Design Patterns

Observer

COMP3607
Object Oriented Programming II

Week 7

Outline

- Design Patterns
  - Observer

6 Observer Design Pattern

The Observer pattern is the gold standard in decoupling - the separation of objects that depend on each other.

The Observer is known as a behavioural pattern, as it's used to form relationships between objects at runtime.

6 Observer Design Pattern

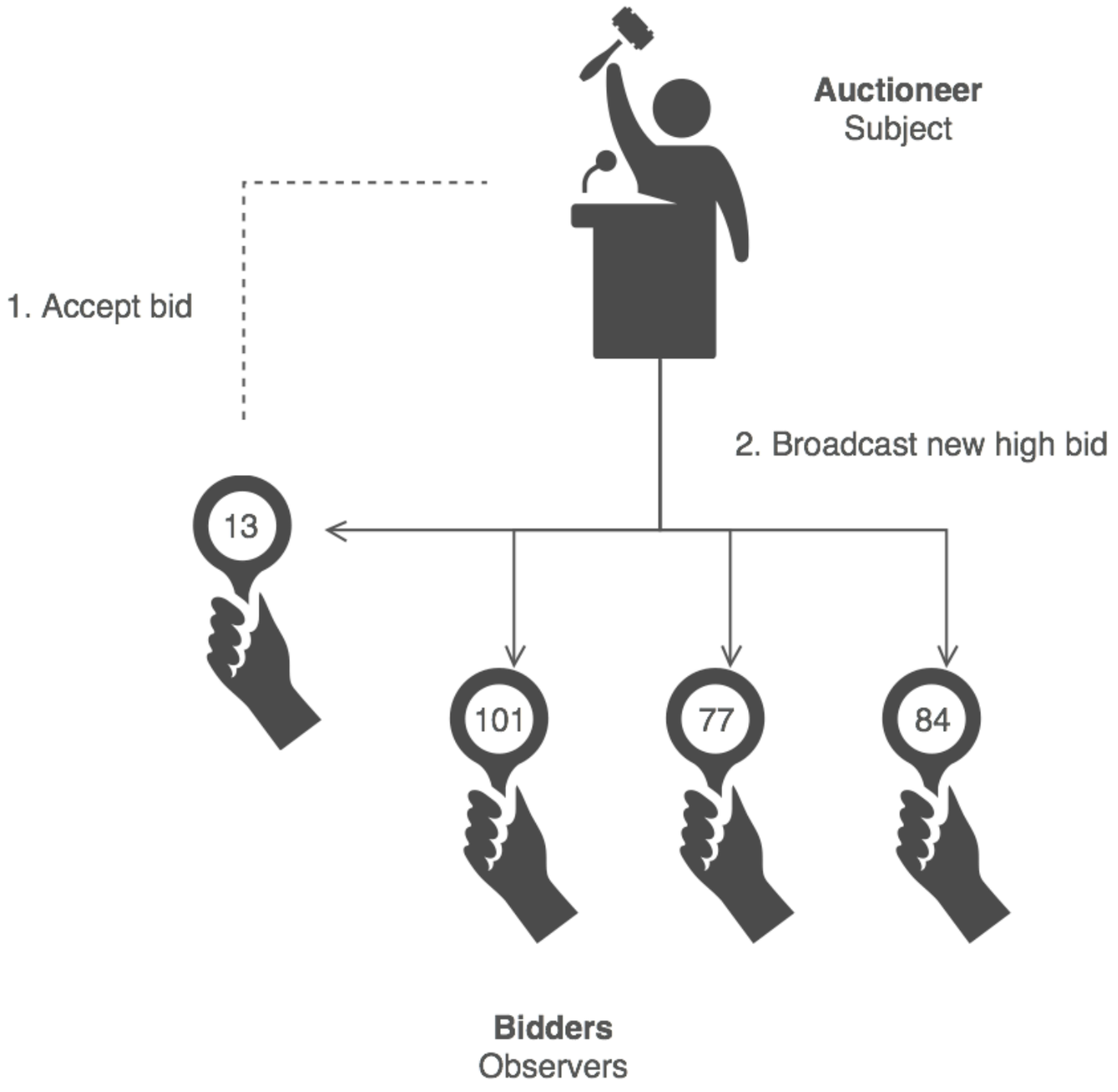
The Observer pattern:

- Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Encapsulates the core (or common or engine) components in a Subject abstraction, and the variable (or optional or user interface) components in an Observer hierarchy.
- Embodies the "View" part of Model-View-Controller.

Example

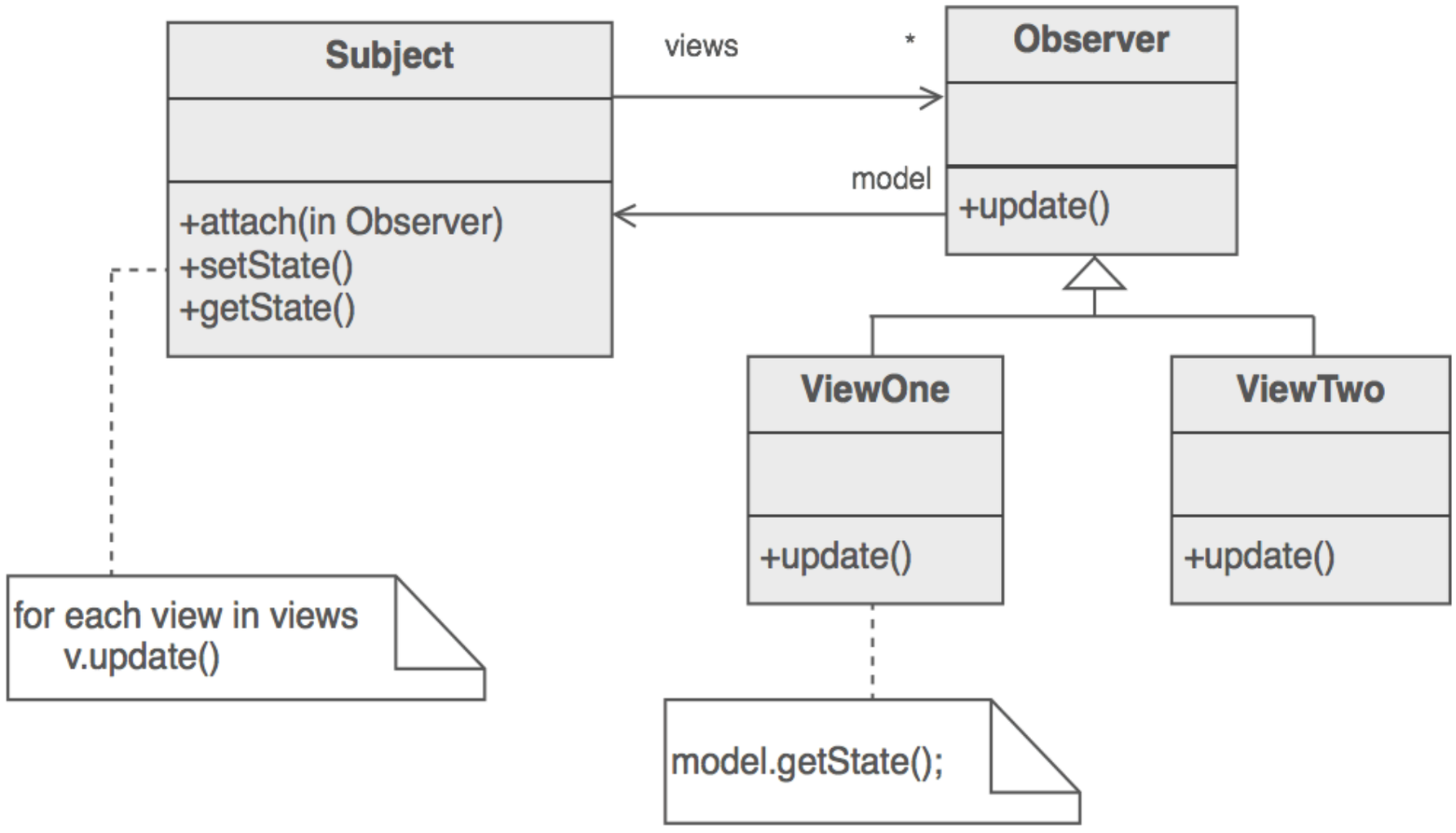
- The Observer defines a one-to-many relationship so that when one object changes state, the others are notified and updated automatically.
- Some auctions demonstrate this pattern.
- Each bidder possesses a numbered paddle that is used to indicate a bid.
- The auctioneer starts the bidding, and "observes" when a paddle is raised to accept the bid.
- The acceptance of the bid changes the bid price which is broadcast to all of the bidders in the form of a new bid.

Example



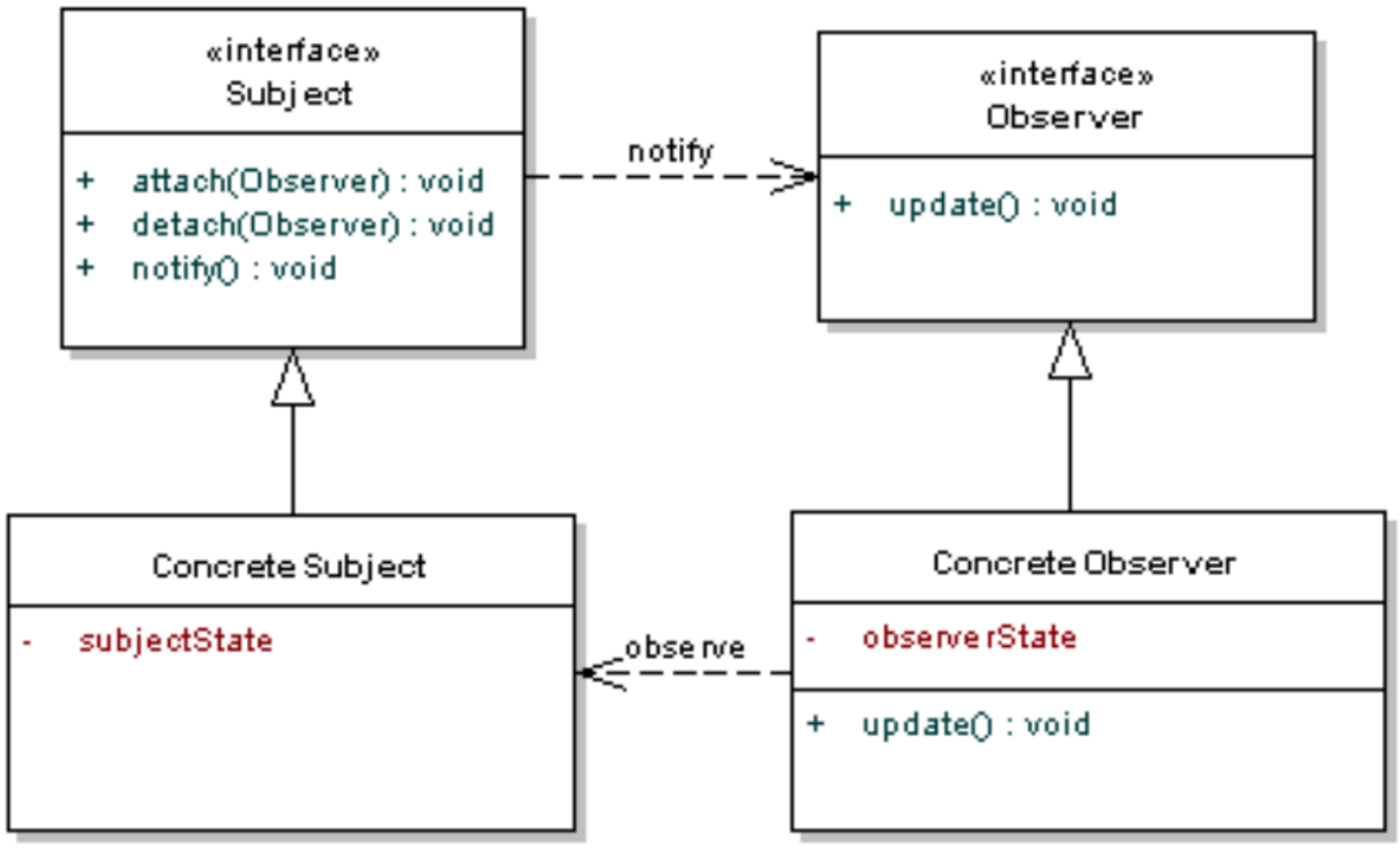
https://sourcemaking.com/design_patterns/observer

Observer (UML) GoF



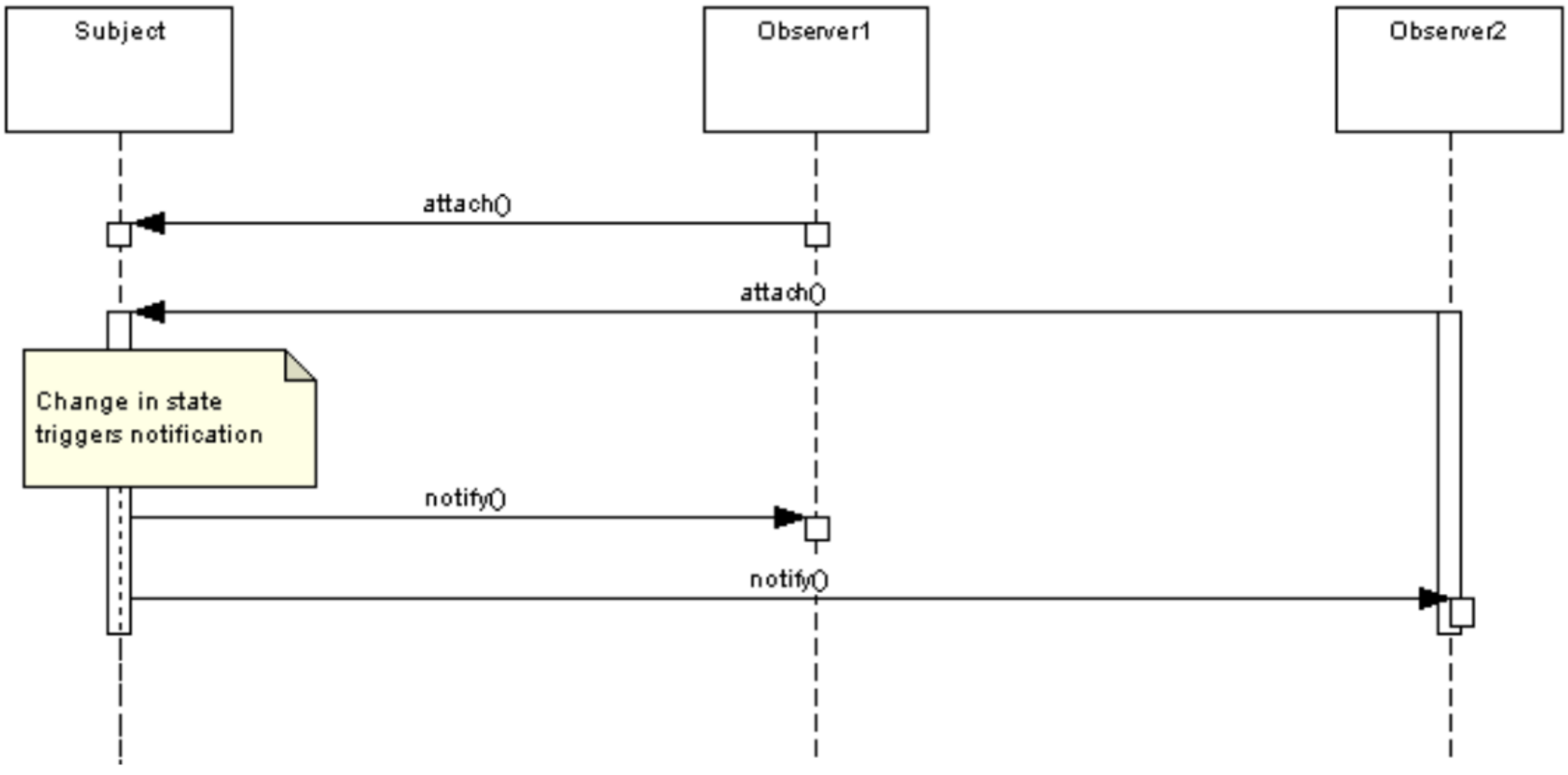
https://sourcemaking.com/design_patterns/observer

Observer (UML)



https://dzone.com/articles/design-patterns-uncovered

Sequence of Events



https://dzone.com/articles/design-patterns-uncovered

Class Inheritance vs Type Inheritance

An object's class defines *how* the object is implemented. In contrast, an object's type only refers to its *interface*.

Class inheritance defines an object's implementation in terms of another object's implementation.

Type inheritance describes when an object can be used in place of another.

Scenario

Consider a sensor system that consists of lighting, gates and surveillance equipment. When the sensor system detects a threat, the lights can be turned on, the gates can be shut, and the surveillance equipment can record video of the scene.

SensorSystem is the “subject” in the scenario.

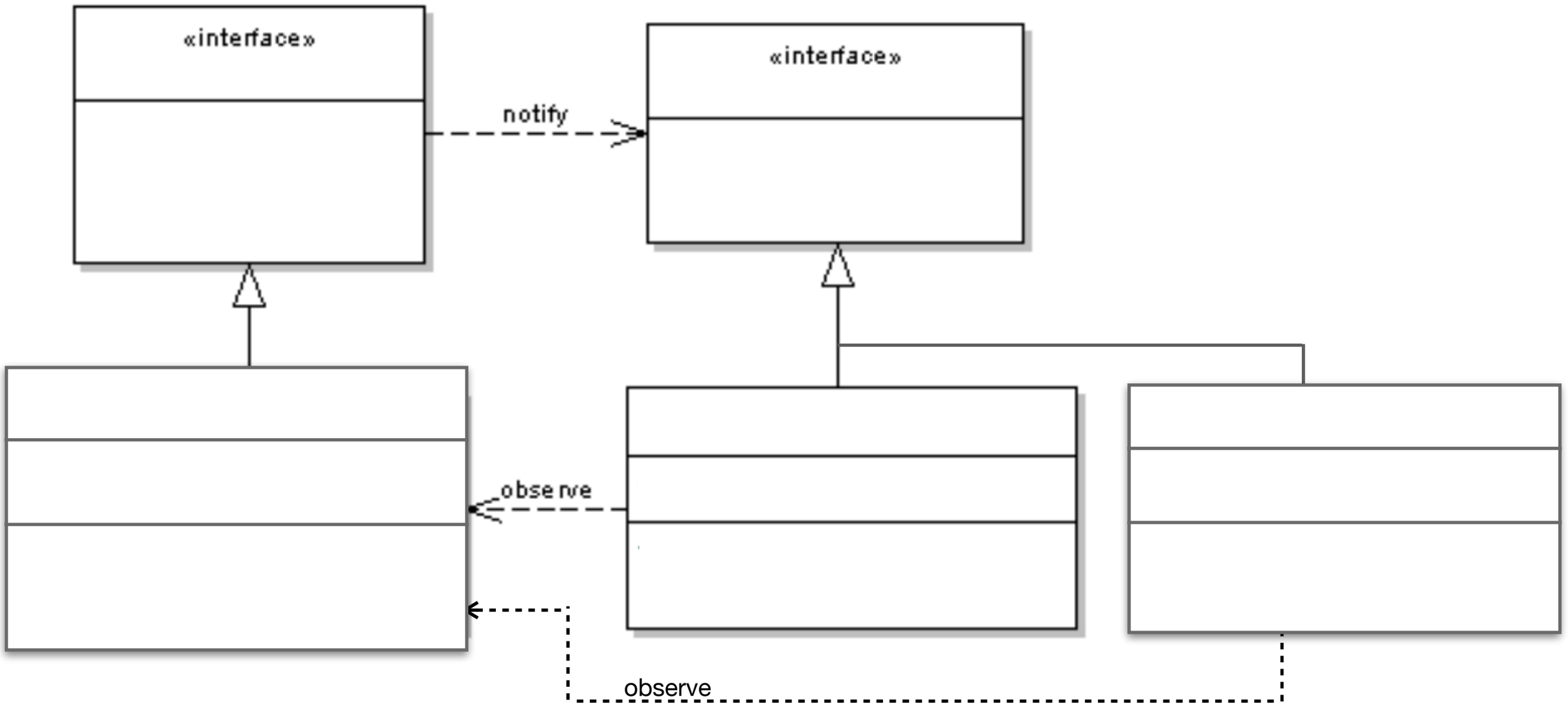
Lighting, Gates, and Surveillance are the "views". These can be types of alarm listeners.

The subject, SensorSystem, is therefore only coupled to the "abstraction" of the views, or AlarmListeners.

Exercise

10 mins

Exercise: Use the Observer Design Pattern to fill in the diagram for the sensor scenario. The *Gates* and *Lighting* are concrete observers of type *AlarmListener*, and the *SensorSystem* is the concrete subject. The *SensorSystem* attaches observers through a *register* method and notifies them through a *soundAlarm* method. The concrete observers 'alarm' when notified of a change in the concrete subject.



Code

```
interface AlarmListener {
    void alarm();
}
```

Observer Interface

```
class Lighting implements AlarmListener {
    public void alarm() {
        System.out.println("lights up");
    }
}

class Gates implements AlarmListener {
    public void alarm() {
        System.out.println("gates close");
    }
}
```

Concrete Observers

(respond to changes in the sensor system by triggering an alarm)

https://dzone.com/articles/design-patterns-uncovered

Code

```
interface Subject{
    void register (Alarm Listener al);
    void soundAlarm( );
}
```

Subject Interface

```
class SensorSystem  implements Subject{
    private Vector listeners = new Vector();    //Collection of Observers

    public void register(AlarmListener alarmListener) {    //attaches observers
        listeners.addElement(alarmListener);
    }

    public void soundTheAlarm() {
        for (Enumeration e = listeners.elements(); e.hasMoreElements();) { //loop through
            ((AlarmListener) e.nextElement()).alarm();                      //notify observers
        }
    }
}
```

Concrete Subject

https://dzone.com/articles/design-patterns-uncovered

Code

```
public class ObserverDemo {
    public static void main( String[] args ) {
        SensorSystem sensorSystem = new SensorSystem();
        sensorSystem.register(new Gates());
        sensorSystem.register(new Lighting());

        sensorSystem.soundTheAlarm();
    }
}
```

//create and attach observers

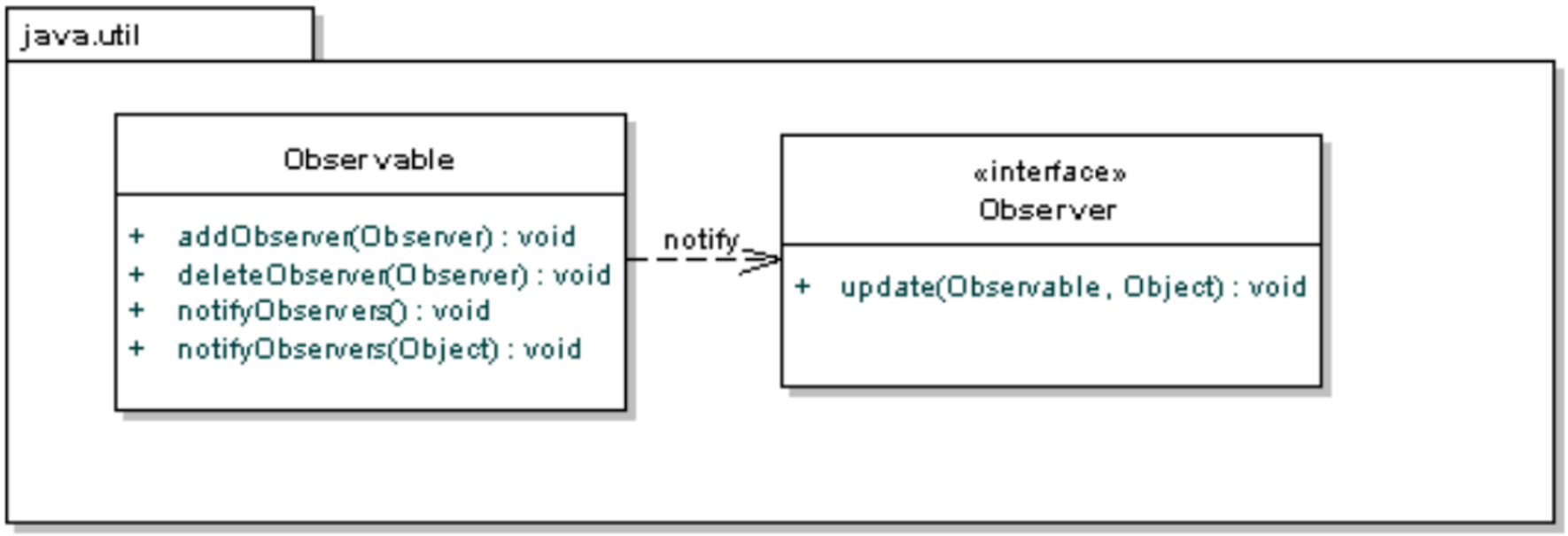
//notify observers

Output

```
gates close
lights up
```

https://dzone.com/articles/design-patterns-uncovered

Observable Interface



While some patterns require you to define the interfaces that make the pattern work, the Observer is a case where Java has the work done for you already in the java.util package.

The slight difference from the classic definition is that **Observable** is used in place of the Subject and is implemented as a class, while the **Observer** interface remains the same

https://dzone.com/articles/design-patterns-uncovered

Applicability: Observer

- In general, you want to use this pattern to reduce coupling. If you have an object that needs to share it's state with others, without knowing who those objects are, the Observer is exactly what you need.
- The whole concept of listeners is based on this pattern. The event listener is the most popular, where you register an ActionListener to a UI control, such a button, and react to action events using the actionPerformed method. In this case, the ActionListener is the Observer and the button is your Subject. As the button changes state, you can react, if you choose to, in your actionPerformed method.
- The typical real world uses of the pattern all revolve around this type of event handling system.

https://dzone.com/articles/design-patterns-uncovered

6

Consequences: Observer

The great strength, and weakness, of observer is that control passes from the subject to the observer implicitly. You can't tell by reading code that an observer is going to fire, the only way you can see what's happening is to use a debugger. As a result of a complex chain of observers can be a nightmare to figure out, change, or debug as actions trigger other actions with little indication why. As a result, it is strongly recommended that you use observers only in a very simple manner.

https://martinfowler.com/eaaDev/OrganizingPresentations.html#observer-gotchas

Consequences: Observer

- Don't have chains of objects observing other objects, observing other objects. One layer of observer relationships is best (unless you use an Event Aggregator
- Don't have observer relationships between objects in the same layer.
  - Domain objects shouldn't observe other domain objects, presentations should not observer other presentations. Observers are best used across the layer boundary, the classic use is for presentations to observe the domain.

https://martinfowler.com/eaaDev/OrganizingPresentations.html#observer-gotchas

Consequences: Observer

- Another issue for observers lies in memory management.
- Assume we have some screens observing some domain objects.
  - Once we close a screen we want it to be deleted, but the domain objects actually carry a reference to the screen though the observer relationship.
- In a memory-managed environment long lived domain objects can hold onto a lot of zombie screens, resulting in a significant memory leak.
- So it's important for observers to de-register from their subjects when you want them to be deleted.

https://martinfowler.com/eaaDev/OrganizingPresentations.html#observer-gotchas

Consequences: Observer

- A similar issue often occurs when you want to delete the domain object.
  - If you rely on breaking all the links between the domain objects this may not be enough since screens may be observing the domain.
  - In practice this turns out be a problem less frequently as the screen departs and the domain objects lifetime are usually controlled through the data source layer.
- But in general it's worth keeping in mind that observer relationships often hang around forgotten and a frequent cause of zombies. Using an Event Aggregator will often simplify these relationships - while not a cure it can make life easier.

https://martinfowler.com/eaaDev/OrganizingPresentations.html#observer-gotchas

References

- Design Patterns: online reading resources
  - https://sourcemaking.com/design_patterns/observer
  - https://dzone.com/articles/design-patterns-uncovered
  - https://martinfowler.com/eaaDev/OrganizingPresentations.html#observer-gotchas