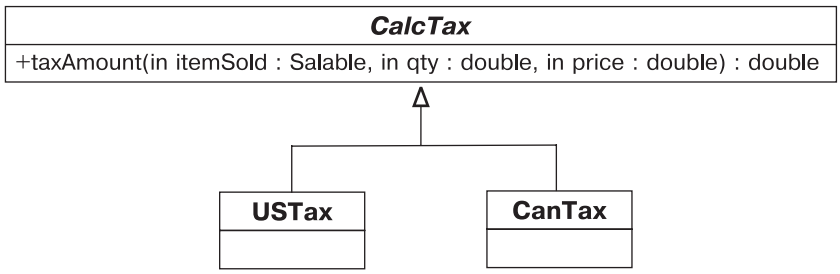

РИС. 14.3. ***Инкапсуляция правил налогообложения***

Теперь на общей схеме приложения вместо наследования классов используется композиция. Имеется в виду, что вместо создания различных версий класса обработки заказа (с помощью механизма наследования) включение вариации выполнено с помощью композиции объектов. Существует только один класс **`SalesOrder`**, который содержит объект класса ***`CalcTax`***, предназначенный для сокрытия и обработки возможных вариаций (рис. 14.4).

---

[4] *Там же, с. 20.*

[5] *Там же, с. 29.*

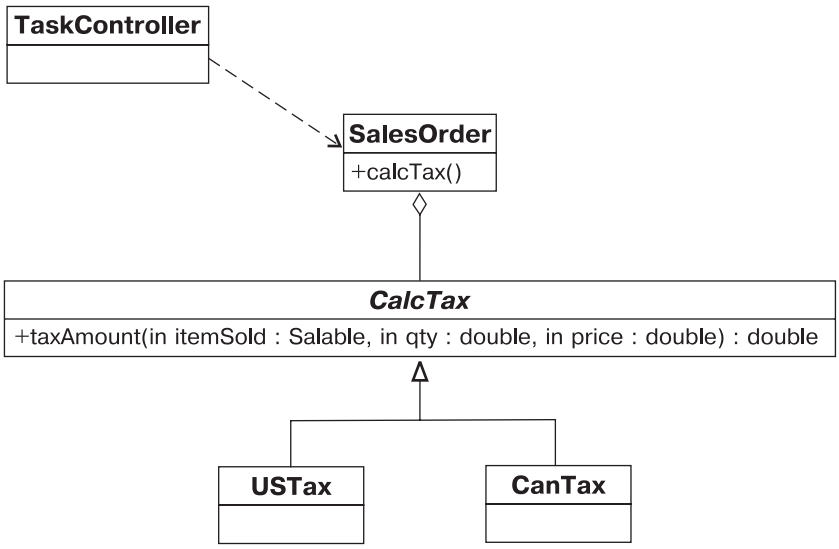

РИС. 14.4. *Вариант схемы приложения, в котором предпочтение отдается композиции, а не наследованию*

UML-диаграммы

Язык UML позволяет определять параметры в методах. Для этого параметр и его тип указываются в скобках, входящих в описание метода.

В частности, на рис. 14.4 показано, что метод `taxAmount()` имеет три параметра:

- `itemSold` (тип **`Salable`**);
- `qty` (тип **`double`**);
- `price` (тип **`double`**).

Все эти параметры являются входными, что обозначено ключевым словом `in`. Возвращаемое методом `TaxAmount` значение также имеет тип **`double`**.

Таким образом, нами определен общий интерфейс для объектов класса ***`CalcTax`***. Вероятно, можно было бы определить класс **`Saleable`**, предназначенный для хранения характеристик товара (и правил его налогообложения). Объект **`SalesOrder`** мог бы передать объект этого класса объекту **`CalcTax`** наряду с данными о заказанном количестве и установленной ценой. Этой информации классу ***`CalcTax`*** было бы вполне достаточно.

Дополнительное преимущество такого подхода состоит в повышении связности в системе, поскольку теперь расчет суммы налога выполняется в специализированном классе. Еще одно преимущество состоит в том, что при добавлении нового варианта правил налогообложения достаточно будет определить еще один класс, производный от класса ***`CalcTax`***, включающий реализацию этих правил.

Наконец, в новой версии проще обеспечить передачу ответственности. В варианте, основанном на использовании механизма наследования, класс **`TaskController`**

должен будет самостоятельно определять, какой тип класса **`SalesOrder`** следует использовать. В новом варианте для этой цели можно использовать как класс **`TaskController`**, так и класс **`SalesOrder`**. В последнем случае потребуется подключить некоторый объект конфигурации, который и будет определять, какой именно вариант начисления налога следует использовать в каждом случае (вероятно, тот же самый объект будет использоваться и классом **`TaskController`**). Данный вариант схемы приложения показан на рис. 14.5.

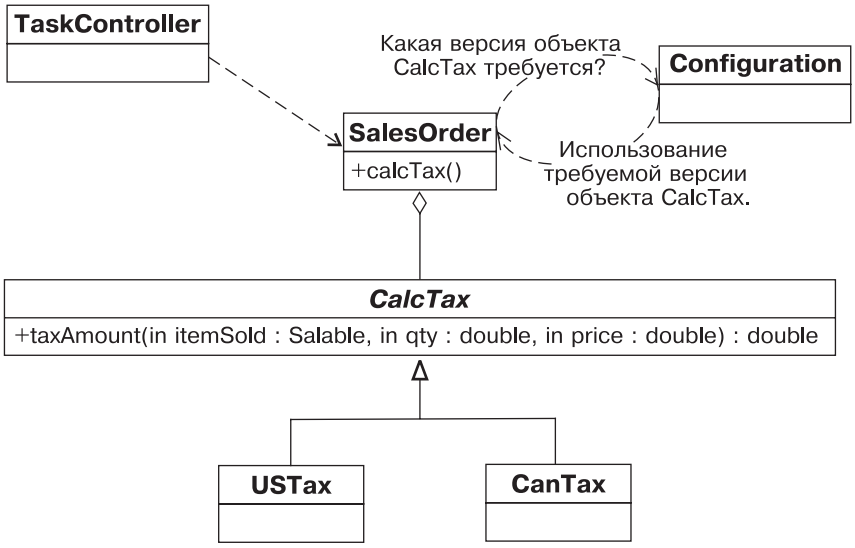

РИС. 14.5. ***Объект `SalesOrder` обращается к объекту `Configuration`, чтобы определить, какой тип объекта класса `CalcTax` следует использовать***

Такой подход позволяет бизнес-правилам изменяться независимо от объекта **`SalesOrder`**, который эти правила использует. Обратите внимание на то, что эта схема хорошо работает как для уже существующих вариаций, так и для любых вариаций, которые могут появиться в будущем. По существу, подобная инкапсуляция семейства алгоритмов в абстрактном классе (***`CalcTax`***) с последующим попеременным выбором одного из членов этого семейства и представляет собой шаблон Strategy.

Назначение шаблона проектирования Strategy

Согласно определению "банды четырех" назначение шаблона Strategy состоит в следующем.

> Определение семейства алгоритмов, инкапсуляция каждого из них и обеспечение их взаимозаменяемости. Шаблон Strategy позволяет менять выбранный алгоритм независимо от объектов-клиентов, которые его используют.[6]

---

[6] *Там же, с. 315.*

В основу шаблона Strategy положено несколько принципов.

- Объекты обладают обязательствами.
- Различные специфические реализации этих обязательств проявляются за счет использования полиморфизма.
- Существует потребность в управлении несколькими различными реализациями того, что концептуально является одним и тем же алгоритмом.
- Следует считать хорошим стилем проектирования отделение существующих в проблемной области поведений друг от друга — т.е. уменьшение связанности между ними. Это позволяет вносить изменения в класс, ответственный за некоторое поведение, без какого-либо неблагоприятного воздействия на другие классы.

Основные характеристики шаблона Strategy

Назначение	Позволяет использовать различные бизнес-правила или алгоритмы в зависимости от контекста
Задача	Выбор алгоритма, который следует применить, в зависимости от типа выдавшего запрос клиента или обрабатываемых данных. Если используется правило, которое не подвержено изменениям, нет необходимости обращаться к шаблону Strategy
Способ решения	Отделение процедуры выбора алгоритма от его реализации. Это позволяет сделать выбор на основании контекста
Участники	• Класс ***Strategy*** определяет, как будут использоваться различные алгоритмы • Конкретные классы **ConcreteStrategy*X*** реализуют эти различные алгоритмы • Класс **Context** использует конкретные классы **ConcreteStrategy*X*** посредством ссылки на конкретный тип абстрактного класса ***Strategy***. Классы ***Strategy*** и **Context** взаимодействуют с целью реализации выбранного алгоритма (в некоторых случаях классу ***Strategy*** требуется посылать запросы классу **Context**). Класс **Context** пересылает классу ***Strategy*** запрос, поступивший от его класса-клиента
Следствия	• Шаблон Strategy определяет семейство алгоритмов • Это позволяет отказаться от использования переключателей и/или условных выражений • Вызов всех алгоритмов должен осуществляться стандартным образом (все они должны иметь один и тот же интерфейс). Взаимодействие между классами **ConcreteStrategy*X*** и **Context** может потребовать введения в класс **Context** дополнительных методов типа `getState`
Реализация	Класс, который использует алгоритм (**Context**), включает абстрактный класс (***Strategy***), обладающий абстрактным методом, определяющим способ вызова алгоритма. Каждый производный класс реализует один требуемый вариант алгоритма. *Замечание*. Метод вызова алгоритма не может быть абстрактным, если требуется реализовать некоторое поведение, принимаемое по умолчанию. *Замечание*. В исходном варианте шаблона Strategy ответственность за выбор конкретной реализации возлагается на объект-клиент – т.е. на контекст шаблона Strategy

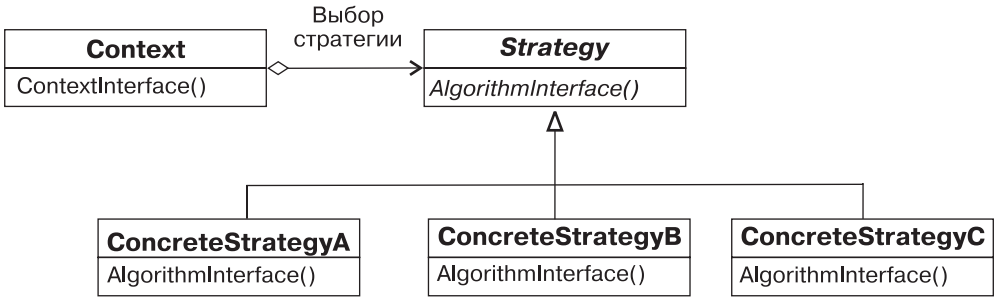

РИС. 14.6. *Стандартное упрощенное представление шаблона Strategy*

Дополнительные замечания о шаблоне Strategy

Однажды, когда на занятиях по изучению шаблонов проектирования обсуждался пример с системой электронной торговли, кто-то в аудитории задал мне вопрос: "А знаете ли вы, что в Англии люди по достижении определенного возраста не платят налог при покупке продовольственных товаров?". Эта информация была для меня новостью, и вполне естественно, что интерфейс объекта `CalcTax` не позволял обрабатывать подобные ситуации. Существует по меньшей мере три способа решения данной проблемы.

1. Передавать сведения о возрасте покупателя от объекта класса `Customer` объекту класса ***`CalcTax`*** и использовать их по мере необходимости.
2. Выбрать более общий подход: передавать объекту класса ***`CalcTax`*** весь объект класса `Customer` в целом и опрашивать его в случае необходимости.
3. Еще повысить уровень обобщения, передавая объекту класса ***`CalcTax`*** ссылку на объект класса `SalesOrder` (т.е. его указатель `this`), и позволить объекту класса ***`CalcTax`*** опрашивать его.

Хотя для обработки подобной ситуации обязательно потребуется изменить классы `SalesOrder` и ***`CalcTax`***, совершенно очевидно, как это можно сделать. Маловероятно, что выполнение требуемых действий вызовет появление в системе каких-либо проблем.

Формально шаблон Strategy представляет собой средство инкапсуляции алгоритмов. Однако на практике он может использоваться для инкапсуляции правил фактически любого вида. В общем случае, если на этапе анализа требований становится известно о существовании различных бизнес-правил, применяемых в различных ситуациях, обязательно следует рассмотреть возможность применения шаблона Strategy, эффективно обрабатывающего вариации такого рода.

Шаблон Strategy требует, чтобы алгоритмы (бизнес-правила) были инкапсулированы вне класса, который их использует (`Context`). Это означает, что информация, необходимая для выполнения этих алгоритмов, должна быть передана или получена каким-то иным способом.

Единственный серьезный недостаток, который я обнаружил в шаблоне Strategy, — это большое количество дополнительных классов, которые потребуется создавать. В случае, когда овчинка стоит выделки, можно предложить несколько рекомендаций по уменьшению количества требуемой работы, если имеется контроль над всеми стратегиями. В подобных случаях при использовании языка C++ можно создать файл заголовка абстрактной стратегии, содержащий имена всех файлов заголовков конкретных классов стратегий. Кроме того, создается cpp-файл абстрактной стратегии, содержащий программный код всех конкретных стратегий. При использовании языка Java в классе абстрактной стратегии создаются внутренние классы, содержащие код конкретных стратегий. Однако этого не следует делать, если отсутствует возможность контроля над всеми стратегиями — т.е. если некоторые алгоритмы будут реализовать другие программисты.

Резюме

Шаблон Strategy — это способ определить семейство алгоритмов. Концептуально все эти алгоритмы решают одну и ту же задачу, но различаются между собой способом ее решения.

Мы рассмотрели пример, в котором используется семейство алгоритмов начисления налогов. В системе поддержки международной электронной розничной торговли необходимо использовать различные алгоритмы начисления налогов для покупателей из различных стран. Шаблон Strategy позволяет инкапсулировать эти правила в одном абстрактном классе, от которого производится требуемое семейство конкретных классов.

За счет определения различных вариантов выполнения алгоритма как производных от одного абстрактного класса главный модуль (в нашем примере это класс **`SalesOrder`**) сможет не принимать во внимание, какую именно версию алгоритма он фактически использует. Такой подход упрощает подключение новых вариантов алгоритма, но, одновременно, требует некоторых мер по управлению ими. Об этом мы поговорим позже, в главе 20, *Матрица анализа*.

Глава 15

Шаблон Decorator

Введение

В этой главе мы продолжим обсуждение нового примера — приложения поддержки электронной розничной торговли, начатое в главе 14, *Шаблон Strategy*.

Здесь мы выполним следующее.

- Сформулируем новое требование к создаваемой системе, связанное с добавлением заголовка и нижнего колонтитула к выводимой на печать накладной.
- Убедимся, что использование шаблона Decorator позволяет удовлетворить новые требования с необходимой гибкостью.
- Рассмотрим, как шаблон Decorator может использоваться для управления вводом-выводом (особенно при работе с языком Java).
- Проанализируем ключевые особенности шаблона Decorator.
- Познакомимся с моим опытом применения шаблона Decorator на практике.

Несколько дополнительных деталей

На рис. 14.2 представлена исходная структура нашего учебного проекта. На рис. 15.1 эта структура изображена более подробно. Здесь показано, что объект класса **`SalesOrder`** для вывода на печать накладной использует объект класса **`SalesTicket`**.

В главе 14 мы решили, что для начисления налога на поступивший заказ объект класса **`SalesOrder`** будет использовать объект класса ***`CalcTax`***. Для выполнения операций печати объект класса **`SalesOrder`** будет обращаться к объекту класса **`SalesTicket`**, предназначенному для вывода на печать накладной. В результате получилась прекрасная и продуманная схема приложения.

Предположим, что в процессе создания приложения выдвигается новое требование, заключающееся в том, что в накладную, выводимую классом **`SalesTicket`**, необходимо добавить заголовок и нижний колонтитул.

Как можно реализовать это новое требование? Если предполагается, что создаваемая система будет использоваться только одной компанией, то вопрос решается очень легко — достаточно добавить код вывода заголовка и колонтитула в класс **`SalesTicket`**, как показано на рис. 15.2.

В этом варианте решения функции управления печатью возложены непосредственно на класс **`SalesTicket`** и реализованы в виде флажков, указывающих, следует ли печатать заголовок и/или нижний колонтитул.

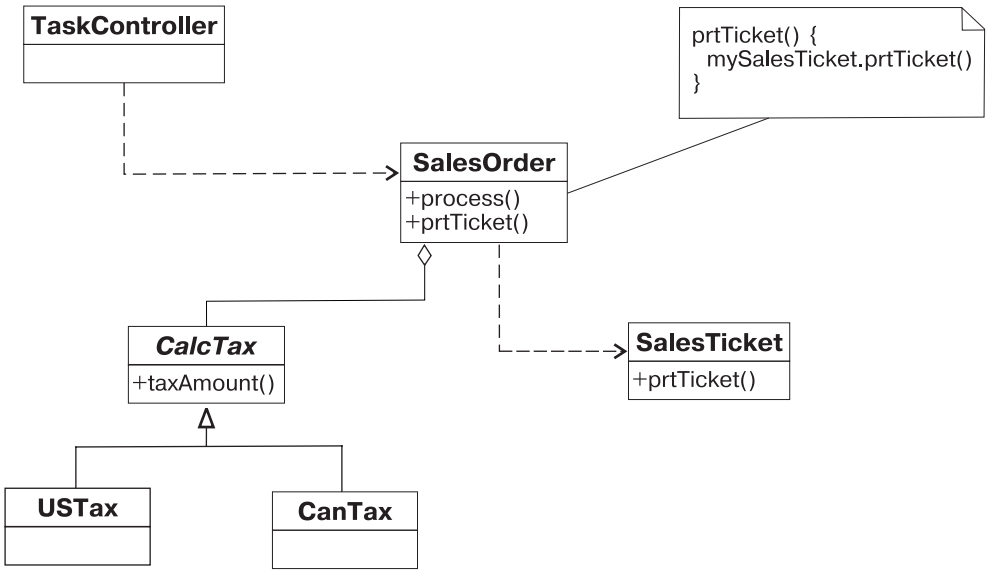

РИС. 15.1. *Класс* `SalesOrder` *использует класс* `SalesTicket`

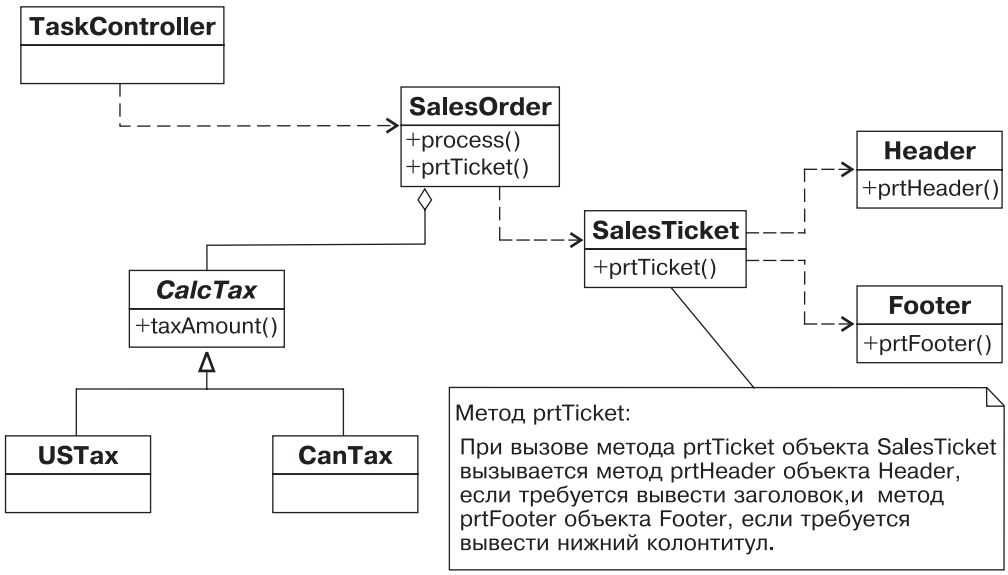

РИС. 15.2. *Класс* `SalesOrder` *использует класс* `SalesTicket` *с различными параметрами*

Такое решение работает вполне удовлетворительно, если не приходится иметь дело со слишком большим количеством параметров или если заголовки накладных не изменяются.

Если же существует множество различных типов заголовков и нижних колонтитулов, и при печати каждой накладной используется только один из возможных вариантов их сочетания, целесообразно перейти к схеме с одним шаблоном Strategy для заголовка и другим шаблоном Strategy для нижних колонтитулов.

А как поступить, если одновременно потребуется напечатать несколько типов заголовка и/или нижнего колонтитула? Или если порядок вывода заголовков и/или нижних колонтитулов будет изменяться? Очевидно, что количество возможных комбинаций в этом случае будет возрастать чрезвычайно быстро.

В ситуациях, подобных данной, пригодится шаблон Decorator (декоратор). Вместо определения новых методов, он предлагает управлять добавленной функциональностью посредством формирования цепочки связанных функций, упорядоченных в требуемой последовательности. Шаблон Decorator отделяет динамическое построение такой цепочки от объекта-клиента, который ее использует (в нашем случае им является объект класса **`SalesOrder`**).

Шаблон Decorator

Согласно определению "банды четырех" назначение шаблона Decorator состоит в следующем.

> Динамическое подключение дополнительных обязательств к объекту. Шаблон Decorator предоставляет гибкую альтернативу практике определения подклассов с целью расширения функциональности.[1]

Шаблон Decorator функционирует посредством создания цепочки объектов, которая начинается с объектов-декораторов, отвечающих за выполнение новых функций, и заканчивается исходным объектом (рис. 15.3).

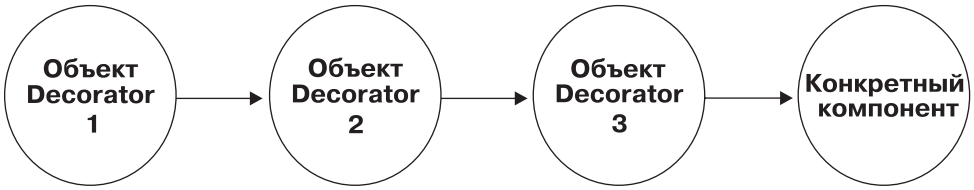

РИС. 15.3. ***Цепочка объектов в шаблоне Decorator***

На рис. 15.4 представлена диаграмма классов шаблона Decorator, соответствующая цепочке объектов, показанных на рис. 15.3. Каждая цепочка начинается с объекта класса ***`Component`*** (это может быть **`ConcreteComponent`** или **`Decorator`**). За каждым объектом класса **`Decorator`** следует либо другой объект **`Decorator`**, либо исходный объект класса **`ConcreteComponent`**. Заканчивается цепочка всегда объектом класса **`ConcreteComponent`**.

---

[1] *Gamma E., Helm R., Johnson R., Vlissides J.* Design Patterns: Elements of Reusable Object-Oriented Software, *Reading, MA: Addison-Wesley, 1995, с. 315.*

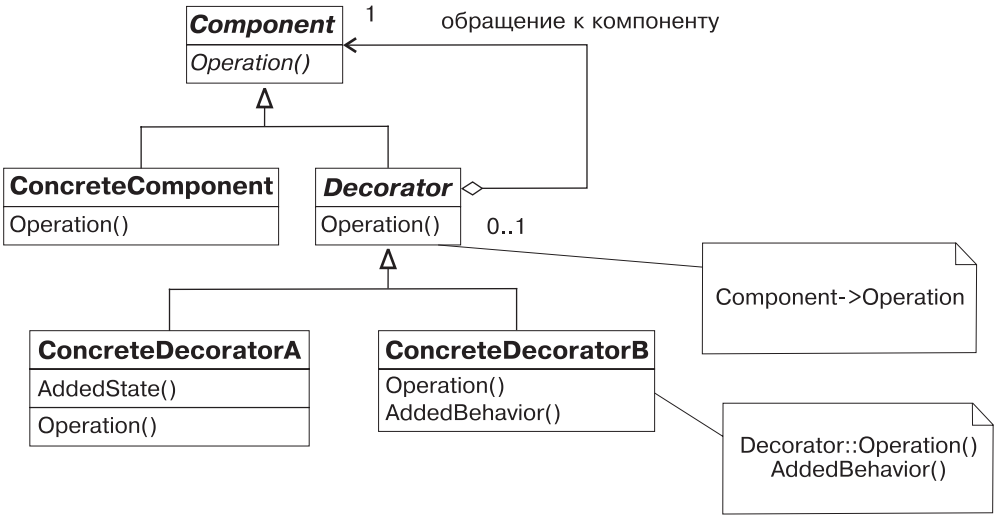

РИС. 15.4. *Диаграмма классов шаблона Decorator*

Например, на рис. 15.4 объект класса **`ConcreteDecoratorB`** выполняет собственный метод `Operation()`, а затем вызывает метод `Operation()` объекта класса **`Decorator`**. Этот вызов объекта **`ConcreteDecoratorB`** завершает выполнение метода `Operation()` класса ***`Component`***.

Применение шаблона Decorator к нашему примеру

В нашем учебном примере классом **`ConcreteComponent`** является класс **`SalesTicket`**, а конкретными представителями класса-декоратора служат заголовки и нижние колонтитулы. На рис. 15.5 показано, как шаблон Decorator может быть использован в проектируемой системе.

Диаграмма объектов на рис. 15.6 демонстрирует применение шаблона Decorator в случае печати документа с одним заголовком и одним нижним колонтитулом.

Каждый объект-декоратор как бы "упаковывает" исходный объект в новую функцию. Отдельный объект-декоратор выполняет свою добавленную функциональность либо перед выполнением функции-декоратора (заголовок), либо после ее выполнения (нижний колонтитул). Проще всего понять, как все это работает, если обратиться к программному коду с конкретным примером и внимательно проанализировать его (листинг 15.1).

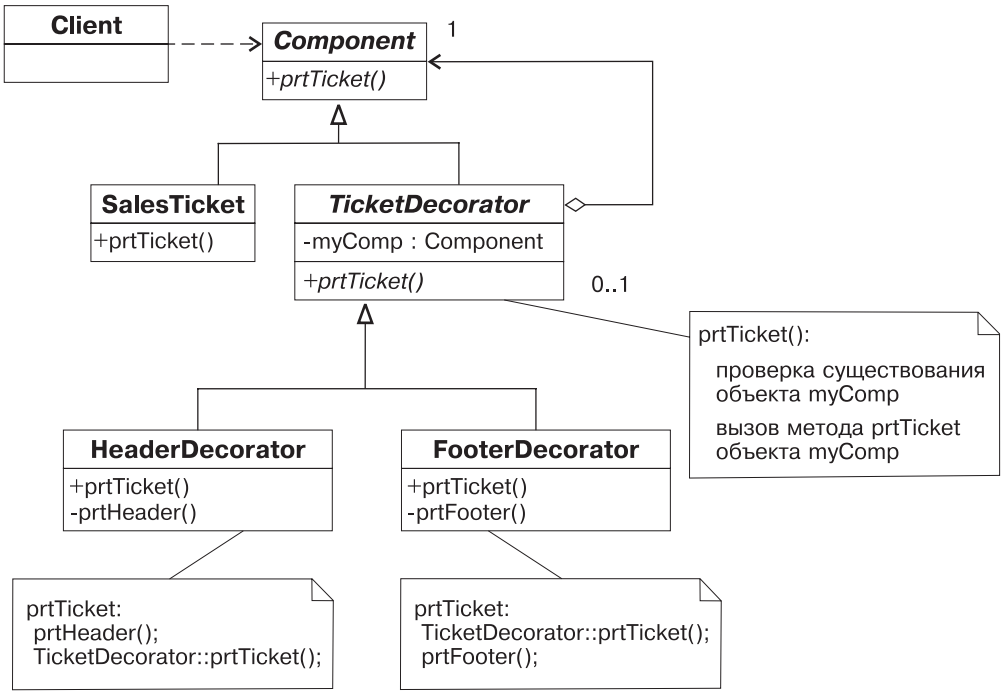

РИС. 15.5. *Формирование документа с заголовком и нижним колонтитулом*

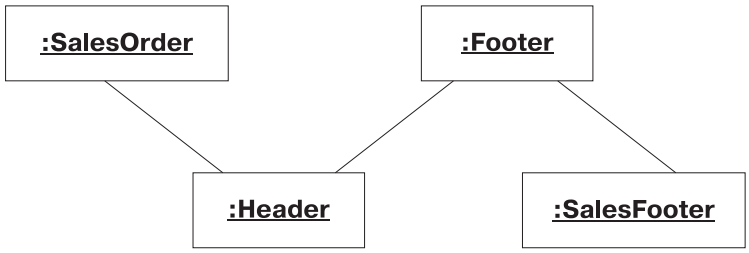

РИС. 15.6. *Пример диаграммы объектов шаблона Decorator*

Листинг 15.1. Реализация шаблона Decorator на языке Java

```
class SalesTicket extends Component {
  public void prtTicket () {
    // Здесь размещается код, выполняющий печать накладной
  }

abstract class Decorator extends Component {
  private Component myComp;
  public Decorator (Component myC) {
    myComp = myC;
  }
  public void prtTicket () {
```

```
    if (myComp != null)
      myComp.prtTicket();
  }
}

class Header1 extends Decorator {
  public void prtTicket () {
    // Здесь размещается код, выполняющий печать Заголовка 1
    super.prtTicket();
  }
}

class Header2 extends Decorator {
  public void prtTicket () {
    // Здесь размещается код, выполняющий печать Заголовка 2
    super.prtTicket();
  }
}

class Footer1 extends Decorator {
  public void prtReport () {
    super.prtTicket();
    // Здесь размещается код, выполняющий печать Колонтитула 1
  }
}

class Footer2 extends Decorator {
  public void prtReport () {
    super.prtTicket();
    // Здесь размещается код, выполняющий печать Колонтитула 2
  }
}

class SalesOrder {
  void prtTicket () {
    Component myST;
    // Получение цепочки, состоящей из нескольких объектов-декораторов
    // и объекта SalesTicket, созданной другим объектом, знающим,
    // как это делается. При необходимости это может выполняться в
    // конструкторе, чтобы избежать выполнения при каждом вызове.
    myST = Configuration.getSalesTicket()
    // Печать накладной с требуемыми заголовками и колонтитулами
    myST.prtTicket();
  }
}
```

Предположим, что накладная должна иметь следующий вид:

Заголовок 1

Текст накладной

Колонтитул 1

Тогда метод `Configuration.getSalesTicket` возвратит следующее:

```
return(new Header1(new Footer1(new SalesTicket())));
```

Это означает, что сначала создается объект **`SalesTicket`**, затем объект **`Footer1`** и, наконец, объект **`Header1`**.

Теперь предположим, что накладная должна иметь такой вид:

Заголовок 1

Заголовок 2

Текст накладной

Колонтитул 1

В этом случае метод `Configuration.getSalesTicket` должен возвращать следующее:

```
return(new Header1(new Header2 (new Footer1(new SalesTicket()))));
```

В результате, первым создается объект **`SalesTicket`**, затем объекты **`Footer1`** и **`Header2`** и, наконец, объект **`Header1`**.

Таким образом, шаблон Decorator позволяет разделить проблему на две подзадачи.

- Каким образом реализовать объекты, включающие новые функциональные возможности.
- Как организовать объекты для выполнения их функциональности в каждом конкретном случае.

Подобный подход позволяет отделить реализацию объектов-декораторов от объекта, определяющего способ их использования. В результате связность в системе возрастает, поскольку каждый из объектов-декораторов отвечает только за выполнение той функции, которую он добавляет, и может не заботиться о том, каким образом он сам будет включен в цепочку объектов.

Еще один пример: организация ввода-вывода информации

Очень часто шаблон Decorator используется для организации потокового ввода-вывода данных. Прежде чем обсуждать возможности использования данного шаблона для указанной цели, целесообразно вспомнить некоторые особенности потокового ввода-вывода данных. Мы ограничимся рассмотрением операции ввода, поскольку вывод данных организуется аналогично (при ясном понимании процесса ввода не составит труда разобраться с тем, как он работает в обратном направлении). Любой заданный входной поток имеет строго один источник, но может предусматривать произвольное количество действий (включая нуль), которые должны быть выполнены в этом входном потоке. Например, чтение данных может выполняться из следующих источников:

- файла;
- сетевого соединения с последующей расшифровкой поступающего потока;
- файла с последующей разархивацией поступающих данных;
- строковой переменной;
- файла с последующей разархивацией и расшифровкой данных.

В зависимости от того, какие данные были посланы (или сохранены), возможна любая комбинация действий. Перефразируем только что сказанное: любой источник данных может быть "декорирован" произвольной комбинацией действий. Некоторые

из возможных типов источников данных и действий по их обработке для входящего потока представлены в табл. 15.1.

Таблица 15.1. Источники данных и действия по их обработке для входного потока

Источник	Обработка
Строковая переменная	Буферированный ввод
Файл	Проверка контрольной суммы
Сетевое соединение (TCP/IP)	Разархивирование
Последовательный порт	Расшифровка данных (любым способом)
Параллельный порт	Избирательные фильтры (любым способом)
Клавиатура	

Разработчики, использующие объектно-ориентированные языки программирования, могут воспользоваться тем преимуществом, что объекты источников и реализации операций (действий) могут быть порождены от общего абстрактного класса. Каждому объекту реализации операции можно назначить источник или предшествующий объект операции непосредственно в его конструкторе. Цепочка действий в этом случае строится в процессе создания экземпляров входящих в нее объектов (каждому объекту передается ссылка на последующий объект). Классы объектов источников являются производными от абстрактного класса `ConcreteComponent` (см. рис. 15.4), в то время как классы объектов операций играют роль объектов-декораторов. Обратите внимание на то, что класс `ConcreteComponent` в данном случае используется иначе, так как здесь он является абстрактным.

Например, чтобы реализовать поведение "читать данные из файла, разархивировать, а затем расшифровать их", необходимо выполнить следующее.

1. Построить цепочку из объектов-декораторов, выполнив следующие действия:
   а) создать экземпляр объекта чтения данных из файла;
   б) передать ссылку на него в конструктор объекта, выполняющего разархивацию данных;
   в) передать ссылку на объект разархивации в конструктор объекта, выполняющего расшифровку данных.
2. Считать, разархивировать и расшифровать данные. Все требуемые операции выполняются для объекта-клиента совершенно прозрачно. Объект-клиент просто знает, что существует некий объект, выполняющий требуемую обработку входного потока данных, с которым он и взаимодействует.

Если объекту-клиенту потребуется считать входной поток данных из другого источника, строится новая цепочка — посредством создания экземпляра объекта требуемого источника с подключением таких же объектов реализации необходимых операций.

Как разобраться с множеством потоков языка Java

Язык Java печально известен запутанным множеством разнообразных входных потоков и связанных с ними классов. Понять назначение этих классов гораздо проще в контексте шаблона Decorator. Все классы, которые являются производными непосредственно от класса `java.io.InputStream` (`ByteArrayInputStream`, `FileInputStream`, `FilterInputStream`, `InputStream`, `ObjectInputStream`, `SequenceInputStream` и `StringBufferInputStream`), выполняют роль декорируемых объектов. Все классы-декораторы являются производными от класса `FilterInputStream` (как напрямую, так и косвенно).

Обращение к шаблону Decorator позволяет понять, почему язык Java требует объединения подобных объектов в цепочку при создании их экземпляров. Такой подход позволяет программистам создавать любое количество комбинаций из различных возможных функций обработки.

Дополнительные замечания о шаблоне Decorator

Чтобы воспользоваться всей мощью шаблона Decorator, необходимо полностью отделить создание цепочки объектов от объекта-клиента, использующего ее. Для достижения этой цели чаще всего применяются объекты-фабрики классов, создающие цепочку экземпляров объектов на основании некоторой переданной им информации о требуемой конфигурации.

Основные характеристики шаблона Decorator

Назначение	Динамическое подключение к объекту дополнительных обязательств
Задача	Объект, который предполагается использовать, выполняет основные функции. Однако может потребоваться добавить к нему некоторую дополнительную функциональность, которая будет выполняться до или после основной функциональности объекта. *Замечание.* Базовые классы в языке Java широко используют шаблон Decorator для организации обработки операций ввода-вывода
Способ решения	Позволяет расширить функциональность объекта без определения соответствующих подклассов
Участники	Класс `ConcreteComponent` – это тот класс, в который с помощью шаблона Decorator добавляется новая функциональность. В некоторых случаях базовая функциональность предоставляется классами, производными от класса `ConcreteComponent`. В подобных случаях класс *`ConcreteComponent`* является уже не конкретным, а абстрактным. Абстрактный класс *`Component`* определяет интерфейс для использования всех этих классов
Следствия	Добавляемая функциональность реализуется в небольших объектах. Преимущество состоит в возможности динамически добавлять эту функциональность до или после основной функциональности объекта `ConcreteComponent`. *Замечание.* Хотя объект-декоратор может добавлять свою функциональность до или после функциональности исходного объекта, цепочка создаваемых объектов всегда должна заканчиваться объектом класса `ConcreteComponent`
Реализация	Создается абстрактный класс, представляющий как исходный класс, так и новые, добавляемые в класс функции. В классах-декораторах новые функции вызываются в требуемой последовательности – до или после вызова последующего объекта

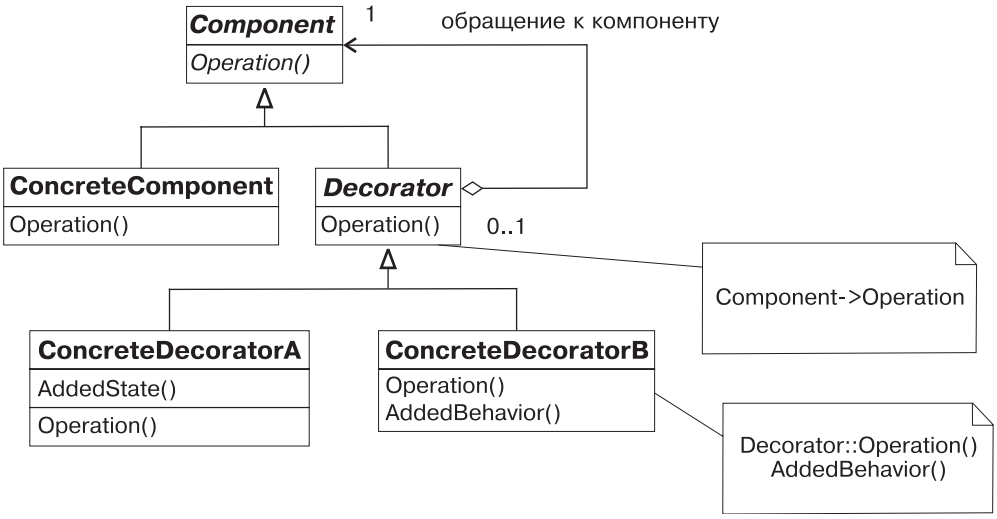

РИС. 15.7. ***Стандартное упрощенное представление шаблона Decorator***

Я использовал шаблон Decorator для того, чтобы "упаковывать" тестируемый объект в классы проверки пред- и постусловий. Этот подход дал очень хорошие результаты. В процессе тестирования первый объект в цепочке выполняет проверку некоторого набора предварительных условий, а затем обращается к следующему объекту в цепочке. Сразу после вызова тестируемого объекта аналогичный объект выполняет проверку любого набора постусловий. Если при каждом прогоне предусматривается выполнять различные тесты, следует описать каждый тест в разных объектах-декораторах, а затем объединить их в цепочку в соответствии с тем набором тестов, который планируется выполнить.

Резюме

Шаблон Decorator предлагает эффективный способ динамического добавления дополнительной функциональности к уже существующей. Для этой цели он предусматривает построение цепочки объектов, которая в совокупности реализует требуемое поведение. Первый объект этой цепочки вызывается объектом-клиентом, который ничего не знает о процедуре создания вызываемой им цепочки. За счет отделения процедуры создания цепочки от процедуры ее использования удается оградить объект-клиент от каких-либо изменений в связи с появлением новых требований по расширению функциональности.

Приложение. Примеры программного кода на языке C++

Листинг 15.2. Фрагмент кода. Реализация шаблона Decorator

```
class SalesTicket : public Component {
  public:
    void prtTicket();
}
SalesTicket::prtTicket() {
  // Здесь размещается код, выполняющий печать накладной
}

class Decorator : public Component {
  public:
    virtual void prtTicket();
    Decorator( Component *myC);
  private:
    Component *myComp;
}
Decorator::Decorator( Component *myC) {
  myComp= myC;
}
Decorator::prtTicket() {
  if (myComp != 0)
    myComp -> prtTicket();
}

class Header1 : public Decorator {
  public:
    void prtTicket();
}
Header1::prtTicket () {
  // Здесь размещается код, выполняющий печать Заголовка 1
  Decorator::prtTicket();
}

class Header2 : public Decorator {
  public:
    void prtTicket();
}
Header2::prtTicket () {
  // Здесь размещается код, выполняющий печать Заголовка 2
  Decorator::prtTicket();
}

class Footer1 : public Decorator {
  public:
    void prtTicket();
}
Footer1::prtTicket () {
  Decorator::prtTicket();
  // Здесь размещается код, выполняющий печать Колонтитула 1
}

class Footer2 : public Decorator {
  public:
    void prtTicket();
}
Footer2::prtTicket () {
  Decorator::prtTicket();
```

```
  // Здесь размещается код, выполняющий печать Колонтитула 2
}
SalesOrder::prtTicket () {
  Component *myST;
  // Получение цепочки, состоящей из нескольких объектов-декораторов
  // и объекта SalesTicket, созданной другим объектом, знающим,
  // как это делается. При необходимости это может выполняться в
  // конструкторе, чтобы избежать выполнения при каждом вызове.
  myST = Configuration.getSalesTicket()

  // Печать накладной с требуемыми заголовками и колонтитулами
  myST -> prtTicket();
}
```

Глава 16

Шаблоны Singleton и Double-Checked Locking

Введение

В этой главе мы продолжим обсуждение нашего учебного примера — приложения поддержки электронной розничной торговли, начатое в главах 14, *Шаблон Strategy*, и 15, *Шаблон Decorator*.

Здесь мы выполним следующее.

- Познакомимся с шаблоном Singleton.0
- Рассмотрим ключевые особенности шаблона Singleton.
- Обсудим один из вариантов шаблона Singleton, называемый шаблоном Double-Checked Locking.
- Познакомимся с моим опытом применения шаблона Singleton на практике.

Шаблоны Singleton и Double-Checked Locking очень просты и широко известны. Оба эти шаблона используются для получения гарантий, что будет создан только один объект определенного класса. Различие между этими двумя шаблонами состоит в том, что шаблон Singleton используется в однопоточных приложениях, а шаблон Double-Checked Locking предназначен для использования в многопоточной среде.[1]

Назначение шаблона проектирования Singleton

Согласно определению "банды четырех" назначение шаблона Singleton (одиночка) состоит в следующем.

Получение гарантий, что будет создан только один экземпляр объекта данного класса, и предоставление глобальной точки доступа к нему.[2]

Шаблон Singleton выполняет свои функции, определяя специальный метод, предназначенный для создания экземпляра требуемого объекта.

---

[1] *Если вам неизвестно, что такое многопоточное приложение, не волнуйтесь — сейчас достаточно будет сконцентрировать все внимание на шаблоне Singleton.*

[2] *Gamma E., Helm R., Johnson R., Vlissides J.* Design Patterns: Elements of Reusable Object-Oriented Software, *Reading, MA: Addison-Wesley, 1995, с. 127.*

- Когда этот метод вызывается, он проверяет, был ли экземпляр требуемого объекта создан ранее. Если это так, то метод просто возвращает ссылку на этот объект. Если же объект еще не существует, метод создает требуемый экземпляр и возвращает ссылку на вновь созданный объект.
- Для получения гарантий, что существует только один способ создания экземпляров объектов данного класса, тип конструктора этого класса следует определить как защищенный или закрытый.

Применение шаблона Singleton к нашему примеру

В главе 14 правила начисления налогов были инкапсулированы в объекты шаблона Strategy. Для этой цели были определены конкретные классы, производные от абстрактного класса ***CalcTax***, — по одному для каждого возможного варианта начисления налога. А это означает, что одни и те же экземпляры объектов будут многократно использоваться, раз за разом вызываясь различными объектами-клиентами.

Из соображений производительности системы нежелательно при каждом обращении заново создавать требуемый экземпляр объекта и уничтожать его после завершения работы с ним. В то же время малоэффективным будет и решение создать все возможные типы объектов начисления налогов непосредственно при запуске системы, особенно если количество вариантов налогообложения достаточно велико. (Не забывайте, что приложение может содержать большое количество и других стратегий.) Оптимальным решением будет создавать экземпляр объекта каждого типа только тогда, когда это действительно необходимо, и при условии, что может быть создан только один такой экземпляр.

Проблема состоит в том, что желательно не создавать дополнительный объект, который будет хранить информацию о тех объектах, экземпляры которых уже были созданы. Гораздо лучше сделать сами объекты (представляющие отдельные стратегии) ответственными за соблюдение правила создания только одного их экземпляра.

В этом и состоит назначение шаблона Singleton. Он позволяет создать экземпляр объекта указанного класса только один раз, не запрашивая у объекта-клиента сведений о том, существует уже требуемый экземпляр объекта или нет.

Пример реализации шаблона Singleton приведен в листинге 16.1. В этом примере определяется метод (`getInstance`), который будет создавать только один экземпляр объекта класса **`USTax`**. Шаблон Singleton предохраняет систему от того, чтобы кто-либо еще мог создать экземпляр объекта класса **`USTax`**, просто объявляя его конструктор закрытым, а это означает, что никакой другой объект не сможет получить доступ к этому конструктору.

Листинг 16.1. Реализация шаблона Singleton на языке Java

```
class USTax {
  private static USTax instance;
  private USTax():
  public static USTax getInstance() {
    if (instance == null)
      instance = new USTax();
    return instance;
  }
}
```

Основные характеристики шаблона Singleton

Назначение	Необходимо иметь в системе только *один* экземпляр объекта заданного класса, не используя никаких глобальных объектов, предназначенных для контроля над созданием этих экземпляров
Задача	Нескольким различным объектам-клиентам требуется обращаться к одному и тому же объекту и иметь гарантии, что объектов этого типа в системе будет не более одного
Способ решения	Гарантированное однократное выполнение метода-конструктора
Участники	Объекты-клиенты получают доступ к методу `getInstance()` класса **`Singleton`** исключительно через его метод `instance()`
Следствия	Объекты-клиенты могут не интересоваться тем, существует ли уже экземпляр объекта класса **`Singleton`**. Контроль над созданием своих экземпляров возложен на сам класс **`Singleton`**
Реализация	• В класс добавляется закрытая статическая переменная-член класса, содержащая ссылку на соответствующий объект (ее начальное значение `NULL`) • В класс добавляется открытый статический метод, который создает экземпляр объекта данного класса, если значение упомянутой выше переменной-члена класса равно `NULL` (и помещает в нее указатель на созданный объект). Метод возвращает значение этого экземпляра класса • Статус конструктора класса устанавливается защищенным или закрытым, так что никто не сможет самостоятельно создать экземпляр объекта данного класса в обход его механизма статического конструктора

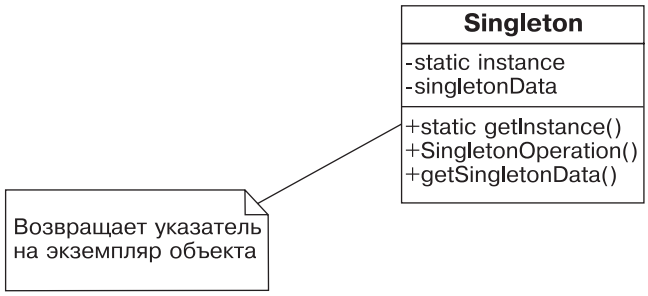

РИС. 16.1. *Стандартный упрощенный вид шаблона Singleton*

Возможный вариант — шаблон Double-Checked Locking

Этот шаблон применяется *только* в случае создания многопоточных приложений. Читатель, не интересующийся этой темой, может безболезненно пропустить этот раздел. Изложение материала предполагает, что читатель имеет базовое представление об особенностях многопоточных приложений, в том числе о синхронизации.

В многопоточной среде применение шаблона Singleton связано с определенными проблемами.

Предположим, что два вызова метода **getInstance** сделаны точно в одно и то же время. Это может закончиться очень плохо. Рассмотрим подробно то, что может произойти в подобной ситуации.

1. Первый поток проверяет, был ли уже создан требуемый экземпляр объекта. Если нет, то управление передается той части программного кода, которая отвечает за создание экземпляров объектов этого класса.
2. Однако прежде чем создание первого экземпляра объекта будет завершено, второй поток также выполняет проверку значения статического указателя на равенство значению NULL. Поскольку первый поток еще не закончил создание экземпляра объекта, значение статической переменной по-прежнему будет равно NULL. Поэтому второй поток также передаст управление той части программного кода, которая отвечает за создание экземпляров объектов этого класса.
3. В результате оба потока создадут по одному экземпляру объекта класса **Singleton**, и в системе окажется два подобных объекта.

Является ли это проблемой? Может быть, да, а может быть, и нет.

- Если класс **Singleton** не хранит сведений о своем состоянии, эта ситуация может не вызывать в системе каких-либо проблем.
- В языке Java проблема будет состоять лишь в том, что несколько лишних байт памяти используется бесполезно.
- В языке C++ эта ситуация может вызвать утечку памяти, поскольку при завершении программа будет удалять только один объект класса **Singleton**, тогда как создано их было два.
- Если объект класса **Singleton** хранит какие-либо сведения о своем состоянии, могут иметь место очень коварные и трудноуловимые ошибки. Например:
  - если объект устанавливает соединение, фактически будет установлено два соединения (по одному для каждого из созданных объектов);
  - если в объекте используется счетчик, в системе будут существовать два разных счетчика.

Обнаружить подобные ошибки очень трудно. Во-первых, повторное создание объектов маловероятно и обычно не имеет места. Во-вторых, может быть совершенно непонятно, что же происходит со счетчиками, когда одни объекты-клиенты будут обращаться к одному объекту класса **Singleton**, а остальные — к другому.

На первый взгляд может показаться, что все, что необходимо сделать, — это синхронизировать выполнение проверки, был ли уже создан объект класса **Singleton**. Единственная проблема заключается в том, что подобная синхронизация может вызвать появление в системе узкого места, резко замедляющего работу программы, поскольку все потоки вынуждены будут ждать в общей очереди права на проверку существования объекта класса **Singleton**.

Можно предположить, что достаточно будет поместить код синхронизации в ветвь *после* успешного выполнения проверки `if (instance == null)`. Однако это решение не будет работать, поскольку синхронизация потоков уже после того, как оба они убедились в наличии значения NULL в статической переменной, опять-таки приведет

к созданию двух объектов класса **Singleton**, но создаваться они будут уже строго поочередно.

Решение данной проблемы состоит в том, чтобы выполнить синхронизацию после проверки статической переменной на NULL, а затем осуществить повторную проверку, чтобы удостовериться, что экземпляр объекта данного класса все еще не создан. Пример соответствующего кода приведен в листинге 16.2. Данный метод получил название *double-checked locking* (блокировка с двойной проверкой)[3]. Идея состоит в том, чтобы избежать ненужной блокировки. В данном случае синхронизация выполняется после первой проверки, так что выполнение ее не приведет к появлению в системе узкого места.

Основные достоинства метода блокировки с двойной проверкой состоят в следующем.

- Удается избежать излишней блокировки процессов за счет помещения запроса на создание нового экземпляра объекта в процедуру проверки другого условия.
- Обеспечивается поддержка многопоточной среды.

Листинг 16.2. Фрагмент реализации шаблона Double-Checked Locking на языке Java

```
class USTax extends CalcTax {
  private USTax instance;
  private USTax () {
  }
  private synchronized static
    void doSync () {
    // Это только для синхронизации
  }
  public USTax getInstance() {
    if (instance == null) {
      USTax.doSync();
      if (instance == null)
        instance= new USTax();
    }
  return instance;
  }
}
```

Дополнительные замечания о шаблонах Singleton и Double-Checked Locking

Если известно, что объект обязательно потребуется, и соображения повышения производительности не заставляют отложить создание экземпляра объекта до тех пор, пока он действительно понадобится, проще всего определить статическую переменную, содержащую ссылку на этот объект.

В многопоточных приложениях класс **Singleton**, как правило, должен быть определен с учетом проблем безопасности работы отдельных потоков (поскольку один и тот же объект может быть доступен многим потокам). Это означает, что класс не дол-

[3] *Martin R., Riehle D., Buschmann F.* Pattern Language of Program Design *Reading, MA: Addison-Wesley, 1998, c. 363.*

жен включать каких-либо данных-членов класса и использовать только такие переменные, область видимости которых не выходит за рамки метода.

Резюме

Шаблоны Singleton и Double-Checked Locking обычно используются в тех случаях, когда необходимо иметь гарантии, что в системе будет создан только один экземпляр объекта заданного класса. Шаблон Singleton используется в однопоточных приложениях, а шаблон Double-Checked Locking предназначен для использования в многопоточном окружении.

Приложение. Примеры программного кода на языке C++

Листинг 16.3. Реализация шаблона Singleton

```
Class USTax {
  public:
    static USTax* getInstance();
  private:
    USTax();
    static USTax* instance;
}
USTax::USTax () {
  instance= 0;
}
USTax* USTax::getInstance () {
  if (instance== 0) {
    instance= new USTax;
  }
return instance;
}
```

Листинг 16.4. Фрагмент кода реализации шаблона Double-Checked Locking

```
class USTax : public CalcTax {
  public:
    static USTax* getInstance();
  private:
    USTax();
    static USTax* instance;
};
USTax::USTax () {
  instance= 0;
}
USTax* USTax::getInstance () {
  if (instance== 0) {
    // здесь выполняется синхронизация
    if (instance== 0) {
    }
  }
return instance;
}
```

Глава 17

Шаблон Observer

Введение

В этой главе мы продолжим обсуждение нашего учебного примера — приложения поддержки электронной розничной торговли, начатое в главах 14–16.

Здесь мы выполним следующее.

- Рассмотрим схему классификации шаблонов.
- Познакомимся с шаблоном Observer в процессе обсуждения дополнительных требований, предъявленных к нашему учебному примеру.
- Применим данный шаблон в системе поддержки электронной торговли.
- Обсудим характеристики этого шаблона.
- Рассмотрим ключевые особенности шаблона Observer.
- Познакомимся с моим опытом применения шаблона Observer на практике.

Категории шаблонов

Количество уже существующих шаблонов достаточно велико. Поэтому для их упорядочения "банда четырех" предложила разделить все шаблоны на три категории, как показано в табл. 17.1.[1]

Таблица 17.1. Категории шаблонов

Категория	Назначение	Примеры в этой книге	Для чего используются
Структурные	Связывают между собой существующие объекты	• Facade (глава 6) • Adapter (глава 7)	Приведение интерфейсов
		• Bridge (глава 9) • Decorator (глава 15)	Связывание реализации с абстракцией
Поведенческие	Определяют способы проявления гибкого (изменяющегося) поведения	• Strategy (глава 14)	Сокрытие вариаций

[1] *Gamma, E., Helm R., Johnson R., Vlissides J.* Design Patterns: Elements of Reusable Object-Oriented Software, *Reading, MA: Addison-Wesley, 1995, c. 10*

Окончание таблицы

Категория	Назначение	Примеры в этой книге	Для чего используются
Создающие	Управляют созданием экземпляров объектов	• Abstract Factory (глава 10) • Singleton (глава 16) • Double-Checked Locking (глава 16) • Factory Method (глава 19)	Создание экземпляров объектов

Впервые приступив к изучению шаблонов проектирования, я очень удивился тому, что шаблоны Bridge и Decorator были отнесены к категории структурных, а не поведенческих шаблонов. На первый взгляд, не вызывало сомнения, что они используются для реализации различных типов поведения. Но, как выяснилось впоследствии, я просто неправильно понял систему классификации "банды четырех". Структурные шаблоны предназначены для связывания между собой уже существующих функций. Работая с шаблоном Bridge, мы, как правило, начинаем с функций абстракции и реализации, а затем связываем их между собой. Шаблон Decorator имеет дело с уже существующим функциональным классом и предназначен для "декорирования" его дополнительными функциями.

Я пришел к заключению, что имеет смысл выделить еще одну, четвертую, категорию шаблонов. К ней я отношу шаблоны, основное назначение которых — уменьшить связанность объектов друг с другом, и, следовательно, повысить возможности масштабируемости системы и ее гибкость. Я назвал эту категорию *развязывающими шаблонами*. Поскольку по классификации "банды четырех" большинство подобных шаблонов принадлежат к категории поведенческих, их можно было бы считать подмножеством этой категории. Я решил выделить их в четвертую категорию только потому, что намеревался в этой книге отразить мой собственный взгляд на шаблоны проектирования, сосредоточив внимание на мотивах их определения — в данном случае это развязывание объектов в системе.

Я не буду более задерживаться на обсуждении различных аспектов классификации шаблонов, поскольку это требует достаточно глубокого понимания принципов их построения и работы.

В этой главе мы познакомимся с шаблоном Observer, который можно считать лучшим примером развязывающего шаблона среди всех прочих. По классификации "банды четырех" шаблон Observer относится к группе поведенческих.

Предъявление новых требований к системе электронной торговли

Предположим, что при разработке нашего приложения поддержки электронной торговли было выдвинуто очередное новое требование. Выяснилось, что необходимо выполнять перечисленные ниже действия всякий раз, когда к системе подключается новый клиент.

- Послать клиенту вежливое приглашение по электронной почте.
- Проверить адрес клиента по списку почтовых отделений.

Все ли требования теперь учтены? Возможны ли какие-либо изменения в будущем?

Если существует обоснованная уверенность в том, что теперь все требования к системе уже известны, можно решить новую проблему, просто внедрив программный код организации отправки приглашения и проверки адреса в класс **Customer**, как показано на рис. 17.1.

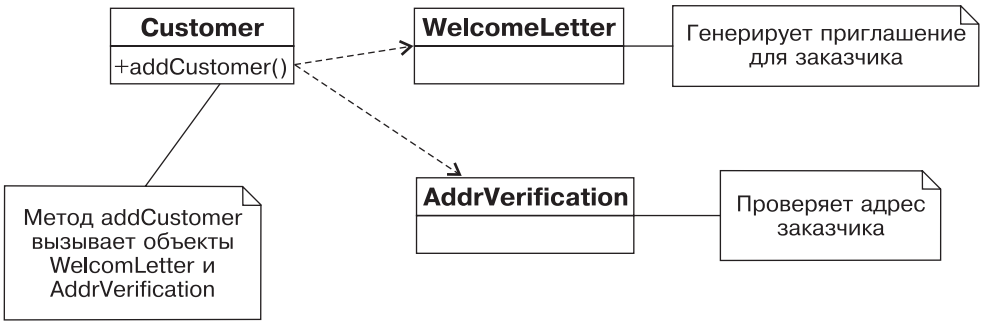

РИС. 17.1. *Жесткое кодирование поведения объекта*

Например, можно расширить метод класса **Customer**, который вносит сведения о новом клиенте в базу данных, добавив в него выполнение запросов к объектам, выполняющим создание и отправку пригласительного письма и проверку адреса по списку почтовых отделений.

В этом случае обязательства будут распределены между классами так, как показано в табл. 17.2.

Таблица 17.2. Исходное распределение обязательств между классами

Класс	Обязательства
Customer	При подключении к системе нового клиента этот объект обращается к другим объектам с требованием выполнить соответствующие действия
WelcomeLetter	Генерирует приветственное письмо клиентам и сообщает им, что сведения о них помещены в систему
AddrVerification	Проверяет адрес любого клиента, который будет ему указан

Метод жесткого кодирования поведения прекрасно работает, но только первое время. К сожалению, требования к системе с течением времени *всегда* изменяются. Можно быть абсолютно уверенным в том, что появление новых требований не заставит себя долго ждать, а это потребует внесения изменений в поведение объектов класса **Customer**. Например, может потребоваться организовать отправку пригласительных писем различным компаниям, при этом придется создавать отдельный объект класса **Customer** для каждой компании. Очевидно, что должно существовать лучшее решение.

Шаблон Observer

Согласно определению "банды четырех" назначение шаблона Observer (наблюдатель) состоит в следующем.

> Определение зависимости типа "один ко многим" между объектами, выполненное таким образом, что при изменении состояния одного объекта все зависимые объекты были бы уведомлены об этом и обновлены автоматически.[2]

Весьма распространена ситуация, при которой существует набор объектов, которые должны получать уведомление всякий раз, когда происходит некоторое событие. Необходимо, чтобы это уведомление выполнялось автоматически. Кроме того, желательно обойтись без внесения изменений в оповещающий объект при каждом изменении в наборе объектов, получающих уведомления. (В противном случае ситуация походила бы на требование внесения изменений в радиопередатчик всякий раз, когда в городе появляется новый автомобиль с радиоприемником.) Таким образом, мы должны снизить уровень связанности между оповещающим и извещаемыми объектами.

Шаблон Observer относится к наиболее широко используемым шаблонам. Иначе его иногда называют *Dependents* (зависимости) или *Publish-Subscribe*[3] (публикация-подписка) Он является аналогом процесса извещения в технологии COM. В языке Java данный шаблон реализован с помощью интерфейса **`Observer`** и класса **`Observable`** (подробнее об этом речь пойдет ниже). В экспертных системах, использующих базы правил, шаблон Observer часто реализуется в демонах правил.

Применение шаблона Observer в системе электронной торговли

Наш подход к решению проблемы состоит в том, чтобы найти то, что может быть изменяемым в системе, а затем попытаться инкапсулировать эти вариации. В нашем учебном примере изменяемым может быть следующее.

- **Различные виды объектов.** Существует список объектов, которые необходимо уведомить об изменении состояния системы. Эти объекты, как правило, принадлежат к различным классам.
- **Различные интерфейсы.** Поскольку извещаемые объекты принадлежат к различным классам, они, вероятнее всего, будут обладать различными интерфейсами.

Прежде всего, выявим все объекты, которые должны получать уведомления. Назовем их *наблюдателями* (observer), так как они находятся в состоянии ожидания некоторого события, которое должно произойти.

Желательно, чтобы все эти объекты имели одинаковый интерфейс. Если интерфейсы не будут одинаковыми, придется модифицировать наблюдаемый *субъект* — т.е. тот объект, который инициирует ожидаемое событие (например, объект класса **`Customer`**), чтобы он мог обращаться к объектам-наблюдателям различного типа.

---

[2] *Gamma E., Helm R., Johnson R., Vlissides J.* Design Patterns: Elements of Reusable Object-Oriented Software, *Reading, MA: Addison-Wesley, 1995, с. 293.*

[3] *Там же, с. 293.*

Если все объекты-наблюдатели будут одного типа, то субъект легко сможет уведомить каждый из них. Чтобы достичь этой цели можно поступить следующим образом.

- В языке Java, вероятно, лучше всего использовать интерфейс (чтобы повысить гибкость или просто по необходимости).
- В языке C++ можно использовать одиночное или множественное наследование — исходя из конкретных обстоятельств.

Обычно требуется, чтобы объекты-наблюдатели знали, за кем они должны наблюдать, а субъект был бы освобожден от необходимости знать, какие именно объекты-наблюдатели от него зависят. Чтобы достичь этого, необходимо предоставить каждому из объектов-наблюдателей возможность зарегистрироваться у субъекта. Поскольку все объекты-наблюдатели имеют один и тот же тип, в класс-субъект необходимо добавить два метода.

- Метод `attach(`**`Observer`**`)`. Добавляет заданный объект класса ***`Observer`*** к списку объектов-наблюдателей данного субъекта.
- Метод `detach(`**`Observer`**`)`. Удаляет заданный объект класса ***`Observer`*** из списка объектов-наблюдателей данного субъекта.

Теперь, когда класс **`Subject`** может регистрировать объекты класса ***`Observer`***, очень просто известить всех зарегистрировавшихся объектов-наблюдателей о наступлении ожидаемого события. Для этого в каждом конкретном типе класса ***`Observer`*** реализуется метод `update` (обновить). В классе **`Subject`** реализуется метод `notify`, который просматривает список зарегистрировавшихся объектов-наблюдателей и вызывает метод `update` каждого из них. Метод `update` объектов-наблюдателей должен включать программный код обработки наступления ожидаемого события.

Однако просто уведомить каждый объект-наблюдатель о наступлении события будет недостаточно. Объекту-наблюдателю может понадобиться дополнительная информации о событии, а не просто извещение о том, что оно произошло. Поэтому в класс-субъект следует добавить еще один или несколько методов, позволяющих объектам-наблюдателям получить ту информацию, в которой они нуждаются. Это решение представлено на рис. 17.2.

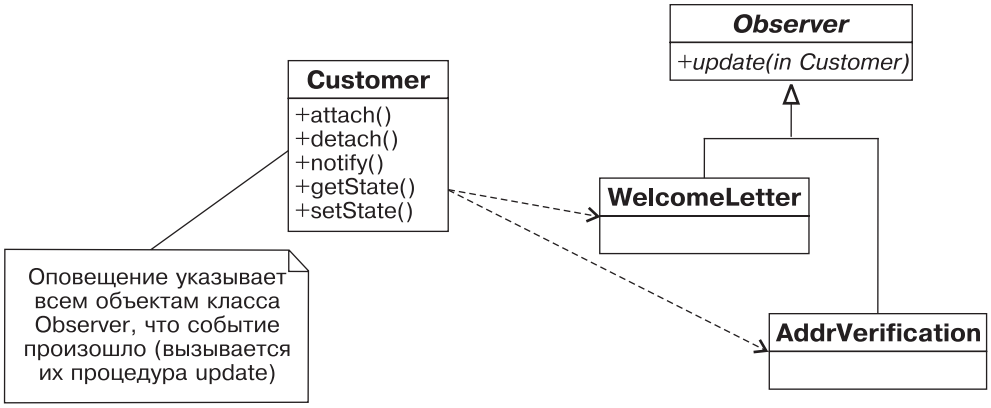

РИС. 17.2. *Реализация взаимодействия классов* ***`Customer`*** *и* `Observer`

На рис. 17.2 классы взаимодействуют между собой следующим образом.

1. Объекты класса ***Observer*** подключаются к классу **Customer** при их инициализации. Если объекту класса ***Observer*** необходима дополнительная информация от субъекта (объект класса **Customer**), он должен передать вызываемому объекту ссылку на свой метод `update`.
2. Каждый раз при добавлении нового объекта класса **Customer**, его метод `notify` вызывает все зарегистрированные объекты класса **Observer**.

Каждый объект класса ***Observer*** вызывает метод `getState` (класс **Customer**) для получения информации о вновь добавленном клиенте — это необходимо ему, чтобы выяснить, какие действия следует предпринять. *Замечание.* Обычно для получения необходимой информации требуется несколько методов.

Обратите внимание на то, что в нашем примере для добавления и удаления оповещаемых объектов в списке используются *статические* методы. Причина в том, что объекты-наблюдатели должны быть уведомлены обо *всех* новых клиентах (т.е. создаваемых объектах класса **Customer**). Вместе с уведомлением наблюдателю передается ссылка на вновь созданный объект описания клиента.

Код реализации описанного выше подхода представлен в листинге 17.1 (частично).

Листинг 17.1. Реализация шаблона Observer на языке Java

```
class Customer {
  static private Vector myObs;
  static {
    myObs= new Vector();
  }
  static void attach(Observer o){
    myObs.addElement(o);
  }
  static void detach(Observer o){
    myObs.remove(o);
  }
  public String getState () {
    // Здесь могут вызываться другие методы,
    // предоставляющие требуемую информацию
  }

  public void notifyObs () {
    for (Enumeration e =
      myObs.elements();
      e.hasMoreElements() ;) {
        ((Observer) e).update(this);
    }
  }
}

abstract class Observer {
  public Observer () {
    Customer.attach( this);
  }
  abstract public void
    update(Customer myCust);
}

class POVerification extends Observer {
  public AddrVerification () {
```

```
    super();
  }
  public void update (
    Customer myCust) {
      // Здесь выполняется проверка адресов.
      // Дополнительная информация о клиенте может быть
      // получена с помощью метода myCust
  }
}

class WelcomeLetter extends Observer {
  public WelcomeLetter () {
    super();
  }
  public void update (Customer myCust) {
    // Здесь выполняется отправка приветственного письма.
    // Дополнительная информация о клиенте может быть
    // получена с помощью метода myCust
  }
}
```

Данный подход позволяет добавлять новые объекты-наблюдатели без какого-либо влияния на любые уже существующие классы. Он также позволяет поддерживать в системе низкий уровень связанности. Организованная подобным образом система работает так, как если бы все объекты в ней отвечали только сами за себя.

Что произойдет в случае предъявления к системе нового требования? Предположим, возникла необходимость посылать письмо с купонами тем клиентам, которые находятся в пределах 20 миль от одного из складов компании.

Чтобы реализовать это требование, достаточно просто определить новый класс-наблюдатель, выполняющий отправку купонов. Он должен отправлять купоны только для тех новых клиентов, которые расположены не далее указанного расстояния от ближайшего из складов компании. Назовем этот класс **`BrickAndMortar`** и определим его как один из объектов-наблюдателей класса **`Customer`**. Данное решение представлено на рис. 17.3.

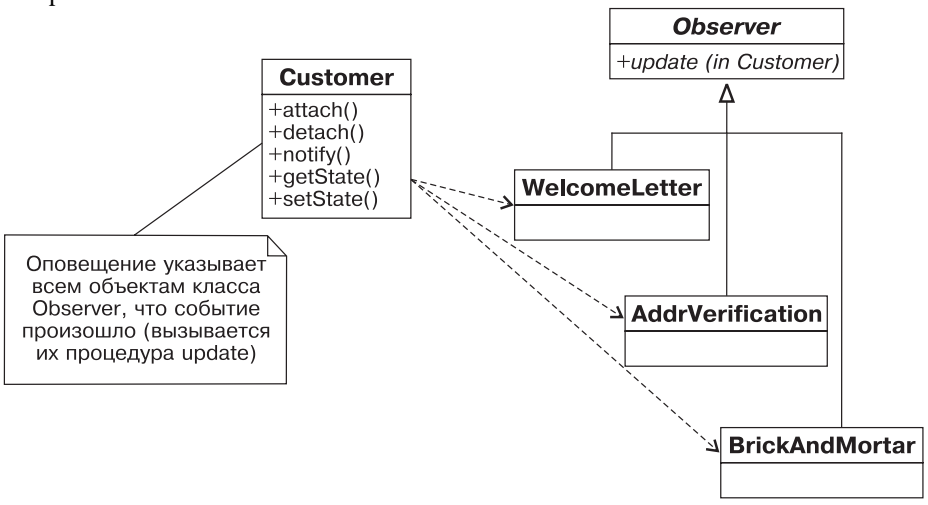

РИС. 17.3. *Добавление нового объекта-наблюдателя* ***`BrickAndMortar`***

Иногда класс, который должен стать новым наблюдателем, может уже существовать в системе. В этом случае, вероятно, будет предпочтительнее не вносить в него никаких изменений. Существует простой путь решить эту проблему: достаточно адаптировать существующий класс с помощью уже знакомого нам шаблона Adapter. Пример подобного решения приведен на рис. 17.4.

Класс `Observable` — замечание для разработчиков, использующих язык Java

Шаблон Observer настолько полезен, что один из пакетов языка Java включает его встроенную реализацию. В данном случае шаблон составлен из класса **`Observable`** и интерфейса **`Observer`**. Класс **`Observable`** выполняет роль субъекта из описания схемы шаблона, представленного в книге "банды четырех". В качестве методов `attach`, `detach` и `notify` в языке Java используются, соответственно, методы `addObserver`, `deleteObserver` и `notifyObservers` (в Java используется и метод `update`). Язык Java также предоставляет несколько других методов, упрощающих работу с данными классами. (Дополнительные сведения об API языка Java для классов **`Observer`** и **`Observable`** можно найти в Internet по адресу `http://java.sun.com/j2se/1.3/docs/api/ index.html`.)

Основные характеристики шаблона Observer

Назначение	Определяет зависимость типа "один ко многим" между объектами таким образом, что когда состояние одного объекта изменяется, все зависимые от него объекты автоматически уведомляются об этом и обновляются
Задача	Необходимо направить уведомления о том, что некоторое событие имело место согласно изменяющемуся списку объектов
Способ решения	Объекты-наблюдатели (класс ***`Observer`***) делегируют ответственность по контролю за появлением некоторого события центральному объекту (класс ***`Subject`***)
Участники	Объект класса ***`Subject`*** знает всех своих объектов-наблюдателей, потому что они регистрируются у него. Объект класса ***`Subject`*** должен уведомить все объекты класса ***`Observer`*** о наступлении интересующего события. Объекты класса ***`Observer`*** отвечают как за регистрацию себя у объекта класса ***`Subject`***, так и за получение от него необходимой им информации после поступления уведомления
Следствия	Объекты класса ***`Subject`*** могут сообщить объектам класса ***`Observer`*** о наступлении событий, которые неинтересны некоторым из них, если последние обрабатывают только часть всех возможных событий (подробнее об этом мы поговорим ниже). Дополнительный обмен информацией может иметь место в случае, если после получения извещения от объекта класса ***`Subject`*** объектам класса ***`Observer`*** потребуются некоторые дополнительные данные
Реализация	• Те объекты (класс ***`Observer`***), которым необходимо знать о наступлении некоего события, подключаются к другому объекту (класс ***`Subject`***), который наблюдает за наступлением событий или непосредственно инициирует требуемое событие • Когда событие происходит, объект класса ***`Subject`*** сообщает объектам класса ***`Observer`*** о том, что событие имело место • Для реализации единого интерфейса у всех объектов различных типов класса ***`Observer`*** иногда используется шаблон Adapter

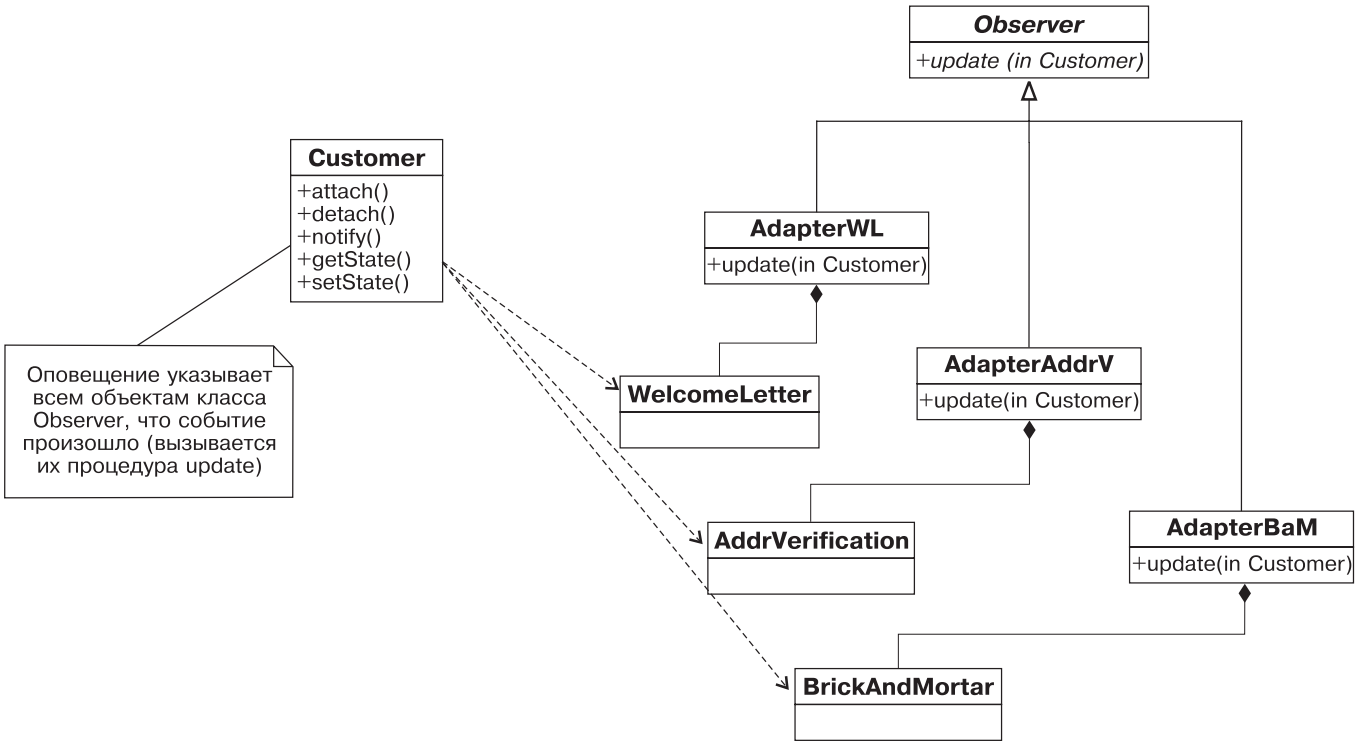

РИС. 17.4. *Реализация классов-наблюдателей с помощью шаблона Adapter*

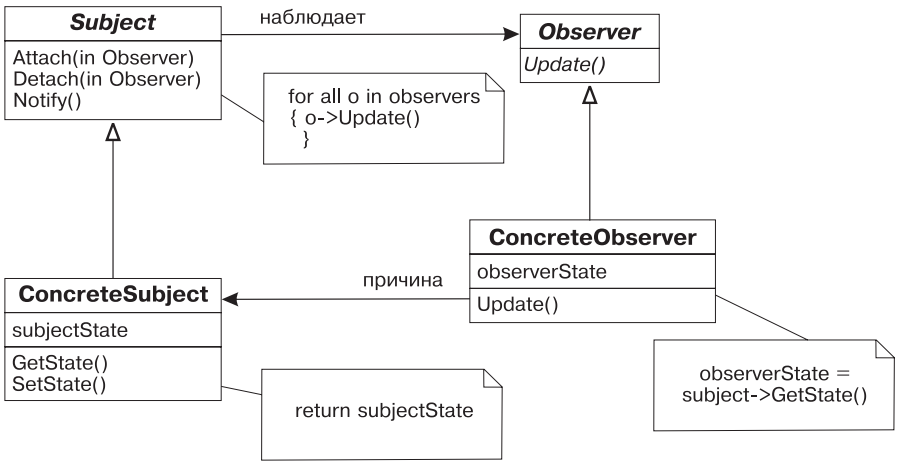

РИС. 17.5. *Стандартный упрощенный вид шаблона Observer*

Дополнительные замечания о шаблоне Observer

Следует заметить, что вовсе не обязательно использовать шаблон Observer всякий раз, когда между объектами возникает зависимость. Например, вполне очевидно, что в процессе подготовки накладной на заказ, объект, отвечающий за начисление налога, должен быть извещен каждый раз, когда в накладную добавляется новая строка, — чтобы сумма налога была пересчитана заново. Применение шаблона Observer в данном случае — это не лучший подход: последовательность операций известна заранее, и вряд ли что-либо когда-либо будет в нее добавлено. Когда зависимости между объектами фиксированы, применение шаблона Observer, скорее всего, только добавит в систему сложности.

Если список объектов, которые должны быть уведомлены о наступлении определенных событий, подвержен изменениям или зависит от конкретных условий, использование шаблона Observer, напротив, принесет большую пользу. Изменения в списке уведомляемых объектов могут быть вызваны как появлением новых требований к системе, так и динамическим появлением или удалением отдельных экземпляров объектов. Шаблон Observer также может быть полезен, если система будет функционировать в различных условиях или у различных заказчиков, каждый из которых имеет собственный список объектов-наблюдателей.

Определенные объекты-наблюдатели могут также обрабатывать только некоторые из возникающих событий. Вспомним пример с объектом **`BrickandMortar`**. В подобной ситуации объект-наблюдатель должен самостоятельно отфильтровать не интересующие его события.

Появление избыточных уведомлений может быть исключено посредством передачи ответственности за фильтрование рассылаемых уведомлений в адрес объекта класса ***`Subject`***. Наилучший способ реализовать это решение — применить шаблон Strategy для проверки необходимости отправки уведомлений. В этом случае каждый

объект-наблюдатель должен предоставить объекту класса ***Subject*** корректную стратегию принятия подобного решения непосредственно при своей регистрации.

Иногда объекты класса ***Subject*** могут вызывать метод update объектов-наблюдателей для передачи им информации. Такой подход помогает обойтись без обратных запросов от объектов-наблюдателей к объекту класса ***Subject***. Однако часто оказывается, что разным объектам-наблюдателям требуется различная информации. В подобном случае вновь можно воспользоваться шаблоном Strategy. На этот раз объект класса ***Strategy*** используется для вызова метода update объектов-наблюдателей. И вновь объекты-наблюдатели должны предоставить для использования объекту класса ***Subject*** соответствующие объекты ***Strategy***.

Резюме

Изучая шаблон Observer, мы определили, какой объект в системе позволит наилучшим образом справиться со вновь появляющимися изменениями. Для шаблона Observer объект, который инициирует событие (объект класса ***Subject***), часто оказывается не в состоянии оповестить все объекты, которые должны быть извещены о наступлении этого события. Для решения проблемы создается интерфейс ***Observer*** и устанавливается правило, согласно которому все объекты-наблюдатели обязаны зарегистрировать себя у объекта класса ***Subject***.

В этой главе наше внимание было сосредоточено на шаблоне Observer, поэтому полезно будет указать несколько общих концепций объектно-ориентированного проектирования, используемых в этом шаблоне (табл. 17.3).

Таблица 17.3. Концепции ООП, используемые в шаблоне Observer

Концепция	Пояснение
Объекты отвечают сами за себя	Существуют различные виды объектов-наблюдателей, но все они получают необходимую им информацию от объекта класса ***Subject***, а затем выполняют необходимые действия согласно их назначению
Абстрактные классы	Абстрактный класс ***Observer*** представляет концепцию объектов, которые нуждаются в получении уведомления. Он предоставляет классу **Subject** (субъекту) общий интерфейс для рассылки всех уведомлений
Полиморфизм и инкапсуляция	Субъект не знает, с каким именно типом объекта класса ***Observer*** он в данный момент взаимодействует. По существу, класс ***Observer*** инкапсулирует конкретные разновидности классов-наблюдателей. Это означает, что если в будущем появятся новые виды классов-наблюдателей, то это не потребует внесения каких-либо изменений в класс **Subject**

Приложение. Пример программного кода на языке C++

Листинг 17.2. Реализация шаблона Observer

```
class Customer {
  public:
    static void attach(Observer *o);
    static void detach(Observer *o);
    String getState();
  private:
    Vector myObs;
    void notifyObs();
}
Customer::attach(Observer *o){
  myObs.addElement(o);
}
Customer::detach(Observer *o){
  myObs.remove(o);
}
Customer::getState () {
  // Могут использоваться другие методы,
  // предназначенные для получения дополнительной информации
}
Customer::notifyObs () {
  for (Enumeration e =
    myObs.elements();
    e.hasMoreElements() ;) {
      ((Observer *) e)->
        update(this);
    }
  }
}

class Observer {
  public:
    Observer();
    void update(Customer *mycust)=0;
      // Создает эту абстракцию
}
Observer::Observer () {
  Customer.attach( this);
}

class AddrVerification : public Observer {
  public:
    AddrVerification();
    void update( Customer *myCust);
}
AddrVerification::AddrVerification () {
}
AddrVerification::update
  (Customer *myCust) {
    // Здесь выполняется проверка адресов.
    // Для получения дополнительной информации о клиенте
    // может использоваться метод myCust
}

class WelcomeLetter : public Observer {
public:
  WelcomeLetter();
```

```
  void update( Customer *myCust);
}
WelcomeLetter::update( Customer *myCust) {
  // Рассылка приветственного письма.
  // Для получения дополнительной информации о клиенте
  // может использоваться метод myCust
}
```

Глава 18

Шаблон Template Method

Введение

В этой главе мы продолжим обсуждение нашего второго учебного примера — приложения поддержки электронной розничной торговли, начатое в главах 14–17.

Здесь мы выполним следующее.

- Познакомимся с шаблоном Template Method при обсуждении новых требований, предъявленных к нашему учебному примеру — системе поддержки электронной розничной торговли.
- Выясним назначение шаблона Template Method.
- Обсудим характеристики шаблона Template Method.
- Познакомимся с моим опытом применения шаблона Template Method на практике.

Предъявление новых требований к системе электронной торговли

Предположим, что в процессе создания приложения заказчик еще раз выдвинул новое требование — обеспечить поддержку системой двух типов СУБД, а именно, Oracle и SQL Server. Обе эти системы используют язык SQL (Structured Query Language — структурированный язык запросов), стандартный язык работы с реляционными БД, упрощающий работу с ними. Однако несмотря на то, что имеется общий стандарт этого языка, существуют некоторые различия в его реализации в разных СУБД.

Будем считать, что выполнение запроса к базе данных включает следующую последовательность действий.

1. Подготовка команды `CONNECT` (установить соединение).
2. Отправка команды `CONNECT` в базу данных.
3. Подготовка команды `SELECT` (выбрать).
4. Отправка команды `SELECT` в базу данных.
5. Получение возвращаемого набора данных.

Конкретные реализации баз данных, о которых идет речь, отличаются друг от друга, поэтому процедуры подготовки команд для них также слегка различаются.

Шаблон Template Method

Назначение шаблона Template Method (шаблонный метод) состоит в том, чтобы помочь выделить (абстрагировать) некоторый общий процесс из нескольких различных процедур. В работе "банды четырех" назначение шаблона Template Method сформулировано следующим образом.

> Определение алгоритма выполнения операции, с передачей некоторых ее этапов для выполнения в подклассах. Переопределение этапов алгоритма без изменения его структуры.[1]

Другими словами, несмотря на то, что процедуры установки соединений и выполнения запросов к базам данных в СУБД Oracle и SQL Server отличаются в деталях, концептуально эти процессы одинаковы. Шаблон Template Method позволяет выявить существующие точки соприкосновения и поместить их в один абстрактный класс, тогда как все различия инкапсулируются в порожденных классах. Шаблон Template Method помогает организовать и управлять выполнением действий, общих в различных процессах.

Применение шаблона Template Method в системе электронной торговли

В нашем учебном примере различия в процедурах доступа к базам данных разных СУБД проявляются в особенностях реализации выполняемых этапов (рис. 18.1).

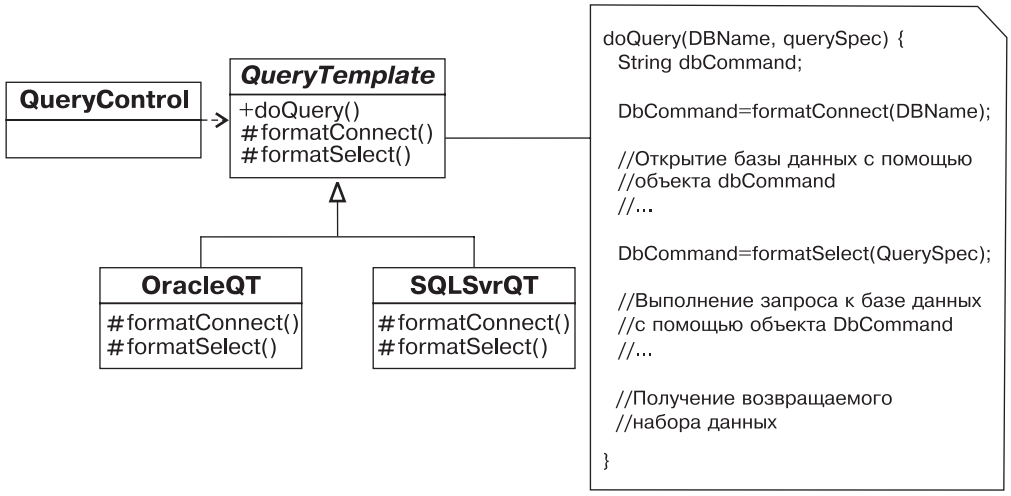

РИС. 18.1. *Использование шаблона Template Method для организации выполнения запросов к БД*

---

[1] *Gamma E., Helm R., Johnson R., Vlissides J.* Design Patterns: Elements of Reusable Object-Oriented Software, *Reading, MA: Addison-Wesley, 1995, с. 325.*

В классе ***QueryTemplate*** создается метод с именем doQuery, предназначенный для выполнения запросов к базе данных. В качестве аргументов ему передается имя базы данных и текст запроса. Метод DoQuery включает код выполнения приведенной выше последовательности из пяти этапов и организует вызов виртуальных методов (formatConnect и formatSelect) для выполнения действий, которые должны быть реализованы различными способами.

Метод DoQuery реализован следующим образом. Как следует из рис. 18.1, прежде всего необходимо подготовить текст команды CONNECT, предназначенной для установки соединения с базой данных. Хотя абстрактный класс (***QueryTemplate***) знает, что эта команда должна быть подготовлена, ему неизвестно, как это можно выполнить на практике. Конкретный код форматирования данной команды предоставляется в соответствующем производном классе. Такой же подход применяется и при подготовке текста команды SELECT.

Шаблон Template Method управляет выполнением указанной процедуры, поскольку вызов методов осуществляется с помощью ссылки, указывающей на один из производных классов. Следовательно, несмотря на то, что объект класса **QueryControl** содержит ссылку типа ***QueryTemplate***, в действительности он ссылается на объект класса **OracleQT** или **SQLSvrQT**. Следовательно, когда метод doQuery вызывается в любом из этих объектов, при разрешении ссылки на метод сначала просматриваются методы в соответствующем производном классе. Допустим, что объект класса **QueryControl** ссылается на объект класса **OracleQT**. Поскольку в классе **OracleQT** метод DoQuery класса ***QueryTemplate*** не переопределяется, вызывается именно этот метод. Данный метод выполняется вплоть до вызова в нем метода formatConnect. Поскольку изначально вызов метода DoQuery поступил к объекту класса **OracleQT**, выполняется метод formatConnect именно этого объекта, т.е. объекта класса **OracleQT**. Затем управление возвращается в метод doQuery объекта ***QueryTemplate*** и выполняется следующий участок кода, общий для всех запросов, — пока не потребуется обработать еще один тип вариаций, инкапсулированный в методе formatSelect. И вновь вызываемый метод находится в том самом объекте, на который ссылается объект класса **QueryControl** (в данном случае это объект класса **OracleQT**).

Если впоследствии потребуется организовать работу с новым типом СУБД, шаблон Template Method предоставит для этой цели готовую структуру (шаблон), которую останется только заполнить. Достаточно будет создать новый конкретный класс, производный от абстрактного класса ***QueryTemplate***, и реализовать в нем выполнение всех специфических для данной СУБД запросов.

Дополнительные замечания о шаблоне Template Method

Иногда в классе используется несколько различных шаблонов Strategy. Впервые увидев диаграмму классов шаблона Template Method, я решил, что шаблон Template Method — это просто совокупность шаблонов Strategy, работающих совместно. Однако это потенциально опасное (и обычно ошибочное) представление. Применение сразу нескольких шаблонов Strategy, соединенных между собой, встречается достаточно редко и просто нежелательно, поскольку уменьшает гибкость системы.

Шаблон Template Method рекомендуется применять в тех случаях, когда имеют место различные в реализации, но концептуально сходные процессы. Вариации для

каждого из процессов объединяются, поскольку они связаны с конкретным процессом. В приведенном выше примере было показано, что если команду `CONNECT` потребовалось отформатировать для базы данных Oracle, то и команды `SELECT` также будут форматироваться для этой базы данных Oracle.

Основные характеристики шаблона Template Method

Назначение	Определение алгоритма выполнения операции с передачей некоторых ее этапов для выполнения в подклассах. Переопределение этапов алгоритма без изменения его структуры
Задача	Существует процедура или последовательность выполняемых этапов, которая является неизменной на верхнем уровне детализации, но ее отдельные этапы могут иметь различные реализации на более низком уровне детализации
Способ решения	Допускается определение изменяющихся этапов с сохранением основного процесса неизменным
Участники	Шаблон Template Method состоит из абстрактного класса ***`AbsrtactClass`***, включающего определение основного метода `TemplateMethod` (рис. 18.2), и абстрактных методов *`PrimitiveOperation`*, которые должны переопределяться в производных классах. Каждый конкретный класс **`ConcreteClass`**, производный от абстрактного класса, должен включать реализацию всех абстрактных методов шаблона, специфическую для конкретной вариации процесса
Следствия	Шаблон TemplateMethod обеспечивают хорошую платформу для достижения многократного использования кода. Он также будет полезен для получения гарантий, что все необходимые этапы будут реализованы. Шаблон предусматривает компоновку всех переопределяемых этапов в один конкретный класс **`ConcreteClass`** и поэтому должен использоваться лишь тогда, когда все эти вариации будут проявляться вместе и только вместе
Реализация	Создается абстрактный класс, в котором основная выполняемая процедура реализуется с использованием абстрактных методов. Эти абстрактные методы должны быть реализованы в подклассах, обеспечивающих выполнение каждого этапа общей процедуры. Если выполняемые этапы изменяются независимо, каждый из них может быть реализован с использованием шаблона Strategy

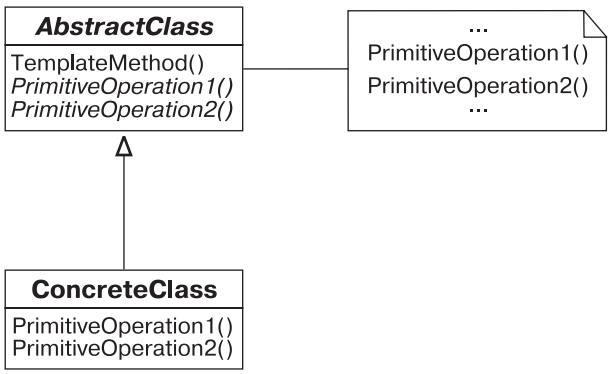

РИС. 18.2. ***Стандартное упрощенное представление шаблона Template Method***

Резюме

Иногда в приложении необходимо организовать выполнение нескольких вариантов одной и той же процедуры. Набор операций этих процедур является общим на концептуальном уровне, но реализация отдельных операций в каждом случае может отличаться от других. Например, процедуры выполнения SQL-запросов в базе данных для различных СУБД на концептуальном уровне одинаковы, но некоторые детали (например, способ установки соединения с базой данных) могут различаться.

Шаблон Template Method позволяет определить общую последовательность выполняемых этапов, а затем при необходимости переопределить способ реализации отдельных этапов в общей последовательности.

Глава 19

Шаблон Factory Method

Введение

В этой главе мы продолжим обсуждение нашего второго учебного примера — приложения поддержки электронной розничной торговли, начатое в главах 14–18.

Здесь мы выполним следующее.

- Познакомимся с шаблоном Factory Method в свете новых требований, предъявленных к нашему учебному примеру — системе поддержки электронной розничной торговли.
- Выясним назначение шаблона Factory Method.
- Обсудим характеристики шаблона Factory Method.
- Познакомимся с моим опытом применения шаблона Factory Method на практике.

Предъявление новых требований к системе электронной торговли

В главе 18, *Шаблон Template Method*, мы оставили без внимания вопрос, как создать экземпляры объектов, необходимых для работы с базой данных в текущем контексте. Возлагать ответственность за создание этих объектов на объект-клиент может оказаться нежелательным. Предпочтительней было бы, чтобы этим занимался непосредственно класс ***QueryTemplate***.

В главе 18 каждый конкретный класс, производный от абстрактного класса ***QueryTemplate***, предназначен для определенной СУБД. Следовательно, на каждый такой конкретный класс можно возложить ответственность за создание экземпляра объекта, работающего с соответствующей СУБД. Это решение можно считать весьма удачным, но при условии, что в системе с объектами взаимодействия с СУБД будет работать только класс ***QueryTemplate*** (и производные от него классы). Схематически это решение представлено на рис. 19.1.

На рис. 19.1 показано, что метод `doQuery` из шаблона Template Method использует метод `makeDB` для создания экземпляра объекта, работающего с соответствующей СУБД. Класс ***QueryTemplate*** ничего не знает о том, экземпляр какого именно объекта работы с СУБД будет создан. Ему известно лишь то, что объект подобного типа непременно *должен* быть создан, и этот класс предоставляет необходимый интерфейс для данного процесса. Обязанность знать, какой именно тип объекта следует создать, возлагается на классы, производные от абстрактного класса ***QueryTemplate***. Таким образом, на данном уровне создание объекта взаимодействия с СУБД требуемого типа осуществляется в методе производного класса.

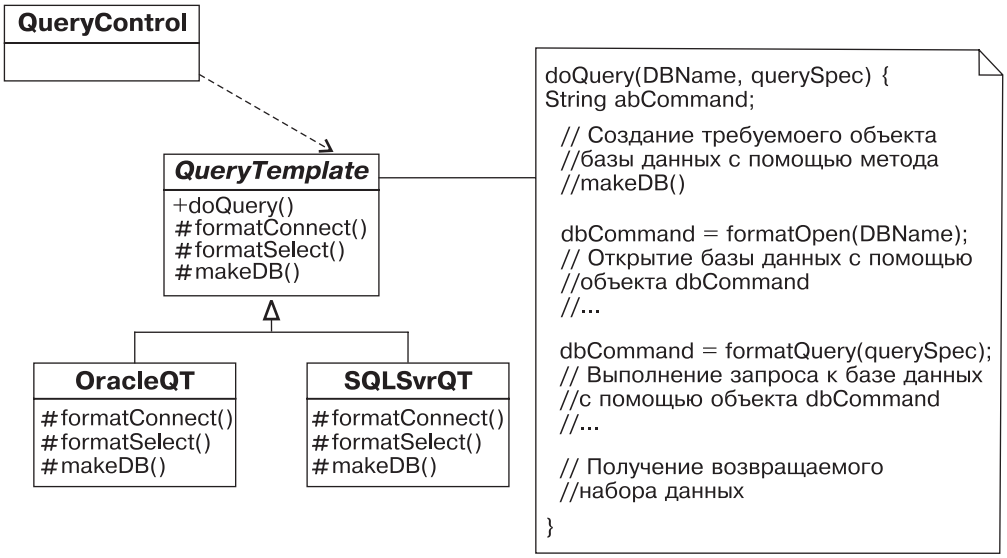

РИС. 19.1. ***Шаблон Template Method (метод `doQuery`) использует шаблон Factory Method (метод `makeDB`)***

Поскольку в данном решении фигурирует метод, предназначенный для создания экземпляра объекта, этот шаблон получил имя Factory Method (метод-фабрика).

Открытые или защищенные методы?

Обратите внимание на то, что на рис. 19.1 методы `makeDB` являются защищенными (об этом говорит символ # перед именем метода). В этом случае только сам класс ***`QueryTemplate`*** и все классы, производные от него, имеют доступ к данным методам. Если необходимо разрешить доступ к этим методам и объектам других классов, отличных от ***`QueryTemplate`***, следует объявить их с типом `public`. Это другой, достаточно распространенный способ использования шаблона Factory Method. В данном случае ответственность за принятие решения относительно того, объект какого именно типа будет создан, также возлагается на конкретный производный класс.

Шаблон Factory Method

Шаблон Factory Method позволяет эффективно распределить ответственность за создание экземпляров объектов. В работе "банды четырех" назначение шаблона Factory Method сформулировано следующим образом.

> Определить интерфейс для создания экземпляра объекта с предоставлением производным классам права принимать решение о том, какой именно тип объекта будет создан. Шаблон Factory Method позволяет абстрактному классу возложить обязанности по созданию экземпляра объекта на его производные классы.[1]

---

[1] *Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software, Reading, MA : Addison-Wesley, 1995, с. 325.*

Дополнительные замечания о шаблоне Factory Method

В классическом варианте реализации шаблона Abstract Factory присутствует абстрактный класс, определяющий методы создания семейств объектов. Для каждого из семейств объектов определяется производный класс, отвечающий за создание экземпляров объектов данного семейства. Каждый метод, определенный в абстрактном классе, а затем переопределенный в производном классе, отвечает требованиям шаблона Abstract Factory.

Иногда полезно создать иерархическую структуру классов, параллельную структуре существующих классов, с делегированием новой иерархии некоторых обязательств. В этом случае важно, чтобы каждый объект в исходной иерархии был способен создать экземпляр объекта соответствующего класса параллельной иерархии. Для этого можно использовать шаблон Factory Method.

Шаблон Factory Method обычно используется при формировании рабочей среды. Поскольку рабочая среда определяется на абстрактном уровне, то здесь обычно отсутствуют какие-либо сведения в отношении правил создания экземпляров конкретных объектов. Право принятия решений о создании экземпляров конкретных объектов делегируется пользователю данной рабочей среды.

Основные характеристики шаблона Factory Method

Назначение	Определить интерфейс для создания объекта с передачей производным классам права принимать решение, экземпляр какого именно класса следует создать
Задача	Классу требуется создать экземпляр объекта конкретного класса, производного от другого класса, но точно не известно, какого именно. Шаблон Factory Method позволяет порожденному классу принимать подобное решение
Способ решения	Производный класс принимает решение, экземпляр какого класса требуется создать и как это следует делать
Участники	Класс ***`Product`*** представляет собой интерфейс для того типа объекта, который создается с помощью шаблона Factory Method. Класс ***`Creator`*** представляет собой интерфейс, который определяет шаблон Factory Method
Следствия	Для создания экземпляра определенного класса `ConcreteProduct` необходимо определить конкретный класс `ConcreteCreator`, производный от абстрактного класса ***`Creator`***
Реализация	Использование метода абстрактного класса, который также является абстрактным (чисто виртуальным в языке C++). Программный код абстрактного класса ссылается на этот метод, когда требуется создать экземпляр объекта конкретного производного класса, но не известно, какого именно

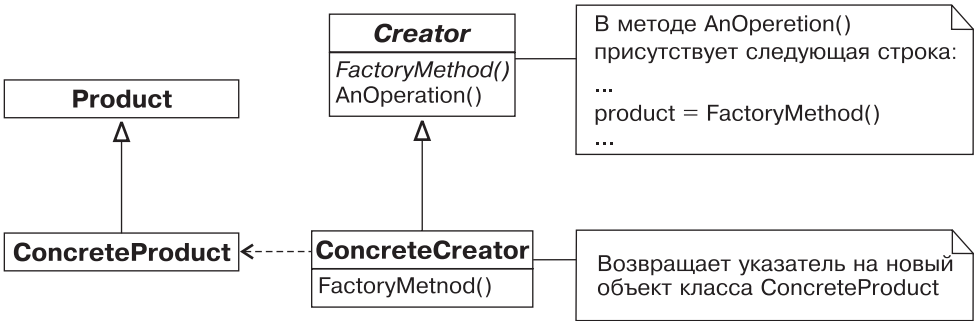

РИС. 19.2. ***Стандартное упрощенное представление шаблона Factory Method***

Резюме

Шаблон Factory Method — один из самых простых и понятных шаблонов, который очень широко используется. К нему прибегают, когда необходимо перенести реализацию правил создания экземпляра объекта в некоторый производный класс. В таких случаях наиболее эффективное решение состоит в том, чтобы поместить реализацию метода в тот класс, который отвечает за требуемое поведение.

Глава 20

Матрица анализа

Введение

В этой главе мы продолжим обсуждение нашего второго учебного примера — приложения поддержки электронной розничной торговли, начатое в главах 14–19.

Теперь, когда мы обсудили полный набор шаблонов по отдельности, пришло время вернуться на шаг назад и вновь обратиться к одной из самых серьезных проблем в разработке программного обеспечения — обработке вариаций, присутствующих в данной проблемной области. Шаблоны проектирования могут оказать большую помощь в выявлении и эффективном управлении этими вариациями.

Здесь мы выполним следующее.

- Рассмотрим проблему вариаций, имеющих место в реальном мире.
- Обсудим ту часть нашего учебного примера (системы поддержки электронной розничной торговли), которая связана с наиболее сложной проблемой обработки вариаций в приложении. По ходу решения этой проблемы мы познакомимся с методом создания матрицы анализа — простейшим вариантом таблицы принятия решений, который может с пользой применяться для обнаружения и обработки имеющихся вариаций в концепциях. Затем мы проведем параллель между этим подходом и концепциями, выдвинутыми Кристофером Александером и Джимом Коплином.
- Познакомимся с моим опытом применения метода создания матрицы анализа на практике.

Вариации в реальном мире

В реальном мире проблемы редко бывают простыми или легко устранимыми. За исключением наиболее тривиальных случаев, любая проблема всегда кажется очень сложной, а возникающие вариации не поддаются систематизации. Каждая проблема воспринимается как западня, ведущая к разрушению столь тщательно разработанной нами модели.

Например, как правило, поступающие в больницу пациенты сначала направляются в приемное отделение. Но когда сложившаяся ситуация создает угрозу жизни пациента, он направляется непосредственно в отделение скорой помощи, тем самым нарушая обычный порядок. Подобные вариации, имеющие место в реальном мире, воспринимаются как разнообразные особые случаи, которые создаваемая система должна уметь обрабатывать.

Понятно, что все это создает дополнительные проблемы для любого аналитика. Могут ли шаблоны проектирования помочь более эффективно обрабатывать вариации?

Воспользуемся следующим подходом. Внесем некоторое изменение в систему, а затем используем аналитические методы для выявления тех шаблонов, которые следует включить в проект. Данный подход предполагает выполнение следующих этапов.

1. Идентификация наиболее важных функций для одного случая и систематизация их в виде матрицы. Обозначим каждую функцию той концепцией, которую эта функция представляет.
2. Распространение этого подхода на все остальные случаи с расширением матрицы по мере необходимости. Каждый случай обрабатывается независимо от других.
3. Расширение матрицы анализа за счет добавления новых концепций.
4. Использование строк для идентификации правил.
5. Применение столбцов для идентификации особых случаев.
6. Идентификация шаблонов проектирования на основании проведенного анализа.
7. Разработка проекта на концептуальном уровне.

Вариации в учебном примере — система международной электронной торговли

Предположим, что создаваемая нами система поддержки электронной розничной торговли должна обрабатывать заказы в нескольких странах мира. Чтобы упростить условия, примем, что этих стран только две — США и Канада. Проанализировав предъявляемые к системе новые требования, можно обнаружить несколько обстоятельств, которые будут изменяться в зависимости от страны (табл. 20.1).

Таблица 20.1. Обстоятельства, зависящие от места жительства клиента

Вариант	Процедура
США	• Стоимость доставки вычисляется по тарифам компании UPS
	• Для проверки адресов используются американские почтовые правила
	• Вычисляется налог на основании цены товара или услуги, в зависимости от места проживания
	• Суммы рассчитываются в долларах США
Канада	• Стоимость доставки вычисляется по тарифам ведущей канадской компании доставки грузов
	• Для проверки адресов используются канадские почтовые правила
	• Вычисляется налог на основании цены на товар или услуги в зависимости от налоговых правил конкретной канадской провинции
	• Суммы рассчитываются в канадских долларах

Вариации, связанные с вновь поставленной задачей, не слишком сложны. Ее решение кажется вполне очевидным. Действительно, указанная проблема проста. Но

она позволяет продемонстрировать метод работы с вариациями, который я многократно использовал. Мой метод прост, и он легко может быть распространен на множество задач в реальном мире. Я назвал этот метод "Матрица анализа".

На данной стадии анализа наша цель состоит в том, чтобы найти те *концепции*, которые изменяются, определить точки соприкосновения и обнаружить недостающие требования. Концепции соответствуют конкретным требованиям для каждого особого случая. Обсуждение вопросов проектирования и реализации мы отложим до более поздних стадий.

Начнем с рассмотрения первого случая.

Проанализируем каждую функцию, которая должна быть реализована, и обозначим концепцию, которую она представляет. Поместим такую функцию в отдельную строку матрицы, а название представляемой ей концепции — в крайний слева столбец.

Рассмотрим данный процесс подробно, начиная с табл. 20.2.

Таблица 20.2. Заполнение матрицы анализа – первая концепция

	США
Вычисление стоимости доставки	Вычисляется по тарифам компании UPS

Теперь обработаем следующую часть информации: "Для проверки адресов используются американские почтовые правила". Добавим в матрицу еще одну строку и поместим в нее данную информацию, как показано в табл. 20.3.

Таблица 20.3. Заполнение матрицы анализа – вторая концепция

	США
Вычисление стоимости доставки	Вычисляется по тарифам компании UPS
Проверка адреса	Используются американские почтовые правила

Далее поместим в матрицу остальные концепции для первого случая, как показано в табл. 20.4.

Таблица 20.4. Заполнение матрицы анализа. Законченный первый вариант — продажа в США

	США
Вычисление стоимости доставки	Вычисляется по тарифам компании UPS
Проверка адреса	Используются американские почтовые правила
Расчет налогов	Рассчитываются государственные и местные налоги США
Валюта	Доллары США

Теперь перейдем к следующему конкретному случаю, а затем к другим, добавляя столбец для каждого нового варианта и заполняя каждую новую ячейку той информацией, которая у нас имеется. Законченная матрица с учетом второго специального случая показана в табл. 20.5.

Таблица 20.5. Заполнение матрицы анализа. Законченный второй вариант — продажа в Канаде

	США	Канада
Вычисление стоимости доставки	Вычисляется по тарифам компании UPS	Вычисляется по тарифам ведущей канадской компании доставки грузов
Проверка адреса	Используются американские почтовые правила	Используются канадские почтовые правила
Расчет налогов	Рассчитываются государственные и местные налоги США	Рассчитываются государственные и местные налоги Канады
Валюта	Доллары США	Канадские доллары

В процессе построения матрицы анализа часто обнаруживаются недостающие требования к системе. Эту информацию следует использовать для расширения области анализа. Подобная неопределенность может возникать из-за того, что поступившая от заказчика информация была неполной. В одном случае заказчик может упоминать какие-то конкретные требования, а в другом опустить их. Например, при формулировании требований по работе с клиентами из США может не упоминаться ограничение по максимальному весу заказа, тогда как для клиентов из Канады такое ограничение может быть установлено — скажем, на уровне 31,5 кг. После сравнения набора требований для каждого конкретного случая необходимо заполнить пустые ячейки в матрице анализа. В нашем случае следует обратиться к пользователям, занимающимся клиентами из США, и задать им конкретный вопрос о максимально допустимом весе заказа (кстати, подобного ограничения может просто не существовать).

Со временем могут появиться новые прецеденты (например, может потребоваться добавить в систему поддержку клиентов из Германии). Всякий раз, когда в одном из прецедентов обнаруживается новая концепция, в матрицу анализа следует добавить новую строку, даже если соответствующая информация для всех остальных случаев пока неизвестна. Сказанное выше иллюстрируется данными в табл. 20.6.

Таблица 20.6. Расширение матрицы анализа. Добавлен третий вариант — продажа в Германии

	США	Канада	Германия
Вычисление стоимости доставки	Вычисляется по тарифам компании UPS	Вычисляется по тарифам ведущей канадской компании доставки грузов	Вычисляется по тарифам ведущей немецкой компании доставки грузов
Проверка адреса	Используются американские почтовые правила	Используются канадские почтовые правила	Используются германские почтовые правила
Расчет налогов	Рассчитываются государственные и местные налоги США	Рассчитываются государственные и местные налоги Канады	Рассчитываются государственные налоги Германии
Валюта	Доллары США	Канадские доллары	Евро
Формат даты	мм/дд/гггг	мм/дд/гггг	дд/мм/гггг

Несколько слов о работе с заказчиком

Обширный опыт общения с заказчиками позволил мне понять несколько важных моментов.

- Как правило, заказчики знают проблемную область очень хорошо (более того, они знают ее лучше, чем я когда-либо буду знать).
- Чаще всего заказчики не воспринимают предметную область на концептуальном уровне, как это принято у разработчиков. Напротив, они говорят о конкретных проявлениях тех или иных частных случаев.
- Заказчики часто используют определение "всегда", но на самом деле следовало бы говорить "обычно".
- Также часто они используют определение "никогда", подразумевая "редко".
- Заказчики часто утверждают, что рассказали обо всех возможных ситуациях, тогда как в действительности речь шла только о том, что случается *обычно*.

Подводя итог, можно сказать, что я полностью доверяю объяснениям заказчиков, когда они дают ответы на заданные мной конкретные вопросы, но с большим сомнением отношусь к их общим рассуждениям. Я всегда стараюсь вести диалог с заказчиком на максимально конкретном уровне. Даже те из заказчиков, которые полагают, что они способны осмыслить задачу на концептуальном уровне, в попытках оказать помощь разработчику часто переоценивают свои возможности.

Теперь, когда все концепции выявлены, как следует поступить с полученными знаниями? С чего начать, чтобы можно было перейти к вопросам реализации?

Еще раз посмотрим на матрицу анализа, представленную в табл. 20.6. Первая строка называется "Вычисление стоимости доставки" и включает ячейки со значениями "Вычисляется по тарифам компании UPS", "Вычисляется по тарифам ведущей канадской компании доставки грузов", "Вычисляется по тарифам ведущей немецкой компании доставки грузов". В этой строке одновременно представлены следующие концепции.

- Общее правило реализации "Вычисления стоимости доставки".
- Конкретный набор правил, который должен быть реализован, — указана компания, тарифами которой следует пользоваться при вычислении стоимости доставки заказа в каждой из стран.

Таким образом, каждая строка включает описание конкретных способов реализации соответствующей обобщенной концепции. В нашем примере две строки (валюта и формат даты) могут быть обработаны непосредственно на уровне объектов. Например, денежные расчеты могут обрабатываться с помощью объектов, включающих данные типа `currency` (валюта). Многие языки программирования поддерживают обработку различных национальных форматов даты с помощью специальных библиотек. В табл. 20.7 представлен концептуальный способ обработки каждой строки матрицы анализа.

А что представляют собой столбцы матрицы анализа? Они включают набор конкретных реализации для каждого отдельного случая, представленного данным столбцом. Эту мысль наглядно иллюстрирует табл. 20.8.

Таблица 20.7. Правила реализации конкретных строк

	США	Канада	Германия
Вычисление стоимости доставки	Конкретные реализации методов вычисления стоимости доставки для каждой страны		
Проверка адреса	Конкретные реализации методов проверки правильности адреса для каждой страны		
Расчет налогов	Конкретные реализации методов расчета налогов для каждой страны		
Объекты Money	Можно использовать такие объекты Money, которые будут содержать поля Currency и Amount, обеспечивающие автоматическую конвертацию валюты		
Объекты Date	Можно использовать такие объекты Date, которые будут автоматически выбирать способ представления даты в соответствии с нормами той страны, в которой живет заказчик		

Таблица 20.8. Правила реализации конкретных столбцов

	США	Канада	Германия
Вычисление стоимости доставки	Конкретные реализации методов, используемые при работе с заказчиками из США	Конкретные реализации методов, используемые при работе с заказчиками из Канады	Конкретные реализации методов, используемые при работе с заказчиками из Германии
Проверка адреса			
Расчет налогов			
Объекты Money			
Объекты Date			

Например, в первом столбце представлены конкретные реализации для отдельных аспектов процедуры обработки заказа для клиента, проживающего в Соединенных Штатах.

Каким же образом можно перейти от понятий, собранных в таблице, к шаблонам проектирования? Еще раз обратимся к табл. 20.7. В каждой ее строке представлен конкретный способ реализации концепции, указанной в крайнем слева столбце.

- В строке "Вычисление стоимости доставки" значения "Вычисляется по тарифам компании UPS" и "Вычисляется по тарифам ведущей канадской компании доставки грузов" в действительности дают ответ на вопрос, "Как следует вычислять стоимость доставки заказа". Алгоритм требуемых вычислений инкапсулирован в ячейке "Вычисление стоимости доставки", а конкретными правилами его использования являются значения "Вычисляется по тарифам компании UPS", "Вычисляется по тарифам ведущей канадской компании доставки грузов" и "Вычисляется по тарифам ведущей немецкой компании доставки грузов".
- Следующие две строки также представляют собой объединение концепций и различных правил, описывающих их конкретные реализации.
- Последние две строки представляют классы, которые могут согласованно использоваться в масштабах всего приложения и будут проявлять разное поведение в зависимости от страны, в которой находится клиент.

Следовательно, каждую из первых трех строк можно воспринимать как шаблон Strategy. Этот подход иллюстрируется в табл. 20.9. Таким образом, объекты в первой строке могут быть реализованы в виде шаблона Strategy, инкапсулирующего правило "Вычисление стоимости доставки".

Аналогично можно проанализировать и столбцы матрицы. Каждый из столбцов описывает правила, которые используются в некотором конкретном случае. Таким образом, столбцы представляют семейства объектов, необходимых для обработки такого случая. Все это напоминает структуру шаблона Abstract Factory, что иллюстрирует табл. 20.10.

Таким образом, выяснив, что в нескольких строках представлен шаблон Strategy, а каждый столбец представляет семейство объектов шаблона Abstract Factory, можно приступить к разработке концептуального решения для проекта нашего учебного приложения. Один из возможных вариантов концептуальной схемы проекта представлен на рис. 20.1.

Дополнительные замечания

На практике, в матрице анализа может быть обнаружен почти каждый из рассмотренных нами шаблонов, использующих полиморфизм (например, Bridge, Decorator, Template Method и Observer). Кроме того, в матрице анализа могут использоваться шаблоны Composite (композит), Proxy (полномочия), Chain of Responsibility (цепь ответственности), Command (команда), Iterator (итератор), Mediator (посредник) и Visitor (посетитель).

Таблица 20.9. Реализация правил с использованием шаблона Strategy

	США	Канада	Германия
Вычисление стоимости доставки	Объекты в этой строке могут быть реализованы с помощью шаблона Strategy, инкапсулирующего правила вычисления стоимости доставки		
Проверка адреса	Объекты в этой строке могут быть реализованы с помощью шаблона Strategy, инкапсулирующего правила проверки адреса		
Расчет налогов	Объекты в этой строке могут быть реализованы с помощью шаблона Strategy, инкапсулирующего правила расчета налогов		
Объекты Money	Можно использовать такие объекты Money, которые будут содержать поля Currency и Amount, обеспечивающие автоматическую конвертацию валюты		
Объекты Date	Можно использовать такие объекты Date, которые будут автоматически выбирать способ представления даты в соответствии с нормами той страны, в которой живет заказчик		

Таблица 20.10. Реализация правил с использованием шаблона Abstract Factory

	США	Канада	Германия
Вычисление стоимости доставки	Работа этих объектов может координироваться с помощью шаблона Abstract Factory	Работа этих объектов может координироваться с помощью шаблона Abstract Factory	Работа этих объектов может координироваться с помощью шаблона Abstract Factory
Проверка адреса			
Расчет налогов			
Объекты Money			
Объекты Date			

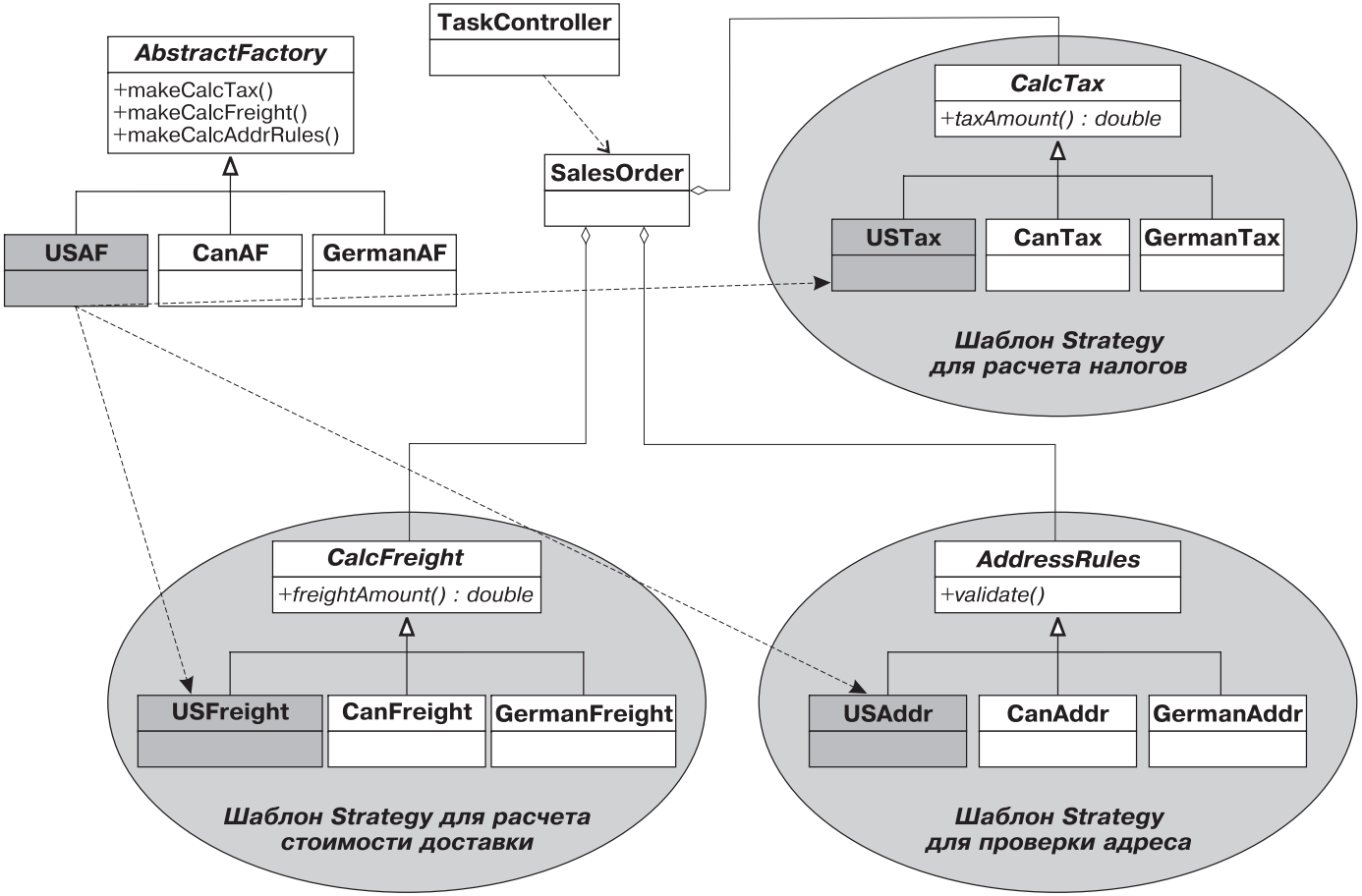

РИС. 20.1. *Концептуальная схема проекта с использованием шаблонов Strategy и Abstract Factory*

Например, пусть в нашем примере с приложением поддержки системы электронной розничной торговли будет добавлено требование выводить на печать накладную с указанием о существовании нескольких разновидностей этого документа:

- для клиентов из США накладная содержит только заголовок;
- для клиентов из Канады накладная содержит заголовок и нижний колонтитул;
- для клиентов из Германии накладная содержит два различных нижних колонтитула.

Данная информация может быть отражена в отдельной строке матрицы анализа, причем каждой стране будет соответствовать свой формат печатаемого документа. Реализацию представленных в этой строке новых требований можно осуществить с помощью шаблона Decorator.

Несмотря на то что матрица анализа редко охватывает все аспекты некоторой проблемной области, я считаю этот метод весьма полезным, по крайней мере, для большинства известных мне проблемных областей. Особенно полезным он будет в ситуации, когда существует столь большое количество специальных случаев, что просто невозможно удержать их все в голове, не утеряв общей картины.

На самом деле все не так просто. Очень редко различные группы требований предоставляются аналитикам или разработчикам в сколько-нибудь согласованном виде. Однако это лишь незначительно усложняет процедуру создания матрицы анализа. В подробных ситуациях я беру очередную функцию и стараюсь найти в левом столбце ту концепцию, вариацией которой она является. Если такая концепция будет обнаружена, данная функция помещается в соответствующую строку. Если требуемая концепция отсутствует в матрице анализа, необходимо создать новую строку.

В чрезвычайно сложных случаях построение матрицы анализа оказывается единственным действенным методом овладеть ситуацией. В моей практике однажды встретился клиент, в требованиях которого особые случаи исчислялись десятками. Каждый особый случай представлял собой независимо разработанную систему управления документами. Проблема состояла в том, чтобы объединить все эти системы управления документами в единое целое. Количество особых случаев было настолько велико (матрица насчитывала почти сотню строк или даже более того), что представлялось просто невозможным осмыслить всю систему в целом. У аналитиков не было подходящего концептуального решения, охватывающего все существующие аспекты проблемы. Они могли лишь выделить общие правила и случаи, являющиеся исключениями. Составляя матрицу анализа, я рассмотрел каждый случай отдельно, затем выделил общие данные и варианты поведения (они были описаны в крайнем слева столбце матрицы). После этого мне осталось лишь выполнить их реализацию с помощью шаблонов проектирования.

Резюме

Учет вариаций в концепциях можно считать одной из самых больших трудностей, с которыми сталкивается аналитик. В этой главе предложен простой инструмент анализа, который может оказаться весьма полезным при решении проблем подобного типа. Я назвал этот инструмент матрицей анализа, и могу сказать, что его идея основана он на концепциях, предложенных Кристофером Александером и Джимом Коплином. Приведенный здесь пример применения этого инструмента для решения относительно простой задачи демонстрирует, как можно выявить типы шаблонов, свойственных данной проблемной области. Хотя данный инструмент может быть чрезвычайно полезен для обработки вариаций и описания характеристик проблемной области, я не претендую на то, что он охватывает все аспекты проектирования.

ЧАСТЬ VI

Завершение и начало

В этой части

В этой части мы продолжим обсуждение нового подхода к объектно-ориентированному проектированию. В частности, будет рассмотрен вопрос о том, как этот новый подход применялся при проектировании и реализации шаблонов проектирования. В завершении приводится обширный список рекомендуемой литературы.

Глава	Предмет обсуждения
21	Обзор действующих сил и взаимосвязей в шаблонах проектирования в контексте нового подхода к объектно-ориентированному проектированию
22	Предлагается литература и указываются другие ресурсы, полезные для дальнейшего изучения