Manufacturing Java Objects with the Factory Method Design Pattern

by Barry A. Burd and Michael P. Redlich

Introduction

This article, the third in a series of design patterns, introduces the Factory Method design pattern, one of the 23 design patterns defined in the legendary 1995 book *Design Patterns – Elements of Reusable Object-Oriented Software*. The authors of this book, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, are known affectionately as the *Gang of Four* (GoF).

Design Patterns

The GoF book defines 23 design patterns. The patterns fall into three categories:

- A creational pattern abstracts the instantiation process.
- A structural pattern groups objects into larger structures.
- A behavioral pattern defines better communication among objects.

The Factory Method design pattern fits into the creational category.

The Factory Method Pattern

According to the GoF book, the Factory Method design pattern “Defines an interface for creating an object, but lets the subclasses decide which to instantiate. Factory Method lets a class defer instantiation to subclasses.” The Factory Methods is also known as “virtual constructor” because of this unique attribute of allowing subclasses to “decide” which concrete class to instantiate.

Motivation

The Factory Method design pattern is for situations in which

- You need to instantiate a particular object from a pool of related objects.
- Until runtime, you don’t know which of the pool's objects to instantiate.

Consider an application for ordering a car. You don't know which car object to instantiate until the user selects a particular make and model. So how do you write the code? Listing 1 shows a clumsy way to write the code:

```
public class OrderCars
      {
      public Car orderCar(String model)
            {
            Car car;
            if(model.equals("Lucerne"))
```

```
                car = new Buick(model);
            else if(model.equals("Corvette"))
                car = new Chevrolet(model);
            else if(model.equals("Fusion"))
                car = new Ford(model);
            else if(model.equals("GTO"))
                car = new Pontiac(model);
            else if(model.equals("Vue"))
                car = new Saturn(model);

            car.buildCar();
            car.testCar();
            car.shipCar();
            }
      }
```

Listing 1: A brute-force solution

The **`OrderCars`** class defines a method, **`orderCar().`** This **`orderCar()`** method gets the job done. But the section of code containing the conditionals will change as you add or delete car makes and models. The rest of the code (building, testing, and shipping a car) does not change.

The GoF book tells you to "Identify the aspects of your application that vary and separate them from what stays the same." But the code in Listing 1 is an inconvenient mix -- a mix of changing and (relatively) unchanging code. Another way to think about this issue is to notice the mix of specific and general code in Listing 1. While the first part of Listing 1 refers to Lucernes, Buicks, and Corvettes, the second part of the code refers to **`car`** -- a reference to any **`Car`** instance.

What's the best way to partition the code into its changing and unchanging parts (into its specific and general parts)? In this scenario, the word "best" has several meanings:

- What's most versatile in terms of future development and maintenance?
- What's most easily recognizable by other developers (by virtue of its being a well-known pattern)?

UML Diagram

Figure 1 shows the official UML diagram for the Factory Method pattern.

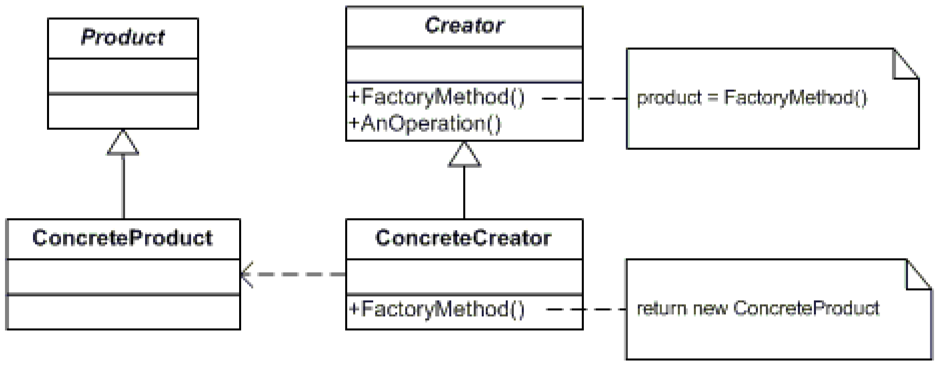

Figure 1: The UML diagram for the Factory Method design pattern

The diagram contains four classes:

- The **`Creator`** abstract class declares an abstract **`factoryMethod()`** method that returns an instance of **`Product`** or **`ConcreteProduct`**.
- The **`ConcreteCreator`** class extends the **`Creator`** class. This **`Creator`** class implements the abstract **`factoryMethod()`** method.
- The **`Product`** abstract class declares a product that's produced by a particular **`Creator`** factory.
- The **`ConcreteProduct`** class extends the **`Product`** class to define all the similar objects that can be instantiated at any given time within the application. With our car ordering application, the concrete products are **`Buick`**, **`Chevrolet`**, **`Ford`**, **`Pontiac`**, **`Saturn`**, and others that can be defined.

Using the Factory Method

Figure 2 shows a Factory Method UML diagram that's specific to the car ordering application:

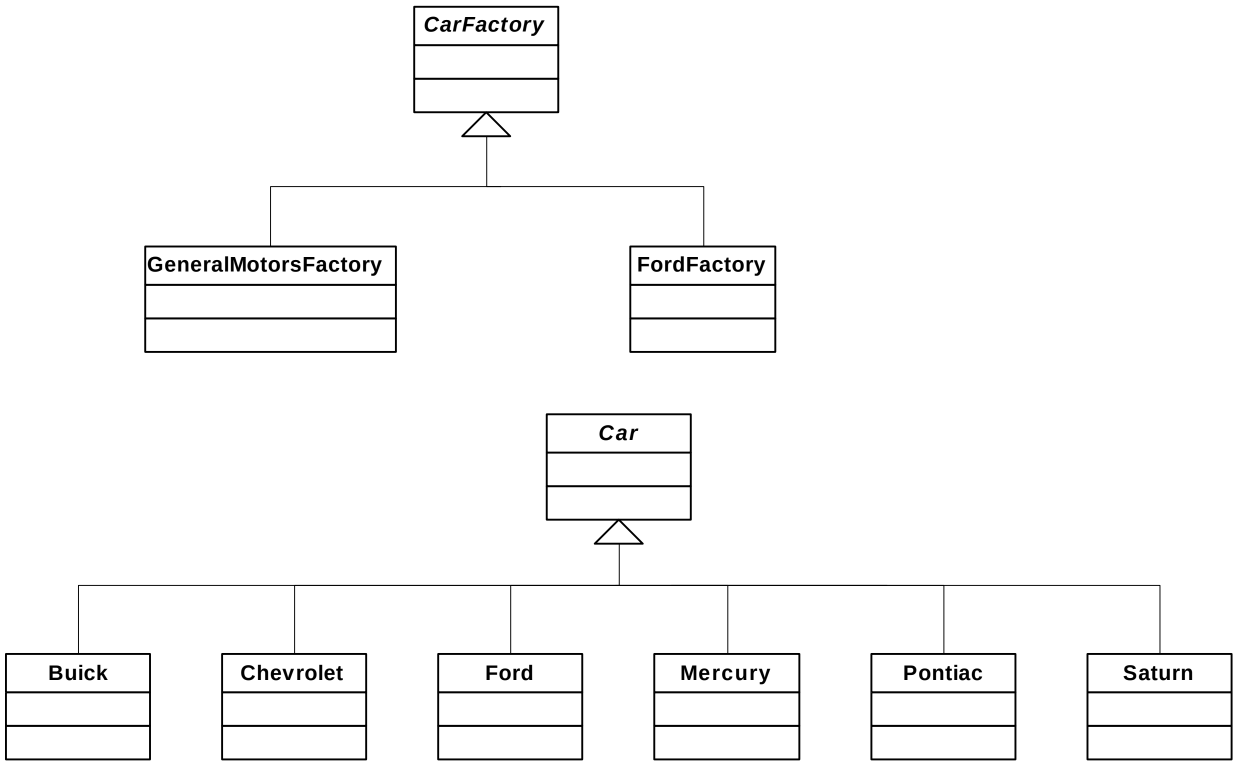

Figure 2: The UML diagram for the car ordering application.

The source code for diagram of Figure 2 is in Listings 2 to 7. The classes of the concrete products (**`Buick`**, **`Chevrolet`**, etc.) are all very similar. For the sake of brevity, only two of them appear in these listings.

```
public abstract class CarFactory
      {
      abstract Car createCar(String model);

      public Car orderCar(String make,String model)
            {
            Car car = createCar(model);
            System.out.println("--- Fulfilling the order for a " +
                  car.getMake() + " " + car.getModel() + " ---");
            car.buildCar();
            car.testCar();
            car.shipCar();
            System.out.println();
            return car;
            }
      }
```

Listing 2: The CarFactory (Creator) abstract class

```
public class GeneralMotorsFactory extends CarFactory
      {
      Car createCar(String model)
            {
            if(model.equals("Lucerne"))
                  return new Buick(model);
            else if(model.equals("Corvette"))
                  return new Chevrolet(model);
            else if(model.equals("GTO"))
                  return new Pontiac(model);
            else if(model.equals("Vue"))
                  return new Saturn(model);
            else
                  return null;
            }
      }
```

Listing 3. The GeneralMotorsFactory (ConcreteCreator) class

```
public class FordFactory extends CarFactory
      {
      Car createCar(String model)
            {
            if(model.equals("Fusion"))
                  return new Ford(model);
            else if(model.equals("Mark IV"))
                  return new Mercury(model);
            else
                  return null;
            }
      }
```

Listing 4. The FordFactory (ConcreteCreator) class

```
public abstract class Car
      {
      String make;
      String model;
      String manufacturer;

      void buildCar()
            {
            System.out.println("Building the " + getMake() + " " +
                  getModel() + " at the " + getManufacturer() +
                  " factory...");
            }

      void testCar()
```

```
            {
            System.out.println("Testing the " + getMake() + " " +
                  getModel() + " at the " + getManufacturer() +
                  " test track...");
            }

      void shipCar()
            {
            System.out.println("Shipping the " + getMake() + " " +
                  getModel() + " at the " + getMake() + " dealership...");
            }

      public String getMake()
            {
            return make;
            }

      public String getModel()
            {
            return model;
            }

      public String getManufacturer()
            {
            return manufacturer;
            }
      }
```

Listing 5. The Car (Product) abstract class

```
public class Buick extends Car
      {
      public Buick(String type)
            {
            make = "Buick";
            model = type;
            manufacturer = "General Motors";
            }
      }
```

Listing 6: The Buick (ConcreteProduct) class

```
public class Mercury extends Car
      {
      public Mercury(String type)
            {
            make = "Mercury";
            model = type;
            manufacturer = "Ford";
            }
      }
```

Listing 7: The Mercury (ConcreteProduct) class

In Listings 2 through 7, notice the clean division between general and specific code. The two abstract classes (**`Car`** and **`CarFactory`**) contain all the general code and the concrete classes (**`FordFactory`**, **`Buick`**, and so on) contain all code specific to particular makes and models.

Listing 8 is the client application. The application starts by creating instances of **`GeneralMotorsFactory`** and **`FordFactory`**. Other defined factories could be instantiated here as well. Each concrete class (**`GeneralMotorsFactory`**, **`FordFactory`**, etc.) has its own **`createCar()`** method, and shares an **`orderCar()`** method with the other concrete classes. (The **`orderCar()`** method comes from the abstract **`CarFactory`** base class.)

```
public class OrderCars
      {
      public static void main(String[] args)
            {
            CarFactory gmfactory = new GeneralMotorsFactory();
            CarFactory fordfactory = new FordFactory();

            gmfactory.orderCar("Pontiac","GTO");
            fordfactory.orderCar("Mercury","Mark IV");
            gmfactory.orderCar("Saturn","Vue");
            fordfactory.orderCar("Ford","Fusion");
            gmfactory.orderCar("Chevrolet","Corvette");
            gmfactory.orderCar("Buick","Lucerne");
            }
      }
```

Listing 8: The OrderCars class – our client application

In Listing 8, the **`orderCar()`** method (defined in the **`CarFactory`** base class) has parameters **`"Pontiac"`** and **`"GTO"`**. The code inside the **`orderCar()`** method creates a new car by calling the appropriate version of **`createCar()`**. In the **`"Pontiac","GTO"`** case, the appropriate **`createCar()`** version is the one defined in the **`GeneralMotorsFactory`** class. The **`GeneralMotorsFactory`** class encapsulates the conditionals that can change with new makes and models.

The **`Pontiac`** class assigns the appropriate values to the **`make`**, **`model`**, and **`manufacturer`** variables. Then the **`orderCar()`** method calls the **`buildCar()`**, **`testCar()`**, and **`shipCar()`** methods to fulfill the order. Listing 8 builds the remaining cars in a similar fashion using the factories in Listings 2, 3, and 4. Figure 3 shows the output of the car ordering application.

```
--- Fulfilling the order for a Pontiac GTO ---
Building the Pontiac GTO at the General Motors factory...
Testing the Pontiac GTO at the General Motors test track...
Shipping the Pontiac GTO at the Pontiac dealership...

--- Fulfilling the order for a Mercury Mark IV ---
Building the Mercury Mark IV at the Ford factory...
Testing the Mercury Mark IV at the Ford test track...
Shipping the Mercury Mark IV at the Mercury dealership...

--- Fulfilling the order for a Saturn Vue ---
Building the Saturn Vue at the General Motors factory...
Testing the Saturn Vue at the General Motors test track...
Shipping the Saturn Vue at the Saturn dealership...

--- Fulfilling the order for a Ford Fusion ---
Building the Ford Fusion at the Ford factory...
Testing the Ford Fusion at the Ford test track...
Shipping the Ford Fusion at the Ford dealership...

--- Fulfilling the order for a Chevrolet Corvette ---
Building the Chevrolet Corvette at the General Motors factory...
```

```
Testing the Chevrolet Corvette at the General Motors test track...
Shipping the Chevrolet Corvette at the Chevrolet dealership...

--- Fulfilling the order for a Buick Lucerne ---
Building the Buick Lucerne at the General Motors factory...
Testing the Buick Lucerne at the General Motors test track...
Shipping the Buick Lucerne at the Buick dealership...
```

Figure 3: The output of the car ordering application

You can use the Factory Method design pattern to create a software framework. The car ordering application (using the Factory Method) is a framework with a pre-written **`orderCar()`** method. To this basic framework, a developer adds concrete classes for new makes and models of cars.

A Simple Factory

The **Simple Factory**, a variation of the Factory Method, isn't one of the original 23 GoF design patterns. In fact, the Simple Factory isn't usually called a "pattern." Instead, the Simple Factory is called an "idiom." Whether you call it a "pattern" or an "idiom," developers make frequent use of the Simple Factory. You can use the Simple Factory when you don't need to let subclasses decide which class to instantiate.

Listings 9 and 10 demonstrate the use of the Simple Factory for the car ordering application. Starting from the code in Listing 1, you pull out any frequently-changing code, and encapsulate the code into a single class.

```
public class SimpleFactory
      {
      public Car createCar(String model)
            {
            Car car = null;
            if(model.equals("Lucerne"))
                  car = new Buick(model);
            else if(model.equals("Corvette"))
                  car = new Chevrolet(model);
            else if(model.equals("Fusion"))
                  car = new Ford(model);
            else if(model.equals("GTO"))
                  car = new Pontiac(model);
            else if(model.equals("Vue"))
                  car = new Saturn(model);
            return car;
            }
      }
```

Listing 9: The SimpleFactory class

```
public class OrderCars
      {
      SimpleFactory factory;

      public OrderCars(SimpleFactory factory)
            {
            this.factory = factory;
            }

      public Car orderCar(String model)
            {
```

```
            Car car = factory.createCar(model);
            car.buildCar();
            car.testCar();
            car.shipCar();
            return car;
            }
    }
```

Listing 10: The OrderCars class – our client application using the SimpleFactory

So what `else` is `new`?

In the end, neither the Factory Method nor the Simple Factory keeps you from constructing concrete objects. Somewhere, your code uses Java's **`new`** keyword and calls an honest-to-goodness constructor. But with the techniques described in this article, you delay the use of **`new`** until the execution reaches a subclass. The effect of this delay is to isolate the code that's subject to frequent change. When a company announces a new make of cars, you know exactly where to put the new code.

Resources

Design Patterns – Elements of Reusable Object-Oriented Software
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides
ISBN 0-201-63361-2

Head First Design Patterns
Eric & Elisabeth Freeman
ISBN 0-596-00712-4

Object-Oriented Software Construction
Bertrand Meyer
ISBN 0-13-629155-4

Data & Object Factory
http://www.dofactory.com/Patterns/Patterns.aspx

About the Authors

Barry Burd is a professor in the Department of Mathematics and Computer Science at Drew University in Madison, New Jersey. When he's not lecturing at Drew University, Dr. Burd leads training courses for professional programmers in business and industry. He has lectured at conferences in America, Europe, Australia and Asia. He is the author of several articles and books, including “Java 2 For Dummies” and “Eclipse For Dummies,” both published by Wiley.

Michael P. Redlich is a Senior Research Technician (formerly a Systems Analyst) at ExxonMobil Research & Engineering, Co. in Clinton, New Jersey with extensive experience in developing custom web and scientific laboratory applications. Mike also has experience as a Technical Support Engineer for Ai-Logix, Inc. where he developed computer telephony applications. He holds a Bachelor of Science in Computer Science from Rutgers University. In his spare time, Mike facilitates the ACGNJ Java Users Group and serves as ACGNJ Secretary. Mike has also co-written several articles for Java Boutique, and his computing experience includes computer security, relational database design and development, object-oriented design and analysis, C/C++, Java, Visual Basic, FORTRAN, Pascal, MATLAB, HTML, XML, ASP, VBScript, and JavaScript in both the PC and UNIX environments.