Глава 1. Введение в паттерны проектирования

Проектирование объектно-ориентированных программ - нелегкое дело, а если их нужно использовать повторно, то все становится еще сложнее. Необходимо подобрать подходящие объекты, отнести их к различным классам, соблюдая разумную степень детализации, определить интерфейсы классов и иерархию наследования и установить существенные отношения между классами. Дизайн должен, с одной стороны, соответствовать решаемой задаче, с другой - быть общим, чтобы удалось учесть все требования, которые могут возникнуть в будущем. Хотелось бы также избежать вовсе или, по крайней мере, свести к минимуму необходимость перепроектирования. Поднаторевшие в объектно-ориентированном проектировании разработчики скажут вам, что обеспечить «правильный», то есть в достаточной мере гибкий и пригодный для повторного использования дизайн, с первого раза очень трудно, если вообще возможно. Прежде чем считать цель достигнутой, они обычно пытаются опробовать найденное решение на нескольких задачах, и каждый раз модифицируют его. И все же опытным проектировщикам удается создать хороший дизайн системы. В то же время новички испытывают шок от количества возможных вариантов и нередко возвращаются к привычным не объектно-ориентированным методикам. Проходит немало времени перед тем, как становится понятно, что же такое удачный объектно-ориентированный дизайн. Опытные проектировщики, очевидно, знают какие-то тонкости, ускользающие от новичков. Так что же это? Прежде всего, опытному разработчику понятно, что не нужно решать каждую новую задачу с нуля. Вместо этого он старается повторно воспользоваться теми решениями, которые оказались удачными в прошлом. Отыскав хорошее решение один раз, он будет прибегать к нему снова и снова. Именно благодаря накопленному опыту проектировщик и становится эксертом в своей области. Во многих объектно-ориентированных системах вы встретите повторяющиеся паттерны, состоящие из классов и взаимодействующих объектов. С их помощью решаются конкретные задачи проектирования, в результате чего объектно-ориентированный дизайн становится более гибким, элегантным, и им можно воспользоваться повторно. Проектировщик, знакомый с паттернами, может сразу же применять их к решению новой задачи, не пытаясь каждый раз изобретать велосипед. Поясним нашу мысль через аналогию. Писатели редко выдумывают совершенно новые сюжеты. Вместо этого они берут за основу уже отработанные в мировой литературе схемы, жанры и образы. Например, трагический герой - Макбет, Введение в паттерны проектирования Гамлет и т.д., мотив убийства - деньги, месть, ревность и т.п. Точно так же в объектно-ориентированном проектировании используются такие паттерны, как «представление состояния с помощью объектов» или «декорирование объектов, чтобы было проще добавлять и удалять их свойства». Все мы знаем о ценности опыта. Сколько раз при проектировании вы испытывали дежавю, чувствуя, что уже когда-то решали такую же задачу, только никак не сообразить, когда и где? Если бы удалось вспомнить детали старой задачи и ее решения, то не пришлось бы придумывать все заново. Увы, у нас нет привычки записывать свой опыт на благо другим людям да и себе тоже. Цель этой книги состоит как раз в том,

чтобы документировать опыт разработки объектно-ориентированных программ в виде паттернов проектирования. Каждому паттерну мы присвоим имя, объясним его назначение и роль в проектировании объектно-ориентированных систем. Некоторые из наиболее распространенных паттернов формализованы и сведены в единый каталог. Паттерны проектирования упрощают повторное использование удачных проектных и архитектурных решений. Представление прошедших проверку временем методик в виде паттернов проектирования облегчает доступ к ним со стороны разработчиков новых систем. С помощью паттернов можно улучшить качество документации и сопровождения существующих систем, позволяя явно описать взаимодействия классов и объектов, а также причины, по которым система была построена так, а не иначе. Проще говоря, паттерны проектирования дают разработчику возможность быстрее найти «правильный» путь. Как уже было сказано, в книгу включены только такие паттерны, которые неоднократно применялись в разных системах. По большей части они никогда ранее не документировались и либо известны самым квалифицированным специалистам по объектно-ориентированному проектированию, либо были частью какойто удачной системы. Хотя книга получилась довольно объемной, паттерны проектирования - лишь малая часть того, что необходимо знать специалисту в этой области. В издание не включено описание паттернов, имеющих отношение к параллельности, распределенному программированию и программированию систем реального времени. Отсутствуют и сведения о паттернах, специфичных для конкретных предметных областей. Из этой книги вы не узнаете, как строить интерфейсы пользователя, как писать драйверы устройств и как работать с объектно-ориентированными базами данных. В каждой из этих областей есть свои собственные паттерны, и, может быть, кто-то их и систематизирует. 1.1. Что такое паттерн проектирования По словам Кристофера Александра, «любой паттерн описывает задачу, которая снова и снова возникает в нашей работе, а также принцип ее решения, причем таким образом, что это решение можно потом использовать миллион раз, ничего не изобретая заново» [AIS+77]. Хотя Александр имел в виду паттерны, возникающие при проектировании зданий и городов, но его слова верны и в отношении паттернов объектно-ориентированного проектирования. Наши решения выражаются Что такое паттерн проектирования в терминах объектов и интерфейсов, а не стен и дверей, но в обоих случаях смысл паттерна - предложить решение определенной задачи в конкретном контексте. В общем случае паттерн состоит из четырех основных элементов: 1. Имя. Сославшись на него, мы можем сразу описать проблему проектирования; ее решения и их последствия. Присваивание паттернам имен позволяет проектировать на более высоком уровне абстракции. С помощью словаря паттернов можно вести обсуждение с коллегами, упоминать паттерны в документации, в тонкостях представлять дизайн системы. Нахождение хороших имен было одной из самых трудных задач при составлении каталога. 2. Задача. Описание того, когда следует применять паттерн. Необходимо сформулировать задачу и ее контекст. Может описываться конкретная проблема проектирования, например способ представления

алгоритмов в виде объектов. Иногда отмечается, какие структуры классов или объектов свидетельствуют о негибком дизайне. Также может включаться перечень условий, при выполнении которых имеет смысл применять данный паттерн. 3. Решение. Описание элементов дизайна, отношений между ними, функций каждого элемента. Конкретный дизайн или реализация не имеются в виду, поскольку паттерн - это шаблон, применимый в самых разных ситуациях. Просто дается абстрактное описание задачи проектирования и того, как она может быть решена с помощью некоего весьма обобщенного сочетания элементов (в нашем случае классов и объектов). 4. Результаты - это следствия применения паттерна и разного рода компромиссы. Хотя при описании проектных решений о последствиях часто не упоминают, знать о них необходимо, чтобы можно было выбрать между различными вариантами и оценить преимущества и недостатки данного паттерна. Здесь речь идет и о выборе языка и реализации. Поскольку в объектно-ориентированном проектировании повторное использование зачастую является важным фактором, то к результатам следует относить и влияние на степень гибкости, расширяемости и переносимости системы. Перечисление всех последствий поможет вам понять и оценить их роль. То, что один воспринимает как паттерн, для другого просто строительный блок. В этой книге мы рассматриваем паттерны на определенном уровне абстракции. Паттерны проектирования - это не то же самое, что связанные списки или хэш-таблицы, которые можно реализовать в виде класса и повторно использовать без каких бы то ни было модификаций. Но это и не сложные, предметно-ориентированные решения для целого приложения или подсистемы. Здесь под паттернами проектирования понимается описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте. Паттерн проектирования именует, абстрагирует и идентифицирует ключевые аспекты структуры общего решения, которые и позволяют применить его для создания повторно используемого дизайна. Он вычленяет участвующие классы и экземпляры, их роль и отношения, а также функции. При описании каждого паттерна внимание акцентируется на конкретной задаче объектно-ориентированного проектирования. Анализируется,- когда следует применять паттерн, можно ли Введение в паттерны проектирования его использовать с учетом других проектных ограничений, каковы будут последствия применения метода. Поскольку любой проект в конечном итоге предстоит реализовывать, в состав паттерна включается пример кода на языке C++ (иногда на Smalltalk), иллюстрирующего реализацию. Хотя, строго говоря, паттерны используются в проектировании, они основаны на практических решениях, реализованных на основных языках объектно-ориентированного программирования типа Smalltalk и C++, а не на процедурных (Pascal, C, Ada и т.п.) или объектно-ориентированных языках с динамической типизацией (CLOS, Dylan, Self). Мы выбрали Smalltalk и C++ из прагматических соображений, поскольку чаще всего работаем с ними и поскольку они завоевывают все большую популярность. Выбор языка программирования безусловно важен. В наших паттернах подразумевается использование возможностей Smalltalk и C++, и от

этого зависит, что реализовать легко, а что - трудно. Если бы мы ориентировались на процедурные языки, то включили бы паттерны наследование, инкапсуляция и полиморфизм. Некоторые из наших паттернов напрямую поддерживаются менее распространенными языками. Так, в языке CLOS есть мультиметоды, которые делают ненужным паттерн посетитель. Собственно, даже между Smalltalk и C++ есть много различий, из-за чего некоторые паттерны проще выражаются на одном языке, чем на другом (см., например, паттерн итератор). 1.2. Паттерны проектирования в схеме MVC в языке Smalltalk В Smalltalk-80 для построения интерфейсов пользователя применяется тройка классов модель/вид/контроллер (Model/View/Controller - MVC) [KP88]. Знакомство с паттернами проектирования, встречающимися в схеме MVC, поможет вам разобраться в том, что мы понимаем под словом «паттерн». MVC состоит из объектов трех видов. Модель - это объект приложения, а вид - экранное представление. Контроллер описывает, как интерфейс реагирует на управляющие воздействия пользователя. До появления схемы MVC эти объекты в пользовательских интерфейсах смешивались. MVC отделяет их друг от друга, за счет чего повышается гибкость и улучшаются возможности повторного использования. M VC отделяет вид от модели, устанавливая между ними протокол взаимодействия «подписка/оповещение». Вид должен гарантировать, что внешнее представление отражает состояние модели. При каждом изменении внутренних данных модель оповещает все зависящие от нее виды, в результате чего вид обновляет себя. Такой подход позволяет присоединить к одной модели несколько видов, обеспечив тем самым различные представления. Можно создать новый вид, не переписывая модель. На рисунке ниже показана одна модель и три вида. (Для простоты мы опустили контроллеры.) Модель содержит некоторые данные, которые могут быть представлены в виде электронной таблицы, гистограммы и круговой диаграммы. Паттерны проектирования в схеме MVC Виды Модель Модель оповещает свои виды при каждом изменении значений данных, а виды обращаются к модели для получения новых значений. На первый взгляд, в этом примере продемонстрирован просто дизайн, отделяющий вид от модели. Но тот же принцип применим и к более общей задаче: разделение объектов таким образом, что изменение одного отражается сразу на нескольких других, причем изменившийся объект не имеет информации о деталях реализации объектов, на которые он оказал воздействие. Этот более общий подход описывается паттерном проектирования наблюдатель. Еще одно свойство MVC заключается в том, что виды могут быть вложенными. Например, панель управления, состоящую из кнопок, допустимо представить как составной вид, содержащий вложенные, - по одной кнопке на каждый. Пользовательский интерфейс инспектора объектов может состоять из вложенных видов, используемых также и в отладчике. MVC поддерживает вложенные виды с помощью класса CompositeView, являющегося подклассом View. Объекты класса CompositeView ведут себя так же, как объекты класса View, поэтому могут использоваться всюду, где и виды. Но еще они могут содержать вложенные виды и управлять ими. Здесь можно было бы считать, что этот дизайн позволяет обращаться с

составным видом, как с любым из его компонентов. Но тот же дизайн применим и в ситуации, когда мы хотим иметь возможность группировать объекты и рассматривать группу как отдельный объект. Такой подход описывается паттерном компоновщик. Он позволяет создавать иерархию классов, в которой некоторые подклассы определяют примитивные объекты (например, Button - кнопка), а другие - составные объекты (CompositeView), группирующие примитивы в более сложные структуры. MVC позволяет также изменять реакцию вида на действия пользователя. При этом визуальное представление остается прежним. Например, можно изменить реакцию на нажатие клавиши или использовать всплывающие меню вместо командных клавиш. MVC инкапсулирует механизм определения реакции в объекте Controller. Существует иерархия классов контроллеров, и это позволяет без труда создать новый контроллер как.вариант уже существующего. Введение в паттерны проектирования Вид пользуется экземпляром класса, производного от Controller, для реализации конкретной стратегии реагирования. Чтобы реализовать иную стратегию, нужно просто подставить другой контроллер. Можно даже заменить контроллер вида во время выполнения программы, изменив тем самым реакцию на действия пользователя. Например, вид можно деактивировать, так что он вообще не будет ни на что реагировать, если передать ему контроллер, игнорирующий события ввода. Отношение вид-контроллер - это пример паттерна проектирования стратегия. Стратегия - это объект для представления алгоритма. Он полезен, когда вы хотите статически или динамически подменить один алгоритм другим, если существует много вариантов одного алгоритма или когда с алгоритмом связаны сложные структуры данных, которые хотелось бы инкапсулировать. В МУС используются и другие паттерны проектирования, например фабричный метод, позволяющий задать для вида класс контроллера по умолчанию, и декоратор для добавления к виду возможности прокрутки. Но основные отношения в схеме МУС описываются паттернами наблюдатель, компоновщик и стратегия. 1.3. Описание паттернов проектирования Как мы будем описывать паттерны проектирования? Графических обозначений недостаточно. Они просто символизируют конечный продукт процесса проектирования в виде отношений между классами и объектами. Чтобы повторно воспользоваться дизайном, нам необходимо документировать решения, альтернативные варианты и компромиссы, которые привели к нему. Важны также конкретные примеры, поскольку они позволяют увидеть применение паттерна. При описании паттернов проектировании мы будем придерживаться единого принципа. Описание каждого паттерна разбито на разделы, перечисленные ниже. Такой подход позволяет единообразно представить информацию, облегчает изучение, сравнение и применение паттернов. Название и классификация паттерна Название паттерна должно четко отражать его назначение. Классификация паттернов проводится в соответствии со схемой, которая изложена в разделе 1.5. Назначение Лаконичный ответ на следующие вопросы: каковы функции паттерна, его обоснование и назначение, какую конкретную задачу проектирования можно решить с его помощью. Известен также под именем

Другие распространенные названия паттерна, если таковые имеются. Мотивация Сценарий, иллюстрирующий задачу проектирования и то, как она решается данной структурой класса или объекта. Благодаря мотивации можно лучше понять последующее, более абстрактное описание паттерна. Описание паттернов проектирования Применимость Описание ситуаций, в которых можно применять данный паттерн. Примеры проектирования, которые можно улучшить с его помощью. Распознавание таких ситуаций. Структура Графическое представление классов в паттерне с использованием нотации, основанной на методике Object Modeling Technique (OMT) [RBP+91]. Мы пользуемся также диаграммами взаимодействий [JCJO92, Boo94] для иллюстрации последовательностей запросов и отношений между объектами. В приложении В эта нотация описывается подробно. Участники Классы или объекты, задействованные в данном паттерне проектирования, и их функции. Отношения Взаимодействие участников для выполнения своих функций. Результаты Насколько паттерн удовлетворяет поставленным требованиям? Результаты применения, компромиссы, на которые приходится идти. Какие аспекты поведения системы можно независимо изменять, используя данный паттерн? Реализация Сложности и так называемые подводные камни при реализации паттерна. Советы и рекомендуемые приемы. Есть ли у данного паттерна зависимость от языка программирования? Пример кода Фрагмент кода, иллюстрирующий вероятную реализацию на языках C++ или Smalltalk. Известные применения Возможности применения паттерна в реальных системах. Даются, по меньшей мере, два примера из различных областей. Родственные паттерны Связь других паттернов проектирования с данным. Важные различия. Использование данного паттерна в сочетании с другими. В приложениях содержится информация, которая поможет вам лучше понять паттерны и связанные с ними вопросы. Приложение А представляет собой глоссарий употребляемых нами терминов. В уже упомянутом приложении В дано описание разнообразных нотаций. Некоторые аспекты применяемой нотации мы поясняем по мере ее появления в тексте книги. Наконец, в приложении С приведен исходный код базовых классов, встречающихся в примерах. Введение в паттерны проектирования 1.4. Каталог паттернов проектирования Каталог содержит 23 паттерна. Ниже для удобства перечислены их имена и назначение. В скобках после названия каждого паттерна указан номер страницы, откуда начинается его подробное описание. Abstract Factory (абстрактная фабрика) (93) Предоставляет интерфейс для создания семейств, связанных между собой, или независимых объектов, конкретные классы которых неизвестны. Adapter (адаптер) (141) Преобразует интерфейс класса в некоторый другой интерфейс, ожидаемый клиентами. Обеспечивает совместную работу классов, которая была бы невозможна без данного паттерна из-за несовместимости интерфейсов. Bridge (мост) (152) Отделяет абстракцию от реализации, благодаря чему появляется возможность независимо изменять то и другое. Builder (строитель) (103) Отделяет конструирование сложного объекта от его представления, позволяя использовать один и тот же процесс конструирования для создания различных представлений. Chain of

Responsibility (цепочка обязанностей) (217) Можно избежать жесткой зависимости отправителя запроса от его получателя, при этом запросом начинает обрабатываться один из нескольких объектов. Объекты-получатели связываются в цепочку, и запрос передается по цепочке, пока какой-то объект его не обработает. Command (команда) (226) Инкапсулирует запрос в виде объекта, позволяя тем самым параметризовывать клиентов типом запроса, устанавливать очередность запросов, протоколировать их и поддерживать отмену выполнения операций. Composite (компоновщик) (162) Группирует объекты в древовидные структуры для представления иерархий типа «часть-целое». Позволяет клиентам работать с единичными объектами так же, как с группами объектов. Decorator (декоратор) (173) Динамически возлагает на объект новые функции. Декораторы применяются для расширения имеющейся функциональности и являются гибкой альтернативой порождению подклассов. Facade (фасад) (183) Предоставляет унифицированный интерфейс к множеству интерфейсов в некоторой подсистеме. Определяет интерфейс более высокого уровня, облегчающий работу с подсистемой. Каталог паттернов проектирования Factory Method (фабричный метод) (111) Определяет интерфейс для создания объектов, при этом выбранный класс инстанцируется подклассами. Flyweight (приспособленец) (191) Использует разделение для эффективной поддержки большого числа мелких объектов. Interpreter (интерпретатор) (236) Для заданного языка определяет представление его грамматики, а также интерпретатор предложений языка, использующий это представление. Iterator (итератор) (173) Дает возможность последовательно обойти все элементы составного объекта, не раскрывая его внутреннего представления. Mediator (посредник) (263) Определяет объект, в котором инкапсулировано знание о том, как взаимодействуют объекты из некоторого множества. Способствует уменьшению числа связей между объектами, позволяя им работать без явных ссылок друг на друга. Это, в свою очередь, дает возможность независимо изменять схему взаимодействия. Memento (хранитель) (272) Позволяет, не нарушая инкапсуляции, получить и сохранить во внешней памяти внутреннее состояние объекта, чтобы позже объект можно было восстановить точно в таком же состоянии. Observer (наблюдатель) (281) Определяет между объектами зависимость типа один-ко-многим, так что при изменении состоянии одного объекта все зависящие от него получают извещение и автоматически обновляются. Prototype (прототип) (121) Описывает виды создаваемых объектов с помощью прототипа и создает новые объекты путем его копирования. Proxy (заместитель) (203) Подменяет другой объект для контроля доступа к нему. Singleton (одиночка) (130) Гарантирует, что некоторый класс может иметь только один экземпляр, и предоставляет глобальную точку доступа к нему. State (состояние)(291) Позволяет объекту варьировать свое поведение при изменении внутреннего состояния. При этом создается впечатление, что поменялся класс объекта. Strategy (стратегия) (300) Определяет семейство алгоритмов, инкапсулируя их все и позволяя подставлять один вместо другого. Можно менять алгоритм независимо от клиента, который им пользуется. введение в паттерны проектирования Template Method (шаблонный метод) (309)

Определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры. Visitor (посетитель) (314) Представляет операцию, которую надо выполнить над элементами объекта. Позволяет определить новую операцию, не меняя классы элементов, к которым он применяется. 1.5. Организация каталога Паттерны проектирования различаются степенью детализации и уровнем абстракции и должны быть каким-то образом организованы. В данном разделе описывается классификация, позволяющая ссылаться на семейства взаимосвязанных паттернов. Она поможет быстрее освоить паттерны, описываемые в каталоге, и укажет направление поиска новых. Мы будем классифицировать паттерны по двум критериям (табл. 1.1). Первый - цель - отражает назначение паттерна. В связи с этим выделяются порождающие паттерны, структурные паттерны и паттерны поведения. Первые связаны с процессом создания объектов. Вторые имеют отношение к композиции объектов и классов. Паттерны поведения характеризуют то, как классы или объекты взаимодействуют между собой. Таблица 1.1. Пространство паттернов проектирования Порождающие паттерны Структурные паттерны Паттерны поведения Класс Фабричный метод Адаптер (класса) Интерпретатор Шаблонный метод Объект Абстрактная фабрика Одиночка Прототип Строитель Адаптер (объекта) Декоратор Заместитель Компоновщик Мост Приспособленец Фасад Итератор Команда Наблюдатель Посетитель Посредник Состояние Стратегия Хранитель Цепочка обязанностей Второй критерий - уровень - говорит о том, к чему обычно применяется паттерн: к объектам или классам. Паттерны уровня классов описывают отношения между классами и их подклассами. Такие отношения выражаются с помощью наследования, поэтому они статичны, то есть зафиксированы на этапе компиляции. Паттерны уровня объектов описывают отношения между объектами, которые могут изменяться во время выполнения и потому более динамичны. Почти все паттерны в какой-то мере используют наследование. Поэтому к категории «паттерны классов» отнесены только те, что сфокусированы лишь на отношениях между классами. Обратите внимание: большинство паттернов действуют на уровне объектов. Порождающие паттерны классов частично делегируют ответственность за создание объектов своим подклассам, тогда как порождающие паттерны объектов передают ответственность другому объекту. Структурные паттерны классов используют наследование для составления классов, в то время как структурные паттерны объектов описывают способы сборки объектов из частей. Поведенческие паттерны классов используют наследование для описания алгоритмов и потока управления, а поведенческие паттерны объектов описывают, как объекты, принадлежащие некоторой группе, совместно функционируют и выполняют задачу, которая ни одному отдельному объекту не под силу. Существуют и другие способы классификации паттернов. Некоторые паттерны часто используются вместе. Например, компоновщик применяется с итератором или посетителем. Некоторыми паттернами предлагаются альтернативные решения. Так, прототип нередко можно использовать вместо абстрактной фабрики. Применение части паттернов приводит к схожему

дизайну, хотя изначально их назначение различно. Например, структурные диаграммы компоновщика и декоратора похожи. Классифицировать паттерны можно и по их ссылкам (см. разделы «Родственные паттерны»). На рис. 1.1 такие отношения изображены графически. Ясно, что организовать паттерны проектирования допустимо многими способами. Оценивая паттерны с разных точек зрения, вы глубже поймете, как они функционируют, как их сравнивать и когда применять тот или другой. 1.6. Как решать задачи проектирования с помощью паттернов Паттерны проектирования позволяют разными способами решать многие задачи, с которыми постоянно сталкиваются проектировщики объектно-ориентированных приложений. Поясним эту мысль примерами. Поиск подходящих объектов Объектно-ориентированные программы состоят из объектов. Объект сочетает данные и процедуры для их обработки. Такие процедуры обычно называют методами или операциями. Объект выполняет операцию, когда получает запрос (или сообщение) от клиента. Посылка запроса - это единственный способ заставить объект выполнить операцию. А выполнение операции - единственный способ изменить внутреннее состояние объекта. Имея в виду два эти ограничения, говорят, что внутреннее состояние объекта инкапсулировано: к нему нельзя получить непосредственный доступ, то есть представление объекта закрыто от внешней программы. Введение в паттерны проектирования Самая трудная задача в объектно-ориентированном проектировании - разложить систему на объекты. При решении приходится учитывать множество факторов: инкапсуляцию, глубину детализации, наличие зависимостей, гибкость, производительность, развитие, повторное использование и т.д. и т.п. Все это влияет на декомпозицию, причем часто противоречивым образом. Методики объектно-ориентированного проектирования отражают разные подходы. Вы можете сформулировать задачу письменно, выделить из получившейся Как решать задачи проектирования фразы существительные и глаголы, после чего создать соответствующие классы и операции. Другой путь - сосредоточиться на отношениях и разделении обязанностей в системе. Можно построить модель реального мира или перенести выявленные при анализе объекты на свой дизайн. Согласие по поводу того, какой подход самый лучший, никогда не будет достигнуто. Многие объекты возникают в проекте из построенной в ходе анализа модели. Но нередко появляются и классы, у которых нет прототипов в реальном мире. Это могут быть классы как низкого уровня, например массивы, так и высокого. Паттерн компоновщик вводит такую абстракцию для единообразной трактовки объектов, у которой нет физического аналога. Если придерживаться строгого моделирования и ориентироваться только на реальный мир, то получится система, отражающая сегодняшние потребности, но, возможно, не учитывающая будущего развития. Абстракции, возникающие в ходе проектирования, - ключ к гибкому дизайну. Паттерны проектирования помогают выявить не вполне очевидные абстракции и объекты, которые могут их использовать. Например, объектов, представляющих процесс или алгоритм, в действительности нет, но они являются неотъемлемыми составляющими гибкого дизайна. Паттерн стратегия описывает способ реализации

взаимозаменяемых семейств алгоритмов. Паттерн состояние позволяет представить состояние некоторой сущности в виде объекта. Эти объекты редко появляются во время анализа и даже на ранних стадиях проектирования. Работа с ними начинается позже, при попытках сделать дизайн более гибким и пригодным для повторного использования. Определение степени детализации объекта Размеры и число объектов могут сильно варьироваться. С их помощью может быть представлено все, начиная с уровня аппаратуры и до законченных приложений. Как же решить, что должен представлять собой объект? Здесь и потребуются паттерны проектирования. Паттерн фасад показывает, как представить в виде объекта целые подсистемы, а паттерн приспособленец - как поддержать большое число объектов при высокой степени детализации. Другие паттерны указывают путь к разложению объекта на меньшие подобъекты. Абстрактная срабрика и строитель описывают объекты, единственной целью которых является создание других объектов, а посетитель и команда - объекты, отвечающие за реализацию запроса к другому объекту или группе. Специфицирование интерфейсов объекта При объявлении объектом любой операции должны быть заданы: имя операции, объекты, передаваемые в качестве параметров, и значение, возвращаемое операцией. Эту триаду называют сигнатурой операции. Множество сигнатур всех определенных для объекта операций называется интерфейсом этого объекта. Интерфейс описывает все множество запросов, которые можно отправить объекту. Любой запрос, сигнатура которого соответствует интерфейсу объекта, может быть ему послан. Тип — это имя, используемое для обозначения конкретного интерфейса. Говорят, что объект имеет тип Window, если он готов принимать запросы на выполнение любых операций, определенных в интерфейсе с именем Window. У одного объекта может быть много типов. Напротив, сильно отличающиеся объекты могут разделять общий тип. Часть интерфейса объекта может быть охарактеризована одним типом, а часть - другим. Два объекта одного и того же типа должны разделять только часть своих интерфейсов. Интерфейсы могут содержать другие интерфейсы в качестве подмножеств. Мы говорим, что один тип является подтипом другого, если интерфейс первого содержит интерфейс второго. В этом случае второй тип называется супертипом для первого. Часто говорят также, что подтип наследует интерфейс своего супертцпа. В объектно-ориентированных системах интерфейсы фундаментальны. Об объектах известно только то, что они сообщают о себе через свои интерфейсы. Никакого способа получить информацию об объекте или заставить его что-то сделать в обход интерфейса не существует. Интерфейс объекта ничего не говорит о его реализации; разные объекты вправе реализовывать сходные запросы совершенно по-разному. Это означает, что два объекта с различными реализациями могут иметь одинаковые интерфейсы. Когда объекту посылается запрос, то операция, которую он будет выполнять, зависит как от запроса, так и от объекта-адресата. Разные объекты, поддерживающие одинаковые интерфейсы, могут выполнять в ответ на такие запросы разные операции. Ассоциация запроса с объектом и одной из его операций во время выполнения называется динамическим связыванием.

Динамическое связывание означает, что отправка некоторого запроса не определяет никакой конкретной реализации до момента выполнения. Следовательно, допустимо написать программу, которая ожидает объект с конкретным интерфейсом, точно зная, что любой объект с подходящим интерфейсом сможет принять этот запрос. Более того, динамическое связывание позволяет во время выполнения подставить вместо одного объекта другой, если он имеет точно такой же интерфейс. Такая взаимозаменяемость называется полиморфизмом и является важнейшей особенностью объектно-ориентированных систем. Она позволяет клиенту не делать почти никаких предположений об объектах, кроме того, что они поддерживают определенный интерфейс. Полиморфизм упрощает определение клиентов, позволяет отделить объекты друг от друга и дает объектам возможность изменять взаимоотношения во время выполнения. Паттерны проектирования позволяют определять интерфейсы, задавая их основные элементы и то, какие данные можно передавать через интерфейс. Паттерн может также «сказать», что не должно проходить через интерфейс. Хорошим примером в этом отношении является хранитель. Он описывает, как инкапсулировать и сохранить внутреннее состояние объекта таким образом, чтобы в будущем его можно было восстановить точно в таком же состоянии. Объекты, удовлетворяющие требованиям паттерна хранитель, должны определить два интерфейса: один ограниченный, который позволяет клиентам держать у себя и копировать хранители, а другой привилегированный, которым может пользоваться только сам объект для сохранения и извлечения информации о состоянии их хранителя. Как решать задачи проектирования Паттерны проектирования специфицируют также отношения между интерфейсами. В частности, нередко они содержат требование, что некоторые классы должны иметь схожие интерфейсы, а иногда налагают ограничения на интерфейсы классов. Так, декоратор и заместитель требуют, чтобы интерфейсы объектов этих паттернов были идентичны интерфейсам декорируемых и замещаемых объектов соответственно. Интерфейс объекта, принадлежащего паттерну посетитель, должен отражать все классы объектов, с которыми он будет работать. Специфицирование реализации объектов До сих пор мы почти ничего не сказали о том, как же в действительности определяется объект. Реализация объекта определяется его классом. Класс специфицирует внутренние данные объекта и его представление, а также операции, которые объект может выполнять. В нашей нотации, основанной на OMT (см. приложение B), класс изображается в виде прямоугольника, внутри которого жирным шрифтом написано имя класса. Ниже обычным шрифтом перечислены операции. Любые данные, которые определены для класса, следуют после операций. Имя класса, операции и данные разделяются горизонтальными линиями. Типы возвращаемого значения и переменных экземпляра необязательны, поскольку мы не ограничиваем себя языками программирования с сильной типизацией. Объекты создаются с помощью инстанцирования класса. Говорят, что объект является экземпляром класса. В процессе инстанцирования выделяется память для переменных экземпляра (внутренних данных объекта), и с этими данными

ассоциируются операции. С помощью инстанцирования одного класса можно создать много разных объектов-экземпляров. Пунктирная линия со стрелкой обозначает класс, который инстанцирует объекты другого класса. Стрелка направлена в сторону класса инстанцированного объекта. Новые классы можно определить в терминах существующих с помощью наследования классов. Если подкласс наследует родительскому классу, то он включает определения всех данных и операций, определенных в родительском классе. Объекты, являющиеся экземплярами подкласса, будут содержать все данные, определенные как в самом подклассе, так и во всех его родительских классах. Такой объект сможет выполнять все операции, определенные в подклассе и его предках. Отношение «является подклассом» обозначается вертикальной линией с треугольником. Класс называется абстрактным, если его единственное назначение - определить общий интерфейс для всех своих подклассов. Абстрактный класс делегирует Введение в паттерны проектирования реализацию всех или части своих операции подклассам, поэтому у него не может быть экземпляров. Операции, объявленные, но не реализованные в абстрактном классе, называются абстрактными. Класс, не являющийся абстрактным, называется конкретным. Подклассы могут уточнять или переопределять поведение своих предков. Точнее, класс может заместить операцию, определенную в родительском классе. Замещение дает подклассам возможность обрабатывать запросы, адресованные родительским классам. Наследование позволяет определять новые классы, просто расширяя возможности старых. Тем самым можно без труда определять семейства объектов со схожей функциональностью. Имена абстрактных классов оформлены курсивом, чтобы отличать их от конкретных. Курсив используется также для обозначения абстрактных операций. На диаграмме может изображаться псевдокод, описывающий реализацию операции; в таком случае код представлен в прямоугольнике с загнутым уголком, соединенном пунктирной линией с операцией, которую он реализует. Подмешанным (mixin class) называется класс, назначение которого - предоставить дополнительный интерфейс или функциональность другим классам. Он родственен абстрактным классам в том смысле, что не предполагает непосредственного инстанцирования. Для работы с подмешанными классами необходимо множественное наследование. Наследование класса и наследование интерфейса Важно понимать различие между классом объекта и его типом. Класс объекта определяет, как объект реализован, то есть внутреннее состояние и реализацию операций объекта. Напротив, тип относится только к интерфейсу Как решать задачи проектирования объекта - множеству запросов, на которые объект отвечает. У объекта может быть много типов, и объекты разных классов могут иметь один и тот же тип. Разумеется, между классом и типом есть тесная связь. Поскольку класс определяет, какие операции может выполнять объект, то заодно он определяет и его тип. Когда мы говорим «объект является экземпляром класса», то подразумеваем, что он поддерживает интерфейс, определяемый этим классом. В языках вроде C++ и Eiffel классы используются для специфицирования, типа и реализации объекта. В программах на языке

Smalltalk типы переменных не объявляются, поэтому компилятор не проверяет, что тип объекта, присваиваемого переменной, является подтипом типа переменной. При отправке сообщения необходимо проверять, что класс получателя реализует реакцию на сообщение, но проверка того, что получатель является экземпляром определенного класса, не нужна. Важно также понимать различие между наследованием класса и наследованием интерфейса (или порождением подтипов). В случае наследования класса реализация объекта определяется в терминах реализации другого объекта. Проще говоря, это механизм разделения кода и представления. Напротив, наследование интерфейса (порождение подтипов) описывает, когда один объект можно использовать вместо другого. Две эти концепции легко спутать, поскольку во многих языках явное различие отсутствует. В таких языках, как C++ и Eiffel, под наследованием понимается одновременно наследование интерфейса и реализации. Стандартный способ реализации наследования интерфейса в C++ - это открытое наследование классу, в котором есть исключительно виртуальные функции. Истинное наследование интерфейса можно аппроксимировать в C++ с помощью открытого наследования абстрактному классу. Истинное наследование реализации или класса аппроксимируется с помощью закрытого наследования. В Smalltalk под наследованием понимается только наследование реализации. Переменной можно присвоить экземпляры любого класса при условии, что они поддерживают операции, выполняемые над значением этой переменной. Хотя в большинстве языков программирования различие между наследованием интерфейса и реализации не поддерживается, на практике оно существует. Программисты на Smalltalk обычно предпочитают считать, что подклассы - это подтипы (хотя имеются и хорошо известные исключения [Coo92]). Программисты на C++ манипулируют объектами через типы, определяемые абстрактными классами. Многие паттерны проектирования зависят от этого различия. Например, объекты, построенные в соответствии с паттерном цепочка обязанностей, должны иметь общий тип, но их реализация обычно различна. В паттерне компоновщик отдельный объект (компонент) определяет общий интерфейс, но реализацию часто определяет составной объект (композиция). Паттерны команда, наблюдатель, состояние и стратегия часто реализуются абстрактными классами с исключительно виртуальными функциями. Программирование в соответствии с интерфейсом, а не с реализацией Наследование классов - это не что иное, как механизм расширения функциональности приложения путем повторного использования функциональности родительских классов. Оно позволяет быстро определить новый вид объектов в терминах уже имеющегося. Новую реализацию вы можете получить посредством наследования большей части необходимого кода из ранее написанных классов. Однако не менее важно, что наследование позволяет определять семейства объектов с идентичными интерфейсами (обычно за счет наследования от абстрактных классов). Почему? Потому что от этого зависит полиморфизм. Если пользоваться наследованием осторожно (некоторые сказали бы правильно), то все классы, производные от некоторого абстрактного класса, будут обладать его интерфейсом. Отсюда следует, что

подкласс добавляет новые или замещает старые операции и не скрывает операций, определенных в родительском классе. Все подклассы могут отвечать на запросы, соответствующие интерфейсу абстрактного класса, поэтому они являются подтипами этого абстрактного класса. У манипулирования объектами строго через интерфейс абстрактного класса есть два преимущества: а клиенту не нужно иметь информации о конкретных типах объектов, которыми он пользуется, при условии, что все они имеют ожидаемый клиентом интерфейс; а клиенту необязательно «знать» о классах, с помощью которых реализованы объекты. Клиенту известно только об абстрактном классе (или классах), определяющих интерфейс. Данные преимущества настолько существенно уменьшают число зависимостей между подсистемами, что можно даже сформулировать принцип объектноориентированного проектирования для повторного использования: программируйте в соответствии с интерфейсом, а не с реализацией. Не объявляйте переменные как экземпляры конкретных классов. Вместо этого придерживайтесь интерфейса, определенного абстрактным классом. Это одна из наших ключевых идей. Конечно, где-то в системе вам придется инстанцировать конкретные классы, то есть определить конкретную реализацию. Как раз это и позволяют сделать порождающие паттерны: абстрактная фабрика, строитель, фабричный метод, прототип и одиночка. Абстрагируя процесс создания объекта, эти паттерны предоставляют вам разные способы прозрачно ассоциировать интерфейс с его реализацией в момент инстанцирования. Использование порождающих паттернов гарантирует, что система написана в терминах интерфейсов, а не реализации. Механизмы повторного использования Большинству проектировщиков известны концепции объектов, интерфейсов, классов и наследования. Трудность в том, чтобы применить эти знания для построения гибких, повторно используемых программ. С помощью паттернов проектирования вы сможете сделать это проще. Как решать задачи проектирования Наследование и композиция Два наиболее распространенных приема повторного использования функциональности в объектно-ориентированных системах - это наследование класса и композиция объектов. Как мы уже объясняли, наследование класса позволяет определить реализацию одного класса в терминах другого. Повторное использование за счет порождения подкласса называют еще прозрачным ящиком (whitebox reuse). Такой термин подчеркивает, что внутреннее устройство родительских классов видимо подклассам. Композиция объектов - это альтернатива наследованию класса. В этом случае новую, более сложную функциональность мы получаем путем объединения или композиции объектов. Для композиции требуется, чтобы объединяемые объекты имели четко определенные интерфейсы. Такой способ повторного использования называют черным ящиком (black-box reuse), поскольку детали внутреннего устройства объектов остаются скрытыми. И у наследования, и у композиции есть достоинства и недостатки. Наследование класса определяется статически на этапе компиляции, его проще использовать, поскольку оно напрямую поддержано языком программирования. В случае наследования классов упрощается также задача модификации существующей реализации. Если

подкласс замещает лишь некоторые операции, то могут оказаться затронутыми и остальные унаследованные операции, поскольку не исключено, что они вызывают замещенные. Но у наследования класса есть и минусы. Во-первых, нельзя изменить унаследованную от родителя реализацию во время выполнения программы, поскольку само наследование фиксировано на этапе компиляции. Во-вторых, родительский класс нередко хотя бы частично определяет физическое представление своих подклассов. Поскольку подклассу доступны детали реализации родительского класса, то часто говорят, что наследование нарушает инкапсуляцию [Sny86]. Реализации подкласса и родительского класса настолько тесно связаны, что любые изменения последней требуют изменять и реализацию подкласса. Зависимость от реализации может повлечь за собой проблемы при попытке повторного использования подкласса. Если хотя бы один аспект унаследованной реализации непригоден для новой предметной области, то приходится переписывать родительский класс или заменять его чем-то более подходящим. Такая зависимость ограничивает гибкость и возможности повторного использования. С проблемой можно справиться, если наследовать только абстрактным классам, поскольку в них обычно совсем нет реализации или она минимальна. Композиция объектов определяется динамически во время выполнения за счет того, что объекты получают ссылки на другие объекты. Композицию можно применить, если объекты соблюдают интерфейсы друг друга. Для этого, в свою очередь, требуется тщательно проектировать интерфейсы, так чтобы один объект можно было использовать вместе с широким спектром других. Но и выигрыш велик. Поскольку доступ к объектам осуществляется только через их интерфейсы, мы не нарушаем инкапсуляцию. Во время выполнения программы любой объект можно заменить другим, лишь бы он имел тот же тип. Более того, поскольку при введение в паттерны проектирования реализации объекта кодируются прежде всего его интерфейсы, то зависимость от реализации резко снижается. Композиция объектов влияет на дизайн системы и еще в одном аспекте. Отдавая предпочтение композиции объектов, а не наследованию классов, вы инкапсулируете каждый класс и даете ему возможность выполнять только свою задачу. Классы и их иерархии остаются небольшими, и вероятность их разрастания до неуправляемых размеров невелика. С другой стороны, дизайн, основанный на композиции, будет содержать больше объектов (хотя число классов, возможно, уменьшится), и поведение системы начнет зависеть от их взаимодействия, тогда как при другом подходе оно было бы определено в одном классе. Это подводит нас ко второму правилу объектно-ориентированного проектирования: предпочитайте композицию наследованию класса. В идеале, чтобы добиться повторного использования, вообще не следовало бы создавать новые компоненты. Хорошо бы, чтобы можно было получить всю нужную функциональность, просто собирая вместе уже существующие компоненты. На практике, однако, так получается редко, поскольку набор имеющихся компонентов все же недостаточно широк. Повторное использование за счет наследования упрощает создание новых компонентов, которые можно было бы применять со старыми. Поэтому наследование и композиция часто используются вместе. Тем не менее,

наш опыт показывает, что проектировщики злоупотребляют наследованием. Нередко дизайн мог бы стать лучше и проще, если бы автор больше полагался на композицию объектов. Делегирование С помощью делегирования композицию можно сделать столь же мощным инструментом повторного использования, сколь и наследование [Lie86, JZ91]. При делегировании в процесс обработки запроса вовлечено два объекта: получатель поручает выполнение операций другому объекту - уполномоченному. Примерно так же подкласс делегирует ответственность своему родительскому классу. Но унаследованная операция всегда может обратиться к объекту-получателю через переменную-член (в C++) или переменную self (в Smalltalk). Чтобы достичь того же эффекта для делегирования, получатель передает указатель на самого себя соответствующему объекту, дабы при выполнении делегированной операции последний мог обратиться к непосредственному адресату запроса. Например, вместо того чтобы делать класс Window (окно) подклассом класса Rectangle (прямоугольник) - ведь окно является прямоугольником, - мы можем воспользоваться внутри Window поведением класса Rectangle, поместив в класс Window переменную экземпляра типа Rectangle и делегируя ей операции, специфичные для прямоугольников. Другими словами, окно не является прямоугольником, а содержит его. Теперь класс Window может явно перенаправлять запросы своему члену Rectangle, а не наследовать его операции. На диаграмме ниже изображен класс Window, который делегирует операцию Area () над своей внутренней областью переменной экземпляра Rectangle. Как решать задачи проектирования Сплошная линия со стрелкой обозначает, что класс содержит ссылку на экземпляр другого класса. Эта ссылка может иметь необязательное имя, в данном случае прямоугольник. Главное достоинство делегирования в том, что оно упрощает композицию поведений во время выполнения. При этом способ комбинирования поведений можно изменять. Внутреннюю область окна разрешается сделать круговой во время выполнения, просто подставив вместо экземпляра класса Rectangle экземпляр класса Circle; предполагается, конечно, что оба эти класса имеют одинаковый тип. У делегирования есть и недостаток, свойственный и другим подходам, применяемым для повышения гибкости за счет композиции объектов. Заключается он в том, что динамическую, в высокой степени параметризованную программу труднее понять, нежели статическую. Есть, конечно, и некоторая потеря машинной производительности, но неэффективность работы проектировщика гораздо более существенна. Делегирование можно считать хорошим выбором только тогда, когда оно позволяет достичь упрощения, а не усложнения дизайна. Нелегко сформулировать правила, ясно говорящие, когда следует пользоваться делегированием, поскольку эффективность его зависит от контекста и вашего личного опыта. Лучше всего делегирование работает при использовании в составе привычных идиом, то есть в стандартных паттернах. Делегирование используется в нескольких паттернах проектирования: состояние, стратегия, посетитель. В первом получатель делегирует запрос объекту, представляющему его текущее состояние. В паттерне стратегия обработка запроса делегируется

объекту, который представляет стратегию его исполнения. У объекта может быть только одно состояние, но много стратегий для исполнения различных запросов. Назначение обоих паттернов - изменить поведение объекта за счет замены объектов, которым делегируются запросы. В паттерне посетитель операция, которая должна быть выполнена над каждым элементом составного объекта, всегда делегируется посетителю. В других паттернах делегирование используется не так интенсивно. Паттерн посредник вводит объект, осуществляющий посредничество при взаимодействии других объектов. Иногда объект-посредник реализует операции, переадресуя их другим объектам; в других случаях он передает ссылку на самого себя, используя тем самым делегирование как таковое. Паттерн цепочка обязанностей Введение в паттерны проектирования обрабатывает запросы, перенаправляя их от одного объекта другому по цепочке. Иногда вместе с запросом передается ссылка на исходный объект, получивший запрос, и в этом случае мы снова сталкиваемся с делегированием. Паттерн мост отделяет абстракцию от ее реализации. Если между абстракцией и конкретной реализацией имеется существенное сходство, то абстракция может просто делегировать операции своей реализации. Делегирование показывает, что наследование как механизм повторного использования всегда можно заменить композицией. Наследование и параметризованные типы Еще один (хотя и не в точности объектно-ориентированный) метод повторного использования имеющейся функциональности - это применение параметризованных типов, известных также как обобщенные типы (Ada, Eiffel) или шаблоны (C++). Данная техника позволяет определить тип, не задавая типы, которые он использует. Неспецифицированные типы передаются в виде параметров в точке использования. Например, класс List (список) можно параметризовать типом помещаемых в список элементов. Чтобы объявить список целых чисел, вы передаете тип integer в качестве параметра параметризованному типу List. Если же надо объявить список строк, то в качестве параметра передается тип String. Для каждого типа элементов компилятор языка создаст отдельный вариант шаблона класса List. Параметризованные типы дают в наше распоряжение третий (после наследования класса и композиции объектов) способ комбинировать поведение в объектно-ориентированных системах. Многие задачи можно решить с помощью любого из этих трех методов. Чтобы параметризовать процедуру сортировки операцией сравнения элементов, мы могли бы сделать сравнение: а операцией, реализуемой подклассами (применение паттерна шаблонный метод); а функцией объекта, передаваемого процедуре сортировки (стратегия); а аргументом шаблона в C++ или обобщенного типа в Ada, который задает имя функции, вызываемой для сравнения элементов. Но между тремя данными подходами есть важные различия. Композиция объектов позволяет изменять поведение во время выполнения, но для этого требуются косвенные вызовы, что снижает эффективность. Наследование разрешает предоставить реализацию по умолчанию, которую можно замещать в подклассах. С помощью параметризованных типов допустимо изменять типы, используемые классом.

Но ни наследование, ни параметризованные типы не подлежат модификации во время выполнения. Выбор того или иного подхода зависит от проекта и ограничений на реализацию. Ни в одном из паттернов, описанных в этой книге, параметризованные типы не используются, хотя изредка мы прибегаем к ним для реализации паттернов в C++. В языке вроде Smalltalk, где нет проверки типов во время компиляции, параметризованные типы не нужны вовсе. Как решать задачи проектирования Сравнение структур времени выполнения и времени компиляции Структура объектно-ориентированной программы на этапе выполнения часто имеет мало общего со структурой ее исходного кода. Последняя фиксируется на этапе компиляции; код состоит из классов, отношения наследования между которыми неизменны. На этапе же выполнения структура программы - быстро изменяющаяся сеть из взаимодействующих объектов. Две эти структуры почти независимы. Рассмотрим различие между агрегированием и осведомленностью (acquaintance) объектов и его проявления на этапах компиляции и выполнения. Агрегирование подразумевает, что один объект владеет другим или несет за него ответственность. В общем случае мы говорим, что объект содержит другой объект или является его частью. Агрегирование означает, что агрегат и его составляющие имеют одинаковое время жизни. Говоря же об осведомленности, мы имеем в виду, что объекту известно о другом объекте. Иногда осведомленность называют ассоциацией или отношением «использует». Осведомленные объекты могут запрашивать друг у друга операции, но они не несут никакой ответственности друг за друга. Осведомленность - это более слабое отношение, чем агрегирование; оно предполагает гораздо менее тесную связь между объектами. На наших диаграммах осведомленность будет обозначаться сплошной линией со стрелкой. Линия со стрелкой и ромбиком вначале обозначает агрегирование. Агрегирование и осведомленность легко спутать, поскольку они часто реализуются одинаково. В языке Smalltalk все переменные являются ссылками на объекты, здесь нет различия между агрегированием и осведомленностью. В C++ агрегирование можно реализовать путем определения переменных-членов, которые являются экземплярами, но чаще их определяют как указатели или ссылки. Осведомленность также реализуется с помощью указателей и ссылок. Различие между осведомленностью и агрегированием определяется, скорее, предполагаемым использованием, а не языковыми механизмами. В структуре, существующей на этапе компиляции, увидеть различие нелегко, но тем не менее оно существенно. Обычно отношений агрегирования меньше, чем отношений осведомленности, и они более постоянны. Напротив, отношения осведомленности возникают и исчезают чаще и иногда длятся лишь во время исполнения некоторой операции. Отношения осведомленности, кроме того, более динамичны, что затрудняет их выявление в исходном тексте программы. Коль скоро несоответствие между структурой программы на этапах компиляции и выполнения столь велико, ясно, что изучение исходного кода может сказать о работе системы совсем немного. Поведение системы во время выполнения должно определяться проектировщиком, а не языком. Соотношения между объектами и их типами нужно проектировать очень аккуратно, поскольку

именно от Введение в паттерны проектирования них зависит, насколько удачной или неудачной окажется структура во время выполнения. Многие паттерны проектирования (особенно уровня объектов) явно подчеркивают различие между структурами на этапах компиляции и выполнения. Паттерны компоновщик и декоратор полезны для построения сложных структур времени выполнения. Наблюдатель порождает структуры времени выполнения, которые часто трудно понять, не зная паттерна. Паттерн цепочка обязанностей также приводит к таким схемам взаимодействия, в которых наследование неочевидно. В общем можно утверждать, что разобраться в структурах времени выполнения невозможно, если не понимаешь специфики паттернов. Проектирование с учетом будущих изменений Системы необходимо проектировать с учетом их дальнейшего развития. Для проектирования системы, устойчивой к таким изменениям, следует предположить, как она будет изменяться на протяжении отведенного ей времени жизни. Если при проектировании системы не принималась во внимание возможность изменений, то есть вероятность, что в будущем ее придется полностью перепроектировать. Это может повлечь за собой переопределение и новую реализацию классов, модификацию клиентов и повторный цикл тестирования. Перепроектирование отражается на многих частях системы, поэтому непредвиденные изменения всегда оказываются дорогостоящими. Благодаря паттернам систему всегда можно модифицировать определенным образом. Каждый паттерн позволяет изменять некоторый аспект системы независимо от всех прочих, таким образом, она менее подвержена влиянию изменений конкретного вида. Вот некоторые типичные причины перепроектирования, а также паттерны, которые позволяют этого избежать: а при создании объекта явно указывается класс. Задание имени класса привязывает вас к конкретной реализации, а не к конкретному интерфейсу. Это может осложнить изменение объекта в будущем. Чтобы уйти от такой проблемы, создавайте объекты косвенно. Паттерны проектирования: абстрактная фабрика, фабричный метод, прототип; а зависимость от конкретных операций. Задавая конкретную операцию, вы ограничиваете себя единственным способом выполнения запроса. Если же не включать запросы в код, то будет проще изменить способ удовлетворения запроса как на этапе компиляции, так и на этапе выполнения. Паттерны проектирования: цепочка обязанностей, команда; а зависимость от аппаратной и программной платформ. Внешние интерфейсы операционной системы и интерфейсы прикладных программ (API) различны на разных программных и аппаратных платформах. Если программа зависит от конкретной платформы, ее будет труднее перенести на другие. Даже на «родной» платформе такую программу трудно поддерживать. Как решать задачи проектирования Поэтому при проектировании систем так важно ограничивать платформенные зависимости. Паттерны проектирования: абстрактная фабрика, мост; а зависимость от представления или реализации объекта. Если клиент «знает», как объект представлен, хранится или реализован, то при изменении объекта может оказаться необходимым изменить и клиента. Сокрытие этой информации от клиентов поможет уберечься от каскада изменений. Паттерны проектирования:

абстрактная фабрика, мост, хранитель, заместитель; а зависимость от алгоритмов. Во время разработки и последующего использования алгоритмы часто расширяются, оптимизируются и заменяются. Зависящие от алгоритмов объекты придется переписывать при каждом изменении алгоритма. Поэтому алгоритмы, вероятность изменения которых высока, следует изолировать. Паттерны проектирования: мост, итератор, стратегия, шаблонный метод, посетитель; а сильная связанность. Сильно связанные между собой классы трудно использовать порознь, так как они зависят друг от друга. Сильная связанность приводит к появлению монолитных систем, в которых нельзя ни изменить, ни удалить класс без знания деталей и модификации других классов. Такую систему трудно изучать, переносить на другие платформы и сопровождать. Слабая связанность повышает вероятность того, что класс можно будет повторно использовать сам по себе. При этом изучение, перенос, модификация и сопровождение системы намного упрощаются. Для поддержки слабо связанных систем в паттернах проектирования применяются такие методы, как абстрактные связи и разбиение на слои. Паттерны проектирования: абстрактная фабрика, мост, цепочка обязанностей, команда, фасад, посредник, наблюдатель; а расширение функциональности за счет порождения подклассов. Специализация объекта путем создания подкласса часто оказывается непростым делом. С каждым новым подклассом связаны фиксированные издержки реализации (инициализация, очистка и т.д.). Для определения подкласса необходимо также ясно представлять себе устройство родительского класса. Например, для замещения одной операции может потребоваться заместить и другие. Замещение операции может оказаться необходимым для того, чтобы можно было вызвать унаследованную операцию. Кроме того, порождение подклассов ведет к комбинаторному росту числа классов, поскольку даже для реализации простого расширения может понадобиться много новых подклассов. Композиция объектов и делегирование - гибкие альтернативы наследованию для комбинирования поведений. Приложению можно добавить новую функциональность, меняя способ композиции объектов, а не определяя новые подклассы уже имеющихся классов. С другой стороны, при интенсивном использовании композиции объектов проект может оказаться трудным для понимания. С помощью многих паттернов проектирования удается построить такое Введение в паттерны проектирования решение, где специализация достигается за счет определения одного подкласса и комбинирования его экземпляров с уже существующими. Паттерны проектирования: мост, цепочка обязанностей, компоновщик, декоратор, наблюдатель, стратегия; а неудобства при изменении классов. Иногда нужно модифицировать класс, но делать это неудобно. Допустим, вам нужен исходный код, а его нет (так обстоит дело с коммерческими библиотеками классов). Или любое изменение тянет за собой модификации множества существующих подклассов. Благодаря паттернам проектирования можно модифицировать классы и при таких условиях. Паттерны проектирования: адаптер, декоратор, посетитель. Приведенные примеры демонстрируют ту гибкость работы, которой можно добиться, используя паттерны при проектировании приложений.

Насколько эта гибкость необходима, зависит, конечно, от особенностей вашей программы. Давайте посмотрим, какую роль играют паттерны при разработке прикладных программ, инструментальных библиотек и каркасов приложений. Прикладные программы Если вы проектируете прикладную программу, например редактор документов или электронную таблицу, то наивысший приоритет имеют внутреннее повторное использование, удобство сопровождения и расширяемость. Первое подразумевает, что вы не проектируете и не реализуете больше, чем необходимо. Повысить степень внутреннего повторного использования помогут паттерны, уменьшающие число зависимостей. Ослабление связанности увеличивает вероятность того, что некоторый класс объектов сможет совместно работать с другими. Например, устраняя зависимости от конкретных операций путем изолирования и инкапсуляции каждой операции, вы упрощаете задачу повторного использования любой операции в другом контексте. К тому же результату приводит устранение зависимостей от алгоритма и представления. Паттерны проектирования также позволяют упростить сопровождение приложения, если использовать их для ограничения платформенных зависимостей и разбиения системы на отдельные слои. Они способствуют и наращиванию функ-. ций системы, показывая, как расширять иерархии классов и когда применять композицию объектов. Уменьшение степени связанности также увеличивает возможность развития системы. Расширение класса становится проще, если он не зависит от множества других. Инструментальные библиотеки Часто приложение включает классы из одной или нескольких библиотек предопределенных классов. Такие библиотеки называются инструментальными. Инструментальная библиотека - это набор взаимосвязанных, повторно используемых классов, спроектированный с целью предоставления полезных функций общего назначения. Примеры инструментальной библиотеки - набор контейнерных классов для списков, ассоциативных массивов, стеков и т.д., библиотека потокового ввода/вывода в C++. Инструментальные библиотеки не определяют Как решать задачи проектирования какой-то конкретный дизайн приложения, а просто предоставляют средства, благодаря которым в приложениях проще решать поставленные задачи, позволяют разработчику не изобретать заново повсеместно применяемые функции. Таким образом, в инструментальных библиотеках упор сделан на повторном использовании кода. Это объектно-ориентированные эквиваленты библиотек подпрограмм. Можно утверждать, что проектирование инструментальной библиотеки сложнее, чем проектирование приложения, поскольку библиотеки должны использоваться во многих приложениях, иначе они бесполезны. К тому же автор библиотеки не знает заранее, какие специфические требования будут предъявляться конкретными приложениями. Поэтому ему необходимо избегать любых предположений и зависимостей, способных ограничить гибкость библиотеки, следовательно, сферу ее применения и эффективность. Каркасы приложений Каркас - это набор взаимодействующих классов, составляющих повторно используемый дизайн для конкретного класса программ [Deu89, JF88]. Например, можно создать каркас для разработки графических редакторов в

разных областях: рисовании, сочинении музыки или САПР [VL90, Joh92]. Другим каркасом рекомендуется пользоваться при создании компиляторов для разных языков программирования и целевых машин [JML92]. Третий упростит построение приложений для финансового моделирования [BE93]. Каркас можно подстроить под конкретное приложение путем порождения специализированных подклассов от входящих в него абстрактных классов. Каркас диктует определенную архитектуру приложения. Он определяет общую структуру, ее разделение на классы и объекты, основные функции тех и других, методы взаимодействия объектов и классов и потоки управления. Данные параметры проектирования задаются каркасом, а прикладные проектировщики или разработчики могут сконцентрироваться на специфике приложения. В каркасе аккумулированы проектные решения, общие для данной предметной области. Акцент в каркасе делается на повторном использовании дизайна, а не кода, хотя обычно он включает и конкретные подклассы, которые можно применять непосредственно. Повторное использование на данном уровне меняет направление связей между приложением и программным обеспечением, лежащим в его основе, на противоположное. При использовании инструментальной библиотеки (или, если хотите, обычной библиотеки подпрограмм) вы пишете тело приложения и вызываете из него код, который планируете использовать повторно. При работе с каркасом вы, наоборот, повторно используете тело и пишете код, который оно вызывает. Вам приходится кодировать операции с предопределенными именами и параметрами вызова, но зато число принимаемых вами проектных решений сокращается. В результате приложение создается быстрее. Более того, все приложения имеют схожую структуру. Их проще сопровождать, и пользователям они представляются более знакомыми. С другой стороны, вы в какой-то мере жертвуете свободой творчества, поскольку многие проектные решения уже приняты за вас. Введение в паттерны проектирования Если проектировать приложения нелегко, инструментальные библиотеки - еще сложнее, то проектирование каркасов - задача самая трудная. Проектировщик каркаса рассчитывает, что единая архитектура будет пригодна для всех приложений в данной предметной области. Любое независимое изменение дизайна каркаса приведет к утрате его преимуществ, поскольку основной «вклад» каркаса в приложение - это определяемая им архитектура. Поэтому каркас должен быть максимально гибким и расширяемым. Поскольку приложения так сильно зависят от каркаса, они особенно чувствительны к изменениям его интерфейсов. По мере усложнения каркаса приложения должны эволюционировать вместе с ним. В результате существенно возрастает значение слабой связанности, в противном случае малейшее изменение каркаса приведет к целой волне модификаций. Рассмотренные выше проблемы проектирования актуальны именно для каркасов. Каркас, в котором они решены путем применения паттернов, может лучше обеспечить высокий уровень проектирования и повторного использования кода, чем тот, где паттерны не применялись. В отработанных каркасах обычно можно обнаружить несколько разных паттернов проектирования. Паттерны помогают адаптировать

архитектуру каркаса ко многим приложениям без повторного проектирования. Дополнительное преимущество появляется потому, что вместе с каркасом документируются те паттерны, которые в нем использованы [BJ94]. Тот, кто знает паттерны, способен быстрее разобраться в тонкостях каркаса. Но даже не работающие с паттернами увидят их преимущества, поскольку паттерны помогают удобно структурировать документацию по каркасу. Повышение качества документирования важно для всех типов программного обеспечения, но для каркасов этот аспект важен вдвойне. Для освоения работы с каркасами надо потратить немало усилий, и только после этого они начнут приносить реальную пользу. Паттерны могут существенно упростить задачу, явно выделив ключевые элементы дизайна каркаса. Поскольку между паттернами и каркасами много общего, часто возникает вопрос, в чем же различия между ними и есть ли они вообще. Так вот, существует три основных различия: а паттерны проектирования более абстрактны, чем каркасы. В код могут быть включены целые каркасы, но только экземпляры паттернов. Каркасы можно писать на разных языках программирования и не только изучать, но и непосредственно исполнять и повторно использовать. В противоположность этому паттерны проектирования, описанные в данной книге, необходимо реализовывать всякий раз, когда в них возникает необходимость. Паттерны объясняют намерения проектировщика, компромиссы и последствия выбранного дизайна; а как архитектурные элементы, паттерны проектирования мельче, чем каркасы. Типичный каркас содержит несколько паттернов. Обратное утверждение неверно; Как выбирать паттерн проектирования а паттерны проектирования менее специализированы, чем каркасы. Каркас всегда создается для конкретной предметной области. В принципе каркас графического редактора можно использовать для моделирования работы фабрики, но его никогда не спутаешь с каркасом, предназначенным специально для моделирования. Напротив, включенные в наш каталог паттерны разрешается использовать в приложениях почти любого вида. Хотя, безусловно, существуют и более специализированные паттерны (скажем, паттерны для распределенных систем или параллельного программирования), но даже они не диктуют выбор архитектуры в той же мере, что и каркасы. Значение каркасов возрастает. Именно с их помощью объектно-ориентированные системы можно использовать повторно в максимальной степени. Крупные объектно-ориентированные приложения составляются из слоев взаимодействующих друг с другом каркасов. Дизайн и код приложения в значительной мере определяются теми каркасами, которые применялись при его создании. 1.7. Как выбирать паттерн проектирования Если в распоряжение проектировщика предоставлен каталог из более чем 20 паттернов, трудно решать, какой паттерн лучше всего подходит для решения конкретной задачи проектирования. Ниже представлены разные подходы к выбору подходящего паттерна: а подумайте, как паттерны решают проблемы проектирования. В разделе 1.6 обсуждается то, как с помощью паттернов можно найти подходящие объекты, определить нужную степень их детализации, специфицировать их интерфейсы. Здесь же говорится и о некоторых иных подходах к решению задач с помощью паттернов; а пролистайте разделы каталога, описывающие

назначение паттернов. В разделе 1.4,перечислены назначения всех представленных паттернов. Ознакомьтесь с целью каждого паттерна, когда будете искать тот, что в наибольшей степени относится к вашей проблеме. Чтобы сузить поиск, воспользуйтесь схемой в таблице 1.1; а изучите взаимосвязи паттернов. На рис. 1.1 графически изображены соотношения между различными паттернами проектирования. Данная информация поможет вам найти нужный паттерн или группы паттернов; а проанализируйте паттерны со сходными целями. Каталог состоит из трех частей: порождающие паттерны, структурные паттерны и паттерны поведения. Каждая часть начинается со вступительных замечаний о паттернах соответствующего вида и заканчивается разделом, где они сравниваются друг с другом; а разберитесь в причинах, вызывающих перепроектирование. Взгляните на перечень причин, приведенный выше. Быть может, в нем упомянута ваша проблема? Затем обратитесь к изучению паттернов, помогающих устранить эту причину; Введение в паттерны проектирования а посмотрите, что в вашем дизайне должно быть изменяющимся. Такой подход противоположен исследованию причин, вызвавших необходимость перепроектирования. Вместо этого подумайте, что могло бы заставить изменить дизайн, а также о том, что бы вы хотели изменять без перепроектирования. Акцент здесь делается на инкапсуляции сущностей, подверженных изменениям, а это предмет многих паттернов. В таблице 1.2 перечислены те аспекты дизайна, которые разные паттерны позволяют варьировать независимо, устраняя тем самым необходимость в перепроектировании. 1.8. Как пользоваться паттерном проектирования Как пользоваться паттерном проектирования, который вы выбрали для изучения и работы? Вот перечень шагов, которые помогут вам эффективно применить паттерн: 1. Прочитайте описание паттерна, чтобы получить о нем общее представление. Особое внимание обратите на разделы «Применимость» и «Результаты» - убедитесь, что выбранный вами паттерн действительно подходит для решения ваших задач. 2. Вернитесь назад и изучите разделы «Структура», «Участники» и «Отношения». Убедитесь, что понимаете упоминаемые в паттерне классы и объекты и то, как они взаимодействуют друг с другом. 3. Посмотрите на раздел «Пример кода», где приведен конкретный пример использования паттерна в программе. Изучение кода поможет понять, как нужно реализовывать паттерн. 4. Выберите для участников паттерна подходящие имена. Имена участников паттерна обычно слишком абстрактны, чтобы употреблять их непосредственно в коде. Тем не менее бывает полезно включить имя участника как имя в программе. Это помогает сделать паттерн более очевидным при реализации. Например, если вы пользуетесь паттерном стратегия в алгоритме размещения текста, то классы могли бы называться SimpleLayoutStrategy или TeXLayoutStrategy. 5. Определите классы. Объявите их интерфейсы, установите отношения наследования и определите переменные экземпляра, которыми будут представлены данные объекты и ссылки на другие объекты. Выявите имеющиеся в вашем приложении классы, на которые паттерн оказывает влияние, и соответствующим образом модифицируйте их. 6. Определите имена

операций, встречающихся в паттерне. Здесь, как и в предыдущем случае, имена обычно зависят от приложения. Руководствуйтесь теми функциями и взаимодействиями, которые ассоциированы с каждой операцией. Кроме того, будьте последовательны при выборе имен. Например, для обозначения фабричного метода можно было бы всюду использовать префикс Create-. 7. Реализуйте операции, которые выполняют обязанности и отвечают за отношения, определенные в паттерне. Советы о том, как это лучше сделать, вы найдете в разделе «Реализация». Поможет и «Пример кода». Как пользоваться паттерном проектирования Все вышесказанное - обычные рекомендации. Со временем вы выработаете собственный подход к работе с паттернами проектирования. Таблица 1.2. Изменяемые паттернами элементы дизайна Назначение Паттерн проектирования Аспекты, которые можно изменять Порождающие паттерны Паттерны поведения Абстрактная фабрика Одиночка Прототип Строитель Фабричный метод Семейства порождаемых объектов Единственный экземпляр класса Класс, из которого инстанцируется объект Способ создания составного объекта Инстанцируемый подкласс объекта Структурные Адаптер Интерфейс к объекту паттерны Декоратор Обязанности объекта без порождения подкласса Заместитель Способ доступа к объекту, его местоположение Компоновщик Структура и состав объекта Мост Реализация объекта Приспособленец Накладные расходы на хранение объектов Фасад Интерфейс к подсистеме Интерпретатор Итератор Команда Наблюдатель Посетитель Посредник Состояние Стратегия Хранитель Цепочка обязанностей Шаблонный метод Грамматика и интерпретация языка Способ обхода элементов агрегата Время и способ выполнения запроса Множество объектов, зависящих от другого объекта; способ, которым зависимые объекты поддерживают себя в актуальном состоянии Операции, которые можно применить к объекту или объектам, не меняя класса Объекты, взаимодействующие между собой, и способ их кооперацій Состояние объекта Алгоритм Закрытая информация, хранящаяся вне объекта, и время ее сохранения Объект, выполняющий запрос Шаги алгоритма Никакое обсуждение того, как пользоваться паттернами проектирования, нельзя считать полным, если не сказать о том, как не надо их применять. Нередко за гибкость и простоту изменения, которые дают паттерны, приходится платить усложнением дизайна и ухудшением производительности. Паттерн проектирования стоит применять, только когда дополнительная гибкость действительно необходима. Для оценки достоинств и недостатков паттерна большую помощь могут оказать разделы каталога «Результаты».