DZone

Abstract Factory Design Pattern

by Yogen Rai MVB · Nov. 06, 18 · Java Zone · Tutorial

The abstract factory pattern has an interface that is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the factory method pattern.

An abstract factory pattern is also called the **Factory of Factories or Kit.** In other words, this pattern has a super-factory that creates other factories This design pattern comes under the creational pattern as it provides one of the best ways to create an object.

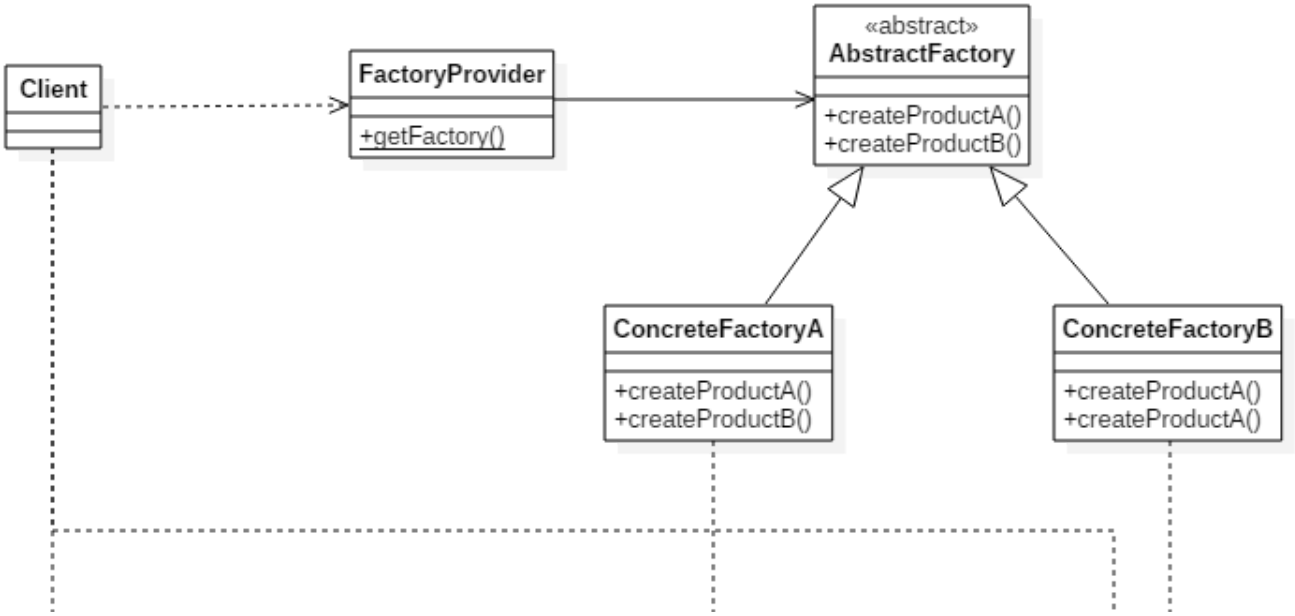
The intent of this pattern, according to *Design Patterns* by Gamma et al, is to:

> **Provide an interface for creating families of related or dependent objects without specifying their concrete classes.**

This pattern provides a level of indirection that abstracts the creation of families of related or dependent objects without directly specifying their concrete classes. The "factory" object has the responsibility for providing creative services for the entire platform family. Clients use the factory pattern to create platform objects but never create them directly.

Structure

The structure of factory method pattern is as shown in the figure below:



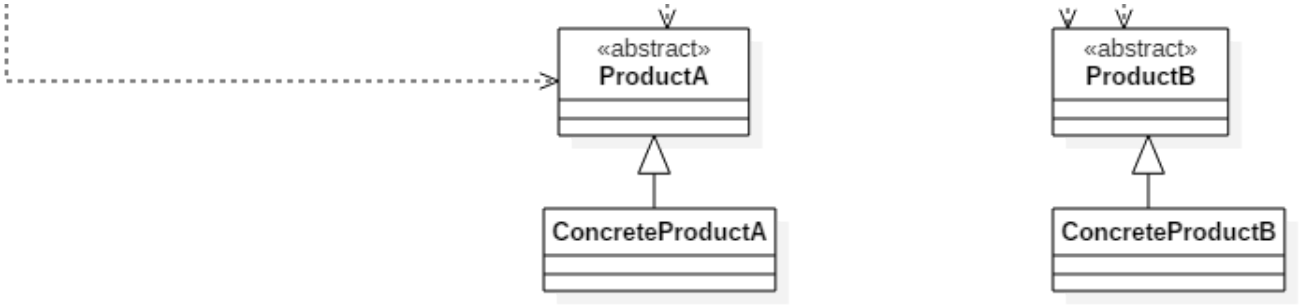


Figure: Abstract Factory Pattern Structure

The following classes are the participants of this pattern:

- `AbstractFactory` — declares an interface for operations that create abstract products.
- `ConcreteFactory` — implements operations to create concrete products.
- `AbstractProduct` — declares an interface for a type of product objects.
- `Product` — defines a product to be created by the corresponding ConcreteFactory; it implements the AbstractProduct interface.
- `Client` — uses the interfaces declared by the AbstractFactory and AbstractProduct classes.

Example

Let us modify an application from the factory method pattern, which draws different geometric shapes, this time including 2D or 3D, on the basis of client demand. The class diagram of the application is as shown below:

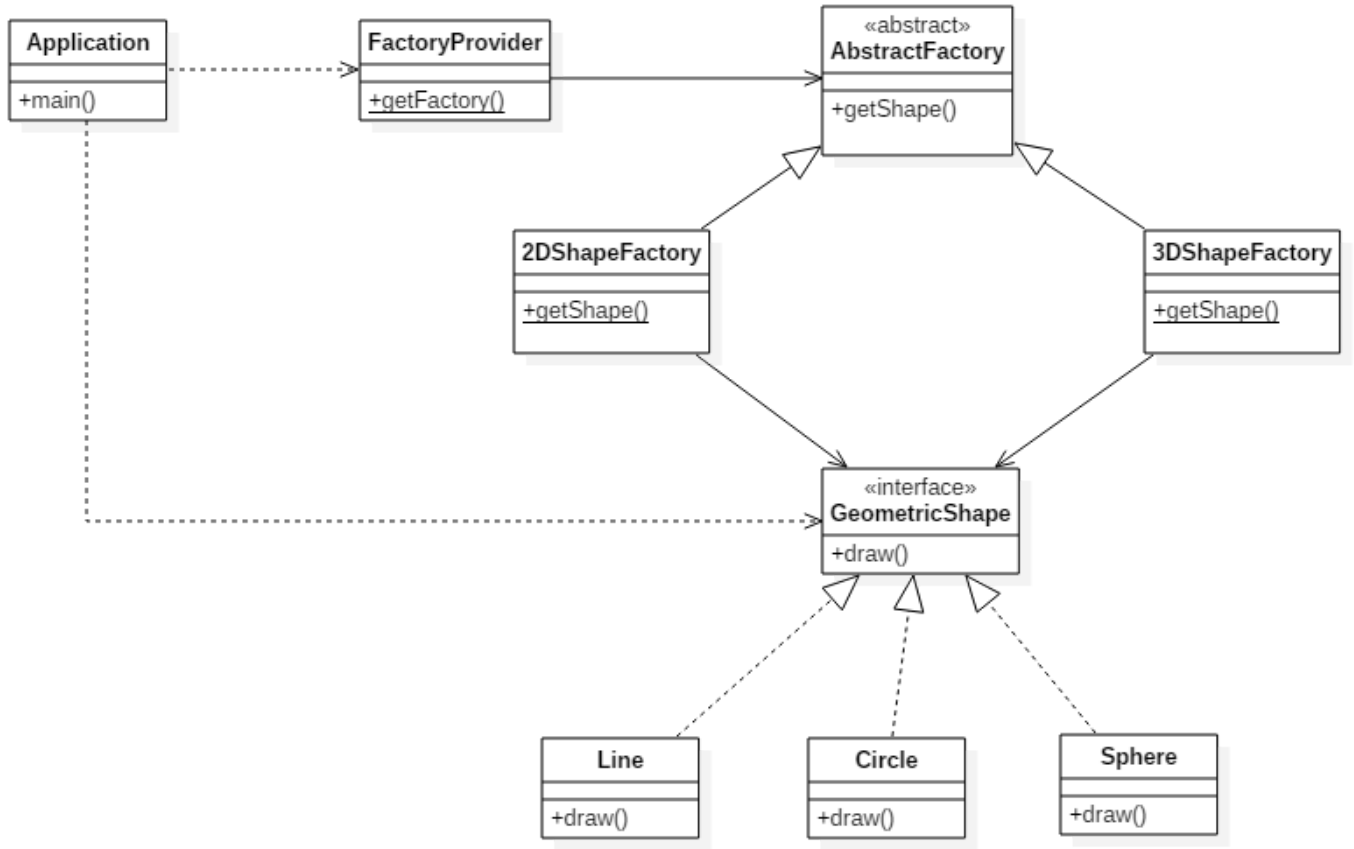


Figure: Abstract Factory Method Example

Figure: Abstract Factory Method Example

Here, the `FactoryProvider` provides factories on the basis of the name provided by the client (application.java in this case). With this factory object, we obtain the corresponding geometric shape, which we'll use to draw the shape.

Java Implementation

In order to implement the above-mentioned design, let's start with the `GeometricShape` interface.

```
/**
 * Product interface
 */
public interface GeometricShape {
    void draw();
}
```

The concrete implementations of the above interface are:

Line.java

```
/**
 * Concrete Product
 */
public class Line implements GeometricShape {
    @Override
    public void draw() {
        System.out.println("Line Drawn.");
    }
}
```

Circle.java

```
/**
 * Concrete product
 */
public class Circle implements GeometricShape {
    @Override
    public void draw() {
        System.out.println("Circle is drawn.");
    }
}
```

Sphere.java

```
/**
 * Concrete product
```

```
 * Concrete product
 */
public class Sphere implements GeometricShape {
    @Override
    public void draw() {
        System.out.println("Sphere drawn.");
    }
}
```

I have added following enums to name the factories as well as shapes:

```
public enum FactoryType {
  TWO_D_SHAPE_FACTORY,
  THREE_D_SHAPE_FACTORY
}

public enum  ShapeType {
    LINE,
    CIRCLE,
    SPHERE
}
```

Now, let's create the `AbstractFactory` .

```
/**
 * Abstract Factory
 */
public abstract class AbstractFactory {
    abstract GeometricShape getShape(ShapeType name);
}
```

The concrete classes for factories are:

TwoDShapeFactory.java

```
/**
 * Concrete factory
 */
public class TwoDShapeFactory extends AbstractFactory {
    @Override
    GeometricShape getShape(ShapeType name) {
        if (ShapeType.LINE == name) {
            return new Line();
        } else if (ShapeType.CIRCLE == name) {
            return new Circle();
        }
```

```
        return null;
    }
}
```

ThreeDShapeFactory.java

```
/**
 * Concrete factory
 */
public class ThreeDShapeFactory extends AbstractFactory {
    @Override
    GeometricShape getShape(ShapeType name) {
        if (ShapeType.SPHERE == name) {
            return new Sphere();
        }
        return null;
    }
}
```

The `FactoryProvider` class is:

```
/**
 * Factory provider
 */
public class FactoryProvider {
    public static AbstractFactory getFactory(FactoryType factoryType) {
        if (FactoryType.TWO_D_SHAPE_FACTORY == factoryType) {
            return new TwoDShapeFactory();
        } else if (FactoryType.THREE_D_SHAPE_FACTORY == factoryType) {
            return new ThreeDShapeFactory();
        }
        return null;
    }
}
```

Finally, the code for the client of this application is:

```
/**
 * Client
 */
public class Application {
    public static void main(String[] args) {
        //drawing 2D shape
        AbstractFactory factory = FactoryProvider.getFactory(TWO_D_SHAPE_FACTORY);
```

```
        if (factory == null) {
            System.out.println("Factory for given name doesn't exist.");
            System.exit(1);
        }
        //getting shape using factory obtained
        GeometricShape shape = factory.getShape(ShapeType.LINE);
        if (shape != null) {
            shape.draw();
        } else {
            System.out.println("Shape with given name doesn't exist.");
        }

        //drawing 3D shape
        factory = FactoryProvider.getFactory(THREE_D_SHAPE_FACTORY);
        if (factory == null) {
            System.out.println("Factory for given name doesn't exist.");
            System.exit(1);
        }
        //getting shape using factory obtained
        shape = factory.getShape(ShapeType.SPHERE);
        if (shape != null) {
            shape.draw();
        } else {
            System.out.println("Shape with given name doesn't exist.");
        }
    }
}
```

The output of the program is:

```
Line Drawn.
Sphere drawn.
```

The output shows that the factory corresponding to shape is required, which, in turn, is used to create an object.

Conclusion

This post talked about the summarized form of the Abstract Factory Method, as one of the GOF patterns, with a simple example.

The source code for all example presented above is available on GitHub.

Happy coding!

Like This Article? Read More From DZone

Design Patterns in the 21st Century, Part One

The Maximal Decoupled "Parameterized Factory-Method-Pattern"

Creational Design Patterns: Builder Pattern

Free DZone Refcard
Java Containerization

Topics: JAVA , DESIGN-PATTERN , ABSTRACT FACTORY PATTERN

Opinions expressed by DZone contributors are their own.

ava Partner Resources

ep insight into your code with IntelliJ IDEA.
tBrains
|
e Build vs. Buy Challenge: Insight Into a Hybrid Approach
elissa
|
ee eBook: SCA Maturity Model - A Framework for Open Source Security and Compliance
exera
|
grating to Microservice Databases
d Hat Developer Program
|