

SE2, Aufgabenblatt 5

Modul: Softwareentwicklung II – Sommersemester 2017

Exceptions behandeln, Modularisierung

Moodle-URL moodle.informatik.uni-hamburg.de
Projektraum Softwareentwicklung 2 SoSe2017
Ausgabedatum 4. Mai 2017

Kernbegriffe

In Softwaresystemen können zur Laufzeit *Fehlerzustände* auftreten. Deren Ursachen unterscheiden wir (orthogonal zu bisherigen Begriffen) aus Sicht des Entwicklungsteams in *Entwicklungsfehler* (vom Team verursacht) und *Umgebungsfehler* (in der Laufzeitumgebung des Produktsystems). Eine Kategorie von Entwicklungsfehlern sind *Verletzungen von Zusicherungen*. Entweder hat ein Klient seine Vorbedingungen nicht eingehalten oder ein Dienstleister hat seine Nachbedingungen verletzt. Weitere Entwicklungsfehler sind beispielsweise ein Methodenaufruf über eine `null`-Referenz, oder eine ganzzahlige Division durch 0. Umgebungsfehler können auftreten, wenn die Systemumgebung sich unerwünscht verhält bzw. ungeeignet konfiguriert wurde. Ein Programm, das z.B. Daten über das Internet verschicken soll, muss auf einem Rechner laufen, der online ist. Ein Programm, das Informationen in eine Datei speichern soll, muss Schreibrechte für die Datei besitzen. Dies sind mögliche Fehlerquellen, die im Programm einkalkuliert und mit denen zur Laufzeit umgegangen werden sollte, denn sie liegen außerhalb des unmittelbaren Einflussbereichs der Programmierer. Bei einem Programm mit direkter Benutzerinteraktion kann im Fall eines Umgebungsfehlers beispielsweise eine Meldung erfolgen, dass der Rechner nicht online ist, oder bei fehlenden Schreibrechten alternativ ein Speichern unter *Eigene Dateien* vorgeschlagen werden.

Java (wie andere objektorientierte Sprachen auch) bietet mit den *Exceptions* ein Sprachmerkmal, um Fehlerzustände zu melden. *Eine Exception ist ein Objekt einer Exceptionklasse*. In Fehlerfällen kann ein Exemplar einer solchen Klasse erzeugt werden und z.B. an den aufrufenden Klienten weitergegeben werden. Man sagt, eine Exception wird vom Dienstleister *geworfen* (engl. *throw an exception*) und kann vom Klienten *gefangen* (engl. *catch an exception*) werden. Hierbei wird der normale Programmablauf unterbrochen. Mit Javas *geprüften Exceptions* (engl. *checked exceptions*) ist es möglich, den Klienten zur Fehlerbehandlung zu „zwingen“. Wenn ein Klient eine geprüfte Exception nicht selbst behandelt oder explizit weiterreicht (mittels `throws` im Methodenkopf), so führt dies zu einem Übersetzungsfehler. Sie werden deshalb vor allem für das Melden von Umgebungsfehlern eingesetzt. *Ungeprüfte Exceptions* (engl. *unchecked exceptions*) hingegen zwingen den Klienten *nicht* dazu, eine Fehlerbehandlung in seinem Quelltext vorzusehen. Sie sind eher geeignet für das Melden von Entwicklungsfehlern, da diese zur Laufzeit nicht sinnvoll behandelt werden können (sondern aus dem Quelltext entfernt werden sollten). Beispiele für ungeprüfte Exceptions sind `NullPointerException` und `AssertionError`.

Module sind klassisch statische Einheiten des Programmtextes mit einer Schnittstelle und einer (gekap-selten) Implementation. Ein Modul deklariert über seine *Import-Schnittstelle* (auch outgoing interface) die Einheiten eines Softwaresystems, die es zur Erfüllung seiner Aufgabe benötigt. Seine eigenen Dienstleistungen bietet es über seine *Export-Schnittstelle* (auch incoming interface) an. Wenn Module in Modulen geschachtelt sein können, dann beeinflusst dies die *Benennung* von Modulen und den *Zugriffsschutz* auf geschachtelte Einheiten. In Java sind die Klassen lediglich modulähnlich: sie können ineinander geschachtelt werden, ihre Implementation verstecken und haben eine implizite Export-Schnittstelle, aber keine explizit deklarierte Import-Schnittstelle. Die *Pakete* (engl.: packages) in Java sind ebenfalls nur modulähnlich: Ihre Schachtelung beeinflusst lediglich ihre Benennung, nicht aber ihren Zugriffsschutz.

Aufgabe 5.1 Exceptions behandeln

Anhand eines Beispiels nähern wir uns nun dem umfangreichen Thema der Exceptions. Die Mediathek soll nun alle Verleih- und Rücknahmevorgänge in einer Datei vermerken. Beim Einlesen und Schreiben von Daten in Dateien spielen geprüfte (checked) Exceptions eine wichtige Rolle.

- 5.1.1 Holt für diese und die folgende Aufgabe eure bisherigen Klassendiagramme hervor (oder zeichnet sie neu) und erweitert sie nach jedem Aufgabenteil, um zu verstehen, welche Klassen welche Aufgaben übernehmen sollen.

- 5.1.2 Jeder Verleihvorgang (Ausleihe und Rückgabe) soll zukünftig festgehalten werden. Erstellt dazu eine Klasse `VerleihProtokollierer`. Diese Klasse soll eine Klassenmethode zur Verfügung stellen, um Verleihkarten zu speichern:

```
public static void protokolliere(String ereignis, Verleihkarte verleihkarte)
```

Überlegt, welche Werte für den Parameter `ereignis` sinnvoll sind und schreibt einen passenden Schnittstellenkommentar. Anfangs soll der Protokollierer die Daten auf die Konsole ausgeben.

- 5.1.3 Sorgt dafür, dass die Klasse `VerleihServiceImpl` ihre Verleihvorgänge mit Hilfe des `VerleihProtokollierers` protokolliert. Testet, ob das Protokollieren funktioniert. Startet das Programm und führt beispielhaft ein paar Ausleih –und Rückgabevorgänge aus.
- 5.1.4 Nun soll die Ausgabe der Verleihvorgänge nicht mehr auf die Konsole erfolgen, sondern in eine Datei. Java stellt im Paket `java.io` eine umfangreiche Bibliothek für den Umgang mit Dateien und weiteren Ein- und Ausgabeschnittstellen zur Verfügung. Wir beschränken uns auf die Möglichkeit, Texte in Dateien zu schreiben und benutzen dazu die Klasse `java.io.PrintWriter`. Für unsere Zwecke relevante Konstruktoren und Operationen dieser Klasse sind:

```
PrintWriter(String fileName, boolean append)
write(String text)
close()
```

Der Parameter `fileName` im Konstruktor kann z.B. „./protokoll.txt“ sein. Der Parameter `append` gibt an, ob beim Öffnen einer nicht leeren Datei ein neuer Text angehängt wird oder ob der alte Text überschrieben wird.

Die Operation `close()` sorgt dafür, dass andere Programme wieder in die Datei schreiben können, sobald wir sie nicht mehr benutzen. Jedes Programm, das eine Datei öffnet, sollte diese nach Benutzung mit `close()` wieder freigeben.

Ändert euren `VerleihProtokollierer`, so dass er nicht mehr auf die Konsole schreibt, sondern mithilfe des `PrintWriters` in eine Datei. Ignoriert dabei erst einmal den folgenden Fehler bei der Übersetzung (aber nur diesen): `Unhandled exception type IOException`.

Anmerkung: Auch Klassen, die mit der Systemumgebung interagieren, also z.B. in Dateien schreiben, lassen sich mit JUnit testen! Dies ist allerdings ein umfangreiches Gebiet, das wir momentan nicht weiter thematisieren können.

- 5.1.5 Schaut euch die Dokumentation der Methode `PrintWriter.write(String)` in der Java-API unter <http://docs.oracle.com/javase/7/docs/api> an (schaut dafür in die Klasse `Writer`, von dieser Klasse wird die Methode geerbt). Hier seht ihr, dass hinter dem Methodenkopf folgendes steht: **throws** `IOException`. Dies bedeutet, dass die Methode `write` im Fehlerfall eine `IOException` wirft, die vom Klienten behandelt werden muss. Dafür verwendet der Klient in Java einen try-catch-Block. In den try-Block kommt der Quelltext, in dem eine Exception auftreten kann:

```
try
{
    ...
    writer.write("Beispiel-Text");
}
catch(IOException e)
{
    // Fehlerbehandlung
}
```

Wenn in dem try-Block eine Exception auftritt, so wird der Programmablauf dort unterbrochen und in dem catch-Block fortgesetzt. Wenn keine Exception auftritt, dann wird der try-Block zu Ende ausgeführt und der catch-Block wird nicht durchlaufen. Normalerweise wird in beiden Fällen danach das Programm hinter dem catch-Block fortgesetzt. *Beispiele für Ausnahmen:* Im catch-Block wird eine weitere Exception geworfen oder im try-Block steht eine return-Anweisung.

- 5.1.6 Sorgt dafür, dass die Klasse `VerleihProtokollierer` eine auftretende `IOException` auffängt und den Fehler mit `System.err.println(String)` auf die Konsole ausgibt. Prüft eure Implementation, indem ihr das Programm startet und Medien verleiht. Wenn der Pfad für die Datei nicht existiert oder die Datei schreibgeschützt ist, tritt ein Fehler auf. Probiert dies aus, indem ihr die Datei von Hand als schreibgeschützt markiert. Dafür könnt ihr den Dialog mit den Dateiei-

enschaften verwenden. Euch fällt sicherlich auf, dass die Lösung für den Benutzer des Programms unbefriedigend ist. Warum ist das so, was könnte man besser machen?

- 5.1.7 Ergänzt nun euer Klassendiagramm um die Klassen `AusleihWerkzeug`, `RueckgabeWerkzeug`, `VerleihServiceImpl`, `VerleihProtokollierer` und `IOException`.
- 5.1.8 **Zusatzaufgabe:** Erstelle für den ersten Parameter der Methode `protokolliere` einen Enum-Typ `VerleihEreignis`. Schau Dir hierzu die Dokumentation zu Enum-Typen an: <http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html>.

```
public static void protokolliere(VerleihEreignis ereignis, Verleihkarte verleihkarte)
```

Aufgabe 5.2 Zuständigkeit für Fehlerbehandlungen festlegen

In Aufgabe 5.1 wurde der Fehler im Dienstleister behandelt: Der `VerleihProtokollierer` hat die Fehlermeldung direkt auf die Konsole ausgegeben. Dies ist offensichtlich keine gute Lösung: Der Benutzer kann die Meldung leicht übersehen, denn das System arbeitet ansonsten mit einer grafischen Benutzungsschnittstelle. Es wäre aber auch keine gute Lösung, wenn der `VerleihProtokollierer` ein GUI-Fenster öffnen würde, denn damit würden wir in den `VerleihProtokollierer` Kenntnisse über die gewählte Form der Benutzerinteraktion einbetten.

Besser wäre es, wenn nicht der Dienstleister (also der `VerleihProtokollierer`), sondern der Klient den Fehler behandelt. Dann wüsste der `VerleihProtokollierer` nichts über die Benutzerinteraktion. Dafür wäre es ideal, wenn der `VerleihService` eine Exception vom `VerleihProtokollierer` bekommt und an seinen Klienten (die GUI) weitergibt, der dem Benutzer einen Fehlerdialog öffnet.

- 5.2.1 Aus diesen Gründen wollen wir im `VerleihProtokollierer` die `IOException` nun nicht mehr selber behandeln, sondern stattdessen eine Exception an den Klienten weiterreichen. Dafür soll der Protokollierer eine eigene, fachliche Exception verwenden. Mit dieser Exception macht er an seiner Schnittstelle deutlich, dass eine Protokollierung fehlgeschlagen ist. Dazu existiert im Projekt bereits die Exception-Klasse `ProtokollierException`.

Mithilfe des Schlüsselwortes `throws` kann im Kopf einer Operation oder Methode deklariert werden, dass sie potentiell eine Exception wirft. Gebt für die Methode `protokolliere` an, dass sie eine `ProtokollierException` werfen kann. Ändert nun die Methode `protokolliere`: Werft im catch-Block eine neue `ProtokollierException`, anstatt den Fehler auf die Konsole zu schreiben. Dazu müsst ihr das Schlüsselwort `throw` verwenden.

- 5.2.2 Die Exception soll geeignet von der GUI angezeigt werden. Der `VerleihService` soll die Behandlung der Exception deshalb an seine Klienten (`AusleiheWerkzeug` und `RuecknahmeWerkzeug`) delegieren. Gebt bei den entsprechenden Operationen an, dass potentiell eine `ProtokollierException` geworfen wird.

- 5.2.3 An mehreren Stellen im Projekt zeigt euch Eclipse an, dass das Kompilieren nun fehlschlägt. Geht zuerst zur Klasse `AusleihWerkzeug`. Behebt hier den Fehler durch das Einbauen einer try-Anweisung. Im catch-Block könnt ihr einen Message-Dialog verwenden, um den Benutzer zu informieren. Nutzt hierzu auch die Fehlermeldung der Exception. Verfährt sinngemäß mit den anderen Fehlerstellen.

```
JOptionPane.showMessageDialog(null, <Sinnvolle Fehlerbeschreibung>,  
    "Fehlermeldung", JOptionPane.ERROR_MESSAGE);
```

- 5.2.4 Nun muss in der Methode `protokolliere` im `VerleihProtokollierer` noch eingebaut werden, dass der `FileWriter` immer geschlossen wird, auch im Falle einer Exception.

Da `FileWriter` das Interface `java.lang.AutoCloseable` implementiert, können wir aus der try-Anweisung eine try-with-resources-Anweisung machen, die dafür sorgt, dass eine angeforderte Ressource nach Ausführung der Anweisungen innerhalb des try-Blocks in jedem Fall geschlossen wird, egal, ob die Ausführung erfolgreich war oder eine Exception geworfen wurde.

Löscht die close-Anweisung in eurem try-Block und verschiebt die Erzeugung des `FileWriters` eingeschlossen in runden Klammern hinter das try-Schlüsselwort. Dadurch wird der erzeugte `FileWriter` nach Ausführung des try-catch-Blocks in jedem Fall automatisch geschlossen.


```
try (FileWriter writer = new FileWriter( ... ))
```

5.2.5 Ergänzt nun Euer Klassendiagramm um die Klassen `ProtokollierException`, `Exception`, und `JOptionPane`.

Aufgabe 5.3 Pakete in Eclipse

Java bietet die Möglichkeit, Software zu modularisieren, indem Klassen auf Pakete (engl: packages) verteilt werden. Die Mediathek ist noch nicht in Pakete aufgeteilt. Dies sollt ihr nun machen. Eclipse unterstützt uns bei solchen Refactorings. (Refactorings sind Änderungen an der Struktur des Quelltextes, die für den Benutzer der laufenden Software nicht sichtbar werden).

5.3.1 Als erstes sollt ihr alle Klassen und Interfaces (also alle Typen) aus dem momentan namenlosen Paket in ein neues Paket bewegen. Öffnet dafür auf dem `src`-Verzeichnis im Package-Explorer das Kontextmenü. Selektiert *New->Package* und erzeugt ein Paket namens *mediathek*. Nun könnt ihr alle Typen in das neue Paket verschieben, indem ihr alle selektiert und dann im Kontextmenü *Refactor->Move...* auswählt.

 5.3.2 Nun wollen wir alle Medien-Typen (und die zugehörigen Testklassen) in ein eigenes Unterpaket *mediathek.medien* verschieben. Falls ihr euch nicht sicher seid, welche Typen das betrifft, klickt mit rechts auf `Medium.java` und wählt dann *Open Type Hierarchy*. **Denkt an die Testklassen!** Welcher Fehler tritt jetzt auf? **Schreibt einen kurzen Text, der die Situation erläutert.** Behebt das Problem anschließend.

5.3.3 Verschiebt die beiden Klassen `Kunde` und `Verleihkarte` zusammen mit ihren Testklassen (also insgesamt 4 Klassen) in ein Unterpaket *mediathek.materialien*.

5.3.4 Verschiebt die Klassen `Datum`, `Geldbetrag`, `Kundennummer` und `PLZ` zusammen mit ihren Testklassen (also insgesamt 8 Klassen) in ein Unterpaket *mediathek.fachwerte*.

5.3.5 Verschiebt alle Typen, in deren Namen mindestens eines der Wörter `Einleser`, `Exception`, `Protokollierer` oder `Service` auftaucht, in ein Unterpaket *mediathek.services*.

5.3.6 Verschiebt alle Typen, in deren Namen mindestens eines der Wörter `Werkzeug`, `UI`, `Model`, `Formatierer` oder `Comparator` auftaucht, in ein Unterpaket *mediathek.werkzeuge*.

5.3.7 Jetzt sollten nur noch zwei Klassen übrig sein, `StartupMediathek` und `AssertTest`. Diese beiden Klassen könnt ihr in ein Unterpaket *mediathek.startup* verschieben.