МИНОБРНАУКИ РОССИИ

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ**

**ВЫСШЕГО ОБРАЗОВАНИЯ**

**“ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ”**

Факультет *компьютерных наук*

Кафедра *программирования и информационных технологий*

*Лабораторная работа 4 “Паттерны”*

Руководитель *Д.Н. Борисов*

Обучающийся *В.М. Беспалов, группа 3.1*

Воронеж 2022

Содержание

[Содержание 2](#_Toc99127187)

[Описание 3](#_Toc99127188)

[1. Системный паттерн (Callback) 3](#_Toc99127189)

[2. Структурный паттерн (Adapter) 4](#_Toc99127190)

[3. Поведенческие паттерны (Visitor) 7](#_Toc99127191)

[4. Производящие паттерны (Abstract Factory) 9](#_Toc99127192)

[5. Паттерны параллельного программирования (lock object) 11](#_Toc99127193)

Описание

1. Системный паттерн (Callback)

Данный шаблон проектирования подразумевает запуск чего-либо с использованием функции обратного вызова. Таким образом, этот подход позволяет выполнять что-либо используя собственный обработчик.

Достоинства:

* Повышает универсальность программы.
* Позволяет добавить пользовательский обработчик чего-либо.

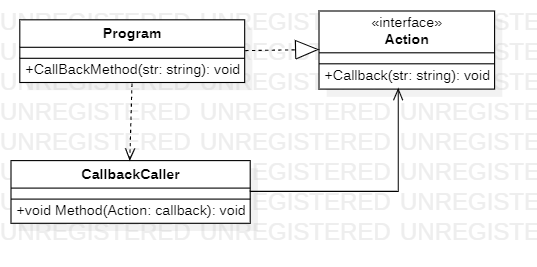
Недостатки:

* Снижает скорость работы программы за счёт увеличения размера стека вызовов.
* В некоторых случаях увеличивает потребление памяти

Ниже приведён код программы-примера на языке C#:

using System;  
  
namespace ConsoleApp1  
{  
 public static class Program  
 {  
 private static void Main(string[] args)  
 {  
 var caller = new CallbackCaller();  
 caller.Method(CallBackMethod);  
 }  
  
 static void CallBackMethod(string str)  
 {  
 Console.WriteLine($"Callback was: {str}");  
 }   
 }  
  
  
 public class CallbackCaller  
 {public void Method(Action<string> callback)  
 {callback("The message to send back");  
 }  
 }  
}

Ниже приведена UML-диаграмма, иллюстрирующая данный код:



1. UML диаграмма, иллюстрирующая паттерн CallBack
2. Структурный паттерн (Adapter)

Задача данного паттерна состоит в том, чтобы привести интерфейс одного класса к другому, в том случае, если функциональность базового класса (или интерфейса) позволяет это сделать.

Достоинства:

* Позволяет не перегружать базовый класс (интерфейс) за счет создания новой, отдельной надстройки.
* Увеличивает гибкость и универсальность программы.

Недостатки:

* Снижает производительность. Также возможно создание цепочки таких адаптеров, которая будет значительно снижать производительность, а также увеличит запутанность кода.

Ниже приведён код программы-примера на языке C# для примерного случая с адаптером для Apple и Android телефонов:

using System;  
  
namespace ConsoleApp1  
{  
 public static class Adapter  
 {  
 public interface ILightningPhone  
 {  
 void ConnectLightning();  
 void Recharge();  
 }  
  
 private interface IUsbPhone  
 {  
 void ConnectUsb();  
 void Recharge();  
 }  
  
 public class AndroidPhone : IUsbPhone  
 {  
 private bool \_isConnected;  
   
 public void ConnectUsb()  
 {  
 this.\_isConnected = true;  
 Console.WriteLine("Android phone connected.");  
 }  
  
 public void Recharge()  
 {  
 Console.WriteLine(this.\_isConnected ? "Android phone recharging." : "Connect the USB cable first.");  
 }  
 }  
  
 private class ApplePhone : ILightningPhone  
 {  
 private bool \_isConnected;  
   
 public void ConnectLightning()  
 {  
 this.\_isConnected = true;  
 Console.WriteLine("Apple phone connected.");  
 }  
  
 public void Recharge()  
 {  
 if (this.\_isConnected)  
 {  
 Console.WriteLine("Apple phone recharging.");  
 }  
 else  
 {  
 Console.WriteLine("Connect the Lightning cable first.");  
 }  
 }  
 }  
  
 private class LightningToUsbAdapter : IUsbPhone  
 {  
 private readonly ILightningPhone \_lightningPhone;  
   
 private bool \_isConnected;  
   
 public LightningToUsbAdapter(ILightningPhone lightningPhone)  
 {  
 this.\_lightningPhone = lightningPhone;  
 }  
   
 public void ConnectUsb()  
 {  
 this.\_isConnected = true;

this.\_lightningPhone.ConnectLightning();  
 Console.WriteLine("Adapter cable connected.");  
 }  
  
 public void Recharge()  
 {  
 if (this.\_isConnected)  
 {  
 this.\_lightningPhone.Recharge();  
 }  
 else  
 {  
 Console.WriteLine("Connect the USB cable first.");  
 }  
 }  
 }  
  
 public static void Main()  
 {  
 ILightningPhone applePhone = new ApplePhone();  
 IUsbPhone adapterCable = new LightningToUsbAdapter(applePhone);  
 adapterCable.ConnectUsb();  
 adapterCable.Recharge();  
 }  
 }  
}

Также возможно привести диаграмму для данного шаблона проектирования:



1. UML диаграмма, иллюстрирующая паттерн Adapter
2. Поведенческие паттерны (Visitor)

Поведенческий паттерн Visitor позволяет определяет простой и удобный способ выполнения какого-либо действия при заходе в определённую структуру. Позволяет легко и удобно «посещать» элементы структуры данных или просто разные классы, вычисляя разный результат при этом.

Достоинства:

* Гибкость (две разные имплементации посетителя могут совершенно по-разному производить сбор необходимой информации)

Недостатки:

* Скорость работы (нужно постоянно передавать имплементацию посетителя в каждый класс, что снижает скорость работы).

Ниже приведён код программы-примера на языке C#:

namespace ConsoleApp1  
{  
 using System;  
using System.Collections.Generic;  
namespace Visitor.Structural  
{  
 public static class Program  
 {  
 public static void Main(string[] args)  
 {  
 ObjectStructure o = new ObjectStructure();  
 o.Attach(new ConcreteElementA());  
 o.Attach(new ConcreteElementB());  
   
 ConcreteVisitor1 v1 = new ConcreteVisitor1();  
 ConcreteVisitor2 v2 = new ConcreteVisitor2();  
   
 o.Accept(v1);  
 o.Accept(v2);  
   
 Console.ReadKey();  
 }  
 }  
   
   
 public abstract class Visitor  
 {  
 public abstract void VisitConcreteElementA(  
 ConcreteElementA concreteElementA);  
 public abstract void VisitConcreteElementB(  
 ConcreteElementB concreteElementB);  
 }  
   
   
 public class ConcreteVisitor1 : Visitor  
 {  
 public override void VisitConcreteElementA(  
 ConcreteElementA concreteElementA)  
 {  
 Console.WriteLine("{0} visited by {1}",  
 concreteElementA.GetType().Name, this.GetType().Name);  
 }  
 public override void VisitConcreteElementB(  
 ConcreteElementB concreteElementB)  
 {  
 Console.WriteLine("{0} visited by {1}",  
 concreteElementB.GetType().Name, this.GetType().Name);  
 }  
 }  
   
   
 public class ConcreteVisitor2 : Visitor  
 {  
 public override void VisitConcreteElementA(  
 ConcreteElementA concreteElementA)  
 {  
 Console.WriteLine("{0} visited by {1}",  
 concreteElementA.GetType().Name, this.GetType().Name);  
 }  
 public override void VisitConcreteElementB(  
 ConcreteElementB concreteElementB)  
 {  
 Console.WriteLine("{0} visited by {1}",  
 concreteElementB.GetType().Name, this.GetType().Name);  
 }  
 }  
   
 public abstract class Element  
 {  
 public abstract void Accept(Visitor visitor);  
 }  
   
 public class ConcreteElementA : Element  
 {  
 public override void Accept(Visitor visitor)  
 {  
 visitor.VisitConcreteElementA(this);  
 }  
 public void OperationA()  
 {  
 }  
 }  
   
 public class ConcreteElementB : Element  
 {  
 public override void Accept(Visitor visitor)  
 {  
 visitor.VisitConcreteElementB(this);  
 }  
 public void OperationB()  
 {  
 }  
 }  
   
 public class ObjectStructure  
 {  
 List<Element> elements = new List<Element>();  
 public void Attach(Element element)  
 {  
 elements.Add(element);  
 }  
 public void Detach(Element element)  
 {  
 elements.Remove(element);  
 }  
 public void Accept(Visitor visitor)  
 {  
 foreach (Element element in elements)  
 {  
 element.Accept(visitor);  
 }  
 }  
 }  
}  
}

Также возможно привести UML диаграмму, иллюстрирующую данный код:



1. UML диаграмма, иллюстрирующая паттерн Visitor с структурой, хранящей объекты
2. Производящие паттерны (Abstract Factory)

Данный шаблон отвечает за создание классов, которые не обязательно связаны друг между другом определёнными связями.

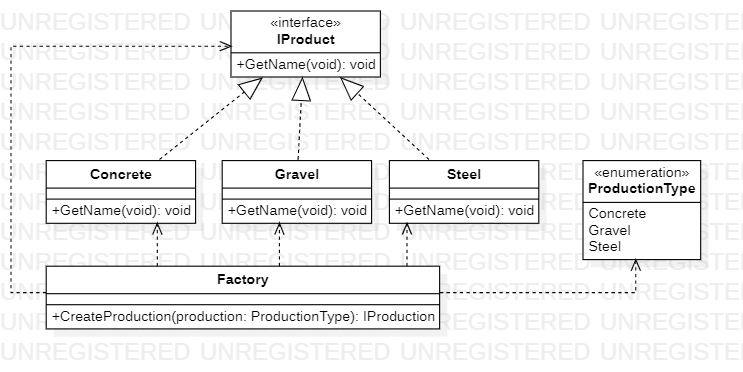
Для данного шаблона сложно выделить какие-то преимущества или недостатки, так как это просто подход к созданию классов. Он может быть применён в системах с весьма ограниченным числом классов, когда классам определённого типа нужно отдать приоритет. Или в системах с закрытым исходным кодом, в которых нельзя создавать объекты классов напрямую.

Но данный код весьма полезен, когда не рекомендуется создавать объекты классов напрямую.

Ниже приведён образец кода на языке C#:

using System;  
  
namespace ConsoleApp1  
{  
 public class Program  
 {  
 public interface IProduction  
 {  
 string GetName();  
 }  
  
 private class ConcreteProduction : IProduction  
 {  
 public string GetName()  
 {  
 return "Concrete";  
 }  
 }  
  
 private class SteelProduction : IProduction  
 {  
 public string GetName()  
 {  
 return "Steel";  
 }  
 }  
   
 private class GravelProduction : IProduction  
 {  
 public string GetName()  
 {  
 return "Gravel";  
 }  
 }  
  
 public enum ProductionType  
 {  
 **Concrete**,  
 **Steel**,  
 **Gravel** }  
  
 public class Factory  
 {  
 public IProduction CreateProduction(ProductionType type)  
 {  
 switch (type)  
 {  
 case ProductionType.**Concrete**:  
 return new ConcreteProduction();  
 case ProductionType.**Gravel**:  
 return new GravelProduction();  
 case ProductionType.**Steel**:  
 return new SteelProduction();  
 default:  
 throw new NotSupportedException();  
 }  
 }  
 }  
  
 public static void Main(string[] args)  
 {  
 Factory factory = new Factory();  
 var concrete = factory.CreateProduction(ProductionType.**Concrete**);  
 var gravel = factory.CreateProduction(ProductionType.**Gravel**);  
 var steel = factory.CreateProduction(ProductionType.**Steel**);  
   
 Console.WriteLine(concrete.GetName());  
 Console.WriteLine(gravel.GetName());  
 Console.WriteLine(steel.GetName());  
 }  
 }  
}

Также возможно привести UML диаграмму, иллюстрирующую данный код:



1. UML диаграмма, иллюстрирующая шаблон abstract factory
2. Паттерны параллельного программирования (lock object)

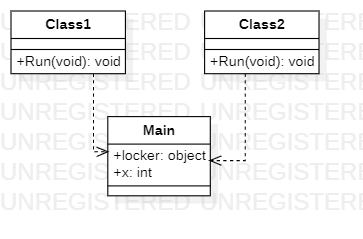
В системах UNIX данный объект обычно называют мьютекс (Mutex). Суть данного объекта блокировки в том, что данный объект должен блокироваться атомарно (за 1 операцию процессора) для того, чтобы избежать обоюдной блокировки (dead-lock).

Ниже приведён образец кода на C#:

using System;  
using System.Threading;  
  
namespace ConsoleApp1  
{   
 public static class Program  
 {  
 static object locker = new();  
 static int x = 0;  
   
 public static void Main(string[] args)  
 {  
 Class1 class1 = new Class1();  
 Class2 class2 = new Class2();  
   
 Thread class1Thread = new(class1.Run);  
 class1Thread.Name = "class1Thread";  
 class1Thread.Start();  
   
 Thread class2Thread = new(class2.Run);  
 class2Thread.Name = "class2Thread";  
 class2Thread.Start();  
   
 }  
  
 private class Class1  
 {  
 public void Run()  
 {  
 lock (locker)  
 {  
 for (x = 1; x < 10; x++)  
 {  
 Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");  
 Thread.Sleep(100);  
 }  
 }  
 }  
 }  
   
 private class Class2  
 {  
 public void Run()  
 {  
 lock (locker)  
 {  
 for (x = 5; x > 0; x--)  
 {  
 Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");  
 Thread.Sleep(100);  
 }  
 }  
 }  
 }  
 }  
}

В данном коде, в качестве общего элемента используется переменная x, которая должна выводится последовательно. (По возрастанию для класса 1, по убыванию для класса 2). В случае, если мы уберем блокирующий элемент, то место последовательного вывода у нас он получится случайным. (в зависимости от того, как планировщик системы будет с этим работать).

Можно привести UML-диаграмму, иллюстрирующую данный код:



1. Диаграмма UML, иллюстрирующая общий доступ двумя потоками к одному объекту.