JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2010**

Vol. 9, No. 1, January-February 2010

A Modern, Compact Implementation of the Parameterized Factory Design Pattern

Harold Fortuin, Principal, Fortuitous Consulting Services, Inc.

Abstract

I propose a modern OO language update of the Parameterized Factory Creational Pattern [Gamma95]. It is compact since it folds the Factory method into an abstract base class, and imposes a package structure on its concrete subclasses. The design is demonstrated in Java, but is also applicable to C# and other modern object-oriented languages.

1 INTRODUCTION

Suppose you are the lead software designer-developer at Consolidated Morsels, which is building a system to control the manufacture of a wide variety of treats including hard candy, filled chocolates, and cookies. Since Consolidated is the parent company for the recently independent Hershzweil, Swietinuff, and Venus companies, many manufacturing details for each kind of treat will vary.

Certain specifics about each hard candy, filled chocolate, and cookie are defined via configuration. However, you can reasonably anticipate that as yet unknown kinds of treats, from existing and future subsidiaries, will be manufactured. Furthermore, you would like to impose order not only on the class implementations, but also upon the Java package structure they inhabit. Since the quantities and kinds of candy to create will vary amongst locations, season of the year, and general economic conditions, a Java client object should only call a generic series of methods for whichever treat.

2 BASE CLASS AND SUBCLASSES

Since the differences in processes between each kind of candy by each subsidiary are quite large, you design a single abstract base class containing method signatures for each top-level processing stage common across the treats:

1. Setup equipment
2. Prepare the mix
3. Mix
4. Shape
5. Package

The 6th step, Cleanup, however, can be handled in a common way in the base class.

Therefore, by now you have most of the elements visible in the class hierarchy in Figure 1:

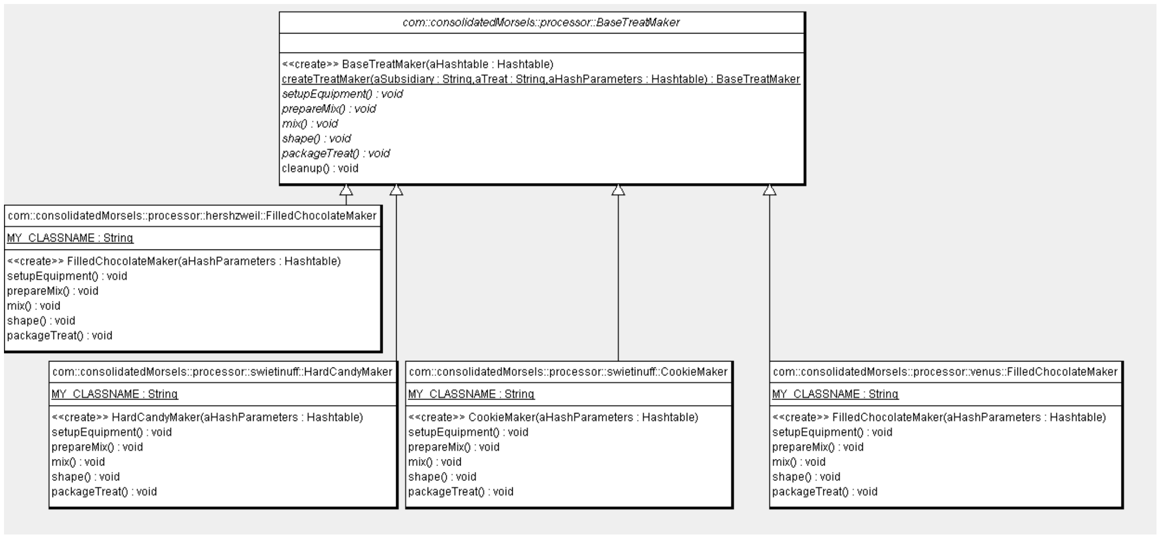
com::consolidatedMorsels::processor::BaseTreatMaker
<<create>> BaseTreatMaker(aHashtable : Hashtable)
createTreatMaker(aSubsidiary : String,aTreat : String,aHashParameters : Hashtable) : BaseTreatMaker
setupEquipment() : void
prepareMix() : void
mix() : void
shape() : void
packageTreat() : void
cleanup() : void
com::consolidatedMorsels::processor::hershzweil::FilledChocolateMaker
MY_CLASSNAME : String
<<create>> FilledChocolateMaker(aHashParameters : Hashtable)
setupEquipment() : void
prepareMix() : void
mix() : void
shape() : void
packageTreat() : void
com::consolidatedMorsels::processor::swietinuff::HardCandyMaker
MY_CLASSNAME : String
<<create>> HardCandyMaker(aHashParameters : Hashtable)
setupEquipment() : void
prepareMix() : void
mix() : void
shape() : void
packageTreat() : void
com::consolidatedMorsels::processor::swietinuff::CookieMaker
MY_CLASSNAME : String
<<create>> CookieMaker(aHashParameters : Hashtable)
setupEquipment() : void
prepareMix() : void
mix() : void
shape() : void
packageTreat() : void
com::consolidatedMorsels::processor::venus::FilledChocolateMaker
MY_CLASSNAME : String
<<create>> FilledChocolateMaker(aHashParameters : Hashtable)
setupEquipment() : void
prepareMix() : void
mix() : void
shape() : void
packageTreat() : void

Figure 1: UML class diagram of BaseTreatMaker and subclasses

3 THE FACTORY METHOD

Next, you need a Factory method of some kind to create the concrete types, which should depend on parameters handed to it. You have reviewed the Creational patterns [Gamma95], but believe there may be a more elegant solution taking advantage of Java language features.

Of course, you know that in contrast to say biological inheritance, any OO superclass lacks the complexity of state of any of its children – and that the Factory by design knows nearly nothing of what it constructs [Gamma95]. Remembering that static methods in Java exist within the class but are not subject to inheritance, you realize that the factory method should in fact reside in the BaseTreatMaker in static (class) scope, and could thereby return its own type, rather than, say, the overly generic java.lang.Object. Thus we have a compact implementation: no separate Factory class required outside the implementation class hierarchy. And by returning the specific type BaseTreatMaker, you impose some useful discipline over the results of the Factory method, which should especially aid junior developers.

The Factory method signature

You impose a package structure through 1..n String arguments, which are appended to one another to build that structure. Here, they are appended to the package of the BaseTreatMaker, in the order of Subsidiary, then Treat, using the Java Reflection API. Also in our case, the class naming logic ensures that the name of the Maker subclass is constructed from the name of the kind of treat – HardCandyMaker, etc.

Then to facilitate the passing of an arbitrary number of parameters to each subclass, with minimal type restrictions on them (in this case with uniquely keyed parameter objects) you add the Hashtable parameter. You might also pass some other Collection framework object like an ArrayList per your real-world needs, but note that in any case, only the implementing subclass and the client of BaseTreatMaker need know how many such parameters it should receive and of what type(s) – the base class by design should not know.

So our factory method signature becomes

```
static public BaseTreatMaker
   createTreatMaker(String aSubsidiary,
                    String aTreat,
                    Hashtable aHashParams)
```

Passing Parameters to the Constructor

After building the canonical form class name using the String arguments, we use the `Class` class methods that allow us to instantiate not merely a no-argument constructor, but a constructor with whatever parameters suit our implementation – here, just the Hashtable. If any of our concrete classes do not share the number and type of arguments that our factory code expects, the runtime `java.lang.NoSuchMethodException` will be thrown. (see the Swietinuff HardCandyMaker for an example you can enable).

The last step is to cast the result from the Reflection API getConstructor method to conform to our return type BaseTreatMaker.

Here is the factory method implementation from BaseTreatMaker, deliberately omitting the catch blocks

(complete code available in a zip archive from http://www.fortuitous-consulting.biz):

```
static public BaseTreatMaker
   createTreatMaker(String aSubsidiary,
                    String aTreat,
                    Hashtable aHashParams)
{
   BaseTreatMaker aTreatMaker = null;
```

```
    Package pkgBaseTreatMaker = null;
    Object objBaseTreatMaker = null;

    Class[] parameterTypes = null;
    Class classDefinition = null;
    Object[] parameters = null;
    Object objTreatMaker = null;

    String strPkg = null;
    String strClassTreatMakerName = null;

    try {
         // Reflection API
         // start from the BaseTreatMaker package
         pkgBaseTreatMaker = BaseTreatMaker.class.getPackage();
         strPkg = pkgBaseTreatMaker.getName();

         StringBuffer sbClassName = new StringBuffer(strPkg);

         sbClassName.append(".");
         sbClassName.append(aSubsidiary);
         sbClassName.append(".");
         sbClassName.append(aTreat);
         sbClassName.append("Maker");

         strClassTreatMakerName = sbClassName.toString();

         System.out.println("Classname = " +
         strClassTreatMakerName);

         // more Reflection API
         parameterTypes = new Class[] { Hashtable.class};
         parameters = new Object[] { aHashParameters };

         classDefinition                                          =
Class.forName(strClassTreatMakerName);
         classDefinition.getConstructor(parameterTypes);
objTreatMaker                                                     =
classDefinition.getConstructor(parameterTypes).
newInstance(aHashParameters);

         aTreatMaker = (BaseTreatMaker)objTreatMaker;
    }

    // real code requires catch block(s)
    return aTreatMaker;
} // end createTreatMaker(...)
```

You'll also see that the example code provides simple (and identical) implementations of some of the subclasses from the design – enough for the reader to flesh out real-world implementations.

But to illustrate how a client could call the factory method, here's the main method from the BaseTreatMakerClient, for our purposes in the same package as BaseTreatMaker:

```
    public static void main(String[] args) {

            BaseTreatMaker theBaseTreatMaker = null;
            System.out.println("------BEGIN ALL-----");

            // -- prep object to put on the argument Hashtable -
            Hashtable aHash = new Hashtable();
            Object obj = new Object();
            System.out.println("Ref of obj:" + obj.toString() );

/* creates a compile warning since the Hashtable type is not
parameterized */
            aHash.put("oneObject", obj);

            // ------- MAKE & call methods on each class ---------
-----

            /* Since neither company or Treat is unique,
               create NameValuePair objects for convenience */
            NameValuePair[] nvsCompanyToTreat =
                    {new
NameValuePair("hershzweil","FilledChocolate"),
                     new NameValuePair("swietinuff","HardCandy"),
                     new NameValuePair("swietinuff","Cookie"),
                     new NameValuePair("venus","FilledChocolate")};

            int nvsLength = nvsCompanyToTreat.length;
            int i = -1;

            /* Iterate over the set of pairings above,
               constructing the corresponding class of each */

            NameValuePair nvPair = null;

            String strCompany = null;
            String strTreat = null;

            for (i = 0; i < nvsLength; i++) {
                    System.out.println("------BEGIN     CREATE     A
TREATMAKER-----");
                    nvPair = nvsCompanyToTreat[i];
                    strCompany = nvPair.name;
                    strTreat = nvPair.value;
```

```
                theBaseTreatMaker =
                     BaseTreatMaker.createTreatMaker(strCompany,
                                                      strTreat,
                                                      aHash);

System.out.println("BaseTreatMaker toString(): " +
      theBaseTreatMaker.toString() );

                System.out.println("------END CREATE A
TREATMAKER-----");
                System.out.println();
        } // end for

        System.out.println("------END ALL-----");
   } /// end main()
```

Here's the NameValuePair class, which was left in the BaseTreatMakerClient.java since it is a very simple class only referenced by BaseTreatMakerClient:

```
   class NameValuePair {
// deliberately accessible
String name;
String value;

NameValuePair(String aName, String aValue) {
   name = aName;
   value = aValue;
} // end NameValuePair constructor
} // end class NameValuePair
```

4 IMPLEMENTATION GUIDELINES

In describing the Factory pattern, [Gamma95] mentions the option of subclassing the Factory class.

However, I recommend against that – it can too easily lead to an unnecececessarily complex class creation hierarchy. Alternatives are to refactor using just the one Factory method, or perhaps to create some additional, independent Factory for an additional, independent class hierarchy should integration be a poor design choice.

Also note that the base Class need not be abstract, though it typically has been in my implementations.

The pattern could also be implemented of course quite naturally in other object-oriented languages besides Java, especially in those including a package-like concept such as C# namespaces [Obasanjo07].

REFERENCES

[Gamma95] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design patterns: Elements of reusable object-oriented systems*, Addison-Wesley, 1995.

[Obasanjo07] Dare Obasanjo: “A Comparison of Microsoft’s C# Programming Language to Sun Microsystem's Java Programming Language”, http://www.25hoursaday.com/CsharpVsJava.html#namespace, 2001/2007

About the author

Author Harold Fortuin is Principal Consultant at Fortuitous Consulting Services, Inc. He has over 15 years of experience in object-oriented design and development for a variety of corporate clients, from Fortune 500 to startups, including manufacturing, educational, financial, and insurance companies. He is also an experienced instructor in Java, J2EE, OO design, and other topics. Contact: hftechnology@yahoo.com