ABOUT ROHIT JOSHI

Rohit Joshi is a Senior Software Engineer from India. He is a Sun Certified Java Programmer and has worked on projects related to different domains. His expertise is in Core Java and J2EE technologies, but he also has good experience with front-end technologies like Javascript, JQuery, HTML5, and JQWidgets.

Abstract Factory Design Pattern

Posted by: Rohit Joshi in Core Java, Java September 30th, 2015

This article is part of our Academy Course titled Java Design Patterns.

In this course you will delve into a vast number of Design Patterns and see how those are implemented and utilized in Java. You will understand the reasons why patterns are so important and learn when and how to apply each one of them. Check it out here!

Table Of Contents

1. Introduction

In the previous lesson, we had developed an application for a product company to parse XMLs and display results to them. We did this by creating different parsers for the different types of communication between the company and its clients. We used the Factory Method Design Pattern to solve their problem.

The application is working fine for them. But now the clients don't want to follow the company's specifics XML rules. The clients want to use their own XML rules to communicate with the product company. This means that for every client, the company should have client specific XML parsers. For example, for the NY client there should be four specific types of XML parsers, i.e. NYErrorXMLParser, NYFeedbackXML, NYOrderXMLParser, NYResponseXMLParser, and four different parsers for the TW client.

The company has asked you to change the application according to the new requirement. To develop the parser application we have used the Factory Method Design Pattern in which the exact object to use is decided by the subclasses according to the type of parser. Now, to implement this new requirement, we will use a factory of factories i.e. an Abstract Factory.

This time we need parsers according to client specific XMLs, so we will create different factories for different clients which will provide us the client specific XML to parse. We will do this by creating an Abstract Factory and then implement the factory to provide client specific XML factory. Then we will use that factory to get the desired client specific XML parser object.

Abstract Factory is the design pattern of our choice and before implementing it to solve our problem, lets us know more about it.

2. What is the Abstract Factory Design Pattern

The Abstract Factory (A.K.A. Kit) is a design pattern which provides an interface for creating families of related or dependent objects without specifying their concrete classes. The Abstract Factory pattern takes the concept of the Factory Method Pattern to the next level. An abstract factory is a class that provides an interface to produce a family of objects. In Java, it can be implemented using an interface or an abstract class.

The Abstract Factory pattern is useful when a client object wants to create an instance of one of a suite of related, dependent classes without having to know which specific concrete class is to be instantiated. Different concrete factories implement the abstract factory interface. Client objects make use of these concrete factories to create objects and, therefore, do not need to know which concrete class is actually instantiated.

The abstract factory is useful for plugging in a different group of objects to alter the behavior of the system. For each group or family, a concrete factory is implemented that manages the creation of the objects and the inter-dependencies and consistency requirements between them. Each concrete factory implements the interface of the abstract factory

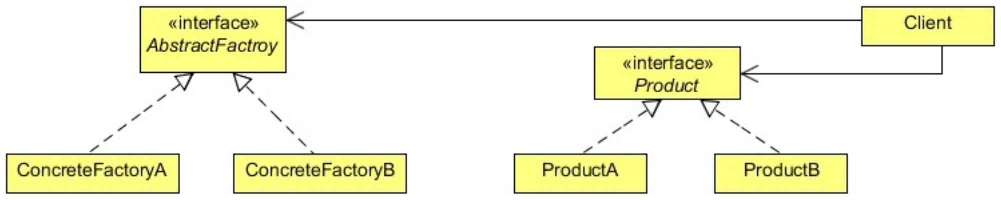

Figure 1

AbstractFactory

- Declares an interface for operations that create abstract product objects.

ConcreteFactory

- Implements the operations to create concrete product objects.

AbstractProduct

- Declares an interface for a type of product object.

ConcreteProduct

- Defines a product object to be created by the corresponding concrete factory.
- Implements the AbstractProduct interface.

Client

- Uses only interfaces declared by AbstractFactory and AbstractProduct classes.

3. Implementing the Abstract Factory Design Pattern

To implement the Abstract Factory Design Pattern will we first create an interface that will be implemented by all the concrete factories.

```java
package com.javacodegeeks.patterns.abstractfactorypattern;

public interface AbstractParserFactory {

    public XMLParser getParserInstance(String parserType);
}
```

The above interface is implemented by the client specific concrete factories which will provide the XML parser object to the client object. The

```
getParserInstance
```

method takes the

```
parserType
```

as an argument which is used to get the message specific (error parser, order parser etc) parser object.

The two client specific concrete parser factories are:

```java
package com.javacodegeeks.patterns.abstractfactorypattern;
```

```java

public class NYParserFactory implements AbstractParserFactory {

    @Override
    public XMLParser getParserInstance(String parserType) {

        switch(parserType){
            case "NYERROR": return new NYErrorXMLParser();
            case "NYFEEDBACK": return new NYFeedbackXMLParser ();
            case "NYORDER": return new NYOrderXMLParser();
            case "NYRESPONSE": return new NYResponseXMLParser();
        }

        return null;
    }

}
```

```java
package com.javacodegeeks.patterns.abstractfactorypattern;

public class TWParserFactory implements AbstractParserFactory {

    @Override
    public XMLParser getParserInstance(String parserType) {

        switch(parserType){
            case "TWERROR": return new TWErrorXMLParser();
            case "TWFEEDBACK": return new TWFeedbackXMLParser ();
            case "TWORDER": return new TWOrderXMLParser();
            case "TWRESPONSE": return new TWResponseXMLParser();
        }

        return null;
    }

}
```

The above two factories implement the

```
AbstractParserFactory
```

interface and overrides the

```
getParserInstance
```

method. It returns the client specific parser object, according to the parser type requested in the argument.

```java
package com.javacodegeeks.patterns.abstractfactorypattern;

public interface XMLParser {

    public String parse();

}
```

The above interface is implemented by the concrete parser classes to parse the XMLs and returns the string message.

There are two clients and four different type of messages exchange between the company and its client. So, there should be six different types of concrete XML parsers that are specific to the client.

```java
package com.javacodegeeks.patterns.abstractfactorypattern;

public class NYErrorXMLParser implements XMLParser{

    @Override
    public String parse() {
        System.out.println("NY Parsing error XML...");
        return "NY Error XML Message";
    }

}
```

```java
package com.javacodegeeks.patterns.abstractfactorypattern;

public class NYFeedbackXMLParser implements XMLParser{

    @Override
    public String parse() {
        System.out.println("NY Parsing feedback XML...");
        return "NY Feedback XML Message";
    }

}
```

```java
package com.javacodegeeks.patterns.abstractfactorypattern;

public class NYOrderXMLParser implements XMLParser{

    @Override
    public String parse() {
        System.out.println("NY Parsing order XML...");
        return "NY Order XML Message";
    }

}
```

```java
package com.javacodegeeks.patterns.abstractfactorypattern;

public class NYResponseXMLParser implements XMLParser{

    @Override
    public String parse() {
        System.out.println("NY Parsing response XML...");
        return "NY Response XML Message";
    }

}
```

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public class TWErrorXMLParser implements XMLParser{

    @Override
    public String parse() {
        System.out.println("TW Parsing error XML...");
        return "TW Error XML Message";
    }

}
```

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public class TWFeedbackXMLParser implements XMLParser{

    @Override
    public String parse() {
        System.out.println("TW Parsing feedback XML...");
        return "TW Feedback XML Message";
    }

}
```

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public class TWOrderXMLParser implements XMLParser{

    @Override
    public String parse() {
        System.out.println("TW Parsing order XML...");
        return "TW Order XML Message";
    }

}
```

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public class TWResponseXMLParser implements XMLParser{

    @Override
    public String parse() {
        System.out.println("TW Parsing response XML...");
        return "TW Response XML Message";
    }

}
```

To avoid a dependency between the client code and the factories, optionally we implemented a factory-producer which has a static method and is responsible to provide a desired factory object to the client object.

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public final class ParserFactoryProducer {

    private ParserFactoryProducer(){
        throw new AssertionError();
    }

    public static AbstractParserFactory getFactory(String factoryType){

        switch(factoryType)
        {
            case "NYFactory": return new NYParserFactory();
            case "TWFactory": return new TWParserFactory();
        }

        return null;
    }

}
```

Now, let's test the code.

```
package com.javacodegeeks.patterns.abstractfactorypattern;

public class TestAbstractFactoryPattern {

    public static void main(String[] args) {

        AbstractParserFactory parserFactory = ParserFactoryProducer.getFactory("NYFactory");
        XMLParser parser = parserFactory.getParserInstance("NYORDER");
        String msg="";
        msg = parser.parse();
        System.out.println(msg);

        System.out.println("************************************");

        parserFactory = ParserFactoryProducer.getFactory("TWFactory");
        parser = parserFactory.getParserInstance("TWFEEDBACK");
        msg = parser.parse();
        System.out.println(msg);
    }
}
```

The above code will result to the following output:

```
NY Parsing order XML...
NY Order XML Message
************************************
TW Parsing feedback XML...
TW Feedback XML Message
```

In the above class, we first got the NY factory from the factory producer, and then the Order XML parser from the NY parser factory. Then, we called the

```
parse
```

method on the parser object and displayed the return message. We did same for the TW client as clearly shown in the output.

4. When to use the Abstract Factory Design Pattern

Use the Abstract Factory pattern when

- A system should be independent of how its products are created, composed, and represented.
- A system should be configured with one of multiple families of products.
- A family of related product objects is designed to be used together, and you need to enforce this constraint.
- You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

5. Abstract Factory Pattern in JDK

- `java.util.Calendar#getInstance()`
- `java.util.Arrays#asList()`
- `java.util.ResourceBundle#getBundle()`
- `java.sql.DriverManager#getConnection()`
- `java.sql.Connection#createStatement()`
- `java.sql.Statement#executeQuery()`
- `java.text.NumberFormat#getInstance()`
- `javax.xml.transform.TransformerFactory#newInstance()`

6. Download the Source Code

This was a lesson on the Abstract Factory Design Pattern. You may download the source code here: **AbstractFactoryPattern-Project**

Tagged with: DESIGN PATTERNS