1. Назначение структурных паттернов?

**Структурные паттерны** рассматривают, как классы и объекты образуют более крупные структуры - более сложные по характеру классы и объекты. К таким шаблонам относятся:

* **Адаптер (Adapter)**
* **Мост (Bridge)**
* **Компоновщик (Composite)**
* **Декоратор (Decorator)**
* **Фасад (Facade)**
* **Приспособленец (Flyweight)**
* **Заместитель (Proxy)**

1. Нарисуете диаграмму классов и поясните принцип работы паттерна Adapter. В каких случаях надо его применять.

Паттерн Адаптер (Adapter) предназначен для преобразования интерфейса одного класса в интерфейс другого. Благодаря реализации данного паттерна мы можем использовать вместе классы с несовместимыми интерфейсами.

**Когда надо использовать Адаптер?**

* Когда необходимо использовать имеющийся класс, но его интерфейс не соответствует потребностям
* Когда надо использовать уже существующий класс совместно с другими классами, интерфейсы которых не совместимы

### Участники

* **Target**: представляет объекты, которые используются клиентом
* **Client**: использует объекты Target для реализации своих задач
* **Adaptee**: представляет адаптируемый класс, который мы хотели бы использовать у клиента вместо объектов Target
* **Adapter**: собственно адаптер, который позволяет работать с объектами Adaptee как с объектами Target.

1. В чем разница между адаптером класса и адаптером объекта?

проще переопределить поведение адаптируемого класса

легче добавить новое поведение

1. Назначение и принцип организации паттерна Декоратор.

Декоратор (Decorator) представляет структурный шаблон проектирования, который позволяет динамически подключать к объекту дополнительную функциональность.

Для определения нового функционала в классах нередко используется наследование. Декораторы же предоставляет наследованию более гибкую альтернативу, поскольку позволяют динамически в процессе выполнения определять новые возможности у объектов.

### Когда следует использовать декораторы?

Когда надо динамически добавлять к объекту новые функциональные возможности. При этом данные возможности могут быть сняты с объекта

Когда применение наследования неприемлемо. Например, если нам надо определить множество различных функциональностей и для каждой функциональности наследовать отдельный класс, то структура классов может очень сильно разрастись. Еще больше она может разрастись, если нам необходимо создать классы, реализующие все возможные сочетания добавляемых функциональностей.

### Участники

* **Component**: абстрактный класс, который определяет интерфейс для наследуемых объектов
* **ConcreteComponent**: конкретная реализация компонента, в которую с помощью декоратора добавляется новая функциональность
* **Decorator**: собственно декоратор, реализуется в виде абстрактного класса и имеет тот же базовый класс, что и декорируемые объекты. Поэтому базовый класс Component должен быть по возможности легким и определять только базовый интерфейс.

Класс декоратора также хранит ссылку на декорируемый объект в виде объекта базового класса Component и реализует связь с базовым классом как через наследование, так и через отношение агрегации.

* Классы **ConcreteDecoratorA** и **ConcreteDecoratorB** представляют дополнительные функциональности, которыми должен быть расширен объект ConcreteComponent.

1. Нарисуйте диаграмму классов для паттерна Proxy. Поясните его назначение. Какие разновидности proxy вы знаете?

Паттерн Заместитель (Proxy) предоставляет объект-заместитель, который управляет доступом к другому объекту. То есть создается объект-суррогат, который может выступать в роли другого объекта и замещать его.

### Когда использовать прокси?

* Когда надо осуществлять взаимодействие по сети, а объект-проси должен имитировать поведения объекта в другом адресном пространстве. Использование прокси позволяет снизить накладные издержки при передачи данных через сеть. Подобная ситуация еще называется **удалённый заместитель (remote proxies)**
* Когда нужно управлять доступом к ресурсу, создание которого требует больших затрат. Реальный объект создается только тогда, когда он действительно может понадобится, а до этого все запросы к нему обрабатывает прокси-объект. Подобная ситуация еще называется **виртуальный заместитель (virtual proxies)**
* Когда необходимо разграничить доступ к вызываемому объекту в зависимости от прав вызывающего объекта. Подобная ситуация еще называется **защищающий заместитель (protection proxies)**
* Когда нужно вести подсчет ссылок на объект или обеспечить потокобезопасную работу с реальным объектом. Подобная ситуация называется **"умные ссылки" (smart reference)**

### Участники паттерна

* **Subject**: определяет общий интерфейс для Proxy и RealSubject. Поэтому Proxy может использоваться вместо RealSubject
* **RealSubject**: представляет реальный объект, для которого создается прокси
* **Proxy**: заместитель реального объекта. Хранит ссылку на реальный объект, контролирует к нему доступ, может управлять его созданием и удалением. При необходимости Proxy переадресует запросы объекту RealSubject
* **Client**: использует объект Proxy для доступа к объекту RealSubject

1. В чем разница между паттернами Decorator и Adapter?

Декоратор представляет расширенный интерфейс, а адаптер совершенно новый.

1. В чем суть паттерна Composite? Поясните в каких случаях его надо применять?

Паттерн Компоновщик (Composite) объединяет группы объектов в древовидную структуру по принципу "часть-целое и позволяет клиенту одинаково работать как с отдельными объектами, так и с группой объектов.

Образно реализацию паттерна можно представить в виде меню, которое имеет различные пункты. Эти пункты могут содержать подменю, в которых, в свою очередь, также имеются пункты. То есть пункт меню служит с одной стороны частью меню, а с другой стороны еще одним меню. В итоге мы однообразно можем работать как с пунктом меню, так и со всем меню в целом.

### Когда использовать компоновщик?

* Когда объекты должны быть реализованы в виде иерархической древовидной структуры
* Когда клиенты единообразно должны управлять как целыми объектами, так и их составными частями. То есть целое и его части должны реализовать один и тот же интерфейс

### Участники

* **Component**: определяет интерфейс для всех компонентов в древовидной структуре
* **Composite**: представляет компонент, который может содержать другие компоненты и реализует механизм для их добавления и удаления
* **Leaf**: представляет отдельный компонент, который не может содержать другие компоненты
* **Client**: клиент, который использует компоненты

1. Расскажите о паттерне Faсade?

Фасад (Facade) представляет шаблон проектирования, который позволяет скрыть сложность системы с помощью предоставления упрощенного интерфейса для взаимодействия с ней.

### Когда использовать фасад?

* Когда имеется сложная система, и необходимо упростить с ней работу. Фасад позволит определить одну точку взаимодействия между клиентом и системой.
* Когда надо уменьшить количество зависимостей между клиентом и сложной системой. Фасадные объекты позволяют отделить, изолировать компоненты системы от клиента и развивать и работать с ними независимо.
* Когда нужно определить подсистемы компонентов в сложной системе. Создание фасадов для компонентов каждой отдельной подсистемы позволит упростить взаимодействие между ними и повысить их независимость друг от друга.

### Участники

* Классы SubsystemA, SubsystemB, SubsystemC и т.д. являются компонентами сложной подсистемы, с которыми должен взаимодействовать клиент
* Client взаимодействует с компонентами подсистемы
* Facade - непосредственно фасад, который предоставляет интерфейс клиенту для работы с компонентами

1. В каких случаях надо применять Bridge? Поясните на диаграмме классов как он применятся

Мост (Bridge) - структурный шаблон проектирования, который позволяет отделить абстракцию от реализации таким образом, чтобы и абстракцию, и реализацию можно было изменять независимо друг от друга.

Даже если мы отделим абстракцию от конкретных реализаций, то у нас все равно все наследуемые классы будут жестко привязаны к интерфейсу, определяемому в базовом абстрактном классе. Для преодоления жестких связей и служит паттерн Мост.

### Когда использовать данный паттерн?

* Когда надо избежать постоянной привязки абстракции к реализации
* Когда наряду с реализацией надо изменять и абстракцию независимо друг от друга. То есть изменения в абстракции не должно привести к изменениям в реализации

### Участники

* **Abstraction**: определяет базовый интерфейс и хранит ссылку на объект Implementor. Выполнение операций в Abstraction делегируется методам объекта Implementor
* **RefinedAbstraction**: уточненная абстракция, наследуется от Abstraction и расширяет унаследованный интерфейс
* **Implementor**: определяет базовый интерфейс для конкретных реализаций. Как правило, Implementor определяет только примитивные операции. Более сложные операции, которые базируются на примитивных, определяются в Abstraction
* **ConcreteImplementorA** и **ConcreteImplementorB**: конкретные реализации, которые унаследованы от Implementor
* **Client**: использует объекты Abstraction