Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Шаблоны проектирования — элементы многократно используемого объектно-ориентированного программного обеспечения.

Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес

patterns@narod.ru

Глава 1

Введение

Проектировать объектно-ориентированного (ОО) программного обеспечения (ПО) трудно, а проектировать *многократно используемое* ОО ПО еще труднее. Вам нужно найти подходящие объекты, объединить их в классы правильного размера, определить интерфейсы классов и иерархию наследования, установить ключевые взаимосвязи между ними. Ваш проект должен быть специфичным для решаемой задачи, но также достаточно общим, чтобы удовлетворять будущим требованиям и задачам. Вам также хочется избежать перепроектирования, или хотя бы минимизировать его. Опытные ОО проектировщики скажут вам, что многократно используемый и гибкий проект трудно, если не невозможно разработать “правильно” с первого раза. До окончания проекта, они обычно пытаются использовать его несколько раз, каждый раз изменяя.

Все же опытные ОО проектировщики делают хорошие проекты. В то же время начинающие проектировщики перегружены доступными возможностями и имеют тенденцию откатываться к не-ОО техникам, которые они использовали раньше. Новичкам требуется много времени, чтобы понять, что же такое хороший ОО проект. Опытные проектировщики, очевидно, знают что-то неизвестное неопытным. Что же это?

Одна из вещей которую эксперты знают *не* делать — это решение каждой задачи с нуля. Наоборот, они используют решения, которые работали в прошлом. Когда они находят хорошее решение, они используют его снова и снова. Такой опыт — часть того, что делает их экспертами. Следовательно, вы найдете повторяющиеся шаблоны классов и сообщающиеся объекты во многих ОО системах. Эти шаблоны решают специфические проектные проблемы и делают ОО проекты более гибкими, элегантными и чрезвычайно подходящими для многократного использования. Они помогают проектировщикам использовать успешные проекты, основывая новые проекты на предыдущем опыте. Проектировщик, ко-

торый знаком с такими шаблонами, может сразу же использовать их в текущих задачах без необходимости переоткрывать их.

Эту идею можно проиллюстрировать с помощью аналогии. Писатели и драматурги редко разрабатывают свои сюжеты с нуля. Вместо этого они следуют шаблонам, таким как "Трагический герой" (Макбет, Гамлет и т.д.) или "Романтическая история" (бесчисленные романтические повести). Аналогично ОО проектировщики следуют таким образцам как "представление состояний с помощью объектов" и "декорация объектов для упрощения добавления/удаления свойств". Если вы знаете шаблон, множество решений следуют автоматически.

Все мы знаем цену проектному опыту. Как много раз у вас было проектное *дежавю* — такое чувство, что вы уже решали подобную задачу раньше, но не помните точно где или как? Если бы вы могли запомнить детали предыдущей задачи и как вы ее решали, то вы смогли бы использовать этот опыт вместо перепридумывания его. Тем не менее, мы не записываем свой опыт в проектировании ПО для того, чтобы его могли использовать другие.

Цель этой книги записать опыт в разработке ОО ПО в виде **шаблонов проектирования.** Каждый шаблон систематизировано именует, объясняет и оценивает важное и повторяющееся проектное решение в ОО системах. Наша цель зафиксировать проектный опыт в такой форме, чтобы люди могли использовать его эффективно. Для этого мы документировали некоторые из самых важных проектных шаблонов и представили их в виде каталога.

Шаблоны проектирования упрощают повторное использование успешных проектных и архитектурных решений. Выражение проверенных техник в виде шаблонов проектирования делает их более доступными для разработчиков новых систем. Шаблоны проектирования помогают вам выбрать проектные альтернативы, которые позволяют многократно использовать систему и избежать альтернатив, которые этого не позволяют. Шаблоны проектирования могут даже улучшить документацию и сопровождение существующих систем, предоставляя явную спецификацию взаимодействий классов, объектов и их назначения. Проще говоря, шаблоны проектирования помогают сделать проект "правильно" быстрее.

Ни один из проектных шаблонов в этой книге не описывает новое или непроверенное проектное решение. Мы включили только решения, которые использовались более одного раза в разных системах. Большинство из них нигде раньше не документировались. Они являются либо частью фольклора ОО сообщества, либо элементами некоторых успешных ОО систем, и то и другое сложно изучить неопытным проектировщикам. Таким образом,

хотя эти проектные шаблоны не новы, мы зафиксировали их новым и доступным способом: в виде каталога проектных шаблонов имеющих последовательный формат.

Несмотря на размер книги, шаблоны проектирования охватывают только часть того, что может знать эксперт. В ней нет шаблонов для параллельного или распределенного программирования, программирования систем реального времени. В ней нет никаких шаблонов для приложений специфических для какой- либо предметной области. Она не рассказывает, как строить интерфейс пользователя, как писать драйверы устройств или как использовать ОО базы данных. Каждая из этих областей имеет свои собственные шаблоны, и их каталогизирование будет стоящим занятием для кого-либо другого.

1.1 Что такое шаблон проектирования?

Кристофер Александер говорит: “Каждый шаблон сначала описывает проблему, которая случается снова и снова в нашем окружении и затем описывает суть решения этой проблемы, таким образом, что вы можете использовать ее снова миллион раз, не повторяя один и тот же путь дважды”. Даже хотя Александер говорит о шаблонах в строениях и городах, его слова истинны и для ОО проектных шаблонов. Наши решения выражены в терминах объектов и интерфейсов вместо стен и дверей, но суть обоих видов шаблонов состоит в решении задачи в определенном контексте.

В общем шаблон состоит из 4 существенных элементов.

1. **Название шаблона** — это то, что мы можем использовать для описания проектной задачи, ее решений и последствий в одном двух словах. Именование шаблона сразу же увеличивает наш проектный словарь. Оно позволяет нам проектировать на более высоком уровне абстракции. Наличие словаря для шаблонов позволяет нам говорить о них с нашими коллегами, в нашей документации и даже с самими собой. Это облегчает обдумывание проектов и сообщение другим о них и их оптимальных вариантах. Поиск хороших имен был одной из сложнейших задач при разработке нашего каталога.

2. **Задача** описывает когда применять шаблон. Она объясняет задачу и ее контекст. Она может описывать специфические проектные проблемы, например как представлять алгоритмы в виде объектов. Она может описывать структуры классов или объектов являющихся симптомами негибкого проекта. Иногда задача включает список условий, которые должны выполняться для того, чтобы имело смысл применить шаблон.

3. **Решение** описывает элементы из которых состоит проектное решение, их связи, обязанности и взаимодействие. Решение не описывает конкретный проект или реализацию, т.к. шаблон может применяться во многих различных ситуациях. Вместо этого, решение представляет абстрактное описание проектной задачи и общее размещение элементов (в нашем случае классов или объектов) решающих ее.

4. **Последствия** — это результаты и оптимальные варианты применения шаблона. Хотя последствия часто замалчиваются, когда мы обсуждаем проектные решения, они важны для оценки проектных альтернатив и для понимания издержек и преимуществ применения шаблона. Последствия для ПО часто включают издержки использования памяти и времени. Они также могут относиться к деталям языка и реализации.

Поскольку многократность использования часто является фактором в ОО проектировании, последствия шаблона включают его влияние на гибкость, расширяемость и переносимость системы. Явное перечисление этих последствий помогает вам понять и оценить их.

Интерпретация того, что является шаблоном проектирования, а что нет, зависит от точки зрения. То, что является шаблоном для одного человека, для другого является примитивным строительным блоком. В этой книге мы сконцентрировались на шаблонах на определенном уровне абстракции. Шаблоны проектирования не описывают такие проектные решения как связные списки или хэш-таблицы, которые могут быть закодированы в виде классов и таким образом многократно использованы. Не являются они и сложными проблемно-ориентированными решениями для всего приложения или подсистемы. Проектные шаблоны в этой книге являются описанием сообщающихся классов и объектов, настроенных для решения общей проектной задачи в определенном контексте.

Проектный шаблон называет, резюмирует, и идентифицирует ключевые аспекты общей проектной структуры, что делает его полезным для создания многократно используемого ОО проекта. Проектный шаблон идентифицирует участвующие классы и их экземпляры, их роли и взаимодействие, а также распределение обязанностей. Каждый проектный шаблон фокусируется на конкретной задаче или вопросе ОО проекта. Шаблон описывает, когда он приложим, может ли он быть использован, учитывая прочие проектные ограничения, а также последствия и результаты его использования. Поскольку в конечном счете мы должны реализовывать наши проекты, шаблон также включает примерный код на C++ и (иногда) на Smalltalk для того, чтобы проиллюстрировать реализацию.

Хотя проектные шаблоны описывают ОО проекты, они основаны на практических решениях реализованных на наиболее распространенных ОО языках программирования, таких как Smalltalk и C++, а не на процедурных языках (Pascal, C, Ada) или более динамичных ОО языках (CLOS, Dylan, Self). Мы выбрали Smalltalk и C++ из прагматических соображений: наш повседневный опыт был в этих языках, и они все более популярны.

Выбор языка программирования важен поскольку он влияет на точку зрения человека. Наши шаблоны подразумевают язык содержащий возможности уровня Smalltalk/C++ , и этот выбор определяет, что может быть легко реализовано, а что нет. Если бы мы выбрали процедурные языки, мы могли бы включить такие шаблоны, как “Наследование”, “Инкапсуляция”, “Полиморфизм”. Аналогично, некоторые из наших шаблонов напрямую поддерживаются менее распространенными ОО языками. CLOS, например, имеет мульти-методы, которые уменьшают необходимость в таком шаблоне, как Посетитель (331). Фактически

между Smalltalk и C++ существует достаточно различий, чтобы какие-либо шаблоны легче реализовывались в одном языке, чем в другом. (см. например Итератор (257)).

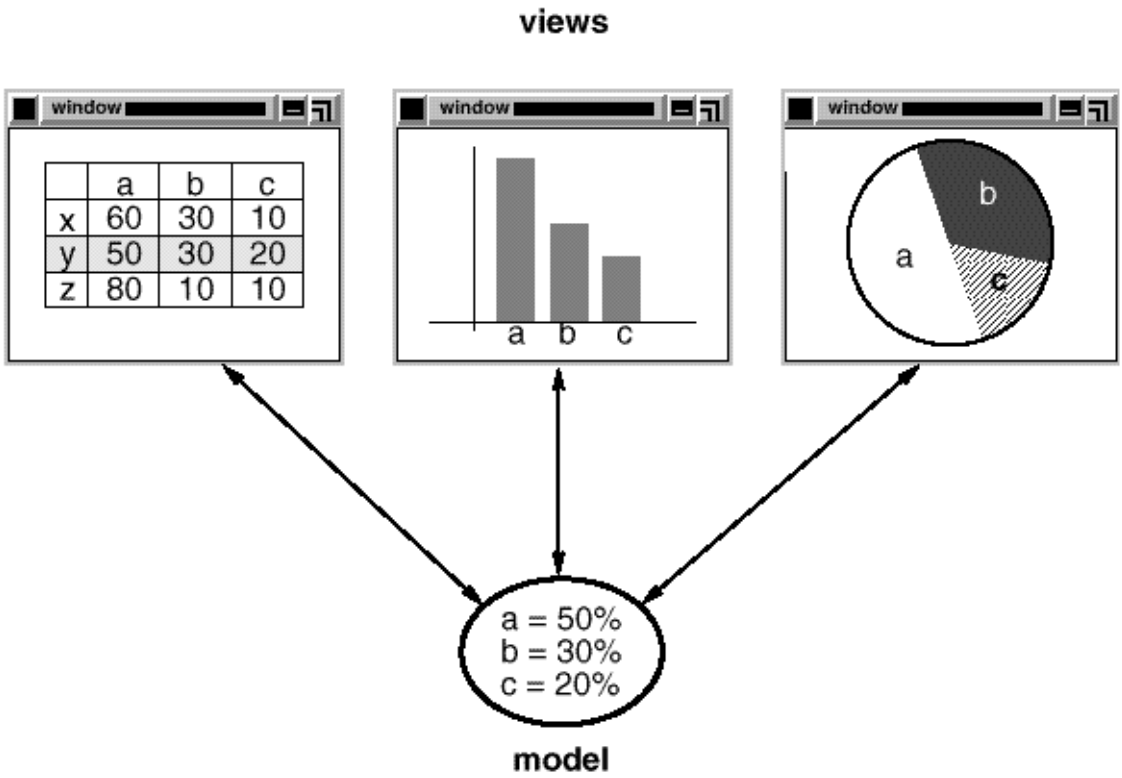
1.2 Шаблоны проектирования в Smalltalk MVC

Триада классов Модель/Вид/Диспетчер (Model/View/Controller – MVC) использовалась для построения интерфейса пользователя в Smalltalk-80. Просмотр проектных шаблонов внутри MVC должен помочь вам понять что мы понимаем под термином “шаблон”.

MVC состоит из трех видов объектов. Модель — это объект приложения, Вид — это его экранное представление, а Диспетчер определяет, как интерфейс реагирует на действия пользователя. До MVC проекты пользовательских интерфейсов имели тенденцию соединять эти объекты вместе. MVC разделяет их чтобы увеличить гибкость и степень многократного использования.

MVC разделяет виды и модели, устанавливая между ними протокол подписки/уведомления. Вид должен удостовериться, что его появление отражает состояние модели. Всякий раз, когда данные модели меняются, модель уведомляет виды, которые зависят от нее. В ответ каждый вид получает возможность обновить себя. Этот подход позволяет вам присоединять несколько видов к модели для получения различных представлений. Также вы можете создавать новые виды для модели без ее переписывания.

Следующая диаграмма показывает модель и три вида. (Для простоты мы опустили объекты-диспетчеры). Модель содержит некоторые данные, а виды определяют электронную таблицу, гистограмму и круговую диаграмму отображающую эти данные различными способами. Модель взаимодействует со своими видами когда данные меняются, а виды сообщаются с моделью для доступа к этим данным.



Номинально этот пример отражает проект, который отделяет виды от моделей. Но проектное решение применимо к более общей задаче: разделение объектов, таким образом, чтобы изменения в одном объекте могли влиять на любое количество других, без необходимости для изменяемого объекта знать о них детали. Этот более общее решение описано шаблоном Наблюдатель (стр. 293).

Другим свойством MVC является возможность вложения видов. Например, контрольная панель с кнопками может быть реализована как составной вид содержащий вложенные виды кнопок. Пользовательский интерфейс для инспектора объектов может состоять из вложенных видов, которые могут быть повторно использованы в отладчике. MVC поддерживает вложенные виды с помощью класса CompositeView, являющегося подклассом View. Объекты класса CompositeView действуют в точности как объекты класса View; составной вид может быть использован везде где может использоваться вид, но он также содержит вложенными виды и управляет ими.

И снова, мы можем думать об этом, как о проектном решении, которое позволяет работать с составным видом так же как с одним из его компонентов. Но проектное решение применимо к более общей задаче, которая появляется всегда, когда мы хотим сгруппировать объекты и работать с группой, как с отдельным объектом. Это более общее решение описано в шаблоне Композит(163). Он позволяет вам создать иерархию классов в которых некоторые подклассы определяют примитивные объекты (например Кнопка), а другие

классы определяют составные объекты (CompositeView), которые собирают примитивы в более сложные объекты.

Также MVC позволяет изменять реакцию вида на ввод пользователя без изменения его визуального представления. Например вы хотите изменить его реакцию на ввод с клавиатуры, или использовать управляющее меню вместо командных клавиш. MVC инкапсулирует механизм ответа в объекте Диспетчер. Существует иерархия классов диспетчеров, упрощающая создание нового диспетчера в виде варианта уже существующего.

Вид использует экземпляр подкласса Диспетчер для реализации конкретной стратегии ответа; для реализации другой стратегии, просто замените экземпляр другим видом диспетчера. Можно даже заменить диспетчер вида на другой во время выполнения, чтобы позволить виду изменить реакцию на ввод пользователя. Например, вид может быть отключен таким образом, чтобы он не воспринимал ввод пользователя просто подключением к нему диспетчера игнорирующего ввод.

Связь Вид-Диспетчер является примером проектного шаблона Стратегия (315). Стратегия — это объект, который представляет собой алгоритм. Он полезен, когда вы хотите заменить алгоритм статически или динамически, когда у вас есть много вариантов алгоритма, или когда алгоритм содержит сложные структуры данных, которые вы хотите инкапсулировать.

MVC использует другие шаблоны проектирования, такие как Заводской Метод (107) для указания диспетчера вида по умолчанию и Декоратор(107) для добавления скроллинга к виду. Но основные связи в MVC представлены проектными шаблонами Наблюдатель, Композит и Стратегия.

1.3 Описание Проектных Шаблонов

Как мы описываем проектные шаблоны? Графические обозначения важны и полезны, но недостаточны. Они просто фиксируют конечный результат процесса проектирования в виде связей между классами и объектами. Для многократного использования проекта, мы должны также записать наши решения, альтернативы и оптимальные варианты, которые к нему привели. Конкретные примеры также важны, потому что они помогают увидеть проект в действии.

Мы описываем проектные шаблоны, используя последовательный формат. Описание каждого шаблона делится на параграфы по следующему шаблону :). Этот шаблон предоставляет единообразную структуру для информации, упрощая изучение, использование и сравнение проектных шаблонов.

Название шаблона и классификация

> Название шаблона кратко передает его сущность. Хорошее имя важно, поскольку оно станет частью вашего проектного словаря. Классификация шаблонов отражает схему из раздела 1.5.

Намерение

> Короткое утверждение отвечающее на следующие вопросы: Что делает проектный шаблон? Каково его обоснование и назначение? Для какой именно проектной задачи он предназначен?

Также Известен Как

> Другие хорошо известные имена для шаблона, если есть.

Мотивация

> Сценарий иллюстрирующий проектную задачу и показывающий как структуры классов и объектов в шаблоне решают ее. Сценарий поможет вам понять более абстрактное описание шаблона.

Сфера применения

> Каковы ситуации в которых может быть применен проектный шаблон? Каковы примеры плохих проектов в которых он может быть применен? Как вы можете распознать такие ситуации?

Структура

Графическое представление классов в шаблоне (используемая нотация основана на Технике Моделирования Объектов) (OMT) [RBP+91]. Мы также используем диаграммы взаимодействия [JCJO92, Boo94] для иллюстрации последовательности запросов и взаимодействий между объектами. В Приложении B эти нотации описываются подробно.

Участники

Классы и/или объекты, участвующие в проектном шаблоне, а также их обязанности.

Взаимодействия

Как участники взаимодействуют для выполнения своих обязанностей.

Последствия

Как шаблон реализует свои цели? Каковы издержки и результаты использования шаблона? Какой аспект структуры системы он позволяет вам изменять независимо?

Реализация

В наличии каких ловушек, трюков и техник вы должны отдавать себе отчет при реализации шаблона? Существуют ли проблемы специфические для определенного языка?

Пример кода

Фрагменты кода, которые показывают как вы можете реализовать шаблон на C++ или Smalltalk.

Известные использования

Примеры использования шаблона в реальных системах. Мы включили как минимум два примера из разных областей применения.

Связь с шаблонами

Какие шаблоны проектирования тесно связаны с этим? Каковы важные различия? С какими другими шаблонами должен использоваться данный шаблон?

Приложения предоставляют дополнительную информацию, которая поможет вам понять шаблоны и их обсуждение. Приложение A — это словарь использованной терминологии. Мы уже упоминали приложение B в котором представлены различные нотации. Мы также опишем аспекты нотаций когда мы представим их в последующих обсуждениях. И

наконец приложение C содержит исходный код для базовых классов используемых нами в примерах.

1.4 Каталог проектных шаблонов

Каталог начинается на странице 79 и содержит 23 проектных шаблона. Их имена и намерения перечислены ниже в виде обзора. Номер в скобках после каждого шаблона означает номер страницы на которой описан шаблон (этого соглашения мы придерживаемся во всей книге)

Абстрактная фабрика (Abstract Factory) (87)

Предоставляет интерфейс для создания семейств связанных или зависимых объектов без указания их конкретных классов.

Адаптер (Adapter) (139)

Преобразует интерфейс класса в другой интерфейс, ожидаемый клиентами. Адаптер позволяет классам с несовместимыми интерфейсами работать вместе.

Мост (Bridge) (151)

Отцепляет абстракцию от ее реализации так, чтобы они могли изменяться независимо.

Строитель (Builder) (97)

Отделяет конструирование сложных объектов от его представления, так чтобы один конструирующий процесс мог создавать различные представления.

Цепь ответственности (Chain of Responsibility) (223)

Избегает связывания объекта пославшего запрос с его получателем путем предоставления возможности обработки запроса более чем одному объекту. Сцепляет объекты получатели и передает запрос по цепочке до тех пор пока какой-либо объект не обработает его.

Команда (Command) (233)

Инкапсулирует запрос в объект, таким образом, позволяя параметризовать клиентов различными запросами, ставить запросы в очередь или протоколировать их, а также поддерживать отменяемые операции.

Композит (Composite) (163)

Соединяет объекты в древовидные структуры для представления иерархий вида целое-часть. Композит позволяет клиентам обращаться с отдельными объектами также как и с группами объектов.

Декоратор (Decorator) (175)

Динамически присоединяет к объекту дополнительные обязанности. Декораторы предоставляют гибкую альтернативу сабклассингу для расширения функциональности.

Фасад (Facade) (185)

Предоставляет унифицированный интерфейс для множества интерфейсов в подсистеме. Фасад определяет интерфейс более высокого уровня для облегчения использования подсистемы.

Заводской метод (Factory Method) (107)

Определяет интерфейс для создания объекта, но позволяет подклассам решать экземпляр какого класса создать. Заводской метод позволяет классу уступать создание подклассам.

Легковес (Flyweight) (195)

Использует разделение для эффективной работы с большим количеством маленьких объектов.

Интерпретатор (Interpreter) (243)

Для данного языка определяет представление его грамматики вместе с интерпретатором, который использует грамматику для интерпретации предложений на этом языке.

Итератор (Iterator) (257)

Предоставляет способ последовательного доступа к элементам составного объекта без раскрытия его представления.

Посредник (Mediator) (273)

Определяет объект, инкапсулирующий взаимодействие некоторого множества объектов. Посредник обеспечивает ослабление сцепления, не давая объектам ссылаться друг на друга явно, и позволяет независимо изменять их взаимодействие.

Память (Memento) (283)

Без нарушения инкапсуляции фиксирует и сохраняет внутреннее состояние объекта, чтобы можно было вернуться к этому состоянию позже.

Наблюдатель (Observer) (293)

Определяет отношение один ко многим между объектами таким образом, что когда один объект изменяет состояние, все зависящие от него объекты уведомляются и обновляются автоматически.

Прототип (Prototype) (117)

Указывает какие виды объектов создавать, используя прототипный экземпляр и создает новые объекты копированием этого прототипа.

Полномочия (Proxy) (207)

Предоставляет заменитель для другого объекта для контроля доступа к нему.

Одиночка (Singleton) (127)

Проверяет что класс имеет только один экземпляр и обеспечивает глобальную точку доступа к нему.

Состояние (State) (305)

Позволяет объекту изменять свое поведение когда его внутреннее состояние меняется. Создает впечатление, что объект сменил свой класс.

Стратегия (Strategy) (315)

Определяет семейство алгоритмов, инкапсулирует каждый и делает их взаимозаменяемыми. Стратегия позволяет изменять алгоритм независимо от клиентов его использующих.

Шаблонный метод (Template Method) (325)

Определяет скелет алгоритма в операции, оставляя некоторые шаги для реализации в подклассах. Шаблонный метод позволяет подклассам переопределять определенные шаги алгоритма без изменения его структуры.

Посетитель [Visitor] (331)

Представляет действие, которое необходимо произвести над элементами объектной структуры. Посетитель позволяет определить новое действие без изменения классов элементов над которыми это действие производится.

1.5 Организация каталога

Проектные шаблоны различаются по степени детализации и уровню абстракции. Поскольку существует много проектных шаблонов, нам нужен способ их организации. В этом разделе проектные шаблоны классифицируются, чтобы мы могли ссылаться на семейства связанных шаблонов. Классификация позволяет вам изучать шаблоны в каталоге быстрее, а также она может направить ваши усилия для поиска новых шаблонов.

Мы классифицируем проектные шаблоны по двум критериям (Таблица 1.1). Первый критерий - назначение, показывает что делает шаблон. Шаблоны могут иметь создающее, структурное или поведенческое назначение. Создающие шаблоны имеют отношение к процессу создания объекта. Структурные шаблоны имеют дело с композицией классов или объектов. Поведенческие шаблоны характеризуют пути взаимодействия объектов и распределения ответственности между ними.

	Создающие	**Структурные**	**Поведенческие**
Классовые	Заводской метод (107)	Адаптер (139)	Интерпретатор (243) Шаблонный метод (325)
Объектные	Абстрактная фабрика(87) Строитель (97) Прототип (117) Одиночка (127)	Адаптер (139) Мост (151) Композит (163) Декоратор (175) Фасад (185) Полномочия(207)	Цепь ответственности(223) Команда (233) Итератор (257) Посредник (273) Память (283) Легковес (195) Наблюдатель (293) Состояние (305) Стратегия (315) Посетитель (331)

Таблица 1.1: Пространство проектных шаблонов

Второй критерий — **область применения**, указывает применим ли шаблон в основном к классам или объектам. Шаблоны классов имеют дело со связями между классами и их подклассами. Эти связи устанавливаются через наследование, таким образом они статически определены во время компиляции. Объектные шаблоны имеют дело со связями между объектами, которые более динамичны и могут быть изменены во время выполнения. Почти все шаблоны в какой-то мере используют наследование. Поэтому только

шаблоны отмеченные как “шаблоны классов” фокусируются на связях классов. Заметьте, что большинство шаблонов работают с объектами.

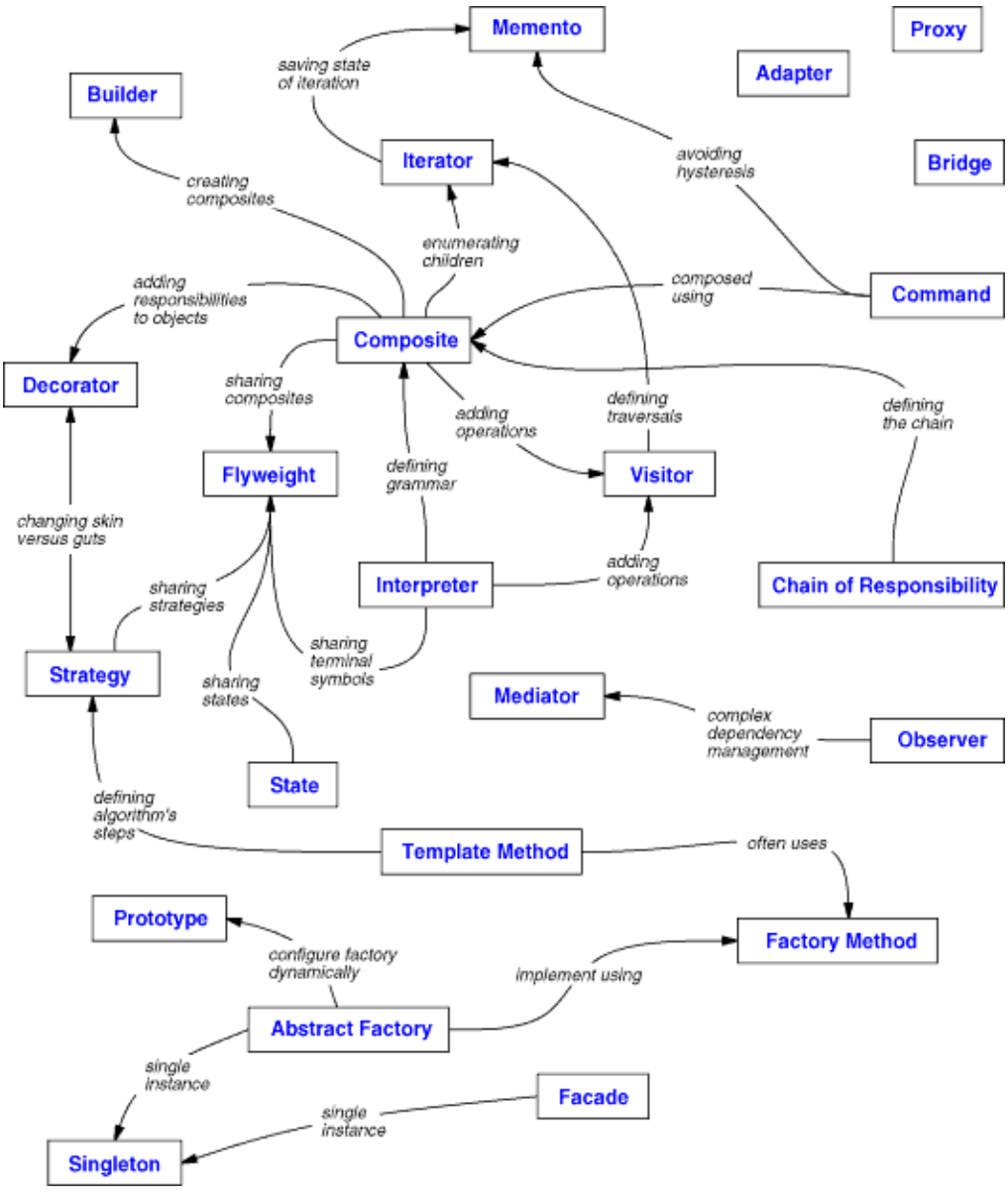
Создающие шаблоны класса оставляют часть процесса создания объекта подклассам, тогда как Создающие объектные шаблоны оставляют часть создания другому объекту. Структурные шаблоны класса используют наследование для соединения классов, тогда как Структурные шаблоны объектов описывают способы сборки объектов. Поведенческие шаблоны класса используют наследование для описания алгоритмов и потока управления, а Поведенческие шаблоны объекта описывают как группа объектов взаимодействует для решения задачи, которую ни один объект не может выполнить самостоятельно.

Существуют другие способы организации шаблонов. Некоторые шаблоны часто используются вместе. Например Композит часто используется с Итератором и Посетителем. Некоторые шаблоны имеют альтернативы: Прототип часто является альтернативой Абстрактной Фабрике. Некоторые шаблоны ведут к похожим проектным решениям, даже если эти шаблоны имеют разные намерения. Например структурные диаграммы Композита и Декоратора похожи.

Еще один способ организации шаблонов — согласно их ссылкам друг на друга в разделе “Связанные шаблоны”. Рисунок 1.1 описывает эти связи графически.

Несомненно существует много способов организации проектных шаблонов. Наличие нескольких способов размышления о шаблонах углубит ваше понимание того, что они делают, как они соотносятся и как их применять.

Рис. 1.1: Взаимосвязи проектных шаблонов



1.6 Как проектные шаблоны решают проектные задачи

Проектные шаблоны решают множество ежедневных проблем с которыми сталкиваются ОО проектировщики, множеством разных способов. Далее перечислены несколько подобных задач и как проектные шаблоны их решают.

1.6.1 Поиск необходимых объектов

ОО программы состоят из **объектов**. Объект содержит как данные так и процедуры работающие с этими данными. Процедуры обычно называются **методами** или **операциями**. Объект выполняет операцию когда он получает **запрос** (или **сообщение**) от **клиента**.

Запросы это единственный способ заставить объект выполнить операцию. Операции это единственный способ изменения внутренних данных объекта. Из-за этих ограничений, внутреннее состояние объекта называется **инкапсулированным**; оно недоступно напрямую и его представление невидимо извне объекта.

Трудной частью ОО проектирования является декомпозиция системы на объекты. Эта задача трудна, потому что на нее влияют многие факторы: инкапсуляция, размер, зависимость, гибкость, производительность, эволюция, повторное использования и т.д. и т.п. Все это влияет на декомпозицию и часто противоречит друг другу.

ОО методологии проектирования используют множество разных подходов. Вы можете написать задачу в виде утверждения, выделить существительные и глаголы и создать соответствующие классы и операции. Или вы можете сфокусироваться на взаимодействиях и ответственности в вашей системе. Или смоделировать реальный мир и перевести объекты найденные в ходе анализа в проект. Всегда будут существовать разногласия, о том какой подход лучше.

Многие объекты приходят в проект из аналитической модели. Но ОО проекты часто приходят к классам у которых нет эквивалентов в реальном мире. Некоторые из них это низкоуровневые классы, такие как массивы. Другие гораздо более высокоуровневые. Например шаблон Композит (163) представляет абстракцию для работы с объектами единообразно, не имеющую физического эквивалента. Строгое моделирование реального мира приводит к системе, которая отражает сегодняшние реалии, но необязательно завтрашние. Абстракции возникающие во время проектирования являются ключевыми для создания гибкого проекта.

Проектные шаблоны помогают вам идентифицировать менее явные абстракции и объекты, которые могут использовать их. Например объекты, представляющие процесс или алгоритм не существуют в природе, но они являются важной частью гибких проектов.

Шаблон Стратегия (315) описывает как реализовать взаимозаменяемые семейства алгоритмов. Шаблон Состояние (305) представляет каждое состояние сущности в виде объекта. Эти объекты редко обнаруживаются во время анализа или даже на ранних стадиях проектирования; они обнаруживаются позже в процессе переделки проекта для большей гибкости и многократного использования.

1.6.2 Определение размера объектов

Объекты могут очень сильно отличаться в размере и количестве. Они могут представлять все от аппаратной части до целых приложений. Как же решить что должно быть объектом?.

Проектные шаблоны решают также и этот вопрос. Шаблон Фасад (185) описывает как представлять целые подсистемы в виде объектов, а шаблон Легковес (195) описывает как удерживать огромное количество объектов в небольшом объеме. Другие проектные шаблоны описывают специфические пути декомпозиции объекта на меньшие объекты. Абстрактная Фабрика (87) и Строитель (97) представляют объекты, единственной обязанностью которых является создание других объектов. Посетитель (331) и Команда (233) собирают объекты единственной обязанностью которых является реализация запроса другого объекта или группы объектов.

1.6.3 Определение Объектных Интерфейсов

Каждая операция в объекте определяется именем, объектами, принимаемыми в качестве параметров, и результатом. Это так называемая **сигнатура** операции. Множество всех сигнатур, определенных в объекте операций, называется **интерфейсом** объекта. Интерфейс объекта характеризует полное множество запросов, которые можно направлять объекту. Любой запрос подходящий под сигнатуру в интерфейсе объекта может быть послан объекту.

Тип это имя используемое для ссылки на соответствующий интерфейс. Мы говорим, что объект имеет тип “Окно”, если он принимает все запросы для операций определенных в интерфейсе “Окно”. Объект может иметь много типов и многие различные объекты могут разделять один тип. Часть интерфейса объекта может быть описана одним типом, а другие части другими типами. Два объекта одного и того же типа должны разделять только части их интерфейсов. Интерфейсы могут содержать другие интерфейсы в качестве подмножеств. Мы говорим, что тип является **подтипом** другого типа, если его интерфейс

содержит интерфейс его **супертипа**. Часто мы говорим о подтипе *наследующем* интерфейс своего супертипа.

Интерфейсы фундаментальны для ОО систем. Объекты известны только через их интерфейсы. Невозможно узнать что-нибудь об объекте или попросить его сделать что-то не через интерфейс. Интерфейс объекта ничего не сообщает о его реализации — разные объекты могут обрабатывать запросы по разному. Это означает, что два объекта, имеющие совершенно разные реализации, могут иметь идентичные интерфейсы.

Когда запрос послан объекту, выполнение соответствующей операции зависит и от запроса и от получающего объекта. Различные объекты поддерживающие одинаковые запросы могут по разному реализовывать операции выполняющие эти запросы. Связывание запроса к объекту с одной из его операций во время выполнения известно как **динамическое связывание**.

Динамическое связывание означает что вызов запроса не привязывает вас к конкретной реализации до времени выполнения. Следовательно вы можете писать программы, которые ожидают объект с конкретным интерфейсом, зная, что каждый объект с корректным интерфейсом обработает запрос. Более того, динамическое связывание позволяет вам заменять объекты с идентичными интерфейсами друг на друга во время выполнения. Такая замещаемость называется **полиморфизмом** и является ключевой концепцией в ОО системах. Это позволяет клиентским объектам не делать много предположений о других объектах, кроме поддержки соответствующего интерфейса. Полиморфизм упрощает определения клиентов, отделяет объекты друг от друга и позволяет им менять связь друг с другом во время выполнения.

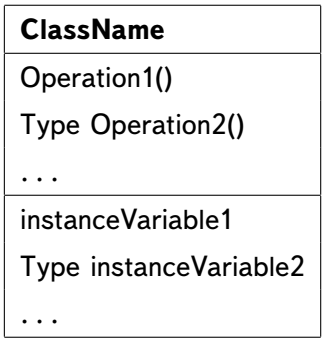
Проектные шаблоны помогают вам определить интерфейсы, идентифицируя их ключевые элементы и данные пересылаемые через интерфейс. Хороший пример этого — шаблон Память (283). Он описывает как инкапсулировать и сохранить внутреннее состояние объекта так, чтобы объект мог вернуться к этому состоянию позже. В шаблоне указывается, что объекты Память должны определять два интерфейса: ограниченный, который позволяет клиентам содержать и копировать объекты, и привилегированный, который может использовать исходный объект для сохранения и получения своего состояния.

Проектные шаблоны также определяют связи между интерфейсами. В частности они часто требуют, чтобы некоторые классы имели похожие интерфейсы. Например шаблоны Декоратор (175) и Полномочия (207) требуют чтобы интерфейсы их объектов были идентичны. В шаблоне Посетитель (331), интерфейс должен отражать все классы объектов которые он может посещать.

1.6.4 Определение Реализаций Объектов

До этого мы мало говорили о том, как мы действительно определяем объект. Реализация объекта определяется его **классом**. Класс специфицирует внутренние данные объекта и его представление, а также определяет операции, которые объект может выполнять.

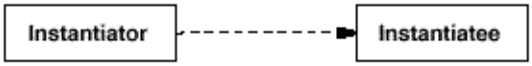
Наша нотация основанная на OMT (описанная в приложении B) изображает класс как прямоугольник, в котором жирным шрифтом написано имя класса. Операции написаны нормальным шрифтом ниже имени класса. Любые данные, которые определяет класс идут после действий. Имя класса, действия и данные разделяются линиями.



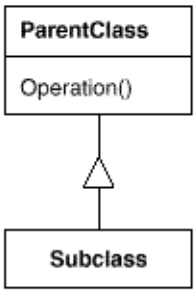
Типы возвращаемых данных и тип экземпляра класса опциональны, т.к. мы не предполагаем использование статически типизированного языка.

Объекты создаются **инстанцированием** класса. Объект называется **экземпляром** класса. Процесс инстанцирования класса выделяет пространство для данных объекта и связывает операции с этими данными. Многие одинаковые экземпляры объектов могут созданы инстанцированием класса.

Пунктирная линия со стрелкой показывает класс, который инстанцирует объекты другого класса. Стрелка указывает на класс создаваемых объектов.



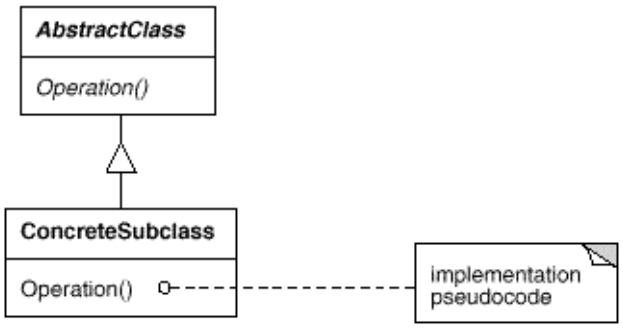
Новые классы могут быть определены в терминах существующих классов с использованием **наследования**. Когда **подкласс** наследует от **родительского класса**, он включает определения всех данных и операций родительского класса. Объекты экземпляры подкласса будут содержать все данные определенные подклассом и его родительскими классами, и они смогут выполнять все операции определенные в подклассе и в его родителях. Мы показываем отношения подкласса вертикальной чертой с треугольником.



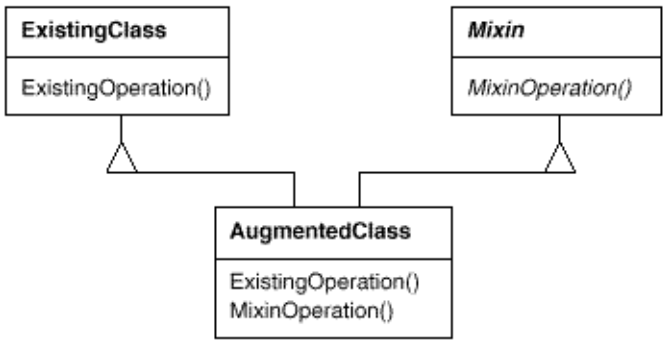
Абстрактный класс это класс, основная цель которого определить общий интерфейс для его подклассов. Абстрактный класс оставляет часть или всю свою реализацию операциям определенных в подклассах. Таким образом, абстрактный класс не может быть инстанцирован. Операции, которые абстрактный класс определяет, но не реализовывает называются **абстрактными операциями**. Не абстрактные классы называются **конкретными**.

Подклассы могут совершенствовать и переопределять поведение родительских классов. Точнее класс может **заменить** операцию определенную в родительском классе. Переопределение дает подклассам возможность обрабатывать запросы вместо их родительских классов. Наследование позволяет вам определять классы, просто расширяя другие классы, и упрощая тем самым создание семейств объектов с похожей функциональностью.

Имена абстрактных классов пишутся наклонным шрифтом для того, чтобы отличить их от конкретных классов. Наклонный шрифт также используется для определения абстрактных операций. Диаграмма может включать псевдокод для реализации операций; в таком случае код изображается в прямоугольнике с загнутым уголком подсоединенным пунктирной линией к операции которую он реализует.



Класс-примесь — это класс, который предоставляет опциональный интерфейс или функциональность для других классов. Он похож на абстрактный класс, в том, что он также не предназначен для инстанцирования. Классы-примеси требуют наличия множественного наследования:



Наследование классов против наследования интерфейсов.

Важно понять разницу между *классом* объекта и его *типом*.

Класс определяет внутреннее состояние объекта и реализацию его операций. С другой стороны тип объекта только ссылается на его интерфейс — множество запросов, на которые объект реагирует. Объект может иметь много типов, и объекты разных классов могут иметь один тип.

Конечно, существует тесная взаимосвязь между классом и типом. Поскольку класс определяет операции выполняемые объектом, он также определяет тип объекта. Когда мы говорим, что объект это экземпляр класса, мы подразумеваем, что объект поддерживает интерфейс определенный классом.

Такие языки как C++ и Eiffel используют классы для указания как типа объекта так и его реализации. Smalltalk программы не определяют типы переменных; следовательно компилятор не проверяет, что типы объектов присвоенных переменной являются подтипами типа этой переменной. Посылка сообщения требует проверки, что класс получателя обрабатывает это сообщение, но не требует проверки того, что получатель является экземпляром определенного класса.

Также важно понять разницу между наследованием классов и интерфейсов. Наследование классов определяет реализацию объекта в терминах реализации другого объекта. Вкратце это механизм совместного использования кода и представления данных. С другой стороны наследование интерфейсов объясняет, когда один объект может быть использован вместо другого.

Легко перепутать эти две концепции, поскольку многие языки явно не разделяют их. В таких языках как C++ и Eiffel, наследование означает одновременно наследование интерфейса и реализации. Стандартный способ унаследовать интерфейс в C++ — унаследовать открыто от класса, который имеет (чисто) виртуальные функции-члены. Чистое наследование интерфейсов может быть приблизительно реализовано в C++ открытым на-

следованием от чисто абстрактных классов. Чистое наследование реализации или класса приблизительно реализуется закрытым наследованием. В Smalltalk наследуется только реализация. Вы можете присваивать экземпляры любого класса переменной, если они поддерживают операции осуществляемые над значением переменной.

Хотя многие языки программирования не поддерживают различие между наследованием реализации и интерфейса, люди делают различие на практике. Программирующие на Smalltalk обычно действуют так если бы подклассы были подтипами (хотя есть несколько известных исключений); Программирующие на C++ работают с объектами через типы определенные в абстрактных классах.

Многие из шаблонов проектирования зависят от этого различия. Например объекты в Цепи Ответственности (223) должны иметь общий тип, но обычно они не разделяют общую реализацию. В шаблоне Композит (163) Компонент определяет общий интерфейс, но Композит часто определяет общую реализацию. Команда (233), Наблюдатель (293), Состояние (305) и Стратегия (315) часто реализуются с помощью абстрактных классов и чистых интерфейсов.

Программирование для интерфейса, а не реализации

Наследование классов в основном только механизм для расширения функциональности приложения с помощью повторного использования функциональности родительских классов. Оно позволяет вам быстро определять новый вид объекта в терминах старого. Оно позволяет получать новые реализации практически даром, наследуя большую часть того что вам нужно от существующих классов.

Тем не менее, использование реализации это только половина дела. Также важна способность наследования определять семейства объектов с *идентичными* интерфейсами (обычно наследуя от абстрактного класса). Почему? Потому что от этого зависит полиморфизм.

Когда наследование используется аккуратно (некоторые скажут *правильно*), все классы унаследованные от абстрактного класса разделят его интерфейс. Это подразумевает, что подкласс просто добавляет или переопределяет операции, и не прячет операции родительского класса. *Все* подклассы могут отвечать на запросы в интерфейсе абстрактного класса, что делает их подтипами абстрактного класса.

У работы с объектами исключительно в терминах интерфейса определенного абстрактным классом есть два преимущества:

1. Клиенты не беспокоятся о конкретных типах используемых объектов, пока объекты придерживаются интерфейса ожидаемого клиентами.

2. Клиенты не беспокоятся о классах, которые реализуют эти объекты. Клиенты знают только об абстрактных класс(ах) определяющих интерфейс.

Это так значительно ослабляет зависимости между подсистемами, что приводит к следующему принципу ОО проектирования:

Программируйте для интерфейса, а не реализации.

Не определяйте переменные экземпляры какого-то конкретного класса. Вместо этого используйте только интерфейс определенный абстрактным классом. Вы обнаружите, что это общая тема проектных шаблонов в этой книге.

Конечно вам придется создавать конкретные классы (то есть указывать конкретную реализацию) где-то в вашей системе, и создающие шаблоны (Абстрактная фабрика (87), Строитель (97), Заводской метод (107), Прототип (117) и Одиночка (127) позволяют вам сделать это. Абстрагированием процесса создания объектов эти шаблоны предоставляют вам разные способы прозрачного ассоциирования интерфейса с реализацией на этапе создания. Создающие шаблоны гарантируют, что ваша система написана в терминах интерфейсов, а не реализаций.

1.6.5 Применение механизмов многократного использования

Большинство людей могут понять концепции вроде объектов, интерфейсов, классов и наследования. Трудность заключается в их применении для создания гибких, многократно используемых программ, и шаблоны проектирования могут показать вам как это сделать.

Наследование против Композиции

Две самые распространенные техники для повторного использования функциональности в ОО системах это наследование классов и **композиция объектов**. Как мы объясняли, наследование классов позволяет вам определить реализацию одного класса в терминах другого. Использование созданием подклассов часто называют использованием **белого ящика**. Термин “белый ящик” говорит о видимости: при наследовании внутренности родительских классов часто видны подклассам.

Альтернативой наследованию является композиция объектов. Здесь новая функциональность получается объединением или *компоновкой* объектов. Композиция объектов требует наличия у них хорошо определенных интерфейсов. Этот стиль использования называется использованием **черного ящика**, поскольку внутреннее состояние объектов не видно. Объекты предстают только в виде “черных ящиков”.

Наследование и композиция имеют свои преимущества и недостатки. Наследование классов определено статически во время компиляции и просто в использовании, поскольку поддерживается непосредственно языком программирования. Наследование также облегчает изменение многократно используемой реализации. Когда подкласс переопределяет некоторые, но не все операции, это может также влиять на операции которые он наследует, если они вызывают переопределенные операции.

Но наследование имеет также и недостатки. Во-первых, вы не можете изменять реализации унаследованные от родительского класса во время выполнения, потому что наследование определено во время компиляции. Во-вторых, что обычно хуже, родительские классы часто определяют как минимум часть физического представления их подклассов. Поскольку наследование посвящает подкласс в детали реализации его родительского класса, часто говорят, что "наследование нарушает инкапсуляцию" [Sny86]. Реализация подкласса становится так связана с реализацией родительского класса, что любое изменение в родительском классе приведет также и к изменению подкласса.

Зависимости реализаций могут вызвать проблемы когда вы пытаетесь повторно использовать подкласс. Если какой-либо аспект унаследованной реализации не подходит для новых специфических задач, родительский класс нужно будет переписать, или заменить на что-либо более подходящее. Эта зависимость ограничивает гибкость и многократное использование. Единственное лекарство против этого — наследовать только от абстрактных классов, т.к. в них мало или совсем нет реализации.

Композиция объектов определяется динамически во время выполнения через ссылки объектов на другие объекты. Композиция требует, чтобы объекты соблюдали интерфейсы других объектов, что в свою очередь требует тщательно разработанных интерфейсов, которые бы не препятствовали использованию объекта со многими другими. Но она того стоит. Поскольку работа с объектами происходит исключительно через их интерфейсы мы не нарушаем инкапсуляцию. Любой объект может быть заменен на другой во время выполнения пока он имеет тот же тип. Более того, поскольку реализация объекта будет написана в терминах интерфейсов, зависимостей будет значительно меньше.

Композиция объектов имеет другой эффект на проект системы. Предпочтение композиции объектов перед наследованием помогает вам поддерживать инкапсуляцию каждого класса и фокусировать его на одной задаче. Ваши классы и иерархии классов останутся небольшими и с меньше вероятностью вырастут в огромных неуправляемых монстров. С другой стороны проект основанный на композиции объектов будет иметь больше объектов (т.к. меньше классов), и поведение системы будет зависеть от их взаимоотношений вместо того, чтобы определяться в одном классе.

Это приводит нас ко второму принципу ОО проектирования:

Предпочитайте композицию объектов наследованию классов.

В идеале, вы не должны создавать новых компонентов для достижения многократного использования. Вы должны иметь возможность получить всю необходимую функциональность, просто объединяя существующие компоненты, с помощью объектной композиции. Но такие случаи редки, поскольку набор доступных компонент практически никогда не бывает достаточно богат. Многократное использование путем наследования упрощает создание новых компонент, которые могут быть соединены с уже имеющимися. Таким образом, наследование и композиция объектов работают вместе.

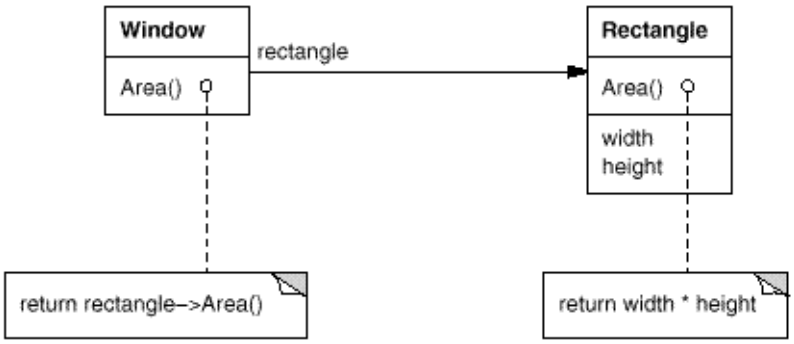
Однако, согласно нашему опыту проектировщики чаще используют наследование в качестве техники повторного использования, тогда как проекты зачастую лучше многократно используются (и становятся проще), если больше полагаться на композицию объектов. Вы увидите, что композиция объектов снова и снова применяется в шаблонах проектирования.

Делегирование

Делегирование — это способ сделать композицию такой же мощной для повторного использования как наследование. При делегировании в обработку запроса вовлечены *два* объекта: получающий объект делегирует операции своему **делегату**. Это похоже на подклассы передающие запросы родительским классам. Но при наследовании, наследуемая операция всегда может сослаться на получающий объект через переменную-член `this` в C++ и `self` в Smalltalk. Для получения аналогичного эффекта при делегировании, получатель передает себя делегату, чтобы позволить делегируемой операции ссылаться на получателя.

Например, вместо создания класса Окно как подкласса Прямоугольника (поскольку окна чаще всего прямоугольны), класс Окно может использовать поведение Прямоугольника включением экземпляра переменной Прямоугольник и *делегированием* ей поведения специфичного для Прямоугольника. Другими словами, Окно вместо того, чтобы *быть* Прямоугольником, будет *содержать* Прямоугольник. Теперь Окно должно явно передавать запросы экземпляру Прямоугольника, тогда как раньше оно бы наследовало эти операции.

Следующая диаграмма описывает класс Окно делегирующий операцию Область экземпляру прямоугольника.



Линия со стрелкой показывает, что класс содержит ссылку на экземпляр другого класса. Ссылка имеет опциональное имя, в данном случае “прямоугольник”

Основное преимущество делегирования в том, что оно облегчает объединение поведения объектов во время выполнения и изменение способа этого объединения. Наше окно может стать круглым во время выполнения просто заменой экземпляра Прямоугольника на экземпляр Окружности, в предположении, что Прямоугольник и Окружность имеют одинаковый тип.

У делегирования есть недостаток, который также присущ другим техникам, делающим ПО более гибким через композицию объектов: Динамичное, сильно параметризованное ПО понять труднее, чем более статичное ПО. Также присутствуют программные издержки во время выполнения, но человеческие издержки более важны в долгосрочной перспективе. Делегирование является хорошим проектным выбором, только когда оно упрощает больше, чем усложняет. Непросто дать правила, которые точно говорят когда использовать делегирование, поскольку его эффективность зависит от контекста и вашего опыта его использования. Делегирование работает лучше всего при использовании хорошо определенными способами — то есть в стандартных шаблонах.

Несколько шаблонов проектирования используют делегирование. Состояние (305), Стратегия (315), Посетитель (331) зависят от него. В шаблоне Состояние объект делегирует запросы объекту Состояние, который представляет его текущее состояние. В шаблоне Стратегия, объект делегирует специфический запрос объекту, который представляет стратегию для выполнения запроса. Объект будет иметь только одно состояние, но он может иметь множество стратегий для различных запросов. Цель обоих шаблонов — изменить поведение объекта, заменяя объекты, которым он делегирует запросы. В Посетителе действие производимое над каждым элементов структуры объекта всегда делегируется объекту Посетитель.

Остальные шаблоны меньше используют делегирование. Посредник (273) представляет объект-посредник между другими взаимодействующими объектами. Иногда объект

Посредник реализует операции, просто передавая их другим объектам; в других случаях он передает ссылку на себя самого и таким образом использует настоящее делегирование. Цепь ответственности (223) обрабатывает запросы, передавая их от одного объекта к другому по цепочке. Иногда этот запрос передается вместе со ссылкой на исходный объект, получающий запрос, в таком случае шаблон использует делегирование. Мост (151) отделяет абстракцию от реализации. Если абстракция и соответствующая реализация тесно согласованы, то абстракция может просто делегировать операции этой реализации.

Делегирование это экстремальный пример композиции объектов. Он показывает, что вы всегда можете заменить наследование на композицию объектов в качестве механизма многократного использования кода.

Наследование против параметризованных типов

Другая (не строго объектно-ориентированная) техника для многократного использования функциональности это **параметризованные типы**, также известные, как **родовые** (generics) (Ada, Eiffel) и **шаблоны** (templates) (C++). Эта техника позволяет вам определять тип без указания всех типов которые он использует. Неуказанные типы передаются в качестве *параметров* в точке использования. Например класс Список может быть параметризован типом содержащихся в нем элементов. Для определения списка целых чисел, вы предоставляете тип “целое” как параметр для параметризованного типа Список. Для определения списка строк вы используете тип “Строка” в качестве параметра. Реализация языка создаст соответствующую версию шаблона класса Список для каждого типа элемента.

Параметризованные типы дают нам третий способ (в добавку к наследованию классов и композиции объектов) для компоновки поведения в ОО системах. Многие проекты могут быть реализованы с использованием любого из этих способов. Для параметризования процедуры сортировки операцией которая сравнивает элементы мы можем сделать сравнение:

1. операцией реализованной подклассами (используем Шаблонный Метод) (325)
2. ответственностью объекта передаваемого в процедуру сортировки (Стратегия (315) или
3. аргументом шаблона C++ или родового типа Ada, который указывает имя функции вызываемой для сравнения элементов.

Существуют важные различия между этими техниками. Композиция объектов позволяет вам изменять создаваемое поведение во время выполнения, но также требует косвенности и может быть менее эффективна. Наследование позволяет вам предоставить реализации операций по умолчанию и позволяет классам переопределять их. Параметризованные типы позволяют вам изменять типы используемые классом. Но ни наследование, ни параметризованные типы не могут изменяться во время выполнения. Какой подход лучше зависит от ограничений проекта и реализации.

Ни один из шаблонов в этой книге не использует параметризованные типы, хотя мы используем их при случае для указания реализации шаблона на C++. Параметризованные типы не нужны в языках вроде Smalltalk, которые не проводят проверку типов во время компиляции.

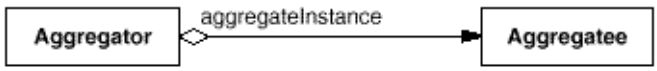
1.6.6 Соотношение структур времени компиляции и времени выполнения

Структура объектно-ориентированной программы во время выполнения зачастую несет слабый отпечаток структуры ее кода. Структура кода заморожена во время выполнения; она состоит из классов жестко связанных отношениями наследования. Структура программы во время выполнения состоит из быстро изменяющихся сетей взаимодействующих объектов. Действительно эти две структуры практически независимы. Попытка понять одну по другой подобна попытке понять динамику живых экосистем по статической анатомии растений и животных и наоборот.

Рассмотрим различие между **агрегацией** и **взаимодействием** объектов и как они различно обнаруживают себя во время компиляции и во время выполнения. Агрегация подразумевает, что один объект владеет или является ответственным за другой объект. В общем мы говорим объекте являющемся *частью* другого объекта. Агрегация подразумевает, что агрегируемый объект и его владелец имеют одинаковое время существования.

Взаимодействие подразумевает что объект только *знает* о другом объекте. Иногда взаимодействие называется отношением “ассоциирования” или “использования”. Взаимодействующие объекты могут запрашивать операции друг друга, но они не отвечают друг за друга. Отношение взаимодействия слабее агрегации и предполагает гораздо меньшую связь между объектами.

В наших диаграммах простая линия со стрелкой означает взаимодействие. Стрелка с ромбиком у основания означает агрегацию:



Легко перепутать **агрегацию** и **взаимодействие**, поскольку они часто реализуются одинаково. В Smalltalk все переменные являются ссылками на другие объекты. В языке нет различия между агрегацией и взаимодействием. В C++ агрегация может быть реализована определение переменных-членов, которые являются непосредственными экземплярами, но более общепринятая практика определять их как ссылки или указатели на экземпляры. Взаимодействие также реализуется с помощью указателей и ссылок.

В конце концов взаимодействие и агрегация определяются больше намерениями, чем явными языковыми механизмами. Различие может быть трудно увидеть в структуре времени компиляции, но оно значительно. Отношений агрегации обычно меньше и они более постоянны. Наоборот, взаимодействия создаются и переделываются более часто, иногда существуя только во время выполнения операции. Также взаимодействия более динамичны, что усложняет их выявление в исходном коде.

Учитывая такое несоответствие между программными структурами времени компиляции и времени выполнения, ясно что код не расскажет всего о том как будет работать система. Структура системы во время выполнения должна больше определяться проектировщиком, чем языком. Отношения между объектами и их типами должны разрабатываться очень осторожно, потому что они определяют насколько хороша или плоха структура времени выполнения.

Многие шаблоны проектирования (в особенности применяемые к объектам) явно фиксируют различие между структурами времени выполнения и времени компиляции. Композит (163) и Декоратор (175) особенно полезны для построения сложных структур времени выполнения. Наблюдатель (293) включает структуры времени выполнения, которые зачастую трудно понять, если вы не знаете шаблона. Цепь ответственности (223) также приводит к взаимодействующим шаблонам, которые не выявляются наследованием. В общем структуры времени выполнения непонятны из кода, пока вы не поймете шаблоны.

1.6.7 Разработка для изменения

Ключ к максимизации повторного использования лежит в предвидении новых требований и изменений к существующим требованиям, и в разработке ваших систем так, чтобы они могли развиваться соответственно изменениям.

Для разработки системы устойчивой к подобным изменениям вы должны предположить, как система может измениться в течении своего существования. Проект не прини-

мающий в расчет изменения рискует глобально перерабатываться в будущем. Эти изменения могут включать переопределение классов и их реализаций, изменение клиентов и новое тестирование. Переработка влияет на многие части программной системы и непредвиденные изменения неизменно дороги.

Проектные шаблоны помогают вам избежать этого, гарантируя, что система может изменяться в определенных направлениях. Каждый проектный шаблон позволяет какому либо аспекту структуры системы изменяться независимо от других аспектов, тем самым делая систему более устойчивой к определенным видам изменений.

Здесь приведены некоторые наиболее общие причины перепроектирования вместе с проектными шаблонами для таких случаев:

1. *Явное указание класса при создании объекта.* Указание имени класса при создании объекта привязывает вас к определенной реализации вместо определенного интерфейса. Эта привязка может усложнить будущие изменения. Во избежание этого создавайте объекты косвенно.

   Проектные шаблоны: Абстрактная Фабрика (87), Заводской Метод (107), Прототип (117).

2. *Зависимость от определенных операций.* Когда вы указываете конкретную операцию, вы привязываетесь к одному способу выполнения запроса. Избегая жестко заданных запросов, вы облегчаете изменение способа выполнения запроса как в во время компиляции, так и во время выполнения.

   Проектные шаблоны: Цепь ответственности (223), Команда (233).

3. *Зависимость от программной или аппаратной платформы.* Внешние интерфейсы операционной системы или интерфейсы программирования приложений (APIs) различны на разных программных и аппаратных платформах. ПО зависящее от конкретной платформы будет труднее переносить на другие платформы. Возможно его будет сложно обновлять на его родной платформе. Поэтому важно так проектировать вашу систему, чтобы ограничить ее зависимости от конкретной платформы.

   Проектные шаблоны: Абстрактная Фабрика (87), Мост (151).

4. *Зависимость от представлений и реализаций объекта.* Клиенты, которые знают как объект представляется, где хранится и находится или как реализуется могут потребовать изменения при изменении объекта. Сокрытие этой информации от клиентов удерживает изменения от каскадного увеличения.

Проектные шаблоны: Абстрактная Фабрика (87), Мост (151), Память (283), Полномочия (207).

5. *Алгоритмические зависимости.* Алгоритмы часто расширяются, оптимизируются и заменяются во время разработки и использования. Объекты, зависящие от алгоритма должны будут измениться при его изменении. Поэтому алгоритмы наиболее подверженные изменению должны быть изолированы.

   Проектные шаблоны: Строитель (97), Итератор (257), Стратегия (315), Шаблонный метод (325), Посетитель (331).

6. *Тесное сцепление.* Тесно сцепленные классы трудно использовать изолированно, поскольку они зависят друг от друга. Тесное сцепление ведет к монолитным системам, в которых вы не можете изменить или убрать класс без понимания и изменения многих других классов. Система становится тяжелой массой, которую трудно изучать, переносить и поддерживать. Ослабление сцепления увеличивает вероятность того, что класс может быть многократно использован самостоятельно, и что система сможет быть изучена, перенесена и модифицирована гораздо проще. Проектные шаблоны используют такие техники как абстрактное сцепление и наслоение для получение слабо сцепленных систем.

   Проектные шаблоны: Абстрактная фабрика (87), Мост (151), Цепь ответственности (223), Команда (233), Фасад (185), Посредник (273), Наблюдатель (293).

7. *Расширение функциональности сабклассингом.* Настройка объекта сабклассингом часто непроста. Каждый новый класс имеет фиксированные издержки в реализации (инициализация, завершение, и т.д.). Определение подкласса также требует глубокого понимания его родительского класса. Например переопределение одной операции может потребовать переопределения другой. Переопределенная операция возможно должна будет вызвать унаследованную операцию. Также сабклассинг может привести к взрывному увеличению числа классов, поскольку вам придется добавлять много новых подклассов даже для простого расширения.

   Композиция объектов вообще и делегирование в частности предоставляют гибкие альтернативы наследованию для комбинирования поведения. Новая функциональность может быть добавлена в приложение соединением существующих объектов новыми способами вместо определения новых подклассов существующих классов. С другой стороны большое использование композиции объектов может усложнить понимание проектов. Многие шаблоны проектирования производят проекты в кото-

рых вы можете добавить функциональность просто определением одного подкласса и соединением его экземпляров с уже существующими классами.

Проектные шаблоны: Мост (151), Цепь ответственности (223), Композит(163), Декоратор (175), Наблюдатель (293), Стратегия (315).

8. *Невозможность простого изменения классов.* Иногда вам нужно изменить класс, который не может быть модифицирован просто. Возможно вам нужен исходный код, а у вас его нет (как например в случае коммерческой библиотеки классов). Или любое изменение потребует множества модификаций существующих подклассов. Проектные шаблоны предлагают способы изменения классов в таких условиях.

   Проектные шаблоны: Адаптер (139), Декоратор (175), Посетитель (331).

Эти примеры показывают гибкость, которую проектные шаблоны помогают встроить в ваше ПО. Насколько критична такая гибкость зависит от типа ПО которое вы создаете. Давайте посмотрим на роль проектных шаблонов в разработке трех широких классов ПО: прикладных программ, пакетов разработчика и сред разработки.

Прикладные программы

Если вы создаете прикладную программу, такую как редактор документов или электронную таблицу, то внутреннее многократное использование, сопровождаемость, и расширяемость имеют высокий приоритет. Внутреннее многократное использование гарантирует, что вы не проектируете и не реализовываете больше чем вы должны. Проектные шаблоны, которые уменьшают зависимости, могут увеличить внутреннее многократное использование. Ослабление сцепления увеличивает вероятность того, что один класс или объект сможет сотрудничать с несколькими другими. Например, когда вы устраните зависимости определенных операций изолировав и инкапсулировав каждую операцию, вы, тем самым, упростите многократное использование операции в разных контекстах. То же самое может случиться если вы также удалите зависимости алгоритмов и представлений.

Проектные шаблоны также делают приложение более сопровождаемым, когда они используются для ограничения зависимостей от платформы и расслоения системы. Они улучшают расширяемость, показывая как расширить иерархии классов и как использовать объектную композицию. Уменьшенное сцепление также улучшает расширяемость. Расширение класса изолированно проще, если класс не зависит от многих других классов.

Пакеты разработчика

Часто приложение включает классы из одной или более библиотек классов, называемых пакетами разработчика. Пакет разработчика это набор взаимосвязанных классов, разработанных с целью предоставления полезной, целевой функциональности. Примером пакета разработчика может быть набор классов для списков, ассоциативных таблиц, стеков и тому подобного. Другой пример — потоковая библиотека ввода/вывода в C++. Пакеты не навязывают определенную структуру проекта вашего приложения; они просто предоставляют функции, которые могут помочь вашему приложению делать свою работу. Они позволяют вам, как программисту избежать повторного кодирования общей функциональности. Пакеты разработчика делают ударение на *многократном использовании кода*. Они являются ОО эквивалентом библиотек подпрограмм.

Разработка пакетов разработчика бесспорно труднее, чем разработка прикладных программ, поскольку для того, чтобы быть полезными, пакеты разработчика должны работать во многих приложениях. Более того, разработчик пакета не знает что это будут за приложения, и каковы будут их специфические требования. Это делает наиболее важным избежание предположений и зависимостей, которые могут ограничить гибкость пакета и, следовательно, его применимость и эффективность.

Среды разработки

Среда разработки это набор взаимодействующих классов, которые составляют многократно используемый проект для специфического типа программного обеспечения [Deu89, JF88]. Например, среда разработки может быть сделана для построения графических редакторов для различных областей применения таких, как художественное рисование, сочинение музыки и механические САПР [VL90, Joh92]. Другая среда разработки может помочь вам создавать компиляторы для разных языков программирования и целевых платформ [JML92]. Еще одна помогает в создании приложения для финансового моделирования [BE93]. Вы настраиваете среду разработки для определенного приложения созданием специфических для него подклассов существующих абстрактных классов.

Среда разработки диктует архитектуру вашего приложения. Она определит общую структуру, ее разделение на классы и объекты и, следовательно, ключевые ответственности, взаимодействие классов и объектов, поток управления. Среда разработки предопределяет эти проектные параметры, так чтобы вы, проектировщик/программист приложения, могли сконцентрироваться на специфике вашего приложения. Среда разработки фиксирует общие проектные решения для своей предметной области. Таким образом, среды разработки подчеркивают преобладание *многократного использования проектных ре-*

шений над многократным использованием кода, хотя среда разработки обычно включает конкретные классы, которые вы сразу же можете использовать.

Многократное использование на этом уровне ведет к инверсии управления между приложением и программным обеспечением на котором оно основано. Когда вы используете пакет разработчика (или стандартную библиотеку подпрограмм), вы пишите основное тело приложения и вызываете код, который хотите использовать. Когда вы используете среду разработки, вы многократно используете основное тело и пишите код, который *оно* вызывает. Вам придется писать операции с определенными именами и условиями вызова, но это уменьшает количество проектных решений, которые вам нужно принять.

В результате вы не только можете строить приложения быстрее, но также эти приложения имеют похожую структуру. Их проще сопровождать и они более единообразны для пользователей. С другой стороны, вы теряете некоторую свободу творчества, т.к. многие проектные решения были сделаны за вас.

Если приложения разрабатывать трудно, а пакеты разработчика труднее, то среды разработки труднее всего. Проектировщик среды разработки ставит на то, что одна архитектура подойдет для всех приложений в предметной области. Любое существенное изменение в проекте среды разработки значительно уменьшит его преимущества, т.к. основной вклад среды разработки в приложение — это архитектура, которую она определяет. Следовательно, необходимо проектировать среду разработки так, чтобы она была гибкой и расширяемой насколько возможно.

Кроме того, поскольку приложения так зависят от среды разработки, в проекте, они особенно чувствительны к изменениям в ее интерфейсах. В ходе развития среды разработки приложения вынуждены развиваться вместе с ней. Это увеличивает важность слабого сцепления; в противном случае даже небольшое изменение в среде разработки будет иметь серьезные последствия.

Обсужденные выше вопросы проектирования наиболее критичны для проектирования среды разработки. Среда разработки, которая решает их, используя шаблоны проектирования, с большей вероятностью достигнет высокого уровня проектирования и многократного использования кода, чем та, которая не использует шаблоны. Зрелые среды разработки обычно включают несколько проектных шаблонов. Шаблоны помогают сделать архитектуру среды разработки подходящей для многих приложений без перепроектирования.

Дополнительная выгода получается, когда среда разработки документирована с помощью используемых ей проектных шаблонов [BJ94]. Люди знающие шаблоны быстрее осваивают среды разработки. Даже люди, которые не знают шаблонов, могут выиграть от структуры которую шаблоны привносят в документацию среды разработки. Улучшение

документации важно для всех видов ПО, но особенно важно для сред разработки. Среды разработки часто имеют крутую кривую обучения, которая должна быть преодолена прежде чем они станут полезны. Хотя проектные шаблоны не могут полностью сгладить кривую обучения, они могут сделать ее менее крутой за счет более явного представления ключевых элементов среды разработки.

Поскольку шаблоны и среды разработки имеют некоторые общие черты, люди часто интересуются чем они отличаются, и отличаются ли вообще. Они различаются в трех основных вопросах:

1. *Шаблоны проектирования более абстрактны, чем среды разработки.* Среды разработки могут быть воплощены в коде, тогда как только *примеры* шаблонов могут быть воплощены в коде. Сила сред разработки в том, что они могут быть написаны на языках программирования, и не только изучены, но и напрямую выполнены и многократно использованы. В противоположность этому шаблоны проектирования из этой книги должны каждый раз реализовываться, когда необходимо их использовать. Шаблоны проектирования также объясняют намерения, выгоды и последствия проекта.

2. *Шаблоны проектирования являются меньшими архитектурными элементами чем среды разработки.* Типичная среда разработки содержит несколько проектных шаблонов, обратное же никогда не верно.

3. *Проектные шаблоны менее специализированы чем среды разработки.* Среды разработки всегда имеют определенную область применения. Среда разработки графического редактора может быть использована в моделировании фабрики, но ее не спутаешь со средой разработки для моделирования. Наоборот, проектные шаблоны в этом каталоге могут быть использованы почти в любых видах приложений. В то же время вполне возможны шаблоны более специализированные чем наши (скажем шаблоны проектирования для распределенных систем или параллельного программирования), даже они не будут диктовать архитектуру приложения так как среда разработки.

Среды разработки становятся все более важны и широкоупотребимы. Они являются способом с помощью которого ОО системы достигают наибольшей степени многократного исполльзования. БОльшие ОО приложения заканчивают как набор слоев взаимодействующих сред разработки. Большая часть проекта и кода в приложении прийдет из используемых сред разработки, или подвергнется их влиянию.

1.7 Как выбрать проектный шаблон.

При наличии более 20 шаблонов в каталоге, поиск того, который предназначен для конкретной проектной задачи может быть затруднен, особенно если каталог незнаком вам. Далее приведены несколько разных подходов к поиску проектного шаблона подходящего к вашей задаче:

1. *Посмотрите как проектные шаблоны решают проектные задачи.* В разделе 1.6 обсуждается как проектные шаблоны помогают вам в поиске подходящих объектов, определяют размер объектов, определяют интерфейсы объектов и несколько других путей в которых шаблоны проектирования решают проектные задачи. Ссылки на эти обсуждения могут направить ваш поиск подходящего шаблона.

2. *Просмотрите параграфы Намерение.* В разделе 1.4 (стр. 14) перечисляются параграфы Намерение всех шаблонов в каталоге. Прочтите намерения каждого шаблона для того чтобы найти одно или несколько намерений звучащих подходяще к вашей задаче. Вы можете использовать схему классификации представленную в Таблице 1.1 (страница 17) для сужения области поиска.

3. *Изучите как шаблоны соотносятся.* Рисунок 1.1 (страница 19) графически показывает взаимосвязи между шаблонами. Изучение этих взаимосвязей может направить вас на нужный шаблон или группу шаблонов.

4. *Изучите шаблоны схожего назначения.* Каталог (страница 79) содержит 3 главы, одна для создающих шаблонов, другая для структурных шаблонов и третья для поведенческих шаблонов. Каждая глава начинается со вступительных комментариев о шаблонах и завершается параграфом, который сравнивает и противопоставляет их. Эти параграфы дают вам понимание сходств и различий между шаблонами имеющими схожие цели.

5. *Проверьте причину перепроектирования.* Просмотрите причины перепроектирования на странице 34 возможно ваша задача включают одну или несколько из них. Затем посмотрите шаблоны помогающие избегать причин перепроектирования.

6. *Определите что в вашем проекте должно быть меняющимся.* Этот подход противоположен фокусировке на причинах перепроектирования. Вместо рассмотрения того что может *потребовать* изменений в проекте, рассмотрите что бы вы хотели видеть изменяющимся без перепроектирования. Здесь упор делается на *инкапсулировании*

меняющейся концепции, это тема многих проектных шаблонов. Таблица 1.2 перечисляет аспекты проекта которые изменяются с помощью шаблонов проектирования, тем самым позволяя изменять их без перепроектирования.

Назначение	**Проектный шаблон**	**Аспект(ы), которые могут изменяться**
Создающие	Абстрактная фабрика (87)	семейства результирующих объектов
	Строитель (97)	как составные объекты создаются
	Заводской метод (107)	подкласс создаваемого объекта
	Прототип (117)	класс создаваемого объекта
	Одиночка (127)	единственный экземпляр класса
Структурные	Адаптер (139)	интерфейс объекта
	Мост (151)	реализация объекта
	Композит (163)	структура и композиция объекта
	Декоратор (175)	ответственности объекта без сабклассинга
	Фасад (185)	интерфейс подсистемы
	Легковес (195)	издержки хранения объектов
	Полномочия (207)	осуществляение доступа к объекту; его положение
Поведенческие	Цепь ответственности (223)	объект, который может выполнить запрос
	Команда (233)	когда и как запрос был выполнен
	Интерпретатор (243)	грамматика и интерпретация языка
	Итератор (257)	доступ к элементам агрегата
	Посредник (273)	как и какие объекты взаимодействуют друг с другом
	Память (283)	какая закрытая информация хранится вне объекта и когда
	Наблюдатель (293)	число объектов зависящих от другого объекта; как зависимые объекты поддерживаются в актуальном состоянии
	Состояние (305)	состояния объекта
	Стратегия (315)	алгоритм
	Шаблонный метод (325)	шаги алгоритма
	Посетитель (331)	операции, которые могут быть произведены над объектом (объектами) без изменения его класса (классов)

Таблица 1.2: Аспекты проекта, которые проектные шаблоны позволяют вам изменять

1.8 Как использовать проектный шаблон

Как только вы выбрали проектный шаблон, как вам его использовать? Ниже приведена пошаговая инструкция по эффективному применению проектного шаблона:

1. *Прочтите обзор шаблона.* Особое внимание уделите параграфам Применимость и Последствия, чтобы удостовериться, что шаблон подходит для вашей задачи.

2. *Вернитесь и изучите параграфы Структура, Участники и Взаимодействия.* Удостоверьтесь, что вы понимаете классы и объекты в шаблоне и как они соотносятся.

3. *Посмотрите параграф Пример Кода для конкретного примера кодирования шаблона.* Изучение этого кода помогает вам понять как реализовать шаблон.

4. *Выберите имена для участников шаблона, значимые в контексте приложения.* Имена участников в проектных шаблонах обычно слишком абстрактны для непосредственного применения в приложении. Тем не менее полезно вставить имя участника в то имя, которое появится в приложении. Это поможет сделать шаблон более явным в реализации. Например если вы используете шаблон Стратегия для алгоритма компоновки текста, то вы можете иметь классы ПростаяСтратегияРасположения ТеХСтратегияРасположения.

5. *Определите классы.* Определите их интерфейсы, установите отношения наследования и определите переменные экземпляры, которые представляют ссылки на данные и объекты. Выделите существующие классы, на которые повлияет шаблон и измените их соответственно.

6. *Определите специфические для приложения имена операций в шаблоне.* Здесь снова имена в основном зависят от приложения. Используйте ответственность и взаимодействия каждой операции в качестве руководства. Также будьте последовательны в правилах именования. Например вы можете последовательно использовать префикс “Создать-” для указания на заводской метод.

7. *Реализуйте операции для выполнения обязанностей и взаимодействий в шаблоне.* Параграф Реализация предлагает подсказки для этого. Также могут помочь примеры из параграфа Пример Кода.

Это только основные принципы для того чтобы вы могли начать. Со временем вы разработаете свой собственный метод работы с проектными шаблонами.

Никакое обсуждение того как использовать шаблоны не может быть завершено без нескольких слов о том как их *не* использовать. Проектные шаблоны не должны применяться неразборчиво. Часто гибкость и варьируемость достигаются добавлением дополнительных уровней косвенности, и это может усложнить проект и/или уменьшить производительность. Проектный шаблон должен применяться только тогда, когда предлагаемая им гибкость действительно нужна. Параграф Последствия наиболее полезен при оценке достоинств и недостатков шаблона.