Institut für Theoretische Informatik
Logik und Formale Methoden
Prof. Dr. B. Beckert, Prof. Dr. P. H. Schmitt, D. Bruns, D. Faragó, Dr. C. Gladisch, C. Scheben, M. Ulbrich

30.10.2012

Praktikum
Formale Entwicklung objektorientierter Software
Übungsblatt 2: RAC und Testen

Bevor Sie in den folgenden Übungsblättern die Konformität zwischen Quelltext und JML-Spezifikation durch statische Analyse überprüfen, wird in diesem Übungsblatt getestet, der Quelltext zur Untersuchung also ausgeführt.

Aufgabe 6 — Einstiegsaufgabe zu RAC

Um zur Laufzeit zu überprüfen, ob die Ausführung die JML-Spezifikation befolgt, kann der JML Runtime Assertion Checker (RAC) benutzt werden. RAC erzeugt mit Hilfe des Compilers `jmlc` Bytecode, der mit Assertions für die JML-Spezifikationen angereichert ist. Der Bytecode kann dann am einfachsten mit `jmlrac` ausgeführt werden.

(a) Benutzen Sie RAC für Ihre Lösung `Klausuren.java` aus Aufgabe 3. Falls Sie in Aufgabe 3 (b) schon eine `main`-Methode haben, benutzen Sie diese. Falls nicht, so implementieren Sie eine und bauen in einer anderen Klasse einen Fehler ein, so dass eine Spezifikation aus Aufgabe 3 (a) verletzt wird.

(b) Zeigt die Fehlermeldung von RAC, dass die Implementierung Ihre Spezifikation nicht erfüllt? Erläutern Sie die Ausgabe von RAC. Zeilennummern beziehen sich dabei nicht auf den ursprünglichen Quelltext, sondern auf eine interne Variante, die um Assertions angereichert wurde. Diese können Sie mit `jmlc -P` erzeugen. Eine grobe Beschreibung von RAC finden Sie im zweiten Kapitel des JMLUnit-Papers auf der Praktikums-Webseite.

Beachten Sie, dass die im Praktikum benutzte Version von RAC (aus JML5.5) Spezifikationen mit bestimmten Allquantoren (z.B. Summe und Produkt) ignoriert. Kommentieren Sie deswegen die JML-Spezifikationsfälle mit diesen Allquantoren aus, damit RAC nicht den kompletten JML-Kontrakt ignoriert.

Abgabe: Ihre Erklärung zu dem Fehler, der in der `main`-Methode aus Aufgabe 6 (a) auftritt, anhand der Ausgabe von RAC.

Aufgabe 7 — Arbeitsprozess mit RAC

Häufig sind sowohl in der Implementierung als auch in der Spezifikation mehrere Fehler vorhanden, so dass mehrere Zyklen bis zum gewünschten Ergebnis nötig sind. Finden Sie Fehler mit RAC und beseitigen Sie diese. Benutzen Sie dazu die Methode `main` der Klasse Klausuren.java von der Praktikums-Webseite für dieses Übungsblatt, zusammen mit Ihrer Lösung aus Aufgabe 3. Am Ende soll die `main`-Methode fehlerfrei mit RAC durchlaufen. Bei der Fehlerbehebung sollen sowohl Implementierungs- als auch Spezifikations-Fehler korrigiert und ggf. JML-Spezifikationen ergänzt werden (z.B. `nullable` statt der impliziten Spezifikation

non_null). Implementierungen und Spezifikationen dürfen aber nicht so stark vereinfacht werden, dass die ursprüngliche Intention nicht mehr gilt.

Abgabe:

- Ihre korrigierte Datei `Klausuren.java`.
- Eine knappe Beschreibung Ihrer Vorgehensweise mit den aufgetretenen RAC-Fehlermeldungen und ihren entsprechenden Korrekturen.

Aufgabe 8 — Einstiegsaufgabe zu JMLUnit

In den Aufgaben 6 und 7 wurde RAC als Testorakel verwendet, d.h. um die Korrektheit der Ausführung zu überprüfen. Die `main`-Methoden dienten als Testtreiber, d.h. zum Aufruf der zu untersuchenden Methoden und zur Lieferung der dafür notwendigen Daten.

Mit JMLUnit muss der Testtreiber nicht mehr manuell implementiert werden, der Entwickler muss sich nur noch um die Testdaten kümmern, indem er für jeden notwendigen Typ genügend Instanzen generiert.

Dafür setzt JMLUnit mit Hilfe von junit[1] auf RAC auf: JMLUnit hilft bei der Organisation und dem Kombinieren der Testdaten, junit bei der Testausführung; RAC fungiert nachwievor als Testorakel, überprüft also ob ein Testfall erfolgreich oder fehlerhaft ist (oder nutzlos wegen geworfener JMLEntryPreconditionError).

Nutzen Sie JMLUnit, um die Klasse `Student` aus Aufgabe 3 zu testen.

(a) Speichern Sie einfachheitshalber alle Ihre Klassen aus Aufgabe 3 in einzelnen, gleichnamigen Dateien. Kompilieren Sie nun die Klasse `Student` mittels `jmlc Student.java`. Danach können Sie mit `jmlunit Student.java` zwei Dateien erzeugen:

- `Student_JML_Test.java`, dem konkreten Testtreiber, der alle mit JML spezifizierten Methoden von `Student.java` testet,
- deren Oberklasse `Student_JML_TestData.java`, die Generatoren für die Testdaten und Testfälle enthält. Falls diese Datei schon existiert, wird sie nicht überschrieben.

(b) Editieren Sie nun die Datei `Student_JML_TestDaten.java`, um für alle Typen einige Werte hinzuzufügen. Für den Typ `Student` z.B.:

```
protected Object make(int n) {
  switch (n) {
    //replace this comment with test data if desired:
    case 0:
      return new Student(...);
    case 1:
    {
       Klausur k = new Klausur(...);
       Student studLinks = new Student(k);
       Student studRechts = new Student(k);
       Student studMitte = new Student(k);
       studMitte.setzeNachbarn(studLinks, studRechts);
       /*hier wird nicht setzeNachbarn() getestet, sondern
         waehrend der Testdaten-Generierung benutzt */
       return studMitte;
    }
    ...
    default:
      break;
```

[1] Eine Kurzeinführung finden Sie auf der Praktikums-Webseite, nur der junit-Teil ist relevant.

```
  }
  throw new java.util.NoSuchElementException();
}
```

Bei der Testdaten-Generierung sollten also keine Exceptions geworfen werden. Zur Testdaten- und Testfall-Generierung stehen auch fortgeschrittenere Methoden zur Verfügung, z.B. Fabrikmethoden und die Differenzierung der Argumentstellen. Weitere Erklärungen finden Sie als Kommentare in der Datei `Student_JML_TestData.java` selbst. Details sind im JMLUnit-Paper (siehe Praktikums-Webseite) beschrieben, insbesondere auf Seite 12 und in Kapitel 5.1.

(c) Führen Sie danach `javac Student_JML_Test*.java` aus (bei Rechnern nicht aus dem Poolraum ggf. `CLASSPATH` erweitern, Beispiel siehe Praktikums-Webseite). Nun können Sie die Tests mit `jmlrac Student_JML_Test` (oder mit `jml-junit Student_JML_Test`) ausführen. Wieviele Tests sind erfolgreich, wieviele fehlerhaft?

Alle Kombinationen der generierten Werte werden für die Methodenparameter und die zu testenden Objekte durchprobiert. Bei sinnlosen Werten sollte die Vorbedingung verletzt sein, so dass das Ergebnis des Testfalls nicht fehlerhaft sondern nutzlos (meaningless) ist.

Abgabe: Ihre Erläuterungen sowie modifizierte `Student_JML_TestDaten.java`.

Aufgabe 9 — Testabdeckung mit JMLUnit

Gehen Sie nun wie in Aufgabe 8 für alle relevanten Klassen vor: Erweitern Sie die Klassen `*_JML_TestDaten.java` so, dass möglichst viele Fälle der Methoden abgedeckt werden. Schreiben Sie keine Testmethoden (`test`X) selbst, sondern nutzen Sie wirklich RAC und die Testdaten-Generatoren.

Die in Aufgabe 7 gefundenen Fehler sollen auch mit den JMLUnit-Tests entdeckt werden. Untersuchen Sie dazu Ihre JMLUnit-Tests per Fehlerinjektion: Bauen Sie alle Fehler aus Aufgabe 7 jeweils einzeln wieder ein und kontrollieren Sie, dass diese entdeckt werden.

Beheben Sie auch alle neuen Fehler, die Sie im Laufe der Benutzung von JMLUnit entdecken. Wie viele waren es? Wie viele Tests werden letztendlich erfolgreich ausgeführt?

Abgabe:

- Ihre modifizierten `*_JML_TestDaten.java`-Dateien.
- Eine knappe Beschreibung Ihrer Vorgehensweise mit den neu aufgetretenen Fehlern und ihren entsprechenden Korrekturen.

Aufgabe 10 — Testabdeckung mit Jet (freiwillige Zusatzaufgabe)

Statt JMLUnit können auch alternative Testverfahren verwendet werden. Damit nicht einmal Testdaten selbst erstellen zu müssen, kann randomisiert getestet werden.

Testen Sie Ihre Klassen mit Hilfe des Werkzeuges Jet, siehe `http://www.cs.utep.edu/cheon/download/jet/index.php`.

Anders als bei den übrigen Werkzeugen sind bei Jet Array-Elemente standardmäßig `nullable` (und nicht `non_null` gdw. das Array selbst `non_null` ist). Erweitern Sie deswegen Ihre Verträge in den Fällen, wo dieser Unterschied zum Tragen kommt.

Vergleichen Sie nun die Testabdeckung mittels JMLUnit in Aufgabe 9 mit der Testabdeckung mittels Jet. Findet Jet Fehler, die bisher unentdeckt blieben? Injizieren Sie einen Fehler in Ihren Quelltext, der von JMLUnit, jedoch nicht von Jet erkannt wird.

Abgabe:

- Die für `non_null` Array-Elemente erweiterten Verträge.
- Falls Jet einen neuen Fehler entdeckt hat: Eine knappe Beschreibung des Fehlers und wie Sie ihn behoben haben.
- Eine Gegenüberstellung der Anzahl erfolgreicher bzw. nutzloser Tests zwischen JML-Unit und Jet, und ein Fazit zur jeweiligen Testabdeckung.
- Den Quelltext zu Ihrer Fehlerinjektion.

Aufgabe 11 — Alternative Frameworks für Java (freiwillige Zusatzaufgabe)

Es gibt viele Frameworks, die Java um Design-By-Contract erweitern. Contract4J, Java Contracts (JC), JASS und Modern Jass[14] werden leider nicht mehr weiter entwickelt. Zwei aktuelle Frameworks sind

- cofoja: `http://cofoja.googlecode.com` (insbesondere `http://cofoja.googlecode.com/files/cofoja-20110112.pdf`)
- C4J: `http://c4j.vksi.de`.

Vergleichen Sie die softwaretechnischen Fähigkeiten von JML zum Spezifizieren und Testen von Verträgen mit denen von cofoja und C4J. Da diese beiden Frameworks unsere JML-Spezifikationen nicht verstehen, soll hier nur grob über den Tellerrand geschaut werden.

Abgabe: Nennen Sie je einen Vorteil von JML, cofoja und C4J im Vergleich zu den beiden anderen.

Abgabe bis Dienstag, 20.11.2012

Abgabe (als Java-, ASCII- oder PDF-Dateien, oder tar-Archiv solcher) per E-Mail an David Faragó. Es braucht pro Gruppe und Aufgabe nur *eine* Lösung abgegeben werden. Bitte dokumentieren Sie Ihre Lösungen ausreichend und seien Sie darauf vorbereitet, sie auf Nachfrage zu erklären.

Praktikums-Webseite: `http://formal.iti.kit.edu/teaching/keypraktWS1213/`

Daniel Bruns:	R. 223,	Tel. 608-45268,	E-Mail: `bruns@kit.edu`
David Faragó:	R. 308,	Tel. 608-47322,	E-Mail: `farago@ira.uka.de`
Christoph Gladisch:	R. 223,	Tel. 608-45268,	E-Mail: `gladisch@ira.uka.de`
Christoph Scheben:	R. 106,	Tel. 608-44338,	E-Mail: `scheben@ira.uka.de`
Mattias Ulbrich:	R. 106,	Tel. 608-44338,	E-Mail: `mulbrich@ira.uka.de`