1. Назначение структурных паттернов? Рассм.как классы и объекты реализ.более крупные структуры

2. Нарисуете диаграмму классов и поясните принцип работы Adapter. В каких случаях применять.

Предназначен для преобразования интерфейса одного класса в интерфейс другого. Благодаря реализации данного паттерна мы можем использовать вместе классы с несовместимыми интерфейсами.

**Когда:** - необходимо исп-ть имеющийся класс, но его интерфейс не соотв.потребностям

-надо исп-ть уже сущ-щий класс совместно с другими классами, интерф.кот.несовместимы

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

Адаптер класса исп.наследование и может переносить только класс . Он не может обернуть интерфейс, поскольку по определению он должен выводиться из некоторого базового класса.  
Адаптер объектов исп.состав и может переносить классы или интерфейсы или и то, и другое. Он может это сделать, т.к.он сод. как закрытый, инкапсулир.член, класс или интерф. экземпляр объекта, кот.он обертывает.

4. Назначение и принцип организации паттерна Декоратор.

Это структурный шаблон проектирования, который позволяет динамически подключать к объекту доп. функциональность.

Для определения нового функционала в классах исп-ся наследование, а декоратор представляет более гибкую альтернативу, поскольку позволяют динамически в процессе выполнения определять новые возможности у объектов.

**Когда:** надо динам.добавлять к объекту новые функциональные возм-сти (д-е мб сняты с объекта)

5. Нарисуйте диаграмму классов для паттерна Proxy. Поясните его назначение + разновидности

Заместитель предоставляет объект-заместитель, который управляет доступом к другом объекту.

То есть создается объект-суррогат, который может выступать в роли другого объекта и замещать его.

**Когда:** - надо осущ.взаимодействие по сети, а объект-проси должен имитировать поведение объекта в другом адресном пространстве. Использование похволит снизить накладные издежки при передаче данных через сеть. Подобная ситуация называется **удалённый заместитель**

**-** нужно управлять доступом к ресурсу, создание которого требует больших затрат. Реальный объект создается только тогда, когда он действительно может понадобиться, а до этого все запросы к нему обрабатывает прокси-объект. Ситуация – **виртуальный заместитель.**

**-** необходимо разграничить доступ к вызываемому объекту в завис-ти от прав вызывающего объекта. Ситуация – **защищающий заместитель**

**-** нужно вести подсчет ссылок на объект или обеспечить потокобезопасную работу с реальным объектом. Ситуация – **умные ссылки**

6. В чем разница между паттернами Decorator и Adapter?

**Адаптер** и **Декоратор** во многом похожи, однако при реализации **Адаптера у** вас нет новой логики, кроме преобразования. При реализации **Decorator** вы фактически добавляете некоторые совершенно новые функциональные возможности, которые никогда не существовали ранее в объекте, который вы украшаете

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

Компоновщик объединяет группы объектов в древовидную структуру по принципу «часть-целое и позволяет клиенту работать как с отдельными объектами, так и с группой объектов.

Реализация – в виде меню, которое имеет различные пункты ( м.содерж.подменю, кот.также им.пункты)

**Когда:** - объекты дб реализованы в виде иерархической древовидной структуры

- клиенты единообразно должны управлять как целыми объектами, так и их составными частями. Т.е. целое и его части д. реализовать один и тот же интерфейс.

8. Расскажите о паттерне Façade?

Позвол.скрыть сложность системы с пом.представления упрощенного интерфейса для взаимодейсвтия с ней

**Когда: -** имеется сложн.сист., и необх.упростить с ней работу. Позвол.опред.одну точку взаимодействия между клиентом и системой.

- надо уменьшить кол-во завис-тей между клиентом и сложн.сист. Объекты позвол.отделить, изолировать компоненты системы от клиента и развивать и работать с ними независимо.

- нужно опред.подсист.компонентов в сложн.сист. Создание фасадов для компонентов каждой отдельной подсистемы позволит упростить взаимод.между ними и повысить их независ-ть друг от друга.

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

Мост – структ.паттерн, кот.позвол.отделить абстракцию от реализации таким образом, чтобы и абстракцию, и реализацию можно было изменять независимо друг от друга. Даже если отделить абстр.от реализ., то наслед.классы все равно будут жестко привязаны к интерфейсу, опред-мому в баз.абстр.классе.

**Когда:** - надо избежать пост.привязки абстракции к реализации

- наряду с реализ.надо измен.и абстракцию независ. друг от друга.(измен.в абстр.!= измен. в реализ)

**ADAPTER**

class Client

{

    public void Request(Target target)

    {

        target.Request();

    }

}

// класс, к которому надо адаптировать другой класс

class Target

{

    public virtual void Request()

    {}

}

// Адаптер

class Adapter : Target

{

    private Adaptee adaptee = new Adaptee();

    public override void Request()

    {

        adaptee.SpecificRequest();

    }

}

// Адаптируемый класс

class Adaptee

{

    public void SpecificRequest()

    {}

}

* **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.

То есть клиент ничего не знает об Adaptee, он знает и использует только объекты Target. И благодаря адаптеру мы можем на клиенте использовать объекты Adaptee как Target

**Decorator (декоратор)**

abstract class Component

{

    public abstract void Operation();

}

class ConcreteComponent : Component

{

    public override void Operation(){}

}

abstract class Decorator : Component

{

    protected Component component;

    public void SetComponent(Component component)

    {this.component = component; }

    public override void Operation()

    {

        if (component != null)

            component.Operation();

    }

}

class ConcreteDecoratorA : Decorator

{

    public override void Operation()

    {base.Operation(); }

}

class ConcreteDecoratorB : Decorator

{

    public override void Operation()

    {base.Operation(); }

}

* **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.

PROXY

class Client

{

    void Main()

    {

        Subject subject = new Proxy();

        subject.Request();

    }

}

abstract class Subject

{

    public abstract void Request();

}

class RealSubject : Subject

{

    public override void Request()

    {}

}

class Proxy : Subject

{

    RealSubject realSubject;

    public override void Request()

    {

        if (realSubject == null)

            realSubject = new RealSubject();

        realSubject.Request();

    }

}

* **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

COMPOSITE(КОМПОНОВЩИК)

abstract class Component

{

    protected string name;

    public Component(string name)

    {

        this.name = name;

    }

    public abstract void Display();

    public abstract void Add(Component c);

    public abstract void Remove(Component c);

}

class Composite : Component

{

    List<Component> children = new List<Component>();

    public Composite(string name)

        : base(name)

    {}

    public override void Add(Component component)

    {

        children.Add(component);

    }

    public override void Remove(Component component)

    {

        children.Remove(component);

    }

    public override void Display()

    {

        Console.WriteLine(name);

        foreach (Component component in children)

        {

            component.Display();

        }

    }

}

class Leaf : Component

{

    public Leaf(string name)

        : base(name)

    {}

    public override void Display()

    {

        Console.WriteLine(name);

    }

    public override void Add(Component component)

    {

        throw new NotImplementedException();

    }

    public override void Remove(Component component)

    {

        throw new NotImplementedException();

    }

}

* **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

FAÇADE(ФАСАД)

class SubsystemA

{ public void A1() {} }

class SubsystemB

{ public void B1() {} }

class SubsystemC

{ public void C1()  {} }

public class Facade

{

    SubsystemA subsystemA;

    SubsystemB subsystemB;

    SubsystemC subsystemC;

    public Facade(SubsystemA sa, SubsystemB sb, SubsystemC sc)

    {

        subsystemA = sa;

        subsystemB = sb;

        subsystemC = sc;

    }

    public void Operation1()

    {

        subsystemA.A1();

        subsystemB.B1();

        subsystemC.C1();

    }

    public void Operation2()

    {

        subsystemB.B1();

        subsystemC.C1();

    }

}

class Client

{

    public void Main()

    {

        Facade facade = new Facade(new SubsystemA(), new SubsystemB(), new SubsystemC());

        facade.Operation1();

        facade.Operation2();

    }

}

* 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

BRIDGE(МОСТ)

class Client

{

    static void Main() {

        Abstraction abstraction;

        abstraction = new RefinedAbstraction(new ConcreteImplementorA());

        abstraction.Operation();

        abstraction.Implementor=new ConcreteImplementorB();

        abstraction.Operation();

    }

}

abstract class Abstraction

{

    protected Implementor implementor;

public Implementor Implementor

    { set { implementor = value; }    }

public Abstraction(Implementor imp)

    { implementor = imp    }

 public virtual void Operation()

    {   implementor.OperationImp(); }

}

abstract class Implementor

{ public abstract void OperationImp();}

class RefinedAbstraction : Abstraction

{

    public RefinedAbstraction(Implementor imp) : base(imp) {}

    public override void Operation(){}

}

class ConcreteImplementorA : Implementor

{ public override void OperationImp() {} }

class ConcreteImplementorB : Implementor

{ public override void OperationImp() {} }

* **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