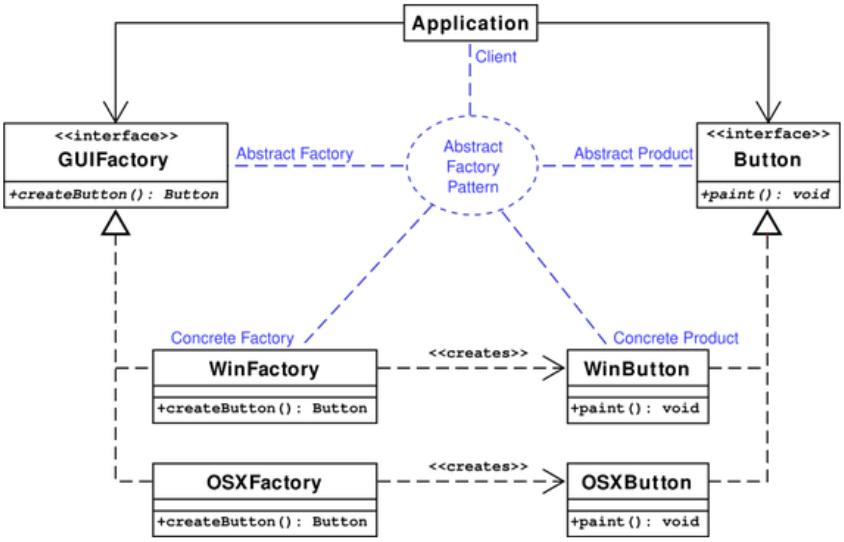
Abstract Factory

From Wikipedia

Rational Class Diagram [edit]



The Abstract Factory is used for encapsulating factories that have a common theme without specifying their classes.

```
Read the configuration file
If the OS specified in the configuration file is Windows, then
  Construct a WinFactory
  Construct an Application with WinFactory
else
  Construct an OSXFactory
  Construct an Application with OSXFactory
```

We want to be able to call Application with either a WinFactory or an OSXFactory.

```
class Application is
  constructor Application(factory) is
      input:  the GUIFactory factory used to create buttons
    Button button := factory.createButton()
    button.paint()
```

The Application constructor takes GUIFactory as its argument. It creates a Button object using the GUIFactory's createButton() method, then calls the Button's paint() method.
However, Application must work with both Windows and Mac, which have different buttons.

```
interface GUIFactory is
  method createButton()
      output:  a button
```

Therefore, GUIFactory is an “abstract factory”, the interface that is implemented by “concrete factories”. GUIFactory has one method, createButton(), which must be implemented by the concrete factories.

```
class WinFactory implementing GUIFactory is
  method createButton() is
      output:  a Windows button
    Return a new WinButton

class OSXFactory implementing GUIFactory is
  method createButton() is
      output:  an OS X button
    Return a new OSXButton
```

These are the concrete factories. They both implement the abstract factory (GUIFactory), and contain the implementation of the createButton() method. However, notice that the WinFactory createButton() method returns a WinButton, while the OSXFactory method returns an OSXButton.

```
interface Button is
  method paint()
```

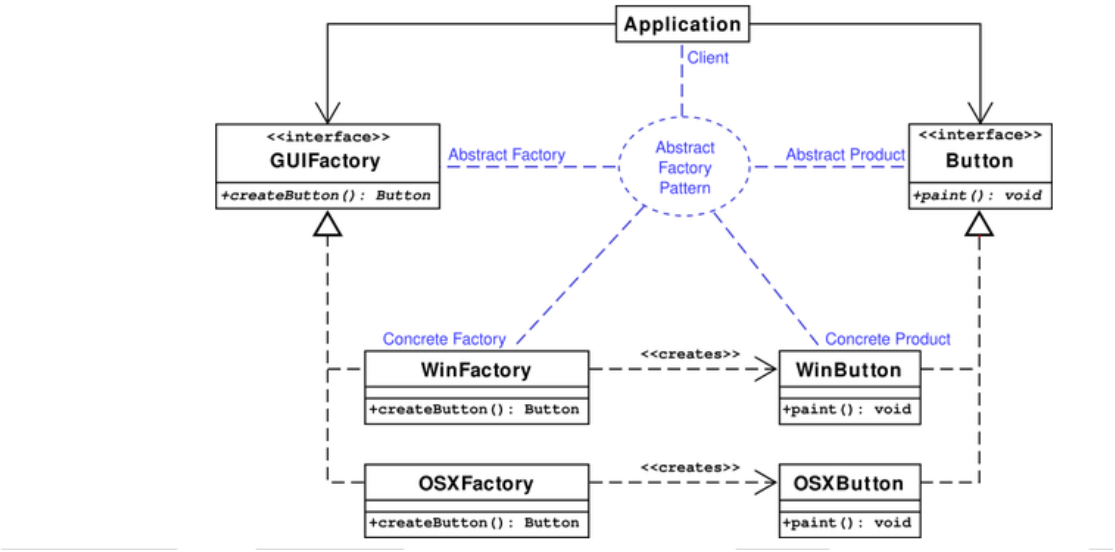
Button is the abstract product. It contains one method that must be implemented, paint().

```
class WinButton implementing Button is
  method paint() is
    Render a button in a Windows style

class OSXButton implementing Button is
  method paint() is
    Render a button in a Mac OS X style
```

WinButton and OSXButton are the concrete products. They both implement Button and contain the implementation of the paint() method. Like the concrete factories, the implementations are different: WinButton’s paint() renders a Windows-style button while OSXButton’s paint() renders a Mac-style button.

Rational Class Diagram [edit]



Looking again at the diagram, the abstract factory pattern's utility should be apparent. The Application can contain an abstract factory and an abstract product, without knowing their specific implementations (as they are abstract). This allows the programmer to create multiple concrete factories and concrete products that implement these interfaces, without having to modify the Application. In other words, the functionality of the Application is independent from what the concrete factories and products actually are. It works so long as they implement their respective interfaces.

Now, let's take a look at the professor's example code.

```
private static TripAdvisorFactory createFactory(String className) throws InstantiationException, Ille
        Class<?> clazz = Class.forName(className);
        return (TripAdvisorFactory) clazz.getConstructor().newInstance();
}
```

```
public static void main(String[] args){
        try{
                new Application(createFactory(System.getenv("FACTORY_IMPL"))).run();
        } catch (Exception e){
                System.out.println("Ooops ... " + e.getMessage());
        }
}
```

Here's the Main class, the main method constructs, and runs, a new Application that takes a TripAdvisorFactory as its argument. The TripAdvisorFactory is the abstract factory. The helper function createFactory returns a new factory object, created based on the function's String input.

```
public class Application {

        // Notice that this class depends only on interfaces, and not on
        // any specific implementation.

        private TripAdvisorFactory factory;


        public Application(TripAdvisorFactory factory){
                this.factory = factory;
        }


        public void run(){
                TripAdvisor advisor = factory.getInstance();

                System.out.println("Now I can test the trip advisor ...");
                double price = advisor.getCheapestPrice("Toronto", "Montreal");
                System.out.println("A cheapest trip from Toronto to Montreal costs " + price + "$.");
        }


}
```

Here's the Application class. It takes a TripAdvisorFactory variable in its constructor. Note that TripAdvisorFactory is the abstract factory, it is an interface. The run() method uses the TripAdvisorFactory's getInstance() method to return a TripAdvisor. TripAdvisor is the abstract product, which we'll see soon. Then the TripAdvisor's getCheapestPrice method is called, which will return a different value depending on which concrete product called it.

```
public interface TripAdvisorFactory {

         public TripAdvisor getInstance();

}
```

This is the TripAdvisorFactory interface. It is the abstract factory. It contains one method that must be implemented: getInstance().

```
public class TripAdvisorFactoryImpl1 implements TripAdvisorFactory{


        public TripAdvisor getInstance(){
                        return new TripAdvisor1();
        }


}
public class TripAdvisorFactoryImpl2 implements TripAdvisorFactory {

        @Override
        public TripAdvisor getInstance() {
                return new TripAdvisor2();
        }

}
```

Here are the concrete factories. Note they both implement the abstract factory, TripAdvisorFactory, and each returns a different concrete product in the implementation of its getInstance() method.

```
package csc301.abstractFactoryExample.tripAdvisor;

public interface TripAdvisor {

        public double getCheapestPrice(String fromStation, String toStation);

}
```

Here's the abstract product, the TripAdvisor interface. It contains one method to be implemented, getCheapestPrice.

```
public class TripAdvisor1 implements TripAdvisor {

        @Override
        public double getCheapestPrice(String fromStation, String toStation){
                return 0;
        }

}
```

```
public class TripAdvisor2 implements TripAdvisor {

        @Override
        public double getCheapestPrice(String fromStation, String toStation) {
                return 2.2;
        }

}
```

These are the concrete products, TripAdvisor1 and TripAdvisor2. They both implement the abstract product, TripAdvisor, but have a different implementation of the getCheapestPrice method (returning 0 and 2.2, respectively).

```
public Application(TripAdvisorFactory factory){
        this.factory = factory;
}


public void run(){
        TripAdvisor advisor = factory.getInstance();

        System.out.println("Now I can test the trip advisor ...");
        double price = advisor.getCheapestPrice("Toronto", "Montreal");
        System.out.println("A cheapest trip from Toronto to Montreal costs " + price + "$.");
}
```

Looking back at this code from the Application class shows the benefit of using the abstract factory pattern. The Application takes in the abstract factory interface (TripAdvisorFactory) and calls getInstance(). Both concrete factories implement this method, therefore it doesn't matter which one it is. This call could return either TripAdvisor1 or TripAdvisor2, but, again it doesn't matter which, because they both implement the abstract product (the TripAdvisor interface) and as such contain the getCheapestPrice method.

The benefit of using the abstract factory pattern is that it allows new concrete factories and concrete products to be created without modifying the higher-level Application. As long as these new concrete factories implement the abstract factory, and the new concrete products implement the abstract product, the Application does not need to be modified.