OBJECT ORIENTED DESIGN PATTERNS

Date: 12 March 2018
Name: Priya Renjith

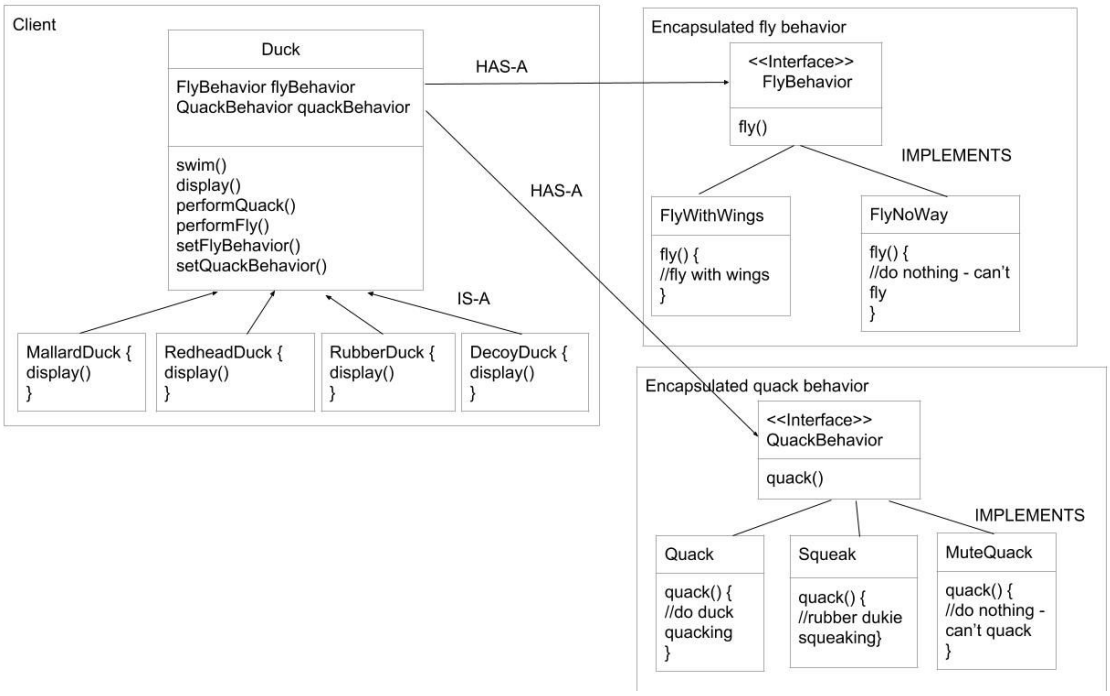
Object Oriented Design Patterns

In this document I will be highlighting the three main OO design patterns discussed in the book Head First Design Patterns by Eric Freeman and Elizabeth Freeman. I have chosen following design patterns for my discussion-Strategy Pattern, Observer Pattern, and Decorator Pattern.

Strategy Pattern:
Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from the clients that use it. This pattern is used to redesign the Si[1]mUDuck app explained in the book. Here the algorithms represent the things a duck would do (such as different ways of quacking or flying). Strategy pattern was used to implement the various behaviors of ducks. This particular design pattern was built understanding the disadvantages of using subclassing to provide specific duck behavior. This tells us that the duck behavior has been encapsulated into its own set of classes that can be easily expanded and changed at runtime.

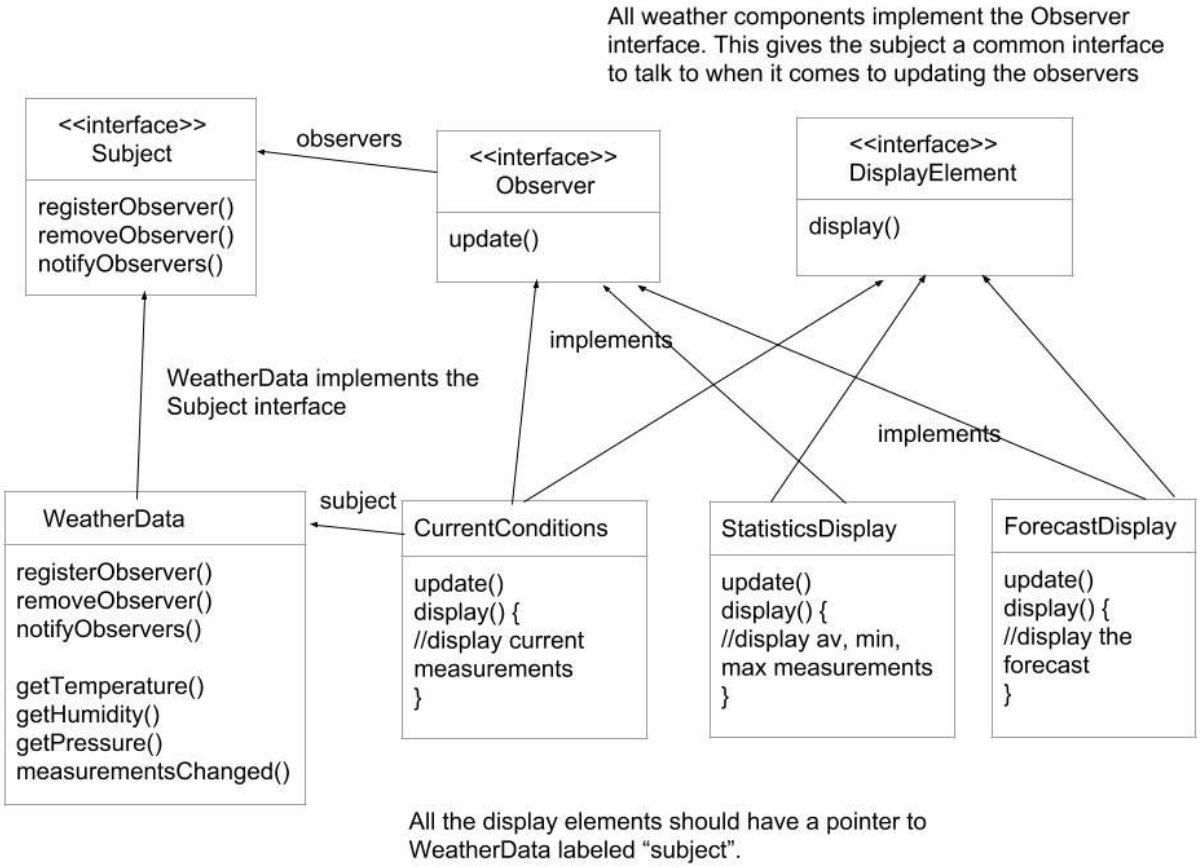
Each set of behaviours considered as a family of algorithms. Each duck has a FlyBehavior and a QuackBehavior to which it delegates flying and quacking. Interface is used to represent each behavior - for instance, a FlyBehavior and a QuackBehavior in this example- and each implementation of a behavior will implement one of those interfaces. Each set of behaviors sit in a separate class implementing a particular behavior interface. That way Duck classes won't need to know any of the implementation details for their own behaviors. With this design, other types of objects can reuse fly and quack behaviors because those behaviors are no longer hidden away in the duck classes. New behaviors can be added without modifying any of the existing behavior classes or touching any of the duck classes that use flying behaviors. Instead of inheriting their behavior, the ducks get their behavior by being composed with the correct behavior object. This is called composition- which gives a lot of flexibility to the system. Not only does it encapsulate a set of behaviors into their own classes, it also lets us change behavior at runtime as long as the object we are composing with implements the correct behavior interface. This can be achieved by creating a setter method in the duck superclass and all duck subclasses inheriting this method. If anytime duck's behavior need to be changed at runtime, just call the duck setter's method for that behavior. Following diagram illustrates the reworked class structure for the SimUDuck app.

1



Observer Pattern:

Observer pattern defines a one to many relationship between objects so that when one object changes state, all of its dependents are notified and updated automatically. The subject and observers define the one to many relationship. Subject is the object that contains the state and controls it. There are many observers and they rely on the subject to tell them when it's state changes. Observer pattern provides an object design where subjects and objects are loosely coupled. Loosely coupled designs allow us to build flexible OO systems that can handle change because they minimize the interdependency between objects. The only thing that subject knows about an observer is that it implements a certain interface (observer interface). We can add new observers at any time. We never need to modify the subject to add new types of observers. All we have to do is implement the Observer interface in the new class and register as an observer. We can reuse subjects or observers independently of each other. Changes to either subject or an observer will not affect the other. Following class diagram illustrates the Observer pattern used in designing a weather station.

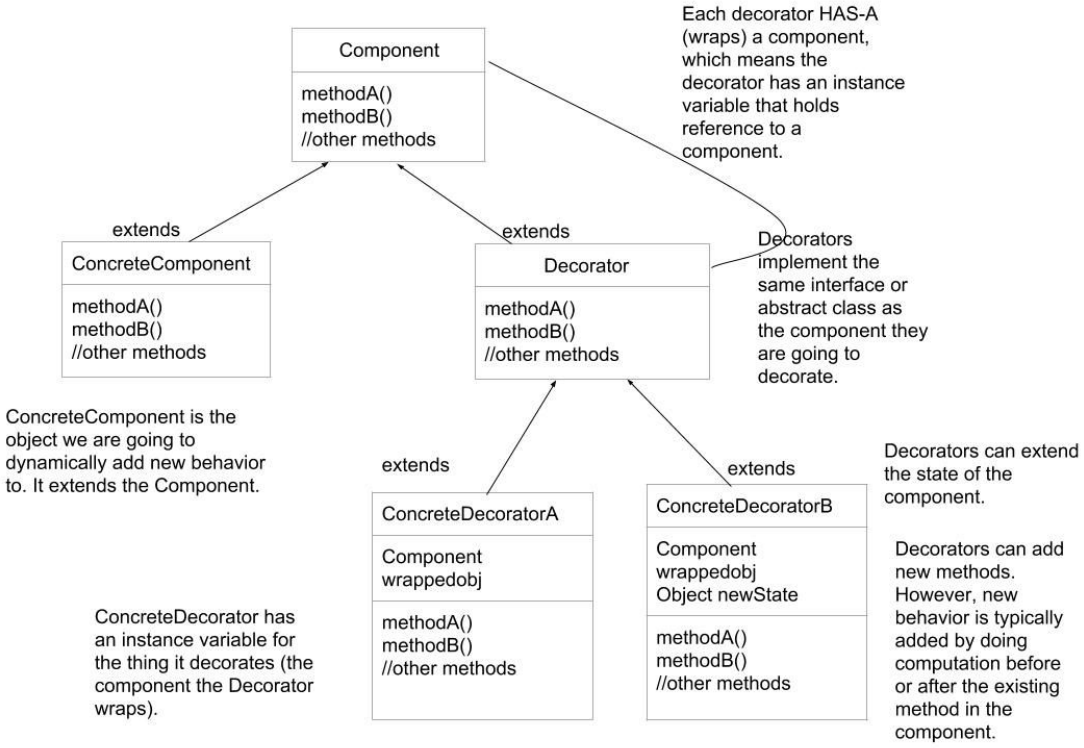


Swing makes heavy use of the observer pattern, as do many GUI frameworks. Java has several implementations of the observer pattern, including the general purpose java.util.Observable. You can push or pull data from the Observable when using the pattern.
Observer patterns follows the three OO design principles namely -a) identify the aspects of your application that vary and separate them from what stays the same, b) program to an interface, not an implementation and c) Favor composition over inheritance.

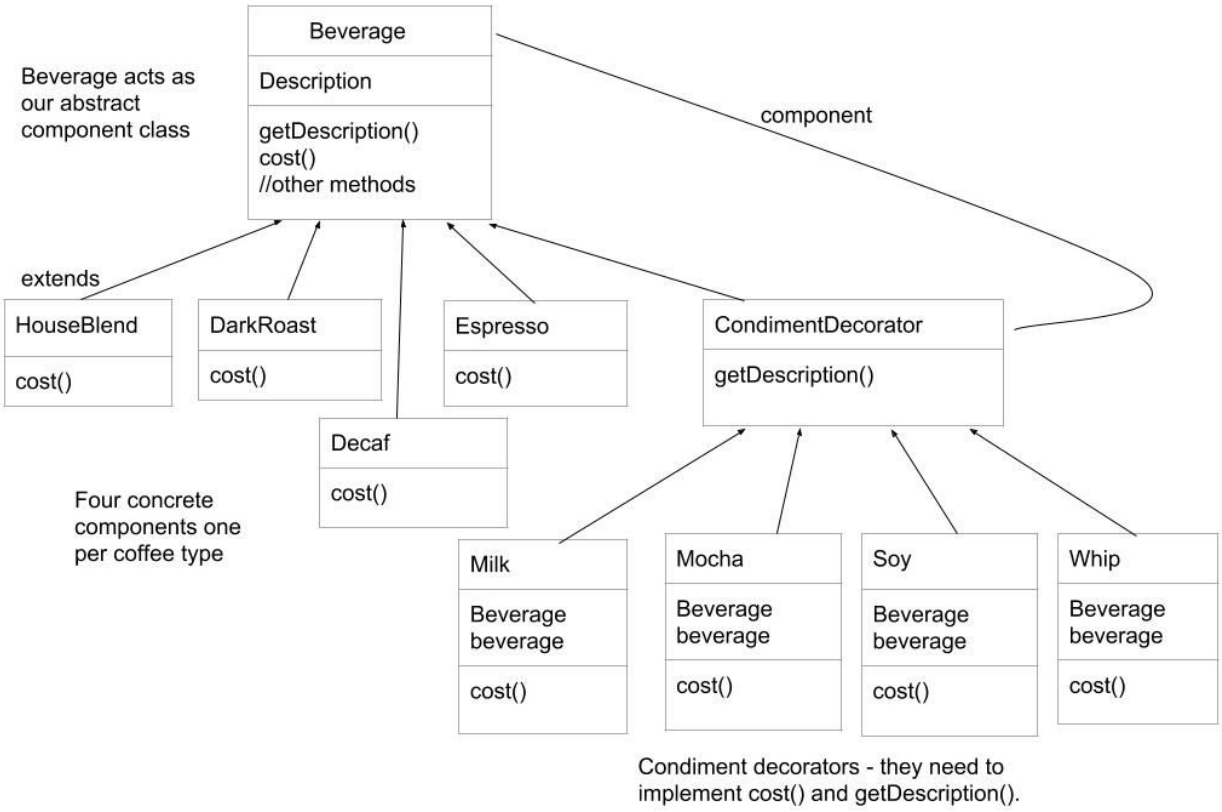
Decorator Pattern:

Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. Decorators have the same supertype as the objects they decorate. You can use one or more decorators to wrap an object. Given that the decorator has the same supertype as the object it decorates, we can pass around a decorated object in place of the original (wrapped) object. The decorator adds it own behavior before/after delegating to the object it decorates to do the rest of the job. Objects can be decorated at any time, so we can decorate objects dynamically at runtime with as many decorators as we like. Using decorator pattern helps in achieving the open closed design

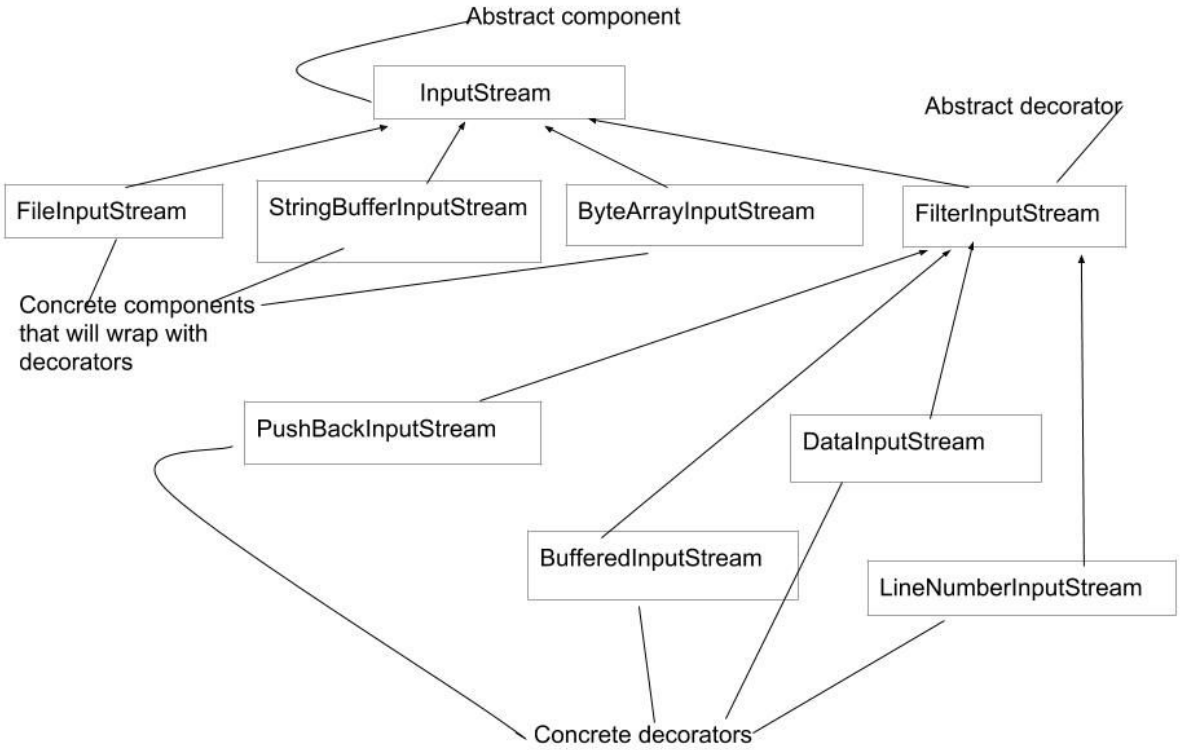
principle of classes should be open for extension but closed for modification. New behavior is acquired by composing decorators with base components as well as with other decorators.
If relied just on inheritance, behavior can only be determined statically at compile time. With composition we can mix and match decorators any way we like at runtime. Decorators can result in many small classes in our design and overuse can be complex.
Following class diagram illustrates a decorator pattern.



Here is an example where decorator pattern is used:



Java I/O package contains lot of components that are used as decorators. One such example that uses decorator components is for reading data from a text file. FileInputStream, StringBufferInputStream, ByteArrayInputStream are all base components used to read data from a text file. BufferedInputStream is a concrete decorator which buffers the input as it reads data from a file.It also augments the interface with a new method readLine() for reading character based input, one line at a time. LineNumberInputStream is a concrete decorator which adds the ability to count the line numbers as it reads data. The class diagram is attached below.



END OF DOCUMENT