GoF Design Patterns - with examples using Java and UML2

written by:

Benneth Christiansson (Ed.)
Mattias Forss,
Ivar Hagen,
Kent Hansson,
Johan Jonasson,
Mattias Jonasson,
Fredrik Lott,
Sara Olsson, and
Thomas Rosevall

Table of Contents

Foreword

This book is the result of a joint effort of the authors with an equal contribution from all. The idea to the book originated during the participation of a Java Architect training program taught at Logica Sverige AB Karlstad office. During the course the authors identified the lack of a quick-guide book to the basic GoF[1] design patterns. A book that could be used as a bare bone reference as well as a learning companion for understanding design patterns. So we divided the workload and together we created an up-to-date view of the GoF design patterns in a structured and uniform manner. Illustrating the choosen patterns with examples in Java and diagrams using UML2 notation. We have also emphasized benefits and drawbacks for the individual patterns and, where applicable. We also illustrate real world usage situations where the pattern has successfully been implemented.

I personally as editor must express my deepest admiration for the dedication and effort the authors have shown during this process, everyone who ever have written a book knows what I mean. I am really proud to be the editor of this very usable book.

--- *Benneth Christiansson, Karlstad, autumn 2008* ---

1 Design Patterns Elements of Reusable Elements by Gamma, Helm, Johnson and Vlissides (1995)

Chapter 1 Creational Patterns

"Creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation."[2]

All the creational patterns define the best possible way in which an object can be created considering reuse and changeability. These describes the best way to handle instantiation. Hard coding the actual instantiation is a pitfall and should be avoided if reuse and changeability are desired. In such scenarios, we can make use of patterns to give this a more general and flexible approach.

2 http://en.wikipedia.org/wiki/Creational_pattern

Factory Pattern

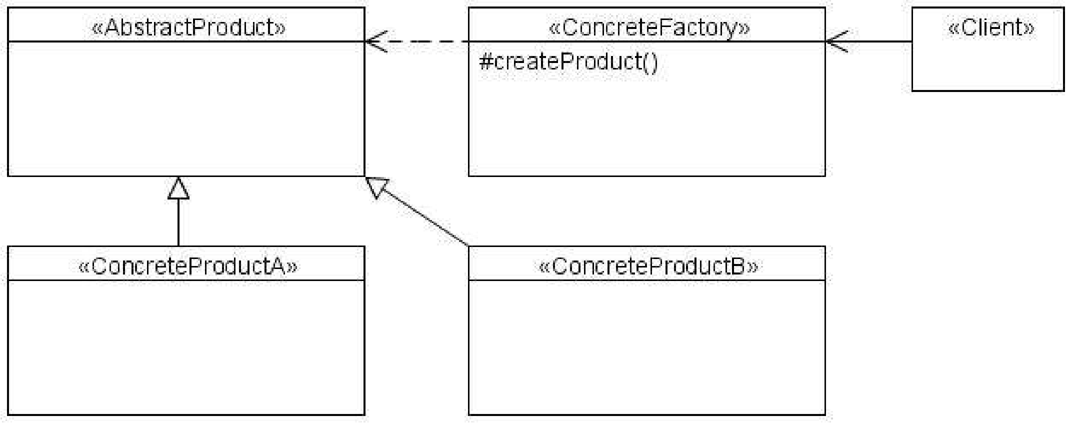
Definition
The Factory pattern provides a way to use an instance as a object factory. The factory can return an instance of one of several possible classes (in a subclass hierarchy), depending on the data provided to it.

Where to use
•When a class can't anticipate which kind of class of object it must create.
•You want to localize the knowledge of which class gets created.
•When you have classes that is derived from the same subclasses, or they may in fact be unrelated classes that just share the same interface. Either way, the methods in these class instances are the same and can be used interchangeably.
•When you want to insulate the client from the actual type that is being instantiated.

Benefits
•The client does not need to know every subclass of objects it must create. It only need one reference to the abstract class/interface and the factory object.
•The factory encapsulate the creation of objects. This can be useful if the creation process is very complex.

Drawbacks/consequences
•There is no way to change an implementing class without a recompile.

Structure



Small example

This example shows how two different concrete Products are created using the ProductFactory. ProductA uses the superclass writeName method. ProductB implements writeName that reverses the name.

```
public abstract class Product {
      public void writeName(String name) {
          System.out.println("My name is "+name);
    }
}


public class ProductA extends Product { }


public class ProductB extends Product {
    public void writeName(String name) {
        StringBuilder tempName = new StringBuilder().append(name);
        System.out.println("My reversed name is" +
            tempName.reverse());
    }
}


public class ProductFactory {
    Product createProduct(String type) {
        if(type.equals("B"))
            return new ProductB();
        else
          return new ProductA();
    }
}
```