PM2, Aufgabenblatt 2

Programmieren 2 – Sommersemester 2017

Implementationsvererbung, Abstrakte Klassen, Fehlerbehandlung mit Exceptions

Ausgabedatum: ..... 17. April 2017

Kernbegriffe

Eine Java-Klasse kann mit dem Schlüsselwort **`extends`** von genau einer anderen Klasse erben (engl. *inherit*, *inheritance*). Die *Unterklasse* (engl. *subclass*, *derived class*) erbt von der *Oberklasse* (auch *Basisklasse*, engl.: *super class*, *base class*) zusätzlich zum Typ auch deren Implementation, also ihre Methoden und Exemplarvariablen. Aus diesem Grund spricht man hier von *Implementationsvererbung*. Wenn man als Autor einer Klasse keine Oberklasse angibt, wird von der Klasse `java.lang.Object` geerbt, in der u.a. `boolean equals(Object o)`, `int hashCode()` und `String toString()` definiert sind.

Geerbte Methoden können in Unterklassen *redefiniert* werden (engl.: *override*). Beim Redefinieren wird eine bestehende Implementation einer Operation durch eine andere ersetzt. Dies ist abzugrenzen vom Überladen (bekannt aus PR1), bei dem eine neue Operation eingeführt wird, die sich in ihrer Signatur von einer bereits bestehenden, gleichnamigen Operation unterscheidet.

Innerhalb einer redefinierenden Methode kann man die jeweilige redefinierte Methode mit Hilfe des Schlüsselworts **`super`** aufrufen:

```
super.methodenname(aktuelleParameter);   // Ruft die redefinierte Methode auf
```

Normalerweise können von einer Klasse mittels `new Klassenname` Exemplare erzeugt werden. Solche Klassen werden *konkrete Klassen* genannt. Durch die Einführung von Implementationsvererbung wird es jedoch sinnvoll, auch solche Klassen zu schreiben, deren Zweck ausschließlich in der Bereitstellung einer Implementationsbasis für Unterklassen besteht. Solche Oberklassen heißen *abstrakte Klassen* (engl. *abstract class*). Sie werden durch das Schlüsselwort **`abstract`** im Klassenkopf gekennzeichnet. Mit abstrakten Klassen sollen üblicherweise Redundanzen in der Implementierung mehrerer Klassen vermieden werden. Abstrakte Klassen haben Eigenschaften von Interfaces und konkreten Klassen: Wie bei Interfaces können keine Exemplare von ihnen erzeugt werden, abstrakte Klassen können aber Exemplarvariablen festlegen und Operationen durch Methoden implementieren.

Abstrakte Klassen deklarieren meist *abstrakte Methoden*, die ebenfalls durch das Schlüsselwort **`abstract`** gekennzeichnet werden und keinen Rumpf haben dürfen. Sie sollen in konkreten Unterklassen implementiert werden. Abstrakte Methoden können aus den Rümpfen der konkreten Methoden einer abstrakten Klasse aufgerufen werden. Eine solche konkrete Methode wird auch *Schablonenmethode* (engl. *template method*) und die verwendete Methode *Einschubmethode* (engl. *hook method*) genannt, wir nennen sie genauer eine *Einschuboperation*. Die Schablonenmethode der Oberklasse legt einen schematischen Ablauf fest, dessen Details eine Unterklasse durch die Implementierung der Einschuboperationen anpassen kann.

Eine gängige Praxis, um die Vorteile von Interfaces (multiples Subtyping) und abstrakten Klassen (Bereitstellung einer Implementationsbasis, Schablonenmethode) zu verbinden ist es, ein Interface bereit zu stellen und mit einer Basisimplementation in Form einer abstrakten Klasse zu ergänzen.

Obwohl von einer abstrakten Klasse keine Exemplare erzeugt werden können, ist sie trotzdem dafür zuständig, ihre Exemplarvariablen zu initialisieren. Deshalb haben Oberklassen, ob abstrakt oder nicht, Konstruktoren. (Wir erinnern uns: ein Konstruktor *erzeugt* kein Objekt, sondern *initialisiert* es nur.)

Die erste Anweisung in einem Konstruktor muss der Aufruf eines anderen Konstruktors sein:

```
super(aktuelleParameter);   // Ruft einen Konstruktor der Oberklasse auf

this (aktuelleParameter);   // Ruft einen Konstruktor der eigenen Klasse auf
```

(Fehlt dieser Konstruktoraufruf, ruft Java den Default-Konstruktor der Oberklasse auf, so als hätte man **`super();`** geschrieben.) Auf diese Weise entsteht eine *Konstruktorkette* (engl. *constructor chain*).

In Softwaresystemen können zur Laufzeit *Fehlerzustände* auftreten. Deren Ursachen unterscheiden wir (orthogonal zu bisherigen Begriffen) aus Sicht des Entwicklungsteams in *Entwicklungsfehler* (vom Team verantwortet) und *Umgebungsfehler* (in der Laufzeitumgebung des Produktivsystems). 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 Aufruf über eine Referenz, deren Wert `null` ist, 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“, denn 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 also eher geeignet für das Melden von Entwicklungsfehlern, da diese zur Laufzeit nicht sinnvoll behandelt werden können (sondern direkt im Quelltext korrigiert werden sollten). Ungeprüfte Exceptions werden z.B. bei fehlgeschlagenen `assert`-Anweisungen geworfen, weitere Beispiele sind Exemplare der Klassen `NullPointerException` oder `ArithmeticException`.

Aufgabe 2.1 Mediathek auf Implementationsvererbung umstellen

`CD`, `DVD` und `Videospiel` besitzen Gemeinsamkeiten, die bisher in jeder Klasse implementiert sind. Diese Codeduplizierung wollen wir nun beseitigen und gleiche Methoden und Exemplarvariablen durch eine gemeinsame Oberklasse bündeln.

2.1.1 Importiert zuerst das Projekt aus der Archivdatei *MediathekBlatt02.zip*. In dieser Vorlage ist es möglich, Medien zurückzunehmen. Startet das Programm und experimentiert mit der Oberfläche, damit ihr wisst, welche Bestandteile hinzugekommen sind.

2.1.2 Erstellt eine abstrakte Klasse `AbstractMedium`, die die Gemeinsamkeiten der Klassen `CD`, `DVD` und `Videospiel` in einer Basisimplementation zusammenführt (Implementationsvererbung).

Lasst die oben genannten Klassen von `AbstractMedium` erben und entfernt den redundanten Code aus ihnen. Stellt durch Ausführen der Testklassen und des Programms sicher, dass die Mediathek wie bisher funktioniert.

2.1.3 **Zeichnet ein UML-Klassendiagramm**, das für die Klassen `DVD`, `CD` und `Videospiel` zeigt, wo Typvererbung und wo Implementationsvererbung eingesetzt wird. Schaut zu dem Thema auch in das PM2-Skript, Teil 1. Führt auch **Operationen** und **Methoden** mit auf, da es auf diesem Blatt verstärkt um deren Zusammenhänge in Vererbungshierarchien geht. Seid großzügig mit dem Platz, da ihr dieses Diagramm in der nächsten Aufgabe noch ergänzt.
Wenn ihr es digital zeichnet: **Bringt einen Ausdruck mit!**

2.1.4 Sofern noch nicht geschehen, sollen jetzt Teile der Methode `getFormatiertenString()` ebenfalls in die Klasse `AbstractMedium` „hochgezogen“ werden: Implementiert die Methode `getFormatiertenString()` in `AbstractMedium` so, dass sie alle dort vorhandenen Attribute als formatierten String zurückgibt. Ruft bei der Implementierung von `getFormatierten-String()` in den Unterklassen mit Hilfe des Schlüsselworts **`super`** die Implementation der Oberklasse auf und hängt danach nur noch die zusätzlichen Attribute an den String. Testet mit den Testklassen und der grafischen Benutzungsoberfläche.

Aufgabe 2.2 Abstrakte Methoden

Die Rückgabe-Ansicht zeigt eine Spalte an, in der die angelaufene Mietgebühr für ein entliehenes Medium dargestellt werden soll. In dieser Aufgabe werden wir dies nun implementieren.

2.2.1 Ergänzt das Interface `Medium` um die folgende Operation:

```
/**
 * Berechnet die Mietgebühr in Eurocent für eine angegebene Mietdauer
 * in Tagen
 *
 * @param mietTage
 *            Die Anzahl der Ausleihtage eines Mediums
 * @return Die Mietgebühr in Eurocent als Geldbetrag
 *
 * @require mietTage > 0
 *
 * @ensure result != null
 */
Geldbetrag berechneMietgebuehr(int mietTage);
```

2.2.2 Ergänzt die Klasse `CDTest` um sinnvolle Testfälle basierend auf der Annahme, dass pro Miet-Tag 300 Euro-Cent berechnet werden. Kopiert diese Testfälle in die anderen Testklassen hinein.

2.2.3 Implementiert nun die Operation `berechneMietgebuehr(int)` in `AbstractMedium`.

2.2.4 Öffnet über *Window->Show View->Tasks* die Tasks-View. In ihr gibt es zwei *ToDo*-Einträge, die sich auf diese Aufgabe beziehen. Über einen Doppelklick gelangt ihr zu den entsprechenden Quelltextstellen, an denen ihr noch etwas erledigen sollt. Jetzt sollte die von euch berechnete Mietgebühr in der Rückgabe-Ansicht angezeigt werden.

ToDos werden oft in Quelltexten verwendet, um sich selbst und andere Programmierer an noch zu erledigende Aufgaben zu erinnern. Eclipse zeigt alle im Projekt vorkommenden ToDos praktischerweise in einer Tasks-View an, so dass man auch keine vergisst.

2.2.5 Die Berechnung der Mietgebühr von *Videospielen* unterscheidet sich doch stärker als gedacht von den anderen Medien. Die Mietgebühr soll immer 200 Euro-Cent betragen, unabhängig davon, wie lange ein Videospiel ausgeliehen wird. Ändert die Klasse `VideospielTest` entsprechend. Der Test sollte nun erstmal fehlschlagen.

Die Methode `berechneMietgebuehr(int)` der abstrakten Oberklasse kann nun in der Klasse `Videospiel` nicht mehr verwendet werden. Überschreibt die Operation `berechneMietgebuehr(int)` in `Videospiel`, so dass der Test erfolgreich durchläuft.

Aufgabe 2.3 Schablonenmethode und Einschub-Operation

2.3.1 Die bisherige Berechnung der Mietgebühren für Videospiele gefällt dem Mediathekar nicht mehr. Er möchte zukünftig sehr unterschiedliche Mietgebühren für PC- und Konsolenspiele erheben. Dafür benötigen wir zwei neue Klassen für PC- und Konsolenvideospiele. Ändert die konkrete Klasse `Videospiel` in eine abstrakte Klasse `AbstractVideospiel`. Ergänzt die konkreten Unterklassen `PCVideospiel` und `KonsolenVideospiel`.
Diese beiden Unterklassen sind erst einmal leer, nur die Konstruktoren solltet ihr schon einfügen.

2.3.2 Erstellt zu den beiden neuen Klassen die zugehörigen Testklassen, indem ihr `VideospielTest` einmal kopiert und die Namen beider Klassen entsprechend der neuen Videospieltypen anpasst. Implementiert nun die Tests. Die Mietgebühren werden wie folgt berechnet:

Die Mietgebühr teilt sich auf in einen fixen Basispreis von 200 Euro-Cent für alle Videospiele und einen zeitabhängigen Preisanteil, der dazu addiert wird. Der zeitabhängige Preisanteil beträgt für `KonsolenVideospiele` 700 Euro-Cent für jeweils 3 volle Tage. Für ein `PCVideospiel` soll für die ersten 7 Tage gar nichts und dann je angefangene 5 Tage 500 Euro-Cent verlangt werden.

Füllt jeweils eine Tabelle (ähnlich den unten angegebenen, nur sinnvollerweise mit mehr Zeilen) mit Testdaten, die ihr in den Testklassen verwendet. Achtet bei der Wahl der Testdaten auf Äquivalenzklassen (Für welche Tage sollte dieselbe Gebühr anfallen?) und Grenzwerte (Bei welchen

Testdaten erwarten wir einen „Sprung“ in den Ergebnissen?).
Die Tests werden erst einmal fehlschlagen, da es ja noch keine Implementation gibt.

KonsolenVideospiel	
Anzahl Tage	Preis

PCVideospiel	
Anzahl Tage	Preis

2.3.3 Ergänzt die Klasse `AbstractVideospiel` um eine Klassenkonstante für den Basispreis. Den zeitabhängigen Preisanteil sollen die konkreten Subklassen definieren. Erstellt dafür in `AbstractVideospiel` die abstrakte Methode `getPreisNachTagen(int)`, die den zeitabhängigen Preisanteil liefern soll. Implementiert diese Operation in `PCVideospiel` und `KonsolenVideospiel`.

Die Operation `berechneMietgebuehr(int)` in der Klasse `AbstractVideospiel` soll nun so implementiert werden, dass sie den Basispreis und den zeitabhängigen Preisanteil addiert. Den zeitabhängigen Preisanteil liefert dabei eure abstrakte Methode `getPreisNachTagen(int)`. Denkt daran, die Testklassen auszuführen.

Bei welcher Methode handelt es sich um die Schablonenmethode, bei welcher um die Einschub-Operation?

2.3.4 Sorgt in der Klasse `Medieneinleser` dafür, dass PC- und Konsolen-Videospiele statt Videospielen eingelesen werden.

2.3.5 Ergänzt und ändert das Klassendiagramm um die neu erstellten Klassen.

Aufgabe 2.4 Vererbung und Testklassen

Die Testklassen, die in den vorigen Aufgaben verändert wurden, enthalten sehr viel duplizierten Quelltext. Diese Form von Redundanz wollen wir vermeiden.

2.4.1 Erstellt eine Testklasse `AbstractVideospielTest`, die die gemeinsamen Anteile der beiden Testklassen zu Videospielen zusammenfasst. Die beiden Testklassen sollen von dieser Klasse erben und nur noch die jeweils spezifischen Anteile enthalten.

Tipp: Die Gemeinsamkeiten der beiden Klassen lassen sich in Eclipse sehr komfortabel mit der Compare-Funktion anzeigen: Beide Testklassen im Explorer selektieren (Strg- bzw. Ctrl-Taste) und dann im Kontext-Menü *Compare With->Each Other* aufrufen.

Um die ähnlichen Anteile noch leichter zusammenfassen zu können, bietet sich eventuell eine abstrakte Hilfsmethode an, die ein zu testendes Exemplar liefert. Diese kann in den Testmethoden verwendet werden.

2.4.2 Geht in ähnlicher Weise auch für `AbstractMedium` vor: Erstellt zuerst eine passende abstrakte Testklasse, lasst dann die bestehenden Testklassen von dieser erben und entfernt redundante Methoden.

2.4.3 Ergänzt erneut euer **Klassendiagramm** um die neuen Klassen.

2.4.4 *Zusatzaufgabe*: Um noch konsequenter Redundanzen zu vermeiden, soll nun `AbstractVideospielTest` von `AbstractMediumTest` erben. Möglicherweise sind dazu wieder einige Anpassungen notwendig.

2.4.5 *Zusatzaufgabe*: Lest Euch im Text zu den Kernbegriffen den Abschnitt zu Schablonen- und Einschubmethode durch. Welche Methoden haben welche Rollen in euren Testklassen? Falls ihr keine Einschubmethode verwendet habt, beschreibt stattdessen die Rollen der Methoden in 2.3.3.

Aufgabe 2.5 Exceptions behandeln

Anhand eines Beispiels nähern wir uns dem Thema 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.

2.5.1 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.

2.5.2 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 passenden Schnittstellenkommentar. Anfangs soll der Protokollierer die Daten auf die Konsole ausgeben.

2.5.3 Sorgt dafür, dass die Klasse `VerleihServiceImpl` ihre Verleihvorgänge mit Hilfe des `VerleihProtokollierers` protokolliert.

2.5.4 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 `FileWriters` 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.

2.5.5 Schaut euch die Dokumentation der Methode `FileWriter.write(String)` in der Java-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.

2.5.6 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 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?

2.5.7 Ergänzt nun euer Klassendiagramm um die Klassen `AusleihWerkzeug, RueckgabeWerkzeug, VerleihServiceImpl`, `VerleihProtokollierer` und `IOException`.

2.5.8 *Zusatzaufgabe*: Erstelle für den ersten Parameter der Methode protokolliere einen Enum-Typ `VerleihEreignis`. Schaue Dir hierzu die Dokumentation zu Enum-Typen an: http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html.

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

Aufgabe 2.6 Zuständigkeit für Fehlerbehandlungen festlegen

In Aufgabe 2.5 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.

2.6.1 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.

2.6.2 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.

2.6.3 An mehreren Stellen im Projekt zeigt euch Eclipse an, dass das Kompilieren nun fehlschlägt. Geht zuerst zur Klasse `AusleiheWerkzeug`. 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. Verfahrt sinngemäß mit den anderen Fehlerstellen.

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

2.6.4 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.

Findet Dokumentation zu try-with-resources in Java und setzt das beschriebene Konzept hier für den `FileWriter` um.

2.6.5 Ergänzt nun euer Klassendiagramm um die Klassen `ProtokollierException`, `Exception`, `AusleihWerkzeug` und `JOptionPane`.