**Паттерны (GoF) –** эффективные решения типовых проблем проектирования ПО. Имеет 2 ключевых аспекта: структура или приемы, которые могут повторяться в различных паттернах и проблема, которую решает паттерн – является отличительной чертой каждого.

*Типы шаблонов:*

Порождающие – отделяют логику создания объектов от остальной логики и предоставляют интерфейс для его создания. (Фокус на создании объектов и его абстрагирование).

Структурные – определяют отношения между классами и объектами указывая, как из классов и объектов создать более сложные структуры.

Поведенческие – определяют способы взаимодействия между классами и объектами. (Фокус на взаимодействии между классами, выделение поведения в классы)

**Creational:**

**Factory method (фабричный метод)** – определяет интерфейс для создания объектов некого базового класса, но решение о том, объект какого класса базового типа инстанциировать определяется наследниками фабричного метода.

Определяется интерфейс фабрики с фабричным методом, реализации этой фабрики будут переопределять этот метод и возвращать специфичный объект базового типа.

Преимущества: Создание объектов отделено от основного кода (ослабление связанности), использование базового класса (упрощает добавление новых классов базового типа).

* В приспособленце также используется для их хранения и определения необходимости создания их создания.

**Abstract Factory (абстрактная фабрика)** – определяет интерфейс для создания семейств связанных объектов, не привязываясь к конкретным классам создаваемых объектов.

Определяется интерфейс фабрики, каждый метод которой специфично реализован наследником, каждая реализация сама решает, какие конкретные типы этого семейства будут определены.

Преимущества: Те же, что и у фабричного метода, плюс гарантирует, что создаваемые семейства типов будут сочетаемы.

* Различие этих фабрик заключается в том, что Abstract Factory используется для создания семейств объектов, factory method для одного объекта. Каждый метод абстрактной фабрики может быть реализован через фабричный метод или прототип.

**Singleton (Одиночка) –** гарантирует, что класс имеет только один экземпляр с глобальной точкой доступа.

Скрывается конструктор, создается static метод, который контролирует ЖЦ одиночки.

Проблемы: Нарушает SRP, имеет проблемы с многопоточностью, при тестировании требует создания Mock-объекта, может иметь проблемы глобальной переменной: глобальный доступ нарушает модульность программы (повышает связанность), изменяемое состояние, если его поля не final.

* Остальные порождающие паттерны могут реализованы с помощью одиночки (кроме абстрактного метода), также может быть реализован фасад.
* Отличие от глобальной переменной в том, что ее состояние можно изменить, синглтон гарантирует его неизменность (саму ссылку не изменить, но состояние полей можно, если они не final, плюс, если они не финал, то проблема с многопоточностью).

**Builder (Строитель) –** предназначен для того, чтобы вынести конструирование объекта за пределы собственного класса, таким образом, чтобы в результате одного и того же процесса конструирования могли получаться разные представления. Позволяет создавать разные варианты одного и того же объекта с помощью пошагового вызова методов вместо использования конструктора. Или же: предоставляет API для конструирования сложного объекта.

Builder:

Определяется интерфейс builder’а, в котором объявляются методы для установки полей этого объекта и метод получения результата. Его реализации определяют специфичные случаи конструирования этого объекта. Опционально может вводится класс директор, который определяет порядок строительства объектов.

* + Чем-то похож на абстрактную фабрику, только на уровне объекта.

Fluent Builder:

Конфигурируется с помощью вложенного статического класса, который имеет те же поля, что и внешний класс и реализует методы setter’ы для их конфигурации и метод build, который устанавливает значения builder во внешний класс. Запрещает создание объекта вне билдера.  
Предпочтителен, так как имеет удобный API в виде цепочки вызываемых методов.

**Prototype (Прототип) –** предназначен для создания объектов класса с помощью клонирования, не вдаваясь в подробности реализации.

Создание интерфейса с методом copy() или имплементация интерфейса Cloneable и метода clone().

Преимущества: Копирование объекта инкапсулировано в самом копируемом классе (уменьшение связанности, решается проблема недоступности приватных полей внешнему классу, позволяет избежать использования конструктора класса).

* Проблема всех порождающих паттернов, кроме Singleton и Fluent Builder в том, что они не инкапсулируют логику создания своих объектов (возможно создание объектов с помощью их конструкторов).

**Structural:**

**Adapter (Адаптер) –** позволяет конвертировать интерфейс класса (публичные методы) в другой интерфейс, ожидаемый клиентом. Т.е. выступает посредником между классом, который имеет нужный и реализованный функционал и интерфейсом, который необходим клиенту.

Реализует интерфейс, ожидаемый клиентом, на основе интерфейса или класса, который нужно адаптировать.

**Bridge (Мост) –** разделяетодин или несколько классов на две отдельные иерархии – абстракцию (слой делегирующий работу реализации) и реализацию, связь которых осуществляется посредством абстракции, а не реализации, что дает возможность к расширению независимо друг от друга.

Пример: Вид управления устройством (пульт, приложение) и Устройство (телик, проектор и тд), если использовать наследование или связь на основе реализации, то нужно будет создать по 1 классу на каждую комбинацию вида управления и устройства. Мост предлагает выделить 2 отдельные иерархии классов Абстрактное устройство и Абстрактное управление устройством и связать их посредством этих абстракций.

**Decorator (Декоратор)** – предоставляет возможность динамического добавления нового поведения объекту, путем создания оболочки для расширяемого объекта, которая будет расширять функционал базового класса. Использует агрегацию или композицию взамен создания подклассов.

Должен предоставлять тот же (или расширенный) интерфейс, что и класс, функционал которого он расширяет (с помощью наследования), должен быть ассоциирован с объектом, который расширяет (иметь поле этого объекта).

Преимущества: Добавление нового функционала, не изменяя сам класс, использованию наследования предпочитает агрегацию или композицию.

**Facade (Фасад)** – предоставляет простой (иногда урезанный) интерфейс, который скрывает сложность системы и инкапсулирует логику взаимодействия этих классов, делая систему проще для использования.

**Преимущества**: создает точку доступа к сложной системе, изолируя клиентов от ее компонентов.

**Composite (Компоновщик) –** шаблон контейнер,позволяет сгруппировать множество объектов в древовидную структуру и работать с ней, как с одним объектом.

Агрегирует все компоненты в виде коллекции этих объектов и расширяет интерфейс этой группы объектов, путем делегирования работы дочерним компонентам (в реализации вызывает метод интерфейса компонентов).

**Flyweight (Легковес/Приспособленец) –** позволяет использовать разделяемые объекты (приспособленцы) с общим и неизменным состоянием сразу в нескольких контекстах.

Легковес **–** класс, в который выделено общее переиспользуемое состояние (должно быть immutable), фабрика легковесов – представлена Object pool (создает, хранит, если уже создан – выдает имеющийся).

* Возможно, string pool отражает идеи этого паттерна.

Информация, которая не хранится в легковесе является уникальной для каждого объекта

Преимущества: помогает избегать порождение одинаковых объектов (экономит оперативную память).

**Proxy (Заместитель/Суррогат) –** позволяет создавать объекты-прослойки между клиентом и сервисным объектом, который выполняет свою функцию, а затем делегирует вызов сервисному объекту. Управляет ЖЦ проксируемого класса.

Расширяет тот же интерфейс, что и сервисный класс, а также ассоциирован с ним. Метод расширяемого интерфейса выполняет свою функцию до или после передачи вызова сервисному объекту.

Применяется для кэширования, логирования, ленивой инициализации, защите доступа, удаленный прокси

Отличия от декоратора: Контролирует ЖЦ объекта (не передается извне – в конструкторе, а создается в классе, то есть не расширяет интерфейс, а, следовательно, клиент не видит разницу в использовании), не добавляется новый функционал к проксируемому классу (*поведение проксируемого класса не изменяется,* добавляется дополнительная механика типа кэширования, логирования и др.).

**Behavioral:**

**Chain of Responsibility (Цепочка обязанностей) –** позволяет передать запрос по цепочке потенциальных обработчиков, где каждый обработчик решает обработает он запрос самостоятельно или передаст следующему обработчику.

Определить классы обработчики – каждый обработчик отражает определенное поведение или состояние обрабатываемого объекта. Определяются метод обработки (объект передается по цепочке с помощью его параметров) и метод связывания обработчиков в цепочку.

Классический вариант – цепочка прерывается, если обработчик способен выполнить работу, но можно реализовать логику обработки всей цепи.

**Command/Action (Команда/Действие) –** преобразует запросы в объекты и эти запросы исполняются по команде снаружи.

Преобразование команды в объект дает возможность использовать преимущества ООП, наполнять параметрами, группировать, сортировать, отменять, логировать.

Создание, вызов и исполнение происходят отдельно друг от друга.

Структура:

Client (Создатель) – создает объекты команд и конфигурирует их контекст, передает их Инициатору (связывает команды с Инициатором).  
Invoker (Инициатор) – сущность, инициирующая исполнение команды (ассоциирован с командами, но не знает, с какой конкретно командой он работает, просто вызывает execute(). Помимо инициации исполнения на него могут быть возложены функции логирования команд, группировки, исполнение по приоритету и др.)  
Receiver (Исполнитель) – объект, который фактически выполняет работу.  
Command (Команда) – каждый класс – объектное представление метода исполнителя (Связывает запрос и его исполнителя, делегирует работу исполнителю)

**Iterator (Итератор)** – позволяет обеспечить универсальный последовательный доступ к элементам структуры данных. Инкапсулирует логику обхода структуры данных в отдельный объект – итератор (т.е. поведение = объект, поведение отделено от структуры данных).

Структура:

Агрегатор – описывает способ генерации итератора коллекции (он же Iterable в java). Итератор – описывает интерфейс для обхода коллекции (он же Iterator в Java).

**Mediator (Посредник)** – позволяет уменьшить связанность множества классов за счет перемещения этих связей в класс-посредник (поведение = объект). Компоненты системы не имеют контактов друг с другом, общаются через медиатор.

Структура: Вызывающий класс ассоциирован с базовым классом медиатора и определяет нужную реализацию посредством инициализации конкретного медиатора. Медиатор ассоциирован с классами исполнителями и получает вызывающий объект для контекста (если нужно), переадресует вызов классу исполнителю.

**Memento (Снимок) –** позволяет сохранять и восстанавливать прошлые состояния объекта, не раскрывая подробности его реализации.

Структура:

Создатель – целевой класс, который создает снимки своего состояния и может восстанавливать своё состояние из снимка. (доступ к приватным полям). Снимок – неизменяемый класс, описывающий состояние создателя. Класс опекун – хранит сохраненные снимки.

Лучше реализовать как вложенный класс.

**Observer (Наблюдатель) –** Создает механизм подписки, который позволяет одним объектам следить и реагировать на события, происходящие в других объектах, их оповещающих.

Структура: Издатель – класс, владеющий внутренним состоянием, которое интересует подписчиков, содержит механизм подписки: хранит своих подписчиков, добавляет и удаляет подписчиков из списка, оповещает их об изменении. Подписчик - наблюдающий класс, имеет метод типа notify, который вызывает издатель при изменении своего состояния (Издатель и Подписчик, естественно, реализованы в виде интерфейсов).

**State (Состояние)** – позволяет объекту изменять свое поведение в зависимости от состояния. Реализуется через вынесение поведения, связанного с конкретным состоянием, в отдельные классы.

Структура:

Класс контекст – содержит поле изменяющегося состояния, которому будет делегирована часть работы, зависящая от конкретного состояния. Классы состояний – содержат поведение, зависящее от состояния контекста.

Реализации:

Переходом из одного состояния в другое управляет либо контекст (главный объект), но тогда придется реализовать метод с большим кол-вом if-else, который будет переключать состояния, либо сами состояния, тогда не будет раздутого метода, но нужно пробрасывать объект контекста для установки его состояния (ну или вообще переключать состояние в клиенте).

**Strategy (Стратегия) –** определяет семейство схожих алгоритмов и помещает их в отдельный класс.

Структура:

Полностью идентичен паттерну Состояние.

**Отличие Состояния и Стратегии:**

Переключение алгоритмов: Стратегия - переходы между стратегиями определяет только клиентский класс (различные реализации стратегий не в курсе друг о друге). Состояние – переключения между состояниями (изменение состояния) могут осуществляться как в классе Контексте, так и в классах Состояниях (или в клиенте) (состояния знают друг о друге и могут инициировать переходы).

Назначение классов, реализующих алгоритмы: Стратегия – стратегии служат одной цели, то есть предоставляют варианты решений одной и той же задачи. Состояние – классы состояния (их поведение) может быть не похоже между собой. Состояние – нацелено на управление состоянием, стратегия – поведением. Также поведение стратегии полностью выполняет работу, Стратегия делегирует только отличающуюся часть работы.

\*Предпочтительным вариантом реализации стратегии и состояния является реализация с помощью вложенных классов\*

**Template method (Шаблонный метод) –** определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги (отличающиеся) на подклассы. Позволяет переопределять некоторые шаги алгоритма, не изменяя его общей структуры.

Структура:

Абстрактный класс – содержит шаблонный метод, который определяет шаги выполнения алгоритма и методы-шаги, реализация которых может отличаться в различных сценариях. Конкретный класс – переопределяет поведение отличающихся шагов.

**Visitor (Посетитель)** – позволяет добавлять новые операции, не изменяя классы, над которыми они могут выполняться.

Структура:

Элемент – иерархия, которую обслуживает посетитель, его интерфейс должен иметь метод принятия посетителя. Посетитель – объявляет набор методов для каждого типа входящего параметра (реализаций иерархии), его реализации определяют поведение для каждого типа таких элементов (класс объединяет в себе логику общего типа, а методы реализацию для каждого подтипа) – поведение выносится из класса.

**Interpretator (Интерпретатор)** – определяет способ интерпретации выражений языка с помощью создания правил для этого языка и интерпретатора, который понимает эту грамматику. (вряд ли пригодится).

Материалы: [1](https://refactoring.guru/ru/design-patterns), видеоматериалы: [1](https://www.youtube.com/playlist?list=PLwcDaxeEINactCC4mly7RQon5juIpH-Q3), [2](https://www.youtube.com/playlist?list=PLCnuT5ArKhsk8su85Z752tyGWL8-r-tRR).

**Не GoF Паттерны:**

**Delegate (Делегат) –** делегирование объектом выполнение своих задач путем ассоциации с этим объектом и вызовом его методов при необходимости делегирования.

**Null Object** **(Нулевой объект)** – Паттерн определяет специальный объект, который не выполняет никаких действий при вызове его методов и служит «мягкой» заменой null (не нужно делать проверки на null и помогает избежать NullPionterException) – чем-то схож с Optional (но он говорит о вероятности отсутствия значения).

**Object pool (Объектный пул) –** содержит в себе набор объектов, готовых к использованию, если объект нужен, то он не создается, а берется из пула, как только он перестает быть нужным, то он не уничтожается, а возвращается в пул.

**DTO (Data transfer object) –** используется для передачи данных между частями приложения. По сути является структурой данных и не должен содержать бизнес-логику или поведения (помимо ответственности хранения данных может иметь методы доступа к этим данным, но это чаще всего не нужно).

*Условные обозначения для UML-диаграммы.*

