Alunos: Adriano Costa, Jorge Nogueira, Priscilla Avelino, Ronaldo Sant'Ana

Inserir a pesquisa na AEP sobre padrões de projetos, com os seguintes itens de cada projeto:

- Definição

- Vantagens

- Desvantagens

- Utilização com exemplo

FACTORY METHOD

- Definição

De forma geral todos os padrões Factory (Simple Factory, Factory Method, Abstract Factory) encapsulam a criação de objetos. O padrão Factory Method fornece uma interface para a criação de objetos, deixando as subclasses escolherem qual classe instanciar.

Deve-se usar, principalmente, quando uma classe não pode antecipar ou conhecer a classe dos objetos que deve criar e quando uma classe quer suas subclasses para especificar os objetos que cria.

- Vantagens

* Não precisa conhecer a classe para retornar uma opção dela através de um método.

* Elimina a necessidade de montar um código em função a uma classe específica.

* Extrema facilidade que temos para incluir novos produtos. Não é necessário alterar NENHUM código, apenas precisamos criar o produto e a sua fábrica. Todo o código já escrito não será alterado.

- Desvantagens

* Especializar uma classe apenas para instanciar um objeto de uma subclasse de outra superclasse pode se revelar bastante improdutiva.

* Para cada novo produto precisamos sempre criar duas classes, um produto e uma fábrica.

- Utilização com exemplo

Suponha que você deve trabalhar em um projeto computacional com um conjunto de carros, cada um de uma determinada fábrica. Para exemplificar suponha os quatro seguintes modelos/fabricantes:

Palio – Fiat

Gol – Volkswagen

Celta – Chevrolet

Fiesta – Ford

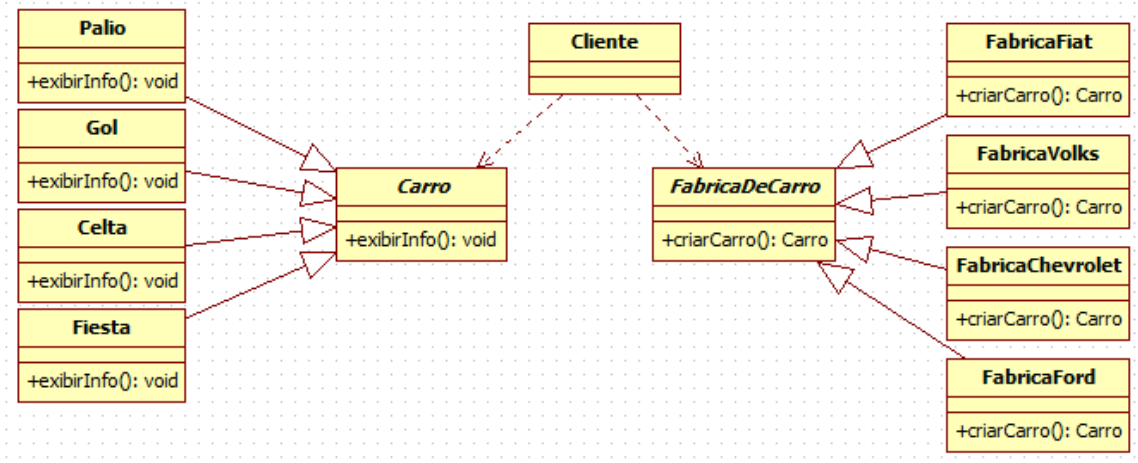
Será necessário manipular este conjunto de carros em diversas operações.

Uma primeira solução, mais simples, seria criar uma classe para representar cada carro, no entanto ficaria muito difícil prever as classes ou escrever vários métodos iguais para tratar cada um dos tipos de objetos.

Poderíamos então criar uma classe base para todos os carros e especializá-la em subclasses que representem cada tipo de carro, assim, uma vez definida uma interface comum poderíamos tratar todos os carros da mesma maneira. O problema surge quando vamos criar o objeto, pois, de alguma forma, precisamos identificar qual objeto queremos criar. Ou seja, precisaríamos criar uma enumeração para identificar cada um dos carros e, ao criar um carro, identificaríamos seguindo essa enumeração.

Um método fábrica cria Objetos concretos que só serão definidos em tempo de execução. No entanto, esta implementação traz um problema quanto a manutenibilidade do código, pois, como utilizamos um switch para definir qual objeto criar, a cada criação de um novo modelo de carro precisaríamos incrementar este switch e criar novas enumerações.

Então, ao invés de criar objetos diretamente em uma classe concreta, nós definimos uma interface de criação de objetos e cada subclasse fica responsável por criar seus objetos. Seria como se, ao invés de ter uma fábrica de carros, nós tivéssemos uma fábrica da Fiat, que cria o carro da Fiat, uma fábrica da Ford, que cria o carro da Ford e etc. As outras fábricas seguem a mesma ideia, cada uma define o método de criação de carros e cria o seu próprio carro.



BUILDER

Definição

É um padrão de projeto que permite separação da separação de um objeto, sendo assim esse mesmo objeto podendo ter diferentes representações.

Vantagens

Encapsulamento de código na construção do mesmo

Ter um controle durante o processo de construção do código

Desvantagens

Um problema com o padrão é que é preciso sempre chamar o método de construção para depois utilizar o produto em si.

Sistema de performance critica é uma das desvantagens do builder.

Utilização com exemplo

```
public class Pizza {

    private int tamanho;
    private boolean queijo;
    private boolean tomate;
    private boolean bacon;

    public static class Builder {

        // requerido
        private final int tamanho;

        // opcional
        private boolean queijo = false;
        private boolean tomate = false;
        private boolean bacon = false;

        public Builder(int tamanho) {
            this.tamanho = tamanho;
```

```
        }

        public Builder queijo() {
            queijo = true;
            return this;
        }

        public Builder tomate() {
            tomate = true;
            return this;
        }

        public Builder bacon() {
            bacon = true;
            return this;
        }

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

    }

    private Pizza(Builder builder) {
        tamanho = builder.tamanho;
        queijo = builder.queijo;
        tomate = builder.tomate;
        bacon = builder.bacon;
    }

}
```

PROTOTYPE

- Definição

A intenção do padrão é especificar tipos de objetos a serem criados usando uma instância protótipo e criar novos objetos pela cópia desse protótipo.

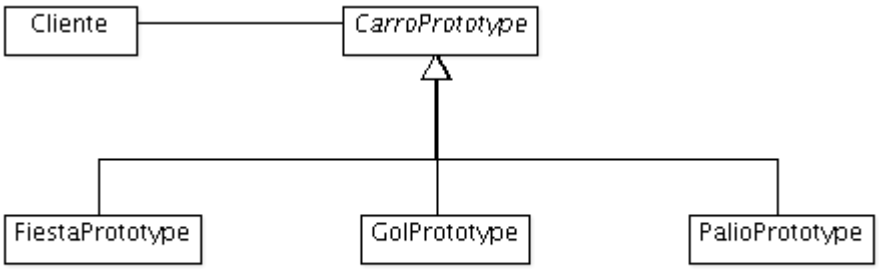
Vantagens

O padrão Prototype leva grande vantagem quando o processo de criação de seus produtos é muito caro, ou mais caro que uma clonagem, pois oferece algumas vantagens como esconder os produtos dos clientes, reduzir o acoplamento e oferecer maior flexibilidade para alterações nas classes produtos.

- Desvantagens

A única desvantagem encontrada foi a utilização do método clone. Ao utilizar este padrão os objetos clonados podem apresentar problemas de inclusão de classes não existentes.

- Utilização com exemplo



A estrutura do padrão inicia então com definição dos objetos protótipos. Para garantir a flexibilidade do sistema, cria-se a classe base de todos os protótipos:

```
public abstract class CarroPrototype {
   protected double valorCompra;

   public abstract String exibirInfo();

   public abstract CarroPrototype clonar();

   public double getValorCompra() {
      return valorCompra;
   }

   public void setValorCompra(double valorCompra) {
      this.valorCompra = valorCompra;
   }
}
```

```
public class FiestaPrototype extends CarroPrototype {

   protected FiestaPrototype(FiestaPrototype fiestaPrototype) {
      this.valorCompra = fiestaPrototype.getValorCompra();
   }

   public FiestaPrototype() {
      valorCompra = 0.0;
   }

   @Override
   public String exibirInfo() {
      return "Modelo: Fiesta\nMontadora: Ford\n" + "Valor: R$"
            + getValorCompra();
   }

   @Override
   public CarroPrototype clonar() {
      return new FiestaPrototype(this);
   }

}
```

```
public static void main(String[] args) {
   PalioPrototype prototipoPalio = new PalioPrototype();

   CarroPrototype palioNovo = prototipoPalio.clonar();
   palioNovo.setValorCompra(27900.0);
   CarroPrototype palioUsado = prototipoPalio.clonar();
   palioUsado.setValorCompra(21000.0);

   System.out.println(palioNovo.exibirInfo());
   System.out.println(palioUsado.exibirInfo());
}
```

ABSTRACT FACTORY

- Definição

Considere uma aplicação onde eu possa ter múltiplas interfaces gráficas, a princípio apenas duas, mas que eu possa adicionar quantas eu quiser. Devemos pensar em fazer isso da forma mais escalável possível, é aí que entra um padrão de Projeto: **Abstract Factory.**

- Vantagens e desvantagens

Ele isola as classes concretas. O padrão Abstract Factory ajuda a controlar as classes de objetos criadas por uma aplicação. Uma vez que a fábrica encapsula a responsabilidade e o processo de criar objetos, isola os clientes das classes de implementação. Os clientes manipulam as instâncias através das suas interfaces abstratas. Nomes de classes ficam isolados na implementação da fábrica concreta.

Ele torna fácil a troca de famílias de produtos. A classe de uma fábrica concreta aparece apenas uma vez numa aplicação, isto é, quando é instanciada. Isso torna fácil mudar a fábrica concreta que uma aplicação usa. Ela pode usar diferentes configurações de objetos simplesmente trocando a fábrica concreta. Uma vez que a fábrica abstrata cria uma família completa de objetos, toda família de objetos muda de uma só vez. No nosso exemplo de construção de carros, podemos mudar de componentes do Polo para componentes do Stilo simplesmente trocando as correspondentes fábricas e recriando o carro.

Ela promove a harmonia entre produtos. Quando objetos numa família são projetados para trabalharem juntos, é importante que uma aplicação use objetos de somente uma família de cada vez. Abstract Factory torna fácil assegurar isso.

É difícil suportar novos tipos de produtos. Estender fábricas abstratas para produzir novos tipos de produtos não é fácil. Isso se deve ao fato de que a interface de Abstract Factory fixa o conjunto de produtos que podem ser criados. Suportar novos tipos de produtos exige estender a interface da fábrica, o que envolve mudar a classe AbstractFactory e todas as suas subclasses.

- Utilização com exemplo

```
public class Palio implements CarroPopular {

   @Override
   public void exibirInfoPopular() {
      System.out.println("Modelo: Palio\nFábrica: Fiat\nCategoria:Popular");
   }

}
```

```
public class Siena implements CarroSedan {

   @Override
   public void exibirInfoSedan() {
      System.out.println("Modelo: Siena\nFábrica: Fiat\nCategoria:Sedan");
   }

}
```

```
public static void main (String [] args) {
   FabricaDeCarro fabrica = new FabricaFiat();
   CarroSedan sedan = fabrica.criarCarroSedan();
```

```
    CarroPopular popular = fabrica.criarCarroPopular();
    sedan.exibirInfoSedan();
    System.out.println();
    popular.exibirInfoPopular();
    System.out.println();

    fabrica = new FabricaFord();
    sedan = fabrica.criarCarroSedan();
    popular = fabrica.criarCarroPopular();
    sedan.exibirInfoSedan();
    System.out.println();
    popular.exibirInfoPopular();
}
```