Einführung in das systematische Testen

Apcon Workplace Solutions
Member of itelligence

- ➔Motivation
- ➔Korrektheit von Software
- ➔Testen ist Handwerkszeug
- ➔Positives und Negatives Testen
- ➔Äquivalenzklassen und Grenzwerte
- ➔Black-Box-, White-Box- und Schreibtischtests

Henning Wolf
APCON Workplace Solutions GmbH
wolf@jwam.de

Software Testing

Apcon Workplace Solutions
Member of itelligence

Robert Binder:
„*Software Testing is the execution of code using combinations of input and state selected to reveal bugs.”*

Definition IEEE610:
“The process of operating a system or component under specified conditions, observing the results, and making an evaluation of some aspect of the system or component.”

Motivation: Warum Testen?

- Am 4. Juni 1996 explodierte die europäische Raumrakete Ariane 5 ca. 40 Sekunden nach dem Start.
- Die Ursache für die Explosion war ein Programmfehler.
- Geschätzter Verlust durch die Explosion: ca. 1 Milliarde DM.

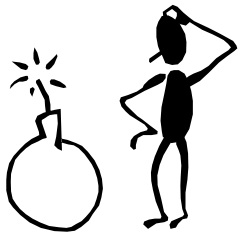
- *Nach Aussage des Untersuchungsberichts hätte der Fehler durch ausführlichere Tests der Steuerautomatik und des gesamten Flugkontrollsystems entdeckt werden können.*
  http://www.esa.int/htdocs/tidc/Press/Press96/ariane5rep.html

➔ Testen kann sich lohnen!

Was ist Testen?

- Testen ist eine Maßnahme zur Qualitätssicherung von Software, mit dem Ziel möglichste fehlerfreie Software zu schreiben.
- Testen dient zum Aufzeigen von Fehlern in Software; es kann *nicht* die Korrektheit der Software zeigen.
- Testen ist damit das geplante und strukturierte Ausführen von Programmcode, um Probleme zu entdecken.

Testen sollte Teil des Entwicklungsprozesses sein!!!



Wann ist Software überhaupt "korrekt"?

Apcon Workplace Solutions
Member of itelligence

- Die *Korrektheit* von Software kann immer nur in Relation zu ihrer Spezifikation gesehen werden - eine Software-Einheit ist korrekt, wenn sie ihre Spezifikation erfüllt.
- Der formale Nachweis, dass eine Software-Einheit ihre Spezifikation erfüllt, ist sehr aufwendig und schwierig und für umfangreichere Programme auch heutzutage noch völlig impraktikabel.
- Voraussetzung für einen formalen Nachweis der Korrektheit ist außerdem, dass die Spezifikation formal definiert ist. Dies ist nur sehr selten der Fall, meist sind Spezifikationen problembedingt nur informell formuliert.
- Selbst wenn eine formale Spezifikation vorliegt: Wie kann nachgewiesen werden, dass die Spezifikation selbst korrekt ist?

Ein Zitat

Apcon Workplace Solutions
Member of itelligence

> *"Beware of bugs in the above code; I have only proven it to be correct, not tested it."*
>
> Donald Knuth

Testen ist Handwerkszeug

- In der Praxis ist Testen nach wie vor das Mittel der Wahl, um die Qualität von Software zu erhöhen.
- Für Software-TechnikerInnen muss Testen zum Handwerkszeug gehören.
- Testen kann zwar keine Korrektheit nachweisen, aber den Eindruck belegen, dass eine Software-Einheit ihre Aufgabe in angemessener Weise erfüllt (" das Vertrauen erhöhen").
- Die Nützlichkeit einer Software kann sich häufig sowieso erst im Gebrauch zeigen.
- Aber: auch Testen hat seine Tücken...

Probleme beim Testen

- Technisch:
  - ➔ Testen ist schwierig (insbesondere bei grafischen Oberflächen)
  - ➔ Testen braucht Zeit (und die ist in den meisten Projekten knapp)
  - ➔ Tests müssen gut vorbereitet sein (Testplan)
  - ➔ Tests müssen wiederholt werden (und damit wiederholbar sein)
- Psychologisch:
  - ➔ Programmierer neigen dazu, nur die Fälle zu testen, die sie wirklich in ihrer Implementierung abgedeckt haben
  - ➔ Testen ist stark beeinflusst durch die Programmiererfahrung
  - ➔ häufig wird nur "positiv" getestet

Positives und negatives Testen

Apcon Workplace Solutions
Member of itelligence

- Die gesamte Funktionalität einer Software-Einheit sollte durch eine Reihe von Testfällen getestet werden.
- Ein *Testfall* besteht aus der Beschreibung der erwarteten Ausgabedaten für bestimmte Eingabedaten.
- Wenn nur erwartete/gültige Eingabewerte getestet werden, spricht man von *positivem Testen*.
- Wenn unerwartete/ungültige Eingabewerte getestet werden, spricht man von *negativem Testen*.
- Positive Tests erhöhen das Vertrauen in die *Korrektheit*, negative Tests das Vertrauen in die *Robustheit*.

Vollständige Tests

Apcon Workplace Solutions
Member of itelligence

- In einem *vollständigen* Test werden *alle* gültigen Eingabewerte getestet.
- Vollständige Tests werden auch *erschöpfende* Tests genannt.
- diese Bezeichnung ist durchaus passend, wie bereits an einem kleinen Beispiel belegt werden kann:

```
public int multiply(int x, int y)
```

- Ein Test für alle gültigen Eingabewerte dieser Funktion würde für Java *sehr* lange dauern...

Äquivalenzklassen und Grenzwerte

Apcon Workplace Solutions
Member of itelligence

- Da vollständige Tests impraktikabel sind, werden verschiedene Eingabewertebereiche in Kategorien eingeteilt.
- Die Werte eines solchen Wertebereichs werden dann als für den Test äquivalent angesehen und brauchen deshalb nicht einzeln getestet werden. Stattdessen reicht es, wenn wenige Vertreter einer solchen *Äquivalenzklasse* getestet werden.
- Insbesondere sind dabei die Werte für die Tests von Interesse, die am Rande einer solchen Äquivalenzklasse liegen: die sogenannten *Grenzwerte*.
- Grenzwertkandidaten für `multiply` sind etwa 0, `Integer.MAX_VALUE` und `Integer.MIN_VALUE`, zwei Äquivalenzklassen wären beispielsweise die positiven und die negativen `int`-Werte.

Modultest und Integrationstest

Apcon Workplace Solutions
Member of itelligence

- Wenn die Einheiten eines Systems (Operationen, Klassen, Systemteile) für sich isoliert getestet werden, spricht man von einem *Modultest* (engl.: unit test). Modultests sind eher technisch motiviert und orientieren sich an den programmiersprachlichen Einheiten eines Systems.
- Wenn die getesteten Einzelteile eines Systems in ihrem Zusammenspiel getestet werden, spricht man von einem *Integrationstest* (engl.: functional test). Integrationstests werden aus Sicht eines Benutzers des Systems formuliert und sind sehr anwendungsbezogen.
- Da erfolgreiche Modultests die Voraussetzung für Integrationstests sind, betrachten wir vorläufig nur Modultests näher.
- Die Methoden zum Modultest lassen sich grob in Black-Box-Tests, White-Box-Tests und Schreibtischtests unterteilen.

Black-Box-Test

Apcon Workplace Solutions
Member of itelligence

- Ein *Black-Box-Test* betrachtet nur das Ein-Ausgabeverhalten einer Software-Einheit und nicht deren interne Implementierung (diese wird als ein "schwarzer Kasten" angesehen). Die Testfälle können also nur auf Basis der Spezifikation bzw. der Schnittstellendefinition der Software-Einheit formuliert werden.
- Ein Black-Box-Test sollte die zu testende Schnittstelle vollständig abdecken. Unter Zuhilfenahme von Grenzwerten und Äquivalenzklassen sollte etwa bei einer Klasse *jede* Operation der Schnittstelle *mindestens einmal* aufgerufen werden.
- Black-Box-Tests sollten sowohl positiv als auch negativ durchgeführt werden.

White-Box-Test

Apcon Workplace Solutions
Member of itelligence

- Bei einem White-Box-Test wird eine Software-Einheit mit Blick auf ihre Implementierung getestet (ein besserer Name wäre deshalb Glas-Box-Test).
- Es werden möglichst *alle* Kontrollpfade des Programmtextes getestet. D.h., bei jedem if-else-Statement sollte sowohl der if- als auch der else-Pfad getestet werden, bei jedem switch-Statement jeder Case-Label angesprungen werden, jede private Operation aufgerufen werden etc.
- Ein White-Box-Test ist somit noch stärker technisch orientiert als ein Black-Box-Test. Er testet nicht das, was eine Software-Einheit machen soll, sondern das, was die Software-Einheit tatsächlich tut.
- White-Box-Tests werden häufig nur als Ergänzung von Black-Box-Tests durchgeführt.

Schreibtischtest

- Beim *Schreibtischtest* wird der Programmtext auf dem Papier durchgegangen und der Programmablauf nachvollzogen. Da der Implementierer oft Fehler in seinem eigenen Programm übersieht, sollte zu solch einem *walk-through* eine zweite Person hinzugezogen werden, der der Programmablauf erklärt wird.
- Eine Variante des Schreibtischtest ist ein *Code-Review*. Dieses wird vom Implementierer vorbereitet, indem die relevanten Quelltextteile für alle Teilnehmer (Größenordnung etwa 5 bis 10) des Reviews ausgedruckt werden. Nachdem alle Teilnehmer den Programmtext gelesen haben, wird das Design und mögliche Alternativen diskutiert.
- Code-Reviews dienen damit eher der Verbesserung (Laufzeit, Speicherplatz) des Quelltextes als dem Finden von Fehlern.

Besonderheiten durch Objektorientierung

- *Kapselung* von Daten:
  Die innere Repräsentation eines Objektes lässt sich nicht observieren.
- *Vererbung*:
  Subklassen müssen auch den Anforderungen der Oberklasse genügen. Abstrakte Klassen können nur durch konkrete Klassen getestet werden.
- *Polymorphismus* / dynamisches Binden
  Unklar welches Objekt eine Anfrage bearbeitet, wenn das erst zur Laufzeit bekannt ist.
- Das *Vertragsmodell*
  ist komplementär zum Testen. Vor- und Nachbedingungen werden im Test beim Ausführen der Methode geprüft. Sie können trotzdem Tests nicht ersetzen.

Beispiel

Apcon Workplace Solutions
Member of itelligence