

Übungen zu Grundlagen der Softwarezuverlässigkeit

Abgabe bis zum Mittwoch, 10.11.2004

Gegenstand dieses Übungsblatts sind die in der Vorlesung vorgestellten Verfahren für das Black- und White-Box Testing. Für die Implementierung von Test-Suites wird die Java-Bibliothek JUnit verwendet werden. Die Coverage-Analyse geschieht mit Hilfe von JCoverage. Zu beiden Tools finden Sie die entsprechenden Links auf der Webpage der Übung.

Schicken Sie Ihre jeweiligen Testfälle (d.h. *ConvertDoubleToString.java* für A2.1, *Clipper-InterfaceTest.java* für A2.2 und die von Ihnen für Aufgabe 2.4 erstellte *java*-Datei) wieder an luttenml@fmi.uni-stuttgart.de.

Aufgabe 2.1 **Black-Box Testing**

Auf der Vorlesungsseite finden Sie wieder ein Zip-Archiv. Dieses enthält das Verzeichnis *ue02_a2.1_convert_bb*, in welchem Sie Implementierungen der folgenden Funktion in Form eines JARs finden:

<i>Parameter</i>	<i>Beschreibung</i>
1. <i>to_convert</i>	Eine Fließkommazahl vom Typ <i>double</i> , aber nicht <i>NaN</i> oder $\pm\infty$.
2. <i>radix</i>	Eine ganze Zahl vom Typ <i>int</i> aus dem Intervall $[2, 10] \cap \mathbb{N}$.
3. <i>n_decimal_places</i>	Eine nicht-negative ganze Zahl vom Typ <i>int</i> .
Rückgabe	Ein <i>String</i> , welcher die Zahl <i>to_convert</i> in der Basis <i>radix</i> mit genau <i>n_decimal_places</i> Nachkommastellen repräsentiert. Hierbei sind die Nachkommastellen durch '.' von dem Ganzzahlanteil getrennt für <i>n_decimal_places</i> größer 0. Kann <i>to_convert</i> nicht exakt bezüglich <i>base</i> und <i>n_decimal_places</i> dargestellt werden, so soll zur nächsten darstellbaren Zahl gerundet werden: $0.25 \mapsto 0.1$, $0.2 \mapsto 0.0$ für <i>radix</i> = 2, <i>n_decimal_places</i> = 1. Entsprechen die übergebenen Parameterwerte nicht obiger Spezifikation, so soll eine <i>Exception</i> vom Typ <i>IllegalArgumentException</i> ausgelöst werden.

Ihre Aufgabe ist es nun entsprechend der Vorlesung, Testfälle für diese Funktion zu entwerfen, mittels JUnit umzusetzen und hiermit die gegebenen Implementierungen zu testen (Genauere Informationen entnehmen Sie bitte der *README* im Archiv).

Geben Sie weiterhin die Kriterien an, nach welchen Sie die Äquivalenzklassen für die Testfälle gewählt haben.

Aufgabe 2.2 White-Box Testing - JUnit - JCoverage

In dieser Aufgabe soll das *white-box testing* auf Aufgabe 1.1 des letzten Blattes angewandt werden.

In dem aktuellen Zip-Archiv finden Sie hierfür das Verzeichnis *ue02_a2.2_clipper_wb*, welches mehrere Implementierungen des Clippers aus Aufgabe 1.1 enthält.

Mit Hilfe des *JCoverage* sollen Sie nun eine Testsuite erstellen, welche in jeder der mitgelieferten Implementierungen 100% Branch- und Statement-Coverage erreicht.

Nähere Informationen über die Softwarewerkzeuge finden Sie wieder in der *README* und auf der Webpage zu den Übungen.

Aufgabe 2.3 Modified Condition/Decision Coverage

Aus der Vorlesung wissen Sie, dass nach *DO-178B* der höchste Sicherheitsstandard verlangt, dass ein Test die sogenannte *Modified Condition/Decision Coverage (MC/DC)* erfüllt.

Dies bedeutet, dass für jedes Atom der betrachteten Bedingung ein Paar von Testfällen gefunden werden muss, so dass sich die Testfälle einzig im Wahrheitswert des betrachteten Atoms unterscheiden und sich die Bedingung einmal zu *true*, einmal zu *false* auswertet.

Als Beispiel betrachte man $A \wedge (B \vee C)$. Für die Variable A wäre ein solches Paar von Belegungen (Testfällen) durch $\{(0, 1, 0), (1, 1, 0)\}$ gegeben. Entsprechend für Variable B durch $\{(1, 0, 0), (1, 1, 0)\}$ und für C durch $\{(1, 0, 0), (1, 0, 1)\}$. Damit erreichen folgende Belegungen $\{(0, 1, 0), (1, 1, 0), (1, 0, 0), (1, 0, 1)\}$ bereits 100% MC/DC.

Zeigen Sie, dass allgemein für eine aussagenlogische Formel mit n aussagenlogischen Variablen mindestens $n + 1$, höchstens $2n$ Testfälle benötigt werden, damit der Test zu 100% der MC/DC-Bedingung genügt.

Geben Sie für das Beispiel auf Folie 71 entsprechend Testfälle an, welche 100% MC/DC erreichen.

Aufgabe 2.4 Test bei unbekanntem Ergebnis

In vielen Fällen kann es aufwendig bis unmöglich sein, das *korrekte* Ergebnis für einen Testfall vorherzusagen. Z.B. könnte Ihnen ein neuer Algorithmus/Implementationen versprechen, zuvor auf Grund des benötigten Speichers und/oder Rechenaufwands nicht lösbare Probleme bewältigen zu können (vgl. $P \stackrel{?}{=} NP$). In anderen Fällen kann das exakte Ergebnis unbekannt sein und nur durch bereits existierende Software approximativ bestimmt werden ('Bsp': π).

Als sehr einfaches Beispiel sollen Sie in dieser Aufgabe daher die Klasse *BigInteger* bzw. eine Funktion zur Berechnung beliebiger Fakultäten 'testen'. *BigInteger* erlaubt dabei - soweit der Speicher es erlaubt - beliebige Zahlen aus \mathbb{Z} darzustellen und mit diesen zu rechnen.

Eine Implementierung der Fakultätsfunktion finden Sie im Verzeichnis *ue02_a2.4_BigIntFactorial*. Überlegen Sie sich, wie Sie ohne Verwendung von *BigInteger* nur unter Verwendung von Operationen und Datentypen, in welche Sie *genügend Vertrauen* besitzen, die Fakultätsfunktion auf (einige) Fehler testen können.

Hinweise

- Mit *long* kann maximal $2^{63} - 1 \in [9.2233 \cdot 10^{18}, 9.2234 \cdot 10^{18}]$ dargestellt werden.
- Es gilt jedoch bereits $21! = 51090942171709440000 \in [5.109, 5.1091] \cdot 10^{19}$.
- *double* kann hingegen bis zu

$$\left(\underbrace{1.1 \dots 1}_{52} \right)_2 \cdot 2^{1023} = \sum_{i=0}^{52} 2^{-i} \cdot 2^{1023} = \frac{1 - 2^{-53}}{1 - 2^{-1}} \cdot 2^{1023} = 2^{1024} - 2^{971} \geq 1.79 \cdot 10^{308}$$

darstellen.

Allgemeine Hinweise

- Vorlesungswebseite:
<http://www.fmi.uni-stuttgart.de/szs/teaching/ws0405/grundsoftzuv/>
- Fragen zu den Übungen oder dem Stoff der Vorlesung können Sie im *Info-Forum* im Bereich *Sichere und Zuverlässige Software* posten:
<http://swt.uni-stuttgart.de/forum>
- Für sonstige Fragen wenden Sie sich bitte an:
Michael.Luttenberger@fmi.uni-stuttgart.de, Raum 1.342.