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.