Einführung in den Softwaretest

Seminar „Simulation und Bildanalyse mit Java“ SS2004

Themenschwerpunkt: Tests in Informatik und Statistik

Arbeit von Christian Aich und Robert Reeb

Thema 2: Einführung in den Softwaretest

Einführung in den Softwaretest

- Was ist ein Testfall?
- Methoden der Testfallfindung
  - Black-Box-Test
  - White-Box-Test
- Testauswertung
- Software-Metriken
- Testende-Kriterien
- Alternativen zum Software-Test: „Manuelles“ Testen

Was ist ein Testfall?

Falsche Definitionen des Testens:

- „Testen ist der Prozess, der zeigen soll, dass keine Fehler vorhanden sind“
- „Der Zweck des Testens ist es zu zeigen, dass ein Programm die geforderten Funktionen korrekt ausführt.
- „Testen ist der Prozess, der das Vertrauen erzeugt, dass ein Programm das tut, was es soll.

Was ist ein Testfall?

Richtige Definition des Testens:

„Testen ist der Prozess, ein Programm mit der Absicht auszuführen, Fehler zu finden“

Testfall:

- Eingabedaten (nicht unbedingt nur Zahlen)
- Ausgabedaten
- Veränderung von Variablen

Testfallentwicklung

„Welche Untermenge aller denkbaren Testfälle bietet die größte Wahrscheinlichkeit möglichst viele Fehler zu finden?“

Blackboxtest (Funktionstest)

Testfälle aus der Spezifikation abgeleitet

Whiteboxtest (Strukturtest)

Testfälle aus dem Programm heraus abgeleitet, Spezifikation zur Wertung aber nötig!

Testfallentwicklung

Testprinzipien:

- Definition der erwarteten Werte
- eigene Programme testen ist ineffizient
- die Ergebnisse eines jeden Tests gründlich überprüfen
- Testfälle müssen für ungültige und unerwartete ebenso wie für gültige und erwartete Eingabedaten definiert werden
- Ein Programm zu untersuchen, um festzustellen, ob es nicht tut, was es tun sollte, ist nur die eine Hälfte der Aufgabe. Die andere Hälfte besteht darin, zu untersuchen, ob das Programm etwas tut, was es nicht tun soll
- Wegwerftestfälle vermeiden, d.h. die Tests sollten reproduzierbar sein

Testfallentwicklung

Blackbox

- Äquivalenzklassen
- Grenzwertanalyse
- Ursache-Wirkungsgraph
- Fehlererwartung (error guessing)

Whitebox

Erfassung (Ausführung) aller

- Befehle
- Entscheidungen
- Bedingungen
- Entscheidungen/Bedingungen
- Mehrfachbedingungen

Whitebox

Whitebox-Verfahren:

- **Line coverage (Ausführung aller Anweisungen)**
- **Decision oder branch coverage**
- **Condition coverage**
- **Decision/condition coverage**
- **Procedure coverage**

Whitebox

Fehlerhafte Implementierung

```
IF (a > 1.0) AND (*statt OR*) (b = 0.0)
THEN x := x/a;
END;

IF (a = 2.0) OR (*AND*) (x > 1.0) (*x <= 1*)
THEN x := x - 1;
END;
```

Whitebox

Test - Implementierung

Art	*a*	*b*	*x*	*Soll*	*Ist*	*„Weg“*	*Bedingungen*
C0	2	0	1	-0.5	-0.5	ACE	
C1	2	0	1	-0.5	-0.5	ACE	
	1	1	1	1	1	ABD	
	2	0	1	-0.5	-0.5	ACE	TT:TF
	1	1	1	1	1	ABD	FF:FF
	1	0	2	2	1	ABE	FT:FT
	2	0	1	-0.5	-0.5	ACE	TT:TF
	1	1	1	1	1	ABD	FF:FF
	1	0	2	2	1	ABE	FT:FT
	2	2	2	0	1	ABE	TF:TT
C7	1	1	1	1	1	ABD	
	2	2	2	0	1	ABE	
	4	0	2	0.5	0.5	ACD	
	2	0	1	-0.5	-0.5	ACE	

Whitebox

Fazit:

In Programmen mit Einfachentscheidungen ist als minimales Kriterium eine hinreichende Anzahl von Testfällen zu betrachten, für die gilt:

1. Alle Zweige einer Entscheidung werden mindestens einmal durchlaufen

2. Jeder Eingang wird mindestens einmal benutzt

- Aufwand der Testfallermittlung beim White-Box-Test kann sehr hoch sein
- Der Wert kann sehr zweifelhaft sein!
- Es wird nur berücksichtigt, was vorhanden ist – fehlende Statements bleiben unberücksichtigt!

Blackbox

Äquivalenzklassen

- reduziert die Anzahl der Testfälle
- überdeckt eine große Menge anderer möglicher Testfälle

Der Entwurf von Testfällen mit Hilfe der **Äquivalenzklassenbildung** erfolgt in zwei Schritten:

1. Bestimmung der Äquivalenzklasse
2. Definition der Testfälle

Blackbox

Beispiel:

Eingabe des Alters für einen Vertragsabschluss.

Erlaubt sind Eingaben >=0.

Bei Eingabe von

- 0 – 17 muss ein Erziehungsberechtigter herangezogen werden
- 18 – 99 wird der Vertrag direkt abgeschlossen
- > 99 muss das Personal aufgesucht werden

Blackbox

Kl.	*Beschreibung*	*erwarteter Output*
1	Alter zwischen 0 und 17	„Erziehungsberechtigter“
2	Alter zwischen 18 und 99	„Vertragsabschluss“
3	Alter über 99	„Personal aufsuchen“
4	Negatives Alter	„Eingabe fehlerhaft“
5	Text enthält nichtnumerisches Zeichen	„Bitte nur Ziffern eingeben“
6	???	???

Blackbox

Grenzwertanalyse

1. In der Grenzwertanalyse muss jeder Rand einer Äquivalenzklasse in einem Testfall auftreten

2. Die Aufmerksamkeit gilt außerdem nicht nur den Eingabebedingungen (Eingaberaum), sondern es werden auch Testfälle entworfen, die den Ergebnisraum berücksichtigen (d.h. Ausgabeäquivalenzklassen).

Blackbox

Richtlinien der Grenzwertanalyse:

1. Verwendung von Randwerten und ungültigen Werten direkt neben den Rändern

2. Ist die Ein- oder Ausgabe eines Programms eine geordnete Menge (z.B. lineare Liste oder Tabelle), so interessiert uns besonders das erste und letzte Element dieser Menge.

3. Suchen nach nicht offensichtlichen Grenzwerten

Blackbox

Ursache-Wirkungs-Graph

Darstellung der Digitallogik eine Programms durch eine formale Sprache

Aber: Wird sehr schnell unübersichtlich!

Error guessing

Bestimmung der Testfälle aus Erfahrungswerten und Intuition des Testers

Beispiel: Navigationssystem im Auto

Welche Fehler sind zu erwarten?

Blackbox

Fazit:

Selbst bei Kombination aller Techniken hat man keinerlei Garantie dafür, dass alle Fehler entdeckt werden, aber es hat sich herausgestellt, dass das ein vernünftiger Kompromiss ist.

Testauswertung

Vergleich des Testergebnisses mit Erfahrungswerten aus alten Tests durch:

- Fehler / Codezeile
- Erstellung eines Testprotokolls für alle durchgeführten Tests
- Erstellung einer zusammenfassenden Mängelliste
- Falls nötig: Erfahrungsbericht zur Verbesserung zukünftiger Tests

Softwaremetriken

Test Coverage:

- Testfälle werden durch eine Software (z. Bsp. JUnit) getestet

Aber: Nicht nur auf Kennzahlen schielen, Zahlen sind nicht alles!

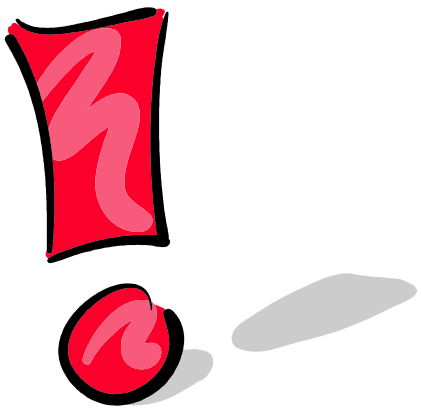
Unterscheidung der Kategorien nicht abgedeckter Codes:

1. Code, der nicht getestet wird, aber getestet werden sollte. Diese Entdeckung weist den größten Nutzen für uns auf.
2. Toter Code, der entfernt werden sollte. Auch das ist sehr nützlich.
3. Automatisch generierter Code, der in unserer Anwendung nicht aufgerufen wird.
4. Code, der nur unter (zu) großem Aufwand in Tests zu erreichen wäre.

Software Metriken

Vorsicht:

Obwohl Coverage-Metriken ein effizientes Werkzeug zum Testen von Software darstellen, darf man nie vergessen, dass sie nur *vorhandenen Code* testen können!



Test-Ende Kriterien

How much is enough?

- Vollständiges Testen ist nicht möglich
- akzeptables Fehlerniveau
- Entwicklungsgeschwindigkeit/Projektlaufzeit
- vom Kunden spezifizierte Akzeptanztests
- zu wenige Tests: Gefühl von falscher Sicherheit

Test-Ende Kriterien

Optimale Testmenge:

- diejenigen Tests, die auch tatsächlich Fehler finden

- Zwei Aspekte:
  - ökonomische Seite

    („was kostet mich welches Fehlerniveau“)
  - technische Seite

    („wie viele Test bringen mir maximale Geschwindigkeit?“)

Praxis:

„Rantasten“ an das richtige Niveau

Test-Ende Kriterien

Aber: oft Anwendung folgender (*falscher*) Regeln:

Abbruch des Testens, wenn

1. die geplante Testzeit abgelaufen ist

2. alle Testfälle ohne Fehler durchgeführt wurden

Test-Ende Kriterien

besseres Kriterium für die Beendigung des Testens:

- erreichen einer bestimmten Anzahl von Fehlern

dazu benötigt man

1. Eine Abschätzung der Gesamtzahl der Fehler im Programm

2. Eine Abschätzung darüber, welcher Anteil dieser Fehler überhaupt gefunden werden kann

3. Schätzungen darüber welcher Anteil an den Fehlern in den einzelnen Testphasen entdeckt werden können

Test-Ende Kriterien

Fehlerabschätzung:

Anwendung von statistischen Methoden:

- Fehlereinpflanzung (error seeding)
  - mehrere Fehler zusammen können ein Fehlverhalten bewirken
  - setzt sehr große Zahl von Fehlern voraus
  - setzt eine gleichmäßige Verteilung der echten Fehler voraus
  - setzt voraus, dass die Verteilung der eingepflanzten Fehler der der echten entspricht
  - setzt voraus, dass echte wie eingepflanzte Fehler mit gleicher Wahrscheinlichkeit gefunden werden
  - setzt auch voraus, dass es keine Wechselwirkung zwischen echten und eingepflanzten Fehlern gibt

Test-Ende Kriterien

Fehlerabschätzung:

- Testen mit zwei unabhängigen Gruppen

  - Zwei unabhängige Gruppen G_1 und G_2 entwickeln jeweils Testdatenmengen T_1 resp. T_2 für Programm P

  - Die Gruppe G_i findet F_i Fehler (i=1,2)

  - F sei die (unbekannte) Anzahl der Fehler in Programm P
    es gelte die Annahme, dass beide Gruppen bei allen Fehlern wie auch bei allen Fehlerteilmengen die gleiche Effizienz haben (gleiche Wahrscheinlichkeit der Fehleraufdeckung)

Manuelles Testen

- Test ohne Computer
- nach Ende der Codierung und vor Beginn des Testens am Computer
- je eher Fehler gefunden werden, desto besser

Techniken:

- Code-Inspektionen
- Walkthroughs
- Schreibtischtests
- Peer Ratings

Manuelles Testen

Code-Inspektionen:

Team versucht durch gemeinsames Lesen des Codes Fehler zu finden

Vorgehen:

1. Der Programmierer / ein Dritter erklärt die Programmlogik Anweisung für Anweisung

2. Das Programm wird mit Hilfe einer Checkliste bekannter Fehler analysiert

Manuelles Testen

Walkthrough:

- Teilnehmer „spielen“ Computer
- Zustand des Programms wird dabei auf dem Papier oder einer Wandtafel festgehalten

Schreibtischtest:

- Ein-Mann-Inspektion /-Walkthrough

Aber: relativ unproduktiv!

Programmierer sollten ihre Programme besser austauschen

Manuelles Testen

Peer Ratings:

Beurteilung von anonymen Programmen

Primäres Ziel ist nicht das Finden von Fehlern, sondern Beurteilung von

- Qualität
- Wartungsfreundlichkeit
- Erweiterbarkeit
- Anwendbarkeit
- Klarheit
- ...

Fazit

Murphy sagt: *„Was schief gehen kann, geht schief!“*

Wir sagen: *„Was nicht getestet ist, läuft nicht!“*

„Testen beginnt am ersten Tag!“

[aus J. Siedersleben, S. 317]

Literatur

G. J. Myers: *Methodisches Testen von Programmen,* Oldenbourg Verlag 1989

J. Link: *Unit Tests mit Java*, dpunkt.verlag, 2002

J. Siedersleben: *Softwaretechnik: Praxiswissen für Sofwareingenieure,*
Carl Hanser Verlag, 2003