Министерство науки и высшего образования

Российской Федерации

ФГАОУ ВО «Крымский федеральный университет

имени В. И. Вернадского»

Таврическая академия (структурное подразделение)

Кафедра компьютерной инженерии и моделирования

**РЕФЕРАТ**

**по дисциплине**

**«ООП»**

*Выполнил:*

*студент (ка) 2 курса*

*кафедры КИМ, ФТИ*

*группа ПИ-б-о-231(2)*

*Аметов Кемран*

Симферополь, 2024

**Порождающие паттерны**

1. **Фабричный метод**

**Фабричный метод (Factory Method)** — это порождающий паттерн проектирования, который предоставляет общий интерфейс для создания объектов, но делегирует их создание подклассам. Таким образом, базовый класс определяет общий контракт для создания объектов, а конкретные классы-наследники решают, какой именно объект будет создан.

**Когда применять:**

1. Если заранее неизвестно, какие именно типы объектов нужно создавать.
2. Если нужно обеспечить независимость системы от конкретных классов создаваемых объектов.
3. Если система должна быть легко расширяема новыми классами без изменения существующего кода.
4. Если создание объектов нужно делегировать подклассам.

abstract class Product

{}

class ConcreteProductA : Product

{}

class ConcreteProductB : Product

{}

abstract class Creator

{

    public abstract Product FactoryMethod();

}

class ConcreteCreatorA : Creator

{

    public override Product FactoryMethod() { return new ConcreteProductA(); }

}

class ConcreteCreatorB : Creator

{

    public override Product FactoryMethod() { return new ConcreteProductB(); }

}

1. **Абстрактная фабрика**

**Паттерн "Абстрактная фабрика" (Abstract Factory)** — это порождающий паттерн, который предоставляет интерфейс для создания семейств связанных объектов, обеспечивая их совместимость, и скрывает от клиента конкретные классы создаваемых объектов.

**Когда использовать:**

1. Когда система должна быть независима от способа создания и композиции объектов.
2. Когда объекты из одного семейства должны использоваться вместе и быть совместимыми.
3. Когда требуется предоставить множество вариантов реализаций семейства объектов.

abstract class AbstractFactory

{

    public abstract AbstractProductA CreateProductA();

    public abstract AbstractProductB CreateProductB();

}

class ConcreteFactory1: AbstractFactory

{

    public override AbstractProductA CreateProductA()

    {

        return new ProductA1();

    }

    public override AbstractProductB CreateProductB()

    {

        return new ProductB1();

    }

}

class ConcreteFactory2: AbstractFactory

{

    public override AbstractProductA CreateProductA()

    {

        return new ProductA2();

    }

    public override AbstractProductB CreateProductB()

    {

        return new ProductB2();

    }

}

abstract class AbstractProductA

{}

abstract class AbstractProductB

{}

class ProductA1: AbstractProductA

{}

class ProductB1: AbstractProductB

{}

class ProductA2: AbstractProductA

{}

class ProductB2: AbstractProductB

{}

class Client

{

    private AbstractProductA abstractProductA;

    private AbstractProductB abstractProductB;

    public Client(AbstractFactory factory)

    {

        abstractProductB = factory.CreateProductB();

        abstractProductA = factory.CreateProductA();

    }

    public void Run()

    { }

}

1. **Одиночка/Singleton**

**Паттерн "Одиночка" (Singleton)** — это порождающий паттерн, который гарантирует существование только одного экземпляра определённого класса и предоставляет глобальную точку доступа к этому экземпляру.

**Когда использовать:**

1. Когда требуется, чтобы для определённого класса существовал только один экземпляр (например, конфигурационный менеджер или логгер).
2. Когда нужно обеспечить контроль над глобальной точкой доступа к ресурсу.
3. Когда объект должен быть создан только при его первой необходимости, чтобы сэкономить ресурсы.

**Пример: Логгер**

class Singleton

{

    private static Singleton instance;

    private Singleton()

    {}

    public static Singleton getInstance()

    {

        if (instance == null)

            instance = new Singleton();

        return instance;

    }

}

1. **Прототип/Prototype**

**Паттерн "Прототип" (Prototype)** — это порождающий паттерн, который позволяет создавать новые объекты, копируя существующие. Этот подход особенно полезен, если создание объекта через конструктор слишком сложное или ресурсоемкое.

**Когда использовать:**

1. Когда тип создаваемого объекта должен определяться во время выполнения.
2. Когда создание объекта через конструктор дорого или сложно.
3. Когда нужно избежать создания параллельной иерархии классов для фабрик, как в случае с паттерном "Абстрактная фабрика".
4. Когда объект может принимать ограниченное число состояний, и клонирование предпочтительнее, чем повторная инициализация.

class Client

{

    void Operation()

    {

        Prototype prototype = new ConcretePrototype1(1);

        Prototype clone = prototype.Clone();

        prototype = new ConcretePrototype2(2);

        clone = prototype.Clone();

    }

}

abstract class Prototype

{

    public int Id { get; private set; }

    public Prototype(int id)

    {

        this.Id = id;

    }

    public abstract Prototype Clone();

}

class ConcretePrototype1 : Prototype

{

    public ConcretePrototype1(int id)

        : base(id)

    { }

    public override Prototype Clone()

    {

        return new ConcretePrototype1(Id);

    }

}

class ConcretePrototype2 : Prototype

{

    public ConcretePrototype2(int id)

        : base(id)

    { }

    public override Prototype Clone()

    {

        return new ConcretePrototype2(Id);

    }

}

1. **Строитель**

**Паттерн "Строитель" (Builder)** — это порождающий шаблон проектирования, который отделяет процесс построения сложного объекта от его представления, так что один и тот же процесс построения может создавать разные представления.

**Когда использовать:**

1. Когда процесс создания объекта сложный и состоит из множества шагов.
2. Когда нужно создавать разные представления объекта, изменяя только процесс его построения.
3. Когда требуется разделить создание объекта и его представление для лучшей читаемости и поддержки кода.

class Client

{

    void Main()

    {

        Builder builder = new ConcreteBuilder();

        Director director = new Director(builder);

        director.Construct();

        Product product = builder.GetResult();

    }

}

class Director

{

    Builder builder;

    public Director(Builder builder)

    {

        this.builder = builder;

    }

    public void Construct()

    {

        builder.BuildPartA();

        builder.BuildPartB();

        builder.BuildPartC();

    }

}

abstract class Builder

{

    public abstract void BuildPartA();

    public abstract void BuildPartB();

    public abstract void BuildPartC();

    public abstract Product GetResult();

}

class Product

{

    List<object> parts = new List<object>();

    public void Add(string part)

    {

        parts.Add(part);

    }

}

class ConcreteBuilder : Builder

{

    Product product = new Product();

    public override void BuildPartA()

    {

        product.Add("Part A");

    }

    public override void BuildPartB()

    {

        product.Add("Part B");

    }

    public override void BuildPartC()

    {

        product.Add("Part C");

    }

    public override Product GetResult()

    {

        return product;

    }

}

**Структурные паттерны**

1. **Адаптер**

проектирования, который преобразует интерфейс одного класса в интерфейс другого. Он позволяет объектам с несовместимыми интерфейсами работать вместе.

**Когда использовать:**

1. Когда необходимо использовать существующий класс, но его интерфейс не соответствует требованиям.
2. Когда требуется интегрировать несовместимые классы в одну систему, не изменяя их код.

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()

    {}

}

1. **Мост**

**Паттерн "Мост" (Bridge)** — это структурный шаблон проектирования, который разделяет абстракцию и реализацию, позволяя им изменяться независимо друг от друга.

**Когда использовать:**

1. Когда нужно избежать жесткой привязки между абстракцией и её реализацией.
2. Когда необходимо, чтобы абстракция и реализация могли развиваться независимо друг от друга.
3. Когда одна из частей системы изменяется намного чаще другой.

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()

    {

    }

}

1. **Компоновщик**

**Описание:**  
Компоновщик позволяет представить древовидную структуру, где отдельные объекты и группы объектов могут обрабатываться единообразно через общий интерфейс.

**Когда использовать:**

1. Когда объекты образуют иерархическую древовидную структуру.
2. Когда требуется единообразно работать с отдельными объектами и их объединениями (например, целым деревом).

class Client

{

    public void Main()

    {

        Component root = new Composite("Root");

        Component leaf = new Leaf("Leaf");

        Composite subtree = new Composite("Subtree");

        root.Add(leaf);

        root.Add(subtree);

        root.Display();

    }

}

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();

    }

}

1. **Декоратор**

**Паттерн "Декоратор" (Decorator)**

**Описание:**  
Декоратор позволяет динамически добавлять объекту новые функциональные возможности без изменения его исходного класса. Это альтернатива использованию наследования, которая позволяет гибко изменять поведение объектов во время выполнения.

**Когда использовать:**

1. Когда нужно динамически добавлять или изменять функциональность объекта.
2. Когда использование наследования приводит к излишнему увеличению числа классов.
3. Когда функциональность может быть "включена" или "выключена" по мере необходимости.

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();

    }

}

1. **Фасад**

**Паттерн "Фасад" (Facade)**

**Описание:**  
Фасад — это структурный паттерн проектирования, который предоставляет упрощенный интерфейс для работы с комплексной системой, скрывая ее внутреннюю сложность. Он позволяет клиенту взаимодействовать с системой через единую точку доступа, не углубляясь в детали работы каждой подсистемы.

**Когда использовать фасад:**

1. **Сложная система**: Когда система слишком сложна и взаимодействие с ней требует упрощения.
2. **Изоляция клиентской логики от системы**: Чтобы уменьшить количество зависимостей между клиентом и системой.
3. **Упрощение взаимодействия между подсистемами**: Когда нужно улучшить независимость подсистем и сделать их более доступными для внешнего взаимодействия.

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();

    }

}

1. **Приспособленец**

**Паттерн "Приспособленец" (Flyweight)**

**Описание:** Паттерн "Приспособленец" — это структурный шаблон проектирования, который позволяет эффективно использовать разделяемые объекты для сокращения потребления памяти, особенно в случае, когда множество объектов имеют однотипные состояния, которые можно вынести на уровень общего использования.

Основная идея паттерна — разделение состояния объекта на **внутреннее** и **внешнее**. Внутреннее состояние может быть разделяемым (например, код символа), а внешнее состояние зависит от контекста (например, позиция символа на странице).

**Когда использовать "Приспособленец":**

1. **Большое количество однообразных объектов**: Когда приложение создает большое количество объектов с одинаковыми или схожими данными, что приводит к большому потреблению памяти.
2. **Вынос изменяемого состояния**: Когда часть состояния объекта, которая изменяется, можно выделить во внешний контекст (например, положение символа на странице), в то время как внутреннее состояние остается неизменным.
3. **Необходимость экономии памяти**: Когда объекты, хранящие одинаковое внутреннее состояние, могут быть разделены и использовать одно общее состояние, а уникальное состояние будет храниться отдельно.

class FlyweightFactory

{

    Hashtable flyweights = new Hashtable();

    public FlyweightFactory()

    {

        flyweights.Add("X", new ConcreteFlyweight());

        flyweights.Add("Y", new ConcreteFlyweight());

        flyweights.Add("Z", new ConcreteFlyweight());

    }

    public Flyweight GetFlyweight(string key)

    {

        if (!flyweights.ContainsKey(key))

            flyweights.Add(key, new ConcreteFlyweight());

        return flyweights[key] as Flyweight;

    }

}

abstract class Flyweight

{

    public abstract void Operation(int extrinsicState);

}

class ConcreteFlyweight : Flyweight

{

    int intrinsicState;

    public override void Operation(int extrinsicState)

    {

    }

}

class UnsharedConcreteFlyweight : Flyweight

{

    int allState;

    public override void Operation(int extrinsicState)

    {

        allState = extrinsicState;

    }

}

class Client

{

    void Main()

    {

        int extrinsicstate = 22;

        FlyweightFactory f = new FlyweightFactory();

        Flyweight fx = f.GetFlyweight("X");

        fx.Operation(--extrinsicstate);

        Flyweight fy = f.GetFlyweight("Y");

        fy.Operation(--extrinsicstate);

        Flyweight fd = f.GetFlyweight("D");

        fd.Operation(--extrinsicstate);

        UnsharedConcreteFlyweight uf = new UnsharedConcreteFlyweight();

        uf.Operation(--extrinsicstate);

    }

}

1. **Заместитель**

**Паттерн "Заместитель" (Proxy)**

**Описание:** Паттерн "Заместитель" предоставляет объект-заместитель, который управляет доступом к другому объекту. Это суррогат, который выполняет роль другого объекта, контролируя доступ к нему и позволяя добавлять различные функциональности, такие как отложенная инициализация, управление доступом или подсчет ссылок.

Основная идея паттерна — предоставить объект, который замещает реальный объект и позволяет работать с ним как с обычным объектом, но при этом прокси может включать дополнительные действия, такие как кэширование, выполнение дополнительных проверок и другие.

**Когда использовать "Прокси":**

1. **Удаленный заместитель (Remote Proxy)**: Когда взаимодействие происходит через сеть, и объект-прокси должен имитировать поведение объекта в другом адресном пространстве. Прокси помогает снизить накладные расходы при передаче данных через сеть.
2. **Виртуальный заместитель (Virtual Proxy)**: Когда нужно управлять доступом к ресурсу, создание которого требует больших затрат (например, тяжелая инициализация объекта). Прокси создает реальный объект только тогда, когда он действительно нужен, экономя ресурсы.
3. **Защищающий заместитель (Protection Proxy)**: Когда нужно ограничить доступ к объекту в зависимости от прав вызывающего. Прокси может проверять права доступа и предоставлять доступ только тем, у кого есть соответствующие разрешения.
4. **Умные ссылки (Smart Reference)**: Когда необходимо вести подсчет ссылок на объект или обеспечить потокобезопасность при работе с реальным объектом. Прокси может следить за количеством ссылок или управлять доступом в многозадачной среде.

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();

    }

}

**Поведенческие паттерны**

1. **Интерпретатор**

**Паттерн "Интерпретатор" (Interpreter)**

**Описание:** Паттерн "Интерпретатор" используется для описания грамматики языка и создания интерпретатора, который может интерпретировать (или вычислять) строки этого языка. Этот паттерн наиболее часто применяется для создания компиляторов, парсеров или других программ, которые должны работать с текстом в определенном формате или языке.

**Суть паттерна:** Интерпретатор позволяет вычислять результат, основанный на грамматике. Он определяет правила синтаксиса, а также предоставляет методы для выполнения операций с использованием этих правил. Обычно грамматика языка представляет собой иерархическую структуру, где каждый элемент грамматики имеет свой собственный интерпретатор.

**Когда использовать паттерн "Интерпретатор"?**

1. **Когда нужно интерпретировать язык или выражения**: Паттерн используется, когда требуется интерпретировать и выполнять выражения, написанные на определенном языке, который может быть легко сформализован с помощью грамматики.
2. **Когда грамматика языка или его правила изменяются часто**: Если вам нужно часто изменять или расширять язык, то паттерн позволяет гибко добавлять новые правила или операторы в интерпретатор без необходимости переписывать всю программу.
3. **Когда выражения часто повторяются**: Если язык или структура выражений повторяется многократно в приложении, например, математические выражения, команды или запросы, паттерн может помочь эффективно обработать их.

class Client

{

    void Main()

    {

        Context context = new Context();

        var expression = new NonterminalExpression();

        expression.Interpret(context);

    }

}

class Context

{

}

abstract class AbstractExpression

{

    public abstract void Interpret(Context context);

}

class TerminalExpression : AbstractExpression

{

    public override void Interpret(Context context)

    {

    }

}

class NonterminalExpression : AbstractExpression

{

    AbstractExpression expression1;

    AbstractExpression expression2;

    public override void Interpret(Context context)

    {

    }

}

1. **Итератор**

**Описание:** Паттерн "Итератор" позволяет предоставлять последовательный доступ ко всем элементам составного объекта без раскрытия его внутренней структуры. Это абстрактный интерфейс для перебора элементов коллекции, обеспечивающий единый способ обхода для различных типов коллекций.

**Когда использовать паттерн "Итератор"?**

1. **Когда необходимо осуществить обход объекта без раскрытия его внутренней структуры**: Итератор позволяет клиенту работать с коллекциями и их элементами, не зная, как они устроены внутри. Например, это полезно для работы с различными типами коллекций (массивы, списки, хеш-таблицы) с единым интерфейсом.
2. **Когда имеется набор составных объектов, и надо обеспечить единый интерфейс для их перебора**: Когда существует несколько типов коллекций (например, массивы, списки, очереди), и требуется одинаковый способ перебора элементов в этих коллекциях, итератор позволяет абстрагироваться от конкретных реализаций.
3. **Когда необходимо предоставить несколько альтернативных вариантов перебора одного и того же объекта**: Итераторы позволяют определить несколько разных способов обхода коллекции. Например, можно создать итератор, который будет перебрать элементы в обратном порядке, или такой, который будет пропускать элементы через определенный интервал.

class Client

{

    public void Main()

    {

        Aggregate a = new ConcreteAggregate();

        Iterator i = a.CreateIterator();

        object item = i.First();

        while (!i.IsDone())

        {

            item = i.Next();

        }

    }

}

abstract class Aggregate

{

    public abstract Iterator CreateIterator();

    public abstract int Count { get; protected set; }

    public abstract object this[int index] { get; set; }

}

class ConcreteAggregate : Aggregate

{

    private readonly ArrayList \_items = new ArrayList();

    public override Iterator CreateIterator()

    {

        return new ConcreteIterator(this);

    }

    public override int Count

    {

        get { return \_items.Count; }

        protected set { }

    }

    public override object this[int index]

    {

        get { return \_items[index]; }

        set { \_items.Insert(index, value); }

    }

}

abstract class Iterator

{

    public abstract object First();

    public abstract object Next();

    public abstract bool IsDone();

    public abstract object CurrentItem();

}

class ConcreteIterator : Iterator

{

    private readonly Aggregate \_aggregate;

    private int \_current;

    public ConcreteIterator(Aggregate aggregate)

    {

        this.\_aggregate = aggregate;

    }

    public override object First()

    {

        return \_aggregate[0];

    }

    public override object Next()

    {

        object ret = null;

        \_current++;

        if (\_current < \_aggregate.Count)

        {

            ret = \_aggregate[\_current];

        }

        return ret;

    }

    public override object CurrentItem()

    {

        return \_aggregate[\_current];

    }

    public override bool IsDone()

    {

        return \_current >= \_aggregate.Count;

    }

}

1. **Посредник**

### Паттерн "Посредник" (Mediator)

**Описание:** Паттерн "Посредник" обеспечивает централизованное управление взаимодействиями между множеством объектов, чтобы они не ссылались друг на друга напрямую. Это позволяет снизить взаимную зависимость и упростить управление связями между объектами. Вместо того чтобы объекты взаимодействовали напрямую, они обращаются к посреднику, который управляет их взаимодействием.

### Когда использовать паттерн "Посредник"?

1. **Когда имеется множество взаимосвязанных объектов, связи между которыми сложны и запутаны**: Если взаимодействия между объектами становятся слишком сложными и трудноуправляемыми, паттерн "Посредник" позволяет сосредоточить всю логику взаимодействия в одном месте, упростив структуру системы.
2. **Когда необходимо повторно использовать объект, однако повторное использование затруднено в силу сильных связей с другими объектами**: Если объект слишком зависит от других объектов, и его повторное использование в другом контексте затруднено, паттерн "Посредник" помогает устранить прямую зависимость, передавая ответственность за взаимодействие посреднику.

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);

    }

}

1. **Хранитель**

### Паттерн "Хранитель" (Memento)

**Описание:** Паттерн "Хранитель" позволяет сохранять внутреннее состояние объекта, не нарушая принципа инкапсуляции, и восстанавливать его позже. С помощью этого паттерна объект может хранить свое состояние в отдельном месте, а затем при необходимости восстанавливать его без того, чтобы напрямую обращаться к его внутренним данным.

Этот паттерн часто используется в случаях, когда требуется возможность отката состояния объекта к ранее сохраненному моменту, как например, в играх, приложениях с возможностью отмены изменений или в системах, где требуется откат изменений (например, отмена операции в текстовом редакторе).

### Когда использовать паттерн "Хранитель"?

1. **Когда нужно сохранить состояние объекта для возможного последующего восстановления**: Если объект имеет сложное внутреннее состояние, которое может изменяться в процессе работы, и его нужно сохранить для отката в будущем.
2. **Когда сохранение состояния должно проходить без нарушения принципа инкапсуляции**: Паттерн помогает сохранить состояние объекта в таком виде, чтобы это не нарушало его инкапсуляцию, то есть не делало его данные доступными для внешнего доступа или изменения.

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);

    }

}

1. **Наблюдатель**

### Паттерн "Наблюдатель" (Observer)

**Описание:** Паттерн "Наблюдатель" (или Publisher-Subscriber) является поведенческим шаблоном проектирования, который позволяет объектам оповещать другие объекты о своем изменении состояния. Этот паттерн используется для организации отношения "один ко многим", когда одно изменение в одном объекте должно привести к изменению в нескольких других объектах. Например, когда объект (издатель) обновляется, все подписчики (наблюдатели) получают уведомления о происходящих изменениях.

### Когда использовать паттерн "Наблюдатель"?

1. **Когда система состоит из множества классов, объекты которых должны находиться в согласованных состояниях**: Когда несколько объектов должны синхронизироваться с одним объектом, например, при изменении его состояния все подписчики должны быть проинформированы о таких изменениях.
2. **Когда общая схема взаимодействия объектов предполагает две стороны**: одна рассылает сообщения и является главным (издателем), а другая получает сообщения и реагирует на них (подписчики). Отделение логики обеих сторон позволяет их рассматривать независимо и использовать отдельно друг от друга.
3. **Когда существует один объект, рассылающий сообщения, и множество подписчиков, которые получают сообщения**. При этом точное число подписчиков заранее неизвестно и может изменяться в процессе работы программы.

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()

    {

    }

}

1. **Состояние**

**Описание:** Паттерн "Состояние" — это поведенческий шаблон проектирования, который позволяет объекту изменять свое поведение в зависимости от его внутреннего состояния. Вместо того чтобы использовать многочисленные условные операторы для изменения поведения объекта, этот паттерн позволяет делегировать изменения состояния в отдельные объекты, представляющие различные состояния. Это упрощает код, делает его более гибким и расширяемым.

### Когда применять данный паттерн?

1. **Когда поведение объекта должно зависеть от его состояния и может изменяться динамически во время выполнения:** Когда объект имеет различные состояния, и его поведение меняется в зависимости от этих состояний. Вместо того чтобы использовать многочисленные условные конструкции для реализации этой логики, лучше использовать паттерн "Состояние", который делегирует поведение объектам-состояниям.
2. **Когда в коде методов объекта используются многочисленные условные конструкции, выбор которых зависит от текущего состояния объекта:** В случае, если объект должен выполнять разные действия в зависимости от своего состояния, часто приходится использовать длинные и сложные условные блоки. Паттерн "Состояние" позволяет избавить код от этих конструкций, делая его более чистым и удобным для модификации.

class Program

{

    static void Main()

    {

        Context context = new Context(new StateA());

        context.Request(); // Переход в состояние StateB

        context.Request();  // Переход в состояние StateA

    }

}

abstract class State

{

    public abstract void Handle(Context context);

}

class StateA : State

{

    public override void Handle(Context context)

    {

        context.State = new StateB();

    }

}

class StateB : State

{

    public override void Handle(Context context)

    {

        context.State = new StateA();

    }

}

class Context

{

    public State State { get; set; }

    public Context(State state)

    {

        this.State = state;

    }

    public void Request()

    {

        this.State.Handle(this);

    }

}

1. **Стратегия**

**Описание:** Паттерн "Стратегия" относится к поведенческим шаблонам проектирования и позволяет инкапсулировать алгоритмы в отдельные классы и обеспечивать их взаимозаменяемость. Это позволяет изменять поведение объекта, не изменяя сам объект, путем замены используемой стратегии. Такой подход помогает избежать большого числа условных операторов и сделать код более гибким.

### Когда использовать паттерн "Стратегия"?

1. **Когда есть несколько родственных классов, которые отличаются поведением.** Вместо того чтобы создавать множество условных конструкций для обработки разных вариантов поведения, можно выделить общее поведение в родительский класс, а различные варианты поведения — в отдельные стратегии. Это позволяет легко добавлять новые стратегии, не изменяя основную логику.
2. **Когда необходимо обеспечить выбор из нескольких вариантов алгоритмов, которые можно легко менять в зависимости от условий.** В некоторых ситуациях может потребоваться изменить алгоритм выполнения операции в зависимости от условий. Паттерн "Стратегия" позволяет динамически менять алгоритмы без изменения самого объекта.
3. **Когда необходимо менять поведение объектов на стадии выполнения программы.** Стратегия позволяет изменять поведение объекта в зависимости от условий, что полезно, когда поведение должно меняться во время выполнения программы.
4. **Когда класс, применяющий определенную функциональность, ничего не должен знать о ее реализации.** Класс, использующий стратегию, должен взаимодействовать с ней через общий интерфейс, не зная о деталях реализации конкретного алгоритма.

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();

    }

}

1. **Шаблонный метод**

### Паттерн "Шаблонный метод" (Template Method)

**Описание:** Паттерн "Шаблонный метод" относится к поведенческим шаблонам проектирования. Он позволяет определить общий алгоритм выполнения задачи в базовом классе, оставив возможность подклассам переопределять отдельные шаги этого алгоритма. Это помогает избежать дублирования кода, позволяя в одном месте определить общий алгоритм, а в подклассах — конкретную реализацию шагов.

### Когда использовать паттерн "Шаблонный метод"?

1. **Когда планируется, что в будущем подклассы должны будут переопределять отдельные шаги алгоритма без изменения его структуры.** В этом случае базовый класс определяет структуру алгоритма, а конкретные детали, которые могут изменяться, реализуются в подклассах.
2. **Когда в классах, реализующим схожий алгоритм, происходит дублирование кода.** Вынесение общего кода в шаблонный метод позволяет избежать дублирования. Общие шаги алгоритма будут выполняться в базовом классе, а специфичные для подкласса части — в его методах.

abstract class AbstractClass

{

    public void TemplateMethod()

    {

        Operation1();

        Operation2();

    }

    public abstract void Operation1();

    public abstract void Operation2();

}

class ConcreteClass : AbstractClass

{

    public override void Operation1()

    {

    }

    public override void Operation2()

    {

    }

}

1. **Посетитель**

**Паттерн "Посетитель" (Visitor)**

**Описание:** Паттерн "Посетитель" позволяет добавить новые операции к объектам различных классов, не изменяя сами классы. Это достигается путем разделения структуры объектов и операций, которые над ними выполняются. Суть паттерна заключается в том, что объект посетителя реализует операцию, которая принимает объекты различных классов, и на основе их типов выполняет определенную логику.

**Когда использовать паттерн "Посетитель"?**

1. **Когда имеется много объектов разнородных классов с разными интерфейсами, и требуется выполнить ряд операций над каждым из этих объектов.** Паттерн помогает организовать выполнение операций над объектами, принадлежащими разным классам, без изменения этих классов.
2. **Когда классам необходимо добавить одинаковый набор операций без изменения этих классов.** Если классы объектов часто изменяются, но операции над ними остаются стабильными, "Посетитель" позволяет добавлять новые операции, не влияя на саму структуру классов.
3. **Когда часто добавляются новые операции к классам, при этом общая структура классов стабильна и практически не изменяется.** Паттерн предоставляет гибкость для расширения функциональности без необходимости модификации существующих классов, что облегчает добавление новых операций.

**Структура паттерна "Посетитель"**

Паттерн "Посетитель" включает две основные иерархии классов:

* **Элементы** (элементы, для которых нужно определить операцию) — это классы, которые представляют объекты, над которыми выполняются операции.
* **Посетители** (классы, определяющие операции) — это классы, которые содержат логику для выполнения операций над элементами.

class Client

{

    void Main()

    {

        var structure = new ObjectStructure();

        structure.Add(new ElementA());

        structure.Add(new ElementB());

        structure.Accept(new ConcreteVisitor1());

        structure.Accept(new ConcreteVisitor2());

    }

}

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()

    { }

}