SE2, Aufgabenblatt 2

Modul: Softwareentwicklung II – Sommersemester 2012

Vertragsmodell, Debugging, Testen, Test-First

CommSy-Projektraum ..... SE2 CommSy SoSe 2012
Ausgabedatum ..... 12. April 2012

Kernbegriffe

Das *Vertragsmodell* (engl.: design by contract) von Bertrand Meyer formalisiert das Aufrufen einer Operation an einem Objekt. Das aufrufende Objekt wird von Meyer als *Klient* bezeichnet, das aufgerufene als *Dienstleister*. Die Beziehung zwischen den beiden wird als ein *Vertragsverhältnis* aufgefasst: Der Klient fordert eine Dienstleistung beim Dienstleister an, indem er eine Operation des Dienstleisters aufruft. Der Dienstleister muss die Dienstleistung nur erbringen, wenn der Klient bestimmte *Vorbedingungen* erfüllt, die der Dienstleister für die Operation formuliert; hält der Klient sich nicht an seinen Teil des Vertrags, muss der Dienstleister es auch nicht tun. Wenn der Klient jedoch die Vorbedingungen erfüllt, dann ist der Dienstleister verpflichtet, die Dienstleistung zu erbringen. In den *Nachbedingungen* wird festgelegt, wie sich die Dienstleistung auswirkt. Vor- und Nachbedingungen werden auch unter dem Begriff *Zusicherungen* zusammengefasst.

Der Begriff *Test-First* beschreibt ein Vorgehen bei der Implementierung einzelner Klassen zusammen mit ihrer jeweiligen Testklasse. Vor der Implementierung einer Klasse wird die zugehörige Testklasse geschrieben. Die zu testenden Operationen der Klasse werden erst *nach* ihren Testmethoden implementiert. Es wird nur implementiert, was auch von der Testklasse überprüft wird. Von diesem Vorgehen verspricht man sich unter anderem besser zu benutzende Schnittstellen, da die Testklasse sich primär auf die Schnittstelle der zu testenden Klasse bezieht. Da man außerdem nur implementiert, was auch in der Testklasse abgeprüft wird, entsteht kein ungetesteter Quelltext. Ein Test beschreibt eine Spezifikation, die eine Klasse einhalten muss.

Konvention für Testklassen in den SE2-Übungen: Es soll *nicht* getestet werden, ob ein Dienstleister überprüft, ob seine Vorbedingungen vom Klienten eingehalten werden (keine Negativtests für Vorbedingungen).

Ab jetzt gilt immer: Bei neuem Code das Vertragsmodell einsetzen (wenn sinnvoll), intensiv testen und gründlich kommentieren (javadoc), auch wenn das nicht explizit gefordert ist.

Aufgabe 2.1 Vertragsmodell einsetzen

Innerhalb der Mediathek wird das Vertragsmodell verwendet. Das Vertragsmodell wurde bisher nur in der Programmiersprache *Eiffel* mit eigenen Sprachmitteln verwirklicht. In Java besteht die Umsetzung aus zwei Konventionen: Der Dienstleister *deklariert* seine Vor- und Nachbedingungen, indem er sie im javadoc-Kommentar einer Operation mit `@require` und `@ensure` angibt; somit kann sich der Entwickler eines Klienten über den Vertrag informieren, indem er die Schnittstellenbeschreibung liest. Außerdem *überprüft* der Dienstleister die Einhaltung des Vertrags, indem er in der implementierenden Methode die Zusicherungen mit assert-Anweisungen testet.

2.1.1 Das Projekt von letzter Woche ist leider fehlerhaft von uns ausgeliefert worden. Bitte löscht es (Rechtsklick auf das Projekt, Delete, [x] Delete project contents on disk, OK) und importiert dann das korrigierte Projekt `Mediathek_Vorlage_Blatt02-03` aus dem CommSy.

2.1.2 Lasst alle im Projekt enthaltenen Tests durchlaufen, indem ihr im Kontextmenü des *src*-Ordners *Run As->JUnit* Test auswählt. Ein Test wird vermutlich nicht durchlaufen. Geht zu der Stelle im Quelltext des fehlgeschlagenen Tests und folgt den dort in den Implementationskommentaren stehenden Anweisungen. Führt die Tests erneut aus. Sie sollten jetzt alle durchlaufen, denn nun werden die assert-Anweisungen immer überprüft. Der Parameter `-ea` (steht für *enable assertions*) wird ab jetzt bei jedem Programmstart an die virtuelle Maschine übergeben. Wenn ihr den Rechner wechselt, müsst ihr diese Einstellung erneut vornehmen!

2.1.3 Seht Euch die Klasse `Verleihkarte` genau an. An welchen Stellen werden Vor- und Nachbedingungen deklariert und überprüft? Welche Regeln gelten für Syntax, Semantik und Pragmatik bei diesen Bedingungen? **Versucht, diese Regeln schriftlich zu formulieren!** Hinweis: Achtet insbesondere auf die Methodenkommentare (Position, Schlüsselwörter!) und die Position und den Aufbau der assert-Anweisungen. Wie sind die Anweisungen syntaktisch aufgebaut? Versucht, Dokumentation zu assert-Anweisungen in Java 1.6 zu finden.

2.1.4 Schaut euch die im Interface `VerleihService` deklarierten Vor- und Nachbedingungen an. Die hier gemachten vertraglichen Zusagen werden nicht von der implementierenden Klasse `VerleihServiceImpl` umgesetzt. Dies müssen wir ändern! Arbeitet hierfür zuerst das Infoblatt *Anlegen von Templates in Eclipse* durch. Jetzt könnt ihr alle fehlenden assert-Anweisungen einfügen.

2.1.5 Das Interface `MedienbestandService` besitzt noch nicht einmal Vor- und Nachbedingungen. Schreibt sinnvolle Vertragsbedingungen in die Kommentare und ergänzt danach die benötigten assert-Anweisungen in der implementierenden Klasse `MedienbestandServiceImpl`.

Aufgabe 2.2 Debugger verwenden

In der vergangenen Woche habt ihr gesehen, dass die Mediathek noch fehlerhaft ist: die Ausleihe funktioniert nicht. Wir verwenden nun den Debugger, um herauszufinden, in welcher Klasse der Fehler liegt. Dabei beginnen wir mit unserer Suche in der Klasse, die die Interaktion an der Benutzungsschnittstelle steuert.

2.2.1 Öffnet die Klasse `AusleiheWerkzeug`. Setzt in die erste Zeile der Methode `leiheAusgewaehlteMedienAus` einen *Haltepunkt* (engl. *breakpoint*). Findet heraus, wie das geht, z.B. über die in Eclipse integrierte Hilfe.

2.2.2 Um eine Anwendung zu debuggen, muss sie im Debug-Modus gestartet werden. Startet erneut die Anwendung, dieses Mal jedoch mit *Debug As->Java Application*. Öffnet dann die *Debug-Perspective* auf die gleiche Weise, wie ihr zuvor die Java-Perspektive geöffnet habt.

Selektiert nun in der Mediathek einen Kunden und mehrere Medien. Drückt dann den Button *ausleihen*. Die Anwendung bleibt an eurem Haltepunkt stehen. Die Fenster der Debug-Perspektive zeigen den aktuellen Zustand des Programms. Oben rechts könnt Ihr die Werte der Variablen betrachten. Oben links findet ihr den *Aufruf-Stack* (im *Debug-View*). Die aktuelle Position im Quelltext wird im mittleren Fenster angezeigt.

2.2.3 Führt die ersten vier Zeilen der Methode mit Hilfe des Debuggers schrittweise aus, indem ihr den Button *step over (F6)* drückt. Beobachtet dabei, wie oben rechts die neuen lokalen Variablen erscheinen.

2.2.4 Springt nun in die Methode `verleiheAn` mit dem Button *step into (F5)*. Beobachtet dabei, wie sich die Variablenanzeige oben rechts ändert und was mit dem Aufruf-Stack passiert. Macht euch insgesamt klar, was die verschiedenen Fenster anzeigen und was die verschiedenen Buttons über dem Debug-View bedeuten. Lauft so lange durch das Programm, bis ihr die fehlerhafte Programmierzeile findet. Diese verbessert ihr aber noch nicht in dieser Aufgabe! Wenn ihr in der letzten Zeile von `verleiheAn` angekommen seid, drückt ihr auf den Button *Resume (F8)*, um das Programm weiterlaufen zu lassen.

2.2.5 Führt bei der Abnahme vor, wie ihr den Debugger verwendet, um von eurem Haltepunkt bis zum Ende der Methode `verleiheAn` zu gehen. Diskutiert mit euren BetreuerInnen, wann man den Debugger einsetzen sollte und wann nicht.

Aufgabe 2.3 Fehler durch Tests finden und beheben

Für die Klasse `VerleihServiceImpl` hat unser Team die Testklasse vergessen! In einem solchen Fall wird zuerst ein Test geschrieben, der den Fehler findet, und danach wird der Fehler in der getesteten Klasse behoben. JUnit 3.8 kennt ihr bereits aus SE1. In SE2 werden wir jedoch JUnit 4.0 verwenden.

2.3.1 JUnit 4.0 in Eclipse: Schaut euch den Quelltext der Klasse `CDTest` an. In ihm findet ihr mehrere Implementationskommentare mit Erklärungen, die beschreiben, welche Änderungen es in JUnit 4.0 gibt. Lest Sie euch durch, damit ihr zukünftig selber Testklassen schreiben könnt. Um einen Test in Eclipse auszuführen, wählt im Package Explorer die Klasse `CDTest` aus. Startet den JUnit-Test mit dem Eintrag *RunAs>JUnitTest* im Kontextmenü. Wenn alles gut geht, seht Ihr den grünen Balken, der euch verrät, dass der Test erfolgreich war.

2.3.2 Ergänzt die Testklasse `VerleihServiceImplTest` um einen Testfall, der den in Aufgabe 2.2.4 gefundenen Fehler aufdeckt. Schreibt außerdem mindestens einen weiteren Testfall.

2.3.3 Nachdem jetzt ein Test existiert, der den Fehler findet, könnt ihr diesen in `VerleihServiceImpl` beheben.

Aufgabe 2.4 Test-First-Entwicklung der Klasse DVD

Ihr sollt nun Software mit Hilfe von JUnit entwickeln. Bei *Test-First-Entwicklung* schreibt man zuerst den Test und implementiert dann eine Klasse, die diesem Test genügt.

2.4.1 Wir wollen die Klasse `DVD` entwickeln. Genau wie die `CD` soll die `DVD` ein `Medium` sein, dass in der Mediathek entliehen werden kann. Damit der Test kompiliert ist es ratsam, im ersten Schritt eine leere Version der zu testenden Klasse zu schreiben. Implementiert die Klasse `DVD` ohne Funktionalität. Es müssen lediglich die Methoden existieren; die Rümpfe der Methoden sollen jedoch noch leer sein bzw. nur ein Dummy-Return-Statement enthalten.

Die Schnittstelle der zu testenden Klasse soll folgendermaßen aussehen:

```
public String getTitel()
public String getKommentar()
public String getRegisseur()
public int getLaufzeit()
public String get MedienBezeichnung()
```

Im Konstruktor sollen die nötigen Daten für eine DVD übergeben werden, sodass keine set-Operationen implementiert werden müssen. Der Konstruktor hat dann die Form:

```
public DVD(String titel, String kommentar, String regisseur, int lauf-
zeit)
```

2.4.2 Implementiert eine Testklasse namens `DVDTest`. Schaut euch dafür die bereits bestehende Klasse `CDTest` als Beispiel an.

2.4.3 Zu diesem Zeitpunkt sollten beide Klassen kompilieren (es sollte bei den Klassen kein rotes x mehr angezeigt werden). Lasst den Test durchlaufen und seht, dass er fehlschlägt.

2.4.4 Nun implementiert ihr in `DVD` eine Methode nach der anderen. Dazu schreibt ihr zuerst für jede Methode einen Schnittstellenkommentar und danach den Quellcode, der die beschriebene Dienstleistung erbringt. Zwischendurch testet ihr immer wieder, bis alle Tests erfolgreich sind.

2.4.5 Jetzt sollen auch DVDs in der Mediathek angeboten werden. In der Methode `leseMediumEin` im `MedienEinleser` steht bereits der dafür nötige Quelltext. Die Zeile, in der DVDs erzeugt werden, ist auskommentiert. Nehmt die Kommentarzeichen weg und startet testweise die Mediathek.