developer.*
The Independent Magazine for Software Professionals

Factory Chain: A Design Pattern for Factories with Generics

by Hugo Troche

Introduction

The recent Java 5 (a.k.a. Java 1.5) generics implementation opens both new problems and new opportunities in the implementation of the "Gang of Four" (GoF) Factory patterns [Gamma, et al 1995]. On the one hand, we want a class factory to determine the type it will instantiate at run time. On the other hand, when we use generics correctly the compiler has to be able to determine the type of any call at compile time; casting with generics is considered unsafe.

As we'll see, these two opposing facts make it cumbersome to implement classic factories with generics. After we explore the problems that arise when we try to use factories and generics together, we will look into a proposed solution: a new hybrid pattern I call Factory Chain. First, though, I will explain the basics of factories and generics when considered separately.

Factories

The Factory pattern is powerful. It provides us with a framework for instantiating classes dynamically at run time according to the logic of the program. Let's take a look at a simple example. (If you already know the Factory pattern, you may choose to skip this section.) Figure 1 shows the UML diagram for a factory that returns `Car` objects.

In Figure 1 we have the interface `Car`. The classes `Ford` and `Chevy` implement the interface `Car`. The factory decides at run time what type of `Car` to use.

Original author owns and reserves all future rights. Reprint only with written permission.

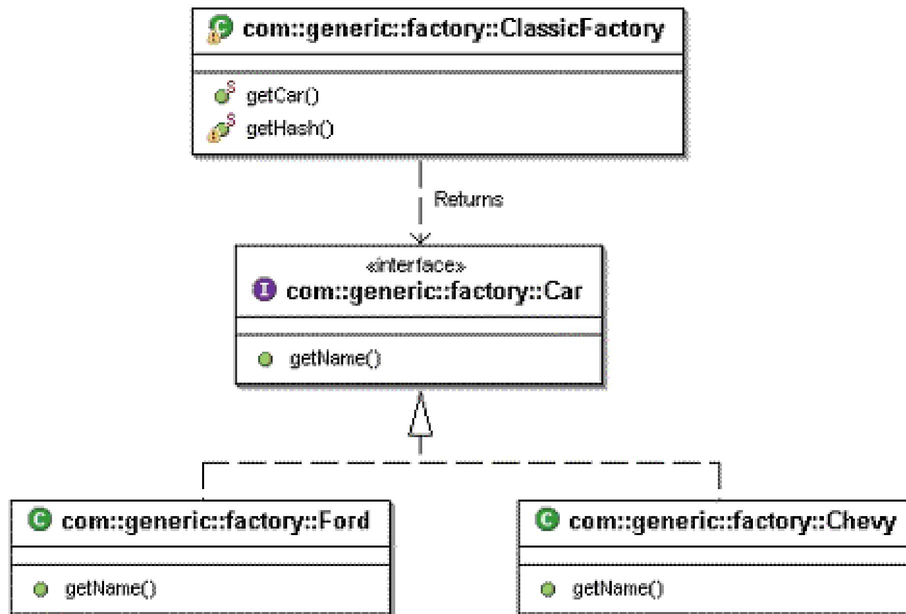

Figure 1. UML Diagram for a classic factory implementation.

To implement this UML, we need is a class that will return a `Car` type according to some logic executed at run time. So we create `ClassicFactory` to return `Car` types to clients. Listing 1 shows the code for all these classes.

```
package com.generic.factory;

public interface Car {
    public String getName();
    }

package com.generic.factory;

public class Ford implements Car {
    public String getName() {
        return "Ford";
    }

}

package com.generic.factory;

public class Chevy implements Car {

    public String getName() {
        return "Chevy";
    }

}
```

Code continues on next page…

Original author owns and reserves all future rights. Reprint only with written permission.

```
package com.generic.factory;
import java.util.*;


public class ClassicFactory {

    public static Car getCar(String carType) {
        try{
            HashMap map = ClassicFactory.getHash();
            //Here we solve the name the client uses to a class name
            String carClass = map.get(carType).toString();
            //With the class name we use reflection
            //and instantiate a new car
            Class clazz = Class.forName(carClass);
            return (Car) clazz.newInstance();
        } catch(Exception e) {e.printStackTrace();}
        return null;
    }

    //This method could as well load the hash from a properties file.
    //That way when you create a new type of car, like a Honda, all
    //you have to do is add it to the properties file.
    public static HashMap getHash() {
        HashMap map = new HashMap();
        map.put("Ford", "com.generic.factory.Ford");
        map.put("Chevy", "com.generic.factory.Chevy");
        return map;
    }
}
```

Listing 1. Classes to implement a factory design pattern.

As we can see in `ClassicFactory`, the type of car that `getCar()` returns is decided at run time. `ClassicFactory` could come in quite handy if you have a screen that shows the detail of a car. The user decides which car to show by selecting an item from a list. When the list sends the car type name to the factory the factory will return an object of the correct `Car` type to the user. Having a factory is nice for encapsulating in one place the knowledge of how to instantiate different classes, but we have to be smart about how we implement the inner code of the factory.

Since one of the main goals of a factory is to promote loose coupling, we want a solution that does not undermine that goal. We know, then, that we don't want to use lines and lines of `if-else` statements to decide the type. In that case both loose coupling and maintainability are reduced because `getCar()` has direct knowledge of all of the `Car` implementations and must be changed every time one is added or removed. So we use a hash (ideally loaded from a configuration file or some other data-driven source) and a type cast. As we'll soon see, the use of casting is what gets us into trouble with generics.

Original author owns and reserves all future rights. Reprint only with written permission.

Generics

The key to understanding generics[*] is to realize that the compiler has to be able to figure out the type of all objects at compile time. This means that any down casting (from parent type to child type) is not safe. In Java 5 the compiler will accept a non safe cast, yet it will throw out a warning. Let's take a look at a non safe cast:

```
Object obj = ….;
//The compiler will produce a warning for this cast.
Integer i = (Integer) obj;
```

On the other hand this is a safe cast:

```
Integer i = …;
//This is safe
Object obj = i;
```

So, a safe cast is a cast that you can do without having to use the class name enclosed in parenthesis to do the cast. Like the cast in the example above. The problem that generics address is how to perform the first type of casting safely. That is where the generics keywords and notations come into play.

The most obvious use of generics is with collection objects. For example, a `Vector` only takes `Object` instances as elements and therefore only returns `Object` instances. You need to cast the object that the `Vector` returns to whatever type it was originally. This example shows how to use a `Vector` without generics:

```
//Create and load the Vector
Vector v = new Vector();
v.add(new Integer(1000));

//Get the Integer out of the vector
//Here we need to cast the Object to Integer,
//because get() returns a Object instance
Integer vi = (Integer) v.get(0);
```

[*] This is only a short introduction to generics. It will show enough to understand the rest of the article. You can find a more detailed and complete discussion of generics in [Bracha 2004].

Original author owns and reserves all future rights. Reprint only with written permission.

Using generics we can make the `get()` operation safe:

```
//This tells the compiler that Vector will hold Integer types.
Vector<Integer> v = new Vector<Integer>();

//Now we can only add Integer objects to the Vector.
//With any other type we will get a compile error
v.add(new Integer(1000));

//The call get() will return an Integer type
Integer vi = v.get(0):
```

The syntax to use generics is usually `<ClassName>`. However, when defining generics it is not necessary to know the class name in advance. We can use formal type parameters. For example, here we have a class with methods that return and take a generic type:

```
public class CustomVector <T> {

    public void add(T element) {//Code to add an element}

    public T get(int index) {//Code to return an element}

}
```

The formal type parameter tells the compiler what type of objects can be used in the class `CustomVector` for the methods `add()` and `get()`. Therefore, if we instantiate a `CustomVector` object with a type `java.lang.String`, the method `add()` will only take `String` objects and the method `get()` will return `String` types.

```
//This CustomVector wil only take String objects
CustomVector<String> cv = new CustomVector<String>();
```

In the example above we made the class `CustomVector` generic. We can make methods of a class generic by placing the generic formal type parameter enclosed in `<>`.

```
//This is a generic method that only accepts type
//that is a subclass of String
public static <T extends String>  T cropString(T t) {
//…..
return T;
}
```


©2005 Hugo Troche
Original author owns and reserves all future rights. Reprint only with written permission.

The tag `<T extends String>` tells the compiler that this is a generic method. It also specifies that the formal parameter type should be a subclass of `String`. `T` specifies the type that the method should return. There are many wildcard keywords and ways of using generics. For further information please refer to [Bracha, 2004].

Factories and Generics

Generics change the basic implementation dynamic of factories. With factories the system defines the type of an object at run time. With generics the compiler has to be able to figure out the type of all objects at compile time. How can we have the flexibility of factories with the safety of generics?

When using generics the code is considered "safe" because there can't be any casting exceptions. The compiler already takes care of checking if you have any invalid casts. In a factory we usually specify to what type to cast in a flat text file. Then the system reads that information and casts objects at run time. This is inherently unsafe because if you even mistype a class name in the flat file, you will get a runtime exception.

As you will see in the sections ahead, implementing a factory with generics is more cumbersome than a classic factory. However, for those time when it is critical to minimize the possibility of runtime exceptions and errors you will have to implement factories with generics. The benefit, though, is that generics make the code safer because the compiler catches type mismatches. If the code is not critical, you might still want to implement factories with generics just to minimize your potential unit testing burden.

Generic Factory Problems

The problem with implementing factories with generics is that you cannot safely instantiate objects with parameters determined at run time. This is the reason why the class `java.lang.Class` is generic in Java 5. The most important line of Listing 1 is not safe with generics:

```
return (Car) clazz.newInstance();
```


Original author owns and reserves all future rights. Reprint only with written permission.

The problem is that the method `newInstance()` returns a type `<?>`. This means that the method returns a generic type. The actual type depends on the type that we use to instantiate the generic `Class` object. So if we wanted to safely instantiate a `Ford` object using `newInstance()` we would have to use the following code:

```
return Ford.class.newInstance();
```

Of course, this mechanism is not useful for a factory. You don't want to refer directly in your factory code to `Ford.class` and `Chevy.class`, using `if-else` statements to decide whether to return a `Ford` or `Chevy` object. That is exactly what we are trying to avoid.

The code below illustrates our dilemma with Factory and generics.

```
//We start by making it return an object that extends (in this
//case implements) Car

public static <T extends Car> T getCar(String carType) {
    try{
        HashMap map = ClassicFactory.getHash();

        //Here we solve the name the client uses
        //to a class name
        String carClass = map.get(carType).toString();

        //With the class name we use reflection and
        //instantiate a new car. The problem here is that
        //Class.forName(String) returns a type<?>,
        //not a type <T>. This code will not compile.
        Class<T> clazz = Class.forName(carClass);

        return clazz.newInstance();
    } catch(Exception e) {e.printStackTrace();}
    return null;
}
```

We know that this code will not compile because of the return type of `Class.forName()`. So what do we do?

One option is to cast the return of `Class.forName()` to `Class<T>`.

```
Class<T> clazz = (Class<T>) Class.forName(carClass);
```

The problem is that we lose the minute we bring in the word cast. Casting in this situation is not safe for Java 5 because it opens the possibility to cast exceptions at run time. Although you will be careful to always pass the correct types, other developers using your code might

Original author owns and reserves all future rights. Reprint only with written permission.

not. That is the power of generics: you know there will be no run time cast exception if there are no compile warnings.

This means that there is no safe way of instantiating objects by retrieving the class name at run time. So are we stuck with long switch statements like this?

```
if(classType.equals("Ford"))
    return Ford.class.newInstance();
if(classType.equals("Chevy"))
    return Chevy.class.newInstance();
//And we keep going for all the classes implementing Car
```

Let's see if we can use generic parameters to help us find a safe way to implement Factory with generics. In the next section I will propose the merging of two common design patterns to avoid these nasty long `if-else` sequences.

The Solution: Factory Chain

The solution to the generic factory problem is to combine the Factory and Chain of Responsibility design patterns, both of which are part of the now standard "Gang of Four" pattern collection [Gamma et al, 1995].[*]

Chain of Responsibility works by creating a linked list of objects. Each object in the linked list knows how to perform a particular part of a major operation. The combination of all of the operations in all of the objects in the chain amount to a larger operation. For example, if you need to parse an HTML document, you can use Chain of Responsibility such that each element knows how to parse one HTML tag.

Our problem is that we need a way to specify which class to instantiate—but we don't want to have switch statements to decide what the correct class is. The solution is to give the knowledge to each class of the class that is next in the "chain." Then we can iterate through all the classes like they were a linked list. Figure 2 shows the UML diagram for a set of classes that implement this concept.

[*] C# and C++ also have their implementation of generics. Using generics with C# is very similar to using generics with Java. With C++ you have to use templates. The problem of creating safe factories with C#, C++ and Java is similar. The design that I present in this article can also be implemented in those languages.

Original author owns and reserves all future rights. Reprint only with written permission.

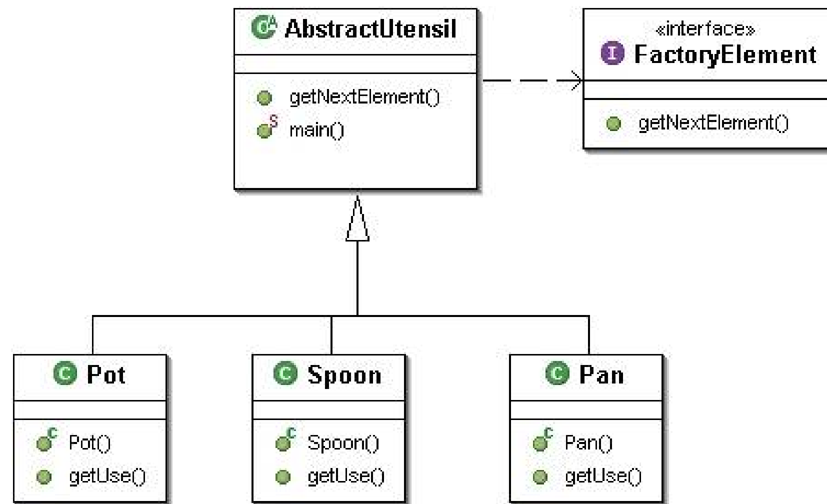

Figure 2. UML Diagram of the Factory Chain pattern

As you can see in Figure 2, the factory object itself does not exist. All we have are factory elements that know the next element in the Factory Chain. In this new example, the classes that we want to construct with the factory are `Pot`, `Spoon` and `Pan`. The strategy is simple: we create an abstract super class for these classes called `AbstractUtensil`. This is the type that our Factory Chain will return. `AbstractUtensil` defines the abstract method `getUse()` (all the sub classes have to implement it).

Here is where we use the Chain of Responsibility design pattern. Each `AbstractUtensil` (`Pot`, `Spoon`, and `Pan`) knows what the next element in the factory is. So, to find an element in the factory, all we need is to know the first element of the Factory Chain and then iterate the chain.

Take a look at the method `main()` in `AbstractUtensil` in Listing 2 (below) for an example of how to do this. Note that I implement `getNextElement` in `AbstractUtensil` so the subclasses of `AbstractUtensil` don't have to. All the subclasses need to specify is the next element. I usually do this in the constructor of the subclasses.

The method `getNextElement()` is in actuality the factory method. This is because this method returns the elements in the factory. Then we have to make sure that each element has a reference to the next element in the factory, giving us a linked list of classes. We have found a type safe way to specify all the elements of a factory without long `if-else` statements. Listing 2 shows the code for these classes.

Original author owns and reserves all future rights. Reprint only with written permission.

```
package com.generic.factory;

//This is a generic interface that you can use to create any
//Factory Chain.
public interface FactoryElement <T> {

    //This method will always return a sub class
    //of the generic definition
    public Class<? extends T> getNextElement();

}

package com.generic.factory;

//This is the class that defines the factory
//and the generic utensil.
//Look at how it implements FactoryElement
//and passed Abstract utensilto the generic parameter.
public abstract class AbstractUtensil implements
FactoryElement<AbstractUtensil> {

    //The member variable that each child of this class should
    //overwrite with the next element in the factory
    protected Class<? extends AbstractUtensil> nextClass = null;

    //This will return the next element in the factory
    public Class<? extends AbstractUtensil> getNextElement() {
        return  nextClass;
    }

    //The method that the children have to implement
    public abstract String getUse();

    //Here we test the factory
    public static void main(String[] args) {
        try{
            //Fist we create the beginning element
            AbstractUtensil pot = new Pot();
            //From that we can get the next element
            Class<? extends AbstractUtensil> clazz = pot.getNextElement();
            while(clazz != null) {
                //We loop through all the elements of the factory
                System.out.println(clazz.getName());
                AbstractUtensil next = clazz.newInstance();
                clazz = next.getNextElement();
            }
        } catch(Exception e) {e.printStackTrace();}
    }
}
```

Code continues on next page…

Original author owns and reserves all future rights. Reprint only with written permission.

```
package com.generic.factory;

public class Pot extends AbstractUtensil {

    public Pot() {
        super.nextClass = Pan.class;
    }

    public String getUse() {
        return "boil";
    }
}

package com.generic.factory;

public class Pan extends AbstractUtensil {

    public Pan() {
        super.nextClass = Spoon.class;
    }

    public String getUse() {
        return "fry";
    }

}

package com.generic.factory;

public class Spoon extends AbstractUtensil {

    public Spoon() {
        super.nextClass = null;
    }

    public String getUse() {
        return "stir";
    }

}
```

Listing 2. Classes to implement the Factory Chain

We can iterate and instantiate any type of `AbstractUtensil` without having a long series of switch statements in the factory. Note how `Pot` points the member variable `nextClass` to the class of `Pan`. `Pan` points to `Spoon`. `Spoon` makes the variable `null` indicating the end of the chain. It is better to link the Class objects of the elements than to link the objects themselves because an element may need specialized construction. For example, the constructors of the element could have many parameters. It is easier to delegate the assignment of loading that constructor to the object using the elements of the chain. The elements might not have that information readily available.

Original author owns and reserves all future rights. Reprint only with written permission.

It is not appropriate to put that logic into each element. That is the job of the client using the Factory Chain.

The Factory Chain pattern solves the problem of having to do a cast to dynamically instantiate classes. Instead of one big factory class that possesses knowledge of the classes of all of its elements, we pushed that knowledge down, distributing it to the elements.

Conclusion

The advantages of using Factory Chain are:

- When implementing a Factory using generics since you don't have those long, hard to maintain switch statements that the use of generics would otherwise force you into.
- The Factory Chain is type safe. You will not have a compile time warning or `ClassCastException`s.

The disadvantages of the using Factory Chain are:

- You have to keep track of where the chain begins and ends manually.
- The elements have less encapsulation because they know about the class of the next element in the chain.

You should use the Factory Chain only when you want a type safe factory. If you don't mind the warnings when compiling and the threat of run time exceptions, it is better to use the classical factory pattern. On the other hand, having type safe code reduces the number unit tests necessary for a factory because you don't need to test the code for run time cast exceptions and errors. When you use Factory Chain you simply know that the factory will not throw a cast run time exception.

###

Hugo Troche is a software engineer originally from Asuncion, Paraguay now living in Auburn, Alabama, USA. He earned his undergraduate degree in Computer Science from Auburn University (Magna Cum Laude) in 2002 and finished his Master's degree in 2004. Hugo a software consultant and contractor specializing in web services and n-tier business systems. Hugo can be reached at *htroche@myway.com*.

Original author owns and reserves all future rights. Reprint only with written permission.

References

[Gamma, et al, 1995] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John. Design Patterns. Boston: Addison-Wesley, 1995.

[Bracha, 2004] Bracha, Gilad. “Generics in the Java Programming Language”. `http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`. Sun Microsystems, 2004.

Original author owns and reserves all future rights. Reprint only with written permission.