Technische Hochschule Rosenheim

Modul - Objektorientierte Programmierung

Bachelor Wirtschaftsinformatik

08 - Annotationen, Refactoring und DP

Prof. Dr. Marcel Tilly

Fakultät für Informatik, Cloud Computing

Annotationen

Technische Hochschule Rosenheim

Annotationen sind Metainformationen, d.h. Informationen *über Informationen*

- Spezielles syntaktisches Element: `@`, z.B. `@Override`, `@Test`
- Können an Klassen, Attribute, Methoden und Argumente geheftet werden.
- Stellen zusätzliche semantische Informationen zum Programm bereit.
- Semantik wird vom Compiler nicht direkt bearbeitet, d.h. sie haben keinen direkten Einfluss auf den Programmablauf
- Annotationen beeinflussen die Semantik von "Dingen," die diese Elemente verwenden

Verwendung von Annotationen

Technische Hochschule Rosenheim

Informationen für den Compiler

- Programmierfehler erkennen, Warnungen erzeugen

Informationen für Werkzeuge (Toolkits)

- Für Toolkits, welche auf dem Quelltext arbeiten
- Zum Übersetzungszeitpunkt (engl. *compile time*), z.B. zur automatischen Codegenerierung
- Zum Deploymentzeitpunkt (engl. *deployment time*), z.B. zur Verifizierung von Anforderungen
- Zur Laufzeit (engl. *run time*), z.B. zum analysieren von Objekten. (→ Reflection)

Technische Hochschule Rosenheim

Beispiele für Annotationen

Vordefinierte Annotationen

Für Sprachfeatures und häufige Verwendung

Objekt-relationale Abbildung

Speicherung von Objekten mit der Java Persistence API (JPA)

Testframeworks

Testen mit JUnit 5

Vordefinierte Annotationen

`@Override`

Typischer Fehler: **überladen** statt **überschreiben**

```
class MeineKlasse {
    // falsch, aber unentdeckt
    public boolean equals(MeineKlasse m) { /* ... */ }

    // so wäre es richtig gewesen!
    public boolean equals(Object o) { /* ... */ }
}
```

Vordefinierte Annotationen

`@Override`

Durch die Annotation mit `@Override` kann der Compiler Fehler erkennen:

```
class MeineKlasse {
    @Override
    public boolean equals(MeineKlasse m) { /* ... */ }  // Compilerfehler!

    @Override
    public boolean equals(Object o) { /* ... */ } // ok
}
```

Vordefinierte Annotationen

`@SuppressWarnings`

- Aktuelle Java Compiler erzeugen sehr viele Warnungen
- Viele sind berechtigt und weisen auf Programmierfehler hin
- Einige sind jedoch manchmal unvermeidbar
- Warnungen abschalten & gleichzeitig dokumentieren
- Art der zu unterdrückenden Warnung ist Parameter (und compilerspezifisch)

```
@SuppressWarnings("unchecked")
public void methodWithScaryWarning() {
    List rawList = new ArrayList();

    // das gäbe normalerweise eine Warnung:
    List<String> stringList = (List<String>)rawList;
}
```

Vordefinierte Annotationen

`@Deprecated`

- Hinweis an den Programmierer, eine Methode/Klasse/etc. nicht (mehr) zu verwenden
- Oft bei Update von Toolkits bzw. Bibliotheken
- Warnung bei Verwendung von Elementen, die mit `@Deprecated` annotiert sind

```
class AlteKlasse {
    @Deprecated
    public static void doofeAlteMethode() { /* ... */ }
}
class NeueKlasse {
    public boolean meineNeueMethode() {
        doofeAlteMethode();  // Compiler Warning
    }
}
```

Definition von Annotationen

- Annotationen werden ähnlich wie Interfaces definiert, jedoch mit dem `@`-Zeichen vor dem Schlüsselwort Interface:
- In der Definition einer Annotation können Methoden deklariert werden, die Elemente der Annotation beschreiben.
  - Methoden in Annotationen besitzen keine Parameter.
  - Erlaubte Rückgabetypen: `byte`, `short`, `int`, `long`, `float`, `double`, `String`, `Class`, `Annotation` und `Enumeration` sowie Felder über diese Typen
  - Definition von Defaultwerten möglich.

Definition von Annotationen

Beispiel: Annotation `@BugFix` um anzuzeigen, wer wann welchen Fehler zu beheben versucht hat.

```
public @interface BugFix {
    String author();
    String date();
    String bugsFixed() default "" ;
}
```

Verwendung von Annotationen

```
@BugFix(author="max", date="04.05.2018")
public void tolleFunktion() {
    // ...
}
```

- Hier als Methodenannotation
- Drittes Argument (`bugsFixed`) ist leer → Defaultwert (`""`)!

Schreibvereinfachungen

Technische Hochschule Rosenheim

Annotationen ohne Methoden: Markerannotationen

- Runde Klammern können entfallen
- Beispiele: `@Override`, `@Deprecated`

Annotationen mit genau einer Methode: Value-Annotationen

- Nur eine einzige Methode mit Namen `value`
- Bezeichner `value` kann bei Verwendung weggelassen werden

```
public @interface ReleaseVersion {
    String value() ;
}
```

```
@ReleaseVersion("1.2.5")
public class Calculator { /* ... */ }
```

Vordefinierte Meta-Annotationen

Technische Hochschule Rosenheim

- Meta-Annotationen annotieren Annotationen. ¯\(ツ)/¯
- `@Target`: Definert, auf welche Elemente die Annotation anwendbar ist
  - Beispiel: `@Target({ElementType.FIELD, ElementType.METHOD})`
- `@Retention`: Legt fest, ob eine Annotation nur im Quelltext ist, oder auch in der `.class` Datei und zur Laufzeit
  - Beispiele
    - `@Retention(RetentionPolicy.RUNTIME)`
    - `@Retention(RetentionPolicy.CLASS)`
    - `@Retention(RetentionPolicy.SOURCE)`
- `@Inherited`: Legt fest, ob eine Annotation durch Vererbung oder Implementierung weitergegeben wird.

Annotationen in der Praxis

Technische Hochschule Rosenheim

Häufig verwenden, selten definieren

Reflection API

- Erlaubt das Abfragen von Annotationen zur Laufzeit eines Java-Programms.
- Interface `AnnotatedElement`
  - Von den Reflection-Klassen `Class`, `Method`, `Field` etc. implementiert
  - Enthält Methoden zum Zugriff auf die Annotationen (wie `getAnnotations` oder `isAnnotationPresent` etc.)

Annotationen für JUnit5

`@Test`

Markiert eine Methode als Testmethode → Ausführung als Testcase in IntelliJ oder `gradle test`.

`@BeforeEach`, `@BeforeAll`, `@AfterEach`, `@AfterAll`

Markiert Methoden, welche vor oder nach Testcases ausgeführt werden soll, um z.B. Datenstrukturen zu initialisieren.

`@Disabled`

Markiert einen Testcase als zu ignorieren.

Annotations für JUnit5

```
class MeineTestSammlung {
    @BeforeAll
    static void initAll() { /* wird einmal vor allen ausgeführt */ }

    @BeforeEach
    void init() { /* wird vor jedem Test erneut ausgeführt */ }

    @Test
    void einTestFall() { // ... }

    @AfterEach
    void tearDown() { /* nach jedem Test erneut ausgeführt */ }

    @AfterAll
    static void tearDownAll() {  /* wird einmal am Ende ausgeführt */ }
}
```

Testen mit JUnit5

Technische Hochschule Rosenheim

- Neue Testklasse erstellen
- Testcasemethoden mit `@Test` annotieren
- Hilfsmethoden zum Testen (mehr Info)
  - `assertEquals(expected, actual)`
  - `assertTrue(actual)`
  - `assertFalse(actual)`
  - `assertThrows(exception, lambda)`
  - `assertArrayEquals(expected, actual)`

```
@Test
void exceptionTesting() {
    Throwable exception = assertThrows(IllegalArgumentException.class,
    () -> {
        throw new IllegalArgumentException("a message");
    });
    assertEquals("a message", exception.getMessage());
}
```

Toolkits, welche Annotationen verwenden

- JUnit5: Automatisiertes Testen
- Java Persistence API: De/Serialisierung von Objekten
- Google Gson: De/Serialisierung von Objekten nach/von JSON
- https://spring.io/: Webapplikationen
- http://square.github.io/retrofit/: REST API Adapter
- http://jakewharton.github.io/butterknife/: Android GUI
- https://github.com/chalup/microorm/: ORM

Refactoring

Technische Hochschule Rosenheim

- Improving the Design of Existing Code -

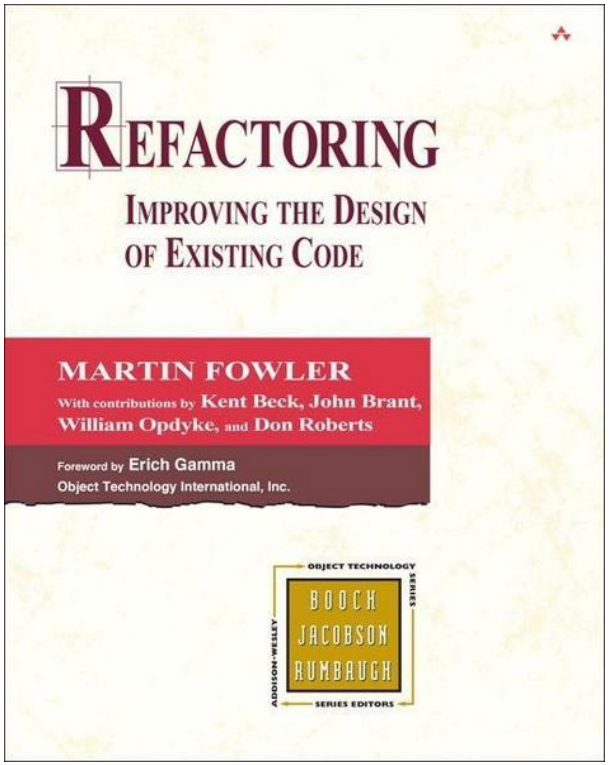
„A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour.“ -- Martin Fowler,1999

Zur Geschichte

Technische Hochschule Rosenheim

- Ward Cunningham und Kent Beck beginnen bereits in den 1980er bei ihren Arbeiten mit Smalltalk explizit mit Refactoring. Sie erarbeiten eine Softwareprozess Xtreme Programming (XP), bei dem Refactoring integraler Bestandteil ist. [-1999]
- Ebenso arbeitet Ralph Johnson bei seinen Arbeiten zu Frameworks mit Refactoring [~1990]
- Die erste wissenschaftliche Arbeit schreibt William Opdyke zu diesem Thema „Erhalt von Semantic beim Einsatz von Refactoring“. Von ihm stammt auch die erste „Refactoring“ Liste [1992]
- Auf Basis dieser Ideen entwickeln John Brant, Don Roberts & Ralph Johnson den Refactoring Browser in Smalltalk [1996]

Buch



[Refactoring.com]

Warum Refactoring?

Technische Hochschule Rosenheim

- Erhöht die Qualität des Designs
  - “Building it this way is stupid, but we did it that way because it was faster”
  - “Ugh, I’ll come back to this later and fix it up”
  - “We should have done X, but ...”
- Verbessert die Verständlichkeit des Codes
- Hilft bei der Fehlersuche
- Verbessert die Wiederverwendbarkeit

Wann Refactoring?

Technische Hochschule Rosenheim

- `Copying and pasting is your enemy!`
  - Gerne kopiert man Fehler!
  - Teilweise kopiert man Code, den man nicht versteht!
- also besser : **„Refactor“**
  - bevor neue Funktionalität hinzukommt!
  - um Fehler zu finden!
  - beim Codereview!
  - wenn „Code Smells“

Code Smells

Technische Hochschule Rosenheim

A Code Smell is a hint that something has gone wrong somewhere in your code [. . .]
Note that a CodeSmell is a hint that something might be wrong, not a certainty [. . .]
Calling something a CodeSmell is not an attack; it's simply a sign that a closer look is warranted.

-- http://xp.c2.com/CodeSmells.html

Technische Hochschule Rosenheim

Code Smell – „Duplizierter Code oder Bad Design“

Vereinheitliche Code, wenn dieselbe Codestruktur an mehreren Stellen auftaucht: » Gleicher Ausdruck in Methoden derselben Klasse » Gleicher Ausdruck in Geschwisterunterklassen » Code ist ähnlich aber nicht gleich (gibt es hier Möglichkeiten?) » Methoden machen dasselbe mit einem anderen Algorithmus

Code Smell: “Gleicher Ausdruck in Unterklassen”

```
class Employee {...}

class Salesman extends Employee {
  get name() {...}
}

class Engineer extends Employee {
  get name() {...}
}
```

```
class Employee {
  get name() {...}
}
class Salesman extends Employee {...}
class Engineer extends Employee {...}
```

Refactoring Technik: “Pull Up to Super”

Technische Hochschule Rosenheim

Wenn Code ist gemeinsame Code über alle Subklassen, gehört er in die Superklasse.

```
class Manager extends Employee {
  public Manager(String name, String id, int grade) {
    this.name = name;
    this.id = id;
    this.grade = grade;
  }
  // ...
}
```

```
class Manager extends Employee {
  public Manager(String name, String id, int grade) {
    super(name, id);
    this.grade = grade;
  }
  // ...
}
```

Code Smell: “Switch Statements”

```
class Bird {

    public static final int EUROPEAN = 1;
    public static final int AMERICAN = 2;
    public static final int NORWEGIAN_BLUE = 3;

    int _type;
    ...

    public double getSpeed() {
        switch (_type) {
            case EUROPEAN:
                return getBaseSpeed();
            case AMERICAN:
                return getBaseSpeed() - getLoadFactor() * _numberOfLoad;
            case NORWEGIAN_BLUE:
                return getBaseSpeed(_voltage);
        }
        throw new RuntimeException("Unreachable");
    }

     ...
}
`
```

Code Smell: “Comments as Design”

```
/**
 * NUR FUER INTERNEN GEBRAUCH (Aufruf durch Controller)!
 *
 * Aktualisiert diese View anhand der uebergebenen ViewDef.
 *
 * @param pViewDef ViewDef, anhand diese View aktualisiert werden soll.
 * @return receiver modified
 */
public void updateWith( ViewDef pViewDef ) {
    if (pViewDef==null) return;
    setUpdatePending(false);
    View lView = replaceOrUpdateViewFor( pViewDef );
    lView.updatePresentationMode( pViewDef );
    if ( isReinitialize() )
        lView.toReinitialize();
}
```

Technische Hochschule Rosenheim

Refactoring Technik: “Extract Method”

Wenn Codefragmente mehrfach verwendet werden oder logisch zusammengehören, verschieben Sie diesen Code in eine separate neue Methode und ersetzen Sie den alten Code durch einen Aufruf der Methode.

```
void printOwing() {
  printBanner();
  // Print details.
  System.out.println("name: " + name);
  System.out.println("amount: " + getOutstanding());
}
```

```
void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}

void printDetails(double outstanding) {
  System.out.println("name: " + name);
  System.out.println("amount: " + outstanding);
}
```

Refactoring Technik: “Extract Method”

```
public void actionPerformed(ActionEvent e) {
        ...
            // out
            for (int i = 0; i < klasses.size(); i++) {
                System.out.println(klasses.get(i));
            }
            // compile
            for (int i = 0; i < klasses.size(); i++) {
                Compiler.compileClass((Class) klasses.get(i));
            }
            // create database
            DatabaseBuilder lDatabase = new DatabaseBuilder(
                "mysql", "jdbc.mysql.MySqlDriver");
            try {
                lDatabase.executeSrcipt(script, ";");
            } catch (Exception ex) {
                ex.printStackTrace();
            }
            // load classes
            Method lMethod = lClass.getDeclaredMethod(
                "newInstance", new Class[]{Properties.class});
            ...
}
```

Refactoring Technik: “Remove Dead Code”

```
if(false) {
  doSomethingThatUsedToMatter();
}
```

Wird ersetzt durch

```
\\nichts
```

Technische Hochschule Rosenheim

Refactoring Technik: “Replace Constructor with Factory Function”

```
leadEngineer = new Employee(document.leadEngineer, 'E');
```

Wird ersetzt durch

```
leadEngineer = createEngineer(document.leadEngineer);
```

Referenzen

- Refactoring Catalog
- Refactoring Buch
- Refactoring Guru

Design Pattern

Technische Hochschule Rosenheim

- Design Pattern (Entwurfsmuster) sind bewährte Lösungswege für wiederkehrende Designprobleme in der Softwareentwicklung
- Sie beschreiben die essenziellen Entwurfsentscheidungen (Klassen- und Objektarrangements)
- Durch den Einsatz von Design Pattern wird ein Entwurf flexibel, wiederverwendbar, erweiterbar, einfacher zu verwenden und änderungsstabil
- In den Design Patterns manifestiert sich die jahrelange Berufserfahrung vieler Softewareentwicklern.
- Zeitgleich schulen Design Pattern die Fähigkeit zur effektiven objektorientierten Modellierung

Design Pattern

- Do not reinvent the wheel!
- Use Patterns and learn from others!

The “gang of four” (GoF): Design Pattern

- 23 verschiedene Design Patterns (Java, C++ & Smalltalk)
- 3 Kategorien (creational, structural, behavioral)
- Definierte Beschreibungsstruktur
  - Name (einschließlich Synonyme)
  - Aufgabe und Kontext
  - Beschreibung der Lösung
  - Struktur (Komponenten, Beziehungen)
  - Interaktionen und Konsequenzen
  - Implementierung
  - Beispielcode

Design Pattern in Kategorien

Technische Hochschule Rosenheim

Creational Pattern

- Abstraktion
- Macht ein System unabhängig davon, wie sein Objekte instanziiert, zusammengesetzt und repräsentiert werden
- `Factory`
- `Abstract Factory`
- `Builder`
- `Prototype`
- `Singleton`

Design Pattern in Kategorien

Technische Hochschule Rosenheim

Structural Pattern

- Zusammensetzung von Klassen/Objekten zu größeren Strukturen, um neue Funktionalität zu realisieren

- `Adapter`

- `Bridge`
- `Composite`
- `Decorator`
- `Facade`
- `Flyweight`
- `Proxy`

Design Pattern in Kategorien

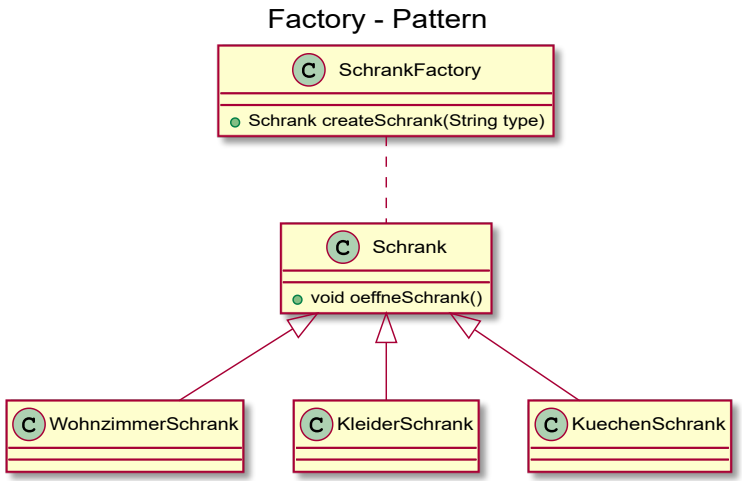
Behavioral Pattern

-Zusammensetzung, Arbeitsteilung, Verantwortlichkeiten zwischen Klassen oder Objekten

`Interpreter` `Template Method` `Chain of Responsibility` `Command` `Iterator` `Mediator` `Memento` `Observer` `State` `Strategy` `Visitor`

Design Pattern: *Factory*

Das Factory Method Entwurfsmuster dient der Entkopplung des Clients von der konkreten Instanziierung einer Klasse. Das erstellte Objekt kann elegant ausgetauscht werden. Oft wird es zur Trennung von (zentraler) Objektverarbeitung und (individueller) Objektherstellung verwendet.



Design Pattern: *Factory*

```
public static void main(String[] args) {
   Schrank schrank = new Kleiderschrank();
   schrank.oeffneSchrank();
}
```

```
public class SchrankFactory {

  public static Schrank createSchrank(String schrankArt) {
    if (schrankArt.equals("Kleiderschrank")){
        return new Kleiderschrank();
    }
    if (schrankArt.equals("Kuechenschrank")){
        return new Kuechenschrank();
    }
    if (schrankArt.equals("Wohnzimmerschrank")){
        return new Wohnzimmerschrank();
    }
    return null;
  }
}
```

Design Pattern: *Factory*

```
public static void main(String[] args) {
  Schrank schrank = SchrankFactory.
      createSchrank("Kleiderschrank");
  schrank.oeffneSchrank();
}
```

Instanzieeren des Schranks ohne *recompile*:

```
public static void main(String[] args) {
  Schrank schrank = SchrankFactory.createSchrank(args[0]);
  schrank.oeffneSchrank();
}
```

Design Pattern: *Singleton*

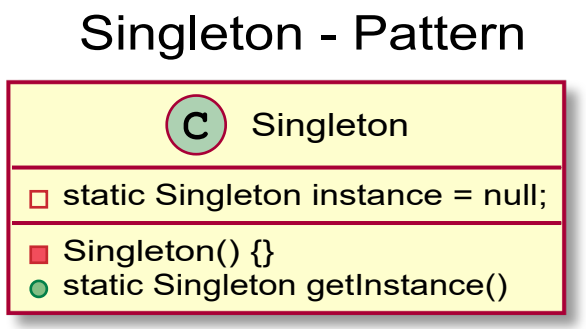
Technische Hochschule Rosenheim

- Das Singleton-Entwurfsmuster gilt im Bezug auf die Komplexität als das einfachste Entwurfsmuster, da es nur aus einer einzigen Klasse besteht.
- Eine Klasse wird immer dann als Singleton implementiert, wenn es nur eine einzige Instanz von ihr geben darf.
- Singleton garantiert, dass es von einer Klasse höchstens eine Instanz geben kann und bietet dazu einen globalen Zugriffspunkt auf diese Instanz.
- Beispiele:
  - Datenbankverbindungen
  - Dialoge
  - Logging-Objekte
  - Objekte, die globale Daten und Einstellungen verwalten

Design Pattern: *Singleton*

Wie bereits erwähnt, besteht das Singleton-Entwurfsmuster nur aus einer einzigen Klasse. Diese verfügt im Wesentlichen über die folgenden Merkmale:

- Eine statische Variable des gleichen Typs
- Einen privaten Konstruktor
- Einer statischen Methode, welche die Instanz zurück gibt



Design Pattern: *Singleton*

```
package singleton;

public class Singleton {
  private static Singleton instance = null;

  private Singleton() {
      /* Initialisation goes here! */
  }

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

Design Pattern: *Composite*

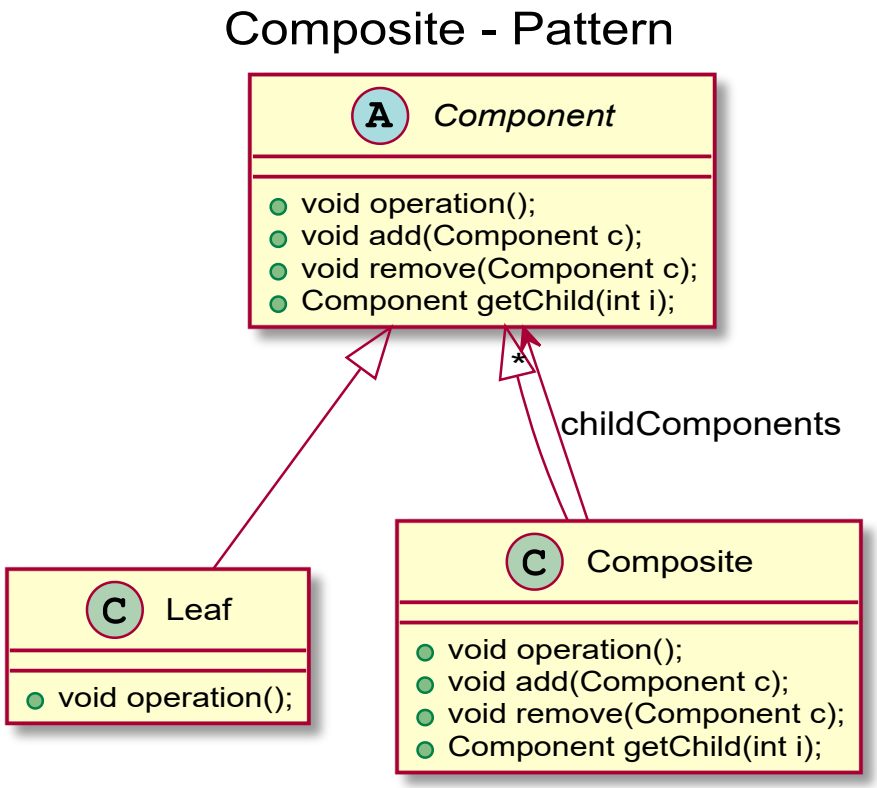
Technische Hochschule Rosenheim

Das Composite Entwurfsmuster ermöglicht es, eine verschachtelte Struktur einheitlich zu behandeln, unabhängig davon, ob es sich um ein atomares Element oder um ein Behälter für weitere Elemente handelt

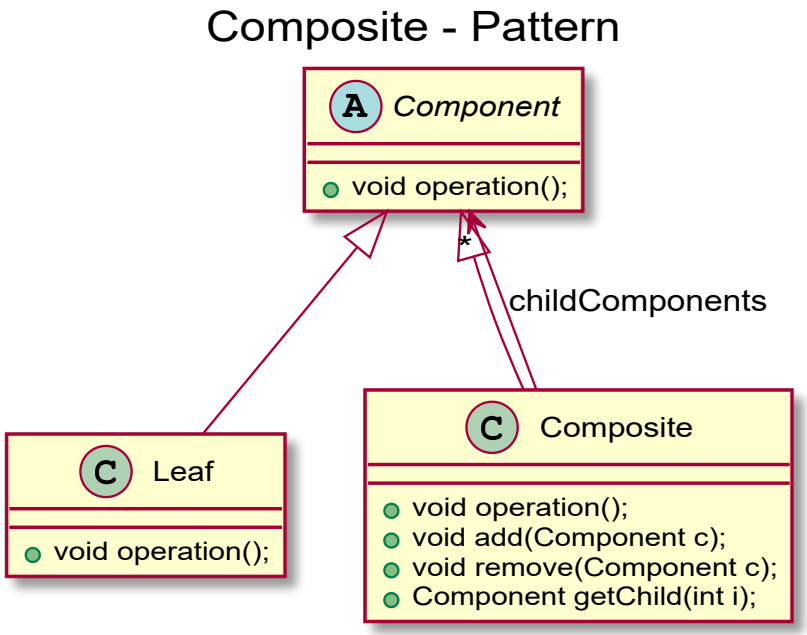
- Es wird eine gemeinsame Schnittstelle für die Elementbehälter (Composite, Kompositum; Aggregat, Knoten) und für die atomaren Elemente (Leaf, Blatt) definiert: Component.
- Diese Schnittstelle Component definiert die Methoden, die gleichermaßen auf Composites und auf Leafs angewandt werden sollen. Composites delegieren oft Aufrufe (operate()) an ihre Components, die atomare Leafs oder wiederrum zusammengesetzte Composites seien können.

Ein Client muss nicht mehr zwischen Composite und Leaf unterscheiden!

Design Pattern: *Composite*



Design Pattern: *Composite* (Alternative!)



Design Pattern: *Composite*

```
class Composite extends Component{

    //hier: Components als Liste vorgehalten
    private List<Component> childComponents = new ArrayList<Component>();

    //rekursiver Aufruf auf kindComponents
    public void operation() {
        System.out.println("Ich bin ein Composite. Meine Kinder sind:");
        for (Component childComps : childComponents) {
            childComps.operation();
        }
    }

    //Überschreiben der Defaulimplementierung
    public void add(Component comp) {
        childComponents.add(comp);
    }
    public void remove(Component comp) {
        childComponents.remove(comp);
    }
    public Component getChild(int index) {
        return childComponents.get(index);
    }
}
```

Referenzen

- Design Patterns @ Refactoring Guru
- TutorialPoint - Design Pattern
- [GeeksForGeeks - Design Pattern] (https://www.geeksforgeeks.org/design-patterns-set-1-introduction/)
- OODesign - GoF Pattern

Fragen?