Padrão de Projeto Builder Completo

Na primeira parte desta aula, apresentamos os diagramas de classes dos dois exemplos mostrados na aula passada: Pizza sem Builder e Builder Aninhado. Depois iremos apresentar diagramas de classes correspondentes a evoluções dos diagramas anteriores.

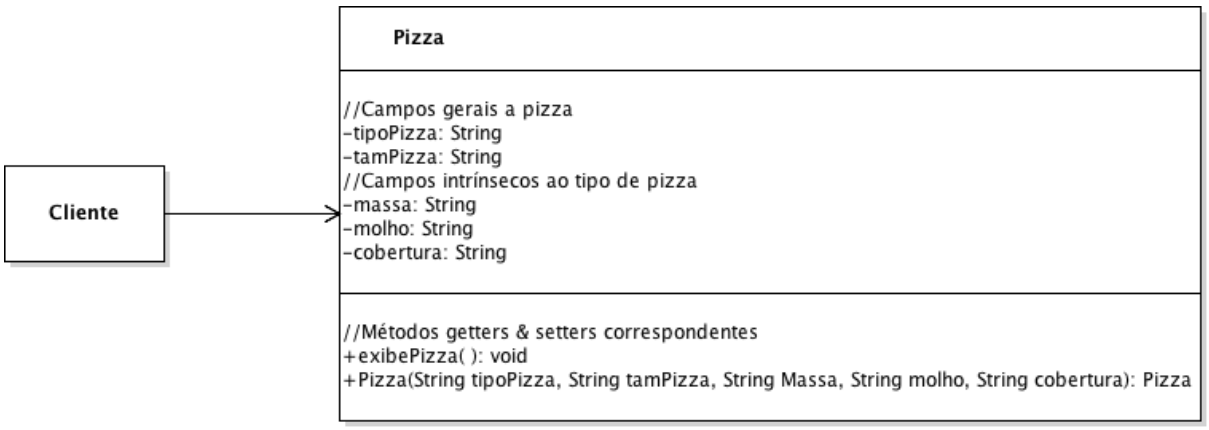
Na segunda parte, apresentamos o diagrama de classes do padrão Builder Quase Completo, bem como o diagrama de classes de uma versão dele usando interface, que se encontra no livro do Prof. Eduardo Guerra (Design Patterns com Java: Projeto Orientado a Objetos Guiado por Padrões. São Paulo: Casa do Código, 2013. [ISBN 978-85-66250-11-4]).

Na terceira parte, apresentamos o Padrão de Projeto ou DP Builder Completo, muito próximo da forma que é descrito no GoF.

Na quarta parte, apresentamos o uso do DP Builder Completo no exemplo de pizzas e usando linguagem Fluente, incluindo diagramas de classes, código fonte de cada classe do padrão, saída exemplo e dois diagramas de sequência do padrão para os casos das pizzas portuguesa e marguerita.

[1] Diagramas de Classe dos Exemplos da Aula Anterior

Pizza sem Builder:

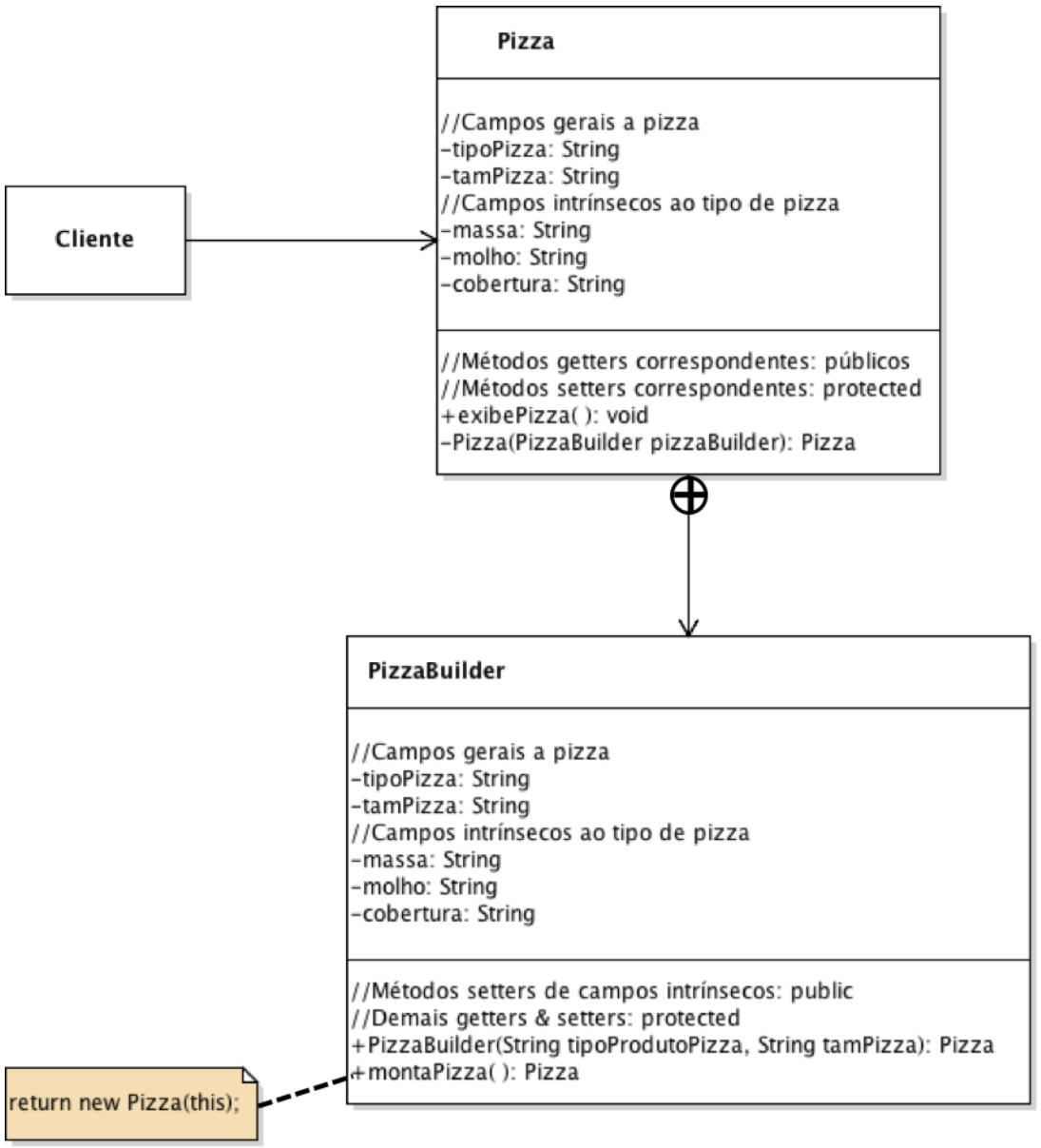


Os problemas com essa solução foram apresentados na aula anterior!

Builder Aninhado:

No diagrama abaixo, estamos usando a notação UML proposta por Uncle Bob no seu livro UML for Java Programmers, por meio da associação adornada com um pequeno círculo com um símbolo "+" interno, para representar uma classe aninhada (inner ou nested), estática ou não. Assim, a classe PizzaBuilder é uma classe aninhada estática da classe Pizza.

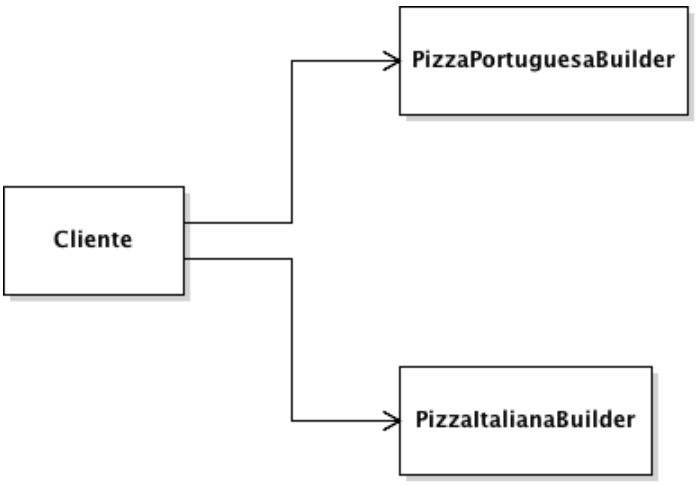
A classe Pizza representa o produto pizza, enquanto PizzaBuilder representa o construtor de um dado tipo de pizza. Esta estrutura de Builder Aninhado propicia criar uma determinada pizza usando a linguagem fluente, conforme apresentado na aula anterior. No exemplo sem Builder, essas duas estruturas estavam misturadas. O que fizemos aqui foi explicitar essa separação para mostrar que vamos ganhar em flexibilidade.



Construtor de Pizza Específico:

Em vez de usar o Builder Aninhado, pode-se definir builders para cada tipo de pizza. Com isso, o construtor passaria a ter um número menor de parâmetros, passando de 5 parâmetros da Pizza sem Builder, para apenas 1, que designaria o tamanho da pizza, se "pequena", "média" ou "grande"!

Qual é o problema desta solução? Dois problemas principais: acoplamento forte entre Cliente e os builders; e muita redundância de código.

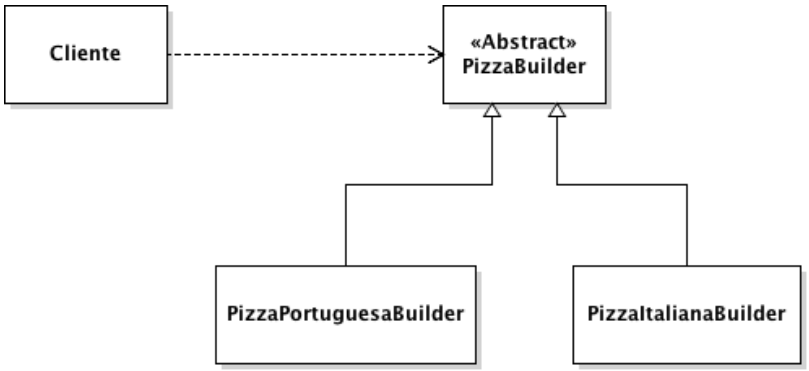


O acoplamento forte implica que temos que criar os dois tipos de objeto pizza no objeto Cliente.

Em relação ao Builder Aninhado, houve uma regressão, pois, por exemplo, a classe PizzaPortuguesaBuilder representa o produto pizza, bem como o construtor do tipo de pizza portuguesa.

Mas há agora uma representação explícita do tipo de pizza (PizzaPortuguesaBuilder e PizzaItalianaBuilder), coisa que não havia anteriormente!

Construtor de Pizza Abstrato:

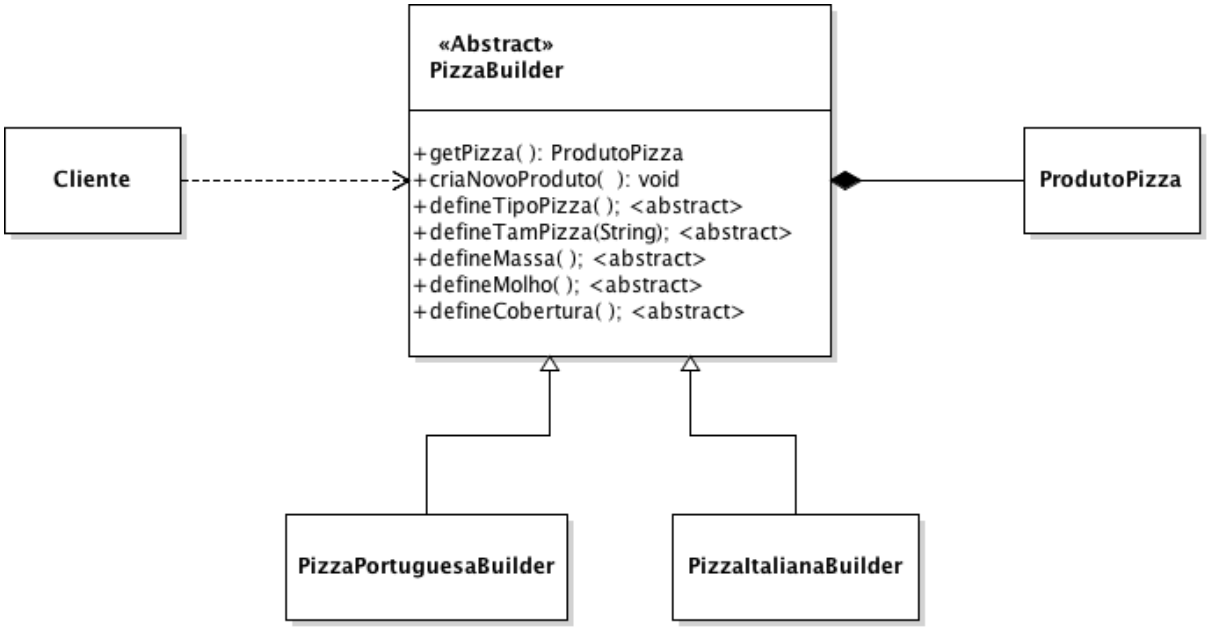


Há um ganho em flexibilidade agora, pois o acoplamento agora é abstrato, de modo que se poderia usar o Builder apropriado no objeto Cliente.

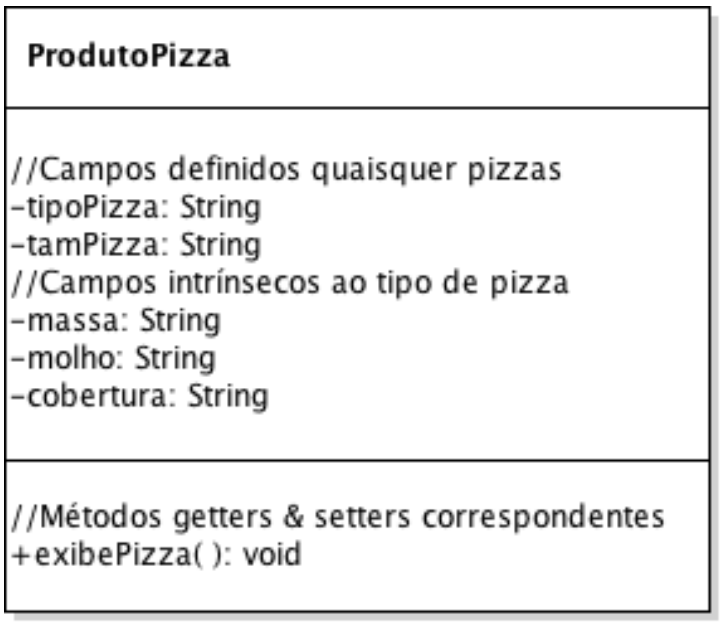
Contudo, as subclasses de PizzaBuilder ainda representam tanto o produto pizza, quanto o construtor do tipo de pizza. Essa separação va acontecer a partir do próximo forma de definir um padrão Builder.

[2] Padrão Builder Quase Completo

O diagrama abaixo constitui o Padrão Builder Quase Completo com linguagem fluente, embora não mostremos o código Java correspondente. Acreditamos que, ao mostrar o código para o Padrão Builder Completo (que vem a seguir nos itens [3] e [4]), não fique tão difícil usar linguagem fluente quando for codificar em Java este tipo de padrão.



Também expandimos a classe ProdutoPizza, para se ter uma ideia de como ela se estrutura neste caso. Fica claro que se pode definir o tipo da pizza e o seu tamanho na hora que o cliente faz o pedido, por exemplo, para o garçom numa pizzaria real. Uma vez definido o tipo de pizza, isso implica que os campos intrínsecos ao tipo de pizza escolhido já ficam determinados de antemão!



Neste caso, desacoplamos a **estrutura complexa da pizza**, aqui referenciada como classe ProdutoPizza, das subclasses de PizzaBuilder, que agora representam os construtores do tipo de pizza disponíveis. A classe ProdutoPizza se assemelha a classe Pizza do padrão Builder Aninhado, enquanto as subclasses de PizzaBuilder se assemelham à classe aninhada PizzaBuilder do Builder Aninhado.

Vale lembrar que agora os tipos de pizza têm construtor próprio, coisa que não acontecia com o padrão Builder Aninhado, uma vez que o construtor era geral. Antes era possível criar uma pizza portuguesa com ingredientes de pizza italiana. Agora não é possível, usando o builder PizzaPortuguesaBuilder, criar uma pizza que não seja com ingredientes de pizza portuguesa; não há como trocar seus ingredientes por ingredientes de pizza italiana ou marguerita!

Nada impede, contudo, que o desenvolvedor coloque o nome pizzaItaliana na variável que referencia um objeto ProdutoPizza criado pelo builder PizzaPortuguesaBuilder! Isso, contudo, é um mau cheiro que deve ser refatorado por uma simples troca do nome inapropriado por um melhor. O que importa aqui é que o objeto ProdutoPizza tenha os ingredientes típicos de pizza portuguesa!

PizzaBuilder faz uma composição de ProdutoPizza e as subclasses PizzaPortuguesaBuilder e PizzaItalianaBuilder herdam essa composição. Ou seja, a classe cliente só vai tomar conhecimento do objeto ProdutoPizza resultante ao final da montagem!

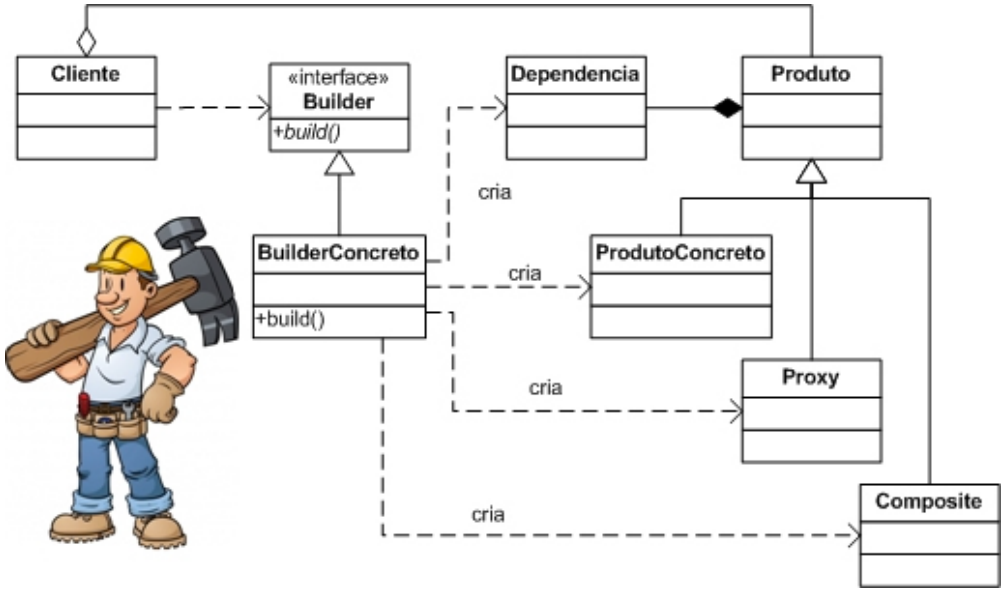
Para se criar um novo tipo de pizza, basta criar uma nova subclasse de PizzaBuilder, por exemplo, PizzaMargueritaBuilder, que vai implementar diretamente os ingredientes intrínsecos de pizza marguerita!

Nós usamos neste padrão uma classe abstrata PizzaBuilder. Poderia ser uma interface PizzaBuilder no lugar, mas aí a classe ProdutoPizza teria que ter outra forma de associação, pois não poderia ser composição de interface Java. Poderia ser uma composição ou agregação explícita criada de antemão pela classe Cliente que disponibilizaria um objeto ProdutoPizza para ser injetado nos builders específicos.

Outra alternativa, para se alcançar um resultado parecido com o do uso da classe abstrata PizzaBuilder, seria fazer com que cada classe Builder que implementa a interface PizzaBuilder faça uma composição da classe ProdutoPizza.

O diagrama abaixo exemplifica esse uso do equivalente da interface PizzaBuilder, chamada apenas Builder, das classes PizzaPortuguesaBuilder e PizzaItalianaBuilder representadas pela classe Concrete Builder, e da classe ProdutoPizza, chamada apenas Produto.

Esse exemplo foi tirado do livro de padrões do Prof. Guerra, que ele usa para ilustrar os dois hands-on sobre o padrão Builder. Como pode ser visto no diagrama, ele ilustra uma combinação do Builder (aqui denominado de Builder Quase Completo, que é um exemplo válido do Padrão Builder) com outros padrões apresentados neste curso.

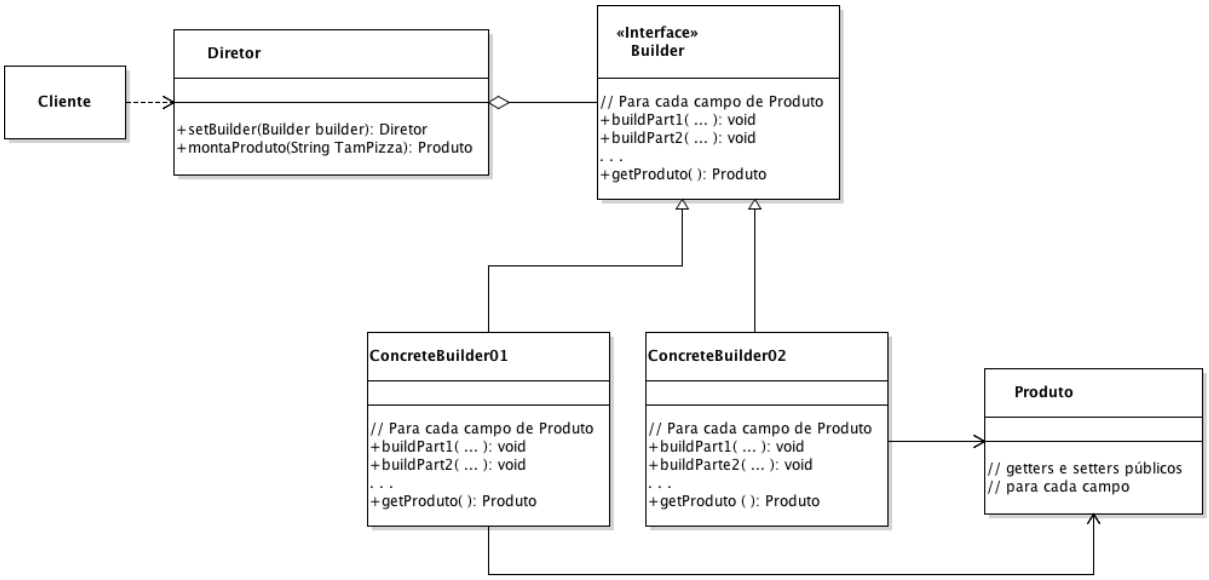


Como dito acima, nessa figura alguns outros padrões estão representados para ilustrar a complexidade que o processo de criação por meio de Builder pode possuir. Veja que a classe representada como Produto pode possuir subclasses, pode ser composta por outras, pode ser envolvida por um Proxy e pode ter suas implementações combinadas por um Composite, sem contar outros padrões que podem ser utilizados dependendo da situação e contexto de projeto.

Vale notar que nenhum desses outros padrões citados são obrigatórios para a utilização de um Builder. Porém, sua presença, apesar de aumentar a complexidade do processo de criação, acaba favorecendo a utilização de outros padrões na criação de objetos, ao mesmo tempo que torna o código mais flexível e fácil de manter.

[3] Padrão Builder Completo

O padrão de projeto Builder (Completo) é apresentado no diagrama UML abaixo:



Os elementos participantes deste padrão são os seguintes:

- **Builder** – Especifica uma interface abstrata para criar partes de um objeto complexo Produto
- **ConcreteBuilder** – 01 e 02 para exemplificar: Cada um constrói e reúne as partes (campos) do produto implementando a interface Builder. Cada um define e acompanha a representação que cria e fornece uma interface para salvar o produto.
- **Diretor** – Constrói o objeto complexo usando a interface Builder
- **Produto** – Representa o objeto complexo que está sendo construído

Builder pode ser tanto uma interface quanto uma classe abstrata. A diferença entre os tipos é que, como interface, a associação com a classe Produto é feita pelos ConcreteBuilders, enquanto que, como classe abstrata, a associação é com a própria classe abstrata!

Na literatura, alguns autores usam Diretor outros não; estou diferenciando e dizendo que com diretor é Builder Completo, mas não existe obrigação de usar uma classe Diretor; tanto é assim, que o exemplo de hands-on do Prof. Guerra faz uso de Builder sem o uso de Diretor!

O Cliente, que pode ser ou outro objeto ou o cliente real que chama o método main () do aplicativo, inicia a classe Builder e Diretor. O Builder representa o objeto complexo que precisa ser construído em termos de objetos e tipos mais simples.

O construtor na classe Diretor recebe um objeto Builder como um parâmetro do Cliente e é responsável por chamar os métodos apropriados da classe Builder. Nós dizemos que fizemos uma injeção por construtor de objeto Builder em objeto Diretor durante a execução do objeto Cliente.

Para fornecer ao Cliente uma interface para todos os Builders concretos, a classe Builder deve ser abstrata ou uma interface. Desta forma, você pode adicionar novos tipos de objetos complexos definindo apenas a estrutura e reutilizando a lógica para o processo de construção real. O Cliente é o único que precisa conhecer os novos tipos de Builders, enquanto o Diretor precisa saber quais os métodos do Builder chamar.

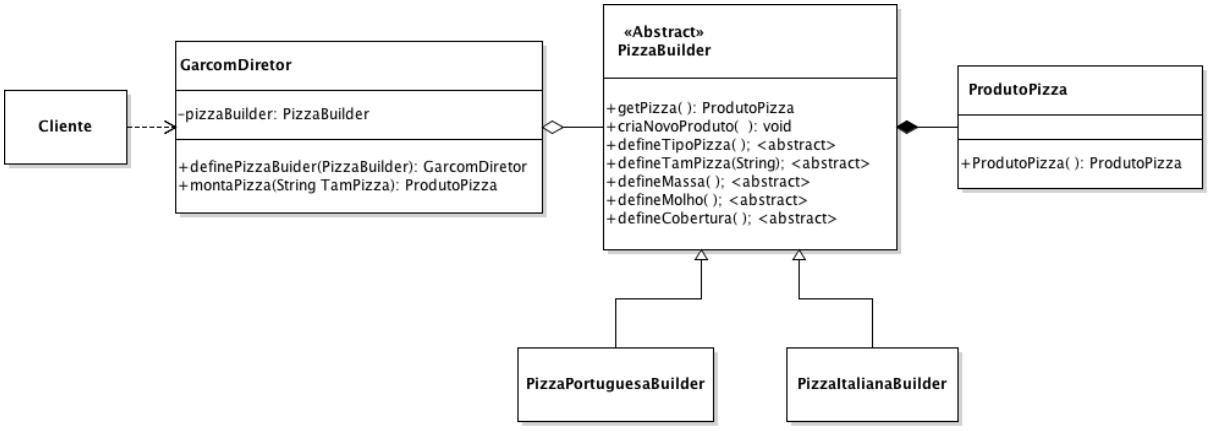
Atendendo o OCP – Open-Closed Principle, se que quiser acrescentar um novo tipo de Builder, bastar criar a classe correspondente como subclasse de Builder. ou seja, o DP Builder Completo está aberto para incluir novos Builders, e fechado a mudanças nas classes já existentes, que não precisam ser modificadas com novos Builders.

Finalmente, dizemos que o Diretor vai montar um objeto Produto de acordo com o Builder injetado. Para isso, faz-se uso do método montaProduto(parâmetros), onde poucos parâmetros variáveis do objeto Produto são passados ao objeto Builder concreto usado. Podemos usar linguagem fluente, o recomendável, ou não! No diagrama, usei um parâmetro String tamPizza para exemplificar o que está variável no exemplo do Produto do tipo pizza, mostrado na Parte [4] a seguir! Já o Builder concreto usado, incorpora aquelas características default dos campos que são intrínsecos ao produto referenciado.

O método montaProduto(parâmetros) vai inicialmente pedir que o Builder concreto escolhido crie um objeto do tipo Produto e em seguida, passando os parâmetros que são variáveis, pedir para que o objeto Builder concreto coloque as características variáveis e intrínsecas (default específicas do tipo do Builder concreto) em cada campo do objeto Produto criado!

[4] Exemplo da Pizza no Padrão Builder Completo e Fluente

Apresentamos abaixo o diagrama de classes do DP Builder Completo e Fluente aplicado ao exemplo da Pizza.



Acrescentamos um nome que associe a classe Diretor ao objetivo do aplicativo, no caso GarcomDiretor, porque ele simula o atendimento de um garçom ao cliente de uma Pizzaria. O garçom pergunta, simplificadamente, qual é o tipo da pizza e também qual o seu tamanho!

Acrescentamos o termo Pizza ao Builder, ficando PizzaBuilder, que vai ser o construtor abstrato do objeto complexo ProdutoPizza, que recebe o posfixo Pizza.

Voltando ao GarçomDiretor, o tipo da pizza desejada pelo cliente vai implicar em definir um objeto PizzaBuilder, que no caso pode ser objeto de uma das classes concretas PizzaPortuguesaBuilder ou PizzaItalianaBuilder.

O método montaPizza(String tamPizza), usando linguagem fluente, vai inicialmente pedir que o Builder concreto escolhido (PizzaPortuguesaBuilder ou PizzaItalianaBuilder) crie um objeto do tipo ProdutoPizza e em seguida, passando o parâmetro variável tamPizza, pedir para que o objeto Builder concreto coloque as características variável (tamPizza) e intrínsecas (tipos de massa, molho e cobertura do tipo do Builder concreto) em cada campo do objeto ProdutoPizza criado!

Abaixo apresentamos as classes Cliente, GarçomDiretor, PizzaBuilder, PizzaPortuguesaBuilder, PizzaItalianaBuilder, PizzaMargueritaBuilder e ProdutoPizza. Apresentamos também uma saída exemplo ao rodar a classe Cliente, bem como dois diagramas de sequência da classe Cliente, um para o exemplo da criação da pizza portuguesa, outro para a marguerita.

Apresento o funcionamento do Cliente, pela numeração, para criar uma pizza de um determinado tipo por meio do DP Builder Completo Fluente:

(1) Cria Diretor --> GarcomDiretor
(2) Cria Builder para pizza Portuguêsa
(3) injeta Builder no GarcomDiretor
(4) Monta e entrega produto: pizza Portuguesa pequena
(5) Usa produto pizza

Mostramos também que o ODP foi satisfeito ao introduzir a pizza marguerita na classe Cliente, de modo que apenas a classe PizzaMargueritaBuildor precisou ser incluída como subclasse da classe PizzaBuilder, sem alteração alguma nas outras classes já existentes no DP Builder Completo Fluente.

No exemplo da criação de uma pizza portuguesa, fizemos uso híbrido de linguagem fluente e não fluente. Nesse caso, pedimos para montar uma pizza portuguesa pequena.

No exemplo da criação de uma pizza italiana, fizemos uso apenas da linguagem não fluente. Nesse caso, pedimos para montar uma pizza italiana média.

E, finamente, no exemplo de uma criação da pizza marguerita, fizemos uso apenas da linguagem fluente. Nesse caso, pedimos para montar uma pizza marguerita grande.

Ou seja, se definirmos o uso da linguagem fluente no DP Builder Completo, obtemos grátis o uso não fluente. O contrário não é verdadeiro! Mas ressaltamos que o aconselhável é usar a linguagem fluente, pelos benefícios que traz para a compreensão e manutenção do código assim produzido!

No DP Builder Completo Fluente, fica bem separado o trabalho do garçom, representado pela classe GarcomDiretor, dos tipos específicos de pizzas (portuguesa, italiana e marguerita), representados pelos Builders específicos subclasses de PizzaBuilder (PizzaPortuguesaBuilder, PizzaItalianaBuilder e PizzaMargueritaBuilder), bem como da representação pura do objeto complexo pizza, representado pela classe ProdutoPizza.

```
-- ---   ----
public class Cliente {
           public static void main(String[] args) {
               // (1) cria Diretor --> GarcomDiretor
               GarcomDiretor garcom = new GarcomDiretor();
               // (2) cria Builder para pizza Portuguêsa
               PizzaBuilder pizzaPortuguesaBuilder = new PizzaPortuguesaBuilder();
               // (3) injeta Builder no GarcomDiretor
               ProdutoPizza pp = garcom.definePizzaBuilder( pizzaPortuguesaBuilder )
                       .montaPizza("pequena");    // (4) monta e entrega produto:
                                                  //     pizza Portuguesa pequena
               pp.exibePizza();     // (5) usa produto pizza

               System.out.println("\n-- ---  ----\n");
               // (2) cria Builder para pizza Italiana
               PizzaBuilder pizzaItalianaBuilder = new PizzaItalianaBuilder();
               // (3) injeta Builder no GarcomDiretor
               garcom.definePizzaBuilder( pizzaItalianaBuilder );
               pp = garcom.montaPizza("media");    // (4) monta e entrega produto:
                                                   //      pizza Italiana média
               pp.exibePizza();     // (5) usa produto pizza

               System.out.println("\n-- ---  ----\n");
               // (2) cria Builder para pizza Margueríta
               PizzaBuilder pizzaMargueritaBuilder = new PizzaMargueritaBuilder();
               // (3) injeta Builder no GarcomDiretor
               garcom.definePizzaBuilder( pizzaMargueritaBuilder )
                     .montaPizza("grande") // (4)  monta e entrega produto:
                                           //      pizza Marguerita grande
                     .exibePizza();// (5) usa produto pizza
         }
}

public class GarcomDiretor {
```

```
	private PizzaBuilder pizzaBuilder;
	public GarcomDiretor definePizzaBuilder(PizzaBuilder pizzaBuilder) {
		// (2.1) associa PizzaBuilder
		this.pizzaBuilder = pizzaBuilder;
		// (2.2) devolve objeto GarcomDiretor
		return this;
	}
	public ProdutoPizza montaPizza(String tamPizza) {
		// (4.1) cria novo Produto, agora com tamanho definido!
		pizzaBuilder.criaNovoProdutoPizza();
		pizzaBuilder.defineTipoPizza();
		pizzaBuilder.defineTamPizza(tamPizza);
		pizzaBuilder.defineMassa();
		pizzaBuilder.defineMolho();
		pizzaBuilder.defineCobertura();
		// (4.2) entrega produto
		return pizzaBuilder.getPizza();
	}
}

public abstract class PizzaBuilder {
	protected ProdutoPizza pizza;
	public ProdutoPizza getPizza() {
		return pizza;
	}
	public void criaNovoProdutoPizza() {
		pizza = new ProdutoPizza();
	}
	public abstract void defineTipoPizza();
	public abstract void defineTamPizza(String tamPizza);
	public abstract void defineMassa();
	public abstract void defineMolho();
	public abstract void defineCobertura();
}

public class PizzaPortuguesaBuilder extends PizzaBuilder {
	@Override
	public void defineTipoPizza() {
		pizza.defineTipoPizza("Portuguesa");
	}
	@Override
	public void defineTamPizza(String tamPizza) {
		pizza.defineTamPizza(tamPizza);
	}
	@Override
	public void defineMassa() {
		pizza.defineMassa("fina");
	}
	@Override
	public void defineMolho() {
		pizza.defineMolho("não apimentado");
	}
	@Override
	public void defineCobertura() {
		pizza.defineCobertura("ovo+azeitona");
	}
}

public class PizzaItalianaBuilder extends PizzaBuilder {
	@Override
	public void defineTipoPizza() {
```

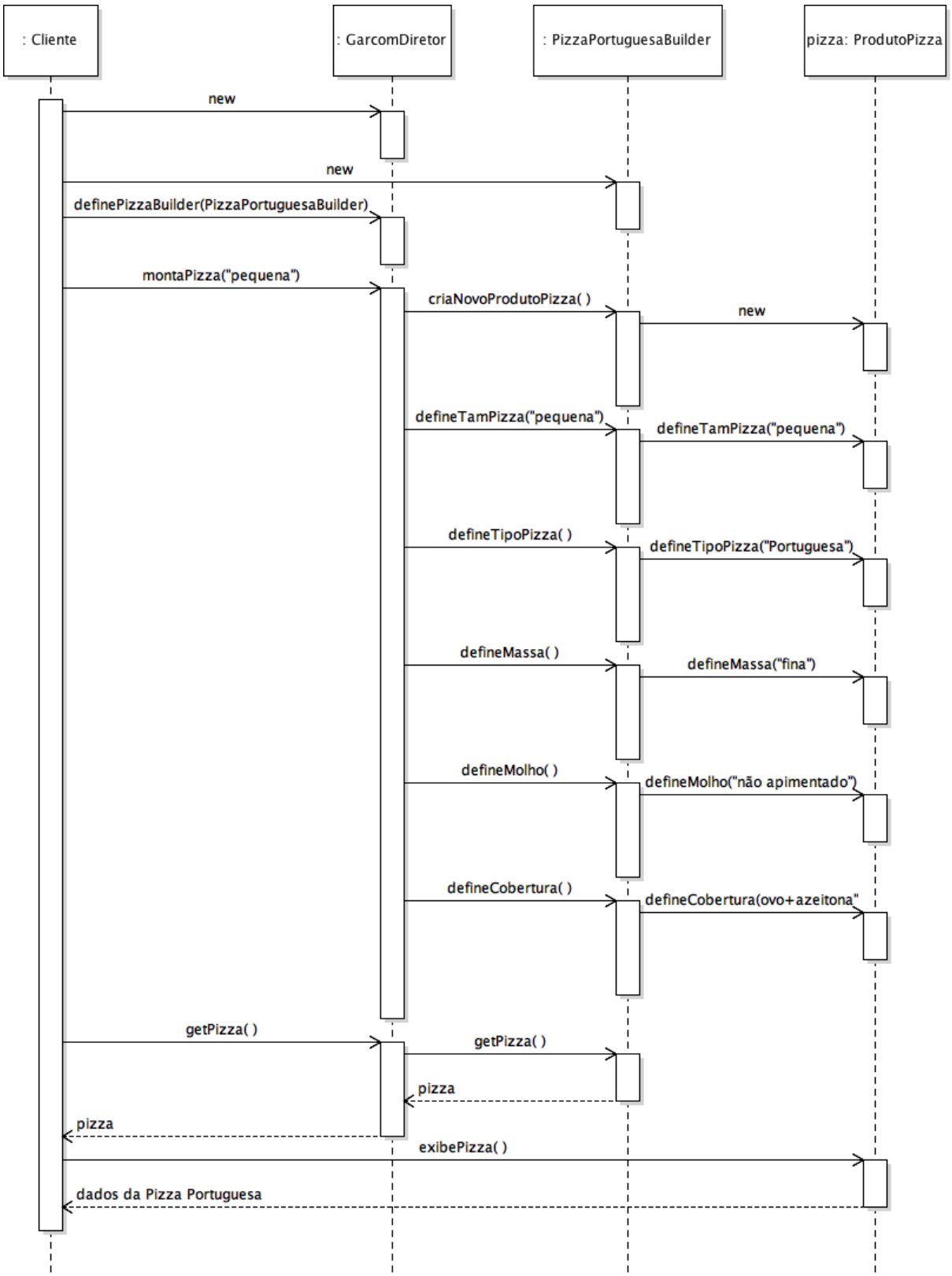
```
                pizza.defineTipoPizza("Italiana");
        }
        @Override
        public void defineTamPizza(String tamPizza) {
                pizza.defineTamPizza(tamPizza);
        }
        @Override
        public void defineMassa() {
                pizza.defineMassa("grossa");
        }
        @Override
        public void defineMolho() {
                pizza.defineMolho("apimentado");
        }
        @Override
        public void defineCobertura() {
                pizza.defineCobertura("pepperoni+salame");
        }
}

public class PizzaMargueritaBuilder extends PizzaBuilder {
        @Override
        public void defineTipoPizza() {
                pizza.defineTipoPizza("Marguerita");
        }
        @Override
        public void defineTamPizza(String tamPizza) {
                pizza.defineTamPizza(tamPizza);
        }
        @Override
        public void defineMassa() {
                pizza.defineMassa("fina");
        }
        @Override
        public void defineMolho() {
                pizza.defineMolho("tomate");
        }
        @Override
        public void defineCobertura() {
                pizza.defineCobertura("tomate+orégano");
        }
}

public class ProdutoPizza {
        public void defineTipoPizza(String tipoPizza) {
                this.tipoPizza = tipoPizza;
        }
        public void defineTamPizza(String tamPizza) {
                this.tamPizza = tamPizza;
        }
        public void defineMassa(String massa) {
                this.massa = massa;
        }
        public void defineMolho(String molho) {
                this.molho = molho;
        }
        public void defineCobertura(String cobertura) {
                this.cobertura = cobertura;
        }
        public String getTipoProdutoPizza() {
                return tipoPizza;
```

```
        }
        public String getTamPizza() {
                return tamPizza;
        }
        public String getMassa() {
                return massa;
        }
        public String getMolho() {
                return molho;
        }
        public String getCobertura() {
                return cobertura;
        }
        public void exibePizza( ){
                System.out.println(
                                "Pizza: "            + getTipoProdutoPizza( ) +
                                "\nTamanho: " + getTamPizza( ) +
                                "\nMassa: "   + getMassa( ) +
                                "\nMolho: "   + getMolho( ) +
                                "\nCobertura: " + getCobertura( ));
        }
        // campos definidos pelo Garcom
        private String tipoPizza    = "";
        private String tamPizza     = "";
        // campos intrínsecos ao tipo de pizza
        private String massa        = "";
        private String molho        = "";
        private String cobertura    = "";
}
```

Saída:

```
Pizza: Portuguesa
Tamanho: pequena
Massa: fina
Molho: não apimentado
Cobertura: ovo+azeitona

-- ---  ----

Pizza: Italiana
Tamanho: media
Massa: grossa
Molho: apimentado
Cobertura: pepperoni+salame

-- ---  ----

Pizza: Marguerita
Tamanho: grande
Massa: fina
Molho: tomate
Cobertura: tomate+orégano
```

<u>Diagrama de Sequência do DP Builder Completo Fluente</u>: exemplo da pizza portuguesa!



<u>Diagrama de Sequência do DP Builder Completo Fluente</u>: exemplo da pizza marguerita!