**Паттерны проектирования(Design patterns)**

**Что это такое?**

**Шаблон проектирования** или **паттерн** (англ. design pattern) — архитектурная конструкция, представляющая собой решение конкретной проблемы в рамках некоторого часто возникающего контекста.

Обычно шаблон не является законченным образцом, который может быть преобразован прямо в код. Это лишь пример решения задачи, который можно использовать в различных ситуациях. Объектно-ориентированные шаблоны показывают отношения и взаимодействия между классами или объектами, без определения того, какие конечные классы ил объекты будут использоваться.

**Классификация**

Классифицируют шаблоны всегда по-разному. В зависимости от применения могут разбиваться на группы. Чаще всего разбиение производится по уровням. «Низкоуровневые» шаблоны, учитывающие специфику конкретного языка программирования, называются [**идиомами**](https://ru.wikipedia.org/wiki/Идиома_(программирование)). Это хорошие решения проектирования, характерные для конкретного языка или программной платформы, и потому не универсальные. На наивысшем уровне существуют **архитектурные шаблоны**, они охватывают собой архитектуру всей [программной системы](https://ru.wikipedia.org/wiki/Система).

[Алгоритмы](https://ru.wikipedia.org/wiki/Алгоритм) по своей сути также являются шаблонами, но не проектирования, а [вычисления](https://ru.wikipedia.org/wiki/Вычисление), так как решают вычислительные задачи.

**Немного истории**

* В [1970-е](https://ru.wikipedia.org/wiki/1970-е) годы [архитектор](https://ru.wikipedia.org/wiki/Архитектор) [Кристофер Александр](https://ru.wikipedia.org/wiki/Александер,_Кристофер) составил набор шаблонов проектирования. В области архитектуры эта идея не получила такого развития, как позже в области программной разработки.
* В [1987 году](https://ru.wikipedia.org/wiki/1987_год) [Кент Бэк](https://ru.wikipedia.org/wiki/Бек,_Кент) (Kent Beck) и [Вард Каннингем](https://ru.wikipedia.org/wiki/Каннингем,_Вард) (Ward Cunningham) взяли идеи Александра и разработали шаблоны применительно к разработке программного обеспечения для разработки графических оболочек на языке [Smalltalk](https://ru.wikipedia.org/wiki/Smalltalk).
* В [1988 году](https://ru.wikipedia.org/wiki/1988_год) [Эрих Гамма](https://ru.wikipedia.org/wiki/Гамма,_Эрих) (Erich Gamma) начал писать докторскую диссертацию при цюрихском университете об общей переносимости этой методики на разработку программ.
* В 1989—1991 годах Джеймс Коплин (James Coplien) трудился над разработкой идиом для программирования на [C++](https://ru.wikipedia.org/wiki/C%2B%2B) и опубликовал в 1991 году книгу Advanced C++ Idioms.

В этом же году Эрих Гамма заканчивает свою докторскую диссертацию и переезжает в [США](https://ru.wikipedia.org/wiki/США), где в сотрудничестве с Ричардом Хелмом (Richard Helm), Ральфом Джонсоном (Ralph Johnson) и Джоном Влиссидсом (John Vlissides) публикует книгу [**Design Patterns** — Elements of Reusable Object-Oriented Software](https://ru.wikipedia.org/wiki/Design_Patterns). В этой книге описаны 23 шаблона проектирования. Также команда авторов этой книги известна общественности под названием «Банда четырёх» ([англ.](https://ru.wikipedia.org/wiki/Английский_язык) *Gang of Four*, часто сокращается до *GoF*). Именно эта книга стала причиной роста популярности шаблонов проектирования.

**Паттерны проектирования классов/объектов**

Описание системы в терминах классов/объектов следует считать низшим уровнем ее представления. В свою очередь, при моделировании системы на уровне классов/объектов обычно проводят дополнительную типологизацию в двух аспектах, а именно, описывают структуру системы в терминах микроскопических элементов и то, каким образом такая система обеспечивает требуемый функционал. Соответственно, среди паттернов проектирования выделены **структурные паттерны**и **паттерны распределения обязанностей между классами/объектами**. Поскольку отдельные объекты создаются и уничтожаются в процессе работы системы, выделена еще одна большая группа **паттернов проектирования**, которые служат для создания объектов.

**Примеры:** Адаптер(Adapter), Декоратор (Decorator), Мост (Bridge), Стратегия (Strategy), Прототип (Prototype), Абстрактная фабрика (Abstract Factory), Одиночка (Singeton), Строитель (Builder).

**Архитектурные системные паттерны**

Архитектурные системные паттерны объединены в группы в соответствии с теми задачами, для решения которых они разработаны. Для организации классов или объектов системы в базовые подструктуры используются **структурные** **архитектурные паттерны**. С другой стороны, для обеспечения требуемого системного функционала первостепенное значение имеет адекватная организация взаимодействия отдельных архитектурных элементов системы - этой цели служат **паттерны управления**.

**Примеры:** MVC (modal-view-controller), MVVM (modal-view-viewmodal), Активная запись (Active Record).

**Паттерны интеграции корпоративных информационных систем**

Паттерны интеграции информационных систем представляют собой, как это описано в разделе 2, верхний уровень классификации паттернов проектирования. Аналогично паттернам более низких уровней классификации, среди паттернов интеграции выделена группа **структурных паттернов**. Структурные паттерны описывают основные компоненты единой интегрированной метасистемы. В свою очередь, для описания взаимодействия отдельных корпоративных систем, включенных в интегрированную метасистему, организована группа паттернов, объединенных в соответствии с тем или иным **методом интеграции**. Далее, интеграция корпоративных информационных систем подразумевает тем или иным способом организованный обмен данными между системами. Для организации обмена информацией между отдельными системами, включенными в интегрированную метасистему, служат **паттерны интеграции по типу обмена данными**. Следует отметить, что в отличие от паттернов проектирования классов/обьектов и архитектурных системных паттернов, отнесение отдельного паттерна интеграции к тому или иному виду является менее условным.

**Примеры:** Взаимодействие «точка - точка», Файловый обмен, Удаленный вызов процедур.

**Паттерны параллельного программирования (Concurrency)**

Представляют собой примитивы синхронизации потоков, обеспечивают асинхронный и безопасный доступ к ресурсу, наблюдают за состояниями объектов.

**Примеры:** Замок (Lock), активный объект (Active Object), Планировщик (Scheduler).

**Конкретные примеры**

**1) Мост**

|  |  |
| --- | --- |
| **Проблема** | Требуется отделить абстракцию от реализации так, чтобы и то и другое можно было изменять независимо. При использовании наследования реализация жестко привязывается к абстракции, что затрудняет независимую модификацию. |
| **Решение** | Поместить абстракцию и реализацию в отдельные иерархии классов. |
| **Рекомендации** | Можно использовать если, например, реализацию необходимо выполнять во время реализации программы. |
| **Пример** | "Абстракция" определяет интерфейс "Абстракции" и хранит ссылку на объект "Реализация", "УточненнаяАбстракция" расширяет интерфейс, определенный "Абстракцией". "Реализация" определяет интерфейс для классов реализации, он не обязан точно соответствовать интерфейсу класса "Абстракция" - оба интерфейса могут быть совершенно различны. Обычно интерфейс класса "Реализация" предоставляет только примитивные операции, а класс "Абстракция" определяет операции более высокого уровня, базирующиеся на этих примитивных. "КонкретнаяРеализация" содержит конкретную реализацию класса "Реализация". Объект "Абстракция" перенаправляет своему объекту "Реализация" запросы "Клиента". |
| **Преимущества** | Отделение реализации от интерфейса, то есть, "Реализацию" "Абстракции" можно конфигурировать во время выполнения. Кроме того, следует упомянуть, что разделение классов "Абстракция" и "Реализация" устраняет зависимости от реализации, устанавливаемые на этапе компиляции: чтобы изменить класс "Реализация" вовсе не обязательно перекомпилировать класс "Абстракция". |

**2) Строитель**

|  |  |
| --- | --- |
| **Проблема** | Отделить конструирование сложного объекта от его представления, так чтобы в результате одного и того же конструирования могли получаться различные представления. Алгоритм создания сложного объекта не должен зависеть от того, из каких частей состоит объект и как они стыкуются между собой. |
| **Решение** | "Клиент" создает объект - распорядитель "Директор" и конфигурирует его объектом - "Строителем". "Директор" уведомляет "Строителя" о том, что нужно построить очередную часть "Продукта". "Строитель" обрабатывает запросы "Директора" и добавляет новые части к "Продукту", затем "Клиент" забирает "Продукт" у "Строителя". |
| **Преимущества** | Объект "Строитель" предоставляет объекту "Директор" абстрактный интерфейс для конструирования "Продукта", за которым может скрыть представление и внутреннюю структуру продукта, и , кроме того, процесс сборки "продукта". Для изменения внутреннего представления "Продукта" достаточно определить новый вид "Строителя". Данный паттерн изолирует код, реализующий создание объекта и его представление. |

**3) Абстрактная фабрика**

|  |  |
| --- | --- |
| **Проблема** | Создать семейство взаимосвязанных или взаимозависимых объектов (не специфицируя их конкретных классов). |
| **Решение** | Создать абстрактный класс, в котором объявлен интерфейс для создания конкретных классов. |
| **Пример** | Какой класс должен отвечать за создание объектов - адаптеров при использовании паттерна "Адаптер". Если подобные объекты создаются неким объектом уровня предметной области, то будет нарушен принцип разделения обязанностей. |
| **Преимущества** | Изолирует конкретные классы. Поскольку "Абстрактная фабрика" инкапсулирует ответственность за создание классов и сам процесс их создания, то она изолирует клиента от деталей реализации классов. Упрощена замена "Абстрактной фабрики", поскольку она используется в приложении только один раз при инстанцировании. |
| **Недостатки** | Интерфейс "Абстрактной фабрики" фиксирует набор объектов, которые можно создать. Расширение "Абстрактной фабрики" для изготовления новых объектов часто затруднительно. |

**4) Адаптер**

|  |  |
| --- | --- |
| **Проблема** | Необходимо обеспечить взаимодействие несовместимых интерфейсов или как создать единый устойчивый интерфейс для нескольких компонентов с разными интерфейсами. |
| **Решение** | Конвертировать исходный интерфейс компонента к другому виду с помощью промежуточного объекта - адаптера, то есть, добавить специальный объект с общим интерфейсом в рамках данного приложения и перенаправить связи от внешних объектов к этому объекту - адаптеру. |
|  |  |

**5) Декоратор(Обертка)**

|  |  |
| --- | --- |
| **Проблема** | Возложить дополнительные обязанности (прозрачные для клиентов) на отдельный объект, а не на класс в целом. |
| **Рекомендации** | Применение нескольких "Декораторов" к одному "Компоненту" позволяет произвольным образом сочетать обязанности, например, одно свойство можно добавить дважды. |
| **Решение** | Динамически добавить объекту новые обязанности не прибегая при этом к порождению подклассов (наследованию). "Компонент"определяет интерфейс для обьектов, на которые могут быть динамически возложены дополнительные обязанности, "КонкретныйКомпонент" определяет объект, на который возлагаются дополнительные обязанности, "Декоратор" - хранит ссылку на объект "Компонент" и определяет интерфейс, соответствующий интерфейсу "Компонента". "КонкретныйДекоратор" возлагает дополнительные обязанности на компонент. "Декоратор" переадресует запросы объекту "Компонент". |
| **Преимущества** | Большая гибкость, чем у статического наследования: можно добавлять и удалять обязанности во время выполнения программы в то время как при использовании наследования надо было бы создавать новый класс для каждой дополнительной обязанности. Данный паттерн позволяет избежать перегруженных методами классов на верхних уровнях иерархии - новые обязанности можно добавлять по мере необходимости. |
| **Недостатки** | "Декоратор" и его "Компонент" не идентичны, и, кроме того, получается что система состоит из большого числа мелких объектов, которые похожи друг на друга и различаются только способом взаимосвязи а не классом и не значениями своих внутренних переменных - такая система сложна в изучении и отладке. |

**6) Синглтон**

|  |  |
| --- | --- |
| **Проблема** | Какой специальный класс должен создавать "Абстрактную фабрику", и как получить к ней доступ? Необходим лишь один экземпляр специального класса, различные объекты должны обращаться к этому экземпляру через единственную точку доступа. |
| **Решение** | Создать класс и определить статический метод класса, возвращающий этот единственный объект. |
| **Рекомендации** | Разумнее создавать именно статический экземпляр специального класса, а не объявить требуемые методы статическими, поскольку при использовании методов экземпляра можно применить механизм наследования и создавать подклассы. Статические методы в языках программирования не полиморфны и не допускают перекрытия в производных классах. Решение на основе создания экземпляра является более гибким, поскольку впоследствии может потребоваться уже не единственный экземпляр объекта, а несколько. |

**Достоинства**

В сравнении с полностью самостоятельным проектированием, шаблоны обладают рядом преимуществ. Основная польза от использования шаблонов состоит в снижении сложности разработки за счёт готовых абстракций для решения целого класса проблем. Шаблон даёт решению свое имя, что облегчает коммуникацию между разработчиками, позволяя ссылаться на известные шаблоны. Таким образом, за счёт шаблонов производится унификация деталей решений: модулей, элементов проекта, — снижается количество ошибок. Применение шаблонов концептуально сродни использованию готовых библиотек кода. Правильно сформулированный шаблон проектирования позволяет, отыскав удачное решение, пользоваться им снова и снова. Набор шаблонов помогает разработчику выбрать возможный, наиболее подходящий вариант проектирования.

**Недостатки**

Хотя легкое изменение кода под известный шаблон может упростить понимание кода, по мнению Стива Макконнелла (Сти́вен Макко́ннелл — американский программист, автор книг по разработке программного обеспечения. Журнал «Software Development» дважды удостоил его книги премии Jolt Excellence как лучшие книги года о разработке программного обеспечения), с применением шаблонов могут быть связаны две сложности. Во-первых, слепое следование некоторому выбранному шаблону может привести к усложнению программы. Во-вторых, у разработчика может возникнуть желание попробовать некоторый шаблон в деле без особых оснований.

Многие шаблоны в ООП можно рассматривать как идиоматическое воспроизведение элементов функциональных языков. Пол Грэхэм (Пол Грэм — американский предприниматель, программист, известный сторонник и пропагандист использования языка программирования Lisp. В числе прочего, создал диалект Lisp, названный им Arc. Автор ряда книг по программированию) считает саму идею шаблонов проектирования — анти-паттерном, сигналом о том, что система не обладает достаточным уровнем абстракции, и необходима её тщательная переработка. Нетрудно видеть, что само определение шаблона как «готового решения, но не прямого обращения к библиотеке» по сути означает отказ от повторного использования в пользу дублирования. Это, очевидно, может быть неизбежным для сложных систем при использовании языков, не поддерживающих комбинаторы и полиморфизм типов, и это в принципе может быть исключено в языках, обладающих этими свойствами (хотя и не обязательно эффективно), так как любой шаблон может быть реализован в виде исполнимого кода.

**Анти-паттерны**

Это классы наиболее часто внедряемых плохих решений проблем. Они изучаются, как категория, в случае когда их хотят избежать в будущем, и некоторые отдельные случаи их могут быть распознаны при изучении неработающих систем.

Термин происходит от паттернов.

* Дым и зеркала (Smoke and mirrors): Демонстрация того, как будут выглядеть ненаписанные функции (название происходит от двух излюбленных способов, которымифокусники скрывают свои секреты)
* Раздувание ПО (Software bloat): Разрешение последующим версиям системы требовать всё больше и больше ресурсов
* Функции для галочки: Превращение программы в конгломерат плохо реализованных и не связанных между собой функций (как правило, для того, чтобы заявить врекламе, что функция есть)
* Полтергейст (Poltergeist): Объекты, чьё единственное предназначение — передавать информацию другим объектам
* Слепая вера (Blind faith): Недостаточная проверка корректности исправления ошибки или результата работы подпрограммы
* Воняющий подгузник (The Diaper Pattern Stinks): Сброс флага ошибки без её обработки или передачи вышестоящему обработчику
* Мыльный пузырь (Soap bubble): Объект, инициализированый мусором, максимально долго притворяется, что содержит какие-то данные
* Бензиновая фабрика (Gas factory): Необязательная сложность дизайна
* Золушкина туфелька: Попытка "натянуть" на объект уже имеющийся малоподходящий по смыслу интерфейс, вместо создания нового.
* Коммит-убийца (Commit assasin): Внесение отдельных изменений в систему контроля версий без проверки влияния их на другие части системы. Как правило, после подобных коммитов работа коллектива парализуется на время исправления проблем в местах, которые ранее работали безошибочно.