Design Patterns

Engr. Abdul-Rahman Mahmood

DPM, MCP, QMR(ISO9001:2000)

armahmood786@yahoo.com

alphapeeler.sf.net/pubkeys/pkey.htm

pk.linkedin.com/in/armahmood

www.twitter.com/alphapeeler

www.facebook.com/alphapeeler

abdulmahmood-sss alphasecure

armahmood786@hotmail.com

http://alphapeeler.sf.net/me

alphasecure@gmail.com

http://alphapeeler.sourceforge.net

http://alphapeeler.tumblr.com

armahmood786@jabber.org

alphapeeler@aim.com

mahmood_cubix 48660186

alphapeeler@icloud.com

http://alphapeeler.sf.net/acms/

Abstract Factory Pattern

What is Decorator pattern?

Decorator is one of the 23 Design Patterns which were selected by the GoF (Gang of Four).

		Purpose		
		Creation	Structure	Behavior
Scope	Class	Factory Method		Interpreter Template
	Objects	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Abstract Factory: Intent

- Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called as factory of factories. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.
- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- A hierarchy that encapsulates: many possible "platforms", and the construction of a suite of "products".
- The new operator considered harmful.

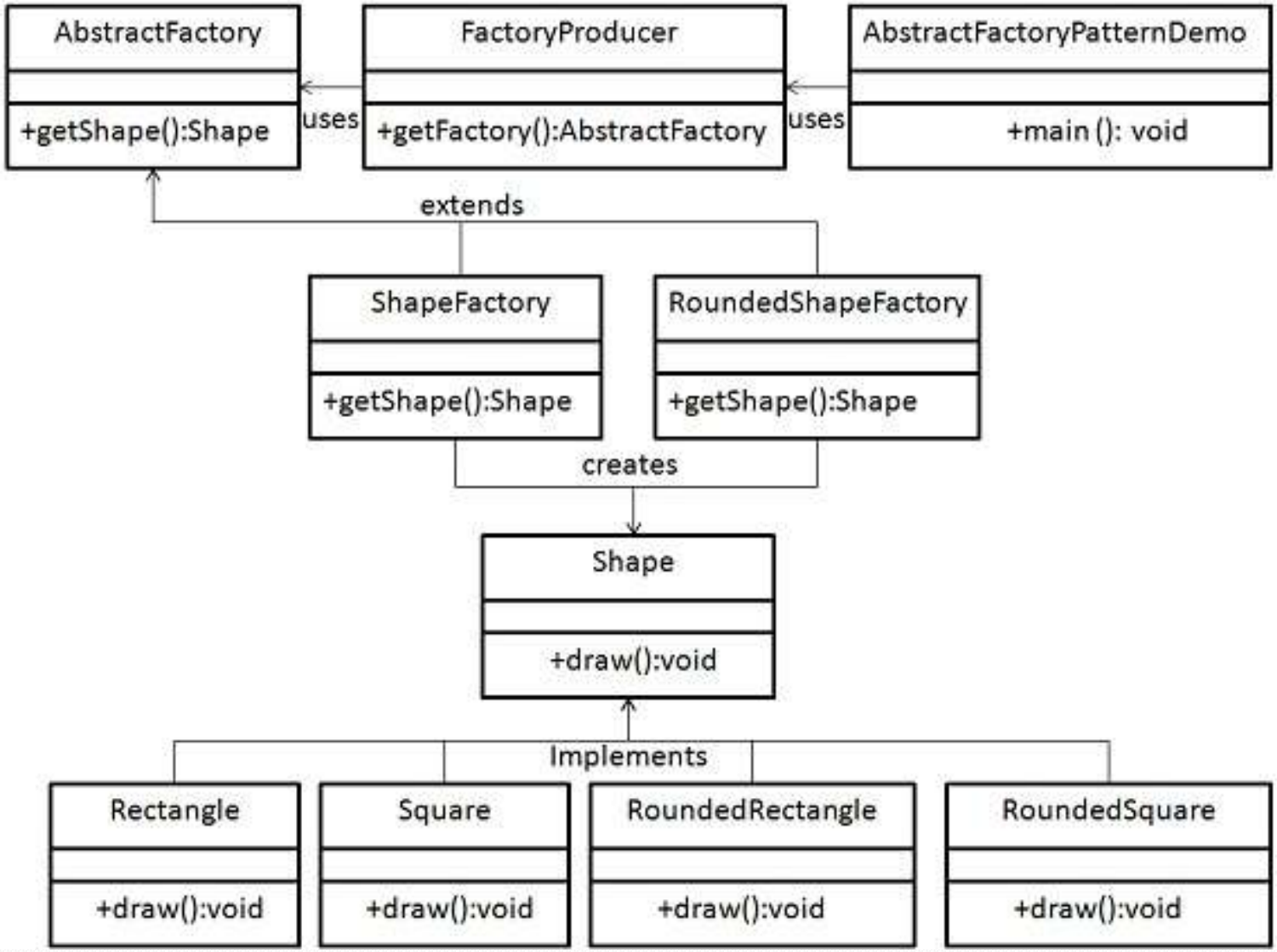
Problem

- If an application is to be **portable**, it needs to **encapsulate platform dependencies**. These "platforms" might include: windowing system, operating system, database, etc. Too often, this encapsulation is not engineered in advance, and lots of **#ifdef case statements** with options for all currently supported platforms begin to procreate throughout the code.
- The "factory" object has the responsibility for **providing creation services for the entire platform** family. **Clients never create platform objects directly,** they ask the factory to do that for them.

Implementation

- We are going to create a *Shape* interface and concrete classes implementing these interfaces. We create an abstract factory class *AbstractFactory* as next step. Factory classe *ShapeFactory* and *RoundedShapeFactory* are defined where each factory extends *AbstractFactory*. A factory creator/generator class *FactoryProducer* is created.
- *AbstractFactoryPatternDemo*, our demo class uses *FactoryProducer* to get a *AbstractFactory* object. It will pass information (*RECTANGLE / SQUARE* for *Shape*) to *AbstractFactory* to get the type of object it needs. It also passes information (Boolean: *rounded*) to *AbstractFactory* to get the type of object it needs.

Abstract Factory - Example



Abstract Factory - Example

- Step 1: Create an interface Shape.java

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

- Step 2: Create concrete classes implementing the same interface.

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

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

Abstract Factory - Example

- Step 2: Create concrete classes implementing the same interface.

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

- **Step 3:** Create an Abstract class to get factories for Normal and Rounded Shape Objects.

```
public abstract class AbstractFactory {
abstract Shape getShape(String shapeType) ;
}
```

Abstract Factory - Example

- **Step 4:** Create Factory classes extending AbstractFactory to generate object of concrete class based on given information.

```
public class ShapeFactory extends AbstractFactory {
 @Override
   public Shape getShape(String shapeType){
      if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();
      }else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
      return null;
   }
}
```

Abstract Factory - Example

- **Step 4:** Create Factory classes extending AbstractFactory to generate object of concrete class based on given information.

```
public class RoundedShapeFactory extends AbstractFactory {
@Override
   public Shape getShape(String shapeType){
      if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new RoundedRectangle();
      }else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new RoundedSquare();
      }
      return null;
   }
}
```

Abstract Factory - Example

- **Step 5:**
- Create a Factory generator/producer class to get factories by passing an information such as Shape

```
public class FactoryProducer {
public static AbstractFactory getFactory(boolean rounded){
      if(rounded){
         return new RoundedShapeFactory();
      }else{
         return new ShapeFactory();
      }
   }
}
```

Abstract Factory - Example

- Step6: *AbstractFactoryPatternDemo.java*

```
public class AbstractFactoryPatternDemo {
public static void main(String[] args) {
    //get rounded shape factory
   AbstractFactory shapeFactory = FactoryProducer.getFactory(false);
   //get an object of Shape Rectangle
   Shape shape1 = shapeFactory.getShape("RECTANGLE");
   //call draw method of Shape Rectangle
   shape1.draw();
   //get an object of Shape Square
   Shape shape2 = shapeFactory.getShape("SQUARE");
   //call draw method of Shape Square
   shape2.draw();
   //get rounded shape factory
   AbstractFactory shapeFactory1 = FactoryProducer.getFactory(true);
   //get an object of Shape Rounded Rectangle
   Shape shape3 = shapeFactory1.getShape("RECTANGLE");
   //call draw method of Shape Rounded Rectangle
   shape3.draw();
   //get an object of Shape Rounded Square
   Shape shape4 = shapeFactory1.getShape("SQUARE");
   //call draw method of Shape Rounded Square
   shape4.draw();
}
}
```

Abstract Factory - Example

- Step7: output

```
Inside Rectangle::draw() method.
Inside Square::draw() method.
Inside RoundedRectangle::draw() method.
Inside RoundedSquare::draw() method.
```

RoundedSqua... | Rectangle.java | Square.java | AbstractFac... | ShapeFactory... | RoundedShap... | FactoryProdu... | *AbstractFac...

```java
9  public class AbstractFactoryPatternDemo {
10     public static void main(String[] args) {
11         // TODO Auto-generated method stub
12         //get rounded shape factory
13           AbstractFactory shapeFactory = FactoryProducer.getFactory(false);
14           //get an object of Shape Rectangle
15           Shape shape1 = shapeFactory.getShape("RECTANGLE");
16           //call draw method of Shape Rectangle
17           shape1.draw();
18           //get an object of Shape Square
19           Shape shape2 = shapeFactory.getShape("SQUARE");
20           //call draw method of Shape Square
21           shape2.draw();
22           //get rounded shape factory
23           AbstractFactory shapeFactory1 = FactoryProducer.getFactory(true);
24           //get an object of Shape Rounded Rectangle
25           Shape shape3 = shapeFactory1.getShape("RECTANGLE");
26           //call draw method of Shape Rounded Rectangle
27           shape3.draw();
28           //get an object of Shape Rounded Square
29           Shape shape4 = shapeFactory1.getShape("SQUARE");
```

Problems | Declaration | Console | Progress | Properties | Model Explorer

```
<terminated> AbstractFactoryPatternDemo [Java Application] C:\Program Files\Java\jre1.8.0_121\bin\javaw.exe (Feb 20, 2019, 11:40:37 PM)
Inside Rectangle::draw() method.
Inside Square::draw() method.
Inside RoundedRectangle::draw() method.
Inside RoundedSquare::draw() method.
```

Factory vs Decorator

- Patterns like builder and factory(and abstract factory) are **used in creation of objects**. And the patterns like decorator (also called as structural design patterns) are **used for extensibility or to provide structural changes to already created objects.**

- Both types of patterns largely **favor composition over inheritance,** so giving this as a differentiator for using builder instead of decorator will not make any sense. **Both give behavior upon runtime rather than inheriting it.**

Factory vs Decorator

- Factory
  - Use builder (factory) if we want to **limit the object creation with certain properties/features.** For example there are 4-5 attributes which are mandatory to be set before the object is created or we want to freeze object creation until certain attributes are not set yet. Basically, use it instead of constructor.
- Decorator
  - Decorator is used to **add new features of an existing object to create a new object.** There is no restriction of freezing the object until all its features are added. Both are using composition so they might look similar but they differ largely in their use case and intention.