

Fragen, Antworten, Kommentare zur aktuellen Vorlesung

Bitte an das Ausfüllen des Evaluationsbogens (s. Mail vom 16.12) denken.

Frage: Sollten die Tests zum Stapel eigentlich alle in einer Klasse stehen?

Antwort: Generell werden die Tests einer Klasse immer in einer Testklasse zusammengefasst. Allerdings gibt es keine Regel, dass es so sein muss. Eine Aufteilung ist insbesondere dann sinnvoll, wenn es sich um komplexe zu testende Methoden handelt oder um Methoden, die häufiger in einem Projekt angepasst werden müssen. Dies ist wieder einer der Fragen mir der Antwort, es hängt vom Projektkontext ab. Für das erste Semester gibt es keine Vorgaben. Keine Vorgabe, aber ein Wunsch ist es, dass Testmethoden sprechende Namen enthalten, also im Methodennamen steht was mit wem gemacht wird und welches Ergebnis erwartet wird, z. B. „testPushAufLeerenStapelErlaubt“. (Ich habe dies bei den Messreihentests leider nicht eingehalten.)

Hinweis: Unter <https://youtu.be/WNw3JbP2RCg> finden Sie die Erklärungen von Beispiellösungen zu zwei Teilaufgaben der Messreihe Teil 2 online. Weiterhin gibt es ebenfalls ein Videos zur Aufgabe mit dem interaktiven Interaktionsbrett unter https://youtu.be/wVNGKb6U_ME und zum Basketballspiel in <https://youtu.be/YhvkBYtCyZ4>.

Hinweis: Bei Programmen im Praktikum ist mir aufgefallen, dass relativ selten „this.“ genutzt wird. Da wir jetzt alle Schreibweisen kennen, zeigt das folgende Beispiel, wie hilfreich solche Bezeichnungen vor Variablen und Methoden sind. Da dies in der Praxis in ordentlichen Unternehmen immer gefordert wird, sollte dieser Stil frühzeitig genutzt werden. Der Hintergrund ist, dass immer alle entwickelnden Personen eines Teams den Code anderer lesen und bearbeiten können sollen. Andere Programmiersprachen haben vergleichbare Regeln.

```
public void sprechenderMethodenname(String produktname, int id) {
    this.name = produktname; // klar erkennbar, auf der linken Seite ist
                             // eine Objektvariable
    super.id = id;           // klar erkennbar, auf der linken Seite ist
                             // eine Objektvariable die (irgendwoher)
                             // geerbt wurde
    Status.count++;         // klar erkennbar, dass eine Klassenvariable
                             // genutzt wird
    thisweiterleiten(id);   // klar erkennbar, es wird erwartet, dass sich
                             // die Objektmethode in dieser Klasse befindet
    super.markieren(this.name); // klar erkennbar, dass eine Objektmethode
                             // (irgendeiner) beerbten Klasse aufgerufen
                             // werden soll
    Status.sichern(this.name); // klar erkennbar, dass eine Klassenmethode
                             // aufgerufen wird
}
```

Kurz ergänzend zum Thema dynamische Polymorphie, dem zentralen Thema der Objektorientierung, was genauer auch in „nur“ objektbasierten Sprachen wie Go die zentrale Rolle spielt. Durch dynamische Polymorphie können Systeme flexibel gestaltet und einfach zur Laufzeit verändert werden, da jeweils auf die passenden Methoden zugegriffen wird. Da es bei der Praktikumsaufgabe 31 damit vereinzelt Probleme gab, hier nochmals eine andere Beispielskizze. Die Abbildung rechts zeigt eine Vererbungshierarchie, also X3 erbt von X2, weiterhin sind nur die Methodennamen von Methoden angegeben, die in diesen Klassen ausprogrammiert sind. Da alle Klassen von der Klasse Object erben, wurde sie nach oben gesetzt. Generell kann man Variablen eines Typen Objekte dieses Typen oder einer erbenden Klasse zuweisen, also

```
X1 x1 = new X1();
X1 x2 = new X2();
X1 x3 = new X3();
ist alles erlaubt, hingegen
```

```
X3 x31 = new X1();
nicht. Der angegebene Typ auf der linken Seite ist relevant,
wenn es darum geht, welche Methoden aufgerufen werden
können. Für x2 ist z. B. der Aufruf x2.m2() so nicht möglich.
```

Die dynamische Polymorphie kommt ins Spiel, wenn für ein Objekt bestimmt werden soll, welche Methode auszuführen ist. Hierbei wird geschaut, welcher Typ, also welche Klasse zur Erzeugung des Objekts genutzt wurde. Wird `x1.m1()` ausgeführt, wird der mit `//1` markierte Code genutzt, bei `x3.m1()` der mit `//2` markierte Code. Sollte dann eine Methode nicht in der erzeugenden Klasse enthalten sein, wird in der Klasse gesucht, von der die erzeugende Klasse geerbt hat. Wird also `x3.m3()` aufgerufen, wird der mit `//3` markierte Code in der Klasse X2 ausgeführt. Dies ist auch der Grund warum es für jedes Objekt einer neuen Klasse gilt, dass immer die `toString()`-Methode ausgeführt werden kann, da sie in der Klasse Objekt definiert wird.

Wichtig ist, dass dieser Ansatz in der zu Erzeugung genutzten Klasse zuerst nach der Methode zu suchen bei jedem neuen einfachen Methodenaufruf angewandt wird. Beim Aufruf von `x3.m3()` wird die Methode `m3()` in X2 genutzt, beim folgenden Aufruf von `this.m4()` wird wieder in X3 nach der Methode gesucht und an der mit `//4` markierten Stelle gefunden.

Vom erwähnten einfachen Methodenaufruf wird nur abgewichen, wenn vor dem Aufruf `super` steht. Damit wird in der Programmierung festgelegt, dass ausgehend von der aktuell genutzten Klasse K die Methode in der Klasse gesucht wird, von der K geerbt hat. Wird z. B. `x3.m4()` aufgerufen, wird die mit `//5` markierte Methode `m4` in X2 aufgerufen, die dann die mit `//6` markierte Methode `mx` in der Klasse X1 aufruft. Da in dem folgenden Aufruf kein `super` steht, wird für `m1()` wieder zuerst in der Klasse X3 gesucht.

Der Ablauf wirkt auf Personen am Anfang etwas verwirrend, später wird so eine wilde Jagd durch die Klassen selten stattfinden. Oftmals ist die obere Klasse ein Interface, zu dem es dann mehrere Klassen gibt, die dieses Interface implementieren/realisieren. Die Suche nach der auszuführenden Methode ist dann trivial und hängt nur vom zur Erzeugung des Objekts genutzten Klasse ab.

```
public class Object {
    // in Java gegeben
}
```

```
public class X1 {
    public void m1() {} //1
    public void mx() { // 6
        this.m1();
    }
}
```

```
public class X2 extends X1 {
    public void m2() {}
    public void m3() { //3
        this.m4();
    }
    public void m4() { //5
        super.mx();
    }
}
```

```
public class X3 extends X2 {
    public void m1() {} //2
    public void m4() { //4
        super.m4();
    }
    public void m3() {}
}
```

