Entwurfsmuster

Jan Fässler

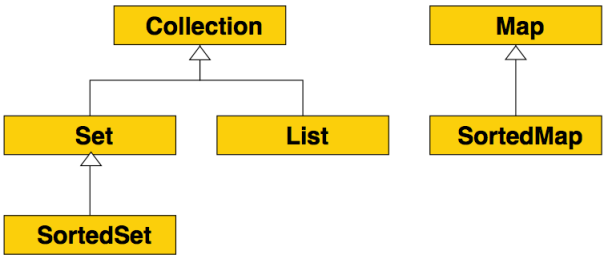
3. Semester (HS 2012)

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

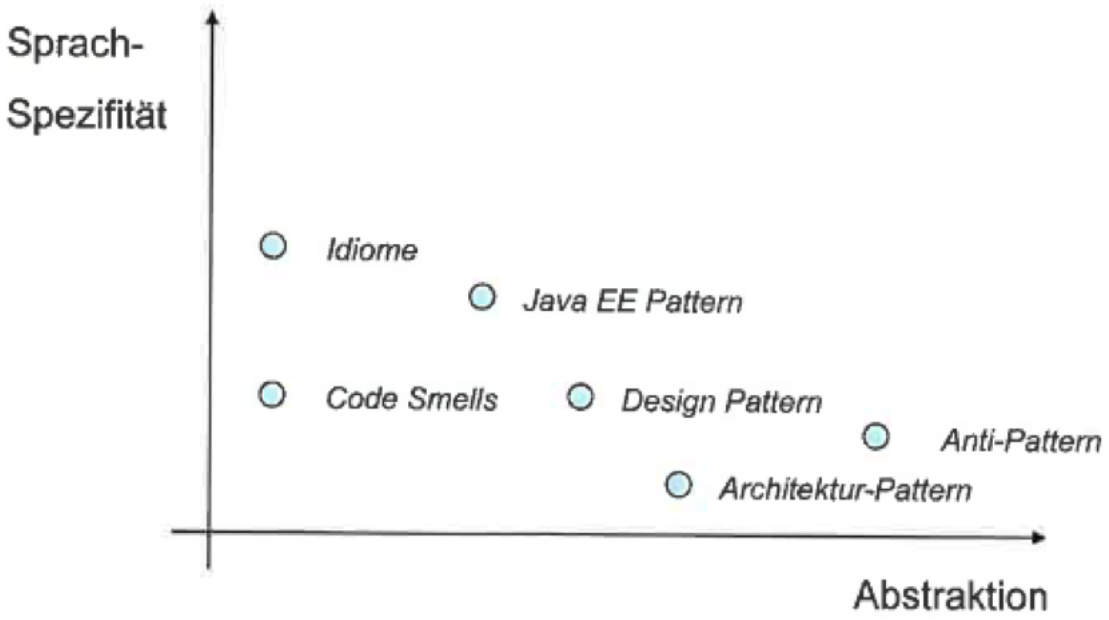
Domain-Specific patterns

- UI patterns, security patterns

Anti-Patterns

- Refactoring

2.2 Pattern Classification



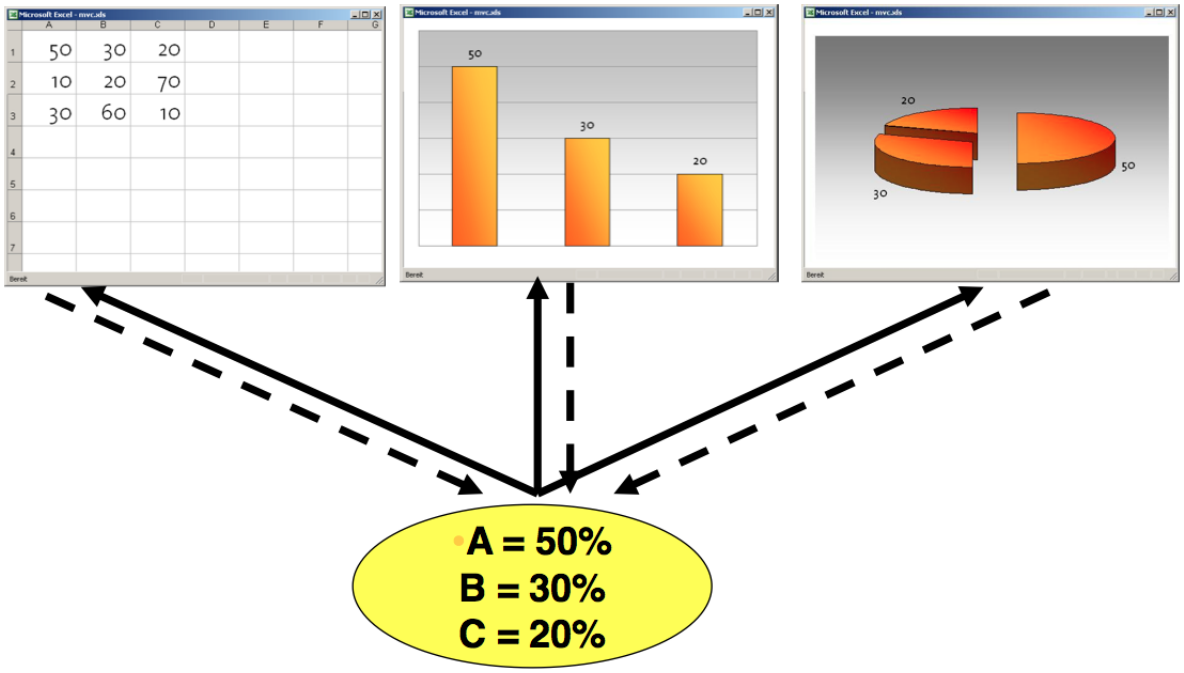
3 Observer Pattern

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

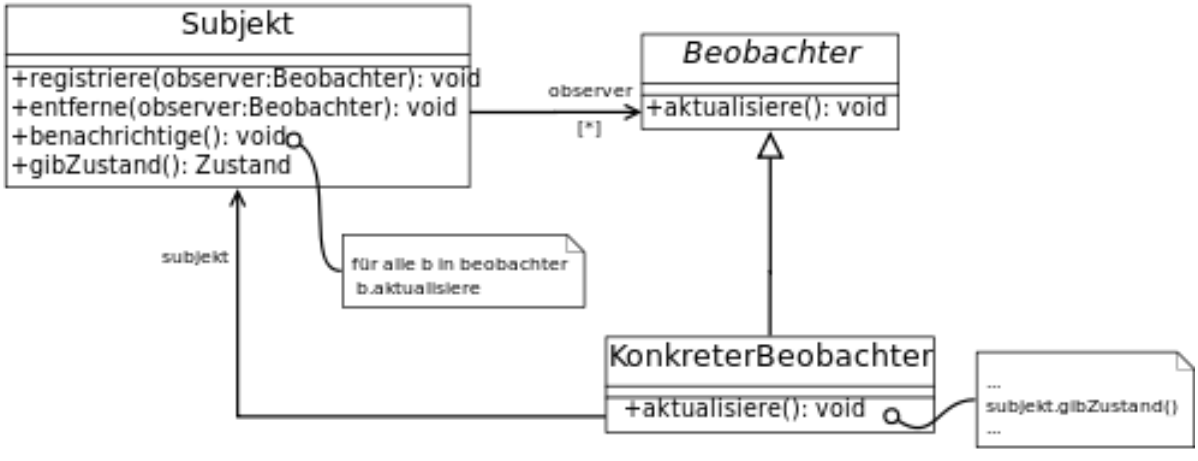
3.1 Ziel

- 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

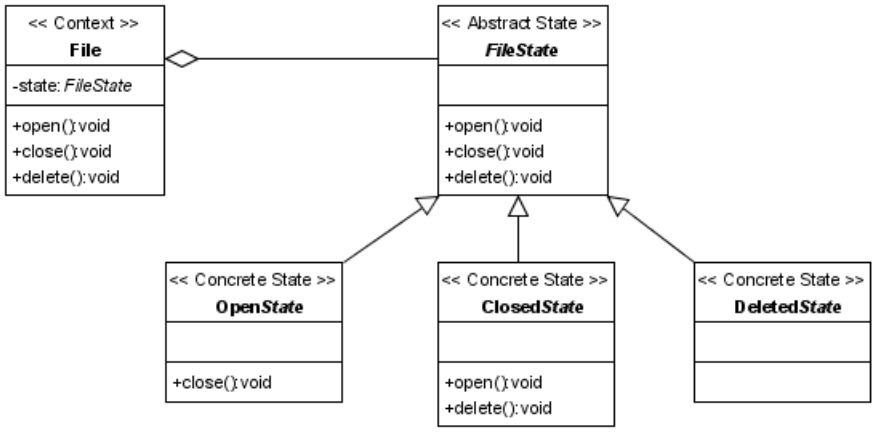
Die View des Grafikeditors hat eine Referenz auf ein abstraktes Tool-Interface. Diese Referenz definiert:

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;
    }
```

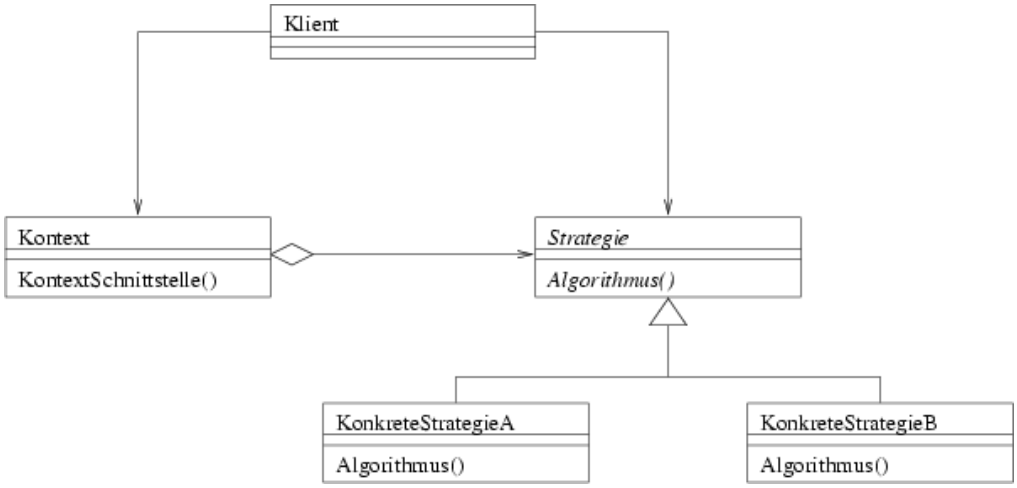
```
}
interface Statelike {
    void writeName(final StateContext STATE_CONTEXT, final String NAME);
}
class StateA implements Statelike {
    public void writeName(final StateContext STATE_CONTEXT, final String
        NAME) {
        System.out.println(Common.firstLetterToUpper(NAME));
        STATE_CONTEXT.setState(new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

```
1  class Klient {
           public static void main(final String[] ARGS) {
                   Kontext k = new Kontext();
                   k.setStrategie(new KonkreteStrategieA());
                   k.arbeite();     // "Weg A"
6                  k.setStrategie(new KonkreteStrategieB());
                   k.arbeite();     // "Weg B"
           }
   }
   class Kontext {
11         private Strategie strategie = null;
           public void setStrategie(final Strategie STRATEGIE) {
                   strategie = STRATEGIE;
           }
           public void arbeite() {
16                 if (strategie != null) strategie.algorithmus();
           }
   }
   interface Strategie { void algorithmus(); }
   class KonkreteStrategieA implements Strategie {
21         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

```
  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();
5   }
    public Dimension preferredLayoutSize(Container parent){
      return parent.getSize();
    }
    public void layoutContainer(Container parent) {}
10 }
```

7 Composite Pattern

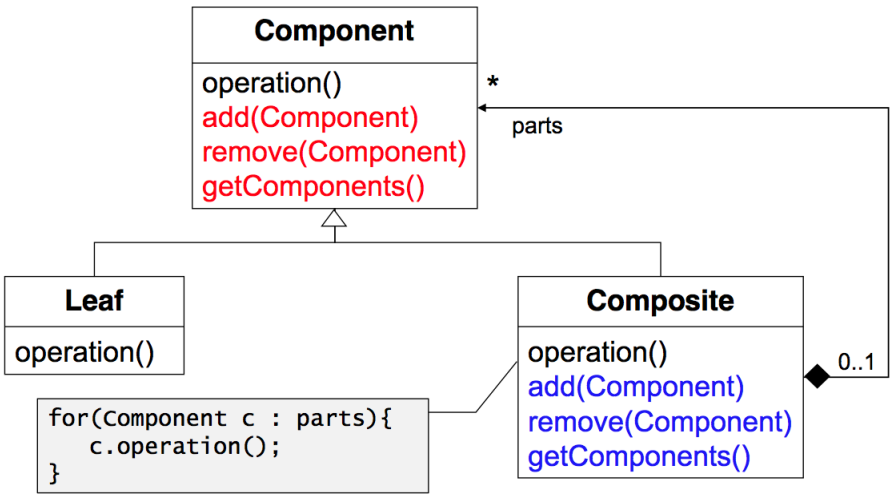
7.1 Ziel

Das Kompositionsmuster wird angewendet, um Teil-Ganzes-Hierarchien zu repräsentieren, indem Objekte zu Baumstrukturen zusammengefügt werden.Die Grundidee des Kompositionsmusters ist, in einer abstrakten Klasse sowohl primitive Objekte als auch ihre Behälter zu repräsentieren. Somit können sowohl einzelne Objekte, als auch ihre Kompositionen einheitlich behandelt werden.

7.2 Motivation

Ein klassisches Beispiel sind hierarchische Dateisysteme, bzw ihre Repräsentation innerhalb von Dateimanagern oder Filebrowsern als Verzeichnisse und Dateien.
Ein modernes Beispiel sind die Klassendefinitionen der grafischen Benutzeroberfläche von Java. Alle Elemente wie Schaltflächen und Textfelder sind Spezialisierungen der Klasse Component. Die Behälter für diese Elemente sind aber ebenfalls Spezialisierungen derselben Klasse.

7.3 Struktur



7.4 Beispiel

Listing 7: Beispiel Composite Pattern

```
interface Grafik {
    public void print();
}
class GrafikKompositum implements Grafik {
    final private List<Grafik> children = new ArrayList<Grafik>(10);
    public void print() {
        for (final Grafik grafik : children) grafik.print();
    }
    public void add(final Grafik component){children.add(component);}
    public void remove(final Grafik component){children.remove(component);}
}
class Ellipse implements Grafik {
    public void print() { System.out.println("Ellipse"); }
};
```

```
15 public class GrafikTest {
       public static void main(final String... args) {
           Ellipse ellipse1 = new Ellipse(),
                   ellipse2 = new Ellipse(),
                   ellipse3 = new Ellipse(),
20                 ellipse4 = new Ellipse();
           GrafikKompositum grafik1 = new GrafikKompositum(),
                            grafik2 = new GrafikKompositum(),
                            grafikGesamt = new GrafikKompositum();
           grafik1.add(ellipse1);
25         grafik1.add(ellipse2);
           grafik1.add(ellipse3);
           grafik2.add(ellipse4);
           grafikGesamt.add(grafik1);
           grafikGesamt.add(grafik2);
30         grafikGesamt.print();
       }
   }
```

8 Prototype Pattern

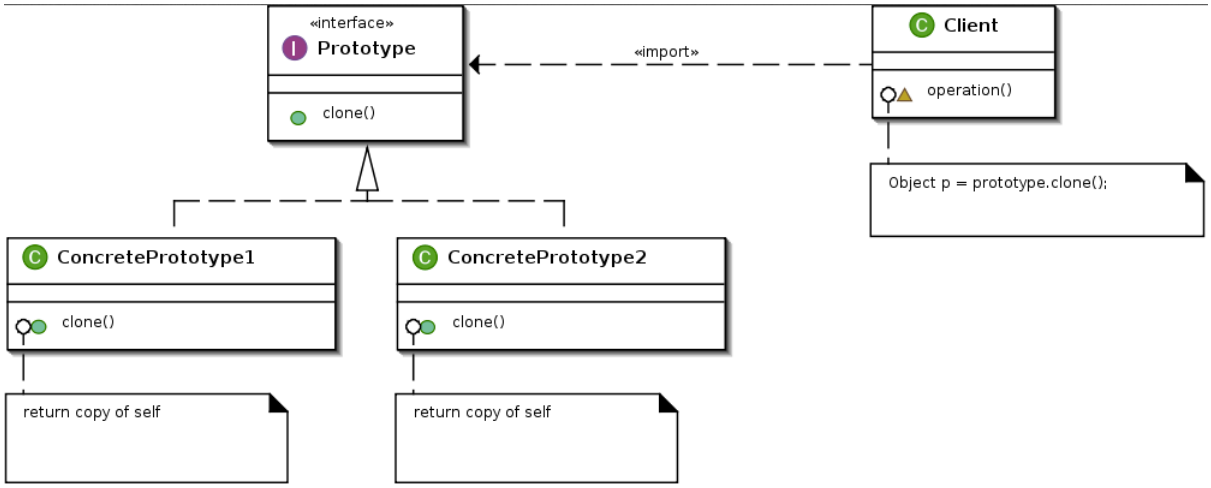
8.1 Ziel

Neue Instanzen werden aufgrund prototypischer Instanzen erzeugt. Dabei wird die Vorlage kopiert und an neue Bedürfnisse angepasst

8.2 Motivation

- Copy und Paste funktion in einem Editor
- Tool palette mit Prototyp-Objekten welche kopiert werden können

8.3 Struktur



8.4 Beispiel

Listing 8: Beispiel Prototype Pattern

```
abstract class Prototype implements Cloneable {
    public Prototype clone() throws CloneNotSupportedException {
        return (Prototype) super.clone();
    }
    public abstract void setX(final short X);
    public abstract short getX();
}
class ConcretePrototype extends Prototype {
    private int x;
    public void setX(int X) { x = X; }
    public short getX() { return x; }
}
public class Client {
    public static void main(String args[])  {
        Prototype origin = new ConcretePrototype(10);
        Prototype clone =  origin.clone();
        clone.setX(4);
    }
}
```

8.5 Unterschiedliche Clone Varianten

Problemstellung	**Cloneable**	**Copy-Constructor**	**Reflection**	**Serialization**
final Felder	Werden richtig kopiert	Können richtig initialisiert werden	Dürfen nicht vorkommen	Werden richtig kopiert
deep copy	Muss selber implementiert werden	Muss selber implementiert werden	Automatisch (keine Kontrolle)	Automatisch (beschränkte Kontrolle: keine Kopie von transienten Feldern)
zyklische Strukturen / Alias-Referenzen	Müssen selber korrekt aufgelöst werden.	Müssen selber korrekt aufgelöst werden	Automatisch	Automatisch
Geschwindigkeit	Sehr effizient	Sehr effizient	Effizient	Sehr ineffizient

9 Immutability Pattern

9.1 Ziel

Das Immutable Pattern stellt sicher, dass eine Instanz der Klasse nach der Initialisierung nicht mehr geändert werden kann.

9.2 Regeln

1. Die Methoden der Klasse dürfen die Daten der Klasse nicht ändern.
2. Die Felder müssen privat oder final deklariert sein, falls es sich um Referenzen auf immutable Objekte oder Felder von primitiven Datentypen handelt.
3. Nur Getter-Methoden definieren, keine Setter Methoden
4. Alle Rückgabe-Werte von Getter-Methoden müssen kopiert werden, falls es nicht Referenzen auf immutable Objekte oder Resultate mit primitivem Datentyp sind.
5. Klassen als final deklarieren damit nicht in der Unterklasse die Regeln verletzt werden können.
6. Parameter in den Konstruktoren, die nicht immutable oder von primitiven Datentypen sind müssen geklont werden

9.3 Beispiel

Listing 9: Beispiel Immutability Pattern

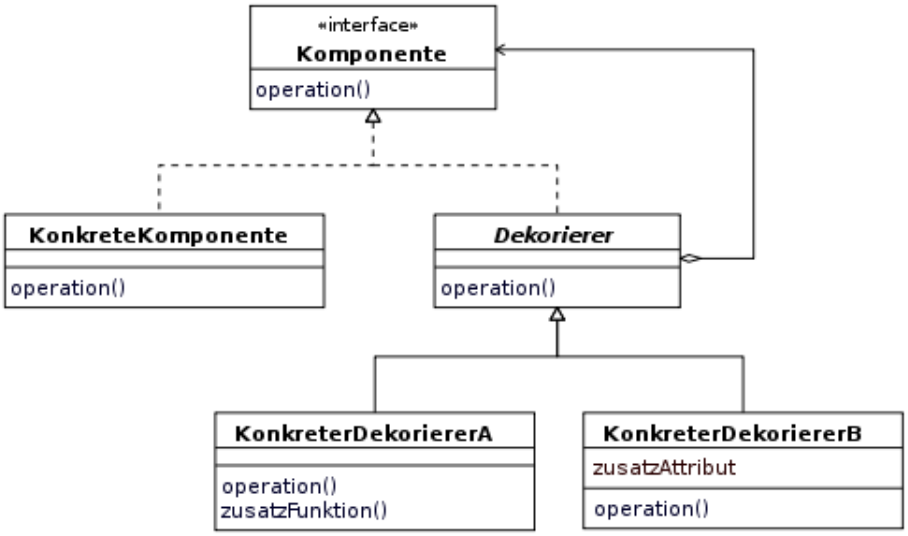
```
class Line {
  private final Point start, stop;
  public Line(Point start, Point stop) {
    this.start = (Point) start.clone();
    this.stop = (Point) stop.clone();
  }
  public Point getStart() { return (Point) start.clone();  }
  public Point getStop() { return (Point) stop.clone(); }
  public Object clone() { return this; }
}
```

10 Decorator Pattern

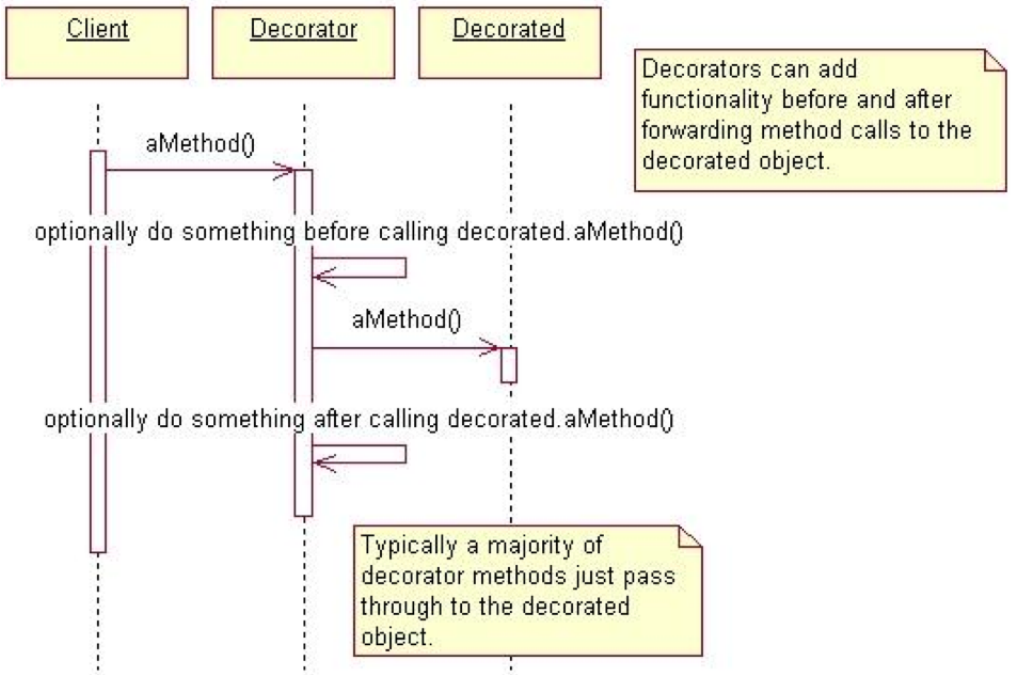
10.1 Ziel

Das Muster ist eine flexible Alternative zur Unterklassenbildung, um eine Klasse um zusätzliche Funktionalitäten zu erweitern. Die Vorteile bestehen darin, dass mehrere Dekorierer hintereinandergeschaltet werden können; die Dekorierer können zur Laufzeit und sogar nach der Instanzierung ausgetauscht werden. Die zu dekorierende Klasse ist nicht unbedingt festgelegt (wohl aber deren Schnittstelle). Zudem können lange und unübersichtliche Vererbungshierarchien vermieden werden.

10.2 Struktur



10.3 Dynamics



10.4 Beispiel

Listing 10: Beispiel Decorator Pattern

```
interface DekoratorMuster
{
    abstract class Spielfigur {
        public abstract void Drohe();
    }
    class Monster extends Spielfigur {
        public void Drohe() { System.out.println("Grrrrrrrrrr."); }
    }
    class Dekorierer extends Spielfigur
    {
        private Spielfigur meineFigur;
        public Dekorierer(Spielfigur s) { meineFigur = s; }
        public override void Drohe() { meineFigur.Drohe(); }
    }
    class HustenDekorierer extends Dekorierer {
        public HustenDekorierer(Spielfigur s) { super(s); }
        public override void Drohe() {
            System.out.println("Hust, hust. ");
            base.Drohe();
        }
    }
    class SchnupfenDekorierer extends Dekorierer {
        public SchnupfenDekorierer(Spielfigur s) { super(s); }
        public override void Drohe() {
            System.out.println("Schniff. ");
            base.Drohe();
        }
    }
    public class Main {
        public static void main(String[] args) {
            Spielfigur monster = new Monster();
            monster.Drohe();
            Spielfigur hustenMonster = new HustenDekorierer(monster);
            hustenMonster.Drohe();
            Spielfigur schnupfenMonster = new SchnupfenDekorierer(monster);
            schnupfenMonster.Drohe();
            Spielfigur hustenSchnupfen = new SchnupfenDekorierer(new
                HustenDekorierer(monster));
            hustenSchnupfen.Drohe();
        }
    }
}
```

11 Command Pattern

11.1 Ziel

- Eine Aktion zu einem Objekt machen, damit man sie weiterleiten und zu einem beliebigen späteren Zeitpunkt ausführen kann.
- Eine Aktion aufrufen von der der Aufrufer weder den Empfänger noch die exakte Methode kennen muss.
- Eine Entkoppelung des Objektes, welches eine Operation aufruft vom Objekt, das weiss, wie die Operation ausgeführt werden soll.

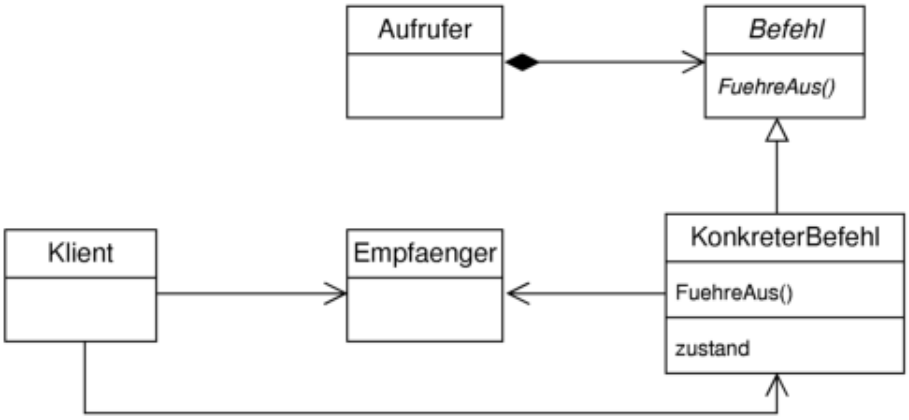
11.2 Beteiligte

Klient
Instanziert ein Kommando und füllt es mit Informationen um später eine Aktion ausführen zu können.

Aufrufer
Entscheidet wann das Kommando ausgeführt wird.

Empfänger
Objekt das die Aktion ausführen kann.

11.3 Struktur



12 Factory Pattern

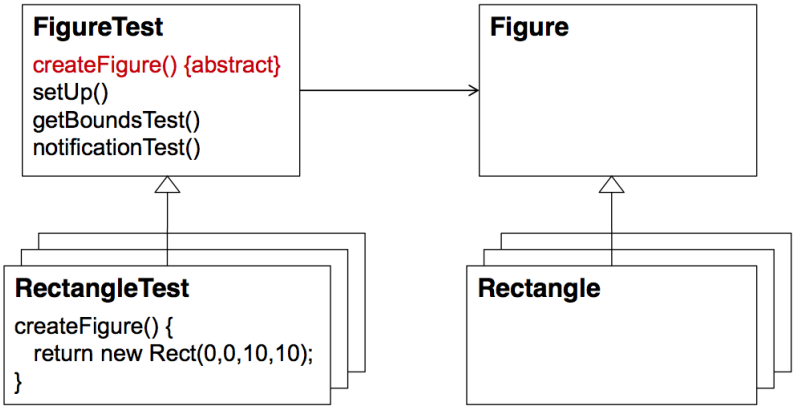
12.1 Idee

Das Muster beschreibt, wie ein Objekt durch Aufruf einer Methode anstatt durch direkten Aufruf eines Konstruktors erzeugt wird. Es gehört somit zur Kategorie der Erzeugungsmuster.

12.1.1 Verwendung

Die Fabrikmethode findet Anwendung, wenn eine Klasse die von ihr zu erzeugenden Objekte nicht kennen kann bzw. soll, oder wenn Unterklassen bestimmen sollen, welche Objekte erzeugt werden. Typische Anwendungsfälle sind Frameworks und Klassenbibliotheken.

12.2 Struktur



13 Singleton

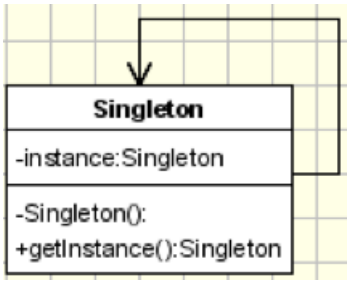
13.1 Idee

Von einer Klasse soll ho?chstens eine Instanz existieren.

13.2 Motivation

- Cache-Implementierungen
- Objekte welche Registryeinstellungen oder Präferenzen verwalten
- Thread-Pool
- Klasse, mit welcher MP3 Dateien abgespielt werden. Falls man wa?hrend dem Abspielen einer Datei eine neue Datei abspielen lässt, so ist das Verhalten unvorhersehbar.
- Treiber (Drucker/Datenbank).
- Kommunikation über Rechnergrenzen hinweg: Versand von Daten u?ber einen Socket

13.3 Struktur



13.4 Implementierung

Listing 11: Java Implementierung

```
public final class Singleton  {
    private static Singleton instance;
    private Singleton() {}
    public synchronized static Singleton getInstance() {
        if (instance == null) instance = new Singleton();
        return instance;
    }
}
```

Soft-Referenzen werden beim Traversieren des Garbage Collectors zerstört, wenn keine weiteren harten Referenzen auf das Objekt mehr zeigen. Dies verhindert Memory-Leak durch Singleton-Patterns.

Listing 12: Java Implementierung (freundlicher für den Garbage Collector)

```
import java.lang.ref.SoftReference;
public final class Singleton {
    private static SoftReference<Singleton> instance;
    private Singleton(){}
    public synchronized static Singleton getInstance(){
        if(instance == null){
            instance = new SoftReference<Singleton>(new Singleton());
        }
        return instance.get();
    }
}
```

Das Initialization-on-demand Holder Idiom nutzt die Funktionsweise der JVM aus und macht effektiv das Gleiche wie die Synchronized-Variante. Sie ist threadsafe, ohne sich dabei auf spezielle Konstrukte wie synchronized oder volatile berufen zu müssen.

Listing 13: Initialization-on-demand holder idiom

```
public class Singleton {
        private Singleton() { }
        /**
        * SingletonHolder is loaded on the first execution
        * of Singleton.getInstance() or the first access
        * to SingletonHolder.INSTANCE, not before.
        */
        private static class SingletonHolder {
                public static final Singleton INSTANCE = new Singleton();
        }
        public static Singleton getInstance() {
                return SingletonHolder.INSTANCE;
        }
}
```