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.

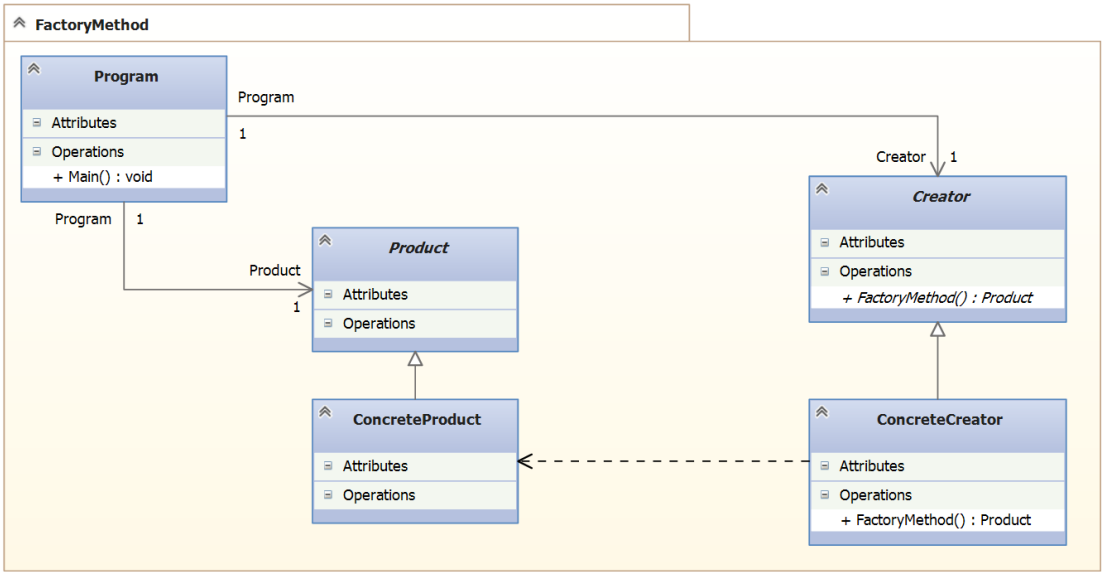
Введение

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`.

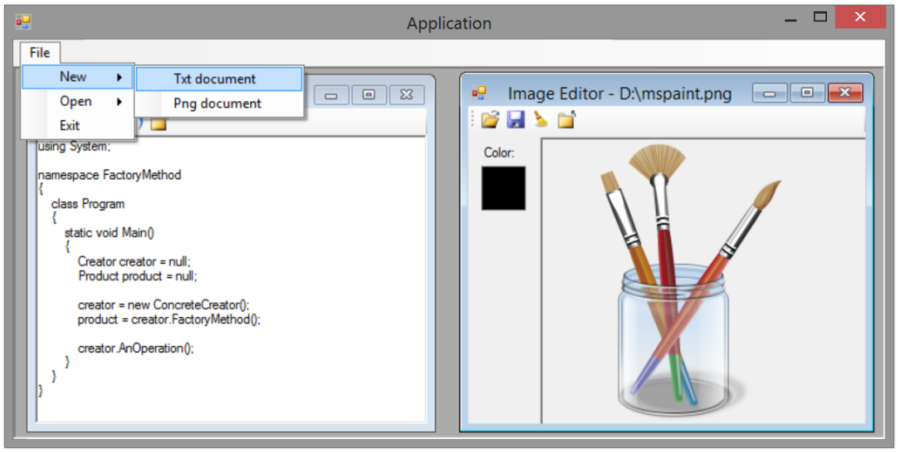
- Класс `ConcreteCreator` связан связью отношения наследования с абстрактным классом `Creator` и связью отношения зависимости с классом порождаемого продукта `ConcreteProduct`.

Отношения между объектами

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

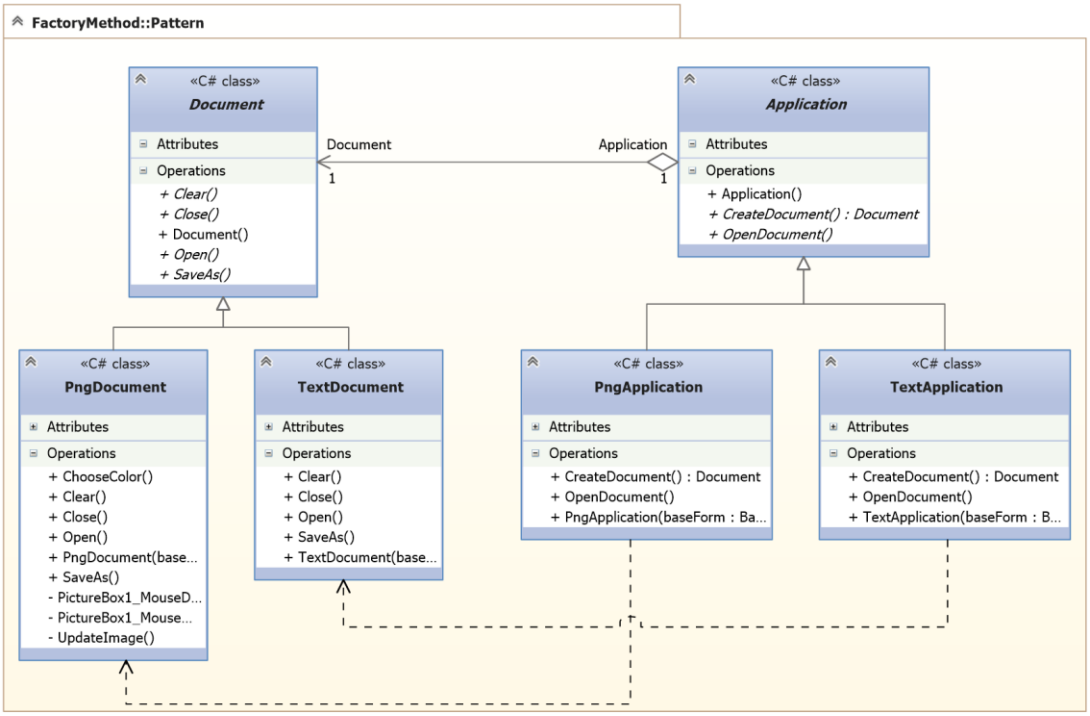
Мотивация

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

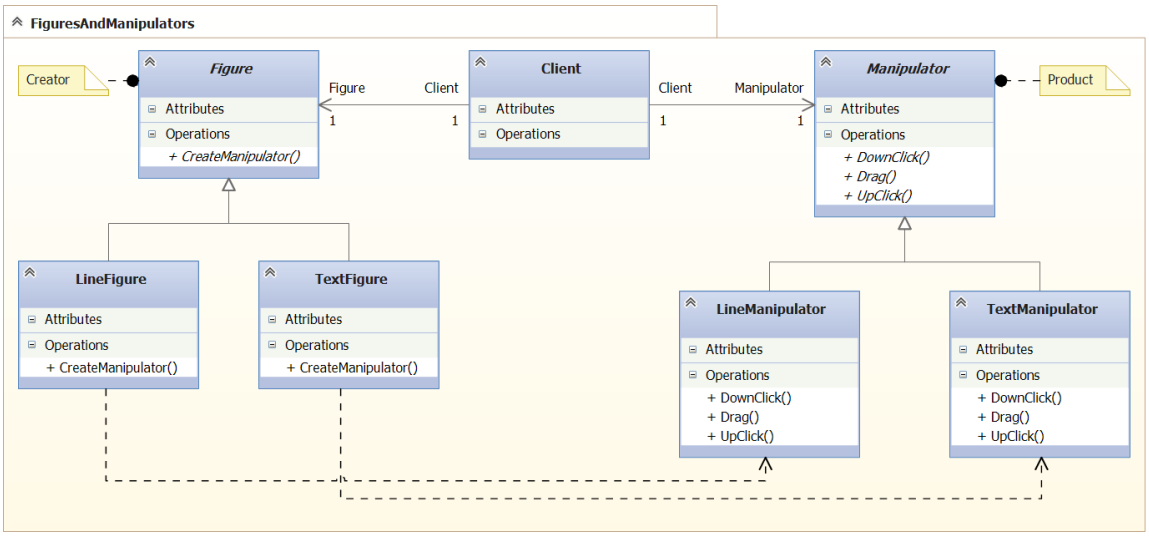
Использование фабричных методов избавляет от необходимости использования конструкторов экземпляров для создания объектов-продуктов непосредственно в месте использования этих объектов-продуктов. Таким образом имеется возможность работать с абстрактным интерфейсом класса `Product`, что в свою очередь позволяет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