

SE2, Aufgabenblatt 4

Modul: Softwareentwicklung II – Sommersemester 2017

Implementationsvererbung, Abstrakte Klassen, Schablonenmethode, Konstruktorkette

Moodle-URL moodle.informatik.uni-hamburg.de

Projektraum Softwareentwicklung 2 SoSe2017

Ausgabedatum 27. April 2017

Kernbegriffe

Bisher haben wir gesehen, wie eine Klasse ein Interface implementiert, wie ein Interface andere Interfaces erweitert und wie sich Typhierarchien bilden lassen, um Ersetzbarkeit zu gewährleisten.

Eine 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 *überschrieben* werden (unglückliche Eindeutschung des engl.: *override, was aufheben, aushebeln, außer Kraft setzen, ersetzen* bedeutet). Beim Überschreiben wird eine bestehende Implementation einer Operation durch eine andere ersetzt. Dies ist abzugrenzen vom Überladen (bekannt aus SE1), bei dem eine neue Operation eingeführt wird, die sich lediglich in ihrer Parameterliste von einer bereits bestehenden, gleichnamigen Operation unterscheidet.

Innerhalb einer überschreibenden Methode kann man die jeweilige überschriebene Methode mit Hilfe des Schlüsselworts **super** aufrufen:

```
super.methodenname(aktuelleParameter); // Ruft die überschriebene 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 definieren meist *abstrakte Methoden*, die ebenfalls durch das Schlüsselwort **abstract** gekennzeichnet werden und keinen eigenen 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. Die Schablonenmethode der Oberklasse legt einen schematischen Ablauf fest, dessen Details eine Unterklasse durch die Implementierung der Einschubmethoden 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 Konstruktoraufwurf, ruft Java den Default-Konstruktor der Oberklasse auf, so als hätte man `super()`; hingeschrieben.) Auf diese Weise entsteht eine *Konstruktorkette* (engl. *constructor chain*).

Aufgabe 4.0 Vierer-Entwicklungsteam für Laborphase 2 bilden


In der Laborphase 2 arbeitet ihr in einem Team aus 4 Entwicklerinnen und Entwicklern. Bildet bis zu eurem nächsten regulären Termin in Übungswoche 5 dieses Team und tragt beim nächsten Termin die Namen in einem Gruppenabnahmезettel ein, den ihr bei den Betreuern erhaltet. Die Erfüllung dieser Aufgabe ist scheinrelevant. Es ist nicht möglich, über Labortermine hinweg Teams zu bilden!

Aufgabe 4.1 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.

- 4.1.1 Importiert zuerst das Projekt aus der Archivdatei `Mediathek_Vorlage_Blatt04-05.zip`. In dieser Vorlage ist es möglich, Medien zurückzunehmen. Startet das Programm und experimentiert kurz mit der Oberfläche herum, damit ihr wisst, welche Bestandteile hinzugekommen sind.
- 4.1.2 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.

-  4.1.3 **Zeichnet ein UML-Klassendiagramm**, das für die Klassen `DVD`, `CD` und `Videospiel` zeigt, wo Typvererbung und wo Implementationsvererbung eingesetzt wird. Seid großzügig mit dem Platz, da ihr dieses Diagramm in der nächsten Aufgabe noch ergänzt.
- 4.1.4 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 `getFormatiertenString()` in den Unterklassen mit dem **super**-Schlüsselworts 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 4.2 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.

4.2.1 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);
```

4.2.2 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.

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

4.2.4 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.

*ToDo*s werden oft in Quelltexten verwendet, um sich selbst und andere Programmierer an noch zu erledigende Aufgaben zu erinnern. Eclipse zeigt alle im Projekt vorkommenden *ToDo*s praktischerweise in einer Tasks-View an, so dass man auch keine vergisst. **Hinweis:** Löscht die Kommentare bitte nicht, damit ihr die *ToDo*s für die Abnahme wiederfindet.

4.2.5 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 nicht mehr verwendet werden. Überschreibt die Operation `berechneMietgebuehr(int)` in der Klasse `Videospiel`, so dass der Test erfolgreich durchläuft.

Aufgabe 4.3 Schablonen- und Einschubmethode

4.3.1 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.

4.3.2 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 volle 3 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 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
1	200
3	900
4	900
6	1600

PCVideospiel	
Anzahl Tage	Preis
7	200
8	700
13	1200
18	1700

- 4.3.3 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 `int 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 Schablonen- bzw. um die Einschubmethode?

- 4.3.4 Arbeitet nun die ToDos im Eclipse-Projekt für diese Aufgabe ab. Nun werden PC- und Konsolenvideospiele im Programm verwendet.

-  4.3.5 Ergänzt und ändert das Klassendiagramm um die neu erstellten Klassen. Öffnet im Interface Medium die Quick Type Hierarchie. Diese eignet sich sehr gut, um die gewählte Variante zu diskutieren.