

# CSC7322: Object Oriented Development

**J Paul Gibson, A207**

`paul.gibson@it-sudparis.eu`

<http://www-public.it-sudparis.eu/~gibson/Teaching/CSC7322/>

## **Design (In Java)**

[.../~gibson/Teaching/CSC7322/L2-Design\(In Java\).pdf](http://www-public.it-sudparis.eu/~gibson/Teaching/CSC7322/L2-Design(In Java).pdf)

## Good design is all about experience:

- Recognising a problem's structure and knowing about alternative solutions
- Knowing about implementation issues specific to potential implementation languages
- Realising that there is often not a perfect fit between the design and the target implementation language
- Being able to adapt the design to the language and/or adapt the language to the design

Note: in OO we can consider the library to be part of the language

Let us return to the Dice/Observer problem with this in mind.

## From the previous lecture –

Choosing to implement a `DiceWithStatistics` as a subclass of `Dice` is a design decision that could have consequences on later stages of the development.

An alternative re-use mechanism is to design a `DiceWithStatistics` class/object to have a `Dice` component class/object.

If you wish, attempt to implement this second design.

In the next PBL session we will see the consequences of each of these designs when we try to implement an observable `Dice` that is observed by a `Dice View`.

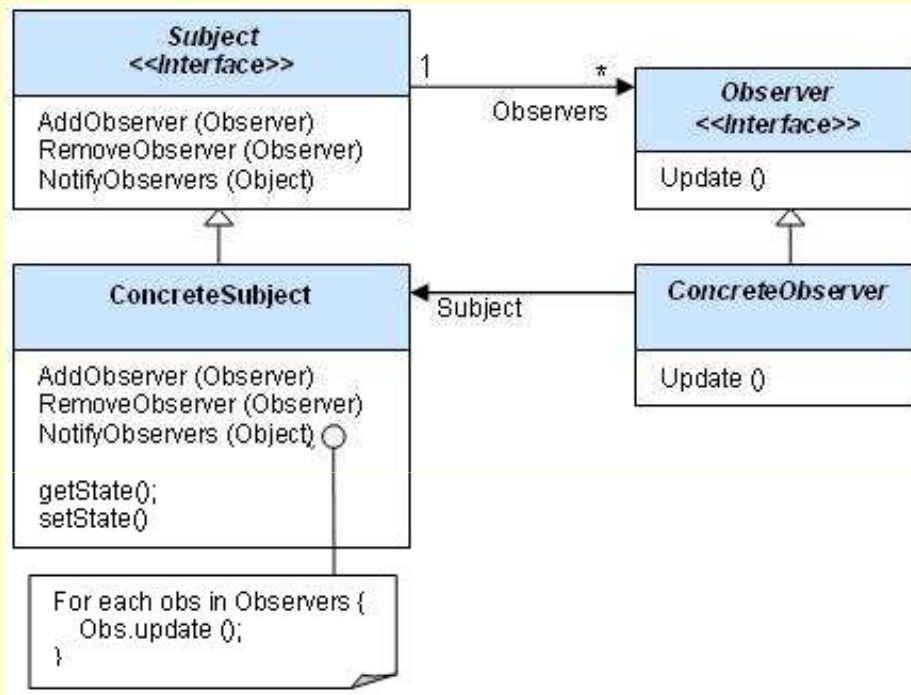
## The Observer Design Pattern

This pattern is among the most useful for object-oriented software design.

It is a key part of the MVC pattern - the JDK itself makes heavy use of a variant of this pattern in the 1.1 AWT event delegation model, and the Swing libraries also facilitate its use

The JDK libraries also provide a reusable implementation of the pattern in the form of the `java.util.Observer` interface and the `java.util.Observable` class.

# The Observer Design Pattern: UML



The idea of the pattern is to model a one-to-many dependency without tightly coupling the observed object with its many observers. When the observed object changes in some interesting way it can automatically notify all of its observers without knowing them directly

## Weaknesses

Unfortunately, a number of weaknesses have been identified in the JDK's Observer/Observable classes.

These weaknesses significantly limit the reusability and power of the classes, which is a shame since powerful reusability is a big part of what object oriented design is all about.

Most of the weaknesses are due to the fact that `java.util.Observable` is a class rather than an interface; or rather, that it is a class without a corresponding interface. This implies that the only way to reuse `Observable` is to subclass it. You can't take an existing class and tack on the role of `Observable` by having it implement an `Observable` interface because there is no `Observable` interface.

## Weaknesses

What if you have a class that is already in a class hierarchy and also needs to play the role of an Observable?

Since Java doesn't support multiple inheritance, you're out of luck. That class cannot extend from Observable because it is already extending from some other class.

It also means that you are stuck with the one and only implementation of Observable in `java.util.Observable`. For a variety of reasons, you may want to use an alternate implementation - e.g., to do the notification in a separate thread or in a particular order. You may even want to vary the implementation of Observable at runtime. There is no Observable interface for your alternate implementations to implement. You cannot reuse Observable by composition so you cannot vary the composed Observable implementation at runtime.

**QUESTION:** Did you discover this when you tried to make an Observable Dice?

## Weaknesses

The designers of the Observable class broke two general principles of object-oriented design with Java:

1. The first principle is to design with interfaces rather than classes. Whenever possible, avoid committing yourself to a particular implementation of an interface.
2. The second principle is to favor reuse by composition over reuse by inheritance unless a class hierarchy is clearly indicated.

By omitting an Observable interface and making some of its methods protected, the designers made it impossible to reuse Observable by composition.

## Weaknesses

A minor weakness in Observable is the necessity to call `setChanged()` before `notifySubscribers()`.

The intention there seems to be to eliminate needless notifications in cases where there is no interesting change to the Observable.

There may be situations in which this two-stage notification is appropriate, but it isn't the simplest case and programmers shouldn't be forced to use this implementation in all situations. Also, `setChanged()` is protected, further reinforcing the necessity to reuse the class only by inheritance.

## Weaknesses

The main weakness in the Observer interface is its tight coupling with the Observable class.

The first parameter to the update() method is unnecessarily typed as an Observable. If it were typed more generally as a simple Object, the Observer interface would be more reusable. It then could be used with any Observable implementation or even in any situation, completely unrelated to Observer/Observable, which called for a void method with two Object parameters.

## Possible Solutions

The general opinion is that one should re-use a better/improved Observer.

Many such improved Observers exist (the version by Coad and Mayfield is perhaps the best known)

You should be able to write your own!

## References

Gamma, E., Johnson, R. and Vlissides, J., "Design Patterns: Elements of Object-Oriented Architecture", Addison-Wesley, Reading, MA, 1995.

Coad, P. and Mayfield, M., "Java Design: Building Better Apps and Applets", Yourdon Press, Upper Saddle River, NJ, 1997.

## Alternative Solutions

Did you manage to do this without using a different Observer class?

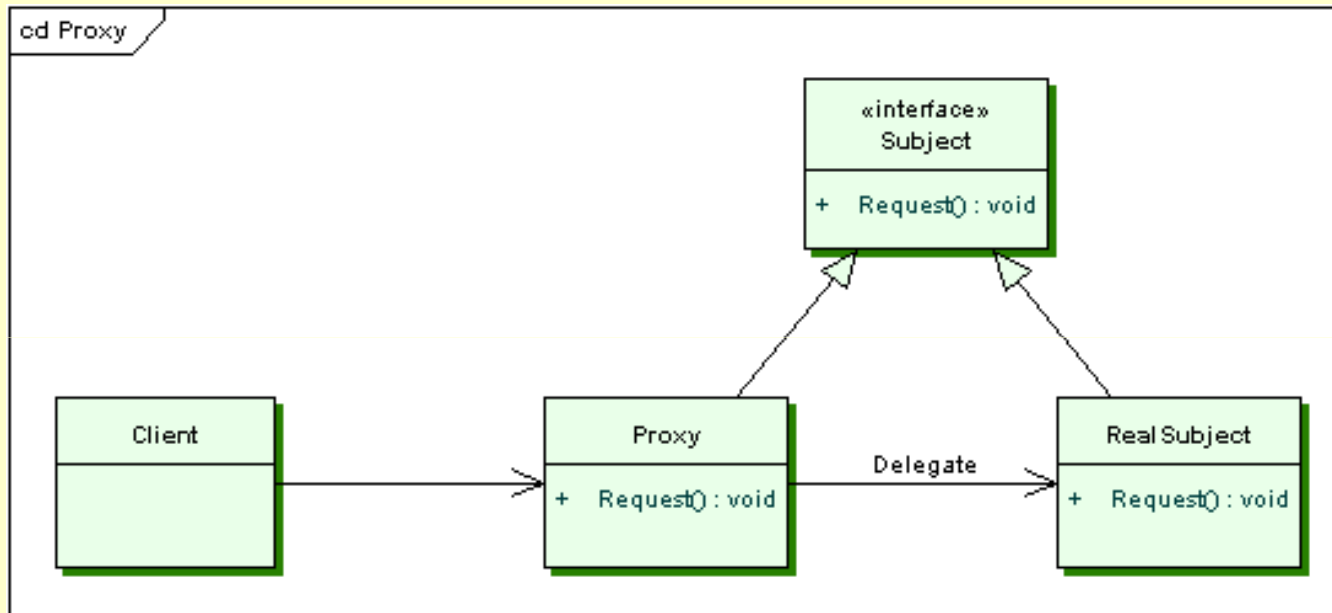
What options are there if you have to use the standard Observer/Observable ?

When looking for solutions you may stumble upon other design patterns that could help, eg:

- Proxy
- Decorator

There are also other ‘poor’ solutions .... We need to look at these on the board.

# Proxy Design Pattern



# Decorator Design Pattern

