ABSTRACT FACTORY PATTERN

M3105 Conception et programmation objet avancées

Sofiane FARKHANI – Elise JOFFRE – Alexandre POMMARAT – Yiyao ZHAI

Plan :

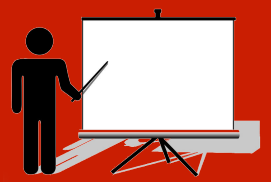
- Présentation générale
- Implémentation
- Quand l'utiliser ?
- Application avec un exemple concret étape par étape
- Sources

Introduction :

Les design pattern c’est quoi ?

- Solution de design et d’implémentation

- ✧ Rendre propre
- ✧ Optimiser
- ✧ Rendre robuste
- ✧ Rendre maintenable et évolutif

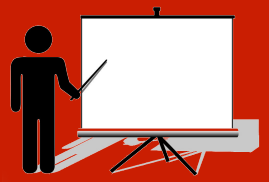


Présentation générale :

Il existe 3 principaux types de « design pattern »:

- Les patterns de création
- Les patterns de structuration
- Les patterns comportementaux

« L'abstract Factory » est un pattern de création (Creational Pattern)

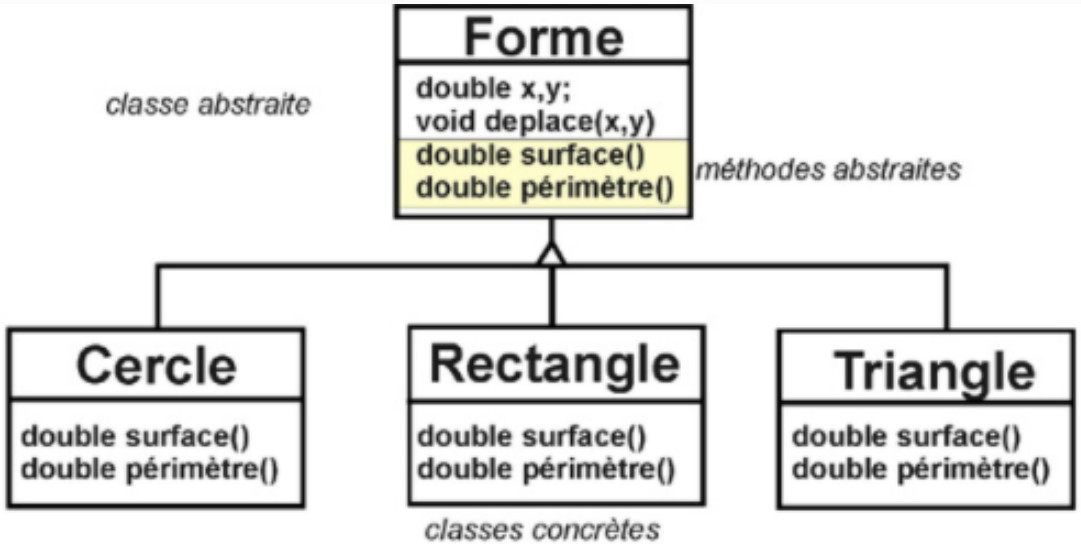


Présentation générale :

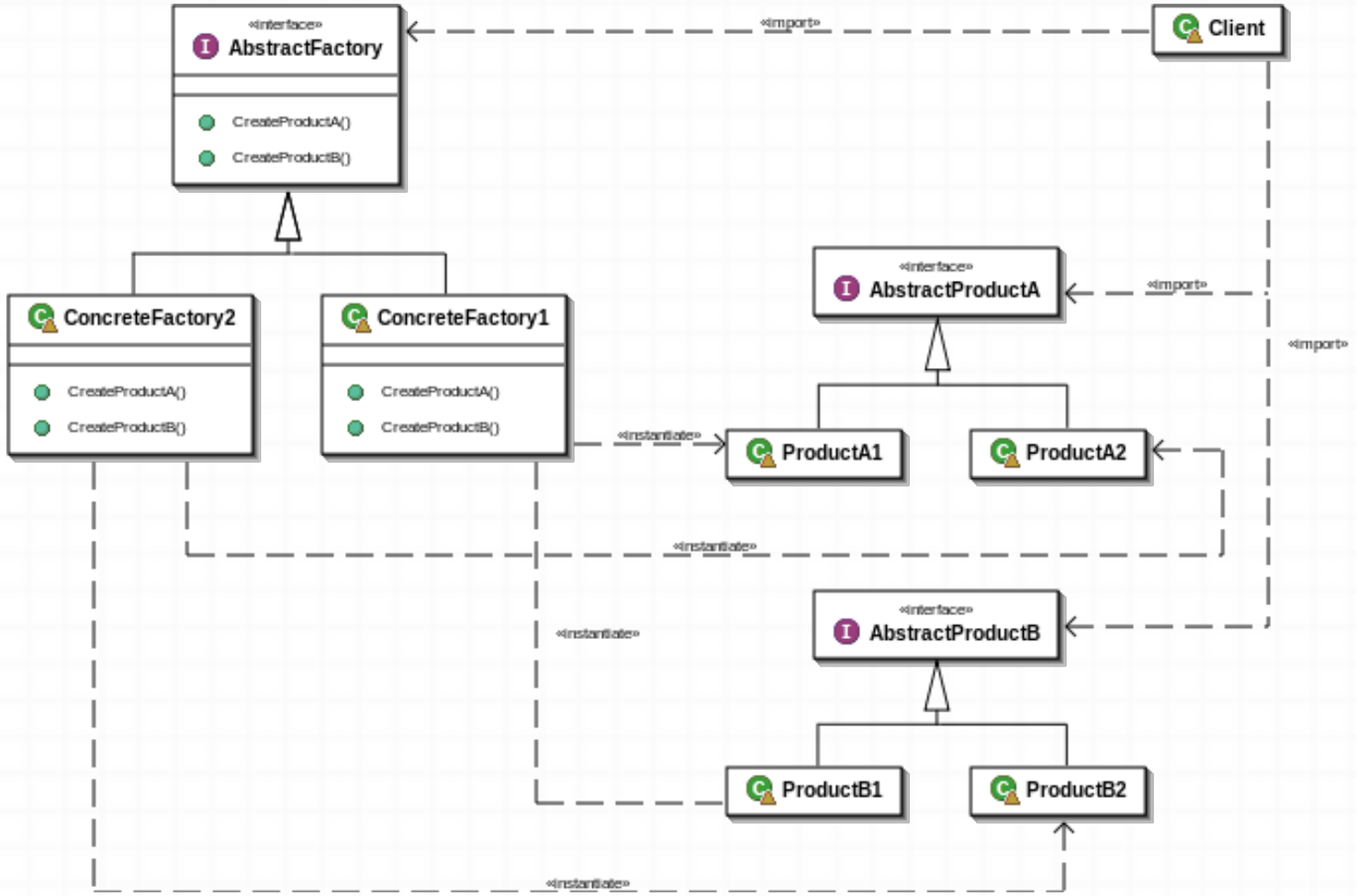
Factory

```
/** Interface de description d'un point */
public interface Point {
 /** Retourne l'abscisse du point */
 public int getX();
 /** Retourne l'ordonnée du point */
 public int getY();
}
```

Abstract



L'implémentation :



Principe OCP :

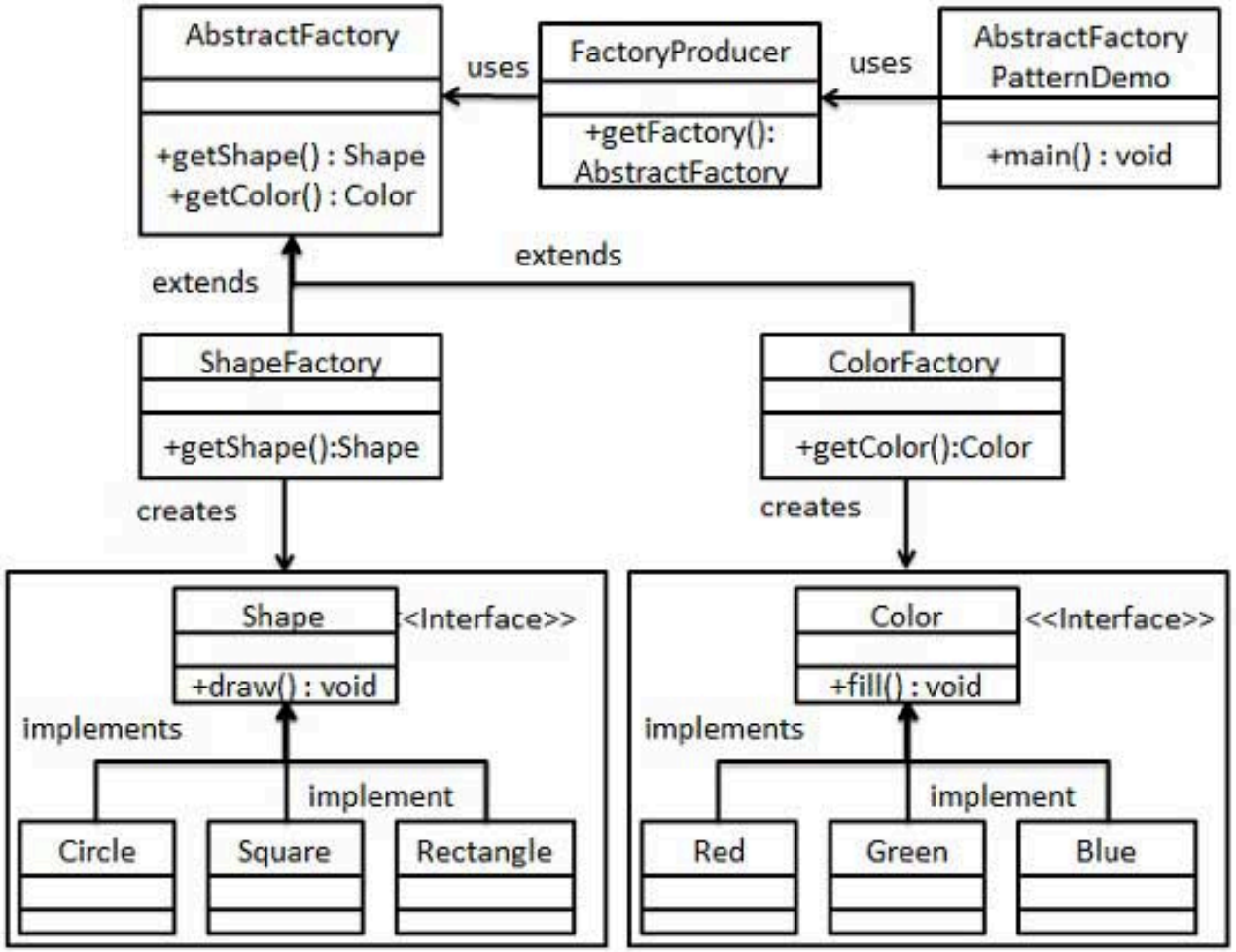
Possibilité d'étendre le comportement sans modifier la classe

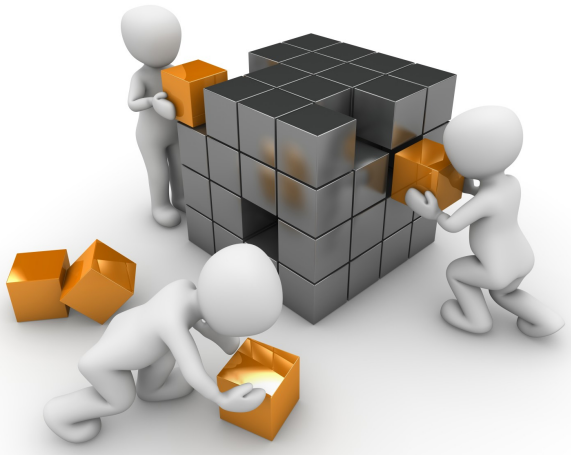
Quand l'utiliser ?

- Pour faire cohabiter des familles d'objets ayant des comportements communs (ce qui arrive très souvent)

- Quand je ne veux pas rendre accessible l'implémentation concrète d'une famille d'objets (le client n'aura accès qu'aux interfaces).

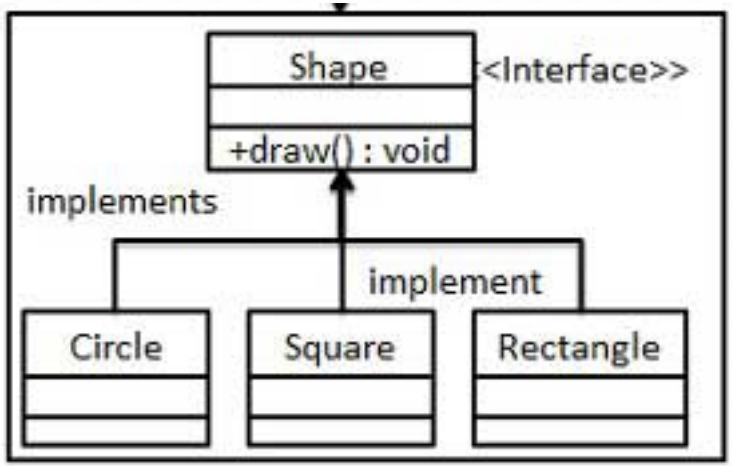
Exemple :


AbstractFactory
+getShape() : Shape
+getColor() : Color
uses
FactoryProducer
+getFactory(): AbstractFactory
uses
AbstractFactory PatternDemo
+main() : void
extends
extends
ShapeFactory
+getShape():Shape
ColorFactory
+getColor():Color
creates
creates
Shape
<Interface>>
+draw() : void
implements
implement
Circle
Square
Rectangle
Color
<<Interface>>
+fill() : void
implements
implement
Red
Green
Blue



Etape 1 :

Création de l'interface pour Forme



Shape.java

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

Etape 2 :

Création des classes concrètes implémentant la même interface

Rectangle.java

```
public class Rectangle implements Shape {

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

Square.java

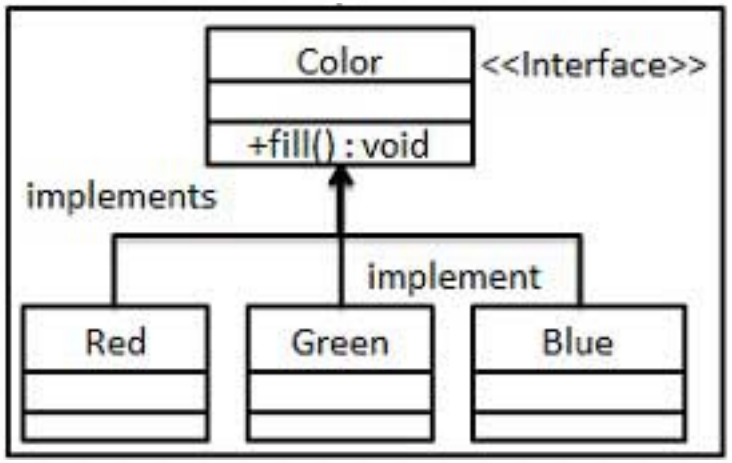
```
public class Square implements Shape {

   @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.");
   }
}
```

Etape 3 :

Création de l'interface pour Color



Color.java

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

Etape 4 :

Création des classes concrètes implémentant la même interface

Red.java

```
public class Red implements Color {

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

Green.java

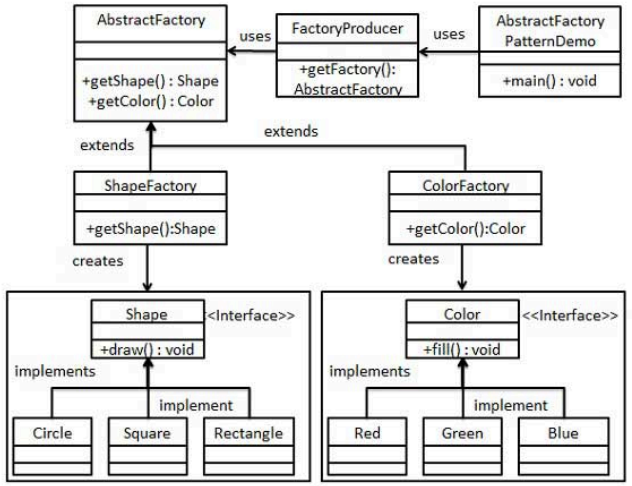
```
public class Green implements Color {

   @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.");
   }
}
```

Etape 5 :

Création de la classe abstraite pour récupérer ShapeFactory et ColorFactory

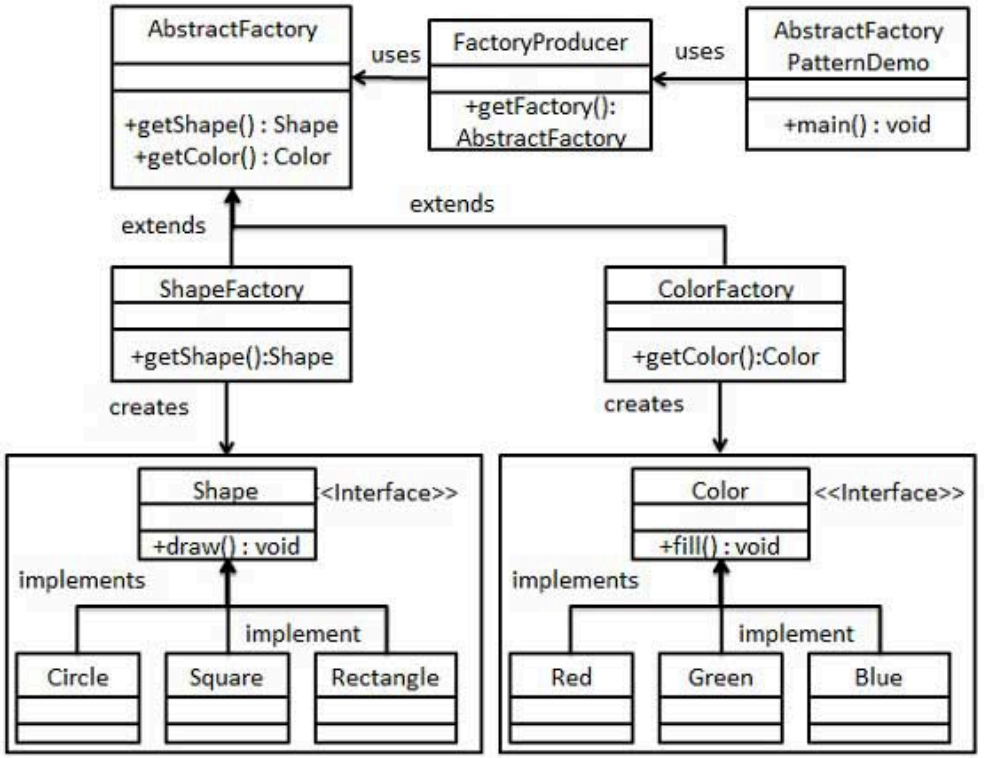


AbstractFactory.java

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

Etape 6 :

Création de ShapeFactory et ColorFactory pour générer un objet de classe concrète basé sur les informations données



Etape 6 (bis) :

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;
   }
}
```

Etape 7 :

La classe FactoryProducer pour passer les informations comme Shape ou Color

```
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;
   }
}
```

Etape 8 :

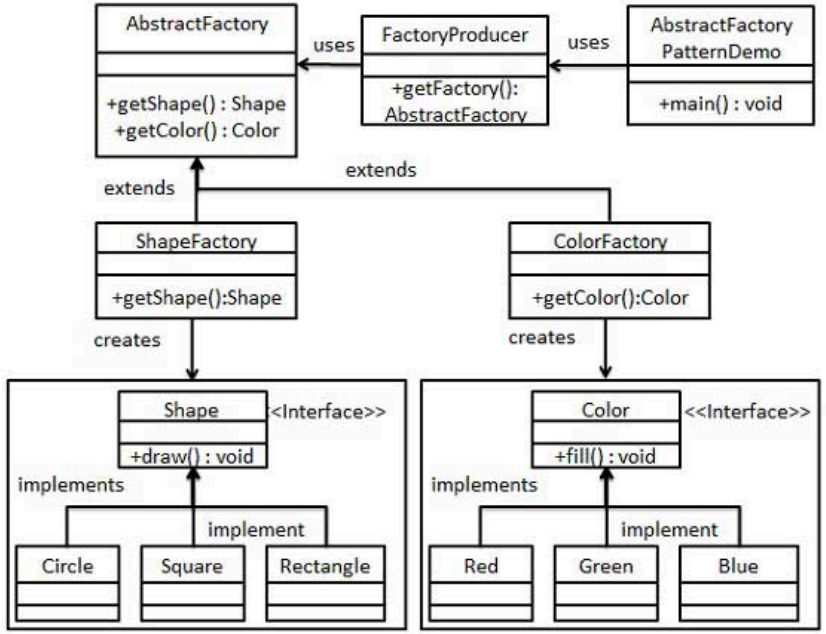
Instanciation des objets

```
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();
```

Etape 8 (bis) :

```
      //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();
   }
}
```

Etape 9 :




```
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
Inside Red::fill() method.
Inside Green::fill() method.
Inside Blue::fill() method.
```

Conclusion :

Que retenir ?

Avantages et inconvénients

Sources :

- http://blog.cellenza.com/uncategorized/abstract-factory-fabrique-abstraite/
- https://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm
- http://jormes.developpez.com/articles/design-pattern-construction/

Retrouvez l'exemple sur GitHub : https://github.com/EliseJoffre/AbstractFactoryExemple