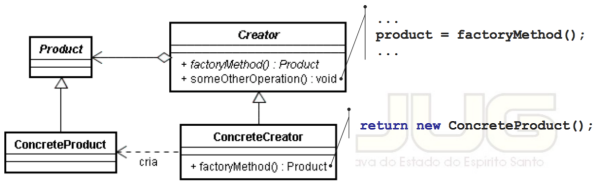
Padrões de Criação

Exemplos Práticos

2015/2016

João Alegria | 68861



Factory Method

A classe **AnimalFactory** irá retornar um objeto animal com base nos dados de entrada.
- Animal é uma classe abstrata.
- A fábrica irá retornar uma subclasse instanciada de **Animal**.

Animal.java
```
public abstract class Animal{
   public abstract String makeSound();
}
```

Animal tem um único método abstrato, makeSound( ).

Dog.java
```
public class Dog extends Animal{
   @Override
   public String makeSound(){
     return "Woof";
   }
```

A classe **Dog** é uma subclasse de Animal. Implementa makeSoud( ) e retorna "Woof".

Cat.java
```
public class Cat extends Animal{
@Override
public String makeSound(){
       return "Miau";
}
```

A classe **Cat** é uma subclasse de Animal. Implementa makeSound( ) e retorna "Miau".

AnimalFactory.java
```
public class AnimalFactory{
   public Animal getAnimal(String type){
     if("canino".equals(type)){
       return new Dog();
     }
     else return new Cat ();
   }
}
```

Na implementação do método fábrica, existe um método getAnimal. Este método utiliza uma sequência de caracteres como parâmetro. Caso a sequência seja "canino" retorna um objeto do tipo cão, caso contrário, retorna um objeto gato.

Demo.java
```
public class Demo{
   public static void main(String []args){
   AnimalFactory animalfactory = new AnimalFactory();
   Animal a1 = animalfactory.getAnimal("felino");
   System.out.println("a1 sound:" + a1.makeSound());

   Animal a2 = animalfactory.getAnimal("canino");
   System.out.println("a2 sound:" + a2.makeSound());
   }
}
```

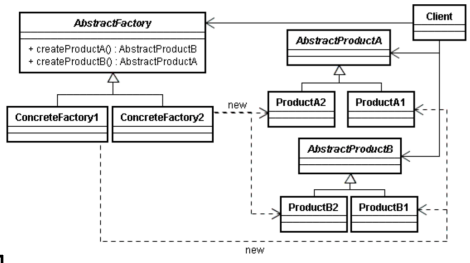
→
```
a1 sound: Miau
a2 sound: Woof
```

A classe **Demo** demonstra o uso da fábrica.
- Cria uma fábrica AnimalFactory. A fábrica cria dois objetos Animal.
- O primeiro objeto é um gato e o segundo é um cão.
- A saída de cada objeto é o método makeSound( ).

Uma fábrica pode ser também utilizada com o padrão Singleton.
Para isso, substitui-se: AnimalFactory animalfactory = new AnimalFactory( );
por AnimalFactory animalfactory = AnimalFactory.getAnimalFactoryInstance( );

Nota:
Neste exemplo, AnimalFactory.getAnimalFactoryInstance() foi implementado para retornar um objeto estático de AnimalFactory. Isto resulta numa fábrica única que é instanciada e usada invés de criar uma fábrica nova sempre que necessário.

Abstract Factory



Uma fábrica abstrata é uma fábrica que retorna fábricas.

AbstractFactory.java

```
public class AbstractFactory {
   public EspeciesFactory getEspeciesFactory(String type){
     if("mamifero".equals(type)){
       return new MamiferoFactory();
     }
     else return new ReptilFactory();
   }
}
```

AbstractFactory retorna MamiferoFactory ou ReptilFactory, através do tipo de retorno de EspeciesFactory - MamiferoFactory e ReptilFactory são subclasses de EspeciesFactory.

EspeciesFactory.java

```
import animals.Animal;
public abstract class EspeciesFactory{
   public abstract Animal getAnimal(String type);
}
```

EspeciesFactory é uma classe abstrata com o método getAnimal( ). Este método retorna um objeto Animal.

O polimorfismo do AbstractFactory é alcançado porque o seu método getEspeciesFactory( ) retorna uma EspecieFactory, independentemente da classe subjacente. Este polimorfismo também podia ser alcançado através de uma interface em vez de uma classe abstrata.

MamiferoFactory.java

```
import animals.Animal;
import animals.Gato;
import animals.Cao;
public class MamiferoFactory extends EspeciesFactory{
   @Override
   public Animal getAnimal(String type){
     if("cao".equals(type)){
       return new Cao();
     }
     else return new Gato();
   }
}
```

MamiferoFactory implementa getAnimal( ). Retorna um Animal, que é um cão ou gato.

ReptilFactory.java

```
import animals.Animal;
import animals.Cobra;
import animals.Dinossauro;
public class ReptilFactory extends EspeciesFactory{
   @Override
   public Animal getAnimal(String type){
     if("cobra".equals(type)){
       return new Cobra();
     }
     else return new Dinossauro();
   }
}
```

ReptilFactory implementa getAnimal( ). Retorna um Animal, que é uma cobra ou dinossauro.

Animal.java

```
public abstract class Animal{
   public abstract String makeSound();
}
```

Animal é uma classe abstrata com o método abstracto makeSound ( ). Subclasses de Animal implementam o método makeSound ( ).

Gato.java

```
public class Gato extends Animal{
   @Override
   public String makeSound(){
     return "Miau";
   }
}
```

Cobra.java

```
public class Cobra extends Animal{
   @Override
   public String makeSound(){
     return "Hiss";
   }
}
```

Cao.java

```
public class Cao extends Animal{
   @Override
   public String makeSound(){
     return "Woof";
   }
}
```

Dinossauro.java

```
public class Dinossauro extends Animal{
   @Override
   public String makeSound(){
     return "Roar";
   }
}
```

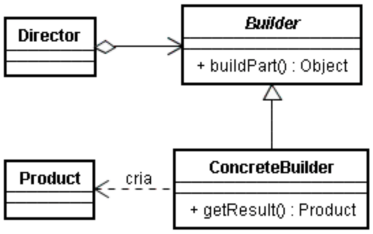
A classe **Demo** contém o método Main.
Cria um objeto AbstractFactory e a partir deste, obtemos o EspeciesFactory (um ReptilFactory) e dois objetos Animal (Dinossauro e Cobra). Após isto, obtemos outro EspeciesFactory (MamiferoFactory) e ontem mais dois objetos Animal (Cão e Gato).

Demo.java

```
import animals.Animal
public class Demo {
   public static void main(String[] args) {
   AbstractFactory abstractFactory = new AbstractFactory();
   EspeciesFactory especiesFactory1 = abstractFactory.getEspeciesFactory(“reptil");
   Animal a1 = especiesFactory1.getAnimal(“dinossauro");
     System.out.println("a1 sound: " + a1.makeSound());
   Animal a2 = especiesFactory1.getAnimal("cobra");
     System.out.println("a2 sound: " + a2.makeSound());
   EspeciesFactory especiesFactory2 = abstractFactory.getEspeciesFactory("mamifero");
   Animal a3 = especiesFactory2.getAnimal("cao");
     System.out.println("a3 sound: " + a3.makeSound());
   Animal a4 = especiesFactory2.getAnimal("gato");
     System.out.println("a4 sound: " + a4.makeSound());
   }
}
```

```
a1 sound: Roar
a2 sound: Hiss
a3 sound: Woof
a4 sound: Miau
```

Nota:
Observe o uso do polimorfismo. Obtemos diferentes fábricas através da superclasse EspeciesFactory. Também obtemos diferentes animais através da superclasse Animal.



Builder

Construção de diferentes tipos de refeição de um restaurante.
A classe **`Meal`** será o nosso *Produto*, que representa os itens numa refeição - prato, bebida, entrada.

Meal.java
```
public class Meal{
   private String prato, bebida, entrada;
   public String getBebida() {return bebida;}
   public void setBebida(String bebida) {this.bebida = bebida;}
   public String getPrato() {return prato;}
   public void setPrato(String prato) {this.prato = prato;}
   public String getEntrada() {return entrada;}
   public void setEntrada(String entrada) {this.entrada = entrada;}

   public String toString() {return "bebida:" + bebida + ",prato principal:" + prato +
                              +",entrada: " + entrada;}
}
```

MealBuilder.java
```
public interface MealBuilder{
   public void buildBebida();
   public void buildPrato();
   public void buildEntrada();
   public Meal getMeal();
```

Interface do Builder - **`MealBuilder`**. Dispõe de métodos para construir uma refeição e um método para retornar a refeição.

ItalianMealBuilder.java
```
public class ItalianMealBuilder implements MealBuilder{
   private Meal meal;
   public ItalianMealBuilder() {meal = new Meal();}
   @Override
     public void buildBebida() {meal.setBebida("vinho tinto");}
   @Override
     public void buildPrato() {meal.setPrato("pizza");}
   @Override
     public void buildEntrada() {meal.setEntrada("pão");}
   @Override
     public Meal getMeal() {return meal;}
}
```

O primeiro *ConcreteBuilder* é o **`ItalianMealBuilder`**. Ele cria a refeição e os seus métodos são implementados para construir as várias partes da refeição. A refeição é retornada através do getMeal( ).

PortugueseMealBuilder.java
```
public class PortugueseMealBuilder implements MealBuilder{
   private Meal meal;
   public PortugueseMealBuilder() {meal = new Meal();}
   @Override
     public void buildBebida() {meal.setBebida("vinho do porto");}
   @Override
     public void buildPrato() {meal.setPrato("cozido");}
   @Override
     public void buildEntrada() {meal.setEntrada("azeitonas");}
   @Override
     public Meal getMeal() {return meal;}
}
```

O segundo *ConcreteBuilder* é o **PortugueseMealBuilder**. Ele cria a refeição e os seus métodos são implementados para construir as várias partes da refeição. A refeição é retornada através do getMeal( ).

MealDiretor.java

```
public class MealDiretor{
   private MealBuilder mealbusilder = null;
   public MealDiretor(MealBuilder mealbuilder){
     this.mealbuilder = mealbuilder
   }
   public void constructMeal(){
     mealBuilder.buildBebida();
     mealBuilder.buildPrato();
     mealBuilder.buildEntrada();
   }

   public Meal getMeal(){
     return mealBuilder.getMeal();
   }
```

função agregadora

retorno da comida

A classe diretora é o **MealDiretor**. É preciso um MealBuilder como parâmetro no seu construtor. Assim, uma interface refeição diferente será construída pelo MealDiretor dependendo do *ConcretBuilder* passado no construtor. A montagem da refeição (*Produto*) ocorre no método constructMeal( ) do MealDiretor. Este método indica os ingredientes que constituirá a refeição.

Demo.java

```
public class Demo {
   public static void main(String[] args) {
     MealBuilder mealBuilder = new ItalianMealBuilder();
     MealDiretor mealDiretor = new MealDiretor(mealBuilder);
     mealDiretor.constructMeal();
     Meal meal = mealDiretor.getMeal();
     System.out.println("meal is = " + meal);

     mealBuilder = new PortugueseMealBuilder();
     mealDiretor = new MealDiretor(mealBuilder);
     mealDiretor.constructMeal();
     meal = mealDiretor.getMeal();
     System.out.println("meal is = " + meal);
```

```
meal is = bebida: vinho tinto, prato: pizza,
                    entrada: pão

meal is = bebida: vinho do porto, prato: cozido,
                  entrada: azeitonas
```

A classe **Demo** permite demonstrar o padrão Builder.
Primeiramente, o diretor constrói uma refeição Italiana. Um ItalianMealBuilder é passado para o construtor do MealDiretor. A refeição é construída através de mealDiretor.constructMeal( ).
A refeição é obtida a partir de mealDiretor via mealDiretor.getMeal( ).
A refeição italiana é exibida. Após isto, realizamos o mesmo para a refeição portuguesa.