Entwurfsmuster

Jan Fässler

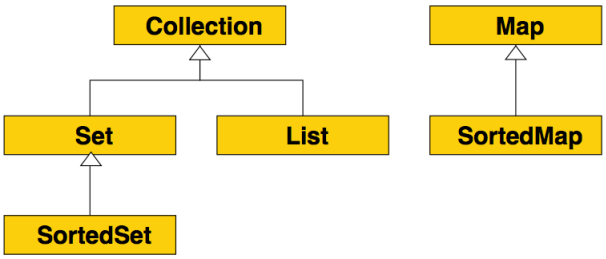
3. Semester (HS 2012)

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> {
    boolean hasNext();      // there is an element which can be jumped over
    E next();               // returns the jumped over element
    void remove();          // removes the last element returned by next
  }
```

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

```
abstract class AbstractCollection<E> implements Collection<E> {
  public abstract Iterator<E> iterator();
  public abstract boolean add(E x);
  public boolean isEmpty() { return size() == 0; }
  public int size() {
    int n = 0; Iterator<E> it = iterator();
    while (it.hasNext()) {
      it.next(); n++;
    }
    return n;
  }
  public boolean contains(Object o) {
    Iterator<E> e = iterator();
    while (e.hasNext()) {
      Object x = e.next();
      if(x == o || (o != null) && o.equals(x)) return true;
    }
    return false;
  }
  public void clear() {
    Iterator<E> it = iterator();
    while (it.hasNext()) {
      it.next();
      it.remove();
    }
  }
  public boolean remove(Object o) {
    Iterator<E> it = iterator();
    while (it.hasNext()) {
      Object x = it.next();
      if(x == o || (o != null && o.equals(x))) {
        it.remove();
        return true;
      }
    }
    return false;
  }
  public boolean containsAll(Collection<?> c) {
    Iterator<?> it = c.iterator();
    while (it.hasNext()) {
      if(!contains(it.next())) return false;
    }
    return true;
  }
  public boolean addAll(Collection<? extends E> c) {
    boolean modified = false;
    Iterator<? extends E> it = c.iterator();
    while (it.hasNext()) {
      if(add(it.next())) modified = true;
    }
    return modified;
  }
```

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

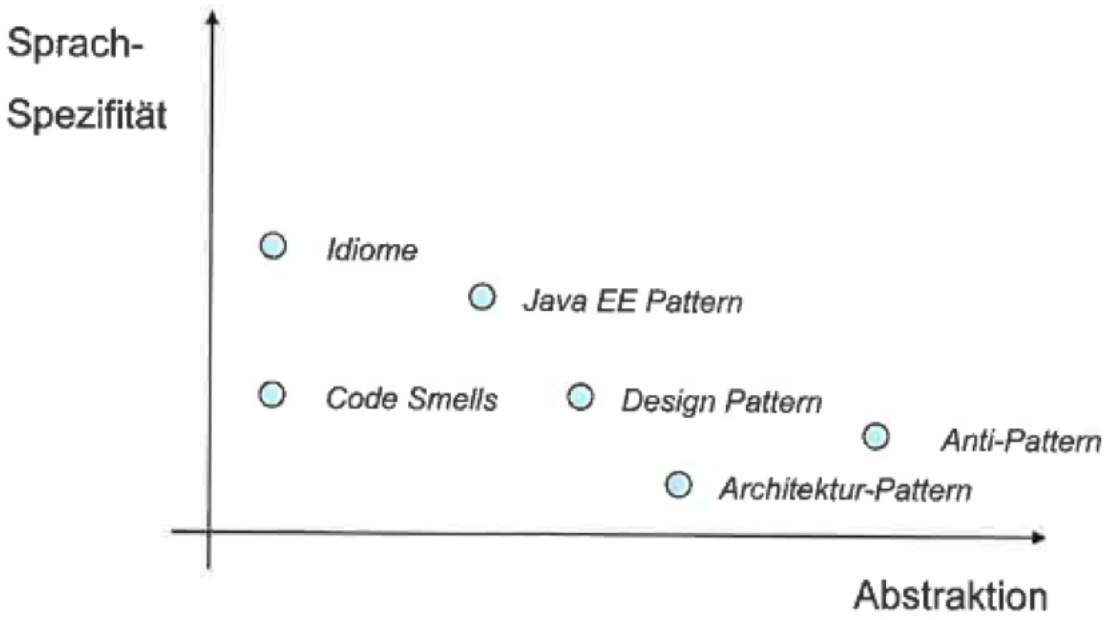
Domain-Specific patterns

- UI patterns, security patterns

Anti-Patterns

- Refactoring

2.2 Pattern Classification



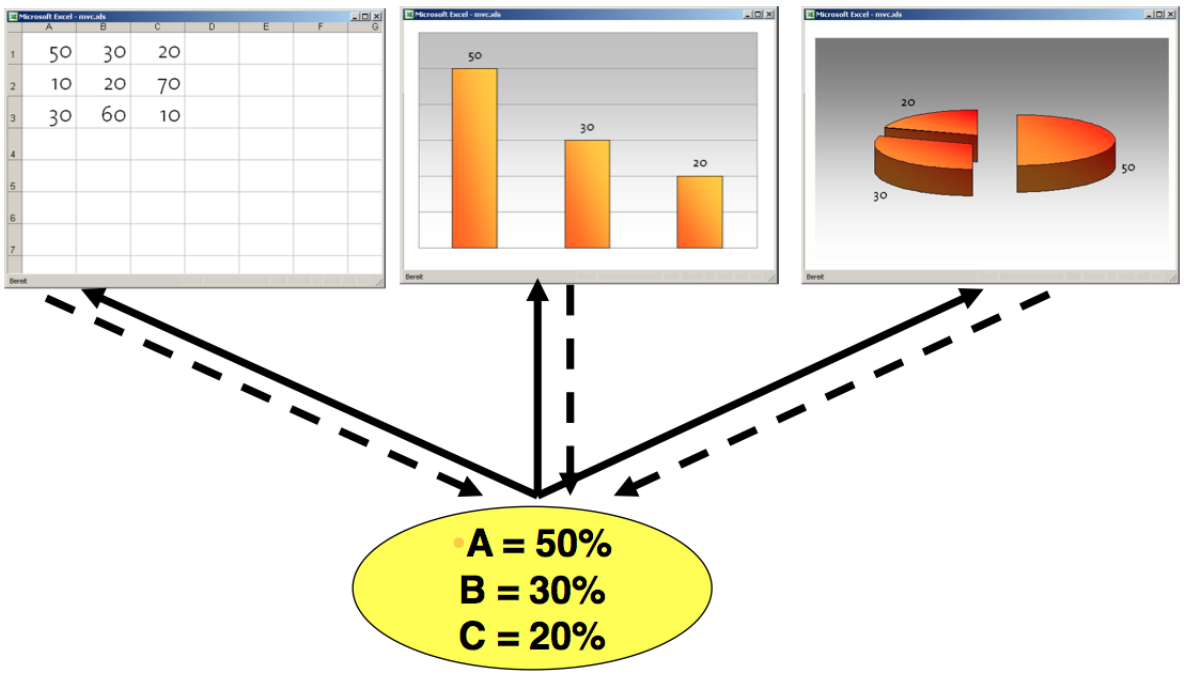
3 Observer Pattern

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

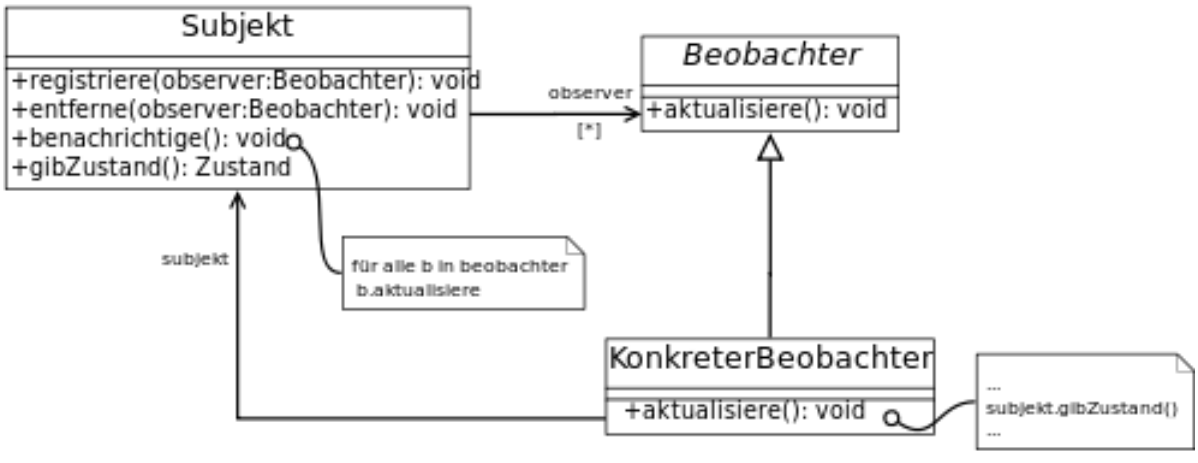
3.1 Ziel

- 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 Struktur



3.4 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("Sensor has changed, new temperature is " + s.
        getTemperatore());
  }
}
```

4 State Pattern

4.1 Ziel

Die Auslagerung von zustandsspezifischem Verhalten in einer Klasse. Ermöglicht mit dem Wechsel des Zustandes eines Objektes auch leicht das Verhalten zu ändern.

4.2 Motivation

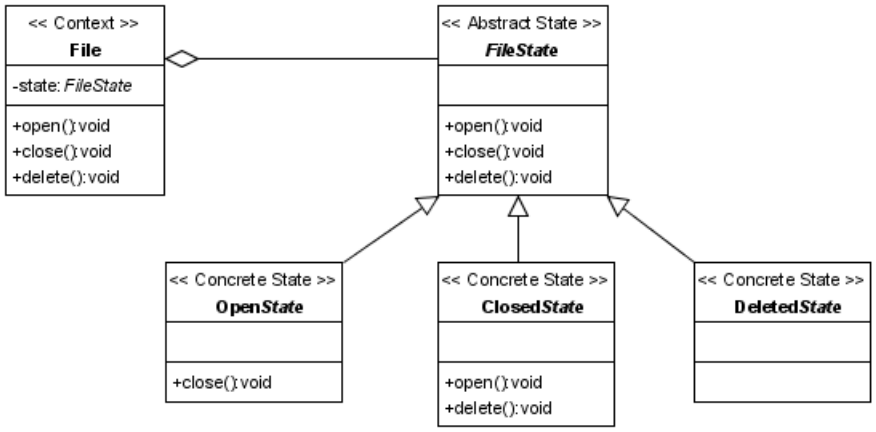
Die View des Grafikeditors hat eine Referenz auf ein abstraktes Tool-Interface. Diese Referenz definiert:

a. Der aktuelle Zeichenmodus

b. Verhalten wenn Maus gedrückt wird

4.3 Struktur

Das zustandsabhängige Verhalten des Objekts wird in separate Klassen ausgelagert, wobei für jeden möglichen Zustand eine eigene Klasse eingeführt wird, die das Verhalten des Objekts in diesem Zustand definiert. Damit der Kontext die separaten Zustandsklassen einheitlich behandeln kann, wird eine gemeinsame Abstrahierung dieser Klassen definiert. Bei einem Zustandsübergang tauscht der Kontext das von ihm verwendete Zustandsobjekt aus.



4.4 Beispiel

Listing 4: Beispiel State Pattern

```
public class Common {
    public static String firstLetterToUpper(final String WORDS) {
        String firstLetter = "";
        String restOfString = "";
        if (WORDS != null) {
            char[] letters = new char[1];
            letters[0] = WORDS.charAt(0);
            firstLetter = new String(letters).toUpperCase();
            restOfString = WORDS.toLowerCase().substring(1);
        }
        return firstLetter + restOfString;
    }
```

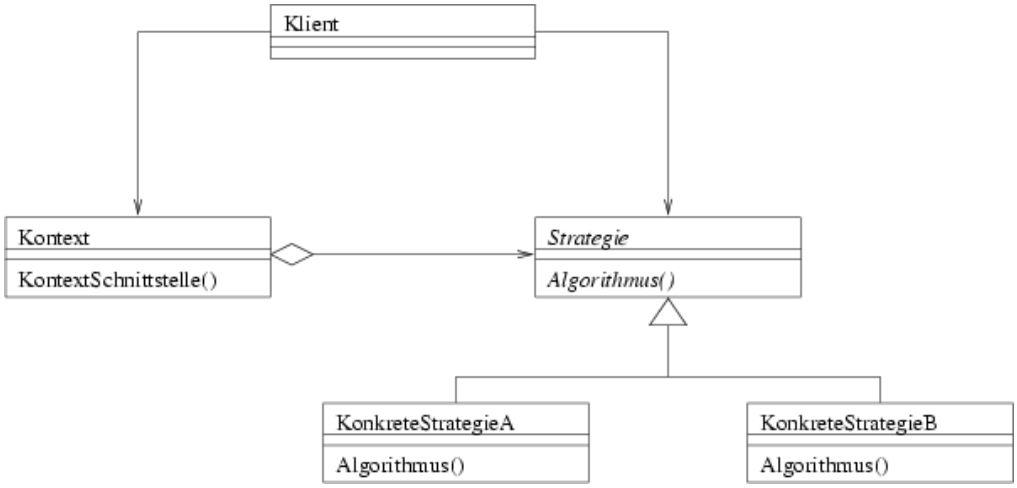
```
}
interface Statelike {
    void writeName(final StateContext STATE_CONTEXT, final String NAME);
}
class StateA implements Statelike {
    public void writeName(final StateContext STATE_CONTEXT, final String
        NAME) {
        System.out.println(Common.firstLetterToUpper(NAME));
        STATE_CONTEXT.setState(new StateB());
    }
}
class StateB implements Statelike {
    public void writeName(final StateContext STATE_CONTEXT, final String
        NAME) {
        System.out.println(NAME.toUpperCase());
        STATE_CONTEXT.setState(new StateA());
    }
}
public class StateContext {
    private Statelike myState;
    public StateContext() { setState(new StateA()); }
    public void setState(final Statelike NEW_STATE) { myState = NEW_STATE; }
    public void writeName(final String NAME) { myState.writeName(this, NAME)
        ; }
}
```

5 Strategy Pattern

5.1 Ziel

Eine Familie von Algorithmen definieren, von dem restlichen Programmcode abtrennen. Diese ermöglicht die Auswahl aus verschiedenen Implementierungen und dadurch erhöht sich die Flexibilität und die Wiederverwendbarkeit.

5.2 Struktur



5.3 Beispiel

Listing 5: Beispiel Strategy Pattern

```
class Klient {
        public static void main(final String[] ARGS) {
                Kontext k = new Kontext();
                k.setStrategie(new KonkreteStrategieA());
                k.arbeite();    // "Weg A"
                k.setStrategie(new KonkreteStrategieB());
                k.arbeite();    // "Weg B"
        }
}

class Kontext {
        private Strategie strategie = null;

        public void setStrategie(final Strategie STRATEGIE) {
                strategie = STRATEGIE;
        }

        public void arbeite() {
                if (strategie != null)
                        strategie.algorithmus();
        }
}

interface Strategie {
        void algorithmus();
}
```

```
class KonkreteStrategieA implements Strategie {
        public void algorithmus() {
                System.out.println("Weg A");
        }
}

class KonkreteStrategieB implements Strategie {
        public void algorithmus() {
                System.out.println("Weg B");
        }
}
```

6 Null Object Pattern

6.1 Beschreibung

Das Entwurfsmuster Nullobjekt findet Anwendung bei der Deaktivierung von Referenzen auf Variablen und besteht darin, der Referenz ein Objekt zuzuweisen, das keine Aktion ausführt, anstatt die Referenz zu invalidieren. Dadurch wird erreicht, dass die Referenz auf die Variable zu jedem Zeitpunkt auf ein gültiges Objekt verweist, was Behandlungen von Sonderfällen (das Nichtvorhandensein) erübrigt.

6.2 Beispiel

Listing 6: Beispiel Null Object Pattern

```
1 public class NullLayout implements LayoutManager {
    public void addLayoutComponent(String name, Component comp) {} public void
        removeLayoutComponent(Component comp) {}
    public Dimension minimumLayoutSize(Container parent){
      return parent.getSize();
    }
6   public Dimension preferredLayoutSize(Container parent){
      return parent.getSize();
    }
    public void layoutContainer(Container parent) {}
  }
```