Общие вопросы проектирования

Прежде всего, опытный разработчик понимает, что не нужно решать каждую новую задачу с нуля. Вместо этого он старается повторно воспользоваться теми решениями, которые оказались удачными в прошлом.

По словам Кристофера Александера (Christopher Alexander), «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, и при этом никакие две реализации не будут полностью одинаковыми»

В общем случае паттерн состоит из четырех основных элементов:

- **Имя**. Указывая имя, мы сразу описываем проблему проектирования, ее решения и их последствия — и все это в одном-двух словах
- **Задача**. Описание того, когда следует применять паттерн. Описание объясняет задачу и ее контекст.
- **Решение**. Описание элементов дизайна, отношений между ними, их обязанностей и взаимодействий между ними. В решении не описывается конкретный дизайн или реализация, поскольку паттерн — это шаблон, применимый в самых разных ситуациях.
- **Результаты**. Следствия применения паттерна, его вероятные плюсыи минусы.

Паттерны проектирования — это не то же самое, что связанные списки или хештаблицы, которые можно реализовать в виде класса и повторно использовать без каких бы то ни было модификаций. С другой стороны, это и не сложные предметно-ориентированные решения для целого приложения или подсистемы. В этой книге под паттернами проектирования понимается описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте.

MVC состоит из объектов трех видов. **Модель** — это объект приложения, а **представление** — его внешний вид на экране. **Контроллер** описывает, как интерфейс реагирует на управляющие воздействия пользователя.

Чтобы повторно воспользоваться дизайном, нам необходимо документировать решения, альтернативные варианты и компромиссы, которые привели к нему. Важны также конкретные примеры, поскольку они демонстрируют практическое применение паттерна.

Паттерны делятся на порождающие, структурные 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.

Второй критерий — уровень — сообщает, 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.

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, совместными усилиями выполняют задачу, которая ни одному отдельному объекту не под силу.

Самая трудная задача в объектно-ориентированном проектировании — **разложить систему на объекты**.

Методологии объектно-ориентированного проектирования отражают разные подходы. Можно сформулировать задачу письменно, выделить из получившейся 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 в свой дизайн. Разработчики никогда не придут к единому мнению относительно того, какой подход самый лучший.

Абстракции, возникающие в ходе проектирования, — **ключ к гибкому дизайну**.

Эти объекты редко возникают во время анализа и даже на ранних стадиях проектирования. Они появляются позднее, при попытках сделать дизайн более гибким и пригодным для повторного использования.

Для любой операции, объявляемой объектом, 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.

Ассоциирование 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.

Примесью (mixin class) называется класс, назначение которого — предоставить дополнительный интерфейс или функциональность другим классам. Он отчасти похож на абстрактные классы в том смысле, что не предполагает непосредственного создания экземпляров.

Класс объекта определяет реализацию объекта, то есть внутреннее состояние и реализацию операций объекта. Напротив, **тип относится только к интерфейсу объекта** — множеству запросов, на которые объект способен ответить. **У объекта может быть много типов**, и объекты разных классов могут иметь один и тот же тип.

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.

Программируйте в соответствии с интерфейсом, а не с реализацией.

Не объявляйте переменные как экземпляры конкретных классов. Вместо этого придерживайтесь интерфейса, определенного абстрактным классом. Этот принцип проходит через все паттерны, описанные в книге.

Использование порождающих паттернов гарантирует, что система написана в категориях интерфейсов, а не реализации.

Повторное использование за счет порождения подкласса называют еще повторным использованием по принципу прозрачного ящика (*white box reuse*). Такой термин подчеркивает, что внутреннее устройство родительских классов часто видимо подклассам.

Композиция объектов — альтернатива наследованию класса. В этом случае новая, более сложная функциональность получается путем объединения или композиции объектов. Для композиции требуется, чтобы объединяемые объекты имели четко определенные интерфейсы. Такой способ называют повторным использованием по принципу черного ящика (*blackbox reuse*), поскольку детали внутреннего устройства объектов остаются скрытыми.

При делегировании в процессе обработки запроса задействованы два объекта: получатель поручает выполнение операций другому объекту — уполномоченному (делегату).

У делегирования есть и недостаток, свойственный и другим подходам, применяемым для повышения гибкости за счет композиции объектов. Динамическую программу с высокой степенью параметризации труднее понять, нежели статическую. Также присутствует и некоторая потеря машинной эффективности, но в долгосрочной перспективе неэффективность работы проектировщика гораздо более существенна. Делегирование можно считать хорошим вариантом только тогда, когда оно позволяет достичь упрощения, а не усложнения дизайна. Нелегко сформулировать правила, которые бы однозначно определяли, 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.

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.

Различие между осведомленностью и агрегированием в конечном итоге определяется, скорее, предполагаемым использованием, а не языковыми механизмами

Системы должны проектироваться 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.

Вот некоторые типичные причины перепроектирования, а также паттерны, которые позволяют этого избежать:

- **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;

- **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;

- **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. Возможно, даже на «родной» платформе такую программу трудно поддерживать. Поэтому при проектировании систем так важно ограничивать платформенные зависимости. Паттерны проектирования: абстрактная фабрика, мост;

- **зависимость от представления или реализации объекта**. Если клиент располагает информацией о том, как объект представлен, хранится или реализован, то, возможно, при изменении объекта придется изменять и клиента. Сокрытие этой информации от клиентов поможет уберечься от каскадных изменений. Паттерны проектирования: абстрактная фабрика, мост, хранитель, заместитель;

- **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, которые с большой вероятностью будут изменяться, следует изолировать. Паттерны проектирования: мост, итератор , стратегия, шаблонный метод, посетитель;

- **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 слабосвязанных 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;

- **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 так же ясно представлять себе устройство родительского класса. Например, замещение одной операции может потребовать замещения и других. Замещение операции может оказаться необходимым для того, чтобы можно было вызвать унаследованную операцию. Кроме того, порождение подклассов ведет к разрастанию количества классов, поскольку даже для реализации простого расширения приходится создавать новые подклассы. 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, интенсивное использование композиции объектов может усложнить понимание кода. С помощью многих паттернов проектирования удается построить такое 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;

- **неудобства при изменении классов**. Иногда нужно модифицировать класс, но делать это неудобно. Допустим, вам нужен исходный код, а он недоступен (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.

Каркас — это набор взаимодействующих классов, составляющих повторно используемый дизайн для конкретного класса программ

Каркас диктует определенную архитектуру приложения. Он определяет общую структуру, ее разделение на классы и объекты, ключевые обязанности тех и других, методы взаимодействия объектов и классов и потоки управления. Данные параметры проектирования задаются каркасом, а проектировщики или разработчики приложений могут сконцентрироваться на специфике приложения. В каркасе отражены 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.

При использовании инструментальной библиотеки (или, если хотите, обычной библиотеки подпрограмм) вы пишете основной код приложения и вызываете из него код, который планируете использовать повторно.

При работе с каркасом вы, наоборот, повторно используете основной код и пишете код, который он вызывает.

Для освоения работы с каркасами надо потратить немало усилий, и только после этого они начнут приносить реальную пользу. Паттерны могут существенно упростить задачу, явно выделяя ключевые элементы дизайна каркаса.

Способ организации информации играет решающую роль при дальнейшем проектировании.

Общая цель всякого проектирования — свести к минимуму зависимость подсистем друг от друга и обмен информацией между ними.

Инкапсуляция вариаций — элемент многих паттернов поведения

О некоторых паттернах

Паттерн компоновщик инкапсулирует сущность рекурсивной композиции в объектно-ориентированных категориях.

Важно понимать, что интерфейс класса Window призван обслуживать ин-тересы прикладного программиста, тогда как интерфейс класса WindowImp в большей степени ориентирован на оконные системы

Порождающие паттерны проектирования абстрагируют процесс создания экземпляров. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов. Паттерн, порождающий классы, использует наследование, чтобы варьировать класс создаваемого экземпляра, а паттерн, порождающий объекты, делегирует создание экземпляров другому объекту.

Для порождающих паттернов характерны два аспекта. Во-первых, эти паттерны инкапсулируют знания о конкретных классах, которые применяются в системе. Во-вторых, они скрывают подробности создания и компоновки экземпляров этих классов. Единственная информация об объектах, известная системе, — это их интерфейсы, определенные с помощью абстрактных классов. Следовательно, порождающие паттерны обеспечивают большую гибкость в отношении того, что создается, кто это создает, как и когда

Отметим, что MazeFactory — всего лишь набор фабричных методов. Это самый распространенный способ реализации паттерна абстрактная фабрика.

Еще заметим, что MazeFactory — не абстрактный класс, то есть он работает и как AbstractFactory, и как ConcreteFactory. Это еще одна типичная реализация для простых применений паттерна абстрактная фабрика. Поскольку MazeFactory — конкретный класс, состоящий только из фабричных методов, легко получить новую фабрику MazeFactory, породив подкласс и заместив в нем необходимые операции.

В библиотеке InterViews для обозначения классов абстрактных фабрик используется суффикс «**Kit**».

В отличие от порождающих паттернов, которые сразу конструируют весь объект целиком, **строитель делает это шаг за шагом под управлением распорядителя**. И лишь когда продукт завершен, распорядитель забирает его у строителя. Поэтому интерфейс строителя в большей степени отражает процесс конструирования продукта, нежели другие порождающие паттерны. Это позволяет обеспечить более тонкий контроль над процессом конструирования, а значит, и над внутренней структурой готового продукта.

Обратите внимание: MazeBuilder не создает лабиринты самостоятельно, его основная цель — просто определить интерфейс для создания лабиринтов. Пустые реализации в этом интерфейсе определены только для удобства. Реальную работу выполняют подклассы MazeBuilder.

Принцип адаптера класса состоит в наследовании интерфейса по одной ветви и реализации — по другой

Декоратор разрешает добавлять новые обязанности по мере необходимости.

Поскольку паттерн декоратор изменяет лишь внешний облик компонента, последнему ничего не надо «знать» о своих декораторах, то есть декораторы прозрачны для компонента.

О применимости шаблонного метода: Это хороший пример техники «вынесения за скобки с целью обобщения», описанной в работе Уильяма Опдайка (William Opdyke) и Ральфа Джонсона (Ralph Johnson). Сначала выявляются различия в существующем коде, которые затем выносятся в отдельные операции. В конечном итоге различающиеся фрагменты кода заменяются шаблонным методом, из которого вызываются новые операции;

По своей сути паттерн посетитель добавляет в классы новые операции без их изменения. Это делается с помощью приема, называемого **двойной диспетчеризацией**. Данная техника хорошо известна. Некоторые языки программирования (например, CLOS) поддерживают ее явно. Языки же вроде C++ и Smalltalk поддерживают только одинарную диспетчеризацию.

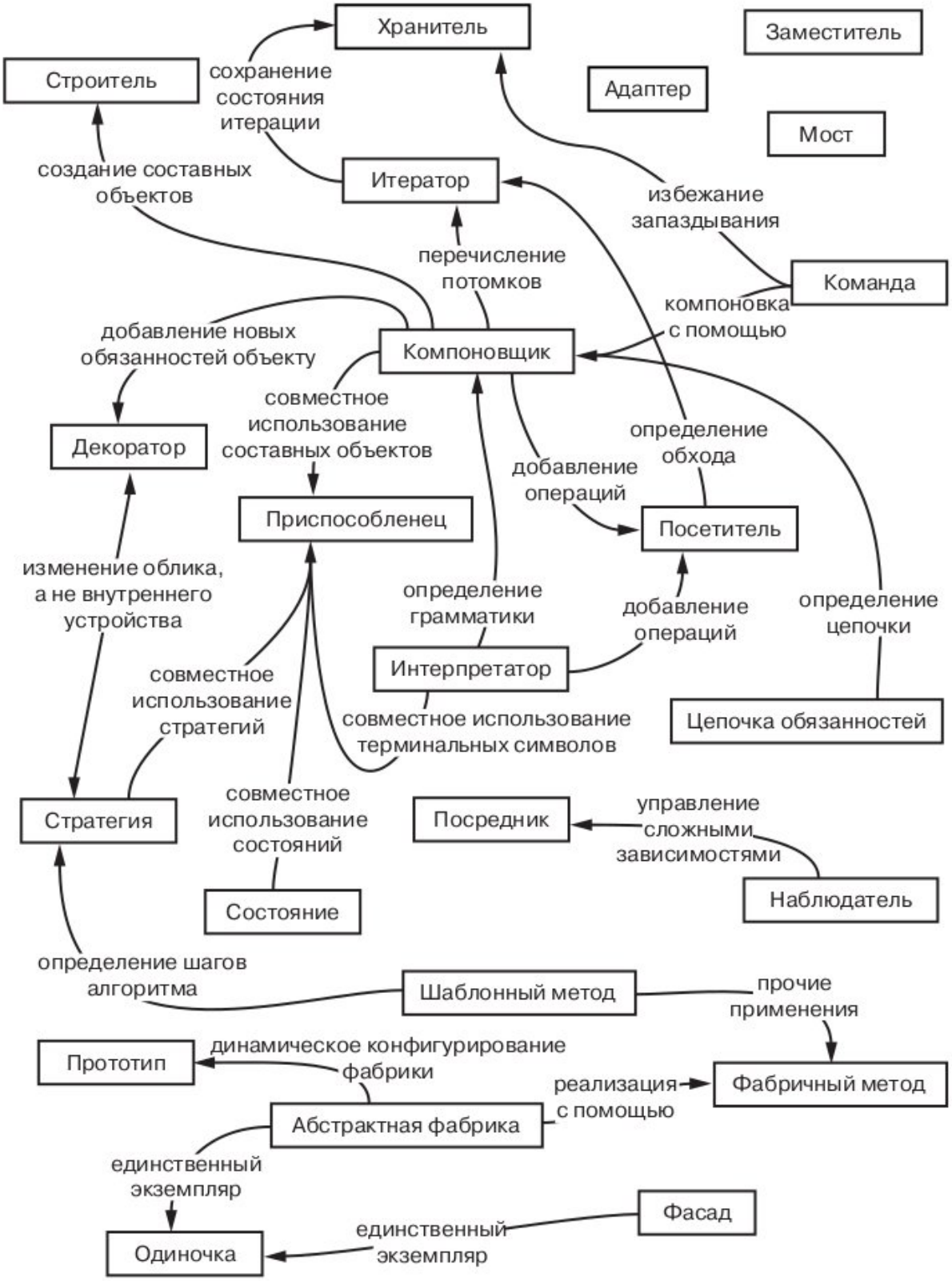

Рис. 1.1. Отношения между паттернами проектирования