

Шаблоны проектирования

В этой части

В этой части мы познакомимся с шаблонами проектирования — узнаем, что они собой представляют и как используются. Здесь описываются четыре шаблона, позволяющих успешно решить задачу о системе САПР, обсуждавшуюся в главе 3, *Проблема, требующая создания гибкого кода*. Мы рассмотрим каждый из них в отдельности, а затем в применении к нашей задаче. Описание этих шаблонов опирается на объектно-ориентированные концепции, изложенные в основополагающей работе "банды четырех" (Гамма, Хелм, Джонсон и Влиссайдес) *Design Patterns: Elements of Reusable Object-Oriented Software*.

Глава	Предмет обсуждения
5	<ul style="list-style-type: none">• Знакомство с шаблонами проектирования• Концепция шаблонов проектирования, их первоначальное появление в архитектуре и последующий перенос этой концепции в область проектирования программного обеспечения• Мотивы для изучения шаблонов проектирования
6	<ul style="list-style-type: none">• Шаблон Facade (фасад), его определение, использование и реализация• Шаблон Facade в применении к задаче, связанной с САПР
7	<ul style="list-style-type: none">• Шаблон Adapter (адаптер), его определение, использование и реализация• Сравнение шаблонов Facade и Adapter• Шаблон Adapter в применении к задаче, связанной с САПР
8	<ul style="list-style-type: none">• Некоторые важные концепции объектно-ориентированного программирования: полиморфизм, абстракция, классы и инкапсуляция. Обсуждение ведется на основе материала, рассмотренного в главах 5–7

- 9
 - Шаблон **Bridge** (мост). Этот шаблон несколько сложнее двух предыдущих, однако намного полезнее, поэтому и рассматривается более подробно
 - Шаблон **Bridge** в применении к задаче, связанной с САПР
 - 10
 - Шаблон **Abstract Factory** (абстрактная фабрика), позволяющий создавать семейство объектов, его определение, использование и реализация
 - Шаблон **Abstract Factory** в применении к задаче, связанной с САПР
-

Ознакомившись с материалом этой части, читатель узнает, что такое шаблоны проектирования, чем они полезны, и получит представление о четырех конкретных шаблонах. Кроме того, он узнает, как эти шаблоны можно применить к нашей задаче о системе САПР. Однако полученной информации будет все еще недостаточно для создания проекта лучшего, чем самый первый, основанный на простейшем механизме наследования. Успеха мы достигнем лишь тогда, когда научимся использовать шаблоны проектирования способом, отличным от того, который применяет большинство разработчиков-практиков.

Первое знакомство с шаблонами проектирования

Введение

В этой главе мы познакомимся с концепцией шаблонов проектирования. Здесь будут рассмотрены:

- концепция шаблонов проектирования, их первоначальное появление в архитектуре и последующее распространение на область проектирования программного обеспечения;
- мотивы для изучения шаблонов проектирования.

Шаблоны проектирования — это один из важнейших компонентов объектно-ориентированной технологии разработки программного обеспечения. Они широко применяются в инструментах анализа, подробно описываются в книгах и часто обсуждаются на семинарах по объектно-ориентированному проектированию. Существует множество групп и курсов подготовки по их изучению. Часто приходится слышать, что приступить к изучению шаблонов проектирования следует только после того, как будут приобретены определенные навыки применения объектно-ориентированной технологии. Однако, по мнению автора, истинно обратное утверждение: предварительное изучение шаблонов помогает лучше понять объектно-ориентированное проектирование и анализ.

В остальной части книги мы будем обсуждать не только шаблоны проектирования, но и то, как в них проявляются и утверждаются принципы объектно-ориентированной технологии. Эта книга с одной стороны должна помочь читателю глубже понять эти принципы, а с другой — проиллюстрировать, каким образом обсуждаемые в ней шаблоны проектирования могут способствовать созданию хороших проектов.

Предлагаемый ниже материал в некоторых случаях может показаться чересчур абстрактным или философским. Но читателю следует набраться терпения, поскольку в этой главе обсуждаются те основы, которые просто необходимы для понимания шаблонов проектирования. Кроме того, изучение этого материала поможет вам лучше понять принципы построения и методы работы с новыми шаблонами.

Многие из идей, изложенных здесь, были найдены автором в книге Кристофера Александра (Christopher Alexander) *The Timeless Way of Building*¹. Эти идеи будут обсуждаться в различных главах книги.

¹ Alexander C., Ishikawa S., Silverstein M. *The Timeless Way of Building*, New York, NY: Oxford University Press, 1979.

Шаблоны проектирования пришли из области архитектуры и культурологии

Много лет назад архитектор по имени Кристофер Александер задумался над вопросом: "Является ли качество объективной категорией?". Следует ли считать представление о красоте сугубо индивидуальным, или люди могут прийти к общему соглашению, согласно которому некоторые вещи будут считаться красивыми, а другие нет? Александер размышлял о красоте с точки зрения архитектуры. Его интересовало, по каким показателям мы оцениваем архитектурные проекты. Например, если некто вознамерился спроектировать крыльцо дома, то как он может получить гарантии, что созданный им проект будет хорош? Можем ли мы знать заранее, что проект будет действительно хорош? Имеются ли объективные основания для вынесения такого суждения? Существует ли необходимая основа для достижения общего согласия?

Александер принял как постулат, что в области архитектуры такое объективное основание существует. Суждение о том, что некоторое здание является красивым, — это не просто вопрос вкуса. Красоту можно описать с помощью объективных критериев, которые могут быть измерены.

К похожим выводам пришли и исследователи в области культурологии. В пределах одной культуры большинство индивидуумов имеют схожие представления о том, что является сделанным хорошо и что является красивым. В основе их суждений есть нечто более общее, чем сугубо индивидуальные представления о красоте. Похоже, что существуют некоторые трансцендентные образы или шаблоны, являющиеся объективным основанием для оценки предметов. Основная задача культурологии состоит в описании подобных шаблонов, определяющих каноны поведения и систему ценности каждой из культур.²

Исходная посылка создания шаблонов проектирования также состояла в необходимости объективной оценки качества программного обеспечения.

Если идея о том, что оценить и описать высококачественный проект возможно, возражений не вызывает, то не пора ли попробовать создать нечто подобное? Я полагаю, Александер сформулировал для себя следующие вопросы.

1. *Что есть такого в проекте хорошего качества, что отличает его от плохого проекта?*
2. *Что именно отличает проект низкого качества от проекта высокого качества?*

Эти вопросы навели Александера на мысль о том, что если качество проекта является объективной категорией, то мы можем явно определить, что именно делает проекты хорошими, а что — плохими.

Александер изучал эту проблему, обследуя множество зданий, городов, улиц и всего прочего, что люди построили для своего проживания. В результате он обнаружил, что все, что было построено хорошо, имело между собой нечто общее.

Архитектурные структуры отличаются друг от друга, даже если они относятся к одному и тому же типу. Однако, не взирая на имеющиеся различия, они могут оставаться высококачественными.

² Антрополог Рут Бенедикт (Ruth Benedict) была одним из создателей метода анализа культур с помощью выявления существующих в ней шаблонов. Например, см. Benedict R., *The Chrysanthemum and the Sword*, Boston, MA: Houghton Mifflin, 1946.

Например, два крыльца могут выглядеть конструктивно различными и, тем не менее, обладать высоким качеством — просто они решают *различные задачи* в различных зданиях. Одно крыльцо может служить для прохода с тротуара ко входной двери, а назначение другого может состоять в создании тени жарким днем. В другом случае два крыльца могут решать одну и ту же задачу, но *различными способами*.

Александр понял это и пришел к выводу, что сооружения нельзя рассматривать обособленно от проблемы, для решения которой они предназначены. Поэтому в своих попытках найти и описать критерии красоты и качества проекта он стал рассматривать различные архитектурные элементы, предназначенные для решения одинаковых задач. Например, на рис. 5.1 продемонстрированы два различных варианта оформления входа в здание.

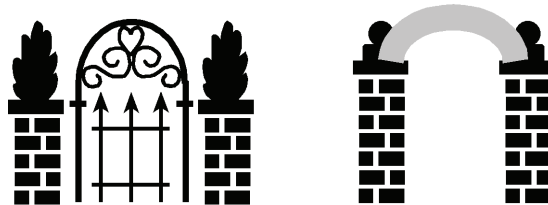


РИС. 5.1. Архитектурные элементы могут выглядеть различными, но выполнять одну функцию

Александр установил, что, сфокусировав внимание на структурах, предназначенных для решения подобных задач, можно обнаружить сходство между различными проектами, которым присуще высокое качество. Он назвал эти сходства *шаблонами*.

Он определил понятие шаблона как "решение проблемы в контексте".³

Каждый шаблон описывает проблему, которая возникает в данной среде снова и снова, а затем предлагает принцип ее решения таким способом, который можно будет применять многократно, получая каждый раз результаты, не повторяющие друг друга.

В табл. 5.1 приведены размышления Александра, взятые мной из его работы⁴. Они иллюстрируют сделанное выше утверждение.

Таблица 5.1. Выдержка из книги "Строительство на века"

Слова Александра	Мой комментарий
Аналогичным образом, правильно спроектированный внутренний двор дома способствует отдыху и восстановлению сил его жителей	Шаблон всегда имеет уникальное имя и предназначен для определенной цели. В данном случае шаблон называется "Внутренний двор", а его назначение состоит в предоставлении жителям дома места для отдыха и восстановления сил

³ Alexander C., Ishikawa S., Silverstein M. The Timeless Way of Building, New York, NY: Oxford University Press, 1979.

⁴ Alexander C., Ishikawa S., Silverstein M. The Timeless Way of Building, New York, NY: Oxford University Press, 1979.

Продолжение таблицы

Слова Александра	Мой комментарий
<p>Рассмотрим, чем занимаются люди во внутреннем дворе. Прежде всего, они ищут возможность уединиться на свежем воздухе и найти такое место, где можно будет посидеть под открытым небом, полюбоваться звездами, насладиться солнечным теплом или посадить цветы. Это совершенно очевидно</p> <p>Но не следует забывать и о других, на первый взгляд, менее важных задачах. В случае, если двор слишком замкнут и не дает необходимого обзора, люди в нем чувствуют себя неуютно. У них возникает желание покинуть это место, поскольку они нуждаются в более широком поле обзора</p> <p>Однако привычка — вторая натура. Когда люди в своей обыденной жизни изо дня в день проходят через внутренний двор множество раз, это место становится для них знакомым, т.е. привычным местом прохода — и именно для этих целей оно будет использоваться ими в дальнейшем</p>	<p>Хотя иногда это может быть вполне очевидно, нам важно четко сформулировать ту основную задачу, которая может быть решена с помощью данного шаблона. Именно это делает Александр при обсуждении шаблона "Внутренний Двор"</p> <p>Здесь подчеркиваются трудности, связанные с упрощенным подходом к решению задачи, а затем предлагается лучший способ решения, позволяющий избежать указанных затруднений</p> <p>Привычка очень часто мешает нам увидеть очевидные вещи. Ценность шаблонов в том, что люди, не обладающие большим опытом, могут воспользоваться преимуществами, найденными их более опытными коллегами. Шаблоны позволяют им определять, какие детали должны быть включены для достижения высокого качества проекта, а также чего следует избегать, чтобы не потерять это качество</p>
<p>Внутренний двор, не имеющий выхода на улицу, становится местом, которое посещают только изредка, когда хочется именно туда. Он остается непривычным местом, которое используется все реже и реже, поскольку люди, в основном, склонны к посещению знакомых им мест.</p> <p>Кроме того, есть что-то неприятное, нежелательное в том, чтобы из дома выйти сразу на улицу. Это вроде бы мелочь, но ее достаточно, чтобы вызвать нервозность или раздражение.</p> <p>Если есть дополнительное пространство в виде крыльца или веранды под навесом, но открытое для воздуха — с психологической точки зрения это промежуточное состояние между домом и улицей. В результате человеку становится легче и проще сделать тот короткий шаг, которые выведет его во внутренний двор</p>	<p>Предлагается решение, способное изменить ваше мнение о преимуществах хорошего внутреннего двора</p>

Окончание таблицы

Слова Александра	Мой комментарий
Если внутренний двор позволяет взглянуть на окружающее пространство, представляет собой удобный путь между различными комнатами, включает в себя некоторый вариант крыльца или веранды, то это наполняет его существование смыслом. Внешний обзор делает посещение внутреннего двора приятным, а пересечение в нем многих путей между помещениями обеспечивает этим посещениям чувство привычности. Наличие калитки или веранды придает дополнительные психологические удобства при выходе из дома на улицу	Александр рассказывает, как построить отличный внутренний двор... ...и поясняет, почему он будет так хорош

В качестве заключения укажем четыре компонента, которые, по мнению Александра, должны присутствовать в описании каждого шаблона.

- Имя шаблона.
- Назначение шаблона и описание задачи которую он призван решать.
- Способ решения поставленной задачи.
- Ограничения и требования, которые необходимо принимать во внимание при решении задачи.

Александр принял как постулат, что с помощью шаблонов может быть решена любая архитектурная задача, с которой столкнется проектировщик. Затем он пошел дальше и высказал утверждение о том, что совместное использование нескольких шаблонов позволит решать комплексные архитектурные проблемы.

Как можно применить одновременно несколько шаблонов, мы обсудим на страницах этой книги позднее. Сейчас же сосредоточимся на пользе отдельных шаблонов при решении специализированных проблем.

Переход от архитектурных шаблонов к шаблонам проектирования программного обеспечения

Но какое отношение может иметь весь этот архитектурный материал к специалистам в области программного обеспечения, коими мы являемся?

В начале 1990-х некоторые из опытных разработчиков программного обеспечения ознакомились с упоминавшейся выше работой Александра об архитектурных шаблонах.

Они задались вопросом, возможно ли применение идеи архитектурных шаблонов при реализации проектов в области создания программного обеспечения.⁵

Сформулируем те вопросы, на которые требовалось получить ответ.

- Существуют ли в области программного обеспечения проблемы, возникающие снова и снова, и могут ли они быть решены тем же способом?
- Возможно ли проектирование программного обеспечения в терминах шаблонов — т.е. создание конкретных решений на основе тех шаблонов, которые будут выявлены в поставленных задачах?

Интуиция подсказывала исследователям, что ответы на оба эти вопроса определенно будут положительными. Следующим шагом необходимо было идентифицировать несколько подобных шаблонов и разработать стандартные методы каталогизации новых.

Хотя в начале 1990-х над шаблонами проектирования работали многие исследователи, наибольшее влияние на это сообщество оказала книга Гаммы, Хелма, Джонсона и Влиссайдеса *Шаблоны проектирования: элементы многократного использования кода в объектно-ориентированном программировании*.⁶ Эта работа приобрела очень широкую известность. Свидетельством ее популярности служит тот факт, что четыре автора книги получили шутовское прозвище "банда четырех".

В книге рассматривается несколько важных аспектов проблемы.

- Применение идеи шаблонов проектирования в области разработки программного обеспечения.
- Описание структур, предназначенных для каталогизации и описания шаблонов проектирования.
- Обсуждение 23 конкретных шаблонов проектирования.
- Формулирование концепций объектно-ориентированных стратегий и подходов, построенных на применении шаблонов проектирования.

Важно понять, что авторы сами не создавали тех шаблонов, которые описаны в их книге. Скорее, они идентифицировали эти шаблоны как уже существующие в разработках, выполненных сообществом создателей программного обеспечения. Каждый из шаблонов отражает те результаты, которые были достигнуты в высококачественных проектах при решении определенных, специфических проблем (этот подход напоминает нам о работе Александера).

На сегодня существует несколько различных форм описания шаблонов проектирования. Поскольку эта книга вовсе не о том, как следует описывать шаблоны проек-

⁵ Консорциум ESPRIT в Европе обратился к этому же вопросу еще в 1980-х. Проекты ESPRIT с номерами 1098 и 5248 состояли в разработке методологии проектирования с использованием шаблонов, получившей название KADS (Knowledge Analysis and Support – анализ и поддержка знаний). Эта методология предусматривала использование шаблонов при создании экспертных систем. Карен Гарднер (Karen Gardner) распространила аналитические шаблоны KADS на объектную технологию. См. Gardner K. *Cognitive Patterns: Problem Solving Frameworks for Object Technology*, New York, NY: Cambridge University Press, 1998.

⁶ Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA: Addison-Wesley, 1995.

тирования, мы не приводим здесь нашего мнения по поводу того, какая из структур описания шаблонов является оптимальной. Однако пункты, приведенные в табл. 5.2, безусловно, должны присутствовать в любом описании шаблона проектирования.

Каждый шаблон, упоминаемый в этой книге, будет представлен в виде краткого резюме, описывающего его важнейшие характеристики.

Таблица 5.2. Важнейшие характеристики шаблонов

Характеристика	Описание
Имя	Все шаблоны имеют уникальное имя, служащее для их идентификации
Назначение	Назначение данного шаблона
Задача	Задача, которую шаблон позволяет решить
Способ решения	Способ, предлагаемый в шаблоне для решения задачи в том контексте, где этот шаблон был найден
Участники	Сущности, принимающие участие в решении задачи
Следствия	Последствия от использования шаблона как результат действий, выполняемых в шаблоне
Реализация	Возможный вариант реализации шаблона. <i>Замечание.</i> Реализация является лишь конкретным воплощением общей идеи шаблона и не должна пониматься как собственно сам шаблон
Ссылки	Место в книге "банды четырех", где можно найти дополнительную информацию

Следствия и действия

Следствия — это термин, который широко используется в отношении шаблонов проектирования, но часто неправильно истолковывается. В применении к шаблонам проектирования под следствием понимается любой эффект, вызванный какой-либо причиной в процессе его функционирования. Иначе говоря, если шаблон реализован так-то и так-то, то следствиями его использования будут результаты любых выполняемых в нем действий.

Зачем нужно изучать шаблоны проектирования

Теперь, когда мы знаем, что такое шаблоны проектирования, можно попытаться ответить на вопрос, зачем нужно их изучать. На то имеется несколько причин, часть из которых вполне очевидна, тогда как об остальных этого не скажешь.

Чаще всего причины, по которым следует изучать шаблоны проектирования, формулируют следующим образом.

- *Возможность многократного использования.* Повторное использование решений из уже завершенных успешных проектов позволяет быстро приступить к решению новых проблем и избежать типичных ошибок. Разработчик получает прямую выгоду от использования опыта других разработчиков, избежав необходимости вновь и вновь изобретать велосипед.



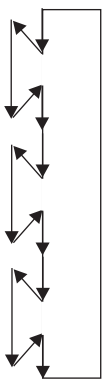
- *Применение единой терминологии.* Профессиональное общение и работа в группе (команде разработчиков) требует наличия единого базового словаря и единой точки зрения на проблему. Шаблоны проектирования предоставляют подобную общую точку зрения как на этапе анализа, так и при реализации проекта.

Однако существует и третья причина, по которой следует изучать шаблоны проектирования.

Шаблоны проектирования предоставляют нам абстрактный высокоуровневый взгляд как на проблему, так и на весь процесс объектно-ориентированной разработки. Это помогает избежать излишней детализации на ранних стадиях проектирования.

Я надеюсь, что, прочитав эту книгу, вы согласитесь с тем, что именно последняя причина является важнейшей для изучения шаблонов проектирования. Я постараюсь изменить ваш образ мышления за счет усиления и развития у вас навыков системного анализа.

Для иллюстрации сказанного выше представим себе разговор двух плотников о том, как сделать выдвижные ящики для письменного стола.⁷

Второй плотник говорит...	Как это выглядит...
“Ну, я думаю, соединение будет прочным, если сначала сделать пропил прямо поперек доски, а затем повернуть обратно под углом 45 градусов...”	
“...после чего снова углубиться в дерево, а затем повернуть назад под 45 градусов, но уже в другую сторону, потом вновь пилить поперек доски, а затем...”	
“...и так до тех пор, пока не получится соединение ласточкин хвост. Именно его изготовление я только что описывал тебе!”	

⁷ Этот диалог взят автором из рассказа Ральфа Джонсона (Ralph Johnson) и соответствующим образом переработан.

Плотник 1. Как, по твоему мнению, следует сделать эти ящики?

Плотник 2. Ну, я думаю, соединение будет прочным, если сначала сделать пропил прямо поперек доски, а затем повернуть обратно под углом 45 градусов, после чего снова углубиться в дерево, а затем повернуть назад под углом 45 градусов, но уже в другую сторону, потом вновь пилить поперек доски, а затем...

Попробуйте понять, о чем здесь идет речь!

Не правда ли, довольно запутанное описание? Что именно второй плотник имеет в виду? Чрезмерное количество подробностей мешает уловить смысл обсуждаемого. Попробуем представить это описание графически.

Не напоминает ли это вам устное описание фрагмента программного кода? Вероятно, программист сказал бы что-то, похожее на следующее:

...а затем я использую цикл WHILE, чтобы выполнить..., за которым следует несколько операторов IF, предназначенных для..., а потом я использую оператор SWITCH для обработки...

Здесь представлено подробное описание фрагмента программы, но из него остается абсолютно неясным, *что эта программа делает и для чего!*

Конечно, никакой уважающий себя плотник не стал бы говорить ничего подобного тому, что было приведено выше.

В действительности мы услышали бы примерно такой диалог.

Плотник 1. Какой вариант соединения мы будем использовать, — "ласточкин хвост" или простое угловое соединение под 45 градусов?

Нетрудно заметить, что новый вариант качественно отличается от прежнего. Плотники обсуждают отличия в качестве решения проблемы, поэтому в данном случае их общение проходит на более высоком, а значит, и более абстрактном уровне. Они не тратят время и силы на обсуждение специфических деталей отдельных типов соединений и не вдаются в излишние подробности их описания.

Когда плотник говорит о простом соединении деревянных деталей под углом 45 градусов, он понимает, что такому решению свойственны следующие характеристики.

- *Это более простое соединение.* Угловое соединение является более простым в изготовлении. Достаточно обрезать торцы соединяемых деталей под углом в 45 градусов, состыковать их, а затем скрепить между собой с помощью гвоздей или клея — как показано на рис. 5.2.
- *Это соединение имеет меньшую прочность.* Угловое соединение является менее прочным, чем соединение "ласточкин хвост", и не способно выдерживать большую нагрузку.
- *Это более незаметное соединение.* Единственный стык в месте соединения деталей меньше заметен глазу, чем многочисленные распилы в соединении типа "ласточкин хвост".



РИС. 5.2. Простое соединение под углом 45 градусов

Когда плотник говорит о соединении типа "ласточкин хвост" (способ выполнения которого был описан выше), он представляет себе его характеристики, которые могут быть вовсе не очевидны для обывателя, но, безусловно, известны любому другому плотнику.

- *Это более сложный вариант.* Выполнить это соединение сложнее, чем простое угловое, поэтому оно дороже.
- *Это соединение устойчиво по отношению к температуре и влажности.* При изменении температуры и влажности воздуха древесина расширяется или сжимается, однако это не оказывает влияния на прочность соединения данного типа.
- *Это соединение не нуждается в дополнительном укреплении.* Благодаря точному соответствию вырезов скрепляемых деталей по отношению друг к другу соединение фактически не нуждается в дополнительном укреплении клеем или гвоздями.
- *Это соединение более эстетично.* Если соединение этого типа выполнено качественно, оно оставляет очень приятное впечатление.

Другими словами, соединение типа "ласточкин хвост" надежно и красиво, но менее технологично, а поэтому изготовление его обходится дороже.

Таким образом, когда первый плотник задает вопрос:

Какой способ соединения мы будем использовать — типа "ласточкин хвост" или обычное, под углом в 45 градусов?

В действительности он спрашивает:

Мы воспользуемся более сложным и дорогим, но красивым и долговечным соединением или сделаем все быстро, не заботясь о высоком качестве, лишь бы ящик простоял до тех пор, пока нам не заплатят?

Можно сказать, что в действительности общение плотников происходит на двух уровнях: внешнем уровне произносимых слов и более высоком уровне (мета-уровне) *реальной* беседы. Последний скрыт от обывателя и намного богаче первого по содержанию. Этот более высокий уровень — назовем его уровнем "шаблонов плотников" — отражает реальные вопросы разработки проекта в нашем примере с плотниками.

В первом варианте второй плотник затеняет реально обсуждаемые проблемы, углубляясь в детали выполнения соединения. Во втором варианте первый плотник предлагает принять решение о выборе типа соединения, исходя из его стоимости и качества.

Чей подход более эффективен? С кем, по вашему мнению, предпочтительнее работать?

Шаблоны позволяют нам видеть лес за деревьями, поскольку помогают поднять уровень мышления. Ниже в этой книге будет показано, что, когда удастся подняться на более высокий уровень мышления, разработчику становятся доступны новые методы проектирования. Именно в этом состоит реальная сила шаблонов проектирования.

Другие преимущества применения шаблонов проектирования

Мой опыт работы в группах разработчиков показал, что в результате применения шаблонов проектирования повышается эффективность труда отдельных исполнителей и всей группы в целом. Это происходит из-за того, что начинающие члены группы видят на примере более опытных разработчиков, как шаблоны проектирования могут применяться и какую пользу они приносят. Совместная работа дает новичкам стимул и реальную возможность быстрее изучить и освоить эти новые концепции.

Применение многих шаблонов проектирования позволяет также создавать более модифицируемое и гибкое программное обеспечение. Причина состоит в том, что эти решения уже испытаны временем. Поэтому использование шаблонов позволяет создавать структуры, допускающие их модификацию в большей степени, чем это возможно в случае решения, первым пришедшего на ум.

Шаблоны проектирования, изученные должным образом, существенно помогают общему пониманию основных принципов объектно-ориентированного проектирования. Я убеждался в этом неоднократно в процессе преподавания курса объектно-ориентированного проектирования. Курс я начинал с краткого представления объектно-ориентированной парадигмы, а затем переходил к освещению темы шаблонов проектирования, используя основные концепции ООП (инкапсуляция, наследование и полиморфизм) лишь для иллюстрации излагаемого материала. К концу трехдневного курса, хотя речь на лекциях шла главным образом о шаблонах, базовые концепции ООП также становились понятными и знакомыми всем слушателям.

"Бандой четырех" было предложено несколько стратегий создания хорошего объектно-ориентированного проекта. В частности, они предложили следующее.

- Проектирование согласно интерфейсам.
- Предпочтение синтеза наследованию.
- Выявление изменяющихся величин и их инкапсуляция.

Эти стратегии использовались в большинстве шаблонов проектирования, обсуждаемых в данной книге. Для оценки полезности указанных стратегий вовсе не обязательно изучать большое количество шаблонов — достаточно всего нескольких. Приобретенный опыт позволит применять новые концепции и к задачам собственных проектов, даже без непосредственного использования шаблонов проектирования.

Еще одно преимущество состоит в том, что шаблоны проектирования позволяют разработчику или группе разработчиков находить проектные решения для сложных проблем, не создавая громоздкой иерархии наследования классов. Даже если шаблоны не используются в проекте непосредственно, одно только уменьшение размера иерархий наследования классов уже будет способствовать повышению качества проекта.

Резюме

В этой главе мы выяснили, что представляют собой шаблоны проектирования. Кристофер Александер сказал: "Шаблоны — это решение проблемы в контексте". Шаблон проектирования это нечто большее, чем просто способ решения какой-либо проблемы. Шаблон проектирования — это способ представления в общем виде как условия задачи, которую необходимо решить, так и правильных подходов к ее решению.

Мы также рассмотрели причины для изучения шаблонов проектирования. Шаблоны могут помочь разработчику в следующем.

- Многократно применять высококачественное решение для повторяющихся задач.
- Ввести общую терминологию для расширения взаимопонимания в пределах группы разработчиков.
- Поднять уровень, на котором проблема решается, и избежать нежелательного углубления в детали реализации уже на ранних этапах разработки.
- Оценить, что было создано — именно то, что нужно, или же просто некоторое работоспособное решение.
- Ускорить профессиональное развитие как всей группы разработчиков в целом, так и отдельных ее членов.
- Повысить модифицируемость кода.
- Обеспечить выбор лучших вариантов реализации проекта, даже если сами шаблоны проектирования в нем не используются.
- Найти альтернативное решение для исключения громоздких иерархий наследования классов.

Шаблон Facade

Введение

Изучение шаблонов проектирования мы начнем с шаблона, который вам, возможно, уже приходилось применять на практике ранее, но вы не представляли себе, что эта конструкция может иметь специальное название — шаблон проектирования Facade.

В этой главе мы выполним следующее.

- Выясним, что представляет собой шаблон проектирования Facade и где он может использоваться.
- Обсудим ключевые особенности этого шаблона.
- Познакомимся с некоторыми разновидностями шаблона Facade.
- Применим шаблон проектирования Facade к задаче о САПР.

Назначение шаблона проектирования Facade

В книге "банды четырех" назначение шаблона Facade (фасад) определяется следующим образом.

Предоставление единого интерфейса для набора различных интерфейсов в системе. Шаблон Facade определяет интерфейс более высокого уровня, что упрощает работу с системой.¹

В основном этот шаблон используется в тех случаях, когда необходим новый способ взаимодействия с системой — более простой в сравнении с уже существующим. Кроме того, он может применяться, когда требуется использовать систему некоторым специфическим образом — например, обращаться к программе трехмерной графики для построения двумерных изображений. В этом случае нам потребуется специальный метод взаимодействия с системой, поскольку будет использоваться лишь часть ее функциональных возможностей.

Описание шаблона проектирования Facade

В моей практике был случай, когда я работал по контракту для большой инженерно-производственной компании. В день моего первого появления на новом

¹ Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software, Reading, MA: Addison-Wesley, 1995, с. 185.

рабочем месте непосредственный руководитель проекта отсутствовал. Компания отказывалась оплатить этот рабочий день просто так, но никто не знал, какое задание мне следует поручить. Они хотели, чтобы я делал хоть что-нибудь, даже если это не принесло бы никакой пользы! Думаю, вам тоже приходилось сталкиваться с чем-либо подобным.

В конце концов, одна из сотрудниц нашла для меня занятие. Она сказала: "Полагаю, что рано или поздно, но вам обязательно потребуется изучить ту САПР, которая используется у нас в настоящее время, — так что вы можете приступить к этому прямо сейчас". После этих слов девушка указала мне на кипу документации, занимавшую на книжной полке почти 3 метра. Книги были напечатаны мелким шрифтом на листах формата А4, и вся эта гора бумаги представляла собой техническое описание одной-единственной, но очень сложной системы (рис. 6.1)!



РИС. 6.1. Множество томов документации — техническое описание одной сложной системы!

Если четыре или пять человек работают с подобной сложной системой, едва ли есть насущная необходимость каждому из них изучить особенности ее функционирования во всех подробностях. Вместо того, чтобы тратить понапрасну время каждого работника группы, лучше будет бросить жребий и поручить *проигравшему* написать подпрограммы, обеспечивающие остальным сотрудникам единый интерфейс для работы с системой.

Тот, кому поручат это задание, должен будет определить, как другие члены группы намереваются использовать систему и какой API (прикладной программный интерфейс) будет оптимальным. Затем ему потребуется определить новый класс (или классы), предоставляющий требуемый интерфейс. Это позволит остальным членам группы просто обращаться к данному интерфейсу, вместо того, чтобы тратить время на подробное изучение сложной системы (рис. 6.2).

Этот подход применяется, когда используется только часть всех возможностей системы или когда взаимодействие с ней осуществляется некоторым специфическим образом. Если же необходимо использовать абсолютно все компоненты и функциональные возможности системы, то маловероятно, что существует какая-либо возмож-

ность создать для нее упрощенный интерфейс (если только разработчики системы не отнеслись к своей работе халатно).

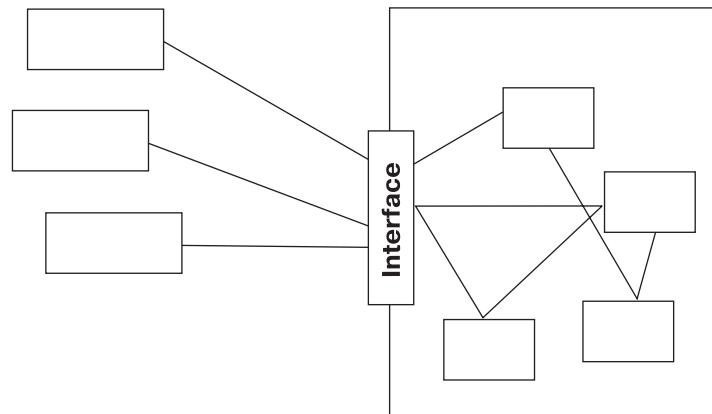


РИС. 6.2. *Изоляция клиентов от внутренних подсистем*

Это и есть шаблон проектирования Facade. Он упрощает использование сложной системы, отдельной части системы или обеспечивает обращение к системе некоторым специфическим образом. Мы имеем сложную систему, и нам требуется использовать только какую-то ее часть (отдельный модуль). В результате применения шаблона Facade мы получим новую, более простую в использовании систему, которая будет точно соответствовать нашим потребностям.

Основная часть работы по-прежнему будет выполняться исходной системой. Шаблон Facade предоставляет лишь коллекцию методов, простых в понимании и использовании. Эти методы обращаются к основной системе для реализации вновь определенных функций внешней системы.

Основные характеристики шаблона Facade

Назначение	Упростить работу с существующей системой, определив собственный интерфейс обращения к ней
Задача	Необходимо использовать только определенное подмножество функций сложной системы или организовать взаимодействие с ней некоторым специфическим образом
Способ решения	Шаблон Facade предоставляет клиентам новый интерфейс для взаимодействия с уже существующей системой
Участники	Клиенту предоставляется специализированный интерфейс, упрощающий работу с системой
Следствия	Применение шаблона Facade упрощает использование требуемой подсистемы, но одновременно лишает пользователя доступа ко всем функциональным возможностям системы — часть их окажется недоступной
Реализация	<ul style="list-style-type: none"> • Определение нового класса (или классов) с требуемым интерфейсом. • В своей работе новый класс должен опираться на функциональные возможности существующей системы (рис. 6.3)

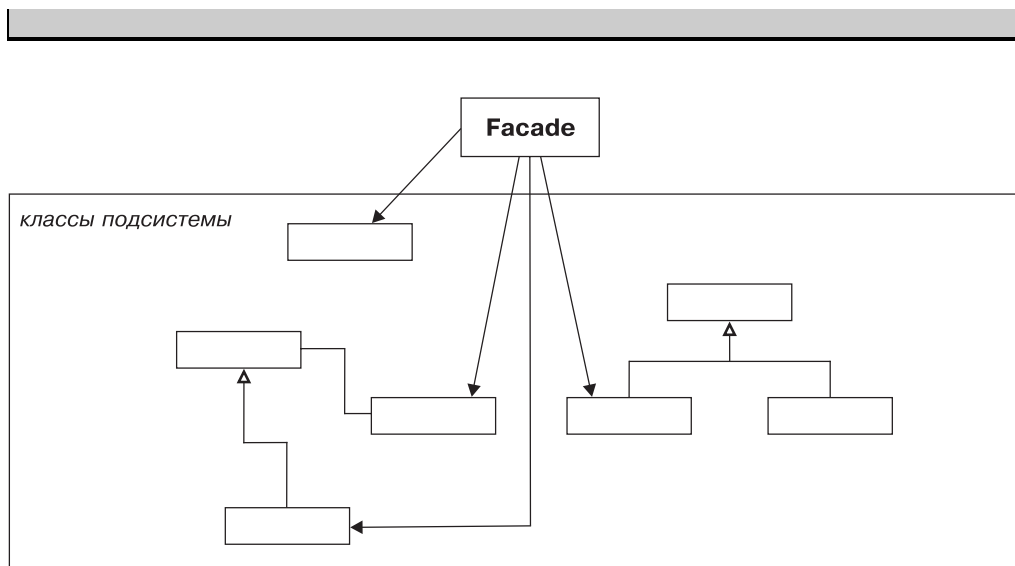


РИС. 6.3. *Стандартное упрощенное представление структуры шаблона проектирования Facade*

Шаблон Facade может применяться не только для создания упрощенного интерфейса с целью вызова методов, но и для уменьшения количества объектов, с которыми клиенту приходится иметь дело. Например, предположим, что существует объект **Client** (клиент), которому необходимо взаимодействовать с объектами **Database** (база данных), **Model** (модель) и **Element** (элемент). Объект **Client** должен будет сначала открыть объект **Database** и получить доступ к объекту **Model**. Затем он должен будет направить объекту **Model** запрос на получение доступа к объекту **Element**. И только после этого он сможет получить от объекта **Element** требуемую информацию. Однако более простым решением будет создать объект **Database Facade**, которому любой объект **Client** сможет направить один запрос непосредственно на выборку информации — как показано на рис. 6.4.

Предположим, что помимо использования функций, реализованных в базовой системе, необходимо предоставить пользователям и некоторую новую функциональность. В этом случае проблема выходит за рамки простого использования некоторого подмножества функциональных возможностей существующей системы.

Необходимые дополнения системы могут быть реализованы за счет добавления в класс **Facade** новых методов, обеспечивающих требуемую функциональность. В этом случае мы по-прежнему имеем дело с шаблоном Facade, но этот шаблон будет расширен новыми функциональными возможностями.

Шаблон Facade определяет общий подход, позволяя нам целенаправленно приступить к работе. Со стороны клиента шаблон фактически создает для него новый интерфейс, которым клиент будет пользоваться вместо уже существующего стандартно-

го интерфейса системы. Это оказывается возможным потому, что объект **Client** не нуждается в использовании всех функций основной системы.

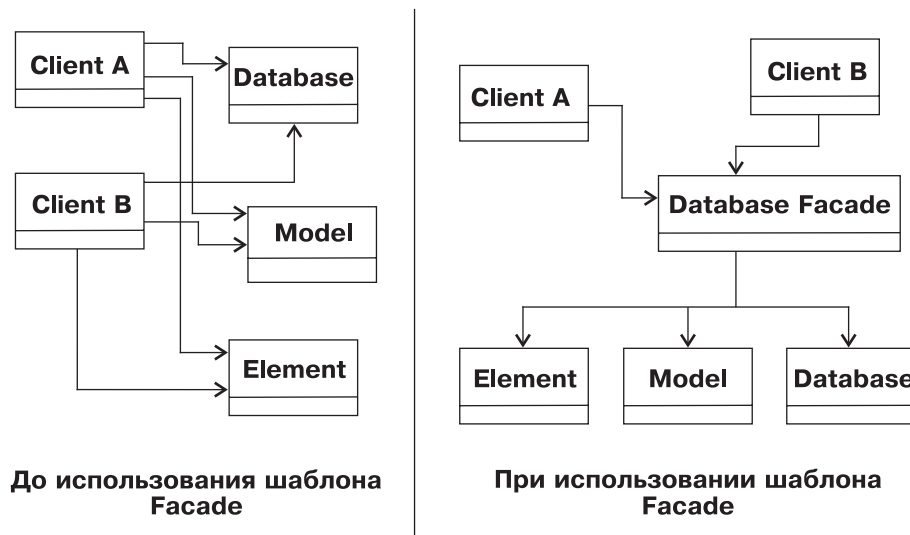


РИС. 6.4. Шаблон Facade позволяет уменьшить количество объектов, видимых клиенту

Шаблоны определяют лишь общий подход

Любой шаблон проектирования определяет лишь общий подход к решению задачи. Потребуется или нет добавлять новые функции, зависит от конкретной ситуации. Шаблоны проектирования в сущности только наброски или эскизы, показывающие, как правильно приступить к работе. Не следует представлять их себе некими догматами, высеченными на камне.

Шаблон Facade также может использоваться для сокрытия или инкапсуляции базовой системы. В этом случае класс **Facade** включает основную систему как свой закрытый член. Поэтому основная система будет взаимодействовать только с классом **Facade**, оставаясь недоступной и невидимой для всех остальных пользователей.

Имеется несколько причин для инкапсуляции основной системы.

- *Контроль за использованием системы.* Вынуждая пользователей все обращения к системе выполнять только через класс **Facade**, легко можно контролировать их доступ к ней.
- *Упрощение замены системы.* В будущем может потребоваться заменить существующую систему новой. Представление базовой системы как закрытого члена класса **Facade** существенно упрощает эту процедуру. Хотя объем необходимых изменений может оказаться достаточно большим, все они будут сосредоточены только в одном месте программного кода — в классе **Facade**.

Применение шаблона Facade к проблеме САПР

Что касается проблемы одновременной работы с различными версиями САПР, то шаблон проектирования Facade может оказаться полезным при организации работы объектов **VisSlots**, **VisHoles** и им подобных с объектом **VisSystem**. Подробнее мы обсудим это в главе 12, *Решение задачи САПР с помощью шаблонов проектирования*.

Резюме

Шаблон проектирования Facade назван так потому, что он как бы устанавливает новый фасад (интерфейс) для основной системы.

Шаблон Facade применяется в следующих случаях.

- Когда нет необходимости использовать все функциональные возможности сложной системы и можно создать новый класс, который будет содержать все необходимые средства доступа к базовой системе. Если предполагается работа лишь с ограниченным набором функций исходной системы, как это обычно и бывает, интерфейс (API), описанный в новом классе, будет намного проще, чем стандартный интерфейс, разработанный создателями основной системы.
- Существует необходимость в инкапсуляции первоначальной системы.
- Если требуется не только использовать существующие функциональные возможности базовой системы, но и дополнить их некоторой новой функциональностью.
- Стоимость разработки нового класса меньше стоимости обучения всех участников проекта работе с базовой системой или суммы будущих затрат на сопровождение создаваемой ими системы.

Шаблон Adapter

Введение

В этой главе мы продолжим изучение шаблонов проектирования и рассмотрим шаблон Adapter. Это шаблон общего характера и, как мы увидим позднее, часто используется в комбинации с другими шаблонами.

Здесь мы выполним следующее.

- Выясним, что собой представляет шаблон Adapter, где он применяется и каким образом реализуется.
- Рассмотрим ключевые особенности этого шаблона.
- Воспользуемся шаблоном Adapter для иллюстрации понятия полиморфизма.
- Обсудим, как язык UML может использоваться на различных уровнях детализации.
- Проанализируем некоторые ситуации из практики автора, связанные с применением шаблона Adapter, а так же сравним шаблоны Adapter и Facade.
- Применим шаблон Adapter к решению задачи о различных версиях САПР.

Замечание. В тексте главы примеры программного кода приведены на языке Java. Соответствующие примеры кода на языке C++ можно найти в конце главы (листинг 7.2).

Назначение шаблона проектирования Adapter

"Банда четырех" так определяет назначение шаблона Adapter.

Преобразование стандартного интерфейса класса в интерфейс, более подходящий для нужд клиента. Применение шаблона Adapter позволяет организовать совместную работу классов с несовместимыми интерфейсами.¹

Другими словами, данный шаблон предполагает создание нового интерфейса для требуемого объекта, который мы не можем использовать из-за его неподходящего интерфейса.

¹ Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software, Reading, MA: Addison-Wesley, 1995, с. 185.

Описание шаблона проектирования Adapter

Самый простой способ понять назначение шаблона Adapter — это рассмотреть его применение на примере. Допустим, существуют такие требования.

- Создать классы для представления в системе точек, линий и квадратов, каждый из которых будет иметь метод `display` (отобразить).
- Используя их объекты не должны знать, с каким именно элементом они имеют дело в действительности — с точкой, линией или квадратом. Клиентским объектам достаточно знать, что они получили доступ к объекту, представляющему одну из указанных фигур.

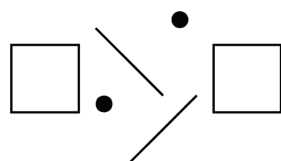
Другими словами, необходимо представить эти конкретные фигуры с помощью концепции более высокого порядка — назовем ее *изображаемой фигурой*.

Теперь расширим наш пример, представив себе еще одну близкую ситуацию.

- Необходимо использовать подпрограмму или метод, написанный кем-то другим, поскольку в нем реализованы именно те функции, которые нам требуются.
- При этом включить готовую подпрограмму непосредственно в создаваемую программу невозможно.
- Интерфейс подпрограммы или способ ее вызова в коде не отвечают тем условиям, в которых она будет использоваться.

Иначе говоря, хотя в системе представлены точки, линии и квадраты, нам нужно, чтобы все *выглядело* так, как будто в ней существуют только некие *абстрактные фигуры*.

- Это позволит объектам-клиентам работать с любыми типами объектов-данных одним и тем же образом, не принимая во внимание существующие между ними различия (рис. 7.1).
- Кроме того, появляется возможность впоследствии добавлять в систему новые виды фигур, не внося никаких изменений в те объекты, которые будут их использовать.



То, с чем мы имеем дело
(точки, линии, квадраты)



То, что видит пользователь
(фигуры)

РИС. 7.1. Все объекты, с которыми мы будем работать,... должны выглядеть одинаково — как абстрактные фигуры

Здесь используется принцип полиморфизма; т.е. в системе присутствуют объекты-данные разных типов, но использующие их объекты-клиенты должны взаимодействовать с ними одним и тем же образом.

В результате объект-клиент может просто указать объекту-данному (независимо от того, представляет он точку, линию или квадрат), что необходимо выполнить то или

иное действие — например, отобразить себя на экране или, наоборот, удалить с экрана свое изображение. В этом случае каждый объект, представляющий точку, линию или квадрат, должен будет нести полную ответственность за корректное выполнение требуемых операций в полном соответствии со своим типом.

Для решения поставленной задачи создадим класс **Shape** (фигура), а затем определим производные от него классы, представляющие точки (**Point**), линии (**Line**) и квадраты (**Square**) — как показано на рис. 7.2.

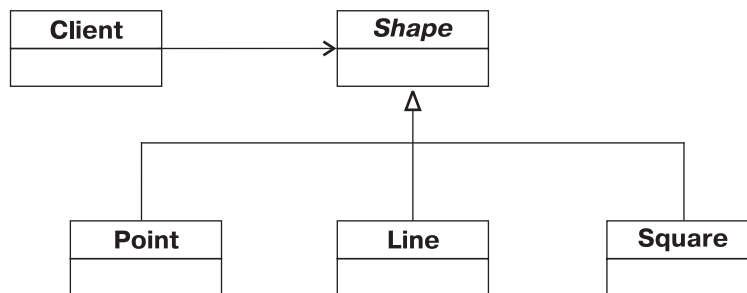


РИС. 7.2. Классы *Point*, *Line* и *Square* представляют собой разновидности класса *Shape*²

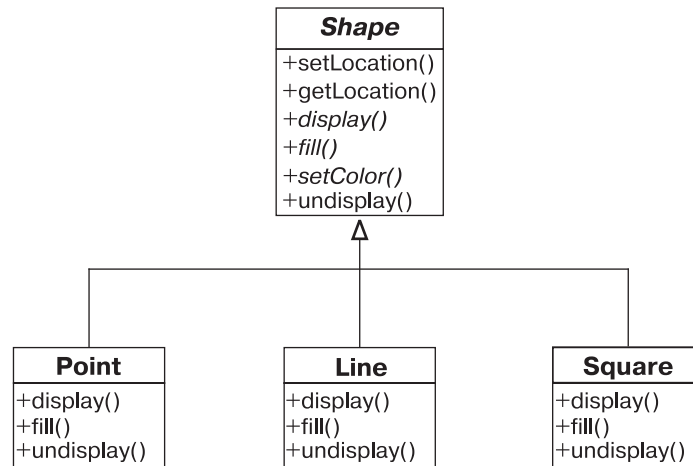
Прежде всего необходимо определить специфическое поведение, которое должен демонстрировать класс **Shape**. Для решения этой задачи следует описать в нем интерфейс вызова методов, ответственных за поведение, а затем реализовать эти методы в каждом из порожденных классов.

Поведение, которое должен демонстрировать класс **Shape**, предусматривает следующие методы (рис. 7.3).

- Получить данные о положении объекта **Shape** (метод `setLocation`).
- Сообщить данные о положении объекта **Shape** (метод `getLocation`).
- Отобразить представленную объектом фигуру на экране (метод `display`).
- Закрасить изображение фигуры указанным цветом (метод `fill`).
- Установить цвет закрашивания фигуры (метод `setColor`).
- Удалить изображение фигуры с экрана (метод `undisplay`).

Предположим, что в систему необходимо включить новый тип объектов класса **Shape**, предназначенный для представления окружностей (не забывайте, что требования постоянно изменяются!). Для этой цели создадим новый класс, **Circle** (окружность), который будет представлять в системе окружности. Реализуем класс **Circle** как производный от класса **Shape**, что позволит воспользоваться преимуществами его полиморфного поведения.

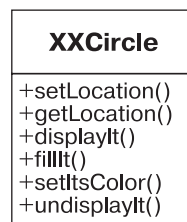
² Эта и все другие диаграммы классов, приведенные в данной книге, выполнены с использованием нотации языка UML. Описание языка UML см. в главе 2, UML – унифицированный язык моделирования.

РИС. 7.3. Методы классов *Shape*, *Point*, *Line* и *Square*

Теперь необходимо написать методы `display`, `fill` и `undisplay` для класса **Circle**. Эта задача может оказаться достаточно сложной.

К счастью, после недолгих поисков альтернативных решений (что всегда должен делать каждый хороший программист), выяснилось, что один из моих коллег уже описал в системе класс **XXCircle**, предназначенный для работы с окружностями (рис. 7.4). К сожалению, он предварительно не посоветовался со мной, как лучше называть методы этого класса, поэтому они получили следующие имена:

- `displayIt`
- `fillIt`
- `undisplayIt`

РИС. 7.4. Класс *XXCircle*

В результате, непосредственно использовать класс **XXCircle** нельзя, поскольку желательно сохранить полиморфное поведение, реализованное в классе **Shape**, но этому препятствуют следующие моменты.

- Класс **Shape** и класс **XXCircle** включают методы с разными именами и различными списками параметров.
- Класс **XXCircle** должен не только иметь совпадающие имена методов, но и обязательно являться производным от класса **Shape**.

Маловероятно, что коллега согласится на изменение имен методов и вывод класса **XXCircle** из класса **Shape**. Дав такое согласие, он вынужден будет внести соответствующие изменения в код всех объектов в системе, которые так или иначе взаимодействуют с классом **XXCircle**. Кроме того, изменение программного кода, созданного другим разработчиком, обычно чревато появлением непредвиденных побочных эффектов.

В результате получается, что использовать готовый класс нельзя, а повторно выполнять всю работу по созданию требуемого класса с нуля нежелательно. Что же делать?

Можно создать новый класс, который действительно будет порожден от класса **Shape** и, следовательно, содержать реализацию интерфейса, описанного в этом классе, но не будет включать методы работы с окружностями, реализованные в классе **XXCircle**. Образуется следующая структура, представленная на рис. 7.5.

Класс **Circle**:

- является производным от класса **Shape**;
- включает класс **XXCircle**;
- переадресует сделанные к нему запросы объекту класса **XXCircle**.

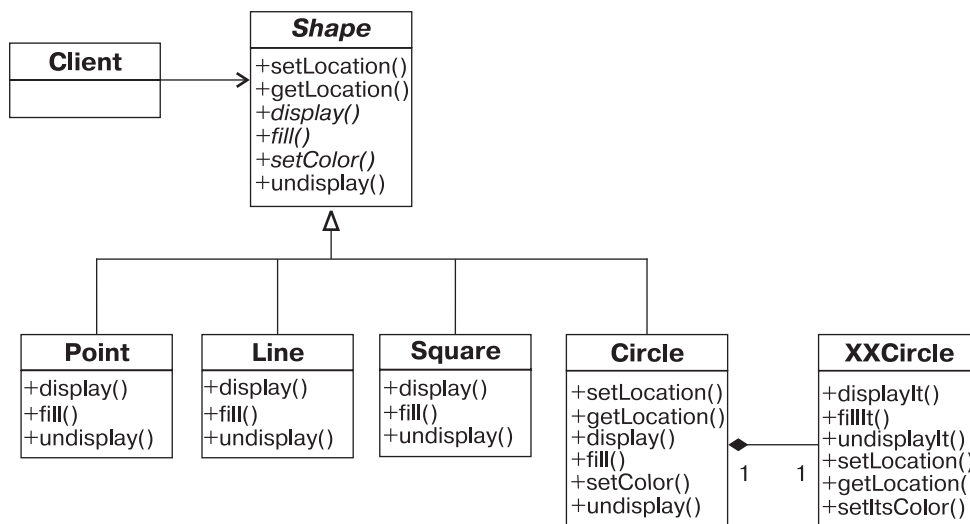


РИС. 7.5. Пример использования шаблона Adapter – класс **Circle** служит оболочкой для класса **XXCircle**

Закрашенный ромб на конце линии, соединяющей классы **Circle** и **XXCircle** (см. рис. 7.5), означает, что класс **Circle** содержит класс **XXCircle**. При создании экземпляра объекта класса **Circle** необходимо будет создать и соответствующий экземпляр объекта класса **XXCircle**. Запросы к объекту **Circle** на выполнение любых действий будут просто передаваться объекту **XXCircle**. Если реализовать все это надлежащим образом, и если объект **XXCircle** будет включать все функциональные возможности, которые должен поддерживать объект **Circle** (чуть позже мы обсудим, что произойдет, если это не так), то объект **Circle** сможет продемонстрировать

нужное поведение, просто возложив на объект **XXCircle** обязанности по выполнению всей необходимой работы.

Пример создания класса-оболочки приведен в листинге 7.1.

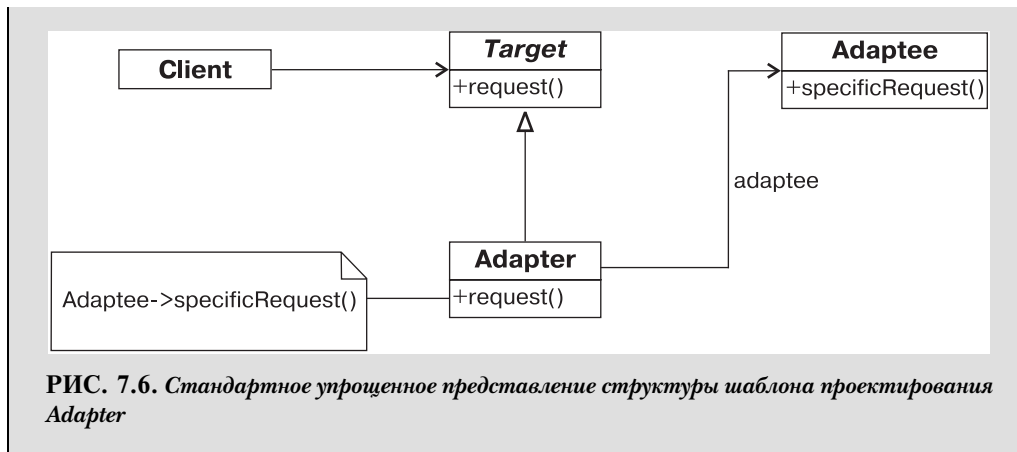
Листинг 7.1. Фрагмент реализации шаблона Adapter на языке Java

```
class Circle extends Shape {
    ...
    private XXCircle pxc;
    ...
    public Circle () {
        pxc= new XXCircle();
    }
    void public display() {
        pxc.displayIt();
    }
}
```

Применение шаблона Adapter позволило сохранить все преимущества полиморфизма при работе с классом **Shape**. Другими словами, объекты, обращающиеся к объекту **Shape**, не знают, какой именно объект в действительности обрабатывает их запросы. Это также пример нового осмысления понятия *инкапсуляция*. Класс **Shape** инкапсулирует в себе представляемые им конкретные фигуры. Шаблон Adapter обычно используется для реализации в системе полиморфизма. Как будет показано в последующих главах, он чаще всего применяется для обеспечения полиморфизма, необходимого другим шаблонам проектирования.

Основные характеристики шаблона Adapter

Назначение	Организовать использование функций объекта, недоступного для модификации, через специально созданный интерфейс
Задача	Система поддерживает требуемые данные и поведение, но имеет неподходящий интерфейс. Чаще всего шаблон Adapter применяется, если необходимо создать класс, производный от вновь определяемого или уже существующего абстрактного класса
Способ решения	Шаблон Adapter предусматривает создание класса-оболочки с требуемым интерфейсом
Участники	Класс Adapter приводит интерфейс класса Adaptee в соответствие с интерфейсом класса Target (от которого класс Adapter является производным). Это позволяет объекту Client использовать объект Adaptee так, словно он является экземпляром класса Target (рис. 7.6)
Следствия	Шаблон Adapter позволяет включать уже существующие объекты в новые объектные структуры, независимо от различий в их интерфейсах
Реализация	Включение уже существующего класса в другой класс. Интерфейс включающего класса приводится в соответствие с новыми требованиями, а вызовы его методов преобразуются в вызовы методов включенного класса



Дополнительные замечания о шаблоне Adapter

Часто встречаются ситуации, подобные описанной выше, но при этом повторно используемый объект не обладает всей требуемой функциональностью.

В этом случае также можно применить шаблон Adapter, хотя это и не позволит сразу получить готовое решение. Правильный подход в подобной ситуации может быть таким.

- Функции, реализованные в уже существующем классе, просто адаптируются по описанной выше схеме.
- Отсутствующие функции реализуются заново — непосредственно в объекте-оболочке.

Это менее эффективное решение, но оно все же позволяет избежать реализации всей требуемой функциональности.

Шаблон Adapter позволяет в процессе проектирования не принимать во внимание возможных различий в интерфейсах уже существующих классов. Если имеется класс, обладающий требуемыми методами и свойствами, — по крайней мере, концептуально, то при необходимости всегда можно будет воспользоваться шаблоном Adapter для приведения его интерфейсов к нужному виду.

Эта особенно важно, если применяется сразу несколько шаблонов, поскольку многие шаблоны требуют, чтобы используемые в них классы были порождены от одного и того же класса. Если в проект предстоит включить уже существующие классы, шаблон Adapter может использоваться для адаптации их к требуемому абстрактному классу (подобно тому, как класс **XXCircle** был адаптирован к абстрактному классу **Shape** с помощью класса-оболочки **Circle**).

Фактически существует два типа шаблонов Adapter.

- **Объектный шаблон Adapter.** Тот вариант шаблона Adapter, который мы обсуждали выше, называют *объектным шаблоном Adapter*, поскольку он реализуется посредством помещения одного объекта (адаптируемого) в другой (адаптирующий).

- **Классовый шаблон Adapter.** Второй вариант реализации шаблона Adapter использует механизм множественного наследования и получил название *классового шаблона Adapter*.

Выбор варианта шаблона Adapter зависит от специфики проблемной области. На концептуальном уровне различия можно игнорировать, однако на этапе реализации проекта потребуется принять во внимание некоторые дополнительные аспекты.³

На моих лекциях по изучению шаблонов проектирования почти всегда находится какой-либо слушатель, который заявляет, что не видит разницы между шаблонами Adapter и Facade. В обоих случаях имеется уже существовавший ранее класс (или классы), не обладающий требуемым интерфейсом, и в обоих случаях мы создаем новый объект, имеющий тот интерфейс, который нам требуется (рис. 7.7).

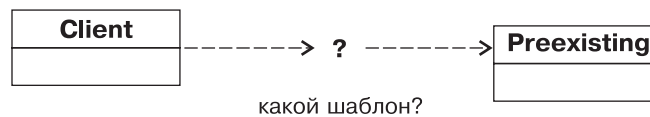


РИС. 7.7. Объект *Client* использует другой, уже существовавший ранее объект *Preexisting*, интерфейс которого нас не устраивает

Оболочка и объект-оболочка — это термины, которые можно слышать довольно часто. Принято считать, что использование объектов-оболочек для уже существующих систем упрощает работу с ними.

На этом самом высоком уровне шаблоны Facade и Adapter действительно кажутся похожими. В каждом из них используются классы-оболочки, но это оболочки разного типа. Очень важно четко понимать то довольно тонкое различие, которое существует между ними. В этом нам поможет сравнение свойств обоих шаблонов (табл. 7.1).

Таблица 7.1. Сравнение шаблона Facade с шаблоном Adapter

	Adapter	Facade
Использование уже существующих классов	Да	Да
Требуется разработка нового интерфейса	Нет	Да
Необходима поддержка полиморфизма	Нет	Вероятно
Требуется упростить существующий интерфейс	Да	Нет

Из сведений, приведенных в табл. 7.1, можно сделать следующие выводы.

- Как шаблон Facade, так и шаблон Adapter используют уже существующие классы.
- Однако для шаблона Facade необходимо самостоятельно разработать требуемый интерфейс, а не использовать уже существующий, как для шаблона Adapter.

³ Рекомендации по выбору конкретного варианта шаблона Adapter можно найти в книге “банды четырех” на с. 142–144.

- Реализация шаблона Facade не предусматривает поддержки полиморфизма, тогда как при применении шаблона Adapter это может оказаться необходимым. (Возможны ситуации, при которых требуется лишь поддержка определенного API, а значит, при использовании шаблона Adapter полиморфизм нас интересовать не будет — именно поэтому здесь употреблено слово "может").
- При использовании шаблона Facade наша цель — упростить существующий интерфейс. А для шаблона Adapter нам требуется поддержка уже существующего интерфейса и не допускаются никакие упрощения, даже если они вполне возможны.

Иногда слушатели приходят к заключению, что различие между шаблоном Facade и шаблоном Adapter состоит в том, что шаблон Facade скрывает множество классов, а шаблон Adapter — только один. Хотя довольно часто это замечание оказывается справедливым, возможны и противоположные ситуации. Например, шаблон Facade может применяться для работы с одним, но очень сложным объектом, в то время как шаблон Adapter может предоставлять оболочку для нескольких мелких объектов с целью объединения той функциональности, которая в них реализована.

Выводы. Шаблон Facade предназначен для упрощения интерфейса, тогда как шаблон Adapter предназначен для приведения различных существующих интерфейсов к единому требуемому виду.

Применение шаблона Adapter к проблеме САПР

В задаче о поддержке различных версий САПР (см. главу 3, *Проблема, требующая создания гибкого кода*) элементы в системе версии V2 были представлены объектами **OOGFeature**. К сожалению, эти объекты имели интерфейс, не отвечающий нашим требованиям (в данной конкретной ситуации), поскольку они были созданы другими разработчиками и их невозможно было сделать производными от класса **Feature**. Тем не менее, в самой системе версии V2 они функционировали вполне корректно.

В данной ситуации вариант создания новых классов, реализующих все функции объектов САПР, даже не рассматривался. Единственно возможное решение состояло в организации взаимодействия с объектами **OOGFeature**, и самый простой способ достижения этой цели заключался в использовании шаблона Adapter.

Резюме

Шаблон Adapter — это очень полезный шаблон, позволяющий привести интерфейс класса (или нескольких классов) к интерфейсу требуемого нам вида. Это достигается посредством определения нового класса-оболочки с требуемым интерфейсом и помещения в него исходного класса, методы которого будут вызываться методами класса-оболочки.

Приложение. Пример программного кода на языке C++

Листинг 7.2. Фрагмент реализации шаблона Adapter на языке C++ ---

```
class Circle : public Shape {
    . . .
private:
    XXCircle *pxc;
    . . .
}
Circle::Circle () {
    . . .
    pxc= new XXCircle;
}
void Circle::display () {
    pxc->displayIt();
}
```

Расширение горизонтов

Введение

В предыдущих главах уже упоминалось о трех фундаментальных концепциях объектно-ориентированного проектирования — объектах, инкапсуляции и абстрактных классах. Очень важно, как проектировщик применяет эти концепции. Традиционный подход к пониманию указанных терминов ограничивает возможности разработки. В этой главе мы вернемся на шаг назад и более подробно рассмотрим темы, уже обсуждавшиеся ранее. Здесь так же раскрываются особенности нового взгляда на объектно-ориентированное проектирование, связанные с появлением шаблонов проектирования.

В этой главе мы выполним следующее.

- Сравним в противопоставлении следующие подходы:
 - традиционный, рассматривающий объекты как совокупность данных и методов, и новый, понимающий под объектом предмет, имеющий некоторые обязательства;
 - традиционный, рассматривающий инкапсуляцию исключительно как механизм сокрытия данных, и новый, понимающий под инкапсуляцией способность к сокрытию чего угодно. Особенно важно понимать, что инкапсуляция может быть применена и для сокрытия различий в поведении;
 - традиционный способ применения механизма наследования с целью реализации специализации и повторного использования кода и новый, понимающий под наследованием метод классификации объектов.
- Проанализируем, как новый подход позволяет объектам демонстрировать различное поведение.
- Рассмотрим, как различные уровни проектирования — концептуальный, спецификаций и реализации — соотносятся с абстрактным классом и его производными классами.

Вероятно, предложенный здесь новый подход вовсе не является оригинальным. Я уверен, что именно он позволил создателям шаблонов проектирования разработать ту концепцию, которую сейчас принято называть шаблоном. Несомненно, что этот подход был известен и Кристоферу Александру, и Джиму Коплину (Jim Coplien), и "банде четырех".

Хотя предлагаемый здесь подход нельзя считать оригинальным, я уверен, что он никогда не обсуждался ранее в таком виде, как это сделано в данной книге. В отличие

от других авторов я предпринял попытку отделить рассмотрение шаблонов от обсуждения особенностей их поведения.

Говоря о новом подходе, я подразумеваю, что для большинства разработчиков это, вероятно, совершенно новый взгляд на объектно-ориентированное проектирование. По крайней мере, для меня он был таковым, когда я впервые приступил к изучению шаблонов проектирования.

Объекты: традиционное представление и новый подход

Традиционное представление об объектах состоит в том, что они представляют собой совокупность данных и методов их обработки. Один из моих преподавателей назвал их "умными данными". Лишь один шаг отделяет их от баз данных. Подобное представление возникает при взгляде на объекты с точки зрения их реализации.

Хотя данное определение является совершенно точным — как объяснялось выше, в главе 1, *Объектно-ориентированная парадигма*, — оно построено при взгляде на объекты с точки зрения их реализации. Более полезным является определение, построенное при взгляде на объекты с концептуальной точки зрения, когда объект рассматривается как сущность, имеющая некоторые обязательства. Эти обязательства определяют поведение объекта. В некоторых случаях мы также будем представлять объект как сущность, обладающую конкретным поведением.

Такое определение предпочтительнее, поскольку оно помогает сосредоточиться на том, что объект должен *делать*, не задаваясь преждевременно вопросом о том, как это можно реализовать. Подобный подход позволяет разделить процедуру разработки программного обеспечения на два этапа.

1. Создание предварительного проекта без излишней детализации всех процессов.
2. Реализация разработанного проекта в кодах.

В конечном счете, этот подход позволяет более точно выбрать и определить объект (в смысле отправной точки любого проекта). Второе определение объекта является более гибким, поскольку позволяет сосредоточиться на том, что объект делает, а механизм наследования предоставляет инструмент реализации его поведения по мере необходимости. При взгляде на объект с точки зрения реализации также можно достичь этого, но гибкость в этом случае, как правило, обеспечивается более сложным путем.

Гораздо проще рассуждать в терминах обязательств, так как это помогает определить открытый интерфейс объекта. Если объект обладает обязательствами, то очевидно, что должен существовать какой-то способ попросить его выполнить данные обязательства. Однако это требование никак не соотносится с тем, что находится внутри объекта. Информация, с которой связаны обязательства объекта, вовсе необязательно должна находиться непосредственно в самом этом объекте.

Предположим, что существует объект **Shape**, имеющий такие обязательства.

- Знать свое местоположение.
- Уметь отобразить себя на экране монитора.
- Уметь удалить свое изображение с экрана монитора.

Этот набор обязательств предполагает существование определенного набора методов, необходимых для их реализации:

- метод `GetLocation(...)`;
- метод `DrawShape(...)`;
- метод `UnDrawShape(...)`.

Для нас не имеет никакого значения, что именно находится внутри объекта **Shape**. Единственное, что нас интересует, это чтобы объект **Shape** обеспечивал требуемое поведение. Для этой цели могут использоваться атрибуты внутри объекта или методы, вычисляющие требуемые результаты или даже обращающиеся к другим объектам. Таким образом, объект **Shape** может как содержать атрибуты, описывающие его месторасположение, так и обращаться к другим объектам базы данных, чтобы получить сведения о своем местоположении. Такой подход предоставляет высокую гибкость, необходимую для эффективного моделирования процессов в проблемной области.

Интересно отметить, что перенос акцента с реализации на мотивацию — характерная черта описания шаблонов проектирования.

Стремитесь всегда рассматривать объекты в указанном ракурсе. Выбор такого подхода в качестве основного позволит вам создавать превосходные проекты.

Инкапсуляция: традиционное представление и новый подход

На лекциях по курсу проектирования с применением шаблонов я часто обращаюсь к студентам со следующим вопросом: "Кто из вас знаком с определением инкапсуляции как механизма сокрытия данных?". В ответ почти каждый человек в аудитории поднимает руку.

Затем я рассказываю им историю о моем зонтике. Следует заметить, что живу я в городе Сиэтл, штат Вашингтон, который отличается очень влажным климатом. Поэтому зонтики и плащи с капюшоном — вещь, совершенно необходимая жителям этого города осенью, зимой и весной.

Так вот, мой зонтик довольно большой — фактически, вместе со мной под ним могут найти укрытие еще три или четыре человека. Спрятавшись от дождя под этим зонтиком, мы можем перемещаться из одного помещения в другое, оставаясь сухими. Зонтик оборудован акустической стереосистемой, помогающей не скучать, пока мы находимся под его защитой. Представьте себе, что в нем имеется даже система кондиционирования воздуха, регулирующая окружающую температуру. Одним словом, это весьма комфортабельный зонтик.

Зонтик чрезвычайно удобен и всегда ждет меня там, где я его оставляю. У него есть колесики и его не нужно переносить с места на место в руках. Более того, перемещение этого зонтика происходит вообще без моего участия, поскольку он оборудован собственным двигателем. Если нужно, можно даже открыть специальный люк вверху моего зонтика, чтобы впустить под него лучи солнца. (Почему я пользуюсь зонтиком при солнечной погоде, я вам объяснять не стану.)

В Сиэтле есть сотни тысяч таких зонтиков различных типов и цветов.

Большинство людей называет их *автомобилями*.

Однако лично я думаю о своем автомобиле как о зонтике, ведь зонтик — это предмет, которым пользуются, чтобы укрыться от дождя. Каждый раз, когда я ожидаю кого-нибудь на улице во время дождя, я сижу в моем "зонтике", чтобы оставаться сухим!

Конечно, в действительности автомобиль — это не зонтик. Но несомненно и то, что его вполне можно использовать для защиты от дождя, хотя это будет очень ограниченное восприятие автомобиля. Точно так же, инкапсуляция — это не просто сокрытие данных. Такое слишком узкое представление ограничивает возможности проектировщика.

Инкапсуляцию следует понимать как "любой вид сокрытия". Другими словами, это механизм, способный скрывать данные. Но этот же механизм может использоваться для сокрытия реализации, порожденных классов и многого другого. Обратимся к диаграмме, представленной на рис. 8.1. Мы уже встречались с этой диаграммой в главе 7, *Шаблон Adapter*.

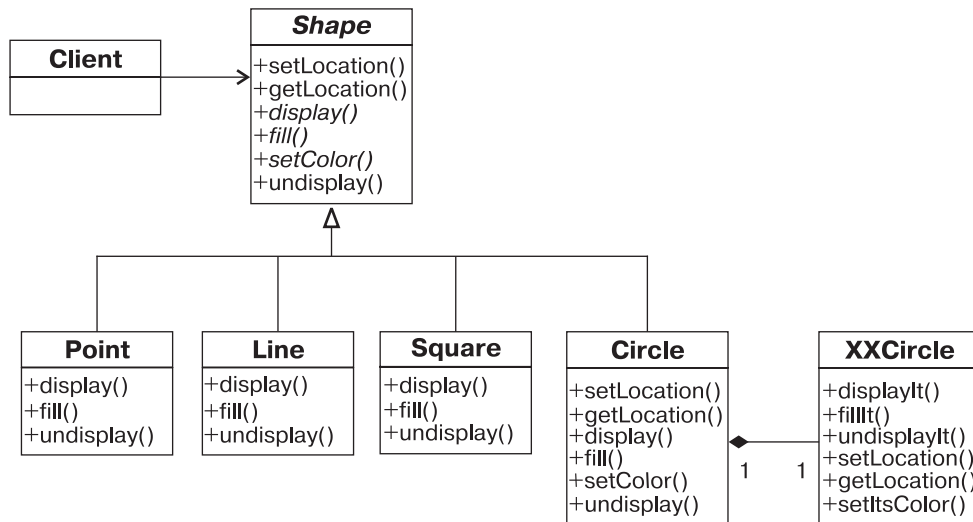


РИС. 8.1. Адаптация класса *XXCircle* с помощью класса *Circle*

На рис. 8.1 представлено несколько видов инкапсуляции.

- *Инкапсуляция данных.* Данные классов **Point**, **Line**, **Square** и **Circle** скрыты от всего остального мира.
- *Инкапсуляция методов.* Например, метод `setLocation()` класса **Circle** является скрытым.
- *Инкапсуляция подклассов.* Объектам-клиентам класса **Shape** классы **Point**, **Line**, **Square** или **Circle** не видны.
- *Инкапсуляция другого объекта.* Ничто в системе, кроме объектов класса **Circle**, не знает о существовании объектов класса **XXCircle**.

Следовательно, один тип инкапсуляция достигается созданием абстрактного класса, который демонстрирует полиморфное поведение, и клиент данного абстрактного класса не знает, с каким именно производным от него классом он имеет дело в действительности. Таким образом, адаптация интерфейса позволяет скрыть, что именно находится за адаптирующим объектом.

Преимущество подобного взгляда на инкапсуляцию состоит в том, что он упрощает разбиение (декомпозицию) программы. Инкапсулированные уровни в этом случае становятся интерфейсами, которые необходимо будет спроектировать. Инкапсуляция различных видов фигур в классе **Shape** позволяет добавлять в систему новые классы без модификации тех клиентских программ, которые их используют. Инкапсуляция объекта **XXCircle** в объекте **Circle** позволяет изменить его реализацию в будущем, если в этом возникнет потребность.

Когда объектно-ориентированная парадигма была впервые предложена, повторное использование классов понималось как одно из ее важнейших преимуществ. Обычно оно достигалось посредством разработки классов с последующим созданием на их основе новых, производных классов. Для процедуры создания этих подклассов, порожденных от других, базовых классов, использовался термин *специализация* (а для обратного перехода к базовому классу — *генерализация*).

Здесь я не собираюсь поднимать вопрос о точности этой терминологии, а просто рассматриваю тот подход, который в моем понимании представляет собой более мощный способ использования наследования. В приведенном выше примере можно было создать проект системы, основываясь на методе специализации класса **Shape** (с получением таких классов, как **Point**, **Line**, **Square** и **Circle**). Однако в этом случае мне, вероятно, не удалось бы спрятать указанные специализированные классы по отношению к методам использования класса **Shape**. Наоборот, скорее всего, я попытался бы воспользоваться преимуществами знания особенностей реализации каждого из производных классов.

Однако, если рассматривать класс **Shape** как обобщение для классов **Point**, **Line**, **Square** и **Circle**, то это упрощает восприятие их как единого понятия. В подобном случае разработчик, вероятнее всего, предпочтет разработать интерфейс и сделать класс **Shape** абстрактным. А это, в свою очередь, означает, что если потребуется ввести в систему поддержку новой фигуры, то необходимая модернизация существующего кода будет минимальна, поскольку никакой из объектов-клиентов не знает, с каким именно подтипом класса **Shape** он имеет дело.

Найдите то, что изменяется, и инкапсулируйте это

В своей книге¹ "банда четырех" предлагает следующее.

Выделите то, что будет изменяемым в вашем проекте. Этот подход противоположен методу, построенному на установлении причины, вынуждающей прибегнуть к переделке проекта. Вместо выявления тех причин, которые будут способны заставить нас внести изменения в проект, следует сосредоточиться на том, что мы хо-

¹ Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software, Reading, MA: Addison-Wesley, 1995, с. 29.

тим уметь изменять, не переделывая проект. Идея состоит в том, что следует *инкапсулировать те концепции, которые могут изменяться*, — лейтмотив многих шаблонов проектирования.

Иными словами это можно выразить так: "Найдите изменяемое и инкапсулируйте это".

Подобное утверждение покажется странным, если понимать инкапсуляцию только как сокрытие данных. Но оно выглядит весьма разумным, если под инкапсуляцией понимать сокрытие конкретных классов с помощью абстрактных. Помещение в текст вносимых изменений на одном уровне, не оказывая влияния на другой. Этим достигается *слабая связанность* между объектами разных уровней.

В действительности, многие шаблоны проектирования используют инкапсуляцию для распределения объектов по различным уровням. Это позволяет разработчику вносить изменения на одном уровне, не оказывая влияния на другой. Этим достигается *слабая связанность* между объектами разных уровней.

Данный принцип очень важен для шаблона Bridge (мост), о котором речь пойдет в главе 9, *Шаблон Bridge*. Однако перед этим хотелось бы обсудить те предубеждения, которые присущи многим разработчикам.

Предположим, что мы работаем над проектом, заключающимся в моделировании различных характеристик животных. Требования к проекту состоят в следующем.

- Каждый тип животного имеет различное количество ног.
 - Представляющий животное объект должен быть способен запоминать и возвращать эту информацию.
- Каждый тип животного использует различный способ передвижения.
 - Представляющий животное объект должен быть способен возвращать сведения о том, сколько времени понадобится животному для перемещения из одного места в другое по земной поверхности заданного типа.

Типичный подход к представлению различного количества ног у животного состоит в том, что объект будет включать данное-член, содержащее соответствующее значение, а также два метода для записи и считывания этого значения. Однако для представления различного поведения обычно используется другой подход.

Предположим, что существует два различных метода передвижения животных: бег и полет. Поддержка этого требования потребует написания двух различных фрагментов программного кода — один для представления бега, а другой для представления полета. Вполне очевидно, что простой переменной в этом случае будет недостаточно. Требование реализации двух различных методов ставит нас перед необходимостью выбрать один из двух возможных подходов.

- Использование элемента данных, предназначенного для хранения сведений о типе передвижения животного, представленного объектом.
- Создание двух различных типов класса **Animal** (животное), производных от абстрактного класса **Animal**. Один из них будет предназначен для представления бегающих животных, а другой — летающих.

К сожалению, оба указанных подхода не лишены недостатков.

- *Сильная связанность*. Первый подход (использование флажка в совокупности с построенным на нем переключателем) ведет к *сильной связанности*, которая

отрицательно проявится, если флажку потребуется принимать какие-то дополнительные значения. В любом случае требуемый программный код будет довольно громоздким.

- *Излишняя детализация.* Второй подход требует жесткого присвоения подтипа класса **Animal** и не позволяет представлять в системе животных, способных и бегать, и летать одновременно.

Существует и третий вариант: пусть класс **Animal** включает объект, представляющий соответствующий способ передвижения (как показано на рис. 8.2).

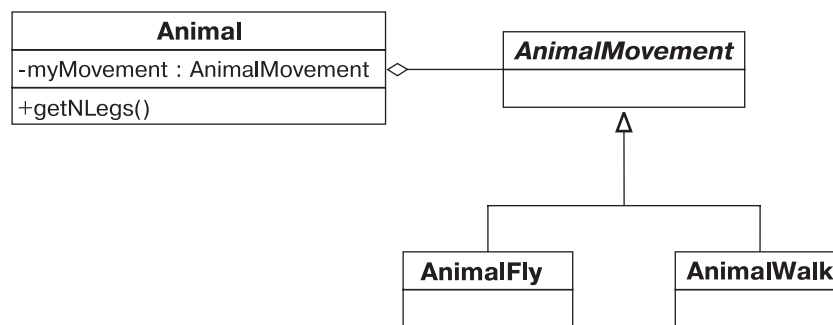


РИС. 8.2. Объект класса **Animal** содержит объект класса **AnimalMovement**

На первый взгляд такое решение кажется чрезмерно сложным. Однако в сущности оно совсем простое, поскольку предполагается лишь то, что класс **Animal** будет включать объект, представляющий способ передвижения животного. Подобный метод решения очень напоминает включение в класс данного-члена, содержащего сведения о количестве ног животного, — отличие лишь в том, что в этом случае сведения содержатся в данном-члене специфического объектного типа. Похоже, что на концептуальном уровне все это отличается сильнее, чем есть на самом деле, поскольку рис. 8.2 и 8.3 совершенно не похожи друг на друга.

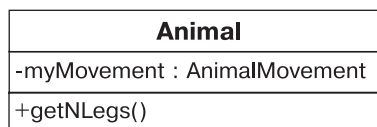


РИС. 8.3. Представление содержимого объекта как данного-члена

Многие разработчики полагают, что объект, содержащий другой объект, существенно отличается от объекта, содержащего только простые элементы данных. Однако данные-члены класса, которые обычно кажутся нам отличными от объектов (например, целые числа или числа двойной точности), на самом деле также являются объектами. В объектно-ориентированном программировании *все* является объектом, даже те встроенные типы данных, поведение которых является сугубо арифметическим.

Способы использования объектов для запоминания значений атрибутов и запоминания различных типов поведения фактически не отличаются друг от друга. Проще всего подтвердить это утверждение на конкретном примере. Предположим, необходимо создать систему управления сетью торговых точек. В этой системе должна обрабатываться приходная накладная. В накладной присутствует итоговая сумма. Для начала представим эту итоговую сумму как число с двойной точностью (тип `Double`). Однако если система предназначена для международной торговой сети, то очень скоро выяснится, что она обязательно должна обеспечивать конвертирование валют и другие подобные операции. Поэтому при расчетах потребуется использовать класс **Money** (деньги), включающий как значение суммы, так и описание типа валюты. Теперь для представления итоговой суммы в накладной необходимо будет использовать объект класса **Money**.

На первый взгляд использование класса **Money** в данном случае вызвано только необходимостью хранения дополнительных данных. Однако, когда потребуется конвертировать сумму в объекте класса **Money** из одного типа валюты в другой, именно данный объект **Money** должен будет выполнить такие преобразования, поскольку каждый объект должен нести ответственность сам за себя. Может показаться, что для выполнения преобразования достаточно просто добавить в класс еще одно данное-член, предназначенное для хранения коэффициента пересчета.

Однако в действительности все может оказаться сложнее. Например, от объекта может потребоваться выполнять конвертирование валюты по курсу прошлых периодов. В этом случае при расширении функциональных возможностей класса **Money** до класса **Currency** (валюта) мы по существу расширяем и функциональные возможности класса **SalesReceipt** (накладная), поскольку он включает в свой состав объекты класса **Money** (или класса **Currency**).

В последующих главах данная стратегия использования вложенных объектов для реализации требуемого поведения класса будет продемонстрирована при обсуждении нескольких новых шаблонов проектирования.

Общность и изменчивость в абстрактных классах

Обратимся к рис. 8.4, на котором отражены взаимосвязи между следующими концепциями.

- Анализ общности и анализ изменчивости.
- Концептуальный уровень, уровень спецификаций и уровень реализации.
- Абстрактный класс, его интерфейс и производные от него классы.

Как показано на рис. 8.4, анализ общности имеет отношение к концептуальному уровню проекта системы, тогда как анализ изменчивости относится к уровню его реализации, т.е. к процедурам конкретного воплощения системы.

Уровень спецификаций занимает промежуточное положение между двумя другими уровнями разработки системы. Поэтому он предусматривает выполнение обоих видов анализа. На уровне спецификаций определяется, как будет происходить взаимодействие с множеством объектов, которые концептуально схожи друг с другом, — т.е. каждый из этих объектов представляет конкретный вариант некоторой общей концепции. На уровне реализации каждая такая общая концепция описывается абстрактным классом или интерфейсом.

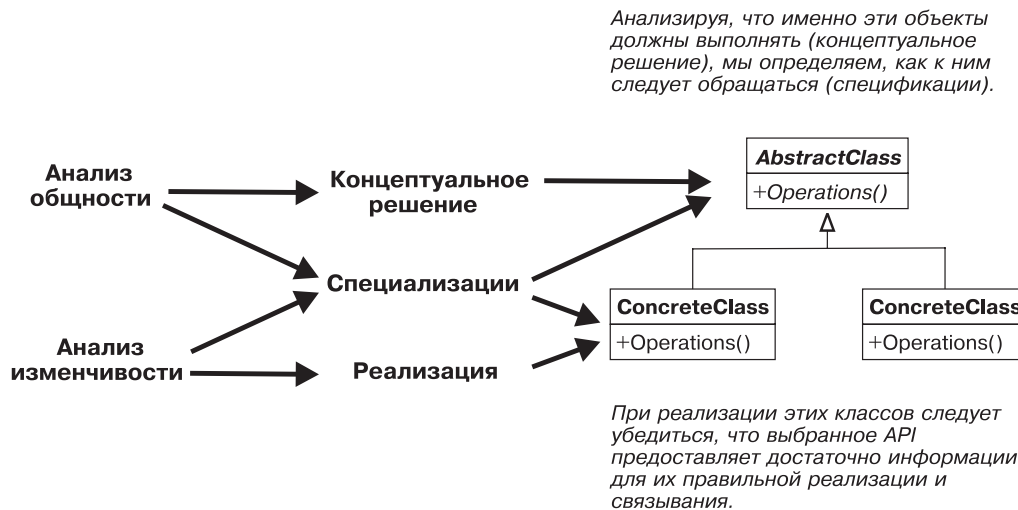


РИС. 8.4. Взаимосвязи между анализом общности/изменчивости, уровнями детализации и абстрактными классами

Новый взгляд на объектно-ориентированное проектирование кратко описывается в табл. 8.1.

Таблица 8.1. Новые концепции ООП

Концепция ООП	Пояснение
Абстрактный класс → центральная объединяющая концепция	Абстрактный класс используется для представления базовой концепции, объединяющей все порожденные от него классы. Эта базовая концепция отражает некоторую общность, имеющую место в предметной области
Общность → какие абстрактные классы должны использоваться	Общности определяют абстрактные классы, которые должны существовать в системе
Вариации → классы, производные от абстрактного класса	Вариации, выявленные в пределах некоторой общности, представляются классами, производными от соответствующего абстрактного класса
Специализация → интерфейс для абстрактного класса	Интерфейс этих производных классов соответствует уровню спецификаций

Подобный подход упрощает процесс проектирования классов и сводит его к процедуре из двух этапов, описанных в табл. 8.2.

Таблица 8.2. Проектирование классов

Когда определяется...	Следует задаться вопросом...
Абстрактный класс (общность)	Какой интерфейс позволит обратиться ко всем <i>обязательствам</i> этого класса?
Порожденные классы (изменчивость)	Как выполнить данную конкретную реализацию (вариацию) исходя из имеющейся спецификации?

Взаимосвязь между уровнем спецификаций и концептуальным уровнем можно описать так. *На уровне спецификаций определяется интерфейс, который необходим для реализации всех вариантов проявления (вариаций) некоторой концепции (общности), выявленной на концептуальном уровне.*

Взаимосвязь между уровнем спецификаций и уровнем реализации можно описать так. *Как, исходя из заданной спецификации, можно реализовать данное проявление (вариацию)?*

Резюме

Традиционное понимание концепций объекта, инкапсуляции и наследования весьма ограничено. Инкапсуляция представляет собой нечто большее, чем просто сокрытие данных. Расширение определения инкапсуляции до концепции сокрытия любых существующих категорий позволяет распределить объекты по разным уровням. Это, в свою очередь, позволяет вносить изменения на одном уровне, исключив нежелательное влияние этих действий на объекты другого уровня.

Наследование лучше использовать как метод определенной обработки различных конкретных классов, являющихся концептуально идентичными, а не как средство проведения специализации.

Концепция использования объектов для поддержки вариаций в поведении ничем не отличается от практики использования элементов данных для поддержки вариаций в значениях данных. Оба подхода предусматривают инкапсуляцию (т.е. расширение) как данных, так и поведения объектов.

Шаблон Bridge

Введение

Продолжим изучение шаблонов проектирования и рассмотрим шаблон Bridge (мост). Шаблон Bridge несколько сложнее тех шаблонов, о которых речь шла раньше, но и намного полезнее их.

В этой главе мы выполним следующее.

- Выведем концепцию шаблона Bridge, исходя из конкретного примера. Обсуждение будет проведено очень подробно, что должно помочь вам понять сущность этого шаблона.
- Рассмотрим ключевые особенности этого шаблона.
- Обсудим несколько поучительных примеров использования шаблона Bridge из моей личной практики.

Назначение шаблона проектирования Bridge

Согласно определению "банды четырех" "назначение шаблона Bridge состоит в отделении абстракции от реализации таким способом, который позволит им обеим изменяться независимо".¹

Я точно помню что впервые прочитав это определение, подумал:

Хм, ерунда какая-то!

А затем:

Я понимаю каждое слово в этом предложении, но не имею ни малейшего представления, что оно может означать.

Я знал следующее.

- *Отделение* означает обеспечение независимого поведения элементов, или, по крайней мере, явное указание на то, что между ними существуют некоторые отношения.
- *Абстракция* — это концептуальное представление о том, как различные элементы связаны друг с другом.

¹ Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software, Reading, MA: Addison-Wesley, 1995, с. 151.

Я также считал, что *реализация* — это конкретный способ представления абстракции, поэтому меня смущало предложение отделить абстракцию от конкретного способа ее реализации.

Как оказалось, мое замешательство было вызвано недопониманием того, что здесь в действительности представляет собой реализация. В данном случае под *реализацией* понимались те объекты, которые абстрактный и производные от него классы использовали для реализации своих функций (а не те классы, которые называются конкретными и являются производными от абстрактного класса). Но, честно говоря, даже если бы я все понимал правильно, не уверен, что это бы сильно мне помогло. Концепция, выраженная в приведенном выше определении, достаточно сложна, чтобы понять ее с первого раза.

Если вам сейчас тоже не до конца понятно, для чего предназначен шаблон Bridge, пусть это вас не беспокоит. Если же назначение этого шаблона вам вполне очевидно, то надо отдать должное вашей сообразительности.

Bridge — один из самых трудных для понимания шаблонов. До некоторой степени это вызвано тем, что он является очень мощным и часто применяется в самых различных ситуациях. Кроме того, он противоречит распространенной практике применения механизма наследования для реализации специальных случаев. Тем не менее, этот шаблон служит превосходным примером следования двум основным лозунгам технологии шаблонов проектирования: "Найди то, что изменяется, и инкапсулируй это" и "Компоновка объектов в структуру предпочтительней обращения к наследованию классов" (в дальнейшем мы убедимся в этом).

Описание шаблона проектирования Bridge на примере

Чтобы лучше понять идею построения шаблона Bridge и принципы его работы, рассмотрим конкретный пример поэтапно. Сначала обсудим установленные требования, а затем проанализируем вывод основной идеи шаблона и способы его применения.

Возможно, данный пример может показаться слишком простым. Однако присмотритесь к обсуждаемым в нем концепциям, а затем попытайтесь вспомнить аналогичные ситуации, с которыми нам приходилось сталкиваться ранее. Обратите особое внимание на следующее.

- Наличие вариаций в абстрактном представлении концепций.
- Наличие вариаций в том, как эти концепции реализуются.

Полагаю, у вас не появится сомнений, что приведенный ниже пример имеет много общего с обсуждавшейся выше задачей поддержки нескольких версий САПР. Однако мы не станем предварительно обсуждать все предъявляемые требования. Как правило, приступая к решению задачи, не удастся сразу увидеть все существующие вариации.

Рекомендация. Формулируя требования к проекту, старайтесь как можно раньше и как можно чаще обдумывать, что в нем может изменяться в дальнейшем.

Предположим, что нам нужно написать программу, которая будет выводить изображения прямоугольников с помощью одной из двух имеющихся графических программ. Кроме того, допустим, что указания о выборе первой (Drawing Program 1 —

DP1) или второй (DP2) графической программы будут предоставляться непосредственно при инициализации прямоугольника.

Прямоугольники определяются двумя парами точек, как это показано на рис. 9.1. Различия между графическими программами описаны в табл. 9.1.

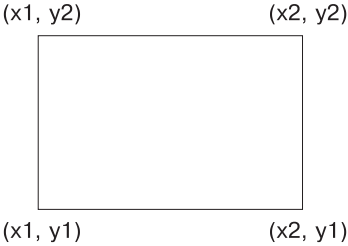


РИС. 9.1. Принцип описания прямоугольника

Таблица 9.1. Различия между двумя графическими программами

	Методы в DP1	Методы в DP2
Метод отображения линий	draw_a_line(x1, y1, x2, y2)	drawline (x1, x2, y1, y2)
Метод отображения окружностей	draw_a_circle(x, y, r)	drawcircle(x, y, r)

Заказчик поставил условие, что объект коллекции (клиент, использующий программу отображения прямоугольников) не должен иметь никакого отношения к тому, какая именно графическая программа будет использоваться в каждом случае. Исходя из этого я пришел к заключению, что, поскольку при инициализации прямоугольника указывается, какая именно программа должна использоваться, можно создать два различных типа объекта прямоугольника. Один из них будет вызывать для отображения программу DP1, а другой – программу DP2. В каждом типе объекта будет присутствовать метод отображения прямоугольника, но реализованы они будут по-разному – как показано на рис. 9.2.

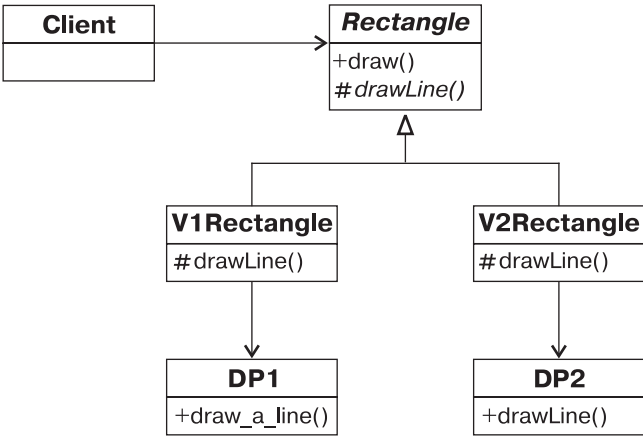


РИС. 9.2. Схема программы отображения прямоугольников с помощью программ DP1 и DP2

Создав абстрактный класс *Rectangle* (прямоугольник), можно воспользоваться тем преимуществом, что единственное несовпадение между различными типами его производных классов *Rectangle* состоит в способе реализации метода *drawLine()*. В классе *V1Rectangle* он реализован посредством ссылки на объект *DP1* и его метод *Draw_a_line()*, а в классе *V2Rectangle* — посредством ссылки на объект *DP2* и его метод *drawline()*. Таким образом, выбрав при инициализации требуемый тип объекта *Rectangle*, в дальнейшем можно полностью игнорировать эти различия. В листинге 9.1 приведен пример реализации этих объектов на языке Java.

Листинг 9.1. Пример реализации классов на языке Java. Версия 1

```
Class Rectangle{
    Public void draw () {
        DrawLine (_x1, _y1, _x2, _y1);
        DrawLine (_x2, _y1, _x2, _y2);
        DrawLine (_x2, _y2, _x1, _y2);
        DrawLine (_x1, _y2, _x1, _y1);
    }
    abstract protected void
        DrawLine (double x1, double y1,
                  double x2, double y2);
}

Class V1Rectangle extends Rectangle {
    DrawLine (double x1, double y1,
              double x2, double y2) {
        DP1.draw_a_line (x1, y1, x2, y2);
    }
}

Class V2Rectangle extends Rectangle {
    DrawLine (double x1, double y1,
              double x2, double y2) {
        // Аргументы в методе программы DP2 отличаются
        // и должны быть переупорядочены
        DP2.drawline (x1, x2, y1, y2);
    }
}
```

Предположим, что сразу после завершения и отладки кода, приведенного в листинге 9.1, на нас обрушивается одно из *трех неизбежных зол* (имеется в виду смерть, налоги и изменение требований). Необходимо включить в программу поддержку еще одного вида геометрических фигур — окружностей. Однако при этом уточняется, что объект коллекции (клиент) не должен ничего знать о различиях, существующих между объектами, представляющими прямоугольники (*Rectangle*) и окружности (*Circle*).

Логично будет сделать вывод, что можно применить выбранный ранее подход и просто добавить еще один уровень в иерархию классов программы. Потребуется только добавить в проект новый абстрактный класс (назовем его *Shape* (фигура)), а классы *Rectangle* и *Circle* будут производными от него. В этом случае объект *Client* сможет просто обращаться к объекту класса *Shape*, совершенно не заботясь о том, какая именно геометрическая фигура им представлена.

Для аналитика, только начинающего работать в области объектно-ориентированной разработки, такое решение может показаться вполне естественным — реализовать изменение в требованиях только с помощью механизма наследования. Начав

со схемы, представленной на рис. 9.2, добавим к ней новый уровень с классом *Shape*. Затем для всех видов фигур организуем их реализацию с помощью каждой из графических программ, создав по отдельному производному классу для вызова объектов *DP1* и *DP2*, как в классе *Rectangle*, так и в классе *Circle*. В итоге будет получена схема, подобная представленной на рис. 9.3.

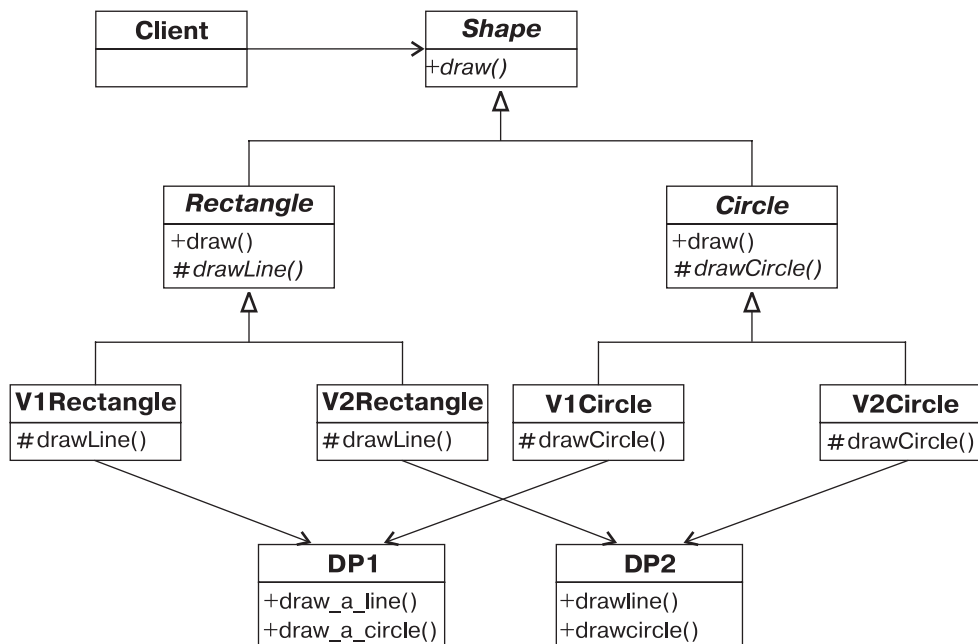


РИС. 9.3. Простейший прямолинейный подход к реализации двух фигур с помощью двух графических программ

Класс *Circle* можно реализовать тем же способом, что и класс *Rectangle* — как показано в листинге 9.2. Единственное отличие состоит в том, что вместо метода *drawLine()* используется метод *drawCircle()*.

Листинг 9.2. Пример реализации классов на языке Java. Версия 2

```

abstract class Shape {
    abstract public void draw ();
}

abstract class Rectangle extends Shape {
    public void draw () {
        drawLine(_x1, _y1, _x2, _y1);
        drawLine(_x2, _y1, _x2, _y2);
        drawLine(_x2, _y2, _x1, _y2);
        drawLine(_x1, _y2, _x1, _y1);
    }

    abstract protected void
    drawLine(

```

```
        double x1, double y1,
        double x2, double y2);
    }

    class V1Rectangle extends Rectangle {
        protected void drawLine (
            double x1, double y1,
            double x2, double y2) {
            DP1.draw_a_line(x1, y1, x2, y2);
        }
    }

    class V2Rectangle extends Rectangle {
        protected void drawLine (
            double x1, double x2,
            double y1, double y2) {
            DP2.drawline(x1, x2, y1, y2);
        }
    }

    abstract class Circle {
        public void draw () {
            drawCircle(x, y, r);
        }
        abstract protected void
            drawCircle (
                double x, double y, double r);
    }

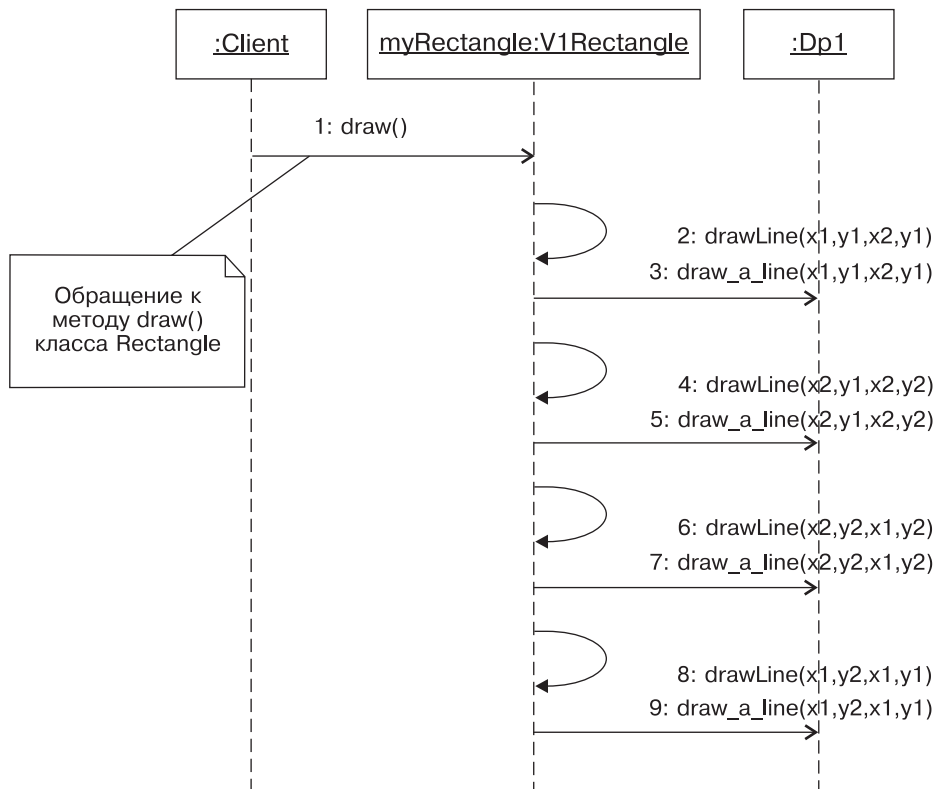
    class V1Circle extends Circle {
        protected void drawCircle() {
            DP1.draw_a_circle(x, y, r);
        }
    }

    class V2Circle extends Circle {
        protected void drawCircle() {
            DP2.drawcircle(x, y, r);
        }
    }
}
```

Для того чтобы лучше понять особенности этого проекта, рассмотрим конкретный пример. Например, проанализируем, как работает метод `draw()` класса **V1Rectangle**.

- Метод `draw()` класса **Rectangle** не изменился (в нем четыре раза подряд вызывается метод `drawLine()`).
- Метод `drawLine()` реализуется посредством обращения к методу `draw_a_line()` объекта **DP1**.

Схематично все это представлено на рис. 9.4.

РИС. 9.4. Диаграмма последовательностей обращения к объекту класса *V1Rectangle*

Чтение диаграмм последовательностей

Выше, в главе 2, мы уже обсуждали язык UML — унифицированный язык моделирования. Диаграмма на рис. 9.4 представляет собой один из видов диаграмм взаимодействия, называемый *диаграммой последовательностей*. Это типичная диаграмма языка UML. Ее назначение состоит в том, чтобы продемонстрировать взаимодействие объектов в системе.

- Каждый прямоугольник вверху диаграммы обозначает объект. Он может быть поименован или нет.
- Если у объекта есть имя, оно указывается слева от двоеточия.
- Класс, которому принадлежит объект, указывается справа от двоеточия. Например, на рис. 9.4 имя среднего объекта myRectangle, и этот объект представляет собой экземпляр класса V1Rectangle.

Диаграмму последовательностей следует читать сверху вниз. На диаграмме каждое пронумерованное предложение представляет собой сообщение, отправленное объектом самому себе или другому объекту.

- Последовательность начинается с непоименованного объекта класса Client, вызывающего метод draw() объекта myRectangle класса V1Rectangle.
- Этот метод четыре раза вызывает метод drawLine() собственного объекта (этапы 2, 4, 6 и 8 на схеме). Обратите внимание на изогнутую стрелку, указывающую на оси времени на сам объект myRectangle.
- В свою очередь, метод drawLine() каждый раз вызывает метод draw_a_line() непоименованного объекта класса Dp1 (этапы 3, 5, 7 и 9 на диаграмме).

Несмотря на то что при взгляде на диаграмму классов складывается впечатление о наличии на ней множества объектов, в действительности, мы имеем дело только с тремя объектами, как показано на рис. 9.5.

- Объект класса **Client**, потребовавший отобразить прямоугольник.
- Объект класса **V1Rectangle**.
- Объект класса **DP1**, представляющий графическую программу.

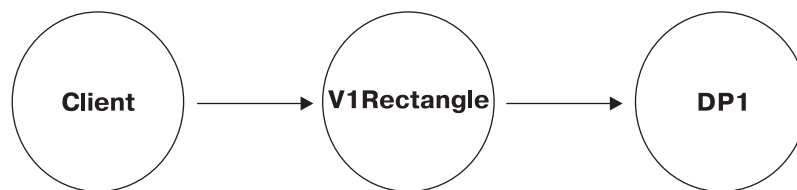


РИС. 9.5. Объекты, представленные на диаграмме последовательностей

Когда объект-клиент посылает сообщение объекту класса **V1Rectangle** (с именем `myRectangle`) с требованием выполнить метод `draw()`, последний реализует метод `draw()` класса **Rectangle**, выполняя этапы со 2 по 9, показанные на диаграмме.

К сожалению, этот подход создает новые проблемы. Взгляните еще раз на рис. 9.3 и обратите внимание на третий ряд классов. Глядя на них, можно сделать следующие выводы.

- Классы в этом ряду представляют те четыре конкретных подтипа класса **Shape**, с которыми мы имеем дело.
- Что произойдет, если появится третий тип графической программы, т.е. еще один возможный вариант реализации графических функций? В этом случае потребуется создать шесть различных подтипов класса **Shape** (для представления двух видов фигур и трех графических программ).
- Теперь предположим, что требуется реализовать поддержку нового, третьего, типа фигур. В этом случае нам понадобятся уже девять различных подтипов класса **Shape** (для представления трех видов фигур и трех графических программ).

Быстрый рост количества требуемых производных классов вызван тем, что в данном решении абстракция (виды фигур) и реализация (графические программы) жестко связаны между собой. Каждый класс, представляющий конкретный тип фигуры, должен точно знать, какую из существующих графических программ он использует. Для решения данной проблемы необходимо отделить изменения в абстракции от изменений в реализации таким образом, чтобы количество требуемых классов возросло линейно — как показано на рис. 9.6.

Вот мы и пришли к приведенному в начале главы определению назначения шаблона Bridge — отделение абстракции от реализации, выполненное таким способом, который позволит им обеим изменяться независимо².



РИС. 9.6. Шаблон Bridge отделяет изменения в абстракции от изменений в реализации

Перед тем как рассмотреть возможное решение проблемы и вывести из него шаблон Bridge, остановимся на нескольких других проблемах (помимо комбинаторного взрыва количества классов).

Еще раз посмотрим на рис. 9.3 и зададимся вопросом: какие еще недостатки имеет данный проект?

- Существует ли в нем избыточность?
- Что характерно для данного проекта — сильная или слабая связность?
- Сильно или слабо связаны между собой отдельные элементы проекта?
- Согласились бы вы выполнять сопровождение программ этого проекта в будущем?

Злоупотребление наследованием

Будучи начинающим разработчиком объектно-ориентированных проектов, я обычно решал проблемы обсуждаемого типа, выделяя их как специальные случаи с последующим применением механизма наследования. Мне нравилась сама идея наследования, поскольку это представлялось мне новым и мощным подходом. Я использовал наследование всегда, когда это было возможным. Такой подход выглядит вполне естественным для новичка, но, в сущности, он весьма наивен.

К сожалению, многие из существующих методик обучения объектно-ориентированному проектированию фокусируют главное внимание на абстракции данных, что делает проектирование чрезмерно зависимым от внутренней организации объектов. И даже когда я стал опытным разработчиком, то еще долго сохранял верность парадигме проектирования, основанной на широком использовании механизма наследования. Иными словами, я определял характеристики создаваемых классов исходя из особенностей их реализации. Однако характеристики объектов

должны определяться их обязательствами, а не их содержанием или внутренней структурой. Скажем, объекты могут отвечать за предоставление информации о самих себе. Например, объекту, представляющему в системе покупателя, может потребоваться умение сообщать его имя. Следует стремиться думать об объектах прежде всего в терминах их обязательств, а не внутренней структуры.

² Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-

Опытные разработчики объектно-ориентированных проектов пришли к выводу, что для полной реализации всей мощи механизма наследования следует применять его выборочно. Использование шаблонов проектирования помогает быстрее это понять. Оно позволяет перейти от выделения отдельной специализации для каждого изменения (методом наследования) к перемещению этих изменений в используемые объекты сторонней или собственной разработки.

Поначалу, столкнувшись с указанными выше проблемами, я решил, что причина их появления кроется в неправильном построении иерархии наследования. Поэтому я попробовал применить альтернативный вариант иерархии классов, представленный на рис. 9.7.

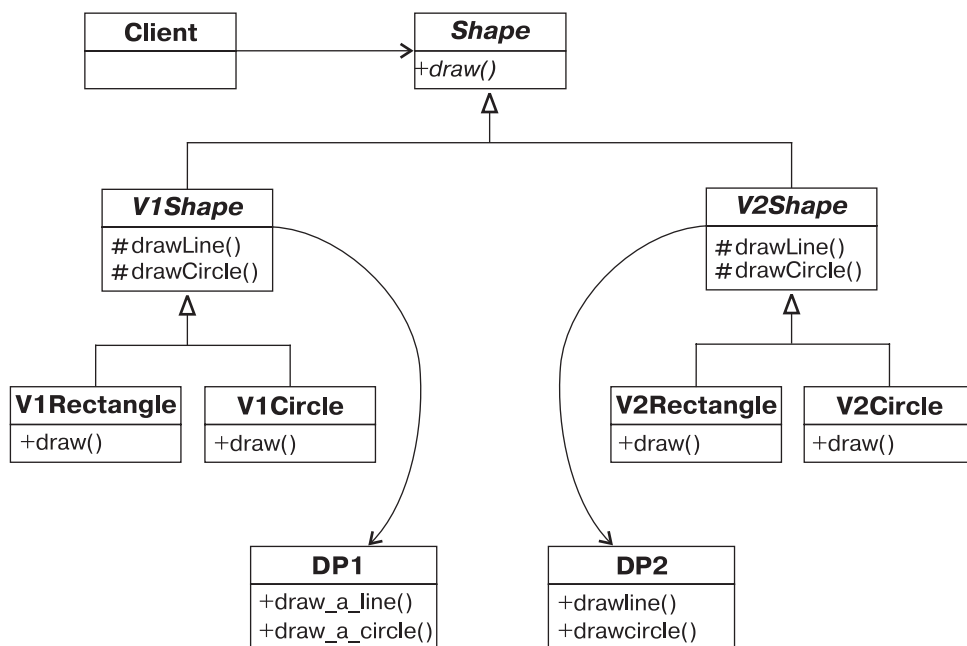


РИС. 9.7. Альтернативный вариант иерархии наследования классов

Здесь по-прежнему используются те же четыре класса, представляющие все возможные комбинации. Однако в этой версии устранена избыточность пакетов классов для графических программ **DP1** и **DP2**.

К сожалению, в этом варианте мне так и не удалось устранить избыточность, связанную с наличием двух типов классов **Rectangle** и двух типов классов **Circle**, причем в каждой паре родственных классов используется один и тот же метод `draw()`.

Как бы там ни было, исключить в новой схеме упомянутое выше комбинаторное увеличение количества классов так и не удалось.

Диаграмма последовательностей для нового решения представлена на рис. 9.8.

Хотя по сравнению с первоначальным решением новый вариант иерархии классов действительно обладает определенными преимуществами, он все же имеет недостат-

ки. Здесь по-прежнему сохраняются проблемы слабой связности классов и их избыточной связанности между собой.

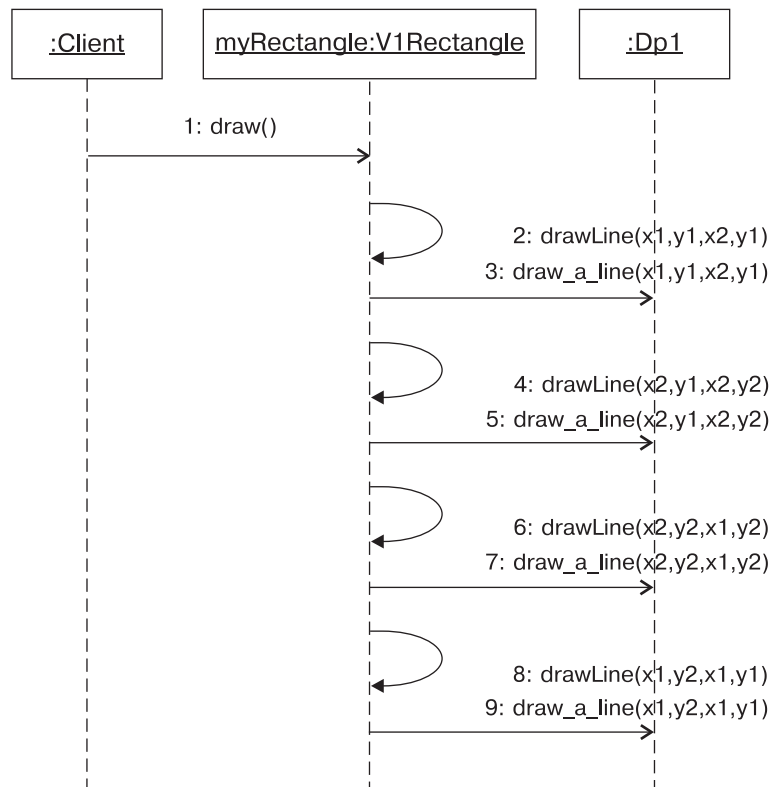


РИС. 9.8. Диаграмма последовательностей для нового варианта иерархии классов

Замечание. Эта версия иерархии классов по-прежнему не вызывает желания сопровождать программный код системы! Должно существовать лучшее решение.

Ищите альтернативы на начальной стадии проектирования

Хотя предложенный здесь альтернативный вариант иерархии наследования классов немного лучше первоначального, следует отметить, что сама по себе практика поиска альтернативы первоначальному варианту проекта — это хороший стиль разработки. Слишком часто проектировщики останавливаются на том варианте, который первый пришел им в голову, и в дальнейшем не предпринимают каких-либо попыток его улучшить. Я не призываю вас к чрезмерно глубокому и исчерпывающему изучению всех возможных альтернатив (это еще один способ свергнуть проект в состояние "паралича за счет анализа"). Однако сделать шаг назад и подумать, как можно было бы преодолеть затруднения, свойственные исходному варианту проекта, — это верный подход. Фактически, именно подобный шаг назад и отказ от дальнейшей работы над заведомо слабым проектным решением привели меня к пониманию всей мощи подхода с использованием шаблонов проектирования, описанию которого, собственно, и посвящена эта книга.

Замечания об использовании шаблонов проектирования

Впервые знакомясь с идеологией шаблонов проектирования, люди часто излишне сосредотачиваются на тех решениях, которые предлагаются шаблонами. Это кажется им вполне разумным, поскольку данная идеология преподносится как совокупность оптимальных решений разнообразных задач.

Однако это не совсем правильный подход. Если изучать шаблоны проектирования преимущественно с точки зрения тех решений, которые они предлагают, то это затрудняет понимание, в каких ситуациях те или иные шаблоны применяются. Каждый шаблон сам по себе сообщает только о том, *что надо сделать*, но умалчивает о том, *когда* это надо делать и *почему*.

Я пришел к выводу, что при изучении шаблонов полезнее сосредоточиться на контексте их применения — т.е. на тех проблемах, которые пытаются решить с их помощью. Это позволяет получить ответы на вопросы *когда* и *почему*. Такой подход перекликается с философией шаблонов Александера: "Каждый шаблон описывает проблему, которая возникает в данной среде снова и снова, а затем предлагает принцип ее решения таким способом..."³.

Попробуем применить данный подход к нашему случаю и сформулируем задачу, для решения которой предназначен шаблон Bridge.

Шаблон Bridge полезен тогда, когда имеется некоторая абстракция и существует несколько различных вариантов ее реализации. Шаблон позволяет абстракции и реализации изменяться независимо друг от друга.

Указанные характеристики как нельзя лучше соответствуют нашей ситуации. Поэтому можно сделать вывод, что шаблон Bridge *следует* использовать, даже еще не зная толком, как он реализуется на практике. То, что абстракцию можно будет менять независимо от ее реализации, означает, что новые элементы абстракции можно будет добавлять, не внося каких-либо изменений на уровне реализации и наоборот.

Существующее у нас решение не обеспечивает подобной независимости изменений. Не вызывает сомнения, что проект был бы гораздо лучше, если бы удалось найти решение, предоставляющее такую возможность.

Замечание. Обратите особое внимание на то, что, даже не зная конкретных способов реализации шаблона Bridge, мы смогли прийти к выводу о возможности и полезности его применения в нашем проекте. Позднее вы поймете, что это замечание справедливо в отношении практически всех шаблонов проектирования. Иными словами, всегда можно установить, что применение того или иного шаблона в данной предметной области будет возможно и полезно, даже не имея точного представления о том, как именно он может быть реализован.

³ Alexander C., Ishikawa S., Silverstein M. A Pattern Language: Towns/Buildings/Construction, New York, NY: Oxford University Press, 1977, с. X.

Описание шаблона проектирования Bridge и его вывод

Теперь, когда достигнуто ясное понимание стоящей перед нами проблемы, пришло время общими усилиями вывести шаблон Bridge. Самостоятельный вывод этого шаблона поможет нам осознать его сложность и, одновременно, мощь.

Используем на практике некоторые из основных положений качественного объектно-ориентированного проектирования — они помогут нам найти решение, очень близкое к шаблону Bridge. С этой целью обратимся к работе Джима Коплина по анализу общности и изменчивости⁴.

Шаблоны проектирования — это проектные решения, применяемые многократно

Каждый шаблон проектирования — это решение, которое успешно применялось в различных задачах на протяжении определенного времени и тем доказало свою надежность и качество. Подход, принятый в этой книге, состоит в том, чтобы *вывести* решение, положенное в основу шаблона, что позволит лучше изучить и понять его основные характеристики.

Мы уже выяснили, что шаблоном, который необходимо вывести в данном случае, является шаблон Bridge, — выше обсуждалось его определение в книге "банды четырех" и рассматривалась возможность его применения к некоторой реальной задаче. Однако следует напомнить, что в действительности этот шаблон нами еще не выведен. По определению всякий шаблон характеризуется повторяемостью — чтобы считаться шаблоном, некоторое решение должно быть применено, по крайней мере, в трех независимых случаях. Под словом "вывести" здесь подразумевается, что в процессе проектирования самостоятельно будет найдено решение, соответствующее идее шаблона, как если бы до этого момента мы не имели о нем никакого представления. Именно этот подход позволит нам выявить ключевые принципы и полезные стратегии использования данного шаблона.

Работа Дж. Коплина по анализу общности и изменчивости включает рекомендации по поиску того, что изменяется и что является общим в заданной проблемной области. Этими рекомендациями мы и воспользуемся. Наша задача состоит в том, чтобы определить, где возможны изменения (анализ общности), а затем установить, как это изменение происходит (анализ изменчивости).

По Коплину, "анализ общности заключается в поиске общих элементов, что поможет понять, чем члены семейства похожи друг на друга."⁵ Таким образом, это процесс поиска общих черт во всех элементах, составляющих некоторое семейство (и, следовательно, их различий).

Анализ изменчивости позволяет установить, каким образом члены семейства изменяются. Изменчивость имеет смысл только в пределах данной общности.

Анализ общности предусматривает выявление структур, которые вряд ли будут изменяться с течением времени, тогда как анализ изменчивости заключается в обнаружении структур, которые, вероятно, изменятся. Анализ изменчивости имеет смысл только в терминах контекста, определенного предварительным анализом общности... В архитектурном смысле анализ общности дает архитектуре ее долговечность, а анализ изменчивости способствует достижению удобства в использовании.

⁴ Coplein J. Multi-Paradigm Design for C++. Reading, MA: Addison-Wesley, 1998.

⁵ Coplein J. Multi-Paradigm Design for C++. Reading, MA: Addison-Wesley, 1998, с. 63.

Другими словами, если изменчивость — это особые случаи в рамках заданной предметной области, то общность устанавливает в ней концепции, объединяющие эти особые случаи между собой. Общие концепции будут представлены в системе абстрактными классами. Вариации, обнаруженные при анализе изменчивости, реализуются посредством создания конкретных классов, производных от этих абстрактных классов.

В объектно-ориентированном проектировании стала уже почти аксиомой практика, когда разработчик, анализируя описание проблемной области, выделяет в нем существительные и создает объекты, представляющие их. Затем он отыскивает глаголы, связанные с этими существительным (т.е. их действия), и реализует их, добавляя к объектам необходимые методы. Подобный процесс проявления повышенного внимания к существительным и глаголам в большинстве случаев приводит к созданию слишком громоздкой иерархии классов. Я считаю, что анализ общности и изменчивости как первичный инструмент выделения объектов, в действительности, предпочтительнее поиска существительных и глаголов в описании предметной области. (Полагаю, что книга Джима Коплина убедительно подтверждает эту точку зрения.)

В практике проектирования для работы с изменяющимися элементами применяются две основные стратегии.

- Найти то, что изменяется, и инкапсулировать это.
- Преимущественно использовать композицию вместо наследования.

Ранее для координации изменяющихся элементов разработчики часто создавали обширные схемы наследования классов. Однако вторая из приведенных выше стратегий рекомендует везде, где только возможно, заменять наследование композицией. Идея состоит в том, чтобы инкапсулировать изменения в независимых классах, что позволит при обработке будущих изменений обойтись без модификации программного кода. Одним из способов достижения подобной цели является помещение каждого подверженного изменениям элемента в собственный абстрактный класс с последующим анализом, как эти абстрактные классы соотносятся друг с другом.

Внимательный взгляд на инкапсуляцию

Чаще всего начинающих разработчиков объектно-ориентированных систем учат, что инкапсуляция заключается в сокрытии данных. К сожалению, это очень ограниченное определение. Несомненно, что инкапсуляция действительно позволяет скрывать данные, но она может использоваться и для многих других целей. Еще раз обратимся к рис. 7.2, на котором показано, как инкапсуляция применяется на нескольких уровнях. Безусловно, с ее помощью скрываются данные для каждой конкретной фигуры. Однако обратите внимание на то, что объект `Client` также ничего не знает о конкретных типах фигур. Следовательно, объект `Client` не имеет сведений о том, что объекты класса `Shape`, с которыми он взаимодействует, в действительности являются объектами разных конкретных классов — `Rectangle` и `Circle`. Таким образом, тип конкретного класса, с которым объект `Client` взаимодействует, скрыт от него (инкапсулирован). Это тот вид инкапсуляции, который подразумевается во фразе "найдите то, что изменяется, и инкапсулируйте это". Здесь как раз было обнаружено то, что изменяется, и оно было скрыто за "стеной" абстрактного класса (см. главу 8, *Расширение горизонтов*).

Рассмотрим данный процесс на примере задачи с вычерчиванием прямоугольника.

Сначала идентифицируем то, что изменяется. В нашем случае это различные типы фигур (представленных абстрактным классом `Shape`) и различные версии графических программ. Следовательно, общими концепциями являются понятия типа фигуры

и графической программы, как показано на рис. 9.9. (Обратите внимание на то, что имена классов выделены курсивом, поскольку это абстрактные классы.)

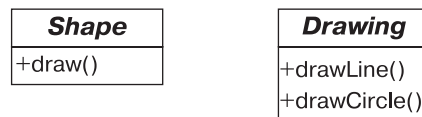


РИС. 9.9. Сущности, которые изменяются в нашем примере

В данном случае предполагается, что абстрактный класс *Shape* инкапсулирует концепцию типов фигур, с которыми необходимо работать. Каждый тип фигуры должен знать, как себя нарисовать. В свою очередь, абстрактный класс *Drawing* (рисование) отвечает за вычерчивание линий и окружностей. На рисунке указанные выше обязательства представлены посредством определения соответствующих методов в каждом из классов.

Теперь необходимо представить на схеме те конкретные вариации, с которыми мы будем иметь дело. Для класса *Shape* это прямоугольники (класс *Rectangle*) и окружности (класс *Circle*). Для класса *Drawing* это графические программы DP1 (класс *V1Drawing*) и DP2 (класс *V2Drawing*). Все это схематически показано на рис 9.10.

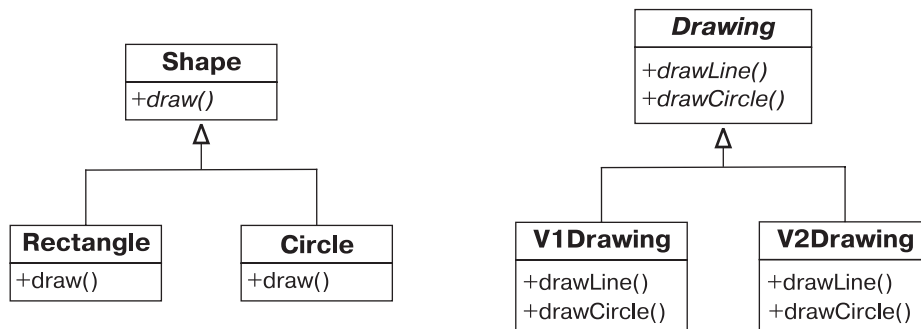


РИС. 9.10. Представление конкретных вариаций

Сейчас наша диаграмма имеет очень упрощенный вид. Определенно известно, что класс *V1Drawing* будет использовать программу DP1, а класс *V2Drawing* — программу DP2, но на схеме еще не указано, как это будет сделано. Пока мы просто описали концепции, присутствующие в проблемной области (фигуры и графические программы), и указали их возможные вариации.

Имея перед собой два набора классов, очевидно, следует задаться вопросом, как они будут взаимодействовать друг с другом. На этот раз попытаемся обойтись без того, чтобы добавлять в систему еще один новый набор классов, построенный на углублении иерархии наследования, поскольку последствия этого нам уже известны (см. рис. 9.3 и 9.7). На этот раз мы подойдем с другой стороны и попробуем определить, как эти классы могут использовать друг друга (в полном соответствии с приведенным выше утверждением о предпочтительности композиции над наследованием). Главный вопрос здесь состоит в том, какой же из абстрактных классов будет использовать другой?

Рассмотрим две возможности: либо класс *Shape* использует классы графических программ, либо класс *Drawing* использует классы фигур.

Начнем со второго варианта. Чтобы графические программы могли рисовать различные фигуры непосредственно, они должны знать некоторую общую информацию о фигурах: что они собой представляют и как выглядят. Однако это требование нарушает фундаментальный принцип объектной технологии: каждый объект должен нести ответственность только за себя.

Это требование также нарушает инкапсуляцию. Объекты класса *Drawing* должны были бы знать определенную информацию об объектах класса *Shape*, чтобы иметь возможность отобразить их (а именно — тип конкретной фигуры). В результате объекты класса *Drawing* фактически оказываются ответственными не только за свое собственное поведение.

Вернемся к первому варианту. Что если объекты класса *Shape* для отображения себя будут использовать объекты класса *Drawing*? Объектам класса *Shape* не нужно знать, какой именно тип объекта класса *Drawing* будет использоваться, поэтому классу *Shape* можно разрешить ссылаться на класс *Drawing*. Дополнительно класс *Shape* в этом случае можно сделать ответственным за управление рисованием.

Последний вариант выглядит предпочтительнее. Графически это решение представлено на рис. 9.11.

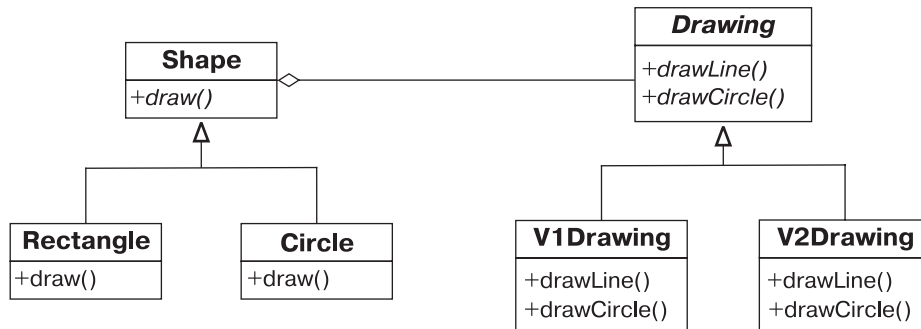


РИС. 9.11. Связывание абстрактных классов между собой

В данном случае класс *Shape* использует класс *Drawing* для проявления собственного поведения. Мы оставляем без внимания особенности реализации классов *V1Drawing*, использующего программу DP1, и *V2Drawing*, использующего программу DP2. На рис. 9.12 эта задача решена посредством добавления в класс *Shape* защищенных методов `drawLine()` и `drawCircle()`, которые вызывают методы `drawLine()` и `drawCircle()` объекта *Drawing*, соответственно.

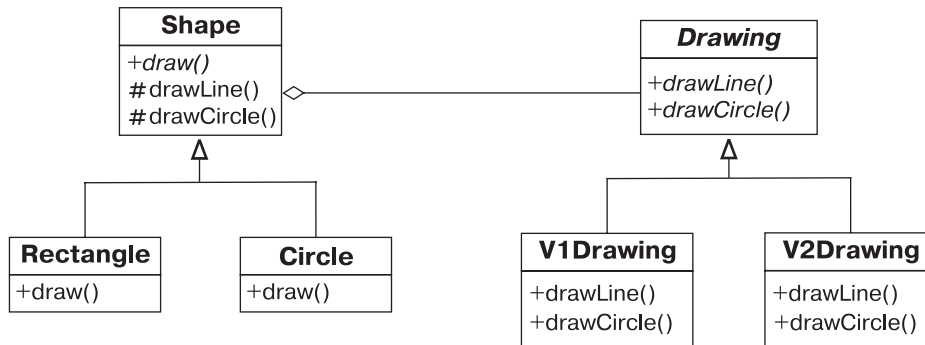


РИС. 9.12. Расширение проекта

Одно правило, одно место

При проектировании реализации очень важно следовать стратегии, согласно которой каждое правило реализуется только в одном месте. Другими словами, если имеется некоторое правило, определяющее способ выполнения каких-либо действий, оно должно быть реализовано только в одном месте. Такой подход обычно приводит к получению программного кода с большим количеством коротких методов. При этом за счет минимальных дополнительных усилий устраняется дублирование и удается избежать множества потенциальных проблем. Дублирование вредно не только из-за выполнения дополнительной работы по многократному вводу одинаковых фрагментов кода, но и в большей степени из-за вероятности того, что при каких-либо изменениях в будущем модификация будет произведена не везде.

В методе `draw()` класса `Rectangle` можно было бы непосредственно вызвать метод `drawLine()` того объекта класса `Drawing`, к которому обращается объект класса `Shape`. Однако, следуя указанной выше стратегии, можно улучшить программный код, создав в классе `Shape` метод `drawLine()`, который и будет вызывать метод `drawLine()` класса `Drawing`.

Я не считаю себя консерватором (по крайней мере, в большинстве случаев), но в этом вопросе я полагаю совершенно необходимым всегда соблюдать установленные правила. В примере, который будет обсуждаться ниже, класс `Shape` содержит метод `drawLine()`, описывающий правило вычерчивания линии объектом `Drawing`. Аналогичный метод `drawCircle()` предназначен для отображения окружностей. Следуя рекомендуемой здесь стратегии, мы готовим фундамент для появления других производных классов фигур, при вычерчивании которых могут потребоваться линии и окружности.

Где впервые была предложена стратегия реализации каждого правила в одном месте? Хотя многие упоминают о ней в своих публикациях, она утвердилась в фольклоре разработчиков объектно-ориентированных проектов уже давно и всегда представлялось как оптимальная практика проектирования. Совсем недавно Кент Бек (Kent Beck) назвал эту стратегию правилом "однажды и только однажды"⁶.

Он определяет это правило так.

- Система (и код, и тесты) должна иметь доступ ко всему, к чему, по вашему мнению, она должна иметь доступ.
- Система не должна содержать никакого дублирующегося кода.

Эти две составляющие вместе и представляют собой правило "однажды и только однажды".

⁶ Beck K. Extreme Programming Explained: Embrace Change, Reading, MA: Addison Wesley, 2000, с. 108, 109.

На рис. 9.13 показано, как абстракция **Shape** может быть отделена от реализации **Drawing**.

С точки зрения методов, новый вариант системы весьма похож на реализацию, построенную на наследовании (см. рис. 9.3). Главное отличие состоит в том, что здесь методы расположены в различных объектах.

Как уже было сказано в начале этой главы, мое замешательство при первом знакомстве с определением шаблона Bridge было вызвано неправильным пониманием термина "реализация". Поначалу я полагал, что этот термин относится к тому, как конкретная абстракция реализуется в программном коде.

Шаблон Bridge позволяет взглянуть на реализацию как на нечто, находящееся вне наших объектов, нечто *используемое* этими объектами. В результате мы получаем намного большую свободу за счет сокрытия вариации в реализации от вызывающей части программы. Разрабатывая объекты по этому принципу, я также обнаружил возможность размещения вариаций различного типа в отдельных иерархиях классов. Иерархия, изображенная на рис. 9.13 слева, включает вариации в абстракциях. Иерархия справа включает вариации в том, как будут реализованы эти абстракции. Такой подход отвечает новой парадигме создания объектов (использование анализа общности и изменчивости), упомянутой выше.

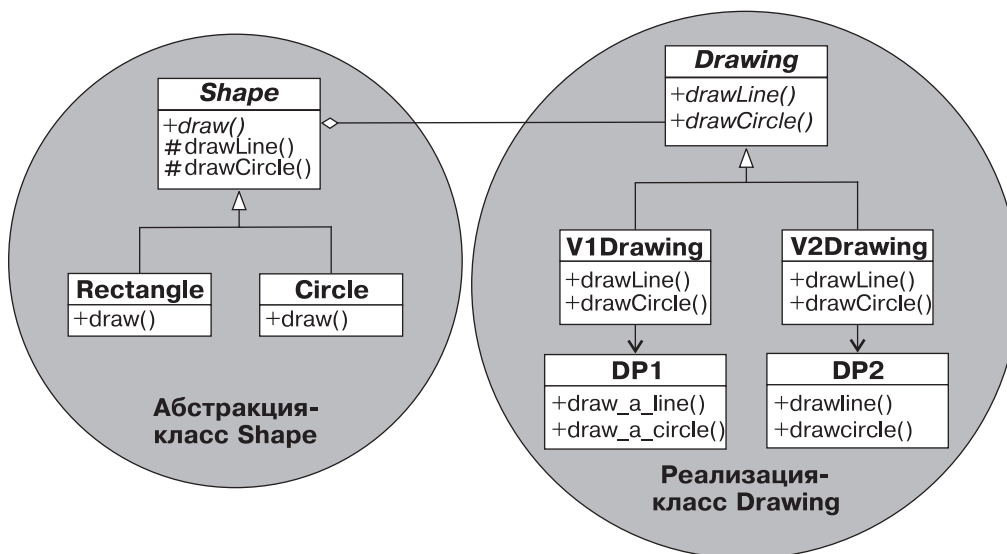


РИС. 9.13. Диаграмма классов, иллюстрирующая отделение абстракции от реализации

Очень легко визуализировать сказанное, если вспомнить, что в любой момент времени в системе существует только три взаимодействующих объекта – несмотря на то, что в ней реализовано около десятка различных классов (рис. 9.14).

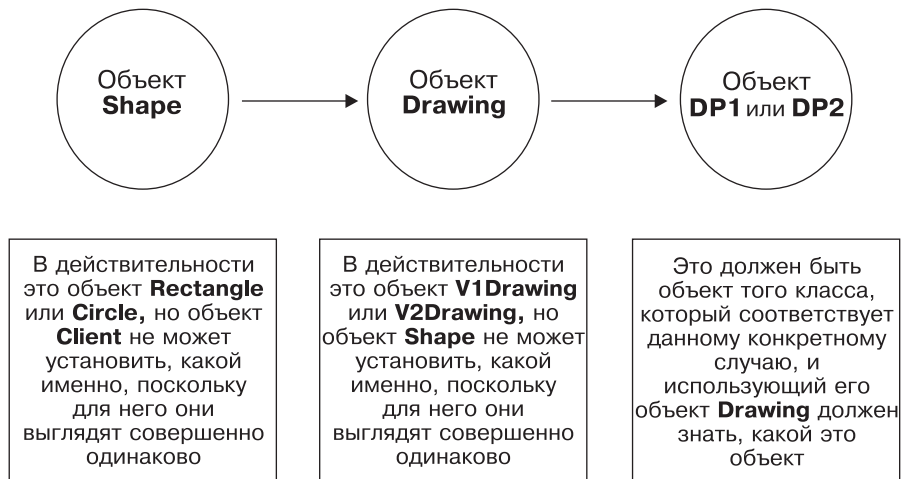


РИС. 9.14. В любой момент времени в программе существует только три объекта

Достаточно полный фрагмент программного кода для нашего примера представлен в листинге 9.3 для языка Java и в листингах 9.4–9.6, помещенных в приложение к этой главе, для языка C++.

Листинг 9.3. Фрагмент кода на языке Java

```

class Client {
    public static void main
        (String argv[]) {
        Shape r1, r2;
        Drawing dp;

        dp = new V1Drawing();
        r1 = new Rectangle(dp, 1, 1, 2, 2);

        dp = new V2Drawing();
        r2 = new Circle(dp, 2, 2, 3);

        r1.draw();
        r2.draw();
    }
}

abstract class Shape {
    abstract public draw() ;
    private Drawing _dp;

    Shape (Drawing dp) {
        _dp= dp;
    }
}

public void drawLine (
    double x1, double y1,
    double x2, double y2) {
    _dp.drawLine(x1, y1, x2, y2);
}

public void drawCircle (

```

```
        double x, double y, double r) {
            _dp.drawCircle(x, y, r);
        }
    }

    abstract class Drawing {
        abstract public void drawLine (
            double x1, double y1,
            double x2, double y2);
        abstract public void drawCircle (
            double x, double y, double r);
    }

    class V1Drawing extends Drawing {
        public void drawLine (
            double x1, double y1,
            double x2, double y2) {
            DP1.draw_a_line(x1, y1, x2, y2);
        }
        public void drawCircle (
            double x, double y, double r) {
            DP1.draw_a_circle(x, y, r);
        }
    }

    class V2Drawing extends Drawing {
        public void drawLine (
            double x1, double y1,
            double x2, double y2) {
            // У программы DP2 порядок аргументов отличается
            // и они должны быть перестроены
            DP2.drawline(x1, x2, y1, y2);
        }
        public void drawCircle (
            double x, double y, double r) {
            DP2.drawcircle(x, y, r);
        }
    }

    class Rectangle extends Shape {
        public Rectangle (
            Drawing dp,
            double x1, double y1,
            double x2, double y2) {
            super(dp) ;
            _x1 = x1; _x2 = x2;
            _y1 = y1; _y2 = y2;
        }

        public void draw () {
            drawLine(_x1, _y1, _x2, _y1);
            drawLine(_x2, _y1, _x2, _y2);
            drawLine(_x2, _y2, _x1, _y2);
            drawLine(_x1, _y2, _x1, _y1);
        }
    }

    class Circle extends Shape {
        public Circle (
            Drawing dp,
            double x, double y, double r) {
            super(dp) ;
            _x = x; _y = y; _r = r ;
        }
    }
```

```
    public void draw () {
        drawCircle(_x, _y, _r);
    }
}

// Ниже приведена упрощенная реализация
// для классов DP1 and DP2

class DP1 {
    static public void draw_a_line (
        double x1, double y1,
        double x2, double y2) {
        // Реализация
    }
    static public void draw_a_circle(
        double x, double y, double r) {
        // Реализация
    }
}

class DP2 {
    static public void drawline (
        double x1, double x2,
        double y1, double y2) {
        // Реализация
    }
    static public void drawcircle (
        double x, double y, double r) {
        // Реализация
    }
}
```

Особенности использования шаблона Bridge

Теперь, когда мы проанализировали, как шаблон Bridge работает, стоит посмотреть на него с более концептуальной точки зрения. Как показано на рис. 9.13, шаблон включает две части — абстрактную (со своими производными классами) и реализации. При проектировании с использованием шаблона Bridge полезно всегда помнить об этих двух частях. Интерфейс в части реализации следует разрабатывать с учетом особенностей различных производных классов того абстрактного класса, который этот интерфейс будет поддерживать. Обратите внимание на то, что проектировщик не обязательно должен помещать в интерфейс реализации всех возможных производных классов абстрактного класса (это еще одна возможная причина возникновения "паралича от анализа"). Следует принимать во внимание только те производные классы, которые действительно необходимо поддерживать. Не раз и не два авторы этой книги получали подтверждение, что даже небольшое усилие по увеличению гибкости в этой части проекта существенно его улучшает.

Замечание. В языке C++ реализация шаблона Bridge должна осуществляться только с помощью абстрактного класса, определяющего открытый интерфейс. В языке Java могут использоваться как абстрактный класс, так и интерфейс. Выбор зависит от того, дает ли преимущество разделение в реализации общих черт абстрактных классов. (Подробности — в книге "Peter Coad, *Java Design*", описание которой можно найти в главе 22.)

Дополнительные замечания о шаблоне Bridge

Обратите внимание на то, что решение, представленное на рис. 9.12 и 9.13, объединяет шаблоны Adapter и Bridge. Это сделано из-за того, что графические программы, которые необходимо использовать, были заранее жестко заданы. В этих программах уже есть интерфейсы, с которыми мы вынуждены работать. Поэтому, чтобы можно было работать с ними одинаково, необходимо воспользоваться шаблоном Adapter.

Несмотря на то что на практике шаблон Adapter зачастую оказывается включенным в шаблон Bridge, он, тем не менее, вовсе не является частью шаблона Bridge.

Основные характеристики шаблона Bridge

Назначение	Отделение набора объектов реализаций от набора объектов, использующих их
Задача	Классы, производные от абстрактного класса, должны использовать множество классов реализации этой абстракции, не вызывая при этом лавинообразного увеличения количества классов в системе
Способ решения	Определение интерфейса для всех используемых версий реализации и использование его во всех классах, порожденных от абстрактного
Участники	Класс <i>Abstraction</i> определяет интерфейс для объектов, представляющих сторону реализации. Класс <i>Implementor</i> определяет интерфейс для конкретных классов реализации. Классы, производные от класса <i>Abstraction</i> , используют классы, производные от класса <i>Implementor</i> , не интересуясь тем, с каким именно классом <i>ConcreteImplementorX</i> они имеют дело
Следствия	Отделение классов реализаций от объектов, которые их используют, упрощает расширение системы. Объекты-клиенты ничего не знают об особенностях конкретных реализаций
Реализация	<ul style="list-style-type: none"> • Инкапсуляция вариантов реализации в абстрактном классе. • Включение методов работы с ним в базовый абстрактный класс, представляющий абстракцию, подлежащую реализации. <p><i>Замечание.</i> В языке Java вместо абстрактного класса, подлежащего реализации, можно использовать интерфейс</p>

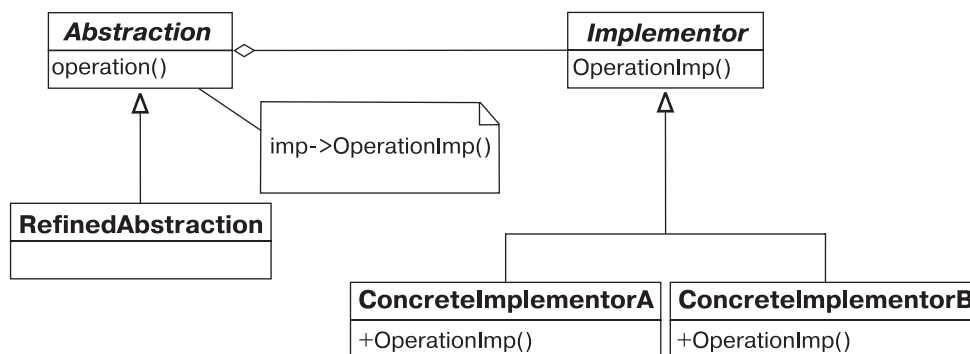


РИС. 9.15. Стандартное упрощенное представление шаблона Bridge

Когда несколько шаблонов интегрируются в одно целое (подобно шаблонам Bridge и Adapter в нашем примере), результатом является так называемый составной

шаблон проектирования.⁷ Это позволяет нам говорить о шаблонах, построенных из шаблонов проектирования!

Еще один интересный момент, который следует отметить, заключается в том, что объекты, представляющие абстракцию (*Shape*), получают доступ к функциям реализации уже после их инициации. Эта особенность не является характерной чертой данного шаблона, однако встречается очень часто.

Теперь, когда мы разобрались, что же такое шаблон Bridge, будет полезно вновь обратиться к работе "банды четырех", а именно — к разделу, в котором описывается реализация шаблона. Здесь обсуждаются различные проблемы, касающиеся того, как объекты абстрактной части шаблона создают и/или используют объекты части реализации.

Иногда при применении шаблона Bridge требуется организовать совместное использование объектов части реализации несколькими объектами, принадлежащими различным абстракциям.

- В языке Java это не вызывает никаких проблем, поскольку, когда все объекты стороны абстракции удаляются, программа автоматической сборки мусора устанавливает, что объекты на стороне реализации также больше не нужны, и удаляет их.
- В языке C++ необходимо каким-то образом управлять объектами реализации, для чего существует много способов. Например, можно организовать хранение счетчика ссылок или даже применить шаблон Singleton. Но все же лучше, когда нет необходимости прилагать дополнительные усилия для решения этого вопроса. Это иллюстрирует еще одно важное преимущество автоматической сборки мусора.

Несмотря на то что решение, которое было получено с применением шаблона Bridge, намного превосходит первоначальное, оно все же несовершенно. Оценить качество проекта можно проанализировав, насколько эффективно он позволяет вносить изменения в систему. При использовании шаблона Bridge внесение новых версий реализации существенно упрощается. Программисту достаточно определить новый конкретный класс и реализовать его. Больше никаких изменений не потребуется.

Однако ситуация оказывается существенно хуже, если в систему необходимо включить новый конкретный вариант абстракции. Безусловно, может оказаться, что для отображения нового типа фигуры будет достаточно тех функций реализации, которые уже существуют в системе. Однако это может быть и такая фигура, отображение которой потребует использования новых функций графической программы. Например, может потребоваться организовать в системе поддержку эллипсов. Существующий класс *Drawing* не содержит метода, позволяющего корректно отображать эллипсы. В этом случае потребуются внести изменения и на стороне реализации. Однако нам, по крайней мере, понятно, как это происходит — модификация интерфейса

⁷ Составные (*compound*) шаблоны проектирования ранее было принято называть сложными (*composite*), но сейчас предпочтение отдается названию составные, чтобы избежать путаницы с шаблоном *Composite*.

Подробности — "Riehl, D., Composite Design Patterns, In, *Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '97)*, New York: ACM Press, 1997, с. 218–228". Кроме того, представляет интерес публикация "Composite Design Patterns (They Aren't What You Think), *C++ Report*, June 1998".

класса *Drawing* (или интерфейса *Drawing* в языке Java) и соответствующие модификации всех его производных классов. Таким образом, место внесения изменений точно локализуется, что снижает риск появления нежелательных побочных эффектов.

Подсказка. Использование шаблонов не всегда означает автоматическое получение самых совершенных решений. Однако благодаря тому, что шаблоны представляют собой коллективный опыт многих проектировщиков, накопленный за долгие годы, они часто оказываются лучше тех решений, которые можно было бы быстро придумать самостоятельно.

В реальном мире новые проекты редко должны включать поддержку многовариантной реализации с самого начала. Иногда мы знаем, что появление новых реализаций *возможно*, но возникают они всегда неожиданно. Один из подходов состоит в том, чтобы заранее подготовиться к многовариантной реализации, всегда разделяя систему на стороны абстракции и реализации. В результате получится хорошо структурированное приложение.

Но этот подход не стоит рекомендовать, поскольку он ведет к нежелательному увеличению количества классов в системе. Важно создавать программный код таким способом, чтобы в случае, если многовариантная реализация все же потребуется (а она будет требоваться часто), существовал простой способ модифицировать систему, включив в нее шаблон Bridge. Модификация кода для улучшения его структуры без добавления функций называется *рефакторингом* (refactoring). По определению Мартина Фаулера (Martin Fowler), "рефакторинг — это процесс такого изменения программного обеспечения, при котором внешнее поведение кода не изменяется, но его внутренняя структура все же улучшается"⁸.

При проектировании программного кода я всегда уделял внимание возможности его рефакторинга, следуя стратегии "одно правило, одно место". Хорошим примером применения подобного подхода является метод `DrawLine()`. Хотя фактически фрагменты кода размещены в различных местах, перемещаться по нему совсем просто.

Рефакторинг

Рефакторинг часто применяется в объектно-ориентированном проектировании. Однако данное понятие принадлежит не только ООП. Рефакторинг — это просто модификация кода с целью улучшить его структуру без добавления новых функций.

При выведении шаблона мы взяли две изменяемые части проекта (фигуры и графические программы) и инкапсулировали каждую из них в собственный абстрактный класс. Вариации в фигурах — в классе *Shape*, а вариации в графических программах были инкапсулированы в классе *Drawing*.

Посмотрим со стороны на две полиморфные структуры и спросим: "Что представляют данные абстрактные классы?". Очевидно, что класс *Shape* представляет различные виды фигур, а класс *Drawing* — то, как эти фигуры будут отображены. Таким образом, даже если к классу *Drawing* предъявляются новые требования (например, появляется необходимость реализовать поддержку эллипсов), между классами сохраняются ясные отношения.

⁸ Fowler M., Refactoring: Improving the Design of Existing Code, Reading, MA: Addison-Wesley, 2000, с. xvi.

Резюме

Для того чтобы изучить шаблон Bridge, мы рассмотрели проблему, при которой в предметной области присутствуют две вариации — геометрических фигур и графических программ. В заданной предметной области каждая вариация изменяется независимо. Трудности появились при попытке воспользоваться решением, построенным на учете всех возможных конкретных ситуаций взаимодействия. Подобное решение, примитивным образом использующее механизм наследования, приводит к созданию громоздкого проекта, характеризующегося сильной связанностью и слабой связностью, а следовательно, крайне неудобного в сопровождении.

При обсуждении шаблона Bridge мы следовали следующим стратегиям работы с вариациями.

- Найдите то, что изменяется, и инкапсулируйте это.
- Отдавайте предпочтение композиции, а не наследованию.

Выявление того, что изменяется, — важный этап изучения предметной области. В примере с графическими программами мы столкнулись с тем, что один набор вариаций использует другой. Это и есть показатель того, что в данной ситуации шаблон Bridge может оказаться весьма полезным.

В общем случае, чтобы определить, какой именно шаблон лучше использовать в данной ситуации, следует сопоставить его с характеристиками и поведением сущностей в проблемной области. Зная ответы на вопросы *зачем* и *что* в отношении всех известных вам шаблонов, несложно будет выбрать среди них именно те, которые позволят решить проблему. Шаблоны могут быть выбраны еще до того, как станет известен способ их реализации.

Применение шаблона Bridge позволяет получить более устойчивые проектные решения для представления элементов абстракции и реализации, упрощая их возможное последующее изменение.

Завершая обсуждение шаблона, будет полезным еще раз напомнить о тех принципах объектно-ориентированного проектирования, которые используются в шаблоне Bridge (табл. 9.2).

Таблица 9.2. Принципы ООП, используемые в шаблоне Bridge

Концепция	Обсуждение
Объекты отвечают только за самих себя	Существует несколько классов, представляющих фигуры, но каждый из них отображает сам себя (с помощью метода <code>draw()</code>). Классы Drawing отвечают за отображение элементов этих фигур
Абстрактный класс	Абстрактные классы используются для представления концепций. Реально в проблемной области мы имели дело с прямоугольниками и окружностями. Концепция "фигура" — это нечто, существующее только в нашей голове, средство для связывания двух реальных вещей между собой. Поэтому она была представлена в системе абстрактным классом Shape . Ни один экземпляр объекта класса Shape никогда не будет создан в системе, поскольку он не существует в проблемной

	области сам по себе — в ней существуют только прямоугольники и окружности. Это же относится и к графическим программам
Инкапсуляция через абстрактный класс	<p>В этой задаче у нас есть два примера инкапсуляции посредством создания абстрактного класса.</p> <ul style="list-style-type: none"> • Объект-клиент, обращающийся к шаблону Bridge, всегда будет иметь дело только с объектом класса, производного от класса <i>Shape</i>. Клиент не будет знать, с каким именно из конкретных подтипов класса <i>Shape</i> он взаимодействует — для клиента это всегда будет только класс <i>Shape</i>. Следовательно, имеет место инкапсуляция информации. Преимущество подобного подхода состоит в том, что появление нового типа фигуры в будущем никак не затрагивает объект-клиент.

Окончание табл. 9.2

Концепция	Обсуждение
	<ul style="list-style-type: none"> • Класс <i>Drawing</i> скрывает от класса <i>Shape</i> наличие в системе различных графических программ. На практике объекты со стороны абстракции смогут узнать, какая именно реализация используется, поскольку именно они создают соответствующие экземпляры объектов. В некоторых случаях это может оказаться полезным. Однако даже в тех случаях, когда это имеет место, знание о конкретной реализации ограничено функцией-конструктором объекта абстракции и, следовательно, легко модифицируемо
Одно правило, одно место	Абстрактный класс часто включает методы, которые непосредственно используют объекты из части реализации. Конкретные классы, производные от данного абстрактного, должны вызывать именно эти методы. Такой подход упрощает модификацию системы, когда в этом появляется необходимость, а также дает хорошую отправную точку в проектировании еще до полной реализации шаблона

Приложение. Примеры программного кода на языке C++

Листинг 9.4. Только прямоугольники

```
void Rectangle::draw () {
    drawLine(_x1, _y1, _x2, _y1);
    drawLine(_x2, _y1, _x2, _y2);
    drawLine(_x2, _y2, _x1, _y2);
}
```

```

        drawLine(_x1, _y2, _x1, _y1);
    }

void V1Rectangle::drawLine
(double x1, double y1,
 double x2, double y2) {
    DP1.draw_a_line(x1, y1, x2, y2);
}

void V2Rectangle::drawLine
(double x1, double y1,
 double x2, double y2) {
    DP2.drawline(x1, x2, y1, y2);
}

```

Листинг 9.5. Прямоугольники и окружности без использования шаблона Bridge

```

class Shape {
public: void draw () = 0;
}
class Rectangle : Shape {
public:
    void draw();
protected:
    void drawLine(
        double x1, y1, x2, y2) = 0;
}
void Rectangle::draw () {
    drawLine(_x1, _y1, _x2, _y1);
    drawLine(_x2, _y1, _x2, _y2);
    drawLine(_x2, _y2, _x1, _y2);
    drawLine(_x1, _y2, _x1, _y1);
}
// Классы V1Rectangle и V2Rectangle порождены от
// класса Rectangle. Заголовок файла не показан
void V1Rectangle::drawLine (
    double x1, y1, x2, y2) {
    DP1.draw_a_line(x1, y1, x2, y2);
}
void V2Rectangle::drawLine (
    double x1,y1, x2,y2) {
    DP2.drawline(x1,x2,y1,y2);
}

class Circle : Shape {
public:
    void draw() ;
protected:
    void drawCircle(
        double x, y, z) ;
}
void Circle::draw () {
    drawCircle();
}

// Классы V1Circle и V2Circle порождены от
// класса Circle. Заголовок файла не показан
void V1Circle::drawCircle (
    DP1.draw_a_circle(x, y, r);
}

```

```
void V2Circle::drawCircle (
    DP2.drawcircle(x, y, r);
}
```

Листинг 9.6. Фрагменты кода — реализация с применением шаблона Bridge

```
void main (String argv[]) {
    Shape *s1;
    Shape *s2;
    Drawing *dp1, *dp2;

    dp1= new V1Drawing;
    s1=new Rectangle(dp, 1, 1, 2, 2);

    dp2= new V2Drawing;
    s2= new Circle(dp, 2, 2, 4);

    s1->draw();
    s2->draw();

    delete s1; delete s2;
    delete dp1; delete dp2;
}

// Замечание. Управление памятью не тестировалось.
// Файлы Include не показаны.

class Shape {
public: draw() = 0;
private: Drawing *_dp;
}
Shape::Shape (Drawing *dp) {
    _dp = dp;
}
void Shape::drawLine(
    double x1, double y1,
    double x2, double y2)
    _dp -> drawLine(x1, y1, x2, y2);
}

Rectangle::Rectangle (Drawing *dp,
    double x1, y1, x2, y2) :
    Shape( dp) {
    _x1 = x1; _x2 = x2;
    _y1 = y1; _y2 = y2;
}
void Rectangle::draw () {
    drawLine(_x1, _y1, _x2, _y1);
    drawLine(_x2, _y1, _x2, _y2);
    drawLine(_x2, _y2, _x1, _y2);
    drawLine(_x1, _y2, _x1, _y1);
}

class Circle {
public: Circle (
    Drawing *dp,
    double x, double y, double r);
};

Circle::Circle (
    Drawing *dp,
```

```

    double x, double y,
    double r) : Shape(dp) {
        _x = x;
        _y = y;
        _r = r;
    }

    Circle::draw () {
        drawCircle(_x, _y, _r);
    }

    class Drawing {
    public: virtual void drawLine (
        double x1, double y1,
        double x2, double y2) = 0;
    };

    class V1Drawing :
    public Drawing {
    public: void drawLine (
        double x1, double y1,
        double x2, double y2);
        void drawCircle(
            double x, double y, double r);
    }
    ;
    void V1Drawing::drawLine (
        double x1, double y1,
        double x2, double y2) {
        DP1.draw_a_line(x1, y1, x2, y2);
    }

    void V1Drawing::drawCircle (
        double x1, double y, double r) {
        DP1.draw_a_circle (x, y, r);
    }

    class V2Drawing : public
    Drawing {
    public:
        void drawLine (
            double x1, double y1,
            double x2, double y2);
        void drawCircle(
            double x, double y, double r);
    };

    void V2Drawing::drawLine (
        double x1, double y1,
        double x2, double y2) {
        DP2.drawline(x1, x2, y1, y2);
    }

    void V2Drawing::drawCircle (
        double x, double y, double r) {
        DP2.drawcircle(x, y, r);
    }

    // Реализация показана для классов DP1 и DP2
    class DP1 {
    public:
        static void draw_a_line (
            double x1, double y1,

```

```
        double x2, double y2);  
    static void draw_a_circle (  
        double x, double y, double r);  
};  
  
class DP2 {  
    public:  
        static void drawline (  
            double x1, double x2,  
            double y1, double y2);  
        static void drawcircle (  
            double x, double y, double r);  
};
```

Шаблон Abstract Factory

Введение

Продолжим изучение шаблонов и рассмотрим шаблон Abstract Factory (абстрактная фабрика), предназначенный для создания семейств объектов.

В этой главе мы выполним следующее.

- Выведем шаблон Abstract Factory на основании конкретного примера.
- Рассмотрим ключевые особенности этого шаблона.
- Применим шаблон Abstract Factory к решению задачи о различных версиях САПР.

Назначение шаблона проектирования Abstract Factory

"Банда четырех" определяет назначение шаблона Abstract Factory как "предоставление интерфейса для создания семейств связанных между собой или зависимых друг от друга объектов без указания их конкретных классов".¹

Иногда встречаются ситуации, когда требуется координированно создать экземпляры нескольких объектов. Например, для отображения интерфейса пользователя система может использовать два различных набора объектов для работы в двух различных операционных системах. Применение шаблона проектирования Abstract Factory гарантирует, что система всегда реализует именно тот набор объектов, который отвечает данной ситуации.

Описание шаблона проектирования Abstract Factory на примере

Представим себе, что перед нами поставлена задача создать компьютерную систему для отображения на мониторе и вывода на печать геометрических фигур, описание которых хранится в базе данных. Значение уровня разрешения при выводе на печать и отображении фигур зависит от характеристик того компьютера, на котором функционирует система, — например, быстродействия процессора и объема доступной оперативной памяти. Иными словами, система должна самостоятельно определять уровень требований, предъявляемых ею к конкретному компьютеру.

¹ Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software, Reading, MA: Addison-Wesley, 1995, с. 87.

Решение данной задачи сводится к тому, что система должна управлять набором используемых драйверов операционной системы. На маломощных компьютерах она должна загружать драйверы, обеспечивающие низкое разрешение, а на более мощных — другие драйверы, обеспечивающие более высокое разрешение (табл. 10.1).

Таблица 10.1. Схема использования драйверов для компьютеров различной мощности

Тип устройства	Компьютер с низкой производительностью	Компьютер с высокой производительностью
Дисплей	Объект LRDD	Объект HRDD
	Драйвер дисплея с низкой разрешающей способностью (Low-Resolution Display Driver)	Драйвер дисплея с высокой разрешающей способностью (High-Resolution Display Driver)
Принтер	Объект LRPD	Объект HRPD
	Драйвер принтера с низкой разрешающей способностью (Low-Resolution Print Driver)	Драйвер принтера с высокой разрешающей способностью (High-Resolution Print Driver)

В данном примере различные наборы драйверов взаимно исключают друг друга, но это требование не является обязательным. Иногда встречаются ситуации, когда различные семейства будут включать объекты одного и того же класса. Например, на компьютере со средними характеристиками система может использовать драйвер монитора с низким разрешением (LRDD) и, одновременно, драйвер печати с высоким разрешением (HRPD).

Какое семейство объектов использовать в каждом конкретном случае, определяется особенностями проблемной области. В нашем случае концепция объединения объектов в семейства строится на требованиях, которые каждый из объектов предъявляет к системе.

- Семейство драйверов с низкой разрешающей способностью — объекты LRDD и LRPD. Это драйверы, предъявляющие низкие требования к оборудованию компьютера.
- Семейство драйверов с высокой разрешающей способностью — объекты HRDD и HRPD. Это драйверы, предъявляющие повышенные требования к оборудованию компьютера.

Первая мысль, которая приходит в голову при обдумывании реализации, — использовать для управления выбором драйверов переключатель, как показано в листинге 10.1.

Листинг 10.1. Вариант 1. Использование переключателя для выбора типа драйвера

// Фрагмент кода на языке Java

```
class ApControl {
    void doDraw () {
        switch (RESOLUTION) {
            case LOW:
```



```
        // Использование драйвера LRDD
    case HIGH:
        // Использование драйвера HRDD
    }
}
void doPrint () {
    switch (RESOLUTION) {
    case LOW:
        // Использование драйвера LRPD
    case HIGH:
        // Использование драйвера HRPD
    }
}
}
```

Конечно, это вполне работоспособное решение, но оно не лишено кое-каких недостатков. Так, правило определения, какой именно драйвер использовать, смешано здесь с собственно использованием этого драйвера. В результате возникают проблемы и со связанностью, и со связностью.

- *Сильная связанность.* Если изменятся правила определения допустимого разрешения (например, необходимо будет ввести средний уровень разрешения), то потребуются изменить программный код в двух местах, чтобы не разрывать существующие связи.
- *Слабая связность.* Методам `doDraw` и `doPrint` приходится дважды передавать соответствующий набор параметров — оба эти метода должны самостоятельно определить, какой именно драйвер следует использовать, а затем отобразить геометрическую фигуру.

Конечно, сильная связанность и слабая связность в настоящее время не представляют большой проблемы, однако они, безусловно, способствуют повышению затрат на сопровождение в будущем. Кроме того, в реальной ситуации мы, вероятнее всего, столкнемся с необходимостью модифицировать код в большем количестве мест, чем показано в этом простом примере.

Еще одна возможная альтернатива состоит в использовании механизма наследования. Мы можем создать два различных класса **ApControl**: один из них будет использовать драйверы с низкой разрешающей способностью, а другой — с высокой. Оба класса будут производными от одного и того же абстрактного класса, что позволит воспользоваться одним и тем же программным кодом. Этот вариант решения представлен на рис. 10.1.

Применение наследования может дать хороший эффект в этом простом случае, однако в целом второй вариант решения также имеет много недостатков, как и первый вариант с переключателем.

- *Комбинаторный взрыв.* Для каждого из существующих семейств и для всех новых семейств, которые появятся в будущем, необходимо создавать новый конкретный класс (т.е. новую конкретную версию класса **ApControl**).

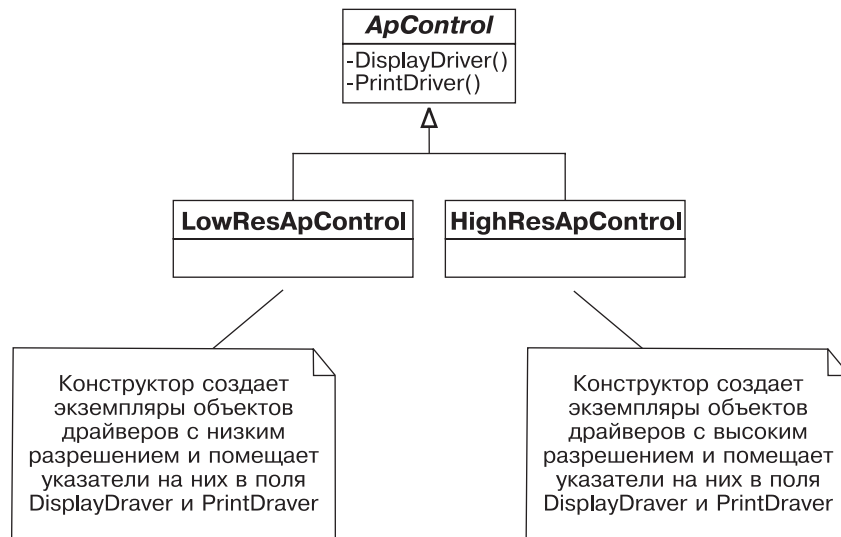


РИС. 10.1. Вариант 2. Реализация вариаций с помощью механизма наследования

- *Неясное назначение.* Вновь образованные классы не способны помочь нам разобраться в том, что происходит в системе. Каждый специализированный класс жестко привязан к конкретному случаю. Чтобы обеспечить простоту сопровождения создаваемого программного кода в будущем, необходимо приложить усилия и сделать назначение каждого фрагмента как можно более понятным. Тогда не потребуется тратить много времени на попытки вспомнить, какой участок кода какие функции выполняет.
- *Композиция объектов предпочтительней наследования.* Наконец, выбранный подход нарушает важнейшее правило разработки — предпочтительнее использовать композицию объектов, а не наследование.

Наличие переключателя свидетельствует о необходимости обращения к абстракции

Чаще всего наличие переключателя свидетельствует о необходимости поддержки полиморфного поведения или о неверном размещении реализации обязательств. В любом случае будет полезно подыскать более общее решение — скажем, ввести новый уровень абстракции или передать соответствующие обязательства другим объектам.

Опыт показывает, что наличие в коде переключателя часто указывает на необходимость применения абстракции. В нашем примере объекты LRDD и HRDD являются драйверами для дисплея, а объекты LRPD и HRPD — драйверами для вывода на принтер. Соответствующие абстракции можно назвать *DisplayDriver* (драйвер дисплея) и *PrintDriver* (драйвер принтера). На рис. 10.2 представлено концептуальное решение для такого подхода. Решение следует считать концептуальным, поскольку классы LRDD и HRDD в действительности не являются производными от одного и того же абстрактного класса.

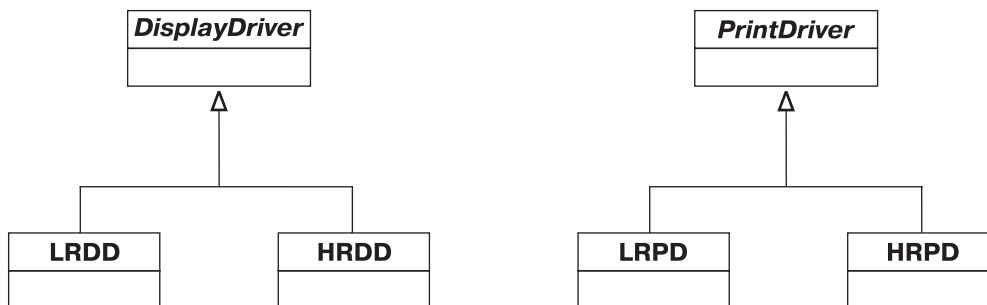


РИС. 10.2. Типы драйверов и соответствующие абстракции

Замечание. На этом этапе мы можем не обращать внимания на то, что классы на схеме являются производными от различных классов, поскольку уже знаем, как можно использовать шаблон Adapter для адаптации драйверов, чтобы создать иллюзию их принадлежности к общему абстрактному классу.

В результате подобного определения объектов класс **ApControl** сможет использовать классы **DisplayDriver** и **PrintDriver** без применения переключателя. Теперь понять работу класса **ApControl** будет намного проще, поскольку отпала необходимость анализировать, какой именно тип драйвера он должен использовать. Другими словами, класс **ApControl** может обращаться к классу **DisplayDriver** или классу **PrintDriver**, не заботясь о разрешении, обеспечиваемом драйверами.

Соответствующая схема представлена на рис. 10.3, а программный код на языке Java приведен в листинге 10.2.

Листинг 10.2. Фрагмент программного кода. Решение задачи с помощью методов полиморфизма

```
// Фрагмент кода на языке Java

class ApControl {
    . . .
    void doDraw () {
        . . .
        myDisplayDriver.draw();
    }
    void doPrint () {
        . . .
        myPrintDriver.print();
    }
}
```

Остается нерешенным только один вопрос — как создать соответствующие объекты в каждом конкретном случае?

Можно возложить эту задачу на класс **ApControl**, однако подобное решение вызовет проблемы с поддержкой системы в будущем. Если потребуется организовать работу с новым набором объектов, класс **ApControl** неизбежно придется модифицировать. Однако если для создания экземпляров нужных нам объектов использовать некоторый специализированный "объект-фабрику", можно решить проблему, связанную с появлением нового семейства объектов.

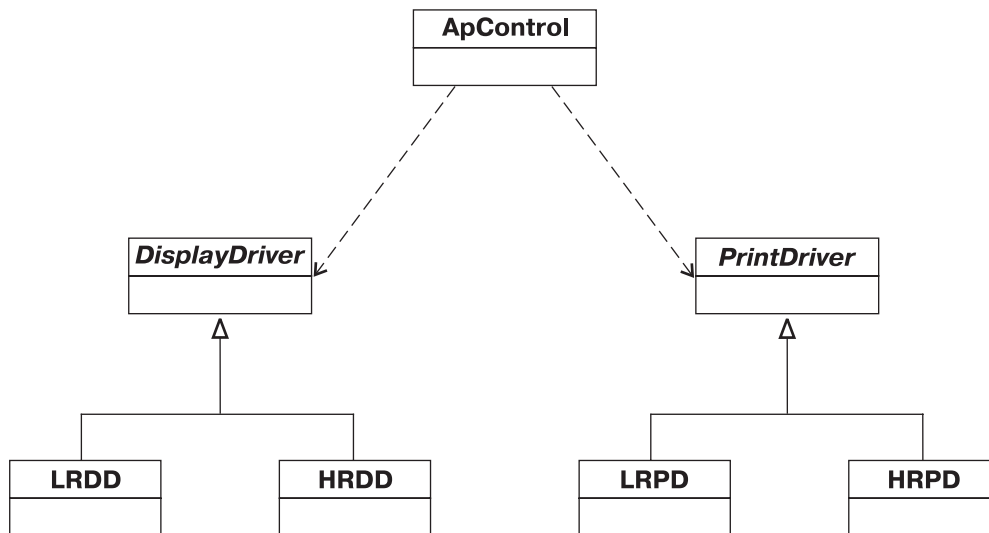


РИС. 10.3. Использование драйверов классом *ApControl* в идеальной ситуации

В нашем примере объект-фабрика будет использоваться для управления созданием требуемого семейства объектов драйверов. Объект *ApControl* будет обращаться к другому объекту — объекту-фабрике, чтобы создать объекты драйверов дисплея и печати с требуемым разрешением, определяемым в зависимости от характеристик используемого компьютера. Схема подобного взаимодействия представлена на рис. 10.4.

Теперь с точки зрения объекта *ApControl* все выглядит предельно просто. Функции определения, какой именно тип драйвера и как требуется создать, возложена на объект *ResFactory*. Хотя перед нами по-прежнему стоит задача написать программный код, реализующий все необходимые действия по созданию требуемых объектов, мы выполнили декомпозицию проблемы с распределением обязательств между отдельными участниками. Класс ***ApControl*** должен знать, как работать с соответствующими случаю объектами, а класс ***ResFactory*** — решать, какие именно объекты являются соответствующими в каждом конкретном случае. Теперь можно использовать либо различные объекты-фабрики, либо один-единственный объект, содержащий переключатель. В любом случае данный подход будет лучше прежних.

Такое решение вносит в систему необходимую законченность. Назначение объекта класса ***ResFactory*** состоит лишь в том, чтобы создавать соответствующие объекты драйверов, а назначение объекта класса ***ApControl*** — в том, чтобы их использовать.

Существуют различные способы избежать использования переключателя в теле класса ***ResFactory***. Их применение позволит вносить возможные изменения, не затрагивая уже существующие объекты-фабрики. Например, можно инкапсулировать возможные вариации за счет определения абстрактного класса, представляющего общую концепцию объекта-фабрики. В нашем примере класс ***ResFactory*** реализует два различных типа поведения (или метода).

- Создание объекта требуемого драйвера дисплея.
- Создание объекта требуемого драйвера печати.

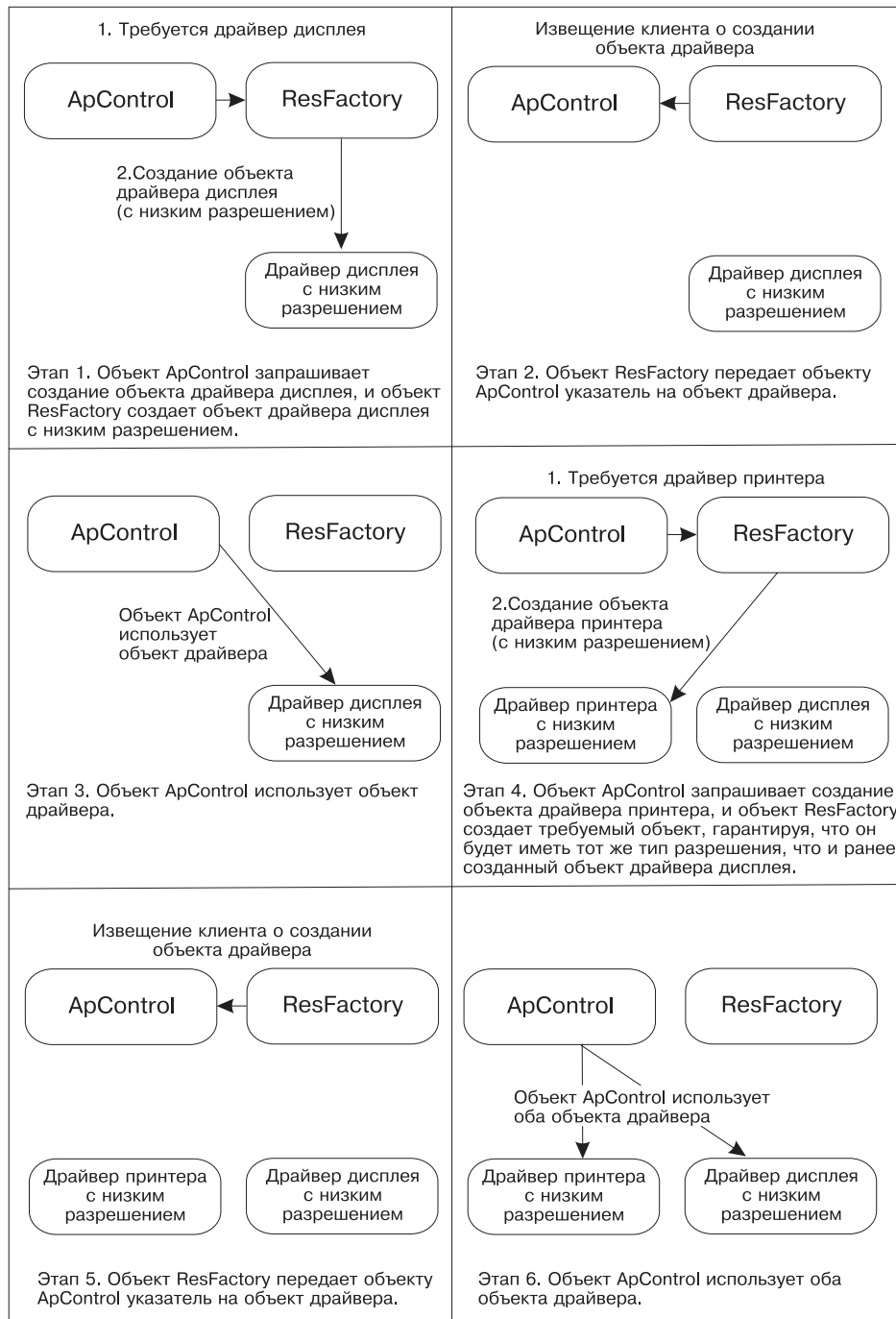


РИС. 10.4. Объект *ApControl* создает объекты драйверов с помощью объекта-фабрики *ResFactory*

В системе абстрактный класс *ResFactory* может быть представлен объектом одного из двух конкретных классов, каждый из которых является производным от этого абстрактного класса и имеет требуемые открытые методы — как показано на рис. 10.5.

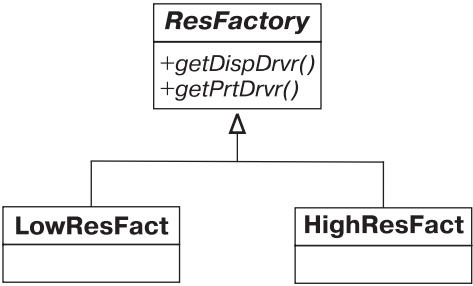


РИС. 10.5. Абстрактный класс *ResFactory* инкапсулирует существующие вариации

Реализация стратегии анализа при проектировании

Ниже описаны три ключевые стратегии проектирования и способы их применения при использовании шаблона Abstract Factory.

Стратегия анализа	Реализация при проектировании
Найдите то, что изменяется, и инкапсулируйте это	В нашем примере изменчивость заключается в выборе типа объекта драйвера, который требуется использовать. Она была инкапсулирована в абстрактный класс <i>ResFactory</i>
Предпочтительное использование композиции, а не наследования	Вместо создания двух различных типов объектов <i>ApControl</i> , изменчивость вынесена в единственный абстрактный класс <i>ResFactory</i> , используемый классом <i>ApControl</i>
Проектирование интерфейсов, а не реализации	Класс <i>ApControl</i> знает, как потребовать от класса <i>ResFactory</i> создать соответствующие экземпляры объектов драйверов, но не знает как именно класс <i>ResFactory</i> реализует это требование

Реализация шаблона проектирования Abstract Factory

В листинге 10.3 показано, как объекты шаблона Abstract Factory могут быть реализованы в нашем примере.

Листинг 10.3. Пример реализации объектов класса *ResFactory* на языке Java

```
class LowResFact extends ResFactory {
    DisplayDriver public
    getDispDrvr() {
        return new lrdd();
    }
}
```

```
    }

    PrintDriver public
    getPrtdrvr() {
        return new lrpdr();
    }
}

class HighResFact extends ResFactory {

    DisplayDriver public
    getDispDrvr() {
        return new hrdd();
    }

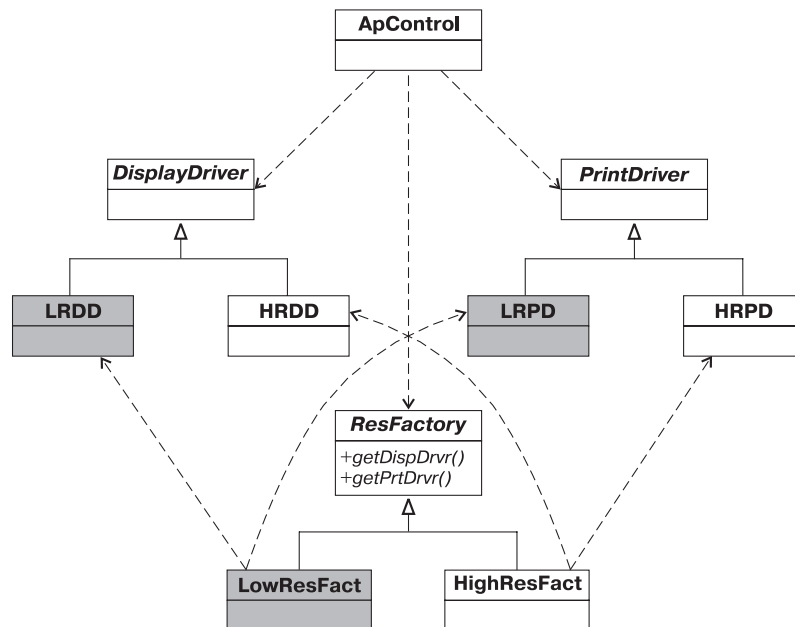
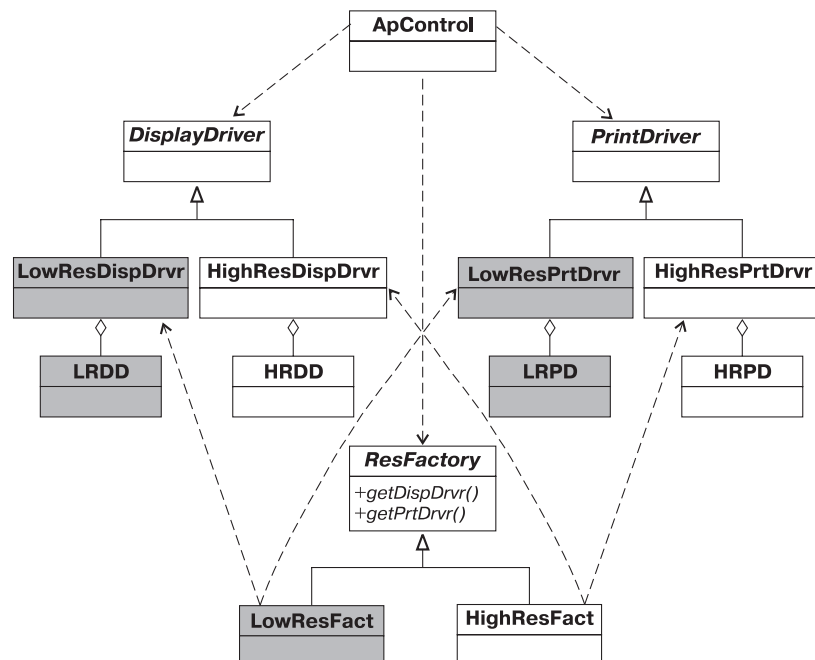
    PrintDriver public
    getPrtdrvr() {
        return new hrpd();
    }
}
```

Для завершения работы над найденным решением необходимо организовать диалог объекта класса **ApControl** с соответствующим объектом-фабрикой (**LowResFact** или **HighResFact**), как показано на рис. 10.6. Обратите внимание на то, что класс **ResFactory** является абстрактным, и именно это сокрытие реализации классом **ResFactory** составляет суть работы данного шаблона. Отсюда и его название — Abstract Factory (абстрактная фабрика).

Объекту класса **ApControl** предоставляется ссылка на объект либо класса **LowResFact**, либо класса **HighResFact**. Он запрашивает создание этого объекта при необходимости получить доступ к соответствующему драйверу. Объект-фабрика создает экземпляр объекта драйвера, самостоятельно определяя его тип (для низкого или высокого разрешения). Причем в этом случае объекту **ApControl** даже нет необходимости знать, какой именно драйвер (с низкой разрешающей способностью или высокой) будет создан, так как со всеми он работает совершенно одинаково.

Мы обошли вниманием один важный вопрос: классы **LRDD** и **HRDD** могут быть производными от разных абстрактных классов (это же замечание может быть справедливо и для классов **LRPD** и **HRPD**). Однако тем, кто знает шаблон Adapter, решить этот вопрос не составит труда. Можно воспользоваться общей схемой, представленной на рис. 10.6, просто добавив к ней элементы адаптации объектов драйверов (рис. 10.7).

Данная реализация проекта по существу неотличима от прежней. Единственное различие состоит в том, что теперь объекты-фабрики создают экземпляры объектов тех классов, которые были добавлены нами для адаптации объектов предыдущего варианта схемы. Это очень важный метод моделирования. Подобное совместное использование шаблонов Adapter и Abstract Factory позволяет обращаться с концептуально сходными объектами так, как будто они близкородственны друг другу, даже если на самом деле это не так. Такой подход позволяет использовать шаблон Abstract Factory в самых разных ситуациях.

РИС. 10.6. Промежуточное решение с использованием шаблона *Abstract Factory*РИС. 10.7. Решение с применением шаблонов *Abstract Factory* и *Adapter*

Вот характерные особенности полученного шаблона.

- Клиентский объект знает только, к какому объекту следует обратиться для создания требуемых ему объектов и как их использовать.
- В классе абстрактной фабрики уточняется, экземпляры каких объектов должны быть созданы, для чего определяются отдельные методы для различных типов объектов. Как правило, объект абстрактной фабрики включает отдельный метод для каждого существующего типа объектов.
- Конкретные объекты-фабрики определяют, какие именно объекты должны создаваться.

Дополнительные замечания о шаблоне Abstract Factory

Выбрать, какой именно объект-фабрика должен использоваться, в действительности то же самое, что и определить, какое следует использовать семейство объектов. Например, в обсуждавшейся выше задаче о драйверах мы имели дело с двумя семействами драйверов — одно для систем с низкой разрешающей способностью, а другое для систем с высокой разрешающей способностью. Как узнать, какой набор драйверов следует выбрать? В этом случае, вероятно, потребуется обратиться к файлу конфигурации системы, содержащему описание характеристик компьютера. В программу придется добавить несколько строк программного кода, где на основе полученной информации будет выбираться объект-фабрика с требуемым набором свойств.

Шаблон Abstract Factory также может быть использован совместно с подсистемами различных приложений. В этом случае объект-фабрика будет пропускным пунктом для подсистем, сообщая им, какие именно объекты из их состава должны использоваться. Как правило, главная система знает, какое семейство объектов должно использоваться каждой подсистемой, поэтому до обращения к некоторой подсистеме можно будет создать требуемый экземпляр объекта-фабрики.

Основные характеристики шаблона Abstract Factory

Назначение	Требуется организовать работу с семействами или наборами объектов для определенных объектов-клиентов (или отдельных случаев)
Задача	Необходимо создать экземпляры объектов, принадлежащих заданному семейству
Способ решения	Координация процесса создания семейств объектов. Предусматривается выделение реализации правила создания тех или иных семейств объектов из объекта-клиента, который в дальнейшем будет использовать созданные объекты, в отдельный объект
Участники	Класс AbstractFactory определяет интерфейс для создания каждого из объектов заданного семейства. Как правило, каждое семейство создается с помощью собственного конкретного уникального класса ConcreteFactory
Следствия	Шаблон отделяет правила, <i>какие</i> объекты нужно использовать, от правил, <i>как</i> эти объекты следует использовать
Реализация	Определяется абстрактный класс, описывающий, какие объекты должны быть созданы. Затем реализуется по одному конкретному классу для каждого семейства объектов (рис. 10.8). Для решения этой задачи также могут использоваться таблицы базы данных или файлы

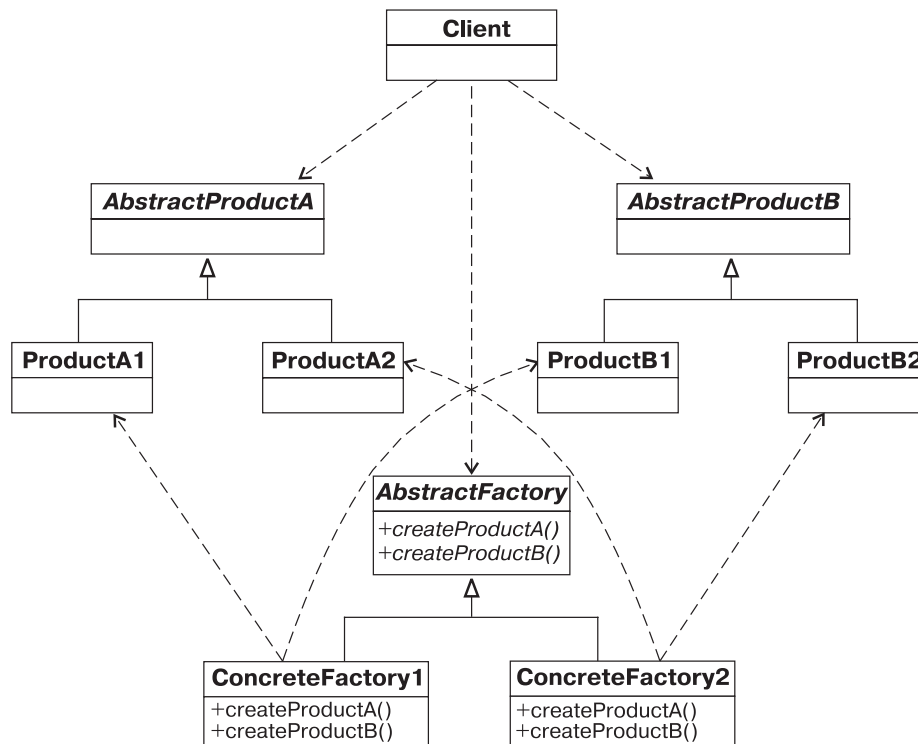


РИС. 10.8. Стандартное упрощенное представление шаблона *Abstract Factory*

На рис. 10.8 показано, что объект **Client** использует объекты, производные от двух различных серверных классов (**AbstractProductA** и **AbstractProductB**). Такой подход упрощает структуру системы, скрывая реализацию, чем упрощает ее сопровождение.

- Объект-клиент не знает, с какой конкретной реализацией серверного объекта он имеет дело, так как ответственность за создание серверного объекта несет объект-фабрика.
- Объект-клиент не знает даже того, к какому конкретному объекту-фабрике он обращается, поскольку он взаимодействует с абстрактным классом **AbstractFactory**, который может быть представлен объектом конкретных классов **ConcreteFactory1** или **ConcreteFactory2**, но каким именно, объект-клиент знать не может.

От объекта класса **Client** полностью скрыта (инкапсулирована) информация о том, какой из серверных объектов используется. Это позволяет в будущем легко вносить изменения в процедуру совершения выбора, так как объект-клиент остается не затронутым.

Шаблон **Abstract Factory** ярко демонстрирует новый вид декомпозиции — декомпозицию ответственности. Подобный подход позволяет разделить задачу на две функциональные стороны.

- Первая сторона, использующая конкретные объекты (класс **ApControl**).
- Вторая сторона, принимающая решение о том, какие именно объекты следует использовать (класс **AbstractFactory**).

Применение шаблона Abstract Factory целесообразно в тех случаях, когда проблемная область включает несколько семейств объектов, причем каждое из семейств используется при различных обстоятельствах.

Семейства могут быть определены на основе одного или нескольких критериев.

Приведем несколько примеров.

- Различные операционные системы (при создании многоплатформенных приложений).
- Различные требования к производительности.
- Различные версии приложений.
- Различные категории пользователей приложения.

Как только семейства и их члены будут идентифицированы, следует принять решение о методе реализации каждого конкретного случая (т.е. каждого семейства). В нашем примере для этой цели определен абстрактный класс, на который возложена ответственность за выбор конкретного типа семейства классов, для которых и будут созданы экземпляры объектов. Затем для каждого семейства от этого абстрактного класса был определен конкретный производный класс, реализующий создание экземпляров объектов для классов — членов семейства.

Иногда встречаются ситуации, когда семейства объектов существуют, но нет необходимости управлять созданием их экземпляров с помощью классов, порожденных специально для каждого семейства. Возможно, более подходящим окажется одно из следующих более динамичных решений.

- Можно использовать файл конфигурации, в котором будет указано, какие объекты следует использовать в каждом случае. Для создания экземпляров требуемых объектов в программе организуется переключатель, управляемый информацией из файла конфигурации.
- Для каждого семейства объектов создается отдельная запись в базе данных, содержащая сведения об используемых объектах. При этом каждый элемент (поле) записи базы данных указывает, какой конкретный класс должен использоваться при вызове соответствующего метода абстрактного класса-фабрики.

При работе с языком Java концепцию использования файла конфигурации можно дополнительно расширить. Имена полей записи файла могут представлять собой имя требуемого класса. В этом случае не требуется даже иметь полное имя класса, достаточно только добавить суффикс или префикс к имени, хранящемуся в файле конфигурации, а затем, используя класс `Class` языка Java, корректно создать экземпляр объекта с данным именем.²

В реально существующих проектах члены различных семейств не всегда являются производными от общего родительского класса. Например, в предыдущем примере с

² Подробное описание класса `Class` языка Java можно найти в *Eckel B. Thinking in Java, Upper Saddle River, N.J.: Prentice Hall, 2000.*

драйверами, возможно, что классы драйверов **LRDD** и **HRDD** будут производными от разных родительских классов. В таких случаях следует так адаптировать эти классы, чтобы шаблон **Abstract Factory** смог работать с ними.

Применение шаблона **Abstract Factory** к проблеме **САПР**

В задаче с различными САПР система должна работать с различными наборами элементов, определяемыми в зависимости от используемой версии САПР. При работе с САПР версии V1 все элементы должны быть реализованы для версии V1, а при работе с САПР версии V2 все элементы должны быть реализованы для версии V2.

Исходя из этого требования, несложно выделить два семейства классов, для управления которыми следует применить шаблон **Abstract Factory**. Первое семейство — это классы элементов версии V1, а второе — классы элементов версии V2.

Резюме

Шаблон **Abstract Factory** применяется, когда необходимо координировать создание семейств объектов. Он позволяет отделить реализацию правил создания экземпляров новых объектов от объекта-клиента, который будет эти объекты использовать.

- Прежде всего, идентифицируйте правила создания экземпляров объектов и определите абстрактный класс с интерфейсом, включающим отдельный метод для каждого из классов, экземпляры которых должны быть созданы.
- Затем для каждого семейства создайте конкретные классы, производные от данного абстрактного класса.
- Реализуйте в объекте-клиенте обращение к абстрактному объекту-фабрике с целью создания требуемых серверных объектов.

Приложение. Примеры программного кода на языке C++

Листинг 10.4. Фрагмент кода. Переключатель управления выбором драйвера

```
// Фрагмент кода на языке C++

// Класс ApControl
.
.
.
void ApControl::doDraw () {
    .
    .
    .
    switch (RESOLUTION) {
        case LOW:
            // Использование LRDD
        case HIGH:
            // Использование HRDD
    }
}
void ApControl::doPrint () {
    .
    .
    .
    switch (RESOLUTION) {
        case LOW:
            // Использование LRDD
    }
```

```
        case HIGH:
            // Использование HRPD
    }
}
```

Листинг 10.5. Фрагмент кода. Использование полиморфизма для решения проблемы

```
// Фрагмент кода на языке C++
// Класс ApControl
void ApControl::doDraw () {
    myDisplayDriver->draw();
}
void ApControl::doPrint () {
    myPrintDriver->print();
}
```

Листинг 10.6. Фрагмент кода. Реализация класса ResFactory

```
// Фрагмент кода на языке C++
class LowResFact : public ResFactory;

DisplayDriver *
LowResFact::getDispDrvr() {
    return new lrdd;
}

PrintDriver *
LowResFact::getPrtDrvr() {
    return new lrpdr;
}

class HighResFact : public ResFactory;

DisplayDriver *
HighResFact::getDispDrvr() {
    return new hrdd;
}

PrintDriver *
HighResFact::getPrtDrvr() {
    return new hrpdr;
}
```
