Grundlagen der Informatik 1
WS 2008/09

Prof. Mühlhäuser, Dr. Rößling, Melanie Hartmann, Daniel Schreiber
http://proffs.tk.informatik.tu-darmstadt.de/gdi1

Übung 13 - Lösungsvorschlag Version: 1.0 02.02.2009

---

1 Mini Quiz

1. Generics
   - Gegeben eine Klasse Shape mit Unterklassen Circle und Rectangle. Welche Typen kann man in einer List<? **extends** Shape> speichern?

     ☒ Shape ☐ Object ☒ Rectangle ☐ String
   - Welche Werte sind für den Platzhalter im Ausdruck Vector<Platzhalter> möglich?

     ☒ ? ☒ A **extends** B ☐ A **implements** B ☒ A
2. Assertions

   ☒ Assertions ermöglichen das Prüfen von Vorbedingungen.

   ☐ Die Fehler die von Assertions ausgelöst werden, sind vom Typ `Exception` und nicht vom Typ `Error`.
3. JUnit / Testing

   ☐ JUnit eignet sich zum Debuggen von komplexen Java-Programmen.

   ☐ Besteht ein System alle Unit-, Integrations- und Systemtests, so kann man davon ausgehen, dass es fehlerfrei ist.

   ☒ Kontrollfluss-orientierte Testverfahren erfordern es, den Programmcode zu kennen.

2 Fragen

1. Nennen Sie einige Vor- und Nachteile der statischen Typprüfung gegenüber dynamischer Typprüfung.

   Lösungsvorschlag:
   Vorteile:
   - *Steigerung der Ausführungsgeschwindigkeit*
   - *Effizientere Kompilierung*
   - *Entwicklungsgewinn im Kleinen ("think before you write"): Vermeidung von Routinefehlern, Dokumentation des Codes, etc.*
   - *Entwicklungsgewinn im Großen: Klarere (weil erzwungene) Schnittstellen, Abstraktionsgewinn*

Nachteile:

- *Längerer Quelltext*
- *unflexibel*
- *Bevormundung des Programmierers ("I know what I am doing")*

2. Was sind die Hauptunterschiede zwischen White-Box Tests und Black-Box Tests?

   Lösungsvorschlag:

   - *White-Box Tests (auch bekannt als Glass-Box Tests oder strukturelle Tests) basieren auf der Programmstruktur. Ein Beispiel für solche Tests sind Überdeckungstestverfahren. Ändert sich die Implementierung einer Funktion oder einer Klasse, müssen auch die entsprechenden White-Box Tests angepasst werden. White-Box Tests können unbenutzten Code entdecken, sie können aber z.B. nicht aufdecken, ob Teile einer Spezifikation nicht implementiert wurden.*
   - *Black-Box Tests (funktionale Tests) ergeben sich aus den Anforderungen an das Programm. In der Objektorientierung werden dabei Klassen als 'schwarze Kästen' betrachtet, d.h. es werden keine Annahmen darüber getroffen, wie eine Klasse implementiert ist. Grundlage für den Test ist nur die öffentliche Schnittstelle. Daher ändern sich Black-Box Tests auch nicht, wenn sich die interne Implementierung einer Klasse ändert.*

     Mit Black-Box Tests werden für gewöhnlich die vorgegebenen Spezifikationen kontrolliert.

   Die beiden Testarten können sich **nicht** *gegenseitig ersetzen, da sie unterschiedliche und voneinander unabhängige Aspekte der Software testen (interne Implementierung vs. Konformität zu einer Spezifikation).*

3. In welcher Phase der Software-Entwicklung sollte man mit dem Testen beginnen? Wann kann man mit dem Testen aufhören und woher weiß man, dass man keine weiteren Tests mehr benötigt?

   Lösungsvorschlag:
   Mit dem Testen sollte so früh wie möglich begonnen werden. Mit dem Entwurf der Tests kann man im Prinzip schon bereits in der Design-Phase beginnen. Bei der Implementierung sollte parallel zu jedem Programm-Baustein (Funktion, Klasse, etc.) ein Unit-Test geschrieben werden (z.B. mit JUnit). Mit dem Testen aufhören sollte man erst dann, wenn die Software Ihre Spezifikation erfüllt und das Programm die gewünschte Funkionalität bietet.

   Hier muss noch erwähnt werden, dass Software eigentlich generell im Laufe ihrer Lebenszeit i.d.R. angepasst und verändert werden. Somit muss man seine Software bis an ihr Lebensende immer wieder testen :-(

4. Wann sollte man Assertions verwenden? Wann sollte man stattdessen lieber Exceptions benutzen?

   Lösungsvorschlag:

   - *Exceptions signalisieren ein außergewöhnliches (exceptional) Terminieren einer Methode, wie zum Beispiel eine unerlaubte Eingabe des Benutzers oder ein Zugriffsversuch auf eine nicht existierende Datei. Exceptions können vom Programmierer* **nicht** *ausgeschlossen werden, er kann für den Fall ihres Auftretens jedoch Gegenmaßnahmen (in Form des try/catch-Blocks) vorsehen.*

- *Assertions hingegen sind Aussagen, die zu jedem Zeitpunkt wahr sein sollten. Ist eine Assertion verletzt, muss ein Fehler im Programmcode vorliegen. Assertions helfen, die korrekte Funktionalität eines Programs zu prüfen, ohne dass dazu notwendigerweise Exceptions auftreten müssen. Zum Beispiel kann die Überprüfung, ob eine bestimmte Variable immer einen bestimmten Mindestwert hat, in einer Assertion festgelegt werden, ohne dass dafür eine Exception geworfen werden muss. Assertions werden in der Regel nur in der Entwicklungs- und Testphase eingesetzt, im Produktivbetrieb schaltet man sie ab.*

5. Welchen Vorteil bieten Generics? Welches Problem wird dadurch gelöst?

   Lösungsvorschlag:
   Generics erlauben die Definition von allgemeinen Methoden oder Klassen, wie z.B. Collection-Typen. Durch Angabe eines Typparameters können verschiedene Implementierungen unterschiedliche Typen konsistent verwenden, beispielsweise List<String>.

   Ohne Generics müssen allgemeine Methoden Parameter vom Typ Object anbieten, wodurch Typecasts notwendig werden, die zu Typfehlern zur Laufzeit führen können. Mit generischen Typen entfallen diese Casts und Typfehler können zur Compilierzeit erkannt werden.

3 Verifikation und Validierung

Zunächst: Ein kleines Beispiel für kritische Fehler in Software...

Im Bereich der Medizin gab es in den Jahren 1985 bis 1987 auf Grund von Softwarefehlern viele Todesfälle. Mehrere Patienten starben, nachdem sie zur Krebsbehandlung mit Therac-25 bestrahlt wurden. Wegen einer zu hohen Strahlendosis wurden ihre Zellen nachhaltig geschädigt.... (Das medizinische Gerät Therac-25 war ein linearer Teilchenbeschleuniger zur Strahlentherapie für krebskranke Patienten.)

Wir betrachten nun nochmals die beiden Begriffe aus der Vorlesung: Verifikation und Validierung. Schauen Sie sich bitte folgenden Text an:

Während die _____ den Output einer Entwicklungsphase auf die Korrektheit mit der vorherigen Phase untersucht, wird die _____ benutzt, um das Gesamtsystem mit den Kundenanforderungen zu vergleichen.
Die zentrale Tätigkeit bei der _____ ist das Testen. Das Gesamtsystem wird bei Ende des Prozesses getestet, ob es der Spezifikation entspricht oder nicht. Die zentrale Tätigkeit bei der _____ ist der Beweis mit der formalen _____.

Setzen Sie die passenden Begriffe (Verifikation / Validierung) korrekt ein.

Lösungsvorschlag:

Während die **Verifikation** *den Output einer Entwicklungsphase auf die Korrektheit mit der vorherigen Phase untersucht, wird die* **Validierung** *benutzt, um das Gesamtsystem mit den Kundenanforderungen zu vergleichen.*
Die zentrale Tätigkeit bei **Validierung** *ist das Testen. Das Gesamtsystem wird bei Ende des Prozess getestet, ob es den Kundenanforderungen entspricht oder nicht. Die zentrale Tätigkeit bei der* **Verifikation** *ist der Beweis mit der formalen* **Verifikation**.

4 Fehler

Starten Sie bitte Eclipse, legen Sie eine einfache Klasse an und rufen Sie die unten angegebene Funktion auf (wird im Portal bereitgestellt als eclipse.java). Welches Ergebnis erhalten Sie? Wenn Sie keinen Computer dabei haben, fragen Sie Ihren Tutor nach der Lösung.

Was für ein Ergebnis hätten Sie erwartet? Was für ein Bug hat sich hier eingeschlichen und wie könnte man diesen beheben?

```
1 public void hmmmm()
2 {
3         double test = 0;
4
5         test += 277.04;
6         test += 222.67;
7
8         System.out.println(test);
9 }
```

Lösungsvorschlag:
Das Ergebnis das Java uns hier liefert lautet: 499.71000000000004

Doch wie kommt das? Es handelt sich hier um einen Genauigkeitsfehler, die nicht nur in Java existieren. Der eigentliche Grund dafür ist die begrenzte Genauigkeit des Datentyps double.

*Um diesen Rundungsfehler (bzw. Genauigkeitsfehler) beheben zu können, benötigen wir entweder einen Datentyp, der mit einer noch höheren Genauigkeit umgehen kann (***BigDecimal** *ist hierfür beispielsweise eine geeignete Klasse) oder aber wir runden das Ergebnis auf 2 Nachkommastellen ab, sodass wir:* **499.71** *erhalten, wie es eigentlich sein sollte.*

5 Generics

1. Gegeben sei eine Klassenhierarchie für Nahrungsmittel mit der Basisklasse *Nahrungsmittel* und ihren Unterklassen *Obst, Fleisch* und *Gemuese*. Die Klasse *Obst* hat zusätzlich noch die Unterklasse *Apfel*.

   Die Methode

   double getNV(LinkedList<Nahrungsmittel>)

   berechnet den Gesamtkaloriengehalt der ausgewählten Nahrungsmittel.

   Geben Sie für jedes Element der folgenden Tabelle an, ob die entsprechende Operation *legal* (T) ist oder nicht (F). So soll in der ersten freien Zelle der ersten Zeile ein T stehen, wenn man die Methode *getNV* mit einem Parameter vom Typ LinkedList auch mit einer Instanz von LinkedList<Nahrungsmittel> aufrufen darf.

Statischer Typ v in LinkedList<v>→ **Methodendeklaration ↓**	<Nahrungsmittel>	<Obst>	<Apfel>
double getNV(LinkedList x)	T	T	T
double getNV(LinkedList <Obst> x)	F	T	F
double getNV(LinkedList <?> x)	T	T	T
double getNV(LinkedList <? **extends** Obst> x)	F	T	T
double getNV(LinkedList <? **super** Obst> x)	T	T	F

2. Gegeben sein nun der folgende Code:

```
1 public void x(ListDef v) {
2   --??-- x = v.get(0);
3   v.add(new Apfel("Boskoop", 43));
4 }
```

In der folgenden Tabelle finden Sie in jeder Zeile wieder eine konkrete Deklaration des Parametertyps *ListDef*.

In der zweiten Spalte tragen Sie bitte ein, welcher Typ für *–??–* in Codezeile 2 eingesetzt werden sollte. Dabei sollten Sie einen möglichst präzisen Typ angeben und daher `Object` nur dann angeben, wenn es keinen konkreteren Typ gibt. Gehen Sie dabei davon aus, dass die Liste existiert und es ein Element an der Position 0 gibt.

In der letzten Spalte geben Sie bitte an, ob die Einfügeoperation in Codezeile 3 jeweils zulässig ist. Verwenden Sie auch hierfür *T* bzw. *F*.

Konkreter Typ für ListDef	Typ von --??-- in Zeile 2	v.add(new Apfel(...))
void x(LinkedList v)	Object	T
void x(LinkedList<Apfel> v)	Apfel	T
void x(LinkedList <?> v)	Object	F
void x(LinkedList <? **extends** Apfel> v)	Apfel	F
void x(LinkedList <? **super** Apfel> v)	Object	T

3. Im Rahmen des Sonderangebots "2 Essen zum Preis von einem" werden zwei Mahlzeiten zusammengefügt. Hierfür gibt es die Methode *createMenu()*, die aus den zwei übergebenen *LinkedList*-Objekten eine *neue* LinkedList des *am besten passenden Typs* machen soll:
   - Das Zusammenlegen einer LinkedList<Gemuese> mit einer LinkedList<Fleisch> führt zu einer LinkedList<Nahrungsmittel>.
   - Das Zusammenlegen von *Äpfeln* mit anderem *Obst* führt zu einer LinkedList<Obst>.
   - Werden nur Listen gleichen Typs zusammengelegt, soll am Ende eine Liste des gleichen Typs entstehen.

Lösen Sie das Problem, indem Sie für die unten angegebene (korrekte) generische Implementierung die passenden Typen *T, U, V* spezifizieren. Begründen Sie in zwei bis drei Sätzen Ihre Typwahl.

```
1 public<T..., U..., V...> LinkedList<V> createMenu
2                 (LinkedList<T> menuA, LinkedList<U> menuB){
3   LinkedList<V> totalMenu = new LinkedList<V>();
4   totalMenu.addAll(menuA);
```

```
5     totalMenu.addAll(menuB);
6     return totalMenu;
7 }
```

Korrekte Deklaration des generischen Typs **T**:

Korrekte Deklaration des generischen Typs **U**:

Korrekte Deklaration des generischen Typs **V**:

Lösungsvorschlag:
Eine, nicht notwendigerweise die einzige, korrekte Lösung nutzt folgende Belegung:

<V **extends** *Nahrungsmittel, T* **extends** *V, U* **extends** *V>*

Hiermit wird sichergestellt, dass die Ergebnis-Liste immer mindestens mit Nahrungsmittel *typisiert wird (eine der obigen Anforderungen). Gleichzeitig müssen T und U identisch sein zu V oder Untertypen davon. In den Beispielkonstellation herrscht damit zwar noch etwas Wahlfreiheit (man kann das Ergebnis auch einer LinkedList<Object> aufzwingen), die obigen Anforderungen werden aber erfüllbar.*