Builder Pattern

Projeto de Software 2018.1
Professor: Baldoino Fonseca

Álvaro Amorim de Albuquerque

https://github.com/alvaroalbuquerque

Problema

```java
public class Employee{

    private String name;
    private String sirName;
    private String email;
    private String cpf;
    private String telephone;
    private double salary;
    private double syndicateTax;
    private String spouse;

    public Employee(String name, String sirName, String email, String cpf, String telephone,
        double salary, double syndicateTax, String spouse) {
        this.name = name;
        this.sirName = sirName;
        this.email = email;
        this.cpf = cpf;
        this.telephone = telephone;
        this.salary = salary;
        this.syndicateTax = syndicateTax;
        this.spouse = spouse;
    }
}

Employee newEmployee = new Employee("Alvaro", "Amorim", "aaa2@ic.ufal.br","12345678900","12123434", 1000.00, 0, null);
Employee newEmployee = new Employee("Alvaro", "Amorim", "aaa2@ic.ufal.br","12123434","12345678900", 1000.00, 0, null);
```

Builder Pattern

- Creational Pattern
- Separa a construção de um objeto complexo de sua representação
- A criação de um objeto é feita por uma Builder Class e não pela classe em si
- Cria diferentes representações a partir de um mesmo processo

Implementação 1

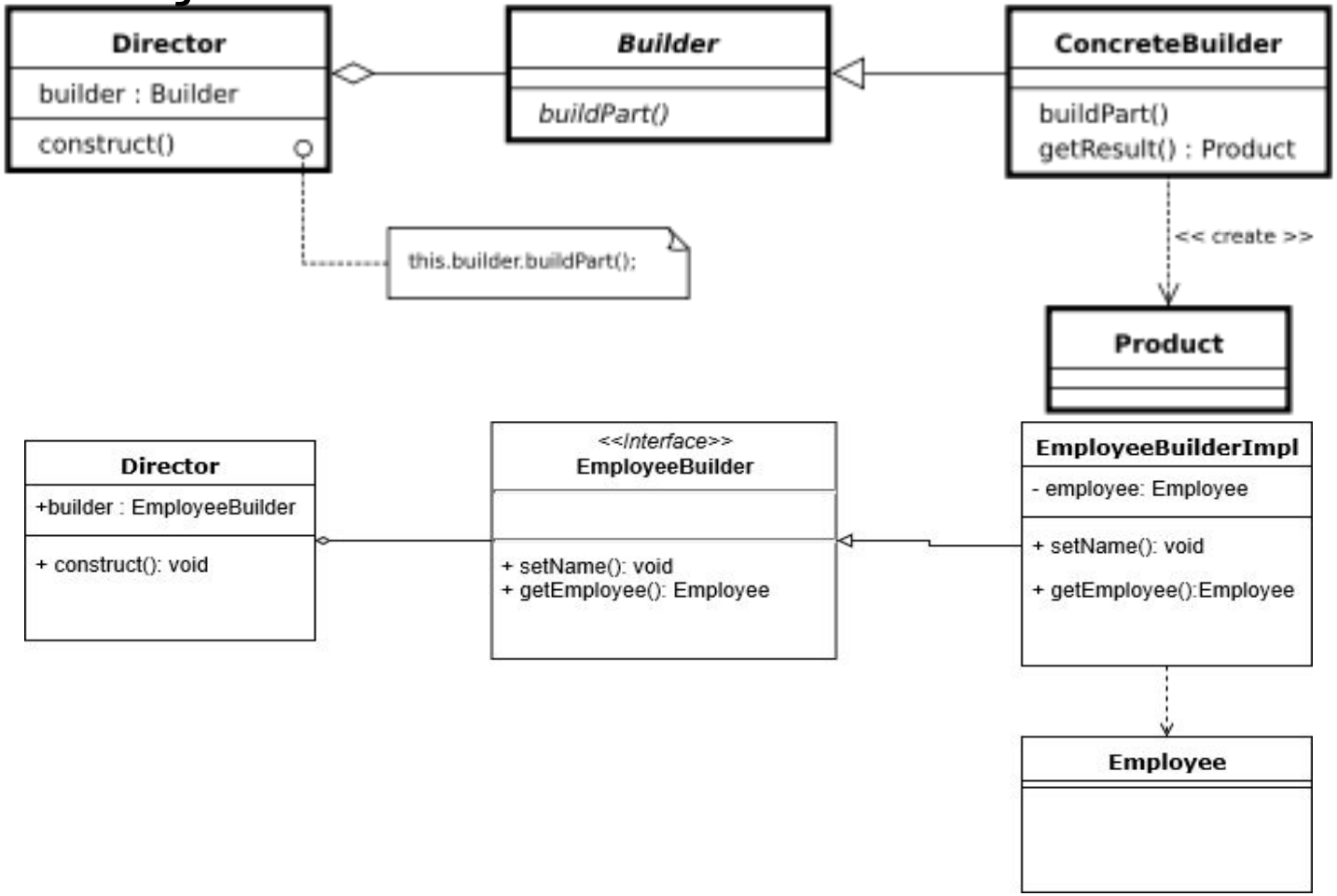
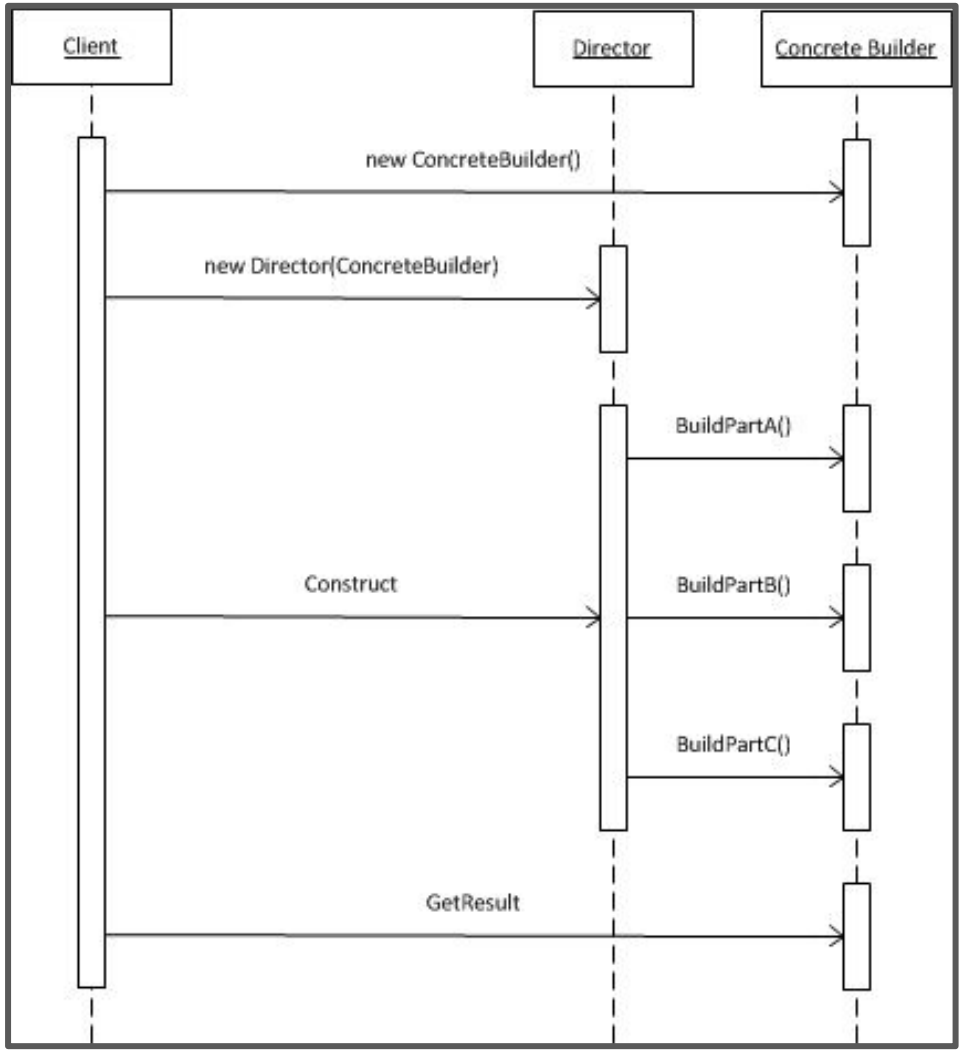
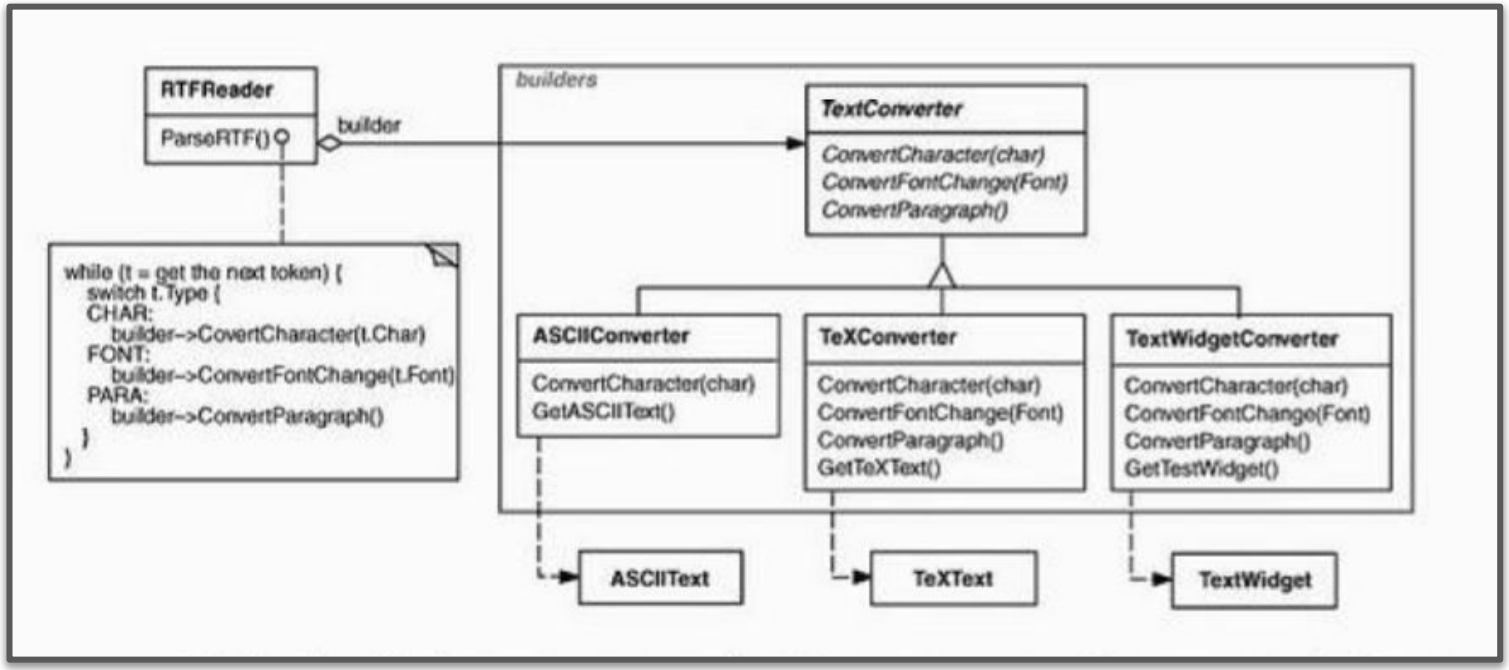


Diagrama de Interação



Motivação

- Um leitor de arquivo RTF deve ser capaz de converter o arquivo de texto em diferentes formatos e o número de conversões possíveis é “infinito”
- Por isso deve ser fácil adicionar um novo tipo de conversão sem modificar o leitor.



Implementação 2

```java
public class Employee{

    private String name;
    private String sirName;
    private String email;
    private String cpf;
    private String telephone;
    private double salary;
    private double syndicateTax;
    private String spouse;

    public Employee(EmployeeBuilder employeeBuilder) {
        this.name = employeeBuilder.name;
        this.sirName = employeeBuilder.sirName;
        this.email = employeeBuilder.email;
        this.cpf = employeeBuilder.cpf;
        this.telephone = employeeBuilder.telephone;
        this.salary = employeeBuilder.salary;
        this.syndicateTax = employeeBuilder.syndicateTax;
        this.spouse = employeeBuilder.spouse;
    }

    public static class EmployeeBuilder{}
}
```

```java
public static class EmployeeBuilder{

    private String name;
    private String sirName;
    private String email;
    private String cpf;
    private String telephone;
    private double salary;
    private double syndicateTax;
    private String spouse;

    public EmployeeBuilder() {
    }

    public EmployeeBuilder setEmployeeName(String employeeName){
        this.employeeName = employeeName;
        return this;
    }

    public Employee build(){
        return new Employee(this);
    }
}
```

```java
Employee newEmployee = Employee.EmployeeBuilder()
        .setEmployeeName("Alvaro")
        .setEmployeeSirName("Amorim")
        .setEmployeeEmail("aaa2@ic.ufal.br")
        .setEmployeeCPF("12345678900")
        .setEmployeeTelephone("12123434")
        .setEmployeeSalary(1000.00)
        .build();
```

Builder

```java
Employee newEmployee = new Employee("Alvaro",
        "Amorim",
        "aaa2@ic.ufal.br",
        "12345678900",
        "12123434",
        1000.00,
        0, null);
```

No Builder

Consequências

- Permite variar a representação interna do produto que é facilitado pela interface
- Isola o código para construção e representação trazendo modularidade para o código. Ou seja, diferentes directors podem usar a mesma interface para converter texto
- Oferece um controle sobre o processo de criação de um objeto
- Necessário a criação de uma BuilderConcrete pra cada representação
- Mais código
- Dificulta a Dependency Injection

```
public class EmployeeData{
    EmployeeMethod employeeMethod;

    public EmployeeData(){
        this.employeeMethod = new EmployeeMethod();
    }
}
```

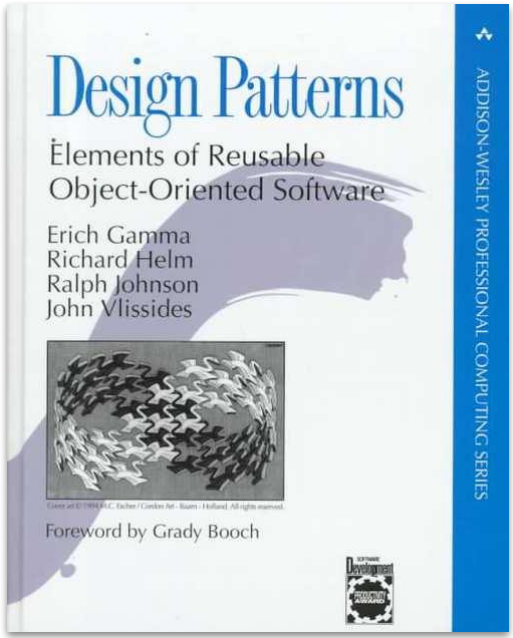
No D.I

```
public class EmployeeData{
    EmployeeMethod employeeMethod;

    public EmployeeData(EmployeeMethod employeeMethod){
        this.employeeMethod = EmployeeMethod;
    }
}
```

D.I

Referência



Design Patterns: Elements of Reusable
Object-Oriented Software