Dr. Michael Eichberg
Software Engineering
Department of Computer Science
Technische Universität Darmstadt

Software Engineering

The Observer Design Pattern

For details see Gamma et al. in “Design Patterns”

TECHNISCHE UNIVERSITÄT DARMSTADT

Observer Design Pattern

Example / Motivation

From the “Lexi” Case Study

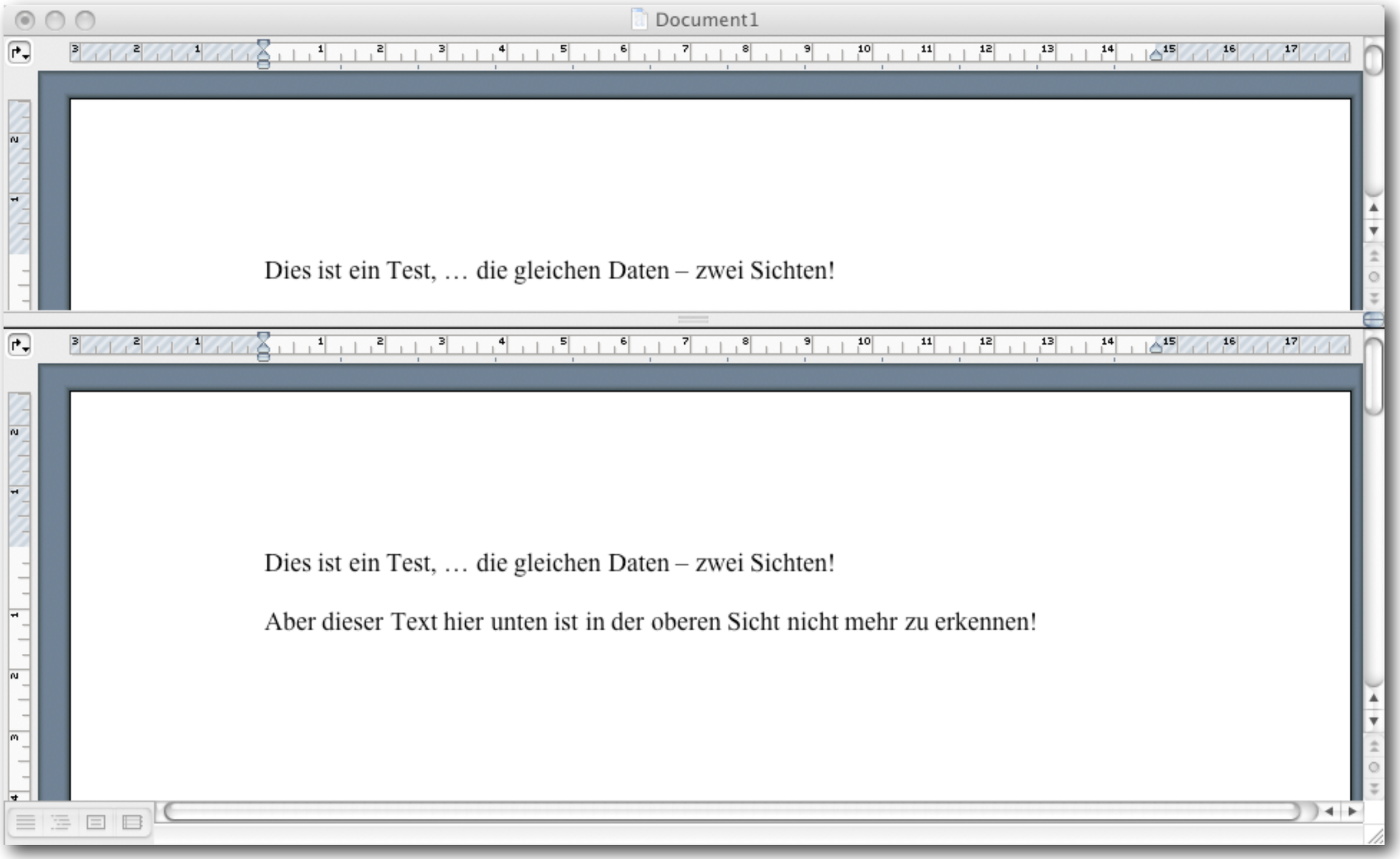
- Presentation components rendering views on the document should be separated from the core document data structures

  Need to establish communication.
- Multiple views on the document should be possible, even simultaneously

  Need to manage updates presenting the document.

Observer Design Pattern

Example / Motivation



Consequences of Object-oriented Programming

Object-oriented programming encourages to break problems apart into objects that have a small set of responsibilities (ideally one)… but can collaborate to accomplish complex tasks.

- Advantage: Makes each object easier to implement and maintain, more reusable, enabling flexible combinations.

- Disadvantage: Behavior is distributed across multiple objects; any change in the state of one object often affects many others.

Observer Design Pattern

Communication without Coupling

- Change propagation (of object states) can be hard wired into objects, but this binds the objects together and diminishes their flexibility and potential for reuse
- A flexible way is needed to allow objects to tell each other about changes without strongly coupling them
- Prototypical Application:
  Separation of the GUI from underlying data, so that classes defining application data and presentations can be reused independently.

Without using the observer pattern

Observer Design Pattern

Communication without Coupling

- **Task**

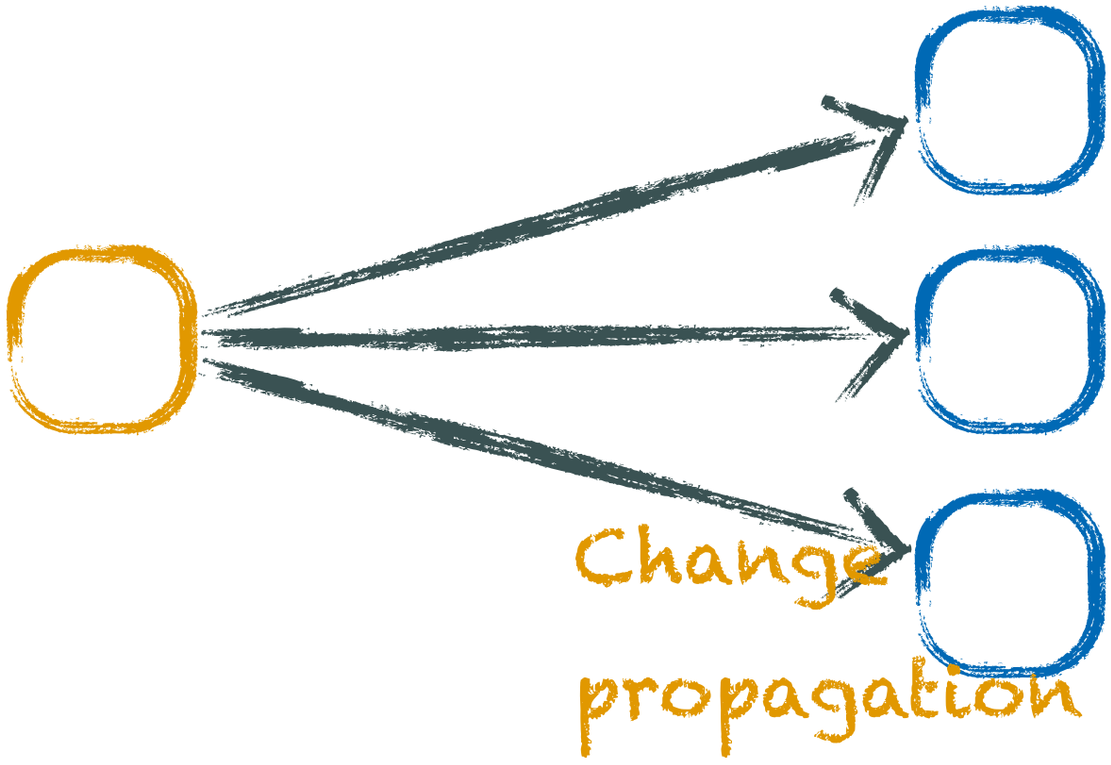
  Decouple a data model (subject) from “parties” interested in changes of its internal state

- **Requirements**
  - subject should not know about its observers
  - identity and number of observers is not predetermined
  - novel receivers classes may be added to the system in the future
  - polling is inappropriate (too inefficient)

Observer Design Pattern

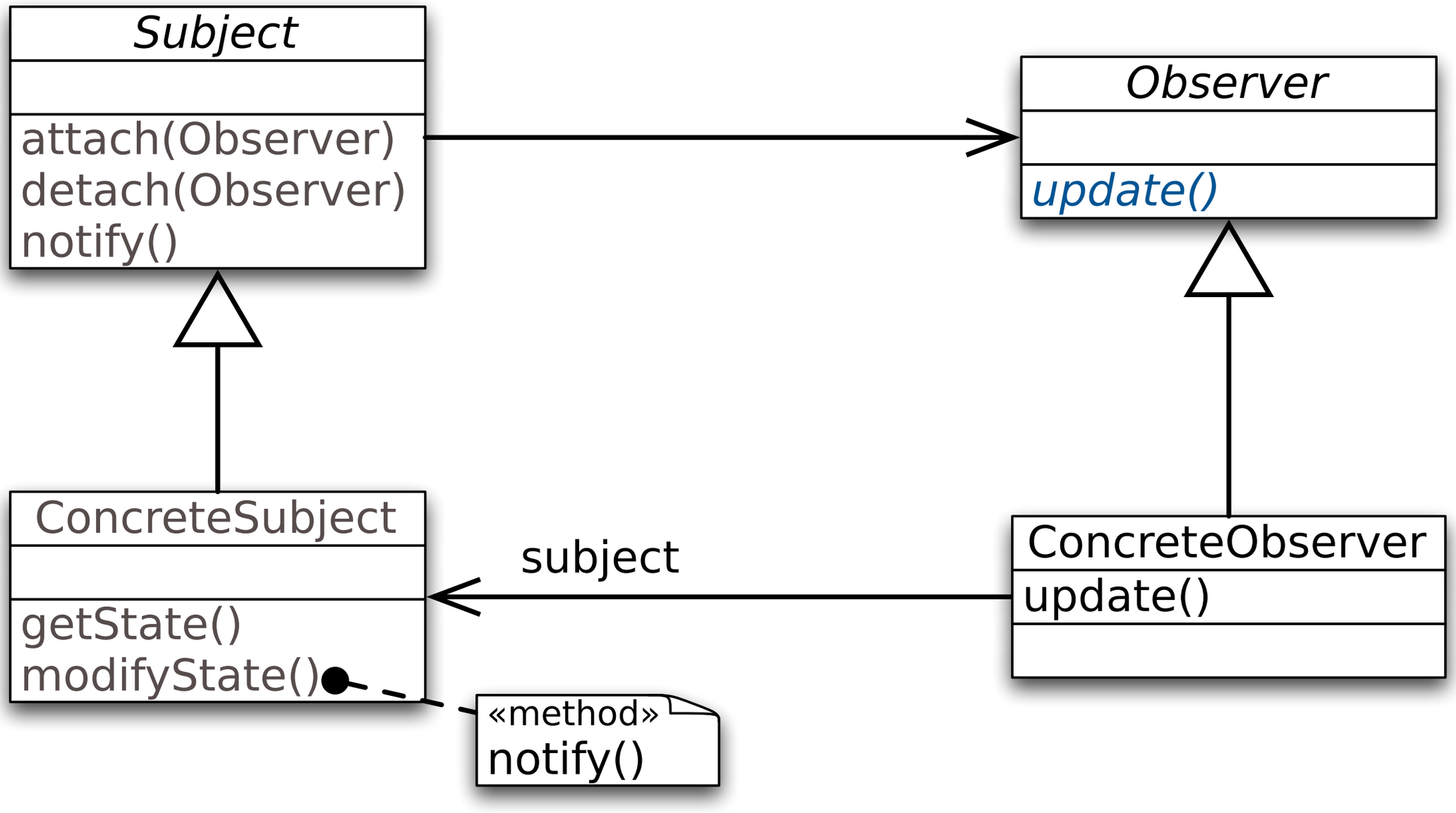
Intent

Define a one-to-many dependency between objects so that when an object changes its state, all its dependents are notified and updated automatically.



Observer Design Pattern

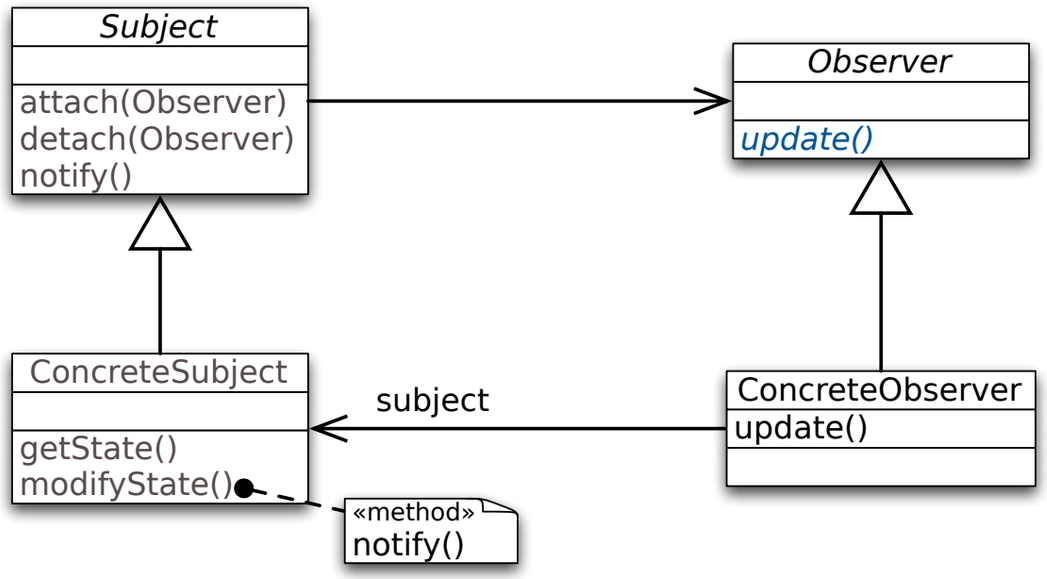
Structure



Observer Design Pattern

Participants

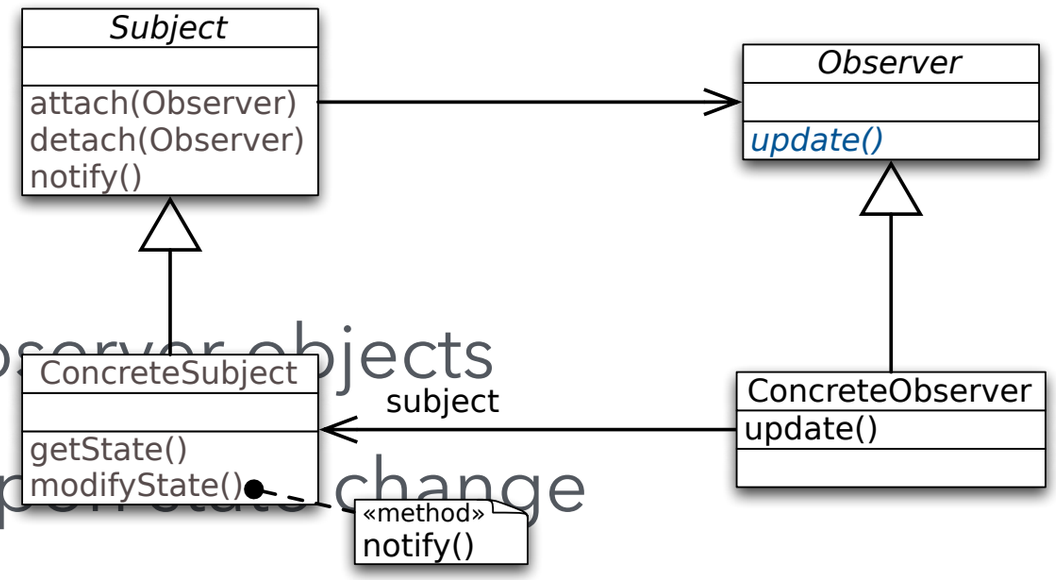
- `Subject`
  - knows its observer(s)
  - provides operations for attaching and detaching Observer objects
- `Observer`
  - defines an updating interface for supporting notification about changes in a Subject
- …



Observer Design Pattern

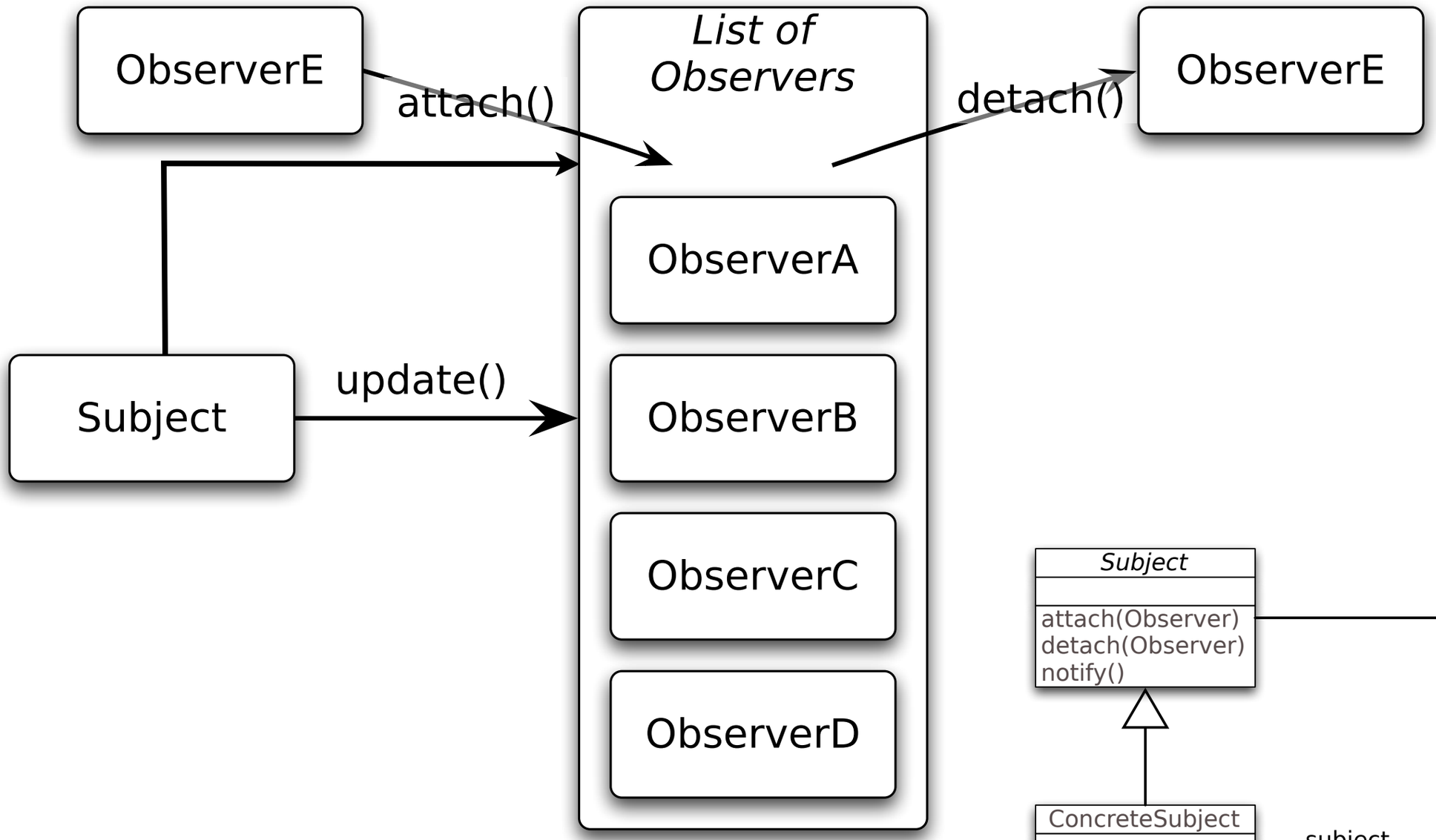
Participants

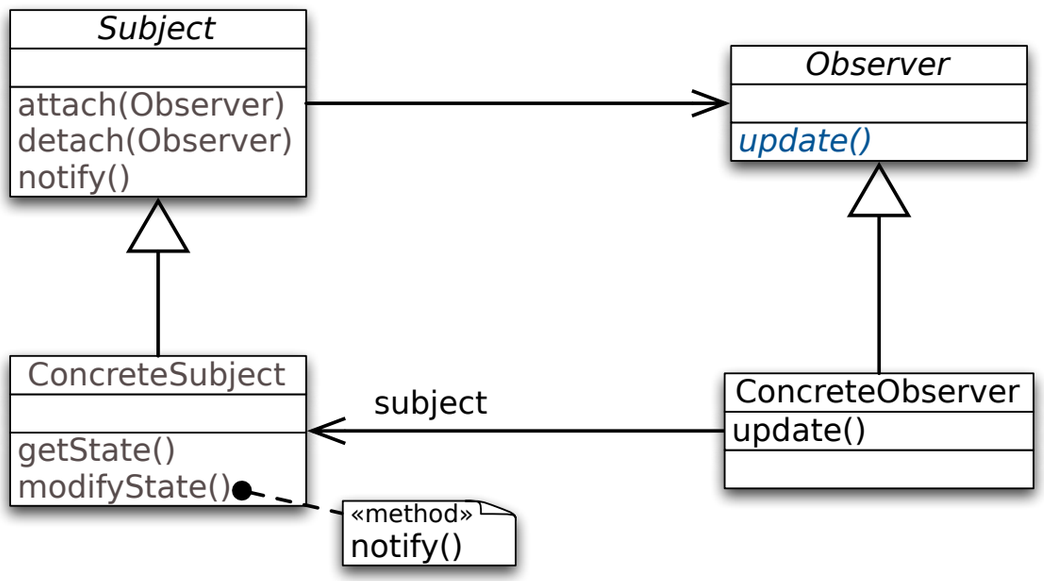
- …
- `ConcreteSubject`
  - stores state of interest to ConcreteObserver objects
  - sends a notification to its observers upon state change
- `ConcreteObserver`
  - maintains a reference to a ConcreteSubject object
  - stores state that should stay consistent with the subject
  - implements the Observer updating interface



Observer Design Pattern

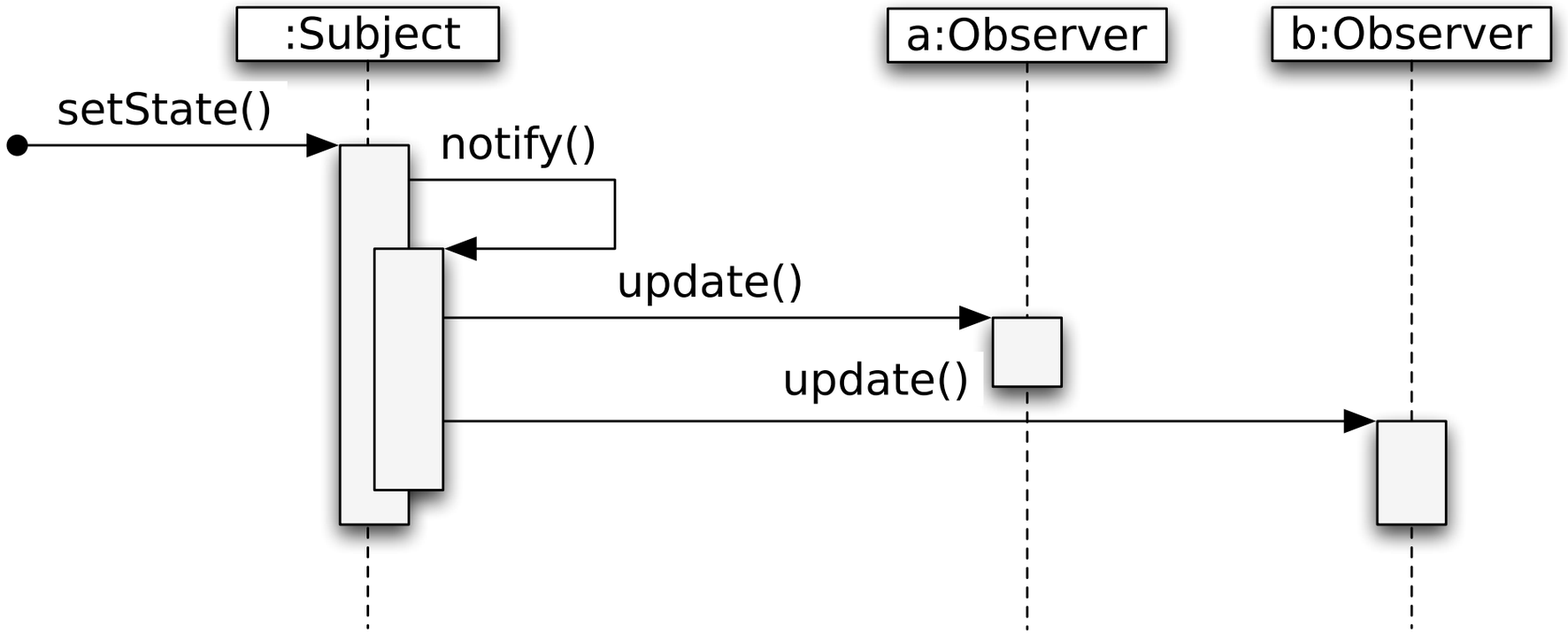
Protocol

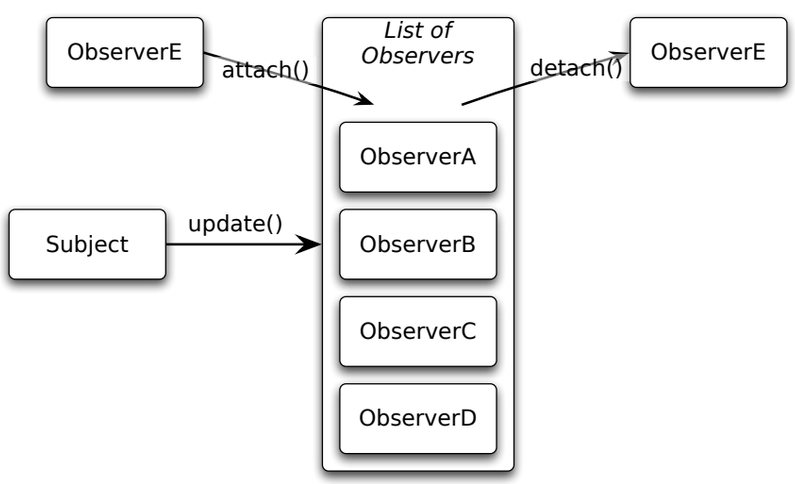


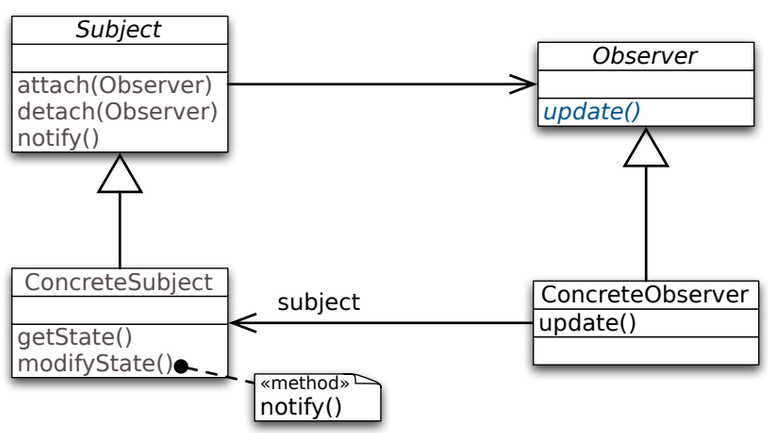


Observer Design Pattern

Interaction







Observer Design Pattern

Consequences

- Abstract coupling between `Subject` and `Observer`
- Support for broadcast communication:
  - notify doesn't specify its receiver
  - the sender doesn't know the (concrete) type of the receiver

Observer Design Pattern

Consequences

- Unexpected / Uncontrolled updates
  - Danger of update cascades to observers and their dependent objects
  - Update sent to all observers, even though some of them may not be interested in the particular change
  - No detail of what changed in the subject; observers may need to work hard to figure out what changed
  - A common update interface for all observers limits the communication interface: Subject cannot send optional parameters to Observers

Observer Design Pattern

"Implementation" - abstract class java.util.Observable

- `addObserver(Observer)` Adds an observer to the observer list
- `clearChanged()` Clears an observable change
- `countObservers()` Counts the number of observers
- `deleteObserver(Observer)` Deletes an observer from the observer list
- `deleteObservers()` Deletes observers from the observer list
- `hasChanged()` Returns a true Boolean if an observable change has occurred
- `notifyObservers()` Notifies all observers about an observable change
- `notifyObservers(Object)` Notifies all observers of the specified observable change which occurred
- `setChanged()` Sets a flag to note an observable change

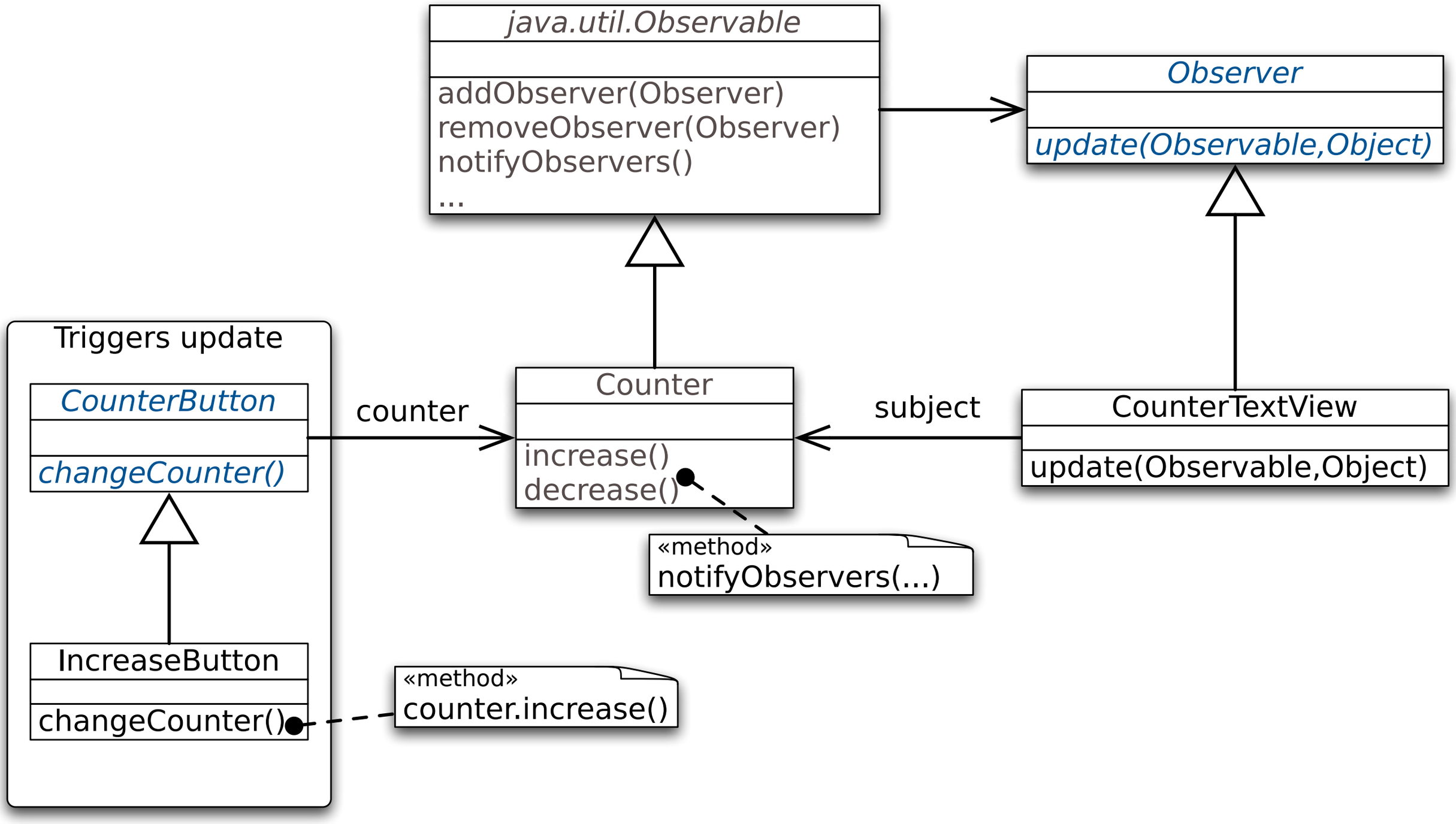
Observer Design Pattern

"Implementation" - interface java.util.Observer

- `void update(Observable o, Object arg)`
  This method is called whenever the observed object is changed. An application calls an observable object's notifyObservers method to have all the object's observers notified of the change.
  Parameters:
  - `o` - the observed object.
  - `arg` - an argument passed to the `notifyObservers` method.

Observer Design Pattern

Example - A Counter, a Controller and a View



Observer Design Pattern

Example - **A Counter**, a Controller and a View


```
class Counter extends java.util.Observable{
   public static final String INCREASE = "increase";
   public static final String DECREASE = "decrease";
   private int count = 0; private String label;
   public Counter(String label) { this.label= label; }
   public String label() { return label; }
   public int value() { return count; }
   public String toString(){ return String.valueOf(count); }
   public void increase() {
      count++;
      setChanged(); notifyObservers(INCREASE);
   }
   public void decrease() {
      count--;
      setChanged(); notifyObservers(DECREASE);
}  }
```

Observer Design Pattern

Example - A Counter, **a Controller** and a View


```
abstract class CounterButton extends Button {

   protected Counter counter;

   public CounterButton(String buttonName, Counter counter) {
      super(buttonName);
      this.counter = counter;
   }

   public boolean action(Event processNow, Object argument) {
      changeCounter();
      return true;
   }

   abstract protected void changeCounter();
}
```

Observer Design Pattern

Example - A Counter, **a Controller** and a View


```
abstract class CounterButton extends Button {

    protected Counter counter;
    public CounterButton(String buttonName, Counter counter) {
        super(buttonName);
        this.counter = counter;
    }
    public boolean action(Event processNow, Object argument) {
        changeCounter();
        return true;
    }

    abstract protected void changeCounter();
}

class IncreaseButton extends CounterButton{
   public IncreaseButton(Counter counter) {
      super("Increase", counter);
   }
   protected void changeCounter() { counter.increase(); }
}
class DecreaseButton extends CounterButton{/* correspondingly… */}
```

Observer Design Pattern

Example - A Counter, a Controller and **a View**


```
class CounterTextView implements Observer{
   Counter model;
   public CounterTextView(Counter model) {
      this.model= model;
      model.addObserver(this);
   }
   public void paint(Graphics display) {
      display.drawString(
        "The value of "+model.label()+" is"+model,1,1
      );
   }

   public void update(Observable counter, Object argument) {
      repaint();
   }
}
```

Observer Design Pattern

Implementation Issues - Triggering the Update

Methods that change the state, trigger update

However, if there are several changes at once, one may not want each change to trigger an update. It might be inefficient or cause too many screen updates

```
class Counter extends Observable {
      public void increase() {
         count++;
         setChanged();
         notifyObservers();
      }
}
```

Clients trigger the update

```
class Counter extends Observable {
      public void increase() {
         count++;
      }
}
class Client {
      public void main() {
         Counter hits = new Counter();
         hits.increase();
         hits.increase();
         hits.setChanged();
         hits.notifyObservers();
      }
}
```

Observer Design Pattern

Implementation Issues:

Passing Information Along with the Update Notification

Pull Mode

Observer asks Subject what happened

```
class Counter extends Observable {
   private boolean increased = false;
   boolean isIncreased() { return increased; }
   void increase() {
      count++;
      increased=true;
      setChanged();
      notifyObservers();
   }
}
class IncreaseDetector extends Counter implements Observer {
   void update(Observable subject) {
      if(((Counter)subject).isIncreased()) increase();
   }
}
```

Observer Design Pattern

Implementation Issues:

Passing Information Along with the Update Notification

Push Mode

Parameters are added to update

```
class Counter extends Observable {
   void increase() {
      count++;
      setChanged();
      notifyObservers(INCREASE);
   }
}
class IncreaseDetector extends Counter implements Observer {
   void update(Observable whatChanged, Object message) {
      if(message.equals(INCREASE)) increase();
   }
}
```

The Observer Design Pattern

Implementation Issues:

Ensure that the Subject State is Self-consistent before Notification


```
class ComplexObservable extends Observable {
   Object o = new Object();
   public void trickyChange() {
      o = new Object();
      setChanged();
      notifyObservers();
   }
}

class SubComplexObservable extends ComplexObservable {
   Object anotherO = …;
   public void trickyChange() {
      super.trickyChange(); // causes notification
      anotherO = …;
      setChanged();
      notifyObservers(); // causes another notification
}  }
```

It's tricky, because the subclass overrides this method and calls it.

The Observer Design Pattern

Implementation Issues:

Ensure that the Subject State is Self-consistent before Notification


```
class ComplexObservable extends Observable {
        Object o = new Object();
        public /*final*/ void trickyChange() {
            doTrickyChange();
            setChanged();
            notifyObservers();
        }
        protected void doTrickyChange() {
            o = new Object();
        }
}
class SubComplexObservable extends ComplexObservable {
        Object anotherO = …;
        protected void doTrickyChange() {
            super.doTrickyChange(); // does not cause notification
            anotherO = …;
            setChanged();
            notifyObservers();
}   }
```

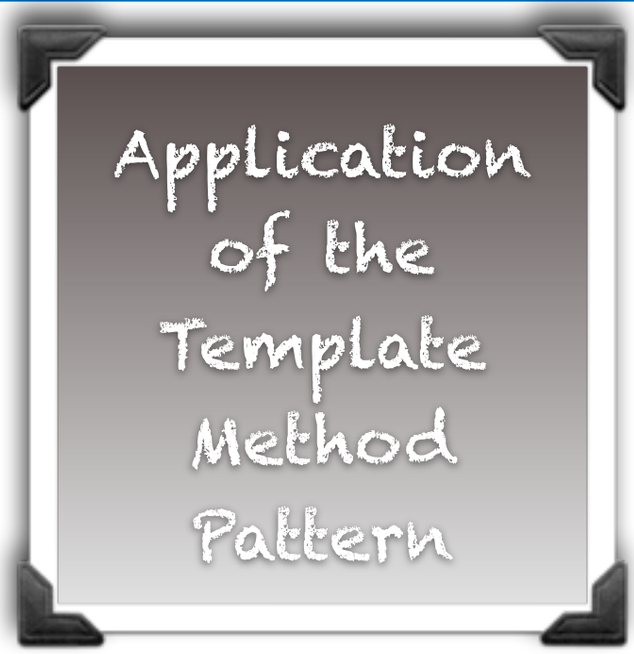
The Observer Design Pattern

Implementation Issues:

Ensure that the Subject State is Self-consistent before Notification


```
class ComplexObservable extends Observable {
    Object o = new Object();
    public /*final*/ void trickyChange() {
        doTrickyChange();
        setChanged();
        notifyObservers();
    }
    protected void doTrickyChange(){
        o = new Object();
    }
}

class SubComplexObservable extends ComplexObservable {
    Object another0 = …;
    public void doTrickyChange() {
        super.doTrickyChange();
        another0 = …;
}   }
```



The Observer Design Pattern

Implementation Issues:
Specifying Modifications of Interest

- The normal `addObserver(Observer)` method is extended to enable the specification of the kind of events the Observer is interested in
- E.g. `addObserver(Observer, Aspect)` where Aspect encodes the type of events the observer is interested in
- When the state of the Subject changes the Subject sends itself a message `triggerUpdateForEvent(anAspect`