

1 Introdução

Neste material, vamos estudar sobre a forma como objetos podem ser construídos utilizando-se boas práticas. Em particular, vamos estudar sobre

- Simple Factory

2 Desenvolvimento

Workspace, novo projeto, VS Code, Typescript

Comece criando uma pasta para desempenhar o papel de Workspace. Ou seja, uma pasta que abriga subpastas, cada qual representando um projeto novo. No Windows, você pode usar algo assim:

```
C:\Users\seuUsuario\Documents\dev\
```

A seguir, abra um terminal e vincule-o ao diretório que acabou de criar com

```
cd C:\Users\seuUsuario\Documents\dev
```

Use

```
mkdir nome_desejado
```

para criar uma pasta que abrigará os arquivos desta nova solução computacional. Vincule o VS Code a ela com

```
code nome_desejado
```

No VS Code, clique Terminal >> New Terminal para abrir um terminal interno do VS Code, simplificando o trabalho. Use

```
npm init -y
```

para criar um projeto. Use

```
npm i typescript @types/node --save-dev
```

para instalar

- typescript: esse é o compilador Typescript. Ele é alimentado com código Typescript e produz código Javascript. Há quem chame isso de “**transpilação**”. O nome historicamente

usado para esse fenômeno, em que ocorre compilação de uma linguagem a outra e ambas possuem nível semelhante de abstração, é “**compilação fonte a fonte**”.

- @types/node: é um pacote que contém definições de tipo do Typescript para o ambiente NodeJS. Viabiliza o funcionamento do “intellisense” (auto complete “inteligente”), verificação de tipos em tempo de compilação, acesso a documentação mais detalhada etc.

Nota. O compilador Typescript e as funcionalidades providas pelo pacote @types/node são necessárias apenas em tempo de compilação. Se um dia o aplicativo for implantado, não será necessário compilar código ou utilizar intellisense em tempo de produção. Por isso usamos a opção --save-dev, instruindo o npm a registrar os pacotes como dependências em tempo de desenvolvimento apenas.

O compilador Typescript pode ter seu funcionamento configurado por meio de um arquivo chamado **tsconfig.json**. Ele pode ser criado utilizando-se o próprio compilador Typescript. Para tal, use

```
npx tsc --init
```

Abra o arquivo tsconfig.json recém criado e ajuste a propriedade **outDir** para que ela fique associada ao valor “./dist”. Ou seja, estamos instruindo o compilador Typescript a armazenar o código Javascript compilado na pasta dist. A propriedade provavelmente já existe no arquivo, basta remover o comentário e ajustar o valor.

```
...  
// "outFile": "./", /* Specify a file that bundles all outputs into  
one JavaScript file. If 'declaration' is true, also designates a file  
that bundles all .d.ts output. */  
"outDir": "./dist", /* Specify an output folder for all emitted  
files. */  
// "removeComments": true, /* Disable emitting comments. */  
// "noEmit": true,  
...
```

Abra o arquivo **package.json** e crie os scripts a seguir.

build: será usado para colocar o compilador Typescript (tsc) em execução, gerando o código Javascript que o NodeJS interpretará.

start: será usado para colocar o NodeJS em execução, alimentado pelo arquivo Javascript compilado.

```
{
  "name": "simplefactory_factorymethod",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "tsc",
    "start": "node dist/app.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@types/node": "^20.8.10",
    "typescript": "^5.2.2"
  }
}
```

Use

```
mkdir src
```

para criar a pasta que armazenará o código-fonte. Crie um arquivo chamado **app.ts** na pasta src. Veja seu conteúdo para o teste inicial.

```
let hello: string = "Hello, Typescript";
console.log(hello)
```

Use

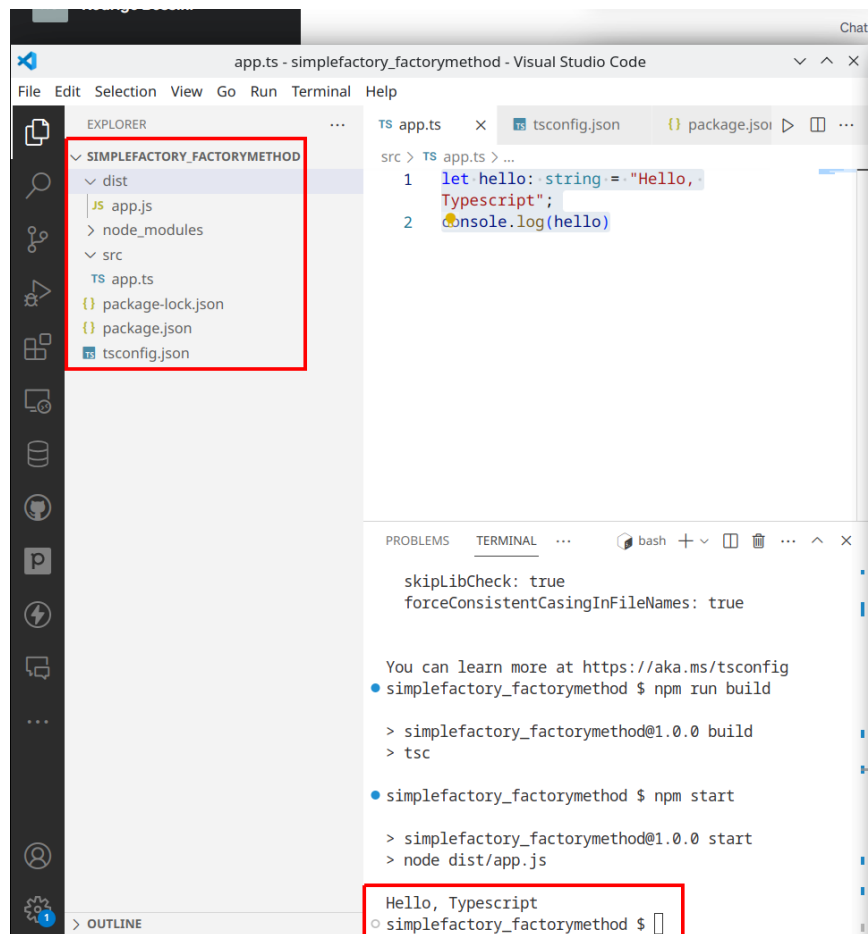
```
npm run build
```

para compilar. E

```
npm start
```

para executar.

Veja o resultado esperado, incluindo a estrutura de pastas e arquivos.



Para simplificar, você pode fazer uso do pacote **ts-node**. Ele permite que a compilação e execução ocorram de uma única vez. Não é apropriado para ambiente de produção. Mas pode ser muito útil para ambientes de desenvolvimento. Instale-o com

```
npm i ts-node --save-dev
```

Depois, ajuste o script start no arquivo **package.json**.

```
...
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "build": "tsc",
  "start": "ts-node src/app.ts"
},
...
```

Você pode escolher entre manter como estava antes ou utilizar o ts-node.

O padrão de projeto Factory (e suas variações) e a construção de objetos

Vamos estudar sobre o padrão de projeto Factory (e suas variações), um Design Pattern que permite encapsular a criação de objetos, o que pode dar origem a designs com menos acoplamento. No código a seguir, utilizamos o operador new (quem diria?) para construir um objeto. O cenário é interessante pois temos uma variável polimórfica e ela está fadada a fazer referência a um objeto de tipo escolhido explicitamente. Ele está fixo ali, é conhecido em tempo de compilação. Além disso, o uso do new está “espalhado” pelo código (ok, tem pouco código agora, vai fazer mais sentido depois).

```
class Pato{  
  
}  
  
class PatoReal extends Pato{  
  
}  
  
function teste(): void{  
    //indesejável!  
    let pato: Pato = new Pato();  
}
```

Embora seja óbvio, o uso do operador new pode ser indesejável. Sempre que utilizamos o operador new, certamente o que vem a seguir é o nome de uma classe concreta, ou seja, uma implementação e não uma interface.

Caso exista um conjunto de subclasses de patos que devem ser utilizados de acordo com a ocasião, o código pode ficar parecido com o seguinte.

```
class Pato{

}

class PatoReal extends Pato{

}

class PatoDeCaca extends Pato{

}

class PatoDeBorracha extends Pato{

}

function teste(): void{
    //indesejável!
    let pato: Pato = new Pato();
}

function qualPato(ocasio: string): Pato | null{
    let pato: Pato | null = null;
    if (ocasio === "piquenique")
        pato = new PatoReal();
    else if (ocasio === "caça")
        pato = new PatoDeCaca();
    else if (ocasio === "banheira")
        pato = new PatoDeBorracha();
    return pato;
}

let ocasioes: string[] = [
    'piquenique',
    'caça',
    'banheira'
]

//indexando ocasioes de 0 a 2
console.log(qualPato(ocasioes[Math.floor(Math.random() * 3)]))
```

O código exibido tem algumas características ruins. Para enxergar uma delas, basta **pensar no que seria necessário fazer para adicionar um novo tipo de pato**. Esse código que já está escrito e funcionando precisaria ser reaberto e editado. Este processo é indesejado pois viola o **princípio aberto-fechado**, que já estudamos.

Princípio aberto-fechado. Um componente de software deve ser aberto para extensão por meio da adição de novo código e fechado para a modificação. Assim, novas funcionalidades podem ser adicionadas sem que código já existente e que já está funcionando seja alterado, reduzindo a chance de adicionarmos bugs ao sistema.

O operador `new` é tão ruim assim?

Mas então, o que há de errado com o operador `new`? Não há nada de errado e nem pode haver, pois sem ele não há objetos. O problema é escrever seu código de modo que ele dependa de muitas classes concretas. **Se seu código depende somente de uma interface (que pode ser definida usando uma interface ou classe abstrata, dependendo da linguagem de programação), novas funcionalidades poderão ser adicionadas criando-se novas classes e referenciando instâncias delas por meio do polimorfismo.**

Agora se seu código depende de múltiplas classes concretas, quando novas classes forem adicionadas, ele terá de ser aberto para modificação (olha o princípio aberto-fechado sendo violado aí).

Como resolver esse problema então? Lembre-se do seguinte Princípio de Design.

Princípio de Design. Identifique aspectos que variam e separe-os daqueles que permanecem os mesmos.

O que desejamos responder neste material é o seguinte:

Como separar as partes da aplicação que constroem objetos do restante?

Implementando uma pizzeria

Para essa seção, crie um arquivo chamado **pizzaria.ts** na pasta **src** e ajuste o script start no arquivo **package.json** da seguinte forma.

```
...
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "build": "tsc",
  "start": "ts-node src/pizzaria.ts"
},
...
```

Crie uma classe que descreve o que é uma pizza, incluindo método para seu preparo. Além disso, escreva uma função que produz uma pizza.

```
class Pizza{
  preparar(): void{
    console.log('Preparando a pizza...');
  }

  assar(): void{
    console.log('Assando a pizza...');
  }

  cortar(): void{
    console.log('Cortando a pizza...');
  }

  empacotar(): void{
    console.log('Empacotando a pizza...');
  }
}

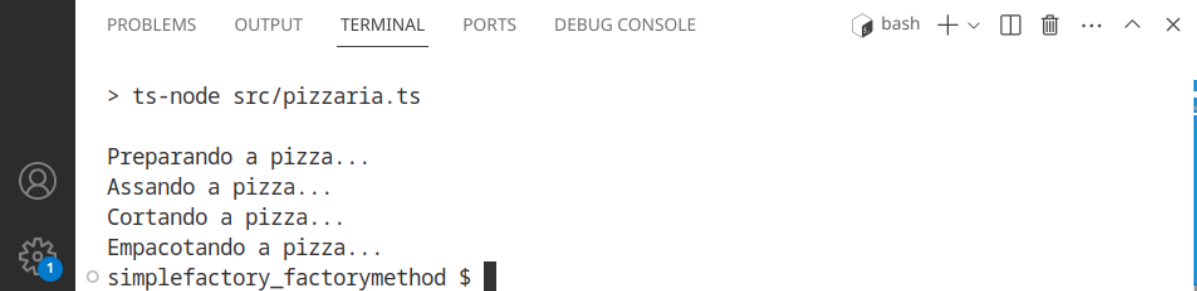
const pedirPizza = (): Pizza => {
  let pizza: Pizza = new Pizza();
  pizza.preparar();
  pizza.assar();
  pizza.cortar();
  pizza.empacotar();
  return pizza;
}

pedirPizza();
```


Execute com

`npm start`

E veja o resultado.



The screenshot shows a VS Code terminal window with the following content:

```
PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE
> ts-node src/pizzaria.ts
Preparando a pizza...
Assando a pizza...
Cortando a pizza...
Empacotando a pizza...
simplefactory_factorymethod $
```

The terminal window has a dark theme and a sidebar on the left with icons for a user profile and a settings gear. The terminal title bar shows 'bash' and standard window controls.

Mais sabores de pizza

Obviamente há muitos sabores de pizza. De alguma forma o método `pedirPizza` deve permitir que se escolha o sabor desejado. Uma nova versão dele pode ser parecida com o código a seguir.

```
class Pizza{
    ...
}

class PizzaDeQueijo extends Pizza{
    preparar(): void{
        console.log('Preparando pizza de queijo...');
    }

    assar(): void{
        console.log('Assando pizza de queijo...');
    }

    cortar(): void{
        console.log('Cortando pizza de queijo...');
    }

    empacotar(): void{
        console.log('Empacotando pizza de queijo...');
    }
}

class PizzaGrega extends Pizza{
    preparar(): void{
        console.log('Preparando pizza grega...');
    }

    assar(): void{
        console.log('Assando pizza grega...');
    }

    cortar(): void{
        console.log('Cortando pizza grega...');
    }

    empacotar(): void{
        console.log('Empacotando pizza grega...');
    }
}

class PizzaDePepperoni extends Pizza{
    preparar(): void{
        console.log('Preparando pizza de pepperoni...');
    }

    assar(): void{
        console.log('Assando pizza de pepperoni...');
    }

    cortar(): void{
        console.log('Cortando pizza de pepperoni...');
    }

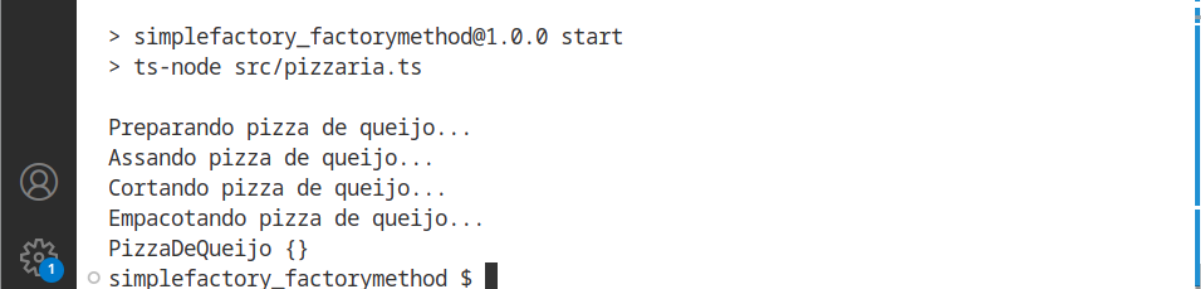
    empacotar(): void{
        console.log('Empacotando pizza de pepperoni...');
    }
}
```

```
const pedirPizza = (tipo: string): Pizza | null => {
  let pizza: Pizza | null = null;
  if (tipo === 'Queijo')
    pizza = new PizzaDeQueijo();
  else if (tipo === 'Grega')
    pizza = new PizzaGrega();
  else if (tipo === 'Pepperoni')
    pizza = new PizzaDePepperoni();
  //operador ?: chama o método somente se o objeto for diferente de
  null
  pizza?.preparar();
  pizza?.assar();
  pizza?.cortar();
  pizza?.empacotar();
  return pizza;
}
console.log(pedirPizza('Grega'));
```

Execute com

npm start

E veja o resultado esperado.



```
> simplefactory_factorymethod@1.0.0 start
> ts-node src/pizzaria.ts

Preparando pizza de queijo...
Assando pizza de queijo...
Cortando pizza de queijo...
Empacotando pizza de queijo...
PizzaDeQueijo {}
simplefactory_factorymethod $
```

A indesejada estrutura if/else e novos sabores de pizzas

Preste bastante atenção no código que escrevemos. A estrutura de seleção if/else decide qual tipo de Pizza criar. Uma vez decidido, o restante do código é exatamente o mesmo: basta preparar, assar, cortar e embalar a pizza. E cada subtipo de pizza sabe como fazer isso (ou seja, o código responsável por preparar uma pizza de queijo, por exemplo, está guardado dentro da classe que implementa esse tipo de pizza).

Mas agora você olha em volta e descobre que seus concorrentes estão vendendo novos tipos de pizza! Eles agora têm as opções "Clam" (molusco) e "Veggie" (Vegetariana). Obviamente você vai adicionar essas opções ao seu menu também! Além disso, você percebe que a pizza "greek" não está vendendo muito e resolve tirá-la do cardápio. Resumindo, aquilo que é **sempre verdade na vida de um desenvolvedor** acontece novamente: **a necessidade de mudança**. E, idealmente, você quer fazer isso da maneira mais rápida possível e com o menor nível de impacto. Observe.

```
class Pizza{
  ...
}

class PizzaDeQueijo extends Pizza{
  ...
}
class PizzaGrega extends Pizza{
  ...
}
class PizzaDePepperoni extends Pizza{
  ...
}

class PizzaDeMolusco extends Pizza{
  preparar(): void{
    console.log('Preparando pizza de molusco...');
  }

  assar(): void{
    console.log('Assando pizza de molusco...');
  }

  cortar(): void{
    console.log('Cortando pizza de molusco...');
  }

  empacotar(): void{
    console.log('Empacotando pizza de molusco...');
  }
}

class PizzaVegetariana extends Pizza{
  preparar(): void{
    console.log('Preparando pizza vegetariana...');
  }

  assar(): void{
    console.log('Assando pizza vegetariana...');
  }

  cortar(): void{
    console.log('Cortando pizza vegetariana...');
  }

  empacotar(): void{
    console.log('Empacotando pizza vegetariana...');
  }
}

const pedirPizza = (tipo: string): Pizza | null => {
  let pizza: Pizza | null = null;
  if (tipo === 'Queijo')
    pizza = new PizzaDeQueijo();
  //não vende, sai do cardápio
  //else if (tipo === 'Grega')
  //pizza = new PizzaGrega();
  else if (tipo === 'Pepperoni')
    pizza = new PizzaDePepperoni();
}
```

```

else if (tipo === 'Molusco')
    pizza = new PizzaDeMolusco();
else if (tipo === 'Vegetariana')
    pizza = new PizzaVegetariana();
//operador ?: chama o método somente se o objeto for diferente de null
pizza?.preparar();
pizza?.assar();
pizza?.cortar();
pizza?.empacotar();
return pizza;
}

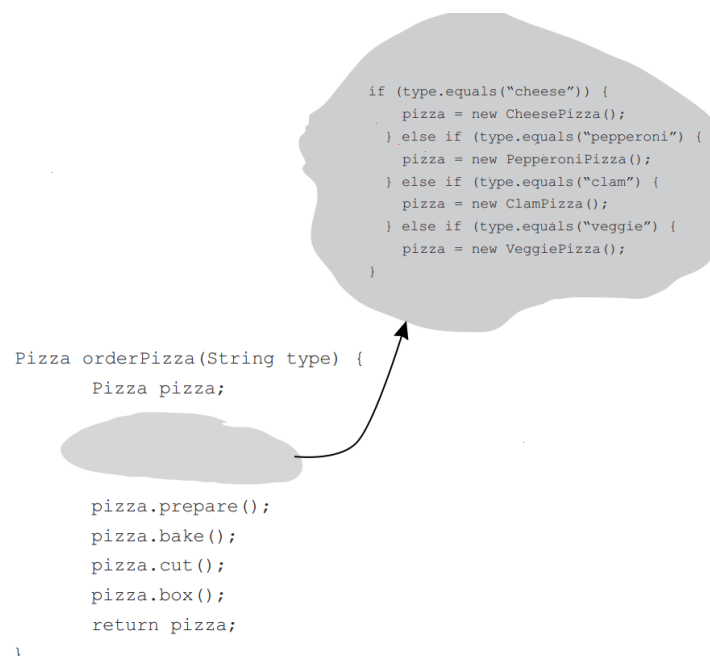
console.log(pedirPizza('Molusco'));

```

Separando o que varia daquilo que permanece o mesmo

Observando em detalhes o código, temos as seguintes características:

- a estrutura de seleção if/else não é fechada para modificação
 - a parte que varia é justamente aquela envolvida no if/else
 - a parte de preparar, assar, cortar e empacotar é o que teoricamente permanece a mesma. É muito pouco provável que esse código seja alterado. A única coisa que é alterada é o tipo de pizza sobre o qual os métodos operam, o qual não aparece explicitamente nesse trecho do método.
- Dadas essas características, o que desejamos fazer é o que a figura a seguir exhibe. Ou seja, separar o que varia daquilo que permanece o mesmo.



O padrão Simple Factory

Como ilustrado, desejamos isolar o código que varia, deixando-o em um único objeto cuja razão de ser será construir objetos do tipo pizza. Esse objeto recebe um nome. Ele é um **Factory**. Um objeto Factory lida com detalhes de criação de objetos. Agora o método pedirPizza não será mais responsável por diferenciar uma PizzaGrega de uma PizzaDeMolusco, por exemplo. Ele somente se preocupa em obter uma pizza (a partir do Factory) e então aplicar os métodos para preparo da pizza. Vamos então criar nosso Factory.

```
...
class SimplePizzaFactory{
  criarPizza(tipo: string): Pizza | null{
    let pizza: Pizza | null = null;
    if (tipo === 'Queijo')
      pizza = new PizzaDeQueijo();
    //não vende, sai do cardápio
    //else if (tipo === 'Grega')
    //pizza = new PizzaGrega();
    else if (tipo === 'Pepperoni')
      pizza = new PizzaDePepperoni();
    else if (tipo === 'Molusco')
      pizza = new PizzaDeMolusco();
    else if (tipo === 'Vegetariana')
      pizza = new PizzaVegetariana();
    return pizza;
  }
}

const pedirPizza = (tipo: string): Pizza | null => {
  //vamos completar ja ja..não está funcionando agora...
  //operador ?: chama o método somente se o objeto for diferente de null
  pizza?.preparar();
  pizza?.assar();
  pizza?.cortar();
  pizza?.empacotar();
  return pizza;
}
```

Qual a vantagem de utilizar a classe SimplePizzaFactory? Trata-se de uma única classe que precisará ser modificada caso novas pizzas sejam adicionadas ao menu. E ela pode ser utilizada por muitas outras classes sem que elas se preocupem com a criação de objetos. Neste exemplo, teremos

- uma classe PizzaStore
- ela tem uma SimplePizzaFactory como variável de instância
- o método pedirPizza também faz parte dela, agora delegando a criação da pizza à fábrica

Veja.

```

...
class SimplePizzaFactory{
    ...
}

class PizzaStore{
    constructor(
        private simplePizzaFactory: SimplePizzaFactory
    ){

    }

    pedirPizza = (tipo: string): Pizza | null => {
        let pizza = this.simplePizzaFactory.criarPizza(tipo);
        //operador ?: chama o método somente se o objeto for diferente de null
        pizza?.preparar();
        pizza?.assar();
        pizza?.cortar();
        pizza?.empacotar();
        return pizza;
    }
}

let pizzaStore: PizzaStore = new PizzaStore(new SimplePizzaFactory());
console.log(pizzaStore.pedirPizza('Molusco'));

```

Nota. Pense um pouco em composição: Já pensou em trocar a implementação do Factory? E construir tipos diferentes de pizza dependendo da implementação do Factory utilizado? Isso pode ser útil para uma rede de franquias, em que cada região (São Paulo, São Caetano etc) produz os mesmos tipos de pizzas, mas com suas “peculiaridades”. Em São Paulo, por exemplo, talvez a massa seja mais grossa. Em São Caetano, é possível que as pessoas gostem de pizzas com mais queijo.

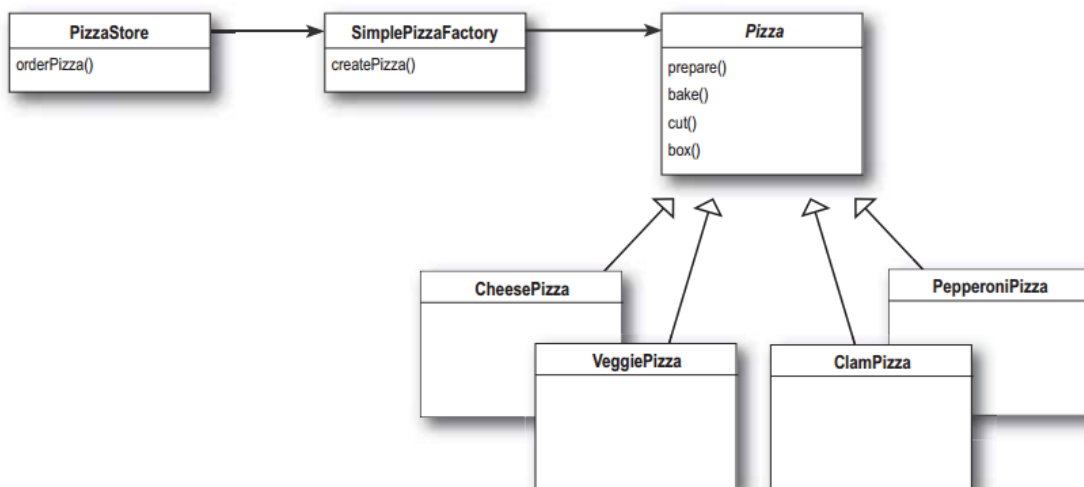
Menção honrosa

O Pattern que temos discutido até agora chama-se Simple Factory e ele, na verdade, não é um Design Pattern propriamente dito. Trata-se de uma boa prática de programação apenas. É bastante utilizado e merece uma menção honrosa.



Diagrama de classes do Simple Factory

De toda forma, vejamos como fica o diagrama de classes do Simple Factory.



Referências

FREEMAN, Eric; FREEMAN, Elisabeth; SIERRA, Kathy; BATES, Bert. **Head First Design Patterns**. 1. ed. São Francisco: O'Reilly Media, 2004.

GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. **Design Patterns: Elements of Reusable Object-Oriented Software**. 1. ed. Reading, Massachusetts: Addison-Wesley, 1995.

TypeScript: JavaScript that scales. Disponível em <<https://www.typescriptlang.org/>>. Acesso em: 03 de novembro de 2023.