

SE2, Aufgabenblatt 2

Modul: Softwareentwicklung II – Sommersemester 2015

Vertragsmodell, Debugging, Testen

CommSy-Projektraum SE2 CommSy SoSe 2015

Ausgabedatum 9. April 2015

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.


Konvention für Testklassen in SE2: Es soll *nicht* getestet werden, ob ein Dienstleister sicherstellt, dass 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. In Java gibt es keine Sprachunterstützung für das Vertragsmodell; stattdessen muss man sich mit Konventionen behelfen. In SE2 sehen diese Konventionen wie folgt aus: Der Dienstleister *deklariert* seine Vor- und Nachbedingungen gegenüber dem Klienten, indem er sie im javadoc-Kommentar einer Operation mit `@require` und `@ensure` angibt. Weiterhin *überprüft* der Dienstleister die Einhaltung der Vorbedingungen im Rumpf der implementierenden Methode mit assert-Anweisungen. Die eigenen Nachbedingungen werden *nicht* überprüft.

2.1.1 Die assert-Anweisungen eines Java-Programms werden nur dann ausgeführt, wenn man die JVM entsprechend konfiguriert; ansonsten werden sie ignoriert. Damit die Vorbedingungen auch wirklich vom Dienstleister überprüft werden, müssen wir Assertions in der Mediathek aktivieren. Lasst alle im Projekt enthaltenen Tests durchlaufen, indem ihr im Kontextmenü des `src`-Ordners *Run As->JUnit Test* auswählt. Einer der enthaltenen Tests prüft, ob Assertions aktiviert sind; er wird vermutlich fehlschlagen. Geht zu der Stelle im Quelltext des fehlgeschlagenen Tests und folgt der Anleitung in den Implementationskommentaren, um Assertions zu aktivieren. Führt die Tests erneut aus. Sie sollten jetzt alle durchlaufen. 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!

 2.1.2 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 zu finden.

2.1.3 Schaut euch die im Interface `VerleihService` deklarierten Vorbedingungen an. Diese werden noch nicht von der implementierenden Klasse `VerleihServiceImpl` geprüft. Dies müssen wir ändern! Arbeitet hierfür zuerst das Infoblatt *Anlegen von Templates in Eclipse* durch. Jetzt könnt ihr alle fehlenden assert-Anweisungen einfügen. Erklärt euren BetreuerInnen an konkreten Beispielen aus dem `VerleihService`, warum die gemachten Vorbedingungen sinnvoll sind.

2.1.4 Das Interface `MedienbestandService` besitzt noch keine Vor- und Nachbedingungen. Schreibt sinnvolle Vertragsbedingungen (nicht nur, dass ein Parameter `!= null` sein soll) in die

Kommentare und ergänzt danach die benötigten assert-Anweisungen in der implementierenden Klasse `MedienbestandServiceImpl`.

Aufgabe 2.2 Debugger verwenden

Vergangene 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 `AusleihWerkzeug`. Setzt in die erste Zeile der Methode `leiheAusgewahlteMedienAus` 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 3 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 Programmzeile 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` gibt es bereits eine Testklasse, die den Fehler aus Aufgabe 2.2 jedoch nicht findet! In einem solchen Fall wird zuerst ein Testfall geschrieben, der den Fehler aufdeckt, und danach wird der Fehler in der getesteten Klasse behoben.

- 2.3.1 Ergänzt die Testklasse `VerleihServiceImplTest` um einen passenden Testfall, der den gefundenen Fehler aufdeckt.
- 2.3.2 Nachdem jetzt ein Test existiert, der den Fehler findet, könnt ihr diesen in `VerleihServiceImpl` beheben.
- 2.3.3 Schreibt mindestens einen weiteren Testfall.

Aufgabe 2.4 Zahlen in Zeichenketten umwandeln, und was dabei alles schief gehen kann

Die Klassenmethode `intToString` der Klasse `Converter` enthält absichtlich mehrere Fehler:

```
class Converter
{
    public static String intToString(int x)
    {
        StringBuilder sb = new StringBuilder();

        String sign = "";
        if (x < 0)
        {
            sign = "-";
            x = -x;
        }

        while (x != 0)
        {
            sb.append((char)('0' + x % 10));
            x = x / 10;
        }

        return sb.insert(0, sign).toString();
    }
}
```

- 2.4.1 Schaut euch die Methode an und diskutiert die Funktionsweise mit eurem Laborpartner. Spielt die Methode im Kopf mit einer beliebigen Zahl durch, zum Beispiel mit der 747 (yay Boeing).
 - 2.4.2 Erstellt ein neues Projekt in Eclipse. Legt darin eine neue Klasse an und tippt den Quelltext ab.
 - 2.4.3 Ergänzt die Klasse um eine `main`-Methode, welche die `intToString`-Methode mit einer Handvoll Werte aufruft und das Ergebnis jeweils auf die Konsole schreibt. Spätestens jetzt sollte euch mindestens ein ganz offensichtlicher Fehler auffallen.
 - 2.4.4 Erstellt eine Testklasse und schreibt darin Testfälle, die eure gefundenen Fehler aufdecken.
 - 2.4.5 Bei welchen eurer Testfälle handelt es sich um Positivtests bzw. Negativtests?
 - 2.4.6 *Zusatzaufgabe:* Behebt die gefundenen Fehler.
-