Design Patterns

with examples in C#

Author D

Паттерны проектирования

с примерами на языке C#

Автор D

2012

Оглавление

Вступление

Представляю вам мануал по паттернам проектирования с примерами на языке C#. Основной материал взят с Википедии http://ru.wikipedia.org/wiki/Design_Patterns .

Целью создания данного мануала послужила потребность в кратком справочнике с ясными примерами на языке C# основных паттернов проектирования.

Связаться со мной можно на моём вебсайте http://go-d.org или по мейлу mail.go.d.org@gmail.com .

Если вам понравился мануал, и вы хотите отблагодарить, просто отправьте https://w.qiwi.com яйцо мне на емейл.

Порождающие шаблоны проектирования

Порождающие шаблоны (англ. Creational patterns) — шаблоны проектирования, которые абстрагируют процесс инстанцирования. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов. Шаблон, порождающий классы, использует наследование, чтобы изменять инстанцируемый класс, а шаблон, порождающий объекты, делегирует инстанцирование другому объекту.

Использование

Эти шаблоны оказываются важны, когда система больше зависит от композиции объектов, чем от наследования классов. Получается так, что основной упор делается не на жестком кодировании фиксированного набора поведений, а на определении небольшого набора фундаментальных поведений, с помощью композиции которых можно получать любое число более сложных. Таким образом, для создания объектов с конкретным поведением требуется нечто большее, чем простое инстанцирование класса.

Порождающие шаблоны инкапсулируют знания о конкретных классах, которые применяются в системе. Они скрывают детали того, как эти классы создаются и стыкуются. Единственная информация об объектах, известная системе, — это их интерфейсы, определенные с помощью абстрактных классов. Следовательно, порождающие шаблоны обеспечивают большую гибкость при решении вопроса о том, что создается, кто это создает, как и когда. Можно собрать систему из «готовых» объектов с самой различной структурой и функциональностью статически (на этапе компиляции) или динамически (во время выполнения).

Иногда допустимо выбирать между тем или иным порождающим шаблоном. Например, есть случаи, когда с пользой для дела можно использовать как прототип, так и абстрактную фабрику. В других ситуациях порождающие шаблоны дополняют друг друга. Так, применяя строитель, можно использовать другие шаблоны для решения вопроса о том, какие компоненты нужно строить, а прототип часто реализуется вместе с одиночкой. Порождающие шаблоны тесно связаны друг с другом, их рассмотрение лучше проводить совместно, чтобы лучше были видны их сходства и различия.

Перечень порождающих шаблонов

- ***абстрактная фабрика (abstract factory);***
- ***строитель (builder);***
- ***фабричный метод (factory method);***
- ***ленивая инициализация (lazy initialization);***
- ***объектный пул (object pool);***
- ***прототип (prototype);***
- ***одиночка (singleton);***
- ***блокировка с двойной проверкой (double checked locking).***

Абстрактная фабрика — Абстрактная фабрика

Абстрактная фабрика (англ. Abstract factory) — порождающий шаблон проектирования, позволяющий изменять поведение системы, варьируя создаваемые объекты, при этом сохраняя интерфейсы. Он позволяет создавать целые группы взаимосвязанных объектов, которые, будучи созданными одной фабрикой, реализуют общее поведение. Шаблон реализуется созданием абстрактного класса Factory, который представляет собой интерфейс для создания компонентов системы (например, для оконного интерфейса он может создавать окна и кнопки). Затем пишутся наследующиеся от него классы, реализующие этот интерфейс.

Цель

Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.

Плюсы

- изолирует конкретные классы;

- упрощает замену семейств продуктов;

- гарантирует сочетаемость продуктов.

Минусы

- сложно добавить поддержку нового вида продуктов.

Применимость

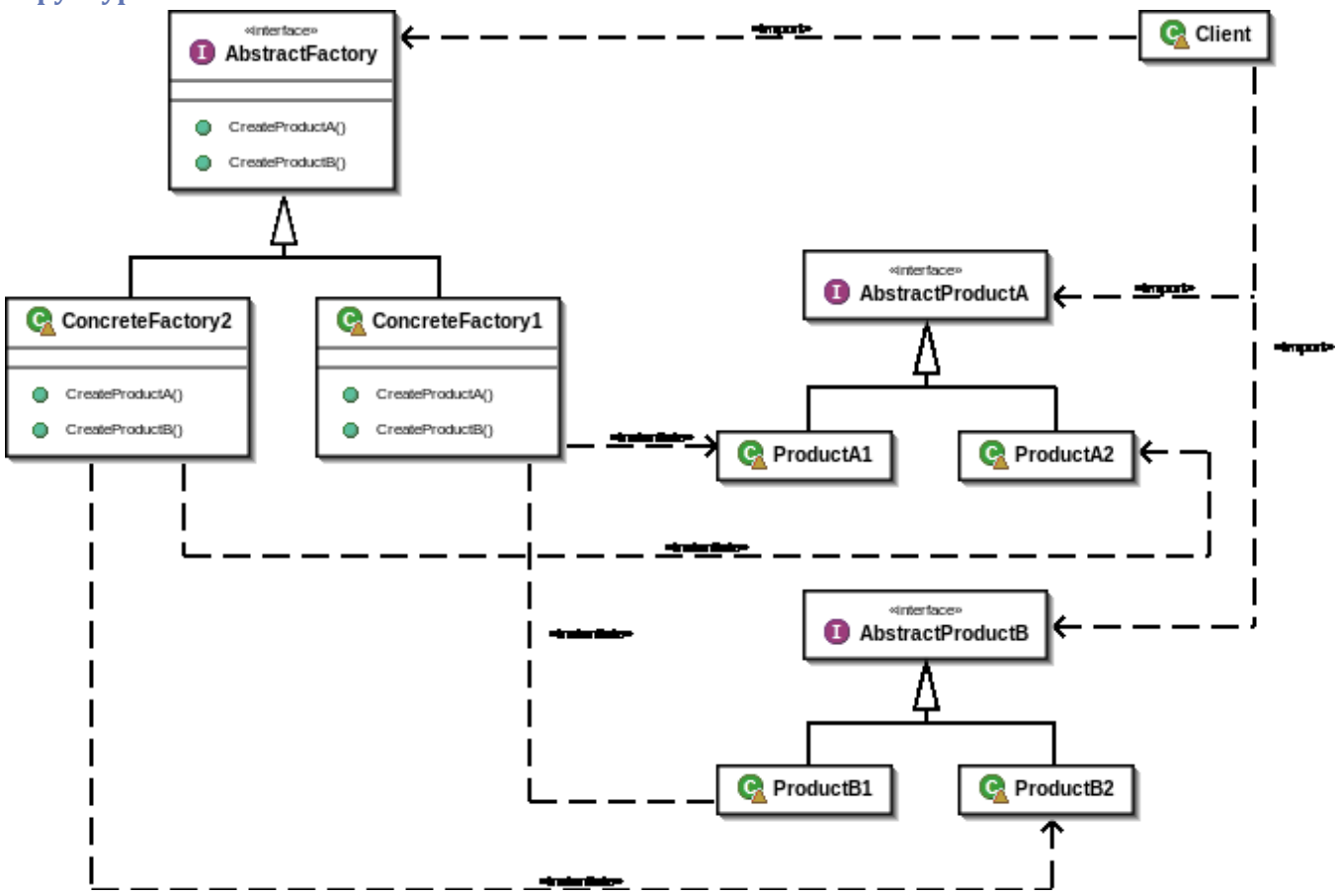
Система не должна зависеть от того, как создаются, компонуются и представляются входящие в нее объекты.

Входящие в семейство взаимосвязанные объекты должны использоваться вместе и вам необходимо обеспечить выполнение этого ограничения.

Система должна конфигурироваться одним из семейств составляющих ее объектов.

Требуется предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию.

Структура



Пример реализации

```
using System;

class MainApp
{
    public static void Main()
    {
        // Abstract factory #1
        AbstractFactory factory1 = new ConcreteFactory1();
        Client c1 = new Client(factory1);
        c1.Run();

        // Abstract factory #2
        AbstractFactory factory2 = new ConcreteFactory2();
        Client c2 = new Client(factory2);
        c2.Run();

        // Wait for user input
        Console.Read();
    }
}


// "AbstractFactory"

abstract class AbstractFactory
{
    public abstract AbstractProductA CreateProductA();
    public abstract AbstractProductB CreateProductB();
}
```

```
// "ConcreteFactory1"

class ConcreteFactory1 : AbstractFactory
{
    public override AbstractProductA CreateProductA()
    {
        return new ProductA1();
    }
    public override AbstractProductB CreateProductB()
    {
        return new ProductB1();
    }
}

// "ConcreteFactory2"

class ConcreteFactory2 : AbstractFactory
{
    public override AbstractProductA CreateProductA()
    {
        return new ProductA2();
    }
    public override AbstractProductB CreateProductB()
    {
        return new ProductB2();
    }
}

// "AbstractProductA"

abstract class AbstractProductA
{
}

// "AbstractProductB"

abstract class AbstractProductB
{
    public abstract void Interact(AbstractProductA a);
}

// "ProductA1"

class ProductA1 : AbstractProductA
{
}

// "ProductB1"

class ProductB1 : AbstractProductB
{
    public override void Interact(AbstractProductA a)
    {
        Console.WriteLine(this.GetType().Name +
          " interacts with " + a.GetType().Name);
    }
}

// "ProductA2"

class ProductA2 : AbstractProductA
{
}

// "ProductB2"
```

```
class ProductB2 : AbstractProductB
{
    public override void Interact(AbstractProductA a)
    {
        Console.WriteLine(this.GetType().Name +
          " interacts with " + a.GetType().Name);
    }
}

// "Client" - the interaction environment of the products

class Client
{
    private AbstractProductA abstractProductA;
    private AbstractProductB abstractProductB;

    // Constructor
    public Client(AbstractFactory factory)
    {
        abstractProductB = factory.CreateProductB();
        abstractProductA = factory.CreateProductA();
    }

    public void Run()
    {
        abstractProductB.Interact(abstractProductA);
    }
}
```

Builder — Строитель

Строитель (англ. Builder) — порождающий шаблон проектирования.

Цель

Отделяет конструирование сложного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться разные представления.

Плюсы

позволяет изменять внутреннее представление продукта;

изолирует код, реализующий конструирование и представление;

дает более тонкий контроль над процессом конструирования.

Применение

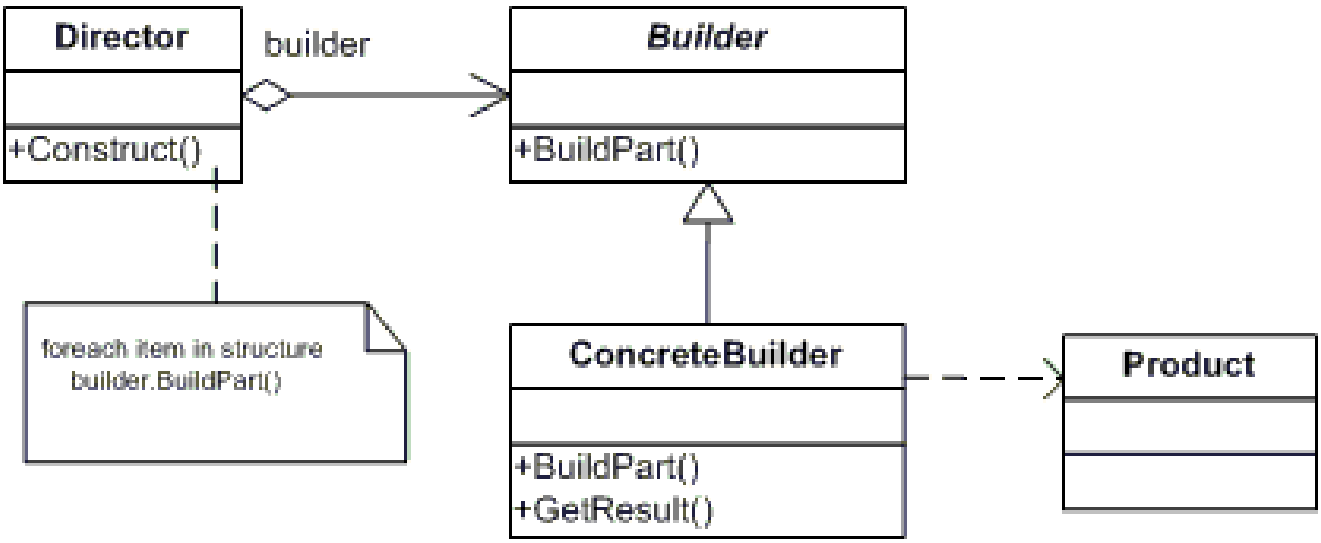
алгоритм создания сложного объекта не должен зависеть от того, из каких частей состоит объект и как они стыкуются между собой;

процесс конструирования должен обеспечивать различные представления конструируемого объекта.

Объекты «моникер» в COM есть Строители, инициализируемые строкой. Более того, для их создания используется другой Строитель — MkParseDisplayNameEx, который определяет по строке класс моникера, создает моникер и инициализирует его этой же строкой.

Один из этих объектов, URL Moniker, используется для всей загрузки страниц, вложений и документов в Microsoft Internet Explorer.

Структура



Пример реализации

```
using System;
using System.Collections.Generic;

namespace Builder
{
    public class MainApp
    {
        public static void Main()
        {
            // Create director and builders
            Director director = new Director();

            Builder b1 = new ConcreteBuilder1();
            Builder b2 = new ConcreteBuilder2();

            // Construct two products
            director.Construct(b1);
            Product p1 = b1.GetResult();
            p1.Show();

            director.Construct(b2);
            Product p2 = b2.GetResult();
            p2.Show();

            // Wait for user
            Console.Read();
        }
    }

    // "Director"

    class Director
    {
        // Builder uses a complex series of steps
        public void Construct(Builder builder)
        {
            builder.BuildPartA();
            builder.BuildPartB();
        }
    }

    // "Builder"

    abstract class Builder
    {
        public virtual void BuildPartA()
        {
        }
        public virtual void BuildPartB()
        {
        }
        public virtual Product GetResult()
        {
        }
    }

    // "ConcreteBuilder1"
    class ConcreteBuilder1 : Builder
    {
        private readonly Product product = new Product();

        public override void BuildPartA()
        {
            product.Add("PartA");
```

```
        }

        public override void BuildPartB()
        {
            product.Add("PartB");
        }

        public override Product GetResult()
        {
            return product;
        }
    }

    // "ConcreteBuilder2"

    class ConcreteBuilder2 : Builder
    {
        private readonly Product product = new Product();

        public override void BuildPartA()
        {
            product.Add("PartX");
        }

        public override void BuildPartB()
        {
            product.Add("PartY");
        }

        public override Product GetResult()
        {
            return product;
        }
    }

    // "Product"

    class Product
    {
        private readonly List<string> parts = new List<string>();

        public void Add(string part)
        {
            parts.Add(part);
        }

        public void Show()
        {
            Console.WriteLine("\nProduct Parts -------");
            foreach (string part in parts)
                Console.WriteLine(part);
        }
    }
}
```

Factory method — Фабричный метод

Фабричный метод (англ. Factory Method) — порождающий шаблон проектирования, предоставляющий подклассам интерфейс для создания экземпляров некоторого класса. В момент создания наследники могут определить, какой класс создавать. Иными словами, Фабрика делегирует создание объектов наследникам родительского класса. Это позволяет использовать в коде программы не специфические классы, а манипулировать абстрактными объектами на более высоком уровне. Также известен под названием виртуальный конструктор **(англ. Virtual Constructor)**.

Цель

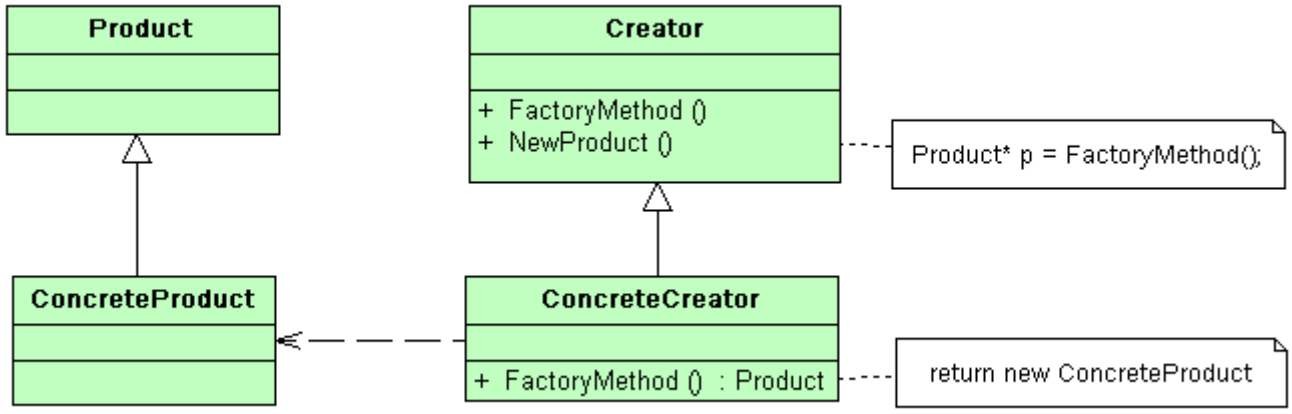
Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанциировать. Фабричный метод позволяет классу делегировать создание подклассов. Используется, когда:

- классу заранее неизвестно, объекты каких подклассов ему нужно создавать.

- класс спроектирован так, чтобы объекты, которые он создаёт, специфицировались подклассами.

Класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и планируется локализовать знание о том, какой класс принимает эти обязанности на себя.

Структура



Product — продукт

определяет интерфейс объектов, создаваемых абстрактным методом;

ConcreteProduct — конкретный продукт

реализует интерфейс Product;

Creator — создатель

объявляет фабричный метод, который возвращает объект типа Product. Может также содержать реализацию этого метода «по умолчанию»;

может вызывать фабричный метод для создания объекта типа Product;

ConcreteCreator — конкретный создатель

переопределяет фабричный метод таким образом, чтобы он создавал и возвращал объект класса ConcreteProduct.

Плюсы

Позволяет сделать код создания объектов более универсальным, не привязываясь к конкретным классам (ConcreteProduct), а оперируя лишь общим интерфейсом (Product);

позволяет установить связь между параллельными иерархиями классов.

Минусы

Необходимость создавать наследника Creator для каждого нового типа продукта (ConcreteProduct).

Пример реализации

```
using System;
using System.Collections.Generic;

namespace Factory
{
    public class MainApp
    {
        public static void Main()
        {
            // an array of creators
            Creator[] creators = { new ConcreteCreatorA(), new ConcreteCreatorB() };

            // iterate over creators and create products
            foreach (Creator creator in creators)
            {
                Product product = creator.FactoryMethod();
                Console.WriteLine("Created {0}", product.GetType());
            }
            // Wait for user
            Console.Read();
        }
    }

    // Product
    abstract class Product
    {
    }

    // "ConcreteProductA"
    class ConcreteProductA : Product
    {
    }

    // "ConcreteProductB"
    class ConcreteProductB : Product
    {
    }

    // "Creator"
    abstract class Creator
    {
        public abstract Product FactoryMethod();
    }

    // "ConcreteCreatorA"
    class ConcreteCreatorA : Creator
    {
        public override Product FactoryMethod()
        {
            return new ConcreteProductA();
        }
    }

    // "ConcreteCreatorB"
    class ConcreteCreatorB : Creator
    {
        public override Product FactoryMethod()
        {
              return new ConcreteProductB();
        }
    }
}
```

Lazy initialization — Ленивая инициализация

Отложенная (ленивая) инициализация (англ. Lazy initialization). Приём в программировании, когда некоторая ресурсоёмкая операция (создание объекта, вычисление значения) выполняется непосредственно перед тем, как будет использован её результат. Таким образом, инициализация выполняется «по требованию», а не заблаговременно. Аналогичная идея находит применение в самых разных областях: например, компиляция «на лету» и логистическая концепция «Точно в срок».

Частный случай ленивой инициализации — создание объекта в момент обращения к нему — является одним из порождающих шаблонов проектирования. Как правило, он используется в сочетании с такими шаблонами как Фабричный метод, Одиночка и Заместитель.Содержание [убрать]

Достоинства

Инициализация выполняется только в тех случаях, когда она действительно необходима;

ускоряется начальная инициализация.

Недостатки

Невозможно явным образом задать порядок инициализации объектов;

возникает задержка при первом обращении к объекту.

Пример реализации

```
public class LazyInitialization<T> where T : new()
{
    protected LazyInitialization() { }

    private static T _instance;
    public static T Instance
    {
        get
        {
            if (_instance == null)
            {
                lock (typeof(T))
                {
                    if (_instance == null)
                        _instance = new T();
                }
            }
            return _instance;
        }
    }
}

public sealed class BigObject: LazyInitialization<BigObject>
{
   public BigObject()
   {
      //Большая работа
      System.Threading.Thread.Sleep(3000);
      System.Console.WriteLine("Экземпляр BigObject создан");
   }
}

class Program
{
    static void Main(string[] args)
    {
      System.Console.WriteLine("Первое обращение к экземпляру BigObject...");
```

```
        //создание объекта происходит только при этом обращении к объекту
        System.Console.WriteLine(BigObject.Instance);

        System.Console.WriteLine("Второе обращение к экземпляру BigObject...");
        System.Console.WriteLine(BigObject.Instance);

        //окончание программы
        System.Console.ReadLine();
    }
}
using System;

public class BigObject
{
   private static BigObject instance;

   private BigObject()
   {
      //Большая работа
      System.Threading.Thread.Sleep(3000);

      Console.WriteLine("Экземпляр BigObject создан");
   }

   public override string ToString()
   {
      return "Обращение к BigObject";
   }

   // Метод возвращает экземпляр BigObject, при этом он
   // создаёт его, если тот ещё не существует
   public static BigObject GetInstance()
   {
      // для исключения возможности создания двух объектов
      // при многопоточном приложении
      if (instance == null)
      {
         lock (typeof(BigObject))
         {
            if (instance == null)
               instance = new BigObject();
         }
      }

      return instance;
   }

   public static void Main()
   {
      Console.WriteLine("Первое обращение к экземпляру BigObject...");

      //создание объекта происходит только при этом обращении к объекту
      Console.WriteLine(BigObject.GetInstance());

      Console.WriteLine("Второе обращение к экземпляру BigObject...");
      Console.WriteLine(BigObject.GetInstance());

      //окончание программы
      Console.ReadLine();
   }
}
```

Object pool — Объектный пул

Объектный пул (англ. object pool) — порождающий шаблон проектирования, набор инициализированных и готовых к использованию объектов. Когда системе требуется объект, он не создаётся, а берётся из пула. Когда объект больше не нужен, он не уничтожается, а возвращается в пул.Содержание [убрать]

Применение

Объектный пул применяется для повышения производительности, когда создание объекта в начале работы и уничтожение его в конце приводит к большим затратам. Особенно заметно повышение производительности, когда объекты часто создаются-уничтожаются, но одновременно существует лишь небольшое их число.

Переполнение

Если в пуле нет ни одного свободного объекта, возможна одна из трёх стратегий:

Расширение пула.

Отказ в создании объекта, аварийный останов.

В случае многозадачной системы, можно подождать, пока один из объектов не освободится.

Примеры

Информация об открытых файлах в DOS.

Информация о видимых объектах во многих компьютерных играх (хорошим примером является движок Doom). Эта информация актуальна только в течение одного кадра; после того, как кадр выведен, список опустошается.

Компьютерная игра для хранения всех объектов на карте, вместо того, чтобы использовать обычные механизмы распределения памяти, может завести массив такого размера, которого заведомо хватит на все объекты, и свободные ячейки держать в виде связного списка. Такая конструкция повышает скорость, уменьшает фрагментацию памяти и снижает нагрузку на сборщик мусора (если он есть).

Ловушки

После того, как объект возвращён, он должен вернуться в состояние, пригодное для дальнейшего использования. Если объекты после возвращения в пул оказываются в неправильном или неопределённом состоянии, такая конструкция называется объектной клоакой (англ. object cesspool).

Повторное использование объектов также может привести к утечке информации. Если в объекте есть секретные данные (например, номер кредитной карты), после освобождения объекта эту информацию надо затереть.

Пример реализации

```
// Пример 1
namespace Digital_Patterns.Creational.Object_Pool.Soft
{
    /// <summary>
    /// Интерфейс для использования шаблона "Object Pool" <see cref="Object_Pool"/>
    /// </summary>
    /// <typeparam name="T"></typeparam>
    public interface ICreation<T>
    {
        /// <summary>
        /// Возвращает вновь созданный объект
        /// </summary>
```

```
        /// <returns></returns>
        T Create();
    }
}

// Пример 2
using System;
using System.Collections;
using System.Threading;

namespace Digital_Patterns.Creational.Object_Pool.Soft
{
    /// <summary>
    /// Реализация пула объектов, использующий "мягкие" ссылки
    /// </summary>
    /// <typeparam name="T"></typeparam>
    public class ObjectPool<T> where T : class
    {
        /// <summary>
        /// Объект синхронизации
        /// </summary>
        private Semaphore semaphore;

        /// <summary>
        /// Коллекция содержит управляемые объекты
        /// </summary>
        private ArrayList pool;

        /// <summary>
        /// Ссылка на объект, которому делегируется ответственность
        /// за создание объектов пула
        /// </summary>
        private ICreation<T> creator;

        /// <summary>
        /// Количество объектов, существующих в данный момент
        /// </summary>
        private Int32 instanceCount;

        /// <summary>
        /// Максимальное количество управляемых пулом объектов
        /// </summary>
        private Int32 maxInstances;

        /// <summary>
        /// Создание пула объектов
        /// </summary>
        /// <param name="creator">Объект, которому пул будет делегировать ответственность
        /// за создание управляемых им объектов</param>
        public ObjectPool(ICreation<T> creator)
            : this(creator, Int32.MaxValue)
        {
        }

        /// <summary>
        /// Создание пула объектов
        /// </summary>
        /// <param name="creator">Объект, которому пул будет делегировать ответственность
        /// за создание управляемых им объектов</param>
        /// <param name="maxInstances">Максимальное количество экземпляров класс,
        /// которым пул разрешает существовать одновременно
        /// </param>
        public ObjectPool(ICreation<T> creator, Int32 maxInstances)
        {
            this.creator = creator;
```

```
    this.instanceCount = 0;
    this.maxInstances = maxInstances;
    this.pool = new ArrayList();
    this.semaphore = new Semaphore(0, this.maxInstances);
}

/// <summary>
/// Возвращает количество объектов в пуле, ожидающих повторного
/// использования. Реальное количество может быть меньше
/// этого значения, поскольку возвращаемая
/// величина - это количество "мягких" ссылок в пуле.
/// </summary>
public Int32 Size
{
    get
    {
        lock(pool)
        {
            return pool.Count;
        }
    }
}

/// <summary>
/// Возвращает количество управляемых пулом объектов,
/// существующих в данный момент
/// </summary>
public Int32 InstanceCount { get { return instanceCount; } }

/// <summary>
/// Получить или задать максимальное количество управляемых пулом
/// объектов, которым пул разрешает существовать одновременно.
/// </summary>
public Int32 MaxInstances
{
    get { return maxInstances; }
    set { maxInstances = value; }
}

/// <summary>
/// Возвращает из пула объект. При пустом пуле будет создан
/// объект, если количество управляемых пулом объектов не
/// больше или равно значению, возвращаемому методом
/// <see cref="ObjectPool{T}.MaxInstances"/>. Если количество управляемых пулом
/// объектов превышает это значение, то данный метод возваращает null
/// </summary>
/// <returns></returns>
public T GetObject()
{
    lock(pool)
    {
        T thisObject = RemoveObject();
        if (thisObject != null)
            return thisObject;

        if (InstanceCount < MaxInstances)
            return CreateObject();

        return null;
    }
}

/// <summary>
/// Возвращает из пула объект. При пустом пуле будет создан
/// объект, если количество управляемых пулом объектов не
```

```
/// больше или равно значению, возвращаемому методом
/// <see cref="ObjectPool{T}.MaxInstances"/>. Если количество управляемых пулом
/// объектов превышает это значение, то данный метод будет ждать до тех
/// пор, пока какой-нибудь объект не станет доступным для
/// повторного использования.
/// </summary>
/// <returns></returns>
public T WaitForObject()
{
    lock(pool)
    {
        T thisObject = RemoveObject();
        if (thisObject != null)
            return thisObject;

        if (InstanceCount < MaxInstances)
            return CreateObject();
    }
    semaphore.WaitOne();
    return WaitForObject();
}



/// <summary>
/// Удаляет объект из коллекции пула и возвращает его
/// </summary>
/// <returns></returns>
private T RemoveObject()
{
    while (pool.Count > 0 )
    {
        var refThis = (WeakReference) pool[pool.Count - 1];
        pool.RemoveAt(pool.Count - 1);
        var thisObject = (T)refThis.Target;
        if (thisObject != null)
            return thisObject;
        instanceCount--;
    }
    return null;
}

/// <summary>
/// Создать объект, управляемый этим пулом
/// </summary>
/// <returns></returns>
private T CreateObject()
{
    T newObject = creator.Create();
    instanceCount++;
    return newObject;
}

/// <summary>
/// Освобождает объект, помещая его в пул для
/// повторного использования
/// </summary>
/// <param name="obj"></param>
/// <exception cref="NullReferenceException"></exception>
public void Release(T obj)
{
    if(obj == null)
        throw new NullReferenceException();
    lock(pool)
    {
```

```
                var refThis = new WeakReference(obj);
                pool.Add(refThis);
                semaphore.Release();
            }
        }
    }
}

// Пример 3
namespace Digital_Patterns.Creational.Object_Pool.Soft
{
    public class Reusable
    {
        public Object[] Objs { get; protected set; }

        public Reusable(params Object[] objs)
        {
            this.Objs = objs;
        }
    }

    public class Creator : ICreation<Reusable>
    {
        private static Int32 iD = 0;

        public Reusable Create()
        {
            ++iD;
            return new Reusable(iD);
        }
    }

    public class ReusablePool : ObjectPool<Reusable>
    {
        public ReusablePool()
            : base(new Creator(), 2)
        {

        }
    }
}

// Пример 4
using System;
using System.Threading;
using Digital_Patterns.Creational.Object_Pool.Soft;

namespace Digital_Patterns
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(System.Reflection.MethodInfo.GetCurrentMethod().Name);
            var reusablePool = new ReusablePool();

            var thrd1 = new Thread(Run);
            var thrd2 = new Thread(Run);
            var thisObject1 = reusablePool.GetObject();
            var thisObject2 = reusablePool.GetObject();
            thrd1.Start(reusablePool);
            thrd2.Start(reusablePool);
            ViewObject(thisObject1);
            ViewObject(thisObject2);
            Thread.Sleep(2000);
```

```
            reusablePool.Release(thisObject1);
            Thread.Sleep(2000);
            reusablePool.Release(thisObject2);

            Console.ReadKey();
        }

        private static void Run(Object obj)
        {
            Console.WriteLine("\t" + System.Reflection.MethodInfo.GetCurrentMethod().Name);
            var reusablePool = (ReusablePool)obj;
            Console.WriteLine("\tstart wait");
            var thisObject1 = reusablePool.WaitForObject();
            ViewObject(thisObject1);
            Console.WriteLine("\tend wait");
            reusablePool.Release(thisObject1);
        }

        private static void ViewObject(Reusable thisObject)
        {
            foreach (var obj in thisObject.Objs)
            {
                Console.Write(obj.ToString() + @" ");
            }
            Console.WriteLine();
        }
    }
}
```

Prototype — Прототип

Прототип, (англ. Prototype) — порождающий шаблон проектирования.Содержание [убрать]

Назначение

Задаёт виды создаваемых объектов с помощью экземпляра-прототипа и создаёт новые объекты путём копирования этого прототипа.

Проще говоря, это паттерн создания объекта через клонирование другого объекта вместо создания через конструктор.

Применимость

Паттерн используется чтобы:

- избежать дополнительных усилий по созданию объекта стандартным путем (имеется в виду использование ключевого слова 'new', когда вызывается конструктор не только самого объекта, но и конструкторы всей иерархии предков объекта), когда это непозволительно дорого для приложения.

- избежать наследования создателя объекта (object creator) в клиентском приложении, как это делает паттерн abstract factory.

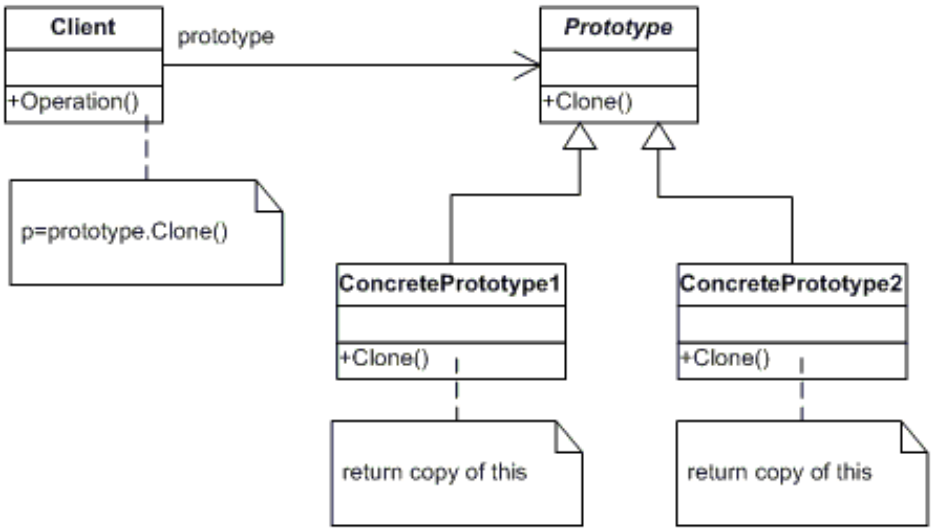
Используйте этот шаблон проектирования, когда система не должна зависеть от того, как в ней создаются, компонуются и представляются продукты:

- инстанцируемые классы определяются во время выполнения, например с помощью динамической загрузки;

- для того чтобы избежать построения иерархий классов или фабрик, параллельных иерархии классов продуктов;

- экземпляры класса могут находиться в одном из нескольких различных состояний. Может оказаться удобнее установить соответствующее число прототипов и клонировать их, а не инстанцировать каждый раз класс вручную в подходящем состоянии.

Структура



Пример реализации

```
using System;

namespace Prototype
{
    class MainApp
    {
        static void Main()
        {
            // Create two instances and clone each

            Prototype p1 = new ConcretePrototype1("I");
            Prototype c1 = p1.Clone();
            Console.WriteLine("Cloned: {0}", c1.Id);

            Prototype p2 = new ConcretePrototype2("II");
            Prototype c2 = p2.Clone();
            Console.WriteLine("Cloned: {0}", c2.Id);

            // Wait for user
            Console.Read();
        }
    }

    // "Prototype"

    abstract class Prototype
    {
        private string id;

        // Constructor
        public Prototype(string id)
        {
            this.id = id;
        }

        // Property
        public string Id
        {
            get
            {
                return id;
            }
        }

        public abstract Prototype Clone();
    }

    // "ConcretePrototype1"

    class ConcretePrototype1 : Prototype
    {
        // Constructor
        public ConcretePrototype1(string id)
            : base(id)
        {
        }

        public override Prototype Clone()
        {
            // Shallow copy
            return (Prototype)this.MemberwiseClone();
        }
    }
```

```
    // "ConcretePrototype2"

    class ConcretePrototype2 : Prototype
    {
        // Constructor
        public ConcretePrototype2(string id)
            : base(id)
        {
        }

        public override Prototype Clone()
        {
            // Shallow copy
            return (Prototype)this.MemberwiseClone();
        }
    }
}
```

Singleton — Одиночка

Одиночка (англ. Singleton) в программировании — порождающий шаблон проектирования.

Цель

Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа. Существенно то, что можно пользоваться именно экземпляром класса, так как при этом во многих случаях становится доступной более широкая функциональность. Например, к описанным компонентам класса можно обращаться через интерфейс, если такая возможность поддерживается языком.

Плюсы

контролируемый доступ к единственному экземпляру;

уменьшение числа имён;

допускает уточнение операций и представления;

допускает переменное число экземпляров;

бóльшая гибкость, чем у операций класса.

Минусы

глобальные объекты могут быть вредны для объектного программирования, в некоторых случаях приводя к созданию немасштабируемого проекта.

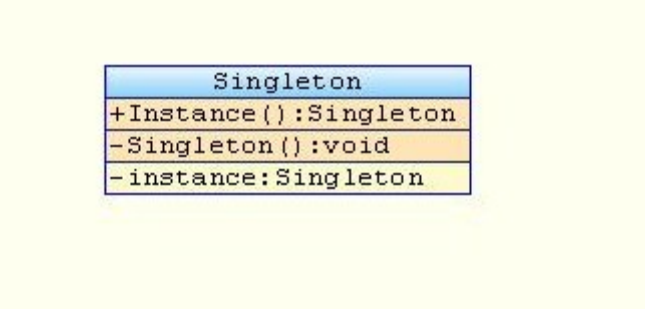
усложняет написание модульных тестов и следование TDD

Применение

должен быть ровно один экземпляр некоторого класса, легко доступный всем клиентам;

единственный экземпляр должен расширяться путем порождения подклассов, и клиентам нужно иметь возможность работать с расширенным экземпляром без модификации своего кода.

Структура



Пример реализации

Для отложенной инициализации Singleton'а в C# рекомендуется использовать конструкторы типов (статический конструктор). CLR автоматически вызывает конструктор типа при первом обращении к типу, при этом обеспечивая безопасность в отношении синхронизации потоков. Конструктор типа автоматически генерируется компилятором и в нем происходит инициализация всех полей типа (статических полей). Явно задавать конструктор типа не следует, так как в этом случае он будет вызываться непосредственно перед обращением к типу и JIT-компилятор не сможет применить оптимизацию (например, если первое обращение к Singleton'у происходит в цикле).

```
/// generic Singleton<T> (потокобезопасный с использованием generic-класса и с отложенной инициализацией)

/// <typeparam name="T">Singleton class</typeparam>
public class Singleton<T> where T : class
{
    /// Защищённый конструктор необходим для того, чтобы предотвратить создание экземпляра класса Singleton.
    /// Он будет вызван из закрытого конструктора наследственного класса.
    protected Singleton()
    {
    }

    /// Фабрика используется для отложенной инициализации экземпляра класса
    private sealed class SingletonCreator<S> where S : class
    {
        //Используется Reflection для создания экземпляра класса без публичного конструктора
        private static readonly S instance = (S)typeof(S).GetConstructor(
                    BindingFlags.Instance | BindingFlags.NonPublic,
                    null,
                    new Type[0],
                    new ParameterModifier[0]).Invoke(null);

        public static S CreatorInstance
        {
            get
            {
                return instance;
            }
        }
    }

    public static T Instance
    {
        get
        {
            return SingletonCreator<T>.CreatorInstance;
        }
    }

}

/// Использование Singleton
public class TestClass : Singleton<TestClass>
{
    /// Вызовет защищенный конструктор класса Singleton
    private TestClass()
    {
    }

    public string TestProc()
    {
        return "Hello World";
    }
}
```

Также можно использовать стандартный вариант потокобезопасной реализации Singleton с отложенной инициализацией:

```
public class Singleton
{
```

```
    /// Защищенный конструктор нужен, чтобы предотвратить создание экземпляра класса Singleton
    protected Singleton()
    {
    }

    private sealed class SingletonCreator
    {
        private static readonly Singleton instance = new Singleton();
        public static Singleton Instance
        {
            get
            {
                return instance;
            }
        }
    }

    public static Singleton Instance
    {
        get
        {
            return SingletonCreator.Instance;
        }
    }

}
```

Если нет необходимости в каких-либо публичных статических методах или свойствах (кроме свойства Instance), то можно использовать упрощенный вариант:

```
public class Singleton
{
  private static readonly Singleton instance = new Singleton();

  public static Singleton Instance
  {
    get { return instance; }
  }

  /// Защищенный конструктор нужен, чтобы предотвратить создание экземпляра класса Singleton
  protected Singleton() { }
}
```

Double checked locking - Блокировка с двойной проверкой

Double checked locking (блокировка с двойной проверкой) — шаблон проектирования, применяющийся в параллельном программировании. Он предназначен для уменьшения накладных расходов, связанных с получением блокировки. Сначала проверяется условие блокировки без какой-либо синхронизации; поток делает попытку получить блокировку только если результат проверки говорит о том, что ни один другой поток не владеет блокировкой.

На некоторых языках и/или на некоторых машинах невозможно безопасно реализовать данный шаблон. Поэтому иногда его называют **анти-паттерном**.

Обычно он используется для уменьшения накладных расходов при реализации ленивой инициализации в многопоточных программах, например в составе шаблона проектирования Одиночка. При ленивой инициализации переменной, инициализация откладывается до тех пор, пока значение переменной не понадобится при вычислениях.

Пример реализации

Microsoft подтверждает, что при использовании ключевого слова *volatile*, использование паттерна Double checked locking является безопасным.

```
public sealed class Singleton
{
    private Singleton()
    {
        // инициализировать новый экземпляр объекта
    }

    private static volatile Singleton singletonInstance;

    private static readonly Object syncRoot = new Object();

    public static Singleton GetInstance()
    {
        // создан ли объект
        if (singletonInstance == null)
        {
            // нет, не создан
            // только один поток может создать его
            lock (syncRoot)
            {
                // проверяем, не создал ли объект другой поток
                if (singletonInstance == null)
                {
                    // нет не создал – создаём
                    singletonInstance = new Singleton();
                }
            }
        }
        return singletonInstance;
    }
}
```

Структурные шаблоны проектирования

Структурные шаблоны — шаблоны проектирования, в которых рассматривается вопрос о том, как из классов и объектов образуются более крупные структуры.

Использование

Структурные шаблоны уровня класса используют наследование для составления композиций из интерфейсов и реализаций. Простой пример — использование множественного наследования для объединения нескольких классов в один. В результате получается класс, обладающий свойствами всех своих родителей. Особенно полезен этот шаблон, когда нужно организовать совместную работу нескольких независимо разработанных библиотек.

Перечень структурных шаблонов

- **адаптер (adapter);**
- **мост (bridge);**
- **компоновщик (composite patern);**
- **декоратор (decorator);**
- **фасад (facade);**
- **front controller;**
- **приспособленец (flyweight);**
- **заместитель (proxy).**

Front Controller — Входная точка

The Front Controller Pattern is a software design pattern listed in several pattern catalogs. The pattern relates to the design of web applications. It "provides a centralized entry point for handling requests."

Front controllers are often used in web applications to implement workflows. While not strictly required, it is much easier to control navigation across a set of related pages (for instance, multiple pages might be used in an online purchase) from a front controller than it is to make the individual pages responsible for navigation.

The front controller may be implemented as a Java object, or as a script in a script language like PHP, ASP, CFML or JSP that is called on every request of a web session. This script, for example an index.php, would handle all tasks that are common to the application or the framework, such as session handling, caching, and input filtering. Based on the specific request it would then instantiate further objects and call methods to handle the particular task(s) required.

The alternative to a front controller would be individual scripts like login.php and order.php that would each then satisfy the type of request. Each script would have to duplicate code or objects that are common to all tasks. But each script might also have more flexibility to implement the particular task required.Contents [hide]

Пример

Several web-tier application frameworks implement the Front Controller pattern, among them:

- Ruby on Rails

- ColdBox, a ColdFusion MVC framework.

- Spring MVC, a Java MVC framework

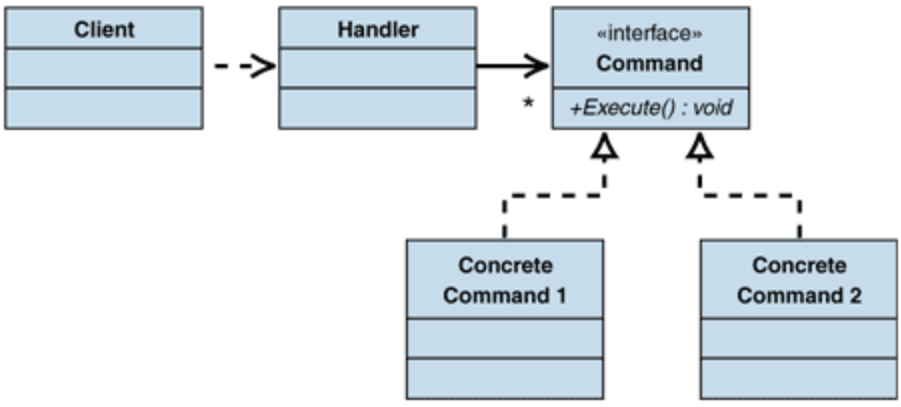
- Yii, Cake, Symfony, Kohana, CodeIgniter and Zend Framework, MVC frameworks written with PHP

- Cairngorm framework in Adobe Flex.

- Microsoft's ASP.NET MVC Framework.

- Yesod web application framework written in Haskell.

Структура



Adapter — Адаптер

Адаптер, Adapter или Wrapper/Обёртка — структурный шаблон проектирования, предназначенный для организации использования функций объекта, недоступного для модификации, через специально созданный интерфейс.

Задача

Система поддерживает требуемые данные и поведение, но имеет неподходящий интерфейс. Чаще всего шаблон Адаптер применяется, если необходимо создать класс, производный от вновь определяемого или уже существующего абстрактного класса.

Способ решения

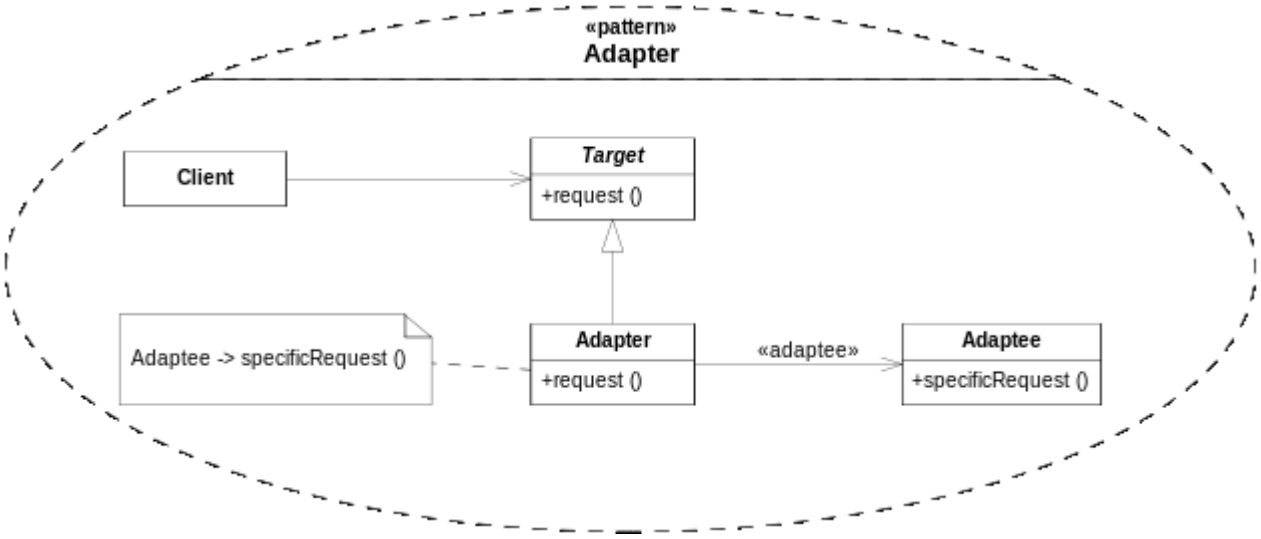
Адаптер предусматривает создание класса-оболочки с требуемым интерфейсом.

Участники

Класс Adapter приводит интерфейс класса Adaptee в соответствие с интерфейсом класса Target (наследником которого является Adapter). Это позволяет объекту Client использовать объект Adaptee (посредством адаптера Adapter) так, словно он является экземпляром класса Target.

Таким образом Client обращается к интерфейсу Target, реализованному в наследнике Adapter, который перенаправляет обращение к Adaptee.

Структура



Следствия

Шаблон Адаптер позволяет включать уже существующие объекты в новые объектные структуры, независимо от различий в их интерфейсах.

Реализация

Включение уже существующего класса в другой класс. Интерфейс включающего класса приводится в соответствие с новыми требованиями, а вызовы его методов преобразуются в вызовы методов включённого класса.

Пример реализации

```
using System;

namespace Adapter
{

    class MainApp
    {
        static void Main()
        {
            // Create adapter and place a request
            Target target = new Adapter();
            target.Request();

            // Wait for user
            Console.Read();
        }
    }

    // "Target"

    class Target
    {
        public virtual void Request()
        {
            Console.WriteLine("Called Target Request()");
        }
    }

    // "Adapter"

    class Adapter : Target
    {
        private Adaptee adaptee = new Adaptee();

        public override void Request()
        {
            // Possibly do some other work
            // and then call SpecificRequest
            adaptee.SpecificRequest();
        }
    }

    // "Adaptee"

    class Adaptee
    {
        public void SpecificRequest()
        {
            Console.WriteLine("Called SpecificRequest()");
        }
    }
}
```

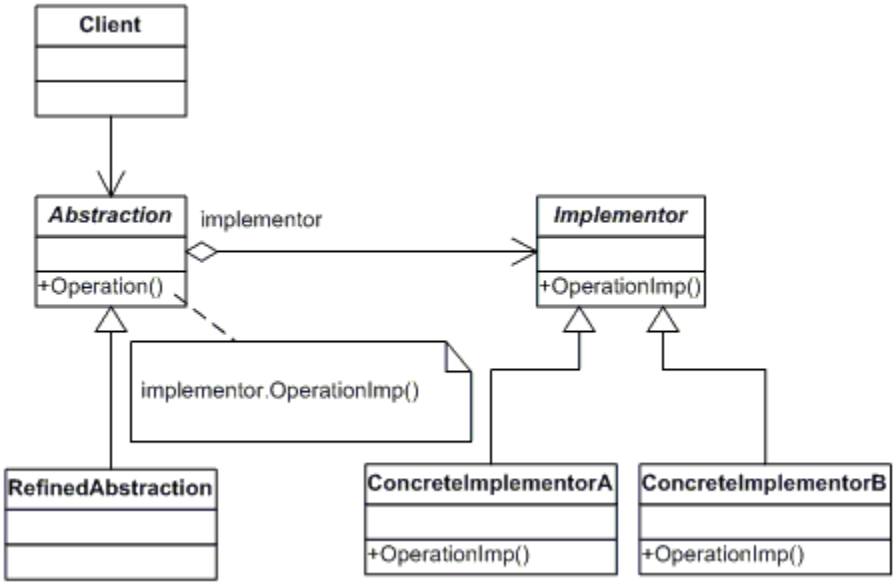
Bridge — Мост

Bridge, Мост — шаблон проектирования, используемый в проектировании программного обеспечения чтобы «разделять абстракцию и реализацию так, чтобы они могли изменяться независимо». Шаблон bridge (от англ. — мост) использует инкапсуляцию, агрегирование и может использовать наследование для того, чтобы разделить ответственность между классами.

Цель

При частом изменении класса преимущества объектно-ориентированного подхода становятся очень полезными, позволяя делать изменения в программе, обладая минимальными сведениями о реализации программы. Шаблон bridge является полезным там, где часто меняется не только сам класс, но и то, что он делает.

Структура

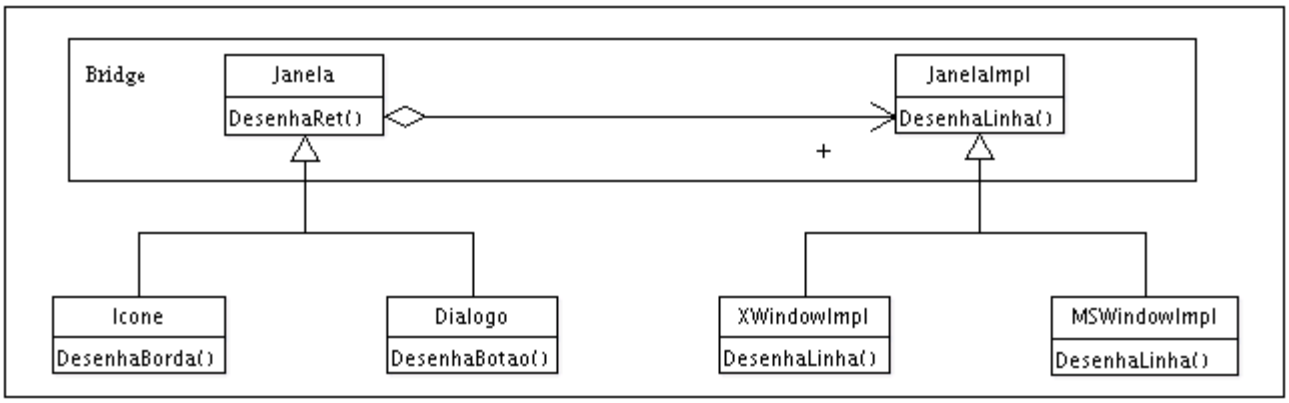


Описание

Когда абстракция и реализация разделены, они могут изменяться независимо. Другими словами, при реализации через паттерн мост, изменение структуры интерфейса не мешает изменению структуры реализации. Рассмотрим такую абстракцию как фигура. Существует множество типов фигур, каждая со своими свойствами и методами. Однако есть что-то, что объединяет все фигуры. Например, каждая фигура должна уметь рисовать себя, масштабироваться и т. п. В то же время рисование графики может отличаться в зависимости от типа ОС, или графической библиотеки. Фигуры должны иметь возможность рисовать себя в различных графических средах, но реализовывать в каждой фигуре все способы рисования или модифицировать фигуру каждый раз при изменении способа рисования непрактично. В этом случае помогает шаблон bridge, позволяя создавать новые классы, которые будут реализовывать рисование в различных графических средах. При использовании такого подхода очень легко можно добавлять как новые фигуры, так и способы их рисования.

Связь, изображаемая стрелкой на диаграммах, может иметь 2 смысла: а) "разновидность", в соответствии с принципом подстановки Б. Лисков и б) одна из возможных реализаций абстракции. Обычно в языках используется наследование для реализации как а), так и б), что приводит к разбуханию иерархий классов.

Мост служит именно для решения этой проблемы: объекты создаются парами из объекта класса иерархии A и иерархии B, наследование внутри иерархии A имеет смысл "разновидность" по Лисков, а для понятия "реализация абстракции" используется ссылка из объекта A в парный ему объект B.



Использование

Архитектура Java AWT полностью основана на этом паттерне - иерархия java.awt.xxx для хэндлов и sun.awt.xxx для реализаций.

Пример реализации

```
using System;

namespace Bridge
{

    // MainApp test application

    class MainApp
    {
        static void Main()
        {
            Abstraction ab = new RefinedAbstraction();

            // Set implementation and call
            ab.Implementor = new ConcreteImplementorA();
            ab.Operation();

            // Change implementation and call
            ab.Implementor = new ConcreteImplementorB();
            ab.Operation();

            // Wait for user
            Console.Read();
        }
    }

    /// <summary>
    /// Abstraction - абстракция
    /// </summary>
    /// <remarks>
    /// <li>
    /// <lu>определяем интерфейс абстракции;</lu>
    /// <lu>хранит ссылку на объект <see cref="Implementor"/></lu>
    /// </li>
    /// </remarks>
    class Abstraction
    {
        protected Implementor implementor;
```

```
    // Property
    public Implementor Implementor
    {
        get
        {
            return implementor;
        }
        set
        {
            implementor = value;
        }
    }

    public virtual void Operation()
    {
        implementor.Operation();
    }
}

/// <summary>
/// Implementor - реализатор
/// </summary>
/// <remarks>
/// <li>
/// <lu>определяет интерфейс для классов реализации. Он не обязан точно
/// соотведствовать интерфейсу класса <see cref="Abstraction"/>. На самом деле оба
/// интерфейса могут быть совершенно различны. Обычно интерфейс класса
/// <see cref="Implementor"/> представляет только примитивные операции, а класс
/// <see cref="Abstraction"/> определяет операции более высокого уровня,
/// базирующиеся на этих примитивах;</lu>
/// </li>
/// </remarks>
abstract class Implementor
{
    public abstract void Operation();
}

/// <summary>
/// RefinedAbstraction - уточненная абстракция
/// </summary>
/// <remarks>
/// <li>
/// <lu>расширяет интерфейс, определенный абстракцией <see cref="Abstraction"/></lu>
/// </li>
/// </remarks>
class RefinedAbstraction : Abstraction
{
    public override void Operation()
    {
        implementor.Operation();
    }
}

/// <summary>
/// ConcreteImplementor - конкретный реализатор
/// </summary>
/// <remarks>
/// <li>
/// <lu>содержит конкретную реализацию интерфейса <see cref="Implementor"/></lu>
/// </li>
/// </remarks>
class ConcreteImplementorA : Implementor
{
    public override void Operation()
    {
```

```
            Console.WriteLine("ConcreteImplementorA Operation");
        }
    }

    // "ConcreteImplementorB"

    class ConcreteImplementorB : Implementor
    {
        public override void Operation()
        {
            Console.WriteLine("ConcreteImplementorB Operation");
        }
    }
}
```

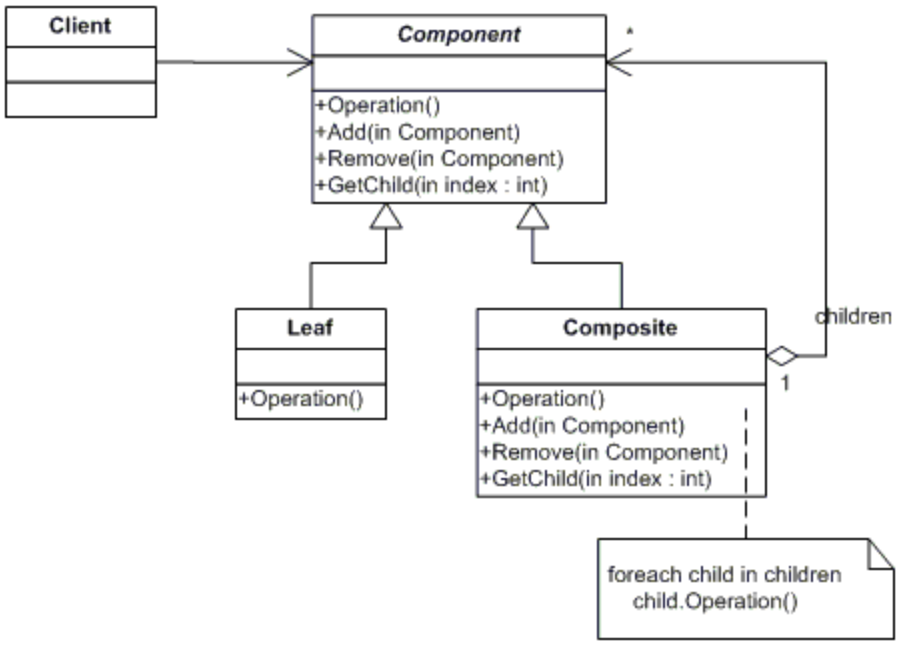
Composite — Компоновщик

Компоновщик (англ. Composite pattern) — шаблон проектирования, относится к структурным паттернам, объединяет объекты в древовидную структуру для представления иерархии от частного к целому. Компоновщик позволяет клиентам обращаться к отдельным объектам и к группам объектов одинаково.

Цель

Паттерн определяет иерархию классов, которые одновременно могут состоять из примитивных и сложных объектов, упрощает архитектуру клиента, делает процесс добавления новых видов объекта более простым.

Структура



Пример реализации

```
class MainApp
{
    static void Main()
    {

        // Create a tree structure
        Component root = new Composite("root");

        root.Add(new Leaf("Leaf A"));
        root.Add(new Leaf("Leaf B"));

        Component comp = new Composite("Composite X");

        comp.Add(new Leaf("Leaf XA"));
        comp.Add(new Leaf("Leaf XB"));
        root.Add(comp);
        root.Add(new Leaf("Leaf C"));

        // Add and remove a leaf
        Leaf leaf = new Leaf("Leaf D");
        root.Add(leaf);
        root.Remove(leaf);

        // Recursively display tree
        root.Display(1);
```

```
        // Wait for user
        Console.Read();

    }
}

/// <summary>
/// Component - компонент
/// </summary>
/// <li>
/// <lu>объявляет интерфейс для компонуемых объектов;</lu>
/// <lu>предоставляет подходящую реализацию операций по умолчанию,
/// общую для всех классов;</lu>
/// <lu>объявляет интерфейс для доступа к потомкам и управлению ими;</lu>
/// <lu>определяет интерфейс доступа к родителю компонента в рекурсивной структуре
/// и при необходимости реализует его. Описанная возможность необязательна;</lu>
/// </li>
abstract class Component
{
    protected string name;

    // Constructor
    public Component(string name)
    {
        this.name = name;
    }

    public abstract void Add(Component c);
    public abstract void Remove(Component c);
    public abstract void Display(int depth);
}

/// <summary>
/// Composite - составной объект
/// </summary>
/// <li>
/// <lu>определяет поведеление компонентов, у которых есть потомки;</lu>
/// <lu>хранит компоненты-потомоки;</lu>
/// <lu>реализует относящиеся к управлению потомками операции и интерфейсе
/// класса <see cref="Component"/></lu>
/// </li>
class Composite : Component
{
    private ArrayList children = new ArrayList();

    // Constructor
    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(int depth)
    {
        Console.WriteLine(new String('-', depth) + name);
```

```
        // Recursively display child nodes
        foreach (Component component in children)
        {
            component.Display(depth + 2);
        }
    }
}

/// <summary>
/// Leaf - лист
/// </summary>
/// <remarks>
/// <li>
/// <lu>представляет листовой узел композиции и не имеет потомков;</lu>
/// <lu>определяет поведение примитивных объектов в композиции;</lu>
/// </li>
/// </remarks>
class Leaf : Component
{

    // Constructor
    public Leaf(string name)
        : base(name)
    {
    }

    public override void Add(Component c)
    {
        Console.WriteLine("Cannot add to a leaf");
    }

    public override void Remove(Component c)
    {
        Console.WriteLine("Cannot remove from a leaf");
    }

    public override void Display(int depth)
    {
        Console.WriteLine(new String('-', depth) + name);
    }
}
```

Decorator — Декоратор

Декоратор, Decorator — структурный шаблон проектирования, предназначенный для динамического подключения дополнительного поведения к объекту. Шаблон Декоратор предоставляет гибкую альтернативу практике создания подклассов с целью расширения функциональности.

Известен также под менее распространённым названием Обёртка (Wrapper), которое во многом раскрывает суть реализации шаблона.

Задача

Объект, который предполагается использовать, выполняет основные функции. Однако может потребоваться добавить к нему некоторую дополнительную функциональность, которая будет выполняться до, после или даже вместо основной функциональности объекта.

Способ решения

Декоратор предусматривает расширение функциональности объекта без определения подклассов.

Участники

Класс ConcreteComponent — класс, в который с помощью шаблона Декоратор добавляется новая функциональность. В некоторых случаях базовая функциональность предоставляется классами, производными от класса ConcreteComponent. В подобных случаях класс ConcreteComponent является уже не конкретным, а абстрактным. Абстрактный класс Component определяет интерфейс для использования всех этих классов.

Следствия

1. Добавляемая функциональность реализуется в небольших объектах. Преимущество состоит в возможности динамически добавлять эту функциональность до или после основной функциональности объекта ConcreteComponent.

2. Позволяет избегать перегрузки функциональными классами на верхних уровнях иерархии

3. Декоратор и его компоненты не являются идентичными

Реализация

Создается абстрактный класс, представляющий как исходный класс, так и новые, добавляемые в класс функции. В классах-декораторах новые функции вызываются в требуемой последовательности — до или после вызова последующего объекта.

При желании остаётся возможность использовать исходный класс (без расширения функциональности), если на его объект сохранилась ссылка.

Замечания и комментарии

Хотя объект-декоратор может добавлять свою функциональность до или после функциональности основного объекта, цепочка создаваемых объектов всегда должна заканчиваться объектом класса ConcreteComponent.

Базовые классы языка Java широко используют шаблон Декоратор для организации обработки операций ввода-вывода.

И декоратор, и адаптер являются обертками вокруг объекта — хранят в себе ссылку на оборачиваемый объект и часто передают в него вызовы методов. Отличие декоратора от адаптера в том, что адаптер имеет внешний интерфейс, отличный от интерфейса оборачиваемого объекта, и используется именно для

стыковки разных интерфейсов. Декоратор же имеет точно такой же интерфейс, и используется для добавления функциональности.

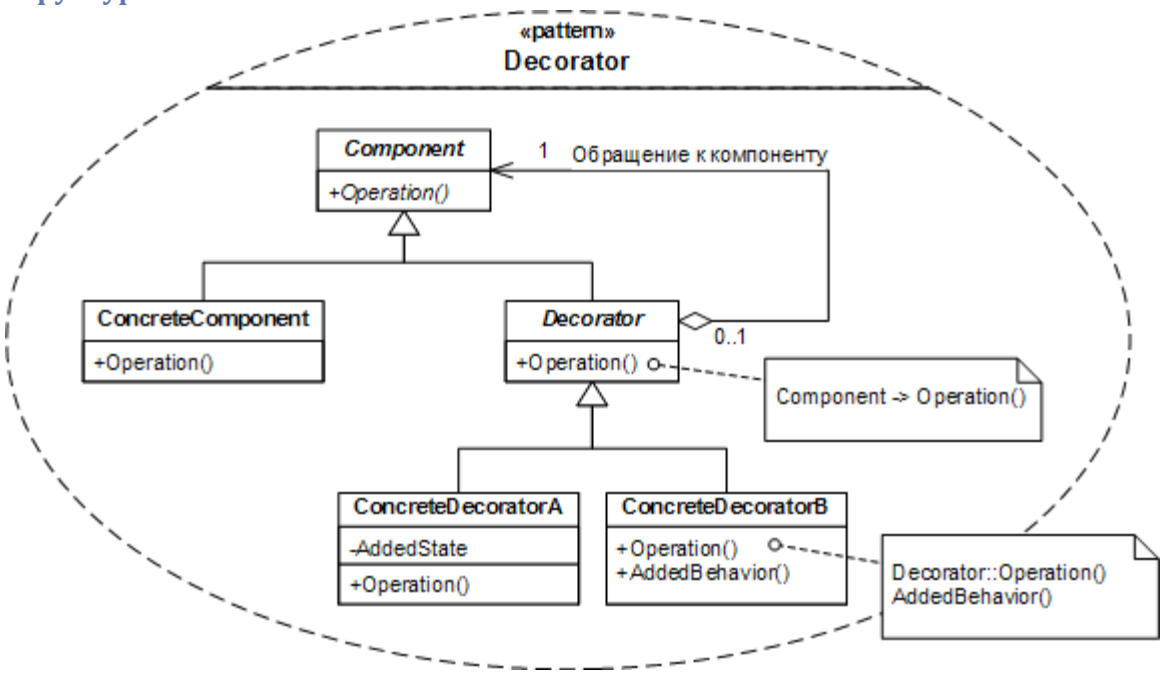
Для расширения функциональности класса возможно использовать как декораторы, так и стратегии. Декораторы оборачивают объект снаружи, стратегии же вставляются в него внутрь по неким интерфейсам. Недостаток стратегии: класс должен быть спроектирован с возможностью вставления стратегий, декоратор же не требует такой поддержки. Недостаток декоратора: он оборачивает ровно тот же интерфейс, что предназначен для внешнего мира, что вызывает смешение публичного интерфейса и интерфейса кастомизации, которое не всегда желательно.

Применение шаблона

Драйверы-фильтры в ядре Windows (архитектура WDM) представляют собой декораторы. Несмотря на то, что WDM реализована на не-объектном языке Си, в ней четко прослеживаются паттерны проектирования — декоратор, цепочка ответственности, и команда (объект IRP).

Архитектура COM не поддерживает наследование реализаций, вместо него предлагается использовать декораторы (в данной архитектуре это называется «агрегация»). При этом архитектура решает (с помощью механизма pUnkOuter) проблему object identity, возникающую при использовании декораторов — identity агрегата есть identity его самого внешнего декоратора.

Структура



Пример реализации

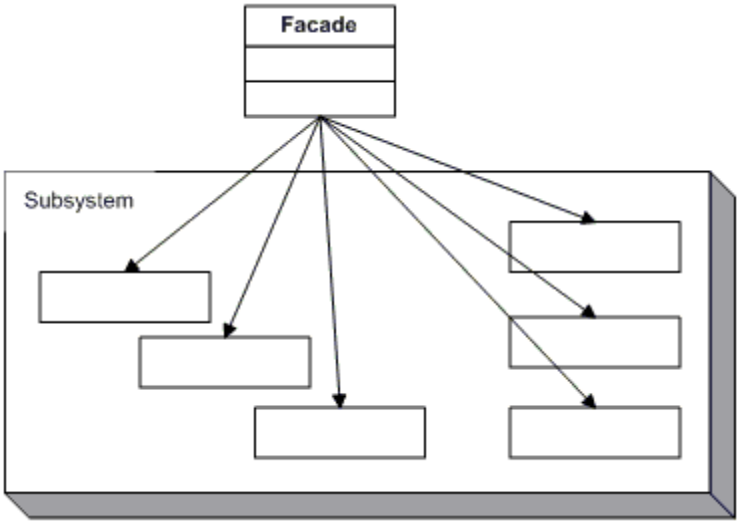
```
using System;

namespace Decorator
{

    class MainApp
    {
        static void Main()
        {
            // Create ConcreteComponent and two Decorators
            ConcreteComponent c = new ConcreteComponent();
            ConcreteDecoratorA d1 = new ConcreteDecoratorA();
            ConcreteDecoratorB d2 = new ConcreteDecoratorB();

            // Link decorators
            d1.SetComponent(c);
            d2.SetComponent(d1);

            d2.Operation();

            // Wait for user
            Console.Read();
        }
    }

    /// <summary>
    /// Component - компонент
    /// </summary>
    /// <remarks>
    /// <li>
    /// <lu>определяем интерфейс для объектов, на которые могут быть динамически
    /// возложены дополнительные обязанности;</lu>
    /// </li>
    /// </remarks>
    abstract class Component
    {
        public abstract void Operation();
    }

    /// <summary>
    /// ConcreteComponent - конкретный компонент
    /// </summary>
    /// <remarks>
    /// <li>
    /// <lu>определяет объект, на который возлагается дополнительные обязанности</lu>
    /// </li>
    /// </remarks>
    class ConcreteComponent : Component
    {
        public override void Operation()
        {
            Console.WriteLine("ConcreteComponent.Operation()");
        }
    }


    /// <summary>
    /// Decorator - декоратор
    /// </summary>
    /// <remarks>
    /// <li>
    /// <lu>хранит ссылку на объект <see cref="Component"/> и определяет интерфейс,
    /// соответствующий интерфейсу <see cref="Component"/></lu>
    /// </li>
```

```
    /// </remarks>
    abstract class Decorator : Component
    {
        protected Component component;

        public void SetComponent(Component component)
        {
            this.component = component;
        }

        public override void Operation()
        {
            if (component != null)
            {
                component.Operation();
            }
        }
    }

    /// <summary>
    /// ConcreteDecorator - конкретный декоратор
    /// </summary>
    /// <remarks>
    /// <li>
    /// <lu>возглагает дополнительные обязанности на компонент.</lu>
    /// </li>
    /// </remarks>
    class ConcreteDecoratorA : Decorator
    {
        private string addedState;

        public override void Operation()
        {
            base.Operation();
            addedState = "New State";
            Console.WriteLine("ConcreteDecoratorA.Operation()");
        }
    }

    // "ConcreteDecoratorB"

    class ConcreteDecoratorB : Decorator
    {
        public override void Operation()
        {
            base.Operation();
            AddedBehavior();
            Console.WriteLine("ConcreteDecoratorB.Operation()");
        }

        void AddedBehavior()
        {
        }
    }
}
```

Facade — Фасад

Шаблон Facade (Фасад) — Шаблон проектирования, позволяющий скрыть сложность системы путем сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы.

Структура



Проблема

Как обеспечить унифицированный интерфейс с набором разрозненных реализаций или интерфейсов, например, с подсистемой, если нежелательно высокое связывание с этой подсистемой или реализация подсистемы может измениться?

Решение

Определить одну точку взаимодействия с подсистемой — фасадный объект, обеспечивающий общий интерфейс с подсистемой и возложить на него обязанность по взаимодействию с её компонентами. Фасад — это внешний объект, обеспечивающий единственную точку входа для служб подсистемы. Реализация других компонентов подсистемы закрыта и не видна внешним компонентам. Фасадный объект обеспечивает реализацию паттерна Устойчивый к изменениям (Protected Variations) с точки зрения защиты от изменений в реализации подсистемы.

Особенности применения

Шаблон применяется для установки некоторого рода политики по отношению к другой группе объектов. Если политика должна быть яркой и заметной, следует воспользоваться услугами шаблона Фасад. Если же необходимо обеспечить скрытность и аккуратность (прозрачность), более подходящим выбором является шаблон Заместитель (Proxy).

Пример реализации

```
using System;

namespace Library
{
    /// <summary>
    /// Класс подсистемы
    /// </summary>
    /// <remarks>
    /// <li>
    /// <lu>реализует функциональность подсистемы;</lu>
    /// <lu>выполняет работу, порученную объектом <see cref="Facade"/>;</lu>
    /// <lu>ничего не "знает" о существовании фасада, то есть не хранит ссылок на него;</lu>
    /// </li>
    /// </remarks>
    internal class SubsystemA
    {
        internal string A1()
        {
            return "Subsystem A, Method A1\n";
        }
        internal string A2()
        {
            return "Subsystem A, Method A2\n";
        }
    }
    internal class SubsystemB
    {
        internal string B1()
        {
            return "Subsystem B, Method B1\n";
        }
    }
    internal class SubsystemC
    {
        internal string C1()
        {
            return "Subsystem C, Method C1\n";
        }
    }
}

/// <summary>
/// Facade - фасад
/// </summary>
/// <remarks>
/// <li>
/// <lu>"знает", каким классами подсистемы адресовать запрос;</lu>
/// <lu>делегирует запросы клиентов подходящим объектам внутри подсистемы;</lu>
/// </li>
/// </remarks>
public static class Facade
{
    static Library.SubsystemA a = new Library.SubsystemA();
    static Library.SubsystemB b = new Library.SubsystemB();
    static Library.SubsystemC c = new Library.SubsystemC();

    public static void Operation1()
    {
        Console.WriteLine("Operation 1\n" +
        a.A1() +
        a.A2() +
        b.B1());
    }
    public static void Operation2()
```

```
    {
        Console.WriteLine("Operation 2\n" +
        b.B1() +
        c.C1());
    }
}

class Program
{
    static void Main(string[] args)
    {
        Facade.Operation1();
        Facade.Operation2();

        // Wait for user
        Console.Read();
    }
}
```

Flyweight — Приспособленец

Приспособленец (англ. Flyweight) — это объект, представляющий себя как уникальный экземпляр в разных местах программы, но по факту не являющийся таковым.

Цель

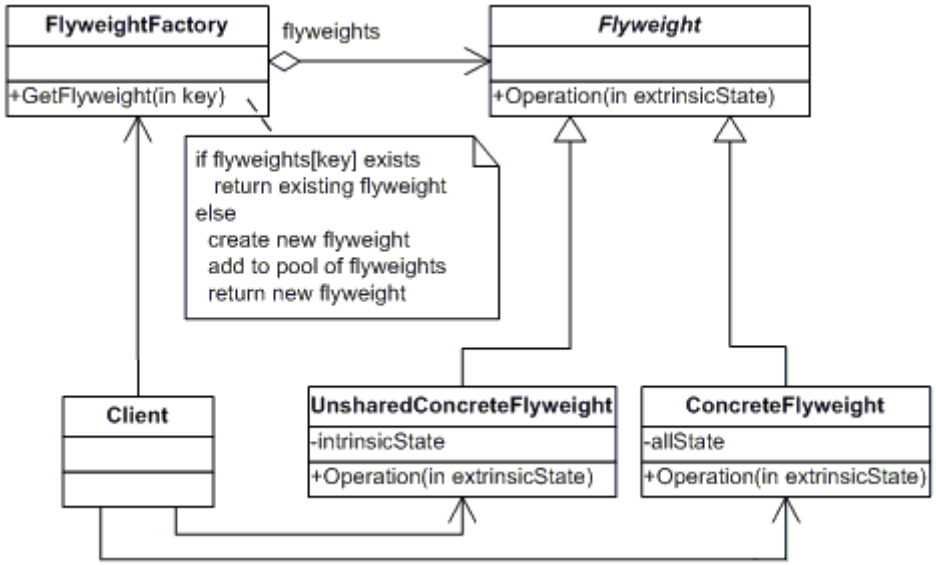
Оптимизация работы с памятью, путем предотвращения создания экземпляров элементов, имеющих общую сущность.

Описание

Flyweight используется для уменьшения затрат при работе с большим количеством мелких объектов. При проектировании приспособленца необходимо разделить его свойства на внешние и внутренние. Внутренние свойства всегда неизменны, тогда как внешние могут отличаться в зависимости от места и контекста применения и должны быть вынесены за пределы приспособленца.

Flyweight дополняет паттерн Factory таким образом, что Factory при обращении к ней клиента для создания нового объекта ищет уже созданный объект с такими же параметрами, что и у требуемого, и возвращает его клиенту. Если такого объекта нет, то фабрика создаст новый.

Суктура



Пример реализации

```
using System;
using System.Collections;

namespace Flyweight
{

    class MainApp
    {
        static void Main()
        {
            // Build a document with text
            string document = "AAZZBBZB";
            char[] chars = document.ToCharArray();

            CharacterFactory f = new CharacterFactory();

            // extrinsic state
            int pointSize = 10;
```

```
        // For each character use a flyweight object
        foreach (char c in chars)
        {
            pointSize++;
            Character character = f.GetCharacter(c);
            character.Display(pointSize);
        }

        // Wait for user
        Console.Read();
    }
}

// "FlyweightFactory"

class CharacterFactory
{
    private Hashtable characters = new Hashtable();

    public Character GetCharacter(char key)
    {
        // Uses "lazy initialization"
        Character character = characters[key] as Character;
        if (character == null)
        {
            switch (key)
            {
                case 'A':
                    character = new CharacterA();
                    break;
                case 'B':
                    character = new CharacterB();
                    break;
                //...
                case 'Z':
                    character = new CharacterZ();
                    break;
            }
            characters.Add(key, character);
        }
        return character;
    }
}

// "Flyweight"

abstract class Character
{
    protected char symbol;
    protected int width;
    protected int height;
    protected int ascent;
    protected int descent;
    protected int pointSize;

    public abstract void Display(int pointSize);
}

// "ConcreteFlyweight"

class CharacterA : Character
{
    // Constructor
    public CharacterA()
    {
```

```
            this.symbol = 'A';
            this.height = 100;
            this.width = 120;
            this.ascent = 70;
            this.descent = 0;
        }

        public override void Display(int pointSize)
        {
            this.pointSize = pointSize;
            Console.WriteLine(this.symbol +
              " (pointsize " + this.pointSize + ")");
        }
    }

    // "ConcreteFlyweight"

    class CharacterB : Character
    {
        // Constructor
        public CharacterB()
        {
            this.symbol = 'B';
            this.height = 100;
            this.width = 140;
            this.ascent = 72;
            this.descent = 0;
        }

        public override void Display(int pointSize)
        {
            this.pointSize = pointSize;
            Console.WriteLine(this.symbol +
              " (pointsize " + this.pointSize + ")");
        }

    }

    // ... C, D, E, etc.

    // "ConcreteFlyweight"

    class CharacterZ : Character
    {
        // Constructor
        public CharacterZ()
        {
            this.symbol = 'Z';
            this.height = 100;
            this.width = 100;
            this.ascent = 68;
            this.descent = 0;
        }

        public override void Display(int pointSize)
        {
            this.pointSize = pointSize;
            Console.WriteLine(this.symbol +
              " (pointsize " + this.pointSize + ")");
        }
    }
}
```

Proxy — Заместитель

Шаблон Proxy (определяет объект-заместитель англ. surrogate иначе -заменитель англ. placeholder) — шаблон проектирования, который предоставляет объект, который контролирует доступ к другому объекту, перехватывая все вызовы (выполняет функцию контейнера).

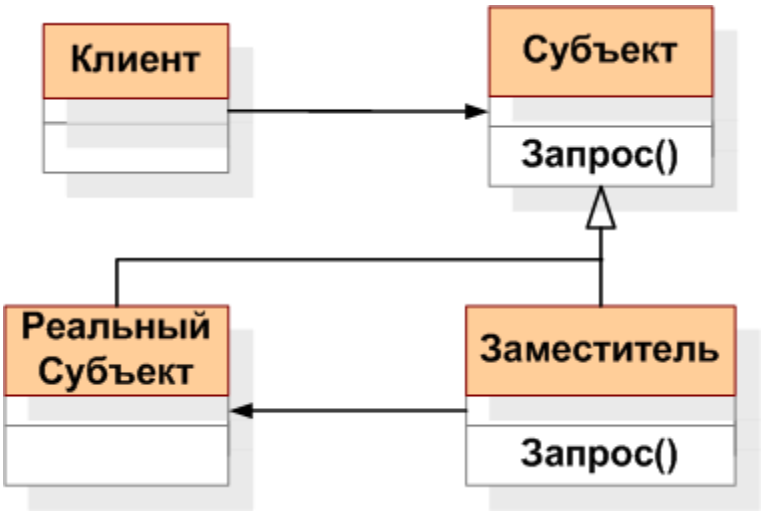
Проблема

Необходимо управлять доступом к объекту так, чтобы создавать громоздкие объекты «по требованию».

Решение

Создать суррогат громоздкого объекта. «Заместитель» хранит ссылку, которая позволяет заместителю обратиться к реальному субъекту (объект класса «Заместитель» может обращаться к объекту класса «Субъект», если интерфейсы «Реального Субъекта» и «Субъекта» одинаковы). Поскольку интерфейс «Реального Субъекта» идентичен интерфейсу «Субъекта», так, что «Заместителя» можно подставить вместо «Реального Субъекта», контролирует доступ к «Реальному Субъекту», может отвечать за создание или удаление «Реального Субъекта». «Субъект» определяет общий для «Реального Субъекта» и «Заместителя» интерфейс, так, что «Заместитель» может быть использован везде, где ожидается «Реальный Субъект». При необходимости запросы могут быть переадресованы «Заместителем» «Реальному Субъекту».

Структура



Шаблон proxy бывает нескольких видов, а именно:

Удаленный заместитель (англ. remote proxies): обеспечивает связь с «Субъектом», который находится в другом адресном пространстве или на удалённой машине. Так же может отвечать за кодирование запроса и его аргументов и отправку закодированного запроса реальному «Субъекту»,

Виртуальный заместитель (англ. virtual proxies): обеспечивает создание реального «Субъекта» только тогда, когда он действительно понадобится. Так же может кэшировать часть информации о реальном «Субъекте», чтобы отложить его создание,

Копировать-при-записи: обеспечивает копирование «субъекта» при выполнении клиентом определённых действий (частный случай «виртуального прокси»).

Защищающий заместитель (англ. protection proxies): может проверять, имеет ли вызывающий объект необходимые для выполнения запроса права.

Кэширующий прокси: обеспечивает временное хранение результатов расчёта до отдачи их множественным клиентам, которые могут разделить эти результаты.

Экранирующий прокси: защищает «Субъект» от опасных клиентов (или наоборот).

Синхронизирующий прокси: производит синхронизированный контроль доступа к «Субъекту» в асинхронной многопоточной среде.

Smart reference proxy: производит дополнительные действия, когда на «Субъект» создается ссылка, например, рассчитывает количество активных ссылок на «Субъект».

Преимущества

- удаленный заместитель;

- виртуальный заместитель может выполнять оптимизацию;

- защищающий заместитель;

- "умная" ссылка;

Недостатки

- резкое увеличение времени отклика.

Сфера применения

Шаблон Proxy может применяться в случаях работы с сетевым соединением, с огромным объектом в памяти (или на диске) или с любым другим ресурсом, который сложно или тяжело копировать. Хорошо известный пример применения — объект, подсчитывающий число ссылок.

Прокси и близкие к нему шаблоны

Адаптер обеспечивает отличающийся интерфейс к объекту.

Прокси обеспечивает тот же самый интерфейс.

Декоратор обеспечивает расширенный интерфейс.

Пример реализации

```
using System;
using System.Threading;

class MainApp
{
    static void Main()
    {
        // Create math proxy
        IMath p = new MathProxy();

        // Do the math
        Console.WriteLine("4 + 2 = " + p.Add(4, 2));
        Console.WriteLine("4 - 2 = " + p.Sub(4, 2));
        Console.WriteLine("4 * 2 = " + p.Mul(4, 2));
        Console.WriteLine("4 / 2 = " + p.Div(4, 2));

        // Wait for user
        Console.Read();
    }
}
```

```
/// <summary>
/// Subject - субъект
/// </summary>
/// <remarks>
/// <li>
/// <lu>определяет общий для <see cref="Math"/> и <see cref="Proxy"/> интерфейс, так что класс
/// <see cref="Proxy"/> можно использовать везде, где ожидается <see cref="Math"/></lu>
/// </li>
/// </remarks>
public interface IMath
{
    double Add(double x, double y);
    double Sub(double x, double y);
    double Mul(double x, double y);
    double Div(double x, double y);
}


/// <summary>
/// RealSubject - реальный объект
/// </summary>
/// <remarks>
/// <li>
/// <lu>определяет реальный объект, представленный заместителем</lu>
/// </li>
/// </remarks>
class Math : IMath
{
    public Math()
    {
        Console.WriteLine("Create object Math. Wait...");
        Thread.Sleep(1000);
    }

    public double Add(double x, double y)
    {
        return x + y;
    }
    public double Sub(double x, double y)
    {
        return x - y;
    }
    public double Mul(double x, double y)
    {
        return x * y;
    }
    public double Div(double x, double y)
    {
        return x / y;
    }
}

/// <summary>
/// Proxy - заместитель
/// </summary>
/// <remarks>
/// <li>
/// <lu>хранит ссылку, которая позволяет заместителю обратиться к реальному
/// субъекту. Объект класса <see cref="MathProxy"/> может обращаться к объекту класса
/// <see cref="IMath"/>, если интерфейсы классов <see cref="Math"/> и <see cref="IMath"/>
одинаковы;</lu>
/// <lu>предоставляет интерфейс, идентичный интерфейсу <see cref="IMath"/>, так что заместитель
/// всегда может быть предоставлен вместо реального субъекта;</lu>
/// <lu>контролирует доступ к реальному субъекту и может отвечать за его создание
/// и удаление;</lu>
```

```
/// <lu>прочие обязанности зависят от вида заместителя:
/// <li>
/// <lu><b>удаленный заместитель</b> отвечает за кодирование запроса и его аргументов
/// и отправление закодированного запроса реальному субъекту в
/// другом адресном пространстве;</lu>
/// <lu><b>виртуальный заместитель</b> может кэшировать дополнительную информацию
/// о реальном субъекте, чтобы отложить его создание.</lu>
/// <lu><b>защищающий заместитель</b> проверяет, имеет ли вызывающий объект
/// необходимые для выполнения запроса права;</lu>
/// </li>
/// </lu>
/// </li>
/// </remarks>
class MathProxy : IMath
{
    Math math;

    public MathProxy()
    {
        math = null;
    }

    /// <summary>
    /// Быстрая операция - не требует реального субъекта
    /// </summary>
    /// <param name="x"></param>
    /// <param name="y"></param>
    /// <returns></returns>
    public double Add(double x, double y)
    {
        return x + y;
    }

    public double Sub(double x, double y)
    {
        return x - y;
    }

    /// <summary>
    /// Медленная операция - требует создания реального субъекта
    /// </summary>
    /// <param name="x"></param>
    /// <param name="y"></param>
    /// <returns></returns>
    public double Mul(double x, double y)
    {
        if (math == null)
            math = new Math();
        return math.Mul(x, y);
    }

    public double Div(double x, double y)
    {
        if (math == null)
            math = new Math();
        return math.Div(x, y);
    }
}
```

Поведенческие шаблоны проектирования

Поведенческие шаблоны (англ. behavioral patterns) — шаблоны проектирования, определяющие алгоритмы и способы реализации взаимодействия различных объектов и классов.

Использование

В поведенческих шаблонах уровня класса используется наследование, чтобы определить поведение для различных классов. В поведенческих шаблонах уровня объекта используется композиция. Некоторые из них описывают, как с помощью кооперации несколько равноправных объектов работают над заданием, которое они не могут выполнить по отдельности. Здесь важно то, как объекты получают информацию о существовании друг друга. Объекты-коллеги могут хранить ссылки друг на друга, но это усиливает степень связанности системы. При высокой связанности каждому объекту пришлось бы иметь информацию обо всех остальных. Некоторые из шаблонов решают эту проблему.

Перечень поведенческий шаблонов

- **цепочка ответственности (chain of responsibility);**
- **команда (action, transaction);**
- **интерпретатор (interpreter);**
- **итератор (cursor);**
- **посредник (mediator);**
- **хранитель (token);**
- **null object (null object);**
- **наблюдатель (dependents, publish-subscribe, listener);**
- **слуга (servant);**
- **specification (specification);**
- **состояние (objects for states);**
- **стратегия (strategy);**
- **шаблонный метод (template method);**
- **посетитель (visitor);**
- **simple Policy;**
- **single-serving visitor;**

Chain of responsibility — Цепочка обязанностей

Цепочка обязанностей — поведенческий шаблон проектирования, предназначенный для организации в системе уровней ответственности.

Применение

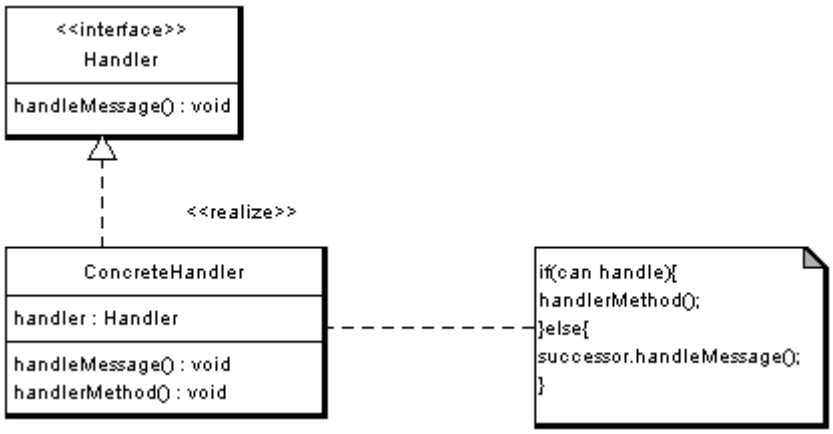
Шаблон рекомендован для использования в условиях:

в разрабатываемой системе имеется группа объектов, которые могут обрабатывать сообщения определенного типа;

все сообщения должны быть обработаны хотя бы одним объектом системы;

сообщения в системе обрабатываются по схеме «обработай сам либо перешли другому», то есть одни сообщения обрабатываются на том уровне, где они получены, а другие пересылаются объектам иного уровня.

Сруктура



Пример реализации

```
// Chain of Responsibility pattern -- Structural example

using System;

namespace DoFactory.GangOfFour.Chain.Structural
{
  /// <summary>
  /// MainApp startup class for Structural
  /// Chain of Responsibility Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      // Setup Chain of Responsibility
      Handler h1 = new ConcreteHandler1();
      Handler h2 = new ConcreteHandler2();
      Handler h3 = new ConcreteHandler3();
      h1.SetSuccessor(h2);
      h2.SetSuccessor(h3);

      // Generate and process request
      int[] requests = { 2, 5, 14, 22, 18, 3, 27, 20 };
```

```
    foreach (int request in requests)
    {
      h1.HandleRequest(request);
    }

    // Wait for user
    Console.ReadKey();
  }
}

/// <summary>
/// The 'Handler' abstract class
/// </summary>
abstract class Handler
{
  protected Handler successor;

  public void SetSuccessor(Handler successor)
  {
    this.successor = successor;
  }

  public abstract void HandleRequest(int request);
}

/// <summary>
/// The 'ConcreteHandler1' class
/// </summary>
class ConcreteHandler1 : Handler
{
  public override void HandleRequest(int request)
  {
    if (request >= 0 && request < 10)
    {
      Console.WriteLine("{0} handled request {1}",
        this.GetType().Name, request);
    }
    else if (successor != null)
    {
      successor.HandleRequest(request);
    }
  }
}

/// <summary>
/// The 'ConcreteHandler2' class
/// </summary>
class ConcreteHandler2 : Handler
{
  public override void HandleRequest(int request)
  {
    if (request >= 10 && request < 20)
    {
      Console.WriteLine("{0} handled request {1}",
        this.GetType().Name, request);
    }
    else if (successor != null)
    {
      successor.HandleRequest(request);
    }
  }
}

/// <summary>
/// The 'ConcreteHandler3' class
```

```
  /// </summary>
  class ConcreteHandler3 : Handler
  {
    public override void HandleRequest(int request)
    {
      if (request >= 20 && request < 30)
      {
        Console.WriteLine("{0} handled request {1}",
          this.GetType().Name, request);
      }
      else if (successor != null)
      {
        successor.HandleRequest(request);
      }
    }
  }
}
```

```
Output
ConcreteHandler1 handled request 2
ConcreteHandler1 handled request 5
ConcreteHandler2 handled request 14
ConcreteHandler3 handled request 22
ConcreteHandler2 handled request 18
ConcreteHandler1 handled request 3
ConcreteHandler3 handled request 27
ConcreteHandler3 handled request 20
```

Command — Команда

Команда — шаблон проектирования, используемый при объектно-ориентированном программировании, представляющий действие. Объект команды заключает в себе само действие и его параметры.

Цель

Создание структуры, в которой класс-отправитель и класс-получатель не зависят друг от друга напрямую. Организация обратного вызова к классу, который включает в себя класс-отправитель.

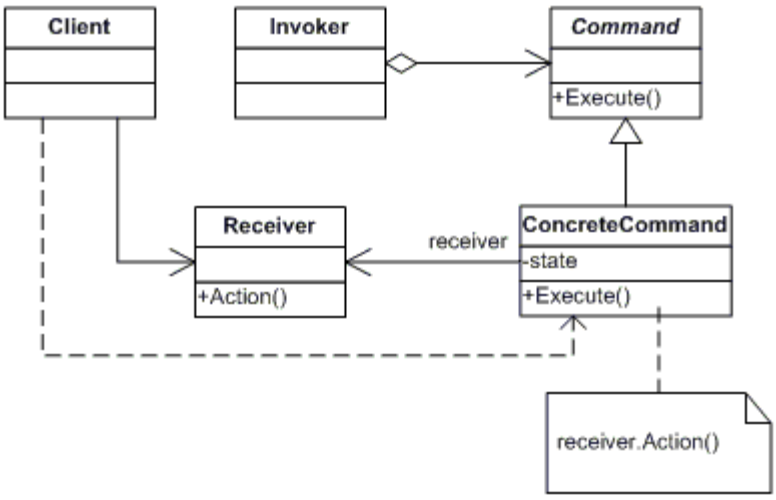
Описание

Паттерн поведения объектов,известен так же под именем Action(действие).

Обеспечивает обработку команды в виде объекта, что позволяет сохранять её, передавать в качестве параметра методам, а также возвращать её в виде результата, как и любой другой объект.

Например, библиотека печати может иметь класс PrintJob. Для его использования можно создать объект PrintJob, установить необходимые параметры, и вызвать метод, непосредственно отсылающий задание на печать.

Суктура



Пример реализации

```
using System;
using System.Collections.Generic;

namespace Command
{

    class MainApp
    {
        static void Main()
        {
            // Создаем пользователя.
            User user = new User();

            // Пусть он что-нибудь сделает.
            user.Compute('+', 100);
            user.Compute('-', 50);
            user.Compute('*', 10);
            user.Compute('/', 2);

            // Отменяем 4 команды
            user.Undo(4);
```

```
        // Вернём 3 отменённые команды.
        user.Redo(3);

        // Ждем ввода пользователя и завершаемся.
        Console.Read();
    }
}

// "Command" : абстрактная Команда

abstract class Command
{
    public abstract void Execute();
    public abstract void UnExecute();
}

// "ConcreteCommand" : конкретная команда

class CalculatorCommand : Command
{
    char @operator;
    int operand;
    Calculator calculator;

    // Constructor
    public CalculatorCommand(Calculator calculator,
      char @operator, int operand)
    {
        this.calculator = calculator;
        this.@operator = @operator;
        this.operand = operand;
    }

    public char Operator
    {
        set
        {
            @operator = value;
        }
    }

    public int Operand
    {
        set
        {
            operand = value;
        }
    }

    public override void Execute()
    {
        calculator.Operation(@operator, operand);
    }

    public override void UnExecute()
    {
        calculator.Operation(Undo(@operator), operand);
    }

    // Private helper function : приватные вспомогательные функции
    private char Undo(char @operator)
    {
        char undo;
        switch (@operator)
        {
```

```
            case '+':
                undo = '-';
                break;
            case '-':
                undo = '+';
                break;
            case '*':
                undo = '/';
                break;
            case '/':
                undo = '*';
                break;
            default:
                undo = ' ';
                break;
        }
        return undo;
    }
}

// "Receiver" : получатель

class Calculator
{
    private int curr = 0;

    public void Operation(char @operator, int operand)
    {
        switch (@operator)
        {
            case '+':
                curr += operand;
                break;
            case '-':
                curr -= operand;
                break;
            case '*':
                curr *= operand;
                break;
            case '/':
                curr /= operand;
                break;
        }
        Console.WriteLine(
          "Current value = {0,3} (following {1} {2})",
          curr, @operator, operand);
    }
}

// "Invoker" : вызывающий

class User
{
    // Initializers
    private Calculator _calculator = new Calculator();
    private List<Command> _commands = new List<Command>();

    private int _current = 0;

    public void Redo(int levels)
    {
        Console.WriteLine("\n---- Redo {0} levels ", levels);

        // Делаем возврат операций
        for (int i = 0; i < levels; i++)
```

```
            if (_current < _commands.Count - 1)
                _commands[_current++].Execute();
        }

        public void Undo(int levels)
        {
            Console.WriteLine("\n---- Undo {0} levels ", levels);

            // Делаем отмену операций
            for (int i = 0; i < levels; i++)
                if (_current > 0)
                    _commands[--_current].UnExecute();
        }

        public void Compute(char @operator, int operand)
        {

            // Создаем команду операции и выполняем её
            Command command = new CalculatorCommand(
              _calculator, @operator, operand);
            command.Execute();

            // Добавляем операцию к списку отмены
            _commands.Add(command);
            _current++;
        }
    }
}
```

Interpreter — Интерпретатор

Шаблон Интерпретатор (англ. Interpreter) — поведенческий шаблон проектирования, решающий часто встречающуюся, но подверженную изменениям, задачу. Также известен как Little (Small) Language

Проблема

Имеется часто встречающаяся, подверженная изменениям задача.

Решение

Создать интерпретатор, который решает данную задачу.

Преимущества

Грамматику становится легко расширять и изменять, реализации классов, описывающих узлы абстрактного синтаксического дерева похожи (легко кодируются). Можно легко изменять способ вычисления выражений.

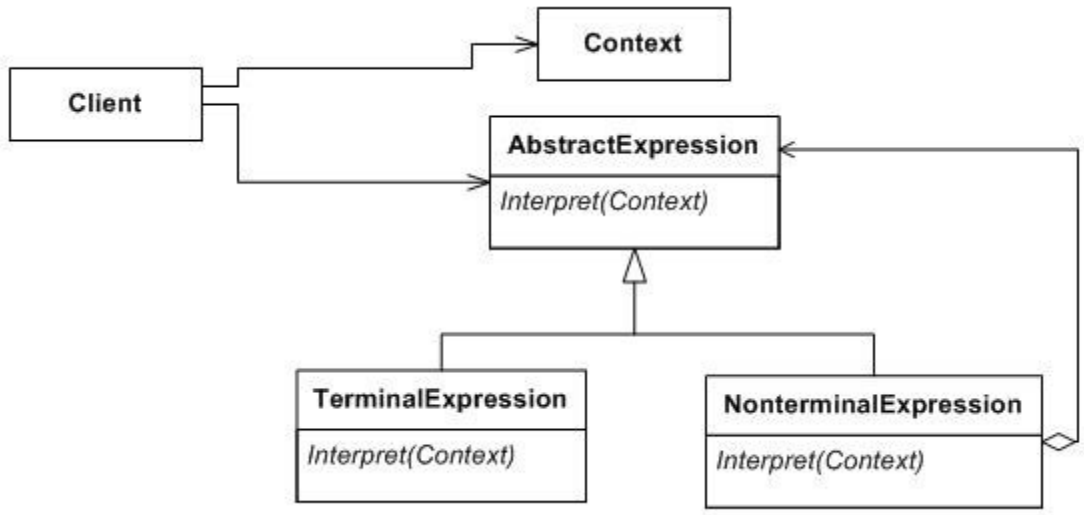
Недостатки

Сопровождение грамматики с большим числом правил затруднительно.

Пример

Задача поиска строк по образцу может быть решена посредством создания интерпретатора, определяющего грамматику языка. "Клиент" строит предложение в виде абстрактного синтаксического дерева, в узлах которого находятся объекты классов "НетерминальноеВыражение" и "ТерминальноеВыражение" (рекурсивное), затем "Клиент" инициализирует контекст и вызывает операцию Разобрать(Контекст). На каждом узле типа "НетерминальноеВыражение" определяется операция Разобрать для каждого подвыражения. Для класса "ТерминальноеВыражение" операция Разобрать определяет базу рекурсии. "АбстрактноеВыражение" определяет абстрактную операцию Разобрать, общую для всех узлов в абстрактном синтаксическом дереве. "Контекст" содержит информацию, глобальную по отношению к интерпретатору.

Структура



Пример реализации

```
using System;
using System.Collections;

namespace DoFactory.GangOfFour.Interpreter.Structural
{
    /// <summary>
```

```
/// MainApp startup class for Structural
/// Interpreter Design Pattern.
/// </summary>
class MainApp
{
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
        Context context = new Context();

        // Usually a tree
        ArrayList list = new ArrayList();

        // Populate 'abstract syntax tree'
        list.Add(new TerminalExpression());
        list.Add(new NonterminalExpression());
        list.Add(new TerminalExpression());
        list.Add(new TerminalExpression());

        // Interpret
        foreach (AbstractExpression exp in list)
        {
            exp.Interpret(context);
        }

        // Wait for user
        Console.ReadKey();
    }
}

/// <summary>
/// The 'Context' class
/// </summary>
class Context
{
}

/// <summary>
/// The 'AbstractExpression' abstract class
/// </summary>
abstract class AbstractExpression
{
    public abstract void Interpret(Context context);
}

/// <summary>
/// The 'TerminalExpression' class
/// </summary>
class TerminalExpression : AbstractExpression
{
    public override void Interpret(Context context)
    {
        Console.WriteLine("Called Terminal.Interpret()");
    }
}

/// <summary>
/// The 'NonterminalExpression' class
/// </summary>
class NonterminalExpression : AbstractExpression
{
    public override void Interpret(Context context)
    {
```

```
            Console.WriteLine("Called Nonterminal.Interpret()");
        }
    }
}

/* Output
Called Terminal.Interpret()
Called Nonterminal.Interpret()
Called Terminal.Interpret()
Called Terminal.Interpret()
*/
```

Iterator — Итератор

Шаблон Iterator (также известный как Cursor) — Шаблон проектирования, относится к паттернам поведения. Представляет собой объект, позволяющий получить последовательный доступ к элементам объекта-агрегата без использования описаний каждого из объектов, входящий в состав агрегации.

Например, такие элементы как дерево, связанный список, хэш-таблица и массив могут быть пролистаны (и модифицированы) с помощью паттерна (объекта) Итератор.

Перебор элементов выполняется объектом итератора, а не самой коллекцией. Это упрощает интерфейс и реализацию коллекции, а также способствует более логичному распределению обязанностей.

Особенностью полноценно реализованного итератора является то, что код, использующий итератор, может ничего не знать о типе итерируемого агрегата.

Конечно же, почти любой агрегат можно итерировать указателем void*, но при этом:

не ясно, что является значением «конец агрегата», для двусвязного списка это &ListHead, для массива это &array[size], для односвязного списка это NULL

операция Next сильно зависит от типа агрегата.

Итераторы абстрагируют именно эти 2 проблемы, используя полиморфный Next (часто реализованный как operator++ в C++) и полиморфный aggregate.end(), возвращающий значение «конец агрегата».

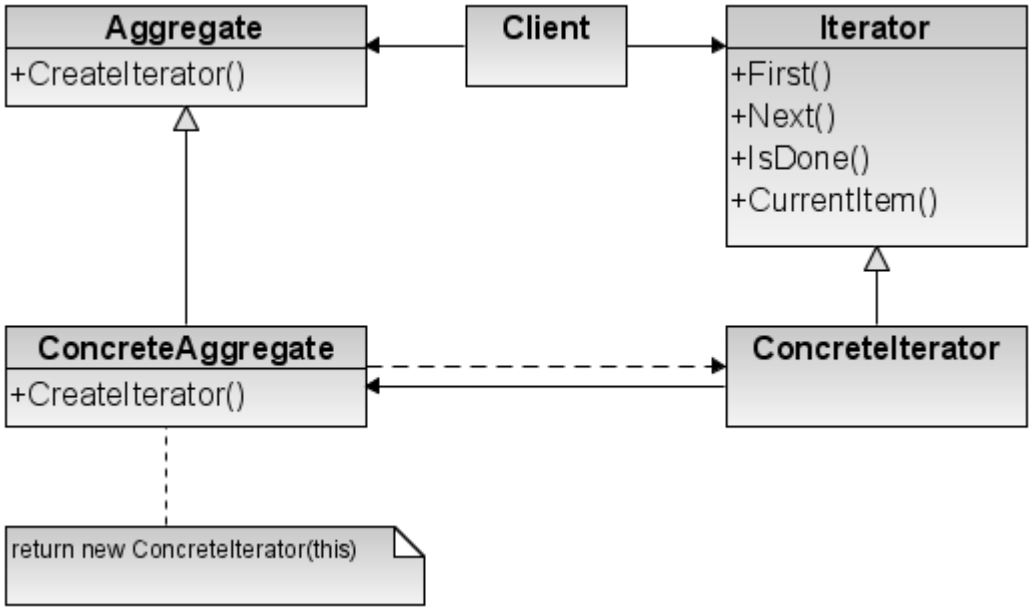
Таким образом, появляется возможность работы с диапазонами итераторов, при отсутствии знания о типе итерируемого агрегата. Например:

```
Iterator itBegin = aggregate.begin();
Iterator itEnd = aggregate.end();
func(itBegin, itEnd);
```

И далее:

```
void func(Iterator itBegin, Iterator itEnd)
{
 for( Iterator it = itBegin, it != itEnd; ++it )
 {
 }
}
```

Структура



Iterator определяет интерфейс для доступа и обхода елементов

ConcreteIterator реализует интерфейс класса Iterator; следит за текущей позицией во время обхода агрегата;

Aggregate определяет интерфейс для создания объекта-итератора;

ConcreteAggregate реализует интерфейс для создания итератора и возвращает экземпляр соответствующего класса *ConcreteIterator*

Пример реализации

```
/*
sample code in C#

This structural code demonstrates the Iterator pattern which provides for a way to traverse
(iterate) over a collection of items without detailing the underlying structure of the
collection.
*/
Hide code

// Iterator pattern -- Structural example

using System;
using System.Collections;

namespace DoFactory.GangOfFour.Iterator.Structural
{
    /// <summary>
    /// MainApp startup class for Structural
    /// Iterator Design Pattern.
    /// </summary>
    class MainApp
    {
        /// <summary>
        /// Entry point into console application.
        /// </summary>
        static void Main()
        {
```

```
        ConcreteAggregate a = new ConcreteAggregate();
        a[0] = "Item A";
        a[1] = "Item B";
        a[2] = "Item C";
        a[3] = "Item D";

        // Create Iterator and provide aggregate
        ConcreteIterator i = new ConcreteIterator(a);

        Console.WriteLine("Iterating over collection:");

        object item = i.First();
        while (item != null)
        {
            Console.WriteLine(item);
            item = i.Next();
        }

        // Wait for user
        Console.ReadKey();
    }
}

/// <summary>
/// The 'Aggregate' abstract class
/// </summary>
abstract class Aggregate
{
    public abstract Iterator CreateIterator();
    public abstract int Count { get; protected set; }
    public abstract object this[int index] { get; set; }
}

/// <summary>
/// The 'ConcreteAggregate' class
/// </summary>
class ConcreteAggregate : Aggregate
{
    private readonly ArrayList _items = new ArrayList();

    public override Iterator CreateIterator()
    {
        return new ConcreteIterator(this);
    }

    // Gets item count
    public override int Count
    {
        get { return _items.Count; }
        protected set { }
    }

    // Indexer
    public override object this[int index]
    {
        get { return _items[index]; }
        set { _items.Insert(index, value); }
    }
}

/// <summary>
/// The 'Iterator' abstract class
/// </summary>
abstract class Iterator
{
```

```
        public abstract object First();
        public abstract object Next();
        public abstract bool IsDone();
        public abstract object CurrentItem();
    }

    /// <summary>
    /// The 'ConcreteIterator' class
    /// </summary>
    class ConcreteIterator : Iterator
    {
        private readonly Aggregate _aggregate;
        private int _current;

        // Constructor
        public ConcreteIterator(Aggregate aggregate)
        {
            this._aggregate = aggregate;
        }

        // Gets first iteration item
        public override object First()
        {
            return _aggregate[0];
        }

        // Gets next iteration item
        public override object Next()
        {
            object ret = null;
            if (_current < _aggregate.Count - 1)
            {
                ret = _aggregate[++_current];
            }

            return ret;
        }

        // Gets current iteration item
        public override object CurrentItem()
        {
            return _aggregate[_current];
        }

        // Gets whether iterations are complete
        public override bool IsDone()
        {
            return _current >= _aggregate.Count;
        }
    }
}

Output
Iterating over collection:
Item A
Item B
Item C
Item D
```

Mediator — Посредник

Шаблон Mediator (также известный как Посредник) – поведенческий шаблон проектирования

Обеспечивает взаимодействие множества объектов, формируя при этом слабую связанность и избавляя объекты от необходимости явно ссылаться друг на друга.

Проблема

Обеспечить взаимодействие множества объектов, сформировав при этом слабую связанность и избавив объекты от необходимости явно ссылаться друг на друга.

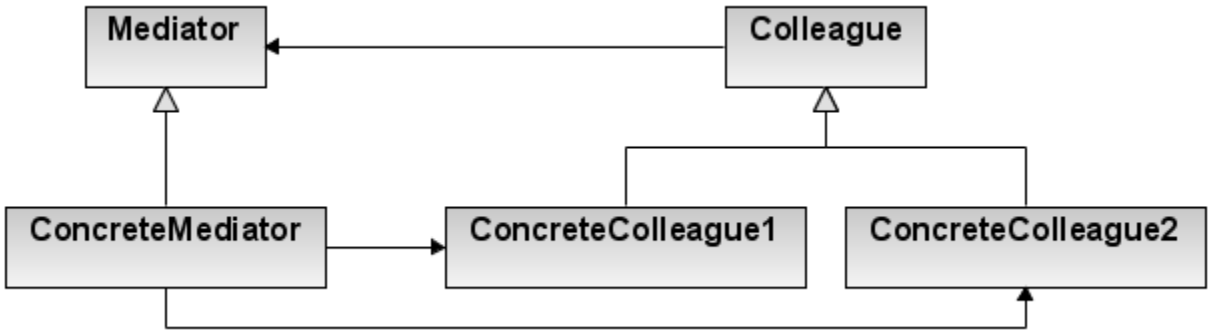
Решение

Создать объект, инкапсулирующий способ взаимодействия множества объектов.

Преимущества

Устраняется связанность между "Коллегами", централизуется управление.

Структура



Mediator – "Посредник"

ConcreteMediator – "Конкретный посредник"

Классы Colleague – "Коллеги"

Описание

"Посредник" определяет интерфейс для обмена информацией с объектами "Коллеги", "Конкретный посредник" координирует действия объектов "Коллеги". Каждый класс "Коллеги" знает о своем объекте "Посредник", все "Коллеги" обмениваются информацией только с посредником, при его отсутствии им пришлось бы обмениваться информацией напрямую. "Коллеги" посылают запросы посреднику и получают запросы от него. "Посредник" реализует кооперативное поведение, пересылая каждый запрос одному или нескольким "Коллегам".

Пример реализации

```
// Mediator pattern – Structural example
using System;

namespace DoFactory.GangOfFour.Mediator.Structural
{
  /// <summary>
  /// MainApp startup class for Structural
  /// Mediator Design Pattern.
  /// </summary>
  class MainApp
  {
```

```
  /// <summary>
  /// Entry point into console application.
  /// </summary>
  static void Main()
  {
    ConcreteMediator m = new ConcreteMediator();

    ConcreteColleague1 c1 = new ConcreteColleague1(m);
    ConcreteColleague2 c2 = new ConcreteColleague2(m);

    m.Colleague1 = c1;
    m.Colleague2 = c2;

    c1.Send("How are you?");
    c2.Send("Fine, thanks");

    // Wait for user
    Console.ReadKey();
  }
}

/// <summary>
/// The 'Mediator' abstract class
/// </summary>
abstract class Mediator
{
  public abstract void Send(string message,
    Colleague colleague);
}

/// <summary>
/// The 'ConcreteMediator' class
/// </summary>
class ConcreteMediator : Mediator
{
  private ConcreteColleague1 _colleague1;
  private ConcreteColleague2 _colleague2;

  public ConcreteColleague1 Colleague1
  {
    set { _colleague1 = value; }
  }

  public ConcreteColleague2 Colleague2
  {
    set { _colleague2 = value; }
  }

  public override void Send(string message,
    Colleague colleague)
  {
    if (colleague == _colleague1)
    {
      _colleague2.Notify(message);
    }
    else
    {
      _colleague1.Notify(message);
    }
  }
}

/// <summary>
/// The 'Colleague' abstract class
/// </summary>
```

```
  abstract class Colleague
  {
    protected Mediator mediator;

    // Constructor
    public Colleague(Mediator mediator)
    {
      this.mediator = mediator;
    }
  }

  /// <summary>
  /// A 'ConcreteColleague' class
  /// </summary>
  class ConcreteColleague1 : Colleague
  {
    // Constructor
    public ConcreteColleague1(Mediator mediator)
      : base(mediator)
    {
    }

    public void Send(string message)
    {
      mediator.Send(message, this);
    }

    public void Notify(string message)
    {
      Console.WriteLine("Colleague1 gets message: "
        + message);
    }
  }

  /// <summary>
  /// A 'ConcreteColleague' class
  /// </summary>
  class ConcreteColleague2 : Colleague
  {
    // Constructor
    public ConcreteColleague2(Mediator mediator)
      : base(mediator)
    {
    }

    public void Send(string message)
    {
      mediator.Send(message, this);
    }

    public void Notify(string message)
    {
      Console.WriteLine("Colleague2 gets message: "
        + message);
    }
  }
}

Output
Colleague2 gets message: How are you?
Colleague1 gets message: Fine, thanks
```

Memento — Хранитель

Хранитель (также известный как Memento, Token, Лексема) — поведенческий шаблон проектирования.

Позволяет, не нарушая инкапсуляцию, зафиксировать и сохранить внутреннее состояния объекта так, чтобы позднее восстановить его в этом состоянии.

Существует два возможных варианта реализации данного шаблона: классический, описанный в книге Design Patterns, и реже встречаемый нестандартный вариант.

Применение

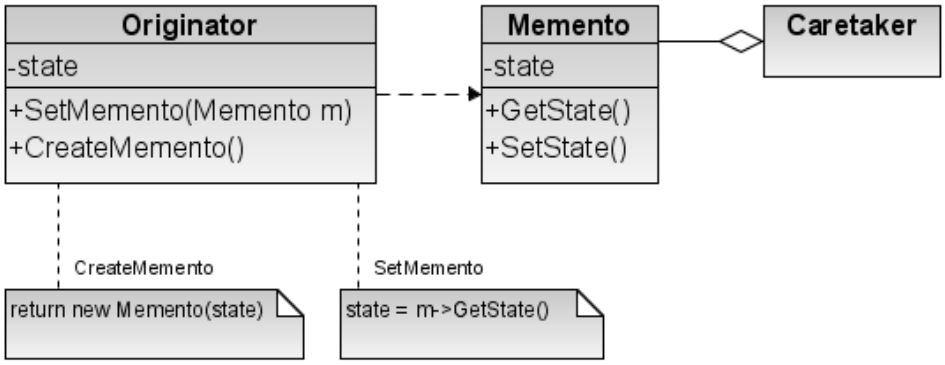
Шаблон Хранитель используется, когда:

- необходимо сохранить снимок состояния объекта (или его части) для последующего восстановления

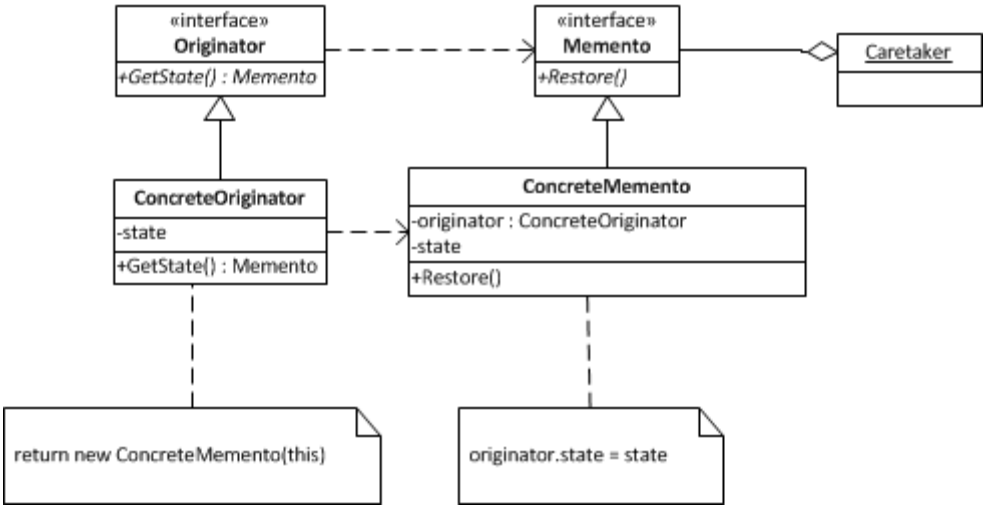
- прямой интерфейс получения состояния объекта раскрывает детали реализации и нарушает инкапсуляцию объекта

Структура

Классический вариант:



Нестандартный вариант:



Originator - "Создатель"

Caretaker - "Опекун"

Memento - "Хранитель"

Описание

Классический вариант: Шаблон Хранитель используется двумя объектами: "Создателем" (originator) и "Опекуном" (caretaker). "Создатель" - это объект, у которого есть внутреннее состояние. Объект "Опекун" может производить некоторые действия с "Создателем", но при этом необходимо иметь возможность откатить изменения. Для этого "Опекун" запрашивает у "Создателя" объект "Хранителя". Затем выполняет запланированное действие (или последовательность действий). Для выполнения отката "Создателя" к состоянию, которое предшествовало изменениям, "Опекун" возвращает объект "Хранителя" его "Создателю". "Хранитель" является непрозрачным (т.е. таким, который не может или не должен изменяться "Опекуном").

Нестандартный вариант: Отличие данного варианта от классического заключено в более жёстком ограничении на доступ "Опекуна" к внутреннему состоянию "Создателя". В классическом варианте у "Опекуна" есть потенциальная возможность получить доступ к внутренним данным "Создателя" через "Хранителя", изменить состояние и установить его обратно "Создателю". В данном варианте "Опекун" обладает возможностью лишь восстановить состояние "Хранителя", вызвав Restore. Кроме всего прочего, "Опекуну" не требуется владеть связью на "Хранителя", чтобы восстановить его состояние. Это позволяет сохранять и восстанавливать состояние сложных иерархических или сетевых структур (состояния объектов и всех связей между ними) путём сбора снимков всех зарегистрированных объектов системы.

Пример реализации

```
using System;
namespace MementoPatte
{
    class Program
    {
        static void Main(string[] args)
        {
            Foo foo = new Foo("Test", 15);
            foo.Print();
            Caretaker ct1 = new Caretaker();
            Caretaker ct2 = new Caretaker();
            ct1.SaveState(foo);
            foo.IntProperty += 152;
            foo.Print();
            ct2.SaveState(foo);
            ct1.RestoreState(foo);
            foo.Print();
            ct2.RestoreState(foo);
            foo.Print();
            Console.ReadKey();
        }
    }

    public interface IOriginator
    {
        object GetMemento();
        void SetMemento(object memento);
    }

    public class Foo
        : IOriginator
    {
        public string StringProperty
        {
            get;
            private set;
        }
```

```
    public int IntProperty
    {
        get;
        set;
    }

    public Foo(string stringPropertyValue, int intPropertyValue = 0)
    {
        StringProperty = stringPropertyValue;
        IntProperty = intPropertyValue;
    }

    public void Print()
    {
        Console.WriteLine("=============");
        Console.WriteLine("StringProperty value: {0}", StringProperty);
        Console.WriteLine("IntProperty value: {0}", IntProperty);
        Console.WriteLine("=============");
    }
    object IOriginator.GetMemento()
    {
        return new Memento
        {
            StringProperty = this.StringProperty,
            IntProperty = this.IntProperty
        };

    }

    void IOriginator.SetMemento(object memento)
    {
        if (Object.ReferenceEquals(memento, null))
            throw new ArgumentNullException("memento");
        if (!(memento is Memento))
            throw new ArgumentException("memento");
        StringProperty = ((Memento)memento).StringProperty;
        IntProperty = ((Memento)memento).IntProperty;
    }

    class Memento
    {
        public string StringProperty
        {
            get;
            set;
        }

        public int IntProperty
        {
            get;
            set;
        }
    }
}

public class Caretaker
{
    private object m_memento;
    public void SaveState(IOriginator originator)
    {
        if (originator == null)
            throw new ArgumentNullException("originator");
        m_memento = originator.GetMemento();
    }
```

```
        public void RestoreState(IOriginator originator)
        {
            if (originator == null)
                throw new ArgumentNullException("originator");
            if (m_memento == null)
                throw new InvalidOperationException("m_memento == null");
            originator.SetMemento(m_memento);
        }
    }
}
```

Нестандартный вариант шаблона:

```
public interface IOriginator
{
    IMemento GetState();
}

public interface IShape : IOriginator
{
    void Draw();
    void Scale(double scale);
    void Move(double dx, double dy);
}

public interface IMemento
{
    void RestoreState();
}

public class CircleOriginator : IShape
{
    private class CircleMemento : IMemento
    {
        private readonly double x;
        private readonly double y;
        private readonly double r;
        private readonly CircleOriginator originator;

        public CircleMemento(CircleOriginator originator)
        {
            this.originator = originator;
            x = originator.x;
            y = originator.y;
            r = originator.r;
        }

        public void Restore()
        {
            originator.x = x;
            originator.y = y;
            originator.r = r;
        }
    }

    double x;
    double y;
    double r;

    public void CircleOriginator(double x, double y, double r)
    {
        this.x = x;
        this.y = y;
```

```
        this.r = r;
    }

    public void Draw()
    {
        Console.WriteLine("Circle with radius {0} at ({1}, {2})", r, x, y);
    }

    public void Scale(double scale)
    {
        r *= scale;
    }

    public void Move(double dx, double dy)
    {
        x += dx;
        y += dy;
    }

    public CircleMemento GetState()
    {
        return new CircleMemento(this);
    }
}

public class RectOriginator : IShape
{
    private class RectMemento : IMemento
    {
        private readonly double x;
        private readonly double y;
        private readonly double w;
        private readonly double h;
        private readonly RectOriginator originator;

        public RectMemento(RectOriginator originator)
        {
            this.originator = originator;
            x = originator.x;
            y = originator.y;
            w = originator.w;
            h = originator.h;
        }

        public void Restore()
        {
            originator.x = x;
            originator.y = y;
            originator.w = w;
            originator.h = h;
        }
    }

    double x;
    double y;
    double w;
    double h;

    public void RectOriginator(double x, double y, double w, double h)
    {
        this.x = x;
        this.y = y;
        this.w = w;
        this.h = h;
    }
```

```
    public void Draw()
    {
        Console.WriteLine("Rectangle {0}x{1} at ({2}, {3})", w, h, x, y);
    }

    public void Scale(double scale)
    {
        w *= scale;
        h *= scale;
    }

    public void Move(double dx, double dy)
    {
        x += dx;
        y += dy;
    }

    public IMemento GetState()
    {
        return new RectMemento(this);
    }
}

public class Caretaker
{
    public void Draw(IEnumerable<IShape> shapes)
    {
        foreach (IShape shape in shapes)
        {
            shape.Draw();
        }
    }

    public void MoveAndScale(IEnumerable<IShape> shapes)
    {
        foreach (IShape shape in shapes)
        {
            shape.Scale(10);
            shape.Move(3, 2);
        }
    }

    public IEnumerable<IMemento> SaveStates(IEnumerable<IShape> shapes)
    {
        List<IMemento> states = new List<IMemento>();
        foreach (IShape shape in shapes)
        {
            states.Add(shape.GetState());
        }
    }

    public void RestoreStates(IEnumerable<IMemento> states)
    {
        foreach (IMemento state in states)
        {
            state.Restore();
        }
    }

    public static void Main()
    {
        IShape[] shapes = { new RectOriginator(10, 20, 3, 5), new CircleOriginator(5, 2, 10) };

        //Выводит:
```

```
        // Rectangle 3x5 at (10, 20)
        // Circle with radius 10 at (5, 2)
        Draw(shapes);

        //Сохраняем состояния фигур
        IEnumerable<IStates> states = SaveStates(shapes);

        //Изменяем положение фигур
        MoveAndScale(shapes);

        //Выводит:
        // Rectangle 30x50 at (13, 22)
        // Circle with radius 100 at (8, 4)
        Draw(shapes);

        //Восстановление старого положения фигур
        RestoreStates(states);

        //Выводит:
        // Rectangle 3x5 at (10, 20)
        // Circle with radius 10 at (5, 2)
        Draw(shapes);
    }
}
```

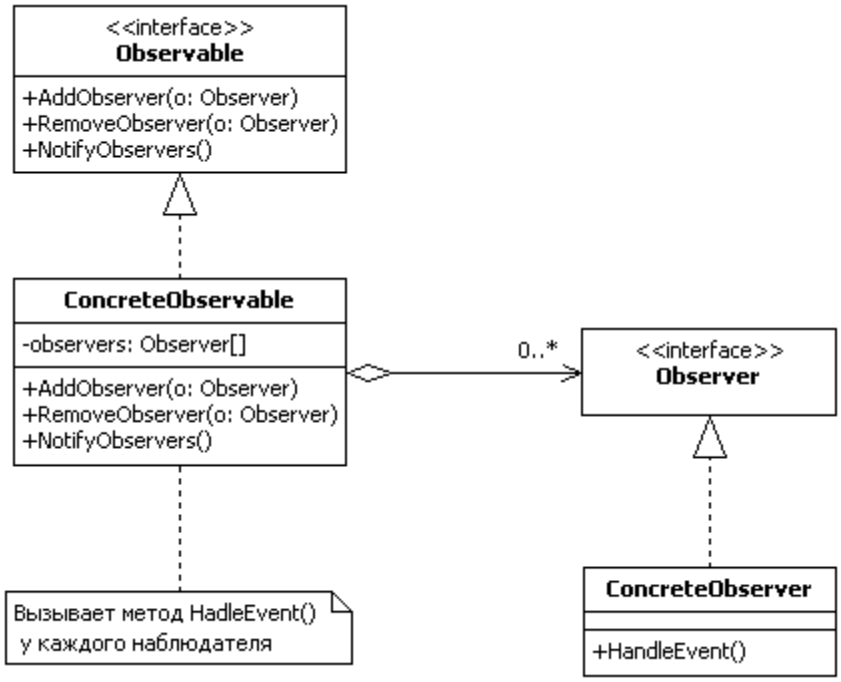
Observer — Наблюдатель

Наблюдатель, Observer — поведенческий шаблон проектирования. Также известен как «подчинённые» (Dependents), «издатель-подписчик» (Publisher-Subscriber).

Назначение

Определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом событии.

Структура



При реализации шаблона «наблюдатель» обычно используются следующие классы.

Observable — интерфейс, определяющий методы для добавления, удаления и оповещения наблюдателей.

Observer — интерфейс, с помощью которого наблюдатель получает оповещение.

ConcreteObservable — конкретный класс, который реализует интерфейс Observable.

ConcreteObserver — конкретный класс, который реализует интерфейс Observer.

Область применения

Шаблон «наблюдатель» применяется в тех случаях, когда система обладает следующими свойствами:

- существует, как минимум, один объект, рассылающий сообщения

- имеется не менее одного получателя сообщений, причём их количество и состав могут изменяться во время работы приложения.

- нет надобности очень сильно связывать взаимодействующие объекты, что полезно для повторного использования.

Данный шаблон часто применяют в ситуациях, в которых отправителя сообщений не интересует, что делают получатели с предоставленной им информацией.

Пример реализации

```
using System;


namespace Observer
{
    /// <summary>
    /// Observer Pattern Judith Bishop Jan 2007
    ///
    /// The Subject runs in a thread and changes its state
    /// independently. At each change, it notifies its Observers.
    /// </summary>
    class Program
    {
        static void Main(string[] args)
        {
            Subject subject = new Subject();
            Observer Observer = new Observer(subject, "Center", "\t\t");
            Observer observer2 = new Observer(subject, "Right", "\t\t\t\t");
            subject.Go();

            // Wait for user
            Console.Read();
        }

    }

    class Simulator : IEnumerable
    {
        string[] moves = { "5", "3", "1", "6", "7" };

        public IEnumerator GetEnumerator()
        {
            foreach (string element in moves)
                yield return element;
        }
    }

    class Subject
    {
        public delegate void Callback(string s);

        public event Callback Notify;

        Simulator simulator = new Simulator();

        const int speed = 200;

        public string SubjectState
        {
            get;
            set;
        }

        public void Go()
        {
            new Thread(new ThreadStart(Run)).Start();
        }

        void Run()
        {
            foreach (string s in simulator)
            {
                Console.WriteLine("Subject: " + s);
                SubjectState = s;
```

```
                Notify(s);
                Thread.Sleep(speed); // milliseconds
            }
        }
    }

    interface IObserver
    {
        void Update(string state);
    }

    class Observer : IObserver
    {
        string name;

        Subject subject;

        string state;

        string gap;

        public Observer(Subject subject, string name, string gap)
        {
            this.subject = subject;
            this.name = name;
            this.gap = gap;
            subject.Notify += Update;
        }

        public void Update(string subjectState)
        {
            state = subjectState;
            Console.WriteLine(gap + name + ": " + state);
        }
    }
}
```

State — Состояние

Состояние (англ. State) — шаблон проектирования. Используется в тех случаях, когда во время выполнения программы объект должен менять свое поведение в зависимости от своего состояния.

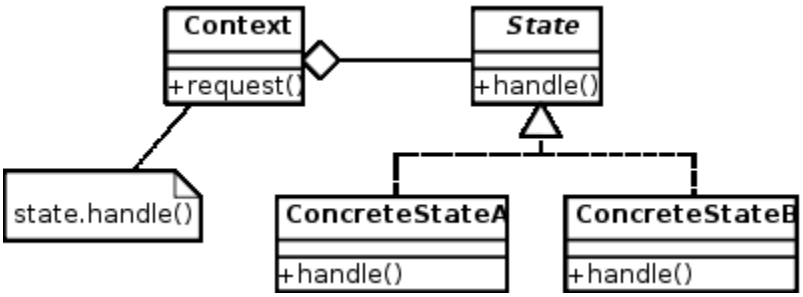
Паттерн состоит из 3 блоков:

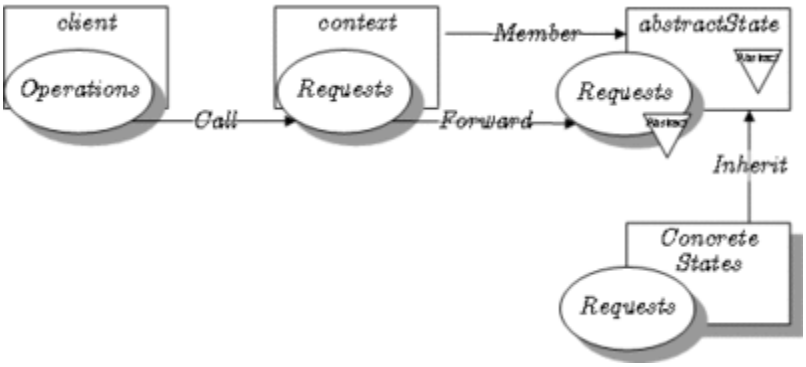
Widget — класс, объекты которого должны менять свое поведение в зависимости от состояния.

IState — интерфейс, который должно реализовать каждое из конкретных состояний. Через этот интерфейс объект Widget взаимодействует с состоянием, делегируя ему вызовы методов. Интерфейс должен содержать средства для обратной связи с объектом, поведение которого нужно изменить. Для этого используется событие (паттерн Publisher — Subscriber). Это необходимо для того, чтобы в процессе выполнения программы заменять объект состояния при появлении событий. Возможны случаи, когда сам Widget периодически опрашивает объект состояние на наличие перехода.

StateA … StateZ — классы конкретных состояний. Должны содержать информацию о том, при каких условиях и в какие состояния может переходить объект из текущего состояния. Например, из StateA объект может переходить в состояние StateB и StateC, а из StateB — обратно в StateA и так далее. Объект одного из них должен содержать Widget при создании.

Структура





Пример реализации

Применение шаблона

```
using System;

namespace Digital_Patterns.Behavioral.State
{
```

```
public interface IAutomatState
{
    String GotApplication();
    String CheckApplication();
    String RentApartment();
    String DispenseKeys();
}
public interface IAutomat
{
    void GotApplication();
    void CheckApplication();
    void RentApartment();

    void SetState(IAutomatState s);
    IAutomatState GetWaitingState();
    IAutomatState GetGotApplicationState();
    IAutomatState GetApartmentRentedState();
    IAutomatState GetFullyRentedState();

    Int32 Count
    {
        get;
        set;
    }
}

public class Automat : IAutomat
{
    private IAutomatState _waitingState;
    private IAutomatState _gotApplicationState;
    private IAutomatState _apartmentRentedState;
    private IAutomatState _fullyRentedState;
    private IAutomatState _state;
    private Int32 _count;

    public Automat(Int32 n)
    {
        _count = n;
        _waitingState = new WaitingState(this);
        _gotApplicationState = new GotApplicationState(this);
        _apartmentRentedState = new ApartmentRentedState(this);
        _fullyRentedState = new FullyRentedState(this);
        _state = _waitingState;
    }

    public void GotApplication()
    {
        Console.WriteLine(_state.GotApplication());
    }

    public void CheckApplication()
    {
        Console.WriteLine(_state.CheckApplication());
    }

    public void RentApartment()
    {
        Console.WriteLine(_state.RentApartment());
        Console.WriteLine(_state.DispenseKeys());
    }

    public void SetState(IAutomatState s)
    {
        _state = s;
    }
```

```
    public IAutomatState GetWaitingState()
    {
        return _waitingState;
    }

    public IAutomatState GetGotApplicationState()
    {
        return _gotApplicationState;
    }

    public IAutomatState GetApartmentRentedState()
    {
        return _apartmentRentedState;
    }

    public IAutomatState GetFullyRentedState()
    {
        return _fullyRentedState;
    }

    public int Count
    {
        get
        {
            return _count;
        }
        set
        {
            _count = value;
        }
    }
}

public class WaitingState : IAutomatState
{
    private Automat _automat;

    public WaitingState(Automat automat)
    {
        _automat = automat;
    }

    public String GotApplication()
    {
        _automat.SetState(_automat.GetGotApplicationState());
        return "Thanks for the application.";
    }

    public String CheckApplication()
    {
        return "You have to submit an application.";
    }

    public String RentApartment()
    {
        return "You have to submit an application.";
    }

    public String DispenseKeys()
    {
        return "You have to submit an application.";
    }
}
```

```
public class GotApplicationState : IAutomatState
{
    private Automat _automat;
    private readonly Random _random;

    public GotApplicationState(Automat automat)
    {
        _automat = automat;
        _random = new Random(System.DateTime.Now.Millisecond);
    }

    public String GotApplication()
    {
        return "We already got your application.";
    }

    public String CheckApplication()
    {
        var yesNo = _random.Next() % 10;

        if (yesNo > 4 && _automat.Count > 0)
        {
            _automat.SetState(_automat.GetApartmentRentedState());
            return "Congratulations, you were approved.";
        }
        else
        {
            _automat.SetState(_automat.GetWaitingState());
            return "Sorry, you were not approved.";
        }
    }

    public String RentApartment()
    {
        return "You must have your application checked.";
    }

    public String DispenseKeys()
    {
        return "You must have your application checked.";
    }
}

public class ApartmentRentedState : IAutomatState
{
    private Automat _automat;

    public ApartmentRentedState(Automat automat)
    {
        _automat = automat;
    }

    public String GotApplication()
    {
        return "Hang on, we're renting you an apartmeny.";
    }

    public String CheckApplication()
    {
        return "Hang on, we're renting you an apartmeny.";
    }

    public String RentApartment()
    {
        _automat.Count = _automat.Count - 1;
```

```
            return "Renting you an apartment....";
        }

        public String DispenseKeys()
        {
            if (_automat.Count <= 0)
                _automat.SetState(_automat.GetFullyRentedState());
            else
                _automat.SetState(_automat.GetWaitingState());
            return "Here are your keys!";
        }
    }

    public class FullyRentedState : IAutomatState
    {
        private Automat _automat;

        public FullyRentedState(Automat automat)
        {
            _automat = automat;
        }

        public String GotApplication()
        {
            return "Sorry, we're fully rented.";
        }

        public String CheckApplication()
        {
            return "Sorry, we're fully rented.";
        }

        public String RentApartment()
        {
            return "Sorry, we're fully rented.";
        }

        public String DispenseKeys()
        {
            return "Sorry, we're fully rented.";
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            var automat = new Automat(9);

            automat.GotApplication();
            automat.CheckApplication();
            automat.RentApartment();
        }
    }
}
```

Тот же пример, без применения шаблона

```
using System;

namespace Digital_Patterns.Behavioral.State
{
    public enum State
```

```
{
    FULLY_RENTED = 0,
    WAITING = 1,
    GOT_APPLICATION = 2,
    APARTMENT_RENTED = 3,
}

public class RentalMethods
{
    private readonly Random _random;
    private Int32 _numberApartments;
    private State _state = State.WAITING;

    public RentalMethods(Int32 n)
    {
        _numberApartments = n;
        _random = new Random(System.DateTime.Now.Millisecond);
    }

    public void GetApplication()
    {
        switch (_state)
        {
            case State.FULLY_RENTED:
                Console.WriteLine("Sorry, we're fully rented.");
                break;
            case State.WAITING:
                _state = State.GOT_APPLICATION;
                Console.WriteLine("Thanks for the application.");
                break;
            case State.GOT_APPLICATION:
                Console.WriteLine("We already got your application.");
                break;
            case State.APARTMENT_RENTED:
                Console.WriteLine("Hang on, we'ra renting you an apartmeny.");
                break;
        }
    }

    public void CheckApplication()
    {
        var yesNo = _random.Next() % 10;

        switch (_state)
        {
            case State.FULLY_RENTED:
                Console.WriteLine("Sorry, we're fully rented.");
                break;
            case State.WAITING:
                Console.WriteLine("You have to submit an application.");
                break;
            case State.GOT_APPLICATION:
                if (yesNo > 4 && _numberApartments > 0)
                {
                    Console.WriteLine("Congratulations, you were approved.");
                    _state = State.APARTMENT_RENTED;
                    RentApartment();
                }
                else
                {
                    Console.WriteLine("Sorry, you were not approved.");
                    _state = State.WAITING;
                }
                break;
            case State.APARTMENT_RENTED:
```

```
                Console.WriteLine("Hang on, we'ra renting you an apartmeny.");
                break;
            }
        }

        public void RentApartment()
        {
            switch (_state)
            {
                case State.FULLY_RENTED:
                    Console.WriteLine("Sorry, we're fully rented.");
                    break;
                case State.WAITING:
                    Console.WriteLine("You have to submit an application.");
                    break;
                case State.GOT_APPLICATION:
                    Console.WriteLine("You must have your application checked.");
                    break;
                case State.APARTMENT_RENTED:
                    Console.WriteLine("Renting you an apartment....");
                    _numberApartments--;
                    DispenseKeys();
                    break;
            }

        }

        public void DispenseKeys()
        {
            switch (_state)
            {
                case State.FULLY_RENTED:
                    Console.WriteLine("Sorry, we're fully rented.");
                    break;
                case State.WAITING:
                    Console.WriteLine("You have to submit an application.");
                    break;
                case State.GOT_APPLICATION:
                    Console.WriteLine("You must have your application checked.");
                    break;
                case State.APARTMENT_RENTED:
                    Console.WriteLine("Here are your keys!");
                    _state = State.WAITING;
                    break;
            }

        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            var rentalMethods = new RentalMethods(9);

            rentalMethods.GetApplication();

            rentalMethods.CheckApplication();
            rentalMethods.RentApartment();
            rentalMethods.DispenseKeys();
        }
    }
}
```

Strategy — Стратегия

Стратегия, Strategy — поведенческий шаблон проектирования, предназначенный для определения семейства алгоритмов, инкапсуляции каждого из них и обеспечения их взаимозаменяемости. Это позволяет выбирать алгоритм путем определения соответствующего класса. Шаблон Strategy позволяет менять выбранный алгоритм независимо от объектов-клиентов, которые его используют.

Задача

По типу клиента (или по типу обрабатываемых данных) выбрать подходящий алгоритм, который следует применить. Если используется правило, которое не подвержено изменениям, нет необходимости обращаться к шаблону «стратегия».

Мотивы

Программа должна обеспечивать различные варианты алгоритма или поведения

Нужно изменять поведение каждого экземпляра класса

Необходимо изменять поведение объектов на стадии выполнения

Введение интерфейса позволяет классам-клиентам ничего не знать о классах, реализующих этот интерфейс и инкапсулирующих в себе конкретные алгоритмы

Способ решения

Отделение процедуры выбора алгоритма от его реализации. Это позволяет сделать выбор на основании контекста.

Участники

Класс Strategy определяет, как будут использоваться различные алгоритмы.

Конкретные классы ConcreteStrategy реализуют эти различные алгоритмы.

Класс Context использует конкретные классы ConcreteStrategy посредством ссылки на конкретный тип абстрактного класса Strategy. Классы Strategy и Context взаимодействуют с целью реализации выбранного алгоритма (в некоторых случаях классу Strategy требуется посылать запросы классу Context). Класс Context пересылает классу Strategy запрос, поступивший от его класса-клиента.

Следствия

Шаблон Strategy определяет семейство алгоритмов.

Это позволяет отказаться от использования переключателей и/или условных операторов.

Вызов всех алгоритмов должен осуществляться стандартным образом (все они должны иметь одинаковый интерфейс).

Реализация

Класс, который использует алгоритм (Context), включает абстрактный класс (Strategy), обладающий абстрактным методом, определяющим способ вызова алгоритма. Каждый производный класс реализует один требуемый вариант алгоритма.

Замечание: метод вызова алгоритма не должен быть абстрактным, если требуется реализовать некоторое поведение, принимаемое по умолчанию.

Полезные сведения

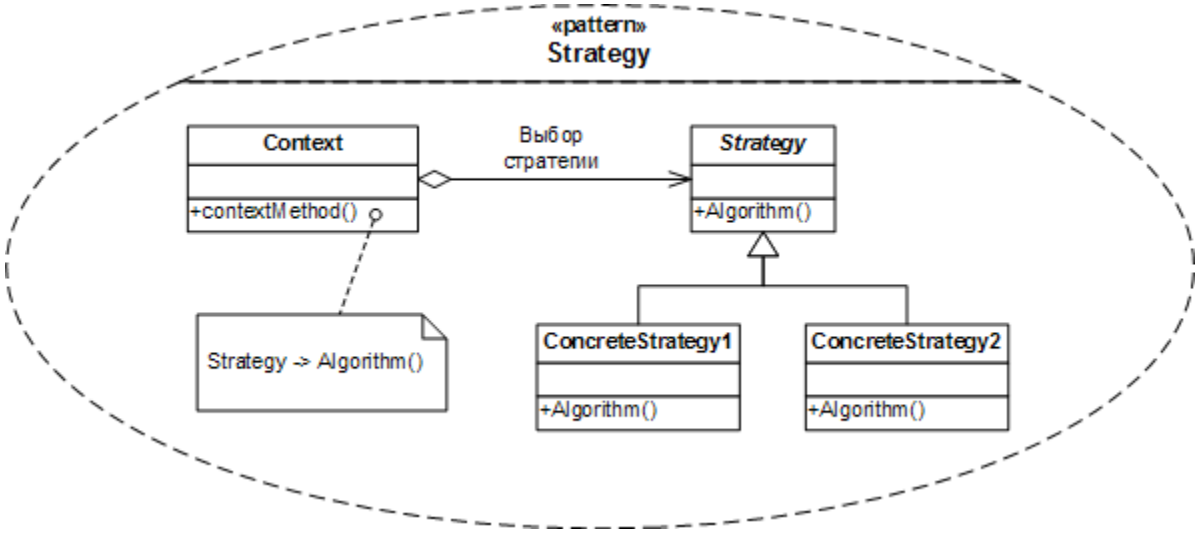
И стратегия, и декоратор может применяться для изменения поведения конкретных классов. Достоинство стратегии в том, что интерфейс кастомизации не совпадает с публичным интерфейсом и может быть куда более удобным, а недостаток в том, что для использования стратегии необходимо изначально проектировать класс с возможностью регистрации стратегий.

Использование

Архитектура Microsoft WDF основана на этом паттерне. У каждого объекта "драйвер" и "устройство" есть неизменяемая часть, вшитая в систему, в которой регистрируется изменяемая часть (стратегия), написанная в конкретной реализации. Изменяемая часть может быть и вовсе пустой, что даст ничего не делающий драйвер, но при этом способный участвовать в PnP и управлении питанием.

Библиотека ATL содержит в себе набор классов threading model, которые являются стратегиями (различными реализациями Lock/Unlock, которые потом используются основными классами системы). При этом в этих стратегиях используется статический полиморфизм через параметр шаблона, а не динамический полиморфизм через виртуальные методы.

Суктура



Пример реализации

```
using System;

namespace DesignPatterns.Behavioral.Strategy
{
    /// <summary>
    /// Интерфейс «Стратегия» определяет функциональность (в данном примере это метод
    /// <see cref="Algorithm">Algorithm</see>), которая должна быть реализована
    /// конкретными классами стратегий. Другими словами, метод интерфейса определяет
    /// решение некой задачи, а его реализации в конкретных классах стратегий определяют,
    /// КАК, КАКИМ ПУТЁМ эта задача будет решена.
    /// </summary>
    public interface IStrategy
    {
        void Algorithm();
    }

    /// <summary>
    /// Первая конкретная реализация-стратегия.
    /// </summary>
    public class ConcreteStrategy1 : IStrategy
    {
        public void Algorithm()
```

```
    {
        Console.WriteLine("Выполняется алгоритм стратегии 1.");
    }
}

/// <summary>
/// Вторая конкретная реализация-стратегия.
/// Реализаций может быть сколько угодно много.
/// </summary>
public class ConcreteStrategy2 : IStrategy
{
    public void Algorithm()
    {
        Console.WriteLine("Выполняется алгоритм стратегии 2.");
    }
}

/// <summary>
/// Контекст, использующий стратегию для решения своей задачи.
/// </summary>
public class Context
{
    /// <summary>
    /// Ссылка на интерфейс <see cref="IStrategy">IStrategy</see>
    /// позволяет автоматически переключаться между конкретными реализациями
    /// (другими словами, это выбор конкретной стратегии).
    /// </summary>
    private IStrategy _strategy;

    /// <summary>
    /// Конструктор контекста.
    /// Инициализирует объект стратегией.
    /// </summary>
    /// <param name="strategy">
    /// Стратегия.
    /// </param>
    public Context(IStrategy strategy)
    {
        _strategy = strategy;
    }

    /// <summary>
    /// Метод для установки стратегии.
    /// Служит для смены стратегии во время выполнения.
    /// В C# может быть реализован также как свойство записи.
    /// </summary>
    /// <param name="strategy">
    /// Новая стратегия.
    /// </param>
    public void SetStrategy(IStrategy strategy)
    {
        _strategy = strategy;
    }

    /// <summary>
    /// Некоторая функциональность контекста, которая выбирает
    /// стратегию и использует её для решения своей задачи.
    /// </summary>
    public void ExecuteOperation()
    {
        _strategy.Algorithm();
    }
}

/// <summary>
```

```
    /// Класс приложения.
    /// В данном примере выступает как клиент контекста.
    /// </summary>
    public static class Program
    {
        /// <summary>
        /// Точка входа в программу.
        /// </summary>
        public static void Main()
        {
            // Создём контекст и инициализируем его первой стратегией.
            Context context = new Context(new ConcreteStrategy1());
            // Выполняем операцию контекста, которая использует первую стратегию.
            context.ExecuteOperation();
            // Заменяем в контексте первую стратегию второй.
            context.SetStrategy(new ConcreteStrategy2());
            // Выполняем операцию контекста, которая теперь использует вторую стратегию.
            context.ExecuteOperation();
        }
    }
}
```

Template — Шаблонный метод

Шаблонный метод (Template method) — паттерн поведения классов, шаблон проектирования , определяющий основу алгоритма и позволяющий наследникам переопределять некоторые шаги алгоритма, не изменяя его структуру в целом.

Применимость

Однократное использование инвариантной части алгоритма, с оставлением изменяющейся части на усмотрение наследникам.

Локализация и вычленение общего для нескольких классов кода для избегания дублирования.

Разрешение расширения кода наследниками только в определенных местах.

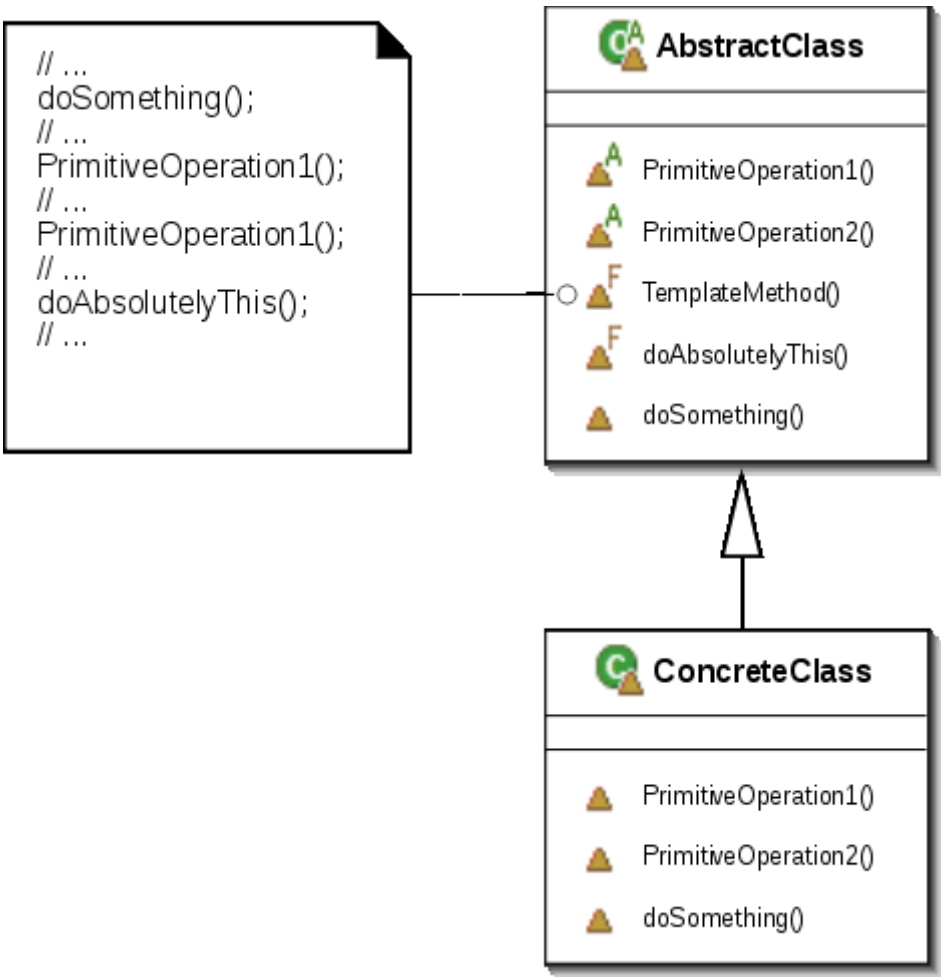
Участники

Abstract class (абстрактный класс) - определяет абстрактные операции, замещаемые в наследниках для реализации шагов алгоритма; реализует шаблонный метод, определяющий скелет алгоритма. Шаблонный метод вызывает замещаемые и другие, определенные в Abstract class, операции.

Concrete class (конкретный класс) - реализует замещаемые операции необходимым для данной реализации способом.

Concrete class предполагает, что инвариантные шаги алгоритма будут выполнены в AbstractClass.

Сруктура



Пример реализации

В примере шаблонный метод реализуется для игр, в которых игроки по очереди делают свой ход.

```
/**
 * An abstract class that is common to several games in
 * which players play against the others, but only one is
 * playing at a given time.
 */

namespace Design_Patterns
{
    class TemplateMethodPattern
    {
        internal abstract class GameObject
        {
            protected int PlayersCount;

            abstract protected void InitializeGame();

            abstract protected void MakePlay(int player);

            abstract protected bool EndOfGame();

            abstract protected void PrintWinner();

            /* A template method : */
            public void PlayOneGame(int playersCount)
            {
                PlayersCount = playersCount;
                InitializeGame();

                var j = 0;

                while (!EndOfGame())
                {
                    MakePlay(j);
                    j = (j + 1) % playersCount;
                }

                PrintWinner();
            }
        }

        //Now we can extend this class in order to implement actual games:

        public class Monopoly : GameObject
        {
            /* Implementation of necessary concrete methods */

            protected override void InitializeGame()
            {
                // Initialize money
            }

            protected override void MakePlay(int player)
            {
                // Process one turn of player
            }

            protected override bool EndOfGame()
            {
                return true;
            }
```

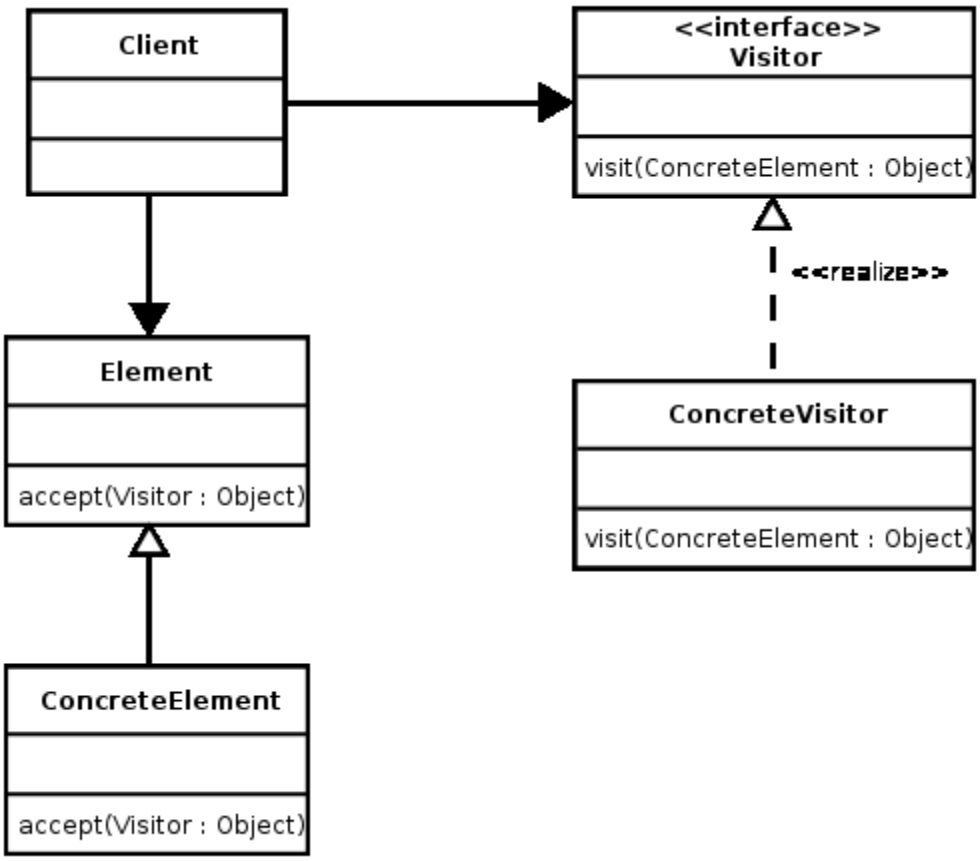
```
            protected override void PrintWinner()
            {
                // Display who won
            }

            /* Specific declarations for the Monopoly game. */

            // ...

        }

        public class Chess : GameObject
        {

            /* Implementation of necessary concrete methods */

            protected override void InitializeGame()
            {
                // Put the pieces on the board
            }

            protected override void MakePlay(int player)
            {
                // Process a turn for the player
            }

            protected override bool EndOfGame()
            {
                return true;
                // Return true if in Checkmate or Stalemate has been reached
            }

            protected override void PrintWinner()
            {
                // Display the winning player
            }

            /* Specific declarations for the chess game. */

            // ...

        }

        public static void Test()
        {
            GameObject game = new Monopoly();

            game.PlayOneGame(2);
        }
    }
}
```

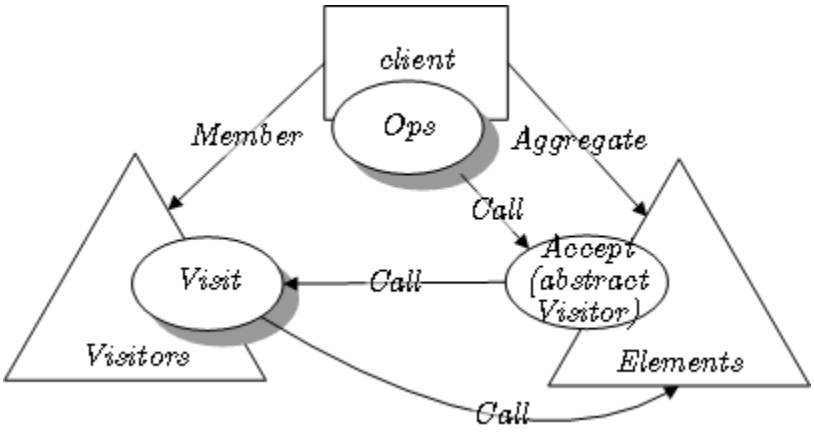
Visitor — Посетитель

Шаблон Посетитель (Visitor) — поведенческий Шаблон проектирования.

Описывает операцию, которая выполняется над объектами других классов. При изменении Visitor нет необходимости изменять обслуживаемые классы.

Структура





Описание средствами псевдокода

```
interface Obj
{
    void visit(Visitor visitor, params);
}

interface Visitor
{
```

```
    void visitA(A a, params);
    void visitB(B b, params);
}

class A implements Obj
{
    void visit(Visitor visitor, params) { visitor.visitA(this, params); }
}

class B implements Obj
{
    void visit(Visitor visitor, params) { visitor.visitB(this, params); }
}

class Visitor1 implements Visitor
{
    void visitA(A a, params);
    void visitB(B b, params);
}

class Visitor2 implements Visitor
{
    void visitA(A a, params);
    void visitB(B b, params);
}
```

Проблема

Над каждым объектом некоторой структуры выполняется одна или более операций. Определить новую операцию, не изменяя классы объектов.

Решение

Для полной независимости посетители имеют отдельную от обслуживаемых структур иерархию. Структуры должны иметь некий интерфейс взаимодействия. При необходимости добавления новых операций необходимо создать новый класс ConcreteVisitor и поместить его в цепочку обхода обслуживаемых структур.

Рекомендации

Шаблон «Посетитель» следует использовать, если:

- в структуре присутствуют объекты разных классов с различными интерфейсами, и необходимо выполнить над ними операции, зависящие от конкретных классов.

- если над обслуживаемой структурой надо выполнять самые различные, порой не связанные между собой операции. То есть они усложняют эту структуру.

- часто добавляются новые операции над обслуживаемой структурой.

- реализация double dispatch. Концептуально это нечто вроде {a; b} -> method(params), где реально вызываемый по стрелочке метод зависит как от типа a, так и от типа b. Так как большинство объектно-ориентированных языков программирования не поддерживает такое на уровне синтаксиса, для такого обычно применяется Visitor в виде a -> visit(b, params), который в свою очередь вызывает b -> visitA(a, params), что дает выбор и по типу a, и по типу b.

Преимущества

- упрощается добавление новых операций

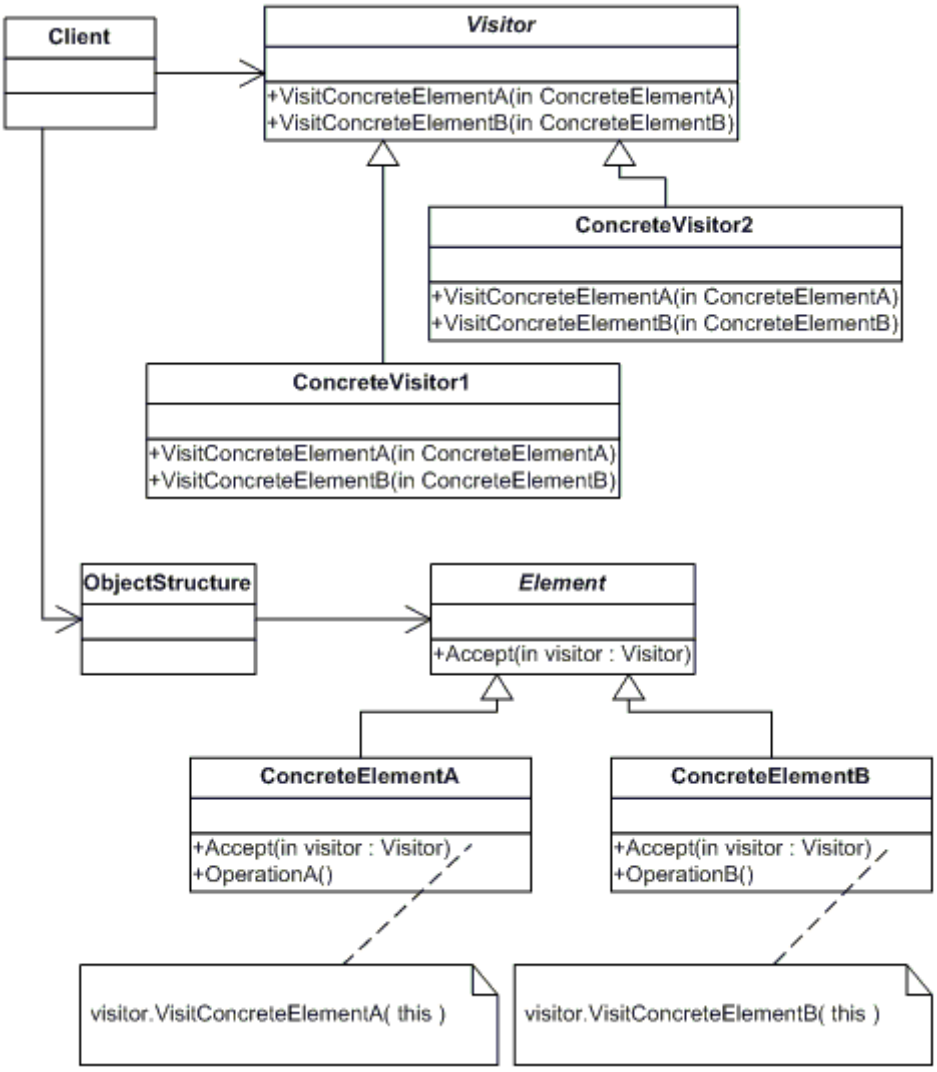
- объединяет родственные операции в классе «Посетитель».

- экземпляр визитора может иметь в себе состояние (например, общую сумму) и накапливать его по ходу обхода контейнера.

Недостатки

Затруднено добавление новых классов, поскольку требуется объявление новой абстрактной операции в интерфейсе визитора, а значит — и во всех классах, реализующих данный интерфейс.

Пример реализации



The classes and/or objects participating in this pattern are:

Visitor (Visitor) declares a Visit operation for each class of *ConcreteElement* in the object structure. The operation's name and signature identifies the class that sends the Visit request to the visitor. That lets the visitor determine the concrete class of the element being visited. Then the visitor can access the elements directly through its particular interface

ConcreteVisitor (IncomeVisitor, VacationVisitor) implements each operation declared by *Visitor*. Each operation implements a fragment of the algorithm defined for the corresponding class or object in the structure. *ConcreteVisitor* provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure.

Element (Element) defines an Accept operation that takes a visitor as an argument.

ConcreteElement (Employee) implements an Accept operation that takes a visitor as an argument

ObjectStructure (Employees) can enumerate its elements

may provide a high-level interface to allow the visitor to visit its elements

may either be a Composite (pattern) or a collection such as a list or a set

This *structural* code demonstrates the Visitor pattern in which an object traverses an object structure and performs the same operation on each node in this structure. Different visitor objects define different operations.

```
// Visitor pattern -- Structural example

using System;
using System.Collections.Generic;

namespace DoFactory.GangOfFour.Visitor.Structural
{
    /// <summary>
    /// MainApp startup class for Structural
    /// Visitor Design Pattern.
    /// </summary>
    class MainApp
    {
        static void Main()
        {
            // Setup structure
            ObjectStructure o = new ObjectStructure();
            o.Attach(new ConcreteElementA());
            o.Attach(new ConcreteElementB());

            // Create visitor objects
            ConcreteVisitor1 v1 = new ConcreteVisitor1();
            ConcreteVisitor2 v2 = new ConcreteVisitor2();

            // Structure accepting visitors
            o.Accept(v1);
            o.Accept(v2);

            // Wait for user
            Console.ReadKey();
        }
    }

    /// <summary>
    /// The 'Visitor' abstract class
    /// </summary>
    abstract class Visitor
    {
        public abstract void VisitConcreteElementA(
          ConcreteElementA concreteElementA);

        public abstract void VisitConcreteElementB(
          ConcreteElementB concreteElementB);
    }

    /// <summary>
    /// A 'ConcreteVisitor' class
    /// </summary>
    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);
    }
}

/// <summary>
/// A 'ConcreteVisitor' class
/// </summary>
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);
    }
}

/// <summary>
/// The 'Element' abstract class
/// </summary>
abstract class Element
{
    public abstract void Accept(Visitor visitor);
}

/// <summary>
/// A 'ConcreteElement' class
/// </summary>
class ConcreteElementA : Element
{
    public override void Accept(Visitor visitor)
    {
        visitor.VisitConcreteElementA(this);
    }

    public void OperationA()
    {
    }
}

/// <summary>
/// A 'ConcreteElement' class
/// </summary>
class ConcreteElementB : Element
{
    public override void Accept(Visitor visitor)
    {
        visitor.VisitConcreteElementB(this);
    }

    public void OperationB()
    {
    }
```

```
    }

    /// <summary>
    /// The 'ObjectStructure' class
    /// </summary>
    class ObjectStructure
    {
        private 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);
            }
        }
    }
}
```

```
Output
 ConcreteElementA visited by ConcreteVisitor1
 ConcreteElementB visited by ConcreteVisitor1
 ConcreteElementA visited by ConcreteVisitor2
 ConcreteElementB visited by ConcreteVisitor2
```

This *real-world* code demonstrates the Visitor pattern in which two objects traverse a list of Employees and performs the same operation on each Employee. The two visitor objects define different operations -- one adjusts vacation days and the other income.

```
// Visitor pattern -- Real World example
using System;
using System.Collections.Generic;

namespace DoFactory.GangOfFour.Visitor.RealWorld
{
    /// <summary>
    /// MainApp startup class for Real-World
    /// Visitor Design Pattern.
    /// </summary>
    class MainApp
    {
        /// <summary>
        /// Entry point into console application.
        /// </summary>
        static void Main()
        {
            // Setup employee collection
            Employees e = new Employees();
            e.Attach(new Clerk());
            e.Attach(new Director());
            e.Attach(new President());

            // Employees are 'visited'
```

```
            e.Accept(new IncomeVisitor());
            e.Accept(new VacationVisitor());

            // Wait for user
            Console.ReadKey();
        }
    }

    /// <summary>
    /// The 'Visitor' interface
    /// </summary>
    interface IVisitor
    {
        void Visit(Element element);
    }

    /// <summary>
    /// A 'ConcreteVisitor' class
    /// </summary>
    class IncomeVisitor : IVisitor
    {
        public void Visit(Element element)
        {
            Employee employee = element as Employee;

            // Provide 10% pay raise
            employee.Income *= 1.10;

            Console.WriteLine("{0} {1}'s new income: {2:C}",
              employee.GetType().Name, employee.Name,
              employee.Income);
        }
    }

    /// <summary>
    /// A 'ConcreteVisitor' class
    /// </summary>
    class VacationVisitor : IVisitor
    {
        public void Visit(Element element)
        {
            Employee employee = element as Employee;

            // Provide 3 extra vacation days
            Console.WriteLine("{0} {1}'s new vacation days: {2}",
              employee.GetType().Name, employee.Name,
              employee.VacationDays);
        }
    }

    /// <summary>
    /// The 'Element' abstract class
    /// </summary>
    abstract class Element
    {
        public abstract void Accept(IVisitor visitor);
    }

    /// <summary>
    /// The 'ConcreteElement' class
    /// </summary>
    class Employee : Element
    {
        private string _name;
        private double _income;
```

```
    private int _vacationDays;

    // Constructor
    public Employee(string name, double income, int vacationDays)
    {
        this._name = name;
        this._income = income;
        this._vacationDays = vacationDays;
    }

    // Gets or sets the name
    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            _name = value;
        }
    }

    // Gets or sets income
    public double Income
    {
        get
        {
            return _income;
        }
        set
        {
            _income = value;
        }
    }

    // Gets or sets number of vacation days
    public int VacationDays
    {
        get
        {
            return _vacationDays;
        }

        set
        {
            _vacationDays = value;
        }
    }

    public override void Accept(IVisitor visitor)
    {
        visitor.Visit(this);
    }
}

/// <summary>
/// The 'ObjectStructure' class
/// </summary>
class Employees
{
    private List<Employee> _employees = new List<Employee>();

    public void Attach(Employee employee)
    {
```

```
            _employees.Add(employee);
        }

        public void Detach(Employee employee)
        {
            _employees.Remove(employee);
        }

        public void Accept(IVisitor visitor)
        {
            foreach (Employee e in _employees)
            {
                e.Accept(visitor);
            }

            Console.WriteLine();
        }
    }

    // Three employee types
    class Clerk : Employee
    {
        // Constructor
        public Clerk() : base("Hank", 25000.0, 14)
        {
        }
    }

    class Director : Employee
    {
        // Constructor
        public Director() : base("Elly", 35000.0, 16)
        {
        }
    }

    class President : Employee
    {
        // Constructor
        public President() : base("Dick", 45000.0, 21)
        {
        }
    }
}
```

```
Output
 Clerk Hank's new income: $27,500.00
 Director Elly's new income: $38,500.00
 President Dick's new income: $49,500.00

 Clerk Hank's new vacation days: 14
 Director Elly's new vacation days: 16
 President Dick's new vacation days: 21
```

Null Object (Null object)

In object-oriented computer programming, a **Null Object** is an object with defined neutral ("null") behavior. The Null Object design pattern describes the uses of such objects and their behavior (or lack thereof). It was first published in the Pattern Languages of Program Design book series.

Мотивация

In most object-oriented languages, such as Java or C#, references may be null. These references need to be checked to ensure they are not null before invoking any methods, because methods typically cannot be invoked on null references.

The Objective-C language takes another approach to this problem and does not invoke methods on nil but instead returns nil for all such invocations.

Описание

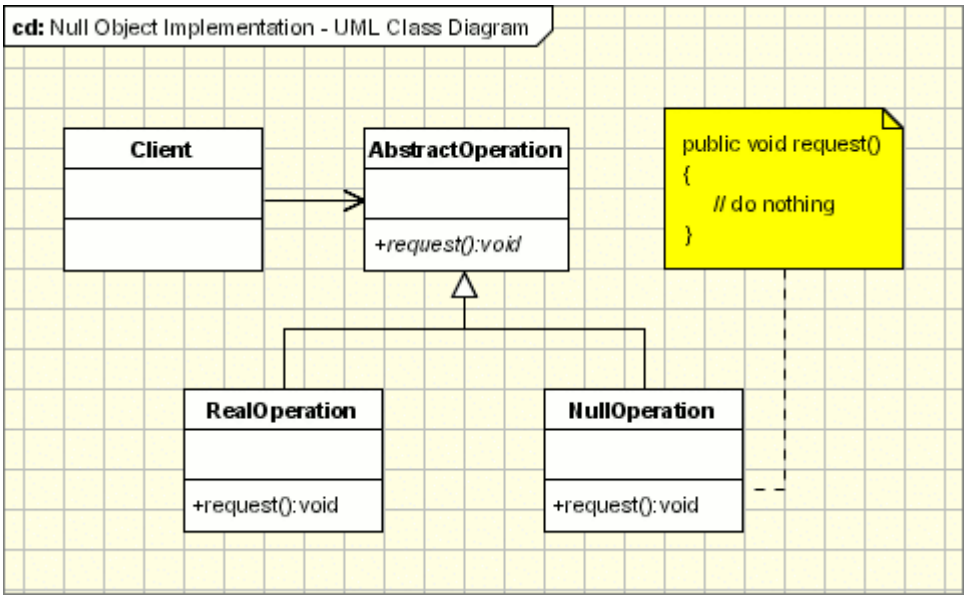
Instead of using a null reference to convey absence of an object (for instance, a non-existent customer), one uses an object which implements the expected interface, but whose method body is empty. The advantage of this approach over a working default implementation is that a Null Object is very predictable and has no side effects: it does nothing.

For example, a function may retrieve a list of files in a directory and perform some action on each. In the case of an empty directory, one response may be to throw an exception or return a null reference rather than a list. Thus, the code which expects a list must verify that it in fact has one before continuing, which can complicate the design.

By returning a null object (i.e. an empty list) instead, there is no need to verify that the return value is in fact a list. The calling function may simply iterate the list as normal, effectively doing nothing. It is, however, still possible to check whether the return value is a null object (e.g. an empty list) and react differently if desired.

The null object pattern can also be used to act as a stub for testing if a certain feature, such as a database, is not available for testing.

Структура



Реализация

The participants classes in this pattern are:

AbstractClass - defines abstract primitive operations that concrete implementations have to define.

RealClass - a real implementation of the *AbstractClass* performing some real actions.

NullClass - a implementation which do nothing of the abstract class, in order to provide a non-null object to the client.

Client - the client gets an implementation of the abstract class and uses it. It doesn't really care if the implementation is a null object or an real object since both of them are used in the same way.

Пример

Given a binary tree, with this node structure:

```
class node {
  node left
  node right
}
```

One may implement a tree size procedure recursively:

```
function tree_size(node) {
  return 1 + tree_size(node.left) + tree_size(node.right)
}
```

Since the child nodes may not exist, one must modify the procedure by adding non-existence or null checks:

```
function tree_size(node) {
  set sum = 1
  if node.left exists {
    sum = sum + tree_size(node.left)
  }
  if node.right exists {
    sum = sum + tree_size(node.right)
  }
  return sum
}
```

This however makes the procedure more complicated by mixing boundary checks with normal logic, and it becomes harder to read. Using the null object pattern, one can create a special version of the procedure but only for null nodes:

```
function tree_size(node) {
  return 1 + tree_size(node.left) + tree_size(node.right)
}
function tree_size(null_node) {
  return 0
}
```

This separates normal logic from special case handling, and makes the code easier to understand.

Связь с другими патернами

It can be regarded as a special case of the State pattern and the Strategy pattern.

It is not a pattern from Design Patterns, but is mentioned in Martin Fowler's Refactoring and Joshua Kerievsky's book on refactoring in the Insert Null Object refactoring.

Критика и комментарии

This pattern should be used carefully as it can make errors/bugs appear as normal program execution.

Пример реализации

C# is a language in which the Null Object pattern can be properly implemented. This example shows animal objects that display sounds and a NullAnimal instance used in place of the C# null keyword. The Null Object provides consistent behaviour and prevents a runtime Null Reference Exception that would occur if the C# null keyword were used instead.

```
/* Null Object Pattern implementation:
 */
using System;

// Animal interface is the key to compatibility for Animal implementations below.
interface IAnimal
{
    void MakeSound();
}

// Dog is a real animal.
class Dog : IAnimal
{
    public void MakeSound()
    {
        Console.WriteLine("Woof!");
    }
}

// The Null Case: this NullAnimal class should be instantiated and used in place of C# null
keyword.
class NullAnimal : IAnimal
{
    public void MakeSound()
    {
        // Purposefully provides no behaviour.
    }
}

/* ========================
 * Simplistic usage example in a Main entry point.
 */
static class Program
{
    static void Main()
    {
        IAnimal dog = new Dog();
        dog.MakeSound(); // outputs "Woof!"

        /* Instead of using C# null, use a NullAnimal instance.
         * This example is simplistic but conveys the idea that if a NullAnimal instance is used
then the program
         * will never experience a .NET System.NullReferenceException at runtime, unlike if C#
null was used.
         */
        IAnimal unknown = new NullAnimal();  //<< replaces: IAnimal unknown = null;
        unknown.MakeSound(); // outputs nothing, but does not throw a runtime exception
    }
}
```

Слуга (Servant)

Servant is a design pattern used to offer some functionality to a group of classes without defining that functionality in each of them. A Servant is a class whose instance (or even just class) provides methods that take care of a desired service, while objects for which (or with whom) the servant does something, are taken as parameters.

Описание

Servant is used for providing some behavior to a group of classes. Instead of defining that behavior in each class - or when we cannot factor out this behavior in the common parent class - it is defined once in the Servant.

For example: we have a few classes representing geometric objects (rectangle, ellipse, and triangle). We can draw these objects on some canvas. When we need to provide a “move” method for these objects we could implement this method in each class, or we can define an interface they implement and then offer the “move” functionality in a servant. An interface is defined to ensure that serviced classes have methods, that servant needs to provide desired behavior. If we continue in our example, we define an Interface “Movable” specifying that every class implementing this interface needs to implement method “getPosition” and “setPosition”. The first method gets the position of an object on a canvas and second one sets the position of an object and draws it on a canvas. Then we define a servant class “MoveServant”, which has two methods “moveTo(Movable movedObject, Position where)” and moveBy(Movable movedObject, int dx, int dy). The Servant class can now be used to move every object which implements the Movable. Thus the “moving” code appears in only one class which respects the “Separation of Concerns” rule.

Структура

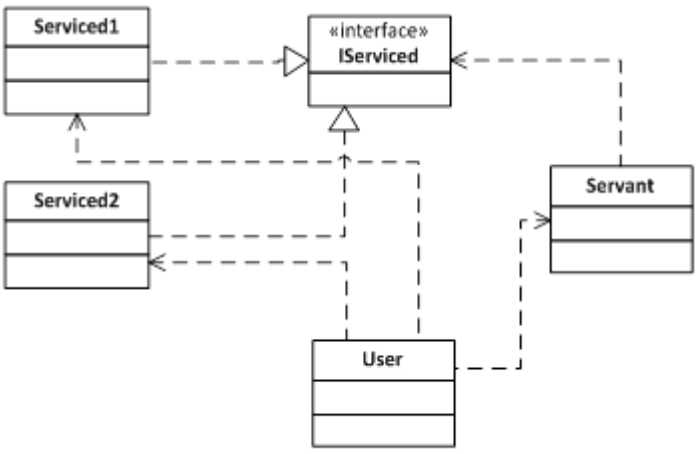
Two ways of implementation

There are two ways to implement this design pattern.

User knows the servant (in which case he doesn’t need to know the serviced classes) and sends messages with his requests to the servant instances, passing the serviced objects as parameters.

Serviced instances know the servant and the user sends them messages with his requests (in which case she doesn’t have to know the servant). The serviced instances then send messages to the instances of servant, asking for service.

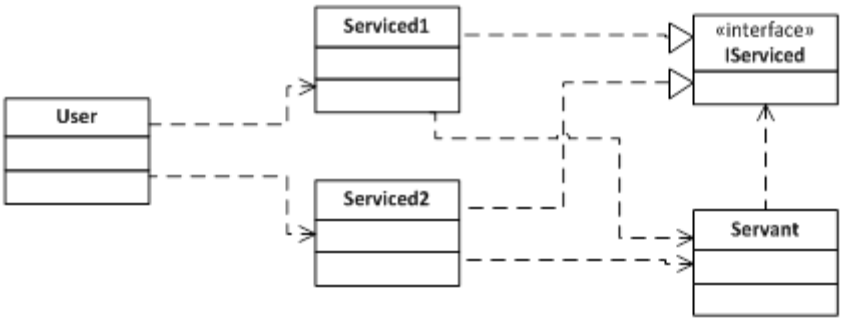
The serviced classes (geometric objects from our example) don’t know about servant, but they implement the “IServiced” interface. The user class just calls the method of servant and passes serviced objects as parameters. This situation is shown on figure 1.



User uses servant to achieve some functionality and passes the serviced objects as parameters.

On figure 2 is shown opposite situation, where user don't know about servant class and calls directly serviced classes. Serviced classes then asks servant themselves to achieve desired functionality.

User requests operations from serviced instances, which then asks servant to do it for them.



Реализаци

Analyze what behavior servant should take care of. State what methods servant will define and what these methods will need from serviced parameter. By other words, what serviced instance must provide, so that servants methods can achieve their goals.

Analyze what abilities serviced classes must have, so they can be properly serviced.

We define an interface, which will enforce implementation of declared methods.

Define an interface specifying requested behavior of serviced objects. If some instance wants to be served by servant, it must implement this interface.

Define (or acquire somehow) specified servant (his class).

Implement defined interface with serviced classes.

Пример реализации

```
// Servant class, offering its functionality to classes implementing
// Movable Interface
public class MoveServant {
        // Method, which will move Movable implementing class to position where
        public void moveTo(Movable serviced, Position where) {
                // Do some other stuff to ensure it moves smoothly and nicely, this is
                // the place to offer the functionality
                serviced.setPosition(where);
        }

        // Method, which will move Movable implementing class by dx and dy
        public void moveBy(Movable serviced, int dx, int dy) {
                // this is the place to offer the functionality
                dx += serviced.getPosition().xPosition;
                dy += serviced.getPosition().yPosition;
                serviced.setPosition(new Position(dx, dy));
        }
}

// Interface specifying what serviced classes needs to implement, to be
// serviced by servant.
public interface Movable {
        public void setPosition(Position p);
```

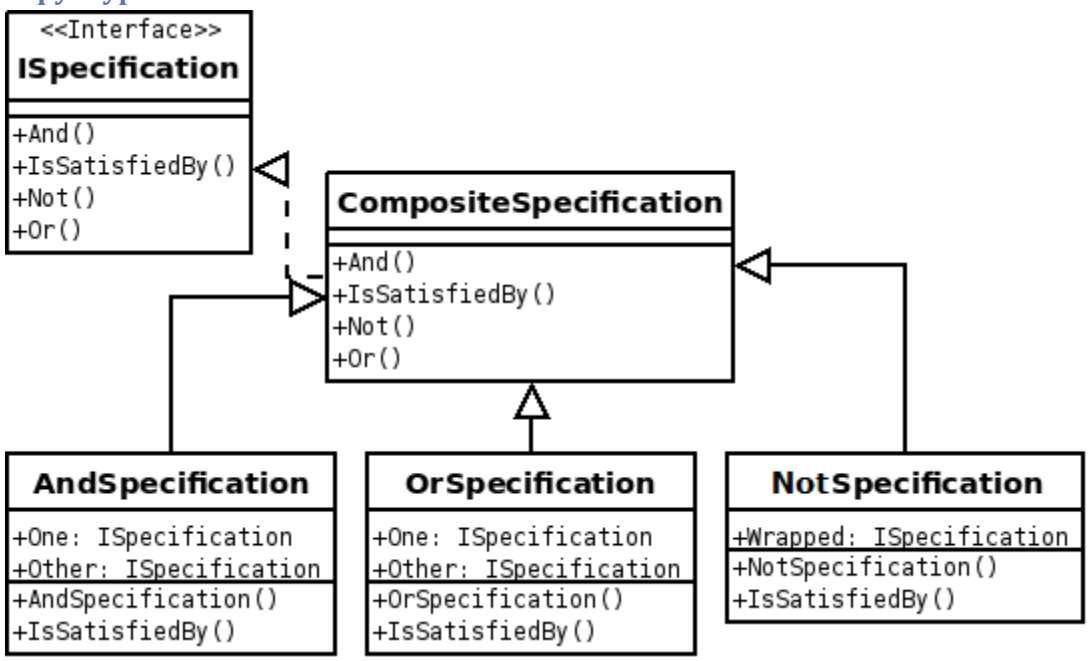
```
        public Position getPosition();
}

// One of geometric classes
public class Triangle implements Movable {
        // Position of the geometric object on some canvas
        private Position p;

        // Method, which sets position of geometric object
        public void setPosition(Position p) {
                this.p = p;
        }

        // Method, which returns position of geometric object
        public Position getPosition() {
                return this.p;
        }
}

// One of geometric classes
public class Ellipse implements Movable {
        // Position of the geometric object on some canvas
        private Position p;

        // Method, which sets position of geometric object
        public void setPosition(Position p) {
                this.p = p;
        }

        // Method, which returns position of geometric object
        public Position getPosition() {
                return this.p;
        }
}

// One of geometric classes
public class Rectangle implements Movable {
        // Position of the geometric object on some canvas
        private Position p;

        // Method, which sets position of geometric object
        public void setPosition(Position p) {
                this.p = p;
        }

        // Method, which returns position of geometric object
        public Position getPosition() {
                return this.p;
        }
}

// Just a very simple container class for position.
public class Position {
        public int xPosition;
        public int yPosition;

        public Position(int dx, int dy) {
                xPosition = dx;
                yPosition = dy;
        }
}
```

Specification (Specification)

In computer programming, the specification pattern is a particular software design pattern, whereby business rules can be recombined by chaining the business rules together using boolean logic.

A **specification pattern** outlines a business rule that is combinable with other business rules. In this pattern, a unit of business logic inherits its functionality from the abstract aggregate Composite Specification class. The Composite Specification class has one function called IsSatisfiedBy that returns a boolean value. After instantiation, the specification is "chained" with other specifications, making new specifications easily maintainable, yet highly customizable business logic. Furthermore upon instantiation the business logic may, through method invocation or inversion of control, have its state altered in order to become a delegate of other classes such as a persistence repository.

Структура



Пример реализации

```
public interface ISpecification<TEntity>
{
    bool IsSatisfiedBy(TEntity entity);
}

internal class AndSpecification<TEntity> : ISpecification<TEntity>
{
    private ISpecification<TEntity> Spec1;
    private ISpecification<TEntity> Spec2;

    internal AndSpecification(ISpecification<TEntity> s1, ISpecification<TEntity> s2)
    {
        Spec1 = s1;
        Spec2 = s2;
    }

    public bool IsSatisfiedBy(TEntity candidate)
    {
        return Spec1.IsSatisfiedBy(candidate) && Spec2.IsSatisfiedBy(candidate);
    }
}

internal class OrSpecification<TEntity> : ISpecification<TEntity>
{
```

```
    private ISpecification<TEntity> Spec1;
    private ISpecification<TEntity> Spec2;

    internal OrSpecification(ISpecification<TEntity> s1, ISpecification<TEntity> s2)
    {
        Spec1 = s1;
        Spec2 = s2;
    }

    public bool IsSatisfiedBy(TEntity candidate)
    {
        return Spec1.IsSatisfiedBy(candidate) || Spec2.IsSatisfiedBy(candidate);
    }
}

internal class NotSpecification<TEntity> : ISpecification<TEntity>
{
    private ISpecification<TEntity> Wrapped;

    internal NotSpecification(ISpecification<TEntity> x)
    {
        Wrapped = x;
    }

    public bool IsSatisfiedBy(TEntity candidate)
    {
        return !Wrapped.IsSatisfiedBy(candidate);
    }
}

public static class ExtensionMethods
{
    public static ISpecification<TEntity> And<TEntity>(this ISpecification<TEntity> s1,
ISpecification<TEntity> s2)
    {
        return new AndSpecification<TEntity>(s1, s2);
    }

    public static ISpecification<TEntity> Or<TEntity>(this ISpecification<TEntity> s1,
ISpecification<TEntity> s2)
    {
        return new OrSpecification<TEntity>(s1, s2);
    }

    public static ISpecification<TEntity> Not<TEntity>(this ISpecification<TEntity> s)
    {
        return new NotSpecification<TEntity>(s);
    }
}
```

Пример использования

In this example, we are retrieving invoices and sending them to a collection agency if they are overdue, notices have been sent and they are not already with the collection agency.

We previously defined an OverdueSpecification class that it is satisfied when an invoice's due date is 30 days or older, a NoticeSentSpecification class that is satisfied when three notices have been sent to the customer, and an InCollectionSpecification class that is satisfied when an invoice has already been sent to the collection agency.

Using these three specifications, we created a new specification called SendToCollection which will be satisfied when an invoice is overdue, when notices have been sent to the customer, and are not already with the collection agency.

```
OverDueSpecification OverDue = new OverDueSpecification();
NoticeSentSpecification NoticeSent = new NoticeSentSpecification();
InCollectionSpecification InCollection = new InCollectionSpecification();

ISpecification SendToCollection = OverDue.And(NoticeSent).And(InCollection.Not());

InvoiceCollection = Service.GetInvoices();

foreach (Invoice currentInvoice in InvoiceCollection) {
    if (SendToCollection.IsSatisfiedBy(currentInvoice))  {
        currentInvoice.SendToCollection();
    }
}
```

Simple Policy

This is a **simple policy** based design sample. They are File Name Policy classes that I use the policy based analysis to design policies. In policy based design, the most important thing is analysis. At first, we must define the policy combined roles and implement them in the policy host classes. Then we must decompose the classes to small policy classes.

C# does not support multiple inheritance, but the multiple inheritance is the key technique in policy based design, so we must use interface to solve it.

Обзор

The central idiom in policy-based design is a class template (called the host class), taking several type parameters as input, which are instantiated with types selected by the user (called policy classes), each implementing a particular implicit interface (called a policy), and encapsulating some orthogonal (or mostly orthogonal) aspect of the behavior of the instantiated host class. By supplying a host class combined with a set of different, canned implementations for each policy, a library or module can support an exponential number of different behavior combinations, resolved at compile time, and selected by mixing and matching the different supplied policy classes in the instantiation of the host class template. Additionally, by writing a custom implementation of a given policy, a policy-based library can be used in situations requiring behaviors unforeseen by the library implementor. Even in cases where no more than one implementation of each policy will ever be used, decomposing a class into policies can aid the design process, by increasing modularity and highlighting exactly where orthogonal design decisions have been made.

While assembling software components out of interchangeable modules, communicating with each other through generic interfaces, is far from a new concept, policy-based design represents an innovation in the way it applies that concept at the (relatively low) level of defining the behavior of an individual class.

Policy classes have some similarity to callbacks, but differ in that, rather than consisting of a single function, a policy class will typically contain several related functions (methods), often combined with state variables and/or other facilities such as nested types.

A policy-based host class can be thought of as a type of metafunction, taking a set of behaviors represented by types as input, and returning as output a type representing the result of combining those behaviors into a functioning whole. (Unlike MPL metafunctions, however, the output is usually represented by the instantiated host class itself, rather than a nested output type.)

A key feature of the policy idiom is that, usually (though it is not strictly necessary), the host class will derive from (make itself a child class of) each of its policy classes using (public) multiple inheritance. (Alternatives are for the host class to merely contain a member variable of each policy class type, or else to inherit the policy classes privately; however inheriting the policy classes publicly has the major advantage that a policy class can add new methods, inherited by the instantiated host class and accessible to its users, which the host class itself need not even know about.) A notable feature of this aspect of the policy idiom is that, relative to object-oriented programming, policies invert the relationship between base class and derived class - whereas in OOP interfaces are traditionally represented by (abstract) base classes and implementations of interfaces by derived classes, in policy-based design the derived (host) class represents the interfaces and the base (policy) classes implement them. It should also be noted that in the case of policies, the public inheritance does not represent an is-a relationship between the host and the policy classes. While this would traditionally be considered evidence of a design defect in OOP contexts, this doesn't apply in the context of the policy idiom.

A disadvantage of policies in their current incarnation is that the policy interface doesn't have a direct, explicit representation in code, but rather is defined implicitly, via duck typing, and must be documented separately and manually, in comments.

The main idea is to use commonality-variability analysis to divide the type into the fixed implementation and interface, the policy-based class, and the different policies. The trick is to know what goes into the main class, and what policies should one create. Andrei's excellent article, mentioned above, gives us the clue: wherever we would need to make a possible limiting design decision, we should postpone that decision, we should delegate it to an appropriately named policy.

Policy classes can contain implementation, type definitions and so forth. Basically, the designer of the main template class will define what the policy classes should provide, what customization points they need to implement.

As we go by the analysis in policy-based design, it is a delicate task to create a good set of policies, just the right number. As little as necessary, but not less. The different customization points, which belong together, should go into one policy argument, such as storage policy, validation policy and so forth. A good rule of thumb during design is that you should be able to give a name to your policy, which represents a concept, and not one which represent an operation or some really tiny implementation detail. Persistence policy seems to be a good choice, while how to save policy does not.

As you do your policy-based design you will see how many other techniques will be useful, even if changed a bit, during your work. One example is that the template method pattern can be reinterpreted for compile time; so that your main class has a skeleton algorithm, which — at customization points — calls the appropriate functions of some of the policies. You will also find yourself in using your policy classes as traits are used, asking type information, delegating type related tasks to it, a storage policy is one example where it can happen.

Простыми словами

I love to eat food, so I thought about the policy recipe of cuisine. Maybe I can define a role to make a cuisine as follows:

Cuisine = Food + Flavor + Cooking

- The cuisine is the host combinable role. Food, Flavor and Cooking are policy groups.

- Food: Beef, Pork, Chicken, Spaghetti ...

- Flavor: Acid, Sweet, Hot, Curry ...

- Cooking: Roast, Cook, Fry, Stew ...

- Cuisine(Steak) = Food(Beef) + Flavor(None) + Cooking(Roast)

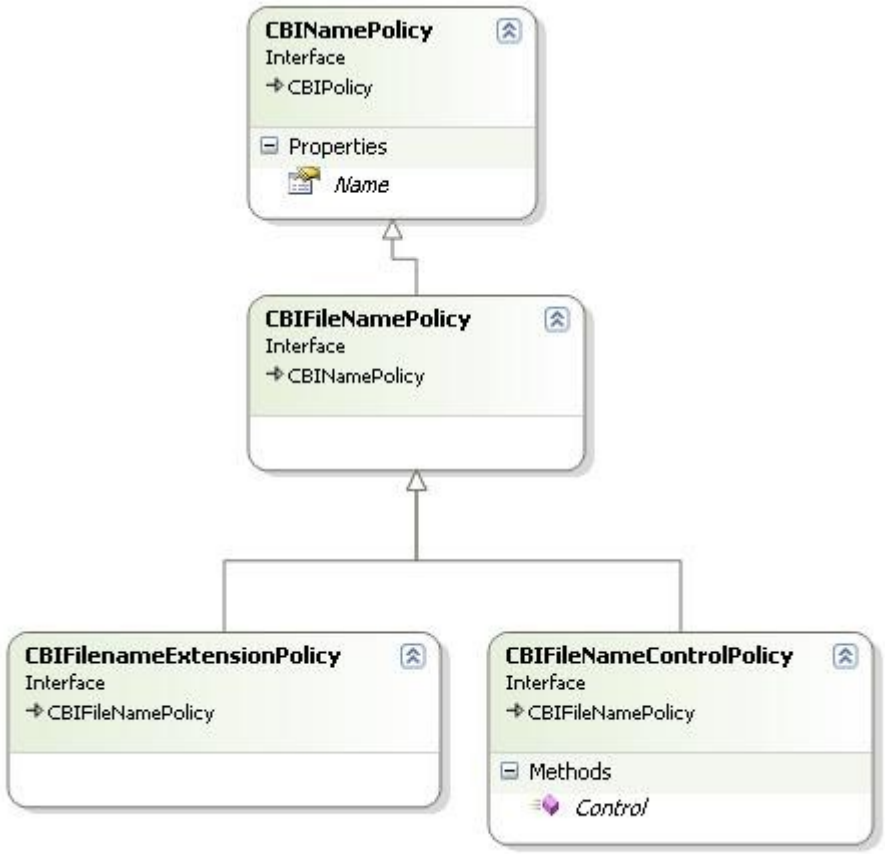
- Cuisine(Roast chicken) = Food(Chicken) + Flavor(Curry) + Cooking(Roast)

The recipe of cuisine is a combination of food, flavor and cooking.

A different policy can combine a different cuisine.

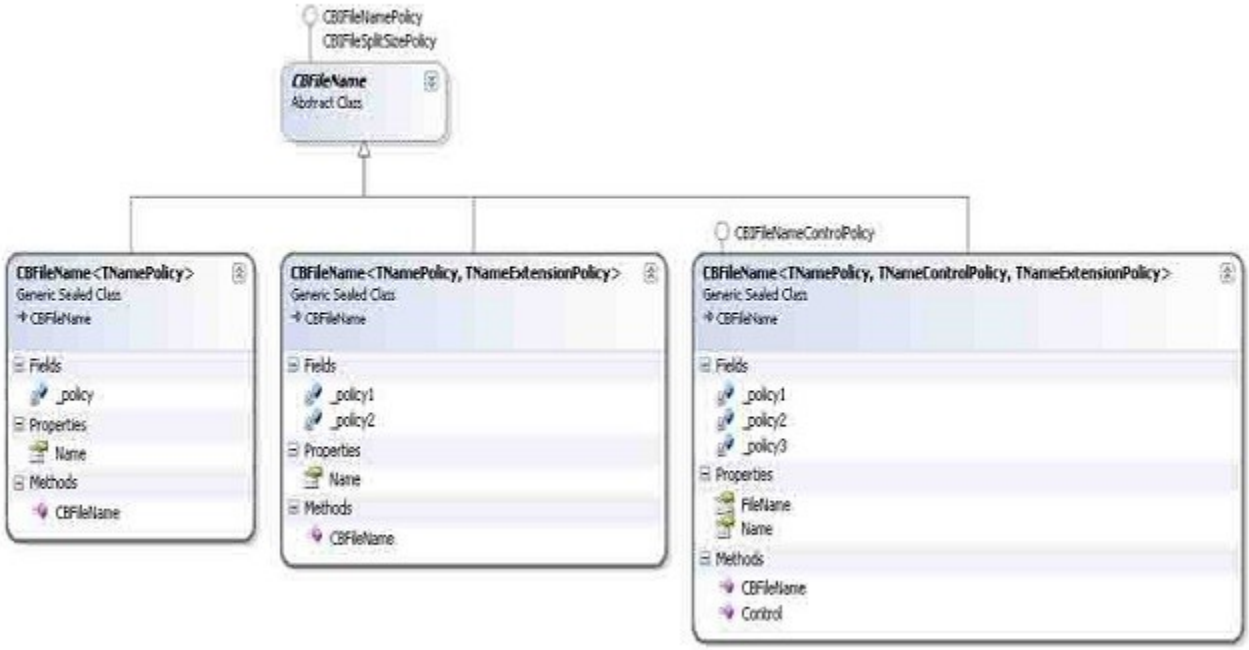
I think this explanation is easiest to understand.

Сруктура

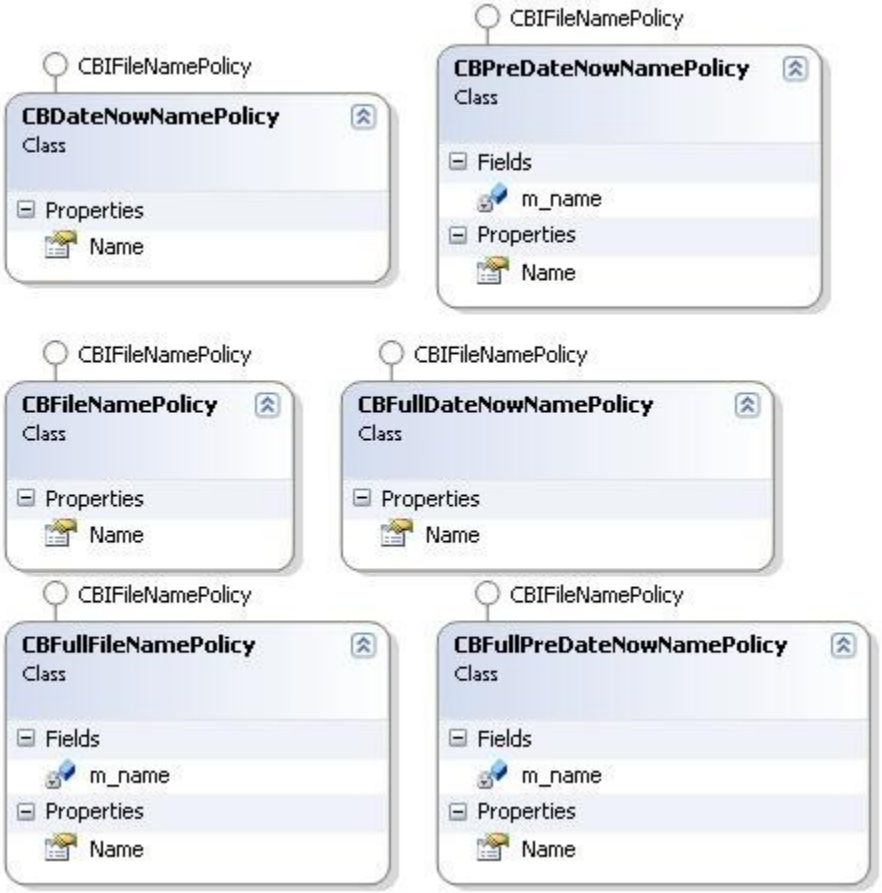


The base interface is CBINamePolicy, it just has a property Name.

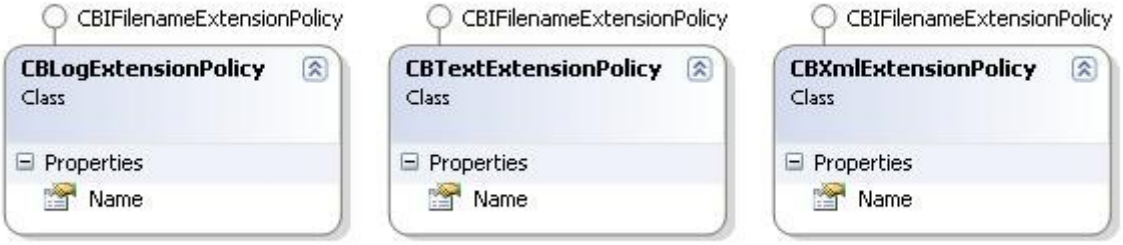
All of FileNamePolicy classes will implement these interfaces including CBIFileNamePolicy, CBIFilenameExtensionPolicy and CBIFileAnmeControl.



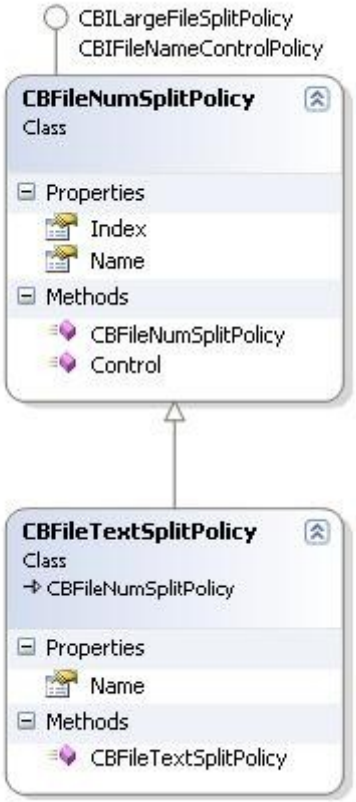
These are policy host classes. The base class is CBFileName implemented.



These classes implement the CBIFileNamePolicy interface.



These classes implement CBIFilenameExtensionPolicy interface.



Пример реализации

These classes implement CBIFileNameControlPolicy and CBILargeFileSplitPolicy interface.

```
/// <summary>
/// Name property interface policy
/// </summary>
public interface CBINamePolicy
{
    /// <summary>
    /// Name Property
    /// </summary>
    string Name
    {
        get;
        set;
    }
}


/// <summary>
/// The base policy interface for file name
/// </summary>
public interface CBIFileNamePolicy : CBINamePolicy
{
}


/// <summary>
/// Basic implement for CBIFileNamePolicy
/// </summary>
public class CBFileNamePolicy : CBIFileNamePolicy
{
    /// <summary>
```

```
    /// File Name
    /// </summary>
    public string Name
    {
        get;
        set;
    }
}

/// <summary>
/// Get full path file name.
/// EX: C:/Test/MyFileName
/// </summary>
public class CBFullFileNamePolicy : CBIFileNamePolicy
{
    /// <summary>
    /// File Name
    /// </summary>
    string m_name;

    /// <summary>
    /// Setter: set value to name
    /// Getter: get full path with name
    /// </summary>
    public string Name
    {
        get
        {
            return CBGeneral.GetFullPath(m_name);
        }
        set
        {
            m_name = value;
        }
    }
}

/// <summary>
/// Readonly policy, to get yyyyMMdd file name
/// EX: 20110923
/// </summary>
public class CBDateNowNamePolicy : CBIFileNamePolicy
{

    #region CBINamePolicy Members

    public string Name
    {
        get
        {
            return DateTime.Now.ToString("yyyyMMdd");
        }
        set
        {
            throw new NotSupportedException
        ("CBDateNowNamePolicy.Name is a readonly property.");
        }
    }

    #endregion
}

/// <summary>
/// File name with date yyyyMMdd
/// EX: MyFileName20110923
```

```
/// </summary>
public class CBPreDateNowNamePolicy : CBIFileNamePolicy
{
    string m_name;

    #region CBINamePolicy Members

    public string Name
    {
        get
        {
            return m_name + DateTime.Now.ToString("yyyyMMdd");
        }
        set
        {
            m_name = value;
        }
    }

    #endregion
}

/// <summary>
/// Full path with date yyyyMMdd
/// EX: C:/Test/20110923
/// </summary>
public class CBFullDateNowNamePolicy : CBIFileNamePolicy
{

    #region CBINamePolicy Members

    public string Name
    {
        get
        {
            return CBGeneral.GetFullPath(DateTime.Now.ToString("yyyyMMdd"));
        }
        set
        {
            throw new NotSupportedException
        ("CBFullPathDateNowNamePolicy.Name is a readonly property.");
        }
    }

    #endregion
}

/// <summary>
/// Full file name with date yyyyMMdd
/// EX: C:/Test/MyFileName20110923
/// </summary>
public class CBFullPreDateNowNamePolicy : CBIFileNamePolicy
{
    string m_name;

    #region CBINamePolicy Members

    public string Name
    {
        get
        {
            return CBGeneral.GetFullPath(m_name + DateTime.Now.ToString("yyyyMMdd"));
        }
        set
        {
```

```
            m_name = value;
        }
    }

    #endregion
}


//
// In these classes, the Name property will get a string for file name.
//
/// <summary>
/// The base policy interface for file extension name.
/// </summary>
public interface CBIFilenameExtensionPolicy : CBIFileNamePolicy
{
}

/// <summary>
/// txt extension policy
/// </summary>
public class CBTextExtensionPolicy : CBIFilenameExtensionPolicy
{

    #region CBIFilenameExtensionPolicy Members

    public string Name
    {
        get
        {
            return "txt";
        }
        set
        {
            throw new NotImplementedException();
        }
    }

    #endregion
}

/// <summary>
/// xml extension policy
/// </summary>
public class CBXmlExtensionPolicy : CBIFilenameExtensionPolicy
{

    #region CBIFilenameExtensionPolicy Members

    public string Name
    {
        get
        {
            return "xml";
        }
        set
        {
            throw new NotImplementedException();
        }
    }

    #endregion
}

/// <summary>
```

```
/// log extension policy
/// </summary>
public class CBLogExtensionPolicy : CBIFilenameExtensionPolicy
{

    #region CBIFilenameExtensionPolicy Members

    public string Name
    {
        get
        {
            return "log";
        }
        set
        {
            throw new NotImplementedException();
        }
    }

    #endregion
}

/// <summary>
/// int extension policy
/// </summary>
public class CBIniExtensionPolicy : CBIFilenameExtensionPolicy
{

    #region CBIFilenameExtensionPolicy Members

    public string Name
    {
        get
        {
            return "ini";
        }
        set
        {
            throw new NotImplementedException();
        }
    }

    #endregion
}

//
// These are readonly classes, just return extension name.
//
/// <summary>
/// The base policy interface for file name control.
/// </summary>
public interface CBIFileNameControlPolicy : CBIFileNamePolicy
{
    void Control();
}

/// <summary>
/// File split size
/// </summary>
public interface CBIFileSplitSizePolicy
{
    long MaxSplitSize
    {
        get;
        set;
```

```
    }
}

/// <summary>
/// Use for large file split
/// </summary>
public interface CBILargeFileSplitPolicy
{
    int Index
    {
        get;
        set;
    }
}

/// <summary>
/// File split policy
/// If file exist and file size large more than setting it will split file.
/// EX: 01, 02 or 03
/// </summary>
public class CBFileNumSplitPolicy : CBILargeFileSplitPolicy, CBIFileNameControlPolicy
{
    public CBFileNumSplitPolicy()
    {
        Index = 1;
    }
    public int Index
    {
        get;
        set;
    }

    public virtual string Name
    {
        get
        {
            return Index.ToString("00");
        }
        set
        {
            throw new NotImplementedException();
        }
    }

    public void Control()
    {
        Index++;
    }

}

/// <summary>
/// File split policy
/// If file exist and file size large more than setting it will split file.
/// EX: AA, AB or AC
/// </summary>
public class CBFileTextSplitPolicy : CBFileNumSplitPolicy
{
    public CBFileTextSplitPolicy()
        : base()
    {
        Index = 0;
    }

    public override string Name
```

```
    {
        get
        {
            int ch1 = (Index / 26) + 0x41;
            int ch2 = (Index % 26) + 0x41;
            return string.Format("{0}{1}", Convert.ToChar(ch1), Convert.ToChar(ch2));
        }
        set
        {
            throw new NotImplementedException();
        }
    }
}

//
// These are file names split with split size property.
// Sometimes the log file will split with file size, so I use this policy to control file name.
//

/// <summary>
/// Abstract base class for file name policy host.
/// How to use:
/// CBFileName filename = new CBFileName<CBIFileNamePolicy>();
/// CBFileName filename = new CBFileName<CBIFileNamePolicy,CBIFilenameExtensionPolicy>();
/// CBFileName filename = new CBFileName<CBIFileNamePolicy,
/// CBIFileNameControlPolicy,CBIFilenameExtensionPolicy>();
/// Just call fliename.Name to get file name
/// filename.MaxSplitSize will use for CBIFileNameControlPolicy
/// </summary>
public abstract class CBFileName : CBIFileNamePolicy, CBIFileSplitSizePolicy
{
    const long DEFAULT_SPLIT_SIZE = 1024 * 1024 * 5;
    //const long DEFAULT_SPLIT_SIZE = 100;
    public virtual string Name
    {
        get;
        set;
    }
    public long MaxSplitSize
    {
        get;
        set;
    }

    public CBFileName()
    {
        MaxSplitSize = DEFAULT_SPLIT_SIZE;
    }
}

//
// The CBFileName is an abstract class and just has two properties:
//

/// <summary>
/// This policy will get name from CBIFileNamePolicy
/// </summary>
/// <typeparam name="TNamePolicy">Must be CBIFileNamePolicy</typeparam>
public sealed class CBFileName<TNamePolicy> : CBFileName
    where TNamePolicy : CBIFileNamePolicy, new()
{
    TNamePolicy _policy;
    public CBFileName()
        : base()
    {
```

```
        _policy = new TNamePolicy();
    }
    public override string Name
    {
        get
        {
            return _policy.Name;
        }
        set
        {
            _policy.Name = value;
        }
    }
}

//
// This is first policy host class.
// It accesses _policy.Name only.
//

/// <summary>
/// This policy will get name from CBIFileNamePolicy.Name + "." +
/// CBIFilenameExtensionPolicy.Name
/// </summary>
/// <typeparam name="TNamePolicy">Must be CBIFileNamePolicy</typeparam>
/// <typeparam name="TNameExtensionPolicy">Must be CBIFilenameExtensionPolicy</typeparam>
public sealed class CBFileName<TNamePolicy, TNameExtensionPolicy> : CBFileName
    where TNamePolicy : CBIFileNamePolicy, new()
    where TNameExtensionPolicy : CBIFilenameExtensionPolicy, new()
{
    TNamePolicy _policy1;
    TNameExtensionPolicy _policy2;
    public CBFileName()
        : base()
    {
        _policy1 = new TNamePolicy();
        _policy2 = new TNameExtensionPolicy();
    }
    public override string Name
    {
        get
        {
            return _policy1.Name + "." + _policy2.Name;
        }
        set
        {
            _policy1.Name = value;
        }
    }
}
```

This policy host class has two generic arguments.

In this role, CBFileName.Name will return _policy1.Name + "." _policy2.Name.

The _policy1 is CBIFileNamePolicy, _policy2 is CBIFilenameExtensionPolicy.

```
CBFileName name1 =
    new CBFileName<CBFileNamePolicy, CBTextExtensionPolicy>();
name1.Name = "Test1";
```

```
Console.WriteLine(name1.Name);
```

We can use it like this sample.

The Name property will get Test1.txt.

```
/// <summary>
/// This policy will get name from
/// CBIFileNamePolicy.Name + "_" + CBIFileNameControlPolicy.Name +
/// "." + CBIFilenameExtensionPolicy.Name
/// </summary>
/// <typeparam name="TNamePolicy">Must be CBIFileNamePolicy</typeparam>
/// <typeparam name="TNameControlPolicy">Must be CBIFileNameControlPolicy</typeparam>
/// <typeparam name="TNameExtensionPolicy">
/// Must be CBIFilenameExtensionPolicy</typeparam>
public sealed class CBFileName<TNamePolicy, TNameControlPolicy,
    TNameExtensionPolicy> : CBFileName, CBIFileNameControlPolicy
    where TNamePolicy : CBIFileNamePolicy, new()
    where TNameControlPolicy : CBIFileNameControlPolicy, new()
    where TNameExtensionPolicy : CBIFilenameExtensionPolicy, new()
{
    TNamePolicy _policy1;
    TNameControlPolicy _policy2;
    TNameExtensionPolicy _policy3;

    public CBFileName()
        : base()
    {
        _policy1 = new TNamePolicy();
        _policy2 = new TNameControlPolicy();
        _policy3 = new TNameExtensionPolicy();
    }

    string FileName
    {
        get
        {
            return _policy1.Name + "_" + _policy2.Name + "." + _policy3.Name;
        }
    }

    public override string Name
    {
        get
        {
            Control();
            return FileName;
        }
        set
        {
            _policy1.Name = value;
        }
    }

    public void Control()
    {
        while (true)
        {
            FileInfo info = new FileInfo(FileName);
            if (info.Exists && info.Length > MaxSplitSize)
            {
                _policy2.Control();
            }
            else
```

```
            {
                break;
            }
        }
    }
}
```

This policy host class has three generic arguments.

In this role, CBFileName.Name will return_policy1.Name + "_" _policy2.Name + "." _policy3.Name.
 The _policy1 is CBIFileNamePolicy, _policy2 is CBIFileNameControlPolicy, _policy3 is CBIFilenameExtensionPolicy.

The implemented control will get file name with MaxSplitSize.

```
CBFileName nameNormal =
   new CBFileName<CBFullPreDateNowNamePolicy,
       CBFileNumSplitPolicy, CBLogExtensionPolicy>();
nameNormal.Name = "Log\\Event_Normal_";
Console.WriteLine(nameNormal.Name);
```

Single-serving visitor

In computer programming, the **single-serving visitor** pattern is a design pattern. Its intent is to optimise the implementation of a visitor that is allocated, used only once, and then deleted (which is the case of most visitors).

Применение

The single-serving visitor pattern should be used when visitors do not need to remain in memory. This is often the case when visiting a hierarchy of objects (such as when the visitor pattern is used together with the composite pattern) to perform a single task on it, for example counting the number of cameras in a 3D scene.

The regular visitor pattern should be used when the visitor must remain in memory. This occurs when the visitor is configured with a number of parameters that must be kept in memory for a later use of the visitor (for example, for storing the rendering options of a 3D scene renderer).

However, if there should be only one instance of such a visitor in a whole program, it can be a good idea to implement it both as a single-serving visitor and as a singleton. In doing so, it is ensured that the single-serving visitor can be called later with its parameters unchanged (in this particular case "single-serving visitor" is an abuse of language since the visitor can be used several times).

Пример использования

```
The single-serving visitor is called through the intermediate of static methods.
Without parameters:
 Element* elem;
 SingleServingVisitor::apply_to(elem);
With parameters:
 Element* elem;
 TYPE param1, param2;
 SingleServingVisitor::apply_to(elem, param1, param2);
Implementation as a singleton:
 Element* elem;
 TYPE param1, param2;
 SingleServingVisitor::set_param1(param1);
 SingleServingVisitor::set_param2(param2);
 SingleServingVisitor::apply_to(elem);
```

Плюси

No "zombie" objects. With a single-serving visitor, it is ensured that visitors are allocated when needed and destroyed once useless.

A simpler interface than visitor. The visitor is created, used and free by the sole call of the apply_to static method.

Минусы

Repeated allocation. At each call of the apply_to method, a single-serving visitor is created then discarded, which is time-consuming. In contrast, the singleton only performs one allocation.

Пример реализации

Implementation as a singleton

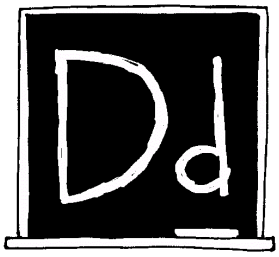
This implementation ensures:

- that there is at most one instance of the single-serving visitor

- that the visitor can be accessed later

```
// Предназначение
public abstract class SingleServingVisitor
{
    protected static SingleServingVisitor instance_ = null;
    protected TYPE param1_ = new TYPE();
    protected TYPE param2_ = new TYPE();
    // Реализация

    protected static SingleServingVisitor get_instance()
    {
        if (this.instance_ == null)
            this.instance_ = new SingleServingVisitor();
        return this.instance_;
    }

    // Примечание: метод get_instance не должен быть публичным
    SingleServingVisitor();
    public static void apply_to(Element elem)
    {
        elem.accept(get_instance());
    }

    // Статические методы для доступа к параметрам
    public static void set_param1(TYPE param1)
    {
        getInstance().param1_ = param1;
    }

    public static void set_param2(TYPE param2)
    {
        getInstance().param2_ = param2;
    }

    public abstract void visit_ElementA(ElementA NamelessParameter);

    public abstract void visit_ElementB(ElementB NamelessParameter);
}
```

Об авторе



Блог: http://go-d.org

Мейл: mail.go.d.org@gmail.com

Твиттер: https://twitter.com/____Dd

Design Patterns

with examples in C#