Builder

Prof. Igor Avila Pereira
igor.pereira@riogrande.ifrs.edu.br

Divisão de Computação
Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul (IFRS)
Câmpus Rio Grande

Agenda

Agenda

Introdução

- A Orientação a Objetos é constantemente mal utilizada.
  - Sem o devido treino, temos a tendência de raciocinar de forma estruturada.
- Esse problema está tão enraizado que muitas vezes encontramos dificuldades nos pontos mais fundamentais da programação, tal como: **a construção de instâncias de objetos um tanto mais complexos que beans simples.**

Consequência:

Nosso código fica difícil de manter e mais propenso a erros

Definição

A definição do padrão *Builder* segundo o *GOF* é:

Definição:

...separar a construção de um objeto complexo de sua representação de modo que o mesmo processo de construção possa criar diferentes representações...

O padrão *Builder* tem como objetivo simplificar a construção de objetos sem que precisemos conhecer os detalhes dessa construção.

Definição

- Ao contrário do Strategy, que é um padrão comportamental, o padrão Builder está na categoria de padrões de criação, junto com Abstract Factory, Factory Method, Prototype e Singleton.
- O padrão Builder deverá ser utilizado quando o algoritmo para criação de um objeto complexo deve ser independente de partes que compõe o objeto e de como elas são montadas.
- O processo de construção deve permitir diferentes representações para o objeto que é construído.

Aplicabilidade

- Utilize quando você precisa separar a criação de um objeto complexo das partes que o constituem e como elas se combinam.
- Outro caso é quando o processo de construção precisa permitir diferentes formas de representação do objeto construído.

Agenda

Exemplo 1

- Suponha que precisamos criar um objeto com diversos atributos **opcionais**.
- Vamos usar uma pizza como exemplo (um dos exemplos clássicos para ilustrar padrões de projetos)

Exemplo 1

Já vi muitos construtores de **pizza** por aí da seguinte forma:

```
1  public class Pizza {
2      private int tamanho; private boolean queijo; private boolean bacon;
3
4      Pizza (int tamanho) {
5          this.tamanho = tamanho;
6      }
7      Pizza (int tamanho, boolean queijo){
8          this(tamanho);
9          this.queijo = queijo;
10     }
11     Pizza (int tamanho, boolean queijo, boolean tomate){
12         this(tamanho, queijo);
13         this.tomate = tomate;
14     }
15     Pizza (int tamanho, boolean queijo, boolean tomate, boolean bacon){
16         this(tamanho, queijo, tomate);
17         this.bacon = bacon;
18     }
19 }
```

Exemplo 1

- Diga a verdade, você já faz isso em algum momento?
  - E pode ficar pior se acrescentarmos construtores com combinações e ordenação diferentes de parâmetros!

Exemplo 1

- A sobrecarga é interessante quando temos algumas poucas variações de parâmetros e há poucas mudanças no conjunto de atributos.
- Porém, chega uma hora que nem sabemos mais o que está acontecendo.

Exemplo 1

- Quanto tempo você já perdeu inspecionando conteúdo de classes de terceiros para entender que valores deveria usar?

```
1 new Pizza(10, true, false, true, false, true, false, true, false...)
```

Que tipo de pizza é essa mesmo?

Criando um objeto com Builder Pattern

```
1  public class Pizza {
2      private int tamanho; private boolean queijo; private boolean
           tomate; private boolean bacon;
3
4      public static class Builder {
5          // requerido
6          private final int tamanho;
7          // opcional
8          private boolean queijo = false;
9          private boolean tomate = false;
10         private boolean bacon = false;
11
12         public Builder(int tamanho){
13             this.tamanho = tamanho;
14         }
15         public Builder queijo(){
16             queijo = true;
17             return this;
18         }
19     }
20 }
```

Criando um objeto com Builder Pattern

```
1         public Builder tomate(){
2             tomate = true;
3             return this;
4         }
5         public Builder bacon(){
6             bacon = true;
7             return this;
8         }
9         public Pizza builder(){
10            return new Pizza(this);
11        }
12    }
13    private Pizza(Builder builder){
14        tamanho = builder.tamanho;
15        queijo = builder.queijo;
16        tomate = builder.tomate;
17        bacon = builder.bacon;
18    }
19 }
```

Exemplo 2

- Aplicando o padrão de projeto Builder, temos agora um objeto **construtor** para o objeto Pizza.
- A classe Pizza está um pouco mais complexa, mas confira como ficou elegante a forma de temperarmos:

```
1 Pizza pizza = new Pizza.Builder(10).queijo().tomate().bacon().build();
```

Exemplo 2

- É muito mais fácil de codificar com essa API e entender o que está acontecendo.
- O Builder Pattern é muito utilizado em boas bibliotecas que disponibilizam APIs intuitivas e fáceis de aprender, como construtores de XML e o Response do JAX-RS, por exemplo.

Exemplo 2

```
1  public class Person {
2      private String firstName;
3      private String middleName;
4      private String lastName;
5      private int age;
6
7      public Person(String firstName, String middleName, String lastName,
           int age){
8          this.firstName = firstName;
9          this.middleName = middleName;
10         this.lastName = lastName;
11         this.age = age;
12     }
13     // getters e setters
14 }
```

Exemplo 2

- Este exemplo é uma classe Person simples com 4 atributos.
- Entretanto, pense o que você teria que fazer para adicionar mais campos nesta classe?
- Como isso adicionaria uma complexidade adicional a este construtor....

Exemplo 2

- Vamos agora adicionar alguns campos extras: **fatherName**, **mothersName**, **height**, **weight** e converte-lo para o padrão Builder.

```
1  public class Person {
2      private String firstName;
3      private String middleName;
4      // ...
5      public Person(String firstName, String middleName, String lastName,
           int age, String fathersName, String mothersName, double
           height, double weight) {
6
7          this.firstName = firstName;
8          this.middleName = middleName;
9          // ...
10     }
11 }
```

Exemplo 2

```
1  public static class Builder {
2      private String firstName;
3      private String middleName;
4      // ...
5      public Builder setFistName(String firstName){
6          this.firstName = firstName;
7          return this;
8      }
9      public Builder setMiddleName(String middleName){
10         this.middleName = middleName;
11         return this;
12     }
13     // ...
14 }
```

Exemplo 2

```
1    public Person build(){
2        return new Person(firstName, middleName, lastName, age,
             fathersName, mothersName, height, weight);
3    }
```

Como resultado teríamos algo parecido com:

```
1  Person person = new Person.Builder()
2                      .setAge(5)
3                      .setFirstName("Bob")
4                      .setHeight(6)
5                      .build();
```

Classes Utilitárias

- Temos ainda classes utilitárias com seus métodos estáticos, usadas nos mais diversos pontos da arquitetura de um sistema.
- Elas causam problemas quando mal planejadas, pois a tendência é acumularmos muitos métodos sobrecarregados com diferentes objetivos que infectam todo o código.
- Então, sem perceber, perdemos o controle, já que agora todo o sistema está acoplado a tais classes e alterações acabam impactando onde não esperamos.

Classes Utilitárias

Vejamos um exemplo de uma típica classe com rotinas de tratamento de datas:

```
1  public class Data {
2      public static Date converteTextoParaData(String dataStr){
3          try {
4              return new SimpleDateFormat("dd/MM/yyyy").parse(dataStr);
5          } catch (ParseException e){
6              return null;
7          }
8      }
9      public static String converteDataParaTexto(Date data){
10         return new SimpleDateFormat("dd/MM/yyyy").format(data);
11     }
12 }
```

Classes Utilitárias

- Eis como ficaria uma manipulação simples de data usando essa classe:

```
1 String inputDateStr = "28/02/2013";
2 Date inputDate = Data.converteTextoParaData(inputDateStr);
3 Date result = Data.avancarDiasCorridos(inputDate, 30);
4 String resultDateStr = Data.converteDataParaTexto(resultDate);
```

- **Entediante.** Vamos refatorar a classe com o conceito de **Interface Fluente**:

Classes Utilitárias

```
1  public class Data {
2      private Date data;
3      public Data(String dataStr){
4          try {
5              data = new SimpleDateFormat("dd/MM/yyyy").parse(dataStr);
6          } catch (ParseException e){
7              throw new IlegalArgumentException(e);
8          }
9      }
10     public String toString(){
11         return new SimpleDateFormat("dd/MM/yyyy").format(data);
12     }
13     public Data avancarDiasCorridos(int dias){
14         Calendar c = Calendar.getInstance();
15         c.setTime(data);
16         c.add(Calendar.DATE, dias);
17         data = c.getTime();
18         return this;
19     }
20 }
```

Classes Utilitárias

E o uso fica assim:

```
1 String dateStr = "";
2 dateStr = new Data("28/02/2013").avancarDiasCorridos(30).toString();
```

Simples, não?

Nota: considere este como um exemplo de estudo. O uso do *new* para instanciar objetos é discutível, assim como o uso da classe *java.util.Date*.

Agenda

Prós

1. Melhora a manutenção do sistema, aumentam a legibilidade do código, pois mostra uma forma elegante de tratar classes com grande número de propriedades se tornando complexos para serem construídos.
2. O código de criação torna-se menos propenso a erros de usuário.
3. O padrão Builder incorpora robustez

Contras

1. O padrão Builder é **verboso**. Exige duplicação de código a fim de copiar todos os campos da classe original.
2. Um dos problemas com o padrão é que é preciso sempre chamar o método de construção para depois utilizar o produto em si.

Prós e Contras

Dica:

- Utilizar o Builder só tem sentido quando há uma grande quantidade de parâmetros para a construção do objeto, ou seja, não deve-se utilizá-lo quando há poucos parâmetros.
- Além disso existe um custo de performance (normalmente não perceptível) já que sempre deve-se chamar o Builder antes de utilizar o objeto, em sistemas de performance crítica pode ser uma desvantagem.

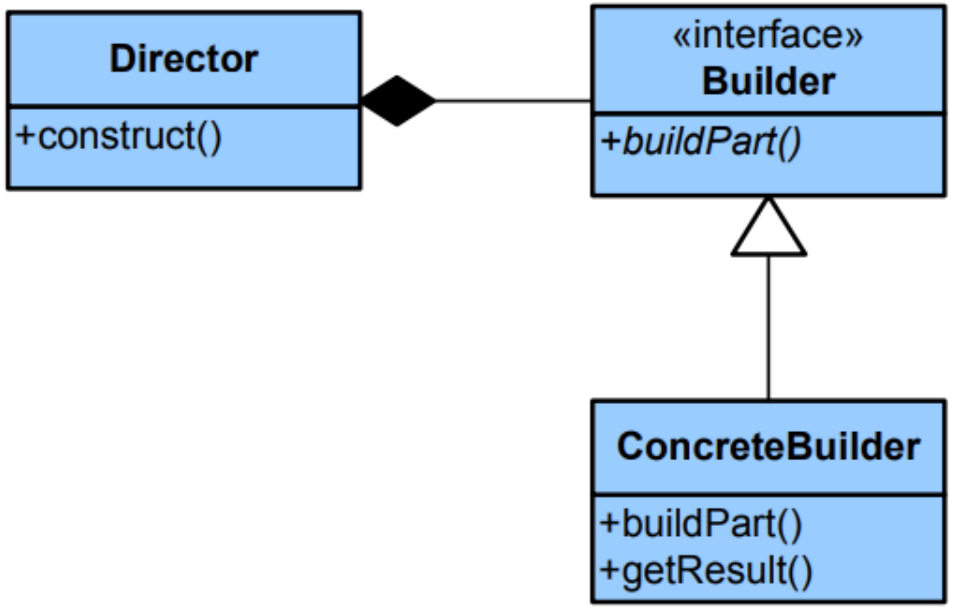
Agenda

UML

Builder

Type: Creational

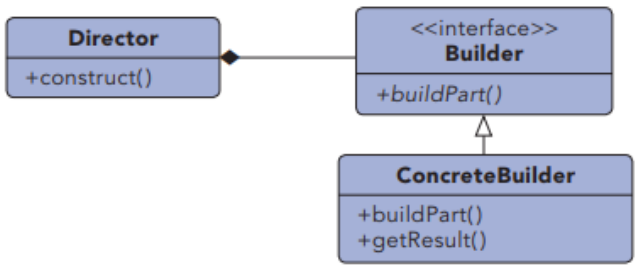
What it is:
Separate the construction of a complex object from its representing so that the same construction process can create different representations.



UML

BUILDER

Object Creational



Purpose

Allows for the dynamic creation of objects based upon easily interchangeable algorithms.

Use When

- Object creation algorithms should be decoupled from the system.
- Multiple representations of creation algorithms are required.
- The addition of new creation functionality without changing the core code is necessary.
- Runtime control over the creation process is required.

Example

A file transfer application could possibly use many different protocols to send files and the actual transfer object that will be created will be directly dependent on the chosen protocol. Using a builder we can determine the right builder to use to instantiate the right object. If the setting is FTP then the FTP builder would be used when creating the object.

Agenda

Conclusões

- Simplificar o código não é luxo.
- Trata-se de uma necessidade na luta contra a crescente complexidade dos sistemas de software.

Builder

Prof. Igor Avila Pereira
igor.pereira@riogrande.ifrs.edu.br

Divisão de Computação
Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul (IFRS)
Câmpus Rio Grande