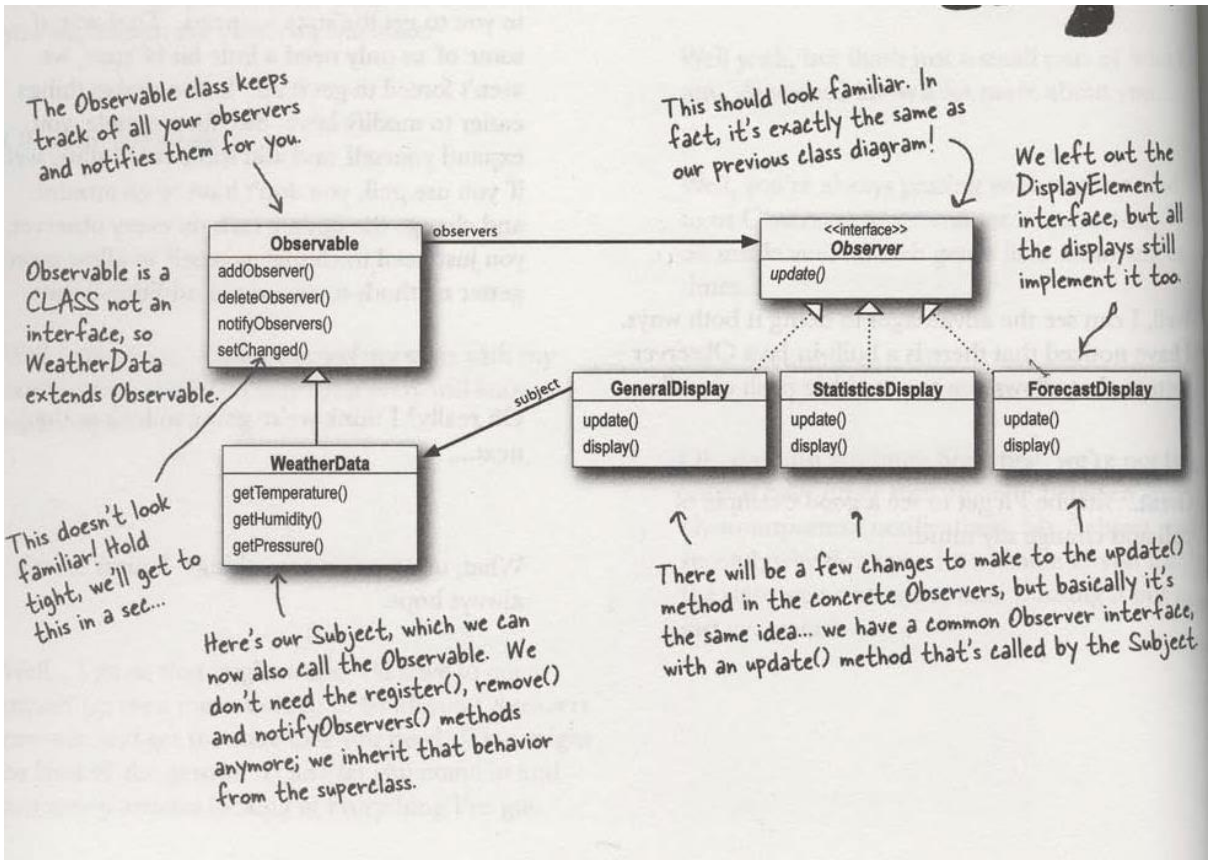
Part 2: Design Patterns.

Design pattern 1: The Observer Pattern

The following example came from Head first design patterns (see references).An example of the observer pattern would be a weather station system. The weather station has data that it reads from somewhere but needs to display it using specialised display modules. The weather station can have many different display modules displaying its data in different formats but will have no knowledge of the display modules. The WeatherData class will be the observable and the display modules will be the observers:



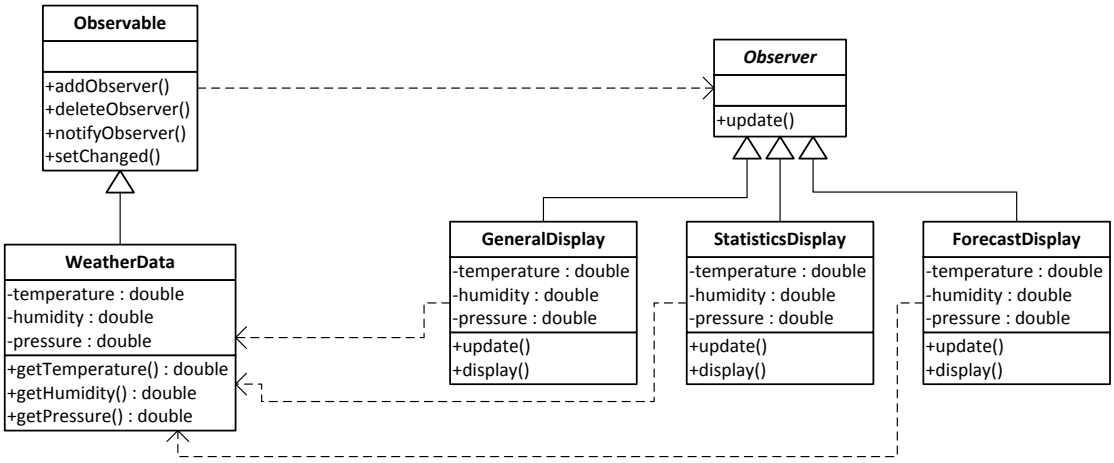
The observer pattern requires the following:

1. A concrete subject that:
    a. Extends a subject or observable class.
    b. Stores the state of some data that the concrete observers are interested in.
    c. Notifies it's observers of any changes.
2. A concrete observer that:
    a. Has a reference to the concrete subject.
    b. Stores data that should remain consistent with that of the subject.
    c. Implements the observer interface to maintain this consistency.

Considering the above criteria we can see that this example does indeed implement the observer pattern:

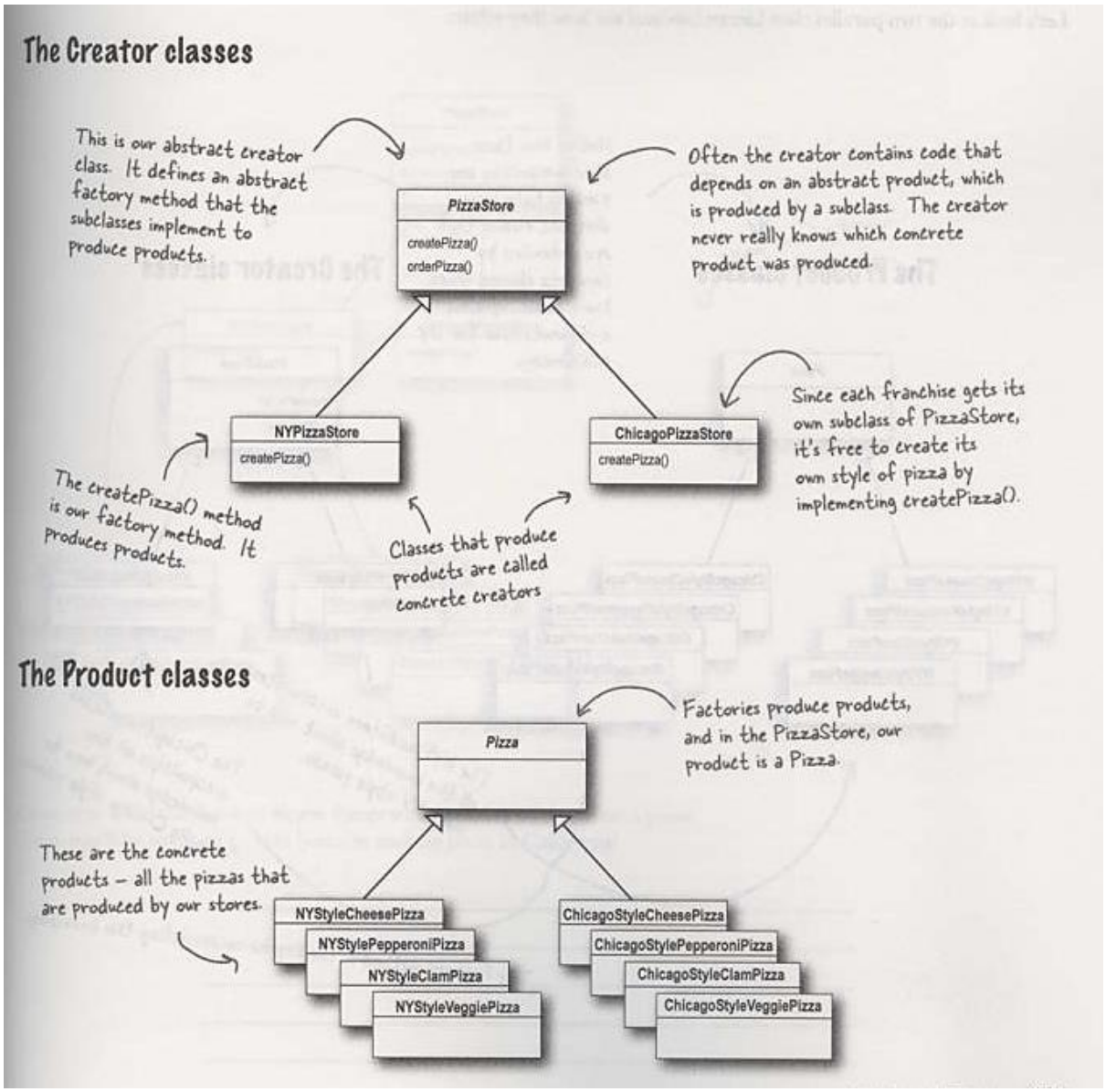
1. The concrete subject in this example is “WeatherData”. It:
    a. Extends the observable class.
    b. Stores temperature, humidity and pressure which the observers are interested in.
    c. Notifies observers if the state of these items changes via the notifyObservers() method inherited from the observable class.
2. The concrete observers in this example are “GeneralDisplay”, “StatisticsDisplay” and “ForecastDisplay”. They:
    a. All will have a reference to the concrete subject, in this case the “WeatherData” class.
    b. Store data that should be consistent with the concrete subject, these are temperature, humidity and pressure.
    c. All implement the observer interface.

The example in head first design patterns did not explicitly show the data in each class so a more detailed class structure is below:



Design Pattern 2: The Factory Method Pattern

The following example came from Head first design patterns (see references). An example of the factory method pattern would be a pizza store that wants to make pizzas all over the country. The menu for the pizza store wants to be the same across the country but wants to have some regional variations depending on the location. The super class “PizzaStore” will have the methods *createPizza()* and orderPizza(). There will be subclasses of “PizzaStore” that will implement the abstract method *createPizza()* which will vary depending on the location of the pizza store, for example NYPizzaStore and ChicagoPizzaStore will make a slightly different Pepperoni Pizza.



The factory method design pattern requires the following:

1. A product interface
2. Concrete products that:
    a. Implement the product interface
3. A Creator that:
    a. Declares the factory method which returns the product.
4. Concrete creators that:
    a. Override the factory method in the creator to return a concrete product.

With the above criteria we can see that the pizza store does implement the factory method design pattern:

1. The product interface in the example is the generic “Pizza”
2. The concrete products in the example are NYStyleCheesePizza, NYStylePepperoniPizza etc and they:
    a. All implement the product interface, in this case the “Pizza” interface.
3. The creator in this case is the “PizzaStore” class. It:
    a. Declares a factory method, in this case the createPizza() method that returns a product, in this case a “Pizza”.
4. The concrete creators in this case are the “NYPizzaStore” and “ChicagoPizzaStore” classes. They both:
    a. Override the factory method createPizza() to create concrete products. For NYPizzaStore this would the NYStyle pizzas and for ChicagoPizzaStore this would bee ChicagoStyle pizzas.

Design Pattern 3: The Singleton Pattern

An example of the Singleton design pattern would be the java.lang.Runtime.

Runtime
+static Runtime : Runtime
+static getRuntime() : Runtime

In order for something to conform to the Singleton design pattern it must have:

1. A singleton (itself) that:
    a. Defines an operation that lets clients access its unique instance
    b. Creates and maintains its own unique instance.

This example does conform to the Singleton design pattern because:

1. It has the singleton “Runtime” that:
    a. Defines the operation getRuntime() that returns its unique instance.
    b. Creates and maintains its own unique instance “currentRuntime”.

Below you can see the declaration of the method getRuntime() that returns its unique instance:

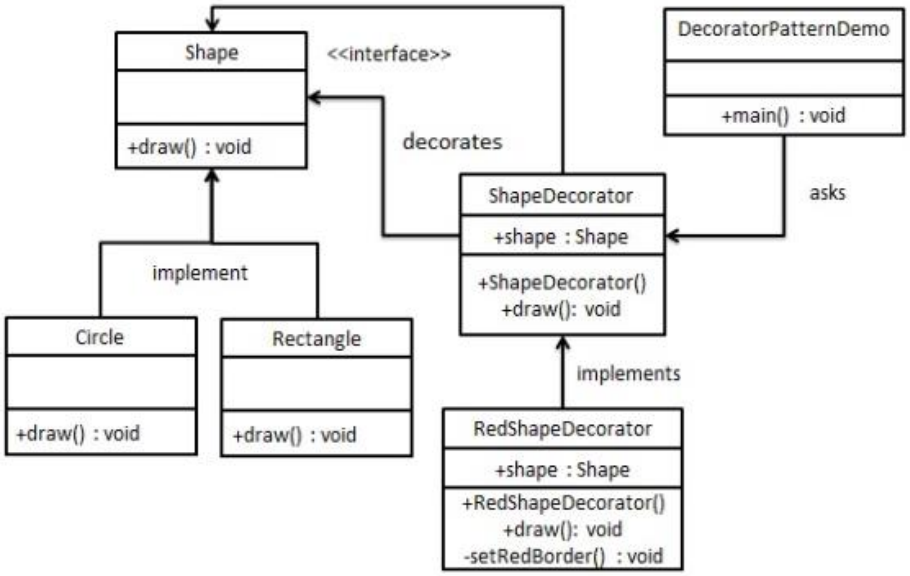
```
public static Runtime  getRuntime() {
          return currentRuntime;
}
```

And here you can see at the beginning of the class declaration it creates its own unique instance “currentRuntime”:

```
public class  Runtime {
      private static Runtime currentRuntime = new Runtime();
```

Design Pattern 4: Decorator Design Pattern

An example of the decorator is from the tutorials point website (see references). This example shows how a shape can be drawn with different attributes by decorating it. It uses specific shapes as concrete components and a concrete decorator “RedShapeDecorator” that in this case just outputs the shape contained within it followed by some text stating that it is red.



In order for a design to conform to the Decorator design pattern it must have the following:

1. A component interface
2. Concrete components that
    a. Implement the component interface.
3. A Decorator that:
    a. Has a reference to a component it will decorate
    b. Implements the component interface
4. Concrete decorators that:
    a. Extend the decorator class
    b. Have particular functions or attributes that they add to the component that they decorate.

Looking at the above criteria we can see that the above example does implement the Decorator design pattern, it has:

1. A component interface “Shape”.
2. Concrete components “Circle” and “Rectangle” that:
    a. Implement the component interface “Shape”.
3. A Decorator class “ShapeDecorator” that:
    a. Has a reference to the component “Shape”.
    b. Implements the component interface “Shape”.
4. A Concrete decorator “RedShapeDecorator” but could have more that:
    a. Extend the decorator class “ShapeDecorator”.

b. Have functions or attributes to add to the concrete component, in the case of "RedShapeDecorator" state that the shape is red.

In this example code implementation is already provided:

```
public interface Shape {
   void draw();
}
```

```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Shape: Rectangle");
   }
}
```

```
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Shape: Circle");
   }
}
```

```
public abstract class ShapeDecorator implements Shape {
   protected Shape decoratedShape;

   public ShapeDecorator(Shape decoratedShape){
      this.decoratedShape = decoratedShape;
   }

   public void draw(){
      decoratedShape.draw();
   }
}
```

```
public class RedShapeDecorator extends ShapeDecorator {
```

```
   public RedShapeDecorator(Shape decoratedShape) {
      super(decoratedShape);
   }

   @Override
   public void draw() {
      decoratedShape.draw();
      setRedBorder(decoratedShape);
   }

   private void setRedBorder(Shape decoratedShape){
      System.out.println("Border Color: Red");
   }
```

}

This can be easily added to so I have chosen to add TranslucentShapeDecorator to it:

```
public class TranslucentShapeDecorator extends ShapeDecorator {

   public TranslucentShapeDecorator (Shape decoratedShape) {
      super(decoratedShape);
   }

   @Override
   public void draw() {
      decoratedShape.draw();
      setTranslucent (decoratedShape);
   }

   private void setTranslucent(Shape decoratedShape){
      System.out.println("Opacity: Translucent");
   }
```

}

References:

1. Unknown.(2012*) Decorator Patter*.[Online]10.11.12. Available from - http://www.tutorialspoint.com/design_pattern/decorator_pattern.htm [Accessed: 03.10.14]

2. FREEMAN, E. et al. (2004) *Head First Design Patterns*. USA: O'Reilly.