# Padrão de Projeto Builder Aninhado

O padrão de projeto Builder resolve o problema de classes complexas, que têm um grande número de parâmetros opcionais e estado inconsistente (que se reflete no/s construtor/es), oferecendo tanto uma maneira de construir o objeto passo a passo, quanto de fornecer um método que realmente retornará o objeto final.

O procedimento abaixo parte de uma classe sem Builder, embutindo nela a capacidade de montar e entregar objetos instância sem os problemas apontados para a classe sem Builder. Iremos mostrar passo a passo como isso se faz, incluindo o uso de interface fluente. Iremos exemplificar usando a classe Pizza sem Builder mostrada na aula anterior.

O procedimento é apenas didático, para entender o que está por trás do Builder Aninhado; na prática, algumas modificações podem se mostrar mais efetivas e realistas, quando já se sabe o que fazer!

## Procedimento:

Entrada: classe Pizza sem Builder

Saída: classe Pizza com Builder Aninhado

**Passos:**

**0** – Caso a classe faça uso do Telescoping Constructor Anti-pattern, eliminar todos os construtores que não tratam todos os atributos da classe Pizza, deixando apenas o construtor completo!

**1** – Criar uma classe aninhada estática PizzaBuilder dentro da classe Pizza, aqui denominada classe externa à classe aninhada PizzaBuilder.

> Obs.: Como poderemos ter vários Builders num programa, uma convenção de nomenclatura aplicável é anexar o termo "Builder" ao final do nome da classe externa. Por exemplo, se o nome da classe for Computador, a classe do construtor deve ser ComputadorBuilder; se for Pessoa, PessoaBuilder. No nosso caso, PizzaBuilder.

**2** – Espelhamento de Atributos: Copiar todos os atributos obrigatórios e opcionais da classe externa Pizza para a classe PizzaBuilder.

**3** – Copiar todos os correspondentes métodos getters e setters associados aos atributos da classe externa Pizza, se eles já existiam. Se não, criá-los em ambas as classes!

**4** – Os métodos setters de atributos obrigatórios da classe PizzaBuilder devem ser tornados protected, se já não estiverem assim!

**5** – Tornar protected os modificadores de visibilidade de todos os métodos setters da classe externa Pizza, se eles já existiam. A ideia é não permitir que os atributos possam ser modificados de clientes da classe Pizza.

**6** – A classe PizzaBuilder deve ter um construtor público com todos os atributos obrigatórios como parâmetros, procurando não passar de 4 parâmetros; se sobrar atributos obrigatórios, eles devem ser tratados como os parâmetros opcionais do Passo 7 abaixo.

**7** – A classe PizzaBuilder deve ter métodos setters públicos para definir os parâmetros opcionais (e eventuais parâmetros obrigatórios restantes), se eles ainda não existiam, como evidenciado pelo Passo 3 acima.

**8** – Interface Fluente: Cada método setter da classe PizzaBuilder deve retornar o mesmo objeto PizzaBuilder, após definir o atributo opcional (ou obrigatório restante).

**9** – Construtor da classe externa Pizza:

> **9.1** – Trocar todos os seus parâmetros anteriores por um único: objeto da classe PizzaBuilder.
>
> **9.2** – Redefinir todos os atributos do objeto da classe externa Pizza com os valores obtidos pelo objeto da classe PizzaBuilder.
>
> **9.3** – Tornar privado o construtor da classe externa Pizza, que ainda assim será acessível pela classe aninhada PizzaBuilder.

**10** – A etapa final é fornecer um método montaPizza( ) na classe PizzaBuilder, que deverá retornar um novo objeto da classe Pizza, passando como parâmetro o próprio objeto da classe PizzaBuilder (this).

> Obs.: Pode-se usar apenas monta( ), build( ) ou anexar o nome da classe externa Pizza ao final do método: montaPizza( ) ou buildPizza( ).

## [2] Exemplificando o Procedimento com os Códigos das Classes Pizza e PizzaBuilder

**Passo 0** – Caso a classe faça uso do Telescoping Constructor Anti-pattern, eliminar todos os construtores que não tratam todos os atributos da classe Pizza, deixando apenas o construtor completo!

```
public class Pizza {

	public Pizza(String tipoPizza, String tamPizza, String massa, String molho,
                                                                String cobertura){
		this.defineTipoProdutoPizza(tipoPizza);
		this.defineTamPizza(tamPizza);
		this.defineMassa(massa);
		this.defineMolho(molho);
		this.defineCobertura(cobertura);
	}
	. . .
}
```

**Passo 1** – Criar uma classe aninhada estática PizzaBuilder dentro da classe Pizza, aqui denominada classe externa à classe aninhada PizzaBuilder.

```
public class Pizza {

	public Pizza(String tipoPizza, String tamPizza, String massa, String molho,
                                                                String cobertura){
		. . .
	}
	. . .
	public static class PizzaBuilder {
		. . .
	}
	. . .
}
```

**Passo 2** – Espelhamento de Atributos: Copiar todos os atributos obrigatórios e opcionais da classe externa Pizza para a classe PizzaBuilder.

```
public class Pizza {

	public Pizza(String tipoPizza, String tamPizza, String massa, String molho,
                                                                String cobertura){
		. . .
	}
	. . .
	public static class PizzaBuilder {
		. . .


		// campos obrigatórios
		private String tipoPizza    = "";
		private String tamPizza     = "";
		// campos opcionais
		private String massa        = "";
		private String molho        = "";
		private String cobertura    = "";
	}

	// campos obrigatórios
	private String tipoPizza    = "";
	private String tamPizza     = "";
	// campos opcionais
	private String massa        = "";
	private String molho        = "";
	private String cobertura    = "";
```

```
}
```

**Passo 3** – Copiar na classe aninhada PizzaBuilder todos os correspondentes métodos getters e setters associados aos atributos da classe externa Pizza, se eles já existiam. Se não, criá-los em ambas as classes!

```
public class Pizza {

	public Pizza(String tipoPizza, String tamPizza, String massa, String molho,
                                                                String cobertura){
		. . .
	}
	. . .
	public static class PizzaBuilder {
		. . .
		protected void defineTipoProdutoPizza(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;
		}
		. . .
	}
	. . .
}
```

**Passo 4**. Os métodos setters de atributos obrigatórios da classe PizzaBuilder devem ser tornados protected, se já não estiverem assim!

A ideia aqui é não permitir que, por serem atributos obrigatórios, eles não possam ser mudados depois da criação do objeto PizzaBuilder! O método defineTipoProdutoPizza( ) já era protected, enquanto o método defineTamPizza( ) precisou ter sua visibilidade mudada para protected!

```
public class Pizza {

	public Pizza(String tipoPizza, String tamPizza, String massa, String molho,
                                                                String cobertura){
		. . .
	}
	. . .
	public static class PizzaBuilder {
		. . .
		protected void defineTipoProdutoPizza(String tipoPizza) {
			this.tipoPizza = tipoPizza;
		}

		protected void defineTamPizza(String tamPizza) {
			this.tamPizza = tamPizza;
		}
		. . .
		// campos obrigatórios
		private String tipoPizza    = "";
		private String tamPizza     = "";
		// campos opcionais
		private String massa        = "";
		private String molho        = "";
		private String cobertura    = "";
	}
	. . .
}
```

**Passo 5**. Tornar protected os modificadores de visibilidade de todos os métodos setters da classe externa Pizza, se eles já existiam. A ideia é não permitir que os atributos possam ser modificados de clientes de objetos da classe Pizza.

```
public class Pizza {

	public Pizza(String tipoPizza, String tamPizza, String massa, String molho,
                                                                String cobertura){
		. . .
	}

	protected void defineTipoProdutoPizza(String tipoPizza) {
		this.tipoPizza = tipoPizza;
	}

	protected void defineTamPizza(String tamPizza) {
		this.tamPizza = tamPizza;
	}

	protected void defineMassa(String massa) {
		this.massa = massa;
```

```
	}

	protected void defineMolho(String molho) {
		this.molho = molho;
	}

	protected void defineCobertura(String cobertura) {
		this.cobertura = cobertura;
	}
	. . .
	public static class PizzaBuilder {
		. . .

	}
	. . .
}
```

**Passo 6**. A classe PizzaBuilder deve ter um construtor público com todos os atributos obrigatórios como parâmetros, procurando não passar de 4 parâmetros; se houver mais de 4 atributos obrigatórios, eles devem ser tratados como os parâmetros opcionais do Passo 7 abaixo.

```
public class Pizza {

	public Pizza(String tipoPizza, String tamPizza, String massa, String molho,
                                                                String cobertura){
		. . .
	}
	. . .
	public static class PizzaBuilder {
		. . .

		public PizzaBuilder (String tipoProdutoPizza, String tamPizza){
			this.defineTipoPizza(tipoProdutoPizza);
			this.defineTamPizza(tamPizza);
		}
		. . .
		// campos obrigatórios
		private String tipoPizza    = "";
		private String tamPizza     = "";
		// campos opcionais
		private String massa        = "";
		private String molho        = "";
		private String cobertura    = "";
	}
	. . .
}
```

**Passo 7**. A classe PizzaBuilder deve ter métodos setters públicos para definir os parâmetros opcionais (e eventuais parâmetros obrigatórios restantes), se eles ainda não existiam, como evidenciado pelo Passo 3 acima.

No caso eles já existiam na classe Pizza e foram disponibilizados na classe PizzaBuilder dessa forma, como exemplificado no Passo 3 acima e reforçado aqui!

```
public class Pizza {
```

```
        public Pizza(String tipoPizza, String tamPizza, String massa, String molho,
                                                                String cobertura){
                . . .
        }
        . . .
        public static class PizzaBuilder {
                . . .

                public PizzaBuilder (String tipoProdutoPizza, String 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;
                . . .
                // campos obrigatórios
                private String tipoPizza    = "";
                private String tamPizza     = "";
                // campos opcionais
                private String massa        = "";
                private String molho        = "";
                private String cobertura    = "";
        }
        . . .
}
```

**Passo 8**. Interface Fluente: Cada método setter da classe PizzaBuilder deve retornar o mesmo objeto PizzaBuilder, após definir no seu corpo o atributo opcional (ou obrigatório restante) correspondente.

```
public class Pizza {

        public Pizza(String tipoPizza, String tamPizza, String massa, String molho,
                                                                String cobertura){
                . . .
        }
        . . .
        public static class PizzaBuilder {
                . . .

                public PizzaBuilder (String tipoProdutoPizza, String tamPizza){
                        …
                }
                . . .
                public PizzaBuilder defineMassa(String massa) {
                        this.massa = massa;
                        return this;
                }

                public PizzaBuilder defineMolho(String molho) {
                        this.molho = molho;
                        return this;
```

```
        }

        public PizzaBuilder defineCobertura(String cobertura) {
            this.cobertura = cobertura;
            return this;
        }
        . . .
        // campos obrigatórios
        private String tipoPizza    = "";
        private String tamPizza     = "";
        // campos opcionais
        private String massa        = "";
        private String molho        = "";
        private String cobertura    = "";
    }
    . . .
}
```

**Passo 9**. Construtor da classe externa Pizza:

**9.1** Trocar todos os seus parâmetros anterior por um único: objeto da classe PizzaBuilder.

```
public class Pizza {

    public Pizza(PizzaBuilder builder){
        . . .
    }

    public static class PizzaBuilder {
        . . .
    }
    . . .
}
```

**9.2** Redefinir todos os atributos do objeto da classe externa Pizza com os valores obtidos pelo objeto da classe PizzaBuilder.

Usamos os getters de PizzaBuilder e os setters de Pizza para que a tarefa seja feita!

```
public class Pizza {

    public Pizza(PizzaBuilder builder){
        this.defineTipoPizza(builder.getTipoProdutoPizza());
        this.defineTamPizza(builder.getTamPizza());
        this.defineMassa(builder.getMassa());
        this.defineMolho(builder.getMolho());
        this.defineCobertura(builder.getCobertura());
    }

    public static class PizzaBuilder {
        . . .
    }
    . . .
}
```

**9.3** Tornar privado o construtor da classe externa Pizza, que ainda assim será acessível pela classe aninhada PizzaBuilder.

Com isso, nenhum cliente pode invocar o construtor de Pizza; só se pode criar um objeto Pizza por meio do PizzaBuilder, que, por ser classe aninhada de Pizza, tem acesso ao construtor de Pizza!

```
public class Pizza {

	private Pizza(PizzaBuilder builder){
		this.defineTipoPizza(builder.getTipoProdutoPizza());
		this.defineTamPizza(builder.getTamPizza());
		this.defineMassa(builder.getMassa());
		this.defineMolho(builder.getMolho());
		this.defineCobertura(builder.getCobertura());
	}

	public static class PizzaBuilder {
		. . .
	}
	. . .
}
```

**Passo 10**. A etapa final é fornecer um método montaPizza( ) na classe PizzaBuilder, que deverá retornar um novo objeto da classe Pizza, passando como parâmetro o próprio objeto da classe PizzaBuilder (this).

```
public class Pizza {

	private Pizza(PizzaBuilder builder){
		. . .
	}

	public static class PizzaBuilder {
		. . .
		public Pizza montaPizza() {
			return new Pizza(this);
		}
		. . .
	}
	. . .
}
```

O código Cliente ilustra o uso da classe Pizza, com a criação de objetos Pizza para pizzas portuguesa (totalmente fluente) e italiana (não totalmente fluente).

```
public class Cliente {
	public static void main(String[] args) {
		// (1) cria Pizza Portuguesa
	 new Pizza
			.PizzaBuilder("Portuguesa", "pequena")
			.defineMassa("fina")
			.defineMolho("não apimentado")
			.defineCobertura("ovo+azeitona")
			.montaPizza( )// (2) monta e entrega produto pizza portuguesa
			.exibePizza();// (3) usa produto
```

```
        System.out.println("\n-- ---  ----\n");

              // (1) cria Pizza Italiana
        Pizza italiana = new Pizza
                    .PizzaBuilder("Italiana", "média")
                    .defineCobertura("pepperoni+salame")
                    .defineMolho("apimentado")
                    .defineMassa("grossa")
                    .montaPizza( );      // (2) monta e entrega produto pizza italiana
        italiana.exibePizza();     // (3) usa produto
              . . .
              // usa o produto pizza italiana em outro contexto!
              . . .
    }
}
```

**Saída**:

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

-- ---  ----

Pizza: Italiana
Tamanho: média
Massa: grossa
Molho: apimentado
Cobertura: pepperoni+salame
- -  -
```

**Observações Adicionais**:

1) Observe que, ao criar a pizza italiana, eu inverti de propósito a ordem "normal" de descrever uma pizza nas mensagens enviadas [defineMolho( ), defineMassa( ) e defineCobertura( )]e isso não atrapalhou em nada o armazenamento dos seus respectivos valores, como se pode ver pela saída abaixo.

2) Note também, como fica mais fácil de ler e entender o que se está fazendo com o uso da interface fluente. Como a intenção está explícita, fica mais fácil entender o código para eventuais futuras mudanças!

3) Ao criar a pizza portuguesa, o Cliente não precisava mais fazer uso dela, a não ser para exibir seu conteúdo. No caso da pizza italiana, desejava-se fazer uso dela além de exibir seu conteúdo; por isso, no primeiro caso não foi preciso guardar o objeto da pizza portuguesa numa variável, enquanto no segundo caso foi estritamente necessário.

4) Na hora de usar, o Cliente precisa realizar três operações: (1) criar o objeto PizzaBuilder; (2) montar o produto pizza; (3) usar o produto criado. Veja como ficou:

```
        // (1) cria Pizza Italiana
 Pizza italiana = new Pizza
                .PizzaBuilder("Italiana", "média")
                .defineCobertura("pepperoni+salame")
                .defineMolho("apimentado")
                .defineMassa("grossa")
                .montaPizza( );       // (2) monta e entrega produto pizza italiana
 italiana.exibePizza();      // (3) usa produto
```

5) Exceto no construtor de PizzaBuilder, em que se pode mudar os valores dos parâmetros correspondentes a atributos obrigatórios, não há mais possibilidade de se cometer esse engano com valores de atributos opcionais! O motivo é a interface fluente empregada em dar nomes apropriados aos métodos, que me ajuda a lembrar a sua finalidade com mais rapidez, sem precisar recorrer à documentação das classes.

6) Nada impedirá ainda o Cliente de fazer um objeto chamado portuguesa receber uma pizza italiana, ou até mesmo de receber uma pizza portuguesa de verdade, mas com ingredientes de italiana, como abaixo:

```
        // (1) cria Pizza Portuguesa
 Pizza portuguesa = new Pizza
                .PizzaBuilder("Italiana", "média")
                .defineCobertura("pepperoni+salame")
                .defineMolho("apimentado")
                .defineMassa("grossa")
                .montaPizza( );       // (2) monta e entrega produto pizza italiana
 portuguesa.exibePizza();    // (3) usa produto
```

➔ A solução para esses problemas só virá com o uso do padrão de projeto Builder Completo!

— — ——

**<u>Listagem dos Códigos da Classe Pizza e da sua Classe Aninhada PizzaBuilder</u>**:

```
public class Pizza {

        private Pizza(PizzaBuilder builder){
                this.defineTipoPizza(builder.getTipoProdutoPizza());
                this.defineTamPizza(builder.getTamPizza());
                this.defineMassa(builder.getMassa());
                this.defineMolho(builder.getMolho());
                this.defineCobertura(builder.getCobertura());
        }

        protected void defineTipoPizza(String tipoPizza) {
                this.tipoPizza = tipoPizza;
        }

        protected void defineTamPizza(String tamPizza) {
```

```
        this.tamPizza = tamPizza;
}

protected void defineMassa(String massa) {
        this.massa = massa;
}

protected void defineMolho(String molho) {
        this.molho = molho;
}

protected 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( ));
}

public static class PizzaBuilder {

        public PizzaBuilder (String tipoProdutoPizza, String tamPizza){
                this.defineTipoPizza(tipoProdutoPizza);
                this.defineTamPizza(tamPizza);
        }

        protected void defineTipoPizza(String tipoPizza) {
                this.tipoPizza = tipoPizza;
        }

        protected void defineTamPizza(String tamPizza) {
                this.tamPizza = tamPizza;
        }

        public PizzaBuilder defineMassa(String massa) {
                this.massa = massa;
                return this;
```

```
        }

        public PizzaBuilder defineMolho(String molho) {
            this.molho = molho;
            return this;
        }

        public PizzaBuilder defineCobertura(String cobertura) {
            this.cobertura = cobertura;
            return this;
        }

        protected String getTipoProdutoPizza() {
            return tipoPizza;
        }

        protected String getTamPizza() {
            return tamPizza;
        }

        protected String getMassa() {
            return massa;
        }

        protected String getMolho() {
            return molho;
        }

        protected String getCobertura() {
            return cobertura;
        }

        public Pizza montaPizza() {
            return new Pizza(this);
        }

        // campos obrigatórios
        private String tipoPizza = "";
        private String tamPizza  = "";
        // campos opcionais
        private String massa  = "";
        private String molho  = "";
        private String cobertura = "";
    }
    // campos obrigatórios
    private String tipoPizza = "";
    private String tamPizza  = "";
    // campos opcionais
    private String massa  = "";
    private String molho  = "";
    private String cobertura = "";
}
```

— — —

**Listagem do Código da Classe Cliente**:

```
// Não todo fluente
package pizzaComBuilderAninhado;

public class Cliente {
	public static void main(String[] args) {
	// (1) cria Pizza Portuguesa
	 Pizza portuguesa = new Pizza
		.PizzaBuilder("Portuguesa", "pequena")
		.defineMassa("fina")
		.defineMolho("não apimentado")
		.defineCobertura("ovo+azeitona")
		.montaPizza( );	// (2) monta e entrega produto pizza portuguesa
	 portuguesa.exibePizza();	// (3) usa produto

	 System.out.println("\n-- ---  ----\n");

	// (1) cria Pizza Italiana
	 Pizza italiana = new Pizza
		.PizzaBuilder("Italiana", "média")
		.defineMassa("grossa")
		.defineMolho("apimentado")
		.defineCobertura("pepperoni+salame")
		.montaPizza( );	// (2) monta e entrega produto pizza italiana
	 italiana.exibePizza();	// (3) usa produto

	 System.out.println("\n-- ---  ----\n");

	// (1) cria Pizza Marguerita
	 Pizza marguerita = new Pizza
		.PizzaBuilder("Marguerita", "grande")
		.defineMassa("fina")
		.defineMolho("tomate")
		.defineCobertura("tomate+orégano")
		.montaPizza( );	// (2) monta e entrega produto pizza marguerita
	 marguerita.exibePizza();	// (3) usa produto
    }
}
- -  --
```

```
// Todo fluente
package pizzaComBuilderAninhado;

public class Cliente {
	public static void main(String[] args) {
	// (1) cria Pizza Portuguesa
	 new Pizza
		.PizzaBuilder("Portuguesa", "pequena")
		.defineMassa("fina")
		.defineMolho("não apimentado")
		.defineCobertura("ovo+azeitona")
		.montaPizza( )// (2) monta e entrega produto pizza portuguesa
		.exibePizza();// (3) usa produto

	 System.out.println("\n-- ---  ----\n");

	// (1) cria Pizza Italiana
	 new Pizza
		.PizzaBuilder("Italiana", "média")
		.defineMassa("grossa")
```

```
                .defineMolho("apimentado")
                .defineCobertura("pepperoni+salame")
                .montaPizza( )// (2) monta e entrega produto pizza italiana
                .exibePizza();// (3) usa produto

         System.out.println("\n-- ---  ----\n");

        // (1) cria Pizza Marguerita
         new Pizza
                .PizzaBuilder("Marguerita", "grande")
                .defineMassa("fina")
                .defineMolho("tomate")
                .defineCobertura("tomate+orégano")
                .montaPizza( )// (2) monta e entrega produto pizza marguerita
                .exibePizza();// (3) usa produto
    }
}
-- --- ----
```

Obs.1: Neste exemplo da classe Cliente abaixo, enfatizo de novo que a ordem para definir molho, cobertura e massa não é fixa, podendo ocorrer em qualquer ordem possível. Contudo, o resultado final, após montar a pizza correspondente e a exibir, será sempre na ordem predeterminada em exibePizza( )!

Obs. 2: No caso da Pizza Italiana, mostro que posso misturar o uso de código fluente com o não todo fluente na mesma classe! Incentiva-se o uso do código fluente sempre!

```
// Com ordem diferente para defineMolho( ),
// defineCobertura( ) e defineMassa( )
// para os 3 tipos de pizza!
// Código da Pizza Italiana Não Todo Fluente
public class Cliente {
       public static void main(String[] args) {
       // (1) cria Pizza Portuguesa
        new Pizza
                .PizzaBuilder("Portuguesa", "pequena")
                .defineMolho("não apimentado")
                .defineCobertura("ovo+azeitona")
                .defineMassa("fina")
                .montaPizza( )// (2) monta e entrega produto pizza portuguesa
                .exibePizza();// (3) usa produto

         System.out.println("\n-- ---  ----\n");

        // (1) cria Pizza Italiana
         Pizza italiana = new Pizza
                .PizzaBuilder("Italiana", "média")
                .defineCobertura("pepperoni+salame")
                .defineMolho("apimentado")
                .defineMassa("grossa")
                .montaPizza( );      // (2) monta e entrega produto pizza italiana
         italiana.exibePizza();     // (3) usa produto

         System.out.println("\n-- ---  ----\n");
```

```
        // (1) cria Pizza Marguerita
         new Pizza
                .PizzaBuilder("Marguerita", "grande")
                .defineMassa("fina")
                .defineCobertura("tomate+orégano")
                .defineMolho("tomate")
                .montaPizza( )// (2) monta e entrega produto pizza marguerita
                .exibePizza();// (3) usa produto
    }
}
-- --- ----
```

Abaixo apresento a saída com a ordem predeterminada em exibePizza(), que é válida para os 3 códigos para Cliente acima apresentados!

## **Saída**:

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

-- ---  ----

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

-- ---  ----

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