TECHNISCHE UNIVERSITÄT DARMSTADT

A Sound Approximation of the

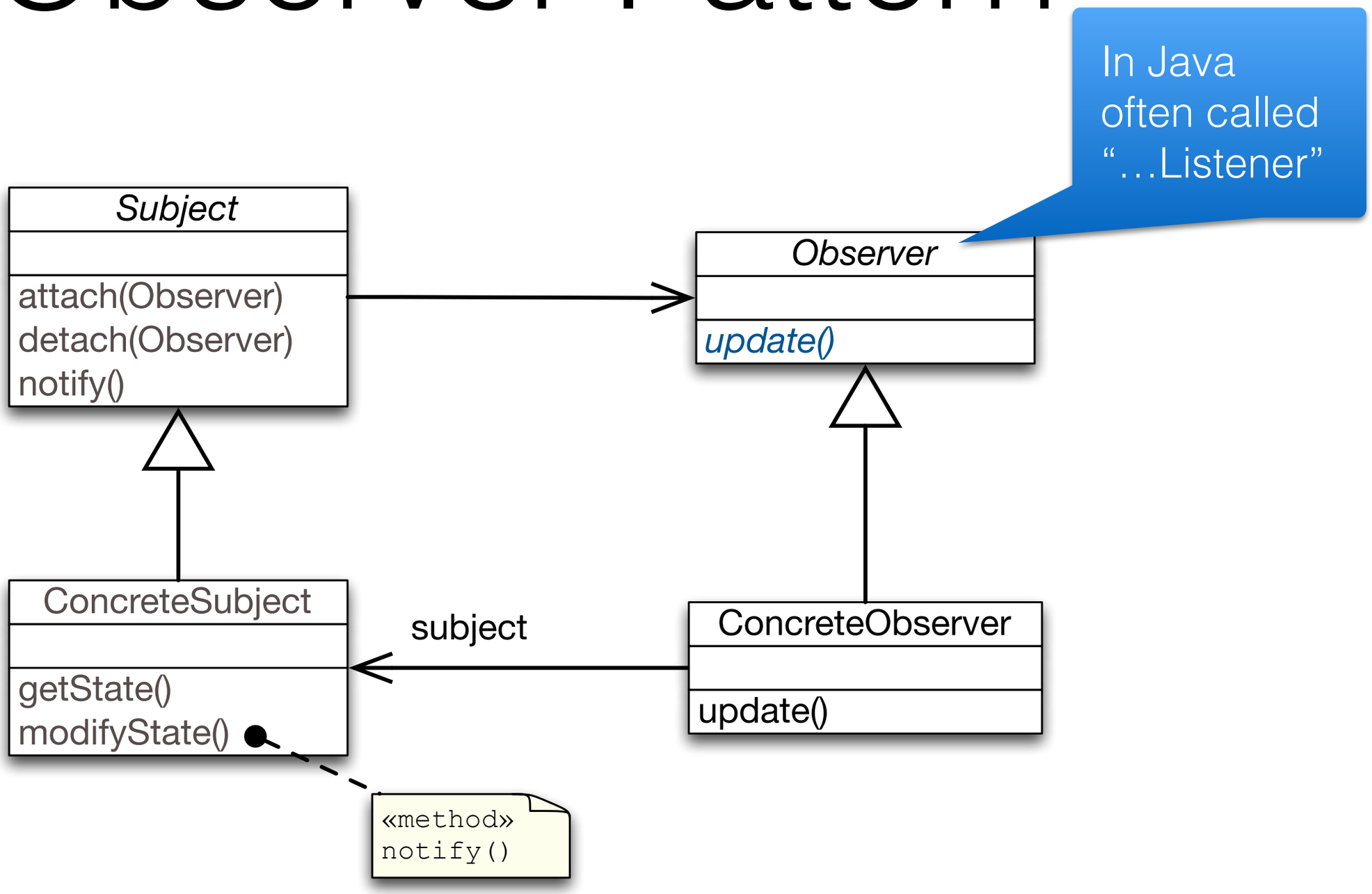
Prevalence of the Observer Design Pattern

(in Java Applications)

BATAI

Software Technology Group
Dr. Michael Eichberg
eichberg@informatik.tu-darmstadt.de
http://bitbucket.org/delors/bat

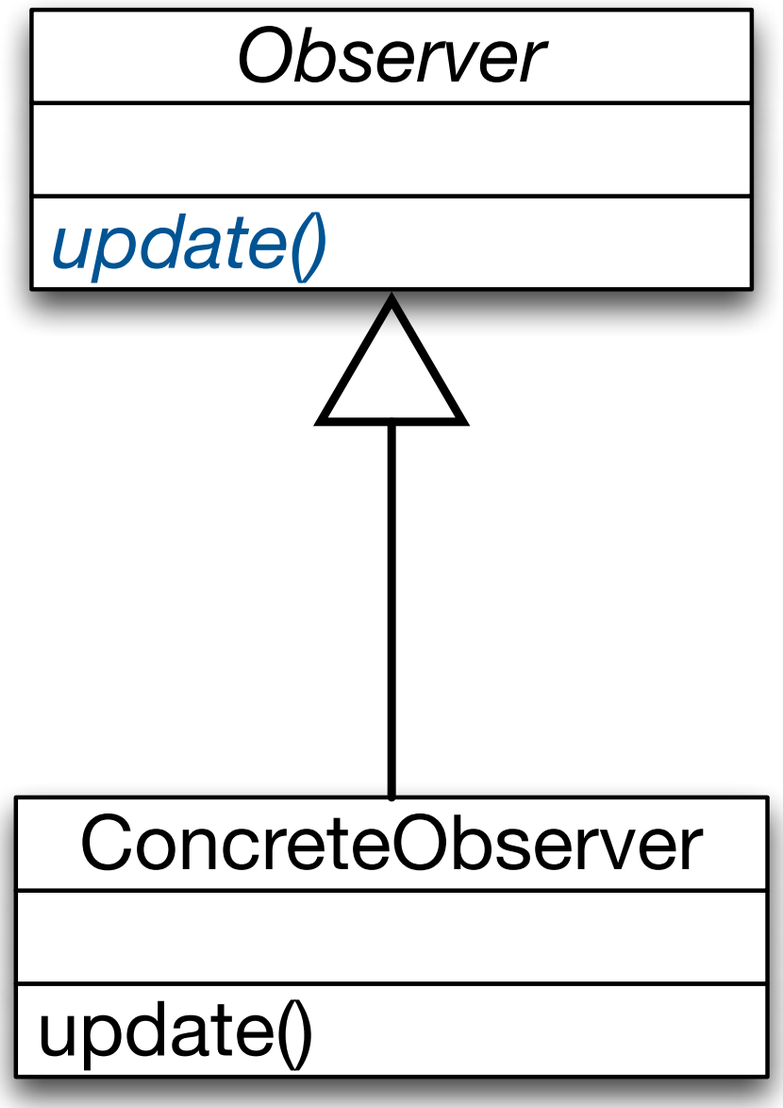
Observer Pattern



Identifying Observers

(i.e., classes that react on some event that happens somewhere else)

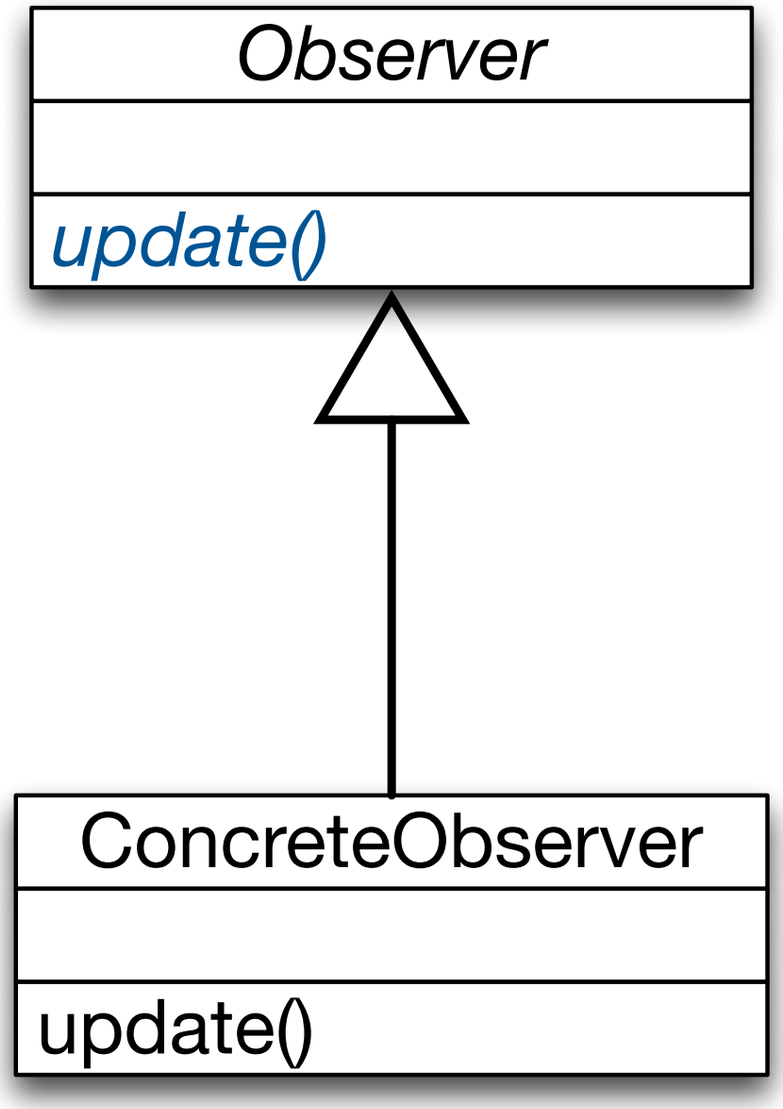
- interfaces that ends with “Listener” or “Observer”; in particular:
  - the interface `java.util.EventListener`
  - the interface `java.util.Observer`
- *ObserverInterfaces = {* all interfaces that are subtypes of the interfaces identified using the above approaches *}*



Identifying Observers

(i.e., classes that react on some event that happens somewhere else)

- *AllObservers = {* all classes that (in-)directly implement an interface in *OberserverInterfaces }*

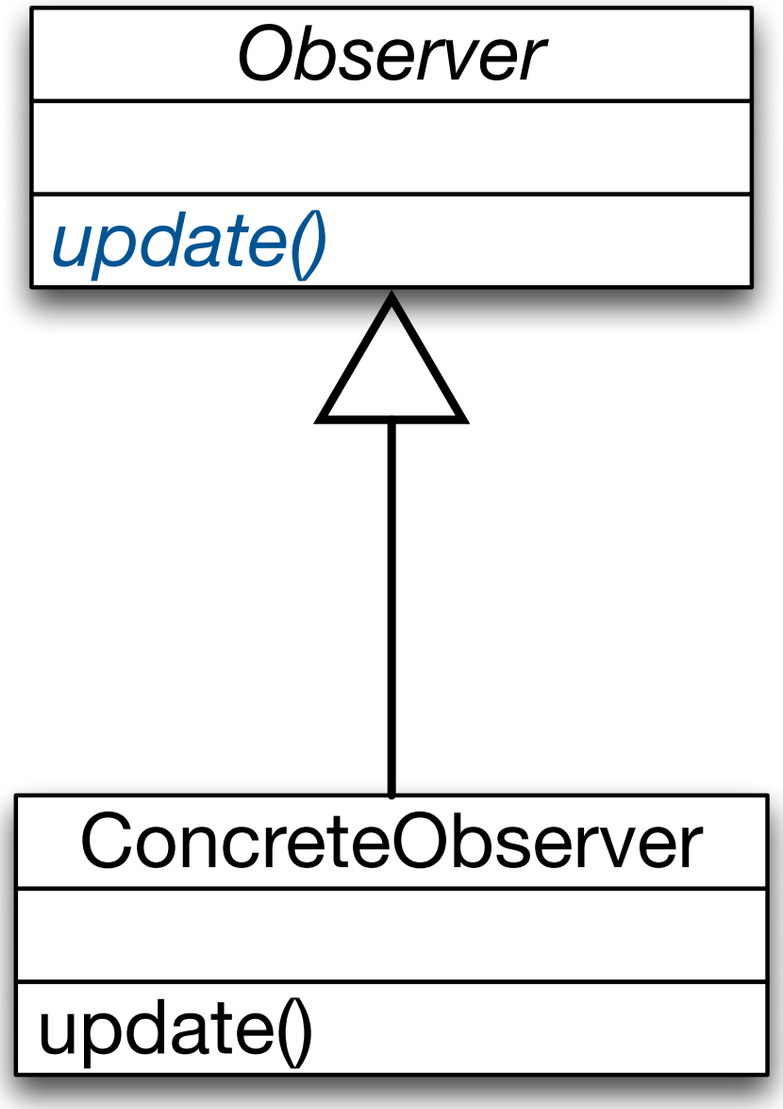


Identifying Update Methods

(methods that are called by the observable to notify the observer)

- *UpdateMethods = {* methods declared by an interface *i* ∈ *ObserverInterfaces }*

- The methods defined by classes that implement an Observer interface are not considered. They are typically not related to the pattern. E.g., the class `javax.swing.JButton` is an Observer (and also Observable), but only the methods defined by the interface `EventListener` are related to it.

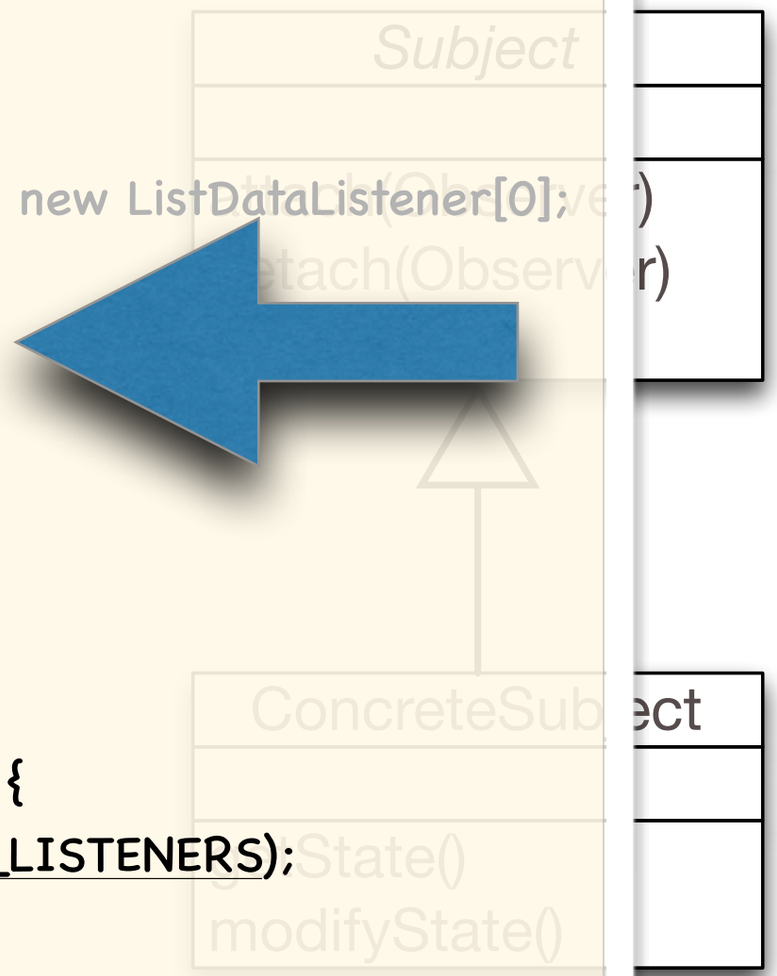


Part I

Managing Observers

(code to manage observers - storing observers)

```
public abstract class AbstractFlashcardSeries
        implements FlashcardSeries {

    public final static ListDataListener[] NO_LISTENERS = new ListDataListener[0];

    private ListDataListener[] listeners = NO_LISTENERS;

    public void addListDataListener(ListDataListener l) {
        this.listeners = Arrays.append(this.listeners, l);
    }

    public void removeListDataListener(ListDataListener l) {
        this.listeners = Arrays.remove(this.listeners, l, NO_LISTENERS);
    }

     :
}
```



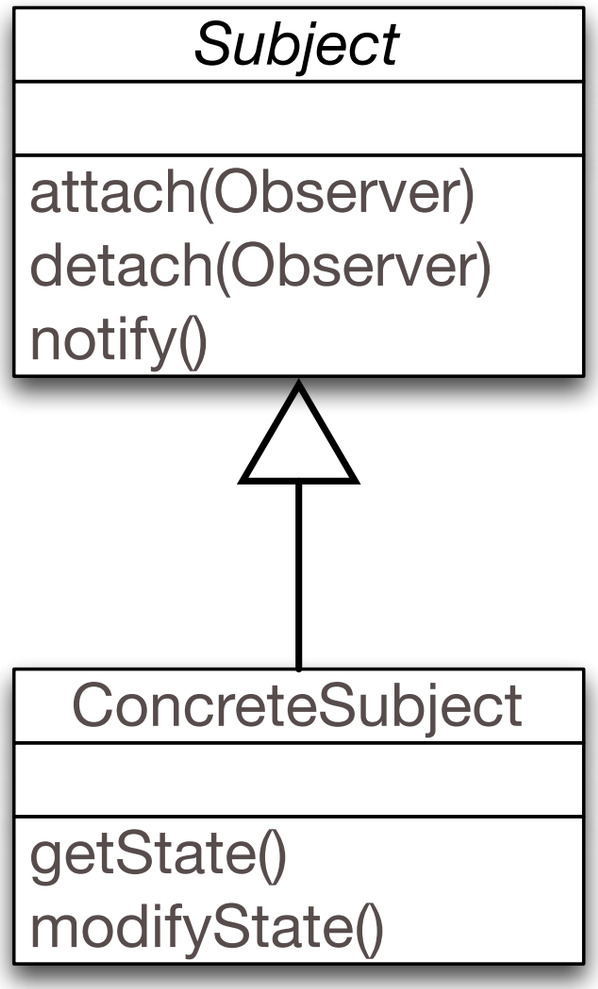
Part I

Managing Observers

(code to manage observers - storing observers)

- *OMCandFields* = { (c,f) | f* is a field of a class ***c*** that has a field with type ***t*** or that is an array of type ***t*** or that has a field that is parameterized using an type ***t*** and ***t*** ∈ *Observers }*

- This enables us to identify subjects that enable the registration of one or more observers (e.g. `List<Observer>`)



* *OMCandFields* ≙ Fields potentially related to the management of observers.

Part I

Managing Observers

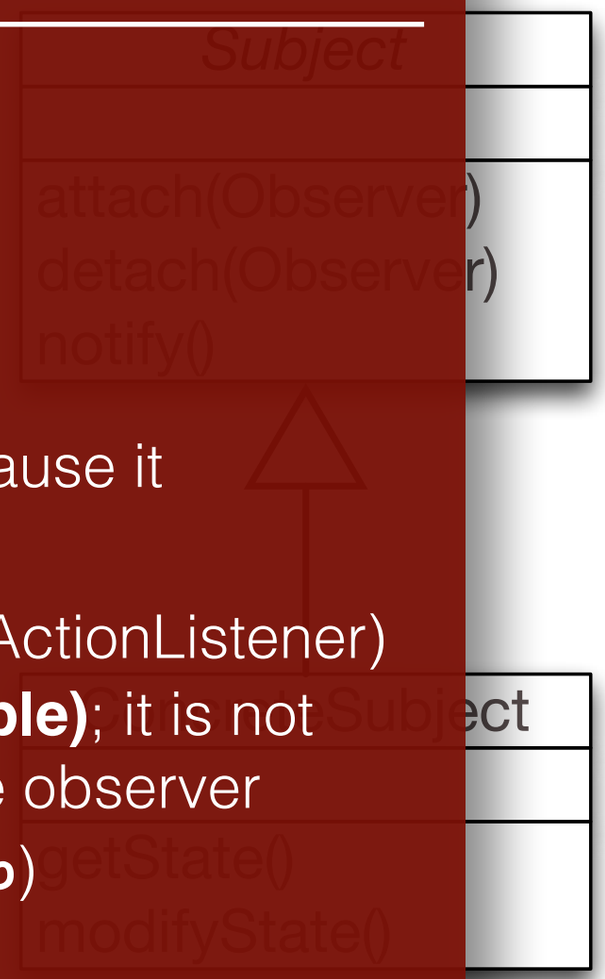
(code to manage observers - storing observers)

- *OMCandFields* = { (c,f) | f is a field of a class c that has a field with type *t* or that is an array of type *t* or that has a field that is parameterized using an type *t* and *t* ∈ *Observer*)
- This enables to identify subjects that have the methods registration/unregistration of observers (e.g. List<Observer>)

Issue

```
class C {
    private JButton b = new JButton();
}
```

- **JButton** is classified as an observer because it implements **java.awt.ImageObserver**
- **JButton** is also observable (add/removeActionListener)
- But, **class C is not a Subject (Observable)**; it is not participating in the implementation of the observer pattern (it does not call back the button **b**)



Part I

Managing Observers

(code to manage observers - storing observers)

- *OMCandFields* = { (c,f) | f is a field of a class c that has a field with type *t* or that is an array of type *t* or that has a field that is parameterized using an type *t* and *t* ∈ *Observer*)

- This enables to identify subjects that enable the registration of one or more observers (e.g. List<Observer>)

Solution

```
class C {
      private JButton b = new JButton();
}
```

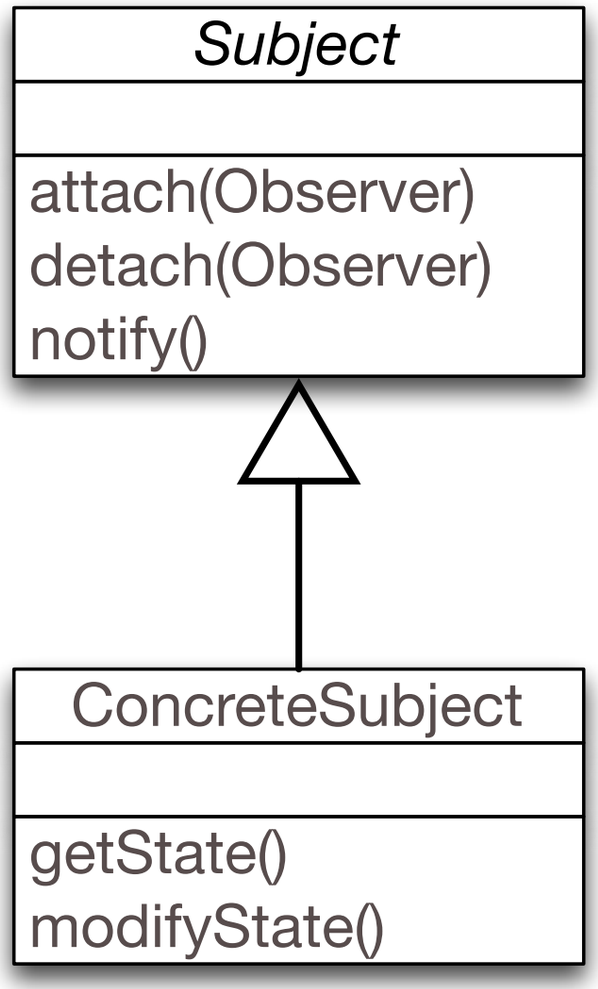
- ignore fields where the type t is not in *ObserverInterfaces*; they are generally not used to make calls to methods in *UpdateMethod.*

Part I

Managing Observers

(code to manage observers - storing observers)

- *OMFields* = { (c,f) | (c,f) ∈ OMCandFields* where the field type *t* is in *OberserverInterfaces }*



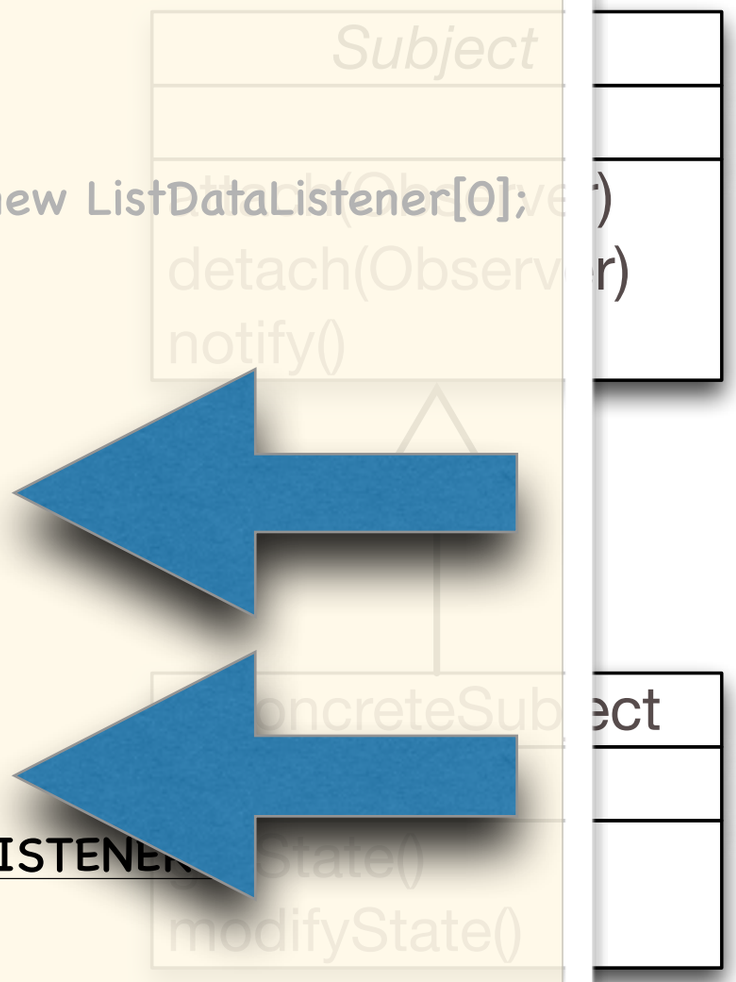
* *OMFields* ≙ Fields that are related to the management of observers.

Part II

Managing Observers

(code to manage observers - registration and notification of observers)

```
public abstract class AbstractFlashcardSeries
        implements FlashcardSeries {

    public final static ListDataListener[] NO_LISTENERS = new ListDataListener[0];

    private ListDataListener[] listeners = NO_LISTENERS;

    public void addListDataListener(ListDataListener l) {
        this.listeners = Arrays.append(this.listeners, l);
    }

    public void removeListDataListener(ListDataListener l) {
        this.listeners = Arrays.remove(this.listeners, l, NO_LISTENERS);
    }

     :
}
```

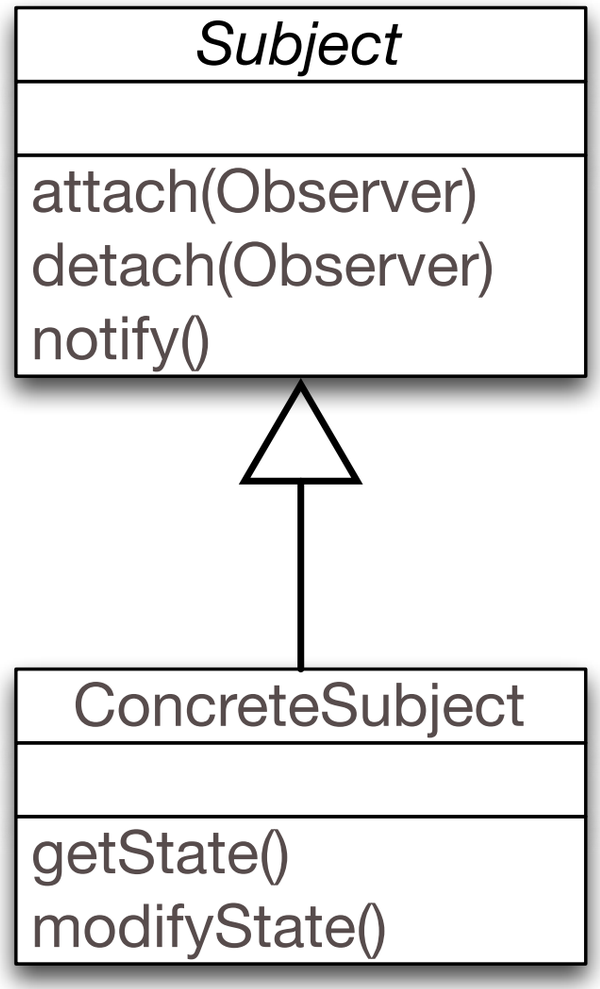


Part II

Managing Observers

(code to manage observers - registration and immediate notification of observers)

- *OMMethods = { (c,m) | m* is a method of a class ***c*** that reads or writes a field ***f*** of that class that is also in *OMFields* }

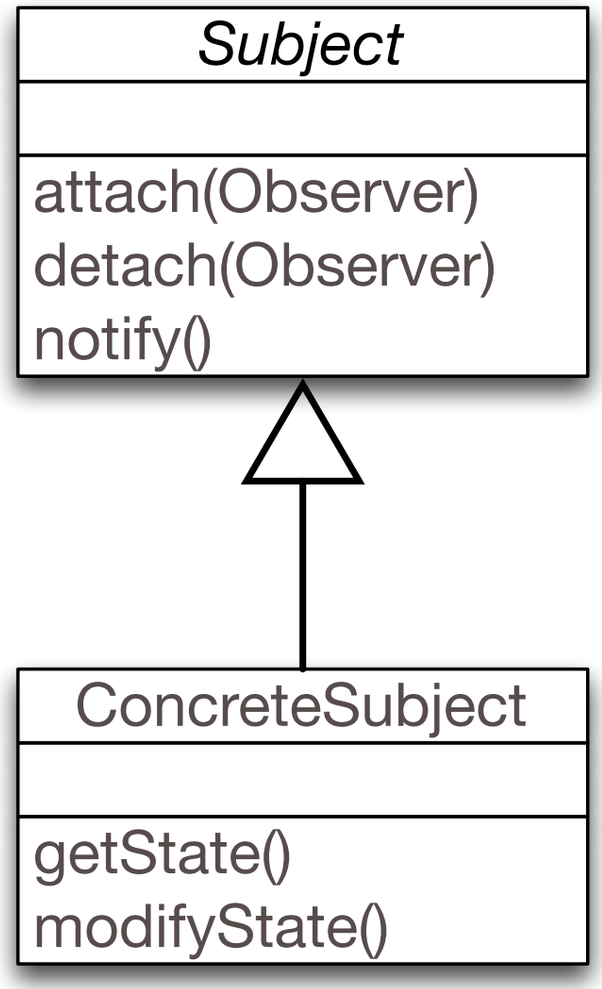


Identifying Observables

(classes that can be observed and will call back observers)

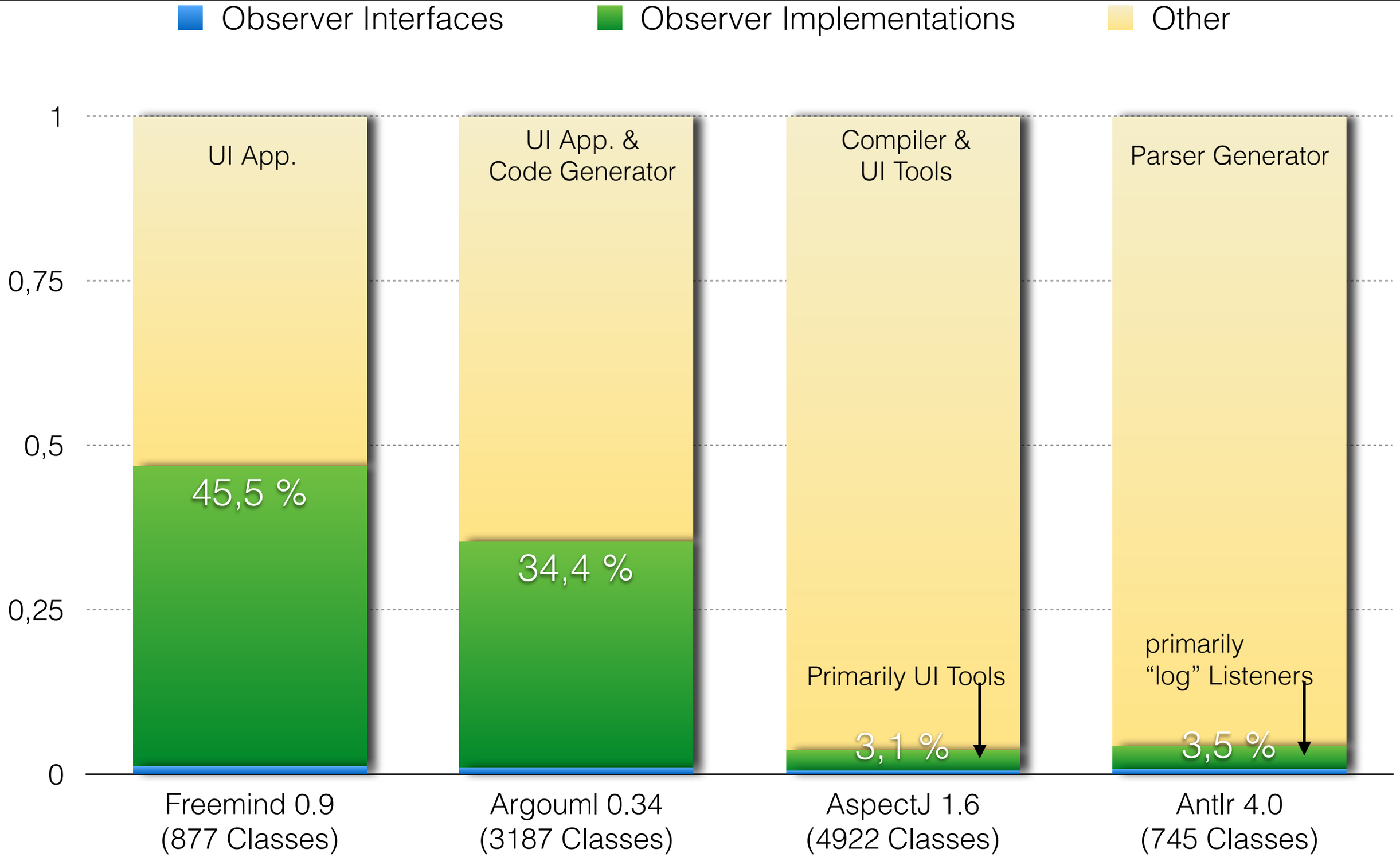
- *Observables* = *{ c | c* has a field *f* that is in *OMField* or *c* implements `java.util.Observable` *}*

- If the subject defines a field: "`List<T> listeners...`", then this class can be considered to be observable.



Overview

- *Key Elements:*
  - *Observers; ObserverInterfaces*
  - *Observables*
  - *OMFields*
  - *OMMethods*
- Next step: estimating the amount of code that is used to instantiate the classes and to call the observers



Classes Directly Related to the Observer Pattern

(Four Applications from the Qualitas Corpus)