Softwaretechnik (WS 2005/2006) - Übungsblatt 9

Jan Hendrik Dithmar - Matrikelnummer 2031259
Übungsgruppe Mi, 8-10 Uhr
Dirk Heine - Matrikelnummer 2030941
Übungsgruppe Di, 14-16 Uhr

Rückgabe: Übungsgruppe Di, 14-16 Uhr

Aufgabe 1

	Abstraktion	Einkapselung	Modularität	Hierarchie
Klassen	ja	ja	ja	ja
Schnittstellen	ja	ja	ja	nein
Sichtbarkeitsmodifizierer	nein	ja	ja	nein
Vererbung	ja	nein	ja	ja
Dynamische Bindung	nein	nein	ja	ja

- **Abstraktion**
  - **Klassen:** Klassen tragen zum Prinzip der Abstraktion bei, da durch sie mittels geeigneter Schnittstellen die Interna der Klassen verborgen werden können, und es somit möglich ist von der konkreten Implementierung des Objektes zu abstrahieren.
  - **Schnittstellen:** Schnittstellen tragen zum Prinzip der Abstraktion bei, da durch sie die gesamten Interna einer Klasse nach außen verborgen werden, und somit von den Details der Klasse abstrahiert werden kann.
  - **Sichtbarkeitsmodifizierer:** Sichbarkeitsmodifizierer tragen nicht zum Prinzip der Abstraktion bei. Sie können zwar einzelne Werte nur bestimmten Objekten zugänglich machen, diese aber nicht effizient nach außen verbergen.
  - **Vererbung:** Vererbung trägt zum Prinzip der Abstraktion bei, da hierdurch abstrakte Oberklassen gebildet werden können, welche die tatsächlichen Details erst einmal ausblenden, und somit eine abstrakte Sicht auf das Objekt ermöglichen.
  - **Dynamische Bindung:** Dynamische Bindung trägt nicht zum Prinzip der Abstraktion bei, da hierdurch nicht unmittelbar ein Verbergen der konkreten Informationen eines Objektes möglich wird.
- **Einkapselung**
  - **Klassen:** Klassen tragen zum Prinzip der Einkapselung bei, da durch sie die Interna der Klasse innerhalb der Klasse selbst bearbeitet werden, und somit nicht nach außen dringen.
  - **Schnittstellen:** Schnittstellen tragen zum Prinzip der Einkapselung bei, da sie es unmöglich machen sich von den Internas eines andern Objektes abhängig zu machen (wenn sie richtig verwendet werden).

  - **Sichtbarkeitsmodifizierer:** Sichtbarkeitsmodifizierer tragen zum Prinzip der Einkapselung bei, da durch sie erst Schnittstellen und damit Klassen ihre Internas verbergen können.
  - **Vererbung:** Vererbung trägt nicht zum Prinzip der Einkapselung bei, da eine Vererbungshierarchie keine zusätzlichen Daten einkapselt, welche nicht schon die einzelnen Klassen eingekapselt haben.
  - **Dynamische Bindung:** Dynamische Bindung trägt nicht zum Prinzip der Einkapselung bei, da auch sie keine Möglichkeit bietet, die internen Daten nach außen zu verstecken.

- **Modularität**
  - **Klassen:** Klassen tragen zum Prinzip der Modularität bei, da sie das Mittel sind, um aus dem Gesamten einzelne Objekte zu formen.
  - **Schnittstellen:** Schnittstellen tragen zum Prinzip der Modularität bei, da durch sie die Interaktion, die Kopplung, sowie die Unabhängigkeit der einzelnen Objekte untereinander sichergestellt werden kann, indem die einzelnen Objekte aufgrund der Schnittstelle automon agieren können müssen.
  - **Sichtbarkeitsmodifizierer:** Sichtbarkeitsmodifizierer tragen zum Prinzip der Modularität bei, da sie sich nicht von den Internas andere Objekte abhängig machen, und somit die Wiederverwendbarkeit der anderen Objekte sicherstellen können.
  - **Vererbung:** Vererbung trägt zum Prinzip der Modularität insofern bei, als dass man abstrakte Oberklassen bilden kann, auf denen man arbeiten kann, und sich somit bei eventuellen Abhängigkeiten nicht auf die Abhängigkeit von einer konkreten Klasse beziehen muss.
  - **Dynamische Bindung:** Die Dynamische Bindung trägt insoweit zum Prinzip der Modularität bei, als dass man sich erst zur Laufzeit entscheiden kann welche Bindungen man eingehen möchte, und dies nicht im Quellcode fest vorgeben muss.

- **Hierarchie**
  - **Klassen:** Klassen tragen zum Prinzip der Hierarchie bei, da sie die Mittel sind um komplexe Vererbungshierarchien zu erstellen.
  - **Schnittstellen:** Schnittstellen tragen nicht zum Prinzip der Hierarchie bei, da durch die Wahl der Schnittstelle erst einmal keine Hierarchie geschaffen werden kann.
  - **Sichtbarkeitsmodifizierer:** Sichtbarkeitsmodifizierer tragen nicht zum Prinzip der Hierarchie bei, da die Modifikation der Sichtbarkeit eines Objektes keine Hierarchie impliziert.

- **Vererbung:** Vererbung trägt zum Prinzip der Hierarchie bei, da durch sie erst große Abhängigkeitsbäume, welche eine Hierarchie vorgeben, ermöglicht werden.
- **Dynamische Bindung:** Dynamische Bindung trägt zum Prinzip der Hierarchie bei, da mittels Dynamischer Bindung das Open/Close Prinzip verwirklicht werden kann.

Aufgabe 2

(a)

- **Open-Close-Prinzip:** Das Open-Close-Prinzip beschreibt, dass ein Objekt zwar auf der einen Seite geschlossen für Änderungen, also in seinem aktuellen Zustand unveränderlich, jedoch auf der anderen Seite offen für Erweiterungen, also jederzeit an eventuelle Änderungen anpassbar sein soll. Dieses Prinzip kann man im Objektorientierten Ansatz zum Beispiel durch Vererbung verwirklichen. Während die Klasse *Auto* unveränderlich ist, kann man jederzeit, falls nötig, durch Vererbung ein Auto erzeugen welches sehr gut im Gelände ist, und auf der anderen Seite auch ein Auto erzeugen welches zwar im Gelände nicht vorwärtskommt, dafür aber auf festen Straßen sehr schnell fährt. Diese beiden Klassen *Geländewagen* und *Rennwagen* sind somit von Auto abgeleitet und selbst auf die neuen Anforderungen angepasst. Die Oberklasse *Auto* wurde jedoch niemals verändert.

- **Abhängigkeits-Prinzip:** Abhängigkeiten sollten nur zu Abstraktionen und nicht zu konkreten Unterklassen bestehen. Als Beispiel kann man ebenfalls obiges Beispiel anführen. Ein spezieller Typ Auto (z.B. ein *Panzer*) sollte nicht von *Geländewagen*, sondern von *Auto* abgeleitet sein.

Im gegebenen Beispiel sieht es so aus, dass die Klasse *University* eine Methode `getTotalSalaries()` hat, um über die Klasse *Employee*, und deren Methode `getSalary()`, welche das Gehalt eines *Employee* ausgibt, um die Gesamtsumme der Gehaltszahlungen der Universität auszugeben. Will eine Universität auf einmal nicht nur Hiwis und Professoren Geld zahlen, sondern auch den Studenten, so reicht es aus, eine weitere Klasse *Student* von der Klasse *Employee* abzuleiten, und die entsprechenden Mehtoden in *Student* zu implementieren. Somit kann die Universität weitere Angestellte bekommen ohne dass sie dazu selbst verändert werden müsste. Auch die Änderungen in der Klasse *Employee* belaufen sich (falls nötig) auf Kleinigkeiten. Somit ist die gesamte Hierarchie einfach um weitere Gehaltsempfänger zu erweitern, ohne dass die Struktur als solches dafür groß geändert werden müsste.

(b)

Problem: Diese Implementierung ist zu speziell: die Auswertung verwendet hauptsächlich Details der Implementierung. Sie ist also sehr nahe an der gegebenen Struktur und es fehlt ihr an Abstraktion. Es gibt keine Klasse, die es erlaubt, das Gehalt aller Professoren zu ermitteln. Änderungen sind nur schwer durchzuführen, da klar definierte Schnittstellen fehlen. Das Demeter-Prinzip wird ebenfalls verletzt, da sich die Funktion zum Ermitteln der Daten ihren Weg durch die ganze Klassenhierarchie bahnt.

Verbesserte Lösung:
Wir müssen zunächst das Design geringfügig ändern:
Die Klasse *University* bekommt eine neue Funktion `double getTotalProfSalaries()`

```
public double getTotalProfSalaries() {
  double sum = 0;
  Iterator e = employees.iterator();
  while (e.hasNext()) {
    if(e instanceof Professor) sum+=((Employee e.next()).getSalary();
  }
  return sum;
}

public double getProfSalaries() {
  double sum = 0;
  University u;
  for (int i = 0; i < universities.size(); i++) {
    u = (University) universities.get(i);
    sum+=u.getTotalProfSalaries();
  }
  return sum;
}
```

Mit diesen Änderungen wird das Demeter-Prinzip eingehalten. Außerdem findet das Open-Close-Prinzip Anwendung, da bei der Implementierung weitestmöglich abstrahiert wird.

Aufgabe 3

(a)

- **Vorbedingungen:** keine
- **Nachbedingungen:**
  - `size = size + 1`
  - `pop(push(x,Stack)) = Stack`
  - `top(push(x,Stack)) = x`

(b)

Die Methode `push()` des *BoundedStack* sollte eine Exception werfen, falls trotz vollem Stack ein Objekt hinzugefügt werden soll. Das hinzuzufügende Objekt selbst sollte nicht hinzugefügt werden, der Stack also unverändert bleiben.

(c)

- **Vorbedingungen:**
  - `isFull() == false` also `size < maxsize`
- **Nachbedingungen:**
  - `size = size + 1`
  - `size` $\leq$ `maxsize`
  - `pop(push(x,Stack)) = Stack`
  - `top(push(x,Stack)) = x`

(d)

In der von uns gewählten Implementierung von *BoundedStack* ist diese keine Erweiterung des Stack nach dem Liskov-Prinzip, da *BoundedStack* zwar stärkere Nachbedingungen hat, jedoch auch die stärkere Vorbedingung, dass der Stack seine maximale Größe beim Hinzufügen eines Elementes noch nicht erreicht haben darf, was dem Prinzip von Liskov widerspricht.

Aufgabe 4

(a)

```
aspect a4a {
  pointcut publicJavaUtilCall():
    (call (public * java.util.*.* (..)));

  before(): publicJavaUtilCall() {
    System.out.println("Entering: " + thisJoinPoint);
  }
}
```

(b)

```
aspect a4b {
  public long MyVector.countNullItems() {
    return this.capacity() - this.size();
  }
}
```

(c)

```
aspect a4c {
  pointcut StrictMathSqrtChecking():
    call (public double Math.sqrt(double));

  after(double a) returning: (
      StrictMathSqrtChecking() &&
      args(a) &&
      !cflow(StrictMathSqrtChecking())
    ) {
    double result_math = Math.sqrt(a);
    double result_strictmath = StrictMath.sqrt(a);
    if (result_math != result_strictmath) {
      System.out.println(
          "In " + thisJoinPoint.getSourceLocation() + ": "
          + thisJoinPoint + ":\n\t"
          + "Math.sqrt(" + a + ")=" + result_math
          + ", while StrictMath.sqrt(" + a + ")=" + result_strictmath);
    }
  }
}
```