Entwurfsmuster

Jan Fässler

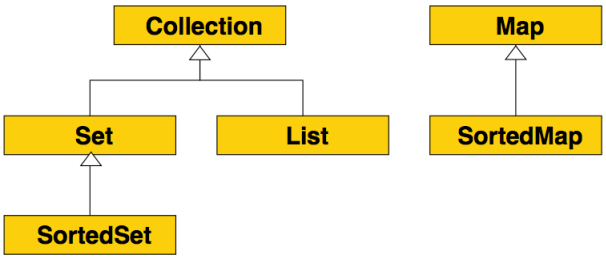
3. Semester (HS 2012)

If you use this documentation for a exam, you should offer a beer to the authors!

Inhaltsverzeichnis

1 Java Collection Framework

1.1 Die Interfaces



1.2 Der Iterator

Ein Iterator ist immer zwischen zwei Elemente. Es gibt also in einer Collection mit n Elementen, n + 1 mögliche Positionen an denen der Iterator stehen kann.

Listing 1: Iterator

```
1 interface Iterator<E> {
2   boolean hasNext();      // there is an element which can be jumped over
3   E next();               // returns the jumped over element
4   void remove();          // removes the last element returned by next
5 }
```

1.3 Die Implementierung

Interface	Implementation				Historical
Set	HashSet		TreeSet		
List		ArrayList		LinkedList	Vector Stack
Map	HashMap		TreeMap		Hashtable Properties

ArrayList
: Eine Implementierung welche ein Array darstellt bei dem man die grösse verändern kann.

LinkedList
: Eine doppelt verkettete Liste.

HashSet
: Die Elemente werden in einer Hash-Tabelle gespeichert.

TreeSet
: Die Elemente werden in einer Baumstruktur gespeichert.

Maps
: Eine Map ist wie ein Wörterbuch aufgebaut. Jeder Eintrag besteht aus einem Schlüssel (key) und dem zugehörigen Wert (value). Jeder Schlüssel darf in einer Map nur genau einmal vorhanden sein.

Wenn eine Klasse ein Interface implementiert, müssen immer alle Funktionen des Interface implementiert werden. In einer abstrakten Collection-Klasse können alle Funktionen bis auf zwei realisiert werden. Für das Hinzufügen und für den Iterator benötigt es Kentnisse über die Datenstruktur. Hier das ein Beispiel einer Abstrakten Klasse:

Listing 2: Abstract Collection

```
1  abstract class AbstractCollection<E> implements Collection<E> {
2    public abstract Iterator<E> iterator();
3    public abstract boolean add(E x);
4    public boolean isEmpty() { return size() == 0; }
5    public int size() {
6      int n = 0; Iterator<E> it = iterator();
7      while (it.hasNext()) {
8        it.next(); n++;
9      }
10     return n;
11   }
12   public boolean contains(Object o) {
13     Iterator<E> e = iterator();
14     while (e.hasNext()) {
15       Object x = e.next();
16       if(x == o || (o != null) && o.equals(x)) return true;
17     }
18     return false;
19   }
20   public void clear() {
21     Iterator<E> it = iterator();
22     while (it.hasNext()) {
23       it.next();
24       it.remove();
25     }
26   }
27   public boolean remove(Object o) {
28     Iterator<E> it = iterator();
29     while (it.hasNext()) {
30       Object x = it.next();
31       if(x == o || (o != null && o.equals(x))) {
32         it.remove();
33         return true;
34       }
35     }
36     return false;
37   }
38   public boolean containsAll(Collection<?> c) {
39     Iterator<?> it = c.iterator();
40     while (it.hasNext()) {
41       if(!contains(it.next())) return false;
42     }
43     return true;
44   }
45   public boolean addAll(Collection<? extends E> c) {
46     boolean modified = false;
47     Iterator<? extends E> it = c.iterator();
48     while (it.hasNext()) {
49       if(add(it.next())) modified = true;
50     }
51     return modified;
52   }
```

2 Design Pattern

Entwurfsmuster (englisch design patterns) sind bewährte Lösungsschablonen für wiederkehrende Entwurfsprobleme sowohl in der Architektur als auch in der Softwarearchitektur und -entwicklung. Sie stellen damit eine wiederverwendbare Vorlage zur Problemlösung dar, die in einem bestimmten Zusammenhang einsetzbar ist.

2.1 Types of Patterns

Software Pattern

- Architectural Pattern (system design)
- Design Pattern (micro architectures)
- Coding Pattern / Idioms (low level)

Analysis Pattern

- Recurring & reusable analysis models used in requirements engineering

Organizational Patterns

- Structure of organizations & projects: XP, SCRUM

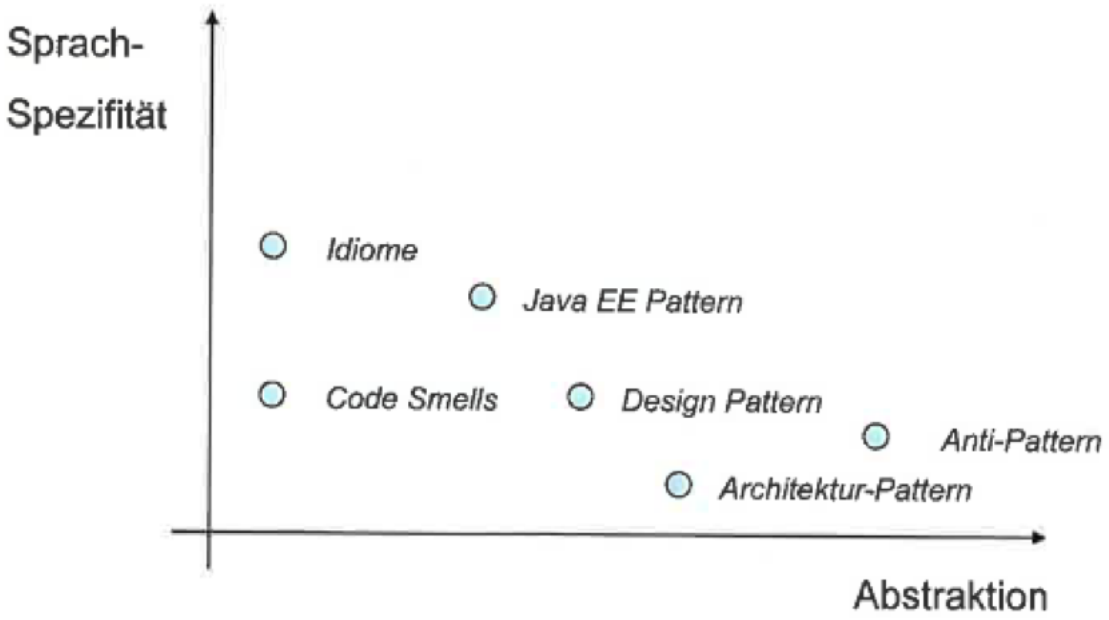
Domain-Specific patterns

- UI patterns, security patterns

Anti-Patterns

- Refactoring

2.2 Pattern Classification



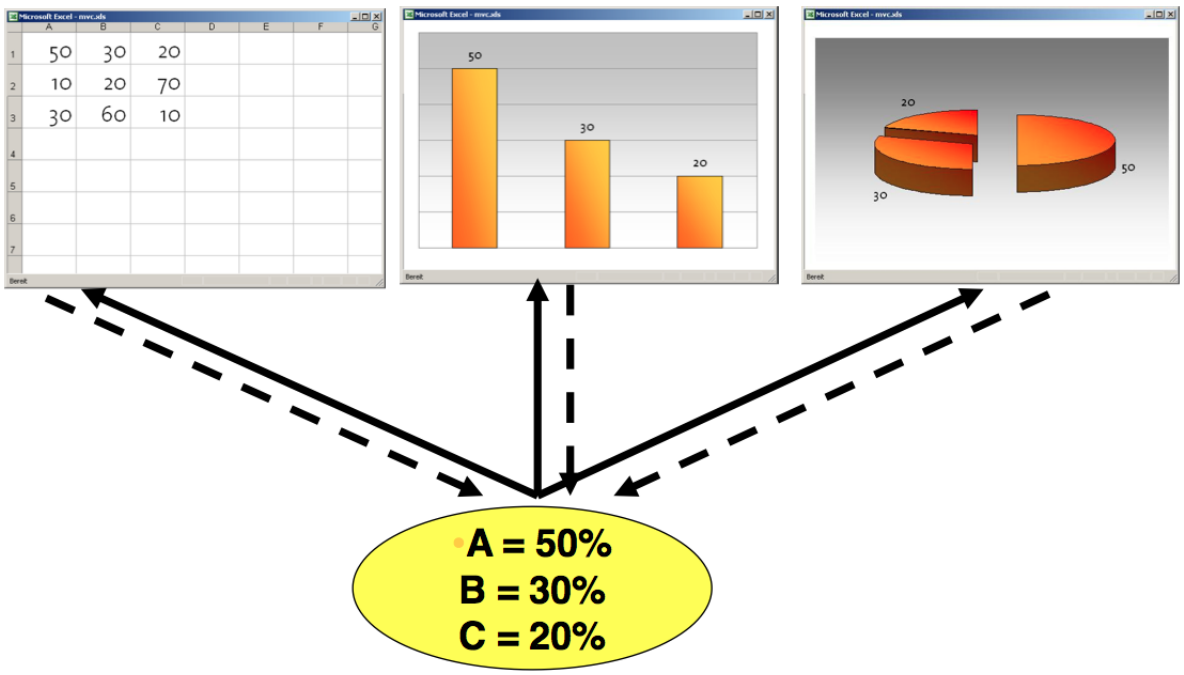
3 Observer Pattern

Also Known As **Publish-Subscribe** or **Listener Pattern**.

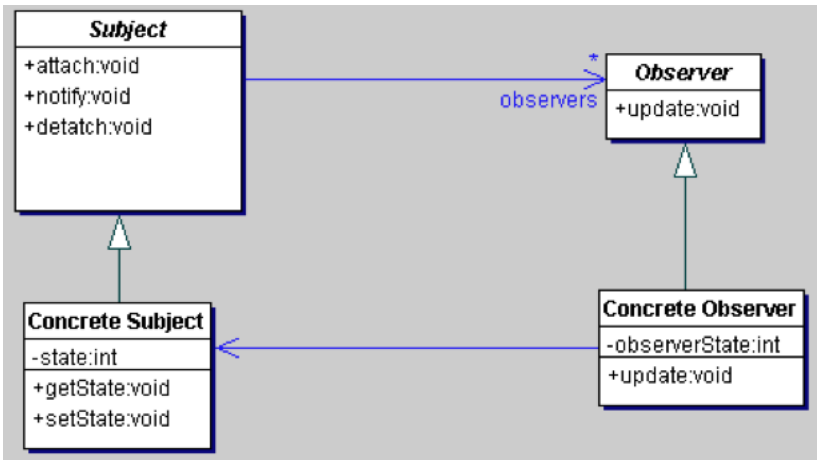
3.1 Intent

- 1:*-Relation between objects which allows to inform the dependent objects about state changes
- Consistency assurance between cooperating objects without connecting them too much
- Notification of a dependent object without knowing it

3.2 Motivation



3.3 Structure



3.4 Participants

Subject

- Knows its observers only through the Observer interface
- Offers methods to attach or detach an observer

Observer

- Defines interface to publish notifications

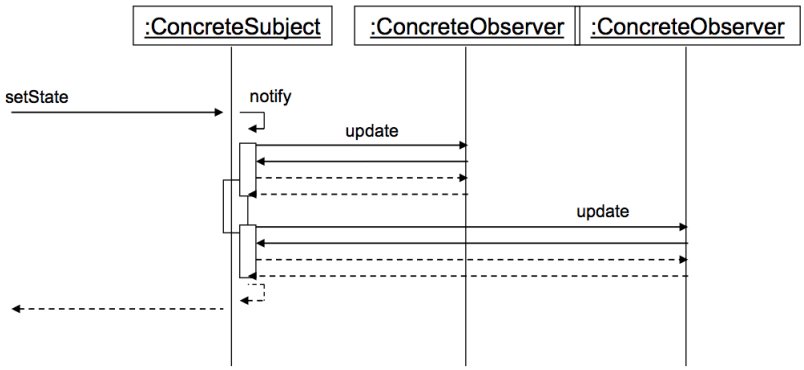
Concrete Subject

- Stores state of interest to concrete observer objects
- Notifies its observers when its state changes

Concrete Observer

- Implements Observer interface to keep its state consistent with the subject
- May maintain a reference to a concrete subject

3.5 Collaborations



3.6 Consequences

Decoupling

- Subject only knows Observer interface, no concrete observers
- Update = dynamically bound =¿ up-calls
- Subject and observer may belong to different abstraction layers

Support for broadcast communication

- Notification is broadcast, subject does not care about number of observers
- It is up to the observer to handle or ignore notifications

Unexpected updates

- A simple operation on a subject may cause a cascade of updates

3.7 Beispiel

Listing 3: Beispiel Observer Pattern

```
interface Observer {
   void update();
}
class Observable {
  private List<Observer> observers = new ArrayList<Observer>();
  public void addObserver(Observer o) {
    observers.add(o);
  }
  public void removeObserver(Observer o) {
    observers.remove(o);
  }
  protected void notifyObservers() {
    for(Observer obs : observers) {
          obs.update();
        }
  }
}
class Sensor extends Observable {
  private int temp;
  public int getTemperature(){
    return temp;
  }
  public void setTemperature(int val){
    temp = val;
      notifyObservers();
    }
}
class SensorObserver implements Observer {
  private Sensor s;
  SensorObserver (Sensor s){
    this.s = s;
    s.addObserver(this);
  }
  public void update(){
    System.out.println(                                          + s.
        getTemperatore());
  }
}
```