

Qualitätssicherung von Software am Beispiel von Unit Testing

Klaus Kusche, Mai 2012

Inhalt

- **Motivation**
- **Definition von „Unit Tests“, Einordnung in der QA, Nutzen**
- **Einordnung im V-Modell (==> Tafel)**
- **Inhalt, Erstellung, Durchführung**
- **„Reine Lehre“ vs. Praxis**
- **Werkzeuge & Beispiel (==> Demo)**

Aufgabe der Quality Assurance

Einer meiner Ex-Chefs:

„ ... sicherstellen,
dass gute Qualität entsteht
(==> vorbeugende Maßnahmen)
und schlechte Qualität
niemals das Haus verläßt!
(==> nachträgliche Prüfungen)“

Mittel der Quality Assurance

- Richtlinien („Coding Guidelines“) und organisatorische Maßnahmen
- Code Reviews (händisch) & statische Code-Analyse (automatisch)
- Tests und Messungen (händisch & automatisiert)
- Formale Verifikation (Beweis der Programm-Korrektheit)

Reichen Tests des Gesamtsystems?

- **Ziel:** Fehler möglichst zeitnah zur Codierung finden:

*„Je später ein Fehler gefunden wird,
umso teurer ist er!“*

Aber: Gesamtsystem-Tests sind
erst in sehr späten Projekt-Phasen möglich!

- **Ziel:** Ursache von Fehlern im Code lokalisieren:

Aber: Bei Gesamtsystem-Tests fast unmöglich!

- **Ziel:** Den Code möglichst gut durch Tests abdecken:

Aber: Bei Modulen tief innen von außen schwierig!

Was sind Unit-Tests?

Unit-Test =

- ... Test möglichst **kleiner Code-Stücke**
(= einzelne Funktion / Methode!)
- ... **isoliert** vom Rest des Codes
- ... auf **spezifikationskonformes Verhalten**.

Im Deutschen:

„Modultest“, „Komponenten-Test“

(im Gegensatz zum Integrations- und Systemtest)

Unit-Tests ...

- ... sind Tests
 - ==> Können keine formale Garantie für die Fehlerfreiheit liefern!
- ... laufen automatisch
 - ==> Erfordern Programmierung & Werkzeuge!
 - ==> Erstellung: Hoher Einmal-Aufwand!
 - ==> Durchführung: Minimaler laufender Aufwand!
- ... sind die erste „nachträgliche“ QA-Maßnahme
(mit vorbeugenden Nebeneffekten!)

Nutzen von Unit-Tests (1)

- Frühe Fehlererkennung, genaue Lokalisierung
- Lieferung flächendeckend getesteter, fehlerfreier Komponenten zur Systemintegration

- Unit-Tests sind auch hervorragende

„Regression Tests“:

„Welche neuen Fehler & Nebenwirkungen hat eine nachträgliche Änderung in ehemals funktionierendem Code verursacht?“

Nutzen von Unit-Tests (2)

Positive Nebeneffekte beim Schreiben der Tests:

- Prüfung der **Spezifikation**:
Fehler? Lücken? Unklarheiten?
- Prüfung des **Feinentwurfes**:
Mangelhafte Funktions- / Methoden-Interfaces?
Konfuse Klassenhierarchie?

Zweit-Nutzung des Test-Codes:

- Die Tests sind zugleich **Beispiele**
für die interne Code-Dokumentation!

Inhalt von Unit-Tests (1)

- Logisch:

1 Testfall pro Verhalten einer Funktion
(laut Spezifikation)

Auch alle

... **Sonderfälle**

... **Randfälle** (gute & schlechte Seite!)

... **Fehlerfälle** (illegaler Aufruf, Überlauf, ...)

Inhalt von Unit-Tests (2)

- Technisch:

- 1 Testfall pro möglichem Codepfad**

- Weiters:

- 1 Testfall für jeden bekannten Bug!**

- ==> Hilft dem Entwickler beim Reproduzieren

- ==> Testet die Behebung des Bugs

- ==> Schützt davor, dass sich derselbe Bug in Zukunft nochmals einschleicht!

Erstellung von Unit-Tests

- ... zeitnah / gleichzeitig mit dem Code,
- ... meist nicht in der Qualitätssicherung, sondern durch die Entwickler des Codes oder durch das Spezifikations-Team

==> Problem falls Test-Autor = Code-Autor

„Blindheit“ gegen eigene Fehler!

==> Autor macht oft dieselben Denkfehler, dieselbe Fehl-Interpretation der Spezifikation im Code und im Test!

„Black Box“ oder „White Box“ ?

Beides!

- Grundsätzlich zuerst „Black Box“:

Ohne Kenntnis des Codes,
nur nach Spezifikation!

*==> Vermeidet Duplikation von Code-Fehlern
und Übersehen von im Code fehlenden Fällen!*

- Danach „White Box“: Vervollständigung
gemäß automatischer Code-Coverage-Analyse
(Abdeckung möglichst vieler Codepfade)

Extremfall „*Test-Driven Development*“

Eines von mehreren Vorgehensmodellen der „agilen Softwareentwicklung“:

- Die Tests werden vor dem Code geschrieben
- Die Tests sind zugleich offizielle Spezifikation
(==> kein separates Textdokument mehr!)

Meist in kurzen, kleinen Iterationen,
nicht für das ganze Projekt auf einmal!

(Tests für ein neues Feature, Code dazu implementieren,
Tests für das nächste Feature, wieder codieren, ...)

Umfang und Anwendungsgebiete

Typischer Umfang von Unit-Tests:

- Rund 1 Dutzend Testfälle pro Funktion!
- Über 100 Testfälle pro Komponente / Klasse!
- Viele 1000 / 10000 Testfälle pro Projekt!
==> Oft über 5 Lines of Test pro Line of Code!

Schlecht für Unit-Tests geeignet:

- Der GUI-Teil eines Projektes
- Multithreaded Code

Durchführung von Unit-Tests

- ... vollautomatisch und „unbemannt“,
in **make** bzw. das Build-System integriert
(==> unmittelbar nach dem Compilieren!).
- ... spätestens jede Nacht
mit jedem neuen „Nightly Build“
==> automatischer Vergleich mit dem Vortag
==> Fehlerhafte Änderungen rasch erkannt!
- ... besser vor jedem Checkin
einer Änderung oder Neuentwicklung
==> Nur korrekter Code wird eingecheckt!

Wichtiger Grundsatz

Testcode, Testdaten und Testergebnisse

- ... sind Bestandteil des Projektes / Produktes
- ... gehören zum jeweiligen Entwicklungsstand
- ... und werden daher so wie der Code selbst

*in der Versionsverwaltung
eingchecked und mitversioniert!*

Was sind „isolierte Tests“?

Laut „reiner Lehre“,
zur Garantie voneinander unabhängiger Testfälle:

Jeder Test darf
nur die zu testende Funktion
aus dem Produktivcode verwenden!

=> Diese Funktion darf keine anderen Funktionen
aus dem Produktivcode aufrufen!

=> Zum Erstellen der Testdaten dürfen
keine Funktionen aus dem Produktivcode
verwendet werden!

Erstellung „isolierter Tests“

Der Aufruf von Realcode wird umgeleitet auf eigens für die Tests geschriebene

„Mock-Objekte“ und „Stub-Funktionen“:

- Haben dieselbe Schnittstelle wie ihre realen Gegenstücke,
- simulieren nur das für den Test nötige Aufruf-Verhalten der realen Funktionen/Objekte,
- täuschen Datenbank- und File-Inhalte oder Ein- und Ausgaben vor, ...

Probleme von „Stubs“ und „Mocks“

- **Sehr hoher Aufwand**
(Erstellung & Wartung des Stub- & Mock-Codes!)
- **Stubs & Mocks sind ungetesteter Code!**
(enthalten auch Fehler...)
- **Divergenz vom Verhalten des Realcodes**
(z.B. verschiedene Auslegung der Spezifikation, nachträgliche Spezifikationsänderung vergessen)
=> Test läuft durch, Echtfall versagt!

Notlösung statt „Stubs“ und „Mocks“

- Verwendung des Realcodes für Testdatenerstellung und Unterfunktionen
- Ausführung der Tests

in „Bottom-Up“-Reihenfolge

==> Nur Aufruf von zuvor schon fertig getesteten Funktionen!

Probleme:

- Testreihenfolge bei zyklischen Abhängigkeiten?
- Fehler in einer Funktion „tief unten“
==> alle nachfolgenden Tests sind wertlos!

Werkzeuge für Unit-Tests

Unit-Tests basieren (fast) immer auf fertigen
Unittest-Frameworks.

Beispiele:

- **JUnit** (Java)
- **CUnit** (C)
- **CppUnit, CppTest, GoogleTest, ...** (C++)
- ... (für fast jede Sprache)

Framework-Komponenten (1)

1) Test-Makros:

- Rufen die zu testende Funktion auf,
- prüfen das Ergebnis & die Nebenwirkungen (z.B. Exception, Signal, File-I/O),
- protokollieren den Test (Name, Codestelle) und sein Ergebnis bzw. den Fehlerwert.

1 Testmakro-Aufruf ~ 1 Testfall

Framework-Komponenten (2)

2) Test-Setup & Test-Tearardown:

Jeder einzelne Test beginnt in einem separaten, frisch initialisierten (leeren) Test-Environment (unabhängig vom vorhergehenden Test!)

=> Das Setup füllt das Test-Environment vor jedem Test mit frischen Testdaten (Datenstrukturen, Objekte, ... = „Test Fixture“), das Tearardown räumt danach auf.

=> Derselbe Setup- und Tearardown-Code wird meist für mehrere Tests verwendet!

Framework-Komponenten (3)

3) Testverwaltung und -ausführung:

- Jeder Test wird im Framework registriert.

Meist hierarchisch:

Test-Suites (1 pro Modul / Klasse)

==> **Tests** (1 pro Funktion / Methode)

==> **Testfälle** (1 pro Funktionalität)

- Ein automatisch generiertes main führt alle registrierten Tests aus.

Framework-Komponenten (4)

4) Auswertungs-Skripts:

- Generieren:
 - Statistiken (# ok, # failed, % failed pro Modul, ...)
 - Fehlerlisten
 - Unterschiede zum vorigen Test-Lauf
- Speichern und Präsentieren die Ergebnisse:
 - als HTML, PDF, Text, Excel, ...
 - in einem interaktivem GUI
 - in einer Datenbank
 - mittels Mail an den jeweiligen Entwickler