**Порождающие паттерны:**

**Abstract Factory** (абстрактная фабрика)

Предоставляет интерфейс для создания групп связанных или зависимых обьектов, не указывая их конкретный класс.

**Builder** (строитель)

Разделяет создание сложного обьекта и инициализацию его состояния так, что одинаковый процесс построения может создать обьекты с разным состоянием.

**FactoryMethod** (Фабричный метод)(Virtual Constructor)

Определяет интерфейс для создания обьекта, но позволяет подклассам решать, какой класс инстанцировать. Позволяет делегировать создание обьекта подклассам.

**Prototype** (прототип)

Определяет несколько видов обьектов, чтобы при создании использовать обьект-прототип и создает новые обьекты, копирую прототип.

**Singleton** (одиночка)

Гарантирует, что класс имеет только один экземпрял и предоставляет глобальную точку доступа к нему.

**Структурные паттерны:**

* **Adapter** (адаптер)

Конвертирует интерфейс класса в другой интерфейс, ожидаемый клиентом. Позволяет объектам с несовместимыми интерфейсами работать вместе. Разрешает проблему, когда мы не можем изменить исходный класс, адаптер является оберткой над нужным классом и наследует необходимый интерфейс. Адаптер оборачивает существующий класс новым интерфейсом, чтобы он стал совместимым с необходимым интерфейсом. Возможно решение через наследование исходного класса и создание инстанса класса внутри нового.

* **Bridge** (мост)

Мост - структурный паттерн, разделяющий бизнеслогику или большой класс на несколько отдельных иерархий, которые можно развивать отдельно друг от друга. Одна из этих иерархий (абстракция) получит ссылку на объекты другой иерархии (реализация) и будет делегировать им основную работу. Благодаря тому, что все реализации будут соблюдать общий интерфейс, их можно будет взаимозаменять внутри абстракции. Если в программе четко выделены классы «управления» и несколько видов классов «платформ», а управляющие объекты делегируют выполнение платформам, тогда можно сказать, что вы используете Мост.

* **Composite** (компоновщик)

Компонует обьекы в древовидную структуру, представляя их в виде иерархии. Позволяет клиенту одинаково обращатся как к отдельному обьекту, так и к целому поддереву. Компоновщик — это структурный паттерн проектирования, позволяющий сгруппировать несколько объектов в древовидную структуру, а затем работать с ней так, будто это единичный объект.

* **Decorator** (декоратор)

Динамически предоставляет обьекту дополнительные возможности. Представляет собой гибкую альтернативу наследованию для расширения функционала.

* **Facade** (фасад)

Представляет единый интерфейс к группе интерфейсов подсистемы. Определяет высокоуровненвый интерфейс, делая подсистему проще для использования. Позволяет скрыть сложность системы путём сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы. Создает единий интерфейс, которий после обращения к нему делигирует работу ниже стоящим обьектам.

* **Flyweight** (приспособленец, легковесный)

Благодаря совместному использованию, поддерживает эффективную работу с большим количеством обьектов. Объект, представляющий себя как уникальный экземпляр в разных местах программы, по факту не является таковым, он содержит общие неизменяемые параметры (внутренее состояние), в то время как изменяемые параметры (внешнее состояние) хранится отдельно, вынесено во внешний клас-контекст. Это позволяет поднять произодительность. При визове создания нового обьекта, ми сперва проверяем, есть ли у нас уже такой обьект с такими характеристиками, если есть - возвращаем его, если нет – создаем новий обьект и кешируем его!!!

* **Proxy** (заместитель)

Предоставляет замену другого обьекта для контроля доступа к нему.

Проблема: Необходимо контролировать доступ к объекту, не изменяя при этом поведение клиента. Необходимо иметь доступ к объекту так, чтобы не создавать реальные объекты непосредственно, а через другой объект, который может иметь дополнительную функциональность.

Решение: Создать суррогат реального объекта. «Заместитель» хранит ссылку, которая позволяет заместителю обратиться к реальному субъекту (объект класса «Заместитель» может обращаться к объекту класса «Субъект», если интерфейсы «Реального Субъекта» и «Субъекта» одинаковы). Поскольку интерфейс «Реального Субъекта» идентичен интерфейсу «Субъекта», так, что «Заместителя» можно подставить вместо «Реального Субъекта», контролирует доступ к «Реальному Субъекту», может отвечать за создание или удаление «Реального Субъекта». «Субъект» определяет общий для «Реального Субъекта» и «Заместителя» интерфейс так, что «Заместитель» может быть использован везде, где ожидается «Реальный Субъект». При необходимости запросы могут быть переадресованы «Заместителем» «Реальному Субъекту».

Виды:

* Протоколирующий прокси
* Удалённый заместитель
* Виртуальный заместитель
* Копирующий-при-записи
* Защищающий заместитель
* Кэширующий прокси
* Экранирующий прокси
* "Умная" ссылка

|  |  |
| --- | --- |
| Wrapper: | Behavior: |
| Адаптер | Позволяет объектам с несовместимыми интерфейсами работать вместе, используются, когда у нас есть класс (Клиент), ожидающий некоторого типа объекта, и у нас есть объект (Адаптируемый), предлагающий те же функции, но предоставляющий другой интерфейс.  Клиент делает запрос к адаптеру, вызывая на нем метод с помощью целевого интерфейса.  Адаптер переводит этот запрос на адаптируемое устройство с помощью интерфейса адаптируемого устройства.  Клиент получает результаты звонка и не знает о присутствии адаптера. |
| Прокси | Предоставляет тот же интерфейс, что и объект, на который он ссылается, и никоим образом не изменяет данные. |
| Декоратор | Обеспечивает расширенный интерфейс, добавляет новый функционал |

**Паттерны поведения:**

**Chain of Responsibility** (цепочка ответственности)

Избегает связывания отправителя запроса с его получателем, давая возможность обработать запрос более чем одному обьекту. Связывает обьекты-получатели и передает запрос по цепочке пока обьект не обработает его.

**Command** (команда)

Инкапсулирует запрос в виде обьекта, позволяя передавать их клиентам в качестве параметра, ставить в очередь, логировать а также поддерживает отмену операций.

**Interpreter** (интерпретатор)

Получая формальный язык, определяет представление его грамматики и интерпретатор, использующий это представление для обработки выражений языка.

**Iterator** (итератор)

Предоставляет способ последовательного доступа к элементам множества, независимо от его внутреннего устройства.

**Mediator** (посредник)

Определяет обьект, инкапсулирующий способ взаимодействия обьектов. Обеспечивает слабую связь, избавляя обьекты от необходимости прямо ссылать друг на друга и дает возможность независимо изменять их взаимодействие.

**Memento** (хранитель)

Не нарушая инкапсуляцию, определяет и сохраняет внутреннее состояние обьекта и позволяет позже восстановить обьект в томже состоянии(хеширование?).

**Observer** (наблюдатель)

Определяет зависимость “один ко многим” между объектами так, что когда один обьект меняет свое состояние, все зависимие оьекты оповещаются и обновляются автоматичекси.

**State** (состояние)

Позволяет обьекту изменять свое поведение в зависимости от внутреннего состояния.

**Strategy** (стратегия)

Определяет группу алгоритмов, инкапсулирует их и делает взаимозаменяемыми. Позволяет изменять алгоритм независимо от клиентов, его использующих.

**Template Method** (шаблонный метод)

Определяет алгоритм, некоторые этапы которого делегируются подклассам. Позволяет подклассам переопределить эти этапы, не меняя структуру алгоритма.

**Visitor** (посетитель)

Представляет собой операцию, которая будет выполнена над обьектами группы классов. Дает возможность определить новую операцию без изменения кода классов, над которыми эта операция проводится.

**OOP:**

* **Encapsulation** – это метод упрощения работы со сложной системой для конечных пользователей. Пользователю не нужно беспокоиться о внутренних деталях и сложностях системы. Инкапсуляция – это процесс объединения данных и кода, который работает с данными, в единую сущность. Вы можете считать его защитной оболочкой, которая останавливает произвольный доступ к коду, определенному вне этой оболочки.
* **Inheritance** – механизм создания одного класса с помощью другого с использованием всех свойств и методов суперкласса добавляя новые свойства и методы при необходимости, отображает иерархичность окружающего мира.
* **Polymorphism** – механизм, позволяющий подклассам изменять реализацию методов суперкласса, сохраняя его сигнатуру, благодаря чему интерфейс суперкласса остается неизменяемым, упрощает изменение модулей. Cпособность программы идентично использовать объекты с одинаковым интерфейсом без информации о конкретном типе этого объекта. Dynamic method dispatch(динамическая диспетчеризация методов). **Covariance and Contravariance.**
* **Abstraction** – направлена на борьбу со сложностью, скрывая от пользователя ненужные детали. Это позволяет пользователю реализовать более сложную логику поверх предоставленной абстракции, не понимая и даже не задумываясь обо всей скрытой сложности. Абстракция позволяет «показывать» только существенные атрибуты и «скрывать» ненужную информацию. Основная цель абстракции — скрыть от пользователей ненужные детали. Абстракция — это выбор данных из большего пула, чтобы показать пользователю только важные сведения об объекте. Это помогает уменьшить сложность и усилия программирования.

|  |  |  |
| --- | --- | --- |
| № | ABSTRACTION | ENCAPSULATION |
| 1 | Процесс или метод получения информации. | Процесс или метод содержания информации. |
| 2 | Проблемы решаются на уровне дизайна или интерфейса. | Проблемы решаются на уровне реализации. |
| 3 | Метод сокрытия нежелательной информации. | Метод сокрытия данных в одном объекте или единице, а также метод защиты информации извне. |
| 4 | Испоьзуем абстрактный класс и интерфейсы. | Испоьзуем модификаторы доступа. |
| 5 | В абстракции сложности реализации скрыты с помощью абстрактных классов и интерфейсов. | В инкапсуляции данные скрываются с помощью методов геттеров и сеттеров. |
| 6 | Обобщение. | Сокрытие. |

**SOLID:**

* **S - Single-responsiblity** (Принцип единственной ответственности)

*“Каждый класс должен отвечать только за одну операцию.”* Если класс отвечает за несколько операций сразу, вероятность возникновения багов возрастает. Внося изменения, касающиеся одной из операций вы, сами того не подозревая, можете затронуть и другие. Принцип служит для разделения типов поведения, благодаря которому ошибки, вызванные модификациями в одном поведении, не распространялись на прочие, не связанные с ним типы.

* **O - Open-closed** (Принцип открытости-закрытости)

*“Классы должны  быть  открыты для расширения, но закрыты для модификации*.” Когда вы меняете текущее поведение класса, эти изменения сказываются на всех системах, работающих с данным классом. Если хотите, чтобы класс выполнял больше операций, то идеальный вариант – не заменять старые на новые, а добавлять новые к уже существующим. Принцип служит для того, чтобы делать поведение класса более разнообразным, не вмешиваясь в текущие операции, которые он выполняет. Благодаря этому вы избегаете ошибок в тех фрагментах кода, где задействован этот класс.

* **L - Liskov Substitution** (Принцип подстановки Барбары Лисков)

*“Если П является подтипом Т, то любые объекты типа Т, присутствующие в программе, могут заменяться объектами типа П без негативных последствий для функциональности программы.”* В случаях, когда класс-потомок не способен выполнять те же действия, что и класс-родитель, возникает риск появления ошибок. Если у вас имеется класс и вы создаете на его базе другой класс, исходный класс становится родителем, а новый – его потомком. Класс-потомок должен производить такие же операции, как и класс-родитель. Это называется наследственностью. Необходимо, чтобы класс-потомок был способен обрабатывать те же запросы, что и родитель, и выдавать тот же результат. Или же результат может отличаться, но при этом относиться к тому же типу.

* **I - Interface Segregation** (Принцип разделения интерфейсов)

*“Не следует ставить клиент в зависимость от методов, которые он не использует.”* Когда классу приходится производить действия, не несущие никакой реальной пользы, это выливается в пустую трату ресурса, а в случае, если класс выполнять эти действия не способен, ведёт к возникновению багов. Класс должен производить только те операции, которые необходимы для осуществления его функций. Все другие действия следует либо удалить совсем, либо переместить, если есть вероятность, что они понадобятся другому классу в будущем. Принцип служит для того, чтобы раздробить единый набор действий на ряд наборов поменьше – таким образом, каждый класс делает то, что от него действительно требуется, и ничего больше.

* **D - Dependency Inversion** (Принцип инверсии зависимостей)

“Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те, и другие должны зависеть от абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.” Согласно данному принципу, класс не должен соединяться с инструментом, который применяет для выполнения операции. Вместо этого он должен быть соединён с интерфейсом, который поможет установить связь между инструментом и классом. Кроме того, принцип гласит, что ни интерфейс, ни класс, не обязаны вникать в специфику работы инструмента. Напротив, это инструмент должен подходить под требования интерфейса. Этот принцип служит для того, чтобы устранить зависимость классов верхнего уровня от классов нижнего уровня за счёт введения интерфейсов.

**DRY (don't repeat yourself):**

Каждая часть знания должна иметь единственное, непротиворечивое и авторитетное представление в рамках системы.

**KISS (keep it simple, stupid):**

Принцип KISS утверждает, что большинство систем работают лучше всего, если они остаются простыми, а не усложняются. Поэтому в области проектирования простота должна быть одной из ключевых целей, и следует избегать ненужной сложности.

**YAGNI (you aren't gonna need it):**

Процесс и принцип проектирования ПО, при котором в качестве основной цели и/или ценности декларируется отказ от избыточной функциональности, — то есть отказ добавления функциональности, в которой нет непосредственной надобности, желание писать код, который не нужен прямо сейчас, но может понадобиться в будущем.

**DI and IoC**

**DDD (Domain-Driven Design)**

**Application Layers**

**CI/CD**