Lesson 5 Factory Method and Abstract Factory

The Factory Method Pattern

1. Intent

   Define an interface for creating an object, but let subclasses (concrete class that implements it) decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

2. Motivation

   a. Sometimes an object may only know that it needs an object of a certain type but does not know exactly which one from the set of subclasses of the parent class is to be selected.

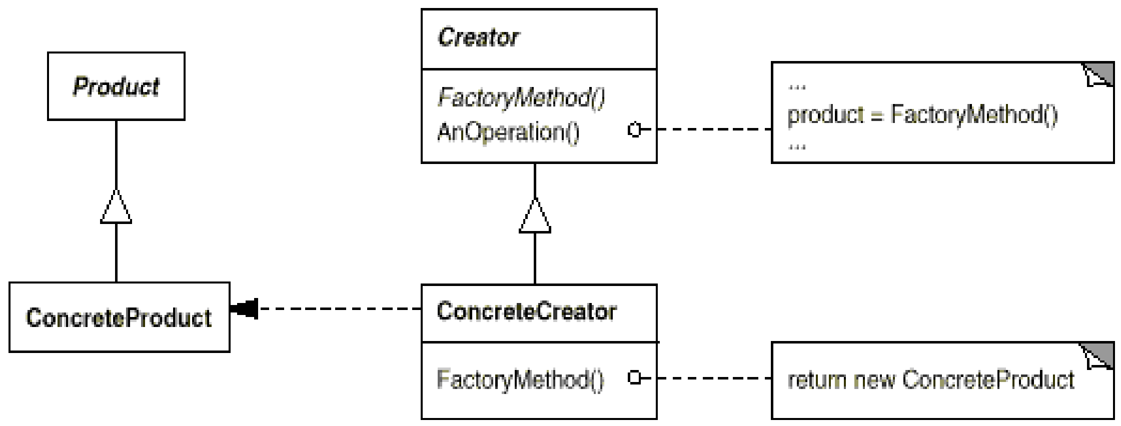
   b. The choice of an appropriate class may depend on factors such as:
   – The state of the running application.
   – Application configuration settings.
   – Expansion of requirements or enhancements.

   c. Factory Method recommends encapsulating the functionality required, to select and instantiate an appropriate class, inside a designated method referred to as a factory method.

   d. A factory method can be defined as a method in a class that:
   – Selects an appropriate class from a class hierarchy based on the application context and other influencing factors.

– Instantiates the selected class and returns it as an instance of the parent class type.

3. Structure



4. Participants

a. Product
- defines the interface of objects the factory method creates.

b. ConcreteProduct
- implements the Product interface.

c. Creator
- declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
- may call the factory method to create a Product object.

d. ConcreteCreator
- overrides the factory method to return an instance of a ConcreteProduct.

5. Applicability

Use the Factory Method pattern when

a. A class can't anticipate the class of objects it must create.
b. A class wants its subclasses to specify the objects it creates.
c. Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

6. Consequences

a. Factory methods eliminate the need to bind application-specific classes into your client code. The client code only deals with the Product interface; therefore it can work with any user-defined concrete product classes.
b. Any change in a concrete product class does not have any impact on the client code.

7. How to implement the Factory Method Pattern

a. Factory interface (or an abstract class)

```
public interface PizzaFactory {
	public Pizza createPizza(String type);
}
```

b. Concrete factory implementation

```
public class SimplePizzaFactory implements PizzaFactory{
	//This factory should often times be a Singleton
	private static PizzaFactory factory = new SimplePizzaFactory();

	private SimplePizzaFactory(){}
```

```
        public static PizzaFactory getFactory(){
              return factory;
        }

        public Pizza createPizza(String type) {
              Pizza pizza = null;

              if (type.equals("cheese")) {
                    pizza = new CheesePizza();
              } else if (type.equals("pepperoni")) {
                    pizza = new PepperoniPizza();
              } else if (type.equals("clam")) {
                    pizza = new ClamPizza();
              } else if (type.equals("veggie")) {
                    pizza = new VeggiePizza();
              }
              return pizza;
        }
  }
```

c. Your product class hierarchy – Pizza superclass and its subclasses, like CheesePizza, PepperoniPizza, etc.

```
abstract public class Pizza {
      String name;
      String dough;
      String sauce;
//…
```

d. Client

```
public class FactoryMethodClient {

   public static void main(String[] args) {
         PizzaFactory factory = SimplePizzaFactory.getFactory();
         Pizza pizza = factory.createPizza("cheese");

         //different from below on when the decision has to be made -
         runtime or compile time?
         //which is one difference between framework and application
         development
         Pizza pizza1 = new CheesePizza();
         System.out.println(pizza.getClass().getSimpleName());
   }

}
```

Lab 5-1

Suppose you are writing a simple debug tool for your colleagues. To start with, you are going to provide 2 implementations - one writes the messages out to the command line (ConsoleTrace), while another writes them to a file (FileTrace). All you want your colleagues to know about the tool is

1) The interface (or what my tool can do for you)

```java
public interface Trace {
    // turn on and off debugging
    public void setDebug( boolean debug );
    // write out a debug message
    public void debug( String message );
    // write out an error message
    public void error( String message );
}
```

2) How to choose a certain debugger by giving an argument to your main(String args[]) method. (for example, use "trace.log" to choose the FileTrace implementation or "console" for the ConsoleTrace debugger.

You may want to add more implementations later as they need it. But they won't have to change any code to switch from one implementation to another. Design/write your debugging tool in Java.

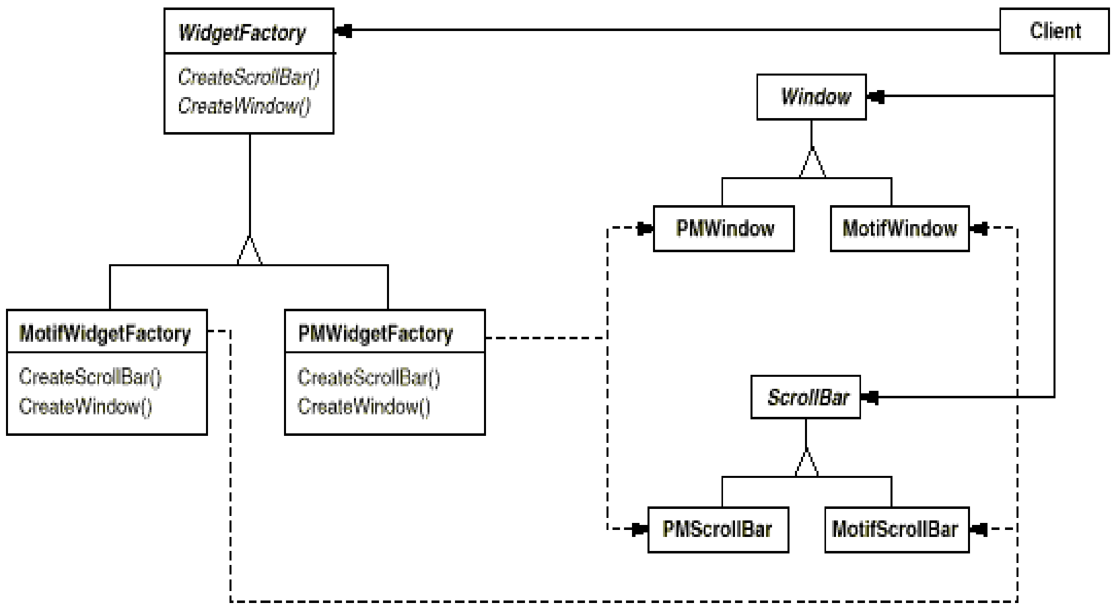
Abstract Factory

1. Intent

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

2. Motivation

   a. Your application needs to create different series of products that belong to different inheritance hierarchies (or suites/ families of related, dependent classes).
   b. You have a common interface to create all different families of products.

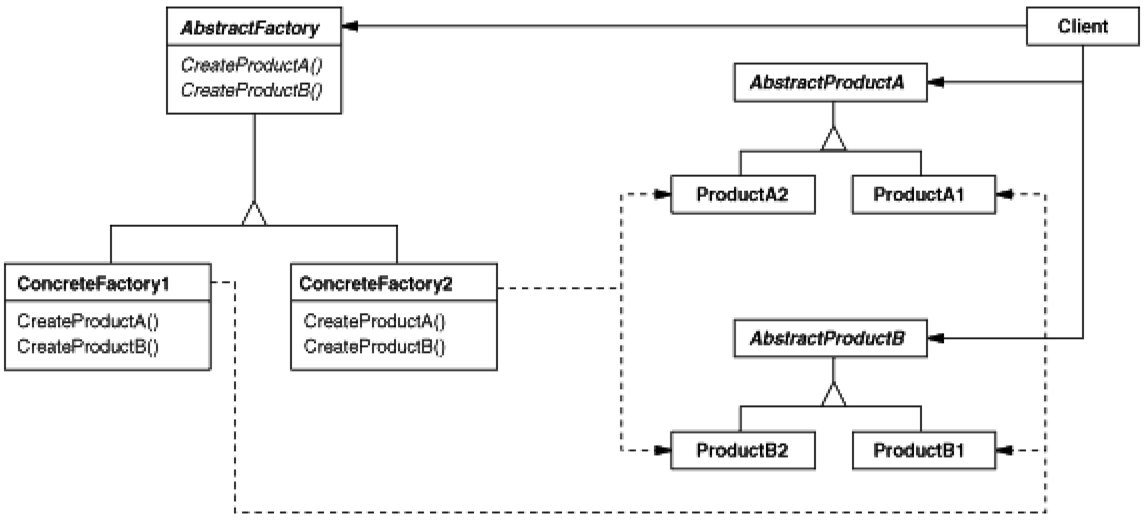


3. Applicability

Use the Abstract Factory pattern in any of the following situations:

a. A system should be independent of how its products are created, composed, and represented.
b. A class can't anticipate the class of objects it must create.
c. A system must use just one of a set of families of products
d. A family of related product objects is designed to be used together, and you need to enforce this constraint.

4. Structure



5. Participants

a. AbstractFactory
   - declares an interface for operations that create abstract product objects.

b. ConcreteFactory

   - implements the operations to create concrete product objects.

c. AbstractProduct
   - declares an interface for a type of product object.

d. ConcreteProduct
   - defines a product object to be created by the corresponding concrete factory.
   - implements the AbstractProduct interface.

e. Client
   - uses only interfaces declared by AbstractFactory and AbstractProduct classes.

6. Consequences

a. It isolates concrete classes. Product class names are isolated in the implementation of the concrete factory; they do not appear in client code.
b. It makes exchanging product families easy.
c. It promotes consistency among products.
d. Supporting new kinds of products is difficult.

7. How to implement the Abstract Factory Pattern?

a. Abstract Factory Class

```
public interface WidgetFactory {
   public abstract AbstractFrameWidget createFrame();
   public abstract AbstractButtonWidget createButton();
   public abstract AbstractCheckboxWidget createCheckbox();
   public abstract AbstractListboxWidget createListbox();
}
```

b. Abstract Product Classes

```
public abstract class AbstractFrameWidget {
   //...
}
```

```java
public abstract class AbstractButtonWidget {
   //…
}

public abstract class AbstractCheckboxWidget {
   //…
}

public abstract class AbstractListboxWidget {

}
```

c. Concrete Factory Classes

```java
public class WindowsWidgetFactory implements WidgetFactory {

   @Override
   public AbstractFrameWidget createFrame() {
         // TODO Auto-generated method stub
         return new WindowsFrame();
   }

   @Override
   public AbstractButtonWidget createButton() {
         return new WindowsButton();
   }

   @Override
   public AbstractCheckboxWidget createCheckbox() {
         return new WindowsCheckbox();
   }

   @Override
   public AbstractListboxWidget createListbox() {
         return new WindowsListbox();
   }

}

public class MacWidgetFactory implements WidgetFactory {

   @Override
   public AbstractButtonWidget createButton() {
         return new MacButton();
   }

   @Override
   public AbstractCheckboxWidget createCheckbox() {
         return new MacCheckbox();
   }

   @Override
   public AbstractListboxWidget createListbox() {
         return new MacListbox();
   }

   @Override
   public AbstractFrameWidget createFrame() {
         // TODO Auto-generated method stub
         return new MacFrame();
   }

}
```

d. Concrete Product Classes

```
public class WindowsFrame extends AbstractFrameWidget {
         //…
}

public class MacFrame extends AbstractFrameWidget {
         //…
}
```

And other product classes…

e. Client

```
public class GUIBuilder {
   private WidgetFactory widgetFactory;

   public GUIBuilder(){
         if(System.getProperty("os.name").startsWith("Windows")){
               widgetFactory  = new WindowsWidgetFactory();
         } else if(System.getProperty("os.name").startsWith("Mac")){
               widgetFactory  = new MacWidgetFactory();
         }
   }

   public WidgetFactory getWidgetFactory(){
         return widgetFactory;
   }

   public static void main(String[] args){
         GUIBuilder guiBuilder = new GUIBuilder();
         WidgetFactory wf = guiBuilder.getWidgetFactory();
         if ( wf == null ){
               System.out.println("This platform is not being 
supported.");
               return;
         }
         AbstractFrameWidget myFrame = wf.createFrame();
         AbstractButtonWidget myButton = wf.createButton();
         AbstractCheckboxWidget myCheckbox = wf.createCheckbox();
         AbstractListboxWidget myListbox = wf.createListbox();
         //...
   }
}
```

Lab 5-2

An online business sells gift items to both individuals and businesses. Each order is treated as a gift pack that may contain 1 or more gift items. When a customer places an order, the type of the gift pack has to be specified as one of the 3 - Business, Adults, or Kids. Depending on the gift pack type, each individual gift item needs to be packaged accordingly. For example, a gift item for a kid can be packaged with a 'Micky Gift Bag', or a 'Cartoon Box' or the 'Happy Kid' gift wrap which has a different cost associated with each option (selected by the person placing the order). For now, there are only 3 available options – gift bag, gift box or gift wrap. For the Business type, the options are 'Merchant Collection' bag, 'Hard Plastic' box and 'Holiday Surprise' wrap. For the Adults, they 'Reusable Shopper' bag, 'Plain Paper' box, and 'Everyday Value' wrap. Prices as listed below.

	Business		*Adults*		*Kids*	
Bag	Merchant Collection	\$0.50	Reusable Shopper	\$0.00	Micky	\$0.25
Box	Hard Plastic	\$1.00	Plain Paper	\$0.25	Cartoon	\$0.50
Wrap	Holiday Surprise	\$0.25	Everyday Value	\$0.00	Happy Kid	\$0.10

Suppose you are going to create a model and implement it with the Abstract Factory Pattern for the business to easily get packaging instructions and calculate the packaging cost from each of the orders. Also provide a client program that prints the packing instructions and the total packaging cost for each gift pack.

Use the following skeleton code for your implementation.

```
public class GiftItem {
      private String giftId;
      private String giftName;
      private String description;
      private String packagingType; //"bag", "box", or "wrap".
      Private Packaging packaging;
      //…

}

public class GiftPack {
      private List<GiftItem> giftItems;
      private Address shippingAddress;
      private String packType; //"Business", "Adults", or "Kids"
      //…

}
```

```java
public interface Packaging {
	public float getCost();
	//...
}

public class Address {
	private String street1;
	private String street2;
	private String city;
	private String state;
	private String zipcode;

}
```