1.Назначение структурных паттернов?

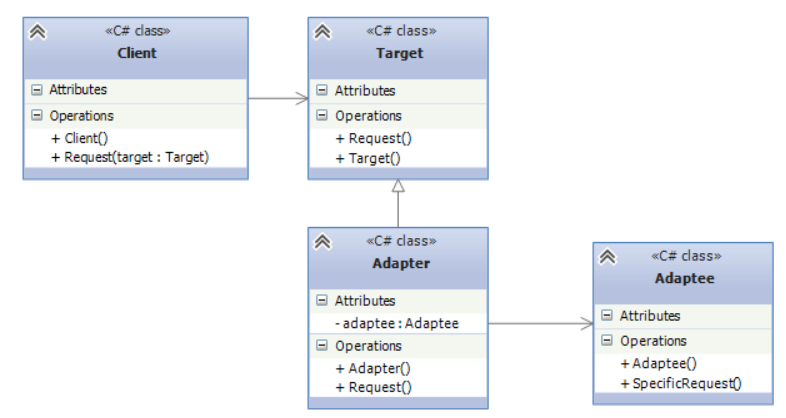
**структурные паттерны** - рассматривает, как классы и объекты образуют более крупные структуры - более сложные по характеру классы и объекты(**Адаптер (Adapter)**, **Мост (Bridge)**, **Компоновщик (Composite)**, **Декоратор (Decorator)**, **Фасад (Facade)**, **Приспособленец (Flyweight)**, **Заместитель (Proxy))**

2. 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



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()

    {}

}

### 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.

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

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

**Адаптер класса использует наследование и может переносить только класс. Он не может обернуть интерфейс, поскольку по определению он должен быть производным от некоторого базового класса.**

**Объектный адаптер использует композицию и может переносить классы или интерфейсы, или и то, и другое. Он может это сделать, поскольку содержит в качестве частного инкапсулированного члена экземпляр объекта класса или интерфейса, который он обертывает.**

4. Назначение и принцип организации паттерна Декоратор.

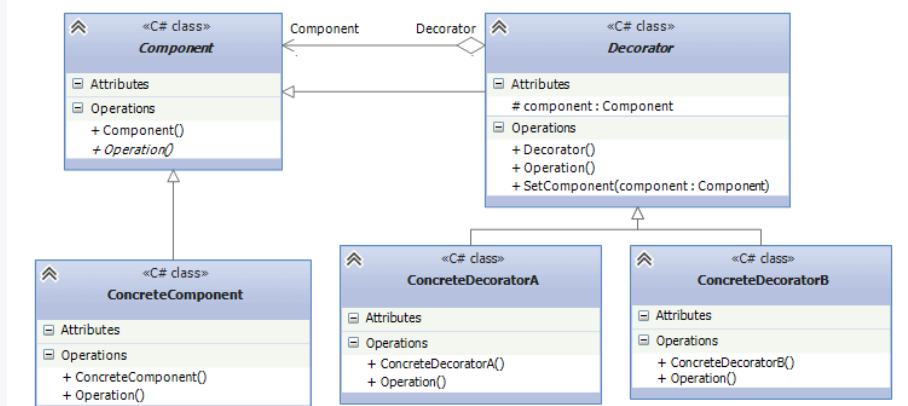
Декоратор (Decorator) представляет структурный шаблон проектирования, который позволяет динамически подключать к объекту дополнительную функциональность.

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.



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();

    }

}

### 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.

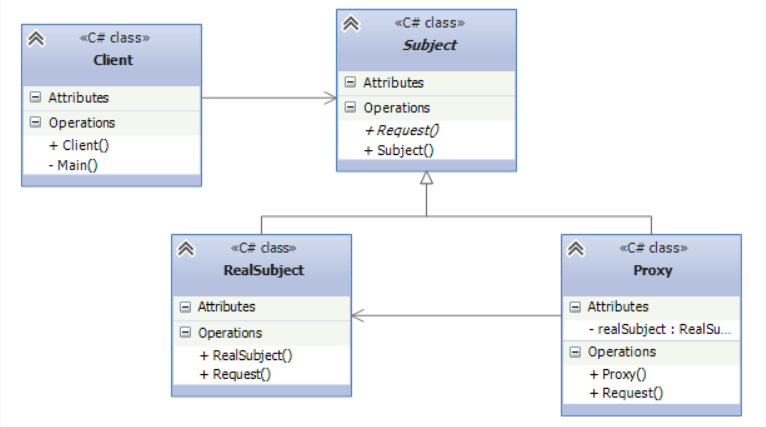
5. Нарисуйте диаграмму классов для паттерна Proxy. Поясните его назначение. Какие разновидности proxy вы знаете?

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)**

С помощью UML паттерн может быть описан так:



6. В чем разница между паттернами Decorator и Adapter?

**Адаптер** адаптирует данный класс/объект к новому интерфейсу. В случае первого используется типичное наследование. В последнем случае объект обертывается соответствующим адаптивным объектом и проходит вокруг него. Проблема, которую мы здесь решаем, это проблема несовместимых интерфейсов.

**Decorator** используется для добавления большего количества пороха к вашим объектам (обратите внимание на термин "объекты" - вы обычно украшаете объекты динамически во время выполнения). Вы не скрываете/не нарушаете существующие интерфейсы объекта, а просто расширяете его во время выполнения.

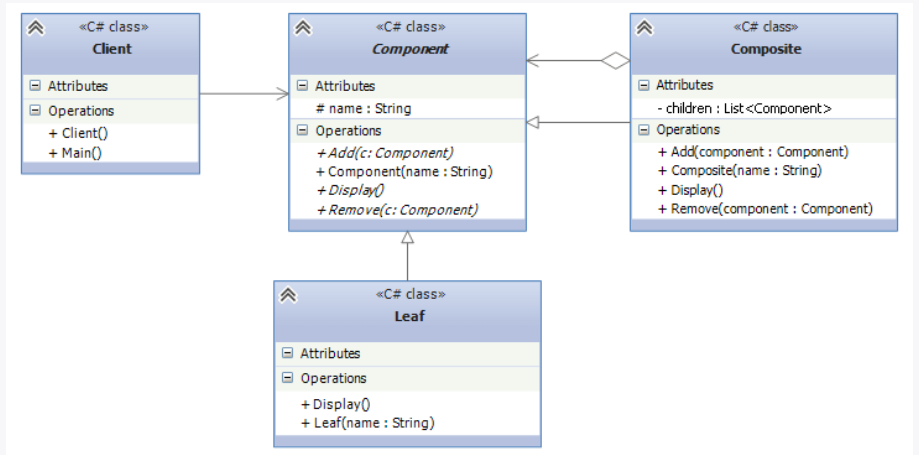
7. В чем суть паттерна Composite? Поясните в каких случаях его надо применять?

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

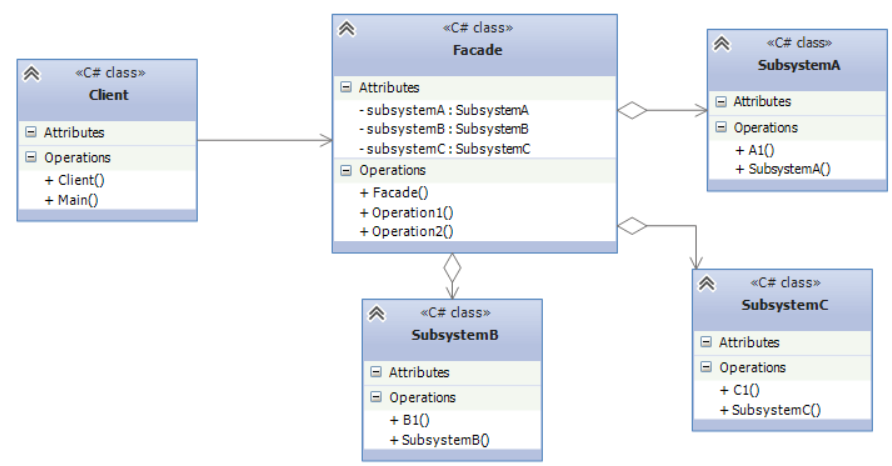


8. Расскажите о паттерне Façade?

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.



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

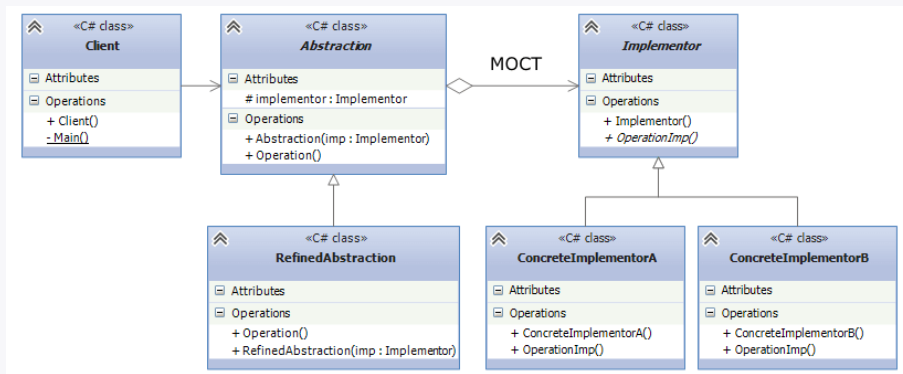
Мост (Bridge) - структурный шаблон проектирования, который позволяет отделить абстракцию от реализации таким образом, чтобы и абстракцию, и реализацию можно было изменять независимо друг от друга.

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

Общая реализация паттерна состоит в объявлении классов абстракций и классов реализаций в отдельных параллельных иерархиях классов.



### 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

10.Назначение паттернов поведения?

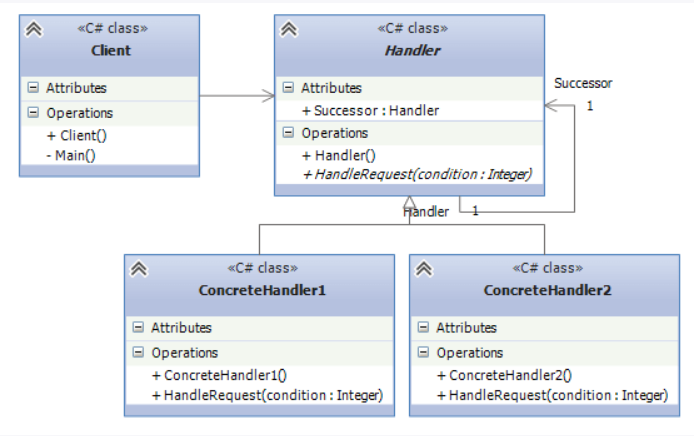
Третья группа паттернов называются **поведенческими** - они определяют алгоритмы и взаимодействие между классами и объектами, то есть их поведение. (**Цепочка обязанностей (Chain of responsibility),Команда (Command)**, **Интерпретатор (Interpreter)**, **Итератор (Iterator)**, **Посредник (Mediator)**, **Хранитель (Memento)**, **Наблюдатель (Observer)**, **Состояние (State)**, **Стратегия (Strategy)**, **Шаблонный метод (Template method)**, **Посетитель (Visitor).**

11.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. Все возможные обработчики запроса образуют цепочку, а сам запрос перемещается по этой цепочке. Каждый объект в этой цепочке при получении запроса выбирает, либо закончить обработку запроса, либо передать запрос на обработку следующему по цепочке объекту.

### 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



12.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.

Команды могут использовать параметры, которые передают ассоциированную с командой информацию. Кроме того, команды могут ставиться в очередь и также могут быть отменены.

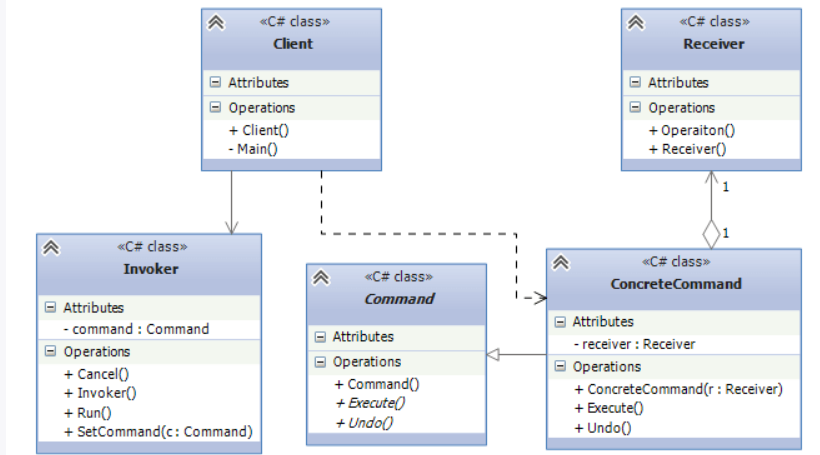
### Когда использовать команды?

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.

Схематично в UML паттерн Команда представляется следующим образом:



### 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()

Таким образом, инициатор, отправляющий запрос, ничего не знает о получателе, который и будет выполнять команду. Кроме того, если нам потребуется применить какие-то новые команды, мы можем просто унаследовать классы от абстрактного класса Command и реализовать его методы Execute и Undo.

В программах на C# команды находят довольно широкое применение. Так, в технологии WPF и других технологиях, которые используют XAML и подход MVVM, на командах во многом базируется взаимодействие с пользователем. В некоторых архитектурах, например, в архитектуре CQRS, команды являются одним из ключевых компонентов.

13.Как реализовать паттерн Observer?

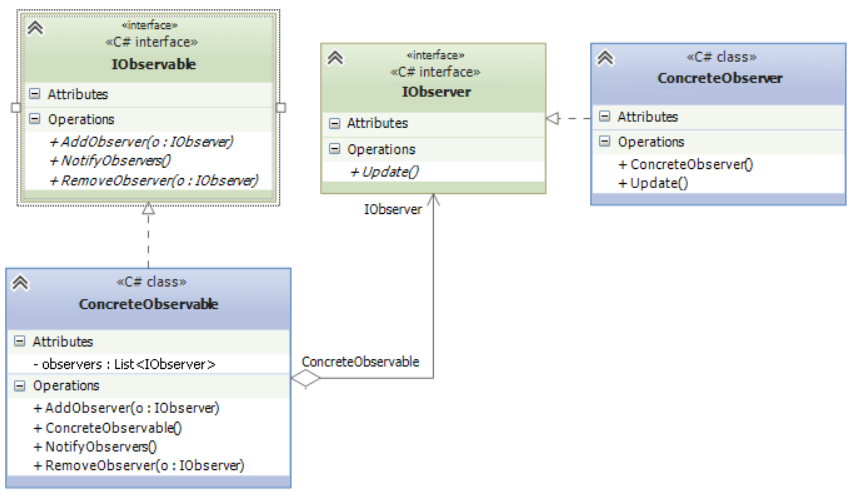
Паттерн "Наблюдатель" (Observer) представляет поведенческий шаблон проектирования, который использует отношение "один ко многим". В этом отношении есть один наблюдаемый объект и множество наблюдателей. И при изменении наблюдаемого объекта автоматически происходит оповещение всех наблюдателей.

Данный паттерн еще называют Publisher-Subscriber (издатель-подписчик), поскольку отношения издателя и подписчиков характеризуют действие данного паттерна: подписчики подписываются email-рассылку определенного сайта. Сайт-издатель с помощью email-рассылки уведомляет всех подписчиков о изменениях. А подписчики получают изменения и производят определенные действия: могут зайти на сайт, могут проигнорировать уведомления и т.д.

### 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.

С помощью диаграмм UML данный шаблон можно выразить следующим образом:



### 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.

При этом наблюдаемому объекту не надо ничего знать о наблюдателе кроме того, что тот реализует метод Update(). С помощью отношения агрегации реализуется слабосвязанность обоих компонентов. Изменения в наблюдаемом объекте не виляют на наблюдателя и наоборот.

В определенный момент наблюдатель может прекратить наблюдение. И после этого оба объекта - наблюдатель и наблюдаемый могут продолжать существовать в системе независимо друг от друга.

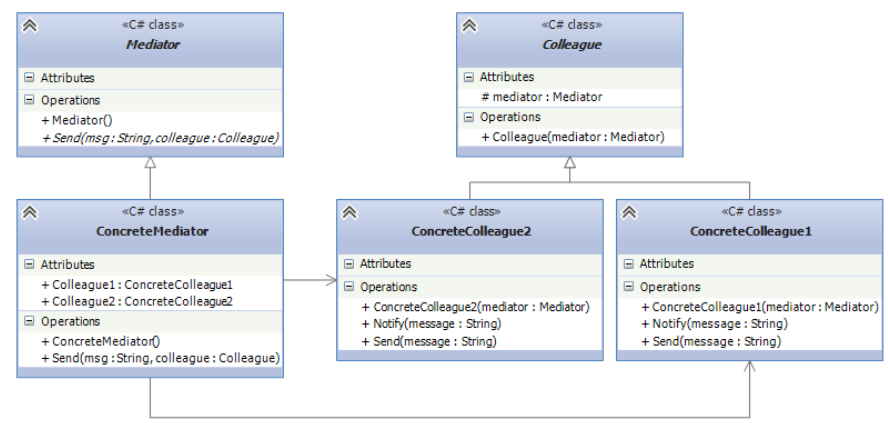
14.Нарисуйте диаграмму классов для паттерна Mediator. Поясните его назначение.

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.

Схематично с помощью UML паттерн можно описать следующим образом:



### 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

15.В чем разница между паттернами Mediator и Facade?

Фасад демонстрирует существующую функциональность, а посредник добавляет к существующей функциональности.

Если вы посмотрите на пример фасада, вы увидите, что вы не добавляете никаких новых функций, а просто придаете текущим объектам новую перспективу. Например, Circle уже существует, и вы просто абстрагируетесь от круга, используя метод drawCircle.

Если вы посмотрите на свой класс-посредник, вы увидите, что метод handle()обеспечивает дополнительную функциональность, проверяя состояние. Если бы вы убрали условия, у вас был бы узор фасада, поскольку дополнительная функциональность пропала.

16.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. Не нарушая инкапсуляции, **паттерн** **Memento** получает и сохраняет за пределами объекта его внутреннее состояние так, чтобы позже можно **было** восстановить объект в таком же состоянии.

Когда использовать Memento?

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

### 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

Теперь рассмотрим реальный пример: нам надо сохранять состояние игрового персонажа в игре:

class Program

{

    static void Main(string[] args)

    {

        Hero hero = new Hero();

        hero.Shoot(); // делаем выстрел, осталось 9 патронов

        GameHistory game = new GameHistory();

        game.History.Push(hero.SaveState()); // сохраняем игру

        hero.Shoot(); //делаем выстрел, осталось 8 патронов

        hero.RestoreState(game.History.Pop());

        hero.Shoot(); //делаем выстрел, осталось 8 патронов

        Console.Read();

    }

}

// Originator

class Hero

{

    private int patrons=10; // кол-во патронов

    private int lives=5; // кол-во жизней

    public void Shoot()

    {

        if (patrons > 0)

        {

            patrons--;

            Console.WriteLine("Производим выстрел. Осталось {0} патронов", patrons);

        }

        else

            Console.WriteLine("Патронов больше нет");

    }

    // сохранение состояния

    public HeroMemento SaveState()

    {

        Console.WriteLine("Сохранение игры. Параметры: {0} патронов, {1} жизней", patrons, lives);

        return new HeroMemento(patrons, lives);

    }

    // восстановление состояния

    public void RestoreState(HeroMemento memento)

    {

        this.patrons=memento.Patrons;

        this.lives = memento.Lives;

        Console.WriteLine("Восстановление игры. Параметры: {0} патронов, {1} жизней", patrons, lives);

    }

}

// Memento

class HeroMemento

{

    public int Patrons { get; private set; }

    public int Lives { get; private set; }

    public HeroMemento(int patrons, int lives)

    {

        this.Patrons = patrons;

        this.Lives = lives;

    }

}

// Caretaker

class GameHistory

{

    public Stack<HeroMemento> History { get; private set; }

    public GameHistory()

    {

        History = new Stack<HeroMemento>();

    }

}

17.Расскажите о паттерне Visitor?

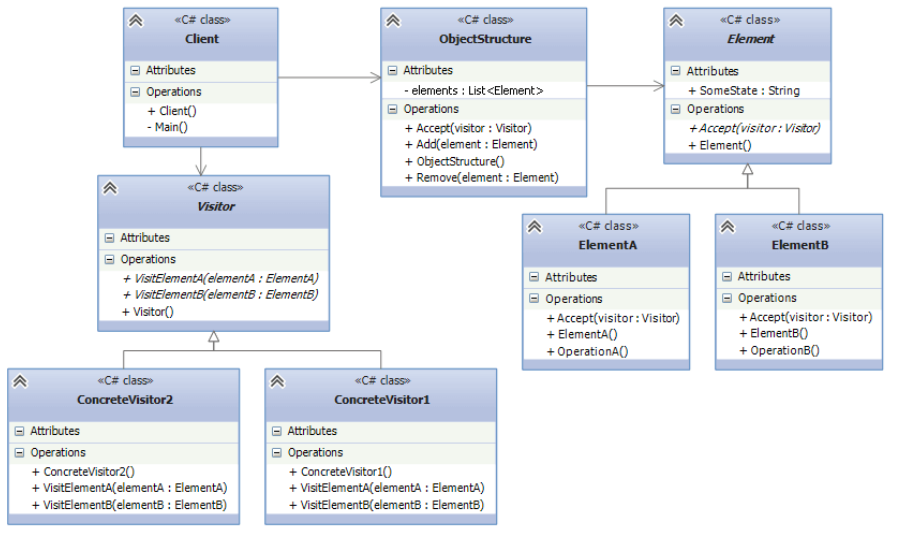
Паттерн Посетитель (Visitor) позволяет определить операцию для объектов других классов без изменения этих классов.

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

Общая структура классов паттерна, описанная с помощью диаграмм UML:



### 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

18.В каких случаях надо применять Null object?

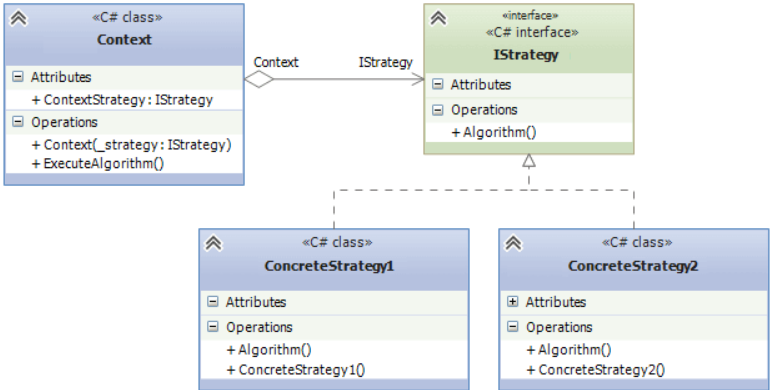
Идея паттерна Null Object заключается в том, что в качестве одной из реализаций некоторого интерфейса добавляется Null-класс, который обладает нейтральным поведением.

19.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.

В данном случае объект IStrategy заключена в свойстве ContextStrategy, хотя также для нее можно было бы определить приватную переменную, а для динамической установки использовать специальный метод.

20.Перечислите и поясните принципы проектирования SOLID

Сам акроним образован по первым буквам названий SOLID-принципов:

* **S**ingle Responsibility Principle (Принцип единственной обязанности)- **У класса должна быть только одна причина для изменения.** Под обязанностью здесь понимается набор функций, которые выполняют единую задачу. Суть этого принципа заключается в том, что класс должен выполнять одну единственную задачу. Весь функционал класса должен быть целостным, обладать высокой связностью (high cohesion).
* **O**pen/Closed Principle (Принцип открытости/закрытости)- *Программные сущности (классы, модули, функции) должны быть открыты для расширения, но не для модификации.*
* **L**iskov Substitution Principle (Принцип подстановки Лисков)- *Необходимо, чтобы подклассы могли бы служить заменой для своих суперклассов.*
* **I**nterface Segregation Principle (Принцип разделения интерфейсов)- *Создавайте узкоспециализированные интерфейсы, предназначенные для конкретного клиента. Клиенты не должны зависеть от интерфейсов, которые они не используют.*
* **D**ependency Inversion Principle (Принцип инверсии зависимостей)-   
  *Объектом зависимости должна быть абстракция, а не что-то конкретное.*