Design Patterns in real life: Abstract Factory

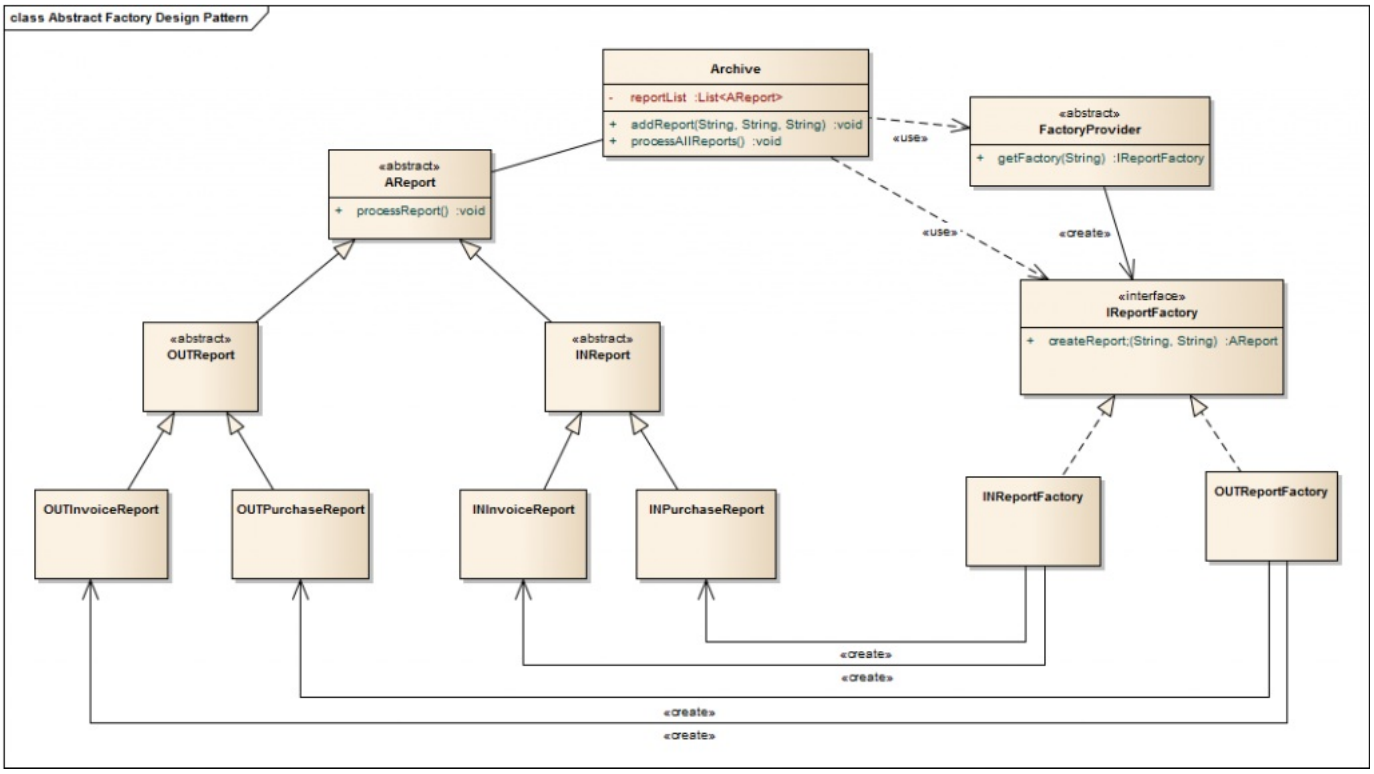
davismol.net/2015/03/18/design-patterns-in-real-life-abstract-factory

The most important thing when we talk about **design patterns** is to recognize when it can be useful to apply them to design something we have to develop. In this post we see a possible real case application of the **Abstract Factory** design pattern.
Suppose we have a system that has to process reports that can be of two categories: reports related to transactions in INPUT and reports related to transactions in OUTPUT. For each category there may be different types of reports, such as invoices reports, report related to purchases, etc.. Each report, depending on the category and type, has its own specific processing mode. The reports come in the form of a list of strings that represent the names of the reports themselves, read from a particular folder in the file system. In the name itself reports have indicated their category and their type. Suppose the names of the reports are in the following format: ##name.txt
So, for example:
IN_INV_001.txt -> indicates a report of INPUT category and "invoice" type
OUT_PUR_001.txt -> indicates a report of OUTPUT category and "purchase" type

Let's see how to use the Abstract Factory pattern to design this kind of system, in order to make it **flexible** and **scalable**.
The objects we need are:

- An interface that defines the method for creating reports, that must be implemented by the concrete factories (IReportFactory)
- The factory implementations for the two report categories (INReportFactory and OUTReportFactory)
- The abstraction of our report objects (AREPORT)
- The abstraction for reports of each category (AINReport and AOUTReport)
- The concrete implementations of the reports for each category and each type (INInvoiceReport, INPurchaseReport, OUTInvoiceReport, OUTPurchaseReport)
- The "client" class that uses the reports and that needs to instantiate them(Archive)
- A factory provider, to decouple the factories instantiation from the client object (FactoryProvider)

As a result of this analysis we can produce the following UML Class Diagram:



We can now implement in Java the structures that we have identified:

We start from the report objects abstraction, where we define a String member that contains the name of the report file. We also define a constructor that initializes this member, which will be invoked by the concrete subclasses constructors and a method which defines the processing operations common to all reports, regardless of category or type.

```
public abstract class AReport {

    protected String name;

    protected AReport(String name) {
        this.name = name;
    }

    public void processReport() {
        System.out.println("Processing report: " + this.name);
    };
}
```

Now let’s define the abstractions for the two categories of reports, INPUT and OUTPUT, which inherit from the previous one and will add in the override of the processReport() method any common operations at the category level.

```
public abstract class AINReport extends AReport {

   protected AINReport(String name) {
      super(name);
   }

   @Override
   public void processReport() {

      super.processReport();
      System.out.println("Performing IN Reports common stuff");

   }
}
```

```
public abstract class AOUTReport extends AReport {

   protected AOUTReport(String name) {
      super(name);
   }

   @Override
   public void processReport() {
      super.processReport();
      System.out.println("Performing OUT Reports common stuff");
   }
}
```

Now we can create the concrete implementations of our report objects, defining one of them for each category and each type.

```
public class INInvoiceReport extends AINReport {

   protected INInvoiceReport(String name) {
      super(name);
   }

   @Override
   public void processReport() {
      super.processReport();
      System.out.println("Performing IN Reports Invoice specific stuff");
   }
}
```

```
public class INPurchaseReport extends AINReport {

    protected INPurchaseReport(String name) {
        super(name);
    }

    @Override
    public void processReport() {
        super.processReport();
        System.out.println("Performing IN Reports Purchase specific stuff");
    }
}
```

```
public class OUTInvoiceReport extends AOUTReport {

    protected OUTInvoiceReport(String name) {
        super(name);
    }

    @Override
    public void processReport() {
        super.processReport();
        System.out.println("Performing OUT Reports Invoice specific stuff");
    }
}
```

```
public class OUTPurchaseReport extends AOUTReport {

    protected OUTPurchaseReport(String name) {
        super(name);
    }

    @Override
    public void processReport() {
        super.processReport();
        System.out.println("Performing OUT Reports Purchase specific stuff");
    }
}
```

Now that we have defined the data model of our report, we move to the definition of the creational pattern which will be used to instantiate them depending on the category and type required.
We start from the interface that defines the factory and the method for creating a report that all the concrete factories of the various categories have to implement.

```
public interface IReportFactory {

    public AReport createReport(String type, String name);

}
```

We define the two concrete factories for the two categories INPUT and OUTPUT, that will take care of instantiation of the actual reports, based on the required report type.

```
public class INReportFactory implements IReportFactory {

   @Override
   public AReport createReport(String type, String name) {
      AReport doc = null;
      switch(type) {
         case "INV":
            doc = new INInvoiceReport(name);
         break;
         case "PUR":
            doc = new INPurchaseReport(name);
         break;
         default:
            break;
      }
      return doc;
   }
}
```

```
public class OUTReportFactory implements IReportFactory {

   @Override
   public AReport createReport(String type, String name) {
      AReport doc = null;
      switch(type) {
         case "INV":
            doc = new OUTInvoiceReport(name);
         break;
         case "PUR":
            doc = new OUTPurchaseReport(name);
         break;
         default:
            break;
      }
      return doc;
   }
}
```

At this point we create FactoryProvider, that’s an object that will perform the task to instantiate the correct concrete factory, needed to the creation of the correct report object requested, according to its category and its type. Often, the logic contained in the factory provider is inserted directly in the “client” object that uses the model objects (in our case would be the Archive class), but it is preferable to use this additional level of decoupling, in order to avoid having to change the consumer class in the case of new categories addition (see below an example).

```
public abstract class FactoryProvider {

    public static IReportFactory getFactory(String factoryType) {
        IReportFactory rf = null;
        switch(factoryType) {
            case "IN":
                    rf = new INReportFactory();
                break;
            case "OUT":
                    rf = new OUTReportFactory();
                break;
            default:
                break;
        }
        return rf;
    }
}
```

Finally we define the Archive class, which will be our report objects user. It will contain a list of report and a method for the insertion of a new report, which will use the FactoryProvider and the factory interface to instantiate new reports, without relying on concrete implementations. We also add a method to carry out the processing of all reports available in the list.

```
import java.util.ArrayList;
import java.util.List;

public class Archive {

        private List<AReport> reportList;

        public void addReport(String fam, String type, String name) {
            IReportFactory rf = FactoryProvider.getFactory(fam);
            if (this.reportList == null) {
                this.reportList = new ArrayList<AReport>();
            }
            this.reportList.add(rf.createReport(type, name));
        }

        public void processAllReports() {

            for (AReport r: this.reportList) {
                r.processReport();
                System.out.println("-----");
            }
        }
}
```

To test if everything works we use a test class, where we create an archive to which we add a list of reports that the system has to process.

```
public class AbstractFactoryTest {

    public static void main(String[] args) {

        String [] reports = {"IN_INV_001.txt","OUT_PUR_001.txt","IN_INV_002.txt", "IN_PUR_001.txt",
"OUT_PUR_002.txt", "OUT_INV_001.txt", "IN_INV_003.txt"};
        String tmp[] = null;

        Archive a = new Archive();

        for (String s: reports) {
            tmp = s.split("_");
            a.addReport(tmp[0], tmp[1], s);
        }

        a.processAllReports();
    }
}
```

Running the test program we get the following result:

```
Processing report: IN_INV_001.txt
Performing IN Reports common stuff
Performing IN Reports Invoice specific stuff
-----
Processing report: OUT_PUR_001.txt
Performing OUT Reports common stuff
Performing OUT Reports Purchase specific stuff
-----
Processing report: IN_INV_002.txt
Performing IN Reports common stuff
Performing IN Reports Invoice specific stuff
-----
Processing report: IN_PUR_001.txt
Performing IN Reports common stuff
Performing IN Reports Purchase specific stuff
-----
Processing report: OUT_PUR_002.txt
Performing OUT Reports common stuff
Performing OUT Reports Purchase specific stuff
-----
Processing report: OUT_INV_001.txt
Performing OUT Reports common stuff
Performing OUT Reports Invoice specific stuff
-----
Processing report: IN_INV_003.txt
Performing IN Reports common stuff
Performing IN Reports Invoice specific stuff
-----
```

The advantages of modeling this type of problems with a solution based on a design pattern like the Abstract Factory are its flexibility and scalability, given mainly by the following aspects:

- Archive depends only on interfaces and does not know anything about how the concrete factories and the related reports are created
- Adding a new category of reports has no impact on the Archive client class, but it is easily handled in FactoryProvider, simply adding a new case in the switch statement for the instantiation of the new concrete factory
- Adding a new type of report to an existing category is completely transparent to both the Archive and FactoryProvider classes and it is simply handled in the category concrete factory by adding a new case in the switch statement
- The FactoryProvider is completely decoupled and can be reused anywhere

Now we try to implement some of the extensions of the system highlighted in the previous list to verify in practice the impacts on the existing code.

Add a new reports category

Suppose we need to add to the system a new report category called “MIXED” which can itself have two types of reports “Invoice” and “Purchase”. To do that we need to create the new abstraction for reports of the MIXED category, extending again the AREPORT abstract class, and the two concrete implementations for puchase and invoice report types for this category.

```
public abstract class AMIXReport extends AReport {

   protected AMIXReport(String name) {
      super(name);
   }

   @Override
   public void processReport() {

      super.processReport();
      System.out.println("Performing MIX Reports common stuff");
   }
}
```

```
public class MIXInvoiceReport extends AMIXReport{

    protected MIXInvoiceReport(String name) {
        super(name);
    }

    @Override
    public void processReport() {
        super.processReport();
        System.out.println("Performing MIX Reports Invoice specific stuff");
    }
}
```

```
public class MIXPurchaseReport extends AMIXReport{

    protected MIXPurchaseReport(String name) {
        super(name);
    }

    @Override
    public void processReport() {
        super.processReport();
        System.out.println("Performing MIX Reports Purchase specific stuff");
    }
}
```

We then create the new concrete factory for the new category, that implements the generic factory interface of course.

```
public class MIXReportFactory implements IReportFactory {

    @Override
    public AReport createReport(String type, String name) {
        AReport doc = null;
        switch(type) {
            case "INV":
                doc = new MIXInvoiceReport(name);
            break;
            case "PUR":
                doc = new MIXPurchaseReport(name);
            break;
            default:
                break;
        }
        return doc;
    }
}
```

Until now we have only created new classes, without changing anything in the existing code. The only change required is in the FactoryProvider and thst’s ok, because it is exactly the decoupling level we intrdocude and on which we want the changes to be concentrated, because it is a class under our control. So, we modify this class as following:

```
public abstract class FactoryProvider {

   public static IReportFactory getFactory(String factoryType) {
      IReportFactory rf = null;
      switch(factoryType) {
         case "IN":
               rf = new INReportFactory();
            break;
         case "OUT":
               rf = new OUTReportFactory();
            break;
         case "MIX":
               rf = new MIXReportFactory();
         default:
            break;
      }
      return rf;
   }
}
```

The Archive class doesn't go through any changes and it is able to handle new reports of the new category in a completely transparent way!

We modify in the test class the list of reports to be processed, by inserting some of the new category created, and we verify that they are managed properly.

```
public class AbstractFactoryTest {

   public static void main(String[] args) {

      String [] reports =
{"MIX_INV_001.txt","MIX_PUR_002.txt","IN_INV_001.txt","OUT_PUR_001.txt","IN_INV_002.txt",
"IN_PUR_001.txt", "OUT_PUR_002.txt", "OUT_INV_001.txt", "IN_INV_003.txt"};

      Archive a = new Archive();

      String tmp[] = null;

      for (String s: reports) {
         tmp = s.split("_");
         a.addReport(tmp[0], tmp[1], s);
      }

      a.processAllReports();
   }
}
```

The result obtained is the following:

```
Processing report: MIX_INV_001.txt
Performing MIX Reports common stuff
Performing MIX Reports Invoice specific stuff
-----
Processing report: MIX_PUR_002.txt
Performing MIX Reports common stuff
Performing MIX Reports Purchase specific stuff
-----
Processing report: IN_INV_001.txt
Performing IN Reports common stuff
Performing IN Reports Invoice specific stuff
-----
Processing report: OUT_PUR_001.txt
Performing OUT Reports common stuff
Performing OUT Reports Purchase specific stuff
-----
Processing report: IN_INV_002.txt
Performing IN Reports common stuff
Performing IN Reports Invoice specific stuff
-----
Processing report: IN_PUR_001.txt
Performing IN Reports common stuff
Performing IN Reports Purchase specific stuff
-----
Processing report: OUT_PUR_002.txt
Performing OUT Reports common stuff
Performing OUT Reports Purchase specific stuff
-----
Processing report: OUT_INV_001.txt
Performing OUT Reports common stuff
Performing OUT Reports Invoice specific stuff
-----
Processing report: IN_INV_003.txt
Performing IN Reports common stuff
Performing IN Reports Invoice specific stuff
-----
```

As we can see the new reports are processed correctly and everything works!

Adding a new report type to an existing category

Now we try to add a new type of report to an existing category. Suppose we want to add the "Order" report type to the existing INPUT category.
We have to create the concrete implementation INOrderReport that extends AINReport.

```
public class INOrderReport extends AINReport {

   protected INOrderReport(String name) {
      super(name);
   }

   @Override
   public void processReport() {

      super.processReport();
      System.out.println("Performing IN Reports Order specific stuff");
   }
}
```

The only change to the existing code that we have to do is in the INPUT category reports factory. Also in this case it is not necessary to modify the Archive class and this time neither the FactoryProvider class.

```
public class INReportFactory implements IReportFactory {

   @Override
   public AReport createReport(String type, String name) {
      AReport doc = null;
      switch(type) {
         case "INV":
               doc = new INInvoiceReport(name);
            break;
         case "PUR":
               doc = new INPurchaseReport(name);
            break;
         case "ORD":
               doc = new INOrderReport(name);
            break;
         default:
            break;
      }
      return doc;
   }
}
```

We add in the reports list of our test class a report of this new type, such as “IN_ORD_004.txt”, and then we execute the program again.

```
public class AbstractFactoryTest {

    public static void main(String[] args) {

        String [] reports =
{"IN_ORD_004.txt","MIX_INV_001.txt","MIX_PUR_002.txt","IN_INV_001.txt","OUT_PUR_001.txt","IN_INV_002.txt",
"IN_PUR_001.txt", "OUT_PUR_002.txt", "OUT_INV_001.txt", "IN_INV_003.txt"};

        Archive a = new Archive();

        String tmp[] = null;

        for (String s: reports) {
            tmp = s.split("_");
            a.addReport(tmp[0], tmp[1], s);
        }

        a.processAllReports();
    }
}
```

The result obtained is shown below:

```
Processing report: IN_ORD_004.txt
Performing IN Reports common stuff
Performing IN Reports Order specific stuff
-----
Processing report: MIX_INV_001.txt
Performing MIX Reports common stuff
Performing MIX Reports Invoice specific stuff
-----
Processing report: MIX_PUR_002.txt
Performing MIX Reports common stuff
Performing MIX Reports Purchase specific stuff
-----
Processing report: IN_INV_001.txt
Performing IN Reports common stuff
Performing IN Reports Invoice specific stuff
-----
Processing report: OUT_PUR_001.txt
Performing OUT Reports common stuff
Performing OUT Reports Purchase specific stuff
-----
Processing report: IN_INV_002.txt
Performing IN Reports common stuff
Performing IN Reports Invoice specific stuff
-----
Processing report: IN_PUR_001.txt
Performing IN Reports common stuff
Performing IN Reports Purchase specific stuff
-----
Processing report: OUT_PUR_002.txt
Performing OUT Reports common stuff
Performing OUT Reports Purchase specific stuff
-----
Processing report: OUT_INV_001.txt
Performing OUT Reports common stuff
Performing OUT Reports Invoice specific stuff
-----
Processing report: IN_INV_003.txt
Performing IN Reports common stuff
Performing IN Reports Invoice specific stuff
-----
```

The example with all the classes can be downloaded here:

Abstract Factory example 6.17 KB
Download