TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

Viện Công nghệ thông tin và Truyền thông

ABSTRACT FACTORY PATTERN

Môn: Phát triển phần mềm theo chuẩn kỹ năng ITSS

Số nhóm: 06
Danh sách sinh viên

Trịnh Thiên Long	:	20142710	Nhóm trưởng
Nguyễn Thăng Long	:	20142685	Thành viên
Nguyễn Phương Nam	:	20143061	Thành viên

Hà Nội, ngày 02 tháng 11 năm 2017

Mục lục

1. **Tổng quan**

Abstract Factory Design Pattern là gì

Abstract Factory là sự mở rộng của tính chất đa hình trong lập trình hướng đối tượng. Mục tiêu hướng đến là có thể tạo ra các đối tượng mà chưa biết trước chính xác kiểu dữ liệu của chúng.

Hãy tưởng tượng, Abstract factory là một nhà máy lớn chứa nhiều nhà máy nhỏ(**Factory Pattern**), trong các nhà máy đó có những xưởng sản xuất, các xưởng đó tạo ra những sản phẩm khác nhau.

Khi nào chúng ta nên sử dụng Abstract Factory Pattern?

- Tạo ra đối tượng mà không cần biết chính xác kiểu dữ liệu.
  Ví dụ, Chúng ta đọc dữ liệu từ một file txt để lấy thông tin tạo đối tượng. Trong file txt đó có chứa các từ : circle, square, rectangle. Trong mã nguồn ta viết rằng, nếu dữ liệu đọc được là chữ "circle" thì tạo ra hình tròn, nếu là "square" thì tạo ra hình vuông, nếu là "rectangle" thì tạo ra hình chữ nhật . Vậy rõ ràng, đối tượng của ta chỉ được tạo ra trong quá trình run time, nghĩa là trong lúc ta viết mã nguồn chương trình, ta chưa xác định được đối tượng mình cần tạo. Abstract Factory có thể giúp chúng ta giải quyết vấn đề này.

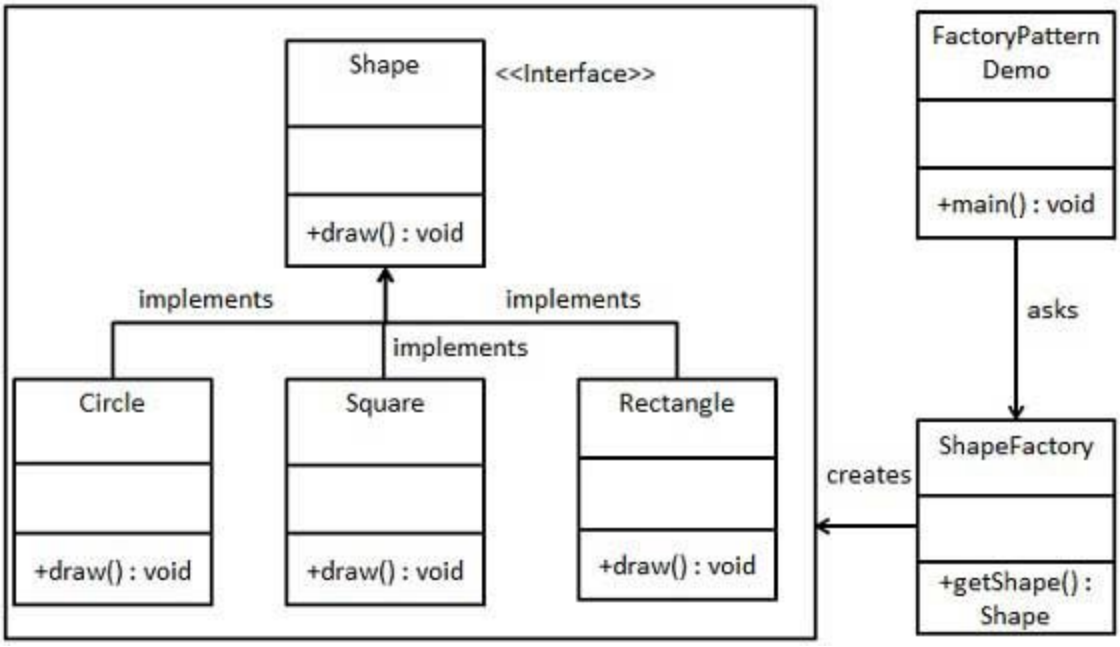
- Giúp mã nguồn của chúng ta trở nên dễ dàng bảo trì nếu có sự thay đổi.
  Abstract Factory được hình dung như là một nhà máy chứa nhiều nhà máy con. Mỗi nhà máy con là một factory. Ta có thể sử dụng Abstract Factory để tạo ra tất cả các đối tượng trong chương trình của ta. Khi ta sửa đổi hoặc thêm mới một đối tượng, các đối tượng khác sẽ không bị ảnh hưởng.

Vậy trước khi đi vào tìm hiểu về Abstract Factory Pattern, chúng ta cùng tìm hiểu về Factory Pattern trước.

2. **Factory Pattern**

Pattern này được sử dụng khi hệ thống của bạn có nhiều đối tượng với thuộc tính, hành vi tương tự nhau. Pattern Factory giống như một nhà máy sản sinh các đối tượng tương tự nhau này cho bạn.

Ví dụ: Chúng ta sẽ minh họa cho pattern này qua ví dụ sau: Chúng ta sẽ tạo ra interface Shape và các lớp cụ thể implements interface này. Một lớp ShapeFactory để tạo ra các object tương ứng với thông tin được truyền vào(CIRCLE / RETANGLE / SQUARE).



Step 1: Tạo interface Shape.java

```
public interface Shape {
   void draw();
}
```

Step 2: Tạo các lớp thực thi interface Shape

Rectangle.java

```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```

Square.java

```
public class Square implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

Circle.java

```
public class Circle implements Shape {
```

```
  @Override
  public void draw() {
    System.out.println("Inside Circle::draw() method.");
  }
}
```

Step 3: Viết lớp ShapeFactory để tạo ra các đối tượng ở Step 2 dựa vào tham số nhận được.

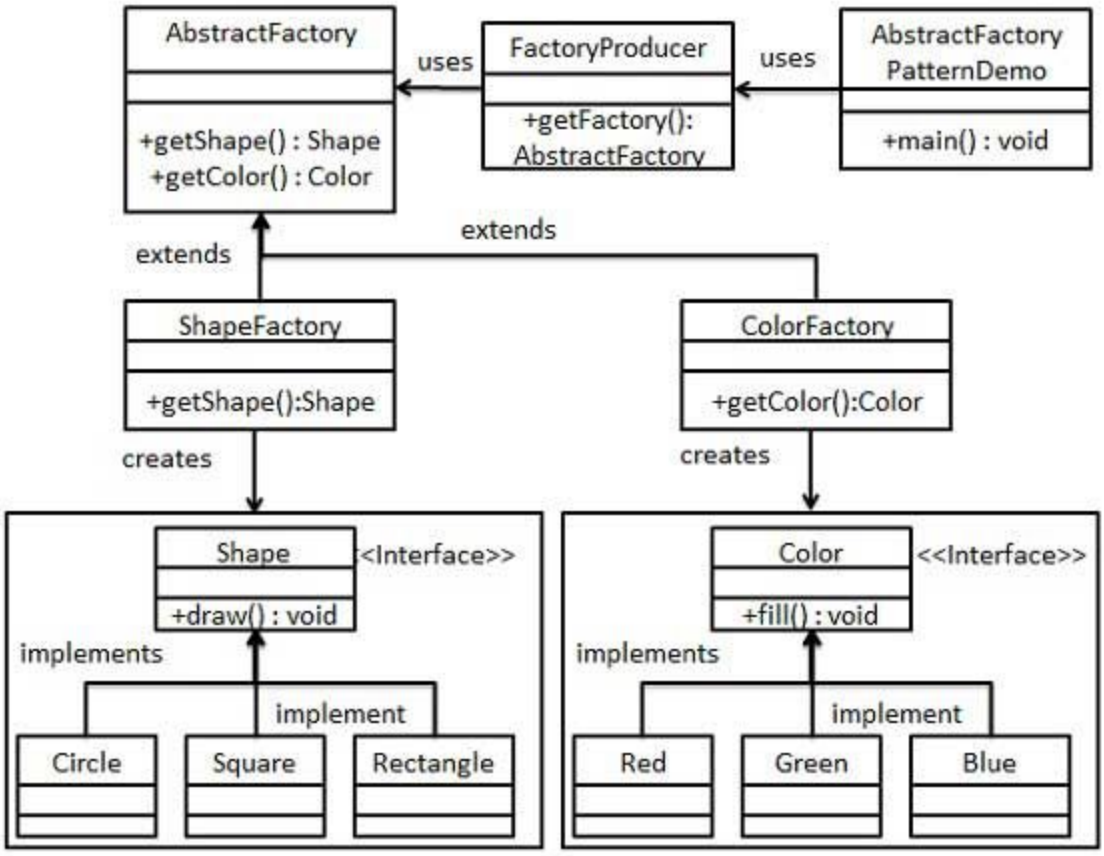
ShapeFactory.java

```
public class ShapeFactory {

  //use getShape method to get object of type shape
  public Shape getShape(String shapeType){
    if(shapeType == null){
      return null;
    }
    if(shapeType.equalsIgnoreCase("CIRCLE")){
      return new Circle();

    } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
      return new Rectangle();

    } else if(shapeType.equalsIgnoreCase("SQUARE")){
      return new Square();
    }

    return null;
  }
}
```

Step 4:

3. **Abstract Factory Design Pattern**

Design pattern này cung cấp một cách hỗ trợ việc quản lý và tạo ra các đối tượng cùng nhóm. Như tên của pattern này thì nó giống là một nhà máy sản sinh ra các đối tượng.



Step 1: Tạo interface Shape.java

```
public interface Shape {
   void draw();
}
```

Step 2: Tạo các lớp thực thi interface Shape

```
Rectangle.java
public class Rectangle implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}

Square.java
public class Square implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
```

```
}

Circle.java
public class Circle implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}
```

Step 3: Tạo interface Color.java

```
Color.java
public interface Color {
   void fill();
}
```

Step 4: Tạo các lớp thực thi interface Color

```
Red.java
public class Red implements Color {
   @Override
   public void fill() {
      System.out.println("Inside Red::fill() method.");
   }
}

Green.java
public class Green implements Color {
   @Override
   public void fill() {
      System.out.println("Inside Green::fill() method.");
   }
}

Blue.java
public class Blue implements Color {
   @Override
   public void fill() {
      System.out.println("Inside Blue::fill() method.");
   }
}
```

Step 5: Xây dựng lớp abstract khởi tạo các đối tượng Color và Shape

```
public abstract class AbstractFactory {
   abstract Color getColor(String color);
   abstract Shape getShape(String shape) ;
}
```

Step 6: Triển khai lớp abstract đã xây dựng ở bước 5

```
ShapeFactory.java
public class ShapeFactory extends AbstractFactory {
   @Override
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }

      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();
      }else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();
      }else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
      return null;
   }

   @Override
   Color getColor(String color) {
      return null;
   }
}

ColorFactory.java
public class ColorFactory extends AbstractFactory {
   @Override
   public Shape getShape(String shapeType){
      return null;
   }

   @Override
   Color getColor(String color) {
      if(color == null){
         return null;
      }
```

```
      if(color.equalsIgnoreCase("RED")){
         return new Red();
      }else if(color.equalsIgnoreCase("GREEN")){
         return new Green();
      }else if(color.equalsIgnoreCase("BLUE")){
         return new Blue();
      }

      return null;
   }
}
```

Step 7: Tạo một Factory generator class để tạo ra Factory dựa trên tham số truyền vào

FactoryProducer.java

```
public class FactoryProducer {
   public static AbstractFactory getFactory(String choice){
      if(choice.equalsIgnoreCase("SHAPE")){
         return new ShapeFactory();
      }else if(choice.equalsIgnoreCase("COLOR")){
         return new ColorFactory();
      }

      return null;
   }
}
```

Step 8: Sử dụng Factory generator class để lấy ra factory của các lớp thực thi dựa trên tham số truyền vào

AbstractFactoryPatternDemo.java

```
public class AbstractFactoryPatternDemo {
   public static void main(String[] args) {

      //get shape factory
      AbstractFactory shapeFactory =
FactoryProducer.getFactory("SHAPE");

      //get an object of Shape Circle
      Shape shape1 = shapeFactory.getShape("CIRCLE");

      //call draw method of Shape Circle
      shape1.draw();
```

```
      //get an object of Shape Rectangle
      Shape shape2 = shapeFactory.getShape("RECTANGLE");

      //call draw method of Shape Rectangle
      shape2.draw();

      //get an object of Shape Square
      Shape shape3 = shapeFactory.getShape("SQUARE");

      //call draw method of Shape Square
      shape3.draw();

      //get color factory
      AbstractFactory colorFactory =
FactoryProducer.getFactory("COLOR");

      //get an object of Color Red
      Color color1 = colorFactory.getColor("RED");

      //call fill method of Red
      color1.fill();

      //get an object of Color Green
      Color color2 = colorFactory.getColor("Green");

      //call fill method of Green
      color2.fill();

      //get an object of Color Blue
      Color color3 = colorFactory.getColor("BLUE");

      //call fill method of Color Blue
      color3.fill();
   }
}
```