SE2, Aufgabenblatt 4

Modul: Softwareentwicklung II – Sommersemester 2012

Implementationsvererbung, Abstrakte Klassen, Schablonenmethode, Konstruktorkette

CommSy-Projektraum .....SE-2 CommSy SoSe 2012
Ausgabedatum .....26. April 2012

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. Nun betrachten wir den Fall, in dem eine Klasse über das Schlüsselwort `extends` von einer anderen *Klasse* erbt (engl. *inherit*, *inheritance*). Sie erbt von der *Oberklasse* (auch *Basisklasse*, engl.: *super class*, *base class*) zusätzlich zum Typ auch deren Implementation – diese besteht aus den von ihr definierten Methoden und Exemplarvariablen; deshalb spricht man hier von *Implementationsvererbung*. Ist keine Oberklasse angegeben, wird in Java immer automatisch von `java.lang.Object` geerbt.

In Java kann eine Klasse nur von *genau einer* anderen Klasse erben, Implementationsvererbung ist also in Java nur *einfach* möglich (*Einfachvererbung*). Wir erinnern uns: Die Typvererbung über Interfaces ist in Java mehrfach möglich.

In der Regel können von einer Klasse über einen Konstruktoraufruf Exemplare erzeugt werden. Derartige 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 erbende Klassen (*Subklassen*, *Unterklassen*, engl. *subclass*) 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 Ablauf oder eine Berechnung fest, eine Unterklasse kann diesen Ablauf anpassen, indem sie die Einschubmethoden implementiert.

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 abstrakten Klassen keine direkten Exemplare erzeugt werden können, sind sie doch dafür zuständig, ihren Zustand (ihre Exemplarvariablen) zu initialisieren. Deshalb haben Oberklassen, ob abstrakt oder nicht, Konstruktoren. Bei der Erzeugung eines Exemplars werden entlang der Klassenhierarchie, bei der obersten Klasse beginnend, Konstruktoren aller Oberklassen ausgeführt, um den inneren Zustand des Objekts schrittweise zu initialisieren. Dabei wird im jeweiligen Unterklassenkonstruktor festgelegt, welcher Konstruktor der Oberklasse – falls sie mehrere hat – zuvor ausgeführt wird: Die erste Anweisung in einem Konstruktor muss der Aufruf des Oberklassenkonstruktors über das Schlüsselwort `super` sein (wenn dieser Aufruf fehlt und die Oberklasse einen parameterlosen Konstruktor hat, wird automatisch dieser verwendet). Es entsteht so eine Konstruktorkette (engl. *constructor chain*).

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 neuen Mediathek ist es nun 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 Klassendiagramm**, das für die Klassen `DVD`, `CD` und `Videospiel` zeigt, wo Typvererbung und wo Implementationsvererbung benutzt wird. Schaut zu dem Thema auch in das OOPM-Skript, Teil 1. Lasst etwas 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 `getFormatierten-String()` in den Unterklassen mittels des **`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 Euro, für den Fall, dass das Medium am
 * Datum "von" ausgeliehen wird und am Datum "bis" zurückgegeben wird.
 * Für ein Medium, das an einem Tag ausgeliehen wird und am nächsten
 * Tag zurückgegeben wird, werden 2 Miettage berechnet.
 *
 * @param von
 *            Das Ausleihdatum.
 * @param bis
 *            Das Rückgabedatum.
 * @return Die Mietgebühr in Euro als Geldbetrag.
 *
 * @require von != null
 * @require bis != null
 * @require von.compareTo(bis) <= 0
 *
 * @ensure result != null
 */
Geldbetrag berechneMietgebuehr(Datum von, Datum bis);
```

Diese Operation soll in `AbstractMedium` so implementiert werden, dass 300 Euro-Cent pro Miet-Tag berechnet werden. Der erste Tag zählt mit, das heißt, wenn man am selben Tag ausleiht und zurückgibt, werden bereits 300 Euro-Cent fällig. Verwendet hierbei die Operation `tageSeit(Datum)` der Klasse `Datum`. Ergänzt *vorher* die drei Testklassen.

4.2.2 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 Quelltext-Stellen, an denen ihr noch etwas erledigen sollt. Danach wird das Ergebnis eurer Methode in der Rückgabe-Ansicht verwendet, um den Mietpreis anzuzeigen.

ToDos werden oft in Quelltexten verwendet, um sich 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.

4.2.3 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 soll nun erstmal fehlschlagen.

4.2.4 Die Methode `berechneMietGebuehr(Datum von, Datum bis)` der abstrakten Oberklasse kann nun nicht mehr verwendet werden. Implementiert die Operation `berechneMietGebuehr(Datum, Datum)` in der Klasse `Videospiel`, so dass der Test erfolgreich durchläuft.

Aufgabe 4.3 Schablonen- und Einschubmethode

4.3.1 Der Mediathekar hat festgestellt, dass das keine gute Idee war. Er möchte stattdessen zukünftig sehr unterschiedliche Mietgebühren für PC- und Konsolenspiele erheben. Aufgrund der unterschiedlichen Berechnung benötigen wir zwei neue Klassen für PC- und Konsolenvideospiele: macht aus der bisherigen Klasse `Videospiel` eine neue abstrakte Klasse `AbstractVideospiel`. Ergänzt die konkreten Unterklassen `PCVideospiel` und `KonsolenVideospiel`. Diese Klassen 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 Gebühr 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. Der erste Tag wird mitgezählt. Wird ein ausgeliehenes Medium am selben Tag wieder zurückgegeben, ist die Anzahl der Tage also 1. Beispiele: für ein Videospiel, das am 1. Mai ausgeliehen und am 2. Mai zurückgegeben wird, werden 200 Euro-Cent berechnet. Wird das Spiel erst am 3. Mai zurückgegeben, werden 900 Euro-Cent fällig.

Beim zeitabhängigen Preisanteil soll für `PCVideospiel` für die ersten 7 Tage gar nichts und dann je angefangene 5 Tage 500 Euro-Cent verlangt werden (der 1. Tag zählt wieder mit).

Füllt jeweils eine Tabelle mit Testdaten, die ihr in den Testklassen verwendet. Achtet bei der Wahl der Testdaten auf Äquivalenzklassen und Grenzwerte. Die Tests werden erstmal fehlschlagen, da es ja noch keine Implementation gibt.

PCVideospiel	
Anzahl Tage	Preis

KonsolenVideospiel	
Anzahl Tage	Preis

4.3.3 Ergänzt die Klasse `AbstractVideospiel` um ein Zustandsfeld für den Basispreis, das im Konstruktor initialisiert wird. 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(Datum, Datum)` 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`. 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.

Aufgabe 4.4 Vererbung und Testklassen

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

4.4.1 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.

4.4.2 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.

4.4.3 *Zusatzaufgabe*: Um noch konsequenter Redundanzen zu vermeiden, soll nun `AbstractVideospielTest` von der abstrakten Testklasse aus 4.4.2 erben. Möglicherweise sind dazu wieder einige Anpassungen notwendig.

4.4.4 *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 4.3.3.