1. Назначение паттернов поведения?

**Поведенческие паттерны** определяют алгоритмы и взаимодействие между классами и объектами, то есть их поведение. Среди подобных шаблонов можно выделить следующие:

* **Цепочка обязанностей (Chain of responsibility)**
* **Команда (Command)**
* **Интерпретатор (Interpreter)**
* **Итератор (Iterator)**
* **Посредник (Mediator)**
* **Хранитель (Memento)**
* **Наблюдатель (Observer)**
* **Состояние (State)**
* **Стратегия (Strategy)**
* **Шаблонный метод (Template method)**
* **Посетитель (Visitor)**

1. Нарисуете диаграмму классов и поясните принцип работы паттерна Chain of responsibility. В каких случаях надо его применять?

Цепочка Обязанностей (Chain of responsibility) - поведенческий шаблон проектирования, который позволяет избежать жесткой привязки отправителя запроса к получателю, позволяя нескольким объектам обработать запрос. Все возможные обработчики запроса образуют цепочку, а сам запрос перемещается по этой цепочке, пока один из ее объектов не обработает запрос. Каждый объект при получении запроса выбирает, либо обработать запрос, либо передать выполнение запроса следующему по цепочке.

**Когда применяется цепочка обязанностей?**

* Когда имеется более одного объекта, который может обработать определенный запрос
* Когда надо передать запрос на выполнение одному из нескольких объект, точно не определяя, какому именно объекту
* Когда набор объектов задается динамически

### Участники

* **Handler**: определяет интерфейс для обработки запроса. Также может определять ссылку на следующий обработчик запроса
* **ConcreteHandler1** и **ConcreteHandler2**: конкретные обработчики, которые реализуют функционал для обработки запроса. При невозможности обработки и наличия ссылки на следующий обработчик, передают запрос этому обработчику

В данном случае для простоты примера в качестве параметра передается некоторое число, и в зависимости от значения данного числа обработчики и принимают решения об обработке запроса.

* **Client**: отправляет запрос объекту Handler

1. Назначение и принцип организации паттерна Command. Поясните как он связан с конечными автоматами

Паттерн "Команда" (Command) позволяет инкапсулировать запрос на выполнение определенного действия в виде отдельного объекта. Этот объект запроса на действие и называется командой. При этом объекты, инициирующие запросы на выполнение действия, отделяются от объектов, которые выполняют это действие.

### Когда использовать команды?

Когда надо передавать в качестве параметров определенные действия, вызываемые в ответ на другие действия. То есть когда необходимы функции обратного действия в ответ на определенные действия.

Когда необходимо обеспечить выполнение очереди запросов, а также их возможную отмену.

Когда надо поддерживать логгирование изменений в результате запросов. Использование логов может помочь восстановить состояние системы - для этого необходимо будет использовать последовательность запротоколированных команд.

### Участники

* **Command**: интерфейс, представляющий команду. Обычно определяет метод Execute() для выполнения действия, а также нередко включает метод Undo(), реализация которого должна заключаться в отмене действия команды
* **ConcreteCommand**: конкретная реализация команды, реализует метод Execute(), в котором вызывается определенный метод, определенный в классе Receiver
* **Receiver**: получатель команды. Определяет действия, которые должны выполняться в результате запроса.
* **Invoker**: инициатор команды - вызывает команду для выполнения определенного запроса
* **Client**: клиент - создает команду и устанавливает ее получателя с помощью метода SetCommand()

1. Как реализовать паттерн Observer?

Паттерн "Наблюдатель" (Observer) представляет поведенческий шаблон проектирования, который использует отношение "один ко многим". В этом отношении есть один наблюдаемый объект и множество наблюдателей. И при изменении наблюдаемого объекта автоматически происходит оповещение всех наблюдателей.

### Когда использовать паттерн Наблюдатель?

* Когда система состоит из множества классов, объекты которых должны находиться в согласованных состояниях
* Когда общая схема взаимодействия объектов предполагает две стороны: одна рассылает сообщения и является главным, другая получает сообщения и реагирует на них. Отделение логики обеих сторон позволяет их рассматривать независимо и использовать отдельно друга от друга.
* Когда существует один объект, рассылающий сообщения, и множество подписчиков, которые получают сообщения. При этом точное число подписчиков заранее неизвестно и процессе работы программы может изменяться.

### Участники

* **IObservable**: представляет наблюдаемый объект. Определяет три метода: AddObserver() (для добавления наблюдателя), RemoveObserver() (удаление набюдателя) и NotifyObservers() (уведомление наблюдателей)
* **ConcreteObservable**: конкретная реализация интерфейса IObservable. Определяет коллекцию объектов наблюдателей.
* **IObserver**: представляет наблюдателя, который подписывается на все уведомления наблюдаемого объекта. Определяет метод Update(), который вызывается наблюдаемым объектом для уведомления наблюдателя.
* **ConcreteObserver**: конкретная реализация интерфейса IObserver.

1. Нарисуйте диаграмму классов для паттерна Mediator. Поясните его назначение.

Паттерн Посредник (Mediator) представляет такой шаблон проектирования, который обеспечивает взаимодействие множества объектов без необходимости ссылаться друг на друга. Тем самым достигается слабосвязанность взаимодействующих объектов.

Когда используется паттерн Посредник?

* Когда имеется множество взаимосвязаных объектов, связи между которыми сложны и запутаны.
* Когда необходимо повторно использовать объект, однако повторное использование затруднено в силу сильных связей с другими объектами.

### Участники

* **Mediator**: представляет интерфейс для взаимодействия с объектами Colleague
* **Colleague**: представляет интерфейс для взаимодействия с объектом Mediator
* **ConcreteColleague1** и **ConcreteColleague2**: конкретные классы коллег, которые обмениваются друг с другом через объект Mediator
* **ConcreteMediator**: конкретный посредник, реализующий интерфейс типа Mediator

1. В чем разница между паттернами Mediator и Facade?

Принципиальное различие между этими двумя паттернами заключается в том, что фасад, как структурный паттерн, всего лишь передает существующую функциональность в медиатор, в то время как медиатор, как поведенческий паттерн, может эту функциональность расширять.

1. В чем суть паттерна Memento? Поясните на примере.

Паттерн Хранитель (Memento) позволяет выносить внутреннее состояние объекта за его пределы для последующего возможного восстановления объекта без нарушения принципа инкапсуляции.

Когда использовать Memento?

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

То есть ключевыми понятиями для данного паттерна являются **сохранение внутреннего состояния** и **инкапсуляция**, и важно соблюсти баланс между ними. Ведь, как правило, если мы не нарушаем инкапсуляцию, то состояние объекта хранится в объекте в приватных переменных. И не всегда для доступа к этим переменным есть методы или свойства с сеттерами и геттерами. Например, в игре происходит управление героем, все состояние которого заключено в нем самом - оружие героя, показатель жизней, силы, какие-то другие показатели. И нередко может возникнуть ситуация, сохранить все эти показатели во вне, чтобы в будущем можно было откатиться к предыдущему уровню и начать игру заново. В этом случае как раз и может помочь паттерн Хранитель.

### Участники

* **Memento**: хранитель, который сохраняет состояние объекта Originator и предоставляет полный доступ только этому объекту Originator
* **Originator**: создает объект хранителя для сохранения своего состояния
* **Caretaker**: выполняет только функцию хранения объекта Memento, в то же время у него нет полного доступа к хранителю и никаких других операций над хранителем, кроме собственно сохранения, он не производит

1. Расскажите о паттерне Visitor?

Паттерн Посетитель (Visitor) позволяет определить операцию для объектов других классов без изменения этих классов.

При использовании паттерна Посетитель определяются две иерархии классов: одна для элементов, для которых надо определить новую операцию, и вторая иерархия для посетителей, описывающих данную операцию.

Когда использовать данный паттерн?

* Когда имеется много объектов разнородных классов с разными интерфейсами, и требуется выполнить ряд операций над каждым из этих объектов
* Когда классам необходимо добавить одинаковый набор операций без изменения этих классов
* Когда часто добавляются новые операции к классам, при этом общая структура классов стабильна и практически не изменяется

### Участники

* **Visitor**: интерфейс посетителя, который определяет метод Visit() для каждого объекта Element
* **ConcreteVisitor1 / ConcreteVisitor2**: конкретные классы посетителей, реализуют интерфейс, определенный в Visitor.
* **Element**: определяет метод Accept(), в котором в качестве параметра принимается объект Visitor
* **ElementA / ElementB**: конкретные элементы, которые реализуют метод Accept()
* **ObjectStructure**: некоторая структура, которая хранит объекты Element и предоставляет к ним доступ. Это могут быть и простые списки, и сложные составные структуры в виде деревьев

1. В каких случаях надо применять Null object?

Целью Null-object является инкапсулирование *отсутствия* объекта путём замещения его другим объектом, который ничего не делает.

Данный шаблон проектирования рекомендуется использовать, когда:

* Объект требует взаимодействия с другими объектами. Null Object не устанавливает нового взаимодействия — он использует уже установленное взаимодействие.
* Какие-то из взаимодействующих объектов должны бездействовать
* Требуется абстрагирование «общения» с объектами, имеющими NULL-значение.[[2]](https://ru.wikipedia.org/wiki/Null_object_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)#cite_note-2)

1. Поясните на диаграмме классов как реализовать Strategy.

Паттерн Стратегия (Strategy) представляет шаблон проектирования, который определяет набор алгоритмов, инкапсулирует каждый из них и обеспечивает их взаимозаменяемость. В зависимости от ситуации мы можем легко заменить один используемый алгоритм другим. При этом замена алгоритма происходит независимо от объекта, который использует данный алгоритм.

### Когда использовать стратегию?

* Когда есть несколько родственных классов, которые отличаются поведением. Можно задать один основной класс, а разные варианты поведения вынести в отдельные классы и при необходимости их применять
* Когда необходимо обеспечить выбор из нескольких вариантов алгоритмов, которые можно легко менять в зависимости от условий
* Когда необходимо менять поведение объектов на стадии выполнения программы
* Когда класс, применяющий определенную функциональность, ничего не должен знать о ее реализации

### Участники

Как видно из диаграммы, здесь есть следующие участники:

* Интерфейс IStrategy, который определяет метод Algorithm(). Это общий интерфейс для всех реализующих его алгоритмов. Вместо интерфейса здесь также можно было бы использовать абстрактный класс.
* Классы ConcreteStrategy1 и ConcreteStrategy, которые реализуют интерфейс IStrategy, предоставляя свою версию метода Algorithm(). Подобных классов-реализаций может быть множество.
* Класс Context хранит ссылку на объект IStrategy и связан с интерфейсом IStrategy отношением агрегации.

1. Перечислите и поясните принципы проектирования SOLID.

**Расшифровка:**

* Single responsibility — принцип единственной ответственности
* Open-closed — принцип открытости / закрытости
* Liskov substitution — принцип подстановки Барбары Лисков
* Interface segregation — принцип разделения интерфейса
* Dependency inversion — принцип инверсии зависимостей

**Принцип единственной обязанности / ответственности** (single responsibility principle / SRP) обозначает, что каждый объект должен иметь одну обязанность и эта обязанность должна быть полностью инкапсулирована в класс. Все его сервисы должны быть направлены исключительно на обеспечение этой обязанности.

**Принцип открытости / закрытости** (open-closed principle / OCP) декларирует, что программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения. Это означает, что эти сущности могут менять свое поведение без изменения их исходного кода.

**Принцип подстановки Барбары Лисков** (Liskov substitution principle / LSP) в формулировке Роберта Мартина: «функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа не зная об этом».

**Принцип разделения интерфейса** (interface segregation principle / ISP) в формулировке Роберта Мартина: «клиенты не должны зависеть от методов, которые они не используют». Принцип разделения интерфейсов говорит о том, что слишком «толстые» интерфейсы необходимо разделять на более маленькие и специфические, чтобы клиенты маленьких интерфейсов знали только о методах, которые необходимы им в работе. В итоге, при изменении метода интерфейса не должны меняться клиенты, которые этот метод не используют.

**Принцип инверсии зависимостей** (dependency inversion principle / DIP) — модули верхних уровней не должны зависеть от модулей нижних уровней, а оба типа модулей должны зависеть от абстракций; сами абстракции не должны зависеть от деталей, а вот детали должны зависеть от абстракций.