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

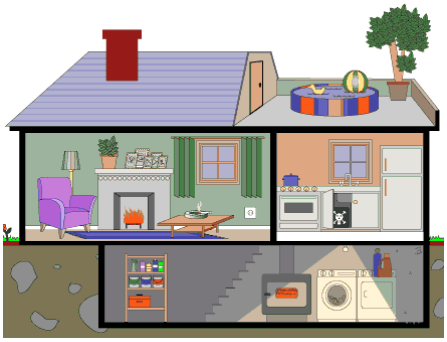
Паттерн Builder – помогает организовать пошаговое построение сложного объекта-продукта так, что клиенту не требуется понимать последовательность шагов и внутреннее устройство строящегося объекта-продукта, при этом в результате одного и того же процесса конструирования могут получаться объекты-продукты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



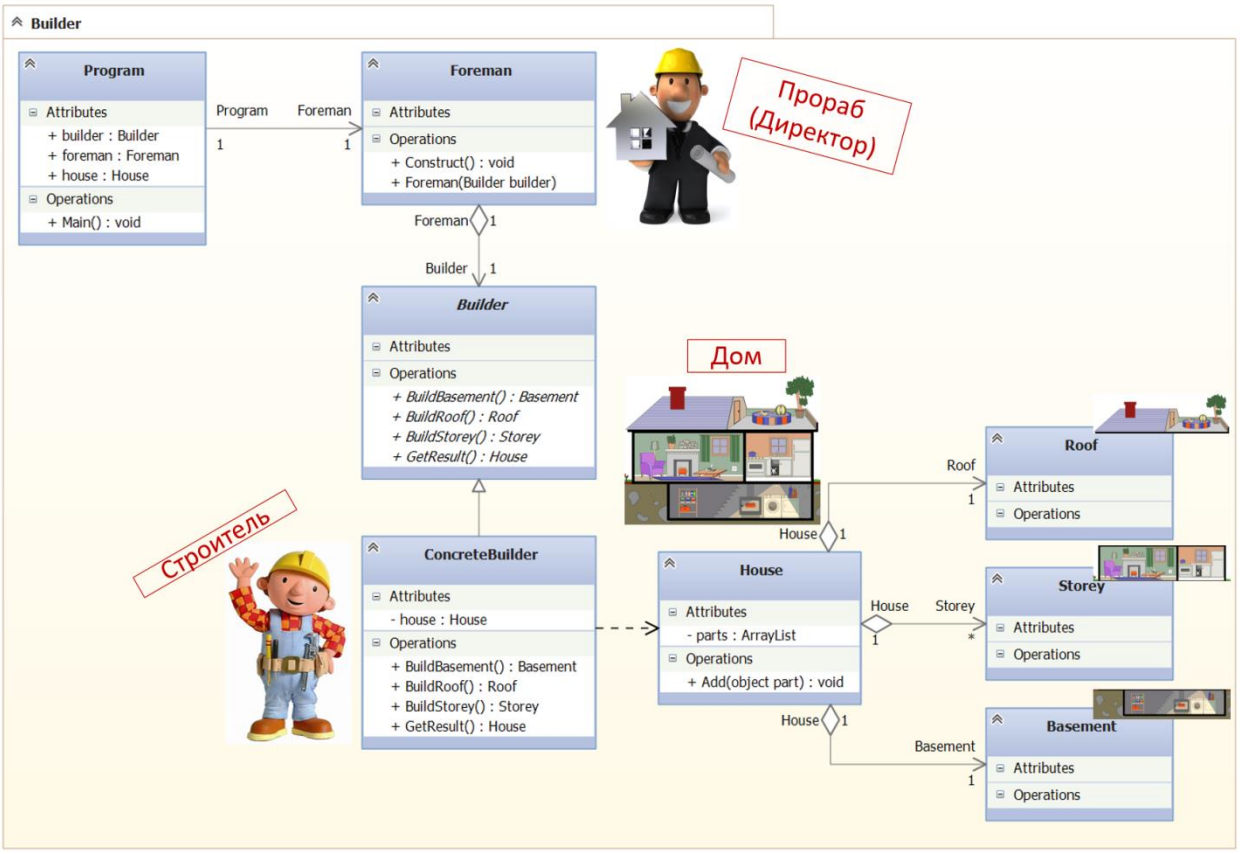
Дом

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

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

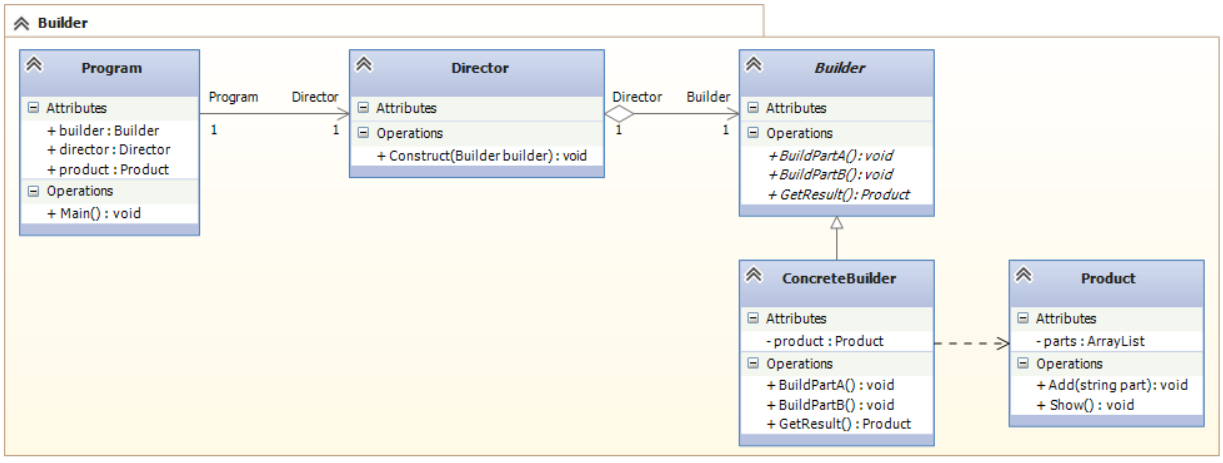
Способ построения дома определяет тип конкретного строителя. Строитель-каменщик, который строит дом из кирпича, будет строить дом отличным способом, от строителя-плотника который будет

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.



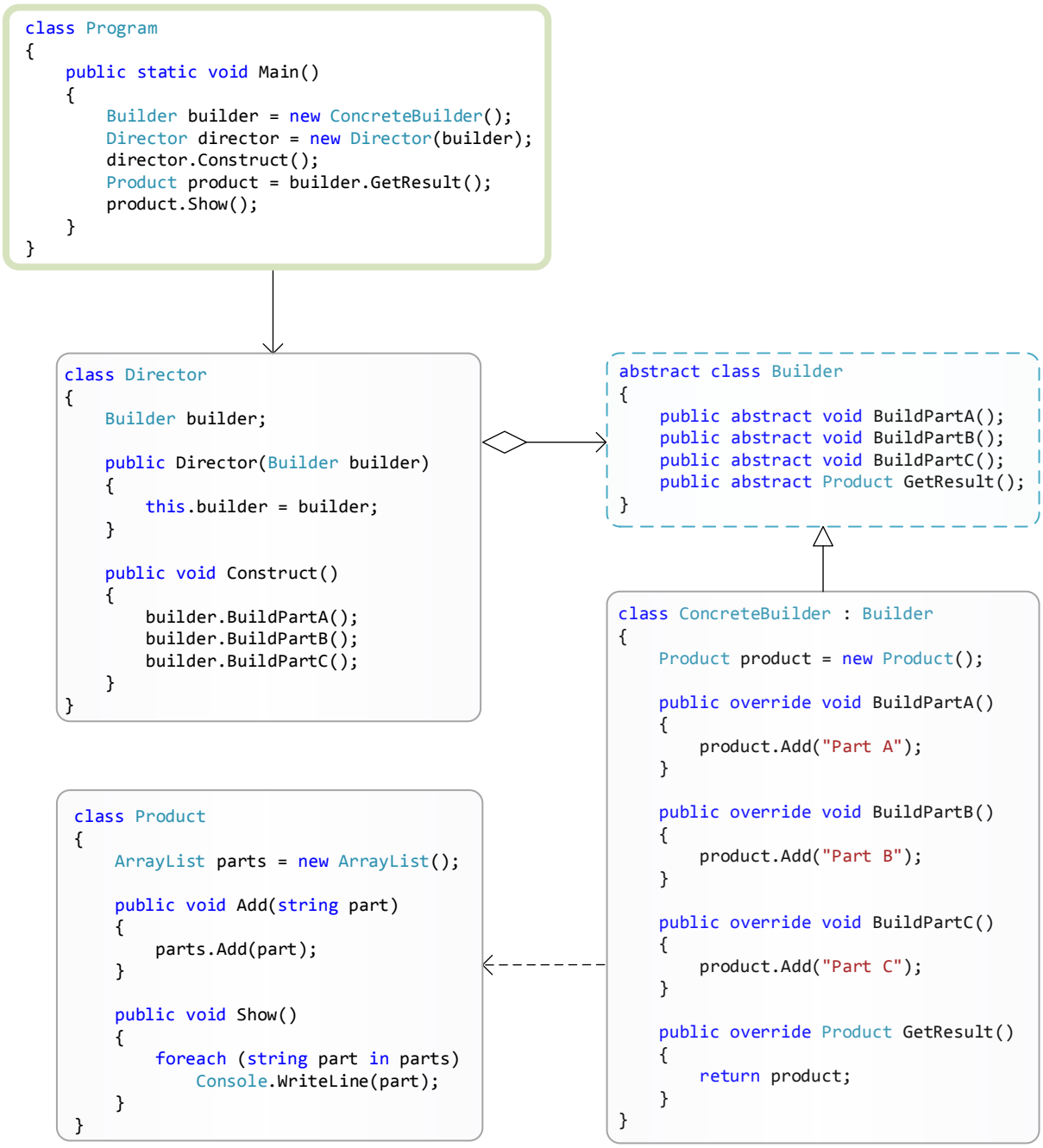
См. Пример к главе: \002_Builder\000_Builder

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`.

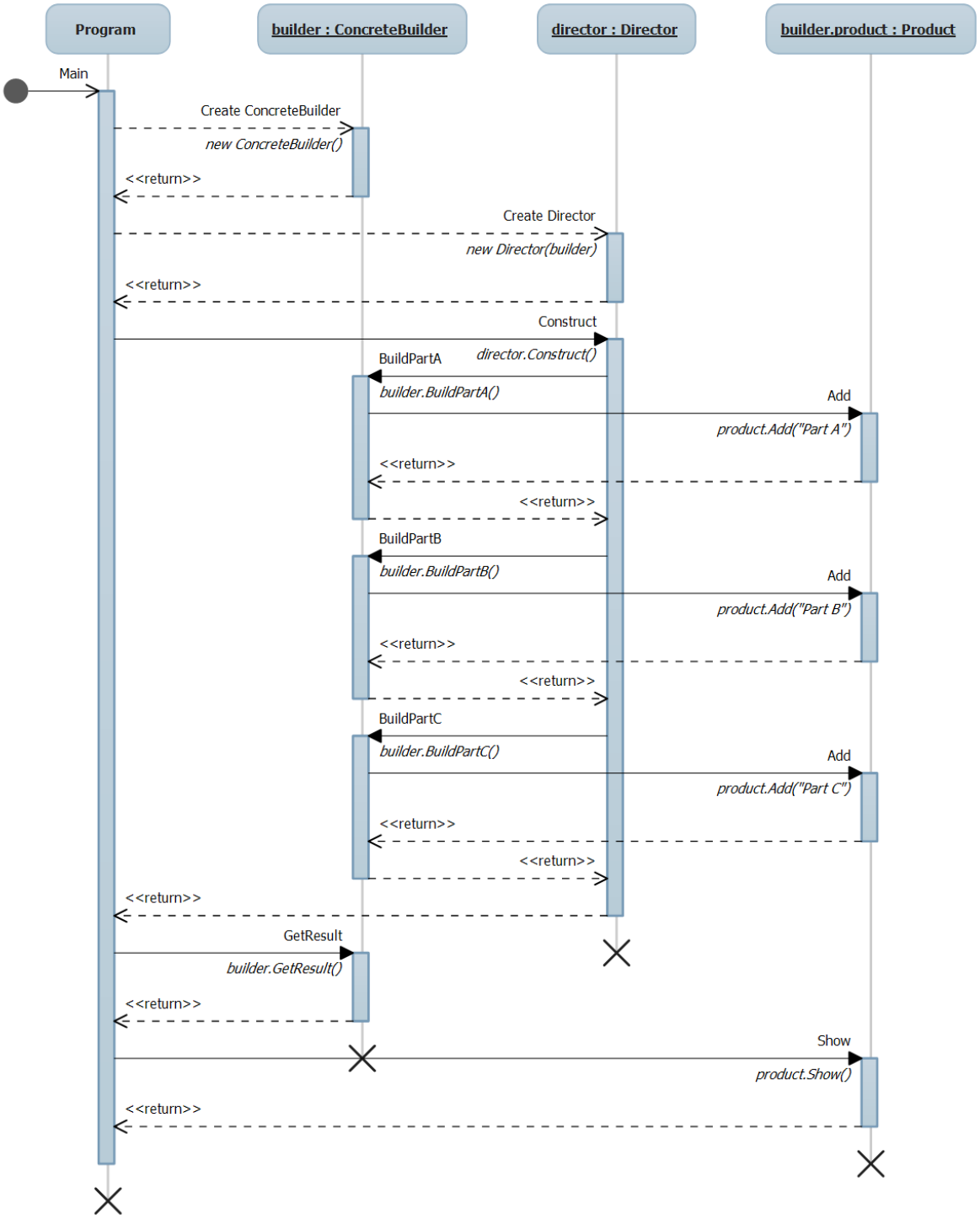
- Клиент создает экземпляр класса `Director` при этом в качестве аргумента конструктора передает ссылку на ранее созданный экземпляр класса `ConcreteBuilder`.

- 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`).

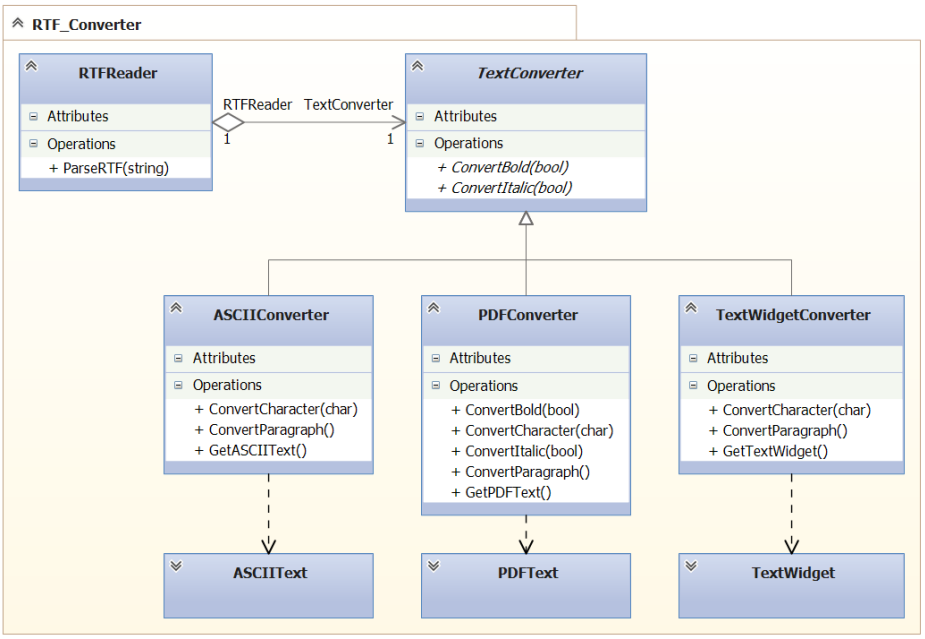
- Клиент получает от строителя ссылку на экземпляр построенного продукта.

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.