SE2, Aufgabenblatt 5

Modul: Softwareentwicklung II – Sommersemester 2012

Exceptions behandeln, Modularisierung

CommSy-Projektraum ..... SE-2 CommSy SoSe 2012
Ausgabedatum ..... 03. Mai 2012

Kernbegriffe

In Softwaresystemen können zur Laufzeit *Fehlerzustände* auftreten. Als Ursachen für Fehlerzustände unterscheiden wir grundsätzlich *Programmierfehler* und *Umgebungsfehler*. Eine Kategorie für Programmierfehler sind *Verletzungen von Zusicherungen*. Entweder hat der Klient die Vorbedingungen vor einem Operationsaufruf nicht eingehalten, oder der Dienstleister hat seine Nachbedingungen verletzt. Weitere Programmierfehler sind beispielsweise ein Aufruf über eine Referenz, deren Wert `null` ist, oder eine ganzzahlige Division durch Null. 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. Hier ergeben sich mögliche Fehlerfälle, die im Programm einkalkuliert werden sollten und mit denen zur Laufzeit umgegangen werden sollte (im Gegensatz zu Programmierfehlern), denn sie liegen außerhalb des unmittelbaren Einflussbereichs des Programmierers (deshalb Umgebungsfehler). Bei einem Programm mit direkter Benutzerinteraktion kann im Fall eines Umgebungsfehlers beispielsweise eine Meldung ausgegeben werden, dass der Rechner nicht online ist, oder bei fehlenden Schreibrechten für eine Datei 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 *geprüften* Exceptions (engl. *checked exceptions*) ist es möglich, den Klienten zur Fehlerbehandlung zu „zwingen“, denn wenn der Klient den Fehler nicht in seinem Quelltext behandelt, so führt das 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 also eher geeignet für das Melden von Programmierfehlern, da diese zur Laufzeit nicht sinnvoll behandelt werden können (sondern bei der Software-Entwicklung korrigiert werden sollten). Ungeprüfte Exceptions werden z.B. bei fehlgeschlagenen `assert`-Anweisungen geworfen, weitere Beispiele sind Exemplare der Klassen `NullPointerException` oder `ArithmeticException`. Da Exceptions den normalen Programmablauf unterbrechen und damit die Komplexität der Software erhöhen, sollten sie äußerst sparsam eingesetzt werden. Die Implementierung des Vertragsmodells sowie Fehlerzustände durch Umgebungseinflüsse sind Beispiele, bei denen Exceptions sinnvoll eingesetzt werden können. Exceptions sollten hingegen nicht als komplexe Möglichkeit zur Steuerung des normalen Kontrollflusses benutzt werden.

Module sind klassisch statische Einheiten des Programmtextes mit einer Schnittstelle und einer (gekapselten) Implementation. Ein Modul deklariert über seine *Import-Schnittstelle* (engl. auch outgoing interface) die Einheiten eines Softwaresystems, die es zur Erfüllung seiner Aufgabe benötigt. Seine eigenen Dienstleistungen bietet es über seine *Export-Schnittstelle* (engl. 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 habe 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. Ein Paket in Java hat eine implizite Export-Schnittstelle (seine als public deklarierte Klassen und Interfaces), deklariert aber ebenfalls keine explizite Import-Schnittstelle.

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 Für diese und die folgende Aufgabe ist es sinnvoll, eure bisherigen **Klassendiagramme hervorzuholen (oder neu zu zeichnen) und zu erweitern**, 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 Operation zur Verfügung stellen, um Verleihkarten zu speichern:

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

Überlegt, welche Werte für den Parameter `ereignis` sinnvoll sind und schreibt einen Schnittstellenkommentar. Im ersten Schritt 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.

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.FileWriter`. Für unsere Zwecke relevante Konstruktoren und Operationen dieser Klasse sind:

```
FileWriter(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 `FileWriter`s 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 `FileWriter.write(String)` in der Java-API unter *http://java.sun.com/javase/6/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 Fäl-

len 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 das aus, indem ihr die Datei von Hand als schreibgeschützt markiert. Dafür könnt ihr den Dialog mit den Dateieigenschaften verwenden. Euch fällt sicherlich auf, dass die Lösung für den Benutzer des Programms unbefriedigend ist. Warum ist das so, was könnte man besser machen?

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 einfach an seinen Klienten (die GUI) weitergibt. Die GUI muss dann einen Fehlerdialog für den Benutzer öffnen.

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 Exception deshalb an seine Klienten (`AusleiheWerkzeug` und `RuecknahmeWerkzeug`) weiter delegieren. Gebt dafür bei den entsprechenden Operationen und Methoden an, dass potentiell eine `ProtokollierException` geworfen wird. Eine Exception dieses Typs kann dadurch einfach durchgereicht werden.

5.2.3 An mehreren Stellen im Projekt zeigt euch Eclipse an, dass das Compilieren nun fehlschlägt.! Geht zuerst zur Klasse `AusleiheWerkzeug`. Behebt hier den Fehler durch das Einbauen eines try-catch-Blockes. Im catch-Block könnt ihr einen MessageDialog verwenden, um den Benutzer zu informieren. Verfahrt 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. Dafür verwenden wir einen finally-Block. Ein finally-Block wird immer ausgeführt, unabhängig davon, ob vorher eine Exception aufgetreten ist oder nicht. Der finally-Block steht direkt hinter dem catch-Block. Löscht die close-Anweisung in eurem try-Block und ergänzt den folgenden finally-Block (mit passendem Variablennamen und passender Fehlermeldung). Da auch in der close-Anweisung eine Exception auftreten kann, brauchen wir in unserem finally-Block den weiteren try-catch-Block. In den insgesamt zwei entstandenen catch-Blöcken innerhalb von `protokolliere` soll dabei die gleiche Exception geworfen werden.

```
finally
{
  if (writer != null)
  {
    try
    {
      writer.close();
    }
    catch (IOException e)
    {
      throw new ProtokollierException("<Fehlerbeschreibung wie im catch>");
    }
  }
}
```

5.2.5 *Zusatzaufgabe*: Die hier vorgestellte Variante zum Umgang mit Dateien basiert auf Java 1.6. Ab Java 1.7 ist es möglich, eine sogenannte „try-with-resources“-Anweisung zu verwenden, wodurch die Behandlung von Exceptions wesentlich vereinfacht wird. Erarbeitet euch die neue Variante durch eigene Recherche und baut den `Verleihprotokollierer` um.

Aufgabe 5.3 Pakete in Eclipse

Java bietet die Möglichkeit, Software zu modularisieren, indem Klassen in Pakete (engl: packages) aufgeteilt 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 in ein eigenes Paket *medien* verschieben. Welcher Fehler Tritt danach auf? **Schreibt einen kurzen Text, der die Situation erläutert.** Behebt das Problem anschließend. Dazu gibt es zwei grundlegende Möglichkeiten: Mitverschieben der benutzenden Typen in das neue Paket oder Exportieren der gebrauchten Typen an der Paketschnittstelle. Denkt daran, auch die zugehörigen Testklassen zu verschieben.

5.3.3 Alle Klassen und Interfaces sollen nun anhand ihrer fachlichen und technischen Ausprägung in passende Pakete verschoben werden:

- Neben Medien gibt es noch weitere Klassen die Dinge oder Personen aus dem Anwendungskontext abbilden. Findet diese und verschiebt sie alle in ein Paket *materialien*.
- Es gibt fachliche Dinge die einen Wertcharakter besitzen, heißt, sie sind unveränderlich. Ein Beispiel hierfür ist `Geldbetrag`. Findet diese Typen und verschiebt sie alle in ein Paket *fachwerte*.
- Einige Klassen und Interfaces verwalten Materialien, bieten systemweite fachliche Funktionen an oder dienen der Persistierung von Daten. Verschiebt diese Klassen in ein Paket *services*.
- Andere Klassen bilden die Benutzungsschnittstelle ab, hierzu zählen beispielsweise `AusleihWerkzeug` und `AusleihUI`. Findet alle Klassen, auf die dies zutrifft und verschiebt sie in ein Paket *werkzeuge*.
- Bei allen übriggebliebenen Klassen und Interfaces müsst ihr nun nachschauen, wer von Ihnen erbt oder wer sie verwendet um eine sinnvolle Einordnung in unser Schema zu finden.

5.3.4 *Zusatzaufgabe:* In Java existiert eine Konvention für Paket-Namen: Als Präfix verwendet man eine umgedrehte URL, um die Namen möglichst eindeutig zu machen. Anstatt des einfachen Präfixes *mediathek* kann beispielsweise der Präfix *de.uni_hamburg.informatik.swt.se2.mediathek* verwendet werden. Ändert die Paket-Namen entsprechend.

5.3.5 *Zusatzaufgabe:* Überlegt euch für jedes Paket, ob es sinnvoll ist, ein oder mehrere untergeordnete Pakete zu ergänzen. Erstellt mögliche weitere Pakete und verschiebt die entsprechenden Klassen.

Aufgabe 5.4 Programmierübung mit regulären Ausdrücken

5.4.1 Die Klasse `PLZ` wird benutzt, um Postleitzahlen abzubilden. In ihr wird eine sehr einfache Überprüfung verwendet, um zu testen, ob eine Postleitzahl korrekt ist. Überlegt euch einen regulären Ausdruck, der für folgende Beispiele passt:

12345 D-01237 d-12345

5.4.2 Baut die Klasse `PLZ` um, so dass in der Methode `istGueltig(String)` anhand des regulären Ausdrucks die Gültigkeit überprüft wird. Denkt daran, zuerst die Testklasse anzupassen!

5.4.3 Eine Postleitzahl soll unabhängig davon, ob D- oder d- der Ziffernkombination vorangestellt ist, gleich sein, wenn die Ziffernkombination identisch ist. Erweitert die Klasse entsprechend.