

1 Введение

Шаблоны проектирования — это один из важнейших компонентов объектно-ориентированной технологии разработки программного обеспечения. Они широко применяются в инструментах анализа, подробно описываются в книгах и часто обсуждаются на семинарах по объектно-ориентированному проектированию.

1.1 Предыстория

Много лет назад архитектор по имени Кристофер Александер задумался над вопросом: «Является ли качество объективной категорией?». Следует ли считать представление о красоте сугубо индивидуальным, или люди могут прийти к общему соглашению, согласно которому некоторые вещи будут считаться красивыми, а другие нет?

Александер размышлял о красоте с точки зрения архитектуры. Его интересовало, по каким показателям мы оцениваем архитектурные проекты. Например, если некто вознамерился спроектировать крыльцо дома, то как он может получить гарантии, что созданный им проект будет хорош? Можем ли мы знать заранее, что проект будет действительно хорош? Имеются ли объективные основания для вынесения такого суждения?

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

К похожим выводам пришли и исследователи в области культурологии. В пределах одной культуры большинство индивидуумов имеют схожие представления о том, что является сделанным хорошо и что является красивым. В основе их суждений есть нечто более общее, чем сугубо индивидуальные представления о красоте.

Исходная посылка создания шаблонов проектирования также состояла в необходимости объективной оценки качества программного обеспечения. Александер сформулировал для себя следующие вопросы:

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

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

Александер изучал эту проблему, обследуя множество зданий, городов, улиц и всего прочего, что люди построили для своего проживания. В результате он обнаружил, что все, что было построено хорошо, имело между собой нечто общее. Архитектурные структуры отличаются друг от друга, даже если они относятся к одному и тому же типу. Однако, не взирая на имеющиеся различия, они могут оставаться высококачественными.

1.2 От архитектурных шаблонов к шаблонам проектирования программного обеспечения

Но какое отношение может иметь весь этот архитектурный материал к специалистам в области программного обеспечения, коими мы являемся? В начале 1990х некоторые из опытных разработчиков программного обеспечения Gangs of Four (GOF) ознакомились с упоминавшейся выше работой Александра об архитектурных шаблонах. Они задались вопросом, возможно ли применение идеи архитектурных шаблонов при реализации проектов в области создания программного обеспечения. Сформулируем те вопросы, на которые требовалось получить ответ.

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

Интуиция подсказывала исследователям, что ответы на оба эти вопроса определено будут положительными. Следующим шагом необходимо было идентифицировать несколько подобных шаблонов и разработать стандартные методы каталогизации новых. Хотя в начале 1990х над шаблонами проектирования работали многие исследователи, наибольшее влияние на это сообщество оказала книга Гаммы, Хелма, Джонсона и Влиссайдеса Шаблоны проектирования: элементы многократного использования кода в объектно-ориентированном программировании. Эта работа приобрела очень широкую известность. Свидетельством ее популярности служит тот факт, что четыре автора книги получили шуточное прозвище "банда четырех". Важно понять, что авторы сами не создавали тех шаблонов, которые описаны в их книге. Скорее, они идентифицировали эти шаблоны как уже существующие в разработках, выполненных сообществом создателей программного обеспечения. Поэтому у некоторых разработчиков возникает вопрос, зачем изучать шаблоны?

1.3 Зачем нужно изучать шаблоны

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

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

- Применение единой терминологии. Профессиональное общение и работа в группе (команде разработчиков) требует наличия единого базового словаря и единой точки зрения на проблему. Шаблоны проектирования предоставляют подобную общую точку зрения как на этапе анализа, так и при реализации проекта.
- Шаблоны проектирования предоставляют нам абстрактный высокоуровневый взгляд как на проблему, так и на весь процесс объектноориентированной разработки. Это помогает избежать излишней детализации на ранних стадиях проектирования.

1.4 Другие преимущества

В результате применения шаблонов проектирования повышается эффективность труда отдельных исполнителей и всей группы в целом. Это происходит из-за того, что начинающие члены группы видят на примере более опытных разработчиков, как шаблоны проектирования могут применяться и какую пользу они приносят. Совместная работа дает новичкам стимул и реальную возможность быстрее изучить и освоить эти новые концепции. Применение многих шаблонов проектирования позволяет также создавать более модифицируемое и гибкое программное обеспечение. Причина состоит в том, что эти решения уже испытаны временем. Поэтому использование шаблонов позволяет создавать структуры, допускающие их модификацию в большей степени, чем это возможно в случае решения, первым пришедшего на ум. Шаблоны проектирования, изученные должным образом, существенно помогают общему пониманию основных принципов объектноориентированного проектирования.

«Бандой четырех» было предложено несколько стратегий создания хорошего объектноориентированного проекта. В частности, они предложили следующее:

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

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

Еще одно преимущество состоит в том, что шаблоны проектирования позволяют разработчику или группе разработчиков находить проектные решения для сложных проблем, не создавая громоздкой иерархии наследования классов.

2 UML-шпаргалка

Напомним о способах взаимосвязи различных классов. Если два класса как-то связаны друг с другом, то в UML диаграмме это отображается через ассоциацию. Существуют два типа ассоциаций: однонаправленная и двунаправленная. В этой связи каждый из объектов класса имеет свой жизненный цикл, и не существует отношения владения. Реализация ассоциации может быть разной. Например, есть две сущности банк и пользователь банка. Банк может хранить деньги пользователя, а тот, в свою очередь, забирать деньги из банка. Это пример двунаправленной ассоциации, которая может быть реализована через импорт в каждый класс ассоциированного класса и использование его в качестве типа параметра некоторого метода.

Пример однонаправленной ассоциации: факультатив и студенты. С каждым факультативом может быть ассоциировано несколько студентов, с свою очередь студент может быть ассоциирован только с одним факультативом. Ассоциация может быть реализована через список ссылок на студентов.

Человек может пользоваться несколькими банками, тогда как студент может быть ассоциирован только с одним факультетом, то есть ассоциации могут выражать различные типы связей таких, как один-к-одному, один-ко-многим, много-ко-многим.

2.1 Аггрегация

Аггрегация — это специализированная форма Ассоциации, где все объекты имеют собственный жизненный цикл, но собственность и дочерние объекты не могут принадлежать другому родительскому объекту.

Например, сотрудники ЕРАМ не могут быть сотрудниками другой фирмы. Дочерний класс сотрудника также не может принадлежать другой фирме. Однако объекты классов могут существовать отдельно друг от друга, так, например, в случае увольнения объект сотрудника не уничтожается, а продолжает свое дальнейшее существование.

2.2 Композиция

Композиция — специальный тип агрегации, когда ассоциированный объект не имеет своего жизненного цикла вне объекта владельца. Например, солнечной системе принадлежат различные планеты. При уничтожении солнечной системы, планеты будут уничтожены также.

2.3 Ассоциация, агрегация, композиция

Можно сказать, что ассоциация является базовым классом для агрегации и композиции, а агрегация — базовым для композиции.

3 Структурные паттерны

Структурные шаблоны связаны со способами формирования более крупных структур. Структурные шаблоны классов используют наследование или композицию для создания интерфейсов или реализаций.

3.1 Фасад

3.1.1 Назначение

В книге «банды четырех» назначение шаблона Facade (фасад) определяется следующим образом:

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

3.1.2 Мотивация

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

Известным примером фасада является JDBC интерфейс в Java, так как его пользователи создают подключение через интерфейс `java.sql.Connection`, реализуется которого их не касается.

3.1.3 Пример

Рассмотрим сложную систему старта автомобиля. В некотором примерном и упрощенном виде, предположим, эта система состоит из подсистем двигатель, топливный инжектор, контроллер температуры, радиатор, топливный насос, датчики температуры.

Старт автомобиля потребует знаний каждой подсистемы и поочередный вызов их методов, причем в необходимом порядке. Для пользователя системы, которому нужно лишь стартовать автомобиль такие знания являются избыточными и усложняют использование системы. Мы можем реализовать класс фасад, который будет предоставлять пользователю простой интерфейс и инкапсулировать в себя сложную логику запуска системы автомобиля.

3.1.4 Использование

- Когда нет необходимости использовать все функциональные возможности сложной системы и можно создать новый класс, который будет содержать все необходимые средства доступа к базовой системе. Если предполагается работа лишь с ограниченным набором функций исходной системы, как это

обычно и бывает, интерфейс (API), описанный в новом классе, будет намного проще, чем стандартный интерфейс, разработанный создателями основной системы.

- Используйте фасад для отделения подсистем от клиентов, способствуя тем самым снижению зависимостей и повышению переносимости
- Если подсистемы системы между собой зависимы, то, используя фасад, можно упростить зависимости между подсистемами.

3.2 Адаптер

3.2.1 Назначение

В книге «банды четырех» назначение шаблона определяется следующим образом:

Преобразование интерфейса класса в интерфейс, нужный пользователю.

3.2.2 Мотивация

Пусть есть некоторая система, и планируется расширить ее функциональность путем добавления некоторого нового класса. Причем новый класс не может быть использован, так как его интерфейс не совместим с интерфейсами уже использующихся классов. В таком случае применяется класс-адаптер, который преобразует интерфейс желаемого класса к виду уже существующих интерфейсов классов.

3.2.3 Пример

Предположим, что есть некоторое устройство-хост, которое занимается сбором показаний температуры с различных датчиков разных фирм. Датчики подключены к хосту по COM порту. В системе есть базовый класс датчик температуры, который имеет публичный метод «считать данные по ком-порту».

Заказчик приобрел очень точный и надежный датчик, который работает по USB порту. Необходимо считывать показания с USB-датчика. С новым датчиком распространяется класс, с помощью объекта которого можно запросить данные с датчика. Однако данный класс реализует интерфейс, который отличается от интерфейса базового класса системы.

Если изменить базовый класс путем добавления в него еще одного метода для считывания данных по USB порту для решения задачи, то тогда придется менять реализации дочерних классов. Такое решение является трудозатратным или его вообще невозможно реализовать, если нет доступа к коду класса какого-то из датчиков.

В такой ситуации можно применить адаптер, который преобразует интерфейс нового класса в необходимый. Эта задача имеет проекцию на реальный мир, когда используются физические адаптеры, решающие ту же задачу.

Адаптер можно реализовать двумя способами:

1. путем создания экземпляра адаптируемого класса;
2. путем наследования интерфейса адаптируемого класса.

3.2.4 Использование

- Вы хотите использовать существующий класс, и его интерфейс не соответствует тому, который вам нужен
- Вы хотите создать повторно используемый класс, который взаимодействует с несвязанными или непредвиденными классами, то есть классами, которые не обязательно имеют совместимые интерфейсы.
- Вам нужно использовать несколько существующих подклассов, не nepraktично адаптировать их интерфейсы путем наследования каждого. Адаптер может адаптировать интерфейс их родителя.

3.3 Мост

3.3.1 Назначение

Отделить абстракцию от реализации, что позволит независимо вносить изменения в абстракцию и в реализацию.

3.3.2 Мотивация

Когда абстракция может иметь одну из нескольких возможных реализаций, стандартным способом их размещения является использование наследования. Абстрактный класс определяет интерфейс к абстракции, а конкретные подклассы реализуют его по-разному. Но этот подход не всегда достаточно гибок. Наследование связывает постоянная реализация абстракции, что затрудняет самостоятельное изменение, расширение и повторное использование абстракций и реализаций.

3.3.3 Пример

Пусть необходимо реализовать систему, которая имеет возможность работать с различными базами данных. На UML диаграмме видно, что решение через наследование заставляет разработчика реализовывать большое число наследников. Это связано с тем, что одновременно мы пытаемся решить две задачи: предоставить интерфейс и реализацию, то есть смешиваем две функциональности: "что сделать" и "как сделать". Выделим "что сделать" в интерфейс, "как сделать" — в реализацию. Интерфейс агрегирует реализацию.

3.3.4 Использование

- избежание постоянной связи между абстракцией и ее реализацией. Например, когда реализация должна быть выбрана или переключена во время выполнения;

- и абстракции, и их реализации должны быть расширяемыми с помощью подклассов. В этом случае паттерн Bridge позволяет комбинировать различные абстракции и реализации и независимо расширять их;
- изменения в реализации абстракции не должны влиять на клиентов; то есть их код не должен быть перекомпилирован.

3.4 Декоратор (обертка)

3.4.1 Назначение

Динамическое расширение функциональности. Декораторы предоставляют гибкую альтернативу подклассам для расширения функциональности.

3.4.2 Мотивация

Добавление функциональных возможностей отдельному объекту, а не всему классу.

3.4.3 Пример

Пусть у нас есть плата, которая позволяет проводить замеры различных физических величин. Периодически к телеметрическому сервису поступит запрос на чтение того или иного датчика. Мы хотим "на лету" считывать показатели в зависимости от запроса. Интерфейс платы имеет метод `read`, который позволяет считывать данные с некоторого датчика. Создадим несколько классов, каждый из которых соответствует определенной физической величине и реализует интерфейс. Декоратор `TelemetryService` также реализует интерфейс платы и включает в себя конкретную реализацию датчика, через которую может производить считывание конкретной величины.

3.4.4 Использование

- динамически и прозрачно добавлять обязанности к отдельным объектам, не затрагивая другие объекты;
- для обязанностей, которые могут быть сняты. То есть можно проводить не только расширение, но и сужение;
- когда расширение подклассами нецелесообразно. Иногда возможно большое количество независимых расширений, что может привести к взрыву подклассов для поддержки каждой комбинации. Или определение класса может быть скрыто или иным образом недоступно для подклассов. .