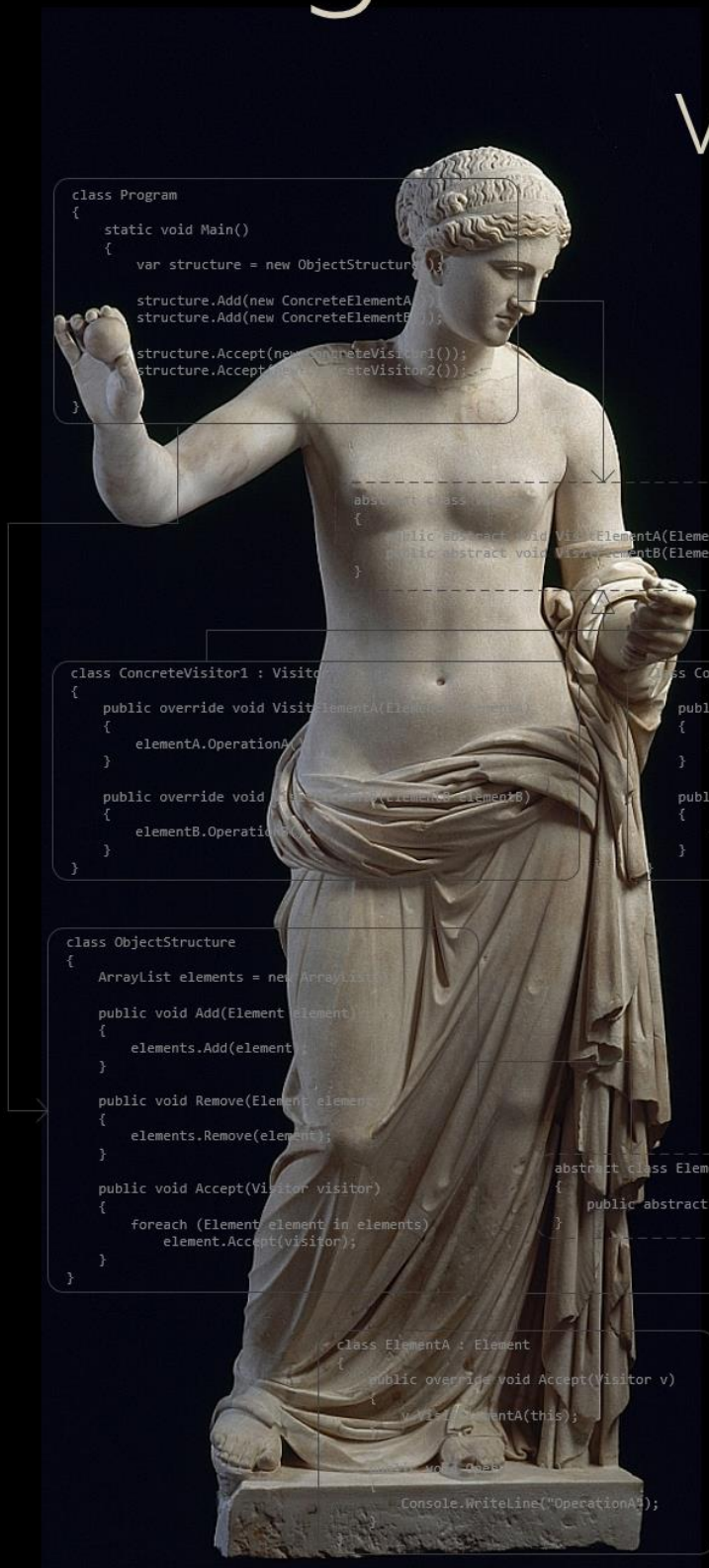
Александр Шевчук, Дмитрий Охрименко, Андрей Касьянов

Design Patterns via C#

Приемы объектно-ориентированного проектирования



```
class Program
{
    static void Main()
    {
        var structure = new ObjectStructure();

        structure.Add(new ConcreteElementA());
        structure.Add(new ConcreteElementB());

        structure.Accept(new ConcreteVisitor1());
        structure.Accept(new ConcreteVisitor2());
    }
}
```

```
abstract class Visitor
{
    public abstract void VisitElementA(ElementA elementA);
    public abstract void VisitElementB(ElementB elementB);
}
```

```
class ConcreteVisitor1 : Visitor
{
    public override void VisitElementA(ElementA elementA)
    {
        elementA.OperationA();
    }

    public override void VisitElementB(ElementB elementB)
    {
        elementB.OperationB();
    }
}
```

```
class ConcreteVisitor2 : Visitor
{
    public override void VisitElementA(ElementA elementA)
    {
        elementA.OperationA();
    }

    public override void VisitElementB(ElementB elementB)
    {
        elementB.OperationB();
    }
}
```

```
class ObjectStructure
{
    ArrayList elements = new ArrayList();

    public void Add(Element element)
    {
        elements.Add(element);
    }

    public void Remove(Element element)
    {
        elements.Remove(element);
    }

    public void Accept(Visitor visitor)
    {
        foreach (Element element in elements)
            element.Accept(visitor);
    }
}
```

```
abstract class Element
{
    public abstract void Accept(Visitor v);
}
```

```
class ElementA : Element
{
    public override void Accept(Visitor v)
    {
        v.VisitElementA(this);
    }

    public void OperationA()
    {
        Console.WriteLine("OperationA");
    }
}
```

```
class ElementB : Element
{
    public override void Accept(Visitor v)
    {
        v.VisitElementB(this);
    }

    public void OperationB()
    {
        Console.WriteLine("OperationB");
    }
}
```

ITVDN

Александр Шевчук, Дмитрий Охрименко, Андрей Касьянов

Design Patterns via C#

Приемы объектно-ориентированного проектирования

2015

Содержание

Предисловие

От авторов

Книга «Design Patterns via C#» не является самостоятельным изданием, описывающим паттерны проектирования, на эту тему уже есть уникальное издание: «Приемы объектно-ориентированного проектирования. Паттерны проектирования», авторами которого являются Эрих Гамма, Ричард Хелм, Ральф Джонсон и Джон Влиссидес. Эта группа авторов известна под творческим псевдонимом - «Банда четырех» (GoF – Gang of Four). Случайным и интересным образом этот псевдоним совпадает с названием левацкой политической фракции Коммунистической Партии Китая, находившейся у власти во времена «Культурной революции» (1966 – 1976 годы). Такое название партии было дано Мао Цзэдуном.

За время своего существования, книга «Приемы объектно-ориентированного проектирования. Паттерны проектирования» зарекомендовала себя как библия объектно-ориентированного проектирования. Каждый представленный в ней паттерн – это отдельный завет, которого необходимо придерживаться и ему следовать. Этой книге не может быть замены.

Книга «Приемы объектно-ориентированного проектирования. Паттерны проектирования» - это научный труд, который заложил основы и сформировал стандарты объектно-ориентированного проектирования, которым все стараются следовать.

Для успешной реализации программных решений, одного стандарта проектирования может оказаться мало. Не менее важным является выбор программной инфраструктуры. Компания Microsoft предоставила миру в свободное использование инфраструктуру .NET.

Что можно сказать об инфраструктуре Microsoft .Net? Её формула проста: .NET = CLR + FCL. И сразу же хочется перефразировать крылатое высказывание неизвестного древнегреческого философа-геометра: *«Высшее проявление духа – это разум. Высшее проявление разума – это .NET. Клетки .NET – FCL. Она так же неисчерпаема, как и Вселенная. CLR – душа .NET. Познайте CLR, и вы не только познаете душу .NET, но и возвысите душу свою»*.

Многие паттерны были использованы для реализации типов, входящих в FCL, а некоторые из них даже нашли выражение в языковых конструкциях и в идеях работы механизмов CLR.

В своей книге «CLR via C#», Джеффри Рихтер, богато и технически ярко описал устройство и особенности работы CLR. Эта книга является глубоким и всеобъемлющим источником знаний по устройству CLR и использованию языка C#, и равных ей в этом нет.

Реализации паттернов с использованием языка C# имеют свои иногда специфические особенности, поэтому рекомендуется обращаться за разъяснениями использования конструкций языка, описания типов FCL и самих механизмов CLR именно к книге Джеффри Рихтера - «CLR via C#».

Цели, которые перед собой ставили авторы книги «Design Patterns via C#» при ее написании:

Разъяснить и в хорошем смысле более «просторечиво» представить определения и положения, представленные в книге «Приемы объектно-ориентированного проектирования. Паттерны проектирования».

Реализовать примеры на языке C# из книги «Приемы объектно-ориентированного проектирования. Паттерны проектирования», которые в книге представлены в форме общих описаний (примеров-идей) или в виде отрывков-кода на языке C++, стараясь при этом максимально сохранить первоначально заложенный смысл-идею.

Представить модели диаграммами с использованием языка UML и выразить их средствами моделирования Microsoft Visual Studio.

Показать варианты реализации паттернов с использованием особенностей конструкций языка C#, типов FCL и механизмов CLR.

Книгу «Design Patterns via C#» рекомендуется воспринимать как приложение к книге «Приемы объектно-ориентированного проектирования. Паттерны проектирования» и, читать параллельно, обращаясь к ней за разъяснениями и описанием реализации примеров на языке C#.

Александр Шевчук
Дмитрий Охрименко
Андрей Касьянов

Об авторах

Александр Шевчук

Тренер-консультант в области построения архитектуры информационных систем, бизнес-анализа и управления IT проектами. Сертифицированный специалист Microsoft (MCTS, MCPD, MCT).

CyberBionic systematics

Дмитрий Охрименко

Тренер-консультант в области построения архитектуры распределенных и веб-ориентированных приложений. Сертифицированный специалист Microsoft (MCTS, MCPD, MCT).

CyberBionic systematics

Андрей Касьянов

Инженер – программист.

https://www.linkedin.com/pub/kasyanov-andrew/78/299/549/en

Благодарности

В создании книги так или иначе участвовали многие люди. Особую благодарность авторы хотели бы выразить всем, кто принимал прямое или косвенное участие в создании этой книги и подготовке примеров к ней.

Назар Рудь, Андрей Рябец, Олег Кулыгин, Давид Бояров, Алексей Назарук, Владислав Поколенко, Андрей Василишин, Илья Попов, Владимир Божек, Юлия Лищук, Тимур Сайфулин, Александр Головатый, Николай Лукьяненко, Виктор Левитский, Дмитрий Живняк, Андрей Павлюк, Алексей Анпилогов, Алексей Яблоков, Богдан Донченко, Александр Самойленко, Владислав Овинников, Полина Смирнова, Дмитрий Черниговский.

Особой благодарности заслуживают студенты – слушатели наших курсов, чьи вопросы, замечания и обсуждения примеров неизменно помогают улучшить учебный материал.

Принятые в книге обозначения

Для облегчения работы с книгой приняты следующие соглашения:

Коды программ обозначены в книге шрифтом (Consolas).

Основные стереотипы языка C# представлены следующими цветами:

```
class MyClass { }
struct MyStruct { }
enum MyEnum { }
interface IInterface { }
delegate void MyDelegate();
```

Смысловые акценты, определения, термины, встретившиеся впервые, и подписи к рисункам, обозначены курсивом, например, *Рисунок 2. Фрагмент танца составленный из двух независимых элементов (шаблонов)*, *closure*, *контейнер* и т.д.

Имена основных участников паттернов выделены полужирным начертанием: **Абстракция**, **Посредник**, **Коллега**, **Конкретный продукт** и т.д.

Имена классов и методов, встречающиеся в тексте книги вне листингов, выделены цветом и шрифтом, как элементы кода программ. Например, имена классов выглядят следующим образом: `Builder`, `MazeGame`, `ConcreteProduct` и т.п. Имена методов: `MemberwiseClone`, `Instance`, `CreateManipulator` и т.п. Имена интерфейсов: `IGraphic`, `IEnumerable`, `IEnumerator`.

Комментарии к листингам приведены в привычном для Visual Studio v12 стиле: шрифт Consolas, текст зеленого цвета (`//комментарий листинга`).

Ссылки на статьи MSDN и других вспомогательных ресурсов приведены шрифтом Calibri, синего цвета с использованием подчеркивания, например, http://MSDN.com.

Ключевые преимущества, недостатки и варианты реализаций паттернов выделены маркерами с использованием полужирного начертания:

- **Вариант реализации, ключевое преимущество или недостаток паттерна**

Технические рекомендации

Для просмотра диаграмм классов, представленных с использованием языка UML, требуется воспользоваться Visual Studio v12 и выше в редакции Ultimate Edition.

Дополнительные ресурсы

Более детально познакомиться с особенностями языка C# и технологий .NET можно на следующих ресурсах:

http://MSDN.com

http://ITVDN.com

Глава 1. Введение

1.1. Понятие паттерна проектирования

Определение

Технически, паттерны (шаблоны) проектирования - это всего лишь абстрактные примеры правильного использования небольшого числа комбинаций простейших техник ООП. Паттерны проектирования - это простые примеры показывающие правильные способы организации взаимодействий между классами или объектами.

Паттерны (шаблоны) проектирования – это 23 примера, которые описывают:

- правильные способы формирования внутреннего состояния (полей) и поведения (методов) объекта или класса;
- правильные способы создания объекта (через вызов конструктора или другим способом);
- правильные способы объединения объектов в группы;
- правильные способы организации информационных потоков (вызовов методов и очередности вызовов) позволяющих наладить гармоничное взаимодействие между объектами и группами этих объектов в объектно-ориентированных системах.

Паттерны проектирования помогают представить объектно-ориентированную систему формально, отображая результаты мышления проектировщика в комбинациях двадцати трех точных понятий и утверждений. Например, каталог паттернов можно было бы представить универсальной алгеброй, а каждый отдельный паттерн – конгруэнцией этой алгебры. Но традиционно принято использовать визуальный формализм представления паттернов с применением диаграмм классов и диаграмм последовательностей языка UML.

Метафора

Для того, чтобы лучше понять, что такое паттерны, предлагается воспользоваться метафорой и провести проекцию элементов объективной реальности на виртуальную, то есть на мир программирования. Проекция модели реального мира на модель программной системы - это та проекция, которая послужила толчком к развитию объектно-ориентированно программирования.

Рассмотрим в качестве метафоры фигурное катание. В фигурном катании можно выделить четыре основных категории базовых элементов: *шаги*, *спирали*, *вращения* и *прыжки*.

- *Шаги:*
  Основной шаг, Шассе, Кроссролл, Подсечка, Беговой шаг, Моухог, Чоктау и др.
- *Спирали:*
  Спираль в арабеске, спираль Шарлотты, Y-спираль, Fan-спираль, Спираль-Керриган и др.
- *Вращения:*
  Вращение стоя назад, Вращение-заклон, Вращение бильман, Вращение в ласточке (либела) и др.
- *Прыжки:*
  Лутц, Тулуп, Флип, Аксель, Риттбергер, Сальхов, Сальто и др.

Базовые элементы фигурного катания, можно назвать шаблонами или паттернами фигурного катания. В итоге полный танец состоит из комбинации базовых элементов в определенной последовательности. И все танцы в фигурном катании отличаются набором и последовательностью исполнения базовых элементов (шаблонов).

У каждого базового элемента имеется своя схема исполнения. Чтобы исполнить определенный элемент танца нужно выполнить ряд простейших действий (Разогнаться, повернуться, поднять ногу, расставить руки и пр.).

Рассмотрим в качестве примера один из базовых элементов – «Axel», который входит в категорию прыжков. На рисунке ниже показана детальная схема исполнения прыжка «Axel» и то как этот элемент разбивается на составляющие.

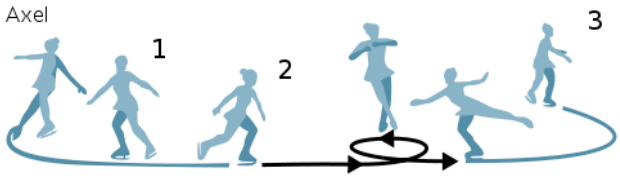


Рисунок 1. Схема исполнения прыжка «Axel»

При проектировании программных систем используется аналогичный подход. Программа строится с учетом наличия готовых подходов, которые представлены своими схемами – шаблонами (паттернами). В общем виде имеется 23 базовых шаблона проектирования программных систем, которые лежат в основе более высокоуровневых и модельно зависимых паттернов (например, паттерны проектирования корпоративных приложений, разработанные Мартином Фаулером, базируются на паттернах из каталога GoF).

Между элементами, показанными на рисунках 2 и 3 можно провести соответствия.

Рисунок 2. Фрагмент танца составленный из двух независимых элементов (шаблонов)

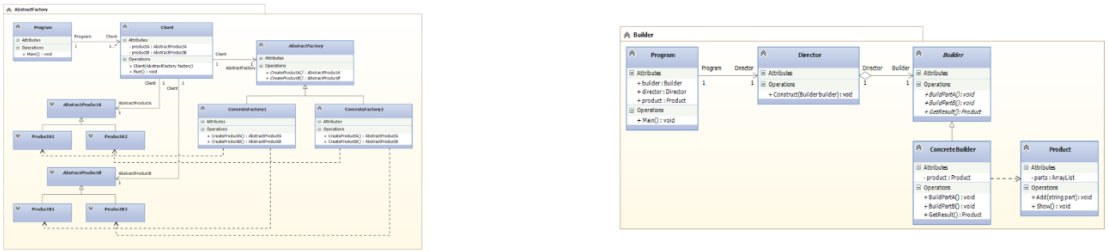


Рисунок 3. Фрагмент программной системы, составленный из двух независимых элементов (шаблонов)

Условно, каждому отдельному движению из элемента (шаблона) входящего в состав танца можно поставить в соответствие отдельный класс из элемента (шаблона) входящего в состав программной системы.

Рисунок 4. Соответствие

Программные системы, построенные с использованием паттернов (шаблонов) удобно сопровождать. Как фигурист понимая, что он плохо исполняет определенный элемент танца, концентрируется на отработке именно этого элемента, при этом не повторяя те элементы, которые он исполняет хорошо, так и проектировщик программных систем, концентрируется на определенном (проблемном) элементе системы (если программная система построена с использованием шаблонов проектирования).

1.2. Формат описания паттернов проектирования

При рассмотрении паттернов проектирования используется единый формат описания. Описание каждого шаблона состоит из следующих разделов:

Название

Название паттерна (на Русском языке) отражающее его назначение.

Также известен как

Альтернативное название паттерна (если такое название имеется).

Классификация

Классификация паттернов производится:

- По цели (порождающий, структурный или поведенческий)
- По применимости (к объектам и/или к классам)

Частота использования

Низкая - **1** 2 3 4 5
Ниже средней - 1 **2** 3 4 5
Средняя - 1 2 **3** 4 5
Выше средней - 1 2 3 **4** 5
Высокая - 1 2 3 4 **5**

Назначение

Краткое описание назначения паттерна и задачи проектирования, решаемые с его использованием.

Введение

Описание паттерна с использованием метафор, позволяющих лучше понять идею, лежащую в основе паттерна, в общем виде охарактеризовать специфические аспекты использования паттерна проводя ассоциации с другими знакомыми процессами, для формирования ясного представления механизма работы паттерна.

Структура паттерна на языке UML

Графическое представление паттерна с использованием диаграмм классов языка UML. На диаграммах показаны основные участники (классы) и связи отношений между участниками.

Структура паттерна на языке C#

Программная реализация паттерна с использованием языка C#.

Участники

Имена участников (классы которые входят в состав паттерна) и описание их назначения.

Отношения между участниками

Описание отношений (взаимодействий) между участниками (классами и/или объектами).

Мотивация

Определение потребности в использовании паттерна. Рассмотрение способов применения паттерна.

Применимость паттерна

Рекомендации по применению паттерна.

Результаты

Особенности и варианты использования паттерна. Результаты применения.

Реализация

Описание вариантов и способов реализации паттерна.

Пример кода

Дополнительные примеры, иллюстрирующие использование паттерна.

Известные применения паттерна в .Net

Использование паттерна в .Net Framework и/или его выражение в языке C#.

1.3. Каталог паттернов проектирования

Каталог состоит из 23 паттернов. Все паттерны разделены на три группы:

Порождающие

1. **Abstract Factory** (Абстрактная Фабрика)
2. **Builder** (Строитель)
3. **Factory Method** (Фабричный Метод)
4. **Prototype** (Прототип)
5. **Singleton** (Одиночка)

Структурные

1. **Adapter** (Адаптер)
2. **Bridge** (Мост)
3. **Composite** (Компоновщик)
4. **Decorator** (Декоратор)
5. **Facade** (Фасад)
6. **Flyweight** (Приспособленец)
7. **Proxy** (Заместитель)

Поведенческие

1. **Chain of Responsibility** (Цепочка Обязанностей)
2. **Command** (Команда)
3. **Interpreter** (Интерпретатор)
4. **Iterator** (Итератор)
5. **Mediator** (Посредник)
6. **Memento** (Хранитель)
7. **Observer** (Наблюдатель)
8. **State** (Состояние)
9. **Strategy** (Стратегия)
10. **Template Method** (Шаблонный Метод)
11. **Visitor** (Посетитель)

1.4. Техники ООП

В основу категоризации каталога паттернов легли три простейшие объектно-ориентированные техники. Это техника использования объектов-фабрик, порождающих объекты-продукты, техника использования объекта-фасада и техника диспетчеризации.

Фабрика - Продукт

Техника использования объекта-фабрики для порождения объектов-продуктов, была положена в основу всех порождающих паттернов. Методы, принадлежащие объекту-фабрике, которые порождают и возвращают объекты-продукты, принято называть фабричными-методами (или виртуальными конструкторами).

```
class Program
{
    static void Main()
    {
        Factory factory = new Factory();

        Product product = factory.FactoryMethod();
    }
}
```

```
class Factory
{
  // Фабричный метод, порождающий продукты.
   public Product FactoryMethod()
   {
      return new Product();
   }
}
```

```
class Product
{
   public Product()
   {
      Console.WriteLine("I'm new Product!");
   }
}
```

См. Пример к главе: \RulesOOP (Creating)

На диаграмме последовательностей можно отследить работу фабричной техники.

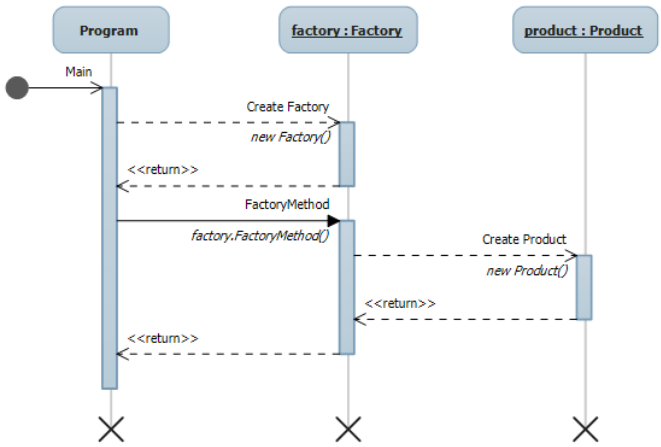


Рисунок 2. Диаграмма последовательностей фабричной техники

Фасад - Подсистема

Техника использования объекта-фасада, скрывающего за собой работу с неким подмножеством объектов, легла в основу структурных паттернов.

```
class Program
{
   static void Main()
   {
      Facade facade= new Facade();
      facade.HelloWorld();
   }
}
```

```
// Большая структура использующая
// в своем составе более мелкие.
class Facade
{
    ElementA elementA;
    ElementB elementB;

    public void HelloWorld()
    {
        elementA = new ElementA();
        elementB = new ElementB();

        elementA.Hello();
        elementB.World();
    }
}
```

```
// Элемент - часть большой структуры.
class ElementA
{
   public void Hello()
   {
      Console.Write("Hello ");
   }
}
```

```
// Элемент - часть большой структуры.
class ElementB
{
   public void World()
   {
      Console.Write("World!");
   }
}
```

См. Пример к главе: \RulesOOP (Structural)

На диаграмме последовательностей можно отследить работу фасадной техники.

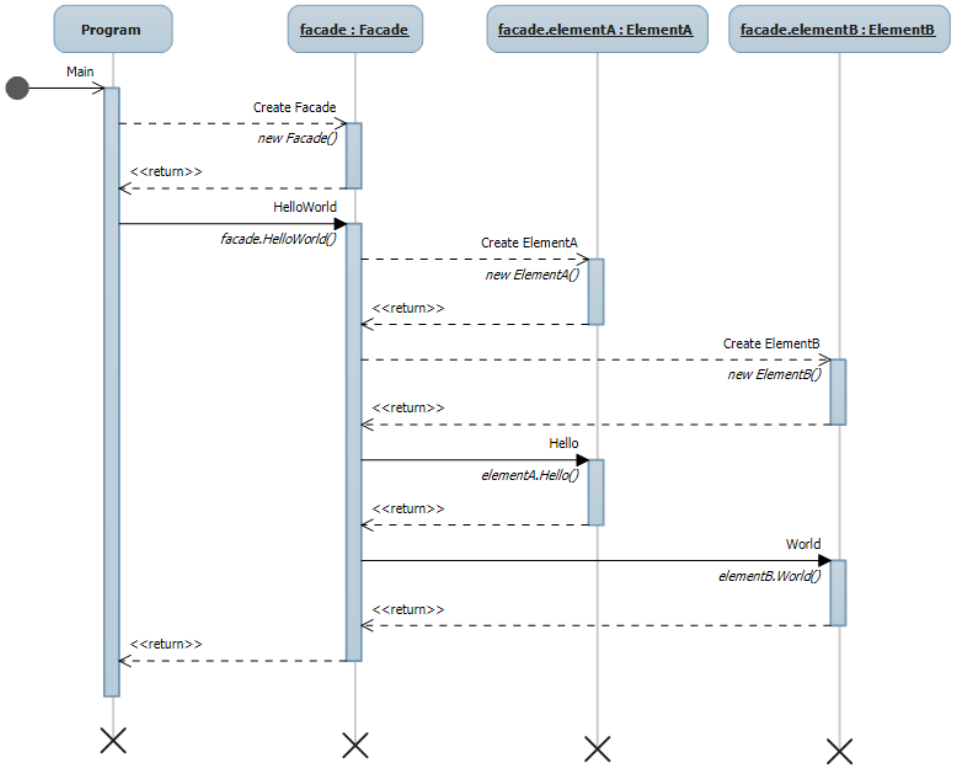


Рисунок 2. Диаграмма последовательностей фасадной техники

Диспетчеризация

Техника использования диспетчеризации имеет две формы: «Цепочка объектов» и «Издатель-Подписчик». Эти техники были положены в основу поведенческих паттернов.

Цепочка объектов

При использовании техники «Цепочка объектов» - объекты связываются в цепочку, вдоль которой происходит серия вызовов методов (посылка сообщений).

```
class Program
{
    static void Main()
    {
        Link3 chain = new Link3();
        chain.Method();
    }
}
```

```
class Link3
{
    public void Method()
    {
        Link2 link = new Link2();
        link.Method();
        Console.WriteLine("Link3");
    }
}
```

```
class Link2
{
    public void Method()
    {
        Link1 link = new Link1();
        link.Method();
        Console.WriteLine("Link2");
    }
}
```

```
class Link1
{
    public void Method()
    {
        Console.WriteLine("Link1");
    }
}
```

См. Пример к главе: \RulesOOP (Chain)

На диаграмме последовательностей можно отследить работу техники «Цепочка объектов».

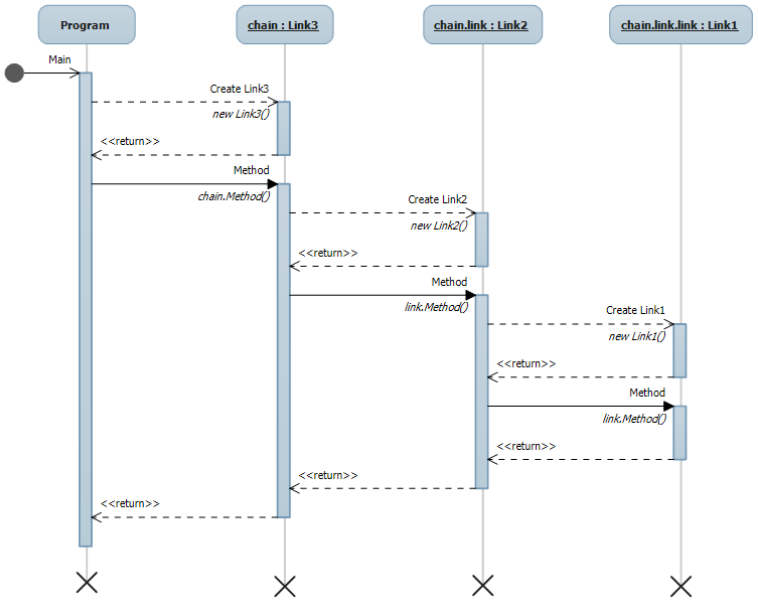


Рисунок 3. Диаграмма последовательностей техники "Цепочка объектов».

Издатель-Подписчик

При использовании техники «Издатель-Подписчик» - объект-издатель вызывает метод на объекте-подписчике, а объект-подписчик после этого вызывает метод на объекте-издателе. Таким образом объект-издатель уведомляет объекта-подписчика о наступлении некоторого события.

```
class Program
{
   static void Main()
   {
      Publisher publisher = new Publisher();
      Subscriber subscriber = new Subscriber();

      // Посылка сообщения подписчику.
      publisher.SendMessageTo(subscriber);
   }
}
```

```
// Издатель.
class Publisher
{
   public void SendMessageTo(Subscriber subscriber)
   {
      subscriber.CallOperationsOn(this);
   }

   public void Operation()
   {
      Console.WriteLine("Выполнена операция.");
   }
}
```

```
// Подписчик.
class Subscriber
{
   public void CallOperationsOn(Publisher publisher)
   {
      publisher.Operation();
   }
}
```

См. Пример к главе: \RulesOOP (Behavioral)

На диаграмме последовательностей можно отследить работу техники «Издатель-Подписчик».

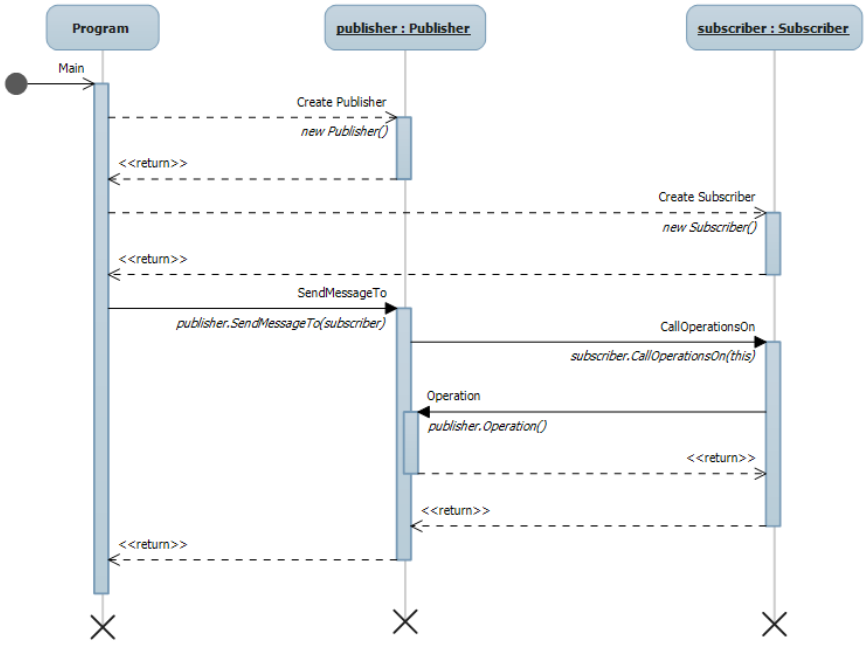


Рисунок 4. Диаграмма последовательностей техники "Издатель-подписчик"

1.5. Принципы организации каталога

Все 23 паттерна классифицируются по двум критериям – цель и применимость (уровень).

Применимость (уровень)	Цель: Порождающие	Цель: Структурные	Цель: Поведенческие
К классам	Фабричный метод	Адаптер (класса)	Интерпретатор Шаблонный метод
К объектам	Абстрактная фабрика Одиночка Прототип Строитель	Адаптер (объекта) Декоратор Заместитель Компоновщик Мост Приспособленец Фасад	Итератор Команда Наблюдатель Посетитель Посредник Состояние Стратегия Хранитель Цепочка обязанностей

Таблица 1. Принципы организации каталога паттернов.

Цель паттерна

Цель паттерна – показывает его назначение.

- Целью порождающих паттернов, является организация процесса создания объектов.
- Целью структурных паттернов, является составление правильно организованных структур из объектов и классов.
- Целью поведенческих паттернов, является организация устойчивого (робастного) взаимодействия между классами или объектами, через правильное формирование информационных потоков.

Уровень паттерна

Уровень паттерна - показывает область применения паттерна: к классам или к объектам.

- Паттерны уровня классов описывают отношения между классами и их подклассами. Такие отношения выражаются при помощи статических связей отношений – наследования и реализации.
- Паттерны уровня объектов описывают взаимодействия между объектами. Такие отношения выражаются при помощи динамических связей отношений – ассоциации, агрегации и композиции.

1.6. Рекомендации по изучению паттернов

Существует две категории программистов, которые решили приступить к изучению паттернов проектирования. Первая категория – начинающие программисты или программисты с небольшим опытом разработки, которые только слышали о таком понятии как паттерны и об их полезности от старших и более опытных коллег. Вторая категория – программисты с опытом разработки, с хорошим пониманием ООП, но по ряду причин не применявшие в своей практике паттерны, при этом осознающие полезность их использования.

Каждая из двух категорий разработчиков делится на три подкатегории: практики (большинство), теоретики (меньшинство) и те, кто гармонично сочетает теорию с практикой. Паттерны не терпят крайностей, для достижения максимального эффекта от обучения рекомендуется знакомство с теорией, изложенной в книге параллельно с рассмотрением прилагаемых примеров.

Начинающим программистам понадобится немного больше времени для изучения и хорошего понимания паттернов. Им рекомендуется сначала приступить к ознакомлению с диаграммой классов и сравнению классов и связей отношений, изображенных на диаграмме с реализацией паттерна в примере на языке C#. Такой подход позволит закрепить понимание простейших техник ООП, которые используются при построении паттерна. Для достижения большего эффекта есть смысл при изучении кода паттерна параллельно рисовать диаграмму объектов (овалы - символизирующие объекты в памяти и стрелки – показывающие как одни объекты ссылаются на другие).

Начать рассмотрение паттерна лучше всего с тела метода `Main`, сперва ознакомившись с интерфейсом взаимодействия используемых объектов. Далее есть смысл переходить к знакомству с классами используемых объектов.

Требуется понять все объектно-ориентированные техники, используемые в коде, мысленно выстроить схему паттерна, а именно запомнить основных участников и связи отношений между ними, а также осознать объектную модель паттерна. И только после этого есть смысл перейти к чтению главы описывающей паттерн и рассмотрению примеров его использования. Таким образом с пониманием абстрактной техники построения паттерна начнет ассоциироваться смысл примеров использования этого паттерна.

Важно понимать, что паттерн - это формула, а пример использования паттерна - это пример применения этой формулы. Формула – первична, ее применение – вторично. Для создателей каталога паттернов формула была вторична. В основу этого каталога была положена докторская диссертация Эриха Гаммы – а это значит сначала исследования в области построения объектно-ориентированных систем, затем формализация результатов исследований и представление их в виде 23 паттернов (формул). Разработчикам исследовать ничего не нужно, им не нужно порождать новых знаний и делать открытий, им просто требуется использовать готовые паттерны (формулы) в повседневной работе для решения проектных задач.

Программистам с опытом разработки ООП, можно предложить читать книгу линейно по ходу чтения рассматривать прилагаемые к книге программные коды или использовать книгу как справочник.

1.7. Рекомендации по применению паттернов

Использовать паттерны просто для тех, кто знает наизусть все 23 паттерна (всех участников и связи отношений между ними). 23 паттерна – это «таблица умножения» проектировщика. Как трудно производить расчеты без знания таблицы умножения, также трудно проектировать приложения без знания паттернов.

Нет надобности искать в каком месте и когда применить тот или иной паттерн. Выбор паттерна – это выбор способа решения задачи. Нет задачи – нет и решения. Поставленная задача – причина. Паттерн - путь к следствию. Решенная задача – следствие.

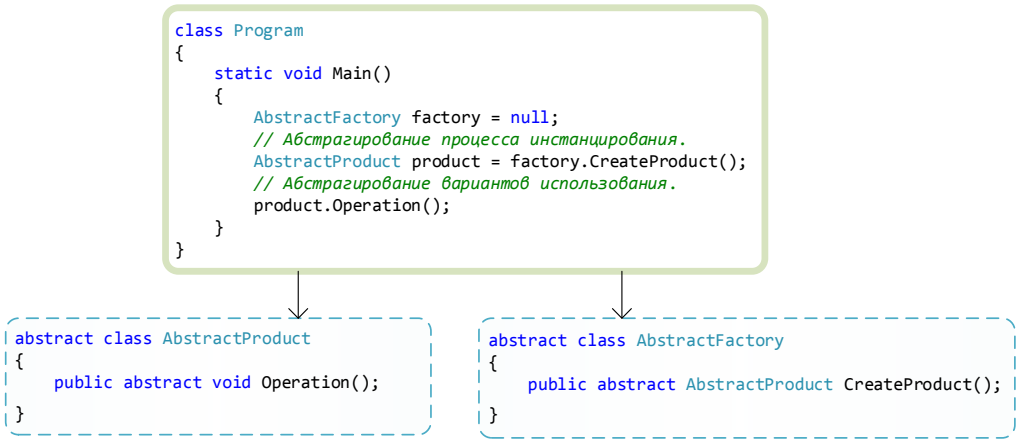
Примеры, приводимые в книге приближены к несложным проектным требованиям и их рассмотрение окажется полезным для понимания использования паттернов. Важно понимать, что пример - это образец чего-либо, как правило – самый яркий и лучший образец (пример для подражания).

Умение увязывать между собой абстрактные примеры и реальные системы или их части, то есть применять знания, полученные во время обучения, к решению проектных задач — есть признак профессионализма. Неумение делать это — основное свойство неопытности. Поиск аналогий — есть перенесение опыта из одной ситуации в другую. Начинающим специалистам не всегда быстро удается перенести опыт из одной проектной ситуации в другую. В этом нет ничего страшного. Потратив определенное количество времени на обучение и рассмотрение примеров, вполне возможно достигнуть желаемых результатов.

Глава 2. Порождающие паттерны

Порождающие паттерны проектирования, не просто описывают процесс создания объектов-продуктов определенных классов, они при этом делают абстрактным сам процесс создания (другими словами абстрагируют процесс инстанцирования). Абстрагирование процесса инстанцирования в ООП, это еще одна простая техника, которая часто используется при порождении объектов-продуктов.

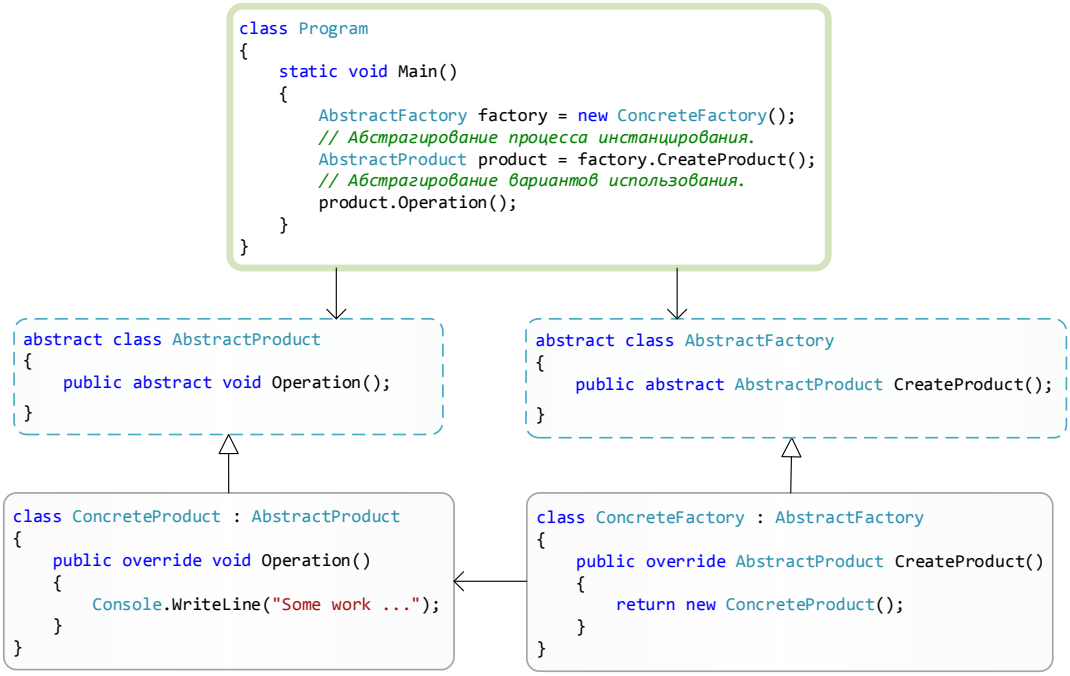
Понятие абстрагирования процесса инстанцирования и есть ничто иное, как абстрактное описание процесса порождения экземпляра типа, через использование абстрактных фабричных методов.



См. Пример к главе: \AbstractInstantiate (AbstractInstantiate)

Сперва создаются абстрактные классы. С их помощью легче выявить и задать нужные типы, выразить типы как собирательные понятия других типов. Описать интерфейсы взаимодействия с каждым типом. Абстрактно (без реализации) описать процессы порождения экземпляров этих типов и варианты использования этих типов, через имеющиеся у них интерфейсы.

После того, как заданы нужные типы, описаны интерфейсы взаимодействия, абстрагированы процессы инстанцирования и варианты использования, можно приступить к выбору структур данных и алгоритмов для их обработки, а также подойти к выбору деталей реализации конкретных классов.



См. Пример к главе: \AbstractInstantiate (AbstractInstantiate2)

Объектно-ориентированная программная система (программа-целое, составленная из соединенных объектов-частей) – состоит из множества объектов, находящихся в определенных отношениях и связях друг с другом. Связанные между собой объекты образуют логическую целостность (единство) системы. Порождающие паттерны проектирования используя технику абстрагирования процесса инстанцирования, помогают построить программную систему так, что эта система не зависит от способа создания в ней объектов, композиции (составления и соединения) объектов и представления (внутреннего состояния) объектов.

Порождающий паттерн уровня классов, использует наследование, чтобы варьировать (видоизменять) инстанцируемый класс, а порождающий паттерн уровня объектов, использует композицию делегируя (передавая ответственность) инстанцирование другому объекту.

Использование порождающих паттернов оказываются полезным, когда в программной системе чаще используется композиция объектов чем наследование классов. При использовании наследования основной акцент делается на жестком кодировании фиксированного набора поведений (методов). В случае использования композиции можно получать любое число новых поведений (методов) составленных из уже существующих поведений.

Порождающие паттерны позволяют скрыть работу с конкретными классами и детали того как эти классы создаются и стыкуются. Единственная известная информация об объектах - это интерфейсы этих объектов, заданные через абстрактные классы. Таким образом упрощается понимание процессов создания объектов: когда, как, кто и что создает.

Игра - Лабиринт

Изучение всех пяти порождающих паттернов, будет сопровождаться примерами построения лабиринта для компьютерной игры. Реализация игры-лабиринта будет изменяться при применении разных паттернов. Для того что бы упростить представление использования паттернов в игре-лабиринте, будут игнорироваться все соображения относительно взаимодействия игры с пользовательским интерфейсом.

В реализации данной игры внутри лабиринта сможет находиться только один игрок. Лабиринт представляет собой множество комнат. Каждая комната лабиринта является ассоциативным объектом класса `Room`, который содержит в себе ссылки на составляющие ее элементы - стены класса `Wall` или двери в другую комнату класса `Door`.

Комната имеет четыре стороны. Для задания северной, южной, восточной и западной сторон используются элементы перечисления `Direction`.

```
enum Direction
{
    North,
    South,
    East,
    West
}
```

Абстрактный класс `MapSite` - является базовым классом для всех классов компонентов лабиринта (`Room`, `Door` и `Wall`). В классе `MapSite` создается абстрактная операция `Enter`.

```
public abstract class MapSite
{
    public abstract void Enter();
}
```

Реализованные в производных классах операции `Enter` позволят игроку перемещаться из комнаты в комнату или оставаться в той же комнате. Например, если игрок находится в комнате и говорит «Иду на восток», то игрой определяется, какой объект типа `MapSite` находится к востоку от игрока, и на этом объекте вызывается операция `Enter`. Если на востоке был объект-дверь (`Door`) то игрок перейдет в другую комнату, а если был объект-стена (`Wall`), то останется в этой же комнате.

Класс `Room` - производный от класса `MapSite`, содержит ссылки на другие объекты типа `MapSite`, а также хранит номер комнаты. Каждая из комнат в лабиринте имеет свой уникальный номер.

```
class Room : MapSite
{
    int roomNumber = 0;
    Dictionary<Direction, MapSite> sides;

    public Room(int roomNo)
    {
        this.roomNumber = roomNo;
        sides = new Dictionary<Direction, MapSite>(4);
    }

    public override void Enter()
    {
        Console.WriteLine("Room");
    }

    public MapSite GetSide(Direction direction)
    {
        return sides[direction];
    }

    public void SetSide(Direction direction, MapSite mapSide)
    {
        this.sides.Add(direction, mapSide);
    }

    public int RoomNumber
    {
        get { return roomNumber; }
        set { roomNumber = value; }
    }
}
```

Классы `Wall` и `Door` описывают стены и двери, из которых состоит комната.

```
class Wall : MapSite
{
    public Wall()
    {
    }

    public override void Enter()
    {
        Console.WriteLine("Wall");
    }
}
```

```
class Door : MapSite
{
    Room room1 = null;
    Room room2 = null;
    bool isOpen;
```

```
    public Door(Room room1, Room room2)
    {
        this.room1 = room1;
        this.room2 = room2;
    }

    public override void Enter()
    {
        Console.WriteLine("Door");
    }


    public Room OtherSideFrom(Room room)
    {
        if (room == room1)
            return room2;
        else
            return room1;
    }
}
```

Класс Maze используется для представления лабиринта, как набора комнат. В классе Maze имеется операция RoomNo(int number) для получения ссылки на экземпляр комнаты по ее номеру.

```
class Maze
{
    Dictionary<int, Room> rooms = null;

    public Maze()
    {
        this.rooms = new Dictionary<int, Room>();
    }

    public void AddRoom(Room room)
    {
        rooms.Add(room.RoomNumber, room);
    }
    public Room RoomNo(int number)
    {
        return rooms[number];
    }
}
```

Класс MazeGame – содержит в себе метод CreateMaze, который создает лабиринт состоящий из двух комнат с одной дверью между ними и возвращает ссылку на экземпляр созданного лабиринта.

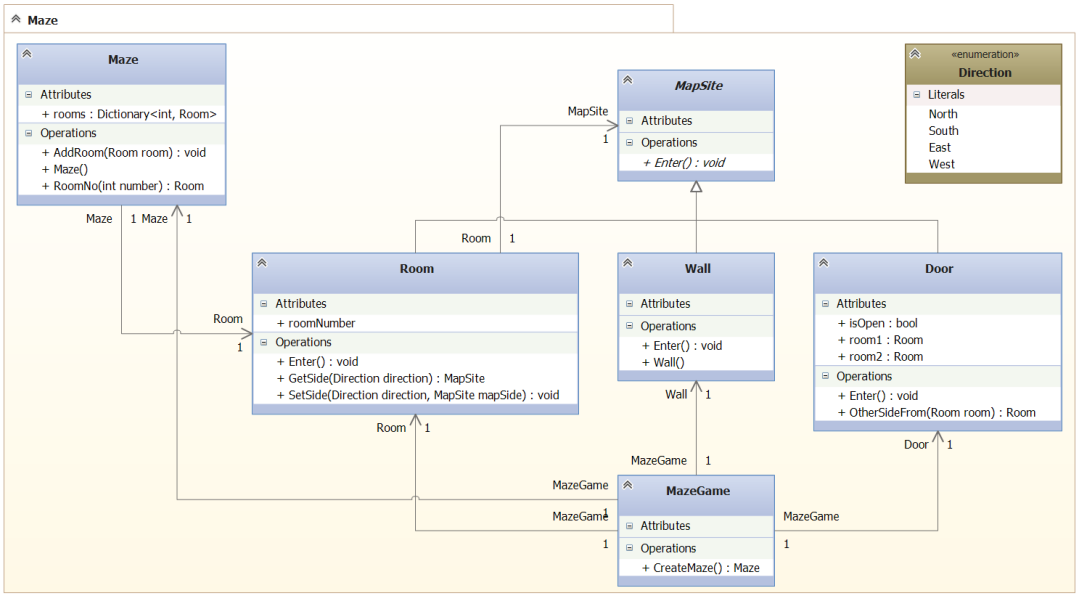
В методе CreateMaze жестко «закодирована» структура лабиринта. Для изменения структуры лабиринта, потребуется внести изменения в тело самого метода или полностью его заместить, что может стать причиной возникновения ошибок и не способствует повторному использованию.

```
class MazeGame
{
    public Maze CreateMaze()
    {
        Maze aMaze = new Maze();
        Room r1 = new Room(1);
```

```
            Room r2 = new Room(2);
            Door theDoor = new Door(r1, r2);

            aMaze.AddRoom(r1);
            aMaze.AddRoom(r2);

            r1.SetSide(Direction.North, new Wall());
            r1.SetSide(Direction.East, theDoor);
            r1.SetSide(Direction.South, new Wall());
            r1.SetSide(Direction.West, new Wall());

            r2.SetSide(Direction.North, new Wall());
            r2.SetSide(Direction.East, new Wall());
            r2.SetSide(Direction.South, new Wall());
            r2.SetSide(Direction.West, theDoor);

            return aMaze;
        }
    }
```

См. Пример к главе: \MAZE\000_Maze

На диаграмме ниже показаны отношения между классами которые использовались при построении игры-лабиринта.



Что можно сказать в общем о дизайне этой программы? Ее можно охарактеризовать как слабо спроектированную и не объектно-ориентированную. Хотя даже в таком виде эта программа работает. Но важно понимать, что в процессе разработки программ участвуют люди, которым будет сложно сопровождать и модифицировать такого вида программу.

Порождающие паттерны, которые будут использоваться в игре-лабиринте помогут сделать дизайн программы более гибким, хотя и необязательно меньшим по размеру. Применение паттернов позволит легко изменять классы компонентов лабиринта.

Паттерн Abstract Factory

Название

Абстрактная фабрика

Также известен как

Kit (Набор инструментов)

Классификация

По цели: порождающий
По применимости: к объектам

Частота использования

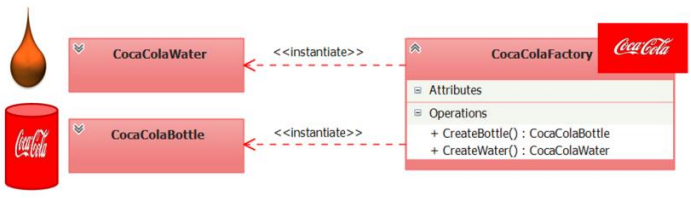
Высокая - 1 2 3 4 **5**

Назначение

Паттерн Abstract Factory - предоставляет клиенту интерфейс (набор методов) для создания семейств взаимосвязанных или взаимозависимых объектов-продуктов, при этом скрывает от клиента информацию о конкретных классах создаваемых объектов-продуктов.

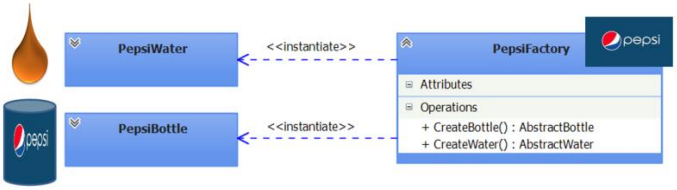
Введение

Что такое фабрика в объективной реальности? Фабрика – это объект имеющий станки (методы), производящие продукты. Например, фабрика компании Coca-Cola производит сладкую газированную воду, разлитую в жестяные банки. Предположим, что в помещении фабрики стоит два станка. Один станок размешивает и газирует сладкую воду, а другой станок формирует жестяные банки. После того как сладкая вода и жестяная банка произведены, требуется воду влить в банку, если сказать другими словами, то требуется организовать взаимодействие между двумя продуктами: водой и банкой. Опишем данный процесс с использованием диаграмм классов языка UML.



На диаграмме видно, что фабрика Coca-Cola порождает два продукта: воду и банку. Эти продукты должны обязательно взаимодействовать друг с другом, иначе воду будет проблематично поставить потребителю, равно как и пустая банка потребителю не нужна. Порождаемые фабрикой Coca-Cola взаимосвязанные продукты (вода и банка) образуют семейство продуктов фабрики Coca-Cola.

Фабрика компании Pepsi также порождает свое собственное семейство взаимодействующих и взаимозависимых продуктов (вода и банка).

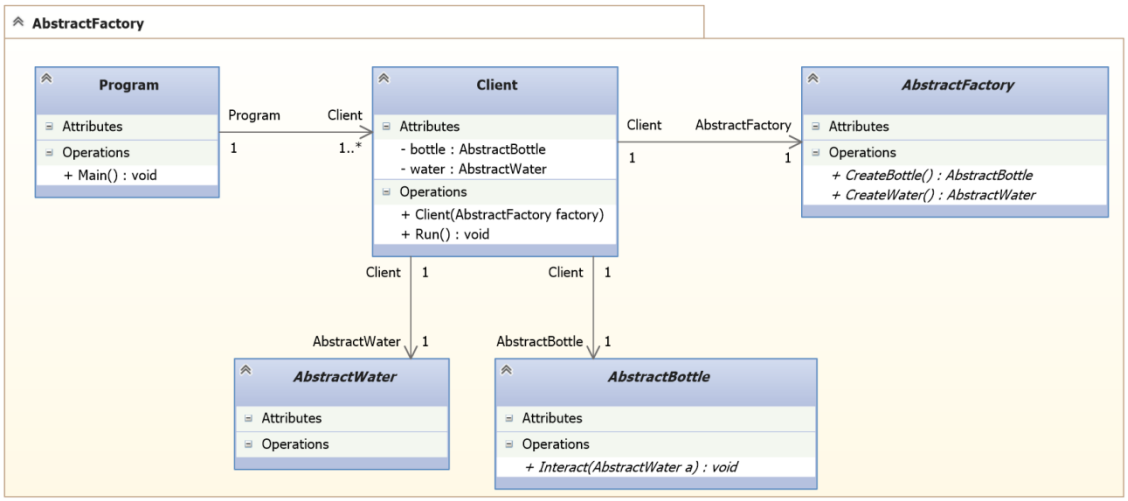


Важно заметить, что не логично пытаться наладить взаимодействие продуктов из разных семейств (например, вливать воду Coca-Cola в банку Pepsi или воду Pepsi в банку Coca-Cola). Скорее всего оба производителя будут против такого взаимодействия. Такой подход представляет собой пример антипатерна.

Представим рассмотренные фабрики и порождаемые ими семейства продуктов в контексте одной программы.

Сперва требуется создать абстрактные классы для задания типов продуктов (`AbstractWater` и `AbstractBottle`) и типа фабрик (`AbstractFactory`). Описать интерфейсы взаимодействия с каждым типом продукта и фабрики.

Далее требуется создать конкретный класс `Client` в котором абстрактно (без реализации) описать процессы порождения экземпляров типов продуктов и варианты использования этих типов продуктов, через имеющиеся у них абстрактные интерфейсы. Также класс `Client` реализует идею инкапсуляции вариаций (сокрытие частей программной системы).




```
class Program
{
    public static void Main()
    {
        Client client = null;
    }
}
```

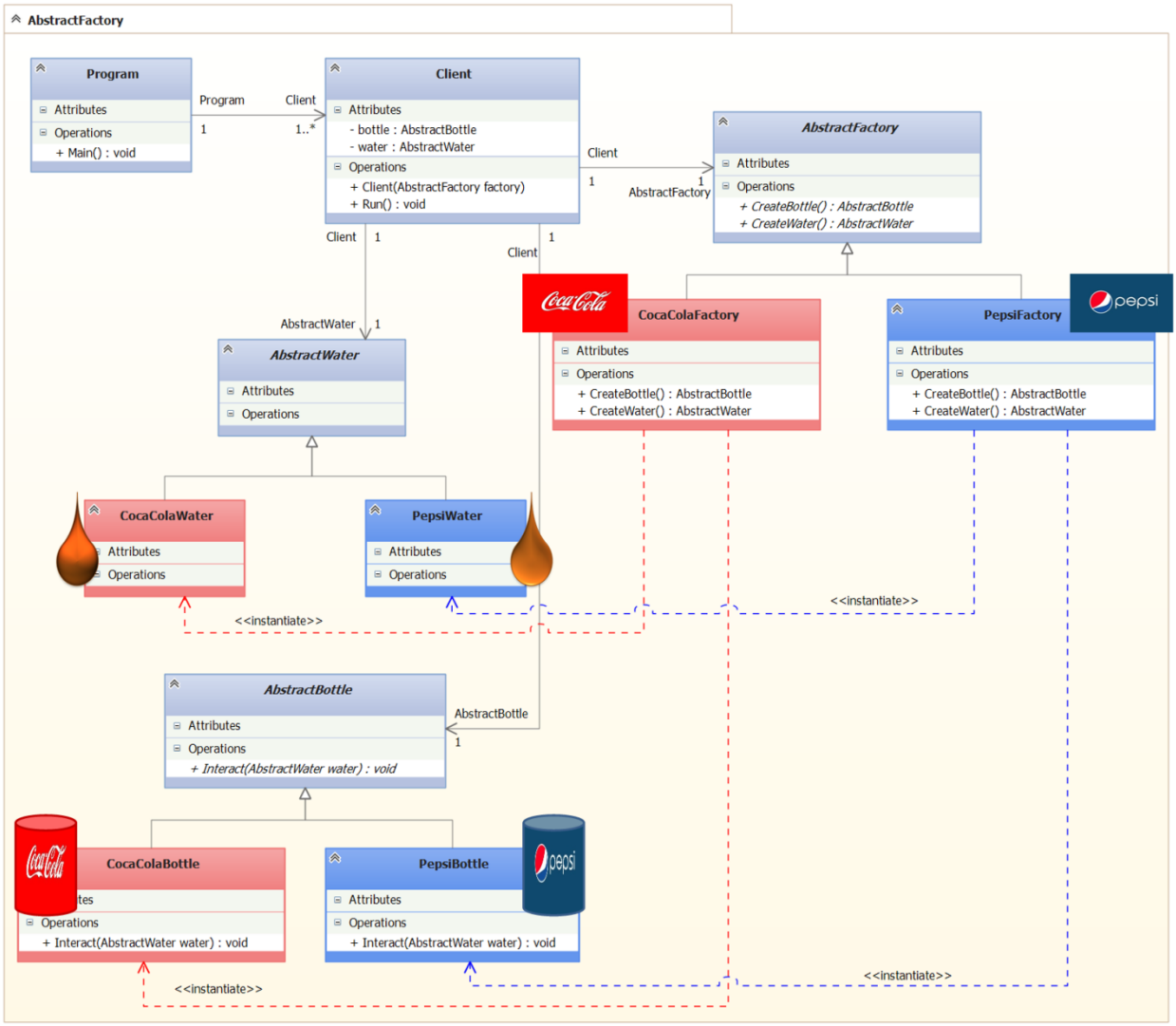
```
class Client
{
    private AbstractWater water;
    private AbstractBottle bottle;

    public Client(AbstractFactory factory)
    {
        // Абстрагирование процессов инстанцирования.
        water = factory.CreateWater();
        bottle = factory.CreateBottle();
    }

    public void Run()
    {
        // Абстрагирование вариантов использования.
        bottle.Interact(water);
    }
}
```

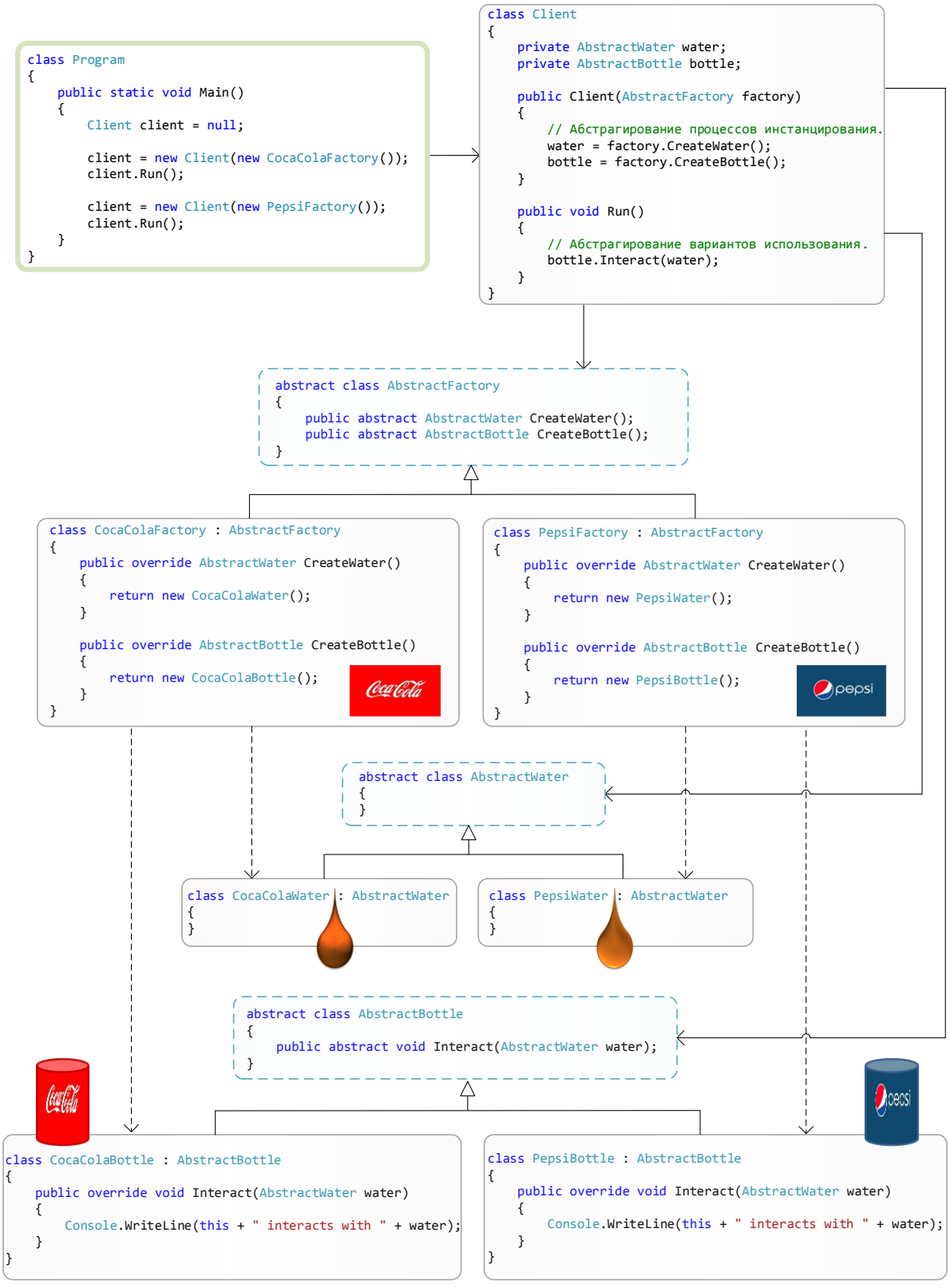
```
abstract class AbstractFactory
{
    public abstract AbstractWater CreateWater();
    public abstract AbstractBottle CreateBottle();
}
```

```
abstract class AbstractWater
{
}
```

```
abstract class AbstractBottle
{
    public abstract void Interact(AbstractWater water);
}
```

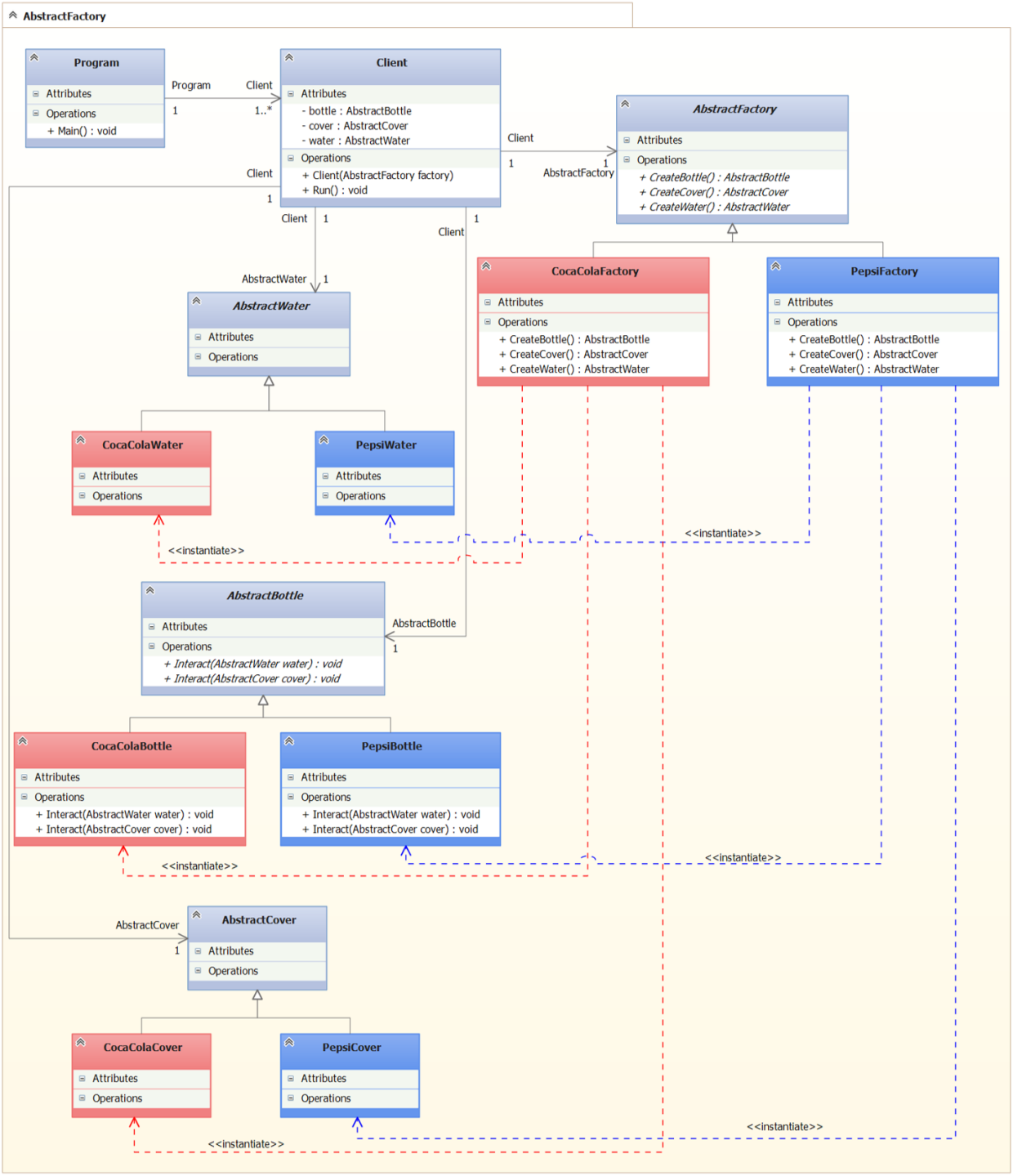
После того, как заданы нужные типы продуктов и фабрик, описаны интерфейсы взаимодействия между продуктами, абстрагированы процессы инстанцирования продуктов и варианты использования продуктов, можно приступить к реализации конкретных классов продуктов и фабрик.



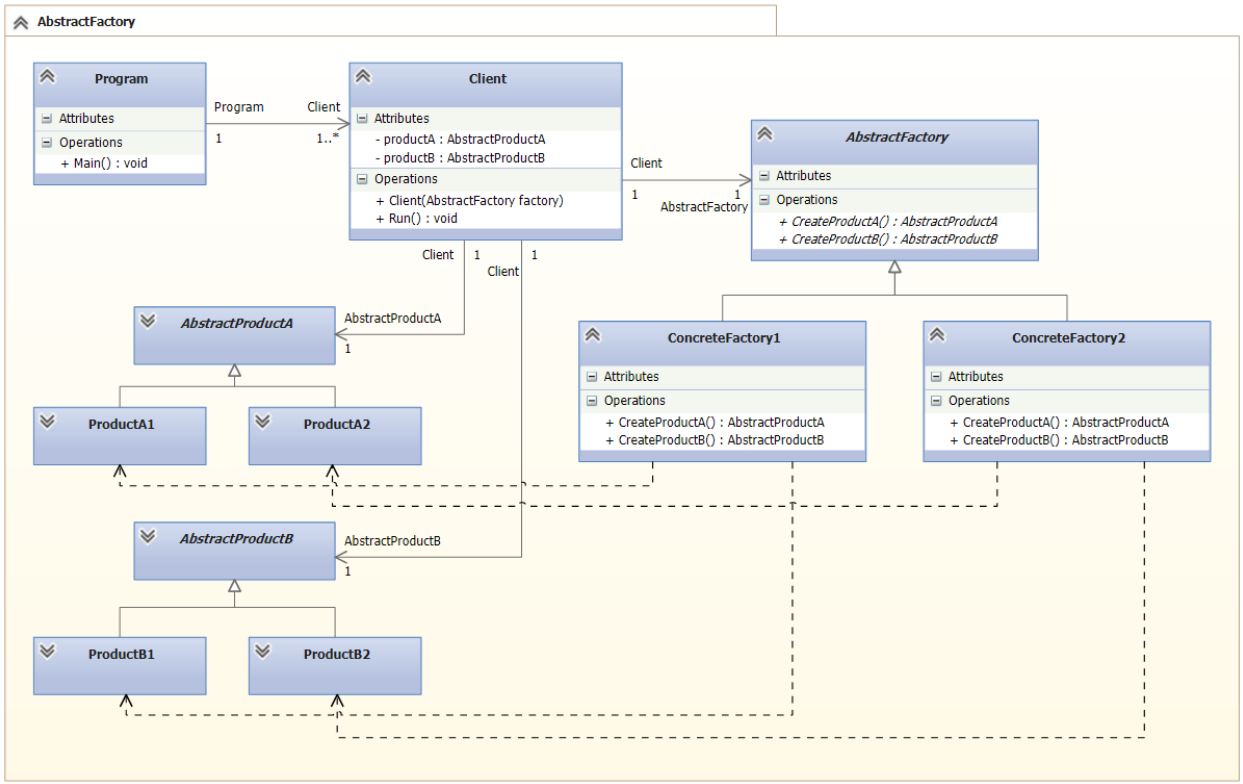


См. Пример к главе: \001_AbstractFactory\000_CocaCola_Pepsi

Используя такой подход к порождению продуктов, теперь не составит труда добавлять новые виды продуктов в систему (например, крышку для закрытия банки с водой).

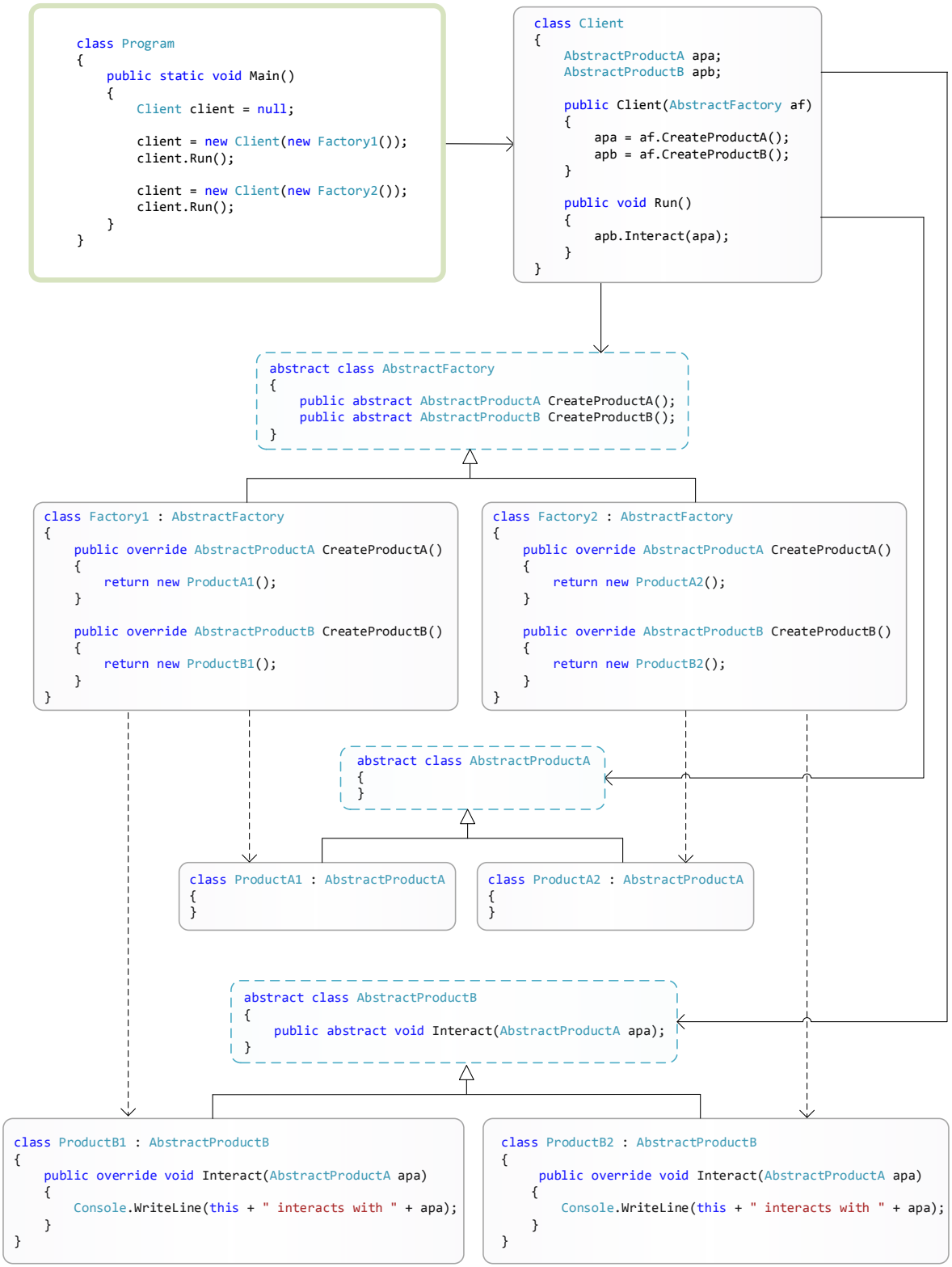

AbstractFactory
Program
Attributes
Operations
+ Main() : void
Program
Client
1
1..*
Client
Attributes
- bottle : AbstractBottle
- cover : AbstractCover
- water : AbstractWater
Operations
+ Client(AbstractFactory factory)
+ Run() : void
Client
1
AbstractFactory
1
AbstractFactory
Attributes
Operations
+ CreateBottle() : AbstractBottle
+ CreateCover() : AbstractCover
+ CreateWater() : AbstractWater
Client
1
Client
1
Client
1
AbstractWater
1
AbstractWater
Attributes
Operations
CocaColaFactory
Attributes
Operations
+ CreateBottle() : AbstractBottle
+ CreateCover() : AbstractCover
+ CreateWater() : AbstractWater
PepsiFactory
Attributes
Operations
+ CreateBottle() : AbstractBottle
+ CreateCover() : AbstractCover
+ CreateWater() : AbstractWater
CocaColaWater
Attributes
Operations
PepsiWater
Attributes
Operations
<<instantiate>>
<<instantiate>>
AbstractBottle
Attributes
Operations
+ Interact(AbstractWater water) : void
+ Interact(AbstractCover cover) : void
AbstractBottle
1
CocaColaBottle
Attributes
Operations
+ Interact(AbstractWater water) : void
+ Interact(AbstractCover cover) : void
PepsiBottle
Attributes
Operations
+ Interact(AbstractWater water) : void
+ Interact(AbstractCover cover) : void
<<instantiate>>
<<instantiate>>
AbstractCover
1
AbstractCover
Attributes
Operations
CocaColaCover
Attributes
Operations
PepsiCover
Attributes
Operations
<<instantiate>>
<<instantiate>>

Структура паттерна на языке UML



См. Пример к главе: \001_AbstractFactory\001_AbstractFactory

Структура паттерна на языке C#



См. Пример к главе: \001_AbstractFactory\001_AbstractFactory

Участники

- `AbstractProduct` - **Абстрактный продукт**:
  Абстрактные классы продуктов предоставляют абстрактные интерфейсы взаимодействия с объектами-продуктами производных конкретных классов.

- `AbstractFactory` - **Абстрактная фабрика**:
  Класс `AbstractFactory` содержит в себе набор абстрактных фабричных методов. Эти абстрактные методы описывают интерфейс взаимодействия с объектами-фабриками и имеют возвращаемые значения типа абстрактных-продуктов, тем самым предоставляя возможность применять технику абстрагирования процесса инстанцирования. Класс `AbstractFactory` не занимается созданием объектов-продуктов, ответственность за их создание ложится на производный класс `ConcreteFactory`.

- `Client` - **Клиент**:
  Класс Client создает и использует продукты, пользуясь исключительно интерфейсом абстрактных классов `AbstractFactory` и `AbstractProduct` и ему ничего не известно о конкретных классах фабрик и продуктов.

- `ConcreteProduct` - **Конкретный продукт**:
  Конкретные классы продукты, наследуются от абстрактных классов продуктов. Объекты-продукты конкретных классов предполагается создавать в телах фабричных методов реализаций соответствующих фабрик.

- `ConcreteFactory` - **Конкретная фабрика**:
  Классы конкретных фабрик, наследуются от абстрактной фабрики и реализуют фабричные методы порождающие объекты-продукты.

Отношения между участниками

Отношения между классами

- Класс `Client` связан связями отношения ассоциации с классами абстрактных продуктов и классом абстрактной фабрики.

- Все конкретные классы продуктов связаны связями отношения наследования с абстрактными классами продуктов.

- Все конкретные классы фабрик связаны связями отношения наследования с классом абстрактной фабрики и связями отношения зависимости (стереотипа - instantiate) с конкретными классами порождаемых продуктов.

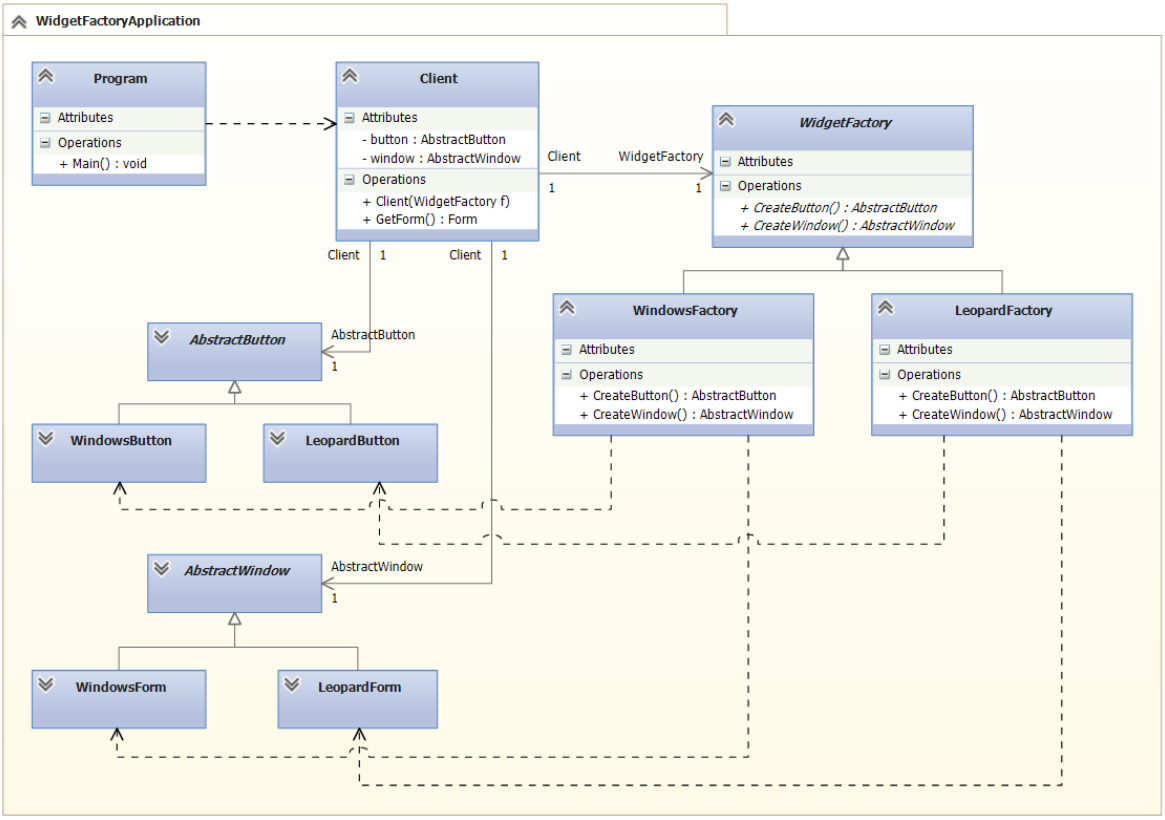
Отношения между объектами

- В системе создается (чаще всего) только один экземпляр конкретной фабрики. Задачей конкретной фабрики является создание объектов продуктов, входящих в определенное семейство.

- При создании экземпляра клиента, клиент конфигурируется экземпляром конкретной фабрики (ссылка на экземпляр фабрики передается в качестве аргумента конструктора клиента).

- Абстрактный класс `AbstractFactory` передает ответственность за создание объектов-продуктов производным конкретным фабрикам.

Мотивация



Рассмотрим простейшую программу, в которой поддерживается возможность создания и использования нескольких стилей пользовательского интерфейса, например, стиль Windows Explorer и стиль Mac OS Snow Leopard. В программе будут использоваться два элемента пользовательского интерфейса («controls - контролы» или иногда используется устаревшее название «widgets - виджеты»): Форма и Кнопка.

Хотелось бы предусмотреть в программе возможности быстрого изменения существующих стилей и легкого добавления новых стилей. Если создание элементов управления для каждого имеющегося стиля разбросано по многим участкам кода приложения, то изменять внешний вид такой программы будет неудобно.

Создадим абстрактный класс `WidgetFactory`, в котором имеется интерфейс (набор абстрактных фабричных методов) для создания элементов управления. Создадим абстрактные классы `AbstractWindow` и `AbstractButton` для описания каждого отдельного вида элемента управления и конкретные подклассы (`WindowsForm`, `LeopardForm` и `WindowsButton`, `LeopardButton`), реализующие элементы управления с определенным стилем. В абстрактном классе `WidgetFactory` имеются абстрактные операции (`CreateWindow` и `CreateButton`), возвращающие ссылки на новые экземпляры элементов управления для каждого абстрактного типа контролов. Клиент вызывает эти операции для получения экземпляров контролов, но при этом ничего не знает о том, какие именно конкретные классы используются для их построения. Соответственно клиент ничего не знает и о реализации выбранного стиля.

Для порождения контролов определенного стиля используются классы `WindowsFactory` и `LeopardFactory` производные от базового класса `WidgetFactory`, которые реализуют операции, необходимые для создания элемента управления определенного стиля. Например, операция CreateButton в классе `WindowsFactory` создает и возвращает кнопку в стиле Windows, тогда как операция `CreateButton` в классе `LeopardFactory` возвращает кнопку в стиле Snow Leopard. Клиент (`Client`) создает элементы управления, пользуясь исключительно интерфейсом, заданным в абстрактном классе `WidgetFactory`, и ему ничего не известно о классах, реализующих контролы для каждого конкретного стиля. Другими словами, клиент должен лишь придерживаться интерфейса, определенного абстрактным классом, а не конкретным классом.

Класс `WidgetFactory` устанавливает зависимости между конкретными классами контролов. Кнопка для Windows должна использоваться только с формой Windows, и это ограничение поддерживается автоматически, благодаря использованию класса `WindowsFactory`.

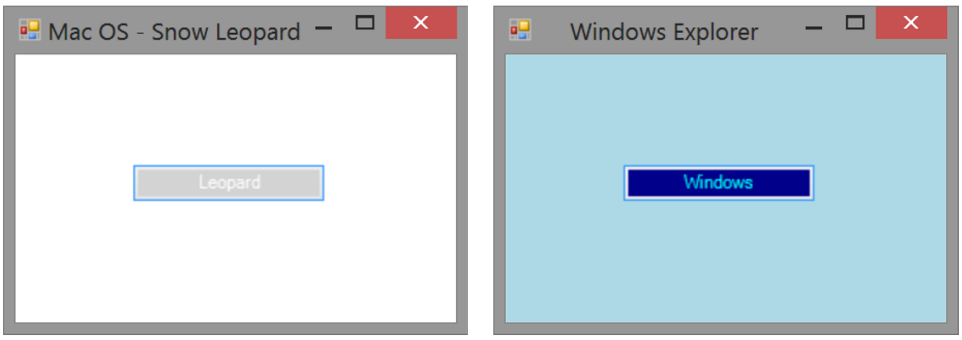


Рисунок. Результат подмены фабрик.

См. Пример к главе: \001_AbstractFactory\002_WidgetFactory

Применимость паттерна

Паттерн Abstract Factory рекомендуется использовать, когда:

- Требуется создавать объекты-продукты разных типов и налаживать между ними взаимодействие, при этом образуя семейства из этих объектов-продуктов. Входящие в семейство объекты-продукты обязательно должны использоваться вместе.

- Требуется построить подсистему (модуль или компонент) таким образом, чтобы ее внутреннее устройство (состояние и/или поведение) настраивалось при ее создании. При этом чтобы ни процесс, ни результат построения подсистемы не был зависим от способа создания в ней объектов, их композиции (составления и соединения объектов) и представления (настройки внутреннего состояния объектов).

- Подсистема или система должна настраиваться (конфигурироваться) через использование одного из семейств объектов-продуктов, порождаемых одним объектом-фабрикой;

Результаты

Паттерн Abstract Factory обладает следующими преимуществами:

- **Сокрытие работы с конкретными классами продуктов.**
  Фабрика скрывает от клиента детали реализации конкретных классов и процесс создания экземпляров этих классов. Конкретные классы-продуктов известны только конкретным фабрикам и в коде клиента они не используются. Клиент управляет экземплярами конкретных классов только через их абстрактные интерфейсы.

- **Позволяет легко заменять семейства используемых продуктов.**
  Экземпляр класса конкретной фабрики создается в приложении в одном месте и только один раз, что позволяет в дальнейшем проще подменять фабрики. Для того чтобы изменить семейство используемых продуктов, нужно просто создать новый экземпляр класса-фабрики, тогда заменится сразу все семейство.

- **Обеспечение совместного использования продуктов.**
  Позволяет легко контролировать взаимодействие между объектами-продуктами, которые спроектированы для совместного использования и входят в одно семейство.

Паттерн Abstract Factory обладает следующим недостатком:

- **Имеется небольшое неудобство добавления нового вида продуктов.**
  Для создания нового вида продуктов потребуется создать новые классы продуктов (абстрактные и конкретные), добавить новый абстрактный фабричный метод в абстрактный класс фабрики и реализовать этот абстрактный метод в производных конкретных классах фабриках, а также изменить код класса `Client`.

Реализация

Полезные приемы реализации паттерна Abstract Factory:

- **Объекты-фабрики существуют в единственном экземпляре.**
  В подсистеме, создается только один экземпляр класса `ConcreteFactory` для порождения, соответствующего семейства продуктов.

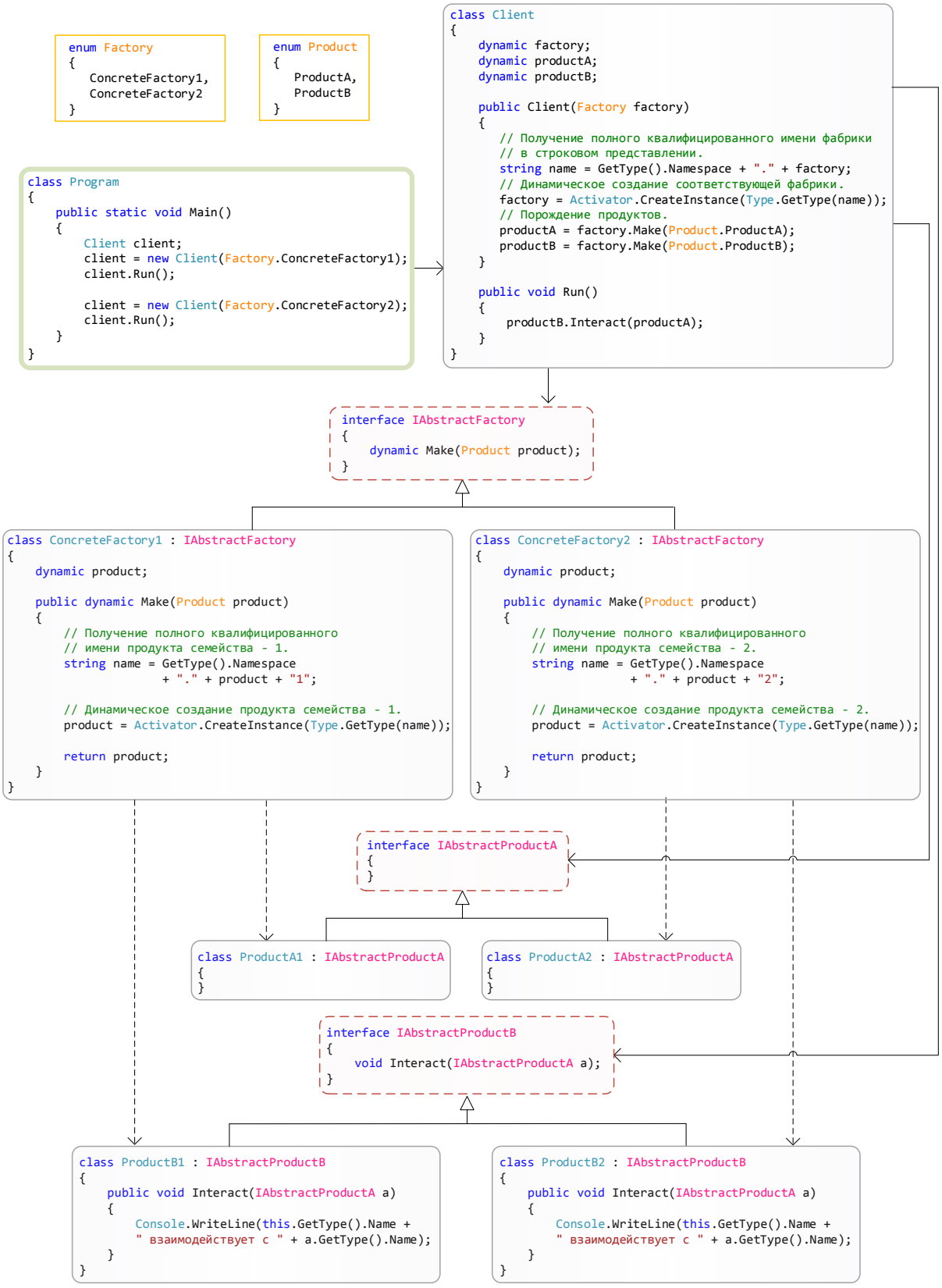
- **Создание объектов-продуктов.**
  Класс `AbstractFactory` предоставляет только интерфейс (набор абстрактных методов) для создания продуктов. Фактически продукты создаются в фабричных методах производных конкретных классов-фабрик. Конкретная фабрика реализует фабричные методы, которые возвращают ссылки на создаваемые ими экземпляры продуктов.

- **Определение расширяемых фабрик.**
  Класс `AbstractFactory` содержит абстрактные операции для создания продуктов. Имена фабричных методов включают в себя название типа порождаемого продукта. Для добавления нового вида продуктов нужно добавить новый абстрактный фабричный метод в абстрактный класс `AbstractFactory` и реализовать этот метод в производных классах-фабриках.

  Более гибкий способ – использовать фабричные методы с аргументами, описывающими виды создаваемых продуктов. Тип аргумента может быть числовой, строковой или типом перечисления (`enum`), однозначно описывающий тип порождаемого продукта. При таком подходе абстрактному классу `AbstractFactory` нужна только одна операция `Make` с аргументом, указывающим тип создаваемого продукта.

  Интересные и гибкие варианты порождения можно организовать в динамически типизированных языках, каким и является C#. Языки с поддержкой динамической типизации позволяют создавать семейства продуктов без наличия общего абстрактного базового класса, а фабричные методы могут иметь возвращаемые значения динамического типа (`dynamic`). Также в языке C# абстрактный класс можно заменить конструкцией языка выражающей такой стереотип как «интерфейс» (`interface IAbstractFactory`).

  Если в клиенте отказаться от приведения продуктов к базовому абстрактному типу то, можно было бы выполнить динамическое приведение типа (например, с помощью оператора `dynamic` в C#), но это не всегда безопасно и не всегда заканчивается успешно. Может возникнуть проблемная ситуация: все продукты будут возвращаться клиенту с интерфейсом, который не отображается в *intellisense* (*intellisense* – механизм автодополнения), клиенту будет сложно различать динамические типы продуктов, и могут возникать сложности с их использованием. Такие варианты представляют собой пример компромисса между гибкостью, расширяемостью интерфейса и производительностью.



См. Пример к главе: \001_AbstractFactory\003_AbstractFactory_Net

Пример кода игры «Лабиринт»

Реализацию игры-лабиринта, которая рассматривалась в начале этой главы можно изменить так, чтобы показать на ее примере возможность использования паттерна Abstract Factory.

Класс `MazeFactory` будет использоваться для создания компонентов лабиринта (комнат, стен и дверей между комнатами).

```
class MazeFactory
{
    public virtual Maze MakeMaze()
    {
        return new Maze();
    }

    public virtual Wall MakeWall()
    {
        return new Wall();
    }

    public virtual Room MakeRoom(int number)
    {
        return new Room(number);
    }

    public virtual Door MakeDoor(Room room1, Room room2)
    {
        return new Door(room1, room2);
    }
}
```

Метод `CreateMaze` класса `MazeGame`, принимает аргумент типа `MazeFactory` и возвращает построенный лабиринт (ссылку на экземпляр класса `Maze`).

```
class MazeGame
{
    MazeFactory factory = null;

    public Maze CreateMaze(MazeFactory factory)
    {
        this.factory = factory;
        Maze aMaze = this.factory.MakeMaze();
        Room r1 = this.factory.MakeRoom(1);
        Room r2 = this.factory.MakeRoom(2);
        Door aDoor = this.factory.MakeDoor(r1, r2);

        aMaze.AddRoom(r1);
        aMaze.AddRoom(r2);

        r1.SetSide(Direction.North, this.factory.MakeWall());
        r1.SetSide(Direction.East, aDoor);
        r1.SetSide(Direction.South, this.factory.MakeWall());
        r1.SetSide(Direction.West, this.factory.MakeWall());
        r2.SetSide(Direction.North, this.factory.MakeWall());
        r2.SetSide(Direction.East, this.factory.MakeWall());
        r2.SetSide(Direction.South, this.factory.MakeWall());
        r2.SetSide(Direction.West, aDoor);
        return aMaze;
    }
}
```

Эта версия метода CreateMaze лишена недостатка создания экземпляра лабиринта и его компонентов через прямой вызов конструкторов, теперь все компоненты создаются через использование фабричных методов, что позволит при создании варьировать типы создаваемых компонентов лабиринта.

Класс-фабрика EnchantedMazeFactory переопределяет фабричные методы базового класса-фабрики MazeFactory.

```
class EnchantedMazeFactory : MazeFactory
{
    public override Room MakeRoom(int number)
    {
        return new EnchantedRoom(number, CastSpell());
    }

    public override Door MakeDoor(Room room1, Room room2)
    {
        return new DoorNeedingSpell(room1, room2);
    }

    protected Spell CastSpell()
    {
        return null;
    }
}
```

Предположим, что в данном лабиринте в одной из комнат заложена бомба, когда бомба взрывается то в комнате обрушиваются стены. Для этого нужно создать классы BombedWall и RoomWithABomb. Класс BoombedWall наследуется от класса Wall, а класс RoomWithABomb наследуется от класса Room.

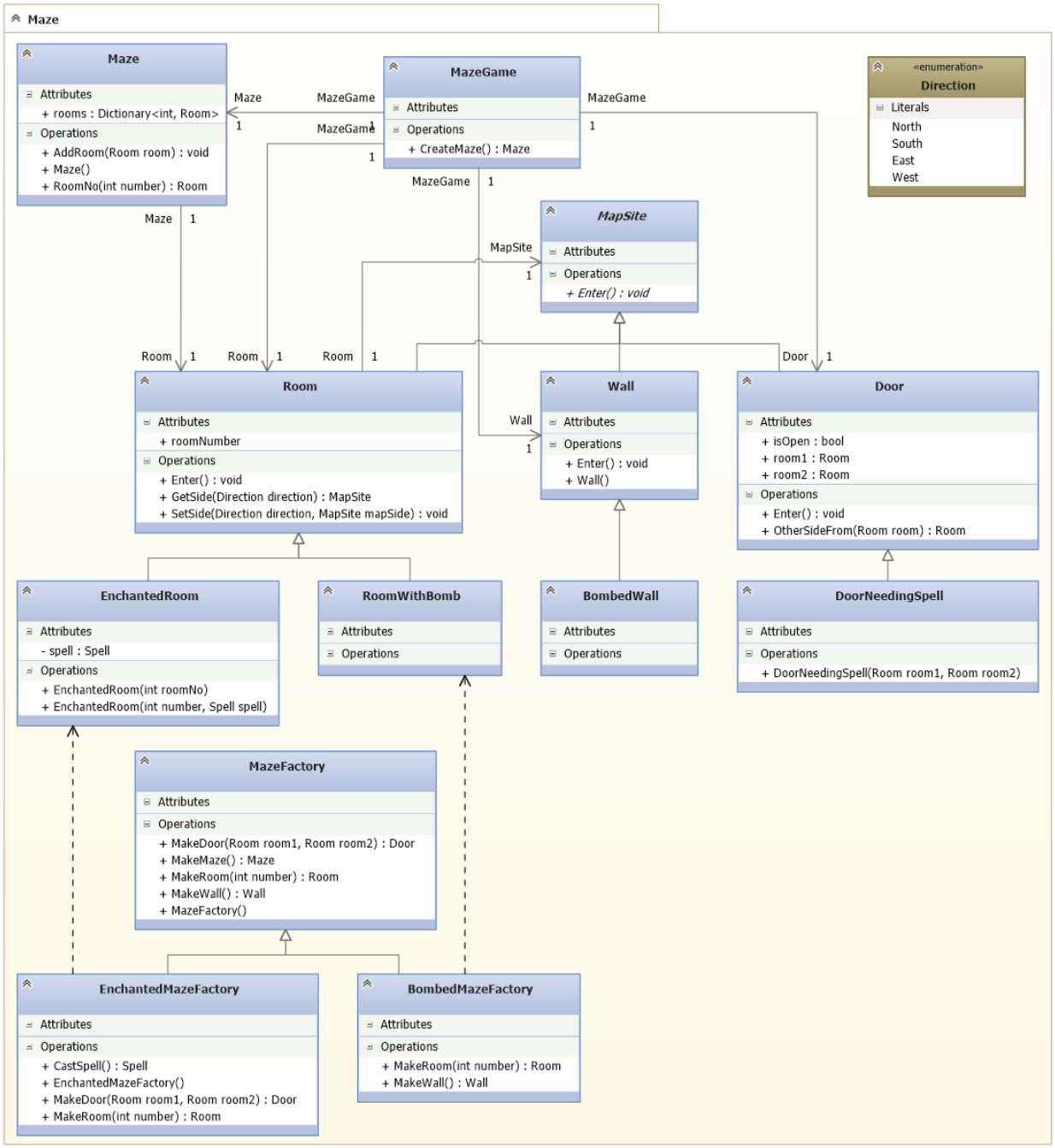
```
class BombedWall : Wall
{
}

class RoomWithBomb : Room
{
    // Конструктор.
    public RoomWithBomb(int roomNo)
        : base(roomNo)
    {
    }
}
```

Класс фабрика BombedMazeFactory переопределяет фабричные методы базового класса фабрики MazeFactory

```
// Фабрика для создания комнат с бомбой.
class BombedMazeFactory : MazeFactory
{
    // Метод создает взорванные стены.
    public override Wall MakeWall()
    {
        return new BombedWall();
    }

    // Метод создает комнату с бомбой.
    public override Room MakeRoom(int number)
    {
        return new RoomWithBomb(number);
    }
}
```

Для построения лабиринта с бомбами вызывается метод `CreateMaze` класса `MazeGame`, которому в качестве аргумента передаётся ссылка на экземпляр класса `BombedMazeFactory`.

Важно заметить, что класс `MazeFactory` является конкретным, а не абстрактным классом, поэтому он используется и как абстрактная фабрика, так и как конкретная фабрика. Такой подход представляет собой еще одну разновидность реализации паттерна Abstract Factory. Так как `MazeFactory` является конкретным классом, хранящим в себе только фабричные методы, легко получить новую фабрику, для этого требуется просто создать новый подкласс класса `MazeFactory` и переопределить в нем фабричные методы.



См. Пример к главе: \MAZE\001_Maze_AF

Известные применения паттерна в .Net

Microsoft.Build.Tasks.CodeTaskFactory
http://msdn.microsoft.com/ru-ru/library/microsoft.build.tasks.codetaskfactory.aspx

Microsoft.Build.Tasks.XamlTaskFactory
http://msdn.microsoft.com/ru-ru/library/microsoft.build.tasks.xamltaskfactory.aspx

Microsoft.IE.SecureFactory
http://msdn.microsoft.com/ru-ru/library/microsoft.ie.securefactory(v=vs.90).aspx

System.Activities.Presentation.Model.ModelFactory
http://msdn.microsoft.com/ru-ru/library/system.activities.presentation.model.modelfactory.aspx

System.Data.Common.DbProviderFactory
http://msdn.microsoft.com/ru-ru/library/system.data.common.dbproviderfactory.aspx

System.Data.EntityClient.EntityProviderFactory
http://msdn.microsoft.com/ru-ru/library/system.data.entityclient.entityproviderfactory.aspx

System.Data.Odbc.OdbcFactory
http://msdn.microsoft.com/ru-ru/library/system.data.odbc.odbcfactory.aspx

System.Data.OleDb.OleDbFactory
http://msdn.microsoft.com/ru-ru/library/system.data.oledb.oledbfactory.aspx

System.Data.OracleClient.OracleClientFactory
http://msdn.microsoft.com/ru-ru/library/system.data.oracleclient.oracleclientfactory.aspx

System.Data.Services.DataServiceHostFactory
http://msdn.microsoft.com/ru-ru/library/system.data.services.dataservicehostfactory.aspx

System.Data.SqlClient.SqlClientFactory
http://msdn.microsoft.com/ru-ru/library/system.data.sqlclient.sqlclientfactory.aspx

System.ServiceModel.ChannelFactory
http://msdn.microsoft.com/ru-ru/library/system.servicemodel.channelfactory.aspx

System.Threading.Tasks.TaskFactory
http://msdn.microsoft.com/ru-ru/library/system.threading.tasks.taskfactory(v=vs.110).aspx

System.Web.Compilation.ResourceProviderFactory
http://msdn.microsoft.com/ru-ru/library/system.web.compilation.resourceproviderfactory.aspx

System.Web.Hosting.AppDomainFactory
http://msdn.microsoft.com/ru-ru/library/system.web.hosting.appdomainfactory(v=vs.90).aspx

System.Xml.Serialization.XmlSerializerFactory
http://msdn.microsoft.com/ru-ru/library/system.xml.serialization.xmlserializerfactory(v=vs.90).aspx

И т.д.

Паттерн Builder

Название

Строитель

Также известен как

-

Классификация

По цели: порождающий
По применимости: к объектам

Частота использования

Ниже средней - 1 **2** 3 4 5

Назначение

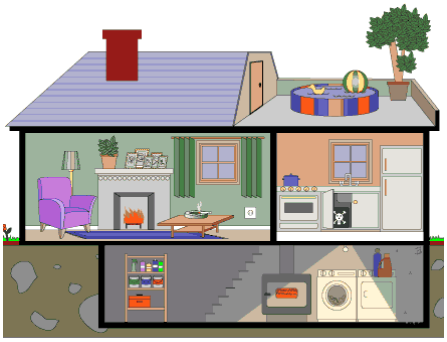
Паттерн Builder – помогает организовать пошаговое построение сложного объекта-продукта так, что клиенту не требуется понимать последовательность шагов и внутреннее устройство строящегося объекта-продукта, при этом в результате одного и того же процесса конструирования могут получаться объекты-продукты с различным представлением (внутренним устройством).

Введение

Кто такой строитель в объективной реальности? Строитель - это человек, который занимается возведением зданий и сооружений (мостов, плотин, туннелей и пр.). Результатом строительства считается возведённое здание (сооружение). Для того чтобы здание было построено по правилам и соответствовало проектным нормам, строителями нужно руководить. Должность руководителя на стройке называется прораб (сокращение от «производитель работ»). Прораб дает указания строителю, как и в каком порядке проводить строительные работы. Паттерн Builder, построен на подобной метафоре.

Прораб

Строитель



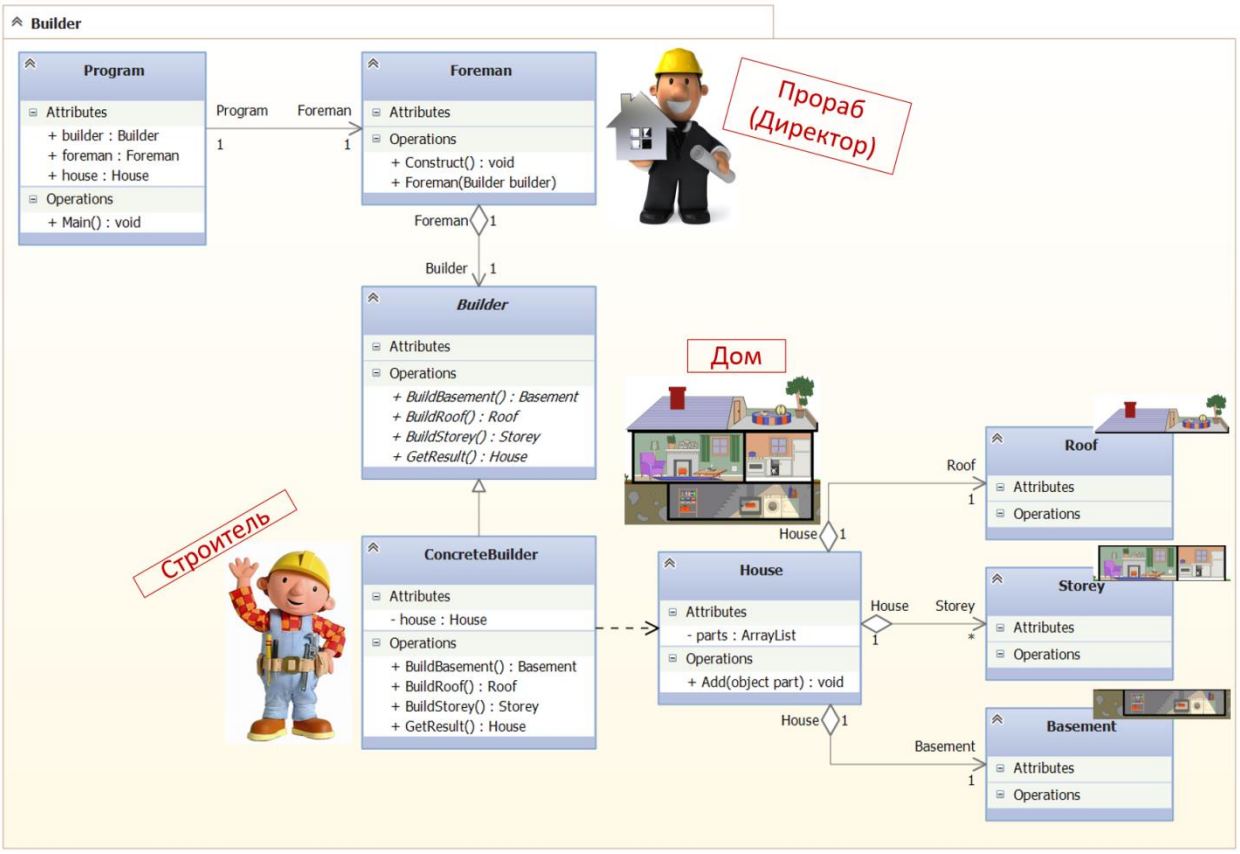
Дом

Прораб, должен давать строителю инструкции по построению частей дома в определенной последовательности. Например,

1. «Построй подвал»,
2. «Построй этаж»,
3. «Построй крышу».

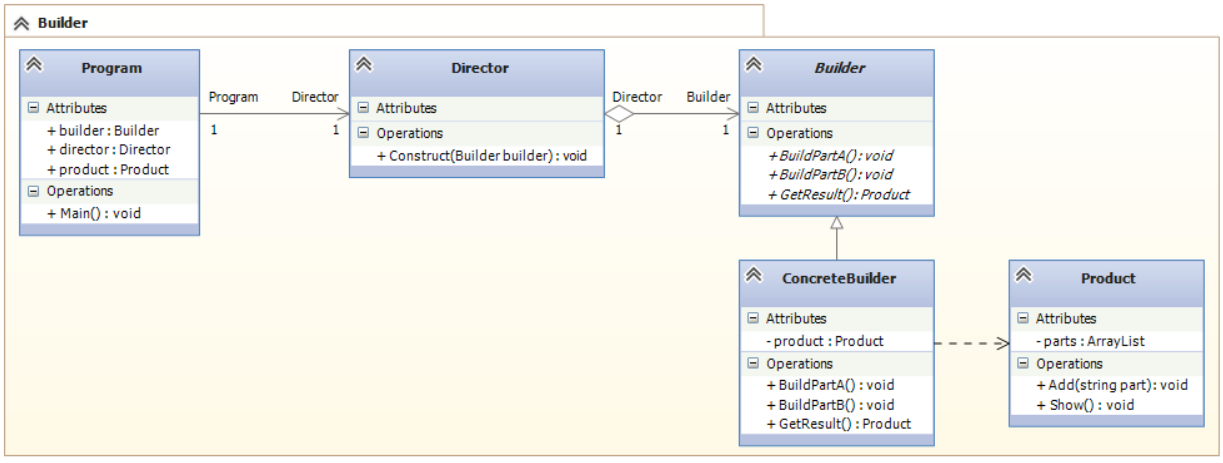
Способ построения дома определяет тип конкретного строителя. Строитель-каменщик, который строит дом из кирпича, будет строить дом отличным способом, от строителя-плотника который будет

строить сруб (деревянный дом) из бревен. Таким образом, согласно проекту, прораб должен вызвать соответствующего строителя и давать ему соответствующие инструкции в определенном порядке. Сначала построить подвал, потом этаж и в последнюю очередь крышу.



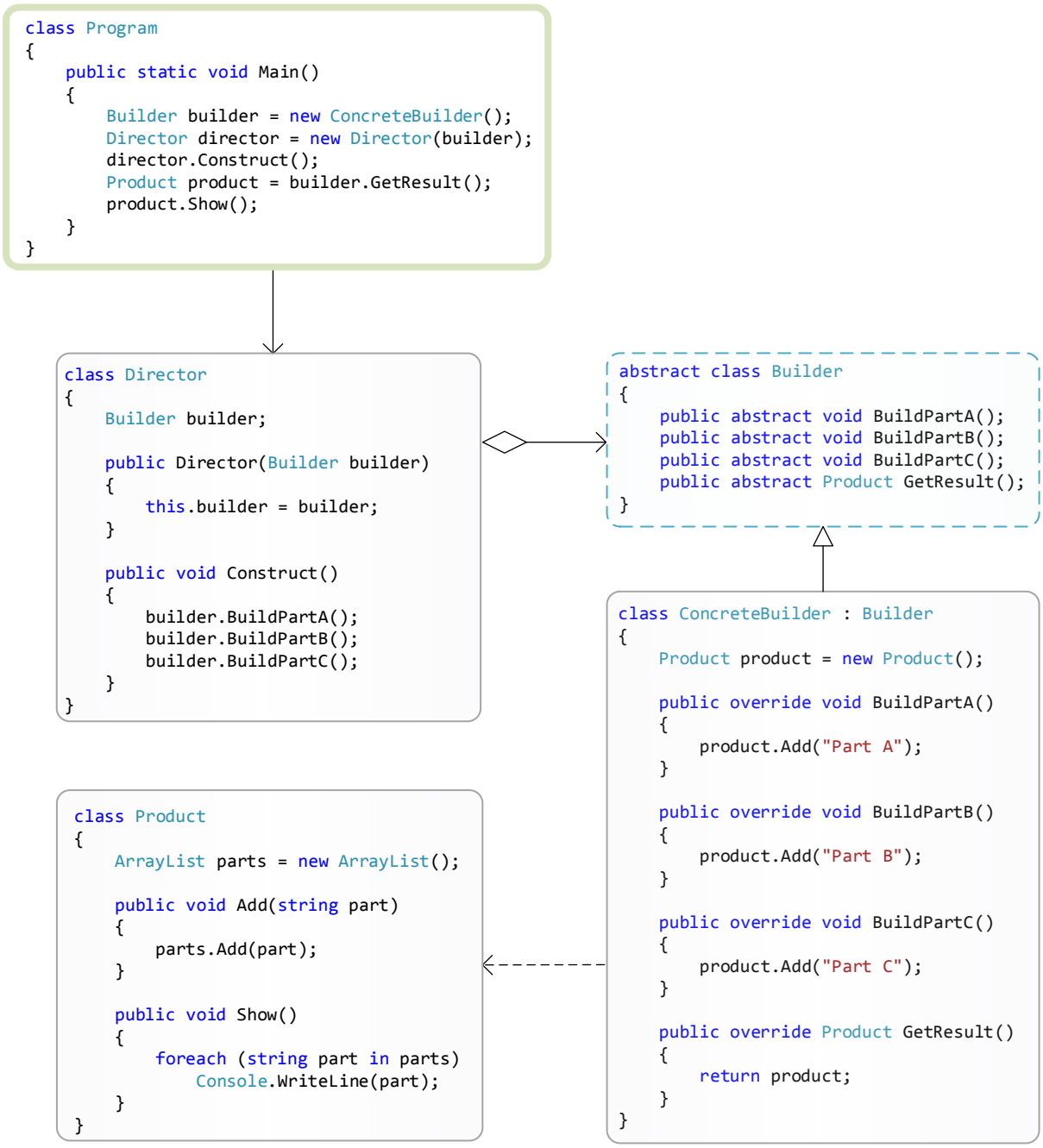
См. Пример к главе: \002_Builder\000_Builder

Структура паттерна на языке UML



См. Пример к главе: \002_Builder\001_Builder

Структура паттерна на языке C#



```
class Program
{
    public static void Main()
    {
        Builder builder = new ConcreteBuilder();
        Director director = new Director(builder);
        director.Construct();
        Product product = builder.GetResult();
        product.Show();
    }
}
```

```
class Director
{
    Builder builder;

    public Director(Builder builder)
    {
        this.builder = builder;
    }

    public void Construct()
    {
        builder.BuildPartA();
        builder.BuildPartB();
        builder.BuildPartC();
    }
}
```

```
abstract class Builder
{
    public abstract void BuildPartA();
    public abstract void BuildPartB();
    public abstract void BuildPartC();
    public abstract Product GetResult();
}
```

```
class ConcreteBuilder : Builder
{
    Product product = new Product();

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

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

    public override void BuildPartC()
    {
        product.Add("Part C");
    }

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

```
class Product
{
    ArrayList parts = new ArrayList();

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

    public void Show()
    {
        foreach (string part in parts)
            Console.WriteLine(part);
    }
}
```

См. Пример к главе: \002_Builder\001_Builder

Участники

- `Product` - **Продукт**:
  Представляет собой класс сложно-конструируемого объекта-продукта и содержит в себе набор методов для сборки конечного результата-продукта из частей. Класс продукта может быть связан связями отношений агрегации, с классами которые описывают составные части создаваемого продукта.

- `Builder` - **Абстрактный строитель**:
  Предоставляет набор абстрактных методов (интерфейс) для создания объекта-продукта из частей и получения готового результата.

- `ConcreteBuilder` - **Конкретный строитель**:
  Конструирует объект-продукт собирая его из частей, реализуя интерфейс, заданный абстрактным строителем (`Builder`). Предоставляет доступ к готовому продукту (возвращает продукт клиенту или в частном случае директору (`Director`)).

- `Director` – **Директор (Распорядитель)**:
  Пользуясь интерфейсом строителя (`Builder`), директор дает строителю указание построить продукт.

Отношения между участниками

Отношения между классами

- Класс `Director` связан связью отношения агрегации с абстрактным классом `Builder`.

- Класс `ConcreteBuilder` связан связью отношения наследования с абстрактным классом `Builder` и связью отношения зависимости с классом `Product`.

- Класс `Product` может быть связан связями отношения агрегации с классами частей (`Part`).

Отношения между объектами

- Клиент создает экземпляр класса `ConcreteBuilder`.

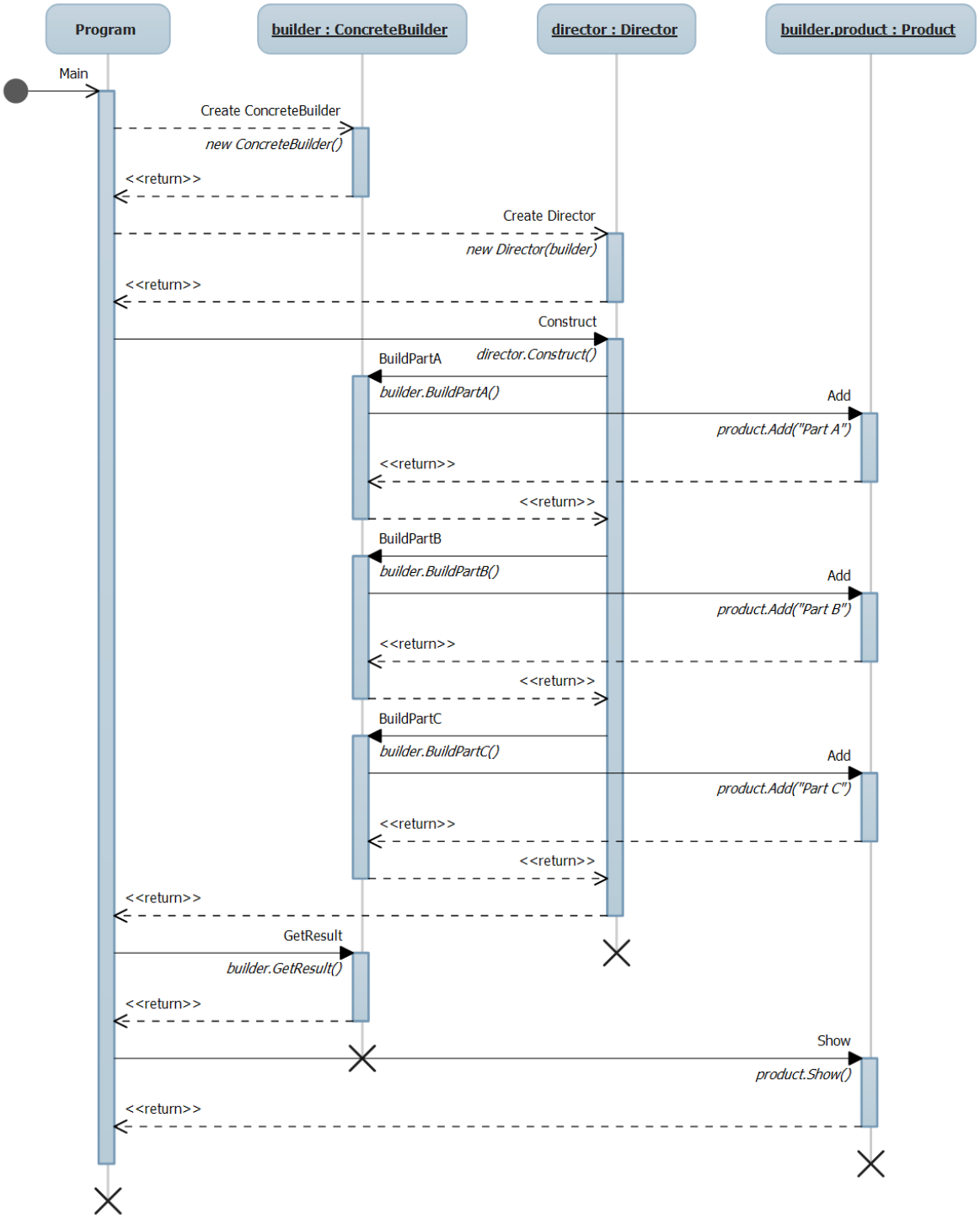
- Клиент создает экземпляр класса `Director` при этом в качестве аргумента конструктора передает ссылку на ранее созданный экземпляр класса `ConcreteBuilder`.

- Директор (`Director`) вызывает на строителе (`ConcreteBuilder`) методы, тем самым уведомляя строителя о том, что требуется построить определенную часть продукта.

- Строитель выполняет операции по построению продукта, добавляя к продукту те части, которые указывает директор (`Director`).

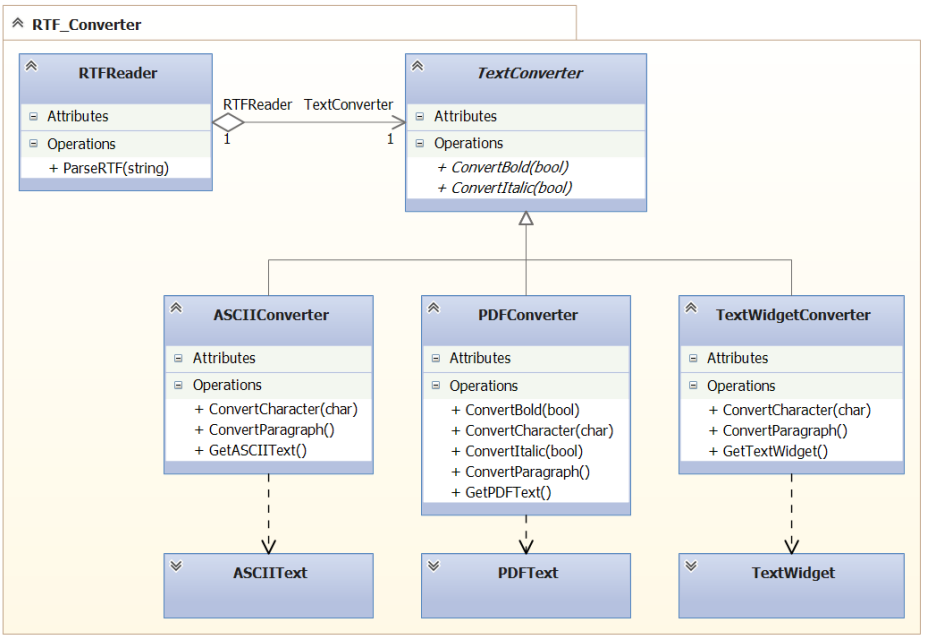
- Клиент получает от строителя ссылку на экземпляр построенного продукта.

На диаграмме последовательностей показаны отношения между объектами (директором и строителем).



Мотивация

Предлагается написать программу для преобразования RTF документа в другие форматы: ASCII, PDF и в представление для отображения на форме в элементе управления. Требуется предусмотреть возможность легкого добавления новых механизмов преобразований в другие форматы. Возможное число форматов, в которые необходимо будет преобразовать документ заранее неизвестно. Поэтому должна быть предусмотрена (обеспечена) возможность легкого добавления нового конвертера, например, конвертера из RTF в html формат.



Таким образом, нужно сконфигурировать экземпляр класса `RTFReader` (Директор) объектом типа `TextConverter` (абстрактный Builder), который мог бы преобразовывать RTF формат в другие форматы. При разборе (анализе) документа в формате RTF, объект класса `RTFReader` дает команду объекту типа `TextConverter` выполнить преобразование формата. При этом каждый раз при распознавании «лексемы RTF» (т.е. простого текста или управляющего слова) `RTFReader` вызывает необходимый метод объекта типа `TextConverter`. Подклассы класса `TextConverter` (конкретные конвертеры) отвечают, как за преобразование данных (текста), так и за представление лексемы в новом формате.

Каждый подкласс класса `TextConverter` (конкретный строитель) позволяет преобразовать RTF формат только в один определенный формат. Например, `ASCIIConverter` преобразовывает RTF формат в простейший ASCII формат, при этом игнорирует преобразование таблиц, изображений и шрифтов. С другой стороны, `PDFConverter` будет преобразовывать все содержимое включая таблицы, изображения, шрифты, стили и пр. А `TextWidgetConverter` построит объект пользовательского интерфейса (контрол), и преобразование формата будет зависеть от возможностей используемого элемента управления (контрола). Например, `TextBox` сможет отображать только простой текст, тогда как `RichTextBox` сможет полноценно отобразить все составляющие документа в RTF формате.

Класс каждого конкретного конвертера (строителя) использует механизм создания и сборки сложного объекта-продукта и скрывает этот механизм за реализацией интерфейса, предоставленного классом `TextConverter`. Конвертер отделен от объекта класса `RTFReader` (директора), который отвечает за синтаксический разбор RTF документа.

В паттерне Builder абстрагированы все отношения между директором и строителями и любой объект типа `TextConverter` будет называться Строителем (`Builder`), а `RTFReader` - Директором (`Director`). Применение паттерна Builder в данном примере позволяет отделить алгоритм интерпретации текста в RTF формате (анализатор RTF документов) от алгоритма конвертации документа в новый формат. Это позволяет повторно использовать алгоритм интерпретации текста в RTF, реализованный в `RTFReader` (Директоре), в связке с различными конвертерами в другие форматы (Строителями). Для этого достаточно сконфигурировать `RTFReader` необходимым конвертером (подклассом класса `TextConverter`).

См. Пример к главе: \002_Builder\002_RTF Converter

Применимость паттерна

Паттерн Строитель рекомендуется использовать, когда:

- Алгоритм пошагового создания сложного объекта-продукта не должен зависеть от того, из каких частей состоит объект-продукт и как эти части стыкуются между собой;

- Процесс создания продукта должен обеспечивать возможность получения различных вариаций создаваемого продукта.

Результаты

Паттерн Builder обладает следующими преимуществами:

- **Позволяет изменять состав продукта.**
  Абстрактный класс `Builder` предоставляет директору набор абстрактных методов (абстрактный интерфейс) для управления построением продукта. За абстрактным интерфейсом `Builder` скрывает внутреннюю структуру создаваемого продукта и процесс его построения. Поскольку построение продукта производится согласно абстрактному интерфейсу, то для изменения структуры продукта достаточно создать новую разновидность строителя;

- **Скрывает код, реализующий конструирование и представление.**
  Паттерн Builder улучшает модульность, скрывая способы построения и представления сложных объектов-продуктов. Клиенты ничего не знают о классах, входящих в состав внутренней структуры продукта, использование этих классов отсутствует в интерфейсе строителя.
  Конкретные строители `ConcreteBuilder` содержат код, необходимый для создания и сборки конкретного вида продукта. Код пишется только один раз и разные директоры могут использовать его повторно для построения различных вариантов продукта из одних и тех же частей комбинируя эти части.

- **Предоставляет полный контроль над процессом построения продукта.**
  В отличие от других порождающих паттернов, которые сразу конструируют весь объект-продукт полностью, строитель строит продукт шаг за шагом под управлением директора.
  И только тогда, когда построение продукта завершено, директор или клиент забирают его у строителя. Поэтому интерфейс строителя в большей степени отражает процесс пошагового конструирования продукта, нежели другие порождающие паттерны. Это позволяет обеспечить полный контроль над процессом конструирования, а значит, и над внутренней структурой (комбинацией частей) готового продукта.

Реализация

Обычно используется абстрактный класс `Builder`, предоставляющий интерфейс для построения каждой отдельной части продукта, который директор может «попросить» создать. В классах конкретных строителей `ConcreteBuilder` реализуются абстрактные операции абстрактного класса `Builder`.

Полезные приемы реализации паттерна строитель:

- **Интерфейс строителя.**
  Строители конструируют продукты шаг за шагом. Интерфейс класса `Builder` должен быть достаточно общим, чтобы обеспечить создание продуктов при любой реализации конкретных строителей. Иногда может потребоваться доступ к частям уже сконструированного, готового продукта и такую возможность желательно предусмотреть.

- **Отсутствие общего базового абстрактного класса для продуктов.**
  Чаще всего продукты имеют настолько разный состав, что создание для них общего базового класса ничего не дает.

Пример кода игры «Лабиринт»

Класс MazeBuilder предоставляет абстрактный интерфейс для построения лабиринта:

```
abstract class MazeBuilder
{
    public abstract void BuildMaze();
    public abstract void BuildRoom(int roomNo);
    public abstract void BuildDoor(int roomFrom, int roomTo);
    public abstract Maze GetMaze();
}
```

Этот интерфейс позволяет создавать три типа объектов: целый лабиринт, комнату с номером и двери между пронумерованными комнатами. Реализация метода GetMaze в подклассах MazeBuilder создает и возвращает лабиринт клиенту.

Класс MazeGame предоставляет собой объектно-ориентированное представление всей игры. Метод CreateMaze класса MazeGame, принимает в качестве аргумента ссылку на экземпляр конкретного строителя типа MazeBuilder и возвращает построенный лабиринт (ссылку на экземпляр класса Maze).

```
public Maze CreateMaze(MazeBuilder builder)
{
    builder.BuildMaze();
    builder.BuildRoom(1);
    builder.BuildRoom(2);
    builder.BuildDoor(1, 2);

    // Возвращает готовый продукт (Лабиринт)
    return builder.GetMaze();
}
```

Эта версия метода CreateMaze показывает, что строитель скрывает внутреннее устройство лабиринта, то есть конкретные классы комнат, дверей и стен.

Как и все другие порождающие паттерны, паттерн строитель позволяет скрывать способы создания объектов. В данном примере сокрытие организуется при помощи интерфейса, предоставляемого классом MazeBuilder. Это означает, что MazeBuilder можно использовать для построения лабиринтов любых разновидностей. В качестве примера для построения альтернативного лабиринта рассмотрим метод CreateComplexMaze принадлежащий классу MazeGame:

```
public Maze CreateComplexMaze(MazeBuilder builder)
{
    // Построение 1001-й комнаты.
    for (int i = 0; i < 1001; i++)
    {
        builder.BuildRoom(i + 1);
    }

    return builder.GetMaze();
}
```

Важно понимать, что MazeBuilder не создает лабиринты напрямую, его главная задача – предоставить абстрактный интерфейс, описывающий создание лабиринта. Реальную работу по построению лабиринта выполняют конкретные подклассы класса MazeBuilder.

Класс StandardMazeBuilder реализует логику построения простых лабиринтов.

```
    // Подкласс StandardMazeBuilder - содержит реализацию построения простых
лабиринтов.
    class StandardMazeBuilder : MazeBuilder
    {

        Maze currentMaze = null;

        // Конструктор.
        public StandardMazeBuilder()
        {
            this.currentMaze = null;
        }


        // Инстанцирует экземпляр класса Maze, который будет собираться другими
        // операциями.
        public override void BuildMaze()
        {
            this.currentMaze = new Maze();
        }


        // Создает комнату и строит вокруг нее стены.
        public override void BuildRoom(int roomNo)
        {
            //if (currentMaze.RoomNo(roomNo) == null)
            {
                Room room = new Room(roomNo);
                currentMaze.AddRoom(room);

                room.SetSide(Direction.North, new Wall());
                room.SetSide(Direction.South, new Wall());
                room.SetSide(Direction.East, new Wall());
                room.SetSide(Direction.West, new Wall());
            }
        }


        // Чтобы построить дверь между двумя комнатами, требуется найти обе комнаты в
        // лабиринте и их общую стену.
        public override void BuildDoor(int roomFrom, int roomTo)
        {
            Room room1 = currentMaze.RoomNo(roomFrom);
            Room room2 = currentMaze.RoomNo(roomTo);
            Door door = new Door(room1, room2);

            room1.SetSide(CommonWall(room1, room2), door);
            room2.SetSide(CommonWall(room2, room1), door);
        }


        // Возвращает клиенту собранный продукт т.е., лабиринт.
        public override Maze GetMaze()
        {
            return this.currentMaze;
        }
```

```
        // CommonWall - Общая стена.
        // Это вспомогательная операция, которая определяет направление общей для
двух
        // комнат стены.
        private Direction CommonWall(Room room1, Room room2)
        {
            if (room1.GetSide(Direction.North) is Wall &&
                room1.GetSide(Direction.South) is Wall &&
                room1.GetSide(Direction.East) is Wall &&
                room1.GetSide(Direction.West) is Wall &&
                room2.GetSide(Direction.North) is Wall &&
                room2.GetSide(Direction.South) is Wall &&
                room2.GetSide(Direction.East) is Wall &&
                room2.GetSide(Direction.West) is Wall)
            {
                return Direction.East;
            }
            else
            {
                return Direction.West;
            }
        }
    }
```

См. Пример к главе: \MAZE\002_Maze_BLD

Известные применения паттерна в .Net

`System.Data.Common.DbCommandBuilder`
http://msdn.microsoft.com/ru-ru/library/system.data.common.dbcommandbuilder(v=vs.90).aspx

`System.Data.Common.DbConnectionStringBuilder`
http://msdn.microsoft.com/ru-ru/library/system.data.common.dbconnectionstringbuilder.aspx

`System.Data.Odbc.OdbcCommandBuilder`
http://msdn.microsoft.com/ru-ru/library/system.data.odbc.odbccommandbuilder.aspx

`System.Data.Odbc.OdbcConnectionStringBuilder`
http://msdn.microsoft.com/ru-ru/library/system.data.odbc.odbcconnectionstringbuilder.aspx

`System.Data.OleDb.OleDbCommandBuilder`
http://msdn.microsoft.com/ru-ru/library/system.data.oledb.oledbcommandbuilder(v=vs.90).aspx

`System.Data.OleDb.OleDbConnectionStringBuilder`
http://msdn.microsoft.com/ru-ru/library/system.data.oledb.oledbconnectionstringbuilder.aspx

`System.Data.SqlClient.SqlCommandBuilder`
http://msdn.microsoft.com/ru-ru/library/system.data.sqlclient.sqlcommandbuilder.aspx

`System.Data.SqlClient.SqlConnectionStringBuilder`
http://msdn.microsoft.com/ru-ru/library/system.data.sqlclient.sqlconnectionstringbuilder.aspx

`System.Reflection.Emit.ConstructorBuilder`
http://msdn.microsoft.com/ru-ru/library/system.reflection.emit.constructorbuilder.aspx

`System.Reflection.Emit.EnumBuilder`
http://msdn.microsoft.com/ru-ru/library/system.reflection.emit.enumbuilder.aspx

`System.Reflection.Emit.EventBuilder`
http://msdn.microsoft.com/ru-ru/library/system.reflection.emit.eventbuilder.aspx

`System.Reflection.Emit.FieldBuilder`
http://msdn.microsoft.com/ru-ru/library/system.reflection.emit.fieldbuilder.aspx

`System.Reflection.Emit.MethodBuilder`
http://msdn.microsoft.com/ru-ru/library/system.reflection.emit.methodbuilder.aspx

`System.Reflection.Emit.ParameterBuilder`
http://msdn.microsoft.com/ru-ru/library/system.reflection.emit.parameterbuilder(v=vs.100).aspx

`System.Reflection.Emit.PropertyBuilder`
http://msdn.microsoft.com/ru-ru/library/system.reflection.emit.propertybuilder.aspx

`System.Reflection.Emit.TypeBuilder`
http://msdn.microsoft.com/ru-ru/library/system.reflection.emit.typebuilder(v=vs.110).aspx

`System.Text.StringBuilder`
http://msdn.microsoft.com/ru-ru/library/system.text.stringbuilder.aspx

И т.д.

Паттерн Factory Method

Название

Фабричный Метод

Также известен как

Virtual Constructor (Виртуальный Конструктор)

Классификация

По цели: порождающий
По применимости: к классам

Частота использования

Высокая - 1 2 3 4 **5**

Назначение

Паттерн Factory Method – предоставляет абстрактный интерфейс (набор методов) для создания объекта-продукта, но оставляет возможность, разработчикам классов, реализующих этот интерфейс самостоятельно принять решение о том, экземпляр какого конкретного класса-продукта создать. Паттерн Factory Method позволяет базовым абстрактным классам передать ответственность за создание объектов-продуктов своим производным классам.

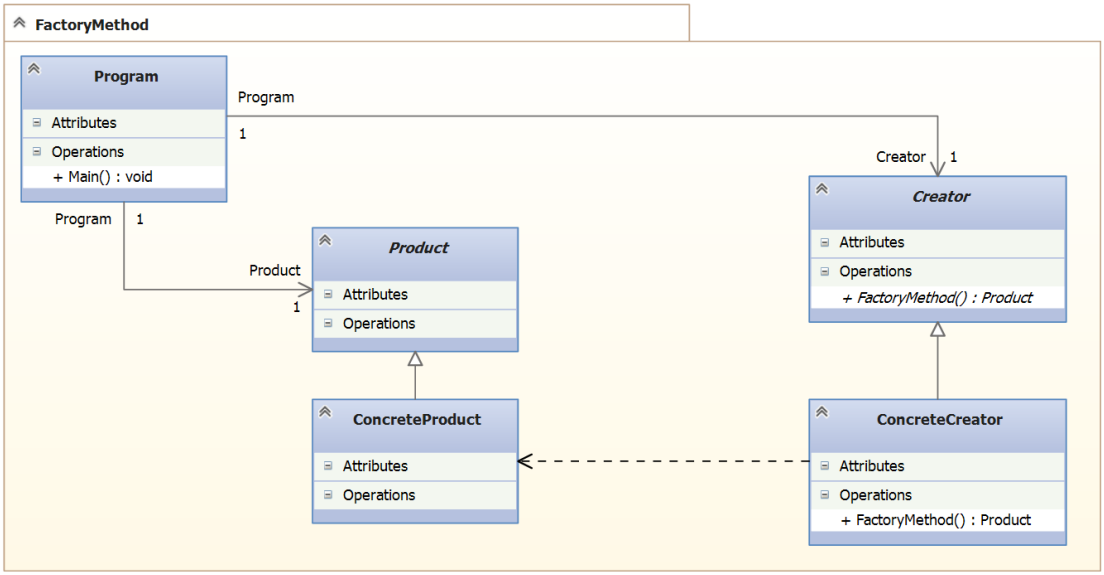
Введение

Паттерн Factory Method лежит в основе всех порождающих паттернов, организовывая процесс порождения объектов-продуктов. Если проектировщик на этапе проектирования системы не может сразу определиться с выбором подходящего паттерна для организации процесса порождения продукта в конкретной ситуации, то сперва следует воспользоваться паттерном Factory Method.

Например, если проектировщик, не определился со сложностью продукта или с необходимостью и способом организации взаимодействия между несколькими продуктами, тогда есть смысл сперва воспользоваться паттерном Factory Method. Позднее, когда требования будут сформулированы более четко, можно будет произвести быструю подмену паттерна Factory Method на другой порождающий паттерн, более соответствующий проектной ситуации.

Важно помнить, что Factory Method является паттерном уровня классов, и он сфокусирован только на отношениях между классами. Основной задачей паттерна Factory Method является организация техники делегирования ответственности за создание объектов продуктов одним классом (часто абстрактным) другому классу (производному конкретному классу). Другими словами – абстрактный класс содержащий в себе абстрактный фабричный метод, говорит своему производному конкретному классу: «Конкретный класс, я поручаю твоему разработчику самостоятельно выбрать конкретный класс порождаемого объекта-продукта при реализации моего абстрактного фабричного метода».

Структура паттерна на языке UML



См. Пример к главе: \003_FactoryMethod\001_FactoryMethod

Структура паттерна на языке C#

```
class Program
{
    static void Main()
    {
        Creator creator = null;
        Product product = null;

        creator = new ConcreteCreator();
        product = creator.FactoryMethod();

        creator.AnOperation();
    }
 }
```

```
abstract class Creator
{
    Product product;

    public abstract Product FactoryMethod();

    public void AnOperation()
    {
        product = FactoryMethod();
    }
 }
```

```
abstract class Product
{
}
```

```
class ConcreteProduct : Product
{
   public ConcreteProduct()
   {
        Console.WriteLine(this.GetHashCode());
   }
}
```

```
class ConcreteCreator : Creator
{
   public override Product FactoryMethod()
   {
       return new ConcreteProduct();
   }
}
```

См. Пример к главе: \003_FactoryMethod\001_FactoryMethod

Участники

- `Product` - **Продукт**:
  Предоставляет интерфейс для взаимодействия с продуктами.

- `Creator` - **Создатель**:
  Предоставляет интерфейс (абстрактные фабричные методы) для порождения продуктов. В частных случаях класс `Creator` может предоставлять реализацию фабричных методов, которые возвращают экземпляры продуктов (`ConcreteProduct`).

- `ConcreteProduct` - **Конкретный продукт**:
  Реализует интерфейс предоставляемый базовым классом `Product`.

- `ConcreteCreator` - **Конкретная фабрика**:
  Реализует интерфейс (фабричные методы) предоставляемый базовым классом `Creator`.

Отношения между участниками

Отношения между классами

- Класс `ConcreteProduct` связан связью отношения наследования с абстрактным классом `Product`.

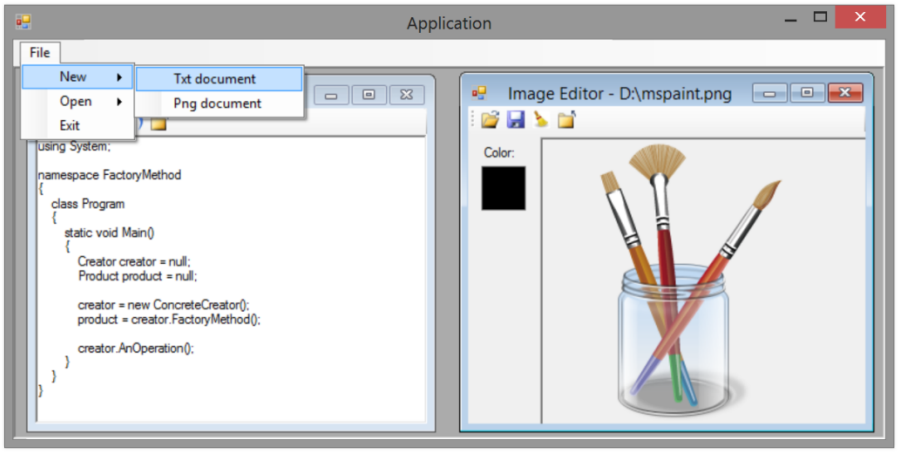
- Класс `ConcreteCreator` связан связью отношения наследования с абстрактным классом `Creator` и связью отношения зависимости с классом порождаемого продукта `ConcreteProduct`.

Отношения между объектами

- Класс `Creator` предоставляет своим производным классам `ConcreteCreator` возможность самостоятельно выбрать вид создаваемого продукта, посредством реализации метода `FactoryMethod`.

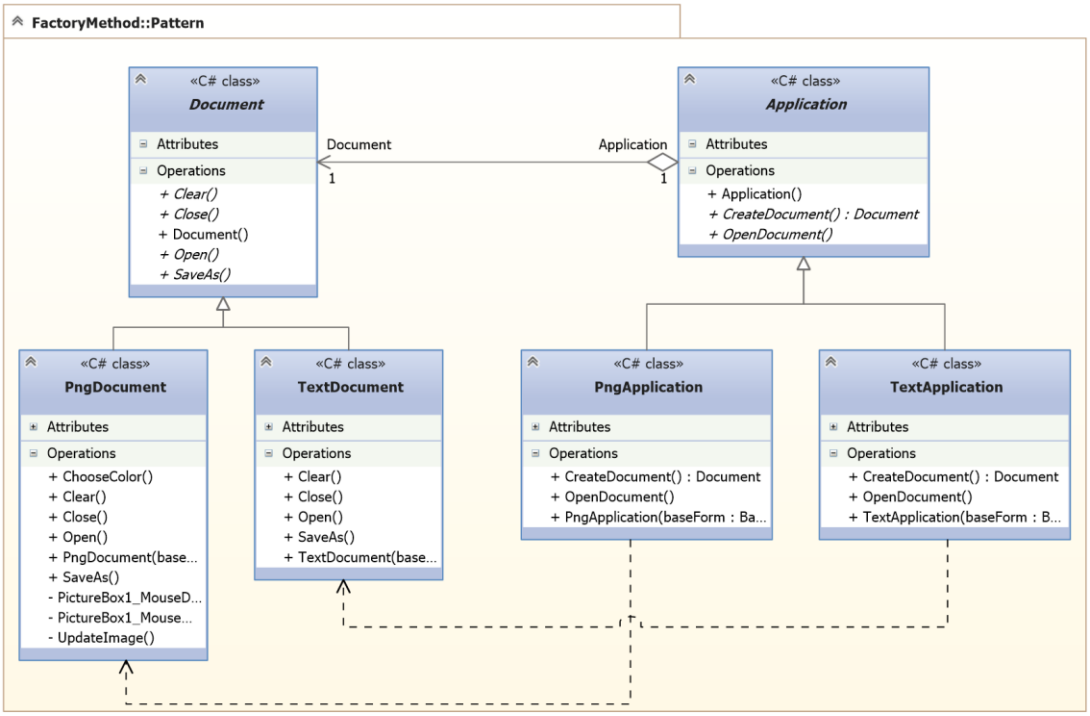
Мотивация

Предлагается рассмотреть простейший каркас (Framework) предоставляющий набор функциональности для построения приложений способных работать с несколькими документами. В таком фреймворке есть смысл выделить две основных абстракции – абстрактные классы `Document` и `Application`.



Чтобы создать приложение, которое могло бы работать как с графикой, так и с текстом, предлагается создать классы `Application` и `Document`. Класс `Application` отвечает за создание и открытие документов, а то с каким документом будет производиться работа определяется его подклассами `PngApplication` и `TextApplication`.

Таким образом можно предоставить пользователю возможность пользоваться пунктами меню: **File.Open** и **File.New** для открытия и создания файлов с расширениями ***.png** и ***.txt**. Файлы, с которыми сможет работать приложение представлены подклассами `PngDocument` и `TextDocument` класса `Document`. Решение с каким документом будет производиться работа принимается пользователем и абстрактный класс `Application` не может заранее спрогнозировать, экземпляр какого конкретного класса-документа (`PngDocument` или `TextDocument`) потребуется создать. Паттерн Factory Method предлагает элегантное решение такой задачи. Подклассы класса `Application` реализуют абстрактный фабричный метод `CreateDocument`, таким образом, чтобы он возвращал ссылку на экземпляр класса требуемого документа (`PngDocument` или `TextDocument`). Как только создан экземпляр одного из классов `PngApplication` или `TextApplication`, этот экземпляр можно использовать для создания документа определенного типа через вызов на этом экземпляре реализации фабричного метода `CreateDocument`. Метод `CreateDocument` называется «фабричным методом» или «виртуальным конструктором», так как он отвечает за непосредственное изготовление объекта-продукта.



См. Пример к главе: \003_FactoryMethod\002_Documents

Применимость паттерна

Паттерн Фабричный Метод рекомендуется использовать, когда:

- Абстрактному базовому классу `Creator` заранее неизвестно, экземпляры каких конкретных классов-продуктов потребуется создать.
- Абстрактный класс `Creator` спроектирован таким образом, чтобы объекты-продукты, которые потребуется создать, описывались производными от него классами (`ConcreteCreator`).
- Процесс создания продукта должен обеспечивать возможность получения различных вариаций создаваемого продукта.
- Абстрактный класс `Creator` планирует передать ответственность за создание объектов-продуктов одному из своих подклассов.
- Требуется собрать в одном месте (в группе наследников) всех `ConcreteCreator`s ответственных за создание объектов-продуктов определенного типа.

Результаты

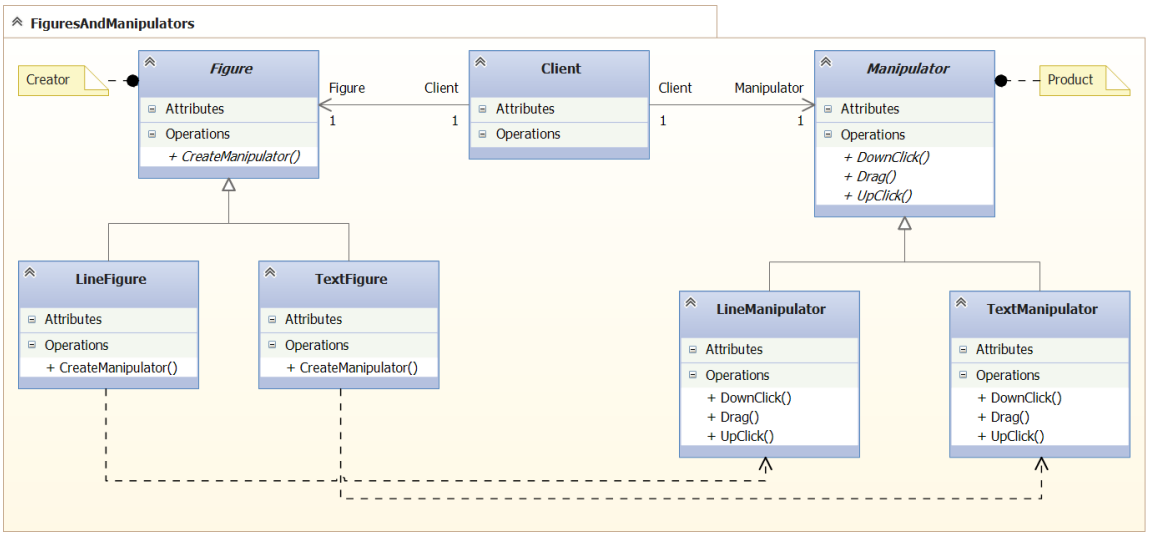
Использование фабричных методов избавляет от необходимости использования конструкторов экземпляров для создания объектов-продуктов непосредственно в месте использования этих объектов-продуктов. Таким образом имеется возможность работать с абстрактным интерфейсом класса `Product`, что в свою очередь позволяет работать с любыми продуктами конкретных классов `ConcreteProduct` производных от `Product`.

Особенности применения паттерна Factory Method:

- **Предоставление производным классам `ConcreteCreator` операций-зацепок (hooks).**
  Создание объектов через использование фабричных методов даже внутри класса оказывается более гибким решением, чем через вызов конструкторов непосредственно. Паттерн Factory Method позволяет использовать в производных классах `ConcreteProduct` операции-зацепки для построения расширенных версий объектов-продуктов.

  См. Пример к главе: \003_FactoryMethod\003_Subclassing Hooks

- **Соединение параллельных иерархий и контроль над ними.**
  Принципы рефакторинга рекомендуют избегать наличия в программном коде параллельных иерархий, но зачастую использование параллельных иерархий может быть хорошим решением. Из этого следует правило: параллельные иерархии в программном коде вполне допустимы в том и только в том случае, если они находятся в контексте паттерна проектирования и их использование регулируется содержащим их паттерном. Неконтролируемое (свободное) использование параллельных иерархий является запахом *(smell)* и требует рефакторинга.
  Параллельные иерархии организуются тогда, когда классу «`A`» требуется возложить часть своих обязанностей на другой класс «`B`» не являющимся производным от класса «`A`».
  Например, имеется приложение, в котором требуется создавать и перемещать графические фигуры при помощи мыши. Приходится сохранять и обновлять информацию о текущем состоянии манипуляций. Но такая информация требуется только в момент перемещения фигуры, поэтому помещать информацию о перемещении в саму фигуру иррационально. В таком случае для каждого типа фигур правильно использовать отдельный объект класса `Manipulator` (манипулятор фигурой). Разные фигуры, будут иметь разные манипуляторы, являющиеся подклассами класса `Manipulator`. Соответствия *«фигура A - манипулятор A»*, *«фигура B - манипулятор B»* и т.д., - представляют собой пример параллельной иерархии. Соответствия *«фигура A - манипулятор A»*, *«фигура B – манипулятор отсутствует»* и т.д., - представляют собой пример частично параллельной иерархии. На диаграмме классов видно, что иерархия классов фигур `Figure` параллельна иерархии классов манипуляторов `Manipulator`.



Важно понимать, что именно фабричные методы (в том числе и в данном примере) определяют соответствия между объектами входящими в параллельные иерархии. Именно фабричные методы содержат механизм порождения необходимых объектов из которых будет составлена параллельная иерархия. Такой подход является надежным и безопасным при эксплуатации параллельных иерархий.

См. Пример к главе: \003_FactoryMethod\004_Figure Manipulators

Реализация

Полезные приемы реализации паттерна Фабричный Метод:

- **Три основных разновидности паттерна.**
  1. Класс `Creator` является абстрактным и содержит только абстрактные фабричные методы. В этом случае требуется создание производных классов в которых будет реализован абстрактный фабричный метод из базового класса.

  2. Класс `Creator` является конкретным классом и содержит реализацию фабричного метода по умолчанию. В этом случае фабричный метод используется главным образом для повышения гибкости. Выполняется правило, которое требует создавать объекты-продукты в фабричном методе, чтобы в производных классах была возможность заместить или переопределить способ создания объектов-продуктов. Такой подход гарантирует, что производным классам будет предоставлена возможность порождения объектов-продуктов требуемых классов.

  3. Класс `Creator` является абстрактным и содержит реализацию фабричного метода по умолчанию.

- **Фабричные методы с параметрами.**
  Допускается создавать фабричные методы принимающие аргументы. Аргумент фабричного метода определяет вид создаваемого объекта-продукта. Переопределение фабричного метода с аргументами, позволит изменять и конфигурировать изготавливаемые продукты.

  См. Пример к главе: \003_FactoryMethod\005_FM_With_Argument

- **Языково-зависимые особенности.**
  Разные языки программирования могут иметь в своем составе свои уникальные конструкции и техники, с использованием которых можно интересным образом выразить идеи использования паттерна – Фабричный Метод.
  В языке C#, фабричные методы могут быть виртуальными или абстрактными (исключительно виртуальными). Нужно осторожно подходить к вызову виртуальных методов в конструкторе класса `Creator`.
  Следует помнить, что в C# невозможно реализовать абстрактный метод базового абстрактного класса как виртуальный в производном классе. Абстрактные методы интерфейсов (`interface`) допустимо реализовывать как виртуальные.
  После переопределения (`override`) виртуального метода в производном классе `ConcreteCreator`, виртуальные методы базового класса `Creator` становятся недоступными для их вызова на экземпляре класса `ConcreteCreator` (неважно, было привидение к базовому типу или нет). Если виртуальный метод вызывается в конструкторе класса `Creator`, а переопределенный (`override`) метод в конструкторе `ConcreteCreator`, то при создании экземпляра класса `ConcreteCreator`, в первую очередь отработает конструктор базового класса `Creator`, в котором произойдет вызов переопределенного метода из производного класса, а невиртуальный метод базового класса `Creator`. В случае замещения виртуального метода такой эффект отсутствует.

  См. Пример к главе: \003_FactoryMethod\006_FM_in_Constructor

  Обойти эту особенность возможно через использование функции доступа `GetProduct`, которая создает продукт по запросу, а с конструктора снять обязанность по созданию продуктов. Функция

доступа возвращает продукт, но сперва проверяет его существование. Если продукт еще не создан, то функция доступа его создает (через вызов фабричного метода). Такую технику часто называют отложенной (или ленивой) инициализацией.

См. Пример к главе: \003_FactoryMethod\007_Lazy Initialization

- **Использование обобщений (Generics).**
  Иногда приходится создавать конкретные классы создателей `ConcreteCreator` производные от базового класса `Creator` только для того чтобы создавать объекты-продукты определенного вида. Чтобы изменить подход порождения продуктов, в языке C#, можно воспользоваться такими конструкциями языка, как обобщения (Generics). Для организации процесса порождения продукта можно использовать технику – Service Locator.

  См. Пример к главе: \003_FactoryMethod\008_ServiceLocator

  При использовании обобщений (Generics), порождать несколько подклассов `ConcreteCreator` от класса `Creator` не потребуется, достаточно при создании экземпляра продукта в качестве параметра-типа фабричного метода `CreateProduct` указать желаемый тип порождаемого продукта.

```
ICreator creator = new StandardCreator();
IProduct productA = creator.CreateProduct<ProductA>();
IProduct productB = creator.CreateProduct<ProductB>();
IProduct productC = creator.CreateProduct<ProductC>();
```

  См. Пример к главе: \003_FactoryMethod\009_FM_Generic

- **Соглашения об именовании.**
  На практике рекомендуется давать такие имена фабричным методам, чтобы можно было легко понять, что используется именно фабричный метод. Например, фабричный метод порождающий документы мог бы иметь имя `CreateDocument`, где в имя метода входит название производимого действия `Create` и название того `Document` что создается.

Пример кода игры «Лабиринт»

Рассмотрим класс MazeGame, который использует фабричные методы. Фабричные методы создают объекты лабиринта: комнаты, стены, двери. В отличие от работы с абстрактной фабрикой, методы: MakeMaze, MakeRoom, MakeWall, MakeDoor – содержатся непосредственно в классе MazeGame.

```
class MazeGame
{
    // Использование Фабричных методов.
    public Maze CreateMaze()
    {
        Maze aMaze = this.MakeMaze();

        Room r1 = MakeRoom(1);
        Room r2 = MakeRoom(2);
        Door theDoor = MakeDoor(r1, r2);

        aMaze.AddRoom(r1);
        aMaze.AddRoom(r2);

        r1.SetSide(Direction.North, MakeWall());
        r1.SetSide(Direction.East, theDoor);
        r1.SetSide(Direction.South, MakeWall());
        r1.SetSide(Direction.West, MakeWall());

        r2.SetSide(Direction.North, MakeWall());
        r2.SetSide(Direction.East, MakeWall());
        r2.SetSide(Direction.South, MakeWall());
        r2.SetSide(Direction.West, theDoor);

        return aMaze;
    }

    public virtual Maze MakeMaze()
    {
        return new Maze();
    }

    public virtual Room MakeRoom(int number)
    {
        return new Room(number);
    }

    public virtual Wall MakeWall()
    {
        return new Wall();
    }

    public virtual Door MakeDoor(Room r1, Room r2)
    {
        return new Door(r1, r2);
    }
}
```

Для того чтобы сделать игру более разнообразной можно ввести специальные варианты частей лабиринта (EnchantedRoom – волшебная комната, DoorNeedingSpell –

дверь, требующая заклинания, RoomWithBomb – комната с бомбой, BombedWall – взорванная стена).

```
    // Класс заклинания необходимый для
    // функционирования лабиринта с заклинаниями.
    class Spell
    {
        public Spell()
        {
            Console.WriteLine("Заклинание...");
        }
    }

    // Класс волшебная комната.
    class EnchantedRoom : Room
    {
        // Поля.
        private Spell spell = null;

        // Конструкторы.

        public EnchantedRoom(int roomNo)
            : base(roomNo)
        {
        }

        public EnchantedRoom(int number, Spell spell)
            : base(number)
        {
            this.spell = spell;
        }
    }

    // Класс двери для которой требуется заклинание.
    class DoorNeedingSpell : Door
    {
        // Конструктор.
        public DoorNeedingSpell(Room room1, Room room2)
            : base(room1, room2)
        {
        }
    }

    // Класс комнаты с бомбой.
    class RoomWithBomb : Room
    {
        // Конструктор.
        public RoomWithBomb(int roomNo)
            : base(roomNo)
        {

        }
    }

    // Класс взорванной стены.
    class BombedWall : Wall
    {
    }
```

Подклассы EnchantedMazeGame и BombedMazeGame класса MazeGame, скрывают работу с такими специфическими классами как: EnchantedRoom, DoorNeedingSpell, RoomWithBomb, BombedWall. Использование фабричных методов позволяет в подклассах класса MazeGame:

EnchantedMazeGame, BombedMazeGame – выбирать различные варианты объектов-продуктов для построения лабиринта.

```
class EnchantedMazeGame : MazeGame
{
    // Конструктор лабиринта с заклинаниями.
    public EnchantedMazeGame()
    {
    }

    // Методы.
    public override Room MakeRoom(int number)
    {
        return new EnchantedRoom(number, this.CastSpell());
    }

    public override Door MakeDoor(Room r1, Room r2)
    {
        return new DoorNeedingSpell(r1, r2);
    }

    // Метод создания заклинания.
    protected Spell CastSpell()
    {
        return new Spell();
    }
}

class BombedMazeGame : MazeGame
{
    // Конструктор лабиринта с бомбами.
    public BombedMazeGame()
    {
    }

    // Методы.
    public override Wall MakeWall()
    {
        return new BombedWall();
    }

    public override Room MakeRoom(int number)
    {
        return new RoomWithBomb(number);
    }
}
```

Известные применения паттерна в .Net

Паттерн Factory Method лежит в основе всех порождающих паттернов, соответственно он используется везде где можно увидеть применение порождающих паттернов.

См. пункты «Известные применения паттерна в .Net» других порождающих паттернов.

Паттерн Prototype

Название

Прототип

Также известен как

-

Классификация

По цели: порождающий
По применимости: к объектам

Частота использования

Средняя - 1 2 **3** 4 5

Назначение

Паттерн Prototype – предоставляет возможность создания новых объектов-продуктов (клонов), используя технику клонирования (копирования) созданного ранее объекта-оригинала-продукта (прототипа). Паттерн Prototype – позволяет задать различные виды (классы-виды) объектов-продуктов (клонов), через настройку состояния каждого нового созданного клона. Классификация клонов-продуктов производится на основании различия их состояний.

Введение

Паттерн Prototype описывает процесс правильного создания объектов-клонов на основе имеющегося объекта-прототипа, другими словами, паттерн Prototype описывает правильные способы организации процесса клонирования.

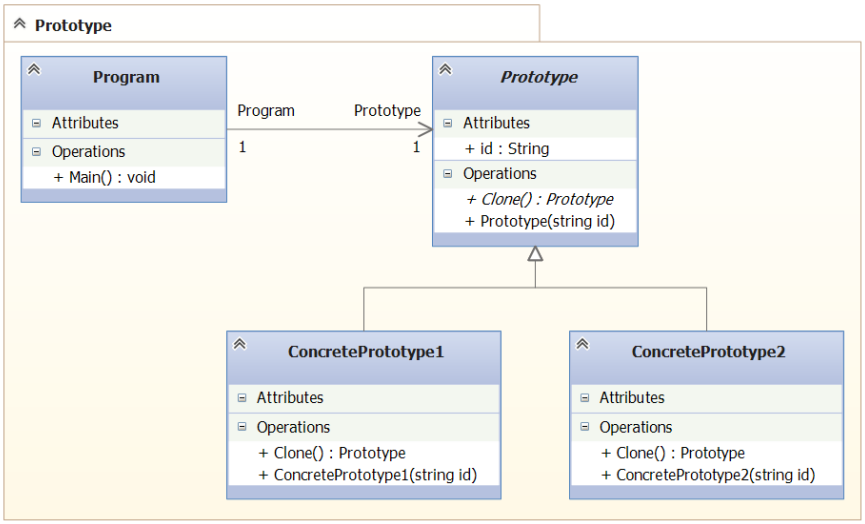
Что такое клонирование в объективной реальности? В биологии термин «клонирование» - обозначает процессы копирования любых живых существ. Но, в информатике термин «клонирование» имеет свое специфическое значение, отличное от его значения в биологии. В биологии процесс клонирования происходит путем создания не готовой копии взрослого организма, а путем выращивания взрослого клона из младенца. Изначально клон-младенец порождается на основе генетического материала организма-прототипа, и развивается до взрослого состояния поэтапно, согласно заложенной в нем генетической программы развития. В информатике же клонирование происходит «мгновенно». В результате клонирования прототипа в информатике, получается сразу «взрослый» клон полностью идентичный прототипу.

В жизни программному клонированию аналогии не найти, так как биологический организм-клон никогда не будет идентичен своему организму-прототипу. Нарушение идентичности происходит за счет вмешательства в развитие клона внешних факторов среды его обитания. С потерей идентичности у биологического клона развивается ярко выраженная индивидуальность. В информатике же, легко добиться идентичности клона и прототипа за счет возможности копирования всего состояния прототипа в клон.

Но можно отойти от подхода к клонированию со строгим контролем идентичности и представить клона, как систему, созданную по образцу другой. Клон должен сохранять основные свойства исходной системы-прототипа не разрушая абстракции (генотипа) прототипа, то есть собирательного понятия исходной системы. Индивидуальные черты (фенотип), которые в процессе развития приобретает клон, не должны разрушать абстракцию (генотип) прототипа и такими индивидуальными чертами клона можно пренебречь.

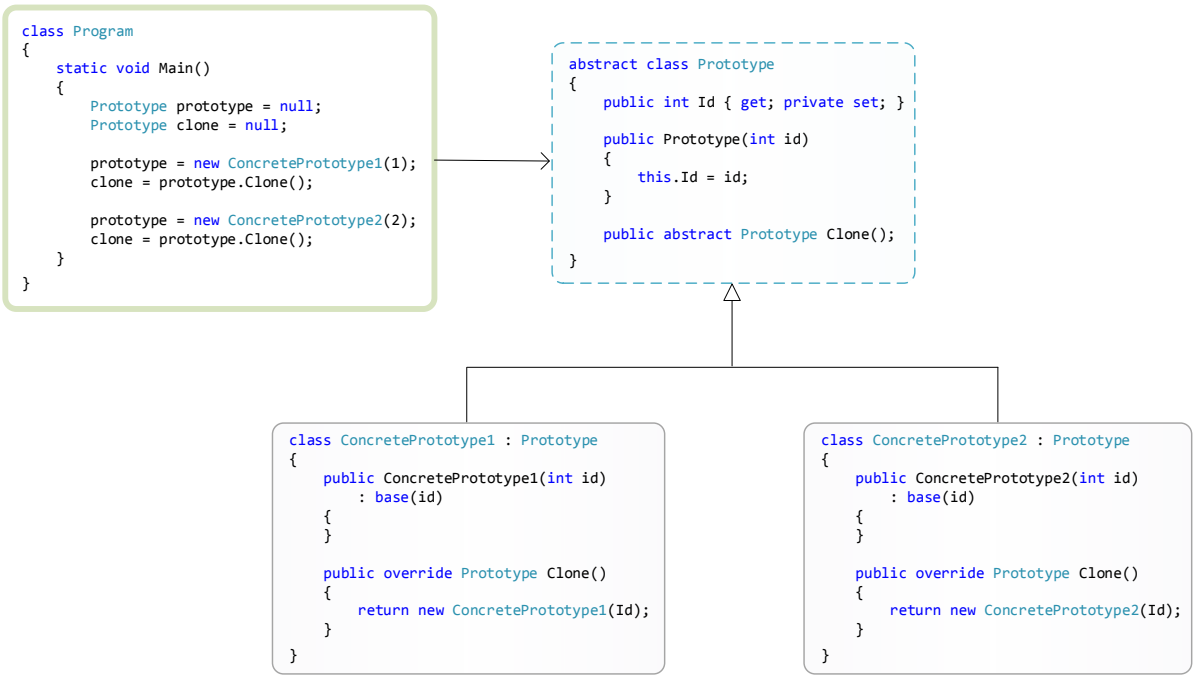
Например, при клонировании овцы Долли, сама Долли и овца-донор имели стопроцентную идентичность генотипа, но не имели стопроцентной идентичности фенотипа. Другими словами, и донор-овца и клон-овца были овцами-близнецами одной породы (сохранение генотипа), а не овцами разных пород, но у них мог отличаться вес, рост, оттенок шерсти, а также приобретенный опыт определяющий индивидуальное поведение (разность фенотипов).

Структура паттерна на языке UML



См. Пример к главе: \004_Prototype\001_Prototype

Структура паттерна на языке C#



```
class Program
{
    static void Main()
    {
        Prototype prototype = null;
        Prototype clone = null;

        prototype = new ConcretePrototype1(1);
        clone = prototype.Clone();

        prototype = new ConcretePrototype2(2);
        clone = prototype.Clone();
    }
}
```

```
abstract class Prototype
{
    public int Id { get; private set; }

    public Prototype(int id)
    {
        this.Id = id;
    }

    public abstract Prototype Clone();
}
```

```
class ConcretePrototype1 : Prototype
{
    public ConcretePrototype1(int id)
        : base(id)
    {
    }

    public override Prototype Clone()
    {
        return new ConcretePrototype1(Id);
    }
}
```

```
class ConcretePrototype2 : Prototype
{
    public ConcretePrototype2(int id)
        : base(id)
    {
    }

    public override Prototype Clone()
    {
        return new ConcretePrototype2(Id);
    }
}
```

См. Пример к главе: \004_Prototype\001_Prototype

Участники

- `Prototype` - **Прототип**:
  Предоставляет интерфейс для клонирования себя.

- `ConcretePrototype` - **Конкретный прототип**:
  Реализует операцию клонирования себя.

- `Client` - **Клиент**:
  Клиент создает экземпляр прототипа. Вызывает на прототипе метод клонирования.

Отношения между участниками

Отношения между классами

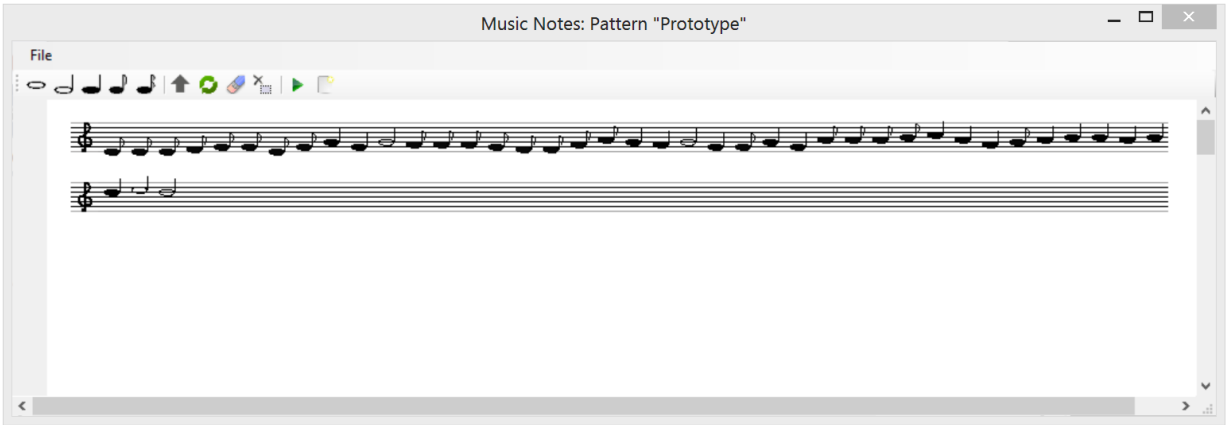
- Класс `ConcretePrototype` связан связью отношения наследования с абстрактным классом `Prototype`.

Отношения между объектами

- Клиент вызывает на экземпляре-прототипе метод `Clone` и этот метод создает экземпляр-клон прототипа и возвращает ссылку на него.

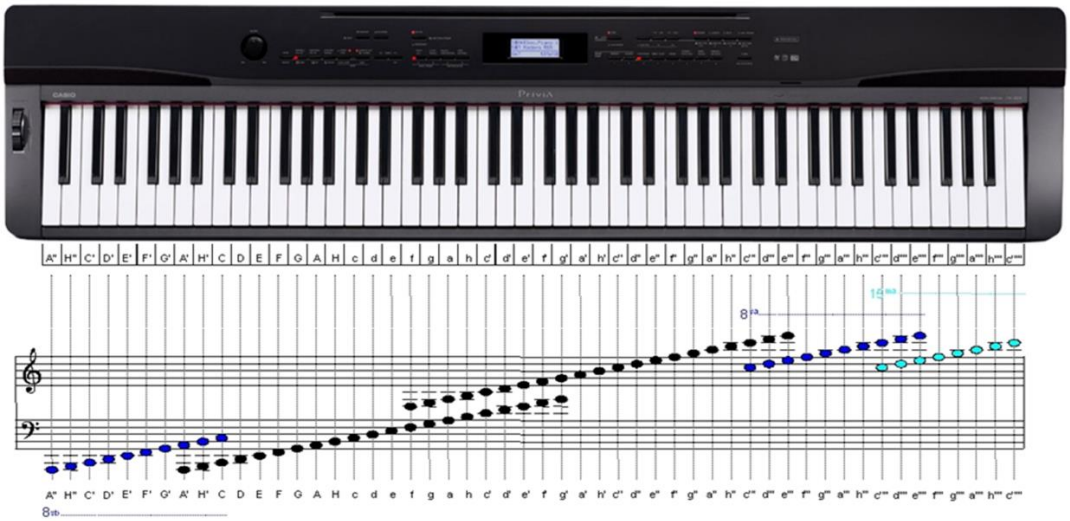
Мотивация

Предлагается построить простейший нотный редактор, позволяющий набирать, редактировать и проигрывать нотный текст. Редактор должен содержать партитуру, нотные станы, ноты и другие музыкальные объекты. Ввод нот может происходить при помощи мыши путем выбора ноты в палитре нот и перемещения выбранной ноты на нотный стан.



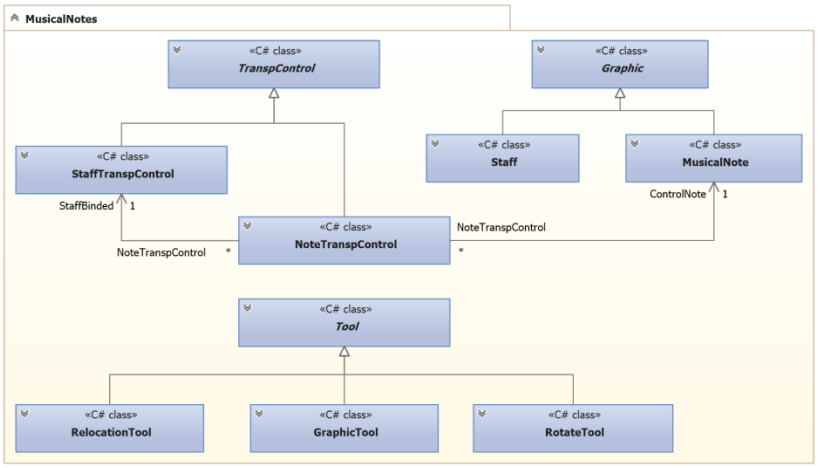
Нота представляет собой графическое изображение звука. Нотная гамма содержит семь нот: «до», «ре», «ми», «фа», «соль», «ля», «си». Ноты распределяются по октавам.

Например, у пианино девять октав: семь полных и две неполных. В каждой октаве семь белых клавиш и пять черных, итого двенадцать клавиш в октаве. Каждая нота имеет длительность звучания: 1, 1/2, 1/4, 1/8, 1/16, 1/32 и 1/64. Соответственно с использованием пианино можно воспроизвести очень большое количество вариаций звуков.



Таким образом может показаться, что потребуется создать по одному классу (`class`) для каждой вариации воспроизводимого звука. Но такой подход является нерациональным и неприемлемым. Более правильным решением проблемы - будет создание ноты-прототипа для ее последующего клонирования с возможностью настройки звучания (свойств) каждой ноты-клона.

Для того чтобы создать приложение с графическим интерфейсом, которое позволяло бы создавать и редактировать музыкальные композиции в наглядной форме, предлагается создать абстрактные классы `Graphic` и `Tool`. Класс `Graphic` служит для графического представления нот и нотных станов на форме приложения, а класс `Tool` определяет функциональные возможности для работы с графическими представлениями, такие как создание графического представления каждой ноты в зависимости от характеристик ноты (реализуется конкретным подклассом `GraphicTool` класса `Tool`), размещение его на нотном стане (реализуется конкретным подклассом `RelocationTool` класса `Tool`) и ориентирование штиля ноты (реализуется подклассом `RotateTool` класса `Tool`).



Чтобы обеспечить гибкость и рациональность разработки приложения, класс `GraphicTool` будет создавать экземпляры классов `MusicalNote` (класс ноты) и `Staff` (класс музыкального стана), которые являются наследниками абстрактного класса `Graphic` и реализуют метод `Clone` базового класса путем клонирования прототипов экземпляров этих классов, переданных в качестве аргумента конструктора класса, и затем добавлять экземпляры-клоны в партитуру.

См. Пример к главе: \004_Prototype\002_MusicalEditor

Применимость паттерна

Паттерн Prototype рекомендуется использовать, когда:

- Программист не должен знать, как в системе создаются, компонуются и представляются объекты-продукты.
- Классы, экземпляры которых требуется создать, определяются во время выполнения программы, например, с использованием техники позднего связывания (динамическая загрузка DLL).
- Требуется избежать построения параллельных иерархий классов (фабрик или продуктов).
- Экземпляры определенного класса могут иметь небольшое количество состояний. Тогда может оказаться удобнее установить прототип в соответствие каждому уникальному состоянию и в дальнейшем клонировать прототипы, а не создавать экземпляры вручную и не настраивать состояние этих экземпляров каждый раз заново.
- Требуется создать некоторое количество экземпляров с одинаковым долго-вычисляемым состоянием. В этом случае для повышения производительности удобнее создать прототип и его клонировать, причем с использованием метода `MemberwiseClone` класса `System.Object` входящего в поставку Microsoft .NET Framework. Клонирование с использованием конструктора повышения производительности не даст.

  См. Пример к главе: \004_Prototype\003_Performance

Результаты

Многие особенности применения паттерна Prototype совпадают с особенностями применения паттернов Abstract Factory и Builder:

- Сокрытие работы с конкретными классами продуктов.
- Возможность легкой замены семейств используемых продуктов.
- Обеспечение совместного использования продуктов.
- Имеется небольшое неудобство добавления нового вида продуктов.
- Возможность изменения состава продукта.
- Сокрытие кода, реализующего конструирование и представление.
- Контроль над процессом построения продукта.

Дополнительные особенности результатов применения паттерна Prototype:

- **Возможность добавления и удаления продуктов во время выполнения.**
  Паттерн Prototype позволяет добавлять новую разновидность «классов-продуктов» во время выполнения программы за счет конфигурирования состояния объекта-клона полученного в результате клонирования объекта-прототипа. «Класс-продукт» следует понимать не как конструкцию языка C# (`class`), а как «вид-продукт» получившийся за счет конфигурации состояния объекта-клона.

  См. Пример к главе: \004_Prototype\004_ObjectClass

- **Описание новых типов объектов (объектов-классов) путем изменения состояния клонов.**
  Программные системы позволяют формировать состояние и поведение сложных объектов-клонов за счет композиции состояний и поведений более простых объектов. При этом для представления сложного состояния объекта, не требуется создавать новых классов (`class`).

- **Уменьшение числа производных классов.**
  Большинство порождающих паттернов используют иерархии классов продуктов параллельные иерархиям классов фабрик. Паттерн Prototype через клонирование и настройку объектов-продуктов позволяет избавиться от наличия иерархии фабрик, порождающих продукты.

- **Динамическое добавление классов во время выполнения программы.**
  Платформа .Net позволяет динамически подключать новые классы к исполняющемуся приложению и создавать экземпляры этих классов (late binding - позднее связывание). Приложение, создающее экземпляры динамически загружаемого класса, не имеет возможности вызывать конструктор напрямую. Вместо этого на классе-объекте `Activator` вызывается метод `CreateInstance`, которому в качестве аргумента передается строковое представление полного квалифицированного имени класса, экземпляр которого требуется создать, например, `Activator.CreateInstance("MyNamespace.MyClass").` Паттерн Prototype в определенных случаях может стать гибкой альтернативой позднему связыванию избавляя от многократного инстанцирования классов, подключаемых динамически. Достаточно создать один экземпляр-прототип и в дальнейшем его клонировать. Такой подход может дать выигрыш в производительности при необходимости создания большого количества однотипных экземпляров классов.

При клонировании могут возникнуть проблемы корректного клонирования ассоциаций. Особое внимание требуется обращать на клонирование двухсторонних ассоциаций.

Реализация

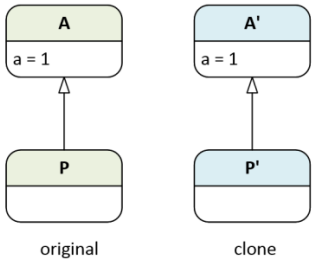
Использование паттерна Prototype может оказаться полезным в статически типизированных языках вроде C#, где классы не являются объектами. Меньше всего паттерн Prototype может оказаться полезным в прототипно-ориентированных языках вроде JavaScript, так как в основе таких языков уже заложена конструкция эквивалентная прототипу, это – «объект-класс». В прототипно-ориентированных языках создание любого объекта производится путем клонирования прототипа и такие языки базируются именно на идеях использования паттерна Prototype.

Вопросы, которые могут возникнуть при реализации прототипов:

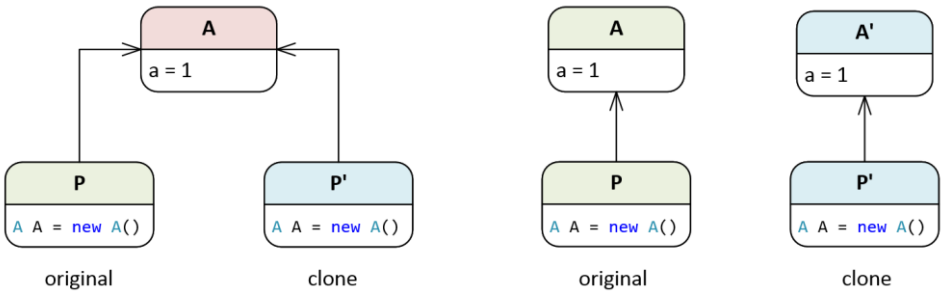
- **Использование диспетчера прототипов.**
  Если в программе требуется создавать большое число прототипов и при этом заранее не известно сколько прототипов может понадобиться, то есть смысл использовать вспомогательный объект для создания прототипов и контроля над ними. Такой объект будет представлять собой реестр прототипов или его можно назвать - «диспетчер прототипов». Реестр прототипов – представляет собой ассоциативную коллекцию типа хеш-таблицы или словаря, в которой в соответствие ключам поставлены объекты-прототипы. Реестр может возвращать ссылку на прототип или непосредственно на построенный клон прототипа, это зависит от желаемой логики.

- **Реализация метода `Clone`.**
  Самый ответственный момент при использовании паттерна Prototype – реализация метода `Clone`. Особое внимание потребуется при наличии ассоциаций в клонируемом прототипе.
  Паттерн Prototype, выражен в платформе .Net в виде техники клонирования, через использование реализации интерфейса `ICloneable` или метода `MemberwiseClone` класса `System.Object`. Но метод `MemberwiseClone` не решает проблему «глубокого и поверхностного клонирования». При клонировании с использованием метода `MemberwiseClone` следует учитывать следующие особенности:

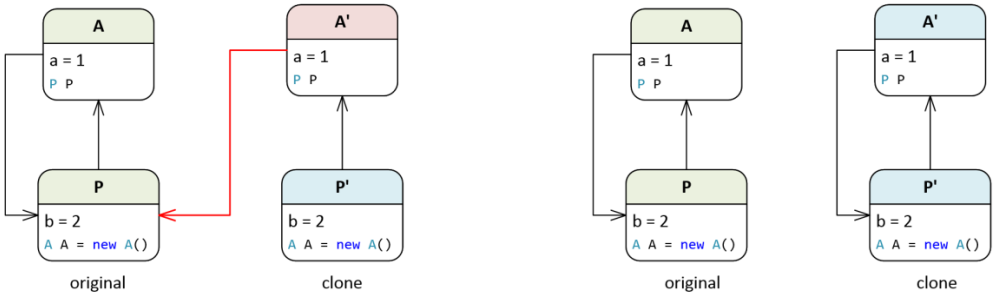
  Граф наследования всегда клонируется глубоко.



Могут возникнуть проблемы корректного клонирования ассоциаций. Так как ассоциации клонируются поверхностно. Перед клонированием следует ответить на вопрос: должны ли при клонировании объекта клонироваться также и другие объекты на которые ссылаются переменные клонируемого объекта, или клон будет использовать эти объекты совместно с оригиналом?



Особо внимание требуется обратить на клонирование двухсторонних ассоциаций.



См. Пример к главе: \004_Prototype\005_Features

- **Инициализация клонов.**
  Часто, сразу после клонирования, требуется производить настройку (изменение) состояния объекта-клона (полностью или частично). Можно задуматься над тем, чтобы передавать параметры в метод `Clone` перед началом клонирования, но такой подход считается неудачным, так как мешает построению единообразного интерфейса клонирования. Для таких случаев, будет целесообразно ввести набор методов для установки или очистки важных элементов состояния, или ввести один метод `Initialize` который будет принимать параметры для установки состояния клона.

Пример кода игры «Лабиринт»

Определим класс `MazePrototypeFactory`, производный от класса `MazeFactory`, который был создан при рассмотрении паттерна Abstract Factory. Конструктор класса `MazePrototypeFactory` в качестве параметров будет принимать прототипы объектов классов `Maze`, `Wall`, `Room` и `Door`, из которых состоит лабиринт.

Класс `MazePrototypeFactory` будет создавать готовые объекты на основе соответствующих прототипов путем клонирования этих прототипов, для этого переопределим в нем фабричные методы `MakeMaze`, `MakeRoom`, `MakeWall` и `MakeDoor` из базового класса `MazeFactory`.

```
class MazePrototypeFactory : MazeFactory
{
    // Поля.
    Maze prototypeMaze = null;
    Room prototypeRoom = null;
    Wall prototypeWall = null;
    Door prototypeDoor = null;

    // Конструктор.
    public MazePrototypeFactory(Maze maze, Wall wall, Room room, Door door)
    {
        this.prototypeMaze = maze;
        this.prototypeWall = wall;
        this.prototypeRoom = room;
        this.prototypeDoor = door;
    }

    // Методы.

    public override Maze MakeMaze()
    {
        return prototypeMaze.Clone();
    }

    public override Room MakeRoom(int number)
    {
        Room room = prototypeRoom.Clone();
        room.Initialize(number);

        return room;
    }

    public override Wall MakeWall()
    {
        // Клонирование.
        return prototypeWall.Clone();
    }

    public override Door MakeDoor(Room room1, Room room2)
    {
        Door door = prototypeDoor.Clone();
        door.Initialize(room1, room2);

        return door;
    }
}
```

Для создания экземпляра фабрики, позволяющей построить простой лабиринт, достаточно сконфигурировать объект класса MazePrototypeFactory объектами классов Maze, Wall, Room и Door, составляющими простой лабиринт. Для создания лабиринта требуется фабрику класса MazePrototypeFactory, передать в качестве аргумента метода CreateMaze класса MazeGame, который был создан при рассмотрении паттерна Abstract Factory.

```
// Создаем генератор лабиринта.
MazeGame game = new MazeGame();

//Конфигурируем фабрику базовыми элементами лабиринта
MazePrototypeFactory simpleMazeFactory =
     new MazePrototypeFactory(new Maze(), new Wall(), new Room(), new Door());

// Создаем лабиринт из двух комнат используя фабричный метод - CreateMaze().
Maze maze = game.CreateMaze(simpleMazeFactory);
```

Поскольку экземпляр класса MazePrototypeFactory может быть сконфигурирован экземплярами классов производных от классов Maze, Wall, Room и Door, то отсутствует необходимость порождать новые классы производные от класса MazePrototypeFactory для создания специальных видов лабиринтов.

Для того чтобы изменить тип простого лабиринта на лабиринт с бомбами требуется организовать следующий вызов:

```
// Конфигурируем фабрику специальными элементами лабиринта
MazePrototypeFactory bombedMazeFactory = new MazePrototypeFactory(new Maze(),
new BombedWall(), new RoomWithBomb(), new Door());
```

Объекты классов типов Maze, Wall, Room и Door, которые предполагается использовать в качестве прототипов, должны содержать в себе реализацию метода Clone.

```
class Door : MapSite
{
    // Поля.
    Room room1 = null;
    Room room2 = null;
    bool isOpen;

    // Конструкторы.
    public Door(){}

    public Door(Door other)
    {
        this.room1 = other.room1;
        this.room2 = other.room2;
    }

    public Door(Room room1, Room room2)
    {
        this.room1 = room1;
        this.room2 = room2;
    }

    // Методы.
    // Отображает дверь.
    public override void Enter()
    {
        Console.WriteLine("Door");
    }


    // Метод возвращает ссылку на другую комнату.
```

```
    public Room OtherSideFrom(Room room)
    {
        if (room == room1)  // Если идем из r1 в r2, то возвращаем ссылку на r2.
            return room2;
        else          // Иначе: Если идем из r2 в r1, то возвращаем ссылку на r1.
            return room1;
    }

    // Клонирование.
    public virtual Door Clone()
    {
        Door door = new Door(this.room1, this.room2);
        // При обращении к закрытым членам экземпляра, в пределах того-же класса
        // инкапсуляция не работает.
        door.isOpen = this.isOpen;
        return door;
    }

    // Метод инициализации...
    public virtual void Initialize(Room room1, Room room2)
    {
        this.room1 = room1;
        this.room2 = room2;
    }
}
```

В подклассах классов `Maze`, `Wall`, `Room` и `Door`, нужно переопределить метод `Clone`.

```
class BombedWall : Wall
{
    // Поля.
    bool bomb;

    // Конструкторы.
    public BombedWall(){}

    public BombedWall(BombedWall other)
    {
        this.bomb = other.bomb;
    }

    // Переопределение базовой операции клонирования.
    public override Wall Clone()
    {
        return new BombedWall(this);
    }

    public bool HasBomb()
    {
        return this.bomb;
    }
}
```

При таком определении метода `Clone` в базовых классах `Maze`, `Wall`, `Room` и `Door`, клиент избавляется от необходимости знать о конкретных реализациях операции клонирования в подклассах этих классов, т.к. клиенту никогда не придется приводить значение, возвращаемое `Clone`, к базовому типу.

Известные применения паттерна в .Net

`System.Data.DataSet`
http://msdn.microsoft.com/ru-ru/library/system.data.dataset.aspx

`System.Array`
http://msdn.microsoft.com/ru-ru/library/system.array.aspx

`System.Collections.ArrayList`
http://msdn.microsoft.com/ru-ru/library/system.collections.arraylist.aspx

`System.Collections.BitArray`
http://msdn.microsoft.com/ru-ru/library/system.collections.bitarray.aspx

`System.Collections.Hashtable`
http://msdn.microsoft.com/ru-ru/library/system.collections.hashtable.aspx

`System.Collections.Queue`
http://msdn.microsoft.com/ru-ru/library/system.collections.queue.aspx

`System.Collections.SortedList`
http://msdn.microsoft.com/ru-ru/library/system.collections.sortedlist.aspx

`System.Collections.Stack`
http://msdn.microsoft.com/ru-ru/library/system.collections.stack.aspx

`System.Data.DataTable`
http://msdn.microsoft.com/ru-ru/library/system.data.datatable.aspx

`System.Data.SqlClient.SqlCommand`
http://msdn.microsoft.com/ru-ru/library/system.data.sqlclient.sqlcommand(v=vs.100).aspx

`System.Delegate`
http://msdn.microsoft.com/ru-ru/library/system.delegate.aspx

`System.Reflection.AssemblyName`
http://msdn.microsoft.com/ru-ru/library/system.reflection.assemblyname.aspx

`System.Text.Encoding`
http://msdn.microsoft.com/ru-ru/library/system.text.encoding(v=vs.90).aspx

`System.Xml.XmlNode`
http://msdn.microsoft.com/ru-ru/library/system.xml.xmlnode.aspx

`System.Globalization.Calendar`
http://msdn.microsoft.com/ru-ru/library/system.globalization.calendar.aspx

`System.Globalization.DateTimeFormatInfo`
http://msdn.microsoft.com/ru-ru/library/system.globalization.datetimeformatinfo(v=vs.110).aspx

`System.Globalization.TextInfo`
http://msdn.microsoft.com/ru-ru/library/system.globalization.textinfo.aspx

`System.String`
http://msdn.microsoft.com/ru-ru/library/system.string.aspx

И т.д.

Паттерн Singleton

Название

Одиночка

Также известен как

Solitaire (Холостяк)

Классификация

По цели: порождающий
По применимости: к объектам

Частота использования

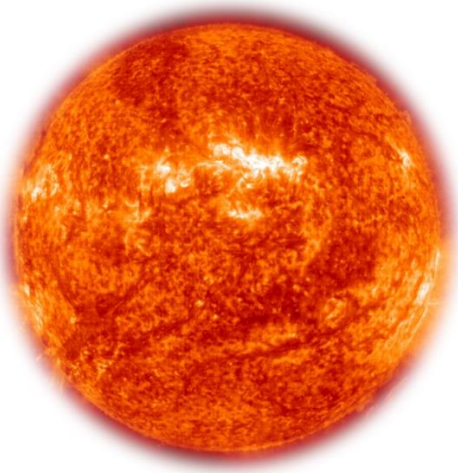
Выше средней - 1 2 3 **4** 5

Назначение

Паттерн Singleton - гарантирует, что у класса может быть только один экземпляр. В частном случае предоставляется возможность наличия, заранее определенного числа экземпляров.

Введение

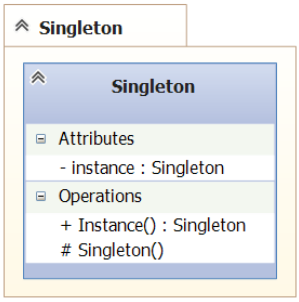
В реальной жизни аналогией объекта в единственном экземпляре может служить Солнце. Как бы мы не смотрели на него, мы всегда будем видеть одно и то же Солнце – звезду, вокруг которой вращается планета Земля, так как не может быть другого экземпляра Солнца, по крайней мере, для нашей планеты.



Важно заметить, что аналогия с Солнцем не является полной в отношении паттерна Singleton. Так как ни один объект в реальной жизни не может контролировать процесс своего порождения, а, следовательно, и существования себя в единственном экземпляре. Такая возможность появляется только у виртуальных объектов программных систем.

При разработке приложений бывает необходимо, чтобы существовал только один экземпляр определенного класса. Например, в играх от первого лица персонаж-анимат, которым управляет игрок, должен быть одним единственным, так как такой анимат, является образом-проекцией, живого игрока на игровой мир.

Структура паттерна на языке UML



См. Пример к главе: \005_Singleton\001_Singleton

Структура паттерна на языке C#

```
class Program
{
    static void Main()
    {
        Singleton instance1 = Singleton.Instance();
        Singleton instance2 = Singleton.Instance();
        Console.WriteLine(ReferenceEquals(instance1, instance2));

        instance1.SingletonOperation();
        string singletonData = instance1.GetSingletonData();
        Console.WriteLine(singletonData);
    }
}
```

```
class Singleton
{
    static Singleton uniqueInstance;
    string singletonData = string.Empty;

    protected Singleton()
    {
    }

    public static Singleton Instance()
    {
        if (uniqueInstance == null)
            uniqueInstance = new Singleton();

        return uniqueInstance;
    }

    public void SingletonOperation()
    {
        singletonData = "SingletonData";
    }

    public string GetSingletonData()
    {
        return singletonData;
    }
}
```

См. Пример к главе: \005_Singleton\001_Singleton

Участники

- `Singleton` - **Одиночка**:
  Предоставляет интерфейс (метод `Instance`) для получения доступа к единственному экземпляру.

Отношения между участниками

Отношения между классами

- Класс `Singleton` не имеет обязательных связей отношений.

Отношения между объектами

- Клиент получает доступ к экземпляру через вызов метода `Instance` на классе-объекте `Singleton`.

Мотивация

Для некоторых классов требуется, чтобы у них мог существовать только один экземпляр их класса. Например, класс `Console`, который представляет собой объектно-ориентированное представление сущности – консоль, из объективной реальности. В объективной реальности консоль, это собирательное понятие, объединяющее два устройства – клавиатуру и экран монитора (устройства ввода/вывода информации).

Класс `Console`, является статическим классом, а это значит, что невозможно создать переменное количество его экземпляров. Поэтому можно сказать, что статические классы в C# являются одним из возможных выражений паттерна Singleton.

Применимость паттерна

Паттерн Singleton рекомендуется использовать, когда:

- В системе должен быть только один экземпляр некоторого класса, или в частном случае заранее определенное пользователем количество экземпляров (два, три и т.д.).

- Требуется организовать расширение класса единственного экземпляра через использование механизма наследования.

Результаты

Особенности применения паттерна Singleton:

- **Контроль доступа к единственному экземпляру.**
  Процесс создания единственного экземпляра скрыт в классе `Singleton`, поэтому класс `Singleton` полностью контролирует доступ к экземпляру через использование метода Instance, который всегда возвращает ссылку на один и тот же экземпляр.

- **Возможность расширения через наследование.**
  Если класс `Singleton` не является статическим или герметизированным / запечатанным (`sealed`), то от него возможно наследование, что позволит расширить существующую функциональность.

- **Возможность наличия переменного числа экземпляров.**
  Паттерн Singleton позволяет создавать фиксированное число экземпляров класса `Singleton`, например, только два или три и т.д.

  См. Пример к главе: \005_Singleton\002_MultySingleton

- **Большая гибкость чем у статических классов.**
  Одним из вариантов реализации паттерна Singleton в C#, является использование статических классов. Но такой подход может в дальнейшем препятствовать изменению дизайна в том случае, если понадобится использование нескольких экземпляров класса `Singleton`. Кроме того, статические классы не сопрягаются с механизмами наследования и статические методы не могут быть виртуальными, что не допускает полиморфных отношений.

Реализация

Особенности, которые следует учесть при реализации паттерна Singleton:

- **Гарантия наличия единственного экземпляра.**
  Паттерн Singleton устроен так, что при его применении не удастся создать более одного экземпляра определенного класса. Чаще всего для получения экземпляра используют статический фабричный метод `Instance`, который гарантирует создание не более одного экземпляра. Если метод `Instance` будет вызван впервые, то он создаст экземпляр при помощи защищенного конструктора и вернет ссылку на него, при этом сохранив эту ссылку в одном из своих полей. При всех последующих вызовах метода `Instance`, будет возвращаться ранее сохраненная ссылка на существующий экземпляр.

  См. Пример к главе: \005_Singleton\001_Singleton

- **Наследование от Singleton.**
  Процесс создания производного класса `SingletonDerived` довольно прост, для этого достаточно заместить в нем статический метод `Instance`, в котором унаследованной из базового класса переменной `instance` присвоить ссылку на экземпляр производного класса `SingletonDerived`.

```
public new static DerivedSingleton Instance()
{
    if (uniqueInstance == null)
        uniqueInstance = new DerivedSingleton();

    return uniqueInstance as DerivedSingleton;
}
```

  См. Пример к главе: \005_Singleton\003_SingletonInheritance

Если в программе имеется несколько производных классов-одиночек (DerivedSingleton1, DerivedSingleton2, …) и далее потребуется организовать выбор использования определенного производного класса-одиночки, то для выбора нужного экземпляра-одиночки удобно будет воспользоваться *реестром одиночек*, который представляет собой ассоциативный массив (ключ-значение). В качестве ключа может выступать имя одиночки, а в качестве значения – ссылка на экземпляр определенного одиночки.

См. Пример к главе: \005_Singleton\004_RegistSingleton

- **Потокобезопасный Singleton**
  Может возникнуть ситуация, когда из разных потоков происходит обращение к свойству Singleton Instance, которое должно вернуть ссылку на экземпляр-одиночку и при этом экземпляр-одиночка еще не был создан и его поле instance равно null. Таким образом, существует риск, что в каждом отдельном потоке будет создан свой экземпляр-одиночка, а это противоречит идеологии паттерна Singleton. Избежать такого эффекта возможно через инстанцирование класса Singleton в теле критической секции (конструкции lock), или через использование «объектов уровня ядра операционной системы типа WaitHandle для синхронизации доступа к разделяемым ресурсам».

```
class Singleton
{
    private static volatile Singleton instance = null;
    private static object syncRoot = new Object();

    private Singleton() { }

    public static Singleton Instance
    {
        get
        {
            Thread.Sleep(500);

            if (instance == null)
            {
                lock (syncRoot) // Закомментировать lock для проверки.
                {
                    if (instance == null)
                        instance = new Singleton();
                }
            }

            return instance;
        }
    }
}
```

См. Пример к главе: \005_Singleton\005_MultithreadedSingleton

- **Сериализация Singleton.**
  В редких случаях может потребоваться сериализовать и десериализовать экземпляр класса Singleton. Например, для сериализации и десериализации можно воспользоваться экземпляром класса BinaryFormatter. Сам процесс сериализации и десериализации не вызывает технических трудностей. Единственный момент, на который следует обратить внимание — это возможность множественной десериализации, когда сериализованный ранее экземпляр-одиночка десериализуется несколько раз в одной программе. В таком случае, может получиться несколько различных экземпляров одиночек, что противоречит идеологии паттерна Singleton. Чтобы предотвратить возможность дублирования экземпляра-одиночки при каждой десериализации, рекомендуется использовать подход, при котором сериализуется и десериализуется не сам экземпляр-одиночка, а его суррогат (объект-заместитель).

```
        BinaryFormatter formatter = new BinaryFormatter();
        formatter.SurrogateSelector = Singleton.SurrogateSelector;
```

Использование объекта-суррогата дает возможность организовать тонкий контроль над десериализацией экземпляра-одиночки.

```
// Класс Singleton не должен сериализовываться напрямую
// и не должен иметь атрибута [Serializable]!
public sealed class Singleton
{
    private static Singleton instance = null;
    public String field = "Some value";

    public static Singleton Instance
    {
        get { return (instance == null) ?
                      instance = new Singleton() : instance; }
    }

    public static SurrogateSelector SurrogateSelector
    {
        get
        {
            var selector = new SurrogateSelector();
            var singleton = typeof(Singleton);
            var context = new StreamingContext(StreamingContextStates.All);
            var surrogate = new SerializationSurrogate();

            selector.AddSurrogate(singleton, context, surrogate);
            return selector;
        }
    }

    // Nested class
    private sealed class SerializationSurrogate : ISerializationSurrogate
    {
        // Метод вызывается для сериализации объекта типа Singleton
        void ISerializationSurrogate.GetObjectData(Object obj,
                          SerializationInfo info, StreamingContext context)
        {
            Singleton singleton = Singleton.Instance;
            info.AddValue("field", singleton.field);
        }

        // Метод вызывается для десериализации объекта типа Singleton
        Object ISerializationSurrogate.SetObjectData(Object obj,
                                               SerializationInfo info,
                                               StreamingContext context,
                                               ISurrogateSelector selector)
        {
            Singleton singleton = Singleton.Instance;
            singleton.field = info.GetString("field");
            return singleton;
        }
    }
}
```

См. Пример к главе: \005_Singleton\006_SerializationSingleton

- **Singleton с отложенной инициализацией.**
  Чаще всего метод `Instance` использует отложенную (ленивую) инициализацию, т.е., экземпляр не создается и не хранится вплоть до первого вызова метода `Instance`. Для реализации техники отложенной инициализации в C# рекомендуется воспользоваться классом `Lazy<T>`, причем по умолчанию экземпляры класса `Lazy<T>` являются потокобезопасными.

```
class Singleton
{
    static Lazy<Singleton> instance = new Lazy<Singleton>();

    public static Singleton Instance
    {
        get
        {
            return instance.Value;
        }
    }
}
```

См. Пример к главе: \005_Singleton\007_LazySingleton

Пример кода игры «Лабиринт»

В игре Лабиринт классом с единственным экземпляром может быть класс `MazeFactory`, который строит лабиринт. Легко понять, что для расширения лабиринта путем строительства большего числа комнат со стенами и дверьми не нужно каждый раз строить новую фабрику, всегда можно использовать одну и ту же уже имеющуюся фабрику. Таким образом, сам класс `MazeFactory` будет контролировать наличие одного единственного своего экземпляра `mazeFactory`, и будет предоставлять доступ к этому экземпляру путем использования метода `Instance`.

```
public static MazeFactory Instance()
{
    if (instance == null)
    {
        // Берем значение свойства MAZESTYLE из файла окружения
        string mazeStyle = GetEnv("MAZESTYLE");

        // 0 - совпадают, 1 - не совпадают
        if (string.Compare(mazeStyle, "bombed") == 0)
        {
            Console.WriteLine("Фабрика для лабиринта с бомбами");
            instance = new BombedMazeFactory();
        }
        else if (string.Compare(mazeStyle, "enchanted") == 0)
        {
            Console.WriteLine("Фабрика для лабиринта с заклинаниями");
            instance = new EnchantedMazeFactory();
        }
        else // По умолчанию.
        {
            Console.WriteLine("Фабрика для обычного лабиринта");
            instance = new MazeFactory();
        }
    }
    return instance;
}
```

Следует заметить, что в данной реализации, метод `Instance` нужно модифицировать при определении каждого нового подкласса `MazeFactory`. Такой подход является неприемлемым, поскольку не обеспечивает необходимую гибкость разработки. Вариантами решения данной проблемы

являются использование принципов «реестра одиночек» и отложенной (ленивой) инициализации, тогда приложению не нужно будет загружать все неиспользуемые подклассы.

Известные применения паттерна в .Net

`System.ServiceModel.ServiceHost`
http://msdn.microsoft.com/ru-RU/library/system.servicemodel.servicehost.singletoninstance(v=vs.100).aspx

`System.Data.DataRowComparer`
http://msdn.microsoft.com/ru-ru/library/system.data.datarowcomparer.aspx

Глава 3. Структурные паттерны

Использование структурных паттернов помогает из классов и объектов строить более крупные структуры.

Структурные паттерны уровня класса используют наследование для составления композиций из интерфейсов и реализаций. Например, в C++ - допустимо использование множественного наследования реализации для объединения нескольких классов в один. В результате получается класс, обладающий свойствами всех своих родителей. В C# - не допустимо множественное наследование реализации (`class`), но допустимо множественное наследование интерфейсов (`interface`), что позволяет организовывать композиции интерфейсов.

Структурные паттерны уровня объекта используют композицию объектов для получения новой функциональности. Дополнительная гибкость в этом случае связана с возможностью изменить композицию объектов во время выполнения, что недопустимо для статической композиции классов.

Многие структурные паттерны в той или иной мере связаны друг с другом.

Паттерн Adapter

Название

Адаптер

Также известен как

Wrapper (обертка)

Классификация

По цели: структурный
По применимости: к классам и объектам

Частота использования

Выше средней - 1 2 3 **4** 5

Назначение

Паттерн Adapter - преобразует интерфейс (набор имен методов) одного класса в интерфейс (набор имен методов) другого класса, который ожидают клиенты. Адаптер обеспечивает совместную работу классов с несовместимыми интерфейсами, такая работа без Адаптера была бы невозможна.

Введение

Рассмотрим самый широко распространенный вид адаптеров в жизни – адаптеры электрической сети. Исторически сложилось так, что в разных странах мира имеются свои технологические стандарты на производство и использование электробытовых устройств. И часто бывает так, что стандарты устройств одной страны не совместимы со стандартами устройств других стран.

Например, ноутбук, произведенный согласно стандартам и требованиям к сети электрического питания США, имеет определенную входную силу тока и напряжение, а также подключается к сети электрического питания с использованием вилки типа “B”. Вилка подключения к электросети этого стандарта имеет 2 плоских контакта и один круглый контакт для заземления, такой тип также известен как PBG (Parallel Blade with Ground) или Hubbell (по имени известного производителя электрофурнитуры).

Владелец такого ноутбука смог бы без проблем подключиться к сети питания в Канаде, Японии и Мексике, так как в этих странах используется такой же стандарт подключения, как и в США. Но как быть, если владельцу такого ноутбука необходимо поехать Великобританию, где принят стандарт розеток типа «G» (три больших плоских контакта, расположенных треугольником) или какую-нибудь другую Европейскую

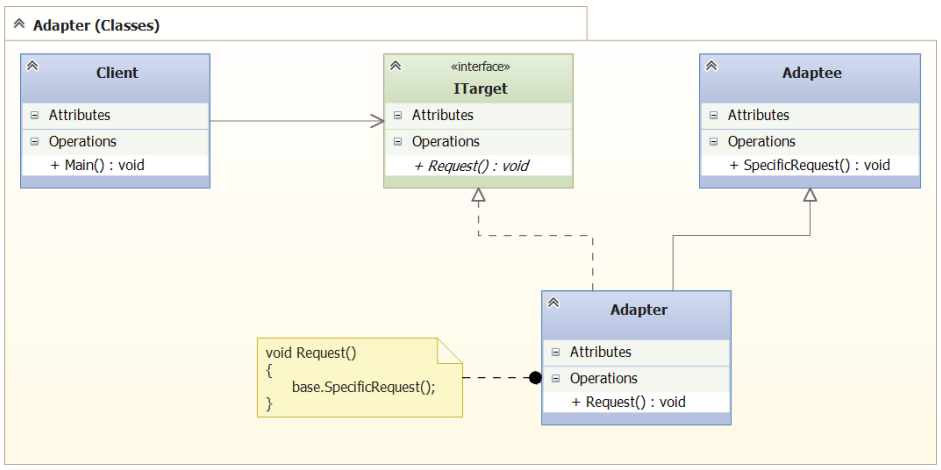
страну, где принят стандарт «F» (Schuko) (два круглых контакта и две заземляющих клеммы вверху и внизу розетки)? Сложности в этой ситуации добавляет еще и тот факт, что стандарт Великобритании на электрические сети и подключение к ним допускает использование в сети тока большой силы, и поэтому обязательным является использование предохранителя непосредственно в вилке подключения к сети.

Как вариант решения данной проблемы, можно было бы производить различные ноутбуки для использования в различных странах, но для человека, который много путешествует по миру, согласитесь, достаточно накладно было бы возить с собой множество ноутбуков, сделанных по различным стандартам и соответствующих техническим условиям разных стран.

Разумеется, есть другой, более рациональный подход – использование адаптеров. То есть приспособлений, которые позволяют «сдружить» различные стандарты подключения электробытовых устройств, таким образом, чтобы выполнялись требования и адаптируемого устройства - вилки ноутбука и электрической розетки, к которой необходимо адаптироваться. При этом, в зависимости от страны в которой пребывает владелец ноутбука, он может использовать различные адаптеры, позволяющие адаптировать ноутбук к стандартам этой страны.

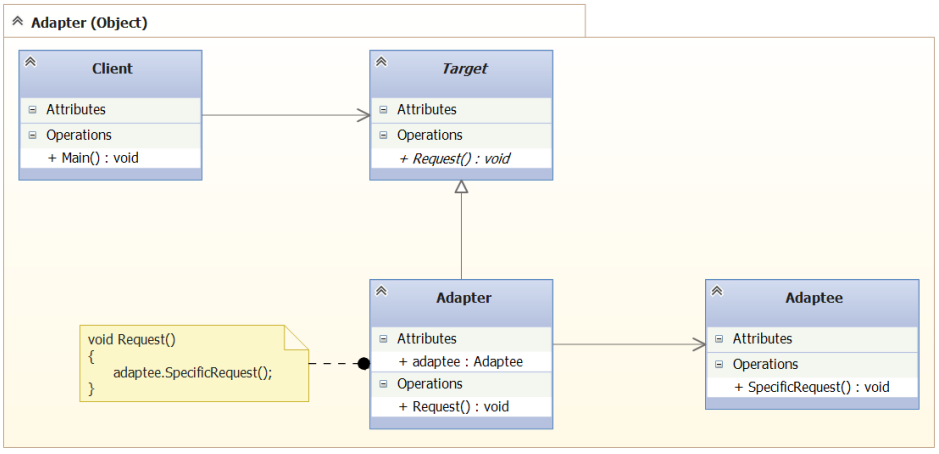
Структура паттерна на языке UML

Adapter уровня классов



См. Пример к главе: \006_Adapter\001_Adapter (Class)

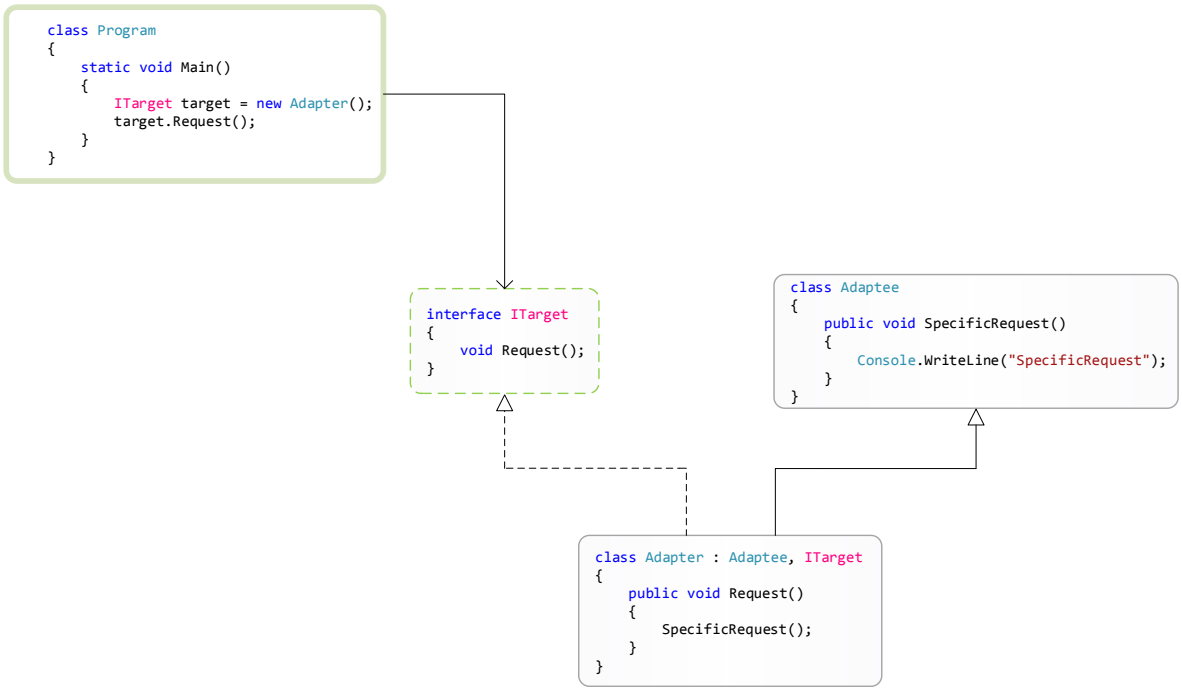
Adapter уровня объектов



См. Пример к главе: \006_Adapter\001_Adapter (Object)

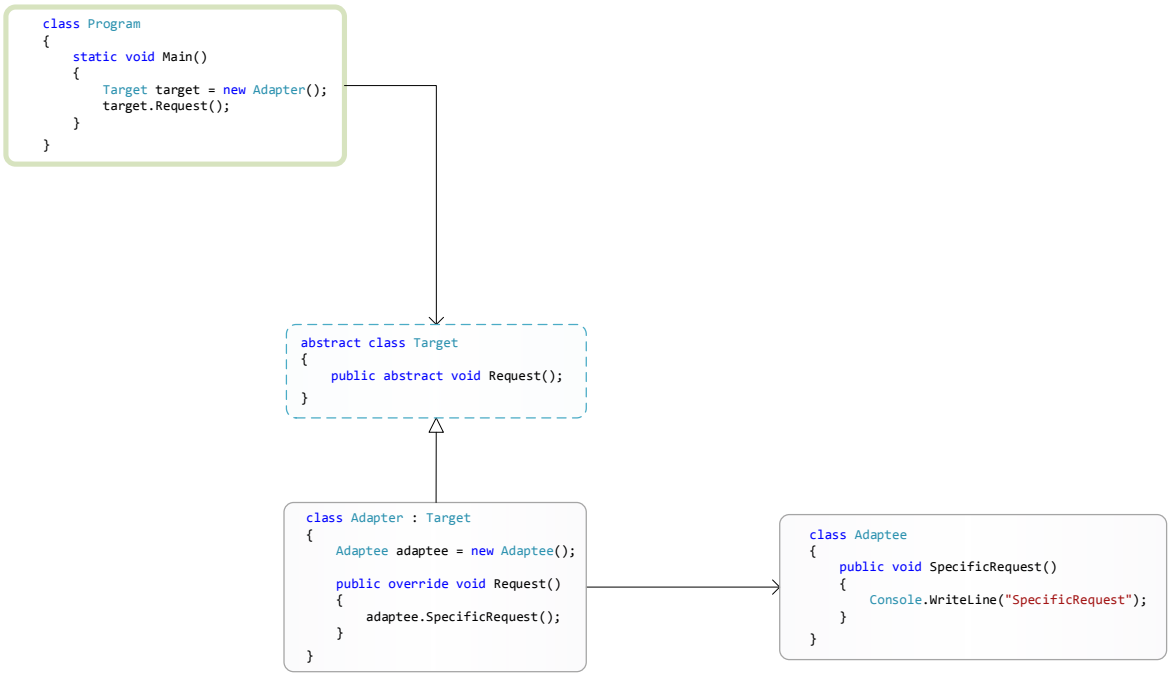
Структура паттерна на языке C#

Adapter уровня классов



См. Пример к главе: \006_Adapter\001_Adapter (Class)

Adapter уровня объектов



См. Пример к главе: \006_Adapter\001_Adapter (Object)

Участники

- `Target` - **Цель**:
  Формирует требуемый клиенту интерфейс (набор имен методов).

- `Client` - **Клиент**:
  Пользуется объектами с интерфейсом `Target`.

- `Adaptee` - **Адаптируемый**:
  Содержит интерфейс (набор методов) требующий адаптации.

- `Adapter` - **Адаптер**
  Адаптирует интерфейс `Adaptee` к интерфейсу `Target`.

Отношения между участниками

Отношения между классами (для адаптера уровня классов)

- Класс `Adapter` связан связью отношения наследования с классом `Adaptee` и связью отношения реализации с интерфейсом `ITarget`.

Отношения между классами (для адаптера уровня объектов)

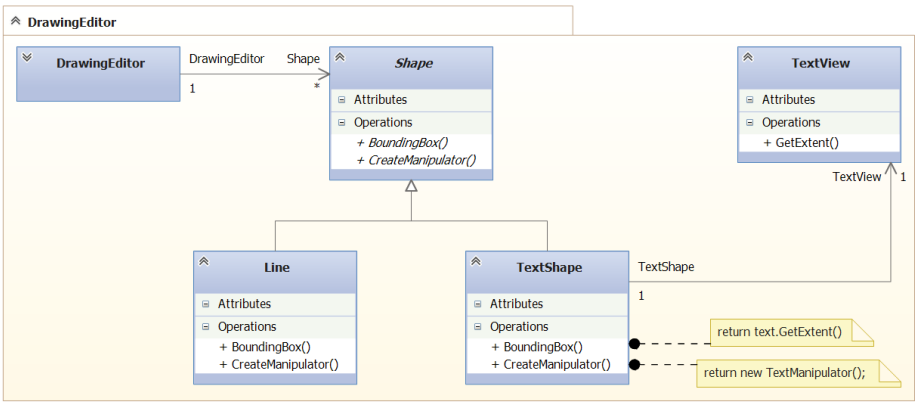
- Класс `Adapter` связан связью отношения наследования с абстрактным классом `Target` и связью отношения ассоциации с классом `Adaptee`.

Отношения между объектами

- Клиент вызывает методы на экземпляре класса `Adapter`. `Adapter` в свою очередь вызывает методы адаптируемого объекта или класса `Adaptee`.

Мотивация

Предлагается построить простейший графический редактор, в котором имеется возможность рисовать линии и вставлять блоки с текстом. Основной абстракцией, с которой будет работать пользователь, является объект реализующий интерфейс, заданный абстрактным классом `Shape`. Этот объект пользователь может создавать непосредственно в рабочей области на форме и изменять его путем перемещения по рабочей области в произвольном направлении. Например, для работы с прямыми линиями, в графическом редакторе определен класс `Line`, а для работы с текстом определен класс `TextShape`, оба эти класса являются производными от абстрактного класса `Shape`.



Реализовать класс `Line` легко, так как выполнения каких-либо сложных действий с линиями в программе не предусматривается. Совсем по-другому обстоит ситуация с классом `TextShape`. Реализовать в нем операции отображения и перемещения по рабочей области значительно сложнее. Можно было бы воспользоваться имеющимся классом `TextView` (условимся, что `TextView` приобретен у стороннего производителя элементов управления), в котором уже реализован необходимый функционал, позволяющий отображать текст нужным образом. В силу обстоятельств, класс `TextView`, не является производным от класса `Shape`, поскольку разрабатывался сторонними производителями, и его нельзя напрямую использовать в базовом классе редактора `DrawingEditor`.

Для организации использования класса `TextView` в данном проекте требуется изменить его интерфейс так, чтобы он соответствовал интерфейсу типа `Shape`. Так как класс `TextView` разрабатывался сторонними разработчиками, то доступ к его исходному коду может отсутствовать. Но, даже при наличии исходного кода, неразумно вносить в него изменения, поскольку нелогично подключаемые библиотеки приспосабливать к интерфейсу целевой программы. Правильным решением проблемы несовместимости интерфейсов, является адаптация интерфейса класса `TextView` к интерфейсу класса `Shape`. Такую адаптацию можно организовать следующим образом: при создании класса `TextShape` следует включить в него функциональность из класса `TextView` через использование композиции, при этом реализовать в классе `TextShape` интерфейс типа `Shape`. Реализация интерфейса типа `Shape` (в данном случае метода `BoundingBox`) в классе `TextShape` будет служить своеобразной оберткой над методом `GetExtend` из класса `TextView`. В таком случае класс `TextShape` будет выступать в роли адаптера (уровня объектов), и графический редактор сможет воспользоваться функциональностью класса `TextView`.

Зачастую, на практике адаптируемый класс не всегда предоставляет всю необходимую разработчикам функциональность, и в таком случае на класс-адаптер возлагается ответственность за добавление новой и расширение существующей функциональности. Класс `TextView` не предоставляет возможности перемещать текст по рабочей области, но класс `TextShape`, такую возможность может предоставить, через реализацию абстрактного метода `CreateManipulator` из базового класса `Shape`. Метод `CreateManipulator` возвращает экземпляр класса `TextManipulator` приведенный к базовому типу `Manipulator`. Абстрактный класс `Manipulator` – класс объектов, которым должно быть известно, как анимировать экземпляры объектов типа `Shape`, при их перетаскивании мышью по рабочей области окна. Для работы с каждым типом объектов производных от `Shape` (`Line`, `TextShape`) у класса `Manipulator` имеются подклассы `LineManipulator` и `TextManipulator`. Таким образом, объект класса `TextShape` расширяет возможности класса `TextView`, для его использования в приложении графического редактора.

См. Пример к главе: \006_Adapter\002_DrawingEditor

Применимость паттерна

Паттерн Adapter рекомендуется использовать, когда:

- Требуется использовать уже существующий класс, но его интерфейс (набор методов) не соответствует требованиям клиента.
- Требуется создать повторно используемый класс который должен взаимодействовать с классами имеющими не совместимые интерфейсы.

Результаты

Варианты использования паттерна Adapter уровня классов и паттерна Adapter уровня объектов, имеют свои особенности.

Adapter уровня классов

- Адаптирует интерфейс `Adaptee` к интерфейсу `ITarget`, делегируя ответственность за выполнение операций (методов) конкретному классу `Adaptee`.
- Позволяет классу `Adapter` переопределить или заместить некоторые операции из базового класса `Adaptee`.
- Оставляет возможность создания только одного экземпляра класса `Adapter`, так как `Adapter` наследуется от `Adaptee`, то не придётся отдельно создавать экземпляр `Adaptee` внутри класса `Adapter`.

Adapter уровня объектов

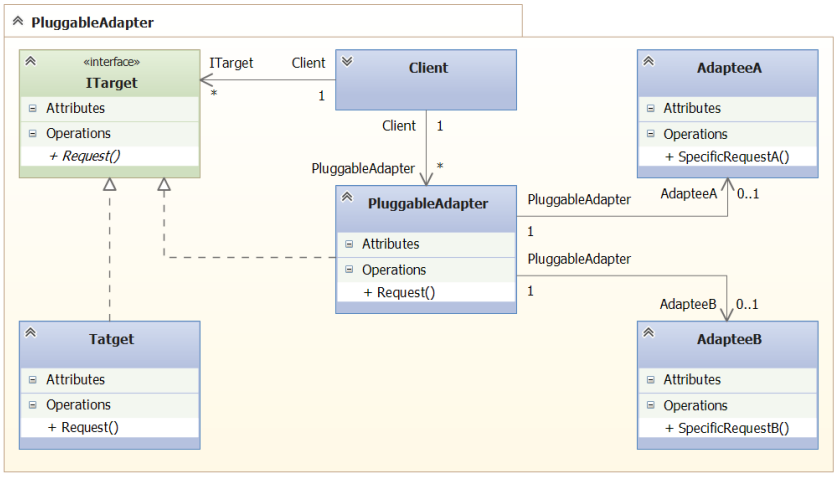
- Позволяет одному экземпляру класса `Adapter`, работать с разными адаптируемыми объектами типа `Adaptee`, как с экземплярами класса `Adaptee`, так и с экземплярами классов производных от `Adaptee`.
- Позволяет классу `Adapter` расширить некоторые операции экземпляра класса `Adaptee` (например, в методе класса `Adapter` сперва выполняется некоторая своя функциональность, а после вызывается метод экземпляра класса `Adaptee`).
- Затрудняет расширение (переопределение или замещение) защищенных виртуальных (`protected virtual`) и невиртуальных методов из класса `Adaptee`. Для расширения защищенных методов потребуется создать класс `DerivedAdaptee` производный от класса `Adaptee`, внести в него необходимые изменения и в адаптере использовать экземпляр класса `DerivedAdaptee`.

Особенности применения паттерна Adapter

- **Объем работ по адаптации.**
  Устройство разных адаптеров может сильно отличаться. Класс `Adapter` может просто изменять имена методов класса `Adaptee`, но может и расширять методы класса `Adaptee`, при этом добавляя свои новые методы с использованием которых формируется желаемый (требуемый) интерфейс взаимодействия с объектом класса `Adapter`.
- **Сменные адаптеры.**
  Степень многократного использования (reusable) класса «`A`» будет высокой, тогда и только тогда, когда разработчик класса «`A`» не будет делать предположений о том, какие классы будут использовать класс «`A`».
  Встраивая механизм адаптации интерфейса (набора методов) класса `Adaptee` в класс `Adapter`, программист отказывается от предположения, что какой-либо другой класс в программе (кроме класса `Adapter`) будет использовать интерфейс класса `Adaptee` напрямую. Другими

словами, адаптация интерфейса (набора методов) класса Adaptee в классе Adapter, позволяет включить интерфейс Adaptee только через класс Adapter в существующую систему (программу), в которой все классы спроектированы так, что не могут воспользоваться интерфейсом класса Adaptee напрямую.

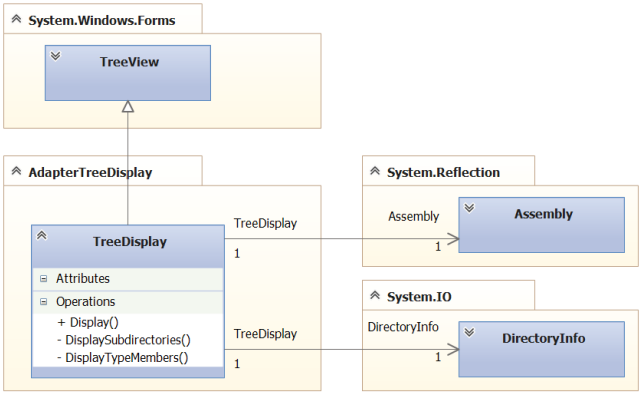
Имеется разновидность адаптеров, которые называются *сменными адаптерами* (pluggable adapters). Сменный адаптер, это такой адаптер PluggableAdapter, в который встроен механизм адаптации интерфейсов нескольких адаптируемых классов AdapteeA, AdapteeB и т.д.



См. Пример к главе: \006_Adapter\003_PluggableAdapter

Рассмотрим элемент управления TreeView, который отображает древовидные структуры. Разные классы имеют свои собственные интерфейсы для получения доступа к элементам древовидных структур, которые эти классы представляют.

Например, можно сравнить интерфейсы взаимодействия с классами DirectoryInfo и Assembly. Как можно убедиться, интерфейсы этих классов различны. Класс TreeView не адаптирован для работы с классами DirectoryInfo и Assembly напрямую. Клиент должен самостоятельно назначать данные узлам TreeNode, класса TreeView, для отображения этих данных в дереве. Для упрощения работы клиента, есть смысл создать пользовательский элемент управления – класс TreeDisplay, для отображения древовидных структур. Элемент управления типа TreeDisplay должен уметь отображать иерархии обоих видов древообразных структур, как DirectoryInfo, так и Assembly. Другими словами, в TreeDisplay должна быть встроена адаптация интерфейсов классов DirectoryInfo и Assembly.



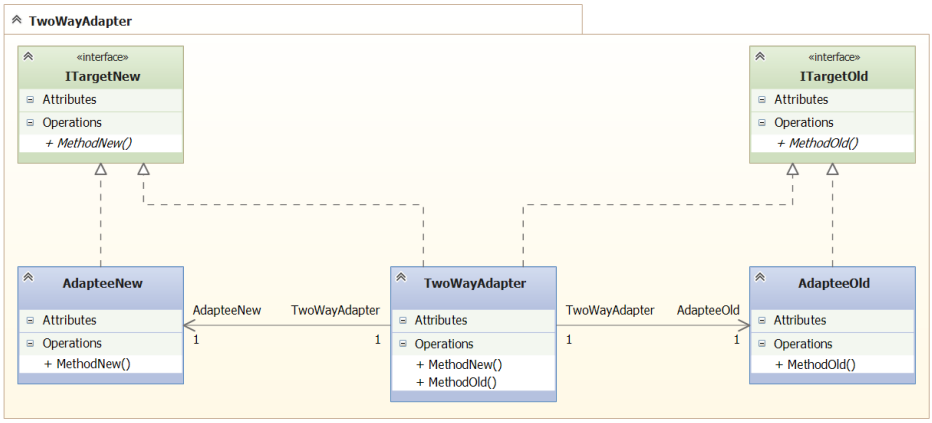
См. Пример к главе: \006_Adapter\004_AdapterTreeDisplay

- **Двусторонние адаптеры.**

Важное правило: клиент не должен знать, что используемый им объект является адаптером. Объект класса Adapter не обладает открытым интерфейсом класса Adaptee, поэтому

объект класса Adapter невозможно использовать там, где может использоваться объект класса Adaptee.

Имеется разновидность адаптеров, которые называются *двусторонними адаптерами* (two-way adapters). Двусторонние адаптеры способны обеспечить возможность использовать адаптер TwoWayAdapter там, где мог использоваться Adaptee.



Интерфейс двустороннего адаптера (класса TwoWayAdapter) представляет собой объединение интерфейсов адаптируемых друг к другу классов AdapteeNew и AdapteeOld.

См. Пример к главе: \006_Adapter\005_TwoWayAdapter

Реализация

Имеется три подхода к реализации сменных адаптеров. Эти способы проще рассмотреть на примере использования пользовательского элемента управления TreeDisplay, для автоматического отображения иерархических древовидных структур (TreeDisplay уже был описан выше).

- **Первый подход – использование абстрактных операций.**
  При таком подходе в теле абстрактного класса-цели TreeDisplay необходимо задать абстрактный метод Display, с аргументом типа object, значение аргумента должно представлять собой ссылку на иерархическую структуру. Метод Display, в данном случае, формирует «узкий» интерфейс для адаптируемых классов со встроенными адаптерами: AssemblyTreeDisplay и DirectoryTreeDisplay. Другими словами, метод Display представляет собой наименьшее подмножество операций, позволяющих выполнить адаптацию, и его реализация фактически является оберткой над специфическими методами адаптируемых классов AssemblyTreeDisplay и DirectoryTreeDisplay, которые позволяют осуществлять доступ к иерархической структуре сборки проекта и структуре каталогов файловой системы.

  См. Пример к главе: \006_Adapter\006_3_AdaptersForTreeDisplay

- **Второй подход – использование объектов-уполномоченных.**
  Для того чтобы реализовать данный подход потребуется создать специальные объекты – «*объекты-уполномоченные*». Тогда класс TreeDisplay сможет переадресовывать запросы для доступа к иерархической структуре этим объектам-уполномоченным, при этом, реализовывая различные стратегии адаптации путем использования необходимых конкретных уполномоченных. Например, одним из таких объектов-уполномоченных для класса TreeDisplay мог бы быть класс DirectoryBrowser. В статически типизированных языках, которым и является C#, потребуется явно задать интерфейс, который необходим классу TreeDisplay, например, `interface ITreeAccessorDelegate`, и реализовать этот интерфейс в выбранном классе-уполномоченном DirectoryBrowser.

- **Третий подход – адаптеры, конфигурируемые при помощи параметров.**
  Этот подход предоставляет возможность использования конструкции `switch` или нескольких условных конструкций `if-else` в теле адаптера, как альтернативу наследованию при реализации

сменных адаптеров. Такой подход позволяет реализовывать механизмы адаптации без порождения подклассов. Применимо, к рассматриваемому примеру это означает, что в классе TreeDisplay может быть два условных блока: первый блок, для преобразования узла в тип GraficNode, а второй блок – для доступа к потомкам узла.

Пример кода

Предлагается рассмотреть реализацию адаптера уровня класса и адаптера уровня объекта на примере, из раздела «Мотивация»:

В классе Shape создается абстрактный фабричный метод CreateManipulator, с возвращаемым значением типа Manipulator. Manipulator обладает функциональностью по анимированию объектов класса Shape в ответ на действия пользователя. В классе Shape также создается абстрактный метод BoundingBox, который отвечает за отображение элементов типа Shape на форме (TextShape и Line).

```
public abstract class Shape
{
    public abstract void BoundingBox();
    public abstract Manipulator CreateManipulator();
}
```

Абстрактный класс Manipulator реализует взаимодействие и контролирует состояние объекта-манипулятора. Манипуляторы представлены как наследники базового класса Control.

```
public abstract class Manipulator : Control
{
    //состояние манипулятора на форме.
    public Point StartPoint { get; protected set; }
    public Point EndPoint { get; protected set; }

    //задание базовых параметров контрола
    public Manipulator()
    {
        SetStyle(ControlStyles.UserPaint, true);
        SetStyle(ControlStyles.SupportsTransparentBackColor, true);
        SetStyle(ControlStyles.Opaque, true);
        this.BackColor = Color.Transparent;
        this.DoubleBuffered = false;
    }

    //определяет параметры необходимые при создании контрола
    protected override CreateParams CreateParams
    {
        get
        {
            const int WS_EX_TRANSPARENT = 0x00000020;
            CreateParams createParams = base.CreateParams;
            createParams.ExStyle |= WS_EX_TRANSPARENT;
            return createParams;
        }
    }
}
```

Класс TextView содержит метод GetExtend, который отвечает за отображение объектов на форме, этот метод является аналогом метода BoundingBox, а аналог фабричного метода для создания манипулятора в классе TextView отсутствует. Таким образом для работы с классом TextView в проекте необходимо использовать класс-адаптер TextShape, который адаптирует интерфейс класса TextView к интерфейсу класса Shape.

```
class TextView
{
    Label label;

    public TextView()
    {
        label = new Label();
    }

    public Label GetExtend()
    {
        label.ForeColor = Color.DarkMagenta;
        label.Font = new Font(label.Font, label.Font.Style | FontStyle.Bold);
        label.BorderStyle = BorderStyle.None;
        label.TextAlign = ContentAlignment.TopLeft;
        label.FlatStyle = FlatStyle.Standard;
        label.AutoSize = true;

        return label;
    }
}


class TextShape : Shape
{
    Point startPoint;
    string text;
    TextView textView;
    Label label;

    public TextShape(Point startPoint, string text)
    {
        textView = new TextView();
        this.startPoint = startPoint;
        this.text = text;
    }

    // Преобразование запроса BoundingBox в запрос GetExtend.
    public override void BoundingBox()
    {
        label = textView.GetExtend();
        label.Text = text;
    }

    // Фабричный метод, возвращающий манипулятор соответствующий
    // конкретной фигуре
    public override Manipulator CreateManipulator()
    {
        return new TextManipulator(startPoint, label);
    }
}
```

Поскольку в классе `TextView` не предусмотрен метод для создания манипулятора, который будет работать с текстом, то необходимо такой метод реализовать самостоятельно. В классе `TextShape` реализация метода `CreateManipulator` не использует повторно никакой функциональности из класса `TextView`.

```
class TextManipulator : Manipulator
{
```

```
Label textInside;
private int deltaX, deltaY;
Point currentMouse;
bool mousePressed = false;

// Метод, возвращающий глобальные координаты курсора на
// контроле относительно формы
Point GetMouseLocation(Point curMouse)
{
     return new Point(curMouse.X + this.Location.X, curMouse.Y +
     this.Location.Y);
}

// Перерисовка манипулятора
void ReDrawControl(Point startPoint)
{
    Size = new Size(textInside.Text.Length * 8, textInside.Size.Height);
    Location = startPoint;
    RecreateHandle();
    Refresh();
}

public TextManipulator(Point startPoint, Label label)
{
    textInside = label;
    StartPoint = startPoint;
    ReDrawControl(startPoint);
    Controls.Add(textInside);
    textInside.MouseDown += TextInside_MouseDown;
    textInside.MouseUp += TextInside_MouseUp;
}

// Методы обработчики базового класса Manipulator
void TextInside_MouseDown(object sender, MouseEventArgs e)
{
    OnMouseDown(e);
}

void TextInside_MouseUp(object sender, MouseEventArgs e)
{
    OnMouseUp(e);
}
```

```
// Метод, определяющий новые координаты манипулятора на форме
protected override void OnMouseUp(MouseEventArgs e)
{
    if (mousePressed)
    {
        base.OnMouseUp(e);
        deltaX = GetMouseLocation(e.Location).X - currentMouse.X;
        deltaY = GetMouseLocation(e.Location).Y - currentMouse.Y;
        StartPoint = new Point(StartPoint.X + deltaX, StartPoint.Y + deltaY);
        ReDrawControl(StartPoint);
        currentMouse = GetMouseLocation(e.Location);
```

```
            }
            else
                mousePressed = false;
            Cursor = Cursors.Arrow;
        }

        // Определение координат текущего манипулятора по клику
        protected override void OnMouseDown(MouseEventArgs e)
        {
            base.OnMouseDown(e);
            currentMouse = GetMouseLocation(e.Location);
            mousePressed = true;
            Cursor = Cursors.Hand;
        }
    }
```

См. Пример к главе: \006_Adapter\002_DrawingEditor

Известные применения паттерна в .Net

Паттерн Adapter, имеет одно из выражений в платформе .Net в виде идеи использования перегрузки операторов преобразования значения типа (`explicit` / `implicit`). А также:

`System.Web.Configuration.AdapterDictionary`
http://msdn.microsoft.com/ru-ru/library/system.web.configuration.adapterdictionary.aspx

`System.AddIn.Pipeline.AddInAdapterAttribute`
http://msdn.microsoft.com/ru-ru/library/system.addin.pipeline.addinadapterattribute.aspx

`System.AddIn.Pipeline.CollectionAdapters`
http://msdn.microsoft.com/ru-ru/library/system.addin.pipeline.collectionadapters(v=vs.90).aspx

`System.AddIn.Pipeline.ContractAdapter`
http://msdn.microsoft.com/ru-ru/library/system.addin.pipeline.contractadapter(v=vs.90).aspx

`System.AddIn.Pipeline.HostAdapterAttribute`
http://msdn.microsoft.com/ru-ru/library/system.addin.pipeline.hostadapterattribute(v=vs.90).aspx

`System.Data.Common.DataAdapter`
http://msdn.microsoft.com/en-us/library/system.data.common.dataadapter(v=vs.110).aspx

`System.Data.Common.DbDataAdapter`
http://msdn.microsoft.com/en-us/library/system.data.common.dbdataadapter(v=vs.110).aspx

`System.Data.Odbc.OdbcDataAdapter`
http://msdn.microsoft.com/ru-ru/library/system.data.odbc.odbcdataadapter(v=vs.110).aspx

`System.Data.OleDb.OleDbDataAdapter`
http://msdn.microsoft.com/en-us/library/system.data.oledb.oledbdataadapter(v=vs.110).aspx

`System.Data.OracleClient.OracleDataAdapter`
http://msdn.microsoft.com/en-us/library/system.data.oracleclient.oracledataadapter(v=vs.110).aspx

`System.Data.SqlClient.SqlDataAdapter`
http://msdn.microsoft.com/ru-ru/library/system.data.sqlclient.sqldataadapter(v=vs.110).aspx

`System.Web.UI.Adapters.ControlAdapter`
http://msdn.microsoft.com/en-us/library/system.web.ui.adapters.controladapter(v=vs.110).aspx

`System.Web.UI.Adapters.PageAdapter`
http://msdn.microsoft.com/en-us/library/system.web.ui.adapters.pageadapter(v=vs.110).aspx

`System.Web.UI.MobileControls.DesignerAdapterAttribute`
http://msdn.microsoft.com/ru-ru/library/system.web.ui.mobilecontrols.designeradapterattribute(v=vs.110).aspx

`System.Web.UI.WebControls.Adapters.DataBoundControlAdapter`
http://msdn.microsoft.com/ru-ru/library/system.web.ui.webcontrols.adapters.databoundcontroladapter(v=vs.90).aspx

`System.Web.UI.WebControls.Adapters.WebControlAdapter`
http://msdn.microsoft.com/ru-ru/library/system.web.ui.webcontrols.adapters.webcontroladapter(v=vs.90).aspx

`System.Web.UI.WebControls.Adapters.MenuAdapter`
http://msdn.microsoft.com/en-us/library/system.web.ui.webcontrols.adapters.menuadapter(v=vs.110).aspx

Паттерн Bridge

Название

Мост

Также известен как

Handle/Body (описатель/тело)

Классификация

По цели: структурный
По применимости: к объектам

Частота использования

Средняя - 1 2 **3** 4 5

Назначение

Паттерн Bridge – позволяет отделить абстракцию от элементов ее реализации так, чтобы и абстракцию, и реализацию можно было изменять независимо друг от друга.

Введение

Абстракция – собирательное понятие дающее возможность просто и обобщенно представить сложную сущность, состоящую из множества частей. Любые абстракции формируются из конкретных понятий или других абстракций которые в свою очередь сформированы из конкретных понятий и т.д.

Конкретика – точное понятие, описывающее единичный (не требуемый разложения на более мелкие части) объект с ясно определенными признаками. Конкретика основана на реализации.

Конкретика и абстракция – философские понятия, поэтому они относительны и могут стать причиной разногласий при формировании уровней детализации как жизненных, так и виртуальных программных сущностей.

Рассмотрим такую абстракцию из объективной реальности как компьютер. Что такое компьютер? Компьютер – это собирательное понятие описывающее устройство, состоящее из конкретных частей – монитор, системный блок, клавиатура, манипулятор-мышь. Но кто-то возразит и предложит рассматривать системный блок как абстракцию, состоящую из конкретных частей таких как – корпус, блок питания, жесткий диск, материнская плата, процессор, ОЗУ, видеокарта, дисковод. Но, и тут можно возразить, и предложить рассмотреть более детально устройство процессора, и представить процессор как абстракцию, состоящую из триггеров и других элементов. Триггеры можно в свою очередь представить в форме абстракции и разложить на более мелкие элементы, например, транзисторы. Транзистор тоже можно представить, как абстракцию, состоящую из трех электродов (эмиттер, база, коллектор) и небольшого количества кремния. А что такое атом кремния? – это абстракция. Атом - собирательное понятие из нейтронов, протонов и электронов. А что такое нейтрон? … Так можно войти в паралич анализа и никогда не закончить декомпозицию каждой составляющей компьютера раскладывая на атомы все его составляющие. Когда же следует закончить декомпозицию и очередную абстракцию представить конкретикой? Тогда, когда мы согласны работать с устройством как с кибернетически черным ящиком (не понимая, как устройство функционирует) и готовы пренебречь гибкостью сопровождения данного устройства. Например, если вышло из строя устройство для чтения DVD дисков, то один человек просто заменит устройство на новое, а другой человек попытается его починить (обладая соответствующими знаниями).

Рассмотрим абстракцию из виртуальной реальности. Например, объект класса `OpenFileDialog`. Что такое `OpenFileDialog`? – это собирательное понятие, состоящее из набора элементов, представляющее собой диалоговое окно для выбора файла. Но, `OpenFileDialog` – конкретный класс, он спроектирован так, чтобы представлять собой точный единичный объект без возможности его расширения (`sealed`). Разработчики этого класса не рекомендуют воспринимать его как абстрактное понятие, и соответственно не рекомендуют детально разбираться в его устройстве. Рекомендуется им просто

пользоваться как черным ящиком, а настоящая абстракция расположена выше в графе наследования – в классе `FileDialog`.

Граф наследования класса `OpenFileDialog` устроен так:

```
sealed class OpenFileDialog : FileDialog {…}
```

в свою очередь:

```
abstract class FileDialog : CommonDialog {…}
```

в свою очередь:

```
abstract class CommonDialog : Component {…}
```

в свою очередь:

```
class Component : MarshalByRefObject, IComponent, IDisposable {…}
```

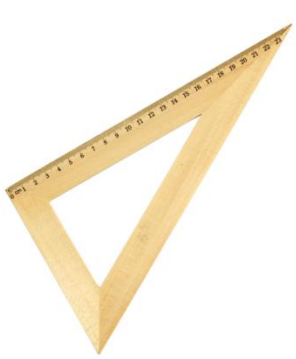
Формирование как конкретных понятий, так и абстрактных понятий, возможно производить двумя способами, через наследование классов и/или через композицию объектов.

Если некоторая абстракция подразумевает несколько реализаций (конкретик), то обычно применяют наследование. Например, абстракция `FileDialog` имеет две конкретные реализации, это классы `OpenFileDialog` и `SaveFileDialog`. Абстракция задает общий интерфейс взаимодействия с конкретными реализациями, тем самым формируя понятие общего типа. Конкретные классы могут по-разному реализовать абстрактный интерфейс, главное, чтобы имена методов, составляющих общий интерфейс взаимодействия, соответствовали роду выполняемой ими деятельности.

Но, подход с наследованием разрушает гибкость и формирует хрупкость. Хрупкость базового класса – фундаментальная проблема в ООП. Проблема хрупкого базового класса заключается в том, что изменения, внесенные в реализацию базового класса, могут негативно сказаться на работе производных классов. Наследование формирует сильнейшую форму связанности (coupling - мера зависимости) из всех возможных и разорвать эту связанность невозможно. Соответственно потеря гибкости означает – затруднение модификации, расширения и повторного использования как абстракции, так и реализации (конкретики). Использование паттерна Bridge позволяет решить описанные выше проблемы.

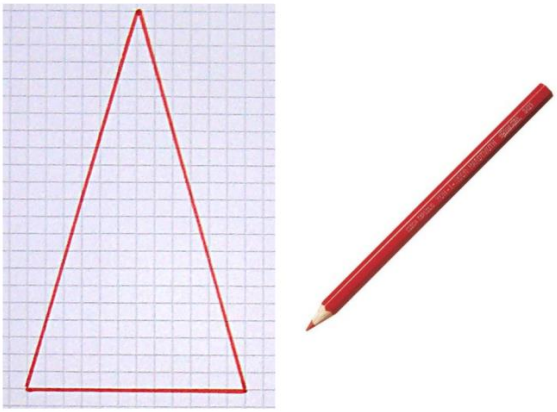
Предлагается воспользоваться паттерном Bridge для построения простейшей программы, которая рисует заданные геометрические фигуры линиями определенного стиля.

В качестве примера рассмотрим треугольник. Треугольник — это абстракция. Треугольник (в евклидовом пространстве) — это геометрическая фигура, образованная тремя отрезками, которые соединяют три не лежащие на одной прямой точки. Предлагается рассмотреть треугольники не в смысле геометрических типов (прямоугольные, равнобедренные и др.) а как выражения треугольных форм предметов из объективной реальности. Например, школьный треугольник, треугольник для бильярда и музыкальный треугольник.

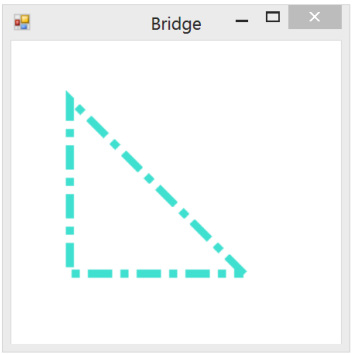


Сам треугольник – это абстракция, а те части из которых он состоит – есть элементы реализации абстракции. Для школьного треугольника и треугольника для бильярда реализацией будут те деревянные стороны из которых он состоит, а для музыкального треугольника реализацией будет являться изогнутый металлический прут.

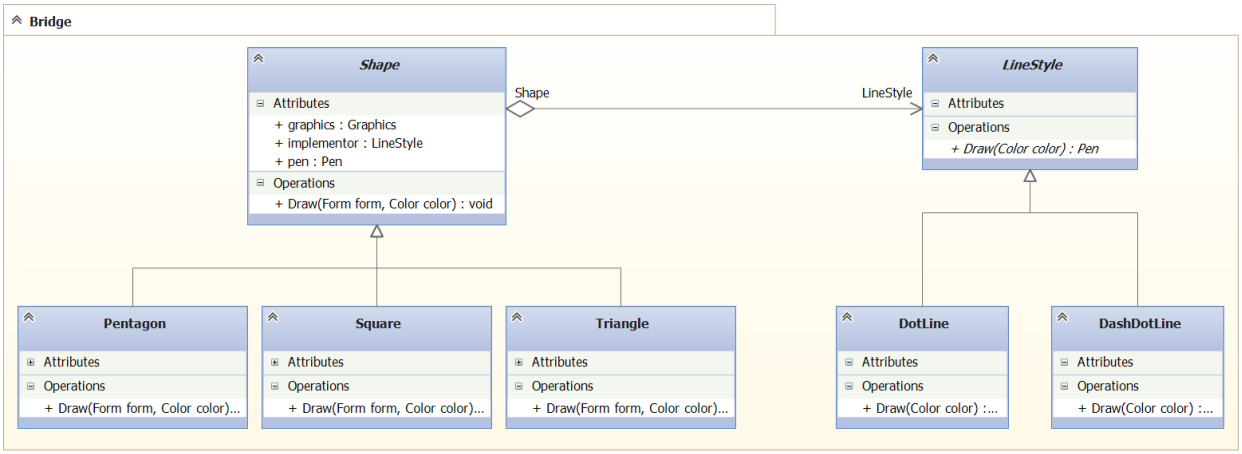
Для нарисованного на листе бумаги треугольника реализацией будет являться краска карандаша, нанесенная на лист бумаги.



Для нарисованного треугольника на Windows форме реализацией будет линия, вычерченная графическим устройством `Graphics` при помощи пера `Pen`.

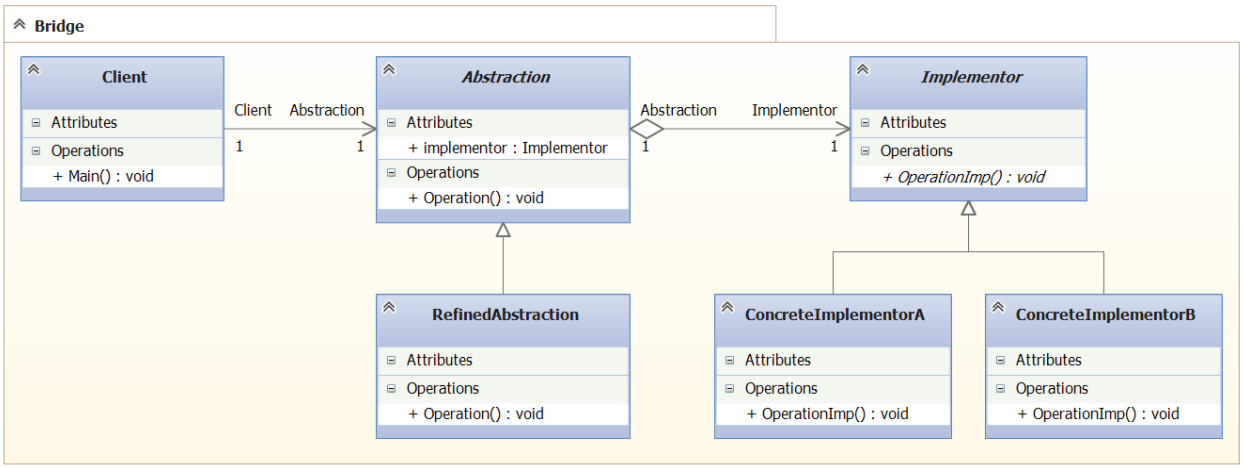


Ниже представлена диаграмма классов на которой представлены две иерархии: классы геометрических форм `Shape` (абстракций) и линий `LineStyle` (реализаций).



См. Пример к главе: \007_Bridge\004_ Bridge WinForm (Shapes and Lines)

Структура паттерна на языке UML



См. Пример к главе: \007_Bridge\001_Bridge

Структура паттерна на языке C#

```
class Program
{
    static void Main()
    {
        Abstraction abstraction;

        abstraction = new RefinedAbstraction(new ConcreteImplementorA());
        abstraction.Operation();

        abstraction = new RefinedAbstraction(new ConcreteImplementorB());
        abstraction.Operation();
    }
}
```

```
abstract class Abstraction
{
    protected Implementor implementor;

    public Abstraction(Implementor imp)
    {
        this.implementor = imp;
    }

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

```
abstract class Implementor
{
    public abstract void OperationImp();
}
```

```
class RefinedAbstraction : Abstraction
{
    public RefinedAbstraction(Implementor imp)
        : base(imp)
    {
    }

    public override void Operation()
    {
        // ...
        base.Operation();
        // ...
    }
}
```

```
class ConcreteImplementorA : Implementor
{
    public override void OperationImp()
    {
        Console.WriteLine("ImplementorA");
    }
}
```

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

См. Пример к главе: \007_Bridge\001_Bridge

Участники

- `Abstraction` - **Абстракция**:
  Предоставляет интерфейс для абстракции. Хранит ссылку на `Implementor`.

- `RefinedAbstraction` - **Уточненная абстракция**:
  Расширяет интерфейс, предоставляемый абстракцией.

- `Implementor` - **Реализатор**:
  Предоставляет интерфейс для реализации. Чаще всего класс `Implementor` предоставляет низкоуровневый интерфейс, а `Abstraction` предоставляет высокоуровневый интерфейс.

- `ConcreteImplementor` - **Конкретный реализатор**:
  Реализует интерфейс класса `Implementor`.

Отношения между участниками

Отношения между классами

- Абстрактный класс `Abstraction` связан связью отношения агрегации с абстрактным классом `Implementor`.

- Конкретный класс `RefinedAbstraction` связан связью отношения наследования с абстрактным классом `Abstraction`.

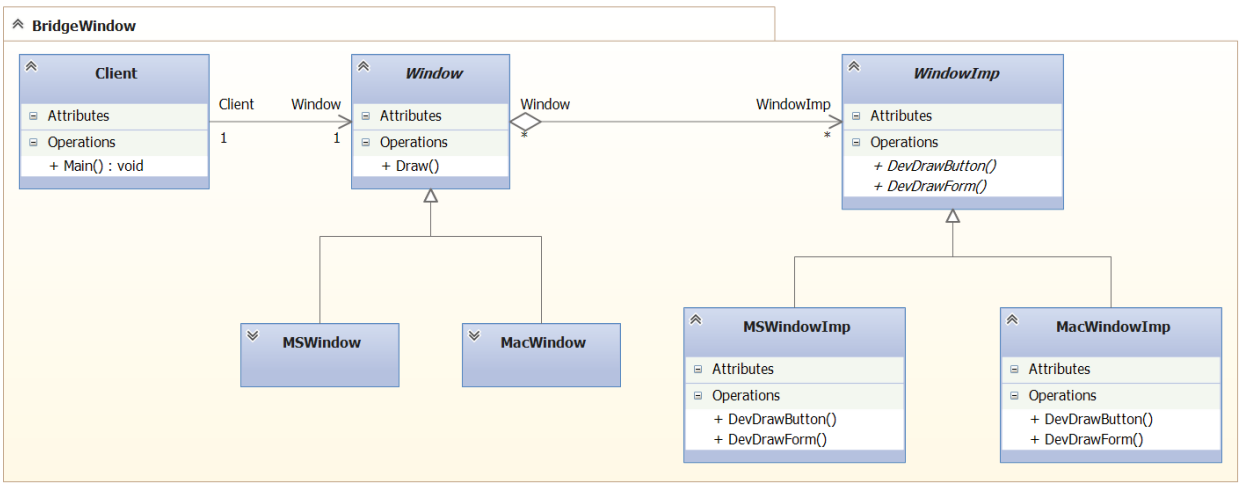
- Конкретные классы-реализаторы `ConcreteImplementorA` и `ConcreteImplementorB` связаны связью отношения наследования с абстрактным классом `Implementor`.

Отношения между объектами

- Объекты типа `Abstraction` перенаправляют запросы клиента объектам типа `Implementor`.

Мотивация

Предлагается рассмотреть такую разновидность абстракции, как «*переносимая абстракция*», когда абстракция и ее реализация помещаются в раздельные (параллельные или частично параллельные) иерархии классов. Рассмотрим использование переносимой абстракции на простом примере построения приложения, способного использовать разные стили пользовательского интерфейса (например, стиль Microsoft Windows и стиль Mac OS).



Создадим две иерархии, одну для интерфейсов окон (`Window`, `MSWindow` и `MacWindow`), а другую для реализаций элементов управления определенного стиля (`WindowImp`, `MSWindowImp` и `MacWindowImp`). Так, например, подкласс `MSWindowImp` предоставляет реализацию в стиле Microsoft Windows.

Все методы классов (`MSWindow` и `MacWindow`) производных от класса `Window`, используют методы из классов (`MSWindowImp` и `MacWindowImp`) производных от класса `WindowImp`. Такой подход отделяет абстракцию окна определенного стиля (например, `MSWindow`) от деталей реализации каждого отдельного элемента стиля (`Form`, `Button` и др.).

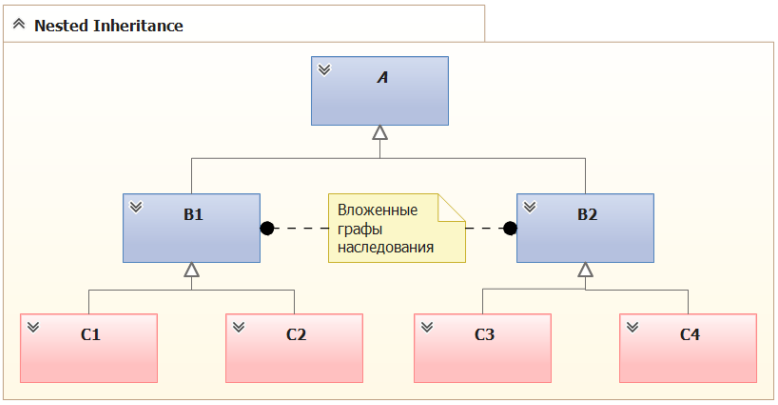
На диаграмме классов видно, что класс `Window` является вершиной иерархии *абстракций*, а класс `WindowImp` является вершиной иерархии *реализаций* (параллельной иерархии абстракций). Связь отношения агрегации между классами `Window` и `WindowImp` называется мостом. Мост между абстракцией (`Window`, `MSWindow` и `MacWindow`) и реализацией (`WindowImp`, `MSWindowImp` и `MacWindowImp`). Это позволяет изменять абстракцию и реализацию независимо друг от друга.

См. Пример к главе: \007_Bridge\002_BridgeMotivation

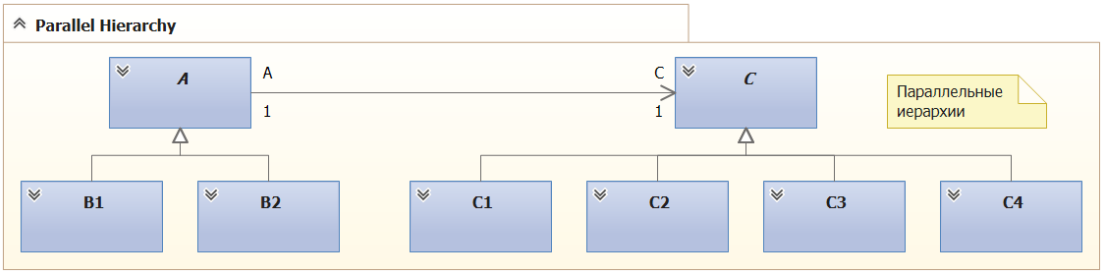
Применимость паттерна

Паттерн Bridge рекомендуется использовать, когда:

- Требуется избежать постоянной привязки абстракции к реализации. Иногда бывает необходимо выбирать нужную реализацию во время выполнения программы.
- Требуется предоставить возможность расширения новыми подклассами и абстракции, и реализации.
- Необходимо разделять одну реализацию между несколькими объектами и этот факт требуется скрыть от клиента. Примером этому может служить использование класса `String` и таблицы интернирования строк.
- Необходимо избавиться от графов наследований, вложенных в графы наследования.



Параллельная иерархия всегда предпочтительней вложенного графа наследования.



Результаты

Паттерн Bridge обладает следующими преимуществами:

- **Отделение реализации от абстракции.**
  Реализация (`Implementor`) не имеет привязки к абстрактному интерфейсу абстракции (`Abstraction`), как это могло бы быть в случае использования вложенных графов наследования. Реализацию абстракции (`RefinedAbstraction`) можно конфигурировать во время выполнения программы, просто подставляя объекты нужных классов (`ConcreteImplementorA` или `ConcreteImplementorB`). Разделение `Abstraction` и `Implementor` устраняет зависимости между абстракцией и реализацией во время компиляции, т.е., позволяет изменять абстракцию и реализацию независимо друг от друга.

- **Повышение степени расширяемости.**
  Имеется возможность независимо друг от друга расширять иерархии классов `Abstraction` и `Implementor`.

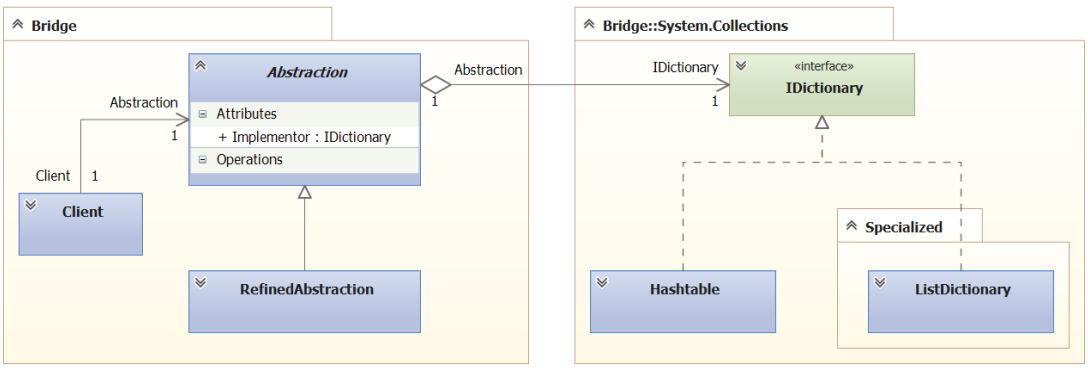
- **Сокрытие реализации от клиента.**
  От клиента можно скрыть наличие иерархии реализации (`Implementor`), предоставив только высокоуровневый интерфейс иерархии абстракции (`Abstraction`), за которым будет скрываться низкоуровневый интерфейс иерархии реализации (`Implementor`).

Реализация

Полезные приемы реализации паттерна Bridge:

- **Наличие только одного класса Implementor.**
  В том случае, если в программе имеется только одна реализация (`ConcreteImplementor`), то создавать абстрактный класс `Implementor` необязательно. Это частный случай использования паттерна Bridge, когда `RefinedAbstraction` ссылается на `ConcreteImplementor`. Тем не менее такое разделение полезно, так как позволит изменять `ConcreteImplementor`, при этом не перекомпилируя клиентскую часть кода.

- **Создание нужного объекта класса ConcreteImplementor.**
  Как принимается решение о том, экземпляр какого конкретного реализатора (`ConcreteImplementorA` или `ConcreteImplementorB`) требуется создать? Если у класса `RefinedAbstraction` имеется информация о классах `ConcreteImplementor`, то класс `RefinedAbstraction` в своем конструкторе может создать экземпляр нужного класса `ConcreteImplementor`. Например, если требуется создать коллекцию, поддерживающую несколько возможных реализаций, то решение о типе коллекции можно принять в зависимости от требуемого размера коллекции. Для хранения маленького числа элементов есть смысл создавать коллекцию типа `ListDictionary`, а для хранения большого числа элементов, коллекцию типа `Hashtable`.



См. Пример к главе: \007_Bridge\003_BridgeCollections

Пример кода

Предлагается рассмотреть пример, из раздела «Мотивация», где рассматривается структура приложения, использующего различные стили пользовательского интерфейса (например, стиль Microsoft Windows и стиль Mac OS). Рассмотрим класс `Window`, который стоит на вершине иерархии *абстракций* и задает высокоуровневую базовую абстракцию для окна клиентских приложений:

```
abstract class Window
{
    protected WindowImp implementor;
    protected Form form;
    protected Button button;

    // Operation
    public virtual void Draw()
    {
        this.form.Controls.Add(button);
        Application.EnableVisualStyles();
        Application.Run(this.form);
    }
}
```

Класс `Window` содержит поле `implementor` типа `WindowImp`. Тип `WindowImp` представлен в виде абстрактного класса, в котором задан интерфейс взаимодействия с оконной системой. `WindowImp` является вершиной иерархии *реализаций*:

```
abstract class WindowImp
{
    protected Button button;
    protected Form form;
    public abstract Form DevDrawForm();
    public abstract Button DevDrawButton();
}
```

Классы `MSWindow` и `MacWindow`, производные от класса `Window` являются представителями иерархии *абстракций* и определяют разнообразные варианты окон с различным внешним видом форм и кнопок:

```
class MacWindow : Window
{
    public MacWindow()
    {
        this.implementor = new MacWindowImp();
        this.form = this.implementor.DevDrawForm();
        this.button = this.implementor.DevDrawButton();
    }

    // Operation
    public override void Draw()
    {
        form.FormBorderStyle = System.Windows.Forms.FormBorderStyle.Fixed3D;
        base.Draw();
    }
}
```

```
class MSWindow : Window
{
    public MSWindow()
    {
        this.implementor = new MSWindowImp();
        this.form = this.implementor.DevDrawForm();
        this.button = this.implementor.DevDrawButton();
    }

    // Operation
    public override void Draw()
    {
        form.FormBorderStyle = System.Windows.Forms.FormBorderStyle.FixedDialog;
        base.Draw();
    }
}
```

Конкретными представителями иерархии реализаций являются подклассы MacWindowImp и MSWindowImp класса WindowImp:

```
class MacWindowImp : WindowImp
{
    public override Form DevDrawForm()
    {
        this.form = new Form();
        this.form.AutoScaleDimensions = new SizeF(6F, 13F);
        this.form.AutoScaleMode = AutoScaleMode.Font;
        this.form.ClientSize = new Size(284, 172);

        this.form.Name = "Mac Form";
        this.form.Text = "Mac OS - Snow Leopard";
        this.form.BackColor = Color.White;
        return this.form;
    }

    public override Button DevDrawButton()
    {
        this.button = new Button();
        this.button.Location = new Point(75, 70);
        this.button.Size = new Size(125, 25);
        this.button.Text = "Leopard";
        this.button.ForeColor = Color.White;
        this.button.BackColor = Color.LightGray;

        return this.button;
    }
}

class MSWindowImp : WindowImp
{
    public override Form DevDrawForm()
    {
        this.form = new Form();
        this.form.AutoScaleDimensions = new SizeF(6F, 13F);
        this.form.AutoScaleMode = AutoScaleMode.Font;
        this.form.ClientSize = new Size(284, 172);

        this.form.Name = "Microsoft Form";
        this.form.Text = "Windows Explorer";
```

```
            this.form.BackColor = Color.LightBlue;

            return this.form;
        }

        public override Button DevDrawButton()
        {
            this.button = new Button();
            this.button.Location = new Point(75, 70);
            this.button.Size = new Size(125, 25);
            this.button.Text = "Windows";
            this.button.ForeColor = Color.Aqua;
            this.button.BackColor = Color.DarkBlue;

            return this.button;
        }
    }
```

Паттерн Composite

Название

Компоновщик

Также известен как

-

Классификация

По цели: структурный
По применимости: к объектам

Частота использования

Выше средней - 1 2 3 **4** 5

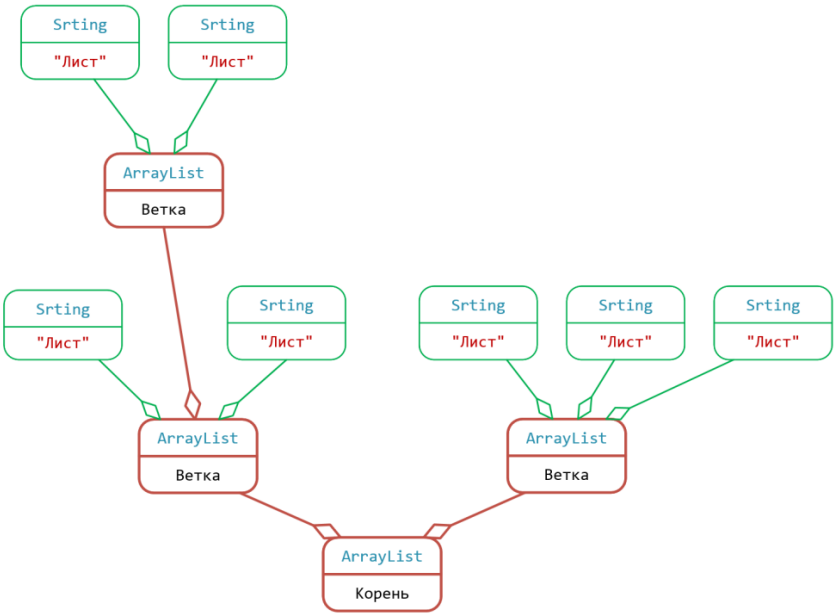
Назначение

Паттерн Composite – составляет из объектов древовидные структуры для представления иерархий «часть – целое». Позволяет клиентам единообразно трактовать индивидуальные объекты (листья) и составные объекты (ветки).

Введение

Паттерн Composite используется в первую очередь для построения специальных структур данных, которые в информатике называются – деревьями. Эти структуры данных называются деревьями, потому что их формальные изображения, напоминают деревья из объективной реальности.

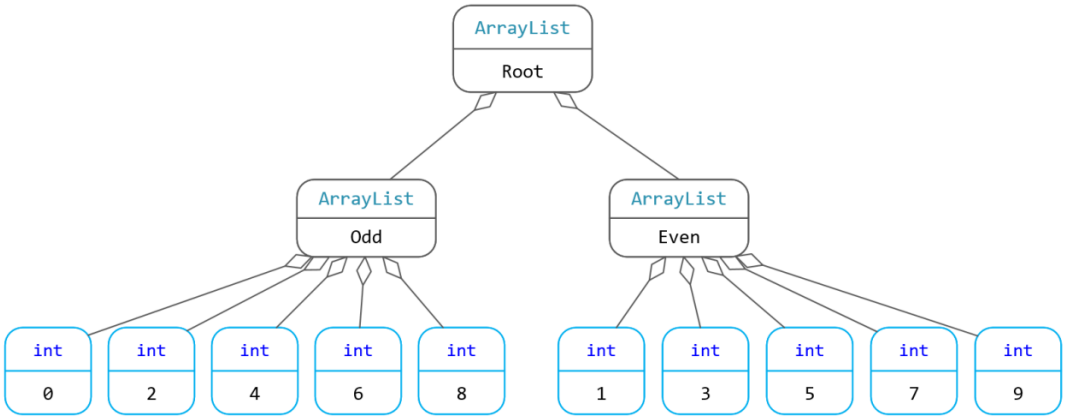
Чтобы построить простейшее дерево, можно воспользоваться двумя классами – `ArrayList` и `String`. Корень и ветки дерева можно представить экземплярами класса (коллекции) `ArrayList`, а листья экземплярами класса `String`. Из корня и веток могут расти как листья, так и другие ветки (аналогично живому дереву). Из листьев ничего произрастать не может.



Но, в виртуальной реальности деревья растут сверху вниз, такая особенность представления *«роста»* деревьев обусловлена адресацией памяти (ОЗУ). В те времена, когда в программировании не использовались объектно-ориентированные подходы, ветви деревьев представлялись массивами, а листья

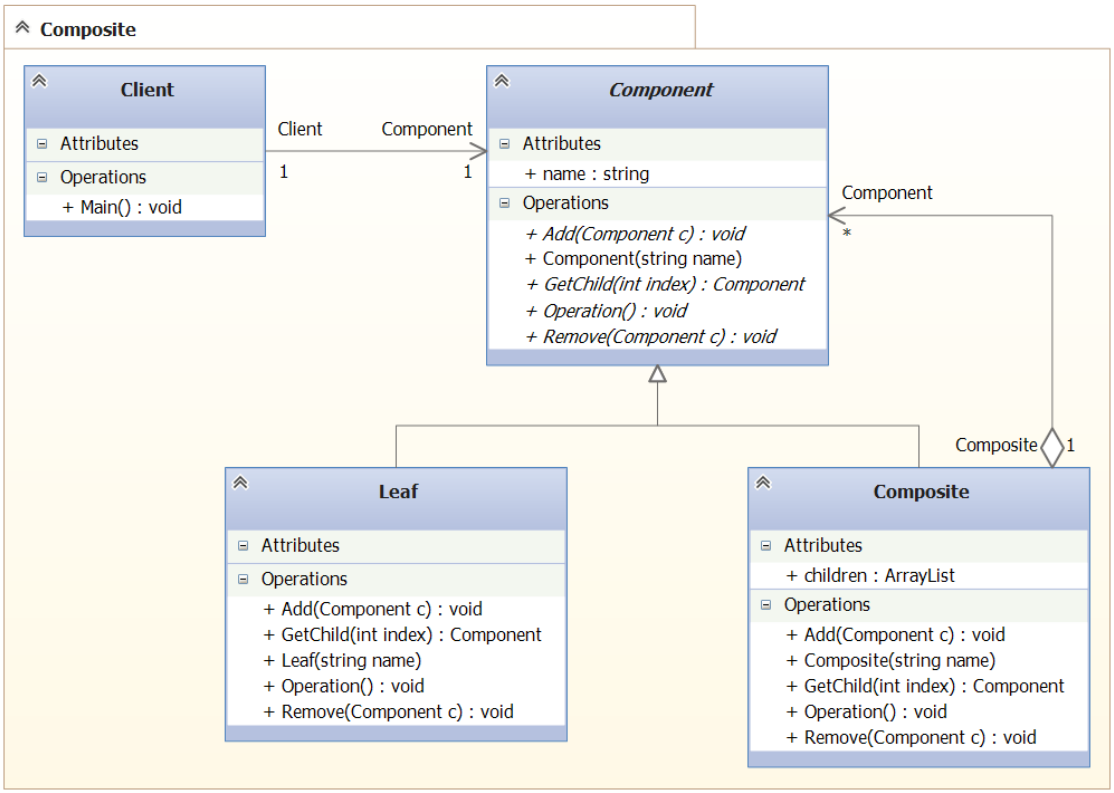
– элементами этих массивов. Как известно, традиционный массив располагается в памяти начиная с области младших адресов и заканчивается в области старших адресов.

Так, например, может формально выглядеть дерево представления четных и нечетных чисел.



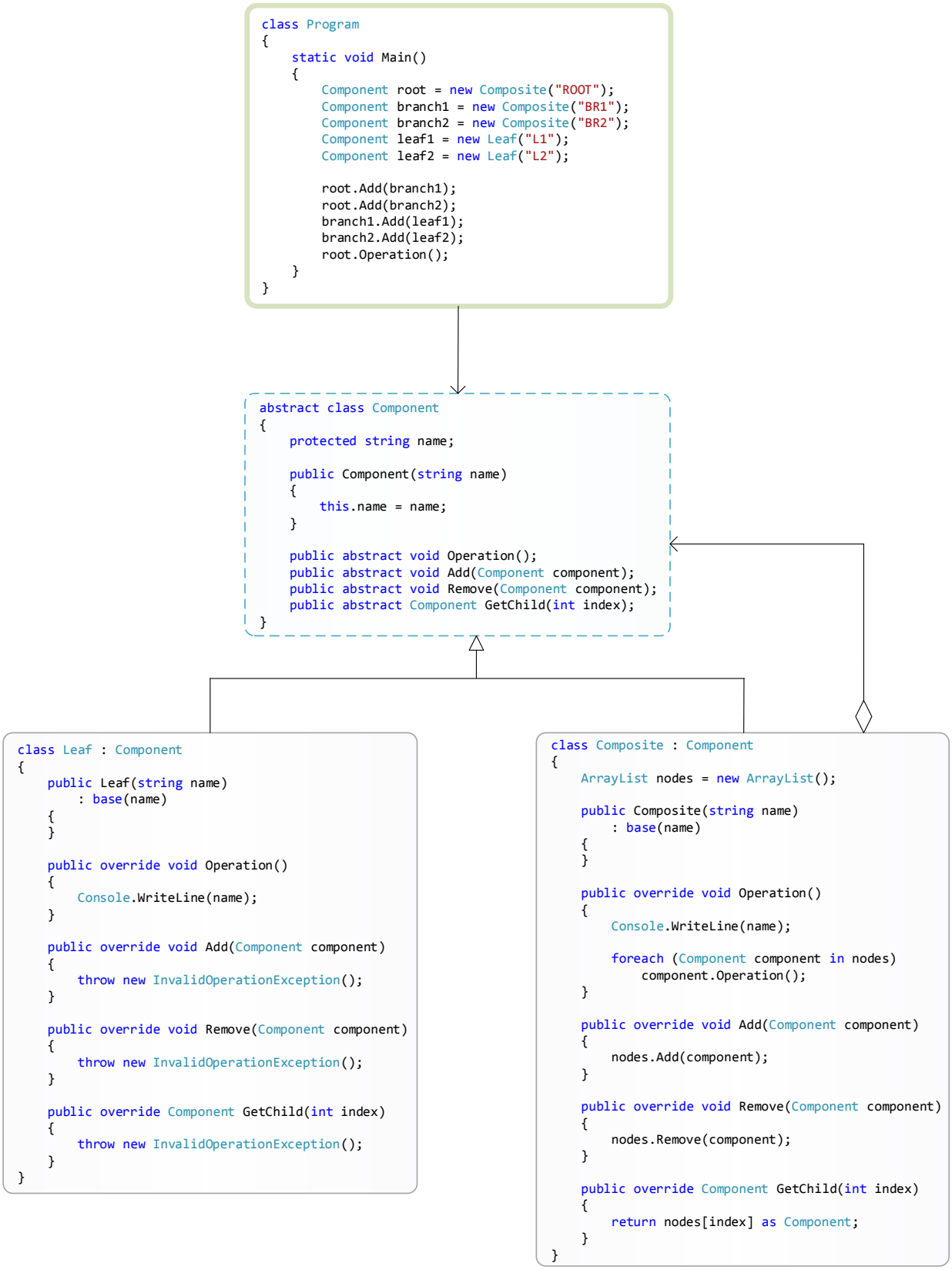
Более детально познакомиться с разновидностями деревьев, способами их построения и обхода, можно в книге Дональда Кнута – Искусство программирования (Глава 2.3. Деревья).

Структура паттерна на языке UML



См. Пример к главе: \008_Composite\001_Composite

Структура паттерна на языке C#



```
class Program
{
    static void Main()
    {
        Component root = new Composite("ROOT");
        Component branch1 = new Composite("BR1");
        Component branch2 = new Composite("BR2");
        Component leaf1 = new Leaf("L1");
        Component leaf2 = new Leaf("L2");

        root.Add(branch1);
        root.Add(branch2);
        branch1.Add(leaf1);
        branch2.Add(leaf2);
        root.Operation();
    }
}
```

```
abstract class Component
{
    protected string name;

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

    public abstract void Operation();
    public abstract void Add(Component component);
    public abstract void Remove(Component component);
    public abstract Component GetChild(int index);
}
```

```
class Leaf : Component
{
    public Leaf(string name)
        : base(name)
    {
    }

    public override void Operation()
    {
        Console.WriteLine(name);
    }

    public override void Add(Component component)
    {
        throw new InvalidOperationException();
    }

    public override void Remove(Component component)
    {
        throw new InvalidOperationException();
    }

    public override Component GetChild(int index)
    {
        throw new InvalidOperationException();
    }
}
```

```
class Composite : Component
{
    ArrayList nodes = new ArrayList();

    public Composite(string name)
        : base(name)
    {
    }

    public override void Operation()
    {
        Console.WriteLine(name);

        foreach (Component component in nodes)
            component.Operation();
    }

    public override void Add(Component component)
    {
        nodes.Add(component);
    }

    public override void Remove(Component component)
    {
        nodes.Remove(component);
    }

    public override Component GetChild(int index)
    {
        return nodes[index] as Component;
    }
}
```

См. Пример к главе: \008_Composite\001_Composite

Участники

- `Component` - **Компонент**:
  Предоставляет интерфейс для объектов из которых составляется дерево. В частном случае может предоставлять реализацию по умолчанию для некоторых методов.

- `Leaf` - **Лист**:
  Является классом листовых узлов дерева и не может иметь потомков, т.е., включать в себя объекты относящиеся к структуре дерева (из листа не может вырасти ветвь или другой лист).

- `Composite` - **Составной объект**:
  Задает поведение объектов, входящих в структуру дерева, у которых есть потомки, а также сам хранит в себе компоненты дерева (объекты потомки), как узловые, так и листовые. Реализует методы интерфейса `Component`, относящиеся к управлению потомками.

- `Client` - **Клиент**:
  Манипулирует объектами, входящими в структуру дерева, через интерфейс, предоставляемый классом `Component`.

Отношения между участниками

Отношения между классами

- Класс `Leaf` связан связью отношения наследования с абстрактным классом `Component`.

- Класс `Composite` связан связью отношения наследования и связью отношения агрегации с абстрактным классом `Component`.

Отношения между объектами

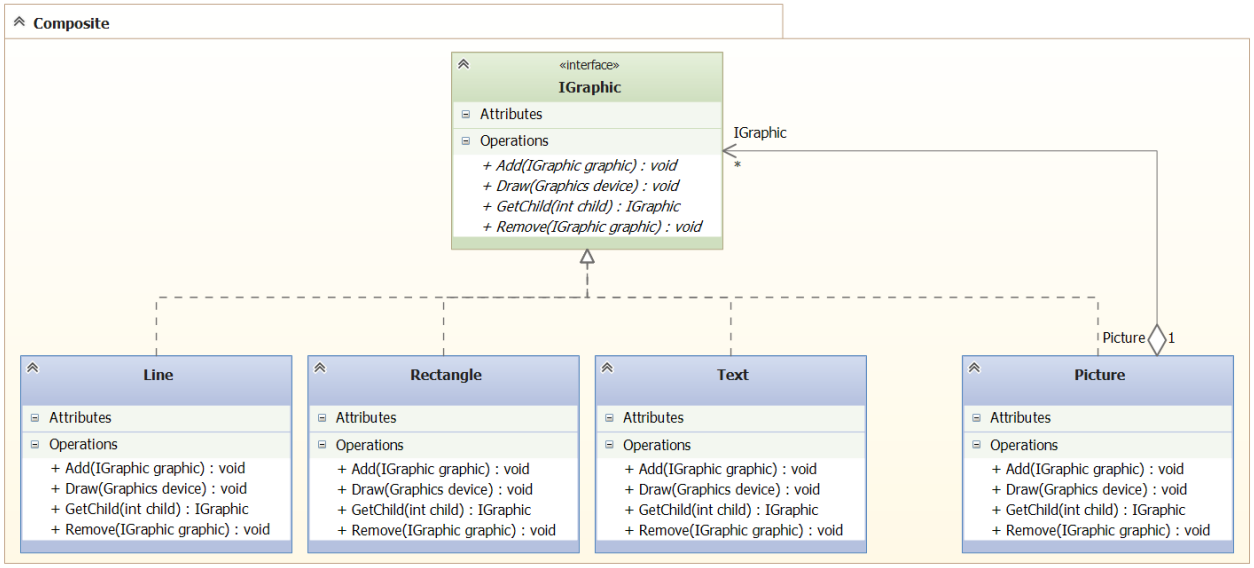
- Клиенты (`Client`) взаимодействуют с элементами, входящими в состав дерева, через интерфейс (набор методов) предоставленный абстрактным классом `Component`.

- Если метод вызывается на листовом объекте (`Leaf`), то объект `Leaf` и обрабатывает поступивший запрос (т.е. выполняет функциональность, заложенную в тело его метода).

- Если метод вызывается на составном (узловом) объекте (`Composite`), то объект `Composite` перенаправляет запрос (вызывает метод) своим потомкам, возможно при этом выполняя некоторые дополнительные действия до или после перенаправления запроса.

Мотивация

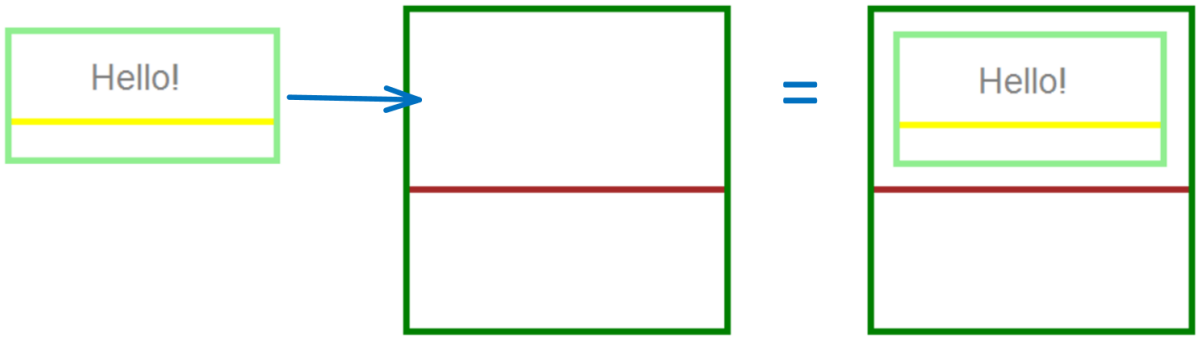
В качестве примера работы с деревом, предлагается рассмотреть построение простейшего редактора векторной графики. Интерфейс `IGraphic` будет являться общим типом как для примитивных объектов (`Line`, `Rectangle`, `Text`), так и для контейнеров (`Picture`).



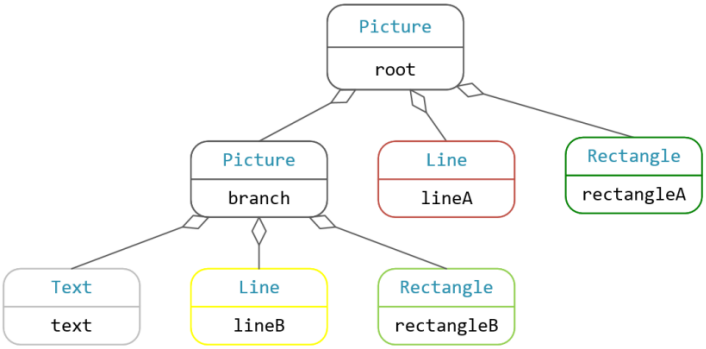
Классы `Line`, `Rectangle`, `Text`, будут представлять примитивные графические объекты, из которых формируется картинка типа `Picture`. В этих классах реализован метод `Draw` для рисования соответствующих геометрических фигур, но поскольку эти классы представляют листовой уровень дерева, то они не могут иметь потомков, т.е. вложенных в них объектов, относящихся к структуре дерева (из листа не может вырасти ветвь или другой лист). Соответственно классы листового уровня не могут полноценно реализовать методы `Add`, `Remove` и `GetChild` интерфейса `IGraphic`, поэтому в их телах будет возбуждаться исключение типа `InvalidOperationException`. Кому-то такая реализация методов покажется не совсем правильной, но эта особенность будет рассмотрена далее достаточно подробно и ее следует принять как должное, несмотря на несоответствие интерфейсов.

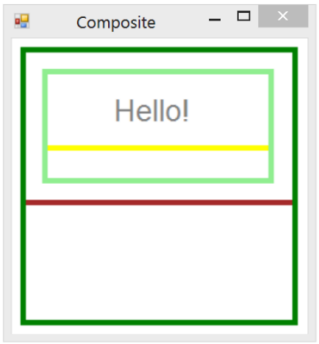
Объекты класса `Picture`, являются узлами (ветвями) в контексте дерева. Класс `Picture`, полноценно реализует все методы интерфейса `IGraphic`. Реализованный в классе `Picture` метод `Draw`, вызывает методы `Draw`, на каждом элементе который входит в состав объекта класса `Picture`.

Объекты класса `Picture` в программе, будут представлять собой объектно-ориентированное представление векторной картинки, которая может состоять как из примитивных объектов типа `Line`, `Rectangle`, `Text`, так и других картинок типа `Picture`. Ниже на рисунке можно увидеть, как большая картинка принимает в себя меньшую картинку.



Ниже на диаграмме показана структура дерева, скомпонованного из объектов типа `IGraphic`, и результат выполнения программы. На диаграмме, цвета рамок представлений объектов в дереве соответствуют цветам фигур на форме, что позволяет отследить работу каждого объекта.





См. Пример к главе: \008_Composite\002_GraphicDesigner

Применимость паттерна

Паттерн Composite рекомендуется использовать, когда:

- Требуется представить иерархию объектов в виде дерева и управлять группами объектов, входящих в эту иерархию.

- Требуется предоставить клиентам возможность единообразно трактовать (использовать) составные (ветви) и индивидуальные (листья) объекты.

Результаты

Паттерн Composite обладает следующими преимуществами:

- **Позволяет строить иерархии, состоящие из примитивных и составных объектов.**
  Из примитивных объектов (`Leaf`) можно строить сложные объекты (`Composite`), из которых в свою очередь можно строить еще более сложные объекты (`Composite`).

- **Позволяет упростить работу клиента (клиентский код).**
  Клиенты могут работать единообразно как с индивидуальными объектами (`Leaf`), так и с составными структурами (`Composite`). В идеале, клиент не должен знать с каким уровнем объектов он взаимодействует с листовым (`Leaf`) или с составным (`Composite`). Такой подход упрощает код клиента.

- **Облегчает добавление подклассов существующих компонентов.**
  Новые подклассы (`SubLeaf`, `SubComposite`) производные от классов `Leaf` и `Composite` смогут без дополнительной адаптации начать работать с существующим деревом, в котором имеются элементы типа `Leaf` и `Composite`. Изменять код клиента при этом может не понадобиться (если клиент работает с уже существующим деревом, которое он сам не строил).

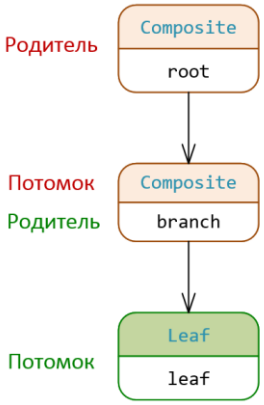
Паттерн Composite обладает следующим недостатком:

- **Отсутствие контроля классов объектов, которые могут входить в определенное дерево.**
  Простота добавления в дерево новых типов компонентов имеет и отрицательную сторону. Иногда требуется наложить ограничение на то, объекты каких классов производных от класса `Component`, могут входить в состав дерева. Паттерн Composite не предоставляет возможности автоматически производить такой контроль. Для этого может понадобиться самостоятельно производить проверку типов, например, с использованием оператора `is` языка C#.

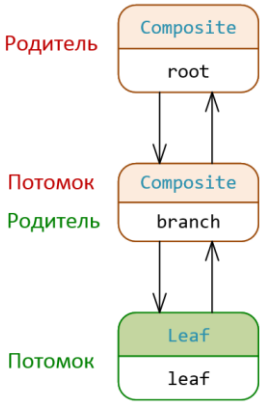
Реализация

Полезные приемы реализации паттерна Composite:

- **Наличие ссылок на родителей.**
  В простейшей форме дерева родительские элементы содержат в себе ссылки на дочерние элементы (т.е., ветви содержат ссылки на листья или на другие *«более тонкие»* ветви). Деревья, которые имеют одностороннюю направленность, еще называют - *ациклическими графами*. Но иногда требуется организовать дерево, в котором определенные потомки содержат ссылки на своих родителей (лист содержит ссылку на ветвь). Деревья, которые имеют двустороннюю направленность, еще называют - *циклическими графами*. На рисунке ниже показан пример ациклического и циклического графов.



Ациклический граф



Циклический граф

Обычно поле-ссылка на родителя создается один раз в базовом абстрактном классе `Component`. Классы `Leaf` и `Composite` наследуют это поле-ссылку, а также возможные методы по работе с этой ссылкой.

- **Максимизация интерфейса класса Component.**
  Одна из целей паттерна Composite – скрыть от клиента информацию о том, с каким уровнем объектов они работают (листовыми или составными). Для этого класс `Component` должен содержать как можно больше методов, общих для классов `Leaf` и `Composite`. Класс `Component` может предоставлять методы по умолчанию, а классы `Leaf` и `Composite` могут замещать или переопределять эти методы. Однако такой подход иногда может противоречить принципу проектирования иерархии классов, согласно которого, класс должен содержать только логичные методы для него самого и всех его подклассов.

- **Создание и использование методов для работы с потомками.**
  Класс `Component` содержит набор методов, не имеющих смысла для класса `Leaf`. Рекомендуется рассматривать класс `Leaf` как полноценный компонент дерева, у которого не может быть потомков. Лист живого дерева, является полноценным компонентом, входящим в состав дерева, при этом из листа не может вырасти новый лист или ветвь.

  Можно было бы в классе `Component` создать методы доступа к потомкам с реализацией по умолчанию – «никогда не работать с потомком» (например, оставить тело метода пустым). Тогда класс `Leaf` сможет использовать методы по умолчанию из базового класса `Component`, а в классе `Composite` можно переопределить эти методы. Но подход с пустыми телами методов, будет не правильным.

  Возникает вопрос: Нужно ли создавать методы по работе с потомками в базовом классе `Component`, делая их доступными для класса `Leaf`, или такие методы следует создавать только в классе `Composite`? Прежде чем ответить на этот вопрос, нужно рассмотреть такие понятия как прозрачность и безопасность, и выбрать что-то одно из них.

  Если задать интерфейс (набор методов) по работе с потомками в корне иерархии (в базовом классе `Component`), то можно добиться прозрачности, так как все компоненты дерева можно трактовать (использовать) единообразно. Однако при таком подходе придется пожертвовать безопасностью, поскольку клиент может попытаться вызвать метод по работе с потомками на элементе листового уровня (например, попытаться добавить или удалить элемент).

Если интерфейс по работе с потомками задать исключительно в классе `Composite`, то безопасность удастся обеспечить (любая попытка добавить или удалить элемент на объекте класса `Leaf`, приведет к ошибке уровня компиляции), но при таком подходе теряется прозрачность, так как у `Leaf` и `Composite` будут различные интерфейсы.

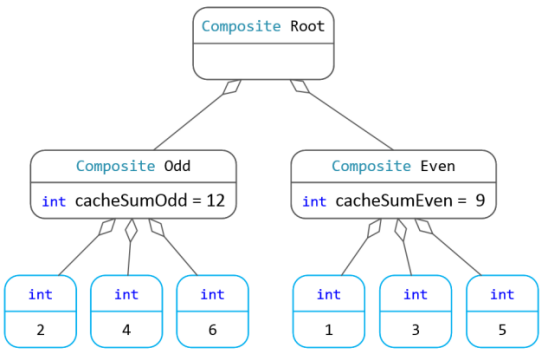
В паттерне Composite предпочтение отдается прозрачности, а не безопасности. Единственный способ обеспечить прозрачность – это включить в класс `Component` методы Add, `Remove` и `GetChild`. При реализации данных методов в классе `Leaf`, можно было бы сделать реализацию данных методов пустой, но такой подход будет вводить разработчиков в заблуждение (Будет получаться так, что в программном коде объект, казалось бы, добавлен в дерево, а при выполнении в дереве этого объекта нет, или в программном коде объект удален из дерева, а при выполнении объект остался в дереве.). Лучшим решением будет такая реализация методов Add, `Remove` и `GetChild`, при которой вызовы этих методов будут завершаться ошибкой, возбуждая исключение типа `InvalidOperationException`.

- **Организация порядка следования элементов (потомков) в дереве.**
  В большинстве случаев важен порядок следования элементов в дереве. Например, может понадобиться организовать расположение элементов в Z-порядке (Аналогично Z-порядку расположения блочных элементов, перекрывающих друг друга в HTML. Или порядку перекрытия фигур в PowerPoint или Visio.). Если порядок следования элементов в дереве важен, то этот порядок необходимо учитывать при проектировании интерфейсов (методов) доступа.

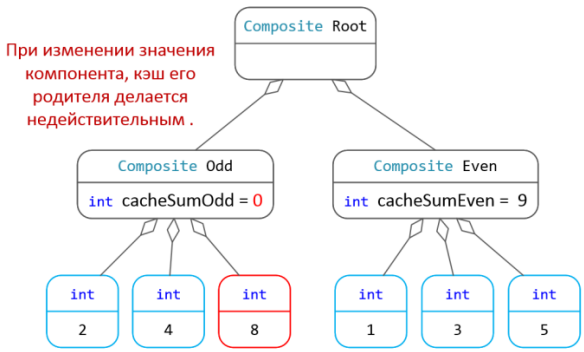
- **Кэширование для повышения производительности.**
  Если приходится часто обходить дерево или производить в нем поиск элементов, то есть смысл организовать в классе `Composite` механизм кэширования, который будет сохранять информацию об обходе или поиске. Для организации механизма кэширования можно использовать техники мемоизации, или воспользоваться готовым решением: библиотекой сквозной функциональности - Enterprise Library.

  Кэшировать рекомендуется либо полученные результаты, либо только информацию, позволяющую ускорить обход или поиск.

  При изменении компонента, кэш всех его родителей делается недействительным. Такой подход может быть наиболее эффективным, когда компоненты знают о своих родителях. Поэтому при организации механизмов кеширования, требуется предусмотреть интерфейс (набор методов), позволяющий уведомлять родителей о недействительности их кэшей.



До изменения



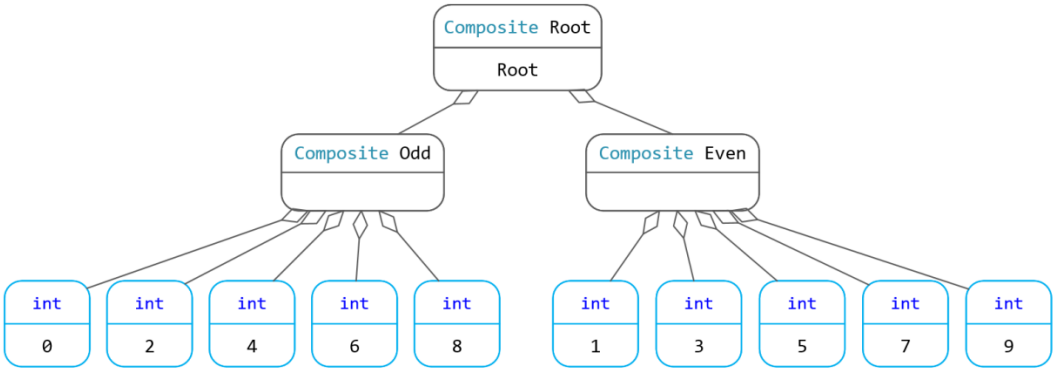
После изменения

- **Какая разновидность коллекций лучше всего подходит для хранения компонентов дерева?**
  Родительские узлы деревьев (объекты класса `Composite`) могут хранить ссылки на своих потомков в самых разных разновидностях массивов и коллекций. Для хранения ссылок на потомков могут использоваться: массивы, списки, связанные списки, хэш-таблицы, словари и прочие разновидности коллекций. Выбор нужной коллекции определяется эффективностью ее использования. В частном случае родительский узел может хранить ссылки на своих потомков в обычных полях (переменных).

- **Рекурсивная композиция.**
  Паттерн Composite описывает правильные способы построения именно объектно-ориентированных представлений древовидных структур и способов их обхода. Построить дерево можно как вручную

(пошагово добавляя узлы), так и с использованием техник рекурсивной композиции. В некоторых языках имеется специальный оператор для поддержки техники рекурсивной композиции, но в языке C# такого оператора нет. В языке C#, рекурсивная композиция может быть реализована через использование рекурсивных вызовов функций. Эффект применения рекурсивной композиции заключается в том, что клиенту при работе с деревом не придется производить различий между простыми объектами (листьями) и составными объектами (узлами), это значит, что клиент сможет работать со всеми компонентами дерева единообразно. При ручном подходе построения дерева – единообразность практически не дает преимуществ, так как приходится вникать в структуру дерева и понимать тип того или иного компонента.

Предлагается рассмотреть пример построения дерева с использованием техники рекурсивной композиции. Дерево состоит из двух узлов (ветвей). Каждый из узлов содержит элементы (листья) представляющие собой четные и нечетные числа. Структура дерева представлена ниже на диаграмме.



См. Пример к главе: \008_Composite\003_RecursionComposition

Известные применения паттерна в .Net

`System.ServiceModel.Channels.CompositeDuplexBindingElement`
http://msdn.microsoft.com/ru-ru/library/system.servicemodel.channels.compositeduplexbindingelement(v=vs.110).aspx

`System.Web.UI.Design.WebControls.CompositeControlDesigner`
http://msdn.microsoft.com/ru-ru/library/system.web.ui.design.webcontrols.compositecontroldesigner(v=vs.90).aspx

`System.Web.UI.WebControls.CompositeControl`
http://msdn.microsoft.com/en-us/library/system.web.ui.webcontrols.compositecontrol(v=vs.110).aspx

`System.Windows.Data.CompositeCollection`
http://msdn.microsoft.com/ru-ru/library/system.windows.data.compositecollection(v=vs.110).aspx

`System.Web.UI.Design.WebControls.TreeViewDesigner`
http://msdn.microsoft.com/ru-ru/library/system.web.ui.design.webcontrols.treeviewdesigner(v=vs.110).aspx

`System.Web.UI.WebControls.TreeNodeCollection`
http://msdn.microsoft.com/ru-ru/library/system.web.ui.webcontrols.treenodecollection(v=vs.110).aspx

`System.Web.UI.WebControls.TreeView`
http://msdn.microsoft.com/en-us/library/system.web.ui.webcontrols.treeview(v=vs.110).aspx

`System.Windows.Controls.TreeView`
http://msdn.microsoft.com/en-us/library/system.windows.controls.treeview.aspx

`System.Windows.Forms.TreeNodeCollection`
http://msdn.microsoft.com/ru-ru/library/system.windows.forms.treenodecollection(v=vs.110).aspx

Паттерн Decorator

Название

Декоратор

Также известен как

Wrapper (Обертка)

Классификация

По цели: структурный
По применимости: к объектам

Частота использования

Средняя - 1 2 **3** 4 5

Назначение

Паттерн Decorator - динамически (в ходе выполнения программы) добавляет объекту новые возможности (состояние и/или поведение). Композиция, используемая при реализации паттерна Decorator, является гибкой альтернативой наследованию (порождению подклассов) с целью расширения функциональности.

Введение

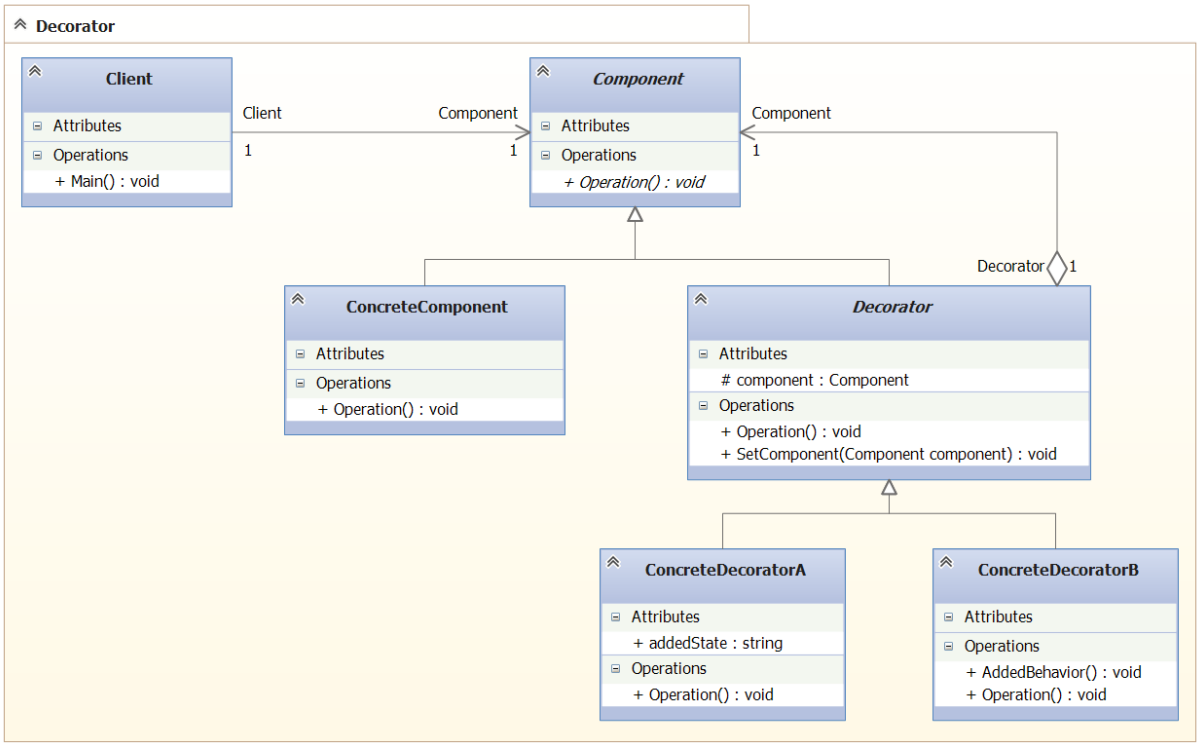
Иногда требуется расширить функциональные возможности только одного объекта, а не всего класса в целом. Добавить новую функциональность (обязанности) можно при помощи наследования, но наследование является статической техникой расширения и соответственно при наследовании теряется гибкость. Более гибким подходом добавления функциональности (обязанностей), является техника *декорирования* объекта. Декорирование следует понимать не как украшение, а как способ добавления функциональности (обязанностей).

Для понимания *декорирования*, предлагается рассмотреть простую аналогию из жизни. Возьмем к примеру человека, который по профессии – летчик истребитель.

Для того чтобы летчик истребитель мог выполнять боевое задание – охранять воздушные границы государства, самого летчика недостаточно. Потребуются следующие компоненты: специальный (высотно-компенсирующий) костюм и самолет СУ-47. Для организации полета потребуется летчика поместить в костюм, а костюм поместить в самолет. Обратите внимание, летчик не касается своим телом кресла самолета, так же, все взаимодействие с самолетом происходит через костюм. Получается некая цепочка связанных объектов: *самолет – костюм – летчик*. Почему именно такая последовательность? Потому что для вражеского пилота, целью в бою будет именно самолет Су-47, а не летчик, который управляет самолетом. Поэтому, если представить такую ситуацию объектно-ориентировано, то получается, что объект-

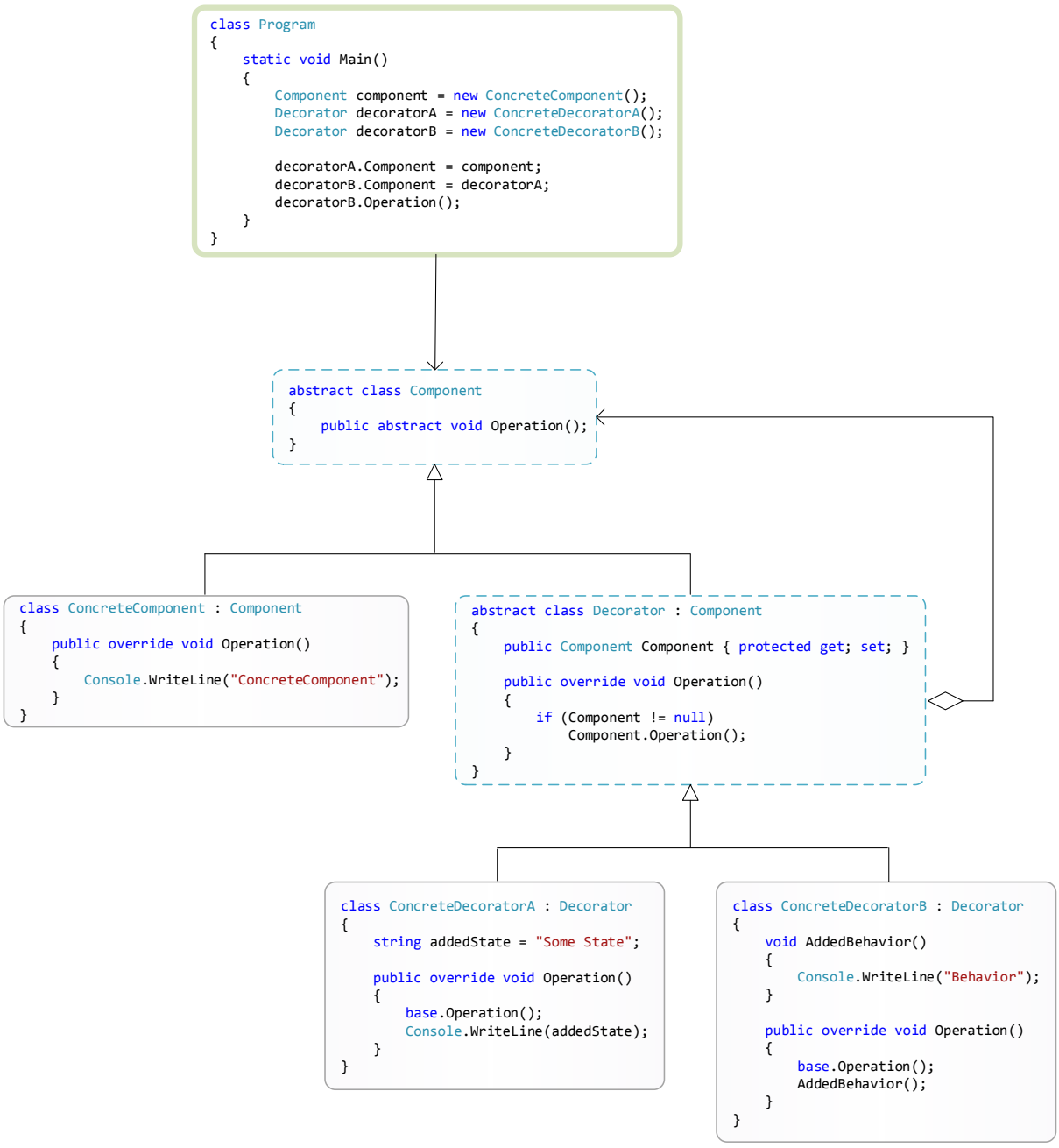
самолет ссылается на объект-костюм, а объект-костюм ссылается на объекта-летчика (по-другому можно сказать, летчик *декорирован* костюмом, а костюм *декорирован* самолетом). Если представить наоборот (*летчик – костюм - самолет*), то получится, что движущая сила летчика тащит за собой костюм, а костюм тащит за собой самолет, что может показаться вздорным.

Структура паттерна на языке UML



См. Пример к главе: \009_Decorator\001_Decorator

Структура паттерна на языке C#



См. Пример к главе: \009_Decorator\001_Decorator

Участники

- `Component` - **Компонент**:
  Предоставляет интерфейс для объектов, которые могут быть декорированы.

- `ConcreteComponent` - **Конкретный компонент**:
  Является классом целевого объекта, который должен быть декорирован.

- `Decorator` - **Декоратор**:
  Предоставляет общий интерфейс для объектов-декораторов.

- `ConcreteDecorator` - **Конкретный декоратор**:
  Содержит дополнительную функциональность, которой будет расширяться целевой объект.

Отношения между участниками

Отношения между классами

- Конкретный класс `ConcreteComponent` связан связью отношения наследования с абстрактным классом `Component`.

- Абстрактный класс `Decorator` связан связью отношения наследования и связью отношения агрегации с абстрактным классом `Component`.

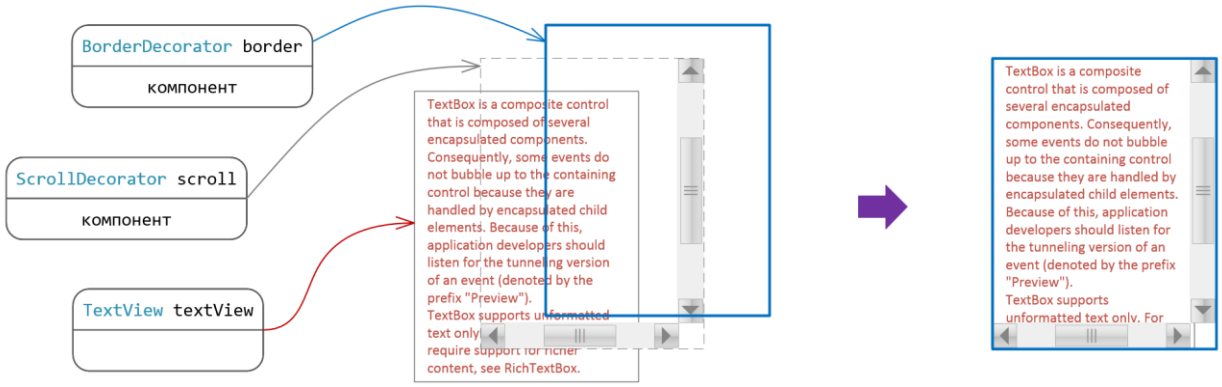
- Конкретные классы `ConcreteDecoratorA` и `ConcreteDecoratorB` связанны связью отношения наследования с абстрактным классом `Decorator`.

Отношения между объектами

- Объекты типа `Decorator` (`ConcreteDecorator`) переадресуют запросы объектам типа `Component` (`ConcreteComponent`, `ConcreteDecorator`) и выполняют дополнительные операции до или после переадресации.

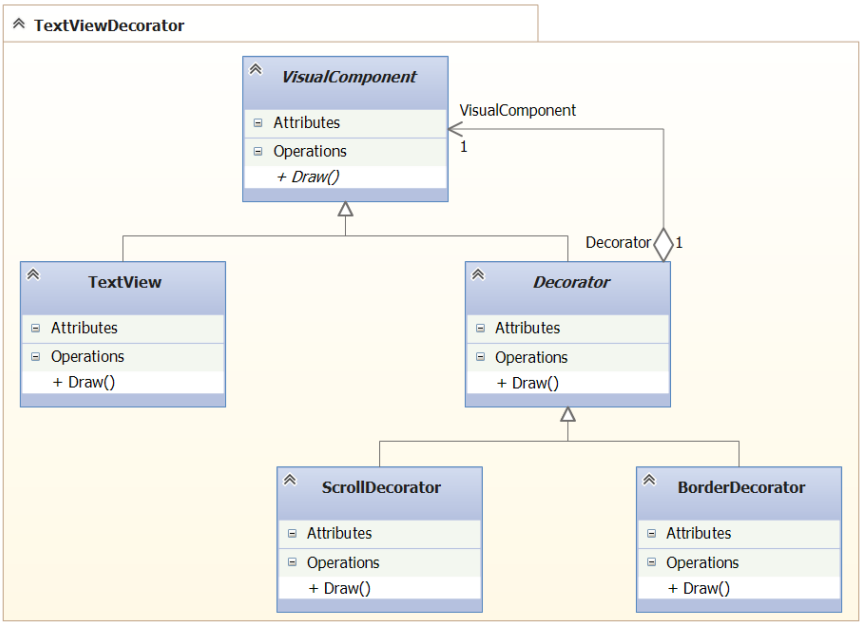
Мотивация

Рассмотрим использование техники декорирования на примере простейшей программы для построения графических интерфейсов. Скажем, программа должна уметь добавлять новое состояние (рамку) или поведение (возможность прокрутки) к элементу управления `TextView` (`TextBox`), который отображает текст в окне. По умолчанию `TextView` не имеет полос прокрутки, поскольку они не всегда нужны. Если полосы прокрутки понадобятся, их можно добавить при помощи декоратора `ScrollDecorator`. Если понадобится добавить рамку вокруг объекта `TextView`, ее можно добавить при помощи декоратора `BorderDecorator`. Также возможно одновременно использовать оба декоратора, тогда отображаемый объект будет иметь и полосы прокрутки, и рамку. На рисунке ниже можно увидеть графическое представление частей результата.



На диаграмме ниже показан объект класса `TextView` декорированный объектами классов `BorderDecorator` и `ScrollDecorator`.

Классы `ScrollDecorator` и `BorderDecorator` являются подклассами абстрактного класса `Decorator`. Класс `VisualComponent` – это абстрактный класс для представления всех визуальных объектов.



Классы `ScrollDecorator` и `BorderDecorator` могут добавлять любые методы для предоставления необходимой функциональности и использоваться (*декорировать*) со всеми объектами типа `VisualComponent`.

См. Пример к главе: \009_Decorator\002_Text View

Применимость паттерна

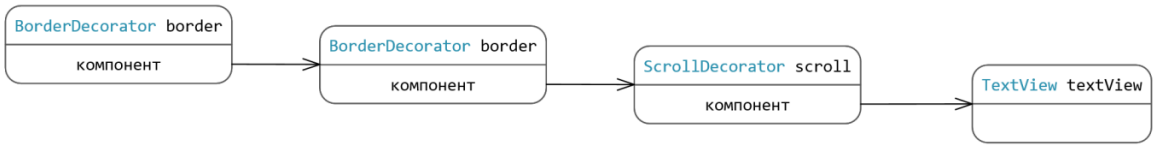
Паттерн Decorator рекомендуется использовать, когда:

- Требуется организовать возможность динамического (во время выполнения) расширения объектов.
- Требуется временно расширить объект новой функциональностью, а позже эту функциональность убрать.

Результаты

Паттерн Decorator обладает следующими преимуществами:

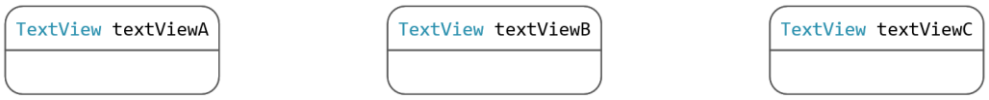
- **Гибкость.**
  Гибкость – это свойство программной системы или компонента, позволяющие быстро, легко и безопасно вносить изменения в существующий код или расширять код новой функциональностью. Паттерн Decorator позволяет гибко расширять существующий объект новой функциональностью. Гибкость достигается за счет использования динамических связей отношения, а именно связей отношения агрегации. Паттерн Decorator позволяет добавлять и удалять функциональность во время выполнения программы. Одно и тоже свойство может быть добавлено несколько раз.
  Например, можно применить к объекту `TextView` двойную рамку, для этого нужно просто добавить два декоратора типа `BorderDecorator`.



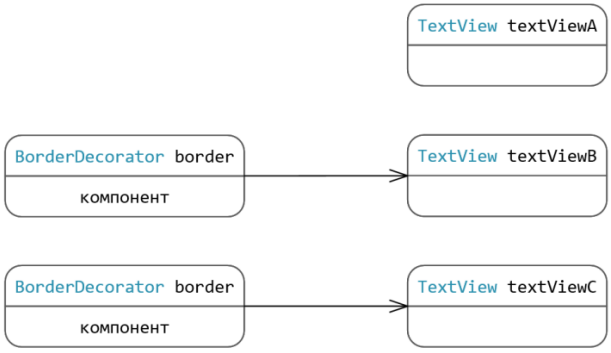
- **Необходимая достаточность.**
  Паттерн Decorator позволяет организовать динамическое добавление новой функциональности объектам по мере необходимости. Вместо того, чтобы пытаться предусмотреть и реализовать в одном классе (`ConcreteComponent`) всю функциональность которая (может быть?!) возможно понадобится, можно создать простой класс конкретного компонента (`ConcreteComponent`) и постепенно наращивать функциональность его объектов, с помощью декораторов (`ConcreteDecorator`). В результате классы не будут содержать неиспользуемые функции, что существенно упростит работу с классом, так как при использовании класса не придется обращать внимания и вникать в работу неиспользуемых методов.

Паттерн Decorator обладает следующими недостатками:

- **Потеря идентичности.**
  Идентичность имеет противопоставление – индивидуальность. Природа идентичности позволяет унифицировано представлять функциональные возможности объектов причисляя их к определенной группе. Индивидуальность часто лишает объект, принадлежности к той или иной группе. Таким образом декорированные объекты теряют идентичность и приобретают индивидуальность. Как в жизни многие люди пытаются выразить свою индивидуальность декорируя себя одеждой, прической и прочими атрибутами.
  Идентичные объекты:



Индивидуальные объекты:



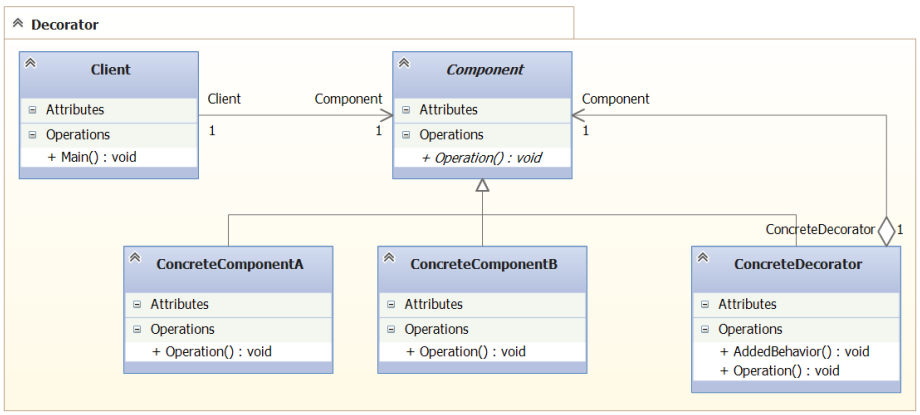
- **Наличие большого числа мелких объектов-декораторов.**
  При использовании паттерна Decorator, иногда получается так, что в системе появляется большое число мелких объектов-декораторов. Программист, знающий паттерны и разбирающийся в устройстве такой системы, сможет легко настраивать разные комбинации вариантов декорирования. Но новичку будет немного сложно изучать и отлаживать такую систему.

Реализация

Полезные приемы реализации паттерна Decorator:

- **Соответствие интерфейсов.**
  Интерфейс декораторов (`Decorator`) должен соответствовать интерфейсу декорируемого объекта (`ConcreteComponent`), поэтому, как класс декорируемого объекта, так и классы декораторов должны наследоваться от общего класса (`Component`).

- **Возможность отсутствия абстрактного класса `Decorator`.**
  Нет необходимости создавать абстрактный класс `Decorator` если планируется добавить только одну обязанность для конкретных компонентов (`ConcreteComponent`). Достаточно будет создать один класс конкретного декоратора (`ConcreteDecorator`), и помимо его основной функциональности, возложить на него ответственность за переадресацию запросов, которая должна была лежать на базовом абстрактном классе (`Decorator`).



См. Пример к главе: \009_Decorator\003_ConcreteDecorator

Известные применения паттерна в .Net

`System.Windows.Controls.Decorator`
http://msdn.microsoft.com/en-us/library/system.windows.controls.decorator.aspx

`System.Data.Entity.Migrations.Infrastructure.MigratorLoggingDecorator`
http://msdn.microsoft.com/ru-ru/library/system.data.entity.migrations.infrastructure.migratorloggingdecorator(v=vs.113).aspx

`System.Windows.Controls.Primitives.BulletDecorator`
http://msdn.microsoft.com/ru-ru/library/system.windows.controls.primitives.bulletdecorator(v=vs.90).aspx

`System.Windows.Documents.AdornerDecorator`
http://msdn.microsoft.com/en-us/library/system.windows.documents.adornerdecorator.aspx

`Microsoft.Windows.Themes.ClassicBorderDecorator`
http://msdn.microsoft.com/en-us/library/microsoft.windows.themes.classicborderdecorator.aspx

`System.Web.HttpBrowserCapabilitiesWrapper`
http://msdn.microsoft.com/ru-ru/library/system.web.httpbrowsercapabilitieswrapper(v=vs.90).aspx

`System.Web.HttpApplicationStateWrapper`
http://msdn.microsoft.com/ru-ru/library/system.web.httpapplicationstatewrapper(v=vs.90).aspx

`System.Web.HttpContextWrapper`
http://msdn.microsoft.com/en-us/library/system.web.httpcontextwrapper(v=vs.110).aspx

`System.Web.HttpFileCollectionWrapper`
http://msdn.microsoft.com/en-us/library/system.web.httpfilecollectionwrapper(v=vs.110).aspx

`System.Web.HttpSessionStateWrapper`
http://msdn.microsoft.com/en-us/library/system.web.httpsessionstatewrapper(v=vs.110).aspx

Паттерн Facade

Название

Фасад

Также известен как

Glue (Клей)

Классификация

По цели: структурный
По применимости: к объектам

Частота использования

Высокая - 1 2 3 4 **5**

Назначение

Паттерн Facade - предоставляет унифицированный интерфейс (набор имен методов) вместо интерфейса некоторой подсистемы (набора взаимосвязанных классов или объектов).

Класс `Facade` предоставляет высокоуровневый интерфейс (набор методов одного класса) вместо низкоуровневого интерфейса подсистемы (наборов методов из разных классов входящих в состав подсистемы).

Введение

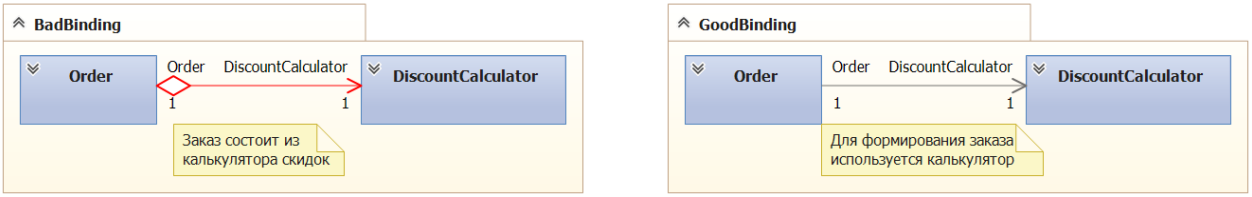
Большие программные системы достаточно сложно проектировать и сопровождать, поэтому обычно, для облегчения работы, большие системы принято разбивать на подсистемы. Главная задача, решаемая при проектировании – это сокращение количества связей отношений между классами или если сказать по-другому – уменьшение зависимостей классов друг от друга.

Зависимость – это технический термин, который описывает количество имеющихся связей отношений. Если на диаграмме мы видим, что у некоторого класса имеется много связей отношений с другими классами - мы говорим, что такой класс сильно зависит от других классов. А если сказать правильнее – работа экземпляра такого класса будет сильно зависеть от работы экземпляров других классов.

Зависимости классов образуют так называемые привязки. Привязка – это логический термин, который заставляет программиста задуматься над смыслом зависимости. С точки зрения ООП, привязки бывают двух типов: *хорошие* - бизнес привязки и *плохие* - технические привязки. Бизнес привязки выражают требования бизнеса, например, класс `Customer` связан связью отношения ассоциации с классом `Order`. Технические привязки выражают системные требования, например, класс `Customer` связан связью отношения ассоциации с классом `DataSet`. Важно заметить, что без технических привязок не обойтись, и техническая привязка может стать *условно-хорошей* тогда и только тогда, когда зависимые сущности находятся в разных слоях системы (например, класс `Customer` располагается в Business Layer, а `DataSet` располагается в слое Data Layer), в таком случае при анализе бизнес логики программной системы можно пренебречь связями, ведущими в нижележащий слой.

Имеются привязки, которые не попадают под классификацию *хороших* и *плохих* привязок – это *«вздорные»* привязки, например, класс `Order` связан связью отношения ассоциации с классом `Customer`, или класс `Customer` связан связью отношения агрегации с классом `Order`. Понятно, что связь отношения ассоциации символизирует знание о чем-то или о ком-то и звучит «*знаю о* или *использую это*», а связь отношения агрегации символизирует составление из частей и звучит «*состою из* или *включаю в себя*». Поэтому приведенные выше примеры *«вздорных»* связей будут звучать так: `Order` знает о `Customer` (заказ знает о человеке), или `Customer` состоит из `Order` (человек состоит из заказа). Связь `Order` знает о `Customer`, можно заменить, например, на `Order` состоит из `CustomersDetails`. Так же класс `Order`

может использовать при построении заказа, например, калькулятор расчета скидок `DiscountCalculator`, такая связь выражается ассоциацией, а не агрегацией, так как не очень удачным решением будет создавать странную гибридную сущность - «*заказо-скидко-калькулятор*».

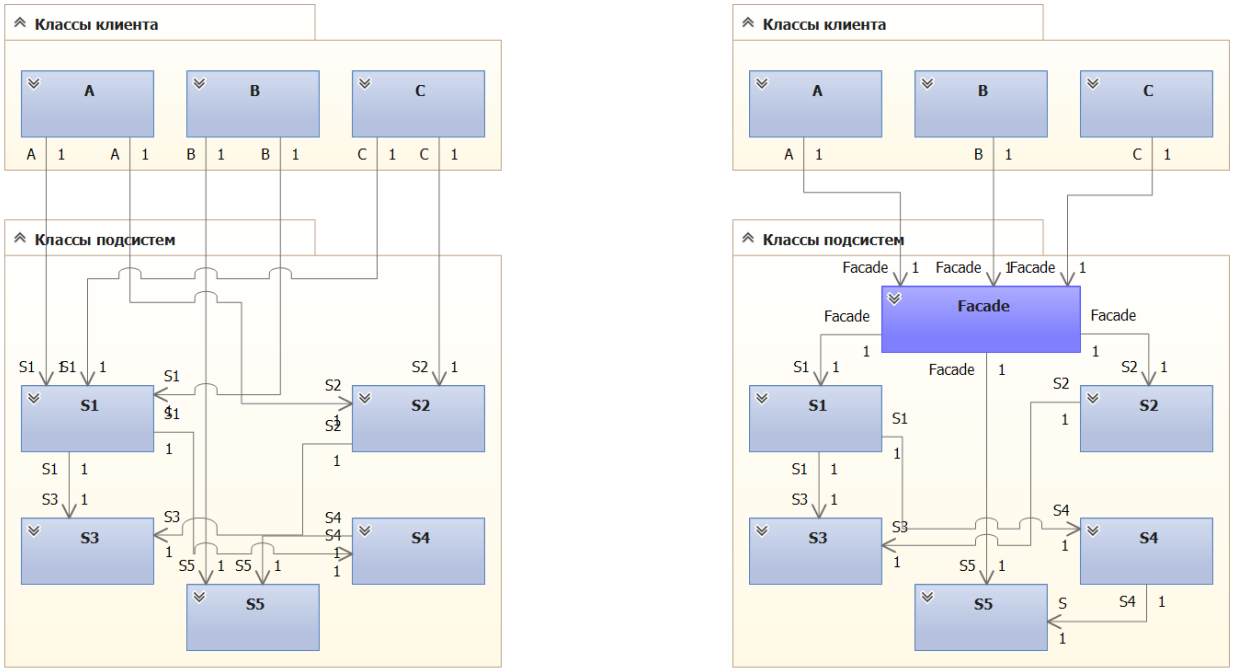


Понятно, что без привязок не обойтись, но нужно стараться минимизировать их количество и при этом сохранять логическую целостность смысла моделируемого процесса.

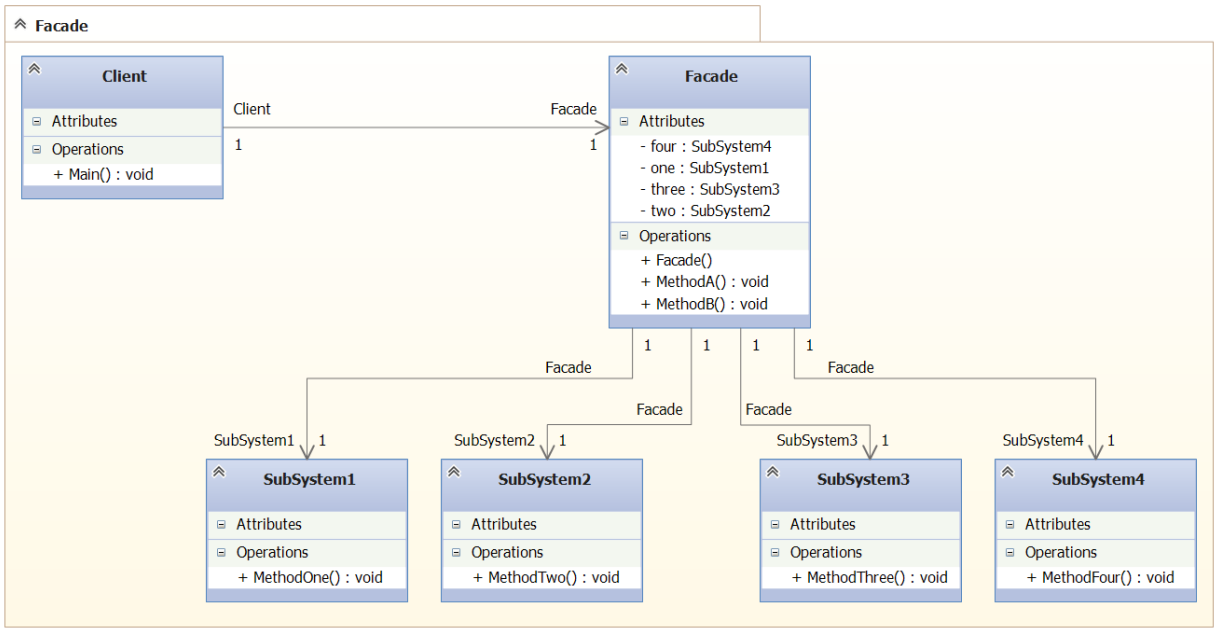
Возможно ли выразить количественно силу зависимости? Конечно, сила зависимости выражается при помощи такого понятия как связанность (*coupling*). Связанность – есть мера зависимости. Связанность имеет более десятка метрик, позволяющих получить численное значение силы зависимости. Детальное рассмотрение метрик связанности не входит в контекст данной книги.

Одним из способов сведения зависимостей к минимуму, это использование паттерна Facade.

На диаграмме слева, видно, что из клиентского кода происходит много обращений к классам подсистем. На диаграмме справа, вводится дополнительный ассоциативный класс `Facade`, через единый и упрощенный интерфейс которого происходит взаимодействие с классами подсистем.

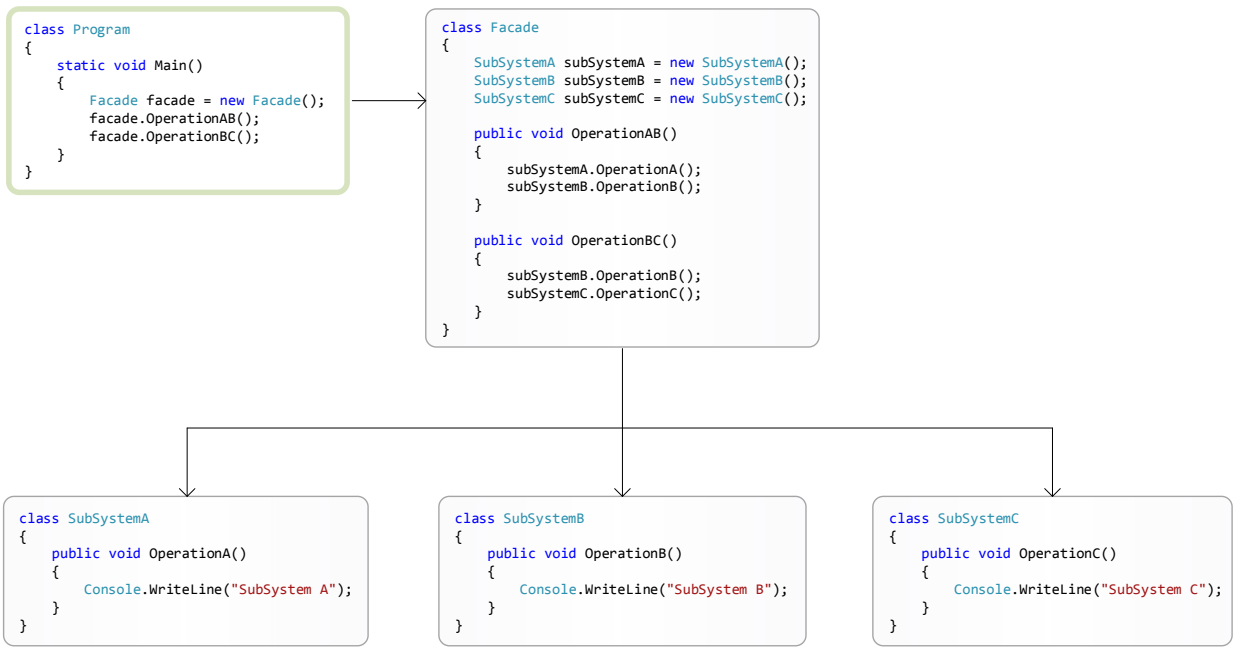


Структура паттерна на языке UML



См. Пример к главе: \010_Facade\001_Facade

Структура паттерна на языке C#



```
class Program
{
    static void Main()
    {
        Facade facade = new Facade();
        facade.OperationAB();
        facade.OperationBC();
    }
}
```

```
class Facade
{
    SubSystemA subSystemA = new SubSystemA();
    SubSystemB subSystemB = new SubSystemB();
    SubSystemC subSystemC = new SubSystemC();

    public void OperationAB()
    {
        subSystemA.OperationA();
        subSystemB.OperationB();
    }

    public void OperationBC()
    {
        subSystemB.OperationB();
        subSystemC.OperationC();
    }
}
```

```
class SubSystemA
{
    public void OperationA()
    {
        Console.WriteLine("SubSystem A");
    }
}
```

```
class SubSystemB
{
    public void OperationB()
    {
        Console.WriteLine("SubSystem B");
    }
}
```

```
class SubSystemC
{
    public void OperationC()
    {
        Console.WriteLine("SubSystem C");
    }
}
```

См. Пример к главе: \010_Facade\001_Facade

Участники

- `Facade` - **Фасад**:
  Перенаправляет запросы клиентов, другим объектам из которых состоит подсистема.

- `Classes subsystems` – **Классы подсистем**:
  Реализуют фактическую функциональность системы. Ничего не знают о фасаде (не имеют ссылок на фасад).

Отношения между участниками

Отношения между классами

- Класс `Facade` связан связями отношения ассоциации с классами подсистем (`Subsystem`).

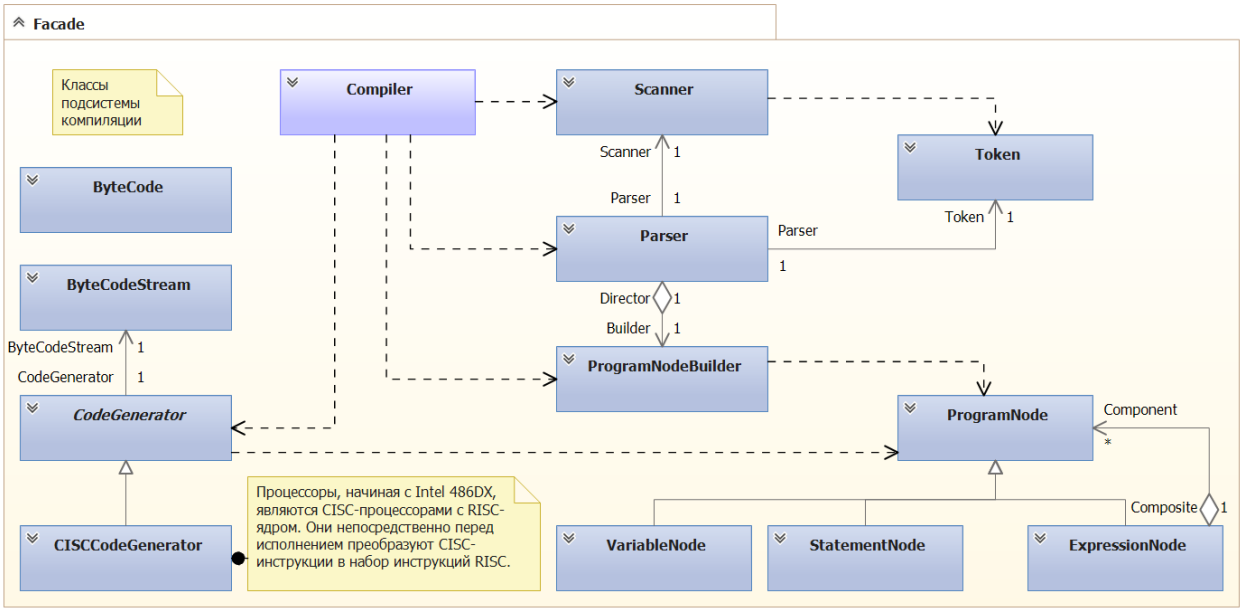
Отношения между объектами

- Клиенты общаются с объектами подсистем, через фасад. Фасад переадресует запросы клиентов подходящим объектам подсистемы.

- Клиенты, которые используют фасад, не должны параллельно работать с объектами подсистем напрямую.

Мотивация

В качестве примера использования паттерна Facade, рассмотрим простейшую программу-компилятор, для ограниченного числа мнемоник (инструкций) языка Assembler.



Имеется простейшая программа на языке Assembler, которую требуется преобразовать в исполняемый файл result.exe:

```
var1 dd 5       ; Переменной с именем var1, типа dd, присваивается значение 5.
var2 dd 2

mov eax, var1 ; В регистр процессора – eax, помещается значение переменной var1.
add eax, var2  ; К значению в регистре, прибавляется значение переменной var2.

call HexMessage
call ExitProcess
```

В компиляторе имеются следующие классы: `Scanner` (лексический анализатор), `Parser` (синтаксический анализатор), `ProgramNode` (узел программы), `BytecodeStream` (поток байтовых кодов), `ProgramNodeBuilder` (строитель узла программы) и другие.

Для того чтобы произвести компиляцию программы, клиенту требуется понимать работу всех классов, составляющих систему компиляции. Чтобы упростить способ взаимодействия с системой компиляции, есть смысл включить в систему класс `Compiler` (компилятор). Класс `Compiler` предоставляет высокоуровневый интерфейс для упрощенного взаимодействия с подсистемой компиляции и будет играть роль фасада этой подсистемы. Теперь, для того чтобы произвести компиляцию, потребуется объекту-компилятору, передать исходный код программы, указать имя исполняемого файла и передать ссылку на механизм обработки сообщений от компилятора.

```
class Program
{
    public static void Main()
    {
        string sourceCode = "var1 dd 5; var2 dd 2;" +
                            "mov eax, var1;" +
                            "add eax, var2;" +
                            "call HexMessage;" +
                            "call ExitProcess;";

        Action<string> message = msg => Console.WriteLine(msg);
        Compiler compiler = new Compiler();
        compiler.Compile(sourceCode, "result.exe", message);
    }
}
```

Благодаря использованию объекта фасада, работа программиста облегчается, так как не нужно вникать в логику работы подсистемы компиляции, при этом, остается возможность обращения к классам подсистем, когда это может потребоваться. Например, чтобы получить коллекцию токенов (`Token`) каждый из которых предоставляет набор лексем, можно обратиться к сканеру (`Scanner`) напрямую.

```
class Program
{
    public static void Main()
    {
        string sourceCode = "var1 dd 5; var2 dd 2;" +
                            "mov eax, var1;" +
                            "add eax, var2;" +
                            "call HexMessage;" +
                            "call ExitProcess;";

        Scanner scanner = new Scanner();
        List<Token> tokens = scanner.Scan(sourceCode);
    }
}
```

См. Пример к главе: \010_Facade\002_Assembler Compiler

Применимость паттерна

Паттерн Facade рекомендуется использовать, когда:

- Требуется предоставить высокоуровневый интерфейс для замены множества низкоуровневых интерфейсов.
- Между клиентскими классами и классами подсистем имеется много зависимостей.
- Требуется разложить систему на слои, при этом объект-фасад будет использоваться в качестве точки входа в слой.

Результаты

Паттерн Facade обладает следующими преимуществами:

- **Сокрытие устройства подсистемы.**
  При использовании объекта-фасада, клиентам не приходится обращаться к объектам из которых состоит подсистема и использовать их напрямую. Такой подход упрощает работу клиентов, так как им не требуется детально знать устройство и способы сопряжения частей сложной подсистемы.

- **Ослабление связанности между клиентским кодом и объектами подсистем.**
  Часто объекты сложной подсистемы сильно связанны между собой, а клиентские объекты еще больше усиливают эту связанность за счет прямого использования объектов подсистем. Сильная связанность усложняет возможность внесения изменений в классы объектов подсистемы. Если клиентский код завязан на интерфейс объектов подсистем напрямую, то внесение изменений в логику работы объектов подсистем может негативно сказываться на работе клиентских объектов. Фасадный объект позволяет сформировать высокоуровневый интерфейс для взаимодействия клиента с системой. Высокоуровневый интерфейс всегда должен предоставлять удобную абстракцию для взаимодействия клиента со сложной системой.

  Например, в жизни людям проще пользоваться современным автомобилем, чем автомобилем начала 19 века. Старинный автомобиль требует от водителя знания множества тонкостей обращения, начиная от запуска двигателя (проворачивают коленвал пусковой (заводной) рукояткой - «*кривой стартёр*») и заканчивая особенностями управления автомобилем в процессе его движения. Современный автомобиль заводится просто, различные системы контроля делают процесс движения автомобиля безопасным как в дождь, гололедицу, в большом потоке транспорта, позволяют безопасно парковать автомобиль и предоставляют много других полезных особенностей. Можно сказать, что интерфейс взаимодействия с современным автомобилем проще интерфейса чем со старинным автомобилем, при этом интерфейс взаимодействия с новым автомобилем функционально богаче интерфейса взаимодействия со старым автомобилем. Водитель взаимодействует с автомобилем через «фасад» представляющий собой руль, педали, ручку переключения скоростей и множество различных кнопок. Фасад автомобиля скрывает за собой части, из которых состоит, например, двигатель автомобиля.

- **Использование объектов подсистем напрямую.**
  Объекты подсистем допускается использовать напрямую. В таком случае рекомендуется отказаться от параллельного использования объекта фасад, так как могут возникнуть путаница при использовании подсистемы через интерфейсы разных уровней. В общем случае рекомендуется придерживаться следующего правила: без полного понимания устройства всех частей подсистемы или с пониманием устройства только некоторых частей подсистемы – следует пользоваться фасадом.

Реализация

Полезные приемы реализации паттерна Facade:

- **Уменьшение связанности клиента с подсистемой.**
  Связанность можно уменьшить, если класс `Facade` будет абстрактным, а производные от него, конкретные классы `ConcreteFacadeA` и `ConcreteFacadeB` будут соответствовать различным реализациям подсистем. Например, требуется создать компилятор, генерирующий машинные коды для различных аппаратных архитектур, таких как «*Intel x86 family*» и «*Motorola 68000 family*». Класс `Compiler` будет абстрактным, а от него будут наследоваться два конкретных класса `CompilerIntelx86` и `CompilerMotorola68000`. В таком случае клиенты смогут взаимодействовать с подсистемой компиляции через унифицированный интерфейс абстрактного класса `Compiler` (не замечая различия между особенностями архитектуры Фон-Неймана (*Intel*) и особенностями Гарвардской архитектуры (*Motorola*)).

- **Открытые и закрытые классы подсистем.**
  Иногда полезно делать некоторые классы подсистемы закрытыми от клиента (имеется ввиду *внутренними* - `internal`), размещая их в отдельной сборке и помечая модификатором доступа `internal`. Класс `Facade` должен принадлежать к открытой части сборки и должен быть помеченным модификатором доступа - `public`.

  См. Пример к главе: \010_Facade\003_OpenAndCloseClasses

  В примере с компилятором к открытой части подсистемы компиляции можно отнести класс `Compiler`, а также важные классы, которые в отдельных случаях есть смысл использовать самостоятельно: `Scanner` и `Parser`, все остальные классы есть смысл отнести к закрытой части.

Пример кода

Предлагается более подробно рассмотреть пример из раздела «Мотивация»: «Реализация подсистемы компиляции исходного кода на языке Assembler в машинный код».

Класс `Compiler` представляет собой фасад, для организации удобного и безопасного доступа к компонентам подсистемы компиляции.

```
public class Compiler
{
    Scanner scanner;
    Parser parser;
    ProgramNodeBuilder builder;
    CodeGenerator generator;
    ByteCodeStream stream;

    public Compiler()
    {
        this.stream = new ByteCodeStream();
        this.scanner = new Scanner();
        this.parser = new Parser();
        this.generator = new CISCCodeGenerator();
        this.builder = new ProgramNodeBuilder(this.generator);
    }

    public void Compile(string sourceCode, string exeFileLocation,
                                           Action<string> notification)
    {
         try
         {
             parser.Parse(scanner, builder, sourceCode);

             ProgramNode product = builder.GetProgramNode();
```

```
            this.stream.SaveStreamToFile(product.ProgramByteCode, exeFileLocation);
            notification("Компиляция завершилась успешно.");
        }
        catch (Exception ex)
        {
            string errorMessage =
                                string.Format("Компиляция не удалась.\r\nОШИБКА: ");
            errorMessage += ex.GetType().Name + ":\n" + ex.Message;
            notification(errorMessage);
        }
    }
}
```

Класс Scanner разбивает программный код на отдельные инструкции представленные токенами (Token).

```
public class Scanner
{
    List<Token> tokens = new List<Token>();

    public List<Token> Scan(string sourceCode)
    {
        string[] commands = sourceCode.Trim().Split(Constants.programDelimiter,
                                            StringSplitOptions.RemoveEmptyEntries);
        foreach (string command in commands)
        {
            string[] lexemes = command.Split(Constants.lexemesDelimiter,
                                             StringSplitOptions.RemoveEmptyEntries);
            this.tokens.Add(new Token(lexemes));
        }

        return tokens;
    }
}
```

Класс Parser проверяет правильность (допустимость) инструкций, анализируя лексемы из токенов. И дает объекту класса ProgramNodeBuilder, команды для построения узлов дерева в соответствии с найденным токеном.

```
internal class Parser
{
    Dictionary<string, int> variables = new Dictionary<string, int>();
    Scanner scanner;
    ProgramNodeBuilder builder;

    public void Parse(Scanner scanner, ProgramNodeBuilder builder, string sourceCode)
    {
        this.scanner = scanner;
        this.builder = builder;

        // Разбить программу на токены состоящие из лексем.
        List<Token> tokens = scanner.Scan(sourceCode);
        // Проверить каждый токен состоящий из лексем на ошибки.
        foreach (Token token in tokens)
            ParseToken(token.Lexemes);

        builder.Build(tokens);
    }

    private void ParseToken(List<string> lexemes)
    {
        // Полная реализация данного метода в примере к главе.
```

```
    }

    private void VerifyParameters(string[] parameters, Command command)
    {
        // Полная реализация данного метода в примере к главе.
    }

    private void VerificationVariableName(string name)
    {
        // Полная реализация данного метода в примере к главе.
    }
}
```

Класс ProgramNodeBuilder используется для построения дерева разбора, состоящего из экземпляров подклассов класса ProgramNode.

```
internal class ProgramNodeBuilder
{
    CodeGenerator codeGenerator = null;
    ProgramNode programNode = new ExpressionNode();

    public ProgramNodeBuilder(CodeGenerator codeGenerator)
    {
        this.codeGenerator = codeGenerator;
    }

    public void Build(List<Token> tokens)
    {
        codeGenerator.Initialize();
        programNode.Traverse(codeGenerator);
        programNode.ProgramByteCode = codeGenerator.GenerateByteCode();
    }

    public ProgramNode AddVariableNode(string name, int value)
    {
        VariableNode node = new VariableNode(name, value);
        programNode.AddNode(node);
        return node;
    }

    public ProgramNode AddStatementNode(string name, string[] parameters)
    {
        StatementNode node = new StatementNode(name, parameters);
        programNode.AddNode(node);
        return node;
    }

    public ProgramNode GetProgramNode()
    {
        return this.programNode;
    }
}
```

Абстрактный класс ProgramNode является базовым классом для классов ExpressionNode, StatementNode, и VariableNode, экземпляры которых будут представлены элементами в дереве разбора.

```
abstract class ProgramNode
{
    public ByteCode ProgramByteCode { get; set; }
```

```
    public abstract void Traverse(CodeGenerator generator);
    public abstract void AddNode(ProgramNode node);
}
```

Класс `ExpressionNode` представляет собой узловой элемент дерева разбора.

```
class ExpressionNode : ProgramNode
{
    List<ProgramNode> nodes = new List<ProgramNode>();

    public override void AddNode(ProgramNode node)
    {
        nodes.Add(node);
    }

    public override void Traverse(CodeGenerator generator)
    {
        foreach (var item in nodes)
            item.Traverse(generator);
    }
}
```

Класс `StatementNode` представляет собой листовой элемент, определяющий команды в исходном коде.

```
class StatementNode : ProgramNode
{
    public StatementNode()
    {
        Offsets = new List<int>();
    }

    public StatementNode(string name, string[] parameters)
    {
        Name = name;
        Parameters = parameters;
        Offsets = new List<int>();
    }

    public string Name { get; set; }
    public List<int> Offsets { get; set; }
    public int Address { get; set; }
    public string[] Parameters { get; set; }

    public override void AddNode(ProgramNode node)
    {
        throw new InvalidOperationException();
    }

    public override void Traverse(CodeGenerator generator)
    {
        // Полная реализация данного метода в примере к главе.
    }
}
```

Класс `VariableNode` представляет собой листовой элемент определяющий переменную в исходном коде.

```
class VariableNode : ProgramNode
{
    public string Name { get; set; }
    public int Value { get; set; }
```

```
    public byte[] Address { get; set; }

    public VariableNode(string name, int value)
    {
        Name = name;
        Value = value;
    }

    public override void AddNode(ProgramNode node)
    {
        throw new InvalidOperationException();
    }

    public override void Traverse(CodeGenerator generator)
    {
        generator.SetDataVariable(this);
    }
}
```

Класс `Token` хранит в себе набор лексем (частей, составляющих инструкцию ассемблера).

```
public class Token
{
    public List<string> Lexemes { get; set;}

    public Token(){  }

    public Token(string [] lexemes)
    {
        this.Lexemes = new List<string>();
        this.Lexemes.AddRange(lexemes);
    }
}
```

Абстрактный класс `CodeGenerator` является базовым для класса `CISCCodeGenerator` и возможных классов-генераторов машинных кодов для других аппаратных архитектур.

```
abstract class CodeGenerator
{
    // Полная реализация данного класса в примере к главе.

    public abstract void Initialize();
    public abstract ByteCode GetNopCode();
    public abstract ByteCode GetCallCode(string[] parameters);
    public abstract ByteCode GetXorCode(string[] parameters);
    public abstract ByteCode GetSubCode(string[] parameters);
    public abstract ByteCode GetAddCode(string[] parameters);
    public abstract ByteCode GetMovCode(string[] parameters);
    public abstract void SetDataVariable(VariableNode node);
    public abstract void FixFunctionAddresses();
    public abstract ByteCode GenerateByteCode();
}
```

Класс `CISCCodeGenerator` генерирует машинные коды для аппаратной архитектуры «*Intel x86 family*».

```
class CISCCodeGenerator : CodeGenerator
{
    // Полная реализация данного класса в примере к главе.
}
```

Класс ByteCode представляет собой объектно-ориентированное представление машинного кода, полученного из узлов дерева разбора.

```
class ByteCode
{
    byte[] code = null;

    public byte[] Code
    {
        get { return code; }
    }

    public ByteCode(params byte[] code)
    {
       this.code = code;
    }
}
```

Класс ByteCodeStream выполняет сохранение байт-кода всей программы в файл.

```
class ByteCodeStream
{
    public void SaveStreamToFile(ByteCode programByteCode, string exeFileLocation)
    {
        using (FileStream fileStream = File.Create(exeFileLocation,
                                                   programByteCode.Code.Length))
        {
            fileStream.Write(programByteCode.Code, 0, programByteCode.Code.Length);
        }
    }
}
```

В этой реализации жестко зашит тип кодогенератора, так как не требовалось задавать целевую аппаратную архитектуру. Такой подход может быть приемлемым, когда имеется всего одна аппаратная архитектура.

Паттерн Flyweight

Название

Приспособленец (*Русское название паттерна Flyweight, не отражает точный перевод*)

Также известен как

-

Классификация

По цели: структурный
По применимости: к объектам

Частота использования

Низкая - **1** 2 3 4 5

Назначение

Паттерн Flyweight – описывает правильное применение техники создания «*разделяемых объектов*», для получения возможности эффективного использования большого числа объектов.

Введение

На «житейском» уровне сложно придумать достойную аналогию или афористически яркую метафору, описывающую использование паттерна Flyweight. Рассмотрим ситуацию, когда один актер играет в одном кинофильме сразу несколько ролей. Например, Майк Майерс (Mike Myers) сыграл целых три роли в кинофильме «Остин Пауэрс».

При просмотре кинофильма, зрители сопереживая вымыслу концентрируют свое внимание на ролях – «Остине Пауэрсе» или «Докторе Зло», а не на актере исполнителе Майке Майерсе. А многие зрители даже не догадываются о том, что один актер сыграл все эти роли.

В ООП имеются такие принципиально различные виды объектов, которые называются «*разделяемые*» и «*неразделяемые*». Сам актер Майк Майерс – является «*разделяемым*» объектом. Актера

разделяют между собой роли-персонажи являющиеся «*неразделяемыми*» объектами – это роль «Остин Пауэрс» и роль «Доктор Зло». Роль – это полноценный объект состоящий из костюма и грима. Костюм и грим творят чудеса, с их помощью, один и тот же актер может сыграть несколько ролей в одной картине. Важно понимать, что кинозрители не взаимодействуют напрямую с человеком-актером, кинозрители сосредотачивают свое внимание именно на игре роли-персонажа, забывая о человеке, играющем роль. Другими словами, кинозрители могут не помнить имя актера, но помнят и обсуждают поступки самого персонажа в контексте сюжета и фильма. Соответственно кинозрители «клиенты» не взаимодействуют напрямую с актером - «*разделяемым*» объектом, но взаимодействуют с ролями - «*неразделяемыми*» объектами. Паттерн Flyweight как раз и описывает способы «правильного обращения» с разделяемыми и неразделяемыми объектами.

Представим рассмотренный пример использования разделяемых и неразделяемых объектов программно.

```
class Program
{
    static void Main()
    {
        ActorMikeMyers mike = new ActorMikeMyers();

        RoleAustinPowers austin = new RoleAustinPowers(mike);
        austin.Greeting("Hello! I'm Austin Powers!");

        RoleDoctorEvil dr = new RoleDoctorEvil(mike);
        dr.Greeting("Hello! I'm Dr.Evil!");
    }
}
```

```
abstract class Flyweight
{
    public abstract void Greeting(string speech);
}
```

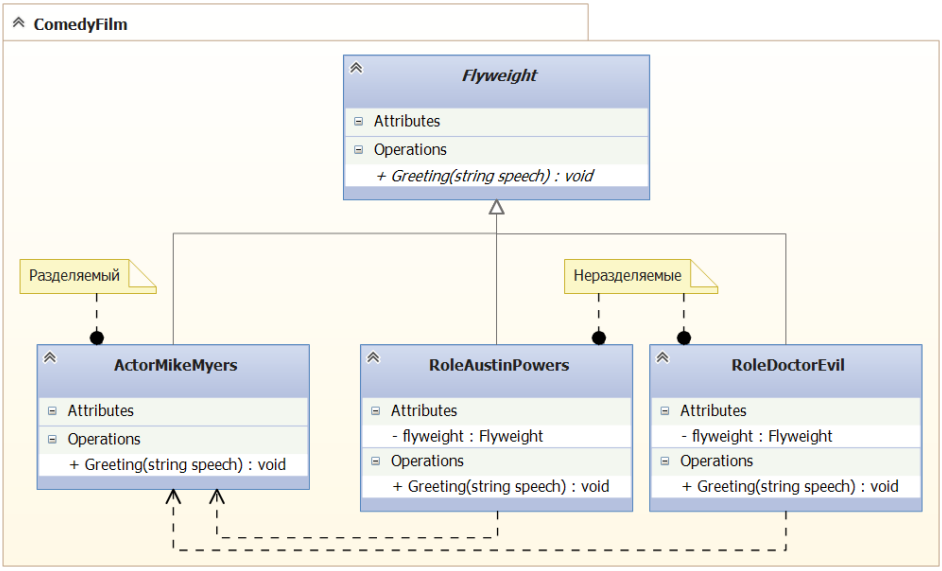
```
// Неразделяемый.
class RoleAustinPowers : Flyweight
{
    Flyweight flyweight;

    public RoleAustinPowers(Flyweight flyweight)
    {
        this.flyweight = flyweight;
    }

    public override void Greeting(string speech)
    {
        this.flyweight.Greeting(speech);
    }
}
```

```
// Разделяемый.
class ActorMikeMyers : Flyweight
{
    public override void Greeting(string speech)
    {
        Console.WriteLine(speech);
    }
}
```

```
// Неразделяемый.
class RoleDoctorEvil : Flyweight
{
    Flyweight flyweight;

    public RoleDoctorEvil(Flyweight flyweight)
    {
        this.flyweight = flyweight;
    }

    public override void Greeting(string speech)
    {
        this.flyweight.Greeting(speech);
    }
}
```

См. Пример к главе: \011_Flyweight\003_ComedyFilm

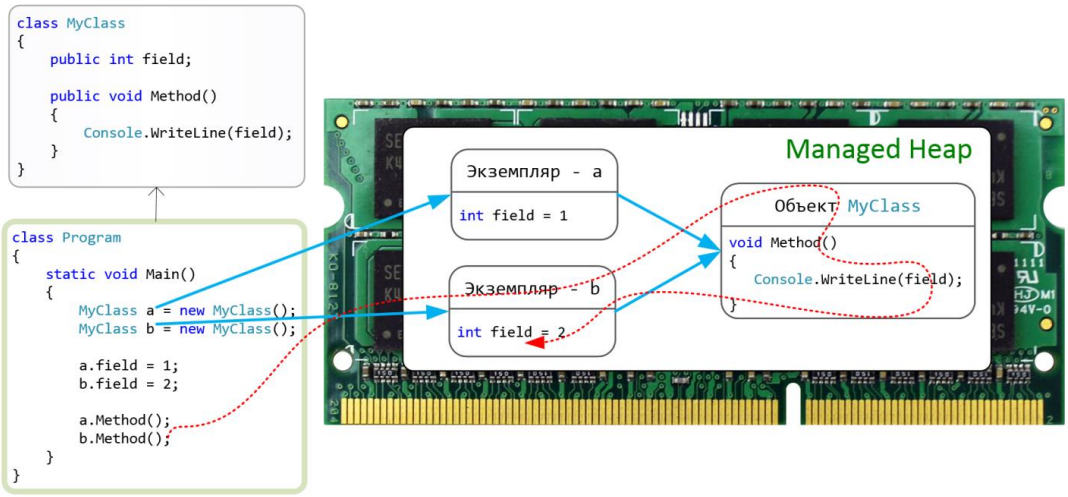
Диаграмма классов будет выглядеть следующим образом:



См. Пример к главе: \011_Flyweight\003_ComedyFilm

Возникает ряд вопросов. Зачем разделять объекты на «*разделяемые*» и «*неразделяемые*»? Где эти объекты использовать в своих программах? Почему указана такая низкая частота использования паттерна Flyweight?

Для того, чтобы ответить на эти вопросы, следует вспомнить различия между объектами и экземплярами, которые строятся и размещаются в управляемой куче (*Managed heap*). На рисунке ниже показано, как происходит разделение «*исполняемой сущности*» на объект и экземпляры. Объект – это область динамической памяти, которая содержит в себе методы (*и статические поля*). Экземпляр – это область динамической памяти, которая содержит в себе только нестатические поля.

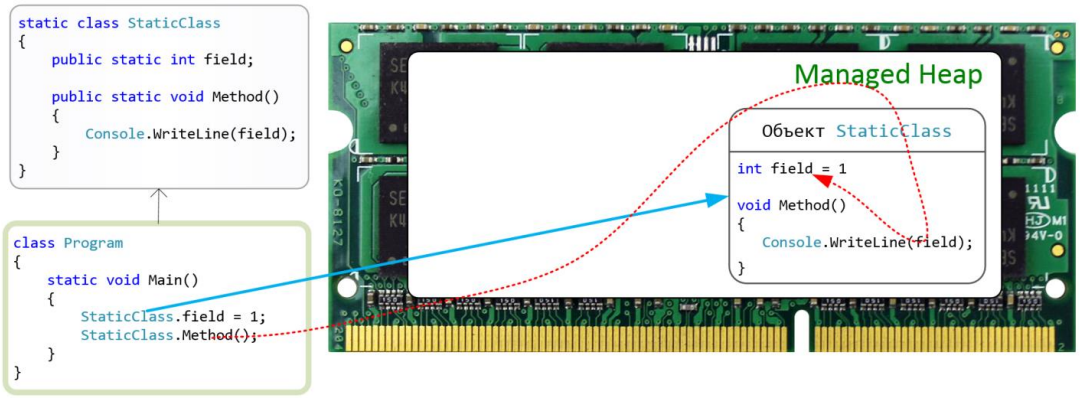


Как видно из рисунка у класса может быть только один объект и сколько угодно экземпляров. Локальные переменные `a` и `b` из метода `Main` содержат ссылки (адреса памяти) на экземпляры. Экземпляры «a» и «b» в свою очередь содержат в специальном служебном блоке (заголовке) адрес самого объекта. На рисунке синими (прямыми) стрелками показана ссылочность: «переменная» - «экземпляр» - «объект». Красная (пунктирная) линия показывает «*маршрут*» прохождения запроса (*вызова метода*). Более детально познакомится с организацией взаимодействий между объектом и экземплярами можно в книге Джефри Рихтера – «CLR via C#».

Разделение «*программной сущности уровня выполнения*» на объект и экземпляры, было организовано с целью добиться экономии использования оперативной памяти. Важно понимать, что тело любого метода (*функции или процедуры*) представляет собой набор инструкций C#, которые будут преобразованы компилятором csc.exe (*C Sharp Compiler*) в байт-код который будет подаваться на вход виртуальной машины (интерпретатора, компилирующего (*JIT*) типа) CLR (*C:\Windows\System32\mscoree.dll*).

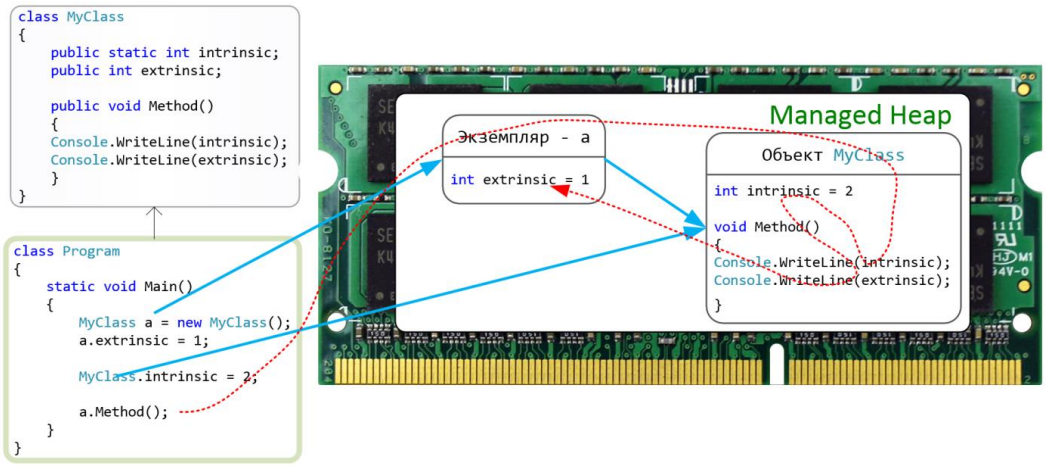
Байт-код тела метода занимает собой определенный объем оперативной памяти и если этот код дублировать в каждом экземпляре, то такой подход может оказаться не эффективным с точки зрения расходования памяти. Поэтому код метода выносится в отдельную исполняемую сущность, которая называется объектом. Экземпляры же в себе хранят только нестатические поля.

Также следует знать, что имеется возможность использовать только объект без построения экземпляра. Прямое обращение к объекту возможно при наличии в программе статического класса или статических членов нестатического класса. На рисунке ниже показано, как происходит прямое обращение к объекту статического класса. В таком случае в программе имеется только «*разделяемая*» сущность (объект) при отсутствии «*неразделяемых*» (экземпляров). В нотации ООП строка `StaticClass.Method()` – читается так: на «*классе-объекте*» `StaticClass` вызывается метод с именем `Method`.



В свою очередь построение только одного экземпляра «*неразделяемого*», без объекта невозможно и это очевидно, так как все классы неявно наследуются от базового класса `Object`, который содержит в себе набор методов, которые и будут размещены в объекте.

Помимо рассмотрения техник разделения на «*разделяемые объекты*» и «*неразделяемые объекты*», следует сделать акцент на таких понятиях как «*внутреннее состояние*» и «*внешнее состояние*». Известно, что состояние объекта представляется совокупностью значений полей этого объекта.

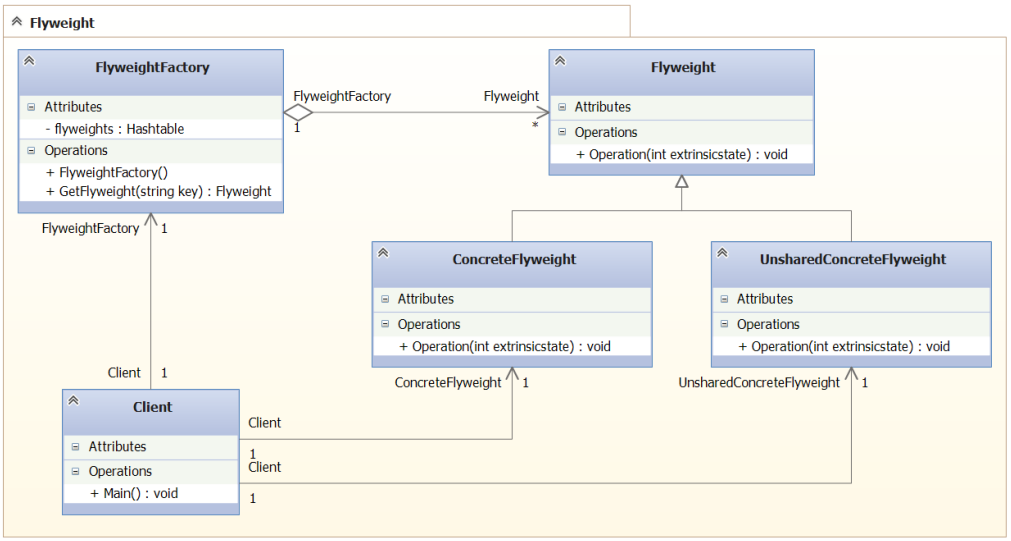


Принято называть состояние «*разделяемого объекта*» - «*внутренним состоянием*», а состояние «*неразделяемого объекта*» - «*внешним состоянием*».

Из всего сказанного выше легко видеть, что паттерн Flyweight нашел свое выражение в организации структуры исполняемых сущностей платформы .Net, и идея использования техники «разделяемых» и «неразделяемых» объектов красной нитью проходит через всю архитектуру организации работы динамической управляемой памяти в .Net. Объекты – представлены в динамической памяти как «*разделяемые*» сущности, а экземпляры – представлены как «*неразделяемые*» сущности. Объект – разделяется между несколькими экземплярами, также как актер между несколькими ролями.

Организация разделения «*программных сущностей уровня выполнения*» на объекты и экземпляры, не избавляет совсем от надобности использования паттерна Flyweight, но позволяет программистам намного реже использовать его в своей практике.

Структура паттерна на языке UML



См. Пример к главе: \011_Flyweight\001_Flyweight

Структура паттерна на языке C#

```
class FlyweightFactory
{
    Hashtable pool = new Hashtable
    {
        {"1", new ConcreteFlyweight()},
        {"2", new ConcreteFlyweight()},
        {"3", new ConcreteFlyweight()}
    };

    public Flyweight GetFlyweight(string key)
    {
        if (!pool.ContainsKey(key))
            pool.Add(key, new ConcreteFlyweight());

        return pool[key] as Flyweight;
    }
}
```

```
abstract class Flyweight
{
    public abstract void Operation(int extrinsicState);
}
```

```
class ConcreteFlyweight : Flyweight
{
    int intrinsicState;

    public override void Operation(int extrinsicState)
    {
        intrinsicState = extrinsicState;
    }
}
```

```
class UnsharedConcreteFlyweight : Flyweight
{
    int allState;

    public override void Operation(int extrinsicState)
    {
        allState = extrinsicState;
    }
}
```

```
class Program
{
    static void Main()
    {
        int extrinsicState = 0;

        Flyweight flyweight = null;
        FlyweightFactory factory = new FlyweightFactory();

        flyweight = factory.GetFlyweight("1");
        flyweight.Operation(extrinsicState);

        flyweight = factory.GetFlyweight("10");
        flyweight.Operation(extrinsicState);

        flyweight = new UnsharedConcreteFlyweight();
        flyweight.Operation(extrinsicState);
    }
}
```

См. Пример к главе: \011_Flyweight\001_Flyweight

Участники

- `Flyweight` – **Приспособленец**:
  Предоставляет интерфейс, при помощи которого *разделяемые* объекты (*приспособленцы*) могут подключать *внешнее* состояние и воздействовать на него.

- `ConcreteFlyweight` – **Разделяемый конкретный приспособленец**:
  Класс разделяемого объекта. Реализует интерфейс класса `Flyweight` и добавляет при необходимости внутреннее состояние.

- `UnsharedConcreteFlyweight` – **Неразделяемый конкретный приспособленец**:
  Приспособленцами принято называть *разделяемые* объекты. Неразделяемый приспособленец – «*неразделяемо-разделяемый* объект». Такая формулировка звучит взаимоисключающе. Понятно, что объект класса `UnsharedConcreteFlyweight` не будет являться приспособленцем, то есть не будет *разделяемым* и соответственно не будет происходить выделения части его *внутреннего* состояния во *внешнее*. Но тем не менее составители каталога включили в имя данного участника слово Flyweight, допуская, что не все подклассы класса `Flyweight` должны быть разделяемыми. Такой подход к именованию данного участника спорный, но он имеет место.

- `FlyweightFactory` – **Фабрика приспособленцев**:
  Создает *разделяемые* и *неразделяемые* объекты. Когда клиент запрашивает *разделяемый* объект, фабрика ищет этот объект в «*пуле приспособленцев*» и если находит, то возвращает ссылку на него, иначе создает новый объект, сохраняет его в «*пуле приспособленцев*» и возвращает ссылку на созданный объект. *Неразделяемые* объекты каждый раз создаются заново.

- `Client` – **Клиент**:
  Работает с *разделяемыми* и *неразделяемыми* объектами. Формирует и может хранить *внешнее* состояние *разделяемых* объектов.

Отношения между участниками

Отношения между классами

- Класс `ConcreteFlyweight` связан связью отношения наследования с абстрактным классом `Flyweight`.

- Класс `UnsharedConcreteFlyweight` связан связью отношения наследования с абстрактным классом `Flyweight`.

- Класс `FlyweightFactory` связан связью отношения агрегации с абстрактным классом `Flyweight`.

Отношения между объектами

- Состояние, используемое при работе с `ConcreteFlyweight` разделяется на *внутреннее* и *внешнее*. Внутреннее состояние хранится непосредственно в самом объекте `ConcreteFlyweight`. Внешнее состояние хранится в клиенте (`Client`) и передается объекту-приспособленцу в качестве аргументов методов.

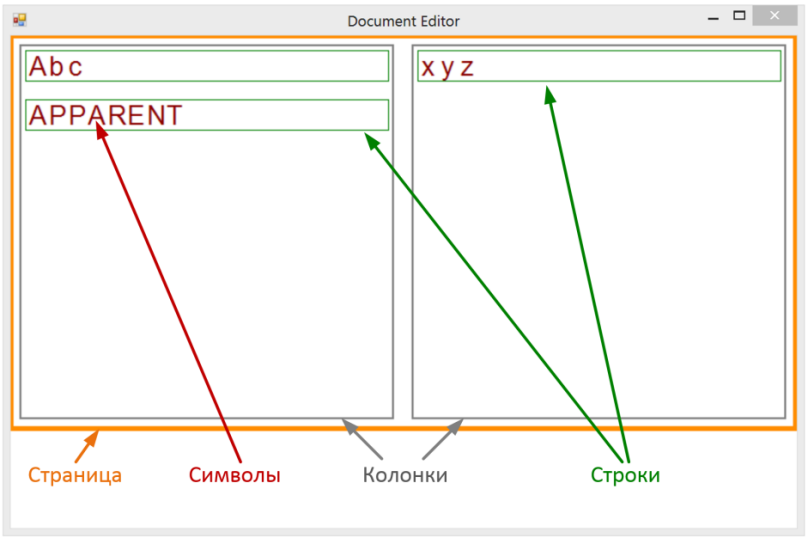
- Клиенты не создают экземпляры `ConcreteFlyweight` напрямую через вызов конструктора, а получают их от объекта `FlyweightFactory`. Такой подход позволяет гарантированно получить *разделяемый* объект.

- Клиентам рекомендуется создавать экземпляры `UnsharedConcreteFlyweight` при помощи объекта `FlyweightFactory`, так как в ходе эволюции программы может потребоваться преобразовать *неразделяемые* объекты в *разделяемые* и если неразделяемые объекты создавались напрямую через вызов конструктора, то придется вносить изменения в клиентский код.

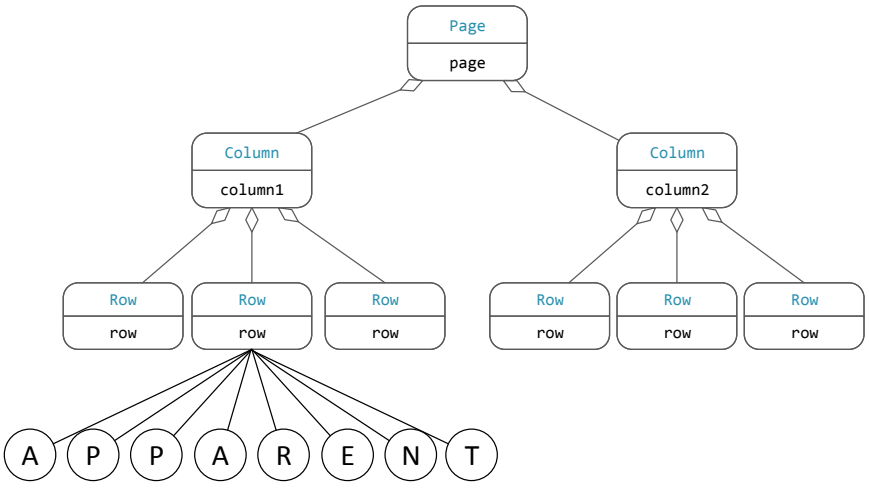
Мотивация

Некоторые разновидности приложений или их части, не всегда могут быть построены полностью с использованием объектно-ориентированного подхода. Рассмотрим такую разновидность приложений, как текстовые редакторы. Объектно-ориентированные текстовые редакторы, для представления таблиц и рисунков обычно используют объекты соответствующих классов (`Table`, `Image`). Но использовать объекты для представления каждого отдельного символа в документе, может оказаться нерациональным с точки зрения расходования памяти, ведь в документе может быть несколько сотен тысяч символов, соответственно в памяти будет создано несколько сотен тысяч объектов класса `Character`. С другой стороны, использование объектов для представления символов повысило бы гибкость использования и сопровождения приложения.

Паттерн Flyweight показывает, как правильно строить подсистемы, в которых требуется использовать очень большое число объектов и при этом избежать недопустимо высокого расходования памяти. На рисунке показано, как редактор документов мог бы использовать объекты для представления символов.



Для построения редактора документов потребуется создать следующие классы: класс `Page` – представляющий страницу, класс `Column` – представляющий колонки (столбцы) на которые может быть разбита страница, класс `Row` – представляющий строки, которые содержатся в колонках и класс `Character` представляющий символы, находящиеся в строках.

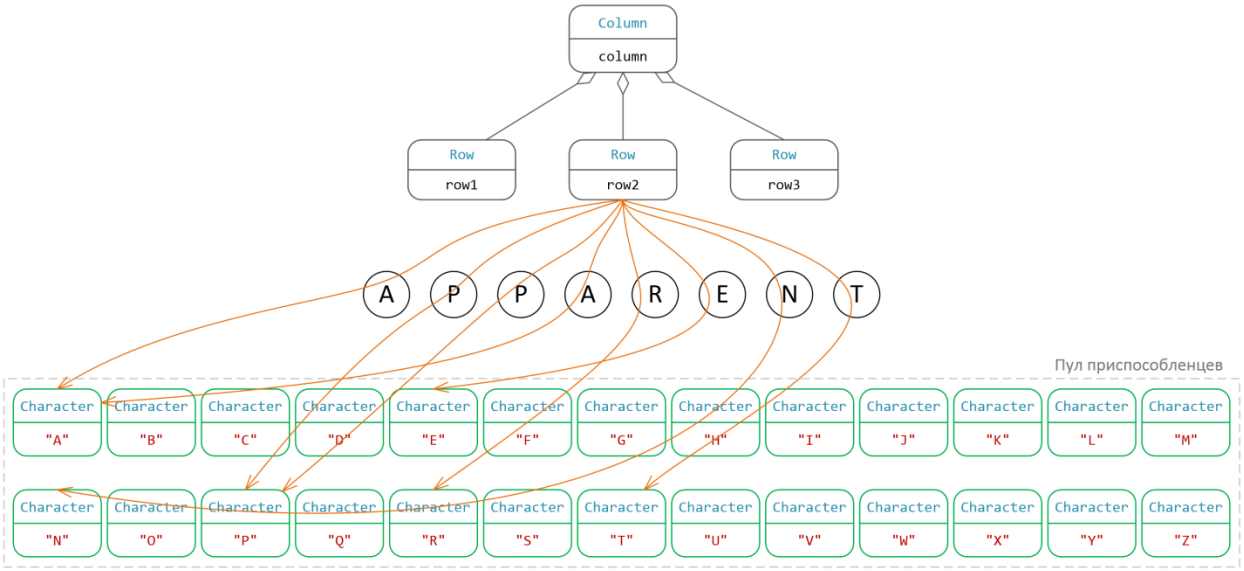


Для того чтобы избежать построения тысяч экземпляров класса `Character`, потребуется использовать так называемые *разделяемые* объекты. *Разделяемый объект* – это такой объект, которым могут (одновременно) пользоваться другие объекты.

Например, в жизни, один актер может играть несколько ролей на протяжении спектакля, выходя на сцену в разных костюмах. У зрителя будет создаваться иллюзия, что роли играют разные актеры. Такой актер одновременно (в сеансе спектакля) *разделяется* на разные роли. Назовем такого актера – *разделяемым актером*. Тех актеров, которые играют в сеансе спектакля только одну роль, назовем – *неразделяемыми актерами*.

Пример из виртуальной реальности: В этой строке имеется только одна буква «а», одна буква «о», одна буква «т» и так далее, не смотря на то, что кажется, что этих букв много. Более того, во всей этой книге имеется только одна маленькая русская буква «а». И в этой строке: a, ***a***, a, - это все одна и та-же буква «а», в разных стилях (размер, тип шрифта, наклон и так далее).

Возникает вопрос: Как возможно добиться такого эффекта в своих программах? Предлагается рассмотреть диаграмму объектов, которые могут входить в состав текстового редактора:



На диаграмме показана древообразная структура объектов, описывающая представление элементов в текстовом документе. Условимся, что представленный на диаграмме документ состоит из одной страницы - объект класса `Page`. Страница состоит только из одной колонки – объект класса `Column` (например, страницы данной книги состоят из одной колонки). В колонке имеется три строки – объекты класса `Row` (`row1`, `row2`, `row3`). Во второй строке содержится набор символов (ссылок на экземпляры класса `Character`) формирующих слово «apparent». Также на диаграмме показан «пул приспособленцев». *Пул* – означает некий набор (коллекцию). *Приспособленец* (`Flyweight`) – это сленговое название *разделяемого объекта*. Иначе можно сказать, что на диаграмме показан набор разделяемых объектов класса `Character`.

Важно понимать, что использование *разделяемых объектов* – это иллюзия в программировании, как иллюзия в жизни, когда фокусник из кармана достает много раз один и тот же шарик, который (хитрым способом) через трубку в рукаве скатывается обратно в карман. Шарик – разделяемый объект. Трубка, рукав, карман – неразделяемые объекты. Фокус с шариком, не получится без трубки. Для работы с разделяемыми объектами понадобится использование неразделяемых объектов. В примере с текстовым редактором, символ (`Character`) – *разделяемый объект*, а страницы (`Page`), колонки (`Column`) и строки (`Row`) – *неразделяемые объекты*.

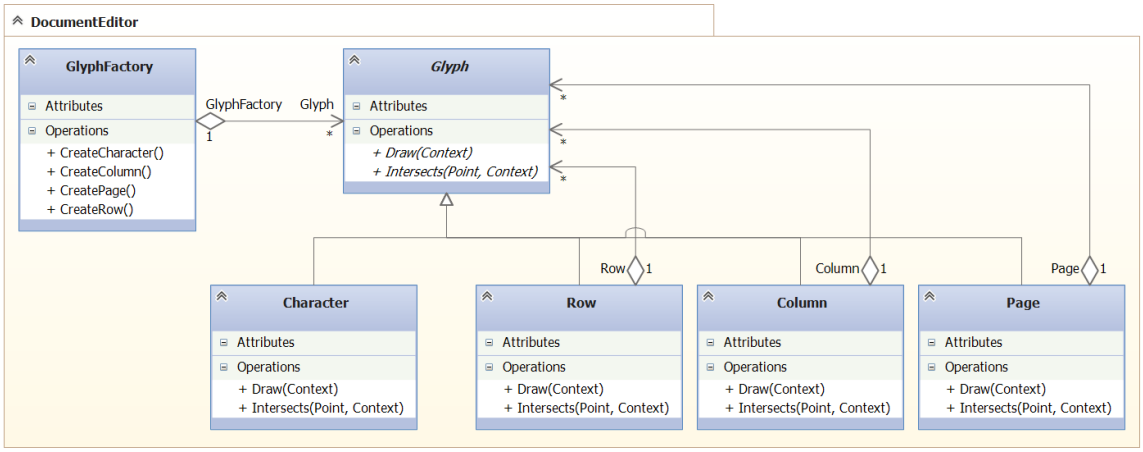
Создавать полноценную иллюзию того, что в документе имеется много экземпляров класса `Character`, помогает специальный объект класса `Graphics`, который занимается рисованием фигур и символов в области формы (`Form`).

```
Graphics graphics = window.CreateGraphics();
graphics.DrawString(character, font, brush, X, Y);
```

Экземпляр класса `Graphics`, еще называют графическим устройством, потому что у него имеется жизненная аналогия – плоттер (устройство которое имеет головку, которая по направляющим передвигается по листу бумаги и вычерчивает рисунки с большой точностью). Также как плоттер, объект класса `Graphics` «*вычерчивает*» на виртуальной форме фигуры и символы.

Каждый экземпляр класса `Character` имеет в себе ссылку на экземпляр класса `Graphics` и в любой момент может дать команду нарисовать себя на форме. Также класс `Character` содержит в себе

ссылку на объект с настройками стиля символа. Перед рисованием, стиль символа может быть изменен, таким образом один и тот же символ может рисоваться много раз, разными стилями. Основная идея такого подхода заключается в предоставлении возможности разделения состояния объекта-символа на *внутреннее* (которое хранится непосредственно в объекте-символе) и *внешнее* (которое подключается как объект со стилями). Такой подход позволяет объекту-приспособленцу разделяться. Внешнее состояние приспособленца считается неразделяемым. На диаграмме классов можно увидеть основных участников, которые входят в состав примера редактора документов.



См. Пример к главе: \011_Flyweight\002_DocumentEditor

Почему в русской интерпретации паттерн Flyweight назван – приспособленцем? Потому что поведение разделяемого объекта напоминает поведение человека – приспособленца. Приспособленец - это человек, который приспосабливается к обстоятельствам, маскируя свои истинные взгляды (чаще в жизни это слово имеет негативный оттенок). Также и разделяемый объект приспосабливается к месту его использования, маскируясь за внешним (меняющимся) состоянием.

Применимость паттерна

Паттерн Flyweight рекомендуется использовать, когда:

- В приложении требуется использовать большое количество однотипных объектов и при этом для их хранения требуется выделение большого объема памяти.
- Состояние объектов можно разделить на *внутреннее* и *внешнее*. Соответственно большие группы объектов можно заменить небольшим количеством *разделяемых объектов*.

Результаты

При использовании разделяемых объектов (приспособленцев), могут возникнуть затраты на вычисление внешнего состояния, его поиск или передачу, особенно если внешнее состояние ранее использовалось как внутреннее. Вычисления и формирование внешнего состояния обычно производят клиенты, которые используют разделяемый объект. Расходы на вычисление внешнего состояния компенсируются экономией памяти (не требуется дублировать одинаковое состояние в каждом экземпляре разделяемого объекта).

Экономия памяти возникает в следующих случаях:

- Уменьшение общего числа экземпляров.
- Сокращение объема памяти, необходимого для хранения внутреннего состояния.
- Вычисление, а не хранение внутреннего состояния.

Чем больше получится создать разделяемых объектов, тем больше будет экономия памяти. Самого большого эффекта экономии памяти возможно добиться, когда внешнее состояние вычисляется, а не хранится, соответственно за счет вычислений сокращается объем выделяемой памяти под хранение внешнего состояния.

Реализация

Полезные приемы реализации паттерна Flyweight:

- **Вынесение внешнего состояния за пределы разделяемого объекта.**
  Применимость паттерна Flyweight зависит от того, насколько легко внутреннее состояние вынести во внешнее состояние, за пределы разделяемых объектов. Вынесение состояния не даст преимуществ, если различных состояний также много, как и разделяемых объектов. Оптимальный вариант -организовать вычисление внешнего состояния, а не хранить его в памяти.

- **Управление разделяемыми объектами.**
  Клиенты не должны создавать экземпляры разделяемых объектов напрямую через вызов конструкторов. Для создания разделяемого объекта, следует воспользоваться объектом-фабрикой, которая позволит найти необходимого приспособленца. Объект-фабрика содержит в себе ассоциативный массив (ключ-значение), обращаясь к которому по ключу, можно находить нужного приспособленца. Если требуемый приспособленец отсутствует в ассоциативном массиве, то он создается, сохраняется в массиве и клиенту возвращается ссылка на него (нового приспособленца). Также требуется задуматься над созданием механизма по удалению приспособленцев, надобность в которых отпадает. Так как ассоциативный массив в ходе работы программы хранит ссылку на приспособленца, который не будет более использоваться, то механизм сборки мусора не может удалить этого приспособленца, поэтому требуется организовать явное удаление неиспользуемого приспособленца из ассоциативного массива.

Пример кода

Программный код, иллюстрирующий реализацию паттерна на языке C#.

Известные применения паттерна в .Net

Паттерн Flyweight, выражен в платформе .Net в виде работы подсистемы управления памятью и особенностью техники разделения исполняемых сущностей на экземпляры и объекты.

Паттерн Proxy

Название

Заместитель

Также известен как

Surrogate (Суррогат)

Классификация

По цели: структурный
По применимости: к объектам

Частота использования

Выше средней - 1 2 3 **4** 5

Назначение

Паттерн Proxy - предоставляет объект-заместитель для контроля доступа к другому объекту.

Введение

Как видно из альтернативного названия паттерна Proxy, его еще называют Surrogate (Суррогат). А что такое суррогат в житейском смысле?

Суррогат - (от лат. surrogatus - поставленный взамен «вместо» другого) это предмет или продукт, заменяющий собой какой-либо другой предмет или продукт, с которым он (суррогат) имеет лишь некоторые общие свойства, но суррогат не обладает всеми качествами оригинального предмета или продукта, например, суррогатом зерен кофе - являются ячмень или желуди, суррогатом сахара – является сахарин. Также суррогатом часто называют различного рода подделки (фальсифицированные продукты), например, подделанный виски или водку, кроссовки **abibas** или батарейки **Panosanic** и др.

Но для формирования метафоры, которая позволила бы быстро и афористически ярко описать принципы работы паттерна Proxy, традиционно-житейского понимания суррогата на достаточно. К реализации паттерна Proxy выдвигается ряд важных требований, а именно то, что «оригинальный (настоящий) объект» и его «суррогат» должны взаимодействовать друг с другом, а также должна быть возможность, чтобы оригинальный объект всегда мог собой заменить суррогат в месте его использования, соответственно интерфейсы взаимодействия оригинального объекта и его суррогата должны совпадать. В объективной реальности трудно найти такие случаи совместной работы оригинального объекта и его суррогата, чтобы удовлетворить всем требованиям паттерна Proxy. Поэтому, чтобы критики не говорили, что данная работа не указывает на то, что ее авторы интересовались когда-нибудь современным кинематографом, обратимся к сюжету фильма «Суррогаты» с Брюсом Уиллисом в главной роли для формирования удовлетворительной метафоры.

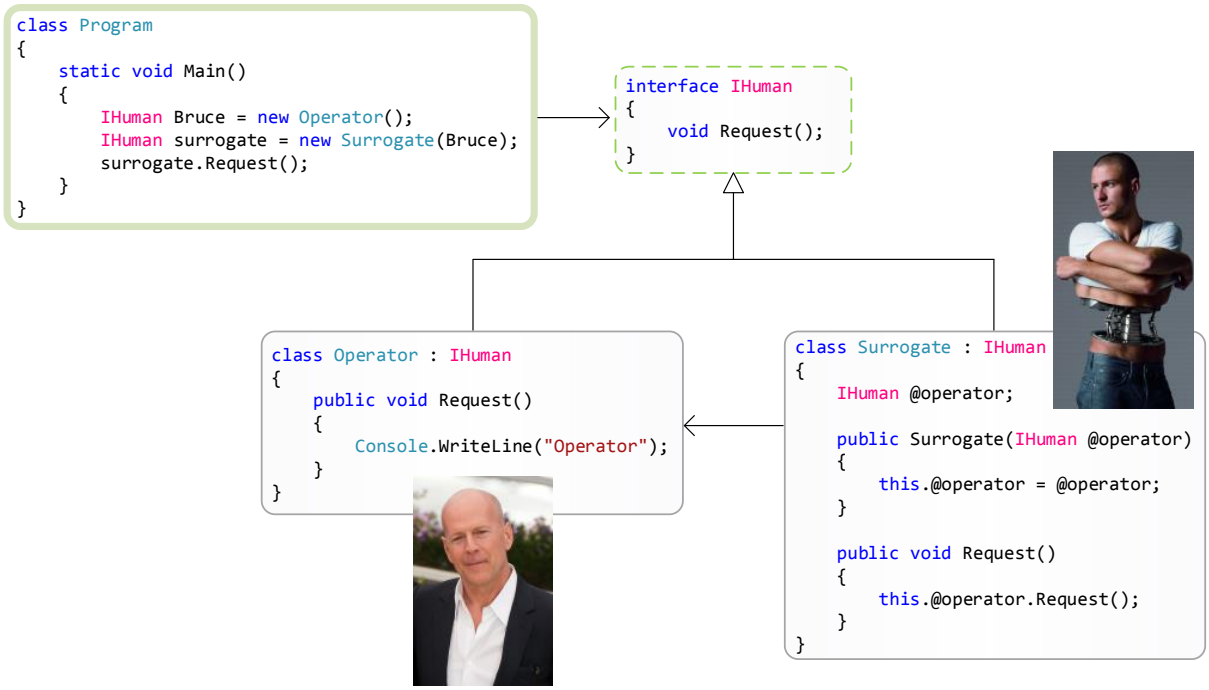
Действие разворачивается в 2057 году. Некий ученый Лайонел Кэнтор разработал нейрокомпьютерный интерфейс (систему для обмена информацией между мозгом человека и электронным устройством, например, компьютером), который позволяет человеку, силой нервных импульсов управлять роботом андроидом. При этом робот андроид внешне похож на человека оператора и ведёт себя полностью как человек. Настоящий же человек-оператор находящийся удаленно от своего робота суррогата — чувствует и переживает все ощущения, которые «воспринимает» его электронный двойник-суррогат через систему встроенных датчиков. В результате такого изобретения мир преобразился. Живые люди не выходят из своих домов, а вместо них по улицам ходят роботы-суррогаты. Главный герой фильма — агент ФБР Том Грир роль которого играет Брюс Уиллис, также, пользуется суррогатом, напоминающим по внешнему виду владельца в его молодые годы.

Мы не будем раскрывать весь сюжет кинофильма «Суррогаты», чтобы не испортить удовольствие от просмотра тем, кто намеревается посмотреть этот фильм. Рассмотрим еще только один интересующий нас момент из этого фильма.

В ходе одной очень важной спецоперации робот-суррогат Брюса Уиллиса выходит из строя, а возможность быстро приобрести нового робота по ряду причин у героя отсутствует. Но подвиги не терпят отлагательств, поэтому уже далеко не молодой Брюс Уиллис рискуя остатками здоровья и жизнью выходит на улицу полную железных суррогатов и отправляется прямо в логово сепаратистов для того чтобы самым решительным образом расправиться с их лидером.

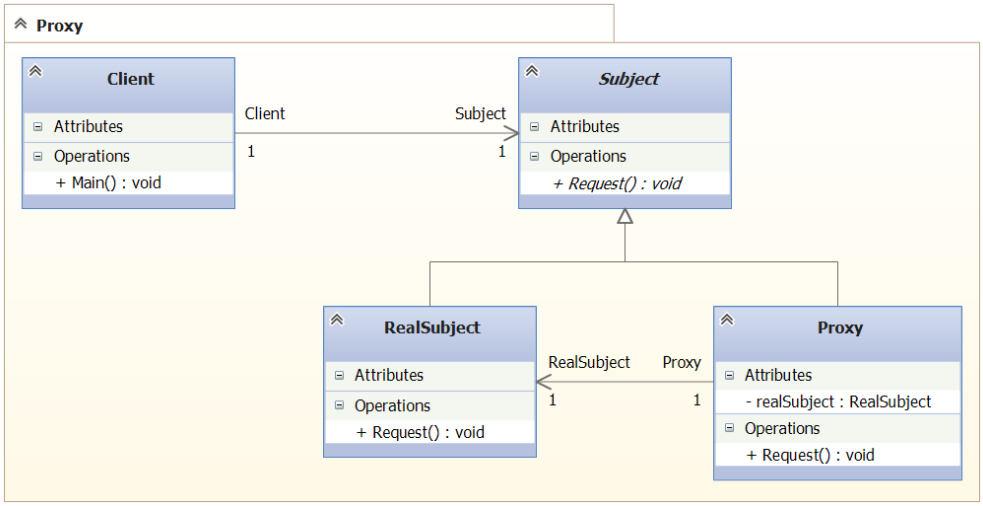
Обратите внимание на связку «человек-оператор» и его «робот-суррогат» в кинофильме «Суррогаты». Суррогат в двустороннем порядке обменивается сигналами со своим оператором (принимая и передавая сигналы), при этом сам оператор оказывается защищен от агрессивной среды обитания, но в случае острой необходимости, оператор может занять место своего суррогата, что говорит о совпадении интерфейсов взаимодействия человека-оператора и робота-суррогата.

Из сказанного выше легко сформировать модель и реализовать ее программно.



См. Пример к главе: \012_Proxy\004_Surrogates

Структура паттерна на языке UML



См. Пример к главе: \012_Proxy\001_Proxy

Структура паттерна на языке C#

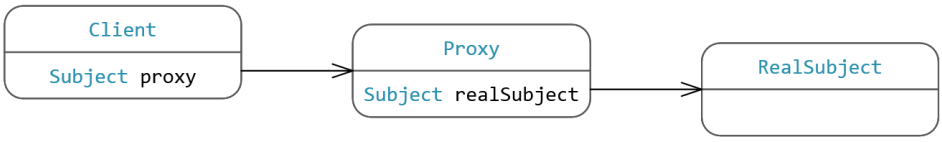
```
class Program
{
    static void Main()
    {
        Subject subject = new Proxy();
        subject.Request();
    }
}
```

```
abstract class Subject
{
    public abstract void Request();
}
```

```
class RealSubject : Subject
{
    public override void Request()
    {
        Console.WriteLine("RealSubject");
    }
}
```

```
class Proxy : Subject
{
    RealSubject realSubject;

    public override void Request()
    {
        if (realSubject == null)
            realSubject = new RealSubject();

        realSubject.Request();
    }
}
```

Так может выглядеть диаграмма объектов при использовании паттерна Proxy:



См. Пример к главе: \012_Proxy\001_Proxy

Участники

- `Proxy` - **Заместитель**:
  Представляет собой класс объекта-заместителя. Объект заместитель хранит в себе ссылку на реальный субъект, что позволяет заместителю обращаться к реальному субъекту напрямую. Заместитель имеет такой же интерфейс, как и реальный субъект, что позволяет в нужный момент подставлять заместителя вместо реального субъекта и наоборот. Заместитель контролирует доступ к реальному субъекту и может отвечать за создание экземпляра реального субъекта если это может оказаться необходимым. Прочие обязанности заместителя зависят от вида заместителя. Заместители бывают четырех видов: удаленный заместитель (посол), виртуальный заместитель, защищающий заместитель и заместитель – умная ссылка.

- `Subject` - **Субъект**:
  Предоставляет общий интерфейс для Proxy и RealSubject. Proxy возможно использовать везде где ожидается использование RealSubject.

- `RealSubject` - **Реальный субъект**:
  Представляет собой класс объекта, для которого требуется создание заместителя.

Отношения между участниками

Отношения между классами

- Класс `RealSubject` связан связью отношения наследования с абстрактным классом `Subject`.

- Класс `Proxy` связан связью отношения наследования с абстрактным классом `Subject` и связью отношения ассоциации с классом `RealSubject`.

Отношения между объектами

- Объект-заместитель класса `Proxy`, переадресует клиентские запросы целевому объекту класса `RealSubject`.

Мотивация

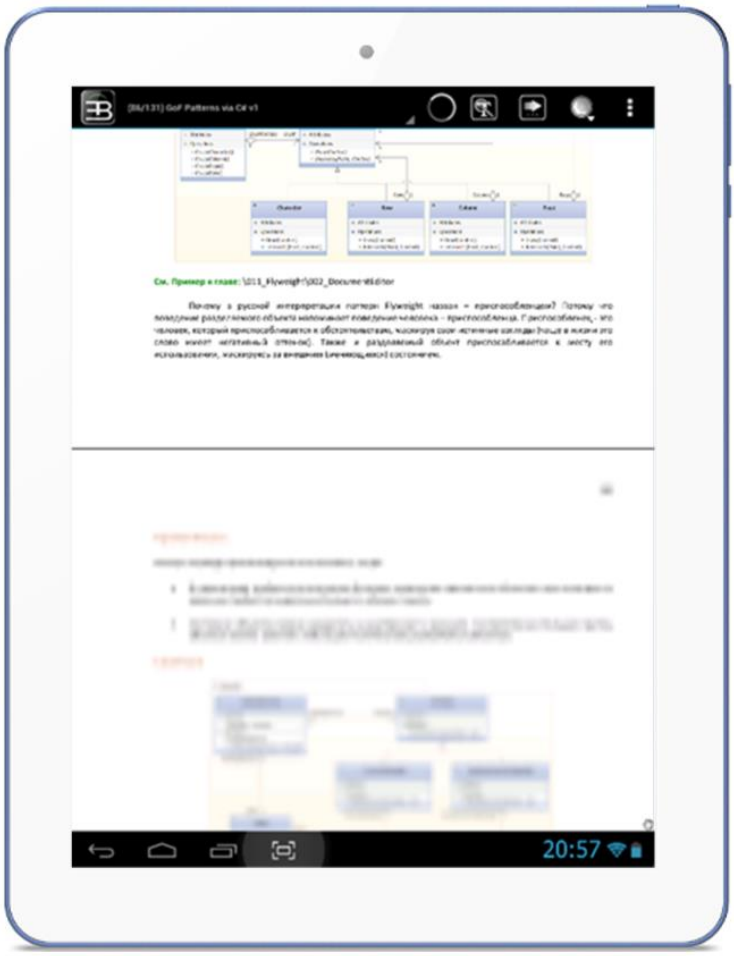
Предлагается рассмотреть редактор документов, который позволяет встраивать в документ изображения (графические объекты). При отображении графических объектов большого размера могут возникать задержки. Но документ должен открываться быстро, поэтому не следует создавать сразу все графические объекты на стадии открытия документа, а создавать их только тогда, когда потребуется показать изображение пользователю.

Вместо реального изображения, можно временно использовать его упрощенный суррогат (заместитель). В роли заместителя можно использовать изображение с более низким разрешением (preview или thumbnail), которое быстрее отобразится на странице в момент, когда изображение станет видимым пользователю.

Например, при использовании EBookDroid (программа просмотра документов и электронных книг) можно увидеть эффект задержки полного отображения новой страницы. Такая задержка может быть связана с аппаратной составляющей (слабый процессор, мало памяти).

На рисунке видно, что, в момент «*перелистывания*» страниц, на появившейся странице (снизу) контент отображается с более низким разрешением чем на странице (сверху).

У разработчиков приложения может быть несколько вариантов организации отображения новой страницы: Первый вариант - отобразить пустую страницу и ожидать пока загрузится контент. Второй вариант – отображать информацию на странице по мере загрузки, что тоже заставляет ждать пользователя. Третий вариант – показать пользователю страницу в более низком качестве, при этом пользователь по очертаниям и силуэтам может принять решение – остаться на данной странице и ожидать ее полной загрузки или продолжить «*листать*» книгу.

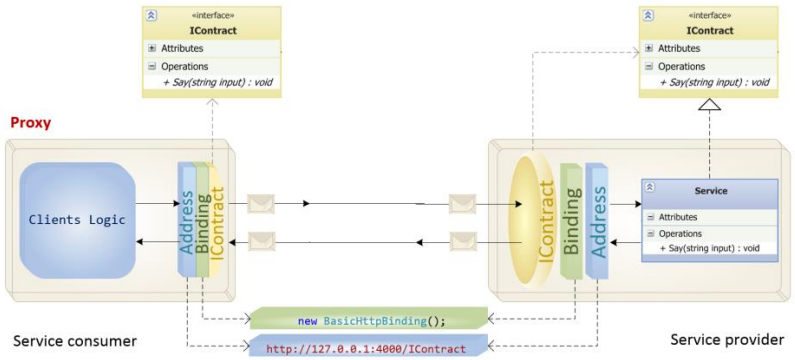


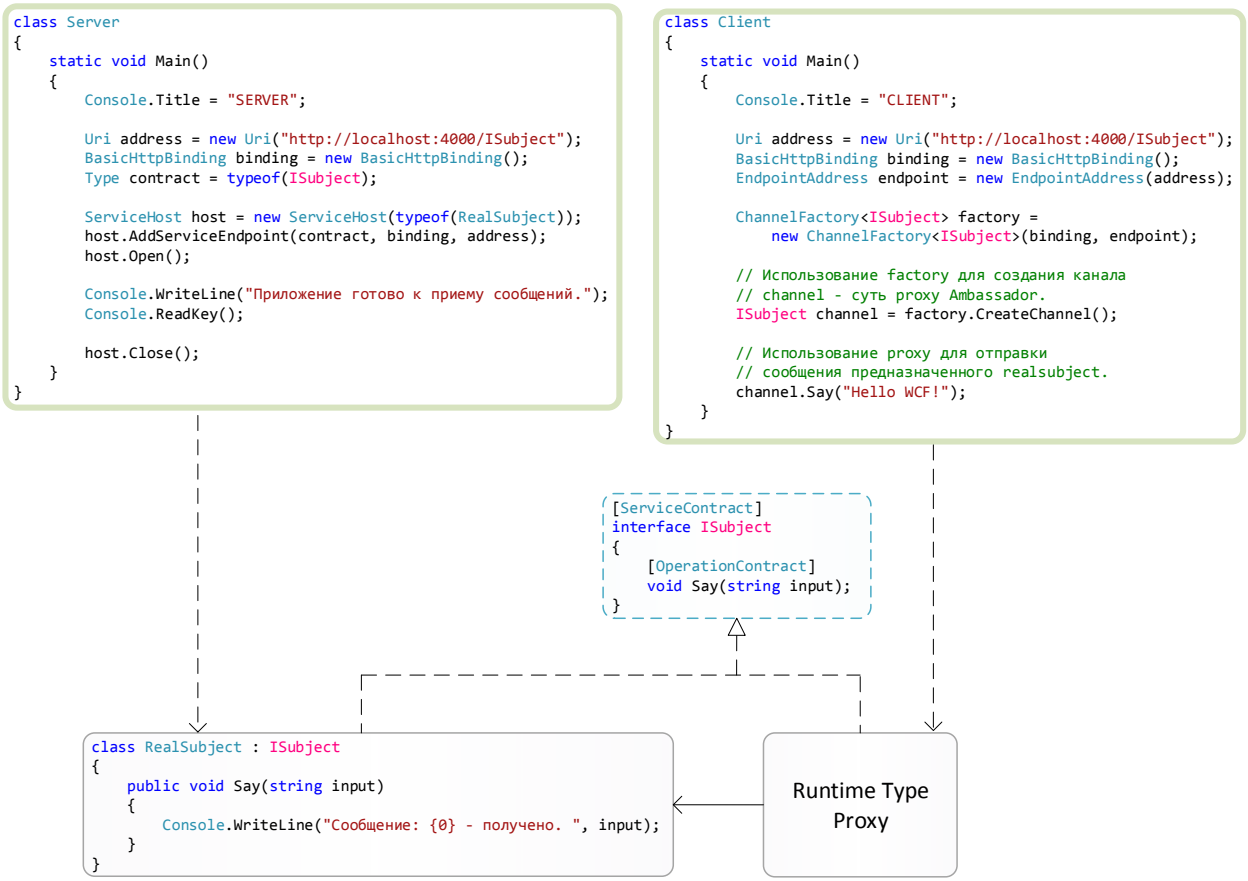
Применимость паттерна

Паттерн Proxy рекомендуется использовать, когда требуется воспользоваться целевым объектом не напрямую, а через объект-заместитель.

Объект-заместитель имеет четыре технологические разновидности:

- **Удаленный заместитель («Посол» или «Ambassador»).**
  Удаленный объект-Proxy – это объект, который находится в другом адресном пространстве относительно целевого объекта и обеспечивает доступ к целевому объекту. Например, при использовании технологии WCF создается объект прокси представляющий собой обертку для сервиса-потребителя (consumer), который связывается с сервисом-поставщиком (provider).





См. Пример к главе: \012_Proxy\005_Ambassador

- **Виртуальный заместитель.**
  Виртуальный заместитель - это объект, который создает «тяжелые» объекты по требованию. Как бывает в жизни. Например, человек хочет приобрести автомобиль. Прежде чем его купить, клиент обычно пользуется объектом-заместителем этого автомобиля, а именно буклетом, который описывает автомобиль. И только после того как клиент убедится, что это именно то что ему требуется, едет в автосалон на «*test-drive*». Пример и техническое описание виртуального заместителя представлены в разделе «Мотивация».
  См. Пример к главе: \012_Proxy\002_ImageProxy

- **Защищающий заместитель.**
  Защищающий заместитель, контролирует доступ к своему целевому объекту. Такие заместители используются, когда требуется установить различные права доступа к целевому объекту. Например, при реализации подхода CRUD популяризированного Джеймсом Мартином. CRUD – (Create, Read, Update, Delete – Создание, Чтение, Обновление, Удаление).
  См. Пример к главе: \012_Proxy\ 003_ProxyCRUD

- **Заместитель «Умная ссылка».**
  Объект «умная ссылка» - представляет собой объектно-ориентированное представление обычного указателя (адреса переменной или метода в памяти). В качестве примера умной ссылки в языке C# можно привести такие синтаксические конструкции как делегат (`delegate`), критическая секция (`lock`), оператор автоматической генерации программного кода итератора (`yield`), переменная запроса LINQ, операторы автоматической генерации программного кода асинхронного выполнения метода (`async` и `await`). Предлагается рассмотреть каждую из упомянутых конструкций, представляющую собой неявное выражение паттерна Proxy вида «умная ссылка».

    Делегат (`delegate`) как «умная ссылка», представляет собой функционально-ориентированную сущность в объектно-ориентированном представлении, предназначенную для хранения и передачи указателя (адреса первого байта тела метода в памяти) на метод. Умность

делегата заключается в том, что программисту не требуется напрямую работать с числовым представлением адреса первого байта тела метода, а также в наличии дополнительной функциональности которая обслуживает хранящийся в делегате указатель. Но не принято «языком ООП» говорить, что мы вызываем метод по указателю содержащемся в делегате, правильно сказать, что мы вызываем метод сообщенный с делегатом. Вызывая метод Invoke, говорят: - «Мы вызываем метод сообщенный с делегатом.» (подразумевая синхронный вызов метода), а вызывая метод BeginInvoke говорят: - «Мы асинхронно вызываем метод сообщенный с делегатом.».

Критическая секция (lock) как «умная ссылка», представляет собой безопасную конструкцию по работе с объектом синхронизации доступа к разделяемому ресурсу, между несколькими потоками (нитями). Умность конструкции lock заключается в том, что она избавляет программиста использовать напрямую вызовы методов Monitor.Enter и Monitor.Exit, использование которых напрямую зачастую приводит к трудно обнаруживаемым ошибкам.

Оператор автоматической генерации программного кода итератора (yield) как «умная ссылка», представляет собой быстрый и безопасный способ получения объекта-итератора. Умность оператора yield заключается в том, что программисту не требуется утруждать себя пониманием устройства сложного итератора, например, создание внутреннего (пассивного) итератора вручную, не всегда оказывается легкой задачей.

Переменная запроса LINQ, как «умная ссылка», представляет собой переменную содержащую ссылку на коллекцию, которую возвращает связанное с ней (*ней - переменной запроса*) выражение запроса LINQ. Умность переменной запроса LINQ заключается в том, что она принимает в себя ссылку на коллекцию (другими словами инициирует выполнение выражения запроса) только тогда, когда эту переменную начинают использовать фактически, если в процессе выполнения программного модуля к переменной запроса не будет произведено обращение, то и выражение запроса не будет выполняться исполняющей средой. Понятно, что любое выражение запроса LINQ трансформируется в последовательность вызовов расширяющих методов, которые могут формировать определенную коллекцию, как результат своего выполнения, что не всегда может положительно сказаться на производительности в случае хаотичного или одновременного выполнения всех имеющихся выражений запроса в программе.

Связка операторов автоматической генерации программного кода, обслуживающего асинхронное выполнение метода (async и await) как «умная ссылка», представляют собой генератор машины состояния для асинхронного выполнения. Умность связки async и await заключается в том, что большинство рутинных операций по организации асинхронной обработки эта связка берет на себя. Например, организация «асинхронной машины состояний» (конечного автомата, обслуживающего асинхронные вызовы – реализация методов MoveNext и SetStateMachine интерфейса IAsyncStateMachine), организация цепочек «продолжений» асинхронного выполнения нескольких операций и др.

Результаты

Паттерн Proxy позволяет дополнительно защитить (скрыть) субъект при попытке доступа к нему, добавляя уровень косвенности во взаимоотношениях объектов. В зависимости от варианта применения паттерна можно выделить следующие полезные особенности:

- **удаленный заместить может скрывать реальное расположение субъекта от клиента** (в адресном пространстве диска или в памяти).
- **виртуальный заместитель может оптимизировать ресурсы**, создавая субъекты по требованию, реализуя технику отложенной (или ленивой) инициализации.
- **защищающий заместитель и «умная» ссылка позволяют проводить дополнительные действия** (например, верификацию, валидацию, кеширование и т.п.) при попытке доступа к субъекту.

Дополнительно паттерн Proxy может скрывать от клиента возможность копирования при записи (copy-on-write). Эта оптимизация очень похожа на технику создания объектов по требованию, она позволяет экономить ресурсы при копировании больших и сложных субъектов. Если проводить действия с суррогатом субъекта вместо оригинала (может требовать значительно меньше вычислительных ресурсов чем действия с оригиналом) и организовать отложенное копирование таким образом, чтобы оно производилось только тогда, когда окончена модификация суррогата и он отличается от первоначального оригинала, то можно существенно уменьшить плату за копирование «тяжелых» субъектов.

Реализация

- **Заместитель может не владеть информацией о типе реального объекта, который необходимо заместить.** Для реализации такой возможности достаточно организовать класс `Proxy` так чтобы он мог работать субъектом только через его абстрактный интерфейс (абстрактный класс `Subject`), в таком случае класс `Proxy` сможет обращаться к любому конкретному классу, реализующему абстрактный интерфейс `Subject`, единообразно. Важно заметить, что если заместитель должен инстанцировать реальных субъектов (в случае с виртуальным заместителем, например), то знание конкретного класса является обязательным.

При реализации паттерна `Proxy` может возникнуть проблема при обращении к еще не инстанцированному экземпляру класса `Subject`. Она возможна в случае, когда заместитель должен работать со своими субъектами вне зависимости от реального места расположения экземпляров субъектов (на диске – персистентные, устойчивые экземпляры, в памяти – временные). В таком случае, нужно использовать при реализации такую форму идентификации, которая бы не зависела от адресного пространства. В примере из раздела «Пример кода» в качестве такого идентификатора используется имя файла.

Пример кода

Рассмотри пример реализации виртуального заместителя. В классе `Graphic` определен абстрактный интерфейс для графических объектов:

```
abstract class Graphic
{
    public string fileName;
    abstract public void Draw();
    abstract public void Load();

    public Image PictureToShow { get; set; }
}
```

Класс `Picture` реализует интерфейс `Graphic`, позволяя работать с файлами картинок:

```
class Picture : Graphic
{
    public Picture(string fileName)
    {
        this.fileName = fileName;
    }

    public override void Draw()
    {
        PictureToShow = Image.FromFile(fileName);
    }

    public override void Load()
    {
        throw new InvalidOperationException();
    }
}
```

Класс `PictureProxy` представляет собой конкретную реализацию виртуального заместителя, которая имеет тот же интерфейс, что и класс `Picture`:

```
class PictureProxy : Graphic
{
    Picture picture;

    public PictureProxy(string fileName)
    {
        this.fileName = fileName;
```

```
            PictureToShow = new Bitmap(Resources.startImg, 52, 62);
        }

        public override void Draw()
        {
            if (picture == null)
            {
                picture = new Picture(fileName);
            }
            picture.Draw();
        }

        public override void Load()
        {
            PictureList.listPictures[PictureList.listPictures.IndexOf(this)] =
this.picture;
        }
    }
```

В конструктор этого класса производится сохранение локальной копии имени файла и инстанцируется уменьшенное кешированное изображение, которое становится доступным через свойство `PictureToShow`.

В результате использования паттерна `Proxy` в клиентской части программы появляется возможность использовать кешированные миниатюрные изображения для предпросмотра графических файлов.

Известные применения паттерна в .Net

`Microsoft.Build.Tasks.Deployment.ManifestUtilities.ProxyStub`
http://msdn.microsoft.com/ru-ru/library/microsoft.build.tasks.deployment.manifestutilities.proxystub(v=vs.110).aspx

`System.Runtime.Remoting.Proxies.ProxyAttribute`
http://msdn.microsoft.com/en-us/library/system.runtime.remoting.proxies.proxyattribute(v=vs.110).aspx

`System.Web.Script.Services.ProxyGenerator`
http://msdn.microsoft.com/ru-ru/library/system.web.script.services.proxygenerator(v=vs.110).aspx

`System.Web.UI.WebControls.WebParts.ProxyWebPart`
http://msdn.microsoft.com/ru-ru/library/system.web.ui.webcontrols.webparts.proxywebpart(v=vs.90).aspx

`System.Net.Configuration.DefaultProxySection`
http://msdn.microsoft.com/en-us/library/system.net.configuration.defaultproxysection(v=vs.110).aspx

`System.Net.GlobalProxySelection`
http://msdn.microsoft.com/ru-ru/library/system.net.globalproxyselection(v=vs.110).aspx

`System.Net.IWebProxy`
http://msdn.microsoft.com/ru-ru/library/system.net.iwebproxy(v=vs.118).aspx

`System.Reflection.AssemblyNameProxy`
http://msdn.microsoft.com/ru-ru/library/system.reflection.assemblynameproxy(v=vs.110).aspx

`System.Runtime.Remoting.Proxies.RealProxy`
http://msdn.microsoft.com/en-us/library/system.runtime.remoting.proxies.realproxy(v=vs.110).aspx

`System.Web.UI.Design.ScriptManagerProxyDesigner`

http://msdn.microsoft.com/ru-ru/library/system.web.ui.design.scriptmanagerproxydesigner(v=vs.90).aspx

System.Runtime.Serialization.SurrogateSelector
http://msdn.microsoft.com/en-us/library/system.runtime.serialization.surrogateselector(v=vs.110).aspx

System.Runtime.Remoting.Messaging.RemotingSurrogateSelector
http://msdn.microsoft.com/ru-ru/library/system.runtime.remoting.messaging.remotingsurrogateselector(v=vs.110).aspx

Глава 4. Паттерны поведения

Паттерны поведения описывают правильные способы организации взаимодействия между используемыми объектами.

Паттерн Chain of Responsibility

Название

Цепочка обязанностей

Также известен как

-

Классификация

По цели: поведенческий
По применимости: к объектам

Частота использования

Ниже средней - 1 **2** 3 4 5

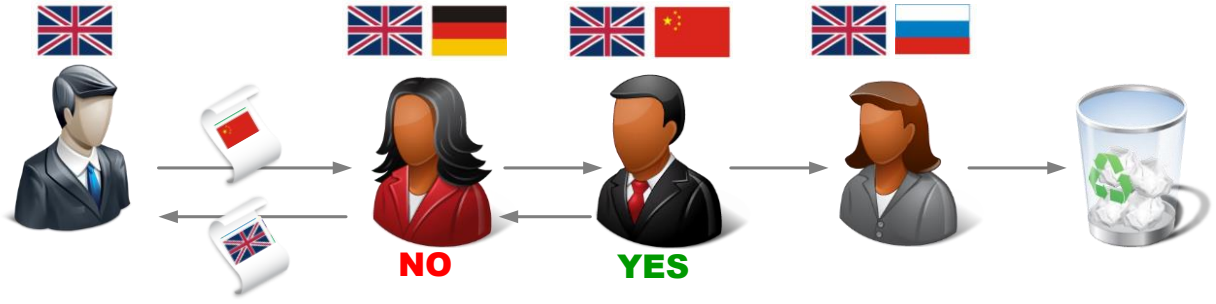
Назначение

Паттерн Chain of Responsibility - позволяет избежать привязки объекта-отправителя запроса к объекту-получателю запроса, при этом давая шанс обработать этот запрос нескольким объектам. Паттерн Chain of Responsibility – связывает в цепочку объекты-получатели запроса и передает запрос вдоль этой цепочки, пока один из объектов, составляющих эту цепочку не обработает передаваемый запрос.

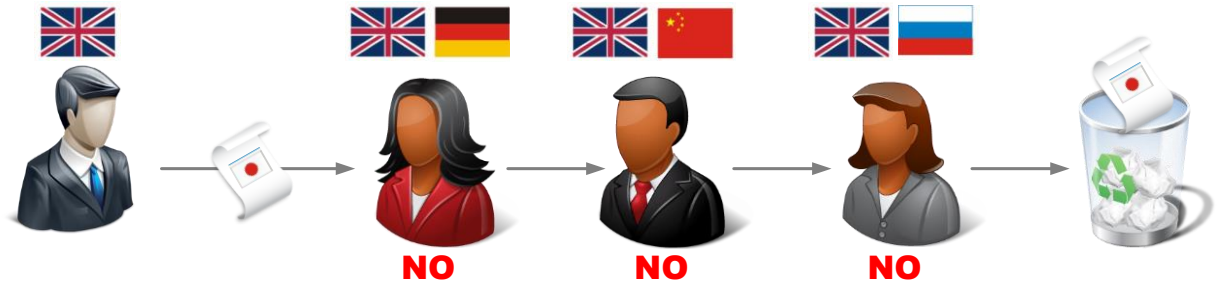
Введение

В реальной жизни, сложно найти рациональный аналог модели, которую описывает паттерн Chain of Responsibility. Для лучшего понимания работы паттерна можно привести нереалистичный пример. Например, предлагается рассмотреть работу бюро переводов с английского языка на другие иностранные языки. Имеется три переводчика англо-немецкий, англо-китайский и англо-русский. Переводчики сидят за столами один за другим. Когда клиент приходит в бюро то он имеет дело только с одним переводчиком, который сидит за первым столом (англо-немецким). Клиент дает англо-немецкому переводчику документ для перевода на определенный язык. Англо-немецкий (первый в цепочке) переводчик определяет язык перевода, и если он в состоянии сделать перевод, то он переводит документ, если не в состоянии, то передает документ дальше по цепочке (сидящему у него за спиной) англо-китайскому переводчику и так далее.

Ниже на рисунке показано, что клиенту требуется сделать перевод документа с китайского языка на английский. Клиент дает документ на китайском языке англо-немецкому переводчику. Англо-немецкий переводчик понимает, что не в состоянии сделать перевод и передает документ дальше по цепочке, следующему переводчику. Следующим переводчиком оказывается англо-китайский переводчик, который в состоянии сделать перевод. Англо-китайский переводчик делает перевод документа на английский язык и возвращает документ клиенту. Англо-русский переводчик остался не задействованным в данном случае.

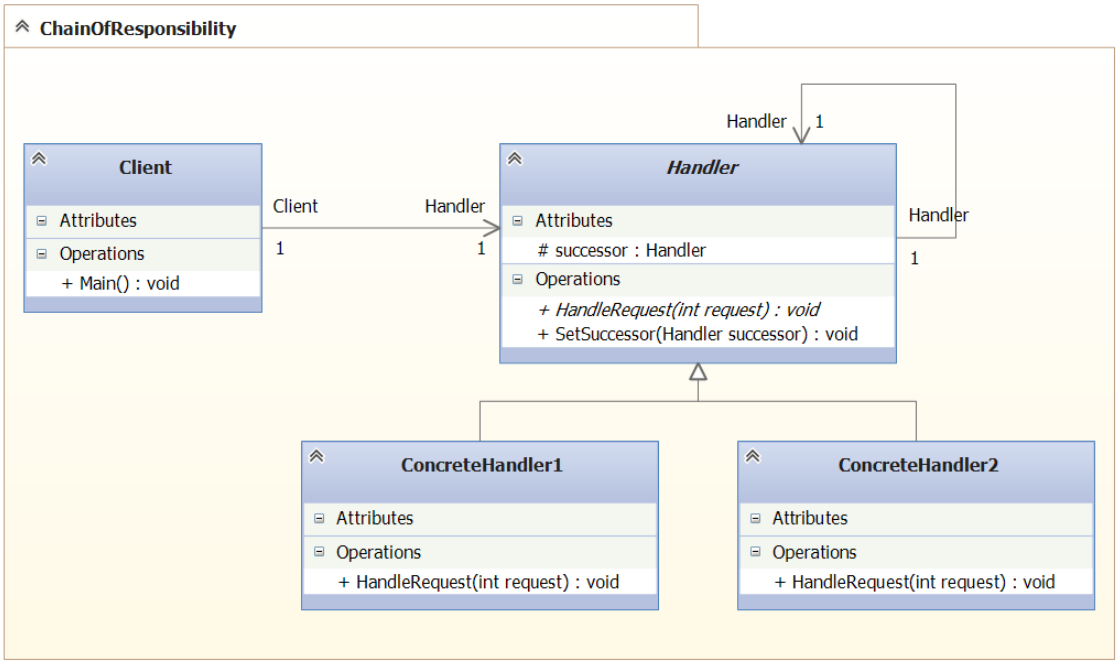


Если ни один из переводчиков не может перевести документ, то документ теряется. Ниже на рисунке показано, что клиенту требуется сделать перевод документа с японского языка на английский. Клиент дает документ на японском языке англо-немецкому переводчику. Англо-немецкий переводчик понимает, что не в состоянии сделать перевод и передает документ дальше по цепочке, следующему переводчику. Следующим переводчиком оказывается англо-китайский переводчик. Англо-китайский переводчик понимает, что не в состоянии сделать перевод и передает документ дальше по цепочке, следующему переводчику. Следующим переводчиком оказывается англо-русский переводчик. Англо-русский переводчик понимает, что не в состоянии сделать перевод и так как за ним уже нет больше переводчиков то просто выбрасывает документ в корзину с мусором (условно).



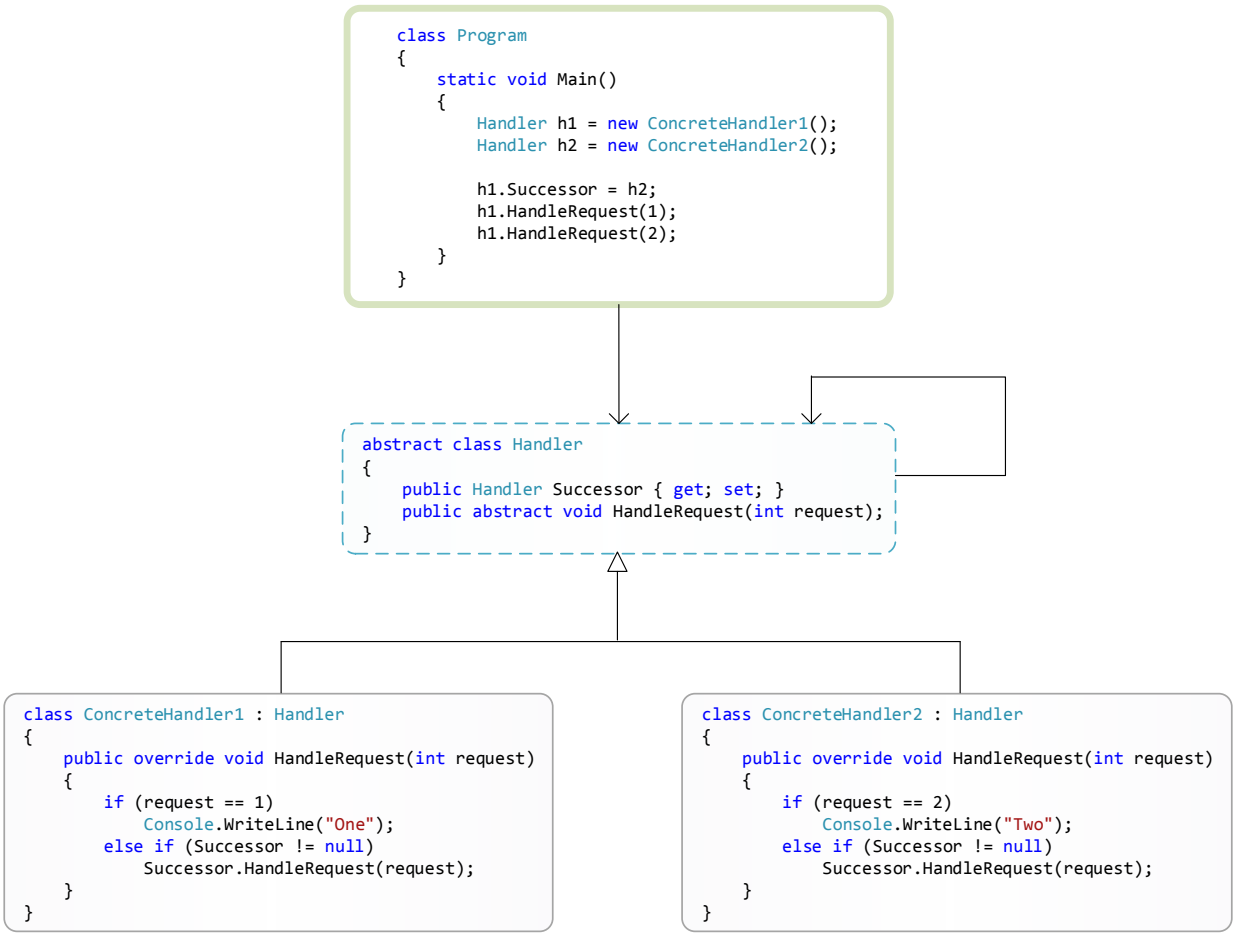
Положительные стороны такого подхода заключаются в том, что клиент не знает о том переводчике который делает перевод, при этом каждый из переводчиков знает только об одном переводчике (который у него за спиной), что обеспечивает слабую связанность всей системы. Недостаток подхода в том, что нет никаких гарантий что перевод будет выполнен, так как документ может достичь конца цепочки и пропасть.

Структура паттерна на языке UML

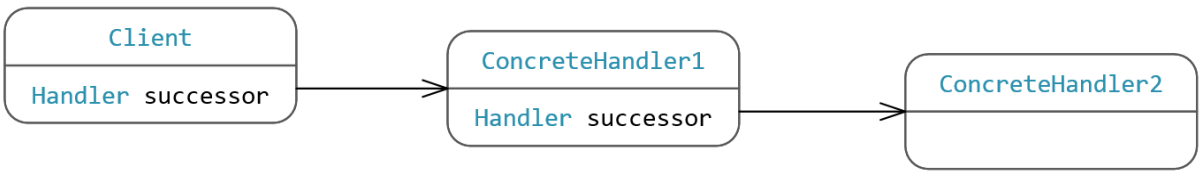


См. Пример к главе: \013_Chain of Responsibility\001_Chain

Структура паттерна на языке C#



Структура объектов:



См. Пример к главе: \013_Chain of Responsibility\001_Chain

Участники

- `Handler` - **Обработчик**:
  Предоставляет интерфейс для обработки запросов.

- `ConcreteHandler` - **Конкретный обработчик**:
  Обрабатывает запрос, который предназначен ему. Имеет ссылку на своего преемника, которому передает запрос в случае, если сам не в состоянии его обработать.

- `Client` - **Клиент**:
  Посылает запрос определенному объекту класса `ConcreteHandler`, который находится в цепочке объектов-обработчиков.

Отношения между участниками

Отношения между классами

- Абстрактный класс `Handler` связан связью отношения самоассоциации.

- Конкретные классы `ConcreteHandler1` и `ConcreteHandler2` связаны связью отношения наследования с абстрактным классом `Handler`.

Отношения между объектами

- Когда клиент посылает запрос, этот запрос передается по цепочке, пока один из объектов `ConcreteHandler` не обработает этот запрос.

Мотивация

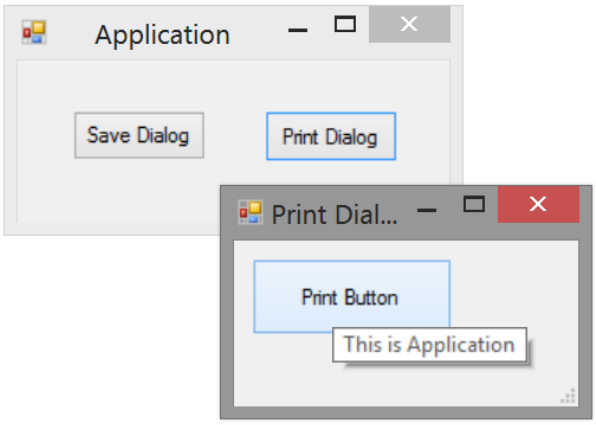
Предлагается рассмотреть в качестве примера реализацию контекстно-зависимой оперативной справки в графическом интерфейсе пользователя. Пользователь может получить информацию, описывающую части пользовательского интерфейса, просто наведя курсор мыши на интересующий элемент управления. Содержание справки обусловлено контекстом, т.е. зависит от того на какой именно форме, в какой именно ее части находится интересующий элемент управления. Например, справка по кнопке в диалоговом окне может отличаться от справки по кнопке с таким же именем и нанесенным текстом в главном окне приложения. В случае если для какого элемента интерфейса нет справочных данных, то система должна отобразить информацию об окружающем этот элемент ближайшем контексте, например, о форме или окне в целом.

Исходя из выше сказанного, логично было бы организовать справочную информацию своеобразной цепочки, от конкретных разделов до более общих.

Проблема организации выдачи контекстно-зависимой справочной информации заключается в том, что объект, который является инициатором запроса (например, кнопка, на которую навел курсор пользователь) не владеет информацией о том, какой объект в конечном итоге предоставит справку. В связи с этим, нужно каким-то образом разделить иерархии объектов – возможных инициаторов запросов справочной информации и объектов, хранящих в себе справочные данные и предоставляющие их по требованию. Паттерн Chain of Responsibility демонстрирует, как этого можно добиться.

Идея паттерна заключается в том, чтобы дать возможность обработать запрос от объекта-инициатора нескольким объектам-получателям запроса и таким образом разорвать биективную связь от конкретного объекта-отправителя к конкретному объекту-получателю/обработчику. Таким образом, запрос будет перемещаться от объекта-отправителя по цепочке объектов-обработчиков, пока один из них не выполнит запрос на предоставление справочной информации. Нужно заметить, что у каждого объекта-обработчика имеется возможность либо обработать запрос (если есть такая возможность), либо передать его другому объекту-обработчику (если такой возможности нет), который находится далее в цепочке контекстов. В результате такой реализации, когда отправитель не знает о конкретном получателе запроса, появляется понятие *анонимного получателя* (implicit receiver).

Предположим, пользователь наводит курсор на кнопку `printButton`, которая находится в диалоговом окне `printDialog`. Приведенная ниже диаграмма взаимодействий показывает, как запрос на получение справки перемещается по цепочке объектов-обработчиков.



В данном случае ни кнопка `printButton`, ни форма `printDialog`, ни даже объекты-обработчики типов `ButtonHandler` или `DialogHandler` не обрабатывают запрос на предоставление справки, и он достигает объекта-обработчика `aplicationHandler` типа `ApplicationHandler`, который может его обработать или игнорировать. Таким образом, объект-отправитель не знает о том какой именно объект-получатель обработает его запрос.

Чтобы гарантировать анонимность получателя, все объекты в цепочке (`ButtonHandler`, `DialogHandler`, `ApplicationHandler`) имеют единый интерфейс `IHelpHandler` для обработки запросов и для доступа к следующему в цепочке объектов-обработчиков. Для обработки запросов классы `ButtonHandler`, `DialogHandler` и `ApplicationHandler` используют метод `HandleHelp`, где по умолчанию запрос передается следующему в цепочке объекту-обработчику, так называемому преемнику.

См. Пример к главе: \013_Chain of Responsibility\002_ContextsHelp

Применимость паттерна

Паттерн Chain of Responsibility рекомендуется использовать, когда:

- Имеется несколько объектов и несколько разновидностей (одноименных) запросов к этим объектам. Каждый объект может обработать только один запрос, предназначенный ему. Отправитель запросов (клиент), не знает заранее какой объект должен обработать определенный запрос, следовательно, объект обработчик запроса должен быть найден автоматически, без участия клиента. При наличии большого числа объектов-обработчиков, клиенту проще запомнить название запроса (действия), чем всех возможных исполнителей.

- Требуется динамически задавать набор объектов-обработчиков запроса.

Результаты

Паттерн Chain of Responsibility обладает следующими преимуществами:

- **Позволяет ослабить связанность между объектами.**
  Паттерн Chain of Responsibility избавляет клиента от необходимости знать, какой объект обработает его запрос. Клиенту и обработчику запроса, ничего не известно друг о друге. «Объект-звено» цепочки ничего не знает о структуре цепочки, в которую он включен.

- **Добавляет гибкости при распределении обязанностей между объектами.**
  Паттерн Chain of Responsibility позволяет добавлять новые и изменять существующие объекты-обработчики входящие в состав цепочки.

Паттерн Chain of Responsibility обладает следующим недостатком:

- **Обработка запроса не гарантирована.**
  В связи с тем, что у запроса может не оказаться в цепочке объекта, который в состоянии обработать данный запрос, запрос может достичь конца цепочки и пропасть.

Реализация

Особенности, которые следует учесть при реализации паттерна Chain of Responsibility:

- **Реализация цепочки преемников.**
  Существует несколько вариантов реализации:
  - Использовать существующие связи между объектами (например, между кнопкой и формой на которой она размещена, существует связь через свойство `Parent`)
    Использование уже существующих связей, которые, уже поддерживают необходимую цепочку, позволяет уйти от использования дублирующих связей. Но если существующая цепочка не отображает необходимой цепочки обязанностей, то построения необходимой цепочки связей избежать не удастся.
  - Определить новые связи (через свойство `Successor` реализованное в абстрактном классе `Handler`, и, возможно, переопределенное в конкретных классах-наследниках `ConcreteHandler1` и `ConcreteHandler2`).

- **Соединение преемников.**
  Если создание цепочки связей является необходимостью, то реализация метода `HandleRequest` в классе `Handler` по умолчанию будет логичной. Этот метод будет перенаправлять запрос преемнику, если тот существует, используя ссылку на преемника, которая доступна через свойство `Successor`. Такой подход позволит наследникам абстрактного класса `Handler` не переопределять метод `HandleRequest` если не планируется его специфической обработки на уровне такого наследника. В таком случае, запрос по умолчанию передастся следующему объекту-обработчику в цепочке.

```
abstract class HelpHandler
{
     public HelpHandler Successor { get; set; }
     public virtual void HandleHelp(HelpHandler h)
     {
         if(Successor!=null)
         {
             Successor.HandleHelp(h);
         }
     }
}
```

- **Представление запросов.**
  В самом простом варианте, например, в случае абстрактного класса `HelpHandler` из раздела мотивации, запрос жестко кодируется в виде вызова метода `HandleHelp`. С одной стороны, такой подход придает определенное удобство и обеспечивает безопасность, но с другой стороны, при этом переадресовывать можно только фиксированное количество запросов, которые определяются интерфейсом заданным абстрактным классом `Handler`. Существует альтернативный подход – использовать метод-обработчик, который в качестве параметра принимает определенный код запроса, и в зависимости от значения этого кода, обрабатывает запрос определенным образом. Так можно организовать обработку заранее известного количества различных запросов.

  Единственное, о чем нужно позаботиться в данном случает, это организации системы кодирования-декодирования, которая поддерживалась бы и объектами-отправителями, и объектами-обработчиками запросов. Естественно, что при этом могут возникнуть проблемы с безопасностью, т.к. такой подход менее безопасен, чем прямые вызовы методов. Решить эту проблему может использование отдельных объектов-запросов, в которых бы инкапсулировались параметры запроса. Такой подход придаст дополнительную гибкость в реализации, поскольку через

наследование от базового класса Request можно порождать новые типы запросов, которые могут иметь дополнительные параметры. В таком случае, объект обработчик должен уметь идентифицировать наследников класса Request и особым способом обрабатывать каждую разновидность. Этого можно добиться, задав в классе Request интерфейс метода доступа, который бы возвращал идентификатор класса. Но такой механизм диспетчеризации нужен, если язык программирования не поддерживает механизмы определения типов, для того чтобы объект обработчик мог определить тип запроса самостоятельно. Платформа .Net позволяет это сделать используя как метод GetType механизма рефлексии, так и ключевые слова is и as.

Следует обратить внимание, что использование GetType совместно с оператором typeof не идентично использованию is или as. Нужно учитывать то, что оператор typeof возвращает имя типа, которое указано при компиляции и не учитывает дерево наследования типов, а GetType возвращает имя типа во время инстанцирования экземпляра, учитывая дерево наследования типов, аналогично работают и операторы is и as т.е. учитывая *upcasting*.

Вот как использована такая возможность в классе DialogHandler из раздела Мотивация:

```
class DialogHandler : IHelpHandler
{
    ToolTip tt = new ToolTip();
    public void HandleHelp(Control ctrl)
    {
        if (ctrl is Dialog && ctrl.Name == "saveDialog")
        {
            tt.Show("This is Save Dialog", ctrl);
        }
        else if (Successor != null)
            Successor.HandleHelp(ctrl);
    }

    public IHelpHandler Successor { get; set; }
}
```

Подклассы абстрактного класса Handler могут расширять схему диспетчеризации, переопределяя метод HandleRequest так чтобы обрабатывать интересующие запросы, а остальные переадресовывать родительскому классу. В таком случае подкласс будет расширять, а не замещать базовый метод HandleRequest родителского класса.

Пример кода

Приведем основные классы. Класс Application является классом формы основного окна программы и в нем определяются методы-обработчики событий наведения курсора на кнопку или форму, а также объявляются объекты-обработчики для соответствующих событий. Процесс создания дочернего диалогового окна и кнопок на нем предлагается рассмотреть в примерах к главе.

```
public partial class Application : Form
{
    Dialog dialog;
    IHelpHandler buttonHandler;
    IHelpHandler dialogHandler;
    IHelpHandler aplicationHandler;
    public Application()
    {
        InitializeComponent();
    }

    private void Application_MouseHover(object sender, EventArgs e)
    {
        this.aplicationHandler = new ApplicationHandler();
        this.aplicationHandler.HandleHelp(this);
    }
```

```
        private void Dialog_MouseHover(object sender, EventArgs e)
        {
            this.dialogHandler = new DialogHandler();
            this.aplicationHandler = new ApplicationHandler();
            this.dialogHandler.Successor = this.aplicationHandler;
            this.dialogHandler.HandleHelp(sender as Dialog);
        }

        private void Button_MouseHover(object sender, EventArgs e)
        {
            this.buttonHandler = new ButtonHandler();
            this.dialogHandler = new DialogHandler();
            this.aplicationHandler = new ApplicationHandler();
            this.buttonHandler.Successor = this.dialogHandler;
            this.dialogHandler.Successor = this.aplicationHandler;
            this.buttonHandler.HandleHelp(sender as Button);
        }
    }
```

Интерфейс IHelpHandler задает контракт для классов-обработчиков ApplicationHandler, DialogHandler, ButtonHandler.

```
    interface IHelpHandler
    {
        void HandleHelp(Control ctrl);
        IHelpHandler Successor { get; set; }\
    }
    class ApplicationHandler:IHelpHandler
    {
        ToolTip tt = new ToolTip();
        public void HandleHelp(Control ctrl)
        {
                tt.Show("This is Application", ctrl);
        }
        public IHelpHandler Successor { get; set; }
    }

    class DialogHandler : IHelpHandler
    {
        ToolTip tt = new ToolTip();
        public void HandleHelp(Control ctrl)
        {
            if (ctrl is Dialog && ctrl.Name == "saveDialog")
            {
                tt.Show("This is Save Dialog", ctrl);
            }
            else if (Successor != null)
                Successor.HandleHelp(ctrl);
        }

        public IHelpHandler Successor { get; set; }
    }

    class ButtonHandler : IHelpHandler
    {
        ToolTip tt = new ToolTip();
        public void HandleHelp(Control ctrl)
        {
            if (ctrl is Button && ctrl.Name == "saveButton")
            {
                tt.Show("This is Save Button", ctrl);
            }
            else if (Successor != null)
```

```
                Successor.HandleHelp(ctrl);
        }

        public IHelpHandler Successor { get; set; }
    }
```

См. Пример к главе: \013_Chain of Responsibility\002_ContextsHelp

Паттерн Command

Название

Команда

Также известен как

Action (Действие), Transaction (Транзакция)

Классификация

По цели: поведенческий
По применимости: к объектам

Частота использования

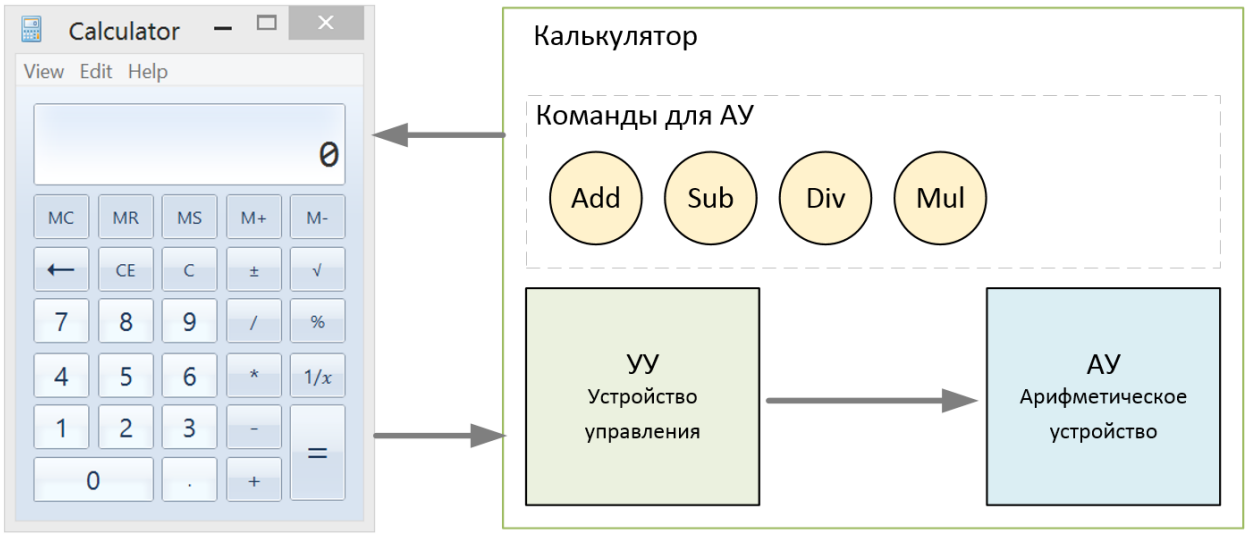
Выше средней - 1 2 3 **4** 5

Назначение

Паттерн Command – позволяет представить запрос в виде объекта, позволяя клиенту конфигурировать запрос (задавая параметры для его обработки), ставить запросы в очередь, протоколировать запросы, а также поддерживать отмену операций.

Введение

В качестве примера системы, в которой возможно использовать паттерн Command предлагается рассмотреть упрощенный программный калькулятор с четырьмя арифметическими операциями `Add`, `Sub`, `Mul`, `Div` и двумя операциями Undo и Redo. Структурная схема калькулятора показана на рисунке ниже.



В структуре калькулятора можно выделить UI (наборное поле с кнопками для цифр, знаков отмены и повтора операций, а также панель для отображения результата). В «корпусе» калькулятора расположены программные блоки, обеспечивающие реакцию калькулятора на поток внешних дискретных событий. Такими событиями являются нажатия кнопок наборного поля.

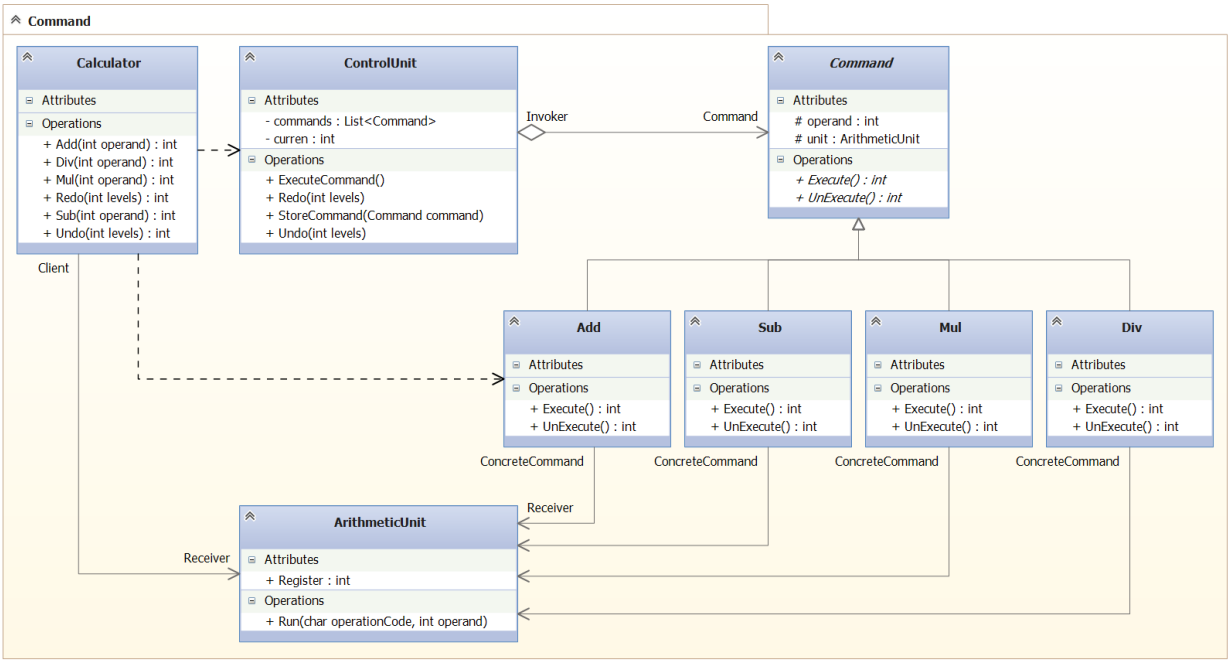
Состав программных блоков калькулятора, следующий:

- УУ - *Устройство управления* (`ControlUnit`). Оно организует всю работу калькулятора, выдавая в подходящий момент элементарные объекты-команды типа: Add, Sub, Mul, Div, Undo, Redo. При этом УУ сохраняет историю использования команд, а также может отменять и восстанавливать ранее выполненные команды.

- АУ - *Арифметическое устройство* (`ArithmeticUnit`). После получения «сигнала» (одной из четырех команд Add, Sub, Mul, Div) на вход выполняет арифметическую операцию.

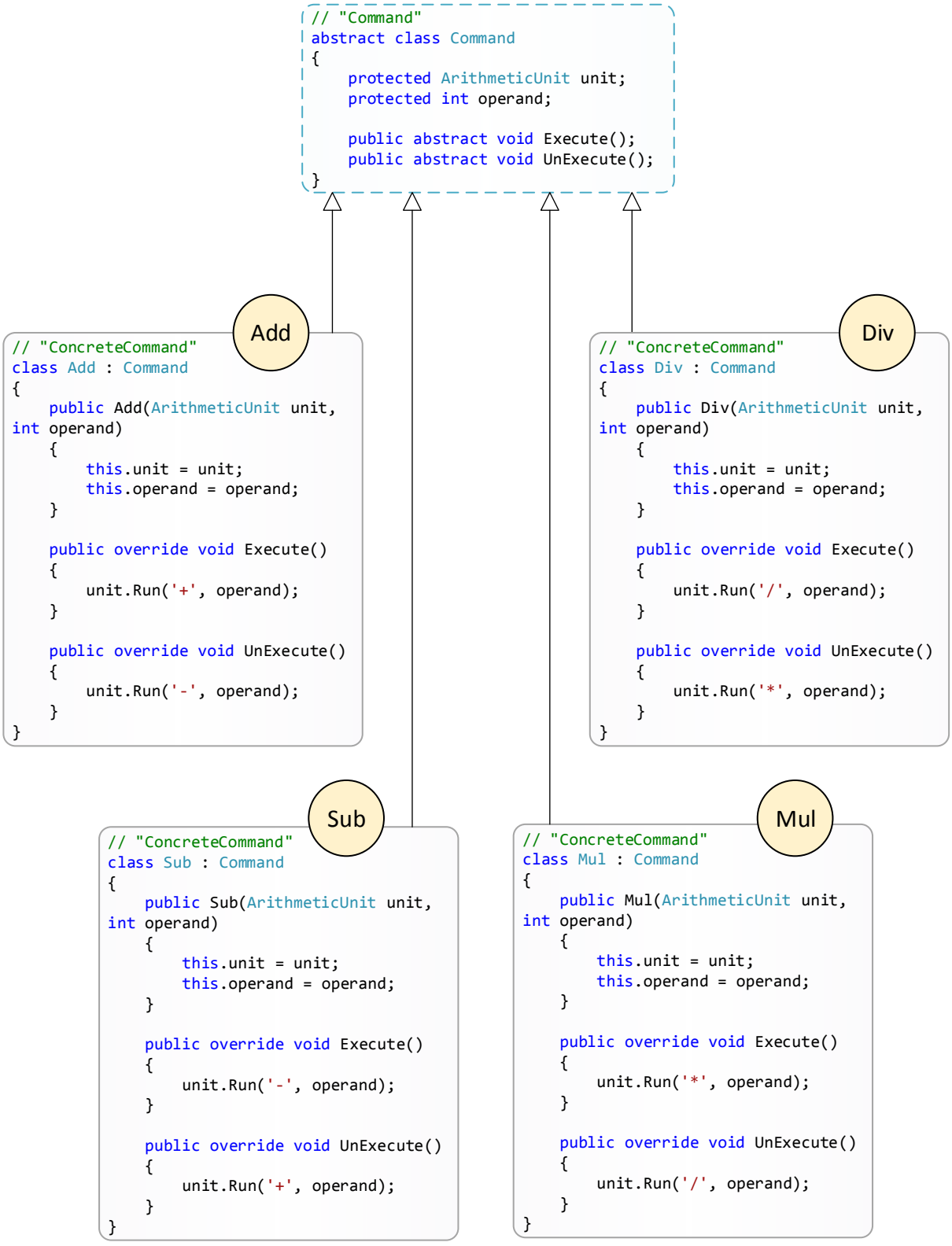
- *Команды* - `Add`, `Sub`, `Mul`, `Div`. Специальные объекты-команды, которые УУ использует для управления АУ. Каждый объект-команда связан с АУ и умеет правильно им управлять.

  Предлагается рассмотреть диаграмму классов на которой представлена модель калькулятора.



См. Пример к главе: \014_Command\002_Command_Undo_Redo

Ниже представлен пример реализации калькулятора.



```
// "Command"
abstract class Command
{
    protected ArithmeticUnit unit;
    protected int operand;

    public abstract void Execute();
    public abstract void UnExecute();
}
```

Add

```
// "ConcreteCommand"
class Add : Command
{
    public Add(ArithmeticUnit unit,
int operand)
    {
        this.unit = unit;
        this.operand = operand;
    }

    public override void Execute()
    {
        unit.Run('+', operand);
    }

    public override void UnExecute()
    {
        unit.Run('-', operand);
    }
}
```

Div

```
// "ConcreteCommand"
class Div : Command
{
    public Div(ArithmeticUnit unit,
int operand)
    {
        this.unit = unit;
        this.operand = operand;
    }

    public override void Execute()
    {
        unit.Run('/', operand);
    }

    public override void UnExecute()
    {
        unit.Run('*', operand);
    }
}
```

Sub

```
// "ConcreteCommand"
class Sub : Command
{
    public Sub(ArithmeticUnit unit,
int operand)
    {
        this.unit = unit;
        this.operand = operand;
    }

    public override void Execute()
    {
        unit.Run('-', operand);
    }

    public override void UnExecute()
    {
        unit.Run('+', operand);
    }
}
```

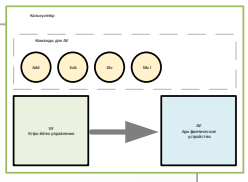
Mul

```
// "ConcreteCommand"
class Mul : Command
{
    public Mul(ArithmeticUnit unit,
int operand)
    {
        this.unit = unit;
        this.operand = operand;
    }

    public override void Execute()
    {
        unit.Run('*', operand);
    }

    public override void UnExecute()
    {
        unit.Run('/', operand);
    }
}
```

См. продолжение примера на следующей странице.

```
class Program
{
    static void Main()
    {
        var calculator = new Calculator();
        int result = 0;

        result = calculator.Add(5);
        Console.WriteLine(result);

        result = calculator.Sub(3);
        Console.WriteLine(result);

        result = calculator.Undo(2);
        Console.WriteLine(result);

        result = calculator.Redo(1);
        Console.WriteLine(result);
    }
}
```

```
// "Client"
class Calculator
{
    ArithmeticUnit arithmeticUnit;
    ControlUnit controlUnit;

    public Calculator()
    {
        arithmeticUnit = new ArithmeticUnit();
        controlUnit = new ControlUnit();
    }

    private int Run(Command command)
    {
        controlUnit.StoreCommand(command);
        controlUnit.ExecuteCommand();
        return arithmeticUnit.Register;
    }

    public int Add(int operand)
    {
        return Run(new Add(arithmeticUnit, operand));
    }

    public int Sub(int operand)
    {
        return Run(new Sub(arithmeticUnit, operand));
    }

    public int Mul(int operand)
    {
        return Run(new Mul(arithmeticUnit, operand));
    }

    public int Div(int operand)
    {
        return Run(new Div(arithmeticUnit, operand));
    }

    public int Undo(int levels)
    {
        controlUnit.Undo(levels);
        return arithmeticUnit.Register;
    }

    public int Redo(int levels)
    {
        controlUnit.Redo(levels);
        return arithmeticUnit.Register;
    }
}
```



УУ
Устройство управления

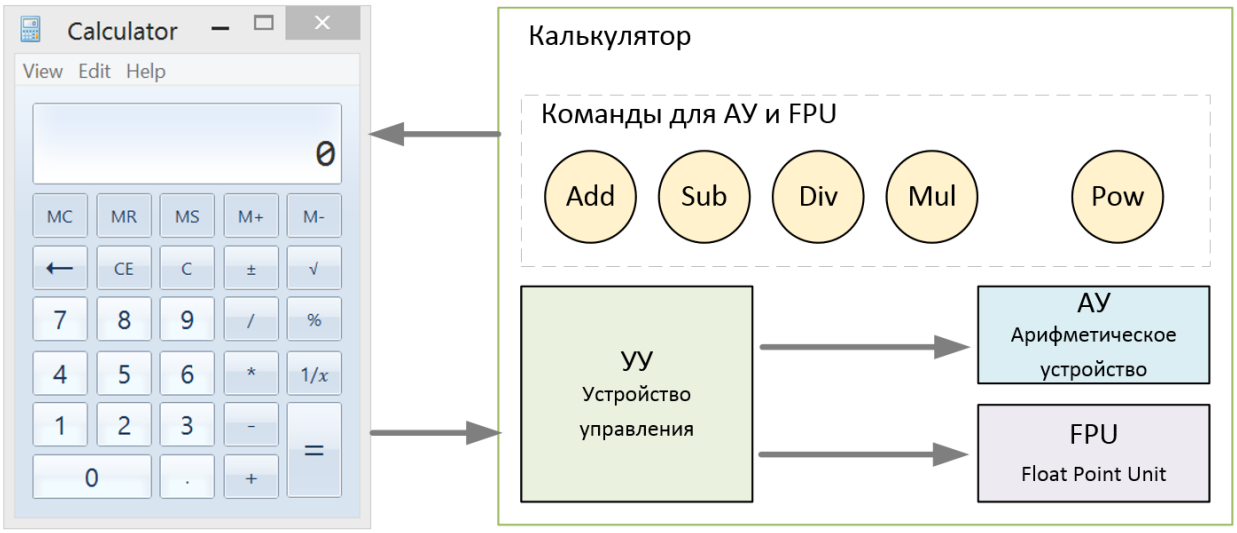
```
// "Invoker" (Инициатор)
// Устройство управления (УУ)
class ControlUnit
{
    private List<Command> commands =
                          new List<Command>();
    private int current = 0;

    public void StoreCommand(Command command)
    {
        commands.Add(command);
    }

    public void ExecuteCommand()
    {
        commands[current].Execute();
        current++;
    }

    public void Undo(int levels)
    {
        for (int i = 0; i < levels; i++)
            if (current > 0)
                commands[--current].UnExecute();
    }

    public void Redo(int levels)
    {
       for (int i = 0; i < levels; i++)
            if (current < commands.Count - 1)
                commands[current++].Execute();
    }
}
```

АУ
Арифметическое устройство

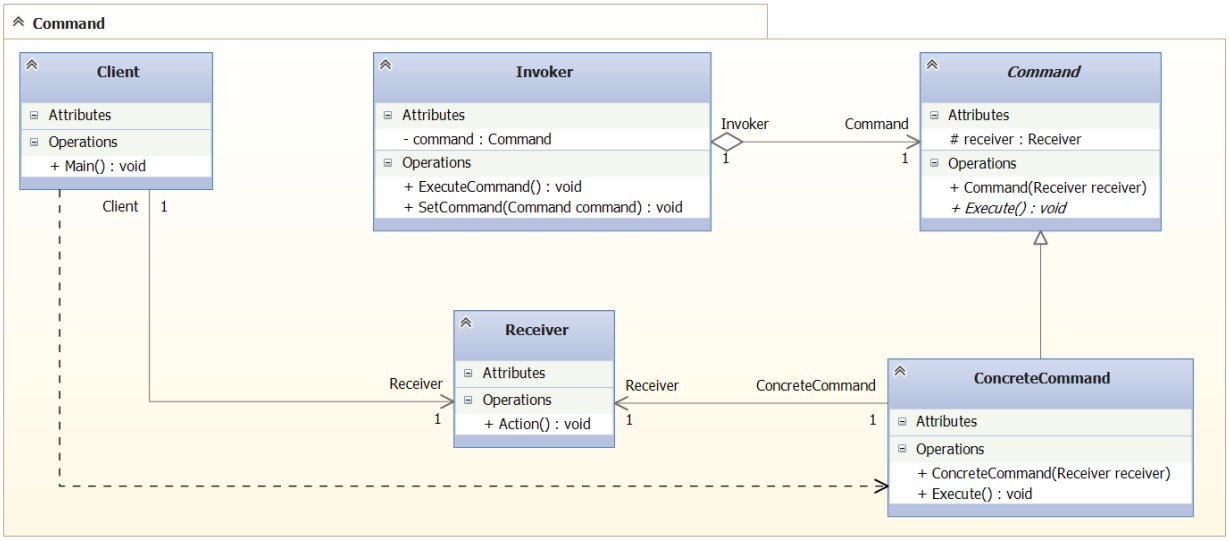
```
// "Receiver" (Получатель)
// Арифметическое устройство (АУ)
class ArithmeticUnit
{
    public int Register { get; private set; }

    public void Run(char operationCode, int operand)
    {
        switch (operationCode)
        {
            case '+':
                Register += operand;
                break;
            case '-':
                Register -= operand;
                break;
            case '*':
                Register *= operand;
                break;
            case '/':
                Register /= operand;
                break;
        }
    }
}
```

См. Пример к главе: \014_Command\002_Command_Undo_Redo

Удобство такого построения калькулятора с использованием паттерна Command заключается в том, что оказывается легко добавлять новые команды, которые можно тонко конфигурировать. Также можно расширять вычислительные возможности самого калькулятора, например, логическим устройством или устройством для вычислений чисел с плавающей точкой (FPU). На рисунке ниже показана структурная схема калькулятора расширенного новой командой Pow и новым вычислительным устройством FPU.

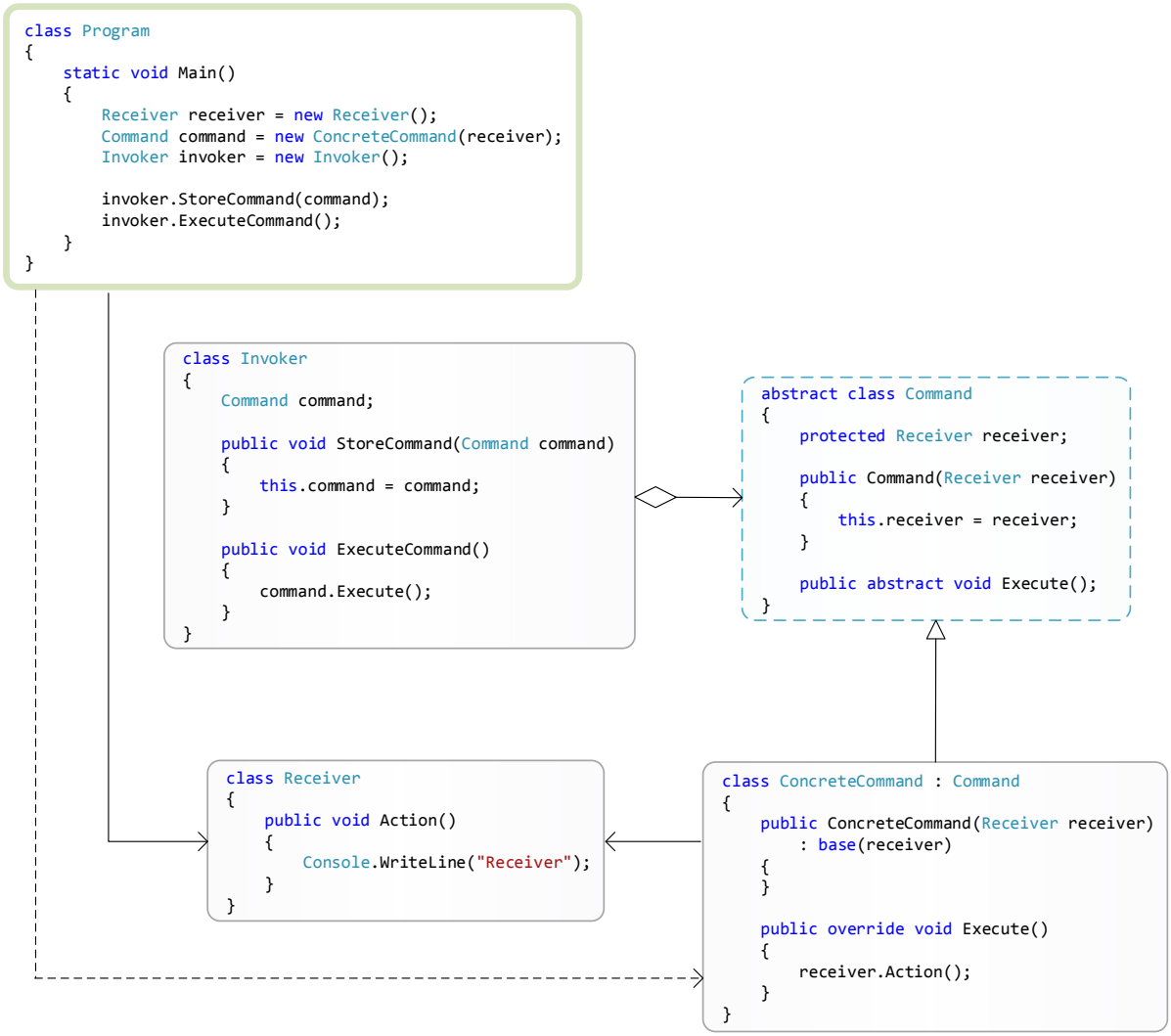


Структура паттерна на языке UML



См. Пример к главе: \014_Command\001_Command

Структура паттерна на языке C#



См. Пример к главе: \014_Command\001_Command

Участники

- `Command` - **Команда**:
  Предоставляет интерфейс для выполнения операции.

- `ConcreteCommand` - **Конкретная команда**:
  Представляет собой объектно-ориентированное выражение соответствия между объектом класса `Receiver` и выполняемым им действием `Action`. Реализует операцию `Execute` которая вызывает соответствующие операции объекта класса `Receiver`.

- `Client` - **Клиент**:
  Создает объект класса `ConcreteCommand` и устанавливает его получателя - объект класса `Receiver`.

- `Invoker` - **Инициатор**:
  Обращается к объекту-команде для выполнения запроса.

- `Receiver` - **Получатель**:
  Обладает функциональностью достаточной для выполнения определенного запроса. В роли получателя запроса может выступать любой класс.

Отношения между участниками

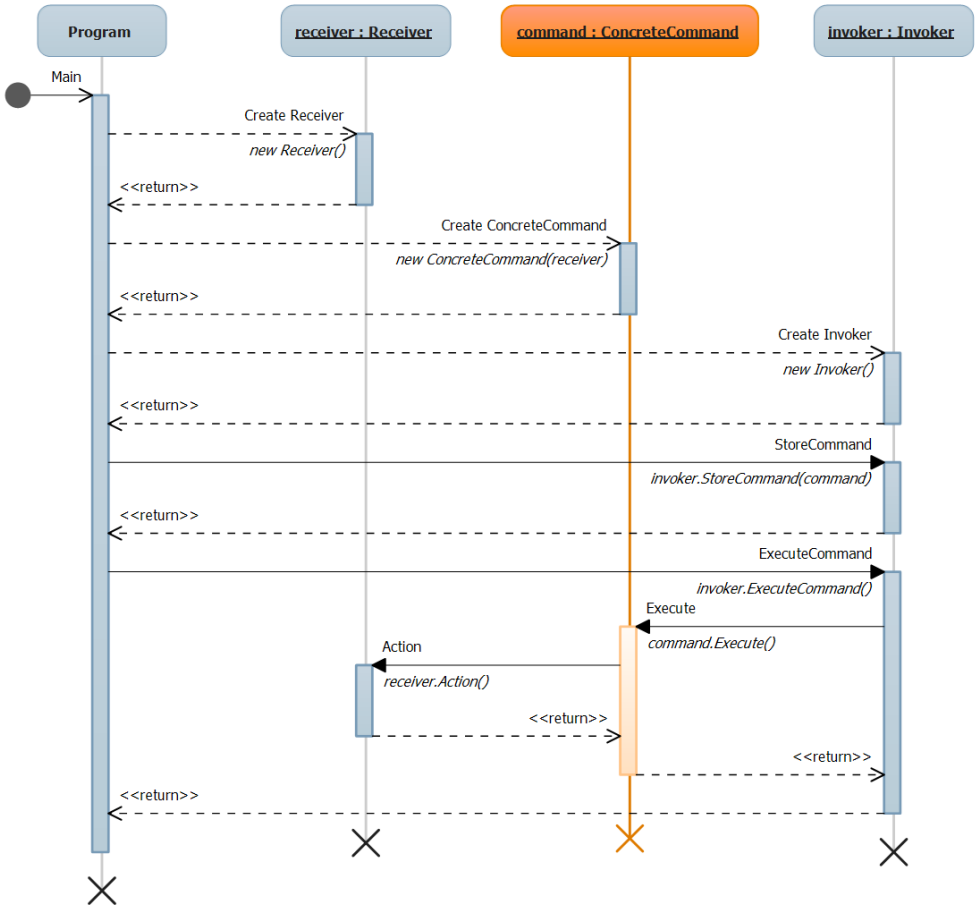
Отношения между классами

- Класс `Invoker` связан связью отношения агрегации с абстрактным классом `Command`.
- Класс `ConcreteCommand` связан связью отношения наследования с абстрактным классом `Command` и связью отношения ассоциации с конкретным классом `Receiver`.

Отношения между объектами

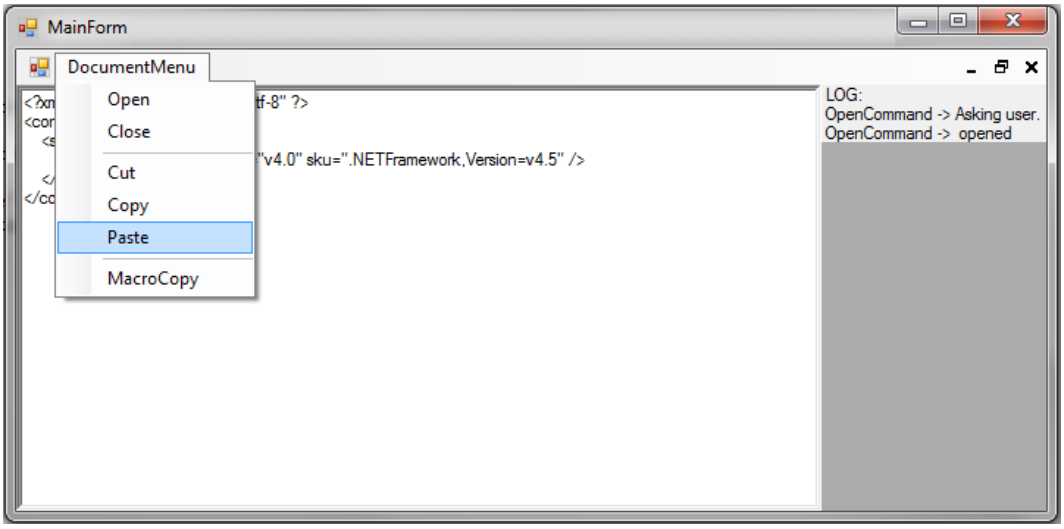
- Клиент создает объект класса `ConcreteCommand` и устанавливает для него получателя (`Receiver`).
- Объект класса `Invoker` сохраняет в списке историй команд объект класса `ConcreteCommand` и отправляет запрос на выполнение команды вызывая метод `Execute` на объекте-команде. Если требуется поддержка отмены (Undo) и повтора (Redo) операций, то объект `ConcreteCommand` перед выполнением тела метода `Execute` должен сохранить информацию о своем текущем состоянии, достаточную для выполнения отката (Undo).
- Объект класса `ConcreteCommand` вызывает операции (`Action`) объекта-получателя (`Receiver`) для выполнения запроса.

На диаграмме взаимодействий между объектами представленной ниже видно, как объект-команда (`Command`) разрывает связь-отношение между инициатором (`Invoker`) и получателем (`Receiver`).



Мотивация

Рассмотрим пример с использованием паттерна Command в приложении, с пользовательским интерфейсом представленном в виде меню. Заметим, что библиотека пользовательских элементов .Net не содержит информации о способах обработки информации при выборе определенных пунктов меню. Варианты и алгоритмы обработки задаются разработчиками приложений, использующих библиотеки .Net. Кроме того, часто необходимо организовать обработку действий пользователя в условиях, когда изначально неизвестно, какой объект является получателем запроса на обработку, и неясно, выполнение какой конкретной задачи запрошено. Например, такое может случиться при разработке в больших командах, где разные группы разработчиков отвечают за логику работы приложения и за интерфейс или когда на этапе проектирования, проектировщик библиотеки интерфейсов не владеет информацией о том, какие классы будут задействованы в приложении, и какие операции они будут выполнять. В таком случае, нужно каким-то образом инкапсулировать запрос и его параметры в определенном объекте и отделить объект интерфейса, инициирующий запрос, от объекта-исполнителя запроса.



Паттерн Command позволяет преобразовать запрос в объект и таким образом сделать возможной его хранение и отправку другим объектам приложения, не раскрывая при этом сути запроса и конкретного адресата. Основой паттерна является абстрактный класс `Command`, который задает интерфейс для выполнения операций, например, в виде абстрактного метода `Execute`, а также имеет защищенный метод логирования выполненных операций `LogExecution`:

```
abstract class Command
{
    public abstract void Execute();

    protected void LogExecution(string text)
    {
        MainForm.MainFormInstance.Log(this.GetType().Name + " -> " + text);
    }
}
```

Конкретные классы, отвечающие интерфейсу заданному классом Command должны хранить в себе получателя (адрес получателя), и реализовывать абстрактный метод `Execute` базового абстрактного класса. Таким образом, эти конкретные классы-команды формируют связки «получатель-действие», подразумевая, что у получателя есть вся необходимая информация для выполнения действия, на которое получен запрос.

С помощью объектов унаследованных от абстрактного класса Command легко реализовать меню, адаптируя абстрактный класс Command к имеющемуся в библиотеке .Net классу `ToolStripMenuItem`:

```
class MyMenuItem : ToolStripMenuItem
{
   public Command MenuCommand { get; set; }
```

```
    public MyMenuItem(string text, Command command)
        : base(text)
    {
        MenuCommand = command;
    }

    protected override void OnClick(EventArgs e)
    {
        base.OnClick(e);
        if (MenuCommand != null)
            MenuCommand.Execute();
    }
}
```

Меню приложения будет состоять из объектов типа MyMenuItem, каждый из которых будет сконфигурирован экземпляром одного из конкретных подклассов класса Command: OpenCommand, CopyCommand, CutCommand, SelectAllTextCommand, PasteCommand, CloseCommand, - или набором из таких объектов типа MacroCommand, которые содержат ссылку на получателя запроса. При выборе некоторого пункта меню связанный с ним объект класса MyMenuItem вызовет метод Execute, реализованный в классе инкапсулированного объекта-команды, который и выполнить необходимую операцию. Заметим, что объекты класса MyMenuItem, работают с интерфейсом заданным абстрактным классом Command и не содержат информацию о том, какую именно операцию они выполняют и каким именно объектом подкласса класса Command они оперируют. Классы конкретных команд содержат информацию о получателе запросов – объекте типа Document, доступному через статическое свойство CurrentDocument класса MainForm, и вызывают одну или целый набор операций этого получателя.

Таким образом, паттерн Command отделяет объект инициирующий операцию от объекта, который знает, как ее выполнить, что придает гибкости проектному решению при проектировании пользовательского интерфейса. Используя паттерн Command легко организовать динамическую подмену команд при реализации контекстно-зависимых меню и организацию сценариев (сложных макрокоманд) из более простых команд.

Применимость паттерна

Паттерн Command рекомендуется использовать, когда:

- **необходимо параметризировать объекты-команды выполняемым действием,** например, кнопки или элементы меню, а также callback методы, благодаря объектно-ориентированному представлению объектов-запросов. Заметим, что такая параметризация не является параметризацией типов в чистом виде, в языке C# - она представлена обобщениями, поддерживаемыми платформой .Net, хотя в этом случае обобщения (generics), также могут быть использованы, например, при параметризации сложными объектами классов Action<T> или Func<T, TResult >, где указатель на место заполнения типа T будет примать тип Command.

- **необходимо организовать разнесенное во времени добавление запросов в очередь и их выполнение.** При этом срок жизни запроса в очереди и объекта-запроса класса, наследуемого от класса Command, могут быть независимыми и вообще существовать в различных процессах.

- **необходимо обеспечить отмену операций.** Для этого нужно в интерфейсе абстрактного класса Command объявить метод Unxecute, вызов которого позволял бы осуществить откат действий, произведенных методом Execute. Возможно, для реализации такого действия, понадобиться предварительно сохранить необходимую информацию при выполнении метода Execute и сохранять последовательность действий Execute/Unxecute в списке истории внутри объекта класса, чтобы при необходимости можно было выполнить откат состояния объекта-получателя, выполняя обратные операции. Чтобы предоставить команде доступ к этой информации, не раскрывая внутреннее состояние объектов-носителей информации, можно воспользоваться паттерном Memento.

- **необходимо протоколировать (логировать) изменения**. Такая возможность может понадобиться для отслеживания времени и инициатора изменений данных, а также для восстановления состояния объектов после сбоя. В таком случае, необходимо дополнить интерфейс абстрактного

класса `Command` методами сохранения и загрузки протокола изменений из внешнего источника. Тогда после сбоя можно будет загрузить протокол изменений и повторно выполнить последовательность операций при помощи методов `Execute/Unxecute`.

- **необходимо создать систему на основе транзакций** т.е. с прозрачной структурой высокоуровневых операций на основе примитивных. *Транзакция* – объектно-ориентированное представление группы логически объединённых последовательных операций по работе с данными, обрабатываемое или отменяемое целиком. Паттерн Command позволяет проектировать системы с учетом транзакционного подхода благодаря наличию у всех низкоуровневых команд и высокоуровневых транзакций общего интерфейса, что позволяет легко добавлять в систему новые команды и организовывать на их основе новые транзакции, а также работать с ними единообразно.

Результаты

Паттерн Command:

- **отделяет объект инициатор операции от объекта, имеющего информацию о том, как ее выполнить.** Достигается благодаря адаптации (см. паттерн Adapter) абстрактного класса или интерфейса `Command` к классу объекта инициатора. При этом сам объект-инициатор может абсолютно не владеть информацией об объекте-исполнителе и о сути выполняемой операции.

- **позволяет оперировать объектами при маршрутизации и обработке запросов.** Конкретные классы команд инкапсулируют в себе адрес обработчика операции и высокоуровневую часть алгоритма обработки.

- **предоставляет возможность структурирования команд,** путем компоновки сложных операций, например `MacroCommand`, из более простых составляющих. При этом объект сложной операции имеет тот же интерфейс, что и все его составные части, т.е. реализует метод `Execute` базового абстрактного класса `Command` (см. паттерн Composite).

- **дает возможность легко добавлять новые типы команд,** поскольку никакие существующие классы, при этом изменять не нужно.

Реализация

Полезные приемы реализации паттерна Command:

- **широкий спектр возможностей организации команд по уровню сложности.** С одной стороны команды могут универсальными, «*умными*» и выполнять широкий круг обязанностей, совмещая в себе большую информированность об адресате и набор возможностей для доступа к его методам, т.е. уметь динамически находить получателя, а также определять и вызывать его методы, для выполнения нужных операций (например, с помощью *механизмов рефлексии*, поддерживаемых платформой .Net). С другой стороны, команды могут принимать и вырожденные формы, т.е. полностью или частично не владеть информацией об адресате (например, если он точно неизвестен или подходящего получателя не существует) и выполнять необходимые операции самостоятельно, без привлечения получателей, используя *автономные методы*. При этом команды, могут даже не владеть информацией о том, что в точности они выполняют, например, создавать объекты, не различая их по типам (кнопки это, меню, окна или объекты пользовательских классов). Между этими двумя крайними вариантами находятся команды, обладающие достаточной информацией для передачи запроса получателю, и использовать заранее прописанные в коде методы объектов-получателей для выполнения необходимых операций.

- **поддержка многоуровневой отмены и повтора операций.** Для реализации такой возможности понадобится реализовать хранение истории изменений с дополнительной информацией об:

    - адресе объекта-получателя класса `Receiver`, который выполняет операции по запросу;

- аргументы и другие параметры используемые при вызове методов класса объекта-получателя;
- исходном состоянии объекта-получателя `Receiver`, т.е. значения его полей, которые могли измениться в результате выполнения команды.

Также необходимым условием является предоставление доступа объектом `Receiver` к методам, позволяющим команде вернуть этот объект в исходное состояние. Обратим внимание, что если в результате выполнения команды меняется состояние не только объекта-получателя, но и самой команды, то объект команды также необходимо хранить в истории изменений, копируя его после выполнения, так как он может быть изменен при следующем вызове, и тогда выполнить откат действий корректно не удастся. Если в результате выполнения команды ее состояние не изменяется (не изменяются атрибуты объекта класса `ConcreteCommand`), то и хранить копию объекта не обязательно, достаточно хранения ссылки на объект-команду. Команды, которые изменяются при выполнении и нуждаются в хранении копий-клонов в списке истории ведут себя подобно прототипам (см. паттерн Prototype).

- **гарантирование целостности и неизменности объектов в процессе хранения истории изменений.** Чтобы гарантировать, что объекты будут полностью восстановлены в процессе отката или повтора команд и буду защищены от несанкционированного изменения можно воспользоваться паттерном Memento. Это позволит команде получить доступ объекта-команды к информации необходимой для произведения операций отмены/повтора, без изменения состояния хранимого объекта, и без раскрытия его внутренней структуры.

- **применение шаблонов.** Для однотипных команд, которые не поддерживают операции отмены/повтора (Undo/Redo) можно воспользоваться подходом создания команды-шаблона – объекта типа `SimpleCommand : Command` на основе делегатов типов `Action`, `Action<T>`, `Func<T, TResult>`, или `delegate`, где метод `Execute`, будет вызывать необходимые callback методы. Такой подход позволяет сократить количество подклассов класса `Command`. Подробнее использование шаблонных команд рассматривается в примере ниже.

Пример кода

Рассмотрим детальнее конкретные классы команд из раздела Мотивация. Класс `OpenCommand`, реализует команду открытия документа выбранного пользователем в приложении, используя метод `AddDocument` класса `MainForm` и реализуя абстрактный метод `Execute` родительского абстрактного класса `Command`:

```
class OpenCommand : Command
{
    public override void Execute()
    {
        var filename = AskUser();
        if (!string.IsNullOrEmpty(filename))
        {
            var doc = new Document();
            doc.Open(filename);
            LogExecution(" opened");
            MainForm.MainFormInstance.AddDocument(doc);
        }
        else
        { LogExecution(" opening cancelled"); }
    }
    string AskUser()
    {
        LogExecution("Asking user.");
        var dlg = new OpenFileDialog();
        dlg.InitialDirectory = Application.StartupPath;
        if (dlg.ShowDialog() == DialogResult.OK)
        {
            return dlg.FileName;
        }
        else
            return string.Empty;
    }
}

public partial class MainForm : Form
{
    internal static Document CurrentDocument { get; set; }
    internal static MainForm MainFormInstance { get; private set; }

    public MainForm()
    {
        InitializeComponent();
        toolStripMenuItem1.DropDownItems.AddRange(new ToolStripItem[] {
            new MyMenuItem("Open",new OpenCommand()),
            new MyMenuItem("Close",new CloseCommand()),
            new ToolStripSeparator(),
            new MyMenuItem("Cut",new CutCommand()),
            new MyMenuItem("Copy",new CopyCommand()),
            new MyMenuItem("Paste",new PasteCommand()),
            new ToolStripSeparator(),
            new MyMenuItem("MacroCopy",new MacroCommand(new OpenCommand(),
                                                        new SelectAllTextCommand(),
                                                        new CopyCommand(),
                                                        new CloseCommand()))
    });
        MainFormInstance = this;
    }


    private void AddMenuItemWithTemplateCommands()
    {
```

```
            toolStripMenuItem2 = new System.Windows.Forms.ToolStripMenuItem();

            this.menuStrip1.Items.AddRange(new System.Windows.Forms.ToolStripItem[] {
            toolStripMenuItem2});

            toolStripMenuItem2.Name = "toolStripMenuItem2";
            toolStripMenuItem2.Size = new System.Drawing.Size(106, 20);
            toolStripMenuItem2.Text = "DocumentTemplate Menu";

            toolStripMenuItem2.DropDownItems.AddRange(new ToolStripItem[] {
                new MyMenuItem("Cut",new SimpleCommand(CurrentDocument.Cut,"cut")),
                new MyMenuItem("Copy",new SimpleCommand(CurrentDocument.Copy, "copy")),
                new MyMenuItem("Paste",new SimpleCommand(CurrentDocument.Paste,"paste")),
            });
        }

        public void AddDocument(Document document)
        {
            document.MdiParent = this;
            document.Show();

            AddMenuItemWithTemplateCommands();
        }

        public void Log(string logString)
        {
            logLabel.Text += Environment.NewLine + logString;
        }
    }
```

Классы CloseCommand, CopyCommand, CutCommand, PasteCommand, SelectAllTextCommand и MacroCommand аналогично реализуют соответственно закрытие документа, копирование и вырезание выделенного фрагмента текста, вставку текста из буфера обмена, выделение всего текстового содержимого в документе и заданную в коде приложения последовательность команд.

```
    class CloseCommand : Command
    {
        public override void Execute()
        {
            if (MainForm.CurrentDocument != null)
            {
                LogExecution("close");
                MainForm.CurrentDocument.Close();
            }
        }
    }

    class CopyCommand : Command
    {
        public override void Execute()
        {
            if (MainForm.CurrentDocument != null)
            {
              LogExecution("copy text: " +
              MainForm.CurrentDocument.DocumentContent.SelectedText);
              MainForm.CurrentDocument.Copy();
            }
        }
    }

    class CutCommand : Command
    {
        public override void Execute()
        {
```

```
            if (MainForm.CurrentDocument != null)
            {
              LogExecution("cut text: " +
              MainForm.CurrentDocument.DocumentContent.SelectedText);
              MainForm.CurrentDocument.Cut();
            }
        }
    }

    class PasteCommand : Command
    {

        public override void Execute()
        {

            if (MainForm.CurrentDocument != null)
            {
                LogExecution("paste text: " + Clipboard.GetText());
                MainForm.CurrentDocument.Paste();
            }
        }
    }

    class SelectAllTextCommand : Command
    {
        public override void Execute()
        {
            if (MainForm.CurrentDocument != null)
            {
                LogExecution(MainForm.CurrentDocument.Text + "select all text");
                MainForm.CurrentDocument.DocumentContent.SelectAll();
            }
        }
    }

    class MacroCommand : Command
    {
        public readonly List<Command> Commands = new List<Command>();

        public MacroCommand(params Command[] commands)
        {
            Commands.AddRange(commands);
        }

        public override void Execute()
        {
            foreach (var c in Commands)
                c.Execute();
        }
    }
```

Заметим, что класс `MacroCommand` не содержит ссылок на объекты-исполнители запросов, т.к. они инкапсулированы в объектах типа `Command`, набор которых хранит в себе данный класс.

```
    class SimpleCommand : Command
    {
        Action action;
        string actionKeyword;

        public SimpleCommand(Action action, string actionKeyword)
```

```
        {
            this.action = action;
            this.actionKeyword = actionKeyword;
        }

        public override void Execute()
        {
            if (action != null)
            {
              LogExecution(actionKeyword + " text: " +
              MainForm.CurrentDocument.DocumentContent.SelectedText);
             action.Invoke();
            }
        }
    }
```

В классе `SimpleCommand` каждая связка «объект-исполнитель – действие» задается делегатом типа `Action` при вызове конструктора класса, а объекты класса `SimpleCommand` оперируют только ссылкой на необходимый callback метод, который передан с делегатом класса `Action`.

Известные применения паттерна в .Net

`System.Data.Odbc.OdbcCommand`
http://msdn.microsoft.com/ru-ru/library/system.data.odbc.odbccommand(v=vs.110).aspx

`System.Data.OleDb.OleDbCommand`
http://msdn.microsoft.com/ru-ru/library/system.data.oledb.oledbcommand(v=vs.110).aspx

`System.Data.OracleClient.OracleCommand`
http://msdn.microsoft.com/en-us/library/system.data.oracleclient.oraclecommand(v=vs.110).aspx

`System.Data.SqlClient.SqlCommand`
http://msdn.microsoft.com/ru-ru/library/system.data.sqlclient.sqlcommand(v=vs.100).aspx

`System.Windows.Input.ICommand`
http://msdn.microsoft.com/en-us/library/system.windows.input.icommand(v=vs.110).aspx

`System.Windows.Input.ComponentCommands`
http://msdn.microsoft.com/ru-ru/library/system.windows.input.componentcommands(v=vs.110).aspx

`System.Windows.Input.MediaCommands`
http://msdn.microsoft.com/ru-ru/library/system.windows.input.mediacommands(v=vs.110).aspx

`System.Windows.Input.NavigationCommands`
http://msdn.microsoft.com/ru-ru/library/system.windows.input.navigationcommands(v=vs.110).aspx

`System.Windows.Input.RoutedCommand`
http://msdn.microsoft.com/ru-ru/library/system.windows.input.routedcommand(v=vs.110).aspx

`System.Windows.SystemCommands`
http://msdn.microsoft.com/ru-ru/library/system.windows.systemcommands(v=vs.110).aspx

`System.Workflow.ComponentModel.Design.WorkflowMenuCommands`
http://msdn.microsoft.com/ru-ru/library/system.workflow.componentmodel.design.workflowmenucommands(v=vs.110).aspx

Паттерн Interpreter

Название

Интерпретатор

Также известен как

-

Классификация

По цели: поведенческий
По применимости: к классам

Частота использования

Низкая - **1** 2 3 4 5

Назначение

Паттерн Interpreter – позволяет сформировать объектно-ориентированное представление грамматики для заданного языка, а также описывает правила создания механизма интерпретации (*толкования*) предложений этого языка.

Введение

Паттерн Interpreter описывает способы и правила построения объектно-ориентированного представления грамматики для заданного формального языка. Формальные языки классифицируются в соответствии с типами грамматик, которыми они задаются. Грамматики классифицируются по иерархии Хомского, согласно которой они разделяются на четыре категории. Иерархия Хомского классифицирует грамматические формальные системы по их порождающей способности – то есть по тому множеству языков, которые они позволяют представить.

Список типов грамматик по классификации Хомского:

- *Рекурсивно перечислимые грамматики.*
  Эти грамматики имеют неограниченные правила и по своей выразительной мощи эквивалентны машинам Тьюринга. Практического применения в силу своей сложности такие грамматики не имеют.

- *Контекстно-зависимые грамматики.*
  Эти грамматики могут использоваться при анализе текстов на естественных языках, но для построения компиляторов практически не используются в силу своей сложности.

- *Контекстно-свободные грамматики.*
  Эти грамматики широко применяются для описания синтаксиса компьютерных языков.

- *Регулярные грамматики.*
  Это самые простые из формальных грамматик. Регулярные грамматики применяются для описания простейших конструкций: идентификаторов, строк, констант, а также языков ассемблера (представления мнемоник), языков регулярных выражений, командных процессоров и др.

Первые две грамматики в иерархии Хомского имеют большую выразительную мощь, но алгоритмы для работы с такими грамматиками являются сложными и малоэффективными. Долгое время усилия лингвистов, в том числе и Аврама Ноама Хомского были в основном направлены на исследование

контекстно-свободных и контекстно-зависимых языков. Но в итоге стали чаще применяться регулярные грамматики в силу простоты обращения с ними. Все регулярные грамматики могут быть разделены на два эквивалентных класса: леволинейные и праволинейные регулярные грамматики.

Регулярные грамматики определяют в точности все регулярные языки, и поэтому эквивалентны конечным автоматам и регулярным выражениям (*Регулярный язык* - это формальный язык, который может быть выражен средствами регулярных выражений, а *Регулярные выражения* – это формальный язык поиска и осуществления манипуляций с подстроками в тексте).

Важной особенностью регулярных грамматик является то, что регулярные грамматики эквивалентны по своей вычислительной мощи конечным автоматам (КА).

Определим язык формально. Есть и другие определения, но мы остановимся на этом.

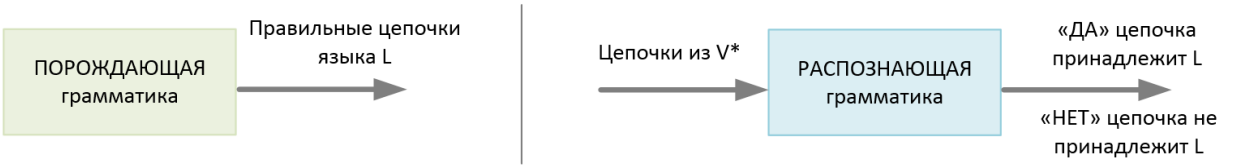
ОПРЕДЕЛЕНИЕ: Конечное множество элементов будем называть словарем, элементы словаря – символами, а последовательности символов словаря – цепочками или предложениями. Множество предложений назовем языком. Язык над словарем **V** будем обозначать $\mathbf{L}_V$, или просто **L**, если **V** очевидно.

Пусть **V** – словарь. Обозначим **V*** - множество всех возможных цепочек, составленных из символов словаря **V**. Если **V= {a, b, c}**, то **V*= {ε, a, b, c, aa, ab, ac, … aaa, aab, … abacbcaacbcacb..., …}**, где **ε** – пустая цепочка, не содержащая символов. Очевидно, что хотя **V** конечно, **V*** - бесконечное счетное множество. Всего языков над словарем **V** (как подмножеств, счетного множества **V***), бесконечное число: это множество мощности континуум.

Один из распространенных методов задания языка использует «*определитель множества*» вида: {**ω|утверждение о цепочке ω**}, где **ω** – это обычно выражение с параметрами, а **утверждение о цепочке ω –** определяет некоторые условия, налагаемые на параметры. Во многих случаях использовать этот метод трудно или невозможно. Например, как задать **L** - множество правильных скобочных выражений, как определить множество всех идентификаторов языка программирования, как задать сам язык программирования? Для упрощения решения большинства таких задач существует два типа грамматик: порождающие и распознающие. Эти два типа грамматик аналогичны двум формам речи человека: распознающая грамматика аналогична импрессивной речи, а порождающая грамматика аналогична экспрессивной речи. Импрессивная (*сенсорная*) речь — это восприятие и понимание речи (В коре головного мозга имеется зона сенсорной речи, которая называется зоной Вернике.). Экспрессивная (*моторная*) речь — произнесение звуков речи самим человеком (В области второй и третьей лобной извилины головного мозга имеется зона моторной речи, которая называется зоной Брока. У правшей зона Брока находится в левом полушарии мозга, а у левшей в большинстве случаев — в правом.).

Под порождающей грамматикой языка **L** понимается конечный набор правил, позволяющий строить все «*правильные*» предложения языка **L** и ни одного «*неправильного*».

Распознающая грамматика задает критерий принадлежности произвольной цепочки данному языку. Это алгоритм, принимающий в качестве входа символ за символом произвольную цепочку над словарем **V** и дающий на выходе один из двух возможных ответов: «данная цепочка принадлежит языку **L**» либо «данная цепочка **НЕ** принадлежит языку **L**». В действительности этот алгоритм должен разделить все возможные входные цепочки на два класса: один – принадлежащие языку **L**, а другой – не принадлежащие языку **L**.



Роль распознающей грамматики может выполнить конечный автомат (КА) без выхода. Если связать с некоторыми состояниями автомата метку «ДА», а с остальными метку «НЕТ», тогда все множество входных цепочек автомата разобьется на два класса: одни - приводящие автомат в одно из состояний, помеченных «ДА», а все другие в одно из состояний, помеченных «НЕТ».

Определим автомат-распознаватель формально.

ОПРЕДЕЛЕНИЕ: Конечным автоматом-распознавателем называется пятерка объектов:

- $\mathbf{A} = < \mathbf{S},\ \mathbf{X},\ \mathbf{s_0},\ \boldsymbol{\delta},\ \boldsymbol{F} >$**, где:**
- **S** - конечное непустое множество (состояний);
- **X** - конечное непустое множество входных сигналов (входной алфавит)
- $\mathbf{s_0} \in \boldsymbol{S}$ - начальное состояние
- $\boldsymbol{\delta}: \boldsymbol{S} \times \boldsymbol{X} \to \boldsymbol{S}$ - функция переходов
- $\boldsymbol{F} \subseteq \boldsymbol{S}$ - множество заключительных (финальных) состояний

ОПРЕДЕЛЕНИЕ: Конечный автомат-распознаватель $\mathbf{A} = < \mathbf{S},\ \mathbf{X},\ \mathbf{s_0},\ \boldsymbol{\delta},\ \mathbf{F} >$ допускает входную цепочку $\boldsymbol{\alpha} \in \mathbf{X}^*$, если $\boldsymbol{\alpha}$ переводит его из начального в одно из заключительных состояний, то есть если $\boldsymbol{\delta}^* : (\mathbf{s_0}, \boldsymbol{\alpha}) \in \mathbf{F}$. Множество всех цепочек, допускаемых автоматом **A**, образуют язык, допускаемый **A**.

Роль порождающей грамматики может выполнить конечный автомат-преобразователь. Конечный автомат-преобразователь (транслятор) - это просто распознающий автомат, дополненный семантическими действиями, выполняемыми при анализе очередного входного символа.

Реализация распознающей и порождающей грамматик не ограничивается использованием конечных автоматов. Паттерн Interpreter дает возможность реализации распознающих и порождающих грамматик без использования «конечно-автоматного» подхода. Объектная структура грамматики, составленная и реализованная по правилам паттерна Interpreter «*теоретически*» эквивалентна по своей вычислительной мощи конечным автоматам (КА). В большинстве случаев предпочтительно использовать КА, но для очень (!) простых грамматик есть смысл использовать подход описываемый паттерном Interpreter. Для сложных грамматик иерархия классов становится слишком громоздкой и практически неуправляемой.

Важно помнить, что не все языки представимы конечными автоматами, а если язык не автоматный, то такой язык практически невозможно реализовать при помощи паттерна Interpreter. Поэтому прежде чем преступить к реализации грамматики определенного языка, требуется проверить, является ли этот язык автоматным. «*Лемма о накачке*», является важным «теоретическим инструментом» позволяющим во многих случаях проверить является ли язык автоматным (*Лемма* – это готовое доказательство теоремы, использующееся для доказательства других теорем). Термин «*накачка*» в названии леммы оттеняет возможность многократного повторения некоторой подстроки в любой строке подходящей длины любого бесконечного автоматного языка.

Более формально лемма о накачке формулируется так:

Лемма о накачке (1). Пусть **L** – автоматный язык над алфавитом **V**. Тогда:

$(\exists \boldsymbol{n} \in \boldsymbol{N})(\forall \boldsymbol{\alpha} \in \mathbf{L}: |\boldsymbol{\alpha}| \geq \boldsymbol{n})(\exists \boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w} \in \boldsymbol{V}^*)$:

$[\boldsymbol{\alpha} = \boldsymbol{uvw} \& |uv| \leq n \& |v| \geq 1 \& (\forall i \in N)(\boldsymbol{uv^i w} \in \boldsymbol{L})]$

Другая форма леммы о накачке, которую иногда удобно применять чтобы показать «*неавтоматность*» некоторого языка, записывается так:

Лемма о накачке (2). Пусть **L** – автоматный язык над алфавитом **V**. Если:

$(\forall \boldsymbol{n} \in \boldsymbol{N})(\exists \boldsymbol{\alpha} \in \mathbf{L}: |\boldsymbol{\alpha}| \geq \boldsymbol{n})(\forall \boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w} \in \boldsymbol{V}^*)$:

$[\boldsymbol{\alpha} = \boldsymbol{uvw} \& |uv| \leq n \& |v| \geq 1 \& (\exists i \in N)(\boldsymbol{uv^i w} \notin \boldsymbol{L})]$

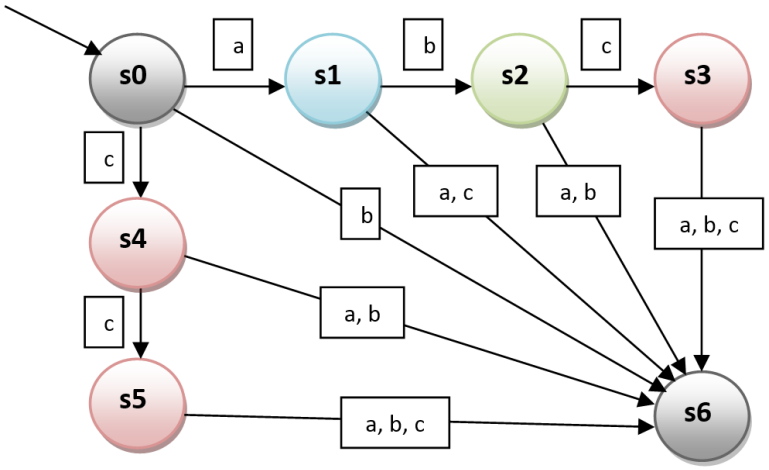
то **L** - не автоматный.

Предлагается рассмотреть пример доказательства теоремы, устанавливающей неавтоматность конкретного языка, с помощью леммы о накачке:

ТЕОРЕМА: Язык $\boldsymbol{L = \{\beta c \beta^R | w \in \{0,1\}^*\}}$ – неавтоматный (здесь $\boldsymbol{\beta^R}$ - это цепочка обратная к $\boldsymbol{\beta}$).

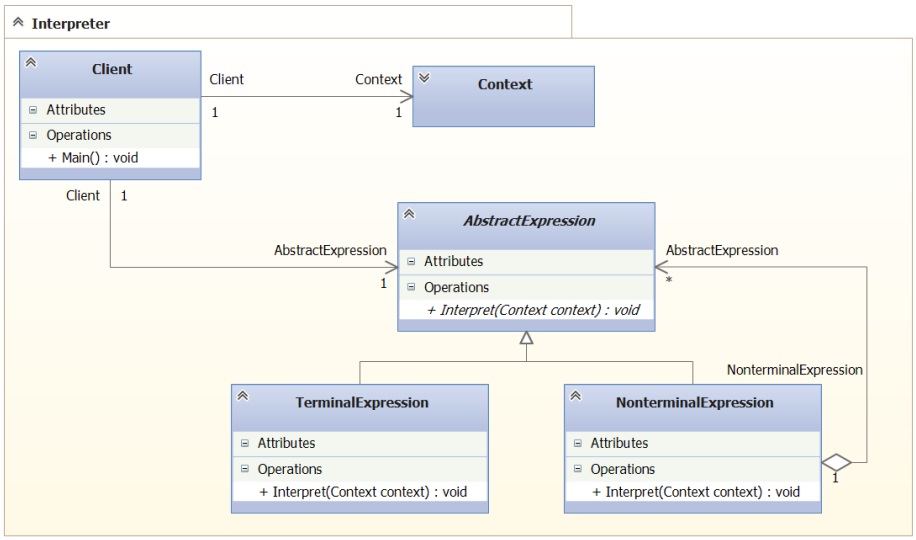
ДОКАЗАТЕЛЬСТВО: Предположим противное, а именно, что язык **L** - автоматный и он распознается конечным автоматом **A** с **n** состояниями. Рассмотрим цепочку $\boldsymbol{\alpha}$ этого языка, такую что $\boldsymbol{|\alpha| \geq n}$. Эта цепочка по предположению допускается автоматом **A**, и поскольку путь, помеченный этой цепочкой на графе переходов **A**, содержит по крайней мере один цикл, то цепочку $\boldsymbol{\alpha}$ можно представить, как конкатенацию трех подцепочек, $\boldsymbol{\alpha = uvw}$, причем цепочка $\boldsymbol{v}$ не пуста. Язык **L** будет автоматным, если **A** вместе с $\boldsymbol{\alpha = uvw}$ допускает также и $\boldsymbol{uvvw}$, и $\boldsymbol{uvvvw}$ и т.д. В какую из подцепочек $\boldsymbol{u}$, $\boldsymbol{v}$ или $\boldsymbol{w}$ может входить символ c? Если c входит в подцепочки $\boldsymbol{u}$ иди $\boldsymbol{w}$, то «*накачка*» $\boldsymbol{v}$ нарушает симметрию допускаемой цепочки, если c входит в $\boldsymbol{v}$, то «*накачка*» $\boldsymbol{v}$ приведет к удвоению, утроению и т.п. символа c, что также недопустимо. Таким образом, в допускаемой автоматом **A** достаточно длинной цепочке $\boldsymbol{\alpha}$ не существует непустой подцепочки, повторение которой произвольное число раз сохраняет структуру $\boldsymbol{\beta c \beta^R}$, то есть также дает цепочку, допускаемую этим автоматом. Следовательно, язык **L** - неавтоматный.

Зададим для примера простой автоматный язык $\boldsymbol{V = \{a, b, c\}}$; $\boldsymbol{L = \{abc, cc\}}$; и построим для него конечный автомат. Построим граф переходов автомата, распознающего **L**. Входные цепочки $\boldsymbol{abc}$ и $\boldsymbol{cc}$ (и только они) переводят автомат из начального, в одно из заключительных состояний **S5** или **S3**.



Предлагается самостоятельно реализовать грамматику для заданного языка $\boldsymbol{V = \{a, b, c\}}$; $\boldsymbol{L = \{abc, cc\}}$; с использованием подхода предоставляемого паттерном Interpreter.

Структура паттерна на языке UML



См. Пример к главе: \015_Interpreter\001_Interpreter

Структура паттерна на языке C#

```
class Program
{
    static void Main()
    {
        Context context = new Context
        {
            Vocabulary = 'a',
            Source = "aaa"
        };

        var expression = new NonterminalExpression();
        expression.Interpret(context);

        Console.WriteLine(context.Result);
    }
}
```

```
class Context
{
    // Контекст распознающей грамматики языка:
    // V = {a}; L = V*;
    // Правильные цепочки: a, aa, aaa, ...
    // Неправильные цепочки: b, ab, aba, ...

    public string Source { get; set; }
    public char Vocabulary { get; set; }
    public bool Result { get; set; }
    public int Position { get; set; }
}
```

```
abstract class AbstractExpression
{
    public abstract void Interpret(Context context);
}
```

```
class TerminalExpression : AbstractExpression
{
    public override void Interpret(Context context)
    {
        context.Result =
        context.Source[context.Position] == context.Vocabulary;
    }
}
```

```
class NonterminalExpression : AbstractExpression
{
    AbstractExpression nonterminalExpression;
    AbstractExpression terminalExpression;

    public override void Interpret(Context context)
    {
        if (context.Position < context.Source.Length)
        {
            terminalExpression = new TerminalExpression();
            terminalExpression.Interpret(context);
            context.Position++;

            if (context.Result)
            {
                nonterminalExpression = new NonterminalExpression();
                nonterminalExpression.Interpret(context);
            }
        }
    }
}
```

См. Пример к главе: \015_Interpreter\001_Interpreter

Участники

- `AbstractExpression` - **Абстрактное выражение**:
  Предоставляет абстрактный метод `Interpret`, который будет реализован во всех узлах абстрактного синтаксического дерева.

- `TerminalExpression` - **Терминальное выражение**:
  Реализует абстрактный метод `Interpret` для анализа терминальных символов грамматики, которые входят в словарь **V**. Для каждого терминального символа в предложении (буквы или слова) требуется создавать отдельный класс `TerminalExpression`.

- `NonterminalExpression` - **Нетерминальное выражение**:
  Представляет собой объектно-ориентированное представление правила **R** (*Rule*). Для каждого правила **R = {R_1, R_2, … R_n}** требуется создавать отдельный класс `NonterminalExpression`. Реализует абстрактный метод `Interpret` для работы с правилами (нетерминальными символами грамматики).

- `Context` - **Контекст**:
  Содержит в себе цепочки определенного языка и другую вспомогательную информацию требуемую для работы интерпретатора.

- `Client` - **Клиент**:
  Строит (или получает в готовом виде) абстрактное синтаксическое дерево, представляющее собой отдельное предложение на языке заданном определенной грамматикой. Дерево составляется из экземпляров классов `TerminalExpression` и `NonterminalExpression`.

Отношения между участниками

Отношения между классами

- Класс `Client` связан связями отношения ассоциации с классами `Context` и `AbstractExpression`.

- Класс `TerminalExpression` связан связью отношения наследования с абстрактным классом `AbstractExpression`.

- Класс `NonterminalExpression` связан связью отношения наследования и связью отношения агрегации с абстрактным классом `AbstractExpression`.

Отношения между объектами

- Клиент строит (или получает в готовом виде) абстрактное синтаксическое дерево, представляющее собой отдельное предложение на языке заданном определенной грамматикой. Дерево составляется из экземпляров классов `TerminalExpression` и `NonterminalExpression`.

Паттерн Iterator

Название

Итератор

Также известен как

Cursor (Курсор)

Классификация

По цели: поведенческий
По применимости: к объектам

Частота использования

Высокая - 1 2 3 4 **5**

Назначение

Паттерн Iterator - предоставляет удобный и безопасный способ доступа к элементам коллекции (составного объекта), при этом не раскрывая внутреннего представления этой коллекции.

Введение

Что такое коллекция в объективной реальности? Коллекция (от происходит от латинского *collectio* - *собирание*, *сбор*) - систематизированное собрание чего-либо, объединённое по какому-то конкретному признаку, имеющее внутреннюю целостность и принадлежащее конкретному владельцу - частному лицу, организации, государству.

Предлагается рассмотреть коллекцию на примере банка, как коллекции денег и ценностей.

Исторически главной функцией банков было безопасное хранение денег клиентов. Во многих странах банки обязаны хранить тайну об операциях, счетах и вкладах клиентов. Может ли клиент банка самостоятельно войти в банковское хранилище и положить деньги на депозит или взять кредит? Конечно же нет. Получается, что клиент банка не имеет прямого доступа к элементам (деньгам) коллекции (банка).

Каким же способом клиент может положить деньги в банк или взять деньги из банка? Только через кассира. Кассир-операционист - лицо банка, ведь он первый, кто встречает посетителей и только он может получить доступ к хранилищу. Именно кассир осуществляет операции по приему, учету, выдаче и хранению денежных средств с обязательным соблюдением правил, обеспечивающих их сохранность.



Если перейти на язык информатики, то банк можно представить программной коллекцией, а кассира механизмом доступа (*итератором*) к элементам коллекции. Коллекция в программировании - это программный объект, содержащий в себе, набор элементов одного или различных типов. Итератор (iterator) - это объект, предоставляющий безопасный доступ к элементам коллекции.

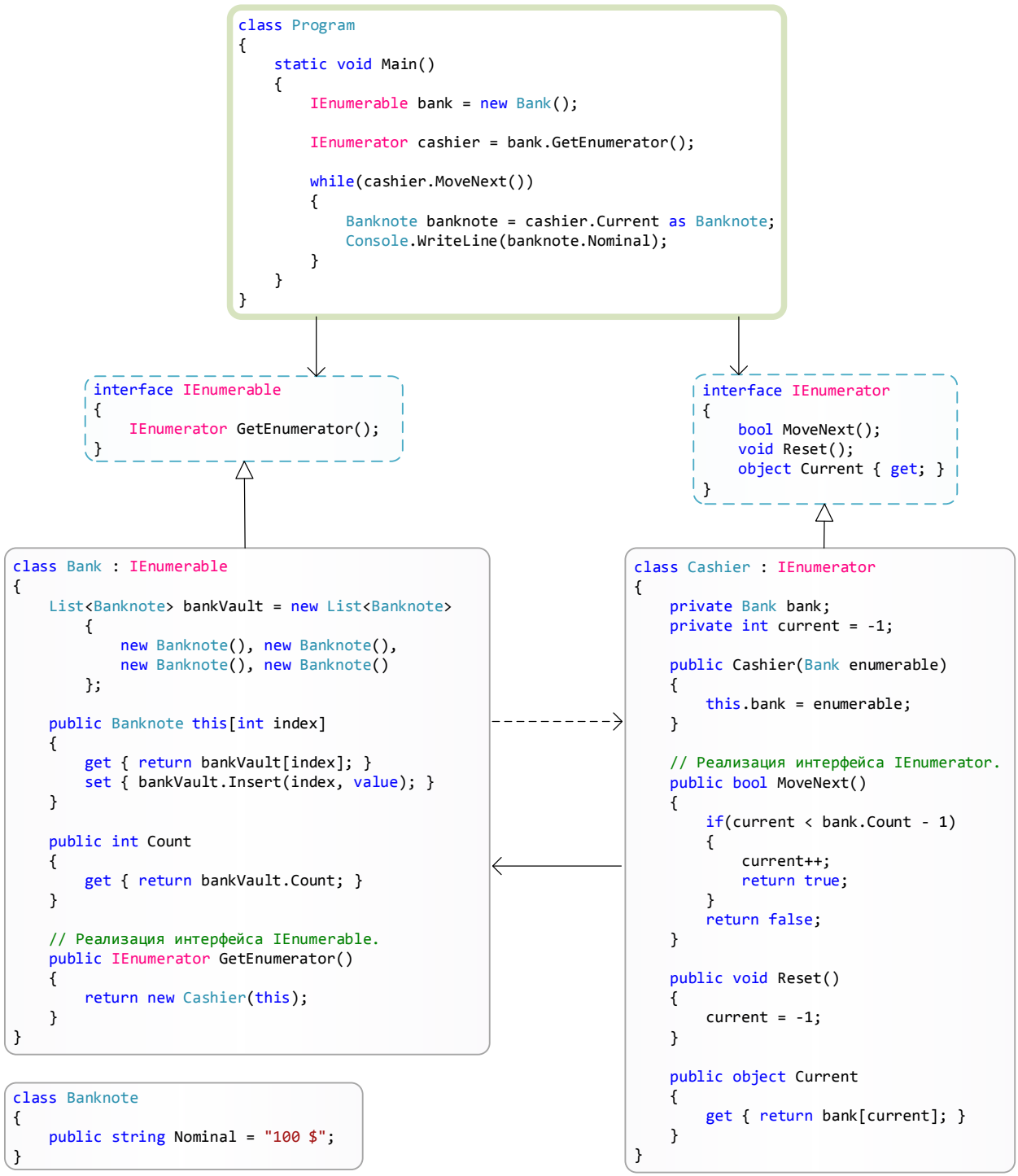
В инфраструктуре .Net большинство объектов-коллекций реализует интерфейс `IEnumerable`, а объекты-итераторы называются «*перечислителями*» (enumerators) и реализуют интерфейс `IEnumerator`.

IEnumerable

IEnumerator

В случае с примером банка и кассира становится понятно, что банк является тем *что перечисляют* (перечисляемый - `IEnumerable`), а кассир является тем, *кто перечисляет* (перечислитель - `IEnumerator`) и идея использования техники «перечисляемый-перечислитель» была положена в основу паттерна Iterator.

Ниже представлен пример использования техники «перечисляемый-перечислитель» в интерпретации Microsoft (с использованием интерфейсов `IEnumerable` и `IEnumerator`).



```
class Program
{
    static void Main()
    {
        IEnumerable bank = new Bank();

        IEnumerator cashier = bank.GetEnumerator();

        while(cashier.MoveNext())
        {
            Banknote banknote = cashier.Current as Banknote;
            Console.WriteLine(banknote.Nominal);
        }
    }
}
```

```
interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

```
interface IEnumerator
{
    bool MoveNext();
    void Reset();
    object Current { get; }
}
```

```
class Bank : IEnumerable
{
    List<Banknote> bankVault = new List<Banknote>
        {
            new Banknote(), new Banknote(),
            new Banknote(), new Banknote()
        };

    public Banknote this[int index]
    {
        get { return bankVault[index]; }
        set { bankVault.Insert(index, value); }
    }

    public int Count
    {
        get { return bankVault.Count; }
    }

    // Реализация интерфейса IEnumerable.
    public IEnumerator GetEnumerator()
    {
        return new Cashier(this);
    }
}
```

```
class Cashier : IEnumerator
{
    private Bank bank;
    private int current = -1;

    public Cashier(Bank enumerable)
    {
        this.bank = enumerable;
    }

    // Реализация интерфейса IEnumerator.
    public bool MoveNext()
    {
        if(current < bank.Count - 1)
        {
            current++;
            return true;
        }
        return false;
    }

    public void Reset()
    {
        current = -1;
    }

    public object Current
    {
        get { return bank[current]; }
    }
}
```

```
class Banknote
{
    public string Nominal = "100 $";
}
```

См. Пример к главе: \016_Iterator\000_ Bank

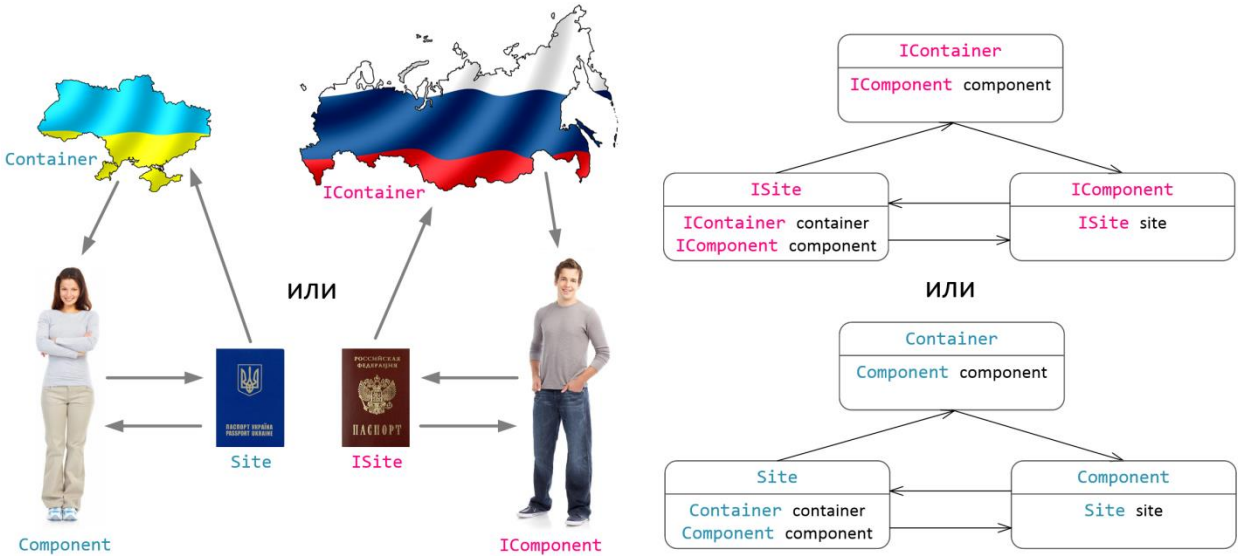
Воспринимать паттерн Iterator как шаблон для создания программных коллекций – это грубо и вульгарно прямолинейно. В действительности паттерн Iterator не описывает построение полноценной коллекции. Паттерн Iterator описывает только технику - «*перечисляемый-перечислитель*» которая лежит в основе понятия *составного объекта*.

Составным объектом же может являться, как программный *контейнер,* так и программная *коллекция*. Термины «*коллекции*» и «*контейнеры*» иногда программистами используются довольно свободно и взаимозаменяемо.

В информатике *контейнер* представляет собой разновидность (настраиваемый класс) *коллекции*. Контейнер содержит в себе специальные объекты-компоненты. Контейнер управляет внутренним взаимодействием компонентов друг с другом (внутри контейнера), а также контролирует взаимодействие внутренних компонентов с другими (внешними) объектами, находящимися за пределами контейнера. Другими словами, *контейнер* – это связующее звено между компонентами, содержащимися в этом контейнере и внешними объектами из кода приложения. Внешние объекты могут получить ссылки на компоненты, содержащиеся в контейнере, не зная настоящего имени этого компонента.

Для того чтобы правильно создать пользовательский контейнер и наполнить его компонентами, потребуется воспользоваться тремя интерфейсами `IContainer`, `IComponent` и `ISite`. Или воспользоваться напрямую или через наследование готовыми классами контейнера `Container` и компонентов `Component`, которые идут в поставке .Net Framework. Доступ к компонентам, содержащимся в контейнере, можно получить с помощью свойства `ComponentCollection Components` этого контейнера.

Для лучшего понимания идеи совместного использования контейнера `Container` и компонентов `Component` предлагается воспользоваться метафорой. Контейнер - `Container` (`IContainer`) можно проассоциировать со страной, в которой проживает человек – `Component` (`IComponent`), у которого имеется паспорт - `Site` (`ISite`) в котором указано имя человека и гражданство (принадлежность к стране).



Объект типа `Site` (`ISite`) предоставляет дополнительное удобство для работы с компонентами. Например, в давние времена не было паспортов и некоторым людям (рабам) ставили клеймо на теле с именем и принадлежностью. Конечно такой подход не удобен.

Ниже представлен пример реализации интерфейсов `IContainer`, `IComponent` и `ISite`, но также можно воспользоваться уже имеющимися классами контейнера `Container` и компонентов `Component`, которые идут в поставке FCL (*Framework Class Library*).

```
class Program
{
    static void Main()
    {
        Container container = new Container();

        Component component1 = new Component();
        Component component2 = new Component();
        Component component3 = new Component();

        container.Add(component1, "First");
        container.Add(component2, "Second");
        container.Add(component3, "Third");

        ComponentCollection components =
                              container.Components;

        foreach (Component component in components)
            Console.WriteLine("Component : " +
                                 component.Site.Name);

        component1.Disposed +=
                     (object sender, EventArgs e) =>
        Console.WriteLine("First Component Disposed");

        container.Dispose();
    }
}
```

```
public interface IContainer : IDisposable
{
    ComponentCollection Components { get; }
    void Add(IComponent component);
    void Add(IComponent component, string name);
    void Remove(IComponent component);
}
```

```
class Container : IContainer
{
    List<IComponent> list = new List<IComponent>();

    public ComponentCollection Components
    {
        get { return new ComponentCollection(list.ToArray()); }
    }

    // Реализация интерфейса IContainer.

    public virtual void Add(IComponent component)
    {
        list.Add(component);
    }

    public virtual void Add(IComponent component, string name)
    {
        component.Site = new Site
        {
            Component = component,
            Container = this,
            Name = name
        };

        list.Add(component);
    }

    public virtual void Remove(IComponent component)
    {
        list.Remove(component);
    }

    // Реализация интерфейса IDisposable.

    public virtual void Dispose()
    {
        list.ForEach(component => component.Dispose());
        list.Clear();
    }
}
```

```
public interface IComponent : IDisposable
{
    ISite Site { get; set; }
    event EventHandler Disposed;
}
```

```
public interface ISite : IServiceProvider
{
    IComponent Component { get; }
    IContainer Container { get; }
    bool DesignMode { get; }
    string Name { get; set; }
}
```

```
class Component : IComponent
{
    // Реализация интерфейса IComponent.
    public virtual ISite Site { get; set; }
    public event EventHandler Disposed;

    // Реализация интерфейса IDisposable.
    public virtual void Dispose()
    {
        if (Disposed != null)
            Disposed.Invoke(this, EventArgs.Empty);
    }
}
```

```
class Site : ISite
{
    // Реализация интерфейса ISite.
    public virtual IComponent Component { get;  set; }
    public virtual IContainer Container { get;  set; }
    public virtual bool DesignMode { get { return false; } }
    public virtual string Name { get; set; }

    // Реализация интерфейса IServiceProvider.
    public virtual object GetService(Type serviceType)
    {
        // В данном примере не используются сервисы.
        return null;
    }
}
```

См. Пример к главе: \016_Iterator\002_Containers

Коллекции в программировании классифицируются по общим характеристикам, по логике организации и по реализации.

Коллекции классифицирующиеся по общим характеристикам могут иметь фиксированный или динамически изменяющийся размер, а также содержать элементы одного типа или разных типов. Коллекция содержащая разнотипные элементы называется – *гетерогенной* коллекцией (например, `ArrayList`).

Коллекции классифицирующиеся по логике организации: вектор, матрица, многомерный массив, зубчатый (рваный) массив, список, стек, очередь, дека, ассоциативный массив «*словарь*», множество, мультимножество, кортеж, сеть и др.

Коллекции классифицирующиеся по уровню реализации: массив, односвязный список, двусвязный список, стек, хеш-таблица, битовый массив, битовый вектор и др.

Для реализации простейшей (в общем виде) коллекции в .NET Framework используется интерфейс `ICollection`. Ниже показан пример реализации интерфейса `ICollection`. Как видно из примера интерфейс коллекции `ICollection` расширяет собой интерфейс простейшей перечисляемой сущности `IEnumerable`.

```
class Program
{
    static void Main()
    {
        var collection = new Collection();

        foreach (var item in collection)
        {
            Console.WriteLine(item);
        }

        int[] array = new int[20];

        collection.CopyTo(array, 2);

        foreach (int element in array)
        {
           Console.Write(element);
        }
    }
}
```

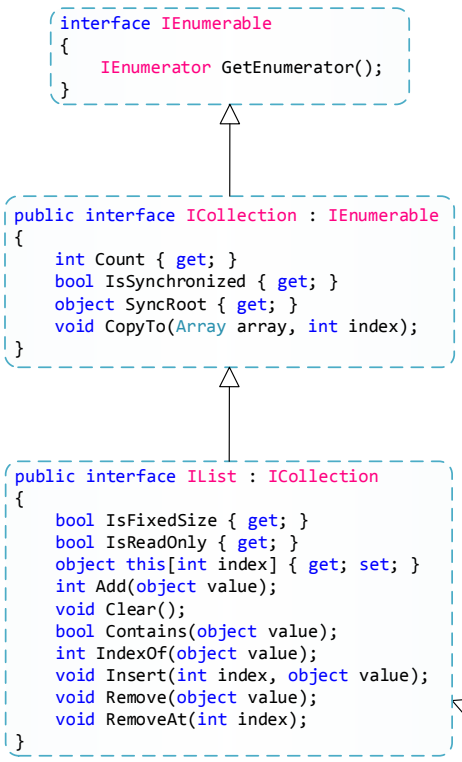
```
interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

```
public interface ICollection : IEnumerable
{
    int Count { get; }
    bool IsSynchronized { get; }
    object SyncRoot { get; }
    void CopyTo(Array array, int index);
}
```

```
class Collection : ICollection
{
    readonly object syncRoot = new object();

    readonly object[] elements = { 1, 2, 3, 4 };

    public void CopyTo(Array array, int userArrayIndex)
    {
        elements.CopyTo(array, userArrayIndex);
    }

    public int Count
    {
        get { return elements.Length; }
    }

    public bool IsSynchronized
    {
        get { return true; }
    }

    public object SyncRoot
    {
        get { return syncRoot; }
    }

    public IEnumerator GetEnumerator()
    {
        return elements.GetEnumerator();
    }
}
```

См. Пример к главе: \016_Iterator\005_ICollection

Для реализации простейшего списка в .NET Framework используется интерфейс `IList`. Ниже показан пример реализации интерфейса `IList`. Как видно из примера интерфейс списка `IList` расширяет собой интерфейс коллекции `ICollection`.



```
interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

```
public interface ICollection : IEnumerable
{
    int Count { get; }
    bool IsSynchronized { get; }
    object SyncRoot { get; }
    void CopyTo(Array array, int index);
}
```

```
public interface IList : ICollection
{
    bool IsFixedSize { get; }
    bool IsReadOnly { get; }
    object this[int index] { get; set; }
    int Add(object value);
    void Clear();
    bool Contains(object value);
    int IndexOf(object value);
    void Insert(int index, object value);
    void Remove(object value);
    void RemoveAt(int index);
}
```

```
class List : IList
{
    ArrayList elements = new ArrayList();

    // Реализация IList

    public int Add(object value)
    {
        return elements.Add(value);
    }

    public void Clear()
    {
        elements.Clear();
    }

    public bool Contains(object value)
    {
        return elements.Contains(value);
    }

    public int IndexOf(object value)
    {
        return elements.IndexOf(value);
    }

    public void Insert(int index, object value)
    {
        elements.Insert(index, value);
    }

    public bool IsFixedSize
    { get { return elements.IsFixedSize; }}

    public bool IsReadOnly
    { get { return elements.IsReadOnly; }}

    public void Remove(object value)
    {
        elements.Remove(value);
    }

    public void RemoveAt(int index)
    {
        elements.RemoveAt(index);
    }

    public object this[int index]
    {
        get { return elements[index]; }
        set { elements[index] = value; }
    }

    // Реализация ICollection

    public void CopyTo(Array array, int index)
    {
        elements.CopyTo(array, index);
    }

    public int Count
    { get { return elements.Count; }}

    public bool IsSynchronized
    { get { return elements.IsSynchronized; }}

    public object SyncRoot
    { get { return elements.SyncRoot; }}

    // Реализация IEnumerable
    public IEnumerator GetEnumerator()
    {
        return elements.GetEnumerator();
    }
}
```

```
class Program
{
    static void Main()
    {
        var collection = new Collection();

        foreach (var item in collection)
        {
            Console.WriteLine(item);
        }

        int[] array = new int[20];

        collection.CopyTo(array, 2);

        foreach (int element in array)
        {
           Console.Write(element);
        }
    }
}
```

См. Пример к главе: \016_Iterator\006_IList

Предлагается рассмотреть создание класса итератора при помощи оператора `yield`, как оператора автоматической генерации программного кода класса итератора. Задача оператора `yield` сгенерировать (написать без участия человека) программный код класса итератора для той коллекции в которой он используется. Оператор `yield` облегчает работу программиста, избавляя его от необходимости вручную писать код класса итератора. При этом сама реализация класса итератора оказывается скрытой от программиста, что добавляет немного «магии», для тех разработчиков, которые не до конца понимают работу оператора `yield`.

Пример ниже показывает создание итератора с использованием оператора `yield` для коллекции `Enumerable` размерностью в один строковой элемент со значением `"Element"`.

```
class Program
{
    static void Main()
    {
        Enumerable enumerable = new Enumerable();

        foreach (var item in enumerable)
        {
            Console.WriteLine(item);
        }
    }
}
```

```
class Enumerable
{
    public IEnumerator GetEnumerator()
    {
        yield return "Element";
    }
}
```

См. Пример к главе: \016_Iterator\007_Yield

Для того чтобы понять работу оператора `yield`, желательно посмотреть на код класса итератора, который генерируется оператором `yield`. Для этого рекомендуется воспользоваться инструментом для обратной инженерии (*дизассемблирования*) – программой dotPeek от компании JetBrains известной как производителя программы ReSharper.

Для анализа потребуется открыть в программе dotPeek исполняемый файл *.exe содержащий код итератора. В результате дизассемблирования будет представлен следующий код *(для упрощения понимания дизассемблированный код был упрощен)*:

```
class Program
{
    static void Main()
    {
        Enumerable enumerable = new Enumerable();

        foreach (var item in enumerable)
        {
            Console.WriteLine(item);
        }
    }
}
```

```
class Enumerable
{
    public IEnumerator GetEnumerator()
    {
        return new Enumerator();
    }

    class Enumerator : IEnumerator
    {
        object current;
        int state = 0;

        bool IEnumerator.MoveNext()
        {
            switch (state)
            {
                case 0:
                    current = "Element";
                    state = 1;
                    return true;
                case 1:
                    state = 0;
                    break;
            }
            return false;
        }
        void IEnumerator.Reset()
        {
            state = 0;
        }
        object IEnumerator.Current
        {
            get { return current; }
        }
    }
}
```

См. Пример к главе: \016_Iterator\008_Yield

При сравнении двух участков кода видно, сколько рутинной работы по созданию класса итератора `Enumerator` и реализации логики по взаимодействию с коллекцией `Enumerable`, взял на себя оператор `yield`. Программисту остается сосредоточится на бизнес логике приложения, а не задумываться над деталями технической реализации класса итератора.

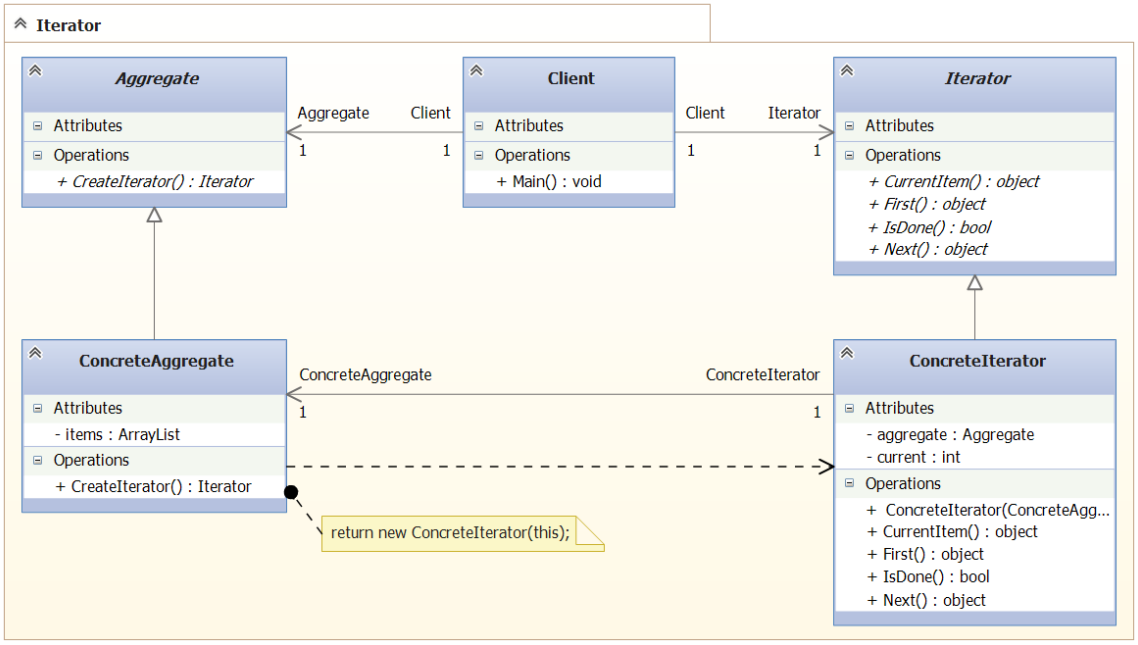
```
class Enumerable
{
    public IEnumerator GetEnumerator()
    {
        yield return "Element";
    }
}
```

=

```
class Enumerable
{
    public IEnumerator GetEnumerator()
    {
        return new Enumerator();
    }

    class Enumerator : IEnumerator
    {
        object current;
        int state = 0;

        bool IEnumerator.MoveNext()
        {
            switch (state)
            {
                case 0:
                    current = "Element";
                    state = 1;
                    return true;
                case 1:
                    state = 0;
                    break;
            }
            return false;
        }
        void IEnumerator.Reset()
        {
            state = 0;
        }
        object IEnumerator.Current
        {
            get { return current; }
        }
    }
}
```

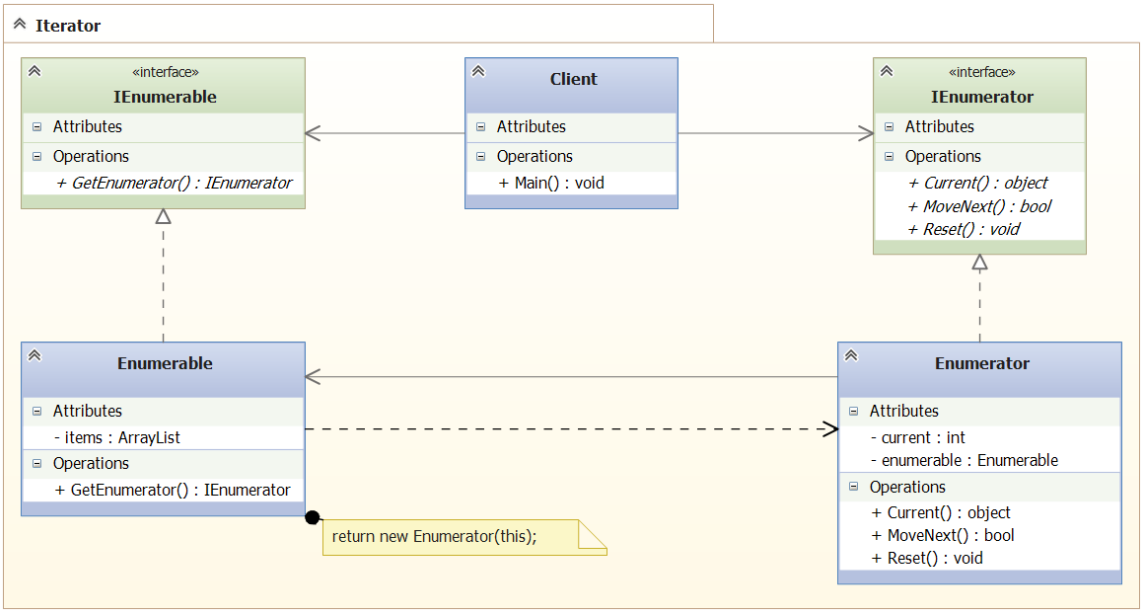
Структура паттерна на языке UML

Классическое представление



См. Пример к главе: \016_Iterator\001_Iterator

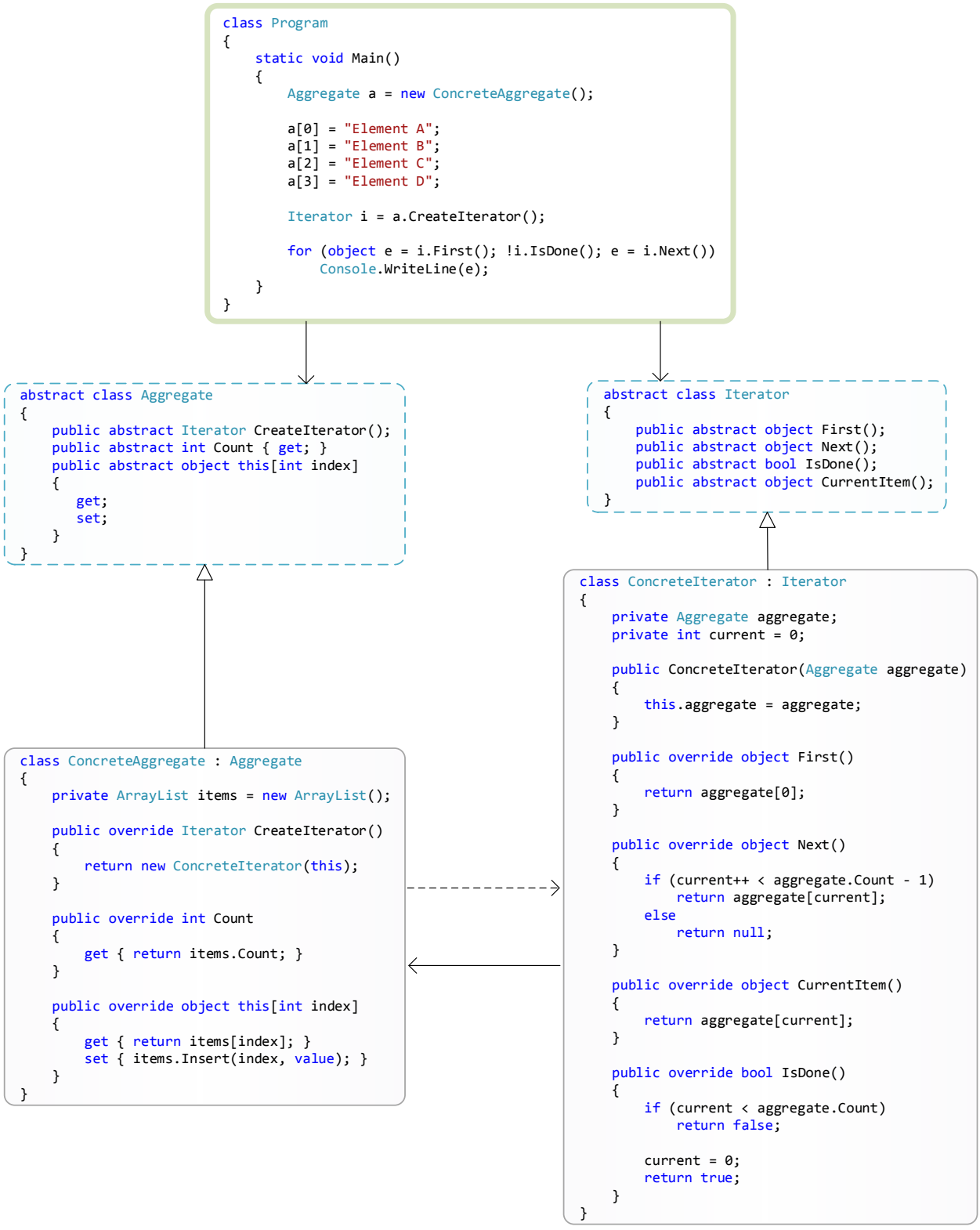
Представление Microsoft .NET



См. Пример к главе: \016_Iterator\003_Enumerator

Структура паттерна на языке C#

Классическое представление



```
class Program
{
    static void Main()
    {
        Aggregate a = new ConcreteAggregate();

        a[0] = "Element A";
        a[1] = "Element B";
        a[2] = "Element C";
        a[3] = "Element D";

        Iterator i = a.CreateIterator();

        for (object e = i.First(); !i.IsDone(); e = i.Next())
            Console.WriteLine(e);
    }
}
```

```
abstract class Aggregate
{
    public abstract Iterator CreateIterator();
    public abstract int Count { get; }
    public abstract object this[int index]
    {
       get;
       set;
    }
}
```

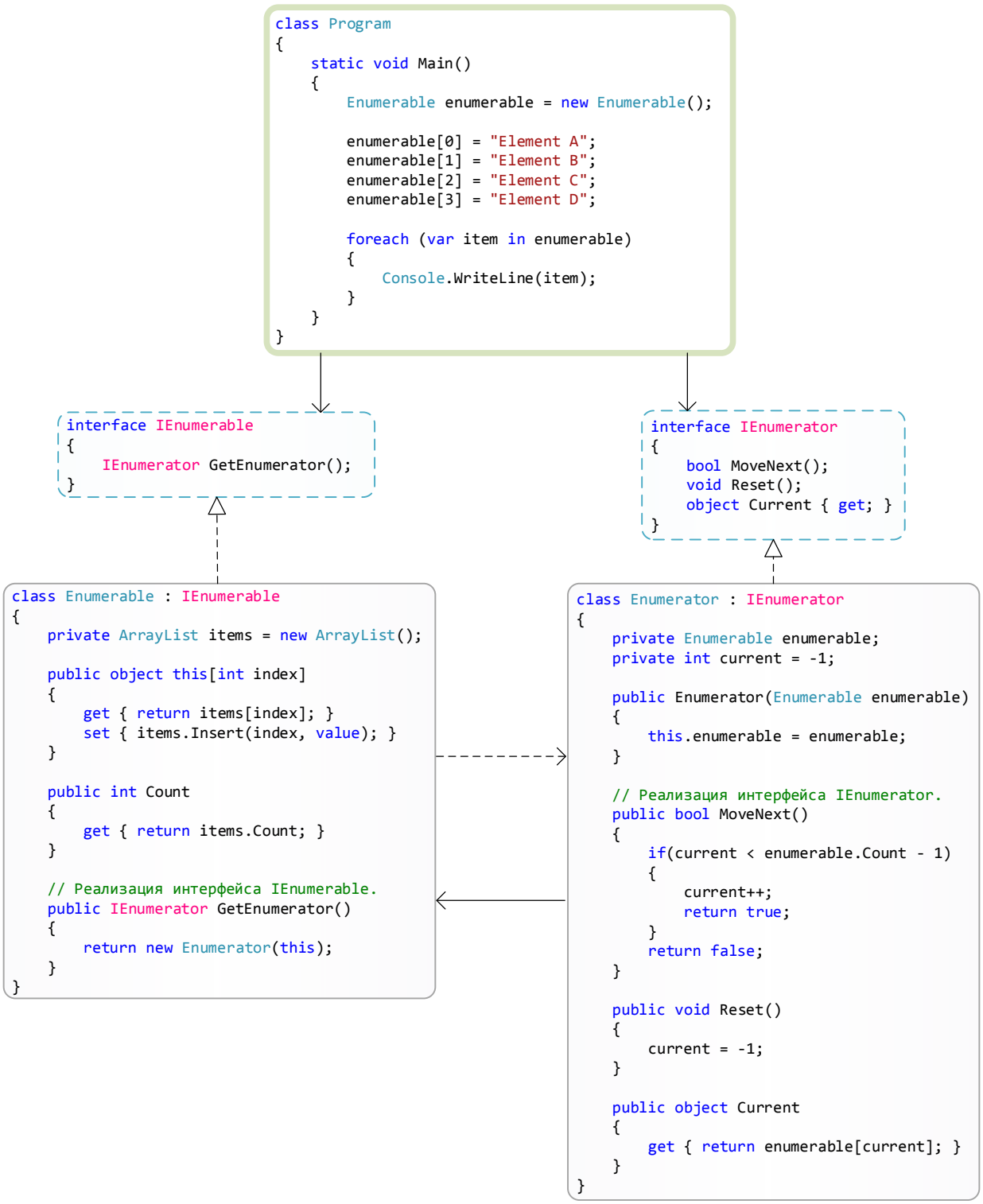
```
abstract class Iterator
{
    public abstract object First();
    public abstract object Next();
    public abstract bool IsDone();
    public abstract object CurrentItem();
}
```

```
class ConcreteAggregate : Aggregate
{
    private ArrayList items = new ArrayList();

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

    public override int Count
    {
        get { return items.Count; }
    }

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

```
class ConcreteIterator : Iterator
{
    private Aggregate aggregate;
    private int current = 0;

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

    public override object First()
    {
        return aggregate[0];
    }

    public override object Next()
    {
        if (current++ < aggregate.Count - 1)
            return aggregate[current];
        else
            return null;
    }

    public override object CurrentItem()
    {
        return aggregate[current];
    }

    public override bool IsDone()
    {
        if (current < aggregate.Count)
            return false;

        current = 0;
        return true;
    }
}
```

См. Пример к главе: \016_Iterator\001_Iterator

Представление Microsoft .NET



См. Пример к главе: \016_Iterator\003_Enumerator

Участники

- `Iterator` (`IEnumerator`) - **Итератор**:
  Предоставляет интерфейс (набор методов) для доступа к коллекции и обхода элементов.

- `ConcreteIterator` - **Конкретный итератор**:
  Реализует интерфейс класса `Iterator`. Следит за позицией текущего элемента при переборе коллекции (`Aggregate`).

- `Aggregate` (`IEnumerable`) - **Агрегат**:
  Предоставляет интерфейс коллекции (набор методов) в том числе методы для создания объекта-итератора.

- `ConcreteAggregate` - **Конкретный агрегат**:
  Реализует интерфейс коллекции и хранит в себе элементы.

Отношения между участниками

Отношения между классами

- Конкретный класс `ConcreteAggregate` связан связью отношения наследования с абстрактным классом `Aggregate` и связью отношения зависимости с конкретным классом `ConcreteIterator`.

- Конкретный класс `ConcreteIterator` связан связью отношения наследования с абстрактным классом `Iterator` и связью отношения ассоциации с конкретным классом `ConcreteAggregate`.

Отношения между объектами

- `ConcreteIterator` отслеживает текущий элемент в коллекции (`Aggregate`) и может вычислить следующий за ним элемент.

Мотивация

Правильно спроектированная коллекция (составной объект), должна предоставлять возможность доступа к своим элементам удобным способом, при этом не раскрывать своего внутреннего устройства (представления). Иногда требуется организовать обход элементов коллекции разными способами. При этом не желательно формировать интерфейс класса `List` из операций различных вариантов обхода списка (коллекции), даже если все операции можно предусмотреть заранее. Кроме того, иногда требуется, чтобы одновременно имелось несколько вариантов обхода списка.

Применимость паттерна

Паттерн Iterator рекомендуется использовать, когда:

- Требуется организовать доступ к элементам (содержимому) коллекции (агрегированного объекта) без раскрытия устройства (внутреннего представления) коллекции.
- Необходимо организовать несколько вариантов обхода коллекции.
- Требуется сформировать единообразный интерфейс (набор методов) с целью унифицированного обращения к элементам из различных коллекций, другими словами для поддержки техники «*полиморфной итерации*» (один итератор для нескольких коллекций).

Результаты

Использование паттерна Iterator предоставляет следующие возможности:

- **Поддержка различных видов обхода коллекции (агрегата).**
  Сложные агрегаты (составные объекты – коллекции или контейнеры) можно обходить по-разному. Например, если внутренне устройство коллекции (агрегата) имеет древообразную структуру, требуется организовать правильный обход частей дерева или генерацию новых частей дерева. Итератор может обходить дерево во внутреннем порядке (только части дерева) или в прямом порядке (от корня к листьям). Использование итераторов позволяет организовать динамическую подмену алгоритмов обхода дерева, для этого достаточно заменить один экземпляр итератора другим. Для поддержки новых алгоритмов обхода можно создать новую разновидность итератора.

- **Итераторы упрощают интерфейс коллекции (агрегата).**
  Наличие интерфейса (набора методов) для обхода элементов коллекции в классе итератор (`Iterator`), избавляет от дублирования этого интерфейса непосредственно в классе агрегата (`Aggregate`), тем самым упрощая интерфейс агрегата.

- **Для одной коллекции (агрегата), одновременно может быть активно несколько вариантов обхода.**
  Имеется возможность организации одновременного обращения к элементам коллекции, параллельно из разных потоков. Такие коллекции называются «*потокобезопасными коллекциями*» (Concurrency collections), а иногда их называют просто «*параллельными коллекциями*». Потокобезопасные коллекции следует использовать вместо обычных коллекций, только тогда, когда требуется организовать одновременный доступ к элементам коллекции из разных потоков. При работе с такими коллекциями, итератор следит за состоянием обхода, отслеживая текущую позицию в коллекции и синхронизируя свою работу с работой коллекции и работой других итераторов.

  В поставку Microsoft .NET FCL включены готовые решения потокобезопасных коллекций. Пространство имен `System.Collections.Concurrent` содержит классы потокобезопасных коллекций, такие как: `BlockingCollection<T>`, `ConcurrentBag<T>`, `ConcurrentDictionary<TKey, TValue>`, `ConcurrentQueue<T>`, `ConcurrentStack<T>`, `OrderablePartitioner<TSource>`, `Partitioner` и `Partitioner<TSource>`. При построении собственной потокобезопасной коллекции рекомендуется реализовать интерфейс `IProducerConsumerCollection<T>`, который задает методы для работы с потокобезопасными коллекциями.

Реализация

Полезные приемы реализации паттерна Iterator:

Существует несколько способов реализации паттерна итератор. Решение о том, какой способ выбрать, часто зависит от управляющих структур, поддерживаемых языком программирования. Некоторые языки (например, C#) поддерживают реализацию паттерна Iterator в своем синтаксисе (оператор - `yield`).

Наиболее употребительные варианты реализации паттерна итератор:

- **Внешний и внутренний итераторы.**
  Важнейший вопрос состоит в том, что управляет итерацией - сам *итератор* или *клиент*, который пользуется итератором. Если итерацией управляет клиент, то итератор называется внешним (*активным*), в противном случае если итерация производится автоматически, без участия клиента, итератор называется внутренним (*пассивным*). Термины «*активный*» и «*пассивный*» относятся к роли клиента, а не к действиям, выполняемым итератором. Клиенты, применяющие внешний (*активный*) итератор, должны явно запрашивать у итератора следующий элемент, чтобы двигаться дальше по коллекции. Напротив, в случае использования внутреннего (*пассивного*) итератора клиент передает итератору некоторую операцию (или лямбда оператор), а итератор уже сам применяет эту операцию к каждому посещенному во время обхода элементу коллекции. Сильные стороны внутренних (*пассивных*) итераторов наиболее отчетливо проявляются в таких языках, как C#, где есть анонимные функции, замыкание (*closure*) и продолжения (*continuation*). Также, внутренние (*пассивные*) итераторы проще в использовании, поскольку они легко конфигурируются новой логикой по работе с коллекцией.

  Пример работы внутреннего итератора реализованного с использованием оператора `yield`:

```
class Program
{
    static double Power2(double n)
    {
        return Math.Pow(n, 2);
    }

    static void Main()
    {
        Enumerable enumerable = new Enumerable();

        IEnumerable power2List = enumerable.Transform(new Function(Power2));

        foreach (var item in power2List)
            Console.WriteLine(item);

        IEnumerable power3List = enumerable.Transform(n => Math.Pow(n, 3));

        foreach (var item in power3List)
            Console.WriteLine(item);
    }
}
```

```
delegate double Function(double arg);
```

```
class Enumerable
{
    List<double> list = new List<double> { 1, 2, 3, 4 };

    public IEnumerable Transform(Function function)
    {
        foreach (double item in list)
            yield return function(item);
    }
}
```

См. Пример к главе: \016_Iterator\004_InteralIterator [001]

Пример работы внутреннего итератора, реализованного с использованием традиционного подхода:

```
class Program
{
    static double Power2(double n)
    {
        return Math.Pow(n, 2);
    }

    static void Main()
    {
        Enumerable enumerable = new Enumerable();

        IEnumerable power2List = enumerable.Transform(new Function(Power2));

        foreach (var item in power2List)
            Console.WriteLine(item);

        IEnumerable power3List = enumerable.Transform(n => Math.Pow(n, 3));

        foreach (var item in power3List)
            Console.WriteLine(item);
    }
}
```

```
delegate double Function(double arg);
```

```
public class Enumerable
{
    List<double> list = new List<double> { 1, 2, 3, 4 };

    public IEnumerable Transform(Function function)
    {
        return new Enumerator
        {
            Enumerable = this,
            Function = function
        };
    }

    // Внутренний (пассивный) итератор. [Nested class]
    class Enumerator : IEnumerable, IEnumerator
    {
        public Enumerable Enumerable { get; set; }
        public Function Function { get; set; }
        object current;
        int position = -1;

        bool IEnumerator.MoveNext()
        {
            if (position < Enumerable.list.Count - 1)
            {
                position++;
                current = Function(Enumerable.list[position]);
                return true;
            }
            return false;
        }

        void IEnumerator.Reset()
        {
            position = -1;
        }

        object IEnumerator.Current
        {
            get { return current; }
        }

        IEnumerator IEnumerable.GetEnumerator()
        {
            return this;
        }
    }
}
```

См. Пример к главе: \016_Iterator\004_InteralIterator [002]

- **Итератор – курсор.**
  Алгоритм обхода коллекции может содержаться не только в итераторе. Алгоритм обхода может находиться непосредственно в коллекции, а коллекция может использовать итератор только для хранения состояния итерации (например, указателя позиции текущего элемента). Такого рода итератор называют курсором, поскольку он всего лишь указывает текущую позицию элемента в коллекции. Клиент может вызывать метод Next коллекции, передавая методу Next в качестве аргумента итератор-курсор. Метод Next в свою очередь состояние итератора-курсора (например, увеличивает значение поля position на один). Если же за алгоритм обхода коллекции отвечает итератор, то для одной и той же коллекции можно использовать разные алгоритмы обхода, а также проще применить один алгоритм обхода к разным коллекциям (*полиморфная итерация*). Часто бывает так, что для выполнения алгоритма обхода может понадобиться предоставить доступ к закрытым членам коллекции. В таком случае потребуется организовать то перенос алгоритма обхода в итератор, что нарушает инкапсуляцию коллекции. Для сохранения инкапсуляции коллекции можно итератор сделать «*nested*» классом, вложенным в класс коллекции, что позволит сохранить инкапсуляцию коллекции.

- **Устойчивый (робастный) итератор.**
  Изменение коллекции в то время, как совершается ее обход, может оказаться опасным. Если во время обхода коллекции добавляются или удаляются элементы, то не исключено, что некоторый элемент может быть «*посещен*» дважды или вообще ни разу. Можно скопировать коллекцию и обходить ее копию, но такой подход не является эффективным и обходится слишком дорого. *Устойчивый итератор* (*robust*) гарантирует, что ни вставки новых элементов, ни удаление существующих не помешают правильному обходу коллекции, причем стабильность работы коллекции достигается без создания ее копии. Имеются различные способы реализации устойчивых итераторов. В большинстве случаев *устойчивый итератор* регистрируется в коллекции. При вставке или удалении элемента, коллекция либо подправляет внутреннее состояние всех созданных ею итераторов, либо организует порядок следования своих элементов так, чтобы обход элементов выполнялся правильно.

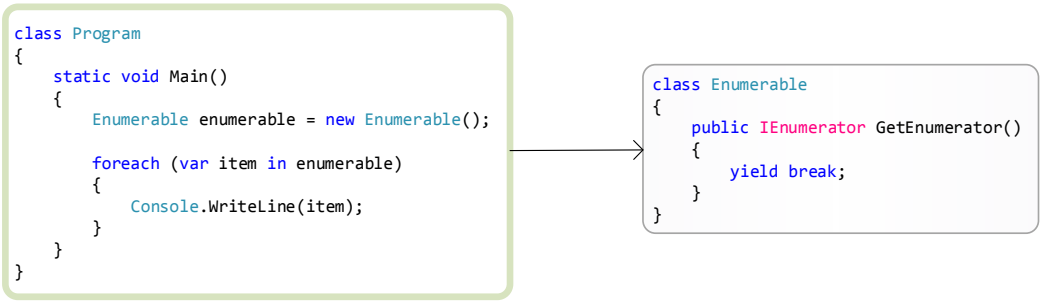
- **Полиморфные итераторы.**
  Полиморфный итератор, это такой итератор который может использоваться для обхода элементов разных коллекций (один итератор для нескольких коллекций). При реализации полиморфных итераторов могут возникать определенные сложности.

- **Итераторы для коллекций с древообразной структурой.**
  Реализация внешних (*активных*) итераторов для рекурсивно агрегированных древообразных структур данных (деревьев) может оказаться затруднительной, так как положение элемента в древообразной структуре данных (дереве) может находиться на нескольких уровнях вложенности (лист, одновременно принадлежащий нескольким ветвям). Поэтому, чтобы отследить позицию текущего элемента, внешний (*активный*) итератор должен хранить все возможные пути к определенному элементу и исключить повторную обработку такого элемента при обходе дерева. Иногда проще воспользоваться внутренним (*пассивным*) итератором

  Деревья часто приходится обходить несколькими способами. Самые распространенные способы - это обход в прямом порядке (от корня к листьям), обратном порядке (от листьев к корню – когда у потомков имеются ссылки на родителей), внутреннем порядке (обход отдельной ветки – части дерева), а также обход в ширину (обход элементов на всех ветках дерева, располагающихся на определенном уровне вложенности. Например, если данную книгу представить в виде дерева, то при обходе в ширину читатель будет переходить к каждой главе и читать только раздел «Мотивация»). Каждый вид обхода дерева можно поддержать реализацией отдельного итератора.

- **Пустой итератор.**
  Пустой итератор `NullIterator` - это вырожденная форма итератора. Пустой итератор `NullIterator` всегда считает, что обход завершен, то есть его операция `IsDone` всегда возвращает истину (`true`). Применение пустого итератора может упростить обход древовидных структур. В каждой точке обхода мы запрашиваем у текущего элемента итератор для его потомков. Ветви дерева обычно, возвращают полноценный итератор, но листовые элементы могут возвращать пустой итератор `NullIterator`, что позволяет реализовать обход элементов дерева единообразно.

В языке C# пустой итератор реализуется при помощи связки операторов `yield break`. Предлагается рассмотреть создание пустого итератора на примере.

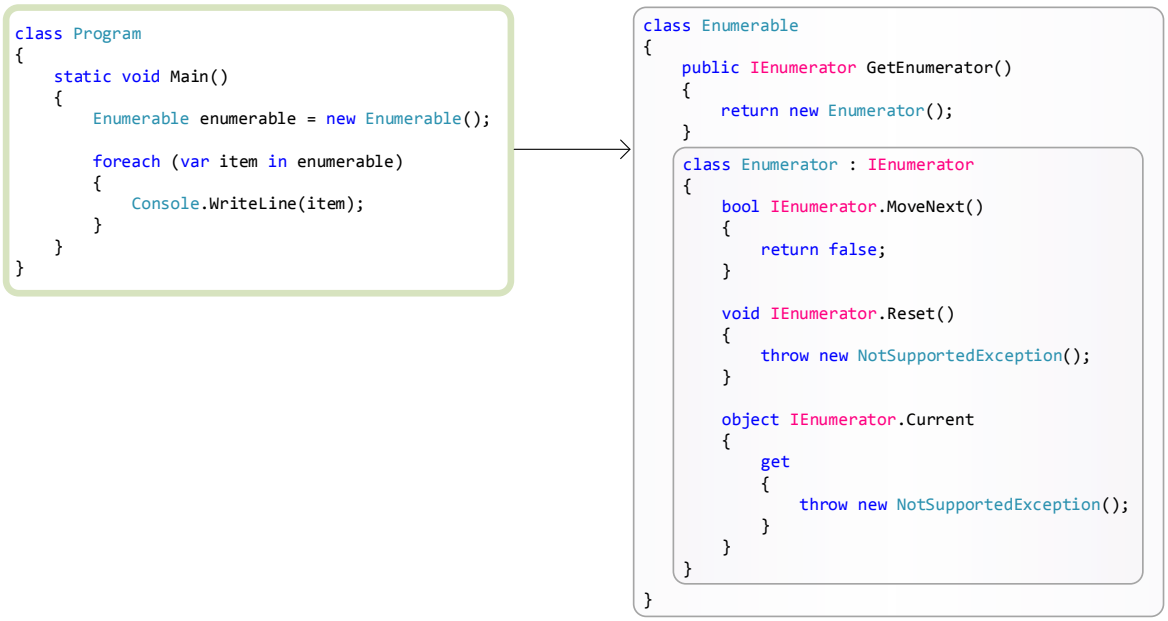


```
class Program
{
    static void Main()
    {
        Enumerable enumerable = new Enumerable();

        foreach (var item in enumerable)
        {
            Console.WriteLine(item);
        }
    }
}
```

```
class Enumerable
{
    public IEnumerator GetEnumerator()
    {
        yield break;
    }
}
```

Для того чтобы понять работу конструкции `yield break`, желательно посмотреть на код класса итератора, который генерируется конструкцией `yield break`. Для этого рекомендуется воспользоваться инструментом для обратной инженерии (*дизассемблирования*) – программой dotPeek от компании JetBrains известной как производителя программы ReSharper.

Для анализа потребуется открыть в программе dotPeek исполняемый файл *.exe содержащий код итератора. В результате дизассемблирования будет представлен следующий код *(для упрощения понимания дизассемблированный код был упрощен)*:



```
class Program
{
    static void Main()
    {
        Enumerable enumerable = new Enumerable();

        foreach (var item in enumerable)
        {
            Console.WriteLine(item);
        }
    }
}
```

```
class Enumerable
{
    public IEnumerator GetEnumerator()
    {
        return new Enumerator();
    }

    class Enumerator : IEnumerator
    {
        bool IEnumerator.MoveNext()
        {
            return false;
        }

        void IEnumerator.Reset()
        {
            throw new NotSupportedException();
        }

        object IEnumerator.Current
        {
            get
            {
                throw new NotSupportedException();
            }
        }
    }
}
```

Пустой итератор `Enumerator` всегда считает, что обход завершен, то есть его операция `MoveNext` всегда возвращает ложь (`false`).

Известные применения паттерна в .Net

Важно отметить, что классическое представление паттерна Iterator отличается от представления, рекомендуемого Microsoft. Но суть использования техники «перечисляемый-перечислитель» которая описывается паттерном Iterator при этом не меняется.

Паттерн Iterator, выражен в платформе .NET в виде техники создания составных объектов, основанных на технике «перечисляемый-перечислитель» коллекций, через реализацию интерфейсов `IEnumerable` и `IEnumerator`. Оператор автоматической генерации программного кода коллекций - `yield`, также выражает идею использования паттерна Iterator.

Практически во всех коллекциях, идущих в поставке Microsoft .NET Framework присутствуют реализации итераторов. Следует внимательно подходить к выбору класса коллекции, так как использование неправильного типа коллекции может привести к неоптимальной работе программы. В общем случае не следует использовать типы в пространстве имен `System.Collections`, если явным образом не используется версия 1.1 платформы .NET Framework. Во всех других случаях рекомендуется использовать универсальные (*generic*) и параллельные коллекции, так как они оптимизированы, обладают большей типобезопасностью и содержат дополнительные функциональные возможности.

Перед использованием той или иной системной коллекции, необходимо ответить на следующие вопросы:

- **Вопрос:** *Требуется ли использовать такой список, элементы которого сразу удаляются после получения их значения?*

  Ответ: Если да, то имеет смысл рассмотреть возможность использования универсального класса `Queue<T>`, если требуется работа по принципу (*FIFO*) или универсального класса `Stack<T>`, если требуется работа по принципу (*LIFO*). Для безопасного доступа из нескольких потоков следует использовать параллельные версии очереди или стека - классы `ConcurrentQueue<T>` или `ConcurrentStack<T>` соответственно.

- **Вопрос:** *Требуется ли получать доступ к элементам коллекции в определенном порядке* (*FIFO, LIFO*) *или в произвольным порядке?*

  Ответ: Если да, то рекомендуется использовать класс `Queue`, универсальный класс `Queue<T>` или универсальный класс `ConcurrentQueue<T>`, которые предоставляют доступ по принципу FIFO.
  Класс `Stack`, универсальный класс `Stack<T>` или универсальный класс `ConcurrentStack<T>` предоставляют доступ по принципу LIFO.
  Универсальный класс `LinkedList<T>` предоставляет последовательный доступ от начала списка к концу списка или наоборот.

- **Вопрос:** *Необходимо ли получать доступ к элементам коллекции по индексу?*

  Ответ: Если да, то рекомендуется использовать классы `ArrayList` и `StringCollection`, и универсальный класс `List<T>`, которые предоставляют доступ к своим элементам по индексу, начиная отсчет с нулевого индекса.
  Классы `Hashtable`, `SortedList`, `ListDictionary` и `StringDictionary`, а также универсальные классы `Dictionary<TKey, TValue>` и `SortedDictionary<TKey, TValue>` предоставляют доступ к своим элементам по ключу.
  Классы `NameObjectCollectionBase` и `NameValueCollection`, а также универсальные классы `KeyedCollection<TKey, TItem>` и `SortedList<TKey, TValue>` предоставляют доступ к своим элементам по индексу с отсчетом от нуля или по ключу.

- **Вопрос:** *Будет ли каждый элемент содержать только одно значение, сочетание из одного ключа и одного значения или сочетание из одного ключа и нескольких значений?*

  Ответ: Одно значение. Можно использовать любую из коллекций, основанных на интерфейсе `IList` или на универсальном интерфейсе `IList<T>`.

Один ключ и одно значение. Можно использовать любую из коллекций, основанных на интерфейсе `IDictionary` или на универсальном интерфейсе `IDictionary<TKey, TValue>`.
Одно значение с внедренным ключом. Можно использовать универсальный класс `KeyedCollection<TKey, TItem>`.
Один ключ и несколько значений. Можно использовать класс `NameValueCollection`.

- **Вопрос:** *Требуется ли сортировка элементов в порядке, отличном от того порядка в котором элементы поступают (добавляются) в коллекцию?*

  Ответ: Если да, то рекомендуется использовать класс `Hashtable`, который сортирует свои элементы по их хэш-коду.
  Класс `SortedList` и универсальные классы `SortedDictionary<TKey, TValue>` и `SortedList<TKey, TValue>` сортируют свои элементы по их ключам на основе реализации интерфейса `IComparer` и универсального интерфейса `IComparer<T>`.
  Класс `ArrayList` предоставляет метод `Sort`, который принимает реализацию IComparer в качестве параметра. Его универсальный аналог — универсальный класс `List<T>` предоставляет метод `Sort`, который принимает реализацию универсального интерфейса `IComparer<T>` в качестве параметра.

- **Вопрос:** *Необходимо ли организовать быстрый поиск по коллекции, а также быстрое помещение новых элементов в коллекцию и быстрое извлечение существующих элементов из коллекции?*

  Ответ: Если да, то рекомендуется использовать класс `ListDictionary`, так как он работает быстрее, чем `Hashtable` для небольших коллекций (10 элементов или меньше). Универсальный класс `Dictionary<TKey, TValue>` предоставляет более быстрый просмотр, чем универсальный класс `SortedDictionary<TKey, TValue>`. Многопоточной реализацией является класс `ConcurrentDictionary<TKey, TValue>`. Класс `ConcurrentBag<T>` предоставляет быструю многопоточную вставку для неупорядоченных данных.

- **Вопрос:** *Требуется ли использовать коллекцию только для хранения строк?*

  Ответ: Если да, то рекомендуется использовать классы `StringCollection` (основанный на `IList`) и `StringDictionary` (основанный на `IDictionary`) которые находятся в пространстве имен `System.Collections.Specialized`.
  Кроме того, можно использовать любой из универсальных классов коллекций из пространства имен `System.Collections.Generic` как строго типизированную строковую коллекцию, указав класс `String` в качестве параметра типа.

Паттерн Mediator

Название

Посредник

Также известен как

-

Классификация

По цели: поведенческий
По применимости: к объектам

Частота использования

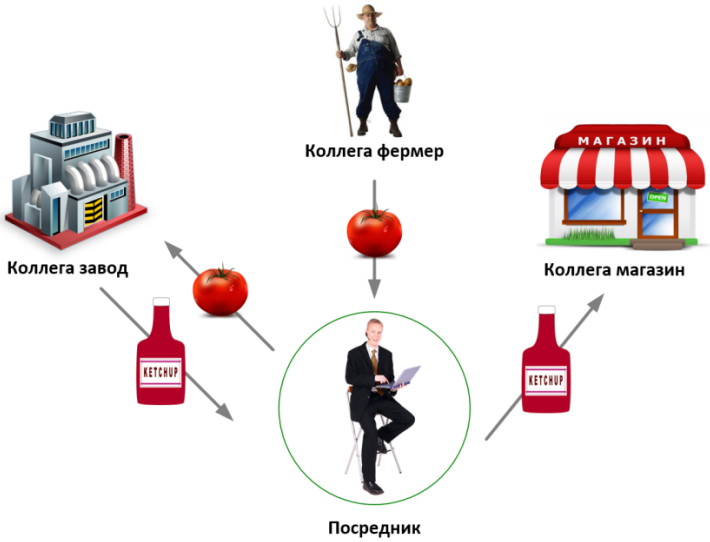
Ниже средней - 1 **2** 3 4 5

Назначение

Паттерн Mediator – предоставляет объект-посредник, скрывающий способ взаимодействия множества других объектов-коллег. Объект-посредник обеспечивает слабую связанность системы, избавляя объектов-коллег от необходимости явно ссылаться друг на друга, позволяя тем самым независимо изменять взаимодействия между объектами-коллегами.

Введение

Предлагается рассмотреть пример работы человека-посредника в объективной реальности с точки зрения его полезности, а не с точки зрения извлечения им прибыли. На рисунке ниже показана роль посредника в процессе производства кетчупа. Посредник покупает свежие помидоры у фермера, отправляет эти помидоры на консервный завод для переработки в кетчуп, а полученный кетчуп оптом продает в магазин розничной торговли.



Достоинство такого подхода заключается в том, что фермеру не требуется знать о заводе по переработке помидоров и в какой продукт будет «превращен» выращенный им помидор. Заводу не требуется заботиться о сбыте своей продукции так как посредник сам позаботится о поиске подходящего магазина розничной торговли для того чтобы товар донести до покупателя. Каждый занимается своим делом.

Предлагается выразить «*программно*» пример с фермером, фабрикой, магазином и посредником.

```
class Program
{
    static void Main()
    {
        var mediator = new ConcreteMediator();
        var farmer = new Farmer(mediator);
        var cannery = new Cannery(mediator);
        var shop = new Shop(mediator);

        mediator.Farmer = farmer;
        mediator.Cannery = cannery;
        mediator.Shop = shop;

        farmer.GrowTomato();
    }
}
```

```
abstract class Mediator
{
    public abstract void Send(string msg,
                      Colleague colleague);
}
```

```
abstract class Colleague
{
    protected Mediator mediator;

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

```
class Farmer : Colleague
{
    public Farmer(Mediator mediator)
        : base(mediator)
    {
    }

    public void GrowTomato()
    {
        string tomato = "Tomato";
        Console.WriteLine(this.GetType().Name
            + " raised " + tomato);
        mediator.Send(tomato, this);
    }
}
```

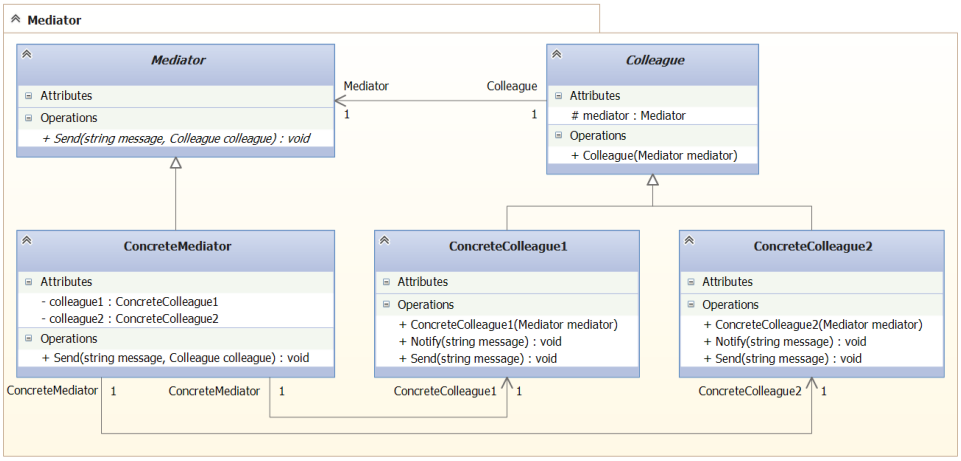
```
class ConcreteMediator : Mediator
{
    public Farmer Farmer { get; set; }
    public Cannery Cannery { get; set; }
    public Shop Shop { get; set; }

    public override void Send(string message,
                      Colleague colleague)
    {
        if (colleague == Farmer)
        {
            Cannery.MakeKetchup(message);
        }
        else if (colleague == Cannery)
        {
            Shop.SellKetchup(message);
        }
    }
}
```

```
class Cannery : Colleague
{
    public Cannery(Mediator mediator)
        : base(mediator)
    {
    }

    public void MakeKetchup(string message)
    {
        string ketchup = message + "Ketchup";
        Console.WriteLine(this.GetType().Name
            + " produced " + ketchup);
        mediator.Send(ketchup, this);
    }
}
```

```
class Shop : Colleague
{
     public Shop(Mediator mediator)
        : base(mediator)
    {
    }

    public void SellKetchup(string message)
    {
        Console.WriteLine(this.GetType().Name
            + " sold " + message);
    }
}
```

См. Пример к главе: \017_Mediator\001_Mediator

Структура паттерна на языке UML



См. Пример к главе: \017_Mediator\001_Mediator

Структура паттерна на языке C#

```
class Program
{
    static void Main()
    {
        var mediator = new ConcreteMediator();
        var colleague1 = new ConcreteColleague1(mediator);
        var colleague2 = new ConcreteColleague2(mediator);

        mediator.Colleague1 = colleague1;
        mediator.Colleague2 = colleague2;
        colleague1.Send("Message 1");
        colleague2.Send("Message 2");
    }
}
```

```
abstract class Mediator
{
    public abstract void Send(string msg, Colleague colleague);
}
```

```
class ConcreteMediator : Mediator
{
    public ConcreteColleague1 Colleague1 { get; set; }
    public ConcreteColleague2 Colleague2 { get; set; }

    public override void Send(string msg, Colleague colleague)
    {
        if (Colleague1 == colleague)
            Colleague2.Notify(msg);
        else
            Colleague1.Notify(msg);
    }
}
```

```
abstract class Colleague
{
    protected Mediator mediator;

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

```
class ConcreteColleague1 : Colleague
{
    public ConcreteColleague1(Mediator mediator)
        : base(mediator)
    {
    }

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

    public void Notify(string message)
    {
        Console.WriteLine(message);
    }
}
```

```
class ConcreteColleague2 : Colleague
{
    public ConcreteColleague2(Mediator mediator)
        : base(mediator)
    {
    }

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

    public void Notify(string message)
    {
        Console.WriteLine(message);
    }
}
```

См. Пример к главе: \017_Mediator\001_Mediator

Участники

- `Mediator` - **Посредник**:
  Предоставляет интерфейс для организации процесса по обмену информацией между объектами типа `Colleague`.

- `ConcreteMediator` - **Конкретный посредник**:
  Реализует алгоритм взаимодействия между объектами-коллегами.

- `Colleague` - **Коллега**:
  Предоставляет интерфейс для организации процесса взаимодействия объектов-коллег с объектом типа `Mediator`.

- `ConcreteColleague` - **Конкретный коллега**:
  Каждый объект-коллега знает только об объекте-медиаторе. Все объекты-коллеги обмениваются информацией только через посредника (медиатора).

Отношения между участниками

Отношения между классами

- Класс `ConcreteMediator` связан связью отношения наследования с абстрактным классом `Mediator` и связями отношения ассоциации с классами `ConcreteColleague`.

- Абстрактный класс `Colleague` связан связью отношения ассоциации с абстрактным классом `Mediator`.

- Конкретные классы `ConcreteMediator` связаны связью отношения наследования с абстрактным классом `Colleague`.

Отношения между объектами

- Объекты-коллеги (`ConcreteColleague`) посылают запросы объекту-посреднику (`ConcreteMediator`), а он в свою очередь перенаправляет полученные запросы другим объектам-коллегам, тем самым организуя процесс взаимодействия между объектами-коллегами.

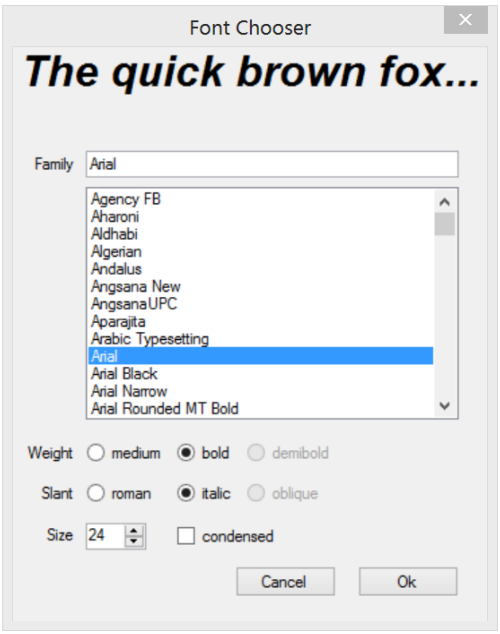
Мотивация

Объектно-ориентированное проектирование предоставляет возможности распределять поведение (функциональность) между различными объектами. При этом между имеющимися объектами может возникать очень много связей отношений, каждый объект (в худшем случае) может ссылаться на множество других объектов, тем самым превращая представление связей отношений в сложно понимаемую и сложно сопровождаемую сеть из связей. Если между объектами связей отношений много, то такая система превращается в монолит и объектно-ориентированный подход к разработке не даст никаких преимуществ. В системах со множеством связей между объектами, очень сложно вносить изменения, поскольку функциональность распределена между многими объектами и соответственно работа одних объектов зависит от функциональных возможностей других объектов. Часто при попытке внесения изменений в определенный класс приходится вносить изменения сразу в несколько других классов.

Использование подхода, предлагаемого паттерном Mediator позволит устранить сильную связанность между объектами при этом упростить правила взаимодействия между объектами, что соответственно сделает систему проще для понимания, сопровождения и расширения.

Предлагается рассмотреть реализацию диалогового окна выбора и настройки шрифта. В диалоговом окне может располагаться ряд элементов управления: кнопки (`Button`), поля ввода (`TextBox`) и т.д.

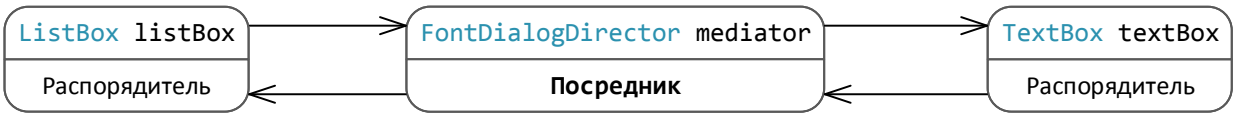
Часто между разными элементами управления в диалоговом окне имеется много зависимостей (одни элементы обращаются к другим). Например, если поле ввода пустое, то определенная кнопка может быть недоступна «загреена», а если в поле ввести текст, кнопка становится активной. На рисунке ниже показано, что при выборе элемента списка, изменяется содержимое поля ввода (выбрав в списке шрифт Arial, имя этого шрифта отображается в поле ввода). И наоборот, ввод текста в поле может автоматически привести к выбору элемента списка. Этот пример показывает, что некоторые элементы управления диалогового окна взаимодействуют друг с другом.



В разных диалоговых окнах взаимодействие между элементами управления может отличаться. Несмотря на то что во всех окнах используется элементы управления из стандартного набора (представленных в окне Toolbox, в Visual Studio), просто взять и использовать существующие классы элементов управления без дополнительной настройки взаимодействия не получится. Индивидуальная настройка каждого элемента управления для организации его взаимодействия с другими элементами управления часто приводит к запутанной логике взаимодействий, особенно когда взаимодействующих элементов управления много.

Проблем со сложностью организации взаимодействия между элементами управления можно избежать, если реализацию логики взаимодействия вынести в отдельный объект-посредник, с которым будут взаимодействовать все элементы управления. Использование объекта-посредника избавит элементы управления от необходимости ссылаться друг на друга. Элементы управления будут взаимодействовать только с объектом-посредником, который будет перенаправлять запросы от одного элемента управления другому.

Например, экземпляр класса `FontDialogDirector` может выступать в качестве посредника между элементами управления в диалоговом окне. Экземпляр класса `FontDialogDirector` «знает» обо всех элементах управления, используемых в контексте диалогового окна и руководит взаимодействием между этими элементами управления, другими словами выполняет функции центра взаимодействий.



См. Пример к главе: \017_Mediator\002_Motivation_Mediator

Применимость паттерна

Паттерн Mediator рекомендуется использовать, когда:

- Имеется некоторое количество объектов, связанных друг с другом при этом связи между этими объектами запутаны и сложны для понимания.

- Возникают трудности повторного использования объекта из-за его тесных связей с другими объектами.

Результаты

Использование паттерна Mediator предоставляет следующие возможности:

- **Уменьшает число создаваемых подклассов.**
  Класс `ConcreteMediator` собирает в себе все поведение (логику взаимодействия между коллегами), которое в противном случае пришлось бы распределять между объектам-коллегами. Для изменения взаимодействия между коллегами потребуется создать производный класс от класса `Mediator`, при этом классы коллег `ConcreteColleague` можно использовать повторно без внесения в них изменений.

- **Устранение сильной связанности между коллегами.**
  Класс `Mediator` обеспечивает слабую связанность между коллегами. Изменять классы посредников `ConcreteMediator` и коллег `ConcreteColleague` можно независимо друг от друга.

- **Упрощение правил (протокола и церемониала) взаимодействия между объектами коллегами.**
  Класс `Mediator` заменяет технику взаимодействия «*все со всеми*» техникой «*один со всеми*», то есть один посредник взаимодействует со всеми коллегами. Отношения «*один ко многим*» проще для понимания, сопровождения и расширения. Коллеги ничего не знают друг о друге. Отношение «*все со всеми*» представляют сложную запутанную сеть, которую сложно анализировать и сопровождать.

- **Алгоритмическое управление связями отношений между коллегами.**
  Механизм «*посредничества*», который расположен внутри объекта-посредника (`ConcreteMediator`), позволяет программисту сосредоточиться именно на взаимодействии объектов-коллег (в общем) а не на их индивидуальных отношениях друг с другом. Посредник использует технику алгоритмического управления связями отношений между объектами, самостоятельно переадресовывая запросы от одного коллеги другому.

- **Централизация управления отношениями между коллегами.**
  Класс (`ConcreteMediator`) сосредотачивает в себе сложные отношения и взаимодействия между коллегами (`ConcreteColleague`). Так как посредник содержит в себе алгоритмическое представление правил взаимодействия между коллегами, то сам посредник может оказаться сложнее отдельных коллег. В результате посредник может оказаться сложным монолитом, что затрудняет его сопровождение.

Реализация

Полезные приемы реализации паттерна Mediator:

- **Отказ от использования абстрактного класса `Mediator`.**
  Если в программной подсистеме коллеги взаимодействуют только с одним посредником (`ConcreteMediator`), то можно не создавать базовый абстрактный класс `Mediator`. Но использование абстрактного класса `Mediator` позволит коллегам работать с разными классами `ConcreteMediator` благодаря технике абстрагирования вариантов использования.

- **Обмен информацией между посредником и коллегами.**
  Коллеги должны обмениваться информацией с посредником тогда, когда возникает надобность послать сообщение об изменении состояния коллеги. Общаясь с посредником, коллега передает ссылку на себя в качестве аргумента вызываемого метода на посреднике, тем самым давая возможность посреднику определить отправителя сообщения и выбрать коллегу-получателя, которому переадресовать сообщение, полученное от коллеги-отправителя.

Пример кода

См. Пример к главе: \017_Mediator\002_Motivation_Mediator

Паттерн Memento

Название

Хранитель

Также известен как

Token (Лексема)

Классификация

По цели: поведенческий
По применимости: к объектам

Частота использования

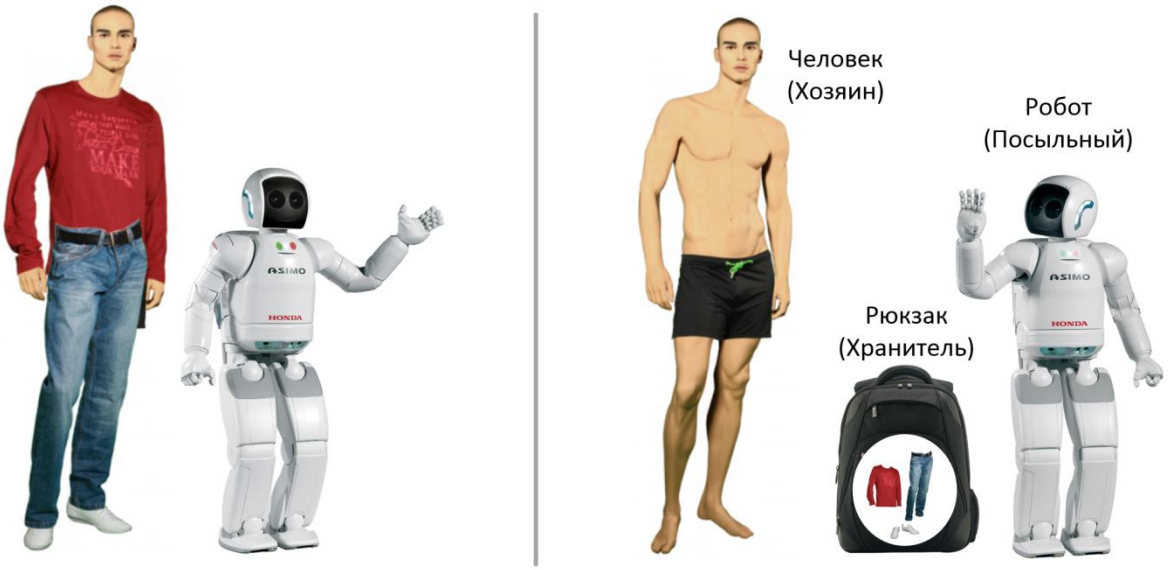
Низкая - **1** 2 3 4 5

Назначение

Паттерн Memento - не нарушая инкапсуляции, фиксирует и выносит за пределы объекта-хозяина его внутреннее состояние так, чтобы позднее это вынесенное состояние можно было восстановить в исходном объекте-хозяине.

Введение

Воспользуемся метафорой для описания работы паттерна Memento. Представьте себе, что Вы приобрели робота **ASIMO** и отправились с ним на прогулку, дошли до городского пляжа и решили искупаться. Вы сняли с себя одежду, сложили одежду в рюкзак, отдали рюкзак роботу, а сами пошли в воду. Задача робота – стеречь рюкзак. Инженеры **HONDA** не пока еще научили **ASIMO** открывать рюкзаки, поэтому он не сможет получить доступ к вещам и, например, поменять Ваши джинсы на новый сервопривод или аккумулятор.



Поплавав, Вы вышли из воды на берег, взяли рюкзак у робота, достали из рюкзака одежду и одели одежду на себя.

Рассмотрим основных участников паттерна Memento:

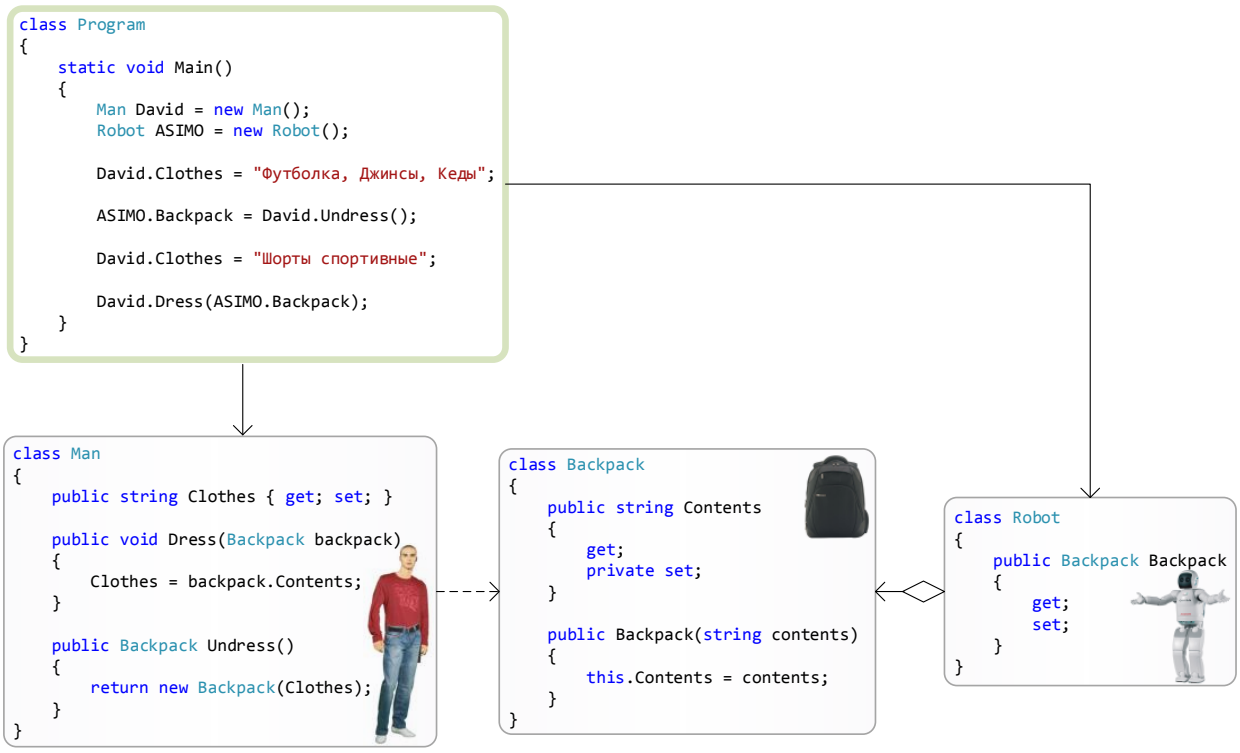
Важно помнить, что состояние объекта – это набор значений полей этого объекта. Одежда человека может входить в понятие одной из разновидностей состояний человека, так как одежду можно описать в виде набора переменных, принадлежащих объекту-человеку. И зачастую, когда нас просят описать человека, то мы помимо биометрических данных, зачастую указываем фасон и цвет одежды в которую одет человек.

Человек (Man) – хозяин (сложный объект со сложной конфигурацией состояния), которому по ряду причин потребовалось изменить свое состояние «из одетого в раздетого», а потом восстановить свое состояние обратно в первоначальный вид «из раздетого в одетого».

Рюкзак (Backpack) – хранитель (объект для хранения в себе копии состояния хозяина). Рюкзак с одеждой (состоянием хозяина) переходит из рук хозяина в руки робота, а позднее обратно из рук робота в руки хозяина. Это удобно, потому что возможно у робота есть еще и сумка с предыдущим состоянием хозяина, например, со смокингом со вчерашнего посещения оперы.

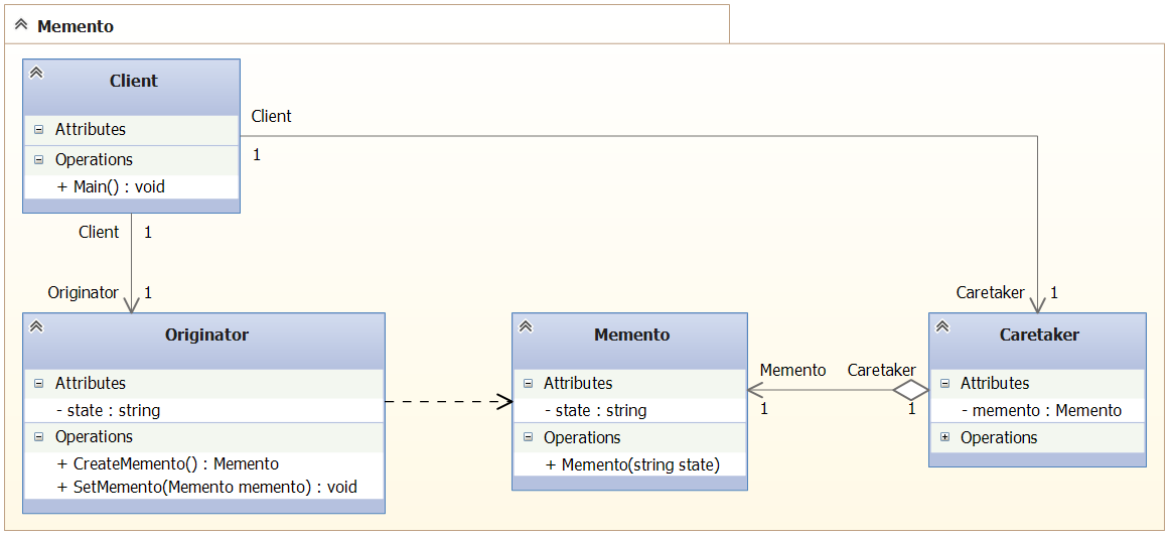
Робот (Robot) – посыльный (объект для ношения рюкзаков и сумок с состояниями хозяина) в любой момент готов дать хозяину нужную сумку или рюкзак с определенным набором вещей (предыдущие состояния). Человеку не требуется заботиться о поиске нужной конфигурации состояния, за него всю заботу за конфигурацию и хранение состояний берет на себя робот.

Из сказанного выше легко сформировать модель и реализовать ее программно.



См. Пример к главе: \018_Memento\002_Memento

Структура паттерна на языке UML



См. Пример к главе: \018_Memento\001_Memento

Структура паттерна на языке C#

```
class Program
{
    static void Main()
    {
        Originator originator = new Originator();
        originator.State = "On";

        Caretaker caretaker = new Caretaker();
        caretaker.Memento = originator.CreateMemento();

        originator.State = "Off";

        originator.SetMemento(caretaker.Memento);
    }
}
```

```
class Originator
{
    public string State { get; set; }

    public void SetMemento(Memento memento)
    {
        State = memento.State;
    }

    public Memento CreateMemento()
    {
        return new Memento(State);
    }
}
```

```
class Memento
{
    public string State
    {
        get;
        private set;
    }

    public Memento(string state)
    {
        this.State = state;
    }
}
```

```
class Caretaker
{
    public Memento Memento
    {
        get;
        set;
    }
}
```

См. Пример к главе: \018_Memento\001_Memento

Участники

- `Memento` - **Хранитель**:
  Хранит в себе состояние объекта-хозяина класса `Originator`. Размер хранимого состояния может быть разный и зависит от потребностей хозяина. Объект-хранитель должен запрещать кому бы то ни было, кроме объекта-хозяина получать доступ к хранимому состоянию.

- `Originator` - **Хозяин**:
  Создает объекта-хранителя и помещает в него свое текущее внутреннее состояние. Впоследствии использует объекта-хранителя для восстановления своего ранее сохраненного состояния.

- `Caretaker` - **Посыльный**:
  Отвечает за сохранность объекта-хранителя и не производит над ним никаких действий, и не исследует его внутреннее содержимое.

Отношения между участниками

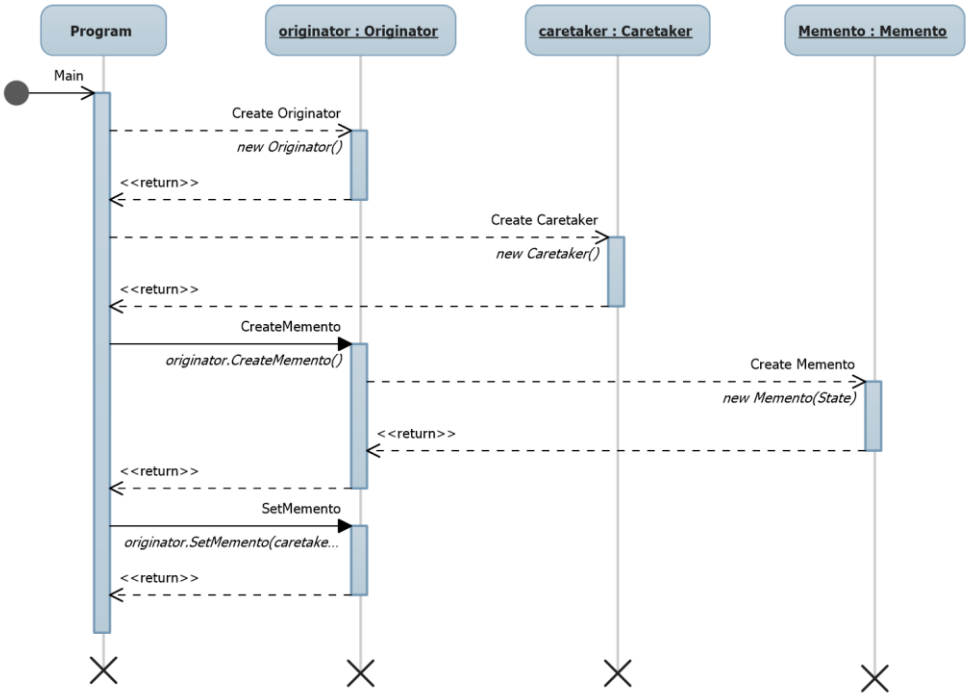
Отношения между классами

- Класс `Originator` связан связью отношения зависимости с классом `Memento`.

- Класс `Caretaker` связан связью отношения агрегации с классом `Memento`.

Отношения между объектами

- Объект-посыльный (`Caretaker`) получает от объекта-хозяина (`Originator`) объект-хранитель (`Memento`), некоторое время держит его у себя, а затем возвращает объекту-хозяину.



См. Пример к главе: \018_Memento\001_Memento

Мотивация

Пример, дающий возможность представить потребность в использовании паттерна и увидеть способ применения паттерна.

Применимость паттерна

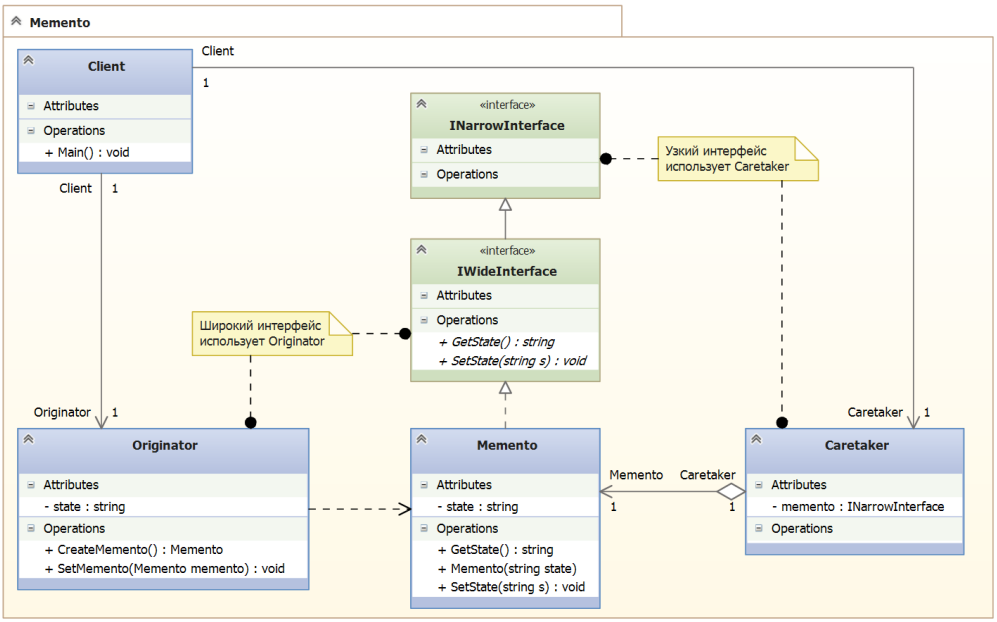
Паттерн Memento рекомендуется использовать, когда:

Необходимо сохранить состояние объекта-хозяина (или только часть состояния), чтобы позднее хозяина можно было восстановить в исходном состоянии.

Реализация

Полезные приемы реализации паттерна Memento:

- **Наличие «узкого» и «широкого» интерфейсов.**
  Объект-хранитель должен запрещать кому бы то ни было, кроме объекта-хозяина получать доступ к хранимому состоянию. Объекту-посыльному класса `Caretaker` может быть доступен только «*узкий*» интерфейс хранителя, а объекту-хозяину может быть доступен «*широкий*» интерфейс, через который объект-хозяин может получить доступ к хранящимся данным.



См. Пример к главе: \018_Memento\003_NarrowAndWideInterfaces

Паттерн Observer

Название

Наблюдатель

Также известен как

Dependents (Подчиненные), Publisher-Subscriber (Издатель-Подписчик)

Классификация

По цели: поведенческий
По применимости: к объектам

Частота использования

Высокая - 1 2 3 4 **5**

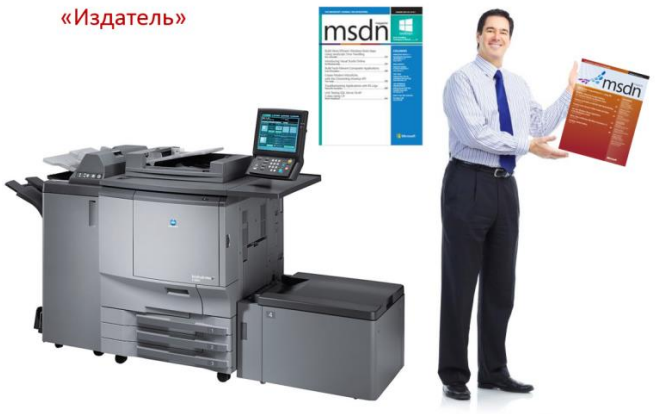
Назначение

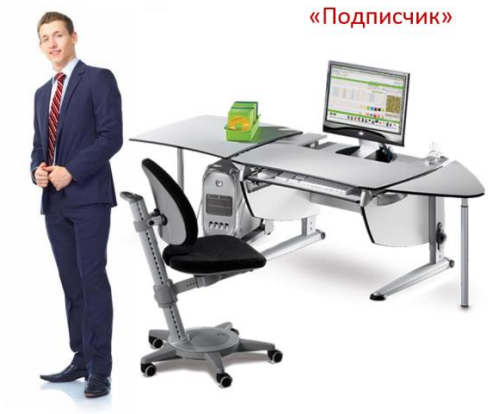
Паттерн Observer – использует связь отношения зависимости «один ко многим» (один издатель ко многим подписчикам). При изменении состояния одного объекта (издателя), все зависящие от него объекты (подписчики) оповещаются об этом и автоматически обновляются.

Введение

Паттерн Observer описывает использование важной техники ООП - «Издатель-Подписчик», другими словами, паттерн Observer описывает правильные способы организации процесса подписки на определенные события.

Кто такие издатель и подписчик в объективной реальности? Издателем может быть издательский центр - ***Microsoft*** **Press** который издает журнал «msdn magazine», а подписчиком может быть программист подписавшийся на данный журнал.



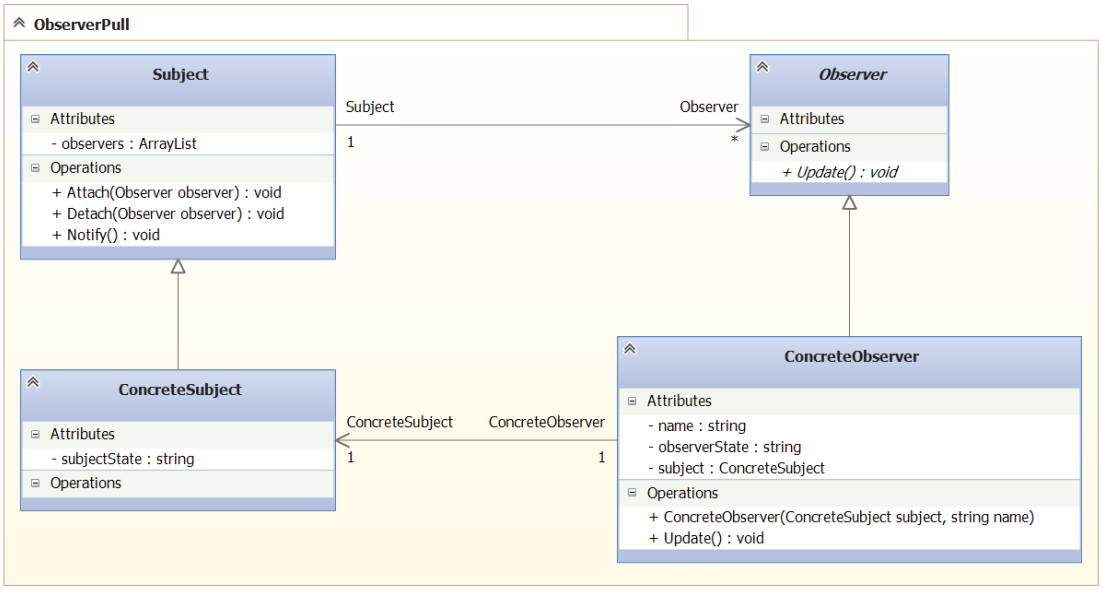


После того как подписчик подписался на журнал, подписчик ожидает пока издатель издаст журнал и оповестит об этом подписчика. Имеется два способа получения подписчиком журнала. Первый способ - «*метод вытягивания*»: После получения уведомления от издателя о том, что журнал выпущен, подписчик должен пойти к издателю и забрать (*вытянуть*) журнал самостоятельно. Второй способ – «*метод проталкивания*»: Издатель не уведомляет подписчика о выпуске журнала, а самостоятельно или через почту доставляет журнал подписчику и, например, бросает (*проталкивает*) журнал в почтовый ящик.

См. Пример к главе: \019_Observer\ 004_MSDN Magazine

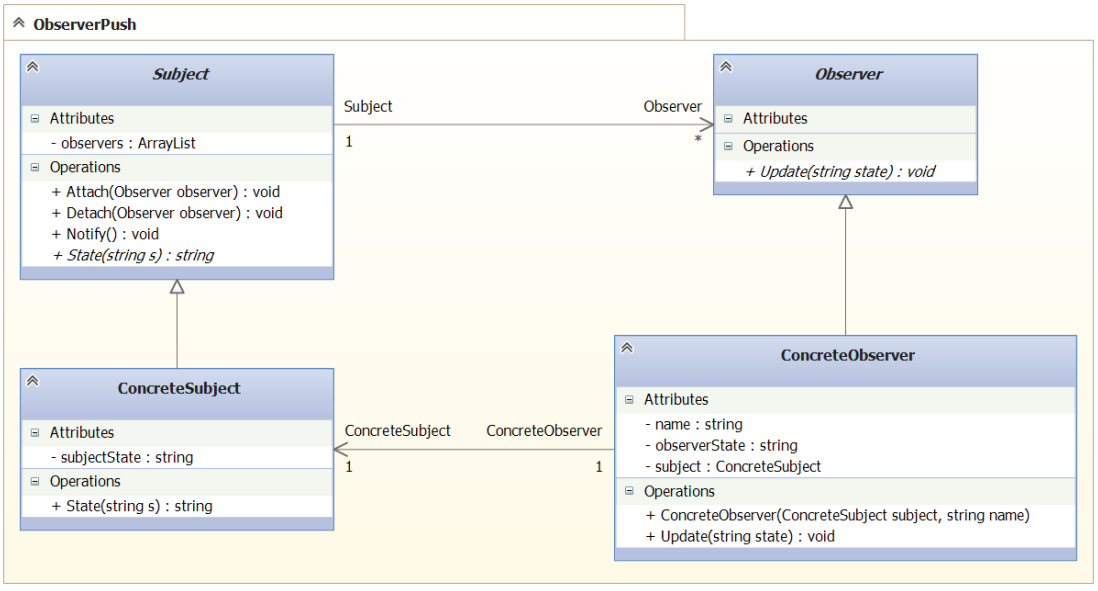
Структура паттерна на языке UML

Модель вытягивания (Pull model)



См. Пример к главе: \019_Observer\001_Observer [project ObserverPull]

Модель проталкивания (Push model)

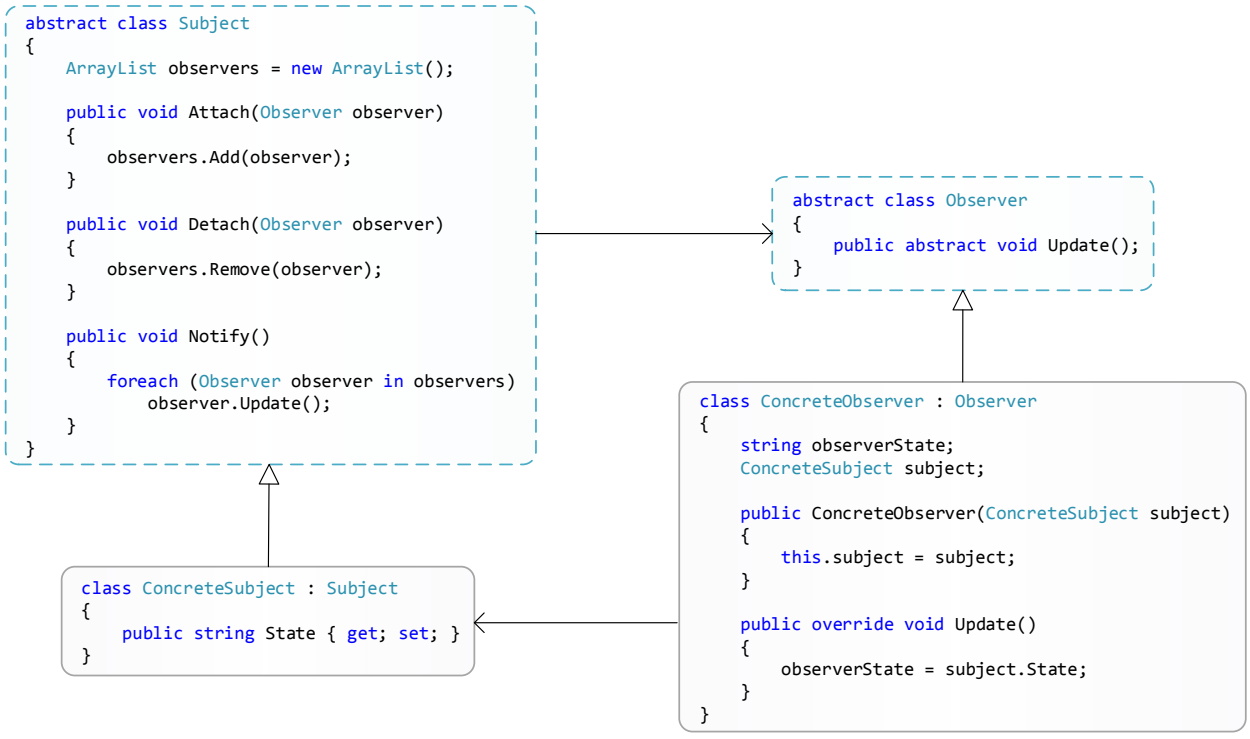


См. Пример к главе: \019_Observer\001_Observer [project ObserverPush]

Структура паттерна на языке C#

Модель вытягивания (Pull model)

```
class Program
{
    static void Main()
    {
        ConcreteSubject subject = new ConcreteSubject();
        subject.Attach(new ConcreteObserver(subject));
        subject.Attach(new ConcreteObserver(subject));
        subject.State = "Some State ...";
        subject.Notify();
    }
}
```



См. Пример к главе: \019_Observer\001_Observer [project ObserverPull]

Модель проталкивания (Push model)

```
class Program
{
    static void Main()
    {
        ConcreteSubject subject = new ConcreteSubject();
        subject.Attach(new ConcreteObserver(subject));
        subject.Attach(new ConcreteObserver(subject));
        subject.State = "Some State ...";
        subject.Notify();
    }
}
```

```
abstract class Subject
{
    ArrayList observers = new ArrayList();

    public void Attach(Observer observer)
    {
        observers.Add(observer);
    }

    public void Detach(Observer observer)
    {
        observers.Remove(observer);
    }

    public override string State { get; set; }

    public void Notify()
    {
        foreach (Observer observer in observers)
            observer.Update(State);
    }
}
```

```
abstract class Observer
{
    public abstract void Update(string state);
}
```

```
class ConcreteSubject : Subject
{
    public override string State { get; set; }
}
```

```
class ConcreteObserver : Observer
{
    string observerState;
    ConcreteSubject subject;

    public ConcreteObserver(ConcreteSubject subject)
    {
        this.subject = subject;
    }

    public override void Update(string state)
    {
        observerState = state;
    }
}
```

См. Пример к главе: \019_Observer\001_Observer [project ObserverPush]

Участники

- `Subject` - **Субъект** *(издатель)*:
  Издатель содержит ссылки на своих подписчиков и предоставляет интерфейс (набор методов) для добавления и удаления подписчиков. На издателя может ссылаться любое число подписчиков.

- `Observer` - **Наблюдатель** *(подписчик)*:
  Подписчик предоставляет интерфейс (набор методов) для обновления своего состояния при изменении состояния издателя.

- `ConcreteSubject` - **Конкретный субъект** *(конкретный издатель)*:
  Конкретный издатель посылает уведомление своим подписчикам и передает им свое состояние.

- `ConcreteObserver` - **Конкретный наблюдатель** *(конкретный подписчик)*:
  Реализует интерфейс обновления, предоставляемый абстрактным классом `Observer` и поддерживает согласованность состояния с издателем.

Отношения между участниками

Отношения между классами

- Абстрактный класс `Subject` связан связью отношения ассоциации с абстрактным классом `Observer`.

- Конкретный класс `ConcreteSubject` связан связью отношения наследования с абстрактным классом `Subject`.

- Конкретный класс `ConcreteObserver` связан связью отношения наследования с абстрактным классом `Observer` и связью отношения ассоциации с конкретным классом `ConcreteSubject`.

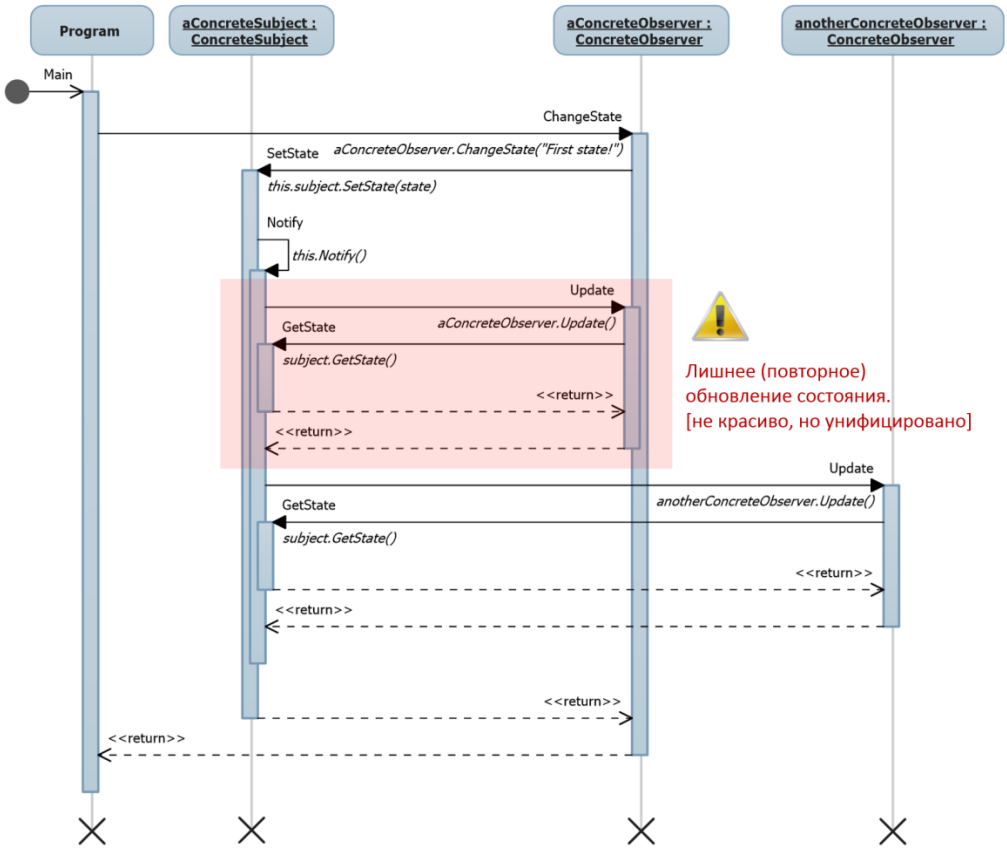
Отношения между объектами

- `ConcreteSubject` (издатель) уведомляет своих наблюдателей (подписчиков) о любом изменении, которое могло бы привести к рассогласованности состояний издателя и подписчика.

- `ConcreteObserver` (подписчик) после получения от `ConcreteSubject` (издателя) уведомления об изменении состояния, может запросить у издателя дополнительную информацию для полного согласования своего состояния.

- Метод `Notify` класса `ConcreteSubject` (издателя) может быть вызван не только издателем самостоятельно, также этот метод могут вызывать и подписчики, и посторонние клиенты.

- Следует отметить, что часто для сохранения унификации способа уведомления подписчиков, приходится повторно уведомлять того подписчика, который передал издателю свое новое состояние. На диаграмме взаимодействий, показан пример, когда у подписчика изменилось состояние, и этот подписчик сообщил издателю об этом. После получения от подписчика уведомления о новом состоянии, издатель начинает обновлять состояние каждого из подписчиков, которые ему известны в том числе и состояние того подписчика, который сам только что передал издателю свое новое состояние.



См. Пример к главе: \019_Observer\005_Observer

Попытка избавиться от дублирования обновления состояния подписчика-инициатора приведет к потере унификации работы с подписчиками, придётся организовывать логику по выявлению (определению) подписчика-инициатора изменения состояния, чтобы потом его определенным образом на время посылки уведомлений исключить из списка уведомляемых подписчиков, что добавит сложности в подсистему.

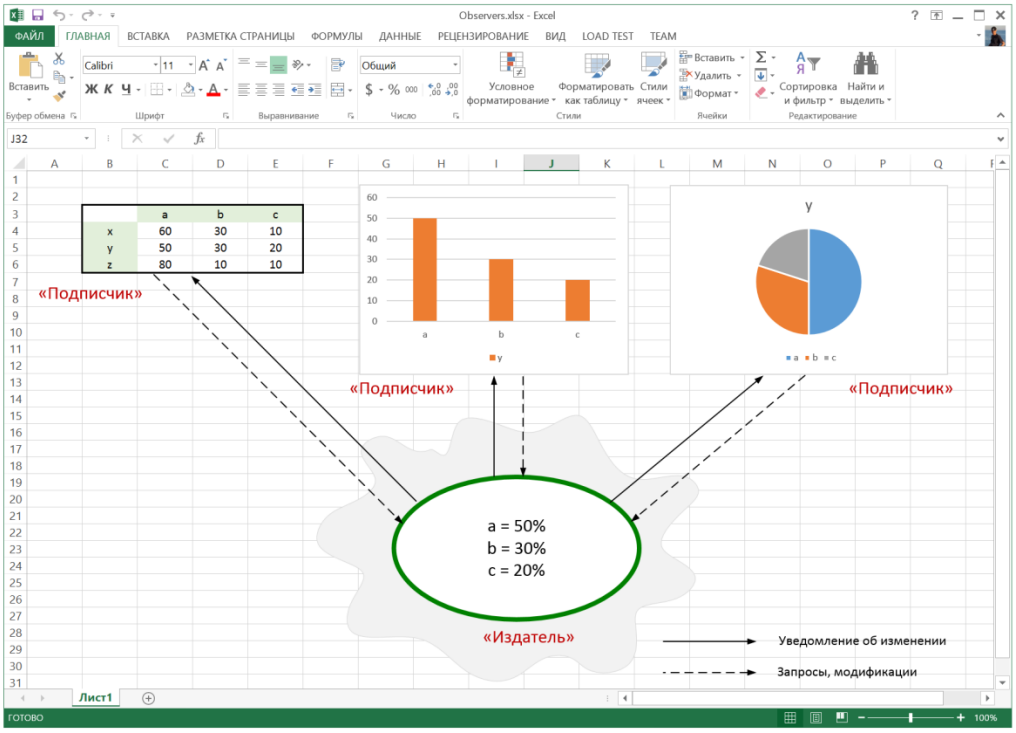
В каждом отдельном случае нужно рассмотреть подход с дублированием обновления состояния подписчика-инициатора. Если обновление стоит не дорого, то проще обновить повторно состояние подписчика-инициатора и сохранить при этом унификацию механизма уведомлений и простоту чтения программы. Если обновление стоит дорого, то следует организовать логику по исключению подписчика-инициатора из текущего сеанса обновления состояния подписчиков.

Мотивация

Часто в ходе разработки программной системы, состоящей из множества классов, требуется организовать поддержку согласования (синхронизации) состояния взаимосвязанных объектов. При организации согласованности между объектами, рекомендуется обеспечить слабую связанность (*Low Coupling*) используемых классов, так как такой подход позволит увеличить гибкость и степень повторного использования элементов системы.

Для построения программной системы чаще всего используется многослойная архитектура, в которой «презентационные аспекты» (*Presentation Layer*) отделены от «аспектов данных» (*Data Layer*) и «бизнес сущностей» (*Business Layer*). Это значит, что классы элементов управления GUI и классы, относящиеся к бизнес логике будет располагаться в разных слоях программной системы и соответственно эти классы можно изменять независимо друг друга и работать с ними автономно.

Например, на рисунке ниже показано, что объект – «электронная таблица» и объект – «диаграмма» ничего не знают друг о друге (не ссылаются друг на друга), поэтому их можно использовать по отдельности.



Когда пользователь работает с электронными таблицами, все изменения сразу же отражаются на диаграммах, и пользователю может показаться что все объекты взаимодействуют друг с другом напрямую. Но на самом деле взаимодействие между объектами «Подписчиками/Наблюдателями» происходит через объект «Издатель/Субъект». При таком подходе, все подписчики (электронная таблица и диаграммы) зависят от издателя (Субъекта). Соответственно, если изменяется состояние одного из подписчиков, этот подписчик уведомляет о своем изменении издателя, а издатель в свою очередь уведомляет всех остальных подписчиков, тем самым приводя согласованности состояния всех подписчиков. При этом нет ограничения на количество подписчиков и для работы с одними данными (a = 50%, b = 30%, c = 20%) может существовать любое число пользовательских интерфейсов (например, диаграмм-подписчиков).

Паттерн Observer описывает способы организации отношений по принципу «издатель-подписчик». Ключевыми объектами в схеме паттерна Observer являются объект «издатель/субъект» и объект «подписчик/наблюдатель». У издателя может быть сколько угодно зависимых от него подписчиков. Все подписчики уведомляются об изменении состояния издателя и синхронизируют с издателем свое состояние. Издатель (субъект) отправляет уведомления, не делая предположений об устройстве и внутренней структуре подписчиков.

Применимость паттерна

Паттерн Observer рекомендуется использовать, когда:

- У абстракции (подсистемы) имеется два аспекта (например, презентационный аспект и бизнес составляющая), при этом один из аспектов зависит от другого. Выражение этих аспектов в разных объектах, позволяет изменять и повторно использовать эти объекты независимо друг от друга.

- При изменении состояния одного объекта требуется изменить состояние других объектов, при этом заранее может быть не известно, сколько объектов нужно изменить.

- Один объект должен оповещать другие объекты, при этом не делая никаких предположений об уведомляемых объектах, другими словами, требуется добиться слабой связанности между объектами.

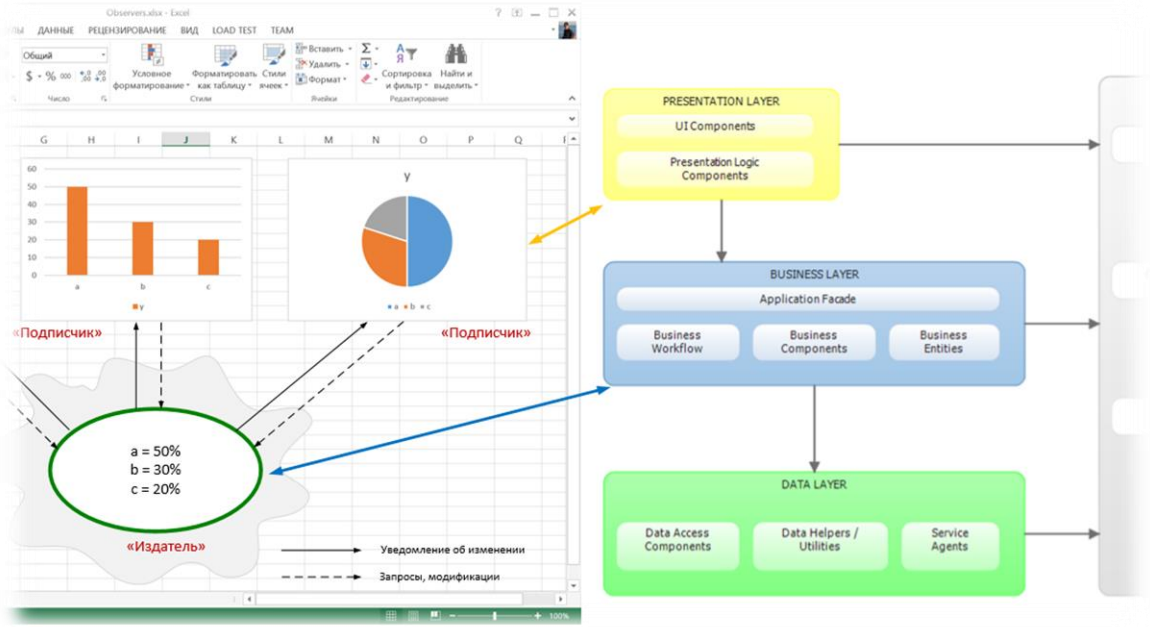
Результаты

Паттерн Observer позволяет изменять субъекты (*издателей*) и наблюдателей (*подписчиков*) независимо друг от друга. Издателей разрешается использовать повторно без участия подписчиков, и наоборот. Такой подход дает возможность добавлять новых подписчиков без внесения изменений в код издателя и других подписчиков.

Паттерн Observer обладает следующими преимуществами:

- **Абстрактная связанность издателя и подписчика.**
  Издатель (*субъект*) знает только о том, что у него имеется ряд подписчиков (*наблюдателей*), каждый из которых имеет интерфейс взаимодействия типа `Observer`. Издателю неизвестны конкретные классы подписчиков. Таким образом связи отношений между издателем и подписчиками имеют абстрактный характер (выражены через использование полей типа абстрактных классов).

  В связи с тем, что издатель и его подписчики не являются сильно связанными, то они могут находиться в разных функциональных слоях системы (например, издатель в бизнес слое «*Business Layer*», а подписчики в слое представления «*Presentation Layer*»).



Издатель, располагающийся в более низком слое «*Business Layer*», может уведомлять подписчиков, располагающихся в более высоком слое «*Presentation Layer*», не нарушая правил построения многослойной системы. Если бы издатель и подписчик представляли собой нечто

единое целое (объединенная логика издателя и подписчика в одном классе), то получившийся объект либо каким-то образом пересекал бы границы функционального слоя (нарушая принципы формирования слоев и их компонентов), либо должен был бы полностью находиться в каком-то одном слое, тем самым компрометируя абстракцию определенного слоя (Например, в объективной реальности, слой «*кухня*», можно скомпрометировать, разместив в этом слое объект-унитаз перенесенный из слоя «*туалет*»).

- **Поддержка широковещательных взаимодействий.**
  Паттерн Observer позволяет организовать посылку запроса от издателя подписчикам, так что при этом не потребуется явно указывать каждого определенного получателя-подписчика. Запрос автоматически поступает всем подписчикам. Издатель пренебрегает информацией о количестве подписчиков, от издателя требуется только уведомить всех имеющихся подписчиков об изменении состояния издателя. Такой подход позволяет в любое время безопасно добавлять и удалять подписчиков (*наблюдателей*), при этом подписчики сами принимают решение о том, что делать с уведомлением, поступившим от издателя (обработать или игнорировать).

Паттерн Observer обладает следующими недостатками:

- **Неожиданность обновлений.**
  Важно понимать, что подписчики не располагают информацией о внутренней структуре издателя и к тому же ничего не знают о наличии и устройстве других подписчиков, поэтому сложно предсказать во что обойдется (какой будет расход памяти и сколько займет процессорного времени) инициация цепочки изменений состояния всей связки «издатель-подписчики». Безобидная на первый взгляд операция над издателем, может вызвать целую волну обновлений подписчиков и зависящих от этих подписчиков других объектов.

  В случае наличия у подписчика ресурсоемкого вычисляемого состояния, потребуется специальный протокол, описывающий схемы избирательного обновления только части состояния, для этого подписчику потребуется понимать, что именно изменилось в издателе. Без такого дополнительного протокола, подписчикам потребуется проделать сложную работу по анализу характера изменений внутреннего состояния издателя или же полностью обновиться, не смотря на стоимость (расход памяти и процессорное время) такого обновления.

Реализация

Полезные приемы реализации паттерна Observer:

- **Хранение ссылок в издателе на подписчиков.**
  С помощью этого подхода издатель может отслеживать всех подписчиков, которым требуется посылать уведомления. Однако, при наличии (очень!) большого числа издателей и всего нескольких подписчиков, такой подход может оказаться накладным, так как в каждом издателе придется создавать коллекцию ссылок на подписчиков, что в совокупности может занять большой объем памяти. Чтобы сократить объем выделяемой памяти для хранения ссылок на подписчиков, можно воспользоваться ассоциативным массивом (например, хэш-таблицей - `Hashtable`) в котором будут храниться соответствия «издатель-подписчик». В таком случае издатели, которые не имеют подписчиков или имеют мало подписчиков не будут расходовать память на хранение ссылок, но при этом увеличится время поиска нужного подписчика в пуле подписчиков (хэш-таблице).

- **Подписка более чем на одного издателя.**
  Иногда подписчик может подписаться на несколько издателей. Например, у электронной таблицы может существовать несколько источников данных. В таких случаях требуется расширить интерфейс обновления - метод `Update(Subject publisher)`, вызываемый на подписчике, чтобы подписчик мог узнать какой издатель прислал уведомление. Издатель (`Subject`) может просто передать ссылку на себя в качестве аргумента метода subscriber.`Update(this)`, тем самым сообщая подписчику (`Observer`) кто именно стал инициатором обновления состояния.

- **Кто может быть инициатором обновлений.**

Для поддержания согласованности состояний между издателем и подписчиками используется механизм уведомлений (вызов метода `Notify` принадлежащего издателю). Возникает вопрос, кто именно должен вызывать метод `Notify` для инициирования обновления? Имеется два варианта:

1. Метод `Notify` вызывается непосредственно самим издателем (из методов класса `Subject`). *Преимущество* такого подхода заключается в том, что клиентам (`Client`) не надо помнить о необходимости вызова метода `Notify` класса `Subject`. *Недостаток* такого подхода в том, что при вызове определенных методов на издателе (`Subject`) эти методы могут вызвать метод `Notify`, тем самым внезапно инициировать волну обновлений подписчиков, что может стать причиной неэффективной работы программы.
2. Метод `Notify` вызывается клиентом на экземпляре класса `Subject`. В роли клиента может выступать как объект класса (`SomeClass`) не входящего в структуру паттерна, так и объекты подписчики (`Observer`). *Преимущество* такого подхода заключается в том, что клиент может отложить инициирование обновления группы подписчиков до определенного времени, тем самым исключив ненужные промежуточные обновления. *Недостаток* такого подхода заключается в том, что у клиентов появляется дополнительная обязанность и клиент должен помнить о том, что в определенный момент нужно инициировать серию обновлений подписчиков. Это увеличит вероятность совершения ошибок клиентом, поскольку клиент должен понимать устройство подписчиков, вникать в технологические трудности работы каждого подписчика и в конце концов клиент может просто забыть вызвать метод `Notify` вовремя.

- **Наличие в подписчиках «*висячих*» ссылок на неиспользуемых издателей.**
  В ходе работы программы, определенный издатель может выполнить свою роль и в последствии должен быть удален механизмом сборки мусора. При этом может возникнуть ситуация, когда подписчики будут продолжать ссылаться на неиспользуемого более издателя, что соответственно не позволит механизму сборки мусора удалить отработавшего издателя. Такая ситуация может привести к нехватке памяти и замедлению работы всего приложения. Чтобы позволить механизму сборки мусора удалить неиспользуемого более издателя, требуется чтобы подписчики удалили ссылки на неиспользуемого ими издателя. Для того чтобы подписчики удалили ссылки на неиспользуемого издателя, издатель должен уведомить своих подписчиков о своей дальнейшей ненадобности.

- **Гарантии непротиворечивости состояния издателя перед отправкой уведомления подписчикам.**
  Перед вызовом метода `Notify` издатель должен находится в (правильном) состоянии. Состояние издателя, которое передается подписчикам не должно вызывать противоречий на стороне подписчиков. Другими словами, все значения полей издателя (состояние) которые передаются на сторону подписчиков, должны быть такими чтобы после передачи не вызвать на стороне подписчика недоразумений и не привести к физическим и логическим ошибкам.

- **Протоколы обновления: Модели вытягивания и проталкивания.**
  В реализациях паттерна Observer, издателю часто требуется передать подписчикам дополнительную информацию о характере изменений. Такая информация передается в качестве аргумента метода `Update` и объем такой передаваемой информации может время от времени изменяться.

  Протокол обновления состояния подписчика имеет две модели: *модель вытягивания (Pull model)* и *модель проталкивания (Push model)*.

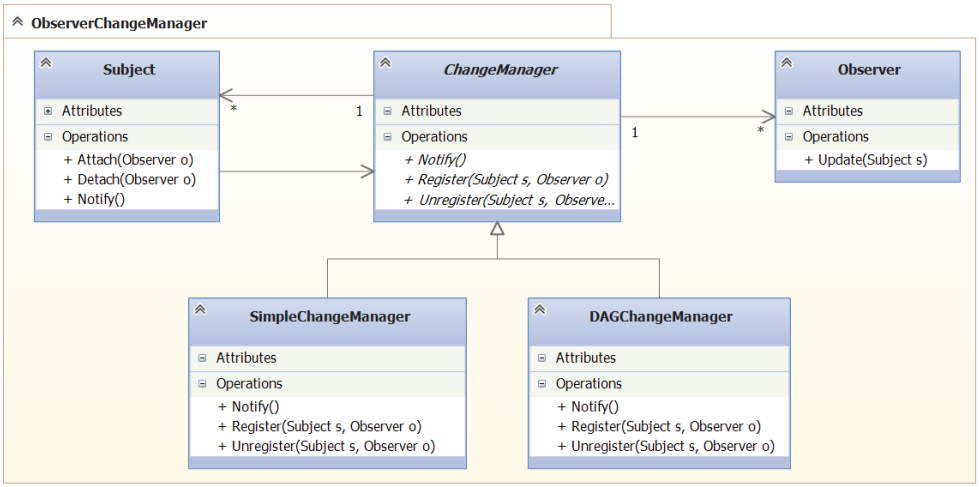
  При использовании *модели вытягивания*, издатель не посылает подписчику ничего кроме минимального уведомления об изменении состояния, а подписчик уже самостоятельно запрашивает детали состояния у издателя (если это требуется подписчику). В *модели вытягивания* подчеркивается неинформированность издателя о своих подписчиках. *Модель вытягивания* может оказаться не эффективной, если подписчикам (`Observer`) потребуется информация о деталях произведенных изменений в состоянии издателя (`Subject`).

  При использовании *модели проталкивания*, издатель посылает подписчикам детальную информацию о деталях измененного состояния, независимо от того нужно это подписчику или нет. В *модели проталкивания* предполагается, что издатель владеет информацией о потребностях подписчиков. В случае использования *модели проталкивания* может снизиться степень повторного использования, так как издатель (`Subject`) строит предположения о потребностях подписчиков (`Observer`), а эти предположения не всегда могут оказаться верны.

- **Явное указание представляющих интерес изменений.**
  Эффективность обновления состояния подписчиков, можно повысить за счет расширения способа регистрации подписчика в издателе, предоставив подписчику возможность выбрать только те события которые его интересуют. Когда произойдет определенное событие, издатель проинформирует только тех подписчиков, которые подписаны именно на это событие.

- **Сокрытие сложного смысла обновления.**
  Если отношения зависимости между издателями и подписчиками становятся сложными и запутанными (например, каждый из подписчиков подписан на несколько издателей), то может понадобиться объект (посредник) который уберет прямые связи отношений между объектами и выразит эти связи в форме алгоритма, а также возьмет под контроль все изменения состояний издателей и подписчиков. Такой объект можно назвать менеджером изменений (`ChangeManager`) и он будет служить для того чтобы подписчики могли корректно и своевременно синхронизироваться с состояниями издателей. Например, если выполнение некоторой операции повлечет за собой изменение состояний нескольких независимых издателей, то подписчики будут уведомлены об изменении состояния издателей только тогда, когда будут изменены состояния всех издателей, чтобы не уведомлять одного и того же подписчика несколько раз. Класс `ChangeManager` имеет три основных обязанности:
  1. Организация *ссылочной целостности* между издателем и подписчиками, а также предоставление интерфейса (набора методов) для поддержания ссылочной целостности в актуальном состоянии. Это освобождает издателей и подписчиков хранить ссылки друг на друга.
  2. Реализация (протокола) плана и правил обновления состояния.
  3. Обновление состояния всех подписчиков по требованию издателя.

- На диаграмме ниже представлена диаграмма классов, описывающая реализацию паттерна Observer с использованием менеджера изменений `ChangeManager`.



См. Пример к главе: \019_Observer\006_ObserverChangeManager

- **Комбинирование издателей и подписчиков.**
  В тех языках, которые не поддерживают множественного наследования реализации (например, C#), обычно не создаются отдельные классы `Subject` и `Observer`, а их интерфейсы комбинируются в одном классе. Такой подход позволяет создать объекты, являющиеся одновременно и издателями, и подписчиками. В языке C# имеется специальный стереотип - `delegate`, выражающий идею технической комбинации издателя `Subject` и подписчика `Observer` в одном объекте. В основе делегатов лежит функциональная природа несмотря на имеющееся объектно-ориентированное выражение данного стереотипа. Функциональная основа делегата, позволила практически полностью убрать использование громоздкой объектно-ориентированной подписки на события, что соответственно привело к уменьшению числа связей в программах. Предлагается рассмотреть пример использования делегатов в схеме «издатель-подписчик»:

```
delegate void SubjectObserver();

class Program
{
    // Update - логически относится к подписчику (Observer).
    static void Update()
    {
        Console.WriteLine("Hello world!");
    }

    static void Main()
    {
        SubjectObserver so = new SubjectObserver(Update);

        // Аналог вызова Notify() - логически относится к издателю (Subject).
        so.Invoke();
    }
}
```

См. Пример к главе: \019_Observer\ 002_Observer Event [001_Observer]

Приведенный пример показывает техническое, вульгарно-прямолинейное применение делегатов в схеме «издатель-подписчик», где делегат является «вещью в себе». Но, на подходе использования делегатов базируется полноценная событийная модель (event) платформы .Net. Событийная модель в .Net является логическим продолжением и более оптимальным выражением использования техники «издатель-подписчик», описываемой при помощи шаблона Observer. Предлагается рассмотреть пример организации событийной модели с использованием конструкции языка C# - событием (event).

```
// Подписчик.
delegate void Observer(string state);


// Издатель.
abstract class Subject
{
    protected Observer observers = null;

    public event Observer Event
    {
        add { observers += value; }
        remove { observers -= value; }
    }

    public abstract string State { get; set; }
    public abstract void Notify();
}
```

```
// Конкретный издатель.
class ConcreteSubject : Subject
{
    public override string State { get; set; }
```

```
        public override void Notify()
        {
            observers.Invoke(State);
        }
    }

    class Program
    {
        static void Main()
        {
            // Издатель.
            Subject subject = new ConcreteSubject();

            // Подписчик, с сообщенным лямбда выражением.
            Observer observer = new Observer(
      (observerState) => Console.WriteLine(observerState + " 1"));

            // Подписка на уведомление о событии.
            subject.Event += observer;
            subject.Event +=
      (observerState) => Console.WriteLine(observerState + " 2");

            subject.State = "State ...";
            subject.Notify();

            Console.WriteLine(new string('-', 11));

            // Отписка от уведомлений.
            subject.Event -= observer;
            subject.Notify();

            // Delay.
            Console.ReadKey();
        }
    }
```

См. Пример к главе: \019_Observer\ 002_Observer Event [002_Observer]

Нельзя смотреть «узко» на событийную модель и пытаться избавиться от объектно-ориентированного выражения издателя и подписчика в программных системах. Делегаты (`delegate`) и события (`event`) в .Net следует воспринимать как вспомогательный, сугубо технический механизм-связку для уменьшения числа явных связей отношений между объектами. По сути сами связи остаются, но уже не выражаются так явно и объектно-ориентированно. Эти связи выражены с использованием функционального подхода, функциональной природы делегата и соответственно на этих связях программисты не делают акцент при анализе системы. Связи в схеме «издатель-подписчик» стали чем-то само собой разумеющимся и неявным.

Пример кода

Интерфейс подписчика определен в абстрактном классе `Observer`.

```
abstract class Observer
{
    public abstract void Update(Subject theChangedSubject);
}
```

При такой реализации поддерживается несколько издателей для одного подписчика. Передача издателя в качестве параметра метода `Update` позволяет подписчику определить, состояние какого из издателей изменилось.

В абстрактном классе `Subject` определен интерфейс издателя.

```
abstract class Subject
{
    protected List<Observer> observers = new List<Observer>();

    public virtual void Attach(Observer observer)
    {
        observers.Add(observer);
        observer.Update(this);
    }

    public virtual void Detach(Observer observer)
    {
        observers.Remove(observer);
    }

    public virtual void Notify()
    {
        foreach (var o in observers)
            o.Update(this);
    }
}
```

`ClockTimer` – это конкретный издатель, который следит за временем и оповещает подписчиков каждую секунду. Класс `ClockTimer` предоставляет интерфейс для получения отдельных компонентов времени: часы, минуты, секунды и т.д.

```
class ClockTimer : Subject
{
    System.Threading.Timer timer;
    private TimeSpan currentTime;

    static void TimerProc(object o)
    {
        (o as ClockTimer).Tick();
    }

    public ClockTimer()
    {
        timer = new System.Threading.Timer(TimerProc, this, 1000, 1000);
    }




    public void Tick()
```

```
        {
            currentTime = DateTime.Now.TimeOfDay;
            Notify();
        }

        public int GetHour()
        {
            return currentTime.Hours;
        }

        public int GetMinute()
        {
            return currentTime.Minutes;
        }

        public int GetSecond()
        {
            return currentTime.Seconds;
        }

        public TimeSpan GetTime()
        {
            return currentTime;
        }
    }
```

Операция Tick вызывается через одинаковые интервалы внутренним таймером, тем самым обеспечивая правильный отсчет времени. При этом обновляется внутреннее состояние объекта ClockTimer и вызывается метод Notify для уведомления подписчиков об изменении времени.

```
        public void Tick()
        {
            currentTime = DateTime.Now.TimeOfDay;
            Notify();
        }
```

Класс DigitalClock отображает время в цифровом формате.

```
    class DigitalClock : Observer
    {
        Label digitalClockLabel;
        Subject subject;
        TimeSpan time;

        public Control GetControl
        {
            get { return digitalClockLabel; }
        }

        public DigitalClock(Control parent, Subject subject)
        {
            digitalClockLabel = new Label { Parent = parent };
            this.subject = subject;
            subject.Attach(this);
        }



        public override void Update(Subject theChangedSubject)
        {
            time = (theChangedSubject as ClockTimer).GetTime();
```

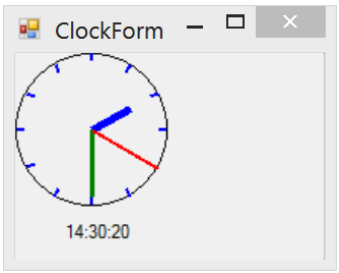
```
            digitalClockLabel.BeginInvoke(new Action(Draw));
        }

        public void Draw()
        {
            digitalClockLabel.Text = time.ToString("hh\\:mm\\:ss");
        }
    }
```

Класс AnalogClock отображает время в аналоговом формате.

```
    class AnalogClock : Observer
    {
        class AnalogClockPanel : Panel
        {
            public AnalogClockPanel()
            {
                SetStyle(ControlStyles.UserPaint |
                         ControlStyles.OptimizedDoubleBuffer |
                         ControlStyles.AllPaintingInWmPaint,
                         true);
                DoubleBuffered = true;
            }
        }

        Panel analogClockPanel;
        Subject subject;
        TimeSpan time;
        Point center = new Point(50, 50);

        public AnalogClock(Control parent, Subject subject)
        {
            analogClockPanel = new AnalogClockPanel() { Parent = parent };

            analogClockPanel.Size = new System.Drawing.Size(100, 100);
            this.subject = subject;
            subject.Attach(this);
        }

        public Control GetControl
        {
            get { return analogClockPanel; }
        }

        public override void Update(Subject theChangedSubject)
        {
            time = (theChangedSubject as ClockTimer).GetTime();
            analogClockPanel.Invoke(new Action(Draw));
        }









        public void Draw()
        {
            analogClockPanel.Refresh();
            var g = analogClockPanel.CreateGraphics();
            var rr = analogClockPanel.ClientRectangle;
```

```
            rr.Width -= 1;
            rr.Height -= 1;
            g.DrawEllipse(new Pen(Color.Black, 1), rr);

            for (int i = 0; i < 12; i++)
            {
                var rrr = (float)(Math.PI * 2f) / 60f * (float)(i * 5);
                var from = GetDestinationPoint(rrr, 45);
                var to = GetDestinationPoint(rrr, 50);
                drawLine(g, from, to, new Pen(Color.Blue, 2));
            }

            drawW(g, (float)(Math.PI * 2f) / 12f * (float)(time.Hours % 12), 30,
                                                        new Pen(Color.Blue, 5));
            drawW(g, (float)(Math.PI * 2f) / 60f * (float)time.Minutes, 45,
                                                        new Pen(Color.Green, 3));
            drawW(g, (float)(Math.PI * 2f) / 60f * (float)time.Seconds, 50,
                                                        new Pen(Color.Red, 2));
        }

        Point GetDestinationPoint(float radians, int length)
        {
            int a = center.Y - (int)(Math.Cos(radians) * (float)length);
            int b = center.X + (int)(Math.Sin(radians) * (float)length);
            return new Point(b, a);
        }

        void drawLine(Graphics g, Point from, Point to, Pen p)
        {
            g.DrawLine(p, from, to);
        }

        void drawW(Graphics g, float radians, int length, Pen p)
        {

            drawLine(g, center, GetDestinationPoint(radians, length), p);
        }
    }
```

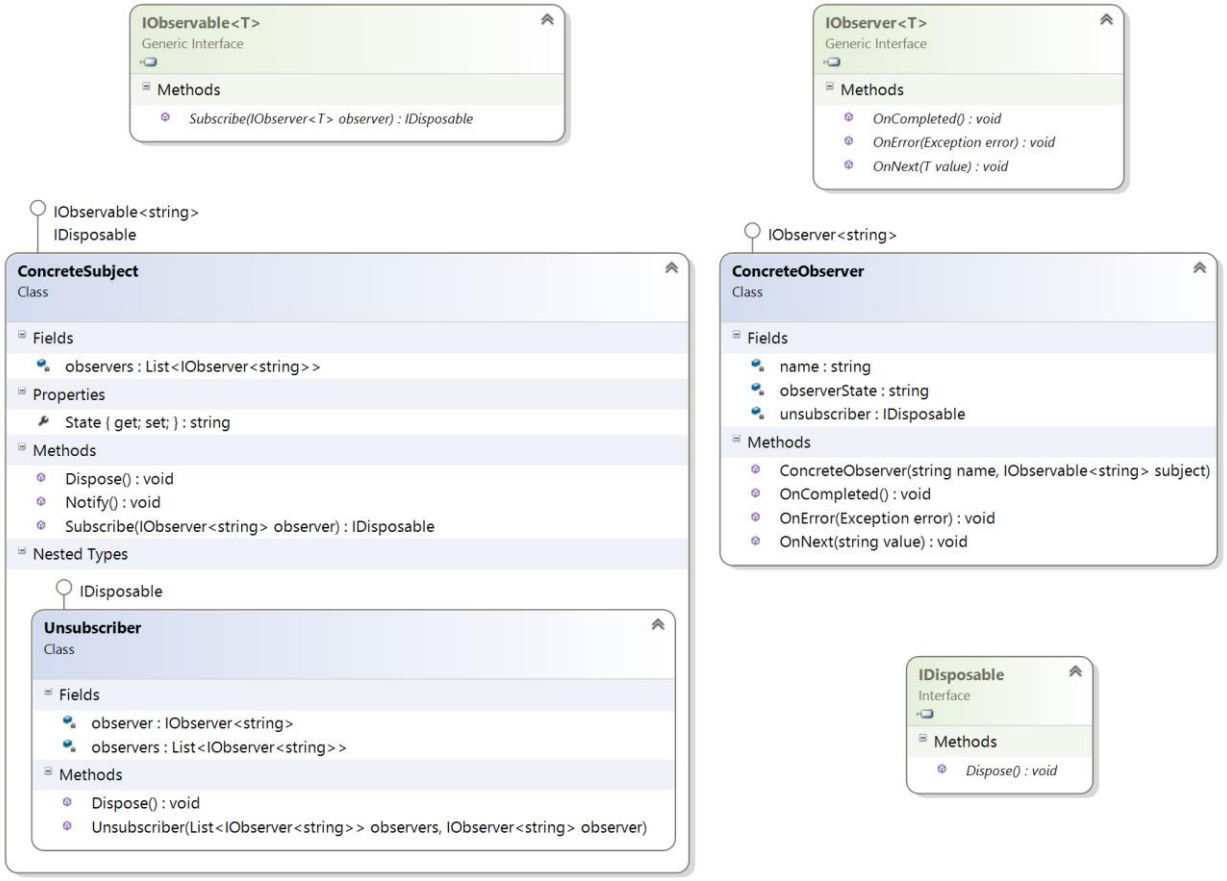
Результат работы программы:



См. Пример к главе: \019_Observer\007_ObserverClocks

Известные применения паттерна в .Net

Паттерн Observer, выражен в языке C# в виде идеи использования функционально-ориентированного стереотипа - делегата (`delegate`) и языковой конструкции - события (`event`), которая в свою очередь строится на использовании делегата. Так же имеется выражение паттерна Observer в FCL (Framework Class Library) виде двух интерфейсов (`interface`) - `IObservable<out T>` и `IObserver<in T>`. Ниже с использованием диаграмм языка DSL представлена структура паттерна Observer.



Пример кода, демонстрирует использование интерфейсов `IObservable<out T>` и `IObserver<in T>`.

Реализация подписчика:

```
class ConcreteObserver : IObserver<string>
{
    string name;
    string observerState;
    IDisposable unsubscriber;

    public ConcreteObserver(string name, IObservable<string> subject)
    {
        this.name = name;
        unsubscriber = subject.Subscribe(this);
    }

    // Реализация интерфейса IObserver<T>
    public void OnCompleted()
    {
        unsubscriber.Dispose();
    }
    public void OnError(Exception error)
    {
```

```
            Console.ForegroundColor = ConsoleColor.Red;
            Console.WriteLine("Observer {0}, Error: {1}", name, error.Message);
            Console.ForegroundColor = ConsoleColor.Gray;
        }

        // Аналог Update(argument) - модель проталкивания.
        public void OnNext(string value)
        {
            observerState = value;
            Console.WriteLine("Observer {0}, State = {1}", name, observerState);
        }
    }
```

Реализация издателя:

```
    class ConcreteSubject : IObservable<string>, IDisposable
    {
        public string State { get; set; }

        List<IObserver<string>> observers = new List<IObserver<string>>();

        public void Notify()
        {
            foreach (IObserver<string> observer in observers)
            {
                if (this.State == null)
                    observer.OnError(new NullReferenceException());
                else
                    observer.OnNext(this.State); // Модель проталкивания.
            }
        }

        // Реализация интерфейса IObservable<T>
        // (UnSubscribe выполняется через IDisposable)

        /// <summary>
        /// Подписать подписчика.
        /// </summary>
        /// <param name="observer">Конкретный подписчик</param>
        /// <returns>Объект отписывающий подписанного подписчика</returns>
        public IDisposable Subscribe(IObserver<string> observer)
        {
            if (!observers.Contains(observer))
                observers.Add(observer);

            return new Unsubscriber(observers, observer);
        }

        // Отписать всех подписчиков.
        public void Dispose()
        {
            observers.Clear();
        }


        // Nested Class
        class Unsubscriber : IDisposable
        {
            List<IObserver<string>> observers;
```

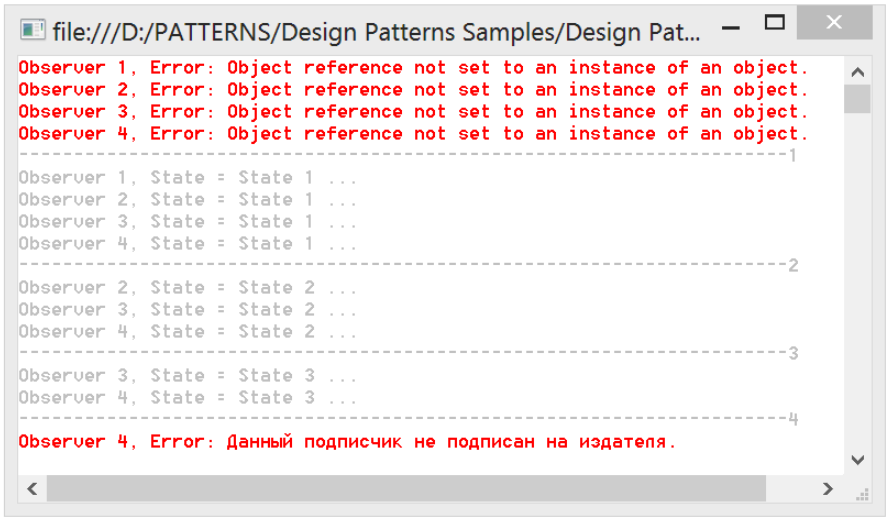
```
        IObserver<string> observer;

        public Unsubscriber(List<IObserver<string>> observers,
                            IObserver<string> observer)
        {
            this.observers = observers;
            this.observer = observer;
        }

        public void Dispose()
        {
            if (observers.Contains(observer))
                observers.Remove(observer);
            else
                observer.OnError(new Exception("Данный подписчик не подписан"));
        }
    }
}
```

Использование:

```
class Program
{
    static void Main()
    {
        // Создание издателя.
        ConcreteSubject subject = new ConcreteSubject();

        // Создание подписчиков.
        ConcreteObserver observer1 = new ConcreteObserver("1", subject);
        ConcreteObserver observer2 = new ConcreteObserver("2", subject);
        ConcreteObserver observer3 = new ConcreteObserver("3", subject);
        ConcreteObserver observer4 = new ConcreteObserver("4", subject);

        // Подписание подписчиков на издателя с получением объекта для отписки.
        IDisposable unsubscriber1 = subject.Subscribe(observer1);
        IDisposable unsubscriber2 = subject.Subscribe(observer2);
        IDisposable unsubscriber3 = subject.Subscribe(observer3);
        IDisposable unsubscriber4 = subject.Subscribe(observer4);

        using (subject)
        {
            // Попытка предоставить подписчикам некорректное состояние.
            subject.State = null;
            subject.Notify();
            Console.WriteLine(new string('-', 70) + "1");

            // Отписка первого подписчика через
            // ConcreteSubject.Unsubscriber.Dispose()
            using (unsubscriber1)
            {
                // Попытка предоставить подписчикам корректное состояние.
                subject.State = "State 1 ...";
                subject.Notify();
            }

            Console.WriteLine(new string('-', 70) + "2");

            // State 2 - получат только три подписчика
            // которые остались подписанными.
```

```
            subject.State = "State 2 ...";
            subject.Notify();

            Console.WriteLine(new string('-', 70) + "3");

            // Отписка второго подписчика через ConcreteObserver.OnCompleted()
            observer2.OnCompleted();

            // State 3 - получат только 2 подписчика
            // которые остались подписанными.
            subject.State = "State 3 ...";
            subject.Notify();
        } // observers.Clear()

        Console.WriteLine(new string('-', 70) + "4");

        // Попытка отписать уже отписанного подписчика, обрабатывается в
        // ConcreteSubject.Unsubscriber.Dispose()
        observer4.OnCompleted();

        // Delay.
        Console.ReadKey();
    }
}
```

Результат работы программы:



См. Пример к главе: \019_Observer\003_IObserver

Паттерн State

Название

Состояние

Также известен как

FSM (Finite-State Machine) - КА (Конечный автомат или Машина состояний)

Классификация

По цели: поведенческий
По применимости: к объектам

Частота использования

Средняя - 1 2 **3** 4 5

Назначение

Паттерн State – позволяет объекту изменять свое поведение в зависимости от своего состояния. Поведение объекта изменяется на столько, что создается впечатление, что изменился класс объекта.

Введение

Паттерн State описывает способы и правила построения объектно-ориентированного представления конечных автоматов (КА). Конечные автоматы классифицируются по логическим свойствам функций переходов и выходов, а также по характеру отсчёта дискретного времени.

По способу формирования функций выходов выделяют автоматы Мили и Мура. У автоматов Мили функции выходов находятся на ребре, а у автоматов Мура функции выходов находятся в состоянии. По характеру отсчёта дискретного времени автоматы делятся на синхронные и асинхронные.

Конечный автомат Мили

Рассмотрим простейший тип конечно-автоматного преобразователя информации: Автомат Мили. Определим конечный автомат формально. Есть и другие определения, но мы остановимся на этом.

ОПРЕДЕЛЕНИЕ: Конечным автоматом Мили называется шестерка объектов:

A = <S, X, Y, s_0, δ, λ>, где:

- **S** - конечное непустое множество (состояний);
- **X** - конечное непустое множество входных сигналов (входной алфавит)
- **Y** - конечное непустое множество выходных сигналов (выходной алфавит)
- $s_0 \in S$ - начальное состояние
- $\delta: S \times X \to S$ - функция переходов
- $\lambda: S \times X \to Y$ - функция выходов

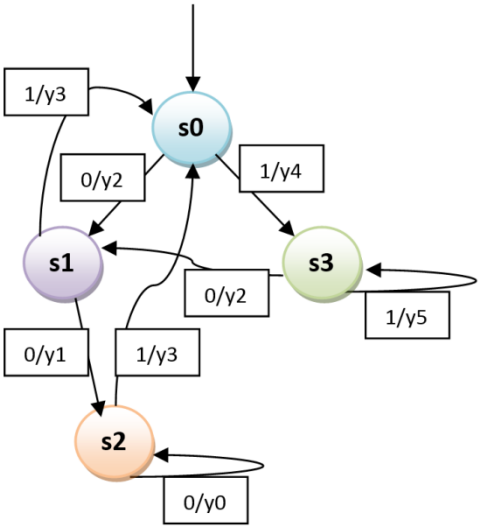
Задавать автомат удобно графом, в котором вершины соответствуют состояниям, а ребро из состояния s_n в состояние s_m, помеченное **x/y**, проводится тогда, когда автомат из состояния s_n под воздействием входного сигнала **x** переходит в состояние s_m с выходной реакцией **y**.

ПРИМЕР:

Зададим конечный автомат, который имеет:

- четыре состояния, **S = {s_0, s_1, s_2, s_3}**
- два входных сигнала **X = {x_0, x_1}**, где: **x_0 = 0, x_1 = 1**
- шесть выходных сигналов **Y = {y_0, y_1, y_2, y_3, y_4, y_5}** где: **y_0 = 1, y_1 = 2, y_2 = 3, y_3 = 4, y_4 = 5, y_5 = 6.**

Теперь представим автомат в виде графа:



Граф 1.

Кроме графического представления автомата, можно использовать и табличное, задавая функции переходов и выходов в виде таблиц. Представим данный автомат таблично. *Таблица 1* - определяет функцию переходов автомата из одного состояния в другое.

Функция переходов **$\delta(s_n, x_n)$** определяется так: **$\delta(s_0, 0) = s_1$; $\delta(s_2, 1) = s_0$; ….**

δ	0	1
s_0	s_1	s_3
s_1	s_2	s_0
s_2	s_2	s_0
s_3	s_1	s_3

Таблица 1.

Таблица 2 - определяет функцию выходов **$\lambda(s_n, x_n)$** так: **$\lambda(s_0, 0) = y_2$; $\lambda(s_2, 1) = y_3$; ….**

λ	0	1
s_0	y_2	y_4
s_1	y_1	y_3
s_2	y_0	y_3
s_3	y_2	y_5

Таблица 2.

Реализация конечного автомата Мили

Для программной реализации будет использоваться язык C# и технология WWF – как инструменты, упрощающие реализацию программных конечных автоматов. На рисунке 1, представлена блок-схема (WWF) программы, реализующей поведение автомата.

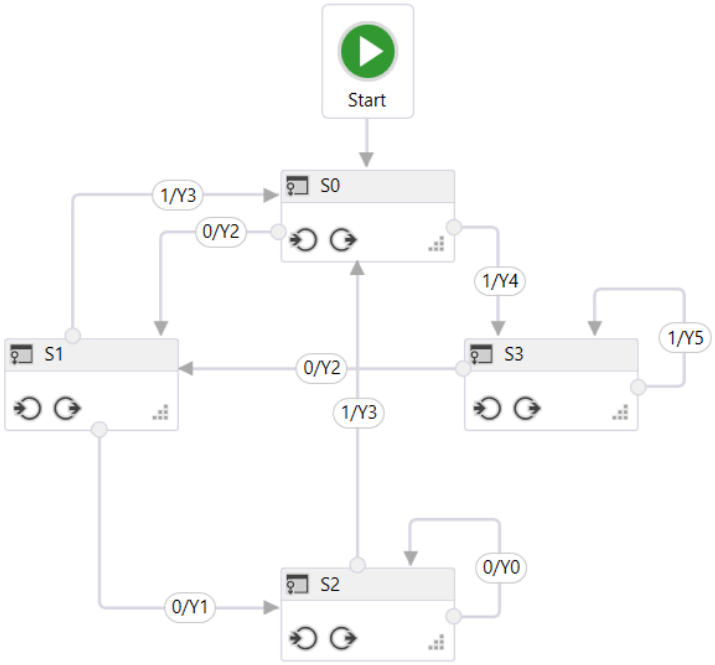


Рисунок 1.

Нетрудно увидеть, что топология блок-схемы программы повторяет топологию графа переходов конечного автомата. С каждым состоянием связана операция, выполняющая функцию ожидания очередного события прихода нового входного сигнала и чтение его в стандартный буфер – `int x`, а также последующий анализ того, какой это сигнал. В зависимости от того, какой сигнал пришел на вход, выполняется та или иная функция $y_0 - y_5$ и происходит переход к новому состоянию. Построив программу подобную этой и добавив активные устройства, реализующие отдельные входные и выходные операции, можно управление каким-либо процессом поручить компьютеру.

См. Пример к главе: \020_State\002_MealyStateMachine

Конечный автомат Мура

Рассмотрим еще один тип конечно-автоматного преобразователя информации: Автомат Мура. Автоматы Мура образуют другой класс моделей, с точки зрения вычислительной мощности полностью эквивалентных классу автоматов Мили. В автомате Мура: **A = <S, X, Y, s_0, δ, λ>**, выходная функция λ определяется не на паре **<состояние, входной сигнал>**, а только на состоянии $\lambda: S \rightarrow Y$. Выход автомата Мура определяется однозначно тем состоянием, в которое автомат переходит после приема входного сигнала.

ПРИМЕР:

Зададим конечный автомат Мура, который будет являться эквивалентным автомату Мили, рассмотренному выше. Не составит труда преобразовать исходный автомат Мили в автомат Мура. На графе автомата Мили легко прослеживается взаимосвязь элементов входного и выходного алфавитов.

Построив таблицу соответствий состояний автомата Мили алфавитным парам, легко увидеть какие состояния требуют расщепления.

	x_0	x_1	Расщепление состояния S
s_0	y_3	y_3	Не требуется
s_1	y_2	y_2	Не требуется
s_2	y_0	y_1	$s_2 = \{s_2, s_4\}$
s_3	y_4	y_5	$s_3 = \{s_3, s_5\}$

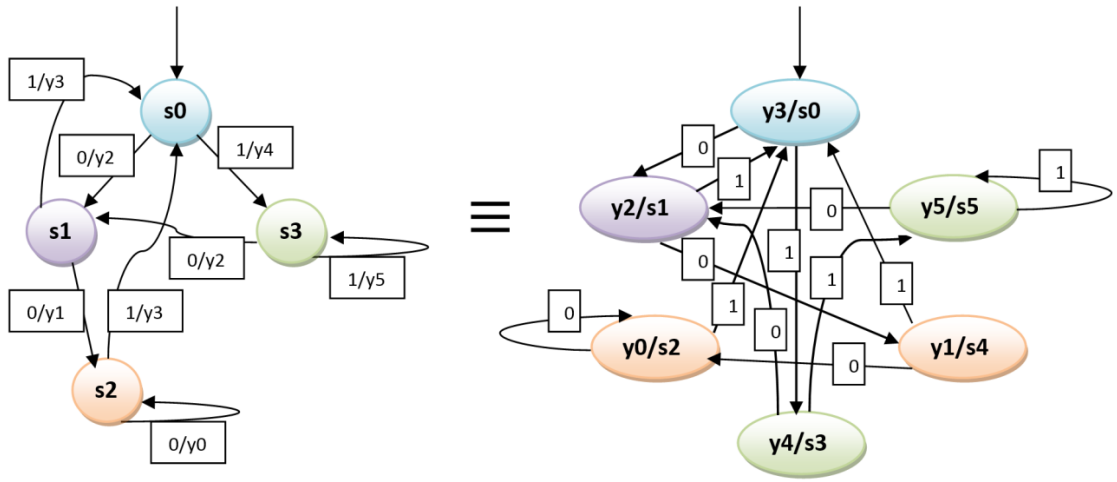
Таблица 3.

Состояния s_2 и s_3 автомата Мили расщепляются на два эквивалентных состояния: $\mathbf{s_2 = \{s_2, s_4\}, s_3 = \{s_3, s_5\}}$, с каждым из которых связывается один выходной символ.

Автомат Мура будет иметь:

- шесть состояний, $\mathbf{S = \{s_0, s_1, s_2, s_3, s_4, s_5\}}$
- два входных сигнала $\mathbf{X = \{x_0, x_1\}}$, где: $\mathbf{x_0 = 0, x_1 = 1}$.
- шесть выходных сигналов $\mathbf{Y = \{y_0, y_1, y_2, y_3, y_4, y_5\}}$ где: $\mathbf{y_0 = 1, y_1 = 2, y_2 = 3, y_3 = 4, y_4 = 5, y_5 = 6.}$

Теперь представим автомат Мура в виде графа (в сравнении с автоматом Мили):



Автомат Мили Автомат Мура

Граф 2.

Реализация конечного автомата Мура

На рисунке 2, представлена блок-схема (WWF) программы реализующей поведение автомата Мура.

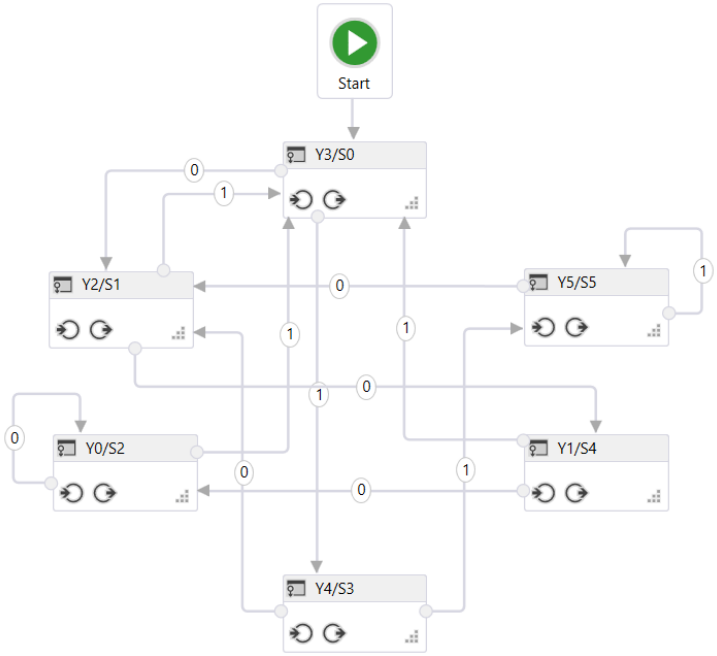


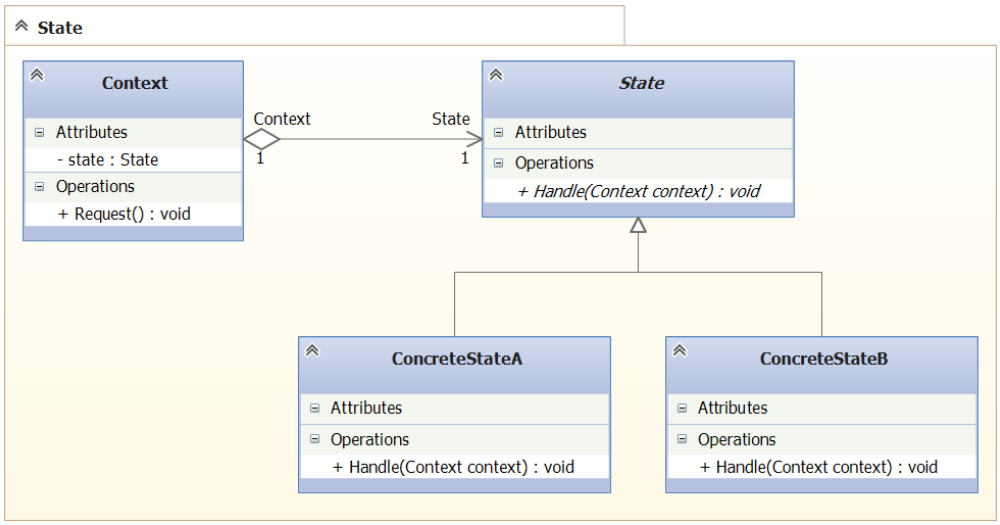
Рисунок 2.

Нетрудно увидеть, что топология блок-схемы программы повторяет топологию графа переходов конечного автомата. С каждым состоянием связана операция, выполнения определенных действий при наступлении данного состояния, ожидания прихода нового входного сигнала, чтение его в стандартный буфер – `int x`, а также последующий анализ того, какой это сигнал и перевод автомата в новое состояние.

См. Пример к главе: \020_State\003_MooreStateMachine

Рассмотренные программные модели автоматов Мура и Мили - полностью эквивалентны.

Структура паттерна на языке UML



См. Пример к главе: \020_State\001_State

Структура паттерна на языке C#

```
class Program
{
    static void Main()
    {
        Context context = new Context(new ConcreteStateA());
        context.Request();
        context.Request();
    }
}
```

```
class Context
{
    public State State { get; set; }

    public Context(State state)
    {
        this.State = state;
    }

    public void Request()
    {
        this.State.Handle(this);
    }
}
```

```
abstract class State
{
    public abstract void Handle(Context context);
}
```

```
class ConcreteStateA : State
{
    public override void Handle(Context context)
    {
        context.State = new ConcreteStateB();
    }
}
```

```
class ConcreteStateB : State
{
    public override void Handle(Context context)
    {
        context.State = new ConcreteStateA();
    }
}
```

См. Пример к главе: \020_State\001_State

Участники

- `Context` - **Контекст**:
  Представляет собой объектно-ориентированное представление конечного автомата. Предоставляет интерфейс взаимодействия, посредством которого клиенты могут оказывать влияние на конечный автомат, например, посредством передачи входных сигналов. Объект класса `Context` содержит ссылку на экземпляр текущего состояния.

- `State` - **Состояние**:
  Задает интерфейс взаимодействия с «объектом-состоянием».

- `ConcreteState` - **Конкретное состояние**:
  Реализует поведение (функции выхода и функции перехода) ассоциированное с определенным состоянием. Очевидно, что данный паттерн реализует модель КА – Мура.

Отношения между участниками

Отношения между классами

- Конкретные классы (`ConcreteStateA` и `ConcreteStateB`) связанны связью отношения наследования с абстрактным классом `State`.

- Конкретный класс `Context` связан связью отношения агрегации с абстрактным классом `State`.

Отношения между объектами

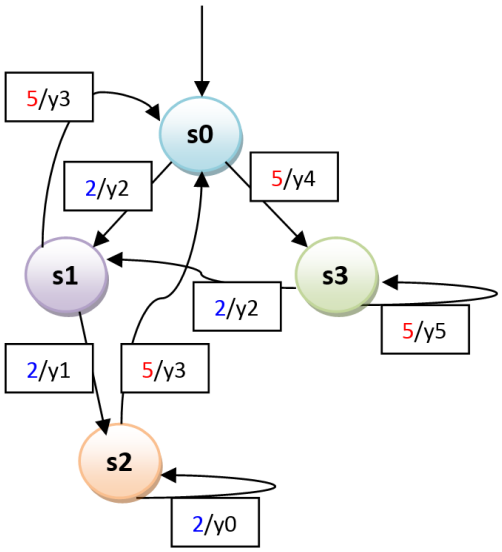
- Объект класса `Context` передает клиентские запросы объекту класса `ConcreteState`.

- Объект класса `Context` может передать себя в качестве аргумента объекту типа `State`, который будет обрабатывать запрос. Такой подход позволит объекту-состоянию получить доступ к контексту и произвести замену состояния.

- Объект класса `Context` обладает интерфейсом для конфигурирования его нужным состоянием. Один раз сконфигурировав контекст, клиенты должны отказаться от дальнейшей конфигурации. Чаще всего объекты `ConcreteState` сами принимают решение о том, при каких условиях и в каком порядке происходит изменение состояния.

Мотивация

В качестве примера использования паттерна State предлагается построить машину состояний описывающую поведение отца, отправившего сына в школу. Условимся, что сын может получать только двойки или пятерки. Отец не хочет браться за ремень каждый раз, когда сын получает двойку и выбирает более тонкую тактику воспитания. Граф автомата, моделирующего умное поведение отца представлен ниже.



Граф 3.

Этот автомат имеет:

- четыре состояния в которых может пребывать отец, **S = { s_0, s_1, s_2, s_3 }**, где:
  - **s_0** – Neutral (Нейтральное состояние)
  - **s_1** – Pity (Жалость)
  - **s_2** – Anger (Гнев)
  - **s_3** – Joy (Радость)
- два входных сигнала – оценки полученные сыном в школе **X = { x_0, x_1 }**, где:
  - **x_0** = 2
  - **x_1** = 5
- шесть выходных сигналов, т.е., действий отца **Y = { y_0, y_1, y_2, y_3, y_4, y_5 }**, где:
  - **y_0** - брать ремень;
  - **y_1** - ругать сына;
  - **y_2** - успокаивать сына;
  - **y_3** - надеяться;
  - **y_4** - радоваться;
  - **y_5** - ликовать;

Сына, получившего одну и туже оценку – ожидает дома совершенно различная реакция отца в зависимости от предыстории его учебы. Отец помнит, как его сын учился раньше и строит модель воспитания с учетом успехов и неудач сына. Например, после третьей двойки в истории (2, 2, 2) сына встретят ремнем (**y_0**), а в истории (2, 2, 5, 2) – будут успокаивать(**y_2**).

Каждая предыстория определяет текущее состояние автомата, при этом некоторые входные предыстории эквивалентны (те истории которые приводят автомат в одно и тоже состояние), например, история (2, 2, 5) эквивалентна пустой истории, которой соответствует начальное состояние.

Представим данный автомат таблично. *Таблица 4* - определяет функцию переходов автомата из одного состояния в другое.

Функция переходов $\delta(s_n, x_n)$ определяется так: $\delta(s_0, 2) = s_1$; $\delta(s_2, 5) = s_0$; ….

δ	2	5
s_0 - Neutral	s_1	s_3
s_1 - Pity	s_2	s_0
s_2 - Anger	s_2	s_0
s_3 - Joy	s_1	s_3

Таблица 4.

Таблица 5 - определяет функцию выходов $\lambda(s_n, x_n)$ так: $\lambda(s_0, 2) = y_2$; $\lambda(s_2, 5) = y_3$; ….

λ	2	5
s_0 - Neutral	y_2 - успокаивать сына	y_4 - радоваться
s_1 - Pity	y_1 - ругать сына	y_3 - надеяться
s_2 - Anger	y_0 - брать ремень	y_3 - надеяться
s_3 - Joy	y_2 - успокаивать сына	y_5 - ликовать

Таблица 5.

Для упрощения программной реализации примера, рекомендуется преобразовать исходный автомат Мили в эквивалентный по вычислительной мощности автомат Мура. Для этого требуется произвести расщепление тех состояний - s_n, которым соответствует более одной функции выхода - $\lambda(s_n, x_n)$.

	x_0	x_1	Расщепление состояния S
s_0 - Neutral	y_3	y_3	Не требуется
s_1 - Pity	y_2	y_2	Не требуется
s_2 - Anger	y_0	y_1	$s_2 = \{s_2, s_4\}$
s_3 - Joy	y_4	y_5	$s_3 = \{s_3, s_5\}$

Таблица 6.

Из таблицы преобразования видно, что произошло формирование двух новых состояний: $\{s_4, s_5\}$. Ранее в состоянии гнева – s_2, отец либо ругал сына – y_1, либо использовал ремень – y_0, а в состоянии радости – s_3, либо просто радовался – y_4, либо ликовал – y_5. Таким образом, функциональность можно распределить по состояниям согласно соответствия силы эмоциональной экспрессии состояния и выполняемого действия.

Например, $s_{2\text{-Гнев}} = \{s_{2\text{-Сильный Гнев}}, s_{4\text{-Гнев}}\}$, $s_{3\text{-Радость}} = \{s_{3\text{-Радость}}, s_{5\text{-Сильная Радость}}\}$. Соответственно: В состоянии «*простого*» гнева - $s_{4\text{-Гнев}}$, можно ругать сына - $y_{1\text{ - ругать сына}}$, а в состоянии сильного гнева - $s_{2\text{-Сильный Гнев}}$, можно использовать ремень - $y_{0\text{ - брать ремень}}$. В состоянии «*простой*» радости - $s_{3\text{-Радость}}$, можно радоваться - $y_{4\text{ - радоваться}}$, а в состоянии сильной радости - $s_{5\text{-Сильная Радость}}$, можно ликовать - $y_{5\text{ - ликовать}}$. Представим новый автомат таблично.

Таблица 7 - определяет функцию переходов автомата из одного состояния в другое.

δ	2	5
s_0 - Neutral	s_1	s_3
s_1 - Pity	s_4	s_0
s_2 - Strong Anger	s_2	s_0
s_3 - Joy	s_1	s_5
s_4 - Anger	s_2	s_0
s_5 - Strong Joy	s_1	s_5

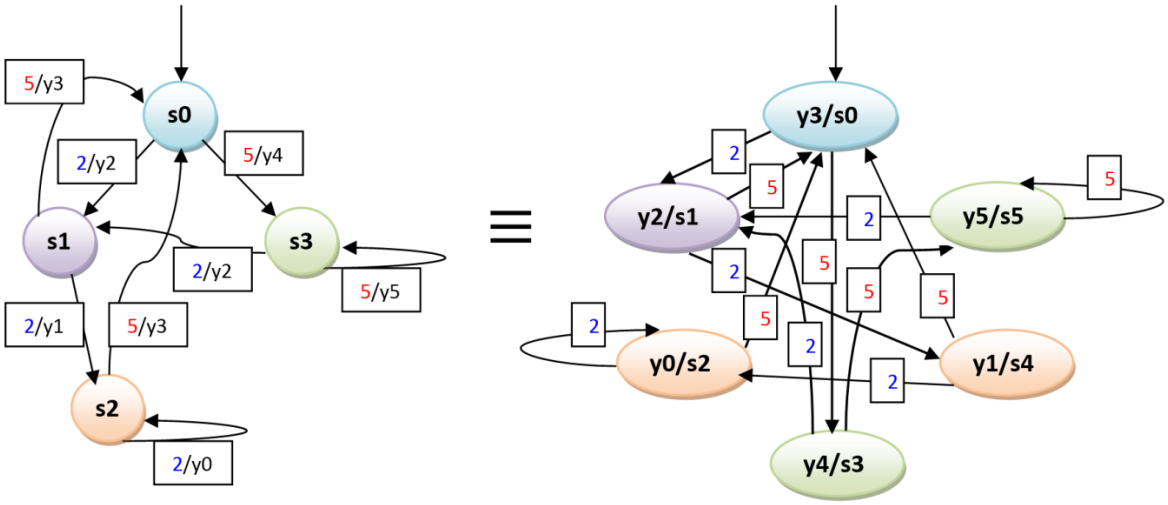
Таблица 7. (Функции переходов - δ (s_n, x_n))

Таблица 8 - определяет функцию выходов.

λ	2	5
s_0 - Neutral	y_2 - успокаивать сына	y_4 - радоваться
s_1 - Pity	y_1 - ругать сына	y_3 - надеяться
s_2 - Strong Anger	y_0 - брать ремень	y_3 - надеяться
s_3 - Joy	y_2 - успокаивать сына	y_5 - ликовать
s_4 - Anger	y_0 - брать ремень	y_3 - надеяться
s_5 - Strong Joy	y_2 - успокаивать сына	y_5 - ликовать

Таблица 8. (Функции выходов - λ (s_n, x_n))

Теперь представим автомат Мура в виде графа (в сравнении с автоматом Мили):



Автомат Мили Автомат Мура

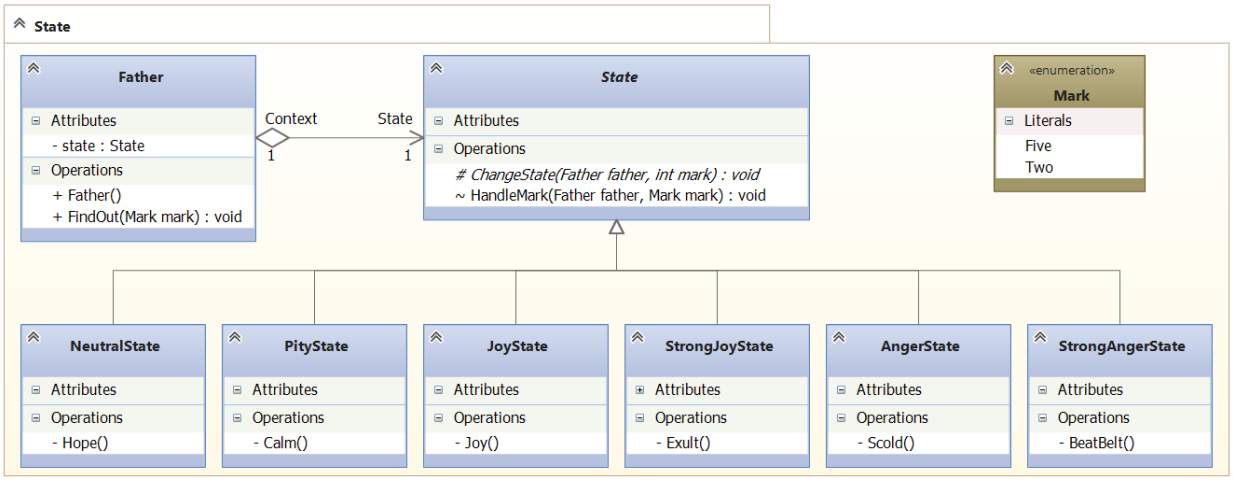
Граф 4.

Получившийся автомат Мура имеет:

- шесть состояний в которых может пребывать отец, **S = { s_0, s_1, s_2, s_3, s_4, s_5 }**, где:
  - **s_0** – Neutral (*Нейтральное состояние*)
  - **s_1** – Pity (*Жалость*)
  - **s_2** – Strong Anger (*Сильный Гнев*)
  - **s_3** – Joy (*Радость*)
  - **s_4** – Anger (*Гнев*)
  - **s_5** – Strong Joy (*Сильная Радость*)
- два входных сигнала – оценки полученные сыном в школе **X = { x_0, x_1 }**, где:
  - **x_0** = 2
  - **x_1** = 5
- шесть выходных сигналов, т.е., действий отца **Y = { y_0, y_1, y_2, y_3, y_4, y_5 }**, где:
  - **y_0** - брать ремень;
  - **y_1** - ругать сына;
  - **y_2** - успокаивать сына;
  - **y_3** - надеяться;
  - **y_4** - радоваться;
  - **y_5** - ликовать;

Построив программу, реализующую логику данного конечного автомата и добавив активные устройства (звуковые динамики и механизм-манипулятор для управления пространственным положением ремня при обработке мягких тканей задней и латеральных поверхностей таза), можно воспитание своего сына поручить компьютеру.

Предлагается более подробно рассмотреть программную реализацию автомата Мура, имитирующего поведение умного отца.



Рассмотрим класс `Father`. Объект класса `Father` может находиться в одном из шести состояний: `NeutralState` (*Нейтральное состояние*), `PityState` (*Жалость*), `JoyState` (*Радость*), `StrongJoyState` (*Сильная Радость*), `AngerState` (*Гнев*), `StrongAngerState` (*Сильный Гнев*). Все классы состояний являются производными от базового абстрактного класса `State`. Объект типа `Father` хранит ссылку на определенный объект состояния.

```
public class Father
{
    internal State State { get; set; }

    public Father()
    {
        State = new NeutralState();
    }

    public void FindOut(Mark mark)
    {
        State.HandleMark(this, mark);
    }
}
```

Объект класса `Father` делегирует все запросы (вызовы метода `HandleMark()`), объекту типа `State`, ссылка на который храниться в автосвойстве `State`.

Основная идея паттерна State заключается в том, чтобы ввести абстрактный класс `State` для представления различных состояний некоторого объекта, в данном примере для представления различных состояний объекта типа `Father`. Класс `State` предоставляет интерфейс для всех производных классов, реализующих различные состояния отца, т.е., объекта типа `Father`.

```
internal abstract class State
{
    internal virtual void HandleMark(Father father, Mark mark)
    {
        ChangeState(father, mark);
    }

    protected abstract void ChangeState(Father father, Mark mark);
}
```

В подклассах класса `State` реализовано поведение, характерное для каждого конкретного состояния.

```
// Нейтральное состояние (S0)
internal class NeutralState : State
{
    internal NeutralState()
    {
        Console.WriteLine("Отец в нейтральном состоянии:");
        Hope();
    }

    protected override void ChangeState(Father father, Mark mark)
    {
        switch (mark)
        {
            case Mark.Two:
                {
                    father.State = new PityState(); // S1
                    break;
                }
            case Mark.Five:
                {
                    father.State = new JoyState();  // S3
                    break;
                }
        }
    }

    private void Hope()  // y3
    {
        Console.WriteLine("Надеется на хорошие оценки.");
    }
}

// Состояние жалости (S1)
internal class PityState : State
{
    internal PityState()
    {
        Console.WriteLine("Отец в состоянии жалости:");
        Calm();
    }

    protected override void ChangeState(Father father, Mark mark)
    {
        switch (mark)
        {
            case Mark.Two:
                {
                    father.State = new AngerState(); // S4
                    break;
                }
            case Mark.Five:
                {
                    father.State = new NeutralState(); // S0
                    break;
                }
        }
    }

    private void Calm()  // y2

    {
```

```
        Console.WriteLine("Успокаивает сына.");
    }
}

// Состояние сильного гнева (S2)
internal class StrongAngerState : State
{
    internal StrongAngerState()
    {
        Console.WriteLine("Отец в состоянии сильного гнева:");
        BeatBelt();
    }

    protected override void ChangeState(Father father, Mark mark)
    {
        switch (mark)
        {
            case Mark.Two:
                {
                    father.State = new StrongAngerState(); // S2
                    break;
                }
            case Mark.Five:
                {
                    father.State = new NeutralState(); // S0
                    break;
                }
        }
    }

    private void BeatBelt()  // y0
    {
        Console.WriteLine("Бьет сына ремнем.");
    }
}

// Состояние радости (S3)
internal class JoyState : State
{
    internal JoyState()
    {
        Console.WriteLine("Отец в состоянии радости:");
        Joy();
    }

    protected override void ChangeState(Father father, Mark mark)
    {
        switch (mark)
        {
            case Mark.Two:
                {
                    father.State = new PityState(); // S1
                    break;
                }

            case Mark.Five:
                {
                    father.State = new StrongJoyState(); // S5
                    break;
                }
        }
```

```
    }

    private void Joy()  // y4
    {
        Console.WriteLine("Радуется успехам сына.");
    }
}

// Состояние гнева (S4)
internal class AngerState : State
{
    internal AngerState()
    {
        Console.WriteLine("Отец в состоянии гнева:");
        Scold();
    }

    protected override void ChangeState(Father father, Mark mark)
    {
        switch (mark)
        {
            case Mark.Two:
                {
                    father.State = new StrongAngerState(); // S2
                    break;
                }
            case Mark.Five:
                {
                    father.State = new NeutralState(); // S0
                    break;
                }
        }
    }

    private void Scold()  // y1
    {
        Console.WriteLine("Ругает сына.");
    }
}

// Состояние сильной радости (S5)
internal class StrongJoyState : State
{
    internal StrongJoyState()
    {
        Console.WriteLine("Отец в состоянии сильной радости:");
        Exult();
    }




    protected override void ChangeState(Father father, Mark mark)
    {
        switch (mark)
        {
            case Mark.Two:
                {
                    father.State = new PityState(); // S1
                    break;
```

```
                }
            case Mark.Five:
                {
                    father.State = new StrongJoyState(); // S5
                    break;
                }
        }
    }

    private void Exult()  // y5
    {
        Console.WriteLine("Ликует и гордится сыном.");
    }
}
```

См. Пример к главе: \020_State\004_StateMotivation

Применимость паттерна

Паттерн State рекомендуется использовать, когда:

- Поведение (методы) объекта зависит от состояния (полей) этого объекта и поведение должно изменяться во время выполнения.

- В телах методов встречаются условные конструкции, состоящие из множества ветвей, при этом выбор необходимой ветви зависит от состояния. Рекомендуется каждую такую ветвь представить отдельным классом, так как это позволит трактовать каждое состояние, как самостоятельный объект, который может изменяться независимо от других объектов.

Результаты

Паттерн State обладает следующими преимуществами:

- **Локализация поведения, зависящего от состояния.**
  Паттерн State, собирает в одном месте зависящее от состояния поведение и помещает поведение, ассоциированное с некоторым состоянием в отдельный объект. Соответственно, для добавления нового состояния и функции перехода потребуется создать новый класс `ConcreteState`.

- **Разделяемые объекты состояния.**
  Разные контексты `Context` могут разделять один и тот же объект типа `State`.

Паттерн Strategy

Название

Стратегия

Также известен как

Policy (политика)

Классификация

По цели: поведенческий
По применимости: к объектам

Частота использования

Выше средней - 1 2 3 **4** 5

Назначение

Паттерн Strategy - определяет набор алгоритмов (часто схожих по роду деятельности), инкапсулирует каждый из имеющихся алгоритмов (в отдельный класс) и делает их подменяемыми. Паттерн Strategy позволяет подменять алгоритмы без участия клиентов, которые используют эти алгоритмы.

Введение

Что такое стратегия в широком понимании? Стратегия - это план действий человека в жизненных или деловых ситуациях, которые могут возникнуть. Стратегию можно сравнить с компьютерным алгоритмом, который описывает действия, выполняемые для реализации плана. Стратегии принято разделять на два типа: *чистые стратегии* и *смешанные стратегии*.

- *Чистая стратегия* - описывает каким образом человек выполнит действия в определенной ситуации.
- *Смешанная стратегия* - описывает использование сочетаний, различных вариантов чистых стратегий, выбранных наугад или с определенной вероятностью.

В программировании, стратегия как способ действий становится необходимой в ситуации, когда для прямого достижения основной цели (выполнения программы) недостаточно наличных ресурсов (объема оперативной памяти, вычислительной мощности процессора или видеокарты). Соответственно, основной задачей стратегии является эффективное использование наличных ресурсов для достижения основной цели.

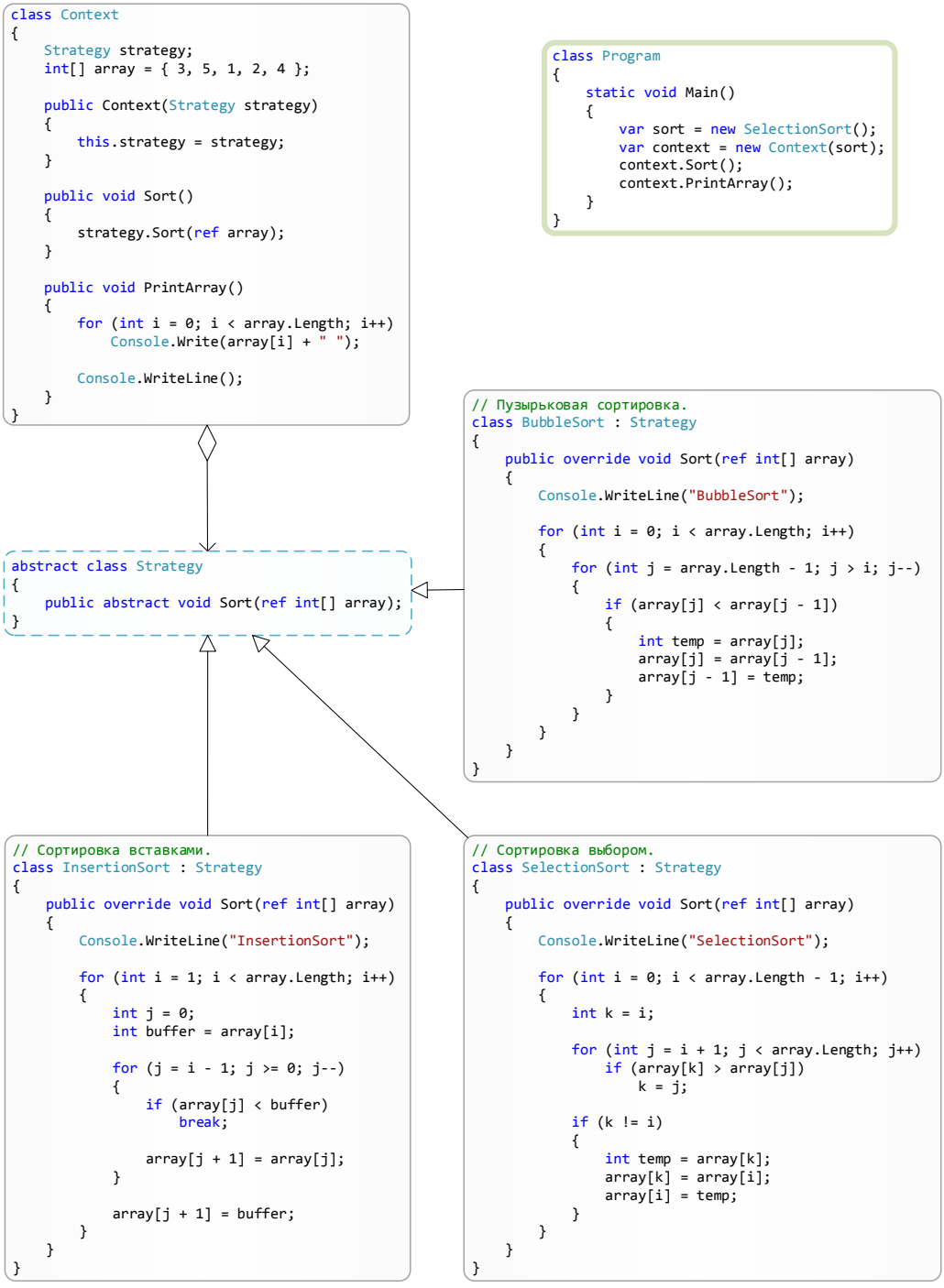
Важно понимать, что достижение основной цели подразумевает два совершенно разных вида деятельности - это *стратегия* и *тактика*.

- *Тактика* – это организация отдельных алгоритмов и выполнение их.
- *Стратегия* – это увязка алгоритмов с общей целью выполнения программы.

Правильно выбранная стратегия называется доминирующей стратегией. *Доминирование* - это ситуация, при которой одна из правильно выбранных стратегий дает больший выигрыш (в производительности) нежели другая. При выборе стратегии из множества всех допустимых стратегий, следует сравнить по предпочтительности исходы от применения имеющихся стратегий.

Принято различать *строгое (сильное) доминирование* и *слабое доминирование*. *Строгое доминирование* стратегии **A** над стратегией **B**, когда применение стратегии **A** дает больший выигрыш чем применение стратегии **B**. *Слабое доминирование* стратегии **A** над стратегией **B**, когда применение стратегии **A** дает меньший выигрыш чем применение стратегии **B**.

Понятие стратегии иногда (ошибочно) путают с понятием *шага*. Например, *шаг* можно сравнить с одиночным действием алгоритма сортировки в какой-то момент его работы (сравнение двух элементов массива и их перестановка). Стратегию можно сравнить с полным алгоритмом сортировки определенного типа (быстрая сортировка, сортировка Шелла, сортировка вставками, пузырьковая сортировка, сортировка выбором и т.д.) для выполнения сортировки всего массива. Алгоритмы сортировки оцениваются по скорости выполнения (время - основной параметр, характеризующий быстродействие алгоритма) и эффективности использования памяти (ряд алгоритмов требует выделения дополнительной памяти под временное хранение результатов промежуточных вычислений). Соответственно, основной задачей стратегии выбора алгоритма сортировки является эффективное использование аппаратных ресурсов вычислительной машины для достижения основной цели - упорядочивания элементов в списке.



```
class Context
{
    Strategy strategy;
    int[] array = { 3, 5, 1, 2, 4 };

    public Context(Strategy strategy)
    {
        this.strategy = strategy;
    }

    public void Sort()
    {
        strategy.Sort(ref array);
    }

    public void PrintArray()
    {
        for (int i = 0; i < array.Length; i++)
            Console.Write(array[i] + " ");

        Console.WriteLine();
    }
}
```

```
class Program
{
    static void Main()
    {
        var sort = new SelectionSort();
        var context = new Context(sort);
        context.Sort();
        context.PrintArray();
    }
}
```

```
abstract class Strategy
{
    public abstract void Sort(ref int[] array);
}
```

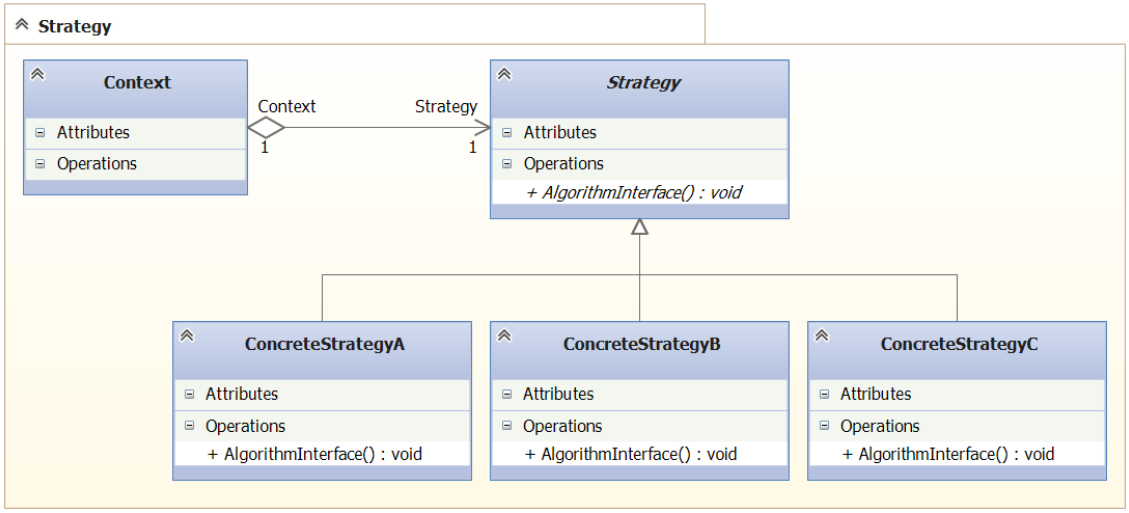
```
// Пузырьковая сортировка.
class BubbleSort : Strategy
{
    public override void Sort(ref int[] array)
    {
        Console.WriteLine("BubbleSort");

        for (int i = 0; i < array.Length; i++)
        {
            for (int j = array.Length - 1; j > i; j--)
            {
                if (array[j] < array[j - 1])
                {
                    int temp = array[j];
                    array[j] = array[j - 1];
                    array[j - 1] = temp;
                }
            }
        }
    }
}
```

```
// Сортировка вставками.
class InsertionSort : Strategy
{
    public override void Sort(ref int[] array)
    {
        Console.WriteLine("InsertionSort");

        for (int i = 1; i < array.Length; i++)
        {
            int j = 0;
            int buffer = array[i];

            for (j = i - 1; j >= 0; j--)
            {
                if (array[j] < buffer)
                    break;

                array[j + 1] = array[j];
            }

            array[j + 1] = buffer;
        }
    }
}
```

```
// Сортировка выбором.
class SelectionSort : Strategy
{
    public override void Sort(ref int[] array)
    {
        Console.WriteLine("SelectionSort");

        for (int i = 0; i < array.Length - 1; i++)
        {
            int k = i;

            for (int j = i + 1; j < array.Length; j++)
                if (array[k] > array[j])
                    k = j;

            if (k != i)
            {
                int temp = array[k];
                array[k] = array[i];
                array[i] = temp;
            }
        }
    }
}
```

См. Пример к главе: \021_Strategy\002_StrategySort

Структура паттерна на языке UML



См. Пример к главе: \021_Strategy\001_Strategy

Структура паттерна на языке C#

```
class Program
{
    static void Main()
    {
        Context context;
        context = new Context(new ConcreteStrategyA());
        context.ContextInterface();
        context = new Context(new ConcreteStrategyB());
        context.ContextInterface();
        context = new Context(new ConcreteStrategyC());
        context.ContextInterface();
    }
}
```

```
class Context
{
    Strategy strategy;

    public Context(Strategy strategy)
    {
        this.strategy = strategy;
    }

    public void ContextInterface()
    {
        strategy.AlgorithmInterface();
    }
}
```

```
abstract class Strategy
{
    public abstract void AlgorithmInterface();
}
```

```
class ConcreteStrategyA : Strategy
{
    public override void AlgorithmInterface()
    {
        Console.WriteLine("ConcreteStrategyA");
    }
}
```

```
class ConcreteStrategyB : Strategy
{
    public override void AlgorithmInterface()
    {
        Console.WriteLine("ConcreteStrategyB");
    }
}
```

```
class ConcreteStrategyC : Strategy
{
    public override void AlgorithmInterface()
    {
        Console.WriteLine("ConcreteStrategyC");
    }
}
```

См. Пример к главе: \021_Strategy\001_Strategy

Участники

- `Strategy` - **Стратегия**:
  Предоставляет интерфейс для реализаций конкретных стратегий.

- `ConcreteStrategy` - **Конкретная стратегия**:
  Классы конкретных стратегий реализуют интерфейс, предоставляемый классом `Strategy`. Реализации интерфейса, предоставляемого классом `Strategy` представляют собой алгоритмы (заданного вида деятельности) направленные на достижение определенной цели.

- `Context` - **Контекст**:
  Объект класса `Context` конфигурируется объектом класса `ConcreteStrategy`.

Отношения между участниками

Отношения между классами

- Конкретные классы `ConcreteStrategy` связаны связями отношения наследования с абстрактным классом `Strategy`.

- Конкретный класс `Context` связан связью отношения агрегации с абстрактным классом `Strategy`.

Отношения между объектами

- Классы `Context` и `Strategy` взаимодействуют друг с другом для реализации техники абстрагирования вариантов использования конкретных стратегий. Контекст может передавать определенной стратегии (`ConcreteStrategy`) все необходимые данные которые будут использоваться алгоритмом из этой стратегии. Также контекст может передавать стратегии ссылку на себя, а стратегия сможет получать доступ к необходимым членам контекста (но, такой подход не способствует построению повторно используемого дизайна, так как каждая стратегия будет делать предположение о контексте который ее использует).

- Обычно клиент создает экземпляр класса `Context` и конфигурирует его экземпляром класса одной из конкретных стратегий (`ConcreteStrategy`), после чего клиент общается только с контекстом, который переадресует запросы клиентов объекту стратегии.

Применимость паттерна

Паттерн Strategy рекомендуется использовать, когда:

- Имеется несколько классов (`Context`), отличающихся только поведением сходным по роду деятельности. Паттерн Strategy позволяет вынести каждое поведение в отдельный класс стратегию (`ConcreteStrategy`) и конфигурировать объект класса `Context` объектом класса `ConcreteStrategy`.

- Требуется иметь несколько вариантов алгоритма (сходных по роду деятельности). Например, можно реализовать два варианта алгоритма сортировки, один из которых требует больше времени для выполнения, а другой больше оперативной памяти для хранения промежуточных результатов вычислений.

Паттерн Template Method

Название

Шаблонный метод

Также известен как

-

Классификация

По цели: поведенческий
По применимости: к классам

Частота использования

Средняя - 1 2 **3** 4 5

Назначение

Паттерн Template Method - формирует структуру алгоритма и позволяет в производных классах реализовать, заместить (перекрыть) или переопределить определенные шаги (участки) алгоритма, не изменяя структуру алгоритма в целом.

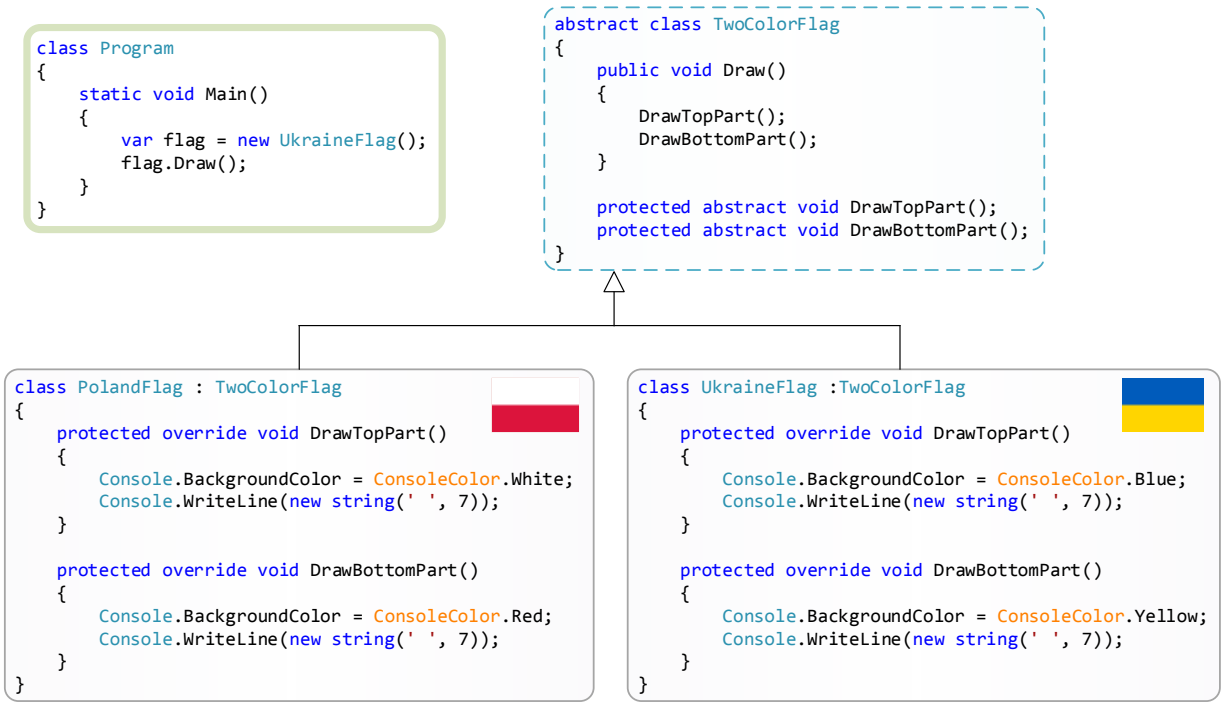
Введение

Одним из важнейших технологических приемов структурного программирования является декомпозиция (разбиение) решаемой задачи на подзадачи. Подзадача – это более простая с точки зрения программирования часть исходной задачи. Алгоритмы решения таких подзадач называются вспомогательными алгоритмами. В связи с этим возможны два пути в построении алгоритма: «сверху-вниз» и «снизу-вверх». Программирование методом «сверху-вниз» (или метод последовательной детализации) – процесс пошагового разбиения алгоритма на более мелкие части с целью получения таких элементов, для которых независимым образом можно легко написать вспомогательные алгоритмы (шаги общего алгоритма).

Для лучшего понимания идеи использования техники «шаблонный метод» предлагается рассмотреть простейшую программу, которая рисует двухцветные государственные флаги.

Польша Украина Гаити

Общий алгоритм рисования двухцветного флага разбивается на отдельные элементарные алгоритмы (шаги) по рисованию отдельной части флага. Пример приведенный ниже демонстрирует технику разбиения общего алгоритма по построению флага на два «подалгоритма», каждый из которых строит только часть флага.



```
class Program
{
    static void Main()
    {
        var flag = new UkraineFlag();
        flag.Draw();
    }
}
```

```
abstract class TwoColorFlag
{
    public void Draw()
    {
        DrawTopPart();
        DrawBottomPart();
    }

    protected abstract void DrawTopPart();
    protected abstract void DrawBottomPart();
}
```

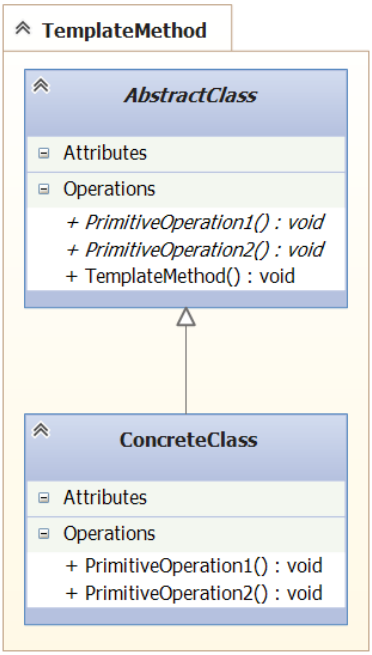
```
class PolandFlag : TwoColorFlag
{
    protected override void DrawTopPart()
    {
        Console.BackgroundColor = ConsoleColor.White;
        Console.WriteLine(new string(' ', 7));
    }

    protected override void DrawBottomPart()
    {
        Console.BackgroundColor = ConsoleColor.Red;
        Console.WriteLine(new string(' ', 7));
    }
}
```

```
class UkraineFlag :TwoColorFlag
{
    protected override void DrawTopPart()
    {
        Console.BackgroundColor = ConsoleColor.Blue;
        Console.WriteLine(new string(' ', 7));
    }

    protected override void DrawBottomPart()
    {
        Console.BackgroundColor = ConsoleColor.Yellow;
        Console.WriteLine(new string(' ', 7));
    }
}
```

См. Пример к главе: \022_Template Method\002_TemplateMethod

Как видно из примера кода, метод Draw из абстрактного класса TwoColorFlag содержит в себе шаблон (набор шагов как последовательность вызовов абстрактных методов DrawTopPart и DrawBottomPart) для рисования полос двухцветного флага, и программист который решит реализовать, например, двухцветный флаг Гаити  должен будет следовать шаблону вывода верхней и нижней полос флага, состоящему из двух шагов: нарисовать верхнюю часть флага синим цветом и нарисовать нижнюю часть флага красным цветом.

Структура паттерна на языке UML



См. Пример к главе: \022_Template Method\001_TemplateMethod

Структура паттерна на языке C#

```
class Program
{
    static void Main()
    {
        AbstractClass instance = new ConcreteClass();
        instance.TemplateMethod();
    }
}
```

```
abstract class AbstractClass
{
    public abstract void PrimitiveOperation1();
    public abstract void PrimitiveOperation2();

    public void TemplateMethod()
    {
        PrimitiveOperation1();
        PrimitiveOperation2();
    }
}
```

```
class ConcreteClass : AbstractClass
{
    public override void PrimitiveOperation1()
    {
        Console.WriteLine("PrimitiveOperation1");
    }

    public override void PrimitiveOperation2()
    {
        Console.WriteLine("PrimitiveOperation2");
    }
}
```

См. Пример к главе: \022_Template Method\001_TemplateMethod

Участники

- `AbstractClass` - **Абстрактный класс**:
  Содержит абстрактные методы `PrimitiveOperation1`, `PrimitiveOperation2` … `PrimitiveOperationN`, реализуемые в конкретных производных классах. Каждая реализация отдельного абстрактного метода `PrimitiveOperation1`, `PrimitiveOperation2` … `PrimitiveOperationN`, представляет собой один шаг общего алгоритма. Абстрактный класс `AbstractClass` содержит реализацию метода `TemplateMethod` в котором вызываются абстрактные методы `PrimitiveOperation1`, `PrimitiveOperation2` … `PrimitiveOperationN`, (через использование техники абстрагирования вариантов использования) в определенной последовательности, тем самым формируя структуру (скелет) общего алгоритма из более мелких алгоритмов представленных методами `PrimitiveOperation1`, `PrimitiveOperation2` … `PrimitiveOperationN`.

- `ConcreteClass` - **Конкретный класс**:
  Реализует абстрактные методы `PrimitiveOperation1`, `PrimitiveOperation2` … `PrimitiveOperationN`, из базового класса `AbstractClass`. Реализации абстрактных методов `PrimitiveOperation1`, `PrimitiveOperation2` … `PrimitiveOperationN`, представляют собой отдельные шаги общего алгоритма.

Отношения между участниками

Отношения между классами

- Конкретный класс `ConcreteClass` связан связью отношения наследования с абстрактным классом `AbstractClass`.

Отношения между объектами

- Конкретный класс `ConcreteClass` предполагает, что инвариантные (постоянные, не изменяющиеся в процессе эволюции системы) шаги алгоритма будут выполнены (через использование техники абстрагирования вариантов использования) в базовом классе `AbstractClass`.

Применимость паттерна

Паттерн Template Method рекомендуется использовать, когда:

- Требуется в базовом классе организовать неизменяемую последовательность шагов алгоритма, позволяя изменять реализацию каждого отдельного шага в производных классах.

- Требуется в базовом классе описать последовательность шагов алгоритма общую для всех подклассов, с целью избежать дублирования кода. Такой подход является примером техники «вынесения за скобки с целью обобщения». Под «вынесением за скобки» подразумевается вынесение за пределы метода `TemplateMethod` или даже класса функциональности, которая будет располагаться в реализациях методов `PrimitiveOperation1` и `PrimitiveOperation2`. Под «целью обобщения» подразумевается наследование.

Результаты

Использование шаблонных методов является одним из фундаментальных приемов повторного использования кода. Применение паттерна Template Method особенно важно в библиотеках классов и каркасах (framework), поскольку этот паттерн дает разработчикам возможность вынести общее поведение (последовательность шагов алгоритма) в библиотечные классы.

Использование шаблонных методов приводит к инвертированной (вывернутой) структуре программного кода. Такой эффект «*вывернутости*» проявляется в том, что в базовом классе `AbstractClass` в методе `TemplateMethod` происходит вызов абстрактных методов `PrimitiveOperation1` и `PrimitiveOperation2` (благодаря технике абстрагирования вариантов использования), но реализация этих методов находится в производных классах `ConcreteClass`. Такую «вывернутую» структуру кода иногда называют принципом Голливуда. Подразумевается, что в Голливуде известных режиссеров мало, а вот актеров, которые хотят получить роль в перспективной кинокартине очень много, соответственно, если каждый актер будет звонить режиссерам и предлагать себя на роль, то режиссеру придётся целый день принимать звонки. Поэтому по правилам Голливуда, не приветствуется, когда актер сам «напрашивается на роль», отсюда и принцип – «Не звоните нам, мы сами позвоним.» (так говорят режиссёры). В данной метафоре абстрактный класс `AbstractClass` – режиссер, метод `TemplateMethod` – кинокартина (сценарий), конкретный класс `ConcreteClass` – сообщество актеров, а методы `PrimitiveOperation1` и `PrimitiveOperation2` – сами актеры. Применительно к паттерну Template Method метафора означает, что базовый класс `AbstractClass` вызывает методы `PrimitiveOperation1` и `PrimitiveOperation2` из производного класса `ConcreteClass`, но не наоборот. Понятно, что в Голливуде одним режиссером ставится некоторое количество кинокартин по сценариям (`TemplateMethod`), и соответственно, как в разных фильмах может играть один актер, так и в случае с методами `PrimitiveOperation1` и `PrimitiveOperation2`, они могут быть вызваны из разных шаблонных методов, например, `TemplateMethod1`, `TemplateMethod2`, `TemplateMethod3`.

Шаблонный метод `TemplateMethod` может вызывать:

- Реализации примитивных операций (реализаций шагов алгоритма) `PrimitiveOperation1` и `PrimitiveOperation2` из производного класса `ConcreteClass`.

- Реализации примитивных операций (реализаций шагов алгоритма) `PrimitiveOperation1` и `PrimitiveOperation2` из того же класса `AbstractClass`.

- Абстрактные методы `PrimitiveOperation1` и `PrimitiveOperation2` используя при этом технику абстрагирования вариантов использования.

- Фабричные методы как абстрактные, так и реализации.

Реализация

Полезные приемы реализации паттерна Template Method:

- **Использование контроля доступа в C#.**
  Под контролем доступа, следует понимать использование невиртуального высокоуровневого интерфейса для работы с виртуальным низкоуровневым интерфейсом. Под невиртуальным интерфейсом следует понимать использование метода `TemplateMethod`, который вызывает методы `PrimitiveOperation1` и `PrimitiveOperation2`, которые рекомендуется по возможности создавать как виртуальные и защищенные (помечать ключевыми словами `virtual` и `protected`), в таком случае гарантируется, что их сможет вызвать только метод `TemplateMethod`.

- **Сокращение числа доступных клиенту «низкоуровневых» методов.**
  Важной целью при проектировании шаблонных методов `TemplateMethod`, является сокращение числа доступных клиенту простых методов `PrimitiveOperation1` и `PrimitiveOperation2`, которые могли бы усложнить собой работу клиента, тем самым избавляя клиента от лишних знаний о работе низкоуровневых операций.

- **Соглашение об именах.**
  В языке C# нет особых рекомендаций для включения в сигнатуру создаваемого шаблонного метода специальных префиксов или постфиксов, для того чтобы оттенить принадлежность шаблонного метода к категории «шаблонных».

Паттерн Visitor

Название

Посетитель

Также известен как

Walker (Бродяга)

Классификация

По цели: поведенческий
По применимости: к объектам

Частота использования

Низкая - **1** 2 3 4 5

Назначение

Паттерн Visitor – позволяет единообразно обойти набор элементов с разнородными интерфейсами (т.е. набор объектов разных классов не приводя их к общему базовому типу), а также позволяет добавить новый метод (функцию) в класс объекта, при этом не изменяя сам класс этого объекта.

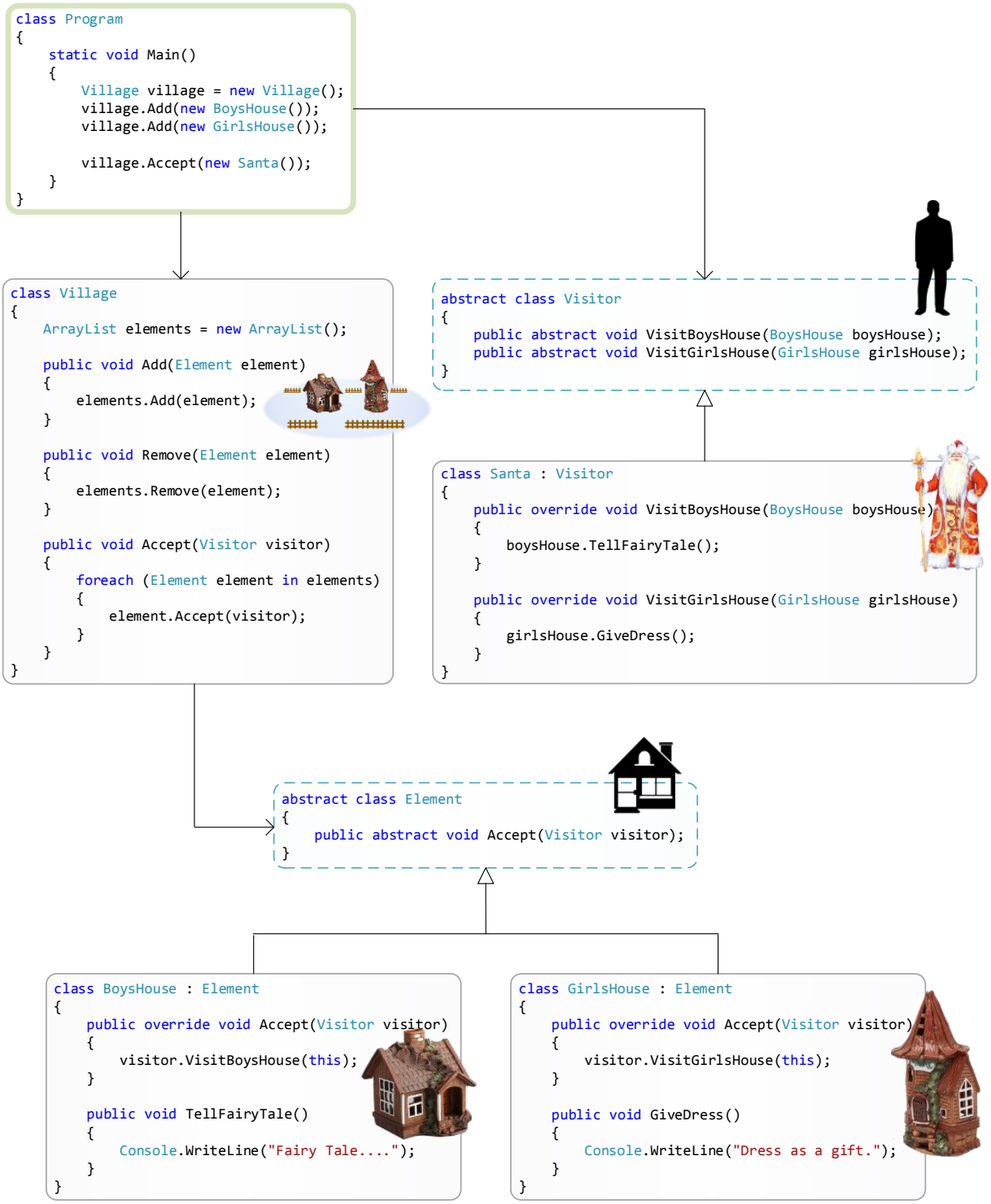
Введение

Предлагается начать рассмотрение паттерна Visitor с использованием метафоры. На минуту заставим себя поверить в существование такого сказочного персонажа как Дед Мороз, который в Новогоднюю ночь поочередно посещает домики в которых живут дети и дарит им подарки. На рисунке ниже, представлена деревня (объектная структура) в которой имеется только два домика (Элемент А и Элемент Б), в одном из них живет мальчик, а в другом девочка.

Понятно, что деревню можно программно представить, как коллекцию домиков. В деревне могут появляться новые домики, а старые ветшать и удаляться из коллекции.

Задача Деда Мороза посетить каждый домик и исполнить желания каждого ребенка (кто что пожелает), другими словами выполнить определенные операции в определенном домике. Например, Дед Мороз мальчику расскажет «волшебную сказку» а девочке подарит «платье прекрасной голубой феи».

Из сказанного выше легко сформировать модель и реализовать ее программно.



См. Пример к главе: \023_Visitor\002_NewYear

Хотелось бы обратить внимание на несколько технических особенностей, которые могли бы смутить читателя в процессе рассмотрения примера.

Возникает, справедливый на первый взгляд вопрос: Почему методы `VisitBoysHouse` и `VisitGirlsHouse` в классе `Visitor` используют в качестве параметра типы конкретных классов `BoysHouse` и `GirlsHouse`?

```
abstract class Visitor
{
    public abstract void VisitBoysHouse(BoysHouse boysHouse);
    public abstract void VisitGirlsHouse(GirlsHouse girlsHouse);
}
```

Ведь если, мы будем использовать абстрактный класс `Element` в качестве типа параметра, то так будет правильней и красивее. Но тут возникает проблема, что после приведения к базовому типу `Element`, нам становятся недоступными для использования операции конкретных элементов (домиков).

```
abstract class Visitor
{
    public abstract void VisitBoysHouse(Element element);
    public abstract void VisitGirlsHouse(Element element);
}
```

```
class Santa : Visitor
{
    public override void VisitBoysHouse(Element element)
    {
        element.TellFairyTale();
    }

    public override void VisitGirlsHouse(Element element)
    {
        element.GiveDress();
    }
}
```

Въедливые умы сразу могут предложить создать абстрактные методы `TellFairyTale` и `CiveDress` в базовом абстрактном классе `Element`, реализовать их в производных классах и код успешно выполниться. Технически организовать такую реализацию не составит труда и пример ниже демонстрирует такой подход.

```
abstract class Element
{
    public abstract void Accept(Visitor visitor);
    public abstract void TellFairyTale();
    public abstract void GiveDress();
}
```

```
class BoysHouse : Element
{
    public override void Accept(Visitor visitor)
    {
        visitor.VisitBoysHouse(this);
    }

    public void TellFairyTale()
    {
        Console.WriteLine("Fairy Tale....");
    }

    public void GiveDress()
    {
        Мальчик в платье голубой феи?!
    }
}
```

```
class GirlsHouse : Element
{
    public override void Accept(Visitor visitor)
    {
        visitor.VisitGirlsHouse(this);
    }

    public void GiveDress()
    {
        Console.WriteLine("Dress as a gift.");
    }

    public void TellFairyTale()
    {
        Девочка ненавидит сказки!
    }
}
```

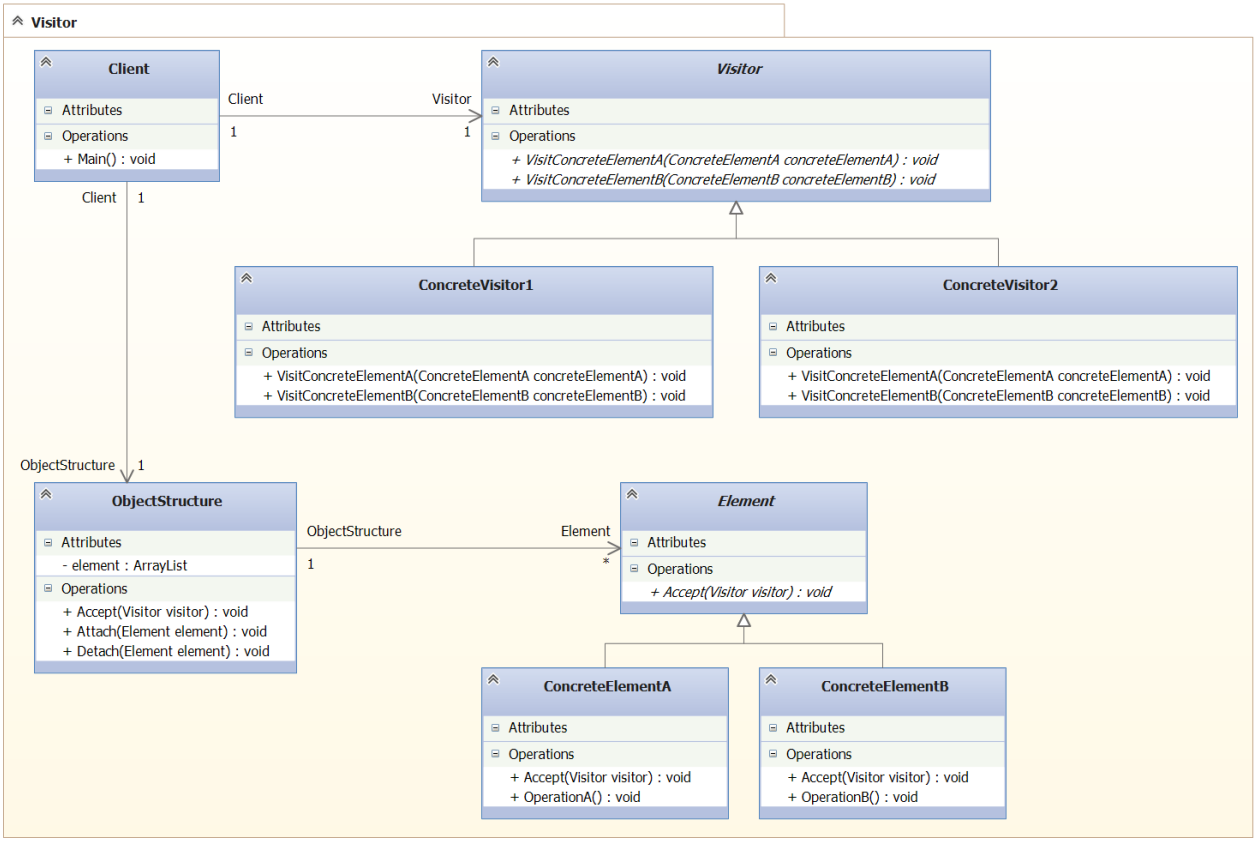
Как видно из примера, мальчик и девочка расширили свой интерфейс взаимодействия с Дедом Морозом, и остается задуматься над реализацией абстрактных методов. Мальчик теперь имеет возможность получить в подарок платье прекрасной голубой феи, а девочка имеет возможность слушать сказки которые она у Деда Мороза не заказывала и вообще не любит сказок (но это еще терпимо). Известно, что добавлять не нужную функциональность тому или иному объекту, не является успешным подходом при построении объектно-ориентированных систем.

Более того, важно то, что оказался испорчен интерфейс взаимодействия с объектами типа `Element`, теперь интерфейс взаимодействия компрометирует абстракцию типа. Аналогично, если читатель являясь мужчиной, публично перед коллегами и друзьями выразит готовность принять в подарок на новый год платье прекрасной голубой феи (и обязательно готовность предстать в нем). Это и есть компрометация

абстракции типа «настоящий мужчина». Компрометация абстракции влечет за собой нарушение концептуальной целостности системы и так далее по наклонной. Поэтому авторы паттерна Visitor отказались от использования подхода с обобщением поведений (методов) классов `BoysHouse` и `GirlsHouse`, и нельзя назвать существующий подход с использование конкретных классов красивым, удобным и способствующим повторному использованию, но такой подход можно назвать вынужденным.

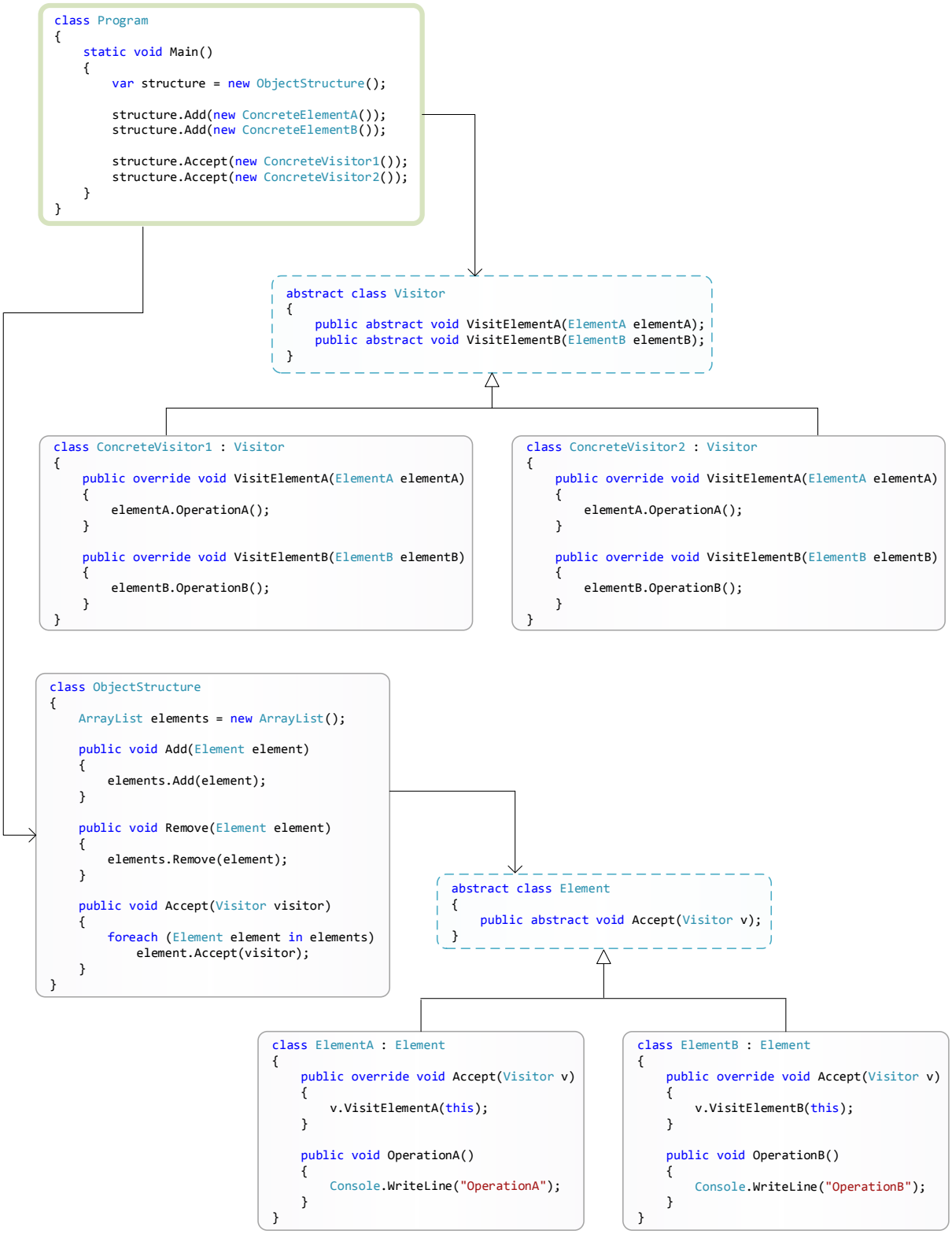
Таким образом из примера видно, что посетитель (в нашем случае Дед Мороз) единообразно обошел элементы (домики) из объектной структуры (деревни), несмотря на наличие разнородности в интерфейсах элементов (домиков) `BoysHouse` и `GirlsHouse`, а это и является одной из важнейших задач которую помогает решить использование паттерна Visitor.

Структура паттерна на языке UML



См. Пример к главе: \023_Visitor\001_Visitor

Структура паттерна на языке C#



См. Пример к главе: \023_Visitor\001_Visitor

Участники

- `Visitor` - **Посетитель**:
  Предоставляет абстрактный интерфейс (набор методов `VisitConcretElementX`) для работы с объектами класса `ConcreteElementX`. Имя метода `VisitConcretElementX` включает в себя имя класса, экземпляр которого вызывает данный метод.

- `ConcreteVisitor` - **Конкретный посетитель**:
  Реализует абстрактный интерфейс, предоставляемый абстрактным классом `Visitor`. Каждая операция `VisitConcretElementX` реализует фрагмент алгоритма, специфичного для каждого отдельного класса `ConcreteElement`.

- `Element` - элемент:
  Предоставляет абстрактный метод `Accept`, который принимает аргумент типа `Visitor`.

- `ConcreteElement` - **Конкретный элемент**:
  Реализует абстрактный метод `Accept`, который принимает аргумент типа `Visitor`.

- `ObjectStructure` - **Структура объектов**:
  Представляет собой набор объектов типа `Element`. Может быть, как обычной коллекцией, так и древообразной структурой.

Отношения между участниками

Отношения между классами

- Конкретные классы `ConcreteVisitor` связаны связью отношения наследования с абстрактным классом `Visitor`.

- Конкретные классы `ConcreteElement` связаны связью отношения наследования с абстрактным классом `Element`.

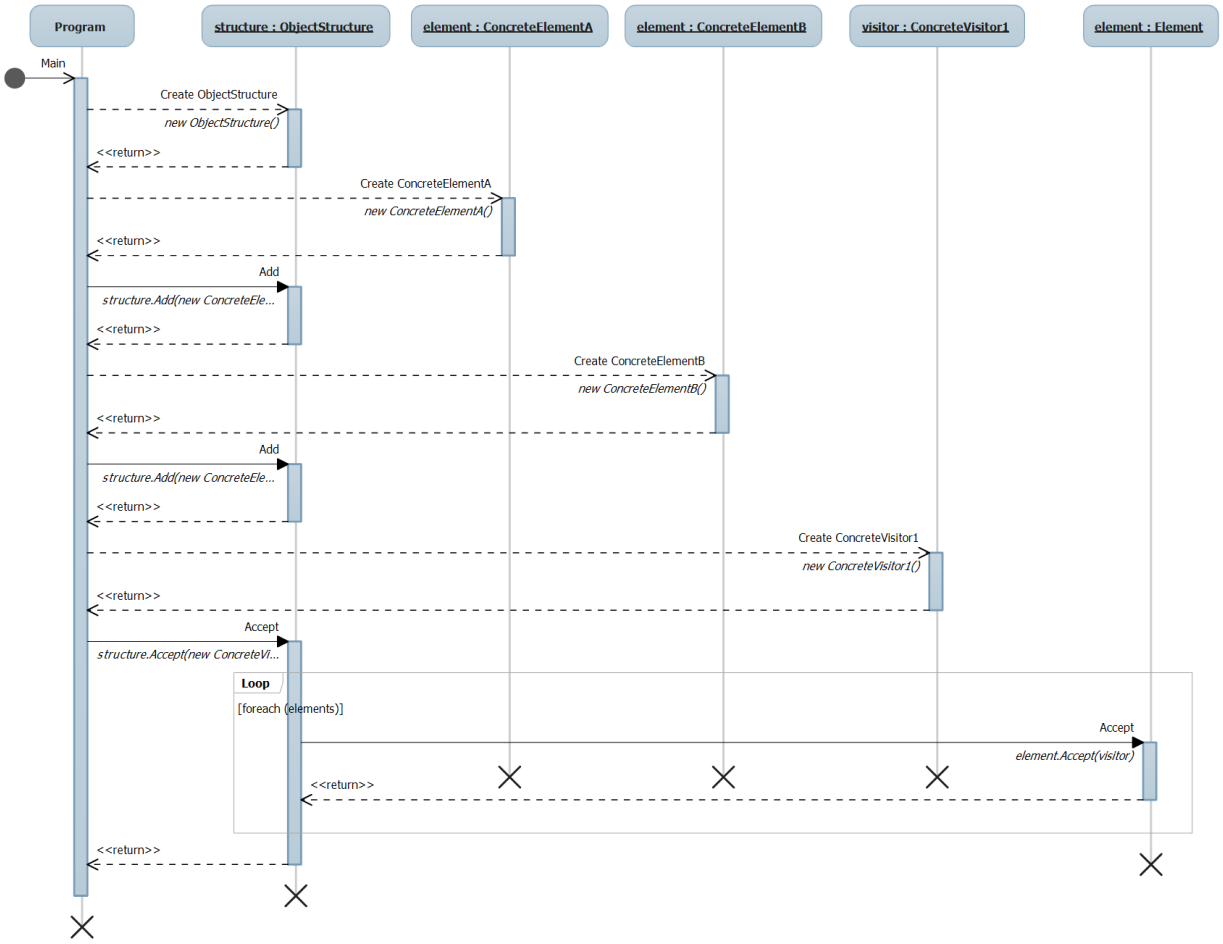
- Конкретный класс `ObjectStructure` связан связью отношения ассоциации с абстрактным классом `Element`.

Отношения между объектами

- Клиент использующий паттерн Visitor должен создать экземпляр класса `ConcreteVisitor` и с его помощью обойти каждый элемент объектной структуры (коллекции).

- При посещении экземпляром класса `ConcreteVisitor` определенного элемента (экземпляра класса `ConcreteElement`) из объектной структуры, этот элемент вызывает на посетителе метод `VisitConcretElementX`, соответствующий классу данного элемента. Элемент передает этому методу себя в качестве аргумента, чтобы посетитель мог получить доступ к членам (состоянию и поведению) данного элемента.

На представленной ниже диаграмме взаимодействия показаны отношения между объектами: клиентом (`Program`), объектной структурой (`ObjectStructure`), посетителем (`ConcreteVisitor`) и двумя элементами (`ConcreteElementA` и `ConcreteElementB`).



См. Пример к главе: \023_Visitor\001_Visitor

Применимость паттерна

Паттерн Visitor рекомендуется использовать, когда:

- В (гетерогенной) коллекции (`ObjectStructure`) должны присутствовать разнотипные объекты (`ConcreteElementA` и `ConcreteElementB`) с разнородными интерфейсами и при этом требуется организовать унифицированный обход элементов этой коллекции и выполнить определенные операции над каждым имеющимся объектом.

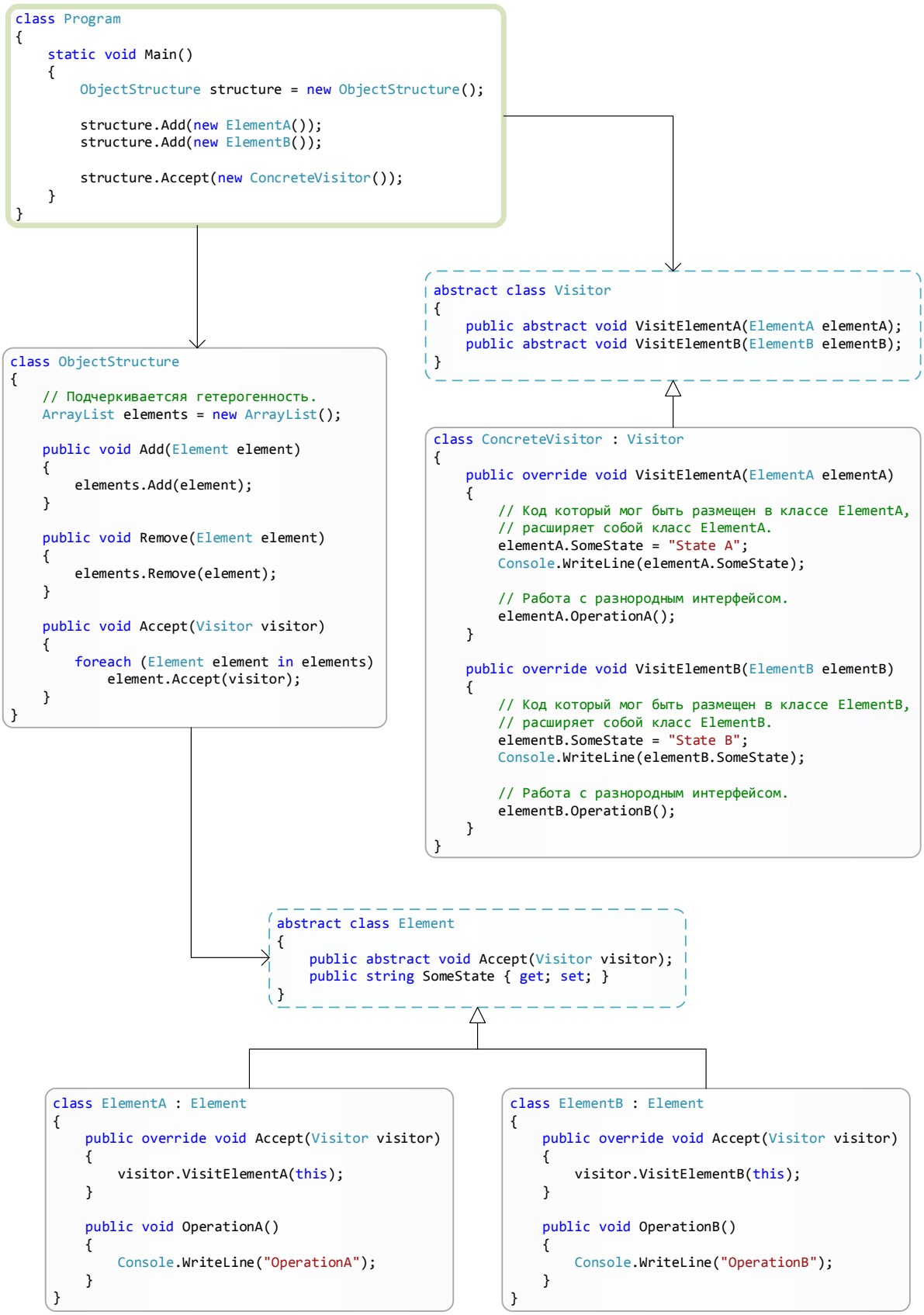
```
class ObjectStructure
{
    // Подчеркивается гетерогенность.
    ArrayList elements = new ArrayList();

    public void Add(Element element)
    {
        elements.Add(element);
    }

    public void Remove(Element element)
    {
        elements.Remove(element);
    }

    public void Accept(Visitor visitor)
    {
        foreach (Element element in elements)
            element.Accept(visitor);
    }
}
```

- Над объектами (ConcreteElementA и ConcreteElementB) входящими в состав коллекции (ObjectStructure) требуется выполнять определенные операции и при этом не хотелось бы повторять эти операции в каждом классе (ConcreteElementA и ConcreteElementB). Код этих операций можно вынести в методы объекта посетителя класса ConcreteVisitor.

```
class ConcreteVisitor : Visitor
{
    public override void VisitElementA(ElementA elementA)
    {
        // Код который мог быть размещен в классе ElementA,
        // расширяет собой класс ElementA.
        elementA.SomeState = "State A";
        Console.WriteLine(elementA.SomeState);

        // Работа с разнородным интерфейсом.
        elementA.OperationA();
    }

    public override void VisitElementB(ElementB elementB)
    {
        // Код который мог быть размещен в классе ElementB,
        // расширяет собой класс ElementB.
        elementB.SomeState = "State B";
        Console.WriteLine(elementB.SomeState);

        // Работа с разнородным интерфейсом.
        elementB.OperationB();
    }
}
```

- Классы объектов (ConcreteElementA и ConcreteElementB), входящих в коллекцию (ObjectStructure) изменяются редко, но новые операции, производимые над коллекцией требуется добавлять часто. Важно понимать, что при изменении интерфейса классов объектов (ConcreteElementA и ConcreteElementB), входящих в состав коллекции, вероятней всего потребуется переопределить и интерфейсы всех объектов посетителей (ConcreteVisitor1 и ConcreteVisitor2), а это может быть затруднительно. Поэтому, если классы (ConcreteElementA и ConcreteElementB) изменяются часто, то лучше все операции определять прямо в них (т.е., не выносить операции в посетителей).

Пример отображающий идеи применимости паттерна Visitor, рассмотренные выше.



```
class Program
{
    static void Main()
    {
        ObjectStructure structure = new ObjectStructure();

        structure.Add(new ElementA());
        structure.Add(new ElementB());

        structure.Accept(new ConcreteVisitor());
    }
}
```

```
class ObjectStructure
{
    // Подчеркивается гетерогенность.
    ArrayList elements = new ArrayList();

    public void Add(Element element)
    {
        elements.Add(element);
    }

    public void Remove(Element element)
    {
        elements.Remove(element);
    }

    public void Accept(Visitor visitor)
    {
        foreach (Element element in elements)
            element.Accept(visitor);
    }
}
```

```
abstract class Visitor
{
    public abstract void VisitElementA(ElementA elementA);
    public abstract void VisitElementB(ElementB elementB);
}
```

```
class ConcreteVisitor : Visitor
{
    public override void VisitElementA(ElementA elementA)
    {
        // Код который мог быть размещен в классе ElementA,
        // расширяет собой класс ElementA.
        elementA.SomeState = "State A";
        Console.WriteLine(elementA.SomeState);

        // Работа с разнородным интерфейсом.
        elementA.OperationA();
    }

    public override void VisitElementB(ElementB elementB)
    {
        // Код который мог быть размещен в классе ElementB,
        // расширяет собой класс ElementB.
        elementB.SomeState = "State B";
        Console.WriteLine(elementB.SomeState);

        // Работа с разнородным интерфейсом.
        elementB.OperationB();
    }
}
```

```
abstract class Element
{
    public abstract void Accept(Visitor visitor);
    public string SomeState { get; set; }
}
```

```
class ElementA : Element
{
    public override void Accept(Visitor visitor)
    {
        visitor.VisitElementA(this);
    }

    public void OperationA()
    {
        Console.WriteLine("OperationA");
    }
}
```

```
class ElementB : Element
{
    public override void Accept(Visitor visitor)
    {
        visitor.VisitElementB(this);
    }

    public void OperationB()
    {
        Console.WriteLine("OperationB");
    }
}
```

См. Пример к главе: \023_Visitor\003_Visitor

Результаты

Паттерн Visitor обладает следующими преимуществами:

- **Упрощение добавления новых методов.**
  Используя объекты-посетители, легко добавлять новую функциональность объектам-элементам. Понятно, что функциональность, предназначенная для объектов-элементов, физически будет находиться в методах объекта-посетителя, создавая иллюзию расширения самого класса элементов новой функциональностью. Для добавления новой операции (функциональности) всем или некоторым объектам-элементам, потребуется создать новый класс посетителя или изменить один из существующих классов посетителей.

- **Объединение сходного поведения.**
  Сходное поведение (функциональность) относящееся к объектам-элементам, не разносится по всем классам (`ConcreteElement`), а локализуется (размещается) в классе объекта-посетителя. Не связанная друг с другом функциональность распределяется по отдельным конкретным классам (`ConcreteVisitor`). Такой подход позволяет упростить как сами классы элементов (`ConcreteElement`), так и алгоритм, располагающийся внутри посетителя (все данные которые относятся к алгоритму можно располагать непосредственно в посетителе, т.е. рядом с алгоритмом).

- **Накопление состояния.**
  Посетитель может накапливать в себе информацию о состоянии элементов, входящих в объектную структуру. Если не использовать паттерн Visitor, то состояние придется передавать в виде дополнительных (`ref`/`out`) параметров методов, выполняющих обход и сохранять в специально отведенном месте, что может вызвать определенные неудобства.

Паттерн Visitor обладает следующими недостатками:

- **Сложность добавления новых классов `ConcreteElement`.**
  При использовании паттерна Visitor, возникают некоторые сложности при добавлении новых классов элементов (`ConcreteElement`). Связано это с тем, что создание нового класса `ConcreteElement`, требует добавления нового абстрактного метода `VisitConcretElementX` в абстрактный класс `Visitor`, этот абстрактный метод потребуется реализовать во всех производных классах `ConcreteVisitor`.
  При принятии решения об использования паттерна Visitor, потребуется определиться, что чаще будет добавляться в подсистему: алгоритм (новая функциональность) применяемый к элементам (`ConcreteElement`) входящим в объектную структуру или сами классы этих элементов (`ConcreteElement`). Если чаще будет добавляться алгоритм, то паттерн Visitor, поможет лучше управлять такими изменениями. Если же, планируется часто добавлять новые классы элементов (`ConcreteElement`), то вероятно, что такую модель классов сопровождать будет не удобно, но вполне возможно.

- **Разрушение инкапсуляции.**
  При работе с элементами (`ConcreteElement`), объект-посетитель в большинстве случаев должен знать их внутреннее устройство, для того чтобы справиться со своей работой по расширению функциональности. Поэтому при использовании паттерна Visitor, требуется организовывать более широкий чем мог бы быть, открытый интерфейс взаимодействия с объектами (`ConcreteElement`), что естественно, разрушает инкапсуляцию.

Реализация

С каждым объектом-элементом класса (ConcreteElement), ассоциирован некий класс (ConcreteVisitor) объекта-посетителя. В классах объектов посетителей (ConcreteVisitor), реализована операция VisitConcretElement, для каждого конкретного класса ConcreteElement. В каждой операции VisitConcretElement, имеется аргумент одного из классов ConcreteElement, благодаря этому посетитель может получить доступ к открытому (public) интерфейсу класса ConcreteElement. Классы ConcreteVisitor реализуют абстрактные методы VisitConcretElement из базового класса Visitor, с целью реализации в посетителе специфического поведения (функциональности), предназначенного для соответствующего класса ConcreteElement.
Каждый класс ConcreteElement реализует метод Accept, который вызывает соответствующий метод VisitConcretElement, на посетителе. Следовательно, какая будет в конечном итоге вызвана операция: OperationA или OperationB, зависит как от класса элемента ConcreteElement, так и от класса посетителя ConcreteVisitor.
Можно было бы использовать перегрузку методов, и вместо методов с разными оттенками имен VisitConcretElementX и VisitConcretElementY, воспользоваться методами с одним именем - Visit. Имеется два мнения по поводу использования перегрузки методов, одно за, другое против. Сторонники перегрузки подчеркивают, что все методы выполняют однотипную деятельность, хоть и имеют разные аргументы. Противники перегрузки бояться, что читателю программы будет затруднительно понимать, что именно происходит (кого требуется посетить) при вызове метода с именем Visit. Считаются оба варианта приемлемыми (допустимыми), применимость перегрузки зависит от предпочтений программиста.

Известные применения паттерна в .Net

Паттерн Visitor, выражен в платформе .Net в виде идеи использования расширяющих методов.

Библиография

Гамма, Э., Хелм, Р., Джонсон, Р., Влиссидес, Д. (1995). *Приемы объектно-ориентированного проектирования. Паттерны проектирования.* Addison Wesley Longman Inc.

Рихтер, Д. (2010). *CLR via C#.* Microsoft Press.

Хейлсберг, А., Торгерсен, М., Вилтамут, С., Голд, П. (2012). *C# Programming Language.* Издательский дом «Питер».

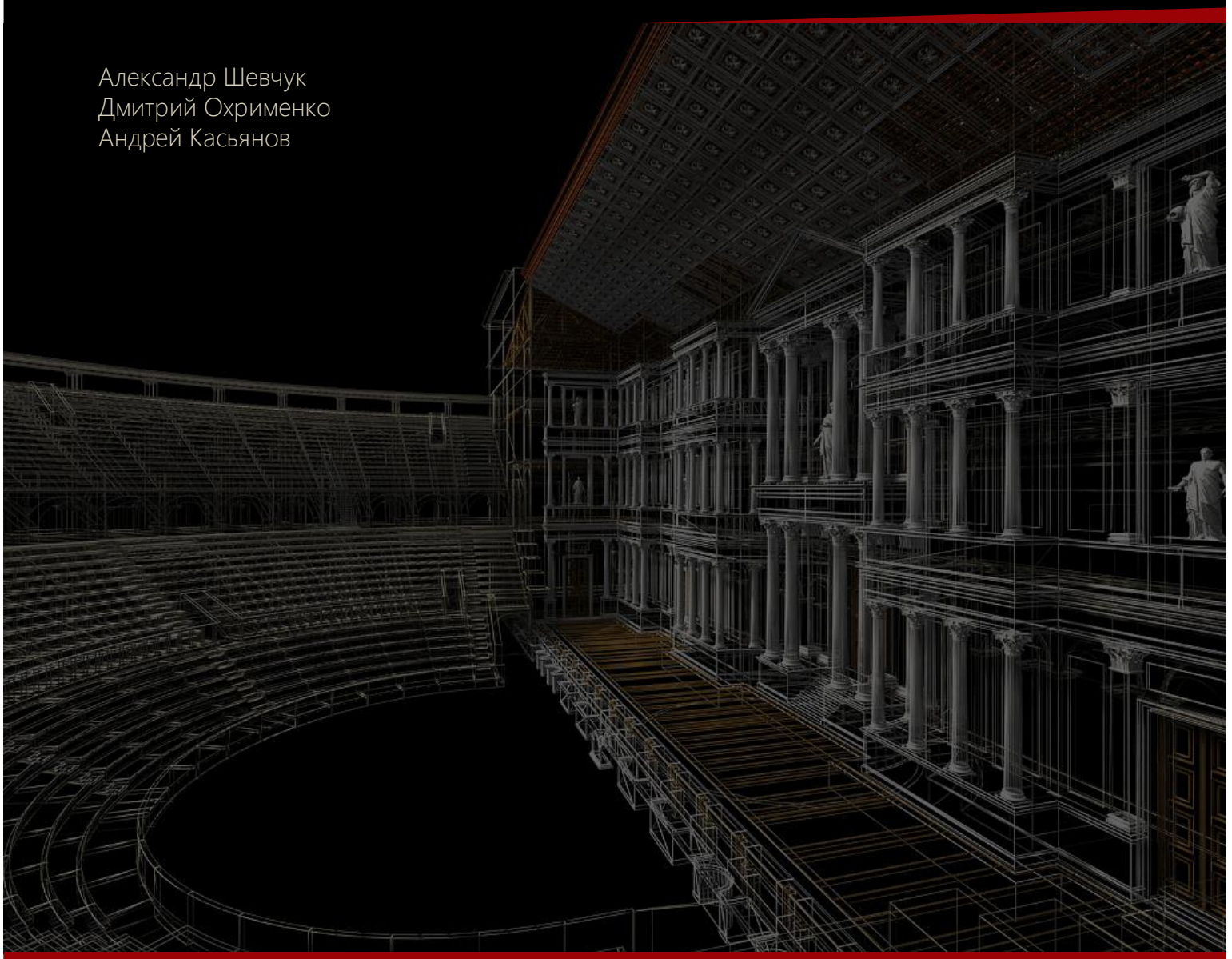
Цвалина, К., Абрамс, Б. (2011). *Инфраструктура программных проектов. Соглашения, идиомы и шаблоны для многократно используемых библиотек .NET.* Издательство Вильямс.

Карпов, Ю. (2003). *Теория автоматов.* Издательский дом «Питер».

Книга «Design Patterns via C#» не является самостоятельным изданием, описывающим паттерны проектирования, на эту тему уже есть уникальное издание: «Приемы объектно-ориентированного проектирования. Паттерны проектирования», авторами которого являются Эрих Гамма, Ричард Хелм, Ральф Джонсон и Джон Влиссидес. Эта группа авторов известна под творческим псевдонимом - «Банда четырех» (GoF – Gang of Four).

Цели, которые перед собой ставили авторы книги «Design Patterns via C#» при ее написании: Разъяснить и в хорошем смысле более «просторечиво» представить определения и положения, представленные в книге «Приемы объектно-ориентированного проектирования. Паттерны проектирования». Реализовать примеры на языке C# из книги «Приемы объектно-ориентированного проектирования. Паттерны проектирования», которые в книге представлены в форме общих описаний (примеров-идей) или в виде отрывков-кода на языке C++, стараясь при этом максимально сохранить первоначально заложенный смысл-идею. Представить модели диаграммами с использованием языка UML и выразить их средствами моделирования Microsoft Visual Studio. Показать варианты реализации паттернов с использованием особенностей конструкций языка C#, типов FCL и механизмов CLR. Книгу «Design Patterns via C#» рекомендуется воспринимать как приложение к книге «Приемы объектно-ориентированного проектирования. Паттерны проектирования» и, читать параллельно, обращаясь к ней за разъяснениями и описанием реализации примеров на языке C#.

Александр Шевчук
Дмитрий Охрименко
Андрей Касьянов



Design Patterns via C#

Приемы объектно-ориентированного проектирования

ITVDN.com