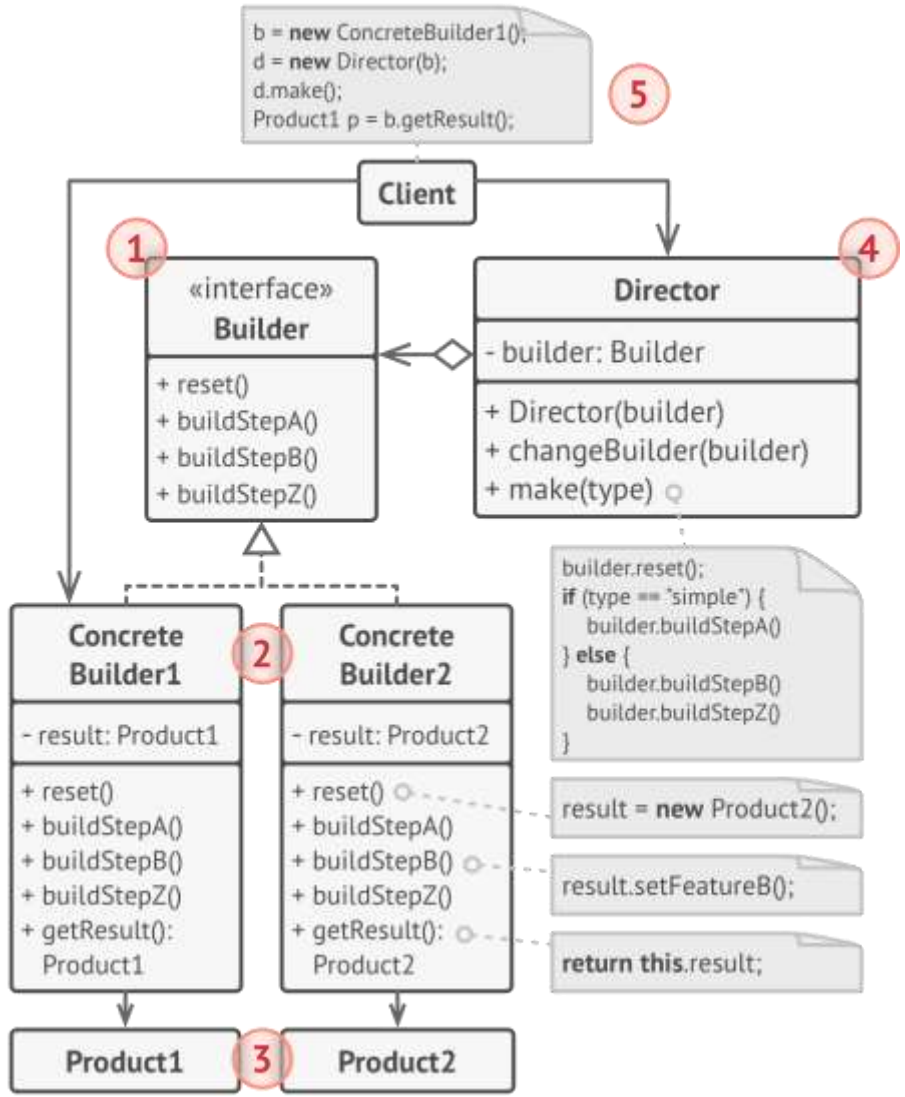
Порождающие паттерны. Часть 2

Строитель

Позволяет создавать сложные объекты пошагово. Строитель дает возможность использовать один и тот же код строительства для получения разных представлений объектов.

Структура


b = new ConcreteBuilder1();
d = new Director(b);
d.make();
Product1 p = b.getResult();
5
Client
1
«interface»
Builder
+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
4
Director
- builder: Builder
+ Director(builder)
+ changeBuilder(builder)
+ make(type)
builder.reset();
if (type == "simple") {
builder.buildStepA()
} else {
builder.buildStepB()
builder.buildStepZ()
}
Concrete
Builder1
2
- result: Product1
+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
+ getResult():
Product1
Concrete
Builder2
- result: Product2
+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
+ getResult():
Product2
result = new Product2();
result.setFeatureB();
return this.result;
Product1
3
Product2

Когда применять?

- Когда процесс создания нового объекта не должен зависеть от того, из каких частей этот объект состоит и как эти части связаны между собой.
- Когда необходимо обеспечить получение различных вариаций объекта в процессе его создания.
- Когда процесс создания объекта разбит на несколько этапов.

Пример

```
// Интерфейс Строителя объявляет создающие методы для различных частей
// объектов Продуктов.
public interface IBuilder
{
    void BuildPartA();

    void BuildPartB();

    void BuildPartC();
}
```

```
// Имеет смысл использовать паттерн Строитель только тогда, когда ваши
// продукты достаточно сложны и требуют обширной конфигурации.
//
// В отличие от других порождающих паттернов, различные конкретные строители
// могут производить несвязанные продукты. Другими словами, результаты
// различных строителей  могут не всегда следовать одному и тому же
// интерфейсу.
public class Product
{
    private List<object> _parts = new List<object>();

    public void Add(string part)
    {
        this._parts.Add(part);
    }

    public string ListParts()
    {
        string str = string.Empty;

        for (int i = 0; i < this._parts.Count; i++)
        {
            str += this._parts[i] + ", ";
        }

        str = str.Remove(str.Length - 2); // removing last ",c"

        return "Product parts: " + str + "\n";
    }
}
```

Пример

```
// Классы Конкретного Строителя следуют интерфейсу Строителя и предоставляют
// конкретные реализации шагов построения. Ваша программа может иметь
// несколько вариантов Строителей, реализованных по-разному.
public class ConcreteBuilder : IBuilder
{
    private Product _product = new Product();

    // Новый экземпляр строителя должен содержать пустой объект продукта
    // который используется в дальнейшей сборке.
    public ConcreteBuilder()
    {
        this.Reset();
    }

    public void Reset()
    {
        this._product = new Product();
    }
```

```
    // Все этапы производства работают с одним и тем же экземпляром
    // продукта.
    public void BuildPartA()
    {
        this._product.Add("PartA1");
    }

    public void BuildPartB()
    {
        this._product.Add("PartB1");
    }

    public void BuildPartC()
    {
        this._product.Add("PartC1");
    }
```

Пример. Всё ещё класс ConcreteBuilder

```
    // Конкретные Строители должны предоставить свои собственные методы
    // получения результатов. Это связано с тем, что различные типы
    // строителей могут создавать совершенно разные продукты с разными
    // интерфейсами. Поэтому такие методы не могут быть объявлены в базовом
    // интерфейсе Строителя (по крайней мере, в статически типизированном
    // языке программирования).
    //
    // Как правило, после возвращения конечного результата клиенту,
    // экземпляр строителя должен быть готов к началу производства
    // следующего продукта. Поэтому обычной практикой является вызов метода
    // сброса в конце тела метода GetProduct. Однако такое поведение не
    // является обязательным, вы можете заставить своих строителей ждать
    // явного запроса на сброс из кода клиента, прежде чем избавиться от
    // предыдущего результата.
    public Product GetProduct()
    {
        Product result = this._product;

        this.Reset();

        return result;
    }
}
```

Пример

```
// Директор отвечает только за выполнение шагов построения в определённой
// последовательности. Это полезно при производстве продуктов в определённом
// порядке или особой конфигурации. Строго говоря, класс Директор
// необязателен, так как клиент может напрямую управлять строителями.
public class Director
{
    private IBuilder _builder;

    public IBuilder Builder
    {
        set { _builder = value; }
    }

    // Директор может строить несколько вариаций продукта, используя
    // одинаковые шаги построения.
    public void BuildMinimalViableProduct()
    {
        this._builder.BuildPartA();
    }

    public void BuildFullFeaturedProduct()
    {
        this._builder.BuildPartA();
        this._builder.BuildPartB();
        this._builder.BuildPartC();
    }
}
```

Пример

```
class Program
{
    static void Main(string[] args)
    {
        // Клиентский код создаёт объект-строитель, передаёт его директору,
        // а затем инициирует  процесс построения. Конечный результат
        // извлекается из объекта-строителя.
        var director = new Director();
        var builder = new ConcreteBuilder();
        director.Builder = builder;

        Console.WriteLine("Standard basic product:");
        director.BuildMinimalViableProduct();
        Console.WriteLine(builder.GetProduct().ListParts());

        Console.WriteLine("Standard full featured product:");
        director.BuildFullFeaturedProduct();
        Console.WriteLine(builder.GetProduct().ListParts());

        // Помните, что паттерн Строитель можно использовать без класса
        // Директор.
        Console.WriteLine("Custom product:");
        builder.BuildPartA();
        builder.BuildPartC();
        Console.Write(builder.GetProduct().ListParts());
    }
}
```

Пример

```
Standard basic product:
Product parts: PartA1

Standard full featured product:
Product parts: PartA1, PartB1, PartC1

Custom product:
Product parts: PartA1, PartC1
```

Строитель и методы расширения

Большую популярность обрела реализация строителя на методах расширениях класса. Это позволяет динамически задавать большое количество свойств объекта, а если какие-либо свойства не задаются явно, то используются значения по умолчанию. К тому же такой способ задания свойств очень хорошо визуально воспринимается, что улучшает удобочитаемость кода и упрощает его отладку и дальнейшую модернизацию

Пример

В качестве примера будем использовать форматируемый с помощью html текст. С помощью методов расширения мы сможем при необходимости задавать цвет, размер и стиль шрифта, а так же задний фон и другие параметры отображения текста с помощью методов расширения

Пример. Перечисление TextLib.Fonts

```
1   namespace TextLib
2   {
3       /// <summary>
4       /// Шрифты.
5       /// </summary>
6       public enum Fonts : int
7       {
8           /// <summary>
9           /// Arial.
10          /// </summary>
11          Arial = 0,
12
13          /// <summary>
14          /// Georgia.
15          /// </summary>
16          Georgia = 1,
17
18          /// <summary>
19          /// Helvetica.
20          /// </summary>
21          Helvetica = 2
22      }
23  }
```

Пример. Класс TextLib.Text

```
1  using System;
2
3  namespace TextLib
4  {
5      /// <summary>
6      /// Форматированный HTML текст.
7      /// </summary>
8      public class Text
9      {
10         /// <summary>
11         /// Цвет шрифта.
12         /// </summary>
13         public ConsoleColor Color { get; internal set; } = ConsoleColor.White;
14
15         /// <summary>
16         /// Цвет заднего фона.
17         /// </summary>
18         public ConsoleColor BackgroundColor { get; internal set; } = ConsoleColor.Black;
19
20         /// <summary>
21         /// Жирный шрифт.
22         /// </summary>
23         public bool Bold { get; internal set; } = false;
24
25         /// <summary>
26         /// Наклонный шрифт.
27         /// </summary>
28         public bool Italic { get; internal set; } = false;
29
30         /// <summary>
31         /// Подчеркнутый шрифт.
32         /// </summary>
33         public bool Underline { get; internal set; } = false;
34
```

Пример. Класс TextLib.Text

```
35          /// <summary>
36          /// Текст.
37          /// </summary>
38          public string Content { get; internal set; } = "";
39
40          /// <summary>
41          /// Уровень заголовка.
42          /// </summary>
43          public int HeaderLevel { get; internal set; } = 0;
44
45          /// <summary>
46          /// Шрифт.
47          /// </summary>
48          public string Font { get; internal set; } = "Arial";
49
50          /// <summary>
51          /// Размер шрифта.
52          /// </summary>
53          public int Size { get; internal set; } = 12;
54
55          /// <summary>
56          /// Создать новый экземпляр класса текст.
57          /// </summary>
58          /// <param name="content"> Текст. </param>
59          public Text(string content)
60          {
61              // Проверяем входные данные на корректность.
62              if(string.IsNullOrEmpty(content))
63              {
64                  throw new ArgumentNullException(nameof(content));
65              }
66
67              // Устанавливаем значение.
68              Content = content;
69          }
```

Пример. Класс TextLib.Text

```
70
71          /// <summary>
72          /// Напечатать текст.
73          /// </summary>
74          /// <returns> Текст с разметкой HTML. </returns>
75          public string Print()
76          {
77              // Если уровень заголовка равен нулю,
78              // то используем тег обычного текста P,
79              // иначе используем тег H1, H2, ... , H6, в зависимости от значения.
80              var mainTag = HeaderLevel == 0 ? "P" : $"H{HeaderLevel}";
81
82              // Форматируем теги в соответствии со свойствами.
83              var formatedContent = $"<{mainTag} style=\"background-color: {BackgroundColor};\">" +
84                  $"<FONT size=\"{Size}\" color=\"{Color}\" face=\"{Font}\">" +
85                  (Bold == true ? "<STRONG>" : "") +
86                  (Italic == true ? "<EM>" : "") +
87                  (Underline == true ? "<U>" : "") +
88                  Content +
89                  (Underline == true ? "</U>" : "") +
90                  (Italic == true ? "</EM>" : "") +
91                  (Bold == true ? "</STRONG>" : "") +
92                  $"</FONT></{mainTag}>";
93
94              // Возвращаем форматированный текст.
95              return formatedContent;
96          }
97
98          /// <summary>
99          /// Приведение объекта к строке.
100         /// </summary>
101         /// <returns> Хранимый текст. </returns>
102         public override string ToString()
103         {
104             return Content;
105         }
106     }
107 }
```

Пример. Класс расширения TextLib.Text Builder

```
1   using System;
2
3   namespace TextLib
4   {
5       /// <summary>
6       /// Строитель, формирующий оформление текста.
7       /// </summary>
8       public static class TextBuilder
9       {
10          /// <summary>
11          /// Установить шрифт.
12          /// </summary>
13          /// <param name="text"> Форматируемый текст. </param>
14          /// <param name="font"> Шрифт. </param>
15          /// <returns> Отформатированный текст. </returns>
16          public static Text Font(this Text text, Fonts font)
17          {
18              // Получаем имя шрифта из перечисления и устанавливаем значение.
19              text.Font = Enum.GetName(typeof(Fonts), font);
20
21              // Возвращаем измененный текст.
22              return text;
23          }
24
```

Пример. Класс расширения TextLib.Text Builder

```
25          /// <summary>
26          /// Установить размер шрифта.
27          /// </summary>
28          /// <param name="text"> Форматируемый текст. </param>
29          /// <param name="size"> Размер шрифта. </param>
30          /// <returns> Отформатированный текст. </returns>
31          public static Text Size(this Text text, int size)
32          {
33              // Устанавливаем крайние допустимые значения
34              const int MinFontSize = 6;
35              const int MaxFontSize = 72;
36
37              if (size <= MinFontSize)
38              {
39                  // Если шрифт меньше либо равен минимальному,
40                  // то устанавливаем минимальное значение.
41                  text.Size = MinFontSize;
42              }
43              else if (size >= MaxFontSize)
44              {
45                  // Если шрифт больше либо равен максимальному,
46                  // то устанавливаем максимальное значение.
47                  text.Size = MaxFontSize;
48              }
49              else
50              {
51                  // Иначе устанавливаем соответствующее значение.
52                  text.Size = size;
53              }
54
55              // Возвращаем измененный текст.
56              return text;
57          }
```

Пример. Класс расширения TextLib.Text Builder

```
59          /// <summary>
60          /// Установить цвет шрифта.
61          /// </summary>
62          /// <param name="text"> Форматируемый текст. </param>
63          /// <param name="color"> Цвет шрифта. </param>
64          /// <returns> Отформатированный текст. </returns>
65          public static Text Color(this Text text, ConsoleColor color)
66          {
67              // Устанавливаем свойства и возвращаем измененный текст.
68              text.Color = color;
69              return text;
70          }
71
72          /// <summary>
73          /// Установить цвет заднего фона.
74          /// </summary>
75          /// <param name="text"> Форматируемый текст. </param>
76          /// <param name="color"> Цвет заднего фона. </param>
77          /// <returns> Отформатированный текст. </returns>
78          public static Text BackgroundColor(this Text text, ConsoleColor color)
79          {
80              // Устанавливаем свойства и возвращаем измененный текст.
81              text.BackgroundColor = color;
82              return text;
83          }
84
85          /// <summary>
86          /// Использовать жирный шрифт.
87          /// </summary>
88          /// <param name="text"> Форматируемый текст. </param>
89          /// <param name="bold"> true - использовать, false - нет. </param>
90          /// <returns> Отформатированный текст. </returns>
91          public static Text Bold(this Text text, bool bold)
92          {
93              // Устанавливаем свойства и возвращаем измененный текст.
94              text.Bold = bold;
95              return text;
96          }
```

Пример. Класс расширения TextLib.Text Builder

```
98          /// <summary>
99          /// Использовать наклонный шрифт.
100         /// </summary>
101         /// <param name="text"> Форматируемый текст. </param>
102         /// <param name="italic"> true - использовать, false - нет. </param>
103         /// <returns> Отформатированный текст. </returns>
104         public static Text Italic(this Text text, bool italic)
105         {
106             // Устанавливаем свойства и возвращаем измененный текст.
107             text.Italic = italic;
108             return text;
109         }
110
111         /// <summary>
112         /// Использовать подчеркнутый шрифт.
113         /// </summary>
114         /// <param name="text"> Форматируемый текст. </param>
115         /// <param name="underline"> true - использовать, false - нет. </param>
116         /// <returns> Отформатированный текст. </returns>
117         public static Text Underline(this Text text, bool underline)
118         {
119             // Устанавливаем свойства и возвращаем измененный текст.
120             text.Underline = underline;
121             return text;
122         }
123
```

Пример. Класс расширения TextLib.Text Builder

```
124         /// <summary>
125         /// Задать уровень заголовка.
126         /// </summary>
127         /// <param name="text"> Форматируемый текст. </param>
128         /// <param name="headerLevel"> 0 - обычный текст, 1-6 - заголовки. </param>
129         /// <returns> Отформатированный текст. </returns>
130         public static Text HeaderLevel(this Text text, int headerLevel)
131         {
132             // Задаем крайние корректные значения.
133             const int NormalText = 0;
134             const int MinHeader = 6;
135 
136             if (headerLevel <= NormalText)
137             {
138                 // Если задано значение меньше или равное нуля используем обычный текст.
139                 text.HeaderLevel = NormalText;
140             }
141             else if (headerLevel >= MinHeader)
142             {
143                 // Если задано значение большее либо равное шести используем заголовок H6.
144                 text.HeaderLevel = MinHeader;
145             }
146             else
147             {
148                 // Иначе устанавливаем соответствующий уровень заголовка.
149                 text.HeaderLevel = headerLevel;
150             }
151 
152             // Возвращаем измененный текст.
153             return text;
154         }
155     }
156 }
```

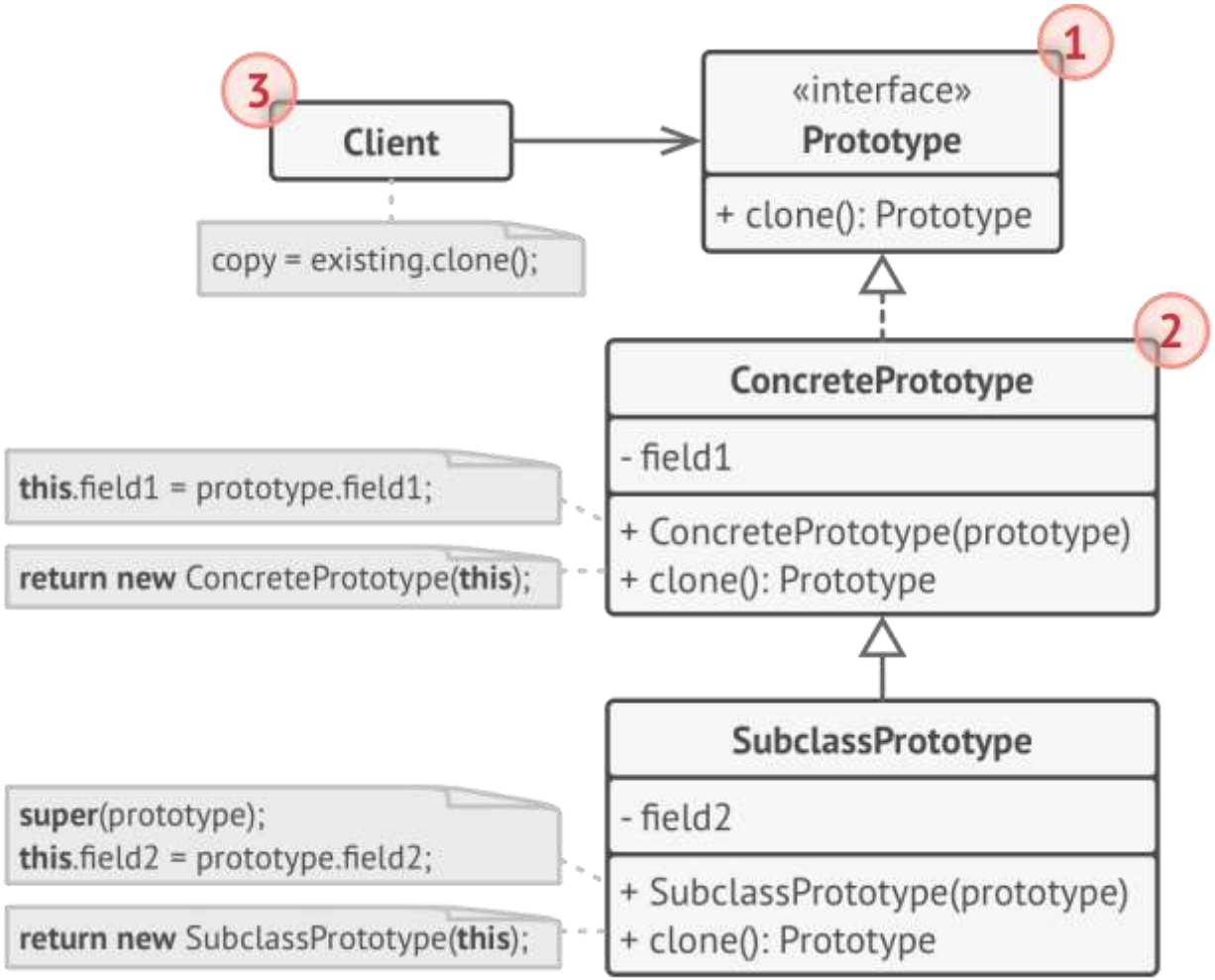
Пример. Класс Builder.Program

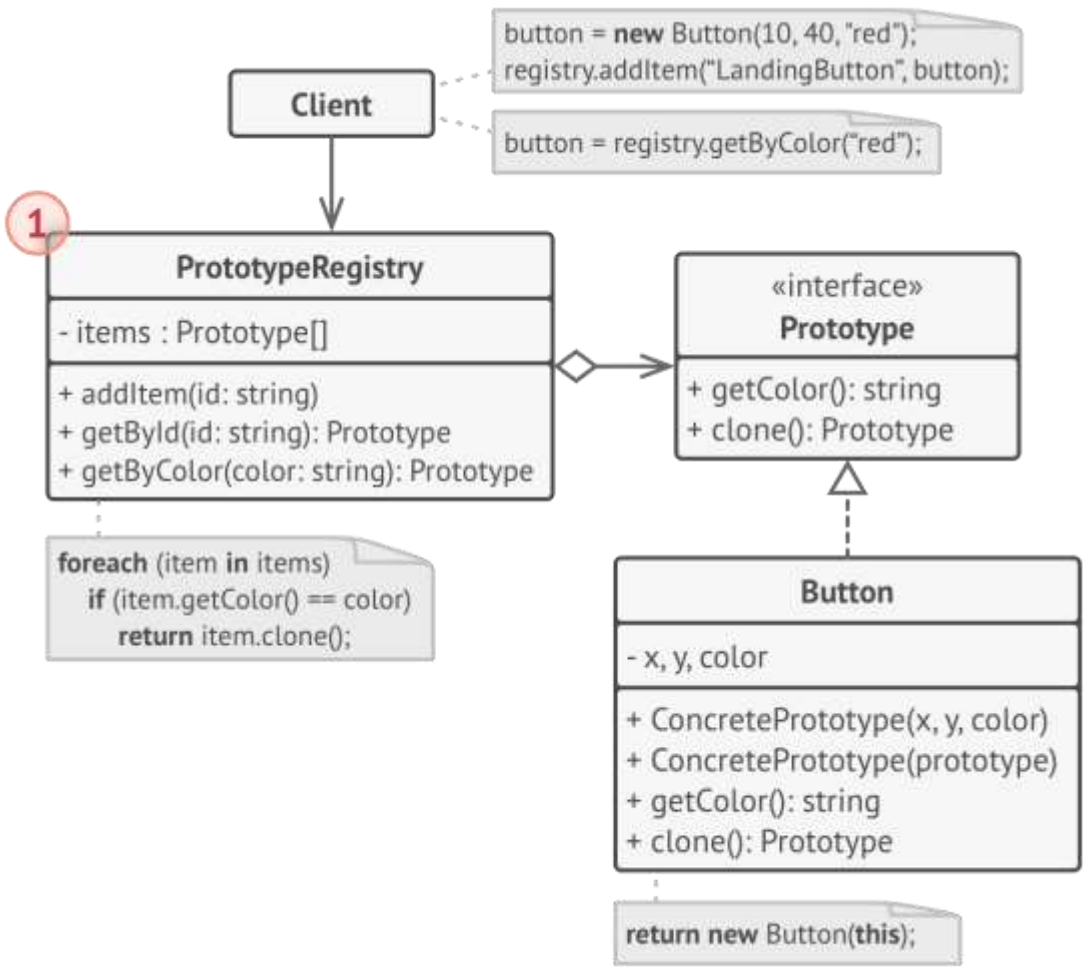
```
using System;
using TextLib;

namespace Builder
{
    class Program
    {
        static void Main(string[] args)
        {
            // Создаем объект текста.
            var text = new Text("Hello, World!");

            // Задаем форматирование текста.
            text.Font(Fonts.Georgia)
                .Size(18)
                .Color(ConsoleColor.Red)
                .BackgroundColor(ConsoleColor.Black)
                .Bold(true)
                .Underline(true);

            // Выводим форматированные текст.
            var html = text.Print();
            Console.WriteLine(html);
            Console.ReadLine();
        }
    }
}
```

Прототип

Позволяет копировать объекты, не вдаваясь в подробности их реализации.

Структура


3
Client
1
«interface»
Prototype
+ clone(): Prototype
copy = existing.clone();
2
ConcretePrototype
- field1
+ ConcretePrototype(prototype)
+ clone(): Prototype
this.field1 = prototype.field1;
return new ConcretePrototype(this);
SubclassPrototype
- field2
+ SubclassPrototype(prototype)
+ clone(): Prototype
super(prototype);
this.field2 = prototype.field2;
return new SubclassPrototype(this);


button = new Button(10, 40, "red");
registry.addItem("LandingButton", button);
Client
button = registry.getByColor("red");
1
PrototypeRegistry
- items : Prototype[]
+ addItem(id: string)
+ getById(id: string): Prototype
+ getByColor(color: string): Prototype
«interface»
Prototype
+ getColor(): string
+ clone(): Prototype
foreach (item in items)
if (item.getColor() == color)
return item.clone();
Button
- x, y, color
+ ConcretePrototype(x, y, color)
+ ConcretePrototype(prototype)
+ getColor(): string
+ clone(): Prototype
return new Button(this);

Когда применять?

- Когда клонирование объекта является более предпочтительным вариантом нежели его создание и инициализация с помощью конструктора. Особенно когда известно, что объект может принимать небольшое ограниченное число возможных состояний.

Пример

```
public class IdInfo
{
    public int IdNumber;

    public IdInfo(int idNumber)
    {
        this.IdNumber = idNumber;
    }
}
```

```
public class Person
{
    public int Age;
    public DateTime BirthDate;
    public string Name;
    public IdInfo IdInfo;

    public Person ShallowCopy()
    {
        return (Person) this.MemberwiseClone();
    }

    public Person DeepCopy()
    {
        Person clone = (Person) this.MemberwiseClone();
        clone.IdInfo = new IdInfo(IdInfo.IdNumber);
        clone.Name = String.Copy(Name);
        return clone;
    }
}
```

Пример

```
class Program
{
    static void Main(string[] args)
    {
        Person p1 = new Person();
        p1.Age = 42;
        p1.BirthDate = Convert.ToDateTime("1977-01-01");
        p1.Name = "Jack Daniels";
        p1.IdInfo = new IdInfo(666);

        // Выполнить поверхностное копирование p1 и присвоить её p2.
        Person p2 = p1.ShallowCopy();
        // Сделать глубокую копию p1 и присвоить её p3.
        Person p3 = p1.DeepCopy();

        // Вывести значения p1, p2 и p3.
        Console.WriteLine("Original values of p1, p2, p3:");
        Console.WriteLine("   p1 instance values: ");
        DisplayValues(p1);
        Console.WriteLine("   p2 instance values:");
        DisplayValues(p2);
        Console.WriteLine("   p3 instance values:");
        DisplayValues(p3);
```

Пример

```
            // Изменить значение свойств p1 и отобразить значения p1, p2 и p3.
            p1.Age = 32;
            p1.BirthDate = Convert.ToDateTime("1900-01-01");
            p1.Name = "Frank";
            p1.IdInfo.IdNumber = 7878;
            Console.WriteLine("\nValues of p1, p2 and p3 after changes to p1:");
            Console.WriteLine("   p1 instance values: ");
            DisplayValues(p1);
            Console.WriteLine("   p2 instance values (reference values have changed):");
            DisplayValues(p2);
            Console.WriteLine("   p3 instance values (everything was kept the same):");
            DisplayValues(p3);
        }

        public static void DisplayValues(Person p)
        {
            Console.WriteLine("      Name: {0:s}, Age: {1:d}, BirthDate: {2:MM/dd/yy}",
                p.Name, p.Age, p.BirthDate);
            Console.WriteLine("      ID#: {0:d}", p.IdInfo.IdNumber);
        }
    }
}
```

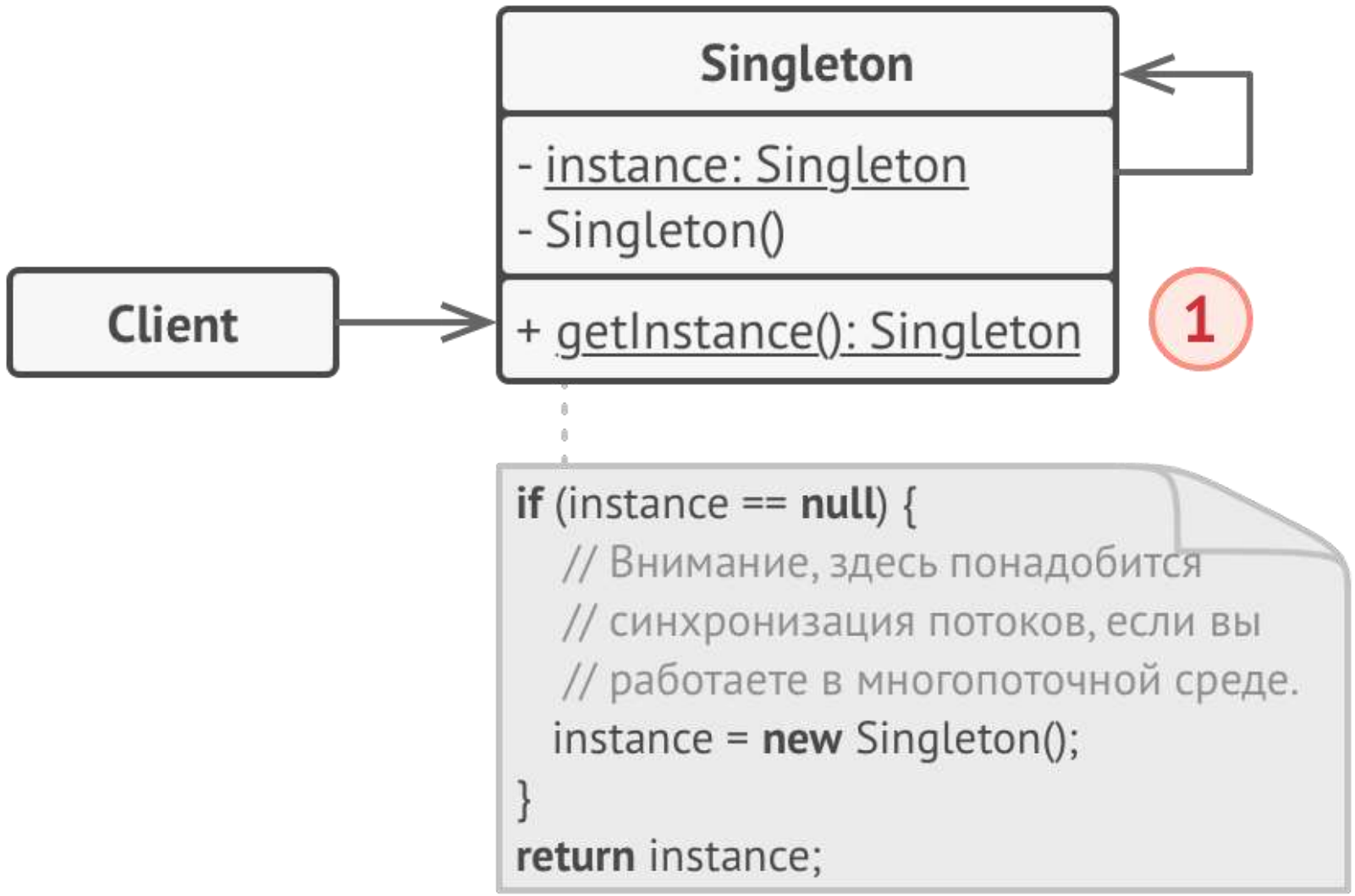
Пример

```
Original values of p1, p2, p3:
   p1 instance values:
      Name: Jack Daniels, Age: 42, BirthDate: 01/01/77
      ID#: 666
   p2 instance values:
      Name: Jack Daniels, Age: 42, BirthDate: 01/01/77
      ID#: 666
   p3 instance values:
      Name: Jack Daniels, Age: 42, BirthDate: 01/01/77
      ID#: 666

Values of p1, p2 and p3 after changes to p1:
   p1 instance values:
      Name: Frank, Age: 32, BirthDate: 01/01/00
      ID#: 7878
   p2 instance values (reference values have changed):
      Name: Jack Daniels, Age: 42, BirthDate: 01/01/77
      ID#: 7878
   p3 instance values (everything was kept the same):
      Name: Jack Daniels, Age: 42, BirthDate: 01/01/77
      ID#: 666
```

Одиночка

Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

Структура


Singleton
- instance: Singleton
- Singleton()
+ getInstance(): Singleton
Client
1
if (instance == null) {
// Внимание, здесь понадобится
// синхронизация потоков, если вы
// работаете в многопоточной среде.
instance = new Singleton();
}
return instance;

Когда применять?

- Когда необходимо, чтобы для класса существовал только один экземпляр.

Пример. Однопоточный одиночка

```
// Класс Одиночка предоставляет метод `GetInstance`, который ведёт себя как
// альтернативный конструктор и позволяет клиентам получать один и тот же
// экземпляр класса при каждом вызове.
class Singleton
{
    // Конструктор Одиночки всегда должен быть скрытым, чтобы предотвратить
    // создание объекта через оператор new.
    private Singleton() { }

    // Объект одиночки храниться в статичном поле класса. Существует
    // несколько способов инициализировать это поле, и все они имеют разные
    // достоинства и недостатки. В этом примере мы рассмотрим простейший из
    // них, недостатком которого является полная неспособность правильно
    // работать в многопоточной среде.
    private static Singleton _instance;

    // Это статический метод, управляющий доступом к экземпляру одиночки.
    // При первом запуске, он создаёт экземпляр одиночки и помещает его в
    // статическое поле. При последующих запусках, он возвращает клиенту
    // объект, хранящийся в статическом поле.
    public static Singleton GetInstance()
    {
        if (_instance == null)
        {
            _instance = new Singleton();
        }
        return _instance;
    }

    // Наконец, любой одиночка должен содержать некоторую бизнес-логику,
    // которая может быть выполнена на его экземпляре.
    public static void someBusinessLogic()
    {
        // ...
    }
}
```

Пример. Однопоточный одиночка

```
class Program
{
    static void Main(string[] args)
    {
        // Клиентский код.
        Singleton s1 = Singleton.GetInstance();
        Singleton s2 = Singleton.GetInstance();

        if (s1 == s2)
        {
            Console.WriteLine("Singleton works, both variables contain the same instance.");
        }
        else
        {
            Console.WriteLine("Singleton failed, variables contain different instances.");
        }
    }
}
```

```
Singleton works, both variables contain the same instance.
```

Пример. Многопоточный одиночка

```
// Эта реализация Одиночки называется "блокировка с двойной проверкой"
// (double check lock). Она безопасна в многопоточной среде, а также
// позволяет отложенную инициализацию объекта Одиночки.
class Singleton
{
    private Singleton() { }

    private static Singleton _instance;

    // У нас теперь есть объект-блокировка для синхронизации потоков во
    // время первого доступа к Одиночке.
    private static readonly object _lock = new object();
```

Пример. Многопоточный одиночка

```
    public static Singleton GetInstance(string value)
    {
        // Это условие нужно для того, чтобы не стопорить потоки блокировкой
        // после того как объект-одиночка уже создан.
        if (_instance == null)
        {
            // Теперь представьте, что программа была только-только
            // запущена. Объекта-одиночки ещё никто не создавал, поэтому
            // несколько потоков вполне могли одновременно пройти через
            // предыдущее условие и достигнуть блокировки. Самый быстрый
            // поток поставит блокировку и двинется внутрь секции, пока
            // другие будут здесь его ожидать.
            lock (_lock)
            {
                // Первый поток достигает этого условия и проходит внутрь,
                // создавая объект-одиночку. Как только этот поток покинет
                // секцию и освободит блокировку, следующий поток может
                // снова установить блокировку и зайти внутрь. Однако теперь
                // экземпляр одиночки уже будет создан и поток не сможет
                // пройти через это условие, а значит новый объект не будет
                // создан.
                if (_instance == null)
                {
                    _instance = new Singleton();
                    _instance.Value = value;
                }
            }
        }
        return _instance;
    }

    // Мы используем это поле, чтобы доказать, что наш Одиночка
    // действительно работает.
    public string Value { get; set; }
}
```

Пример. Многопоточный одиночка

```
class Program
{
    static void Main(string[] args)
    {
        // Клиентский код.

        Console.WriteLine(
            "{0}\n{1}\n\n{2}\n",
            "If you see the same value, then singleton was reused (yay!)",
            "If you see different values, then 2 singletons were created (booo!!)",
            "RESULT:"
        );

        Thread process1 = new Thread(() =>
        {
            TestSingleton("FOO");
        });
        Thread process2 = new Thread(() =>
        {
            TestSingleton("BAR");
        });

        process1.Start();
        process2.Start();

        process1.Join();
        process2.Join();
    }

    public static void TestSingleton(string value)
    {
        Singleton singleton = Singleton.GetInstance(value);
        Console.WriteLine(singleton.Value);
    }
}
```

```
FOO
FOO
```

Спасибо за внимание!