Proseminar Programmiermethodik

Universität Innsbruck - Institut für Informatik

A. Blaas, M. Mayerl, B. Murauer, M. Tschuggnall

Blatt 5: Exceptions und Testing

Diskussionsteil

a) ★ Die folgende Methode liest eine Datei ein, verarbeitet diese, und gibt ein Ergebnis vom Typ `Result` aus. Falls die angegebene Datei nicht existiert, soll eine `FileNotFoundException` geworfen werden. Vervollständigen Sie den unterstehenden Code entsprechend.

```
1       public static double processFile(String path) {
2           Path p = Paths.get(path);
3           if (Files.notExists(p)) {
4               //ToDo: wirf FileNotFoundException
5           }
6
7           double result = .0;
8           //...snip (Logik zum Verarbeiten der Datei) ...
9           return result;
10      }
```

Hinweis: Sie müssen an **zwei** Stellen im Code Ergänzungen/Änderungen vornehmen. Gehen Sie dabei davon aus, dass `java.io.FileNotFoundException` bereits importiert ist.

Hinweis: Falls Sie den Code in Eclipse bearbeiten wollen, müssen Sie in ihrer .java-Datei `import`-Statements für die folgenden Klassen hinzufügen:

- `java.nio.file.Files`
- `java.nio.file.Paths`
- `java.nio.file.Path`
- `java.io.FileNotFoundException`

b) ★ In dieser Aufgabe werden Sie Unit Testing anhand eines ersten kleinen Beispiels kennen lernen.

Als ersten Schritt dafür müssen Sie auf dem Rechner, mit dem Sie arbeiten, das Eclipse-Plugin für TestNG installieren. TestNG ist das Testing-Framework, das wir in diesem Kurs verwenden werden.

Öffnen Sie für die Installation den Eclipse Marketplace (Help > Eclipse Marketplace...) und tippen sie *TestNG* in die Suchleiste ein. Wählen Sie anschließend aus den Suchergebnissen *TestNG for Eclipse* aus und klicken Sie auf Install. Falls das Plugin bei Ihnen bereits installiert ist, ist der Install-Button inaktiv und mit *Installed* beschriftet.

Importieren Sie jetzt das Projekt `MyFirstUnitTest.zip` (OLAT) in Eclipse (File > Import... > General > Existing Projects into Workspace). Das Projekt besteht aus zwei Klassen. Die Klasse `Factorial`

bietet eine statische Methode zum Berechnen der Fakultät. Die Klasse `FactorialTest` enthält Unit Tests für diese Methode.

`FactorialTest` soll die folgenden drei Tests enthalten:

- Test, ob der Versuch, die Fakultät einer negativen Zahl zu berechnen, korrekt zu einer `IllegalArgumentException` führt.
- Test, ob die Fakultät von 0 korrekt als 1 berechnet wird.
- Test, ob Folgendes gilt:
  `Factorial.calculateFactorial(n - 1) * n == Factorial.calculateFactorial(n)`

Die ersten beiden dieser Tests sind bereits implementiert. Ergänzen Sie den letzten Test und lassen Sie die Tests anschließend durchlaufen. Alle drei Tests sollten erfolgreich sein.

c) ★★ Diskutieren Sie folgende Fragen mit Ihren Kolleginnen und Kollegen:

- Was unterscheidet Exceptions von anderen Möglichkeiten zur Fehlerbehandlung (spezielle Rückgabewerte, globale Fehler-Variable, ...)? Welche Vorteile haben Exceptions?
- Was ist der Unterschied zwischen Exceptions und Errors?
- Was ist der Unterschied zwischen checked Exceptions und unchecked Exceptions?

d) ★★★ Diskutieren Sie folgende Fragen mit Ihren Kolleginnen und Kollegen:

- Was ist ein Stacktrace? In welchem Zusammenhang steht dieser mit der Behandlung von Exceptions?
- Was versteht man unter Exception Chaining?
- Warum kompiliert der folgende Code nicht?

```
1   class BaseClass {
2       public int calculate() {
3           // ... snip ...
4       }
5   }
6
7   class DerivedClass extends BaseClass {
8       public int calculate() throws FileNotFoundException {
9           // ... snip ...
10      }
11  }
```

Übungsteil (selbstständig zu lösen)

Aufgabe 1 (Testen allgemein) [2 Punkte]

Der Softwareentwickler Toni Testardo hat einen Job bei einem Online-Versandhaus ergattert. Obwohl man es aufgrund seines Namens (bzw. eines Teils davon) nicht vermuten würde, hält er überhaupt nichts von `Unit-Tests`. Sein Vorgesetzter sieht das zum Glück ein wenig anders und verdon-

nert Toni[1] deshalb, beim aktuellen Auftrag eines Online-Versandhändlers die `Unit-Tests` vor der Implementierung zu entwerfen.
Da Toni leider ein ziemlicher Sturkopf ist[2] – und er das Thema 'Testen' an der Uni nicht mit dem notwendigen Interesse verfolgt hat, beschließt er dennoch, zuerst mit der Implementierung zu beginnen. Als sein Vorgesetzter dies allerdings bemerkt, ist Schluss mit Lustig. Toni wird mit Kündigung gedroht, falls er sich nicht am Ende des Tages bei seinem Vorgesetzten mit den `Unit-Tests` meldet.
Spätestens jetzt bereut Toni, bei den Vorlesungen zum Thema Softwaretesten nicht besser aufgepasst zu haben. Er beginnt deshalb zu recherchieren, was sein Vorgesetzter da von ihm verlangt.
Helfen Sie Toni seine Deadline einzuhalten, indem Sie folgende Fragen beantworten:

a) *0.5 Punkte* Warum testet man bzw. wieso ist das Testen von großer Bedeutung?

b) *0.5 Punkte* Wie nennt man diese Art der Softwareentwicklung, bei welcher Tests vor der eigentlichen Implementierung geschrieben werden?

c) *0.5 Punkte* Welche Vorteile erhofft sich Toni's Vorgesetzter davon?

d) *0.5 Punkte* Welche Kritikpunkte an dieser Entwicklungsweise könnte Toni anbringen, um zumindest nicht komplett das Gesicht zu verlieren?

Aufgabe 2 (Onlineversand – Zuerst die Tests) [2 Punkte]

Dank Ihres heldenhaften Einsatzes schöpft Toni wieder neuen Mut. Jetzt müssen aber `Unit-Tests` her, damit er seinen aktuellen Job vorerst behalten kann. Als ersten Schritt sucht sich Toni die Anforderungsspezifikation[3] des Onlineversandhandels heraus und beginnt sich zu überlegen, welche Tests denn sinnvoll wären.
Helfen Sie Toni dabei, indem Sie ihm über die Schulter schauen und sich überlegen, welche `Unit-Tests` für die nachfolgenden Anforderungen notwendig sein könnten.

Hinweis ⚠

Jede dieser Anforderungen lässt sich jeweils mit einer Methode realisieren. Je nach übergebenen Parametern bzw. äußeren Umständen verhält sich die Methode aber anders und liefert gegebenenfalls ein unterschiedliches Resultat. Definieren Sie gegebenenfalls pro Methode mehrere `Unit-Tests`, um jeden dieser Fälle abzudecken.
Sie müssen die Tests bei dieser Aufgabe noch nicht implementieren, sondern lediglich beschreiben, welche verschiedenen Fälle bei der Abarbeitung der entsprechenden Methoden auftreten können. Sehen Sie dazu die nachfolgenden Beispiele (Bekannt aus Blatt 2).

Beispiel 1:
Ein Student soll zu einer Proseminar-Gruppe hinzugefügt werden.
Unterschiedliche Fälle:

1. In der Gruppe ist noch Platz
2. Die Gruppe ist bereits voll

[1] Aus Toni's Sicht betrachtet natürlich.
[2] Seinem Namen alle Ehre machend.
[3] Salopp gesagt: Ein Schriftstück, wo beschrieben wird, was die zu entwickelnde Software können muss (mehr dazu im 4. Semester in einem eigenen Kurs).

Beispiel 2:
Eine Proseminar-Gruppe soll erstellt werden, welche eine eindeutige ID sowie einen Proseminarleiter benötigt.
Unterschiedliche Fälle:

1. Die erstellte Gruppe ist gültig
2. Die Gruppen-ID ist nicht eindeutig
3. Der Gruppe wurde kein Proseminarleiter zugewiesen

Hinweis ⚠

Beim Beispiel 2 schließen sich die Fälle 2 und 3 nicht gegenseitig aus, d.h. es gäbe auch den Fall, wo die Gruppen-ID nicht eindeutig ist **und** der Gruppe kein Proseminarleiter zugewiesen wurde.
Für die folgenden Aufgaben reicht es aus, wenn Sie alle 'einfachen' Fälle nennen (Wie in Beispiel 2 gezeigt). Sie müssen NICHT alle möglichen Fall-Kombinationen auflisten.

Beschreiben Sie nun analog zu den obrigen Beispielen bei jeder Anforderung die unterschiedlichen Fälle, welche auftreten könnten:

a) *0.5 Punkte* Es sollen Produkte zu einem Warenbestand (unendlich groß) hinzugefügt werden können. Die hinzugefügten Produkte müssen dabei gültige Werte besitzen (Name und Beschreibung dürfen nicht leer bzw. `null` sein, der Preis muss >0 sein). Jedes Produkt besitzt eine eindeutige ID (Ein `Integer` reicht aus). Im Produktkatalog darf eine ID nur einmal vergeben werden!

b) *0.5 Punkte* Es muss möglich sein, Produkte zu einem Einkaufswagen hinzuzufügen. Das gleiche Produkt kann auch mehrmals zum Einkaufswagen hinzugefügt werden.

Hinweis ⚠

Es soll außerdem eine Abfrage unterstützt werden, die zurückgibt, wie viele Produkte sich insgesamt im Einkaufswagen befinden (z.B.: 10 Laptops, 2 Fernseher also 12 Produkte insgesamt).

c) *0.5 Punkte* Der Gesamtpreis aller Produkte im Einkaufswagen soll berechnet werden. Wenn von einem Produkt z.B.: 3 Stück gekauft werden, dann soll die Menge klarerweise in der Preisberechnung beachtet werden.

d) *0.5 Punkte* Bei jedem bestätigten Einkauf wird die Einkaufssumme vom Guthaben des Kunden abgebucht. Der Kunde kann nur dann einkaufen, wenn die Einkaufssumme von seinem Guthaben abgedeckt wird. Außerdem soll gleichzeitig die Menge der gekauften Produkte vom noch vorhandenen Warenbestand abgezogen werden. Auch hier kann der Kauf nur dann abgewickelt werden, wenn noch genügend Produkte im Warenbestand vorhanden sind.

Aufgabe 3 (Onlineversand – Testimplementierung) [3 Punkte]

Toni ist erleichtert, da ein großer Teil der Arbeit bereits erledigt ist. Allerdings muss er jetzt die Testbeschreibungen in ausführbare `Unit-Tests` umformen. Zum Glück fällt ihm wieder ein, dass

er ja schon mal mit `TestNG` gearbeitet hat. Im nächsten Schritt gilt es also, die zuvor konzipierten `Unit-Tests` zu implementieren. Toni juckt es bereits in den Fingern, die Anforderungen direkt umzusetzen, doch da ihm sein Vorgesetzter einen kritischen, aber durchaus drohenden, Blick zuwirft, entscheidet sich Toni in seinem eigenen Interesse, sein Glück nicht überzustrapazieren – und wirklich zuallererst mit der Implementierung der Tests zu beginnen[4].
Doch wie ging das noch gleich mit `TestNG`? Richtig erkannt: Unser lieber Toni steht erneut auf der Leitung. Und wieder liegt es an Ihnen, die Situation zu retten. Beginnen Sie also damit, die `Unit-Tests` mit `TestNG` zu implementieren. Halten Sie sich dabei strikt an die beschriebene Vorgehensweise bzw. Reihenfolge:

a) *0.5 Punkte* Erstellen Sie die folgenden Klassen und generieren Sie die benötigten `getter`- und `setter` Methoden:

- `User.java`: Ein eindeutig identifizierbarer `user`, welchem ein `shopping cart` zugeordnet ist.
- `ShoppingCart.java`: Ein `shopping cart`, zu welchem `products` (inkl. die Menge der Produkte) hinzugefügt werden können.
- `Product.java`: Ein eindeutig identifizierbares `product`.
- `Stock.java`: Simuliert den Warenbestand. Hier sind alle `products` sowie ihre Menge enthalten.
- `OrderManager.java`: Wickelt alle 'Transaktionen' ab. D.h. über diese Klasse wird der gesamte Online-Versandhandel gesteuert.

Hinweis ⚠

Beschränken Sie sich nicht ausschließlich auf die vorgegebenen Klassen. Sollten weitere Klassen sinnvoll erscheinen bzw. verhindern, dass Sie redundanten Code schreiben müssen, dann zögern Sie nicht, diese hinzuzufügen.

b) *0.5 Punkte* Erstellen Sie für jene Methoden, welche die beschriebenen Anforderungen umsetzen, sogenannte `stubs` (Methoden ohne Logik und mit hardcodierten Rückgabewerten; siehe Beispiel). Somit kann die Methode im Test bereits aufgerufen werden (d.h. der `Unit-Test` kann schon implementiert und ausgeführt werden, da Ihr Programm kompilierbar ist). Ein `Stub` für eine Methode sieht folgendermaßen aus:

```
1  public int addNumbers(int number1, int number2){
2   /* TODO: implement */
3   return 0;
4  }
```

Erstellen Sie für die notwendigen Methoden **nur** die `stubs`. Wichtig ist an dieser Stelle, dass Ihr Programm kompiliert und Sie in der nachfolgenden Teilaufgabe die Methoden bereits aufrufen können, um `Unit-Tests` dafür zu schreiben. Die Tests sollen bei dieser Vorgehensweise **vor** der eigentlichen Implementierung der Methoden umgesetzt werden (d.h. **alle** Tests sollen nach diesem Schritt ausführbar sein, während Ihre Methoden noch keine Logik beinhalten).

[4] Wir haben es bereits 15:00 Uhr und sein Vorgesetzter muss das Büro bereits um 16:00 Uhr anstatt um 17:00 Uhr verlassen, da er heute seinen 10. Hochzeitstag feiert und noch ein Geschenk für seine Frau kaufen muss. Dass er dass nicht schon früher erledigt hat – und ihm bis jetzt noch nichts Passendes eingefallen ist, schlägt ihm auch gehörig auf die Stimmung.

Hinweis ⚠

Der Prozess des Einkaufens (Siehe Aufgabe 2d) sollte in einer eigenen Klasse gemanaged werden.

c) *2 Punkte* Erstellen Sie nun die `Unit-Tests` mit `TestNG`. Da Sie in der vorhergehenden Aufgabe sich bereits Gedanken darüber gemacht haben, was Sie testen möchten, sollte dieser Teil der Aufgabe leicht von der Hand gehen. Schauen Sie sich auch die Dokumentation von `TestNG` an und suchen sich die passenden `assert`-Methoden heraus.

Hinweis ⚠

Sie sollten Ihre `Unit-Tests` in mehrere Klassen auslagern (gemäß Funktionalität) und nicht alle Tests in eine einzige Klasse packen! Trotzdem sollen alle `Unit-Tests` mit einem Befehl (Mausklick) ausführbar sein. Erstellen Sie deshalb eine eigene TestSuite.

Hinweis ⚠

Beim erstmaligen Ausführen der Tests werden Sie feststellen, dass fast alle Tests fehlschlagen. Nur keine Bange, das muss so sein.

Aufgabe 4 (Onlineversand – Die Umsetzung) [4 Punkte]

Gerade noch rechzeitig (oder eben pünktlich wie ein Schweizer Uhrwerk, je nach Betrachtungsweise) reicht Toni seine `Unit-Tests` um 15:59 Uhr ein. Sein Vorgesetzter nimmt die Tests mit mürrischem Blick, aber doch mit einem anerkennenden Nicken, zur Kenntnis und verlässt dann das Büro.
Toni konnte die Deadline einhalten, die Gefahr ist also (vorerst) gebannt. Jedoch hat er heute keine gute Figur gemacht und möchte deshalb bei seinem Vorgesetzten Pluspunkte sammeln. Aus diesem Grund beschließt er, eine möglichst gute Implementierung der definierten Methoden abzuliefern. Worauf warten Sie noch? Machen Sie sich an die Arbeit!

a) *0.5 Punkte* Statten Sie die erstellten Methoden-`Stubs` nun mit Funktionalität aus (d.h. implementieren Sie sie). Führen Sie dabei in regelmäßigen Abständen Ihre `Unit-Tests` aus.

Hinweis ⚠

Im Idealfall müssen Sie den Testcode, welchen Sie in der vorherigen Aufgabe geschrieben haben, nicht mehr ändern.

Hinweis ⚠

Nach jeder korrekten Implementierung und erneutem Ausführen der Tests schalten immer mehr Tests auf grün. Ein gutes Gefühl, oder?

Hinweis ⚠

Da Sie sich jetzt bereits mit `Collections` auskennen, sollten Sie diese auch bei dieser Hausübung einsetzen.

b) *2 Punkte* Beim Recherchieren und durchstöbern seiner alten Folien stößt Toni auf das Thema `Exceptions`. Damit könnte man doch auch die Implementierung gehörig verbessern, oder? Er möchte also an folgenden Stellen `Exceptions` einbauen:

- Wenn beim Hinzufügen eines Produktes zum Warenbestand ungültige Werte übergeben wurden.
- Wenn beim Hinzufügen eines Produktes die ProduktID bereits vergeben wurde.
- Wenn ein Kunde z.B.: 10 Laptops zu seinem Einkaufswagen hinzufügen möchte, aber nur mehr 3 auf Lager sind.
- Wenn die Summe aller Produkte im Einkaufswagen größer als das Guthaben des Kunden ist.
- Wenn ein Kunde mit einem leeren Einkaufswagen bezahlen möchte.

Statten Sie Ihre Implementierung also mit `Exception-Handling` aus. Überlegen Sie sich nun, unter welchen Umständen welche `Exceptions` bei welchen Methoden Sinn machen, wo man sie abfängt und wie man gegebenenfalls darauf reagieren kann. Realisieren Sie auch eigene `Exceptions`.

Halten Sie sich dabei an folgende Vorgaben:

- Egal welche Operation Sie durchführen, es soll immer (auch im Fehlerfall) eine Statusmeldung (Ausgabe) erfolgen (z.B.: 'Hinzufügen des Produktes xy erfolgreich/nicht erfolgreich abgeschlossen').
- Jede auftretende `Exception` soll protokolliert werden (Möglicherweise in einer `Collection` gesammelt?).
- Wenn z.B.: 20 Produkte in einer Schleife dem Warenbestand hinzugefügt werden sollen, ein Produkt aber fehlerhaft ist, dann sollen am Ende der Operation trotzdem 19 Produkte hinzugefügt worden sein (und ein Eintrag im Error-Log vorhanden sein).

c) *0.5 Punkte* Erweitern Sie nun Ihre `Unit-Tests` bwz. ändern Sie den bestehenden Testcode ab indem Sie auch sogenannte 'Negativ-Tests' hinzufügen. Sehen Sie sich dazu folgendes Beispiel an und recherchieren Sie gegebenenfalls in den Vorlesungsfolien bzw. im Internet:

```
1  @Test(expectedExceptions = ArithmeticException.class)
2  public void testSomething(){
3   /*
4    * Der Test ist erfolgreich, wenn eine ArithmeticException auftritt.
5    * Kommt es zu keiner Exception, scheitert der Test.
6    */
7  }
```

Wichtig: Laden Sie bitte Ihre Lösung (.txt, .java oder .pdf) in OLAT hoch und geben Sie mittels der Ankreuzliste auch unbedingt an, welche Aufgaben Sie gelöst haben. Die Deadline dafür läuft am Vortag des Proseminars um 23:59 (Mitternacht) ab.