The Observer Pattern

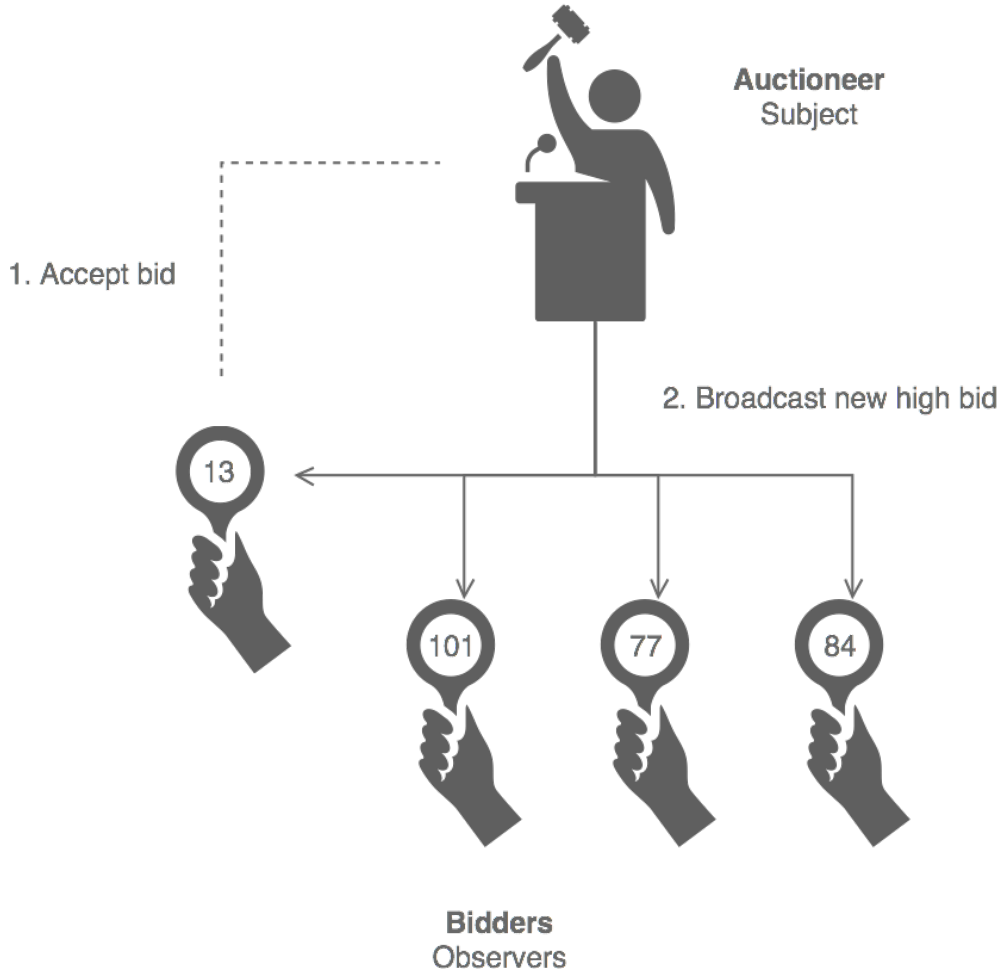
Mel Ó Cinnéide
School of Computer Science
University College Dublin

Introduction to the Observer Pattern

- Intent
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependants are notified and updated accordingly.

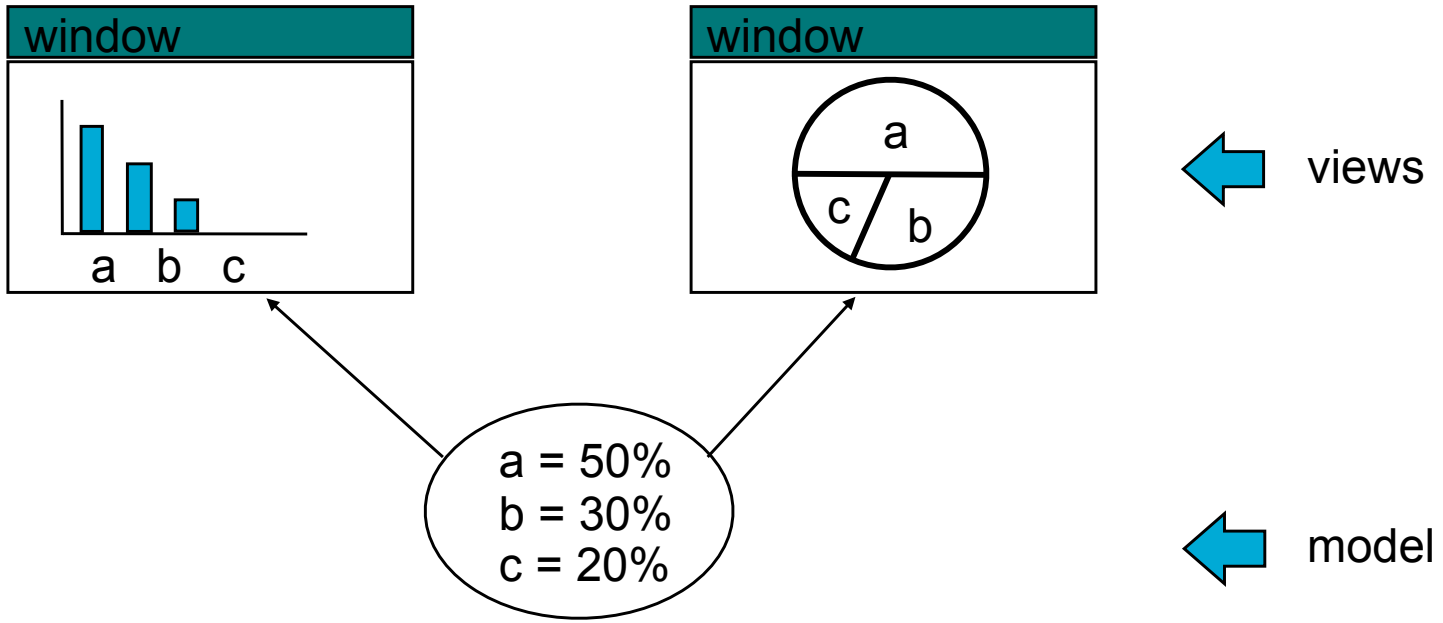
- Examples
  - Examples of this pattern abound, both in the real world and in the software domain.
  - To motivate this pattern, we consider two examples.

Observer -- Real World Example



Observer -- Motivation

- In a spreadsheet application, several views of the same data are to be displayed on the screen simultaneously, e.g.,



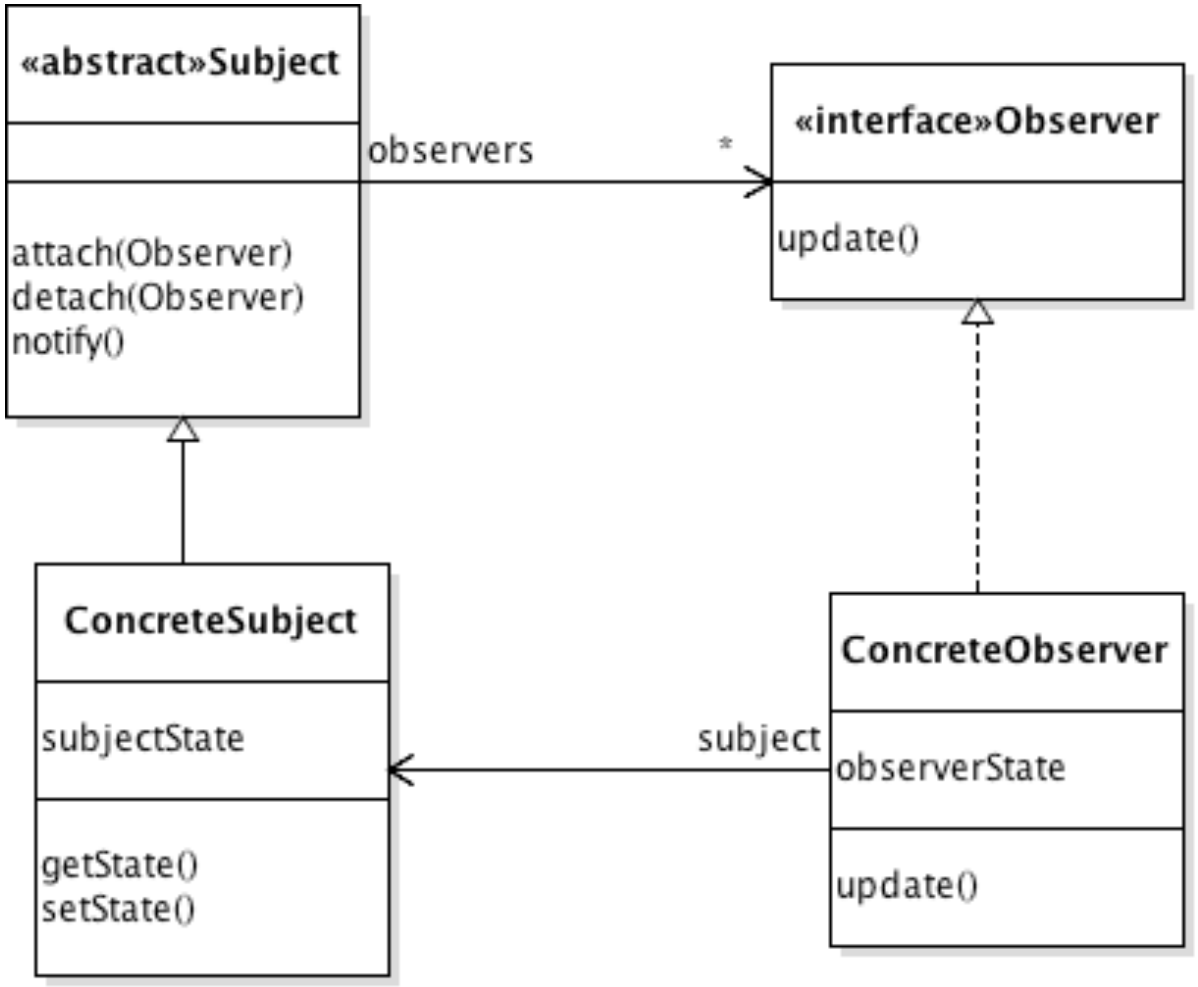
- When the data changes, the views must be updated
  - How best to design this in software?

Observer -- Applicability

Use the Observer pattern when:

- An abstraction has two aspects, one dependent on the other, and it is necessary to model them as separate objects;
- A change to one object requires changing others, and you don't know how many objects need to be changed;
- An object should be able to notify other objects, without making assumptions about who these objects are (loose coupling).

Observer -- typical class structure

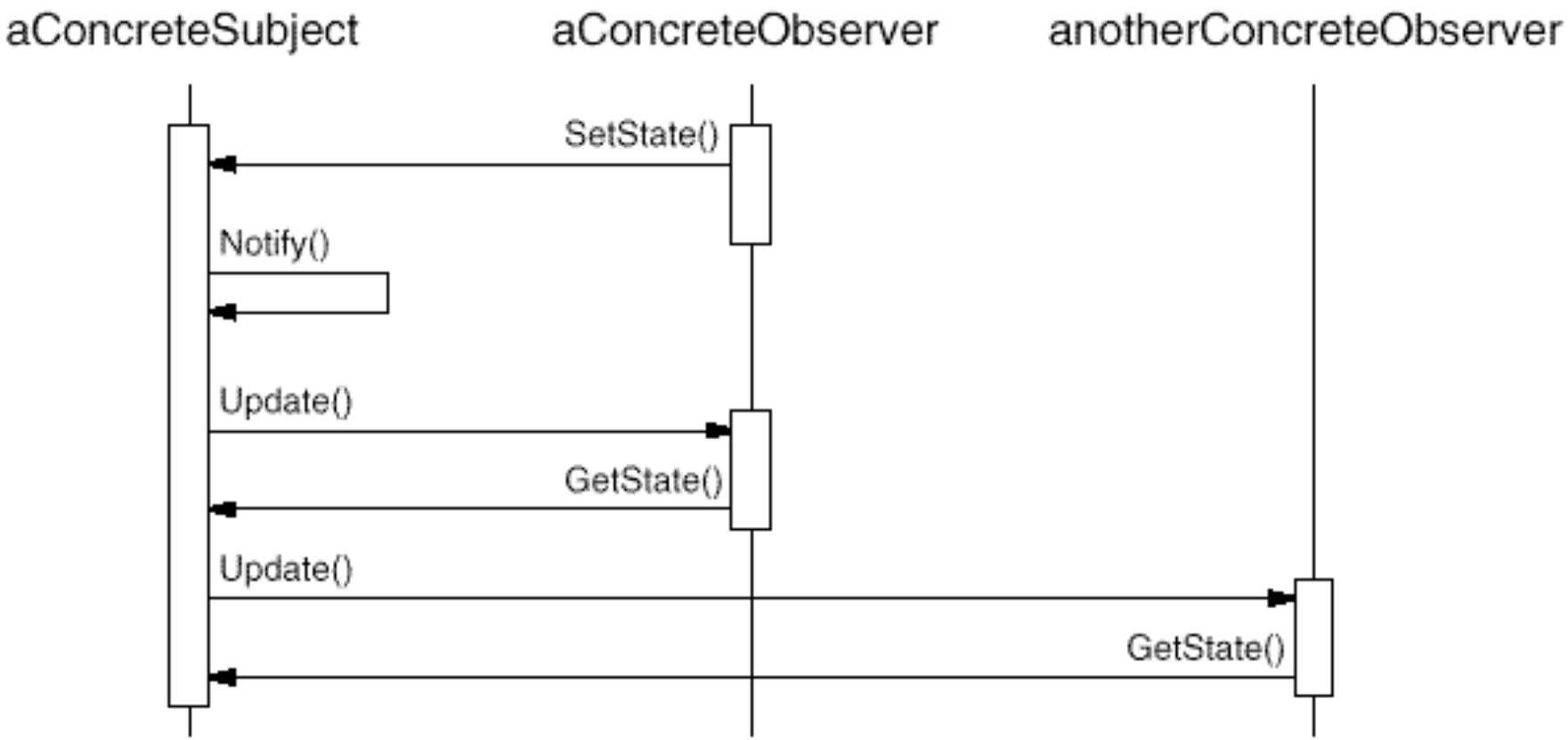


notify() invokes **update()** on all observers

getState() returns the contents of **subjectState**

update() invokes **getState()** on **subject** and stores result in **observerState**.

Observer -- Collaborations



Observer -- Consequences

Benefits and liabilities of Observer include:

- *Abstract coupling from Subject to Observer*. (The coupling is tighter in the opposite direction… why couldn't it be weakened?)
- *Support for broadcast communication*. Sending a single **`notify`** request results in multiple **`update`** messages being sent (known as *multicasting*).
- *Unexpected updates*. A single change to the subject may cause a cascade of updates to observers and their dependants.
  - can be a big challenge in a complicated system

Observer Implementation – Triggering the Updates

- *Who triggers the update?* When a client updates the subject, the subject must be sent a **`notify`** message in order for the **`update`** messages to be sent to the observers.
- Who sends the subject the **`notify`** message? Two possibilities:
  - All state-setting operations on the subject call **`notify`** after they change the subject state (less efficient but safe).
  - Clients that update the subject state call **`notify`** at the appropriate time (more efficient, but also more error-prone).

Observer Implementation – Push and Pull Models

- **Pull Model**: the observer is notified that a change has occurred and must find out itself what changes have occurred.
- **Push Model**: the subject sends observers detailed information about the change that has occurred (in the simplest case, the entire new state itself).
- The Pull model is simple, but leads to further requests from the observer to the subject.
- The dumb 'push everything' model is simplest, but may be inefficient.
- Extending the Subject registration interface to enable smarter pulling is possible, but increases subject→observer coupling.

Observer – Other Implementation Issues

Issues related to the implementation of Observer include:

- *Mapping Subjects to Observers*. Simplest is to store a list of references to observers in the subject.
  - A central look-up table to store the Subject->Observer mapping is another possibility.

- *Observing multiple subjects*. In this case the observer can receive updates from several subjects, so it needs to know the source of any **`update`** message. Simplest solution is for the subject to pass a reference to itself with the **`update`** message.

- *Garbage Collection*. The reference the Subject holds to an Observer may prevent the Observer from ever being garbage collected. Either Observers must detach from Subject, or possibly a **weak reference** can be used.

Observer Implementation – Java

- Java explicitly supports the Observer pattern through `java.util.Observer` and `java.util.Observable`.
- The **`Observer`** interface contains the method update
  - public void update(Observable o, Object arg)
- The **`Observable`** class provide a full implementation of Subject behaviour including:
  - addObserver(Observer o)
  - deleteObserver(Observer o)
  - notifyObservers()

Uses of Observer

- First and best-known use is as part of the MVC (Model/View/Controller) framework in Smalltalk.
- Observer is a very commonly-occurring pattern:
  - Java Listeners are essentially a specialisation of the Observer pattern.
  - Language support in Java, C#, Ruby and others.
- Non-UI uses of this pattern are common as well, e.g. consider any type of monitoring system that is interested in (observes) the state of multiple sensor devices.

Observer – Sample Code and Related Patterns

- For sample code see GoF text.
  - Or code in Observer practical
- Related Patterns include: Ignore for 47480 2018
  - **Template Method**
  - **Strategy** can be used where the `notify` or `update` operations are complex and added flexibility is required.
  - **Mediator** may be used to coordinate updating of multiple interdependent observers
  - **Adapter** can be applied if an observer wanna-be doesn't implement the required `update` interface.
- Do any other related patterns come to mind?