TU Wien:Software-Qualitätssicherung VU (Biffl)/Prüfung 2013-05-29

< TU Wien:Software-Qualitätssicherung VU (Biffl)

Hinweis: ich habe mir die Fragen nicht speziell gemerkt, sondern schreibe hier nur, was mir eben noch einfällt. Die Fragen könnten auch anders gelautet haben.

Contents

Theorie

Nennen Sie Vorgehensweisen zu Sicherung/Verbesserung der Qualität von Projekt-Anforderungen? (4 Punkte)

Projektanforderungen werden oft im Zuge eines *Project Requirement Review* überprüft. Grundsätzlich ist es sinnvoll, in dieser Phase großzügig zu testen, kontrollieren und prüfen, da frühe Fehler am günstigsten zu beheben sind. Es kann zum Beispiel mit Hilfe von Checklisten überprüft werden, ob für alle Use-Cases entsprechende Parameter angegeben sind.

Was ist der Unterschied zwischen Testing und Debugging? (2 Punkte)

Testing: Wird von einem Tester durchgeführt, um ein System bzw. eine Komponente auf Fehler zu prüfen. Entscheidend ist hier nur das Finden von Fehlern, nicht deren genaue Ursache.
Debugging: Wird vom Entwickler durchgeführt, um die Quelle eines Fehlers zu lokalisieren.

Was ist der Unterschied zwischen Äquivalenzklassenzerlegung und Grenzwertanalyse? (4 Punkte)

Äquivalenzklassenzerlegung: Befasst sich damit, mögliche Eingaben basierend auf dem erwarteten Systemverhalten in Klassen gleichen Verhaltens aufzuteilen. Beispiel: Beschränkung eines YouTube-Videos durch Volljährigkeitsprüfung - zwei Äquivalenzklassen: <18 und >=18
Grenzwertanalyse: Ist eine Vertiefung der Äquivalenzklassenzerlegung, bei der gezielt die Ränder (= Grenzwerte) von den einzelnen Klassen überprüft werden, da dort am häufigsten Fehler (vgl. falscher Vergleichsoperator, oder Index one-off Fehler) auftreten.

Eine Methode string10(String s) retourniert die ersten 10 Zeichen des Strings s. Ist der String kürzer als 10, so wird der String unverändert zurückgegeben. Wie viele Äquivalenzklassen werden mindestens benötigt, um die Methode vollständig zu testen. (4 Punkte)

Es sollten zumindest Tests mit weniger als 10 (bspw. 9), genau 10 und mehr als 10 (bspw. 11) Zeichen durchgeführt werden. Idealerweise werden auch noch Tests mit einem leeren String, sowie einem null-Parameter durchgeführt. D.h. 3 (+2) Äquivalenzklassen sind notwendig.

Nennen Sie 3 Prinzipien des Qualitätsmanagements und erklären Sie diese. Was ist der Unterschied zwischen Qualitätsmanagement und Qualitätssicherung? (6 Punkte)

Qualitätsmanagement beschreibt die Gesamtheit aller Maßnahmen und Aktivitäten, die zur Herstellung und Wahrung von Qualität notwendig sind. Dazu zählen unter anderem Qualitätsplanung, -lenkung, -sicherung und insbesondere die Qualitätsverbesserung. Wichtige Prinzipien:

- Kontinuierliche Verbesserung durch Bewertung von Maßnahmen und Prozessen.
- Frühzeitige Entdeckung und Behebung von Fehlern
- Unabhängigkeit bei Qualitätsprüfungen (vgl. QS-Abteilung getrennt von Entwicklung)
- Produkt- und projektabhängige Planung von QS-Maßnahmen

Qualitätssicherung hingegen beschreibt Verfahren, Aktivitäten und Prozesse, welche sicherstellen, dass ein Produkt gegebenen Anforderungen (Spezifikation und Kundenanforderungen) gerecht wird (kurz: Testen, Reviewen, ...).

Erklären Sie den Begriff "Continuous Integration". Nennen Sie die Aufgaben des CI-Systems. Welche davon können automatisiert werden? (6 Punkte)

Kontinuierliche Integration beschreibt ein System, das bei Änderungen (= neue Version im Versionsverwaltungssystem) automatisch Tests durchführt und ein Softwareprodukt zusammenführt. Dadurch wird gewährleistet, dass Fehler früh erkannt und behandelt werden können. Kurzum ist ein CI-System also dafür zuständig, bei Änderungen die Funktionalität einer neuen Version sicherzustellen. Im Idealfall übernimmt das CI-System auch entsprechende Dokumentationsarbeiten (Test-Berichte u.Ä.).

Was versteht man unter dem Begriff Refactoring? Wâs wird dadurch erreicht? (4 Punkte)

Refaktorisierung ist die Umgestaltung (= Umstrukturierung) von Programmcode, ohne die Programmfunktionalität dabei zu verändern, mit den Zielen, die Lesbarkeit, Verständlichkeit, Wartbarkeit und Erweiterbarkeit des Codes zu verbessern. Refaktorisierung wird auch zur Vermeidung von Redundanzen (z.B. mittels Ersetzung durch generischer Klassen) eingesetzt.

Praxis

[diese Angaben sind nicht 1:1 die Angaben vom Test, sondern nur eine Rekonstruktion von dem, was mir noch im Kopf hängen geblieben ist]

1. Beispiel: Äquivalenzklassenzerlegung

Gegeben war eine Funktion berechneRabatt(String land, int totalPriceInCent). Normalerweise wird ab einem Bestellwert von 20€ ein Rabatt von 5% gewährt. Erfolg die Bestellung aus Österreich, wird IMMER ein Rabatt von 5% gewährt (auch bei < 20€). Stellen Sie die Äquivalenzklassen für diese Funktion auf und definieren Sie zwei wichtige Testfälle (NICHT JUnit).

2. Beispiel: Zuweisung von Anforderungen auf Testarten

Gegeben waren diverse Anforderungen (~7 kurze Anforderungen, jeweils 1 Satz), die mit einer Verbindungslinie auf die entsprechenden Testarten gemappt werden sollten (Testarten: Unittest, Systemtest, Regressionstest, Usabilitytest, Performancetest, Lastentest, ...).

3. Beispiel: Mocking-Beispiel

Gegeben war ein Code-Snippet mit einem TimeService-Interface, das gemockt werden musste und mittels dieses Mocks mussten 2 JUnit-Tests einer gegebenen Funktion erstellt werden.

4. Beispiel: JUnit-Tests

Hier musste man 2 JUnit-Tests für einen Iterator schreiben. Die Namen der Tests waren bereits gegeben und daher wusste man schon was verlangt ist. Tests waren sowas wie: test_shouldReturnNextElement() und test_shouldThrowExceptionWhenNextElementIsNotAvailable() -> der 2. Test hieß sicher nicht so, aber ich weiß nicht wie ichs besser benennen soll xD

Hier ist extrem darauf zu achten GENAU zu testen.. Ich war da sehr ungenau und hab die Dinge sehr falsch getestet. Dachte beispielsweise, dass assertEquals("foo", it.next()); beim 1. Test genügt, ("foo" wähle ich statt einer fixen Konstante, die in setUp in die zu testende Liste eingetragen wurde). Gewollt war jedoch, dass man den gesamten Inhalt der Liste überprüft. (in setUp() wurden 2 Elemente der Liste geaddet -> zuerst "foo" dann "bar") sowas wie

assertEquals("foo", it.next());

assertEquals("bar", it.next());

wäre also gewünscht gewesen.

Beim 2. Test muss man besonders aufpassen. Da hab' ich auch gefailt. Man musste so lange next() aufrufen, bis eine Exception geworfen wird. Ich dacht mir dabei, oh, easy.

@Test (expected = Exception.class)

public void test_shouldThrowExceptionWhenNextElementIsNotAvailable() {

it.next();

it.next();

it.next();

}

FALSCH!

Dadurch weiß man nämlich nicht wo die Exception geworfen wurde. Es kann genauso gut sein, dass das erste it.next() bereits die Exception geworfen hat, was den Testfall unsinnig macht.

Gewünscht wäre gewesen:

@Test (expected = Exception.class)

public void test_shouldThrowExceptionWhenNextElementIsNotAvailable() throws Exception {

```
try {
it.next();
it.next();
} catch(Exception e) {
fail("next()-Aufruf fehlgeschlagen.");
}
it.next(); // Dieser next()-Aufruf sollte die Exception auslösen
}
```

Dieses Beispiel hat 10/30 Punkte gegeben, die Gewichtung beim Test lag also SEHR auf JUnit-Testing. Bei der Einsichtnahme meinten die Tutoren ebenfalls, dass sehr viel Wert darauf gelegt wird, dass die Studenten JUnit-Tests verstehen und anwenden können.

5. Beispiel: Test-Refactoring

Hier waren JUnit-Tests gegeben, die ein Queue-Objekt testeten. Worauf bei solchen Beispielen besonders zu achten ist:

- korrekte Benennung der Testfälle

- assertEquals ist assertTrue vorzuziehen

- ist eine spezifische Exception-Klasse bekannt, so sollte diese expected werden, nicht die Exception-Oberklasse

- eventuell fehlende throws-Annotationen

- while(true) vermeiden (war bei uns vorhanden)

- ...

Praxis für Mutige (Retake)

1. Beispiel: Äquivalenzklassenzerlegung

"Im gregorianischen Kalender ist jenes Jahr ein Schaltjahr, das durch 4 teilbar ist, jedoch nicht durch 100. Ausnahme dabei sind die Jahre, die durch 400 teilbar sind. Erstellen Sie die Äquivalenzklassen für die Funktion boolean isLeapYear(int jahr) und schreiben Sie 2 Testfälle, die aufgrund der Äquivalenzklassenzerlegung sinnvoll sind." (kein JUnit - lediglich Parameter angeben -> Grenzwerte)

2. Beispiel: Zweigabdeckung

Dabei war ein Code-Snippet gegeben mit Variablen a, b, c und man musste sich Testfälle überlegen, mit denen eine vollständige c1-Abdeckung (Zweigabdeckung) gegeben war. Dabei reicht es die Werte, die a, b und c dabei annehmen, hinzuschreiben. (z.B: Testcase 1: a > 0 (z.B. 1), b < 0 (z.B. -1), c egal)

3. Beispiel: Mocking/Stubbing

Es war wieder ein Code-Snippet gegeben. Dabei gab es ein Interface Dice, das die Methode roll() definierte und die Klasse DiceGame, das eine Methode int play(int coins) besaß. Wenn der Spieler 5 gewürfelt hat, bekommt er das 5-fache das Einsatzes zurück und bei 6 das 6-fache. Ansonsten verliert der Spieler alle Münzen. Nun war die Aufgabe das Interface Dice zu mocken oder zu stubben und zwei Testfälle zu schreiben. 1. Test: Spieler würfelt einen 6er mit 4 Coins und 2. Test Spieler würfelt einen 2er mit 5 Coins.

4. Beispiel: JUnit-Testing

Hierbei war die Definition eines Stacks gegeben und man musste die Tests pop_shouldRemoveFirstElement() und pop_whenStackIsEmpty() implementiert werden.

Korrekte Lösung:

```
@Test
public void pop_shouldRemoveElement() {
try {
assertEquals(ELEM2, stack.pop());
assertEquals(ELEM1, stack.pop());
} catch(Exception e) {
fail("irgeneine Message");
}
@Test (expected = StackEmptyException.class)
public void pop_whenStackIsEmpty() {
try {
while (!stack.isEmpty()) {
stack.pop();
} catch(Exception e) {
fail("meeh");
```

}

stack.pop(); //diese Zeile soll erst Exception auslösen

}

5. Beispiel: JUnit-Test-Refactoring

Gegeben waren 3 Tests, die diverse Fehler beinhalteten und darunter eine Tabelle, wo die gefundenen Fehler (5 Fehler waren zu finden) dokumentiert werden sollten.

Dabei ist wieder auf die selben Dinge zu achten, die ich bereits beim 1. Praxistest erläutert habe.

Peace.
