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

1. Назначение паттернов поведения?

Паттерны, опред-щие алгоритмы и взаимодействие между классами и объектами

2. Принцип работы Chain of responsibility + применять?

Цепочка обязанностей  
позвол. избежать жесткой привязки отправителя запроса к получателю, позволяя нескольким объектам обработать запрос.  
запрос перемещается по цепочке, каждый объект выбирает, либо обработать запрос, либо передать следующему.

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

Позвол. инкапсулировать запрос на вып-ние опред. действия в виде отдельного объекта  
Этот объект запроса на действие и называется **командой**

**Когда: -** необх. ф-ции обратного действия в ответ на опр. действия.

- необх. обесп. вып-ние очереди запросов, а также их возможную отмену.

4. Как реализовать паттерн Observer?

Исп.отношение один ко многим  
один наблюдаемый объект и мн-во наблюдателей  
При изменении наблюдаемого объекта автомат. происходит оповещение всех наблюдателей.

\* инт1: опр. методы для доб., удал., оповещения наблюдателей  
\* инт2: с пом. кот наблюдатель получ. оповещение   
2 класса : инт 1,2 - один уведомл, другой добавляет

5. Паттерн Mediator. Поясните его назначение.

Обеспеч. взаимод. мн-ва объектов без необх. ссылаться друг на друга  
 -> слабосвязанность взаимод. объектов

6. В чем разница между паттернами Mediator и Facade?

фасад предоставляет существующие функции, а медиатор добавляет к существующим функциям.

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

Позвол. сохранить состояние объекта для возможного послед. восстановления без нарушения принципа инкапсуляции

8. Расскажите о паттерне Visitor?

доб. один. набор операций разнородным классам без изменения этих классов

9. В каких случаях надо применять Null object? когда требуется инкапсулировать отсутствие объекта.

10. реализовать Strategy.

- неск.родственных классов, кот. отлич.поведением. Созд.1 осн.класс, а разные варики поведения вынести в отдельные классы и при необх.их применять

- необх.обеспеч.выбор из неск.вариков алгоритмов, кот.можно легко менять в завис-ти от условий

- необх.менять поведение объектов на стадии вып-ния программы

11. Перечислите и поясните принципы проектирования SOLID.

Способствует созданию такой системы, кот.будет легко поддерживать и расширять в течении долгого времени. Принципы солид – это руководства, кот.также м.примен.во время работы над сущ.ПО

**CHAIN OF RESPONSIBILITY(ЦЕПОЧКА ОБЯЗАННОСТЕЙ)**

abstract class Handler

{

    public Handler Successor { get; set; }

    public abstract void HandleRequest(int condition);

}

class ConcreteHandler1 : Handler

{

    public override void HandleRequest(int condition)

    {

        if (condition == 1)

        {

            // обработка;

        }

        else if (Successor != null)

        {

            Successor.HandleRequest(condition);

        }

    }

}

class ConcreteHandler2 : Handler

{

    public override void HandleRequest(int condition)

    {

        if (condition==2)

        {

            // обработка;

        }

        else if (Successor != null)

        {

            Successor.HandleRequest(condition);

        }

    }

}

* 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

**COMMAND (КОМАНДЫ)**

{

    public abstract void Execute();

    public abstract void Undo();

}

// конкретная команда

class ConcreteCommand : Command

{

    Receiver receiver;

    public ConcreteCommand(Receiver r)

    {

        receiver = r;

    }

    public override void Execute()

    {

        receiver.Operation();

    }

    public override void Undo()

    {}

}

// получатель команды

class Receiver

{ public void Operation()  { } }

class Invoker

{

    Command command;

    public void SetCommand(Command c)

    {

        command = c;

    }

    public void Run()

    {   command.Execute();    }

    public void Cancel()

    { command.Undo();   }

}

* **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()

**OBSERVER (НАБЛЮДАТЕЛЬ)**

interface IObservable

{

    void AddObserver(IObserver o);

    void RemoveObserver(IObserver o);

    void NotifyObservers();

}

class ConcreteObservable : IObservable

{

    private List<IObserver> observers;

    public ConcreteObservable()

    {

        observers = new List<IObserver>();

    }

    public void AddObserver(IObserver o)

    { observers.Add(o);    }

    public void RemoveObserver(IObserver o)

    { observers.Remove(o);  }

    public void NotifyObservers()

    {

        foreach (IObserver observer in observers)

            observer.Update();

    }

}

interface IObserver

{ void Update(); }

class ConcreteObserver :IObserver

{ public void Update() {} }

* 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.

**MEDIATOR (ПОСРЕДНИК)**

abstract class Mediator

{ public abstract void Send(string msg, Colleague colleague); }

abstract class Colleague

{

    protected Mediator mediator;

    public Colleague(Mediator mediator)

    { this.mediator = mediator    }

}

class ConcreteColleague1 : Colleague

{

    public ConcreteColleague1(Mediator mediator) : base(mediator)  { }

    public void Send(string message)

    { mediator.Send(message, this);    }

    public void Notify(string message) { }

}

class ConcreteColleague2 : Colleague

{

    public ConcreteColleague2(Mediator mediator) : base(mediator)  { }

    public void Send(string message)

    { mediator.Send(message, this);    }

    public void Notify(string message) { }

}

class ConcreteMediator : Mediator

{

    public ConcreteColleague1 Colleague1 { get; set; }

    public ConcreteColleague2 Colleague2 { get; set; }

    public override void Send(string msg, Colleague colleague)

    {

        if (Colleague1 == colleague)

            Colleague2.Notify(msg);

        else

            Colleague1.Notify(msg);

    }

}

* 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

**MEMENTO (ХРАНИТЕЛЬ)**

class Memento

{

    public string State { get; private set;}

    public Memento(string state)

    {

        this.State = state;

    }

}

class Caretaker

{

    public Memento Memento { get; set; }

}

class Originator

{

    public string State { get; set; }

    public void SetMemento(Memento memento)

    {

        State = memento.State;

    }

    public Memento CreateMemento()

    {

        return new Memento(State);

    }

}

* 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

**VISITOR (ПОСЕТИТЕЛЬ)**

abstract class Visitor {

    public abstract void VisitElementA(ElementA elemA);

    public abstract void VisitElementB(ElementB elemB); }

class ConcreteVisitor1 : Visitor {

public override void VisitElementA(ElementA elementA) { elementA.OperationA();   }

    public override void VisitElementB(ElementB elementB) { elementB.OperationB();   }

}

class ConcreteVisitor2 : Visitor

{

    public override void VisitElementA(ElementA elementA) { elementA.OperationA();   }

    public override void VisitElementB(ElementB elementB) { elementB.OperationB();   }

}

class ObjectStructure

{

    List<Element> elements = new List<Element>();

    public void Add(Element element) { elements.Add(element); }

    public void Remove(Element element) { elements.Remove(element);  }

    public void Accept(Visitor visitor)

    {

        foreach (Element element in elements)

            element.Accept(visitor);

    }

}

abstract class Element {

    public abstract void Accept(Visitor visitor);

    public string SomeState { get; set; } }

class ElementA : Element{

    public override void Accept(Visitor visitor) { visitor.VisitElementA(this);   }

    public void OperationA(){ } }

class ElementB : Element {

public override void Accept(Visitor visitor) { visitor.VisitElementB(this);    }

    public void OperationB() { }

}

* **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

**STRATEGY (СТРАТЕГИЯ)**

public interface IStrategy

{

    void Algorithm();

}

public class ConcreteStrategy1 : IStrategy

{

    public void Algorithm()

    {}

}

public class ConcreteStrategy2 : IStrategy

{

    public void Algorithm()

    {}

}

public class Context

{

    public IStrategy ContextStrategy { get; set; }

    public Context(IStrategy \_strategy)

    {

        ContextStrategy = \_strategy;

    }

    public void ExecuteAlgorithm()

    {

        ContextStrategy.Algorithm();

    }

}

* 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.