The Observer Pattern

Mel Ó Cinnéide
School of Computer Science
University College Dublin

Introduction to the Observer Pattern

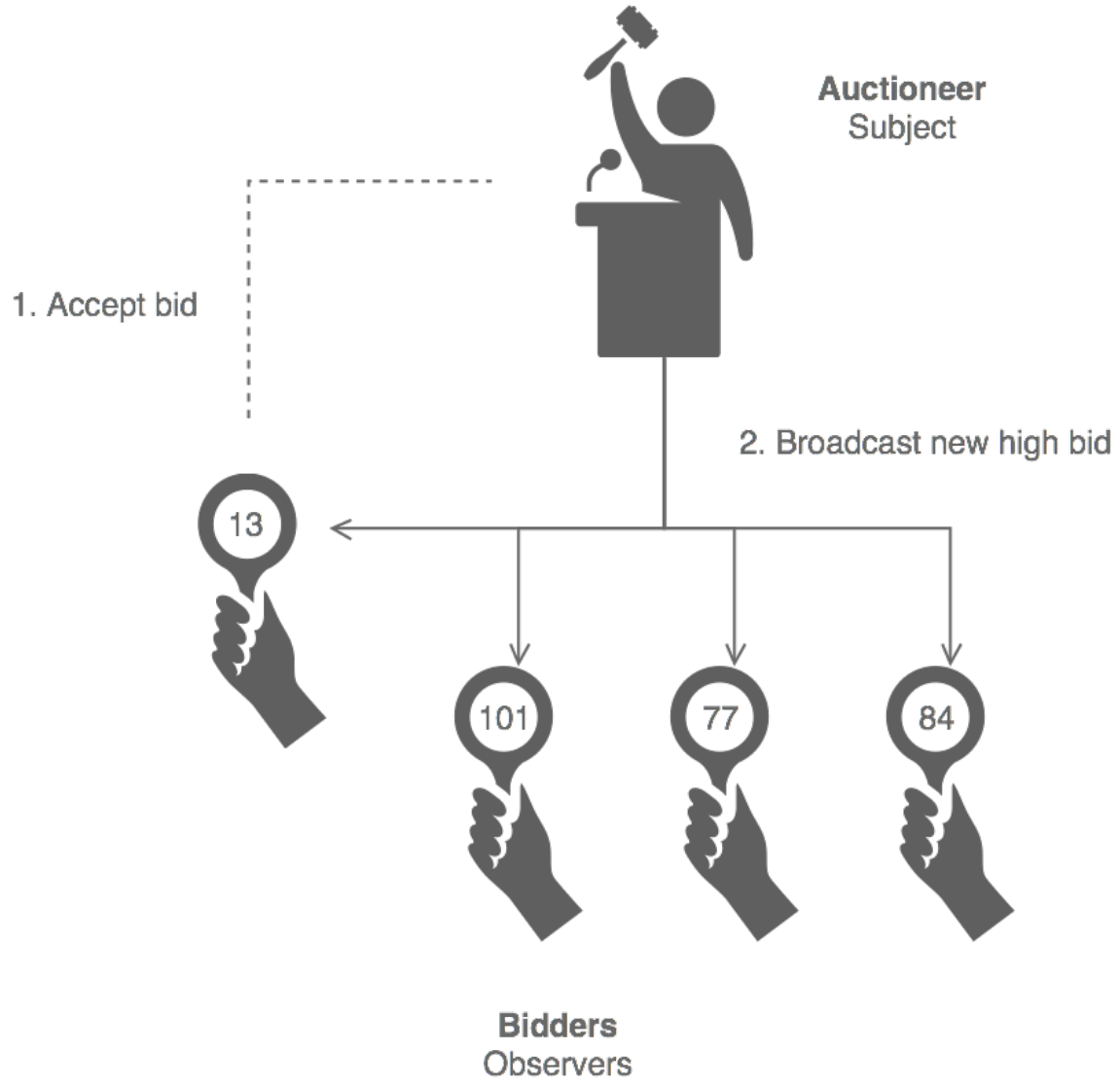
Intent

- 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 a real-world example.

Real World Example

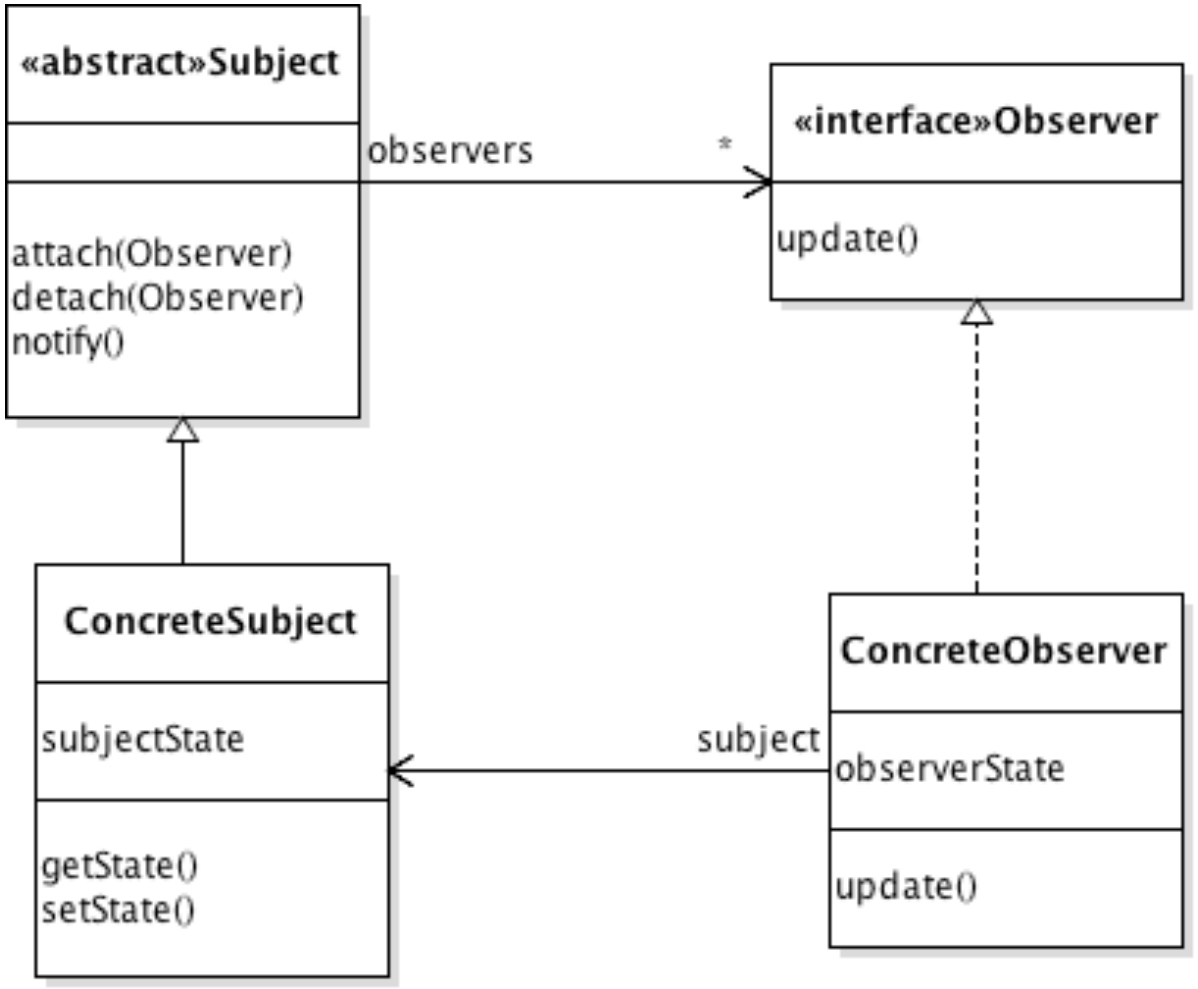


Applicability

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 those objects are (loose coupling).

Observer *typical* class structure

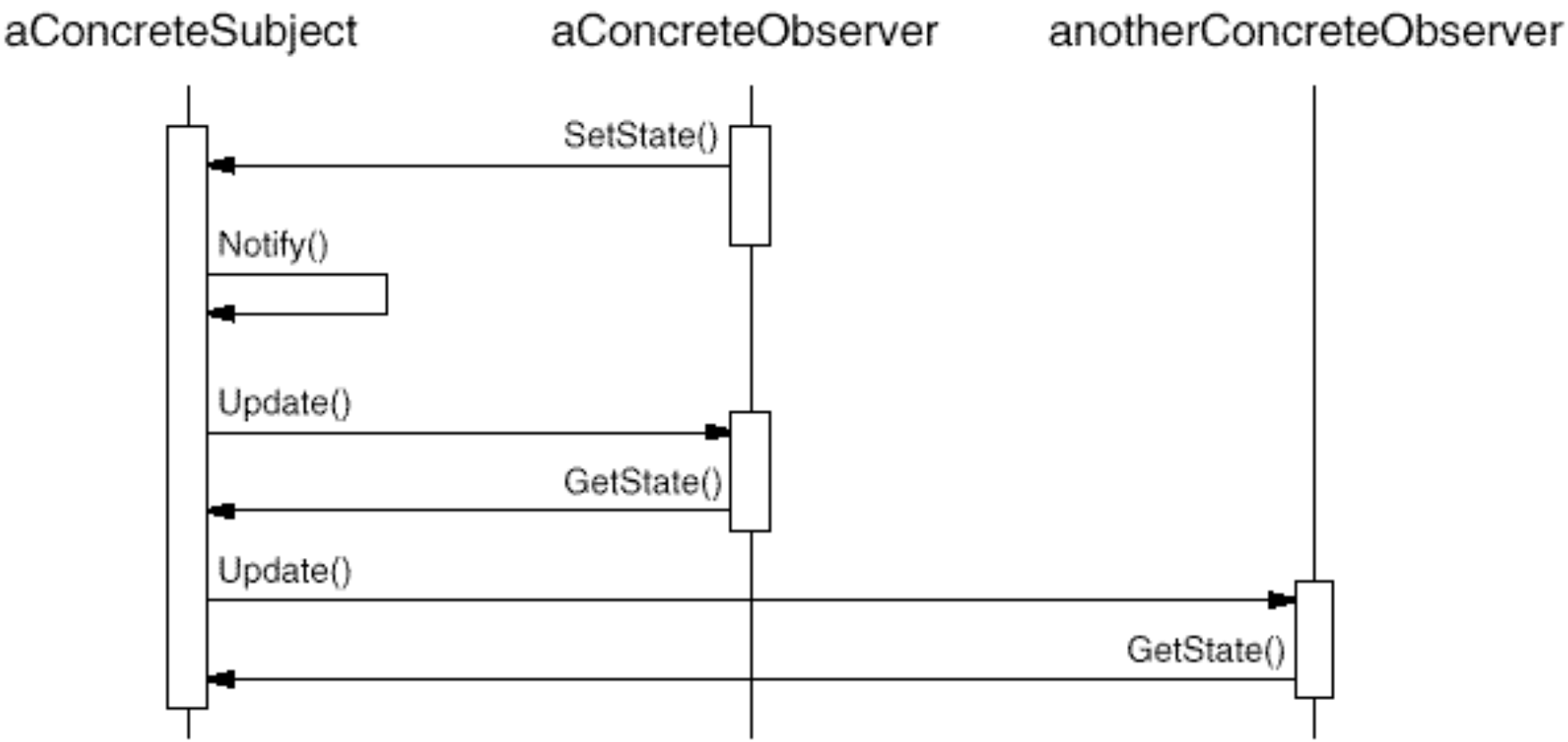


`notify` invokes **`update`** on all observers

`getState` returns the contents of **`subjectState`**

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

Collaborations



Consequences of applying Observer

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 serious challenge in practice

Triggering the Updates

Who triggers the update? When the subject gets updated, **`notify`** must be invoked in order for the **`update`** messages to be sent to the observers.

Who invokes **`notify`** on the subject? 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).
- Clients can switch mode using a `setNotificationsOn/Off` protocol, but this remains error-prone.

Push and Pull Models

Pull Model: the observer is notified that a change has occurred and must find out itself what the changes are.

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 pushing is possible, but increases subject→observer coupling.

More on the Push and Pull Models

These models make more sense with a slightly more complex example, so let's extend the Auctioneer example to an *Auction House*, where multiple items are auctioned simultaneously. The AuctionHouse is the Subject, and let's assume the Observers comprise online bidders.

Pull Model: A bidder registers with the AuctionHouse and receives a message `update(AuctionHouse)` whenever a new bid is made. This AuctionHouse is easy to implement, but now the bidder has to repeatedly interrogate the AuctionHouse to see if the specific item they are interested in has a new bid.

Push Model: A bidder registers their interest in a certain item and only receives a message `update(Item, NewBid)` whenever there is a new bid on the item in which they are interested. This is sweet for the bidder, but the Subject will have to manage the Observer -> Item mapping.

The discriminating factor is where is the specific interest of the Observer is stored: in the Observer (pull model) or in the Subject (push model).

A "push/pull" model is possible, where e.g. the AuctionHouse sends an update message to say that Item 37 has a new bid, but leaves it to the bidder to interrogate the AuctionHouse to find out what the actual bid was.

The *Lapsed Listener* Problem

Say at some point the observer no longer needs to receive updates and just ignores them.

The observer should detach, but let's say it doesn't, either due to programmer oversight or a bug.

This leads to a negligible performance degradation, and possibly a more serious **memory leak**.

> the reference in the subject to the observer prevents the observer from being garbage collected.

A **weak reference** may be an effective solution.

> A weak reference does not prevent the object from being garbage collected. Supported in many languages.

Other Implementation Issues

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.

O*bserving multiple subjects*. Here 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.

Observer Implementation in Java

Java explicitly supports the Observer pattern through `java.util.Observer` and `java.util.Observable`.

The **`Observer`** interface contains the `update` method

```
public void update(Observable o, Object arg)
```

The **`Observable`** class provides a full implementation of Subject behaviour including:

```
addObserver(Observer o)
deleteObserver(Observer o)
notifyObservers()
```

Aside: In Scala we can use a *self-type* to tighten the type of the argument to `update` to be the actual type of what's being observed.

Uses of Observer

First used 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.
- Closely related to the **Publish/Subscribe** pattern commonly used in message-oriented middleware
- Language support in Java, C#, Ruby and others.

All social media applications embody the notion of Observer (e.g. 'follow' on X/LinkedIn etc.).

Related Patterns

Related Patterns include:

- **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.

- How does Observer relate to the SOLID principles?

Observer – Summary

- Observer is a very pervasive design pattern, and is a central part most UI architectures.
- We looked that the pattern in a general way. You'll develop a concrete implementation in practicals.