**Паттерн –** набор принципов построения программного кода для решения часто встречающихся проблем проектирования.

Плюсы:

* Не нужно изобретать велосипед. Есть готовый шаблон, который можно применить для решения конкретной проблемы.
* Паттерны не зависят от языка программирования (как правило).
* Упрощает командную работу. Другой программист сможет лучше понять код, если он работает по общепринятым шаблонам.

Не всегда паттерны улучшают программу. Иногда их использование может привести к неоправданному усложнению кода.

**ПОРОЖДАЮЩИЕ ПАТТЕРНЫ**

Порождающие паттерны отвечают за создание объектов и позволяют системе быть независимой от типов этих объектов и от процесса порождения.

**Фабричный метод –** паттерн, который определяет общий интерфейс для создания объектов в суперклассе. Наследники решают, объект какого типа создавать. Базовый класс делегирует создание объектов классам-наследникам.

Для реализации в базовом классе описываем фабричный метод, который будет возвращать какой-то базовый тип (например транспорт). Наследники базового класса будут переопределять фабричный метод, и возвращать какой-то конкретный вид транспорта (автомобиль, корабль, самолет).

*ПРИМЕНИМОСТЬ*:

* Когда заранее неизвестно, объекты каких типов необходимо создавать.
* Когда система должна быть независимой от процесса создания новых объектов и расширяемой: в нее можно легко вводить новые классы, объекты которых система должна создавать.
* Когда создание новых объектов необходимо делегировать из базового класса классам наследникам.

**Абстрактная фабрика –** паттерн, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.

Для реализации сначала выделяем общие интерфейсы для отдельных «продуктов» (стол, стул, диван). Далее создаем абстрактную фабрику – общий интерфейс, который содержит методы создания всех связанных продуктов, которые возвращают абстрактные типы продуктов (createTable, createChair, createSofa). Далее создаем наследников абстрактной фабрики, каждый из которых будет возвращать определенные типы продуктов (createModernTable, createModernChair). Клиент должен работать как с фабриками, так и с продуктами через интерфейсы.

*ПРИМЕНИМОСТЬ*

* Когда система не должна зависеть от способа создания и компоновки новых объектов.
* Когда создаваемые объекты должны использоваться вместе и являются взаимозаменяемыми.

**Строитель –** паттерн, который инкапсулирует создание объекта и позволяет разделить его на различные этапы. Конструирование объекта выносится в отдельный класс, называемый строителем.

В строителе конструирование разбито на отдельные шаги, которые можно вызывать поочередно либо из клиентского кода, либо из директора. Можно создать различных строителей, которые могут выполнять одни и те же шаги по-разному.

Если есть несколько способов конструирования продукта, шаги можно вынести в класс директора. Обычно Клиент подаёт в конструктор директора уже готовый объект-строитель, и в дальнейшем данный директор использует только его. Можно передавать строителя через параметр строительного метода директора. В этом случае можно каждый раз применять разных строителей для производства различных представлений объектов.

*ПРИМЕНИМОСТЬ*

* Когда нужно обеспечить получение различных вариаций объекта.
* Когда хотите избавиться от конструктора с большим количеством параметров.
* Когда процесс создания нового объекта не должен зависеть от того, из каких частей этот объект состоит и как эти части связаны между собой.

**Прототип** – паттерн, который позволяет создавать объекты на основе уже ранее созданных объектов-прототипов, клонируя их. Позволяет копировать объекты, не вдаваясь в их реализацию.

Клонированием объекта занимается сам объект. Он создает новый объект такого же класса, и копирует в него поля.

*ПРИМЕНИМОСТЬ*

* Когда не нужно привязываться к классам клонируемых объектов.
* Когда клонирование объекта более предпочтительно, нежели его создание и инициализация через конструктор. Особенно когда известно, что объект может принимать небольшое ограниченное число состояний.
* Когда нежелательно создание отдельной иерархии классов фабрик для создания объектов-продуктов.

**Singlton (Одиночка) –** паттерн, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа. Экземпляр создается только при необходимости.

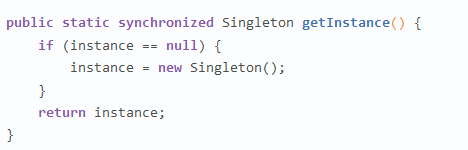
Для реализации мы делаем конструктор приватным. Доступ к объекту получаем через специальный публичный метод. Внутри метода прописываем условие: если объекта не существует, создаем его и возвращаем; иначе – возвращаем уже существующий объект.

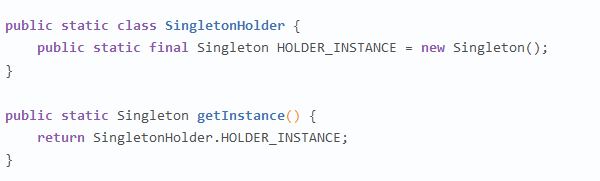
Имеет проблемы с многопоточностью, например, когда несколько потоков пытаются инициализировать объект (Решается синхронизацией, либо блокировкой с двойной проверкой).

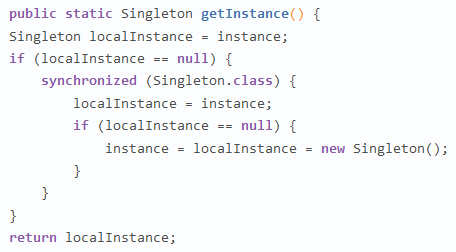
*ПРИМЕНИМОСТЬ*

* Когда нужно, чтобы в программе был только один экземпляр данного класса и имелся простой доступ к нему.

Есть несколько реализаций:

* Неленивый синглтон. Экземпляр инициализируется сразу при объявлении.
* EnumSinglton. Создать Enum с одним полем экземпляра.
* Полный синхронизированный доступ.  
  
* Через вложенный статический класс.



* Double Checked Locking & volatile  
  

**СТРУКТУРНЫЕ ПАТТЕРНЫ**

Паттерны, в которых рассматривается вопрос о том, как из классов и объектов образуются более крупные структуры (как удобно организовывать связи между объектами).

**Адаптер –** предназначен для преобразования интерфейса одного класса в интерфейс другого. Позволяет использовать вместе классы с несовместимыми интерфейсами.

Реализовать можно адаптер объектов: Реализует интерфейс клиента и содержит ссылку на объект сервиса (объект у которого несовместимый интерфейс). Адаптер получает вызовы от клиента, и приводит их к формату, понятному сервису. Работая с адаптером через интерфейс, мы можем добавлять в программу новые адаптеры.

Адаптер классов: можно реализовать только в языке, поддерживающем множественное наследование. Наследуемся и от класса клиента, и от сервиса. Таким образом у нас в одном классе будут методы обоих классов.

*ПРИМЕНИМОСТЬ*

* Когда нужно использовать класс, интерфейс которого несовместим с приложением.
* Когда нужно расширить функциональность наследников каким-то одинаковым методом, при этом нельзя трогать суперкласс. Недостающую функциональность можно вынести в Адаптер, и приспособить его для работы с суперклассом.

**Мост –** паттерн, который позволяет отделить абстракцию от реализации таким образом, чтобы и абстракцию, и реализацию можно было изменять независимо друг от друга. Абстракция – это образный слой управления чем-либо (не связано с абстрактными классами и интерфейсами). Он не делает работу самостоятельно, а делегирует ее слою реализации.

Для реализации описываем абстракцию, которая содержит управляющую логику и делегирует реальную работу реализации. Описываем интерфейс реализации, и наследуем от него конкретные реализации.

Интерфейсы абстракции и реализации могут как совпадать, так и быть совершенно разными. Но обычно в реализации живут базовые операции, на которых строятся сложные операции абстракции. Можно описать расширенные абстракции, которые содержат различные вариации управляющей логики.

Клиент работает только с объектами абстракций, в которые предварительно нужно передать объект реализации.

*ПРИМЕНИМОСТЬ*

* Когда нужно изменять абстракцию и реализацию независимо друг от друга.
* Когда хотите разделить монолитный класс, который содержит несколько различных реализаций какой-то функциональности.
* Когда нужно расширять класс в двух независимых плоскостях.

**Компоновщик –** паттерн, который позволяет сгруппировать множество объектов в древовидную структуру, и затем работать одинаково как с отдельными объектами, так и с группой объектов.

Объявляем общий интерфейс для простых и составных компонентов дерева. От него наследуем лист, не имеющий потомков, и контейнер, который будет содержать набор элементов (это могут быть как листья, так и контейнеры, интерфейс у них один и тот же).

*ПРИМЕНИМОСТЬ*

* Когда нужно построить древовидную структуру объектов.
* Когда клиенты должны единообразно трактовать простые и составные объекты.

**Декоратор –** паттерн, позволяющий динамически добавлять объектам новую функциональность, оборачивая их в обертки (агрегация вместо наследования).

И сами декораторы, и оборачиваемые объекты должны иметь общий интерфейс, чтобы их можно было взаимозаменять. Декоратор содержит ссылку на оборачиваемый объект. Декоратор делегирует работу вложенному объекту, при этом добавляет некоторое свое поведение. Один объект можно обернуть в несколько декораторов, комбинируя различные варианты поведения объекта.

*ПРИМЕНИМОСТЬ*

* Когда вам нужно динамически добавлять функционал объектам во время выполнения программы.
* Когда нет возможности расширить класс с помощью наследования.

**Фасад –** паттерн, который позволяет скрыть сложности системы, с помощью предоставления упрощенного интерфейса для взаимодействия с ней.

Фасад переадресовывает вызовы клиента нужным объектам сложной подсистемы. Если фасад начнет раздуваться, можно вводить дополнительные фасады.

*ПРИМЕНИМОСТЬ*

* Когда нужно предоставить клиенту простой доступ к сложной системе. Это также позволит менять функциональность, изменяя только код фасада.
* Фасад можно применять для разбиения системы на слои, повышая независимость слоев друг от друга и упрощая взаимодействие между слоями.

**Приспособленец (Легковес)** – структурный шаблон, позволяющий оптимизировать расход памяти, вынося общее состояние объектов в отдельный объект, и ссылаясь на него.

При создании приспособленца, внешнее (изменяемое) состояние выносится. Приспособленец хранит только общее состояние. Изменяемые поля передаются в методы легковесов в качестве параметров.

Объекты “контексты” хранят изменяемое, уникальное состояние и ссылаются на легковесы.

Обычно создается фабрика легковесов. Клиент запрашивает у фабрики объект с определенным внутренним состоянием. Фабрика хранит объекты в каком-нибудь пуле (коллекции). Если объект с таким состоянием существует в пуле, он возвращается. Если нет, то фабрика создает его.

*ПРИМЕНИМОСТЬ*

* Когда приложение использует большое количество однообразных объектов.
* Когда часть состояния объекта, которое не изменяется можно вынести во вне.

**Заместитель (Прокси)** – паттерн, предоставляющий объект-заместитель, который управляет доступом к другому объекту.

Сам объект и прокси имеют общий интерфейс. Прокси содержит ссылку на сервисный объект, и может управлять его жизненным циклом (создавать, удалять). Прокси также делегирует работу сервисному объекту, если это необходимо (например, если данных нет в кэше, берем из бд).

*ПРИМЕНИМОСТЬ*

* Если нужна ленивая инициализация. Если есть тяжелый объект (делающий очень трудоёмкую задачу), можно сэкономить ресурсы и создать объект тогда, когда он действительно понадобится.
* С помощью прокси можно реализовать проверку прав доступа, и передавать выполнение служебному объекту, если доступ разрешен.
* Прокси может логировать обращения к объекту.
* Кэширование объектов. Прокси может хранить в своем буфере данные, чтобы не брать их из бд.
* Удаленный прокси – имитирует поведение объекта, когда настоящий сервисный объект находитсяна удаленном сервере.
* Когда нужно вести подсчет ссылок на объект, и освобождать объект, если он никем не используется.

**ПОВЕДЕНЧЕСКИЕ ПАТТЕРНЫ**

Паттерны, определяющие алгоритмы и способы реализации взаимодействия различных объектов и классов. Они отвечают на вопрос «Как организовать поведение программного компонента и его общение с другими компонентами?»

**Цепочка обязанностей** – шаблон, позволяющий избежать жесткой привязки отправителя запроса к получателю. Все обработчики запроса образуют цепочку. Запрос перемещается по цепочке, каждый последующий обработчик решает, может ли он сам обработать запрос и стоит ли передавать запрос дальше по цепи.

Цепочка может обрываться, если обработчик не смог обработать запрос, либо наоборот, если найден обработчик который сможет обработать запрос.

Все обработчики имеют общий интерфейс. Обработчик может содержать ссылку на следующий обработчик. Клиент формирует цепочку обработчиков, и может перестраивать ее динамически. Запрос может отправляться любому объекту цепочки, не обязательно первому. Существует вероятность того, что запрос дойдет до конца цепочки, и останется необработанным.

*ПРИМЕНИМОСТЬ*

* Когда имеется более одного обработчика запроса, причем они могут задаваться динамически.
* Когда нужно чтобы обработчики выполнялись один за другим в строгом порядке.
* Когда есть несколько обработчиков, и мы не знаем заранее, какой обработчик сможет обработать запрос.

**Команда –** паттерн, в котором запросы представляются как объекты. Это позволяет передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.

В реализации имеется отправитель, который взаимодействует с командами через интерфейс. Обычно команда имеет метод запуска, хранит ссылку на объект получателя и делегирует работу ему. Команда так же может хранить список параметров для вызова метода получателя.

Таким образом отправители и получатели не будут зависеть друг от друга, так как отправитель не знает какую команду и с каким получателем мы ему передаем.

*ПРИМЕНИМОСТЬ*

* Когда одни и те же действия могут вызываться из различных частей программы.
* Когда вы хотите ставить операции в очередь, выполнять их по расписанию, или передавать по сети.
* Когда вам нужно отменять операции.

**Итератор** – паттерн, позволяющий последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.

Итератор реализует алгоритм обхода коллекции. Объект итератора должен сам отслеживать свою позицию при обходе коллекции, чтобы разные итераторы могли обходить одну и ту же коллекцию независимо.

Обычно коллекция сама создает итератор, и связывает его с текущим объектом коллекции, так как она знает, какие именно итераторы способны с ней работать.

Клиентский код перебирает объекты коллекции с помощью итератора.

*ПРИМЕНИМОСТЬ*

* Когда нужно осуществить обход объекта без раскрытия его внутренней структуры.
* Когда нужно иметь несколько вариантов обхода одной и той же структуры. Их можно вынести в отдельный класс, чтобы не захламлять коллекцию.
* Когда нужно иметь единый интерфейс обхода различных структур данных.

**Посредник (Медиатор)** – паттерн, который позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.

Обычно посредник содержит один метод, для оповещения о каком-либо событии. Он также содержит ссылки на объекты связанных компонентов. Компоненты должны иметь ссылку на посредника, чтобы уведомлять его о событиях. Посредника удобнее всего передавать в конструктор компонента.

Класс посредник знает, кому нужно перенаправить тот или иной запрос. Компонент-отправитель не знает, кто обработает запрос, а компонент-получатель не знает кто его прислал.

Благодаря этому, компоненты системы будут зависеть только от посредника, а не от кучи других компонентов.

*ПРИМЕНИМОСТЬ*

* Когда имеется множество взаимосвязанных объектов, связи между которыми сложны и запутаны.
* Когда вы не можете повторно использовать класс, поскольку он зависит от уймы других классов.

**Снимок (Momento)** – паттерн, позволяющий выносить внутреннее состояние объекта за его пределы для последующего возможного восстановления объекта без нарушения принципа инкапсуляции.

Для сохранения состояния, нужно копировать поля объекта. Если делать это извне, то поля нужно делать публичными, что не есть хорошо (мы привяжемся к структуре класса, и при любых изменениях в его полях придется менять и код копирования состояния). С другой стороны, приватные поля тоже невозможно копировать извне.

Паттерн снимок поручает создание копии самому объекту, у которого есть доступ к своим приватным полям.

В языках, поддерживающих вложенные классы, класс снимка объявляют внутри класса, состояние которого нужно хранить. Таким образом ему будут доступны все поля внешнего класса. Снимки хранятся в специальном классе хранителе (опекуне). Он не имеет доступа к состоянию снимков, и занимается только хранением.

Если язык не поддерживает вложенные классы: То создаем общий интерфейс без методов. Опекун будет взаимодействовать со снимком через общий интерфейс, а класс которому нужно хранить состояние с конкретной реализацией снимка.

*ПРИМЕНИМОСТЬ*

* Когда нужно сохранять состояние объекта для возможного последующего восстановления.
* Когда сохранение состояния должно проходить без нарушения принципа инкапсуляции.

**Наблюдатель** – паттерн, который создает механизм подписки, позволяющий объектам следить и реагировать на события, происходящие в других объектах.

В рамках паттерна объекты делятся на издателей (объект который производит событие) и подписчиков (объекты, которые ждут события от издателя и как-то реагируют на него).

Издатель хранит у себя список подписчиков. Подписчики сами подписываются на события издателя с помощью специальных методов. В итоге, при наступлении события издатель пройдется по списку подписчиков и вызовет у них метод оповещения.

Если подписчику нужно получить состояние издателя, можно передать ссылку на объект издателя в методе оповещения, либо хранить ссылку на этот объект в объекте подписчика.

*ПРИМЕНИМОСТЬ*

* Когда после изменения состояния одного объекта требуется что-то делать в других объектах, но неизвестно наперед, какие именно объекты должны отреагировать.
* Когда нужно динамически изменять подписчиков.

**Состояние –** паттерн, который позволяет объектам менять поведение в зависимости от своего состояния.

Для каждого состояния, в котором может находится объект, создается отдельный класс. В этих классах описывается и поведение, соответствующее этим состояниям.

Сам объект будет делегировать работу объекту-состоянию. В данном паттерне, в отличии от «Стратегии», контекст и сами конкретные состояния могут знать друг о друге и инициировать переходы от одного состояния к другому.

*ПРИМЕНИМОСТЬ*

* Когда поведение объекта должно зависеть от его состояния и может изменяться динамически во время выполнения.
* Когда в коде методов объекта куча if-ов, выбор которых зависит от текущего состояния объекта.
* Когда код при некоторых состояниях дублируется. Общий код состояний можно описать в базовом классе состояний.

**Стратегия –** паттерн, который определяет набор алгоритмов, инкапсулирует их и позволяет взаимозаменять в зависимости от ситуации.

Создается класс контекст, который будет содержать ссылку на стратегию, и делегировать ей работу. Чтобы сменить алгоритм, достаточно передать в объект другой объект стратегию. Стратегии должны иметь общий интерфейс.

* Когда нужно использовать разные вариации какого-то алгоритма внутри одного объекта.
* Когда нужно выбирать алгоритм динамически, в зависимости от условий.
* Когда нужно менять алгоритмы, не меняя интерфейс.

**Шаблонный метод –** паттерн, в котором какой-то алгоритм разбивается на отдельные шаги. Подклассы затем могут переопределять некоторые шаги алгоритма, не меняя его общей структуры.

Эти шаги будут вызываться в так называемом шаблонном методе. Можно сделать все шаги абстрактными, либо добавить какую-то реализацию по умолчанию. Наследники переопределяют нужные шаги, но общая структура алгоритма не меняется.

*ПРИМЕНИМОСТЬ*

* Когда подклассы должны расширять базовый алгоритм, не меняя его структуры.
* Когда в нескольких классах есть схожие по структуре алгоритмы, с незначительными отличиями.

**Посетитель –** паттерн, который позволяет добавлять в программу новые операции, не изменяя классы, над которыми эти операции могут выполняться.

Используется если над разными типами классов, нужно выполнять одно и то же действие, причем реализация этих действий немного отличается (например экспорт объектов в xml).

Эти действия выносятся в отдельный класс – Посетитель. Для каждого типа объекта объявляется свой метод (если язык поддерживает перегрузку методов, можно назвать их одинаково, но они будут принимать разные типы параметров).

Элемент, над которым выполняется действие, должен иметь метод для принятия посетителя. В этом методе вызывается нужный метод посетителя, и передается ссылка на объект, который его вызывает.

Таким образом, создавая новых посетителей, можно будет выполнять над объектами различные действия, не меняя сами объекты.

*ПРИМЕНИМОСТЬ*

* Когда часто добавляются новые операции над группой элементов.
* Когда имеется много объектов разнородных классов с разными интерфейсами, и требуется выполнить операцию над каждым из этих объектов.
* Когда классам нужно добавить одинаковый набор операций, без изменений этих классов.
* Когда новое поведение имеет смысл только для некоторых классов из существующей иерархии.

**СРАВНЕНИЕ НЕКОТОРЫХ ПАТТЕРНОВ**

**Адаптер vs Фасад**

Адаптер позволяет работать вместе объектам с несовместимыми интерфейсами. Фасад предоставляет упрощенный доступ к сложной системе.

Фасад задает новый (упрощенный интерфейс), тогда как Адаптер позволяет двум существующим интерфейсам работать сообща, вместо того чтобы задавать новый.

Адаптер оборачивает только один класс, а фасад целую подсистему.

**Адаптер vs Мост**

Мост разделяет абстракцию и реализацию, позволяя изменять их независимо. Мост проектируется заранее, чтобы развивать отдельные части приложения отдельно друг от друга.

Адаптер применяется постфактум, когда нужно заставить несовместимые классы работать вместе.

**Фасад vs Прокси**

Прокси имеет тот же интерфейс, что и его служебный объект, благодаря чему они становятся взаимозаменяемы. Фасад же предоставляет альтернативный интерфейс, упрощающий доступ к сложной системе.

**Наблюдатель vs Цепочка обязанностей**

Цепочка обязанностей передает запрос последовательно через цепочку потенциальных получателей, ожидая, что какой-то из них обработает запрос.

Наблюдатель передает запрос сразу всем заинтересованным получателям, которые подписались на оповещения.

**Декоратор vs Прокси**

Они очень похожи. Оба построены на принципе композиции и делегируют работу другим объектам.  
Прокси сам управляет жизнью сервисного объекта (создает его когда нужно. Если объект больше не нужен, может освобождать ресурсы).

Обертывание декораторов контролируется клиентом – клиент создает объект, и передает его в декоратор. Декоратор может также расширять интерфейс.

**Фабричный метод vs Builder**

Фабричный метод используется когда не требуется сложной инициализации объекта. Он построен на наследовании, и в наследниках будут переопределяться типы создаваемых объектов. Используется для создания различных типов объектов из одной иерархии.

Строитель концентрируется на создании различных представлений одного и того же объекта. Разные строители могут выполнять шаги по-разному. Кроме того, строители из одной иерархии могут производить разнородные объекты, не имеющие общего интерфейса.