Softwaretests

Florian Ehmke

21. März 2011

Inhaltsverzeichnis

1 Einleitung

Though scientific software is an engine for scientific progress and provides data for critical decisions, the testing of scientific software is often anything but scientific.

Diane Kelly and Rebecca Sanders

Softwaretests gehören zu den grundlegenden Prinzipien in der Softwaretechnik. Es gibt eigens Softwaretester, die Experten darin sind Software zu testen und Fehler zu finden, die der Entwickler des zu testenden Codes nicht gefunden hat / hätte. Wissenschaftliche Software wird in den meisten Fällen nicht von professionellen Softwareentwicklern entwickelt und somit auch nicht angemessen getestet. Softwaretests finden auf allen Ebenen der Softwareentwicklung statt – die kleinsten Einheiten des Codes werden genauso getestet wie die fertige Software als Ganzes. Oft fehlt die Motivation oder das Verständnis dafür Tests für eine Routine, die auf den ersten Blick korrekt aussieht und erwartete Ausgaben produziert, zu entwickeln.

1.1 Warum überhaupt testen?

Softwaretests sollen das Vertrauen in die Software erhöhen und Fehler aufdecken. Nicht nur Fehler die beispielsweise im Code dafür verantwortlich sind, dass falsche Ausgaben produziert werden, sondern auch Fehler in Spezifikationen, die dadurch entstanden sind, dass Nutzer und Entwickler sich falsch verstanden haben. Weiterhin muss verifiziert werden, dass die Software alle Anforderunge, die in der Softwarespezifikation formuliert sind, erfüllt.

1.2 Verifikation und Validation

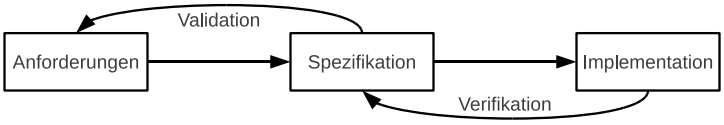


Abbildung 1: Verifikation und Validation

Im Bereich der Softwaretests unterscheidet man zwischen Validierung und Verifizierung. Abbildung 1 veranschaulicht den Unterschied zwischen diesen beiden Prozeduren. Auf allen Ebenen der Softwareentwicklung findet man den gleichen Prozess – es existieren Anforderungen aus denen Spezifikationen abgeleitetet werden und anhand dieser Spezifikation findet eine Implementation statt. Wann immer etwas gegen eine Spezifikation getestet wird spricht man von einer Verifizierung, während Tests gegen die Anforderungen Validierungen sind.

1.3 Definition Test

Es gibt sehr viel Literatur zu diesem Thema und entsprechend viele Definitionen, was genau man unter einem Softwaretest zu verstehen hat. Folgende Definition ist sehr gut, da sie auf allen Ebenen des Softwaretestens „passt“.

> Unter Testen versteht man den Prozess des Planens, der Vorbereitung und der Messung, mit dem Ziel, die Eigenschaften eines IT-Systems festzustellen und den Unterschied zwischen dem tatsächlichen und dem erforderlichen Zustand aufzuzeigen.
>
> Pol, Koomen und Spillner

Der erforderliche Zustand ist hierbei der Zustand, der alle Anforderungen des Nutzers erfüllt und nicht nur eine Spezifikation (die möglicherweise Fehler enthält) [7].

Im folgenden wird auf die einzelnen Ebenen des Softwareentwicklungsprozesses eingegangen (Abschnitt 3). Wie umfangreich das Testen ausfallen sollte und was alles dazu gehört wird in Abschnitt 2 besprochen. Unterschiedliche Strategien und die Rollenverteilung – testet und entwickelt ein und dieselbe Person oder testet jemand anderes? – sind in Abschnitt 4 dargestellt. Außerdem wird auf die Probleme und den derzeitigen Zustand der Softwareentwicklung, insbesondere des Testens, in der Wissenschaft eingegangen (Abschnitt 5). Abschließend befindet sich in Abschnitt 6 eine Zusammenfassung der besprochenen Inhalte.

2 Umfang der Tests

In welchem Maß das Testen von Software stattfinden sollte ist eine immer wiederkehrende Frage. Testabdeckung bezeichnet das Verhältnis zwischen den tatsächliche vorhandenen Tests und den insgesamt möglichen Tests [8]. Ein nah verwandtes Maß ist die Codeabdeckung (vgl. 2.3). Mit 100% Codeabdeckung sind jedoch noch nicht 100% Testabdeckung erreicht, da die vollständige Codeabdeckung bereits mit einer Teilmenge der möglichen Eingabedaten erreichbar ist.

2.1 Ein-/Ausgabedaten

Listing 1: Summierer

```
1 class Trivial {
2   static int sum(int a, int b) {
3     return a + b;
4   }
5 }
```

Es ist unmöglich alle Eingabedaten für Funktionen / Methoden zu testen. Schon ein einfachstes Beispiel genügt, um dies zu demonstrieren (Listing 1). Um die Klasse `Trivial` mit allen Wertekominationen zu testen werden Jahre benötigt. Daraus folgt, dass es in den meisten Fällen nicht möglich ist zu beweisen, dass die Software keine Fehler hat.

> Testing can be used to show the presence of bugs, but never show their absence!
>
> E. W. Dijkstra

Um in solchen Fällen, in denen es nicht möglich ist alle Eingabedaten zu testen, Testfälle zu kreieren, die das Vertrauen in die Software erhöhen, greift man auf Randwerte und Äquivalenzklassen (vgl. Abschnitt 2.2) zurück.

2.2 Äquivalenzklassen

Von einer Äquivalenzklasse erwartet man, dass sich Werte aus ihr als Eingabedaten äquivalent verhalten. Das hat den Vorteil, dass man nicht alle Werte der Äquivalenzklasse testen muss — einige wenige Vertreter genügen. Besonders von Interesse sind Werte, die am Rande einer Äquivalenzklasse liegen. Im

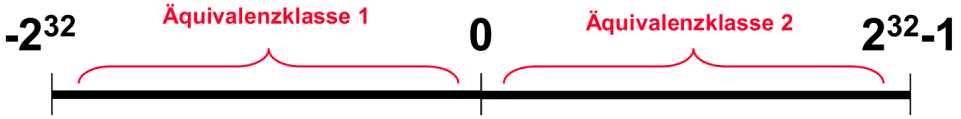


Abbildung 2: Äquivalenzklassen (Quelle: SE1 Skript)

Beispiel (Abbildung 2) wären dies `Integer.MIN_VALUE`, `Integer.MAX_VALUE` sowie `0`.

2.3 Codeabdeckung

Listing 2: Beispiel Codeabdeckung

```
1 int foo(int x, int y)
2 {
3     int z = 0;
4     if ((x > 0) && (y > 0)) {
5         z = x;
6     }
7     return z;
8 }
```

Codeabdeckung ist eine Form der Testabdeckung. Codeabdeckung lässt sich noch weiter spezifizieren. Es wird unterschieden zwischen Funktionsabdeckung, Zeilenabdeckung, Entscheidungsabdeckung und Bedingungsabdeckung. Am Beispiel (Listing 2) erklärt: (Beispiel aus [1])

- Funktionsabdeckung ist durch einen Aufruf von `foo(int x, int y)` mit beliebigen Parametern befriedigt.
- Zeilenabdeckung genügt folgender Aufruf: `foo(1, 1)`. In diesem Fall ist die `if` Anweisung in Zeile 4 erfüllt und jede Zeile des Codes wird ausgeführt.
- Für Entscheidungsabdeckung sind 2 Aufrufe nötig. Einmal `foo(1, 1)` damit die Zeile `5 (z = x;)` innerhalb der if Anweisung ausgeführt wird, sowie `foo(1, 0)` damit dies nicht passiert.
- Um Bedingungsabdeckung zu erfüllen, muss der `if` Teil auf jede mögliche Art ausgeführt bzw. nicht ausgeführt werden. Das heißt einmal muss er nicht ausgeführt werden da `x > 0` nicht erfüllt ist und einmal da `y > 0` nicht erfüllt ist. Somit sind für Bedingungsabdeckung die Aufrufe `foo(1, 1)`, `foo(1, 0)`, `foo(0, 0)` von nöten.

2.4 Error-Seeding

Tests werden verwendet um die Korrektheit der Implementation von Software zu verifizieren. Doch das Erstellen von Tests führt unweigerlich zu der Frage, ob denn die Tests korrekt sind. Dieser Test von Tests[1] lässt sich durch willentlich in den Code injizierte Fehler realisieren. Hat man eine Menge von Fehlern im Code verteilt und die bisherigen Tests finden nicht alle davon, muss man die Tests erweitern. Während beim Error-Seeding die Arbeit von Hand gemacht wird, gibt es Techniken die systematisch den kompletten Code verändern. Man spricht dabei vom Mutationen-Test (vgl. Abschnitt 2.5).

2.5 Mutationen-Test

Listing 3: Beispiel Mutationen

```
1 if (a && b)
2   c = 1;
3 else
4   c = 0;
```

```
1 if (a || b)
2   c = 1;
3 else
4   c = 0;
```

Mutations-Tests ähneln stark dem Error-Seeding. Beim Mutations-Test werden Operatoren und Variablen nach vordefinierten mustern mutiert (Listing 3). Populäre Mutations-Verfahren („mutation operators“) sind:

- `true` und `false` vertauschen
- Arithmetische Operatoren austauschen ($+ \rightarrow -$)
- Vergleichsoperatoren vertauschen (`==` $\rightarrow$ `!=`)

Die Mutationen werden nicht alle auf einmal durchgeführt, sondern eine pro Testlauf, um das Ergebnis nicht zu verfälschen. Schlägt ein /mehrere Test(s) fehl, nachdem der Code mit Mutation getestet wurde, sagt man die Mutation wurde „gekillt“.

Der Unterschied zum Error-Seeding ist, dass man beim Mutationen-Test systematischer vorgeht und eine höhere Abdeckung erreicht. Durch Mutationen-Tests werden zudem tote (nie ausgeführte) Codeteile aufgedeckt.

[1] „Who will guard the guards?“

2.6 Blackbox / Whitebox

Blackbox- und Whitebox-Tests bezeichnen zwei unterschiedliche Betrachtungen des zugrundeliegenden Systems während des Testens. Bei einem Whitebox-Test kennt man die Struktur und innere Funktionsweise der Software und macht Gebrauch von diesem Wissen. Es wird also der Quellcode betrachtet. Qualitätskritierien für Whitebox-Tests sind (unter anderem) die verschiedenen Formen der Codeabdeckung (vgl. Abschnitt 2.3). Bei einem Blackbox-Test wird das System gegen seine Spezifikation getestet.

3 Testebenen

Das Testen von Software findet auf unterschiedlichen Ebenen des Softwareentwicklungsprozesses statt. Bei höheren Ebenen spricht man von System- oder Abnahmetests — auf niedrigeren Ebenen finden Komponententests sowie Integrationstests statt. Grafisch dargestellt wird dies im populären V-Modell, welches den jeweiligen Ebenen des Softwareentwicklungsprozesses entsprechende Testformen zuordnet.

3.1 V-Modell

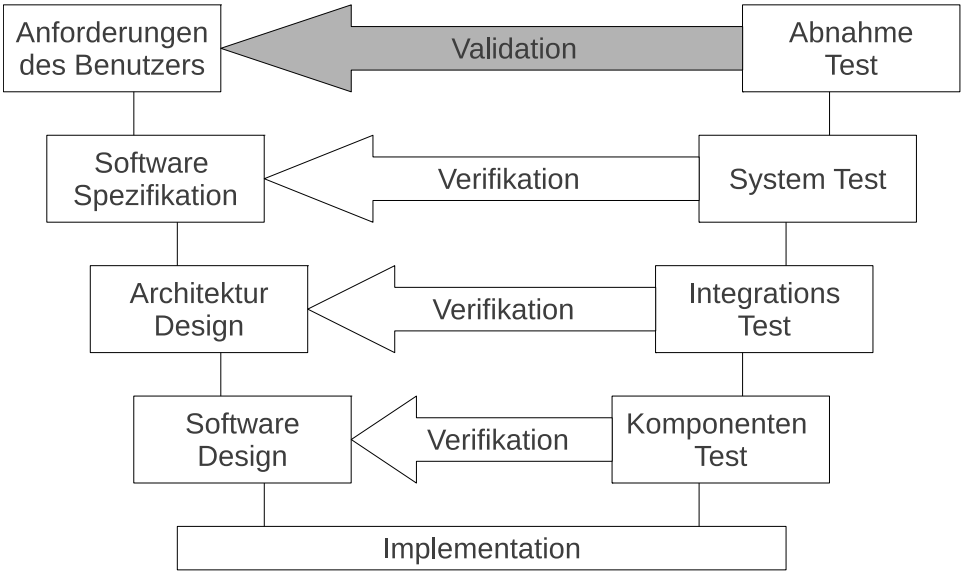


Abbildung 3: Das V-Modell

Validation muss nicht nur in Form eines Abnahmetests stattfinden. Ist der zukünftige Anwender der Software während des Entwicklungsprozesses anwesend bzw. in Kommunikation mit den Entwicklern, kann Validation auch auf anderen Ebenen durchgeführt werden. So kann zum Beispiel die Softwarespezifikation, die aus den Anforderungen des Benutzers entstanden, ist zusammen mit dem Benutzer überprüft werden.

3.2 Komponententest

Die unterste Testebene — der Komponententest — testet die kleinsten testbaren Einheiten der Software. Das kann eine Klasse, eine Methode oder eine

Funktion sein. Diese Komponenten (oder Module) werden später zusammengefügt (integriert) und mittels Integrationstests getestet. Bei testgetriebener Entwicklung (TDD – Test-driven development) wird der Komponententest vor der eigentlichen Komponente entwickelt und implementiert. Komponententests sind im Idealfall unabhängig von einander (vgl. Abschnitt 4.1).

3.3 Integrationstest

In der Phase der Integrationstests werden Komponenten kombiniert und in Gruppen getestet. Bei der Integrationsphase können Fehler entstehen, die selbst durch ausgiebe Komponententests nicht aufgedeckt werden. So kann es beispielsweise vorkommen, dass 2 Komponenten jeweils eine temoräre Datei benötigen. Liegen diese temporären Dateien mit dem selben Namen im selben Verzeichnis entsteht ein Konflikt, der bei den jeweiligen Komponententests nicht entdeckt worden wäre. Integrationstests decken auch Fehler auf, die durch nicht eindeutige Spezifikiationen entstanden sind. So ist die Maßeinheit *Meilen* nicht eindeutig. Es könnten sowohl *Seemeilen* als auch *Landmeilen* gemeint sein. Da diese unterschiedliche Werte haben kommt es bei der Interaktion zweier Komponenten, die unterschiedliche Meilen implementieren, zu Konflikten.

3.4 Systemtest

Der Systemtest ist die Teststufe in der gegen die Softwarespezifikation getestet wird. Das System befindet sich in dieser Phase in einem kompletten, integriertem Zustand. Die Testumgebung sollte der Produktivumgebung möglichst ähnlich sein oder ihr entsprechen. Da in dieser Phase keine Einblicke in den Code benötigt werden, handelt es sich um einen Blackboxtest. Im Zuge eines Systemtests werden unter anderem Performance-, Installations-, Wiederinbetriebnahme-, Stress- und Usability-Tests durchgeführt (vgl. Abschnitt 3.6).

3.5 Abnahmetest

Der Abnahmetest unterscheidet sich dadurch grundlegend von allen anderen Tests, dass er vom Benutzer durchgeführt wird und nicht vom Entwickler / Tester. Der Test findet außerdem in der Produktivumgebung statt und nicht länger in einer Testumgebung. Es wird getestet, ob das fertige Produkt aus Sicht des Benutzers alle Anforderungen erfüllt. Zu den geprüften

Anforderungen gehören:

- Ist der Leistungsumfang vollständig? Wurden alle Spezifikationen (aus Nutzersicht) eingehalten?
- Ist die Grafische Benutzeroberfläche verständlich und benutzbar?
- Ist die Dokumentation angemessen umfangreich und verständlich?

3.6 Weitere Testarten

Performancetest Ein Test der gezielt die Performance eines System testet/misst und so den Entwickler dabei unterstützt Engpässe / Flaschenhälse aufzudecken. Mit den erlangten Erkenntnissen lässt sich die Performance des Systems leicht(er) optimieren.

Installationstest Installationstests simulieren eine Installation in der vorgesehenen Produktivumgebung und sollen Fehler, die durch die veränderte Umgebung (Entwicklungsumgebung → Produktivumgebung) entstanden sind aufdecken. Zu den Veränderungen gehören beispielsweise:

- CPU-Architektur
- Software-Bibliotheken
- Installierte Software
- Betriebssystem
- Netzwerk

Üblicherweise gehört auch ein Test der Deinstallations-Routinen zum Installationstest.

Wiederinbetriebnahmetest Wie verhält sich die Software nach einem Ausfall bedingt durch z.B. einen Stromausfall? Man kann 3 Szenarien unterscheiden:

1. Die Software wird nicht beeinträchtigt und macht an exakt der Stelle weiter, an der zuvor der Stromausfall auftrat.
2. Die Software wird bedingt beeinträchtigt und muss einen Teil von zuvor bereits fertiggestellter „Arbeit“ wiederholen.
3. Die Software besitzt keine Wiederinbetriebnahme-Funktion. Sämtliche Ergebnisse gehen verloren, alles muss wiederholt werden.

Stresstest Wie verhält sich das System unter extremer Last? Es wird kontrolliert ob es nach wie vor möglich ist Eingaben vorzunehmen oder anderweitig das Programm zu kontrollieren, wenn große Last auf dem System anliegt. Man kann hierbei zwischen 2 verschiedenen Lasten unterscheiden:

1. Last, die durch das zu testende Programm selbst erzeugt wird.
2. Last, die durch externe Programme entsteht und somit nicht immer vorhersehbar ist.

Usabilitytest Usability ist ein sehr großes und umfangreiches Thema. In der Regel werden Usabilitytests von Nutzern oder Testern, die die Rolle des Nutzers simulieren, durchgeführt. Dabei liegt das Hauptaugenmerk darauf, ob Schwierigkeiten bei der Benutzung auftreten. Solche Schwierigkeiten könnten zum Beispiel Buttons sein, deren Beschriftung es für den Benutzer nicht eindeutig ersichtlich macht, welche Aktion durch eine Betätigung ausgelöst wird.

Regressionstest Wann immer Teile der Software verändert werden, sei es durch Erweiterung oder Veränderung der Funktionen oder Behebung eines Fehlers, ist es notwendig alle oder eine Teilmenge der vorhandenen Tests neu durchzuführen. Denn selbst kleinste Änderungen können zu unverhergesehenen Fehlfunktionen führen. Regression liegt immer dann vor, wenn eine Codeänderung entgegen der Intention ein „Rückschritt“[2] ist. Dem zur Folge ist “*nonregression*“ ein Qualitätskritierium. Ein simpler Ansatz, um Regressionstests durchzuführen, wäre einfach nach jeder Änderung alle vorhanden Tests durchzuführen und die Ergebnisse mit denen vor der Änderung zu vergleichen. Dieser Ansatz ist wenig praktikabel und sehr zeitaufwändig. Es ist ratsam nur eine Teilmenge der Tests auszuführen, die Auswahl dieser Teilmenge ist, worauf es ankommt.

[2] Regression engl. *Rückschritt*

4 Teststrategien

4.1 Testumgebung

Software sollte isoliert von der Produktivumgebung getestet werden um, im Falle eines Fehlers, Beeinträchtigungen des Produktivbetriebes zu vermeiden. Bei der Testumgebung kommt es vor allem darauf an, dass sie der Produktivumgebung so ähnlich wie es geht ist, dabei aber so isoliert wie möglich. Zur Testumgebung gehören zahlreiche Dinge wie:

- Installierte Bibliotheken
- Installierte Software (wie Datenbanken)
- Zustand von Datenbanken
- Softwarekomponenten
- Hardware

4.2 Scaffolding

Wird eine neue Softwarekomponente entwickelt, gehört dazu auch ein geeignete Menge von Tests. Häufig bestehen aber Abhängigkeiten zwischen Komponenten, die ein isoliertes Testen der Komponente, wie es ein Komponententest vorsieht, erschweren. Weiter erschwert wird die Situation durch Abhängigkeiten von Komponenten, die noch nicht implementiert sind. Es ist nicht ratsam in einem solchen Fall zunächst diese Komponenten zu entwickeln, da die Abhängigkeiten oft wechselseitig sind. Stattdessen wird ein „Mock Objekt“ (auch „Dummy“ oder Attrappe genannt) implementiert, welches das unfertige Modul simuliert implementiert. Solche Objekte werden auch verwendet, wenn anstelle des „Mock Objekts“ eine bereits fertige Komponente zur Verfügung steht, um dem Anspruch gerecht zu werden, dass die Komponenten unabhängig von einander getestet werden können.

4.3 Rollenverteilung

Es hat viele Vorteile wenn nicht ein und dieselbe Person den Code und die dazugehörigen Tests entwickelt. Während der Planung und Implementation einer Komponente hat der Entwickler meist ein Anwendungsszenario und

damit verbundene Eingabe- / Ausgabedaten im Kopf. Es liegt nahe, dass der Entwickler beim Testen ähnliche Werte verwendet. Ein Tester hat an dieser Stelle eine völlig unvoreingenomme Sicht auf die Komponente und zieht eventuell Szenarien in Betracht, die der Entwickler nicht getestet hätte. Zudem sind Tester auf ihrem Gebiet Experten und haben in der Regel mehr Erfahrung auf diesem Gebiet.

4.4 Automatisieren

Automatisierung kann die Effizienz des Entwicklungsprozesses enorm steigern. Beispielsweise Regressionstests (vgl. Abschnitt 3.6) werden immer wieder aufs neue ausgeführt und sind dadurch prädestiniert um automatisiert zu werden. Es ist auch sinnvoll Tests automatisiert zu einer bestimmten Uhrzeit auszuführen. Zum Beispiel jeden Abend – auf diese Art bekommt man sehr gut vergleichbare Ergebnisse und kann anhand derer Rückschlüsse ziehen, wann eine Fehler das erste Mal aufgetreten ist oder wann eine Aufgabe abgeschlossen ist. Es gibt Tools, die den Entwickler beim Automatisieren unterstützen, aber bei kleineren Projekten kann die Automatisierung auch über Skripte erreicht werden.

4.5 Tools

Es gibt zahlreiche Tools die den Entwickler beim Testen unterstützen. Unter anderen die xUnit Familie, zu der neben dem wohl bekanntesten Werkzeug JUnit auch CUnit[3] für C gehört.

4.5.1 CUnit

CUnit ist Open Source und auch über die Ubuntu Paketquellen erhältlich. Zum Testen wird CUnit als statische Bibliothek (`-lcunit`) mit dem Programm verlinkt. Das Programm enthält mehrere Interfaces die für die Ausgabe der Ergebnisse zuständig sind.

Interfaces

Basic Die Ergebnisse der Tests werden auf der Konsole ausgegeben (Abbildung 4). Die Tests werden ausgeführt, ausgegeben und das Programm

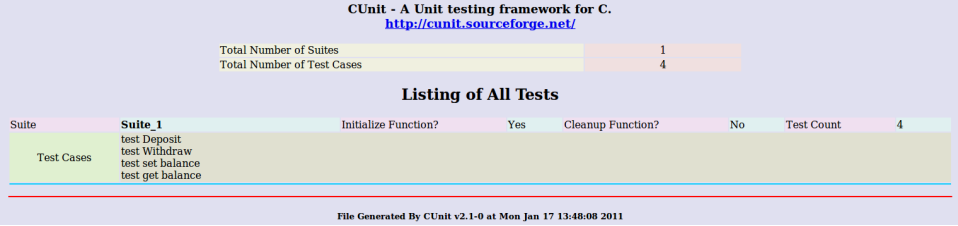
[3] http://cunit.sourceforge.net/

```
     CUnit - A Unit testing framework for C - Version 2.1-0
     http://cunit.sourceforge.net/


Suite: Suite_1
  Test: test Deposit ... passed
  Test: test Withdraw ... passed
  Test: test set balance ... passed
  Test: test get balance ... FAILED
    1. testBank.c:37  - CU_ASSERT_EQUAL(getBalance(),5000)

--Run Summary: Type      Total     Ran  Passed  Failed
               suites        1       1     n/a       0
               tests         4       4       3       1
               asserts       4       4       3       1
```

Abbildung 4: CUnit Ausgabe Basic-Mode



CUnit - A Unit testing framework for C.
http://cunit.sourceforge.net/

Total Number of Suites	1
Total Number of Test Cases	4

Listing of All Tests

Suite	Suite_1	Initialize Function?	Yes	Cleanup Function?	No	Test Count	4
Test Cases	test Deposit test Withdraw test set balance test get balance						

File Generated By CUnit v2.1-0 at Mon Jan 17 13:48:08 2011

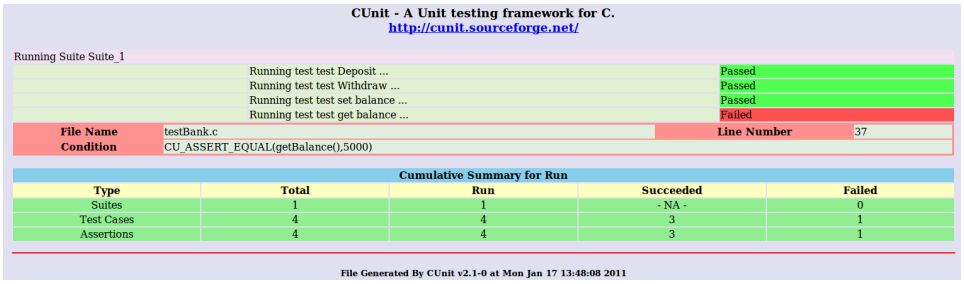
Abbildung 5: CUnit Automated listing.xml

beendet sich wieder.

Automated Es werden 2 anschauliche XML-Dateien generiert – eine (Abbildung 5) stellt übersichtlich die vorhanden Tests dar und die andere (Abbildung 6) die Testergebnisse.

Console Die Ausgabe findet interaktiv auf der Konsole statt. Die Funktionen der interaktiven Konsole umfassen: `(R)un all, (S)elect suite, (L)ist suites, Show (F)ailures, (Q)uit.`

Ncurses Die Ausgabe findet interaktiv (mittels `ncurses`) auf der Konsole statt. Die interaktiven Funktionen sind identisch mit dem Console-Interface jedoch ist das GUI ansprechender. Das Ergebnis der Tests wird durch einen Balken visualisert. Dieser ist grün wenn keine Fehler auftraten und andernfalls rot.



CUnit - A Unit testing framework for C.
http://cunit.sourceforge.net/

Running Suite Suite_1

Running test test Deposit ...	Passed
Running test test Withdraw ...	Passed
Running test test set balance ...	Passed
Running test test get balance ...	Failed

File Name	testBank.c	**Line Number**	37
Condition	CU_ASSERT_EQUAL(getBalance(),5000)		

Cumulative Summary for Run

Type	Total	Run	Succeeded	Failed
Suites	1	1	- NA -	0
Test Cases	4	4	3	1
Assertions	4	4	3	1

File Generated By CUnit v2.1-0 at Mon Jan 17 13:48:08 2011

Abbildung 6: CUnit Automated result.xml

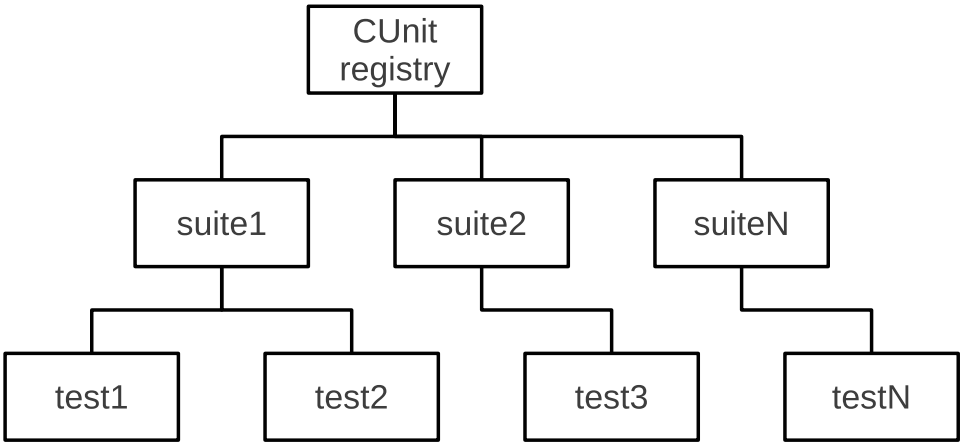


Abbildung 7: CUnit Hierarchie

Hierarchie CUnit ist sehr konventionell aufgebaut (Abbildung 7). Man unterscheidet zwischen der Registry, Test Suiten und Tests. Grob gesagt werden Tests zu Suiten zusammengefasst und diese Suites in der Registry bekannt gemacht. Zu einer Suite (kann) eine Initialisierungs- und eine Beenden-Funktion gehören.

Erste Schritte Zunächst müssen die entsprechenden Header-Dateien im Programm eingebunden werden, `CUnit.h` wird auf jeden Fall benötigt. Dazu kommen noch Header für die verwendeten Interfaces.

Listing 4: CUnit Header

```
1 #include <CUnit/CUnit.h>
2 #include <CUnit/Basic.h>
3 #include <CUnit/Automated.h>
```

Dieser Beispiel-Test prüft mittels `CU_ASSERT_EQUAL` ob die Funktion `deposit()` korrekt funktioniert. Das Guthaben des Kontos wurde vorher auf den Wert

`userBalance` initialisiert.

Listing 5: CUnit Test

```
1 #include "bankFunctions.h"
2
3 void testDeposit(void) {
4   float amount = 5000;
5   deposit(amount);
6   CU_ASSERT_EQUAL(getBalance(), amount + userBalance);
7 }
```

Die benötigten Schritte um die Tests durchlaufen zu lassen sind tatsächlich auf `CU_initialize_registry`, `CU_add_suite`, `CU_add_test` und `CU_basic_run_tests` beschränkt.

Listing 6: CUnit main

```
1 int main()
2 {
3    CU_pSuite pSuite = NULL;
4    CU_initialize_registry();
5    pSuite = CU_add_suite("Suite_1", setup, NULL);
6    CU_add_test(pSuite, "test Deposit", testDeposit)) {
7    CU_basic_set_mode(CU_BRM_VERBOSE);
8    CU_basic_run_tests();
9 }
```

Assertions Zum Testen werden in CUnit Assertions[4] verwendet. Für die Assertions, die `double` Werte auf Gleichheit Testen, muss die Mathematik-Bibliothek verlinkt sein. Für jede der aufgelisteten Assertions gibt es auch eine FATAL-Version (`_FATAL` am Ende des Namens). Wird bei einer FATAL-Assertion ein Fehler entdeckt bricht der komplette Test ab.

- `CUCU_ASSERT(int expression)`
- `CUCU_TEST(int expression)`
- `CUCU_ASSERT_TRUE(value)`
- `CUCU_ASSERT_FALSE(value)`
- `CUCU_ASSERT_EQUAL(actual, expected)`
- `CUCU_ASSERT_NOT_EQUAL(actual, expected))`
- `CUCU_ASSERT_PTR_EQUAL(actual, expected)`

[4] engl. Aussage/Behauptung

- CUCU_ASSERT_PTR_NOT_EQUAL(actual, expected)
- CUCU_ASSERT_PTR_NULL(value)
- CUCU_ASSERT_PTR_NOT_NULL(value)
- CUCU_ASSERT_STRING_EQUAL(actual, expected)
- CUCU_ASSERT_STRING_NOT_EQUAL(actual, expected)
- CUCU_ASSERT_NSTRING_EQUAL(actual, expected, count)
- CUCU_ASSERT_NSTRING_NOT_EQUAL(actual, expected, count)
- CUCU_ASSERT_DOUBLE_EQUAL(actual, expected, granularity)
- CUCU_ASSERT_DOUBLE_NOT_EQUAL(actual, expected, granularity)

5 Testen wissenschaftlicher Software

Das Problem wissenschaftlicher Software ist, dass sie von Personen entwickelt wird, für die sie lediglich ein Mittel zum Zweck ist. Die Software entsteht nicht um ihrer selbst willen sondern weil sie nötig ist, um bestimmte wissenschaftliche Ziele zu erreichen [2]. So ist es die Regel, dass der Entwickler softwaretechnisch nicht sehr versiert ist, was direkten Einfluss auf die Qualität der Tests hat.

5.1 Ist-Zustand

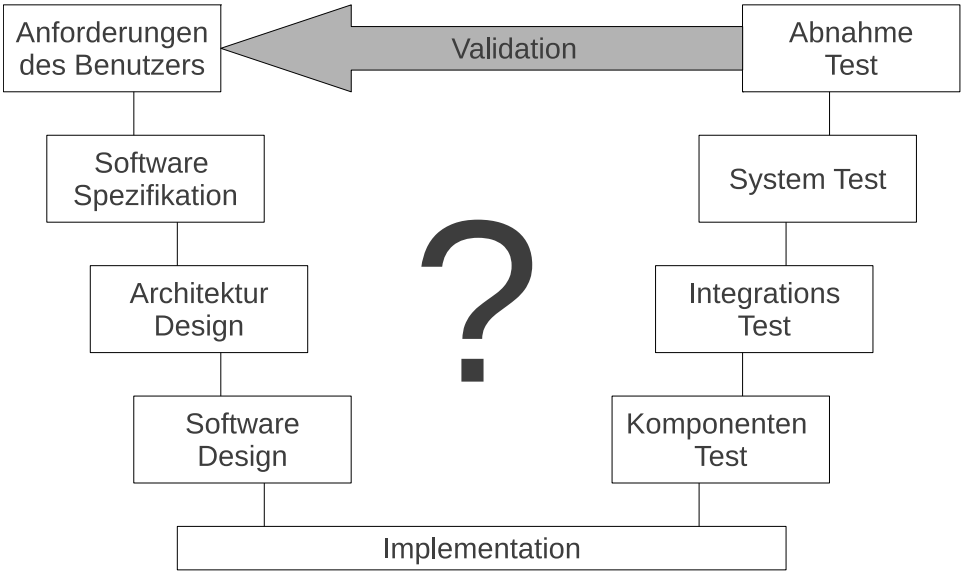


Abbildung 8: V-Modell Ist-Zustand Wissenschaft

Es ist falsch zu sagen, dass wissenschaftliche Software gar nicht getestet wird. Validation der Ausgabedaten des zu testenden Programmes findet auf jeden Fall statt. Das große Problem ist, dass die Wissenschaftler sich oft nicht klar darüber sind, wie wichtig Softwareverifikation ist – die Wichtigkeit von Validation demgegenüber ist über jeden Zweifel erhaben, schließlich geht es dabei direkt um die Modelle, derentwegen die Software überhaupt entsteht. Zudem sind die Wissenschaftler auf diesem Gebiet gewissermaßen wieder Fachmänner, denn sie können überlichweise besser beurteilen ob Ausgabedaten der „wissenschaftlichen“ Realität entsprechen oder völlig unrealistisch sind. Wie in der Grafik (Abbildung 8) dargestellt findet Testen also nur der höchsten Ebene statt. In [6, S. 2] wird der Testvorgang als „Verhält sich die Software mit normalen Eingabendaten wie erwartet? Wenn man keine ech-

ten Erwartungen an die Ausgabedaten hat, verhält sich die Software mit normalen Eingabedaten auf eine Art die man *überhaupt* nicht erwartet hat?“ beschrieben.

5.2 Probleme

Man kann davon ausgehen, dass die Entwickler wissenschaftlicher Software, wenn es die Wissenschaftler selbst sind, nicht ausgebildet im Testen von Software sind. Man kann von einem Wissenschaftler nicht verlangen Experte in seiner Domäne und der Softwaretechnik zu sein. Soll die Software eine gewisse Zeit überdauern kommt man aber nicht komplett ohne Qualitätanspruch aus. Neben gut strukturiertem, verständlichen und kommentiertem Code gehören dazu auch Tests. Ist der Wissenschaftler mit der Situation überfordert, beziehungsweise es ist einfach keine Zeit für solche Maßnahmen vorhanden, liegt es nahe sich helfen zu lassen. Die Modelle der Wissenschaftler sind häufig extrem komplex und für Außenstehende schwer zu verstehen. Es ist jedoch schwer Tests zu entwickeln, wenn man nicht versteht, was die Software überhaupt tut. Einen externen Tester / Entwickler einzustellen um Zeit zu sparen, kann so schnell mehr Zeit in Anspruch nehmen als geplant, wenn die Einarbeitung (die Erklärung des Modells, der Software) sehr lange dauert. In [2, S. 3] ist ein Beispiel gegeben in dem Wissenschaftler Softwareentwickler zum Testen und Entwickeln ihrer wissenschaftlichen Software eingestellt haben. Die Kollaboration verlief wegen dem schlechten Verständnis der Softwareentwickler für den wissenschaftlichen Kern der Software so schlecht, dass die Softwareentwickler am Ende nur bei der Grafischen Benutzeroberfläche helfen konnten und die Wissenschaftler eine solche Zusammenarbeit für zukünftige Projekte nicht länger in Betracht zogen. Es gibt auch Beispiele für erfolgreiche Kollaborationen – die verlangten aber von den Softwareentwicklern sich *hunderte* Stunden in die Materie einzuarbeiten. Hinzu kommt, dass mit steigender Rechenleistung auch komplexere Modelle berechnet werden können, was die Ausgangslage noch ungünstiger macht. Das spiegelt sich auch in [10, S. 2] wieder. In einer Umfrage antworteten 45% aller Wissenschaftler, dass sie mehr oder viel mehr Zeit mit der Entwicklung wissenschaftlicher Software verbringen als 5 Jahre zuvor. 70% antworteten auf die Frage wie viel Zeit sie mit der Benutzung wissenschaftlicher Software verbringen mit „mehr oder viel mehr “.

5.3 Wie viel testen?

Manchmal ist ein Kompromiss die beste Lösung. Wenn es durch Zeitdruck und Mangel an Wissen im Bereich Softwaretechnik nicht möglich ist Softwa-

re auf dem herkömmlichen Weg zu entwickeln müssen irgendwo Einstriche gemacht werden. Kommentiert, Validiert und Verifiziert wird „So wenig wie nötig, so viel wie möglich“. In [6, S. 2] wird vorgeschlagen, dass wenigstens in einigen Situationen Wissenschaftlern nicht der komplette Softwaretechnik-Apperat aufgezwungen werden sollte. Um Zeit zu sparen können auch einstriche bei der Codeabdeckung (vgl. Abschnitt 2.3) gemacht werden. Für das eigentliche Ergebnisse irrelevanter Code (Listing 7) muss nicht zwangsläufig durch Tests abgedeckt sein.

Listing 7: (Relativ) unwichtiger Code

```
1 FILE *fp = fopen("logfile","w");
2 fprintf(fp, "%d:%d = %d\n", hour, minute, value);
3 fclose(fp);
```

Stattdessen sollte man den Fokus eher auf Genauigkeit der Ergebnisse liegen [2, S. 5]. Die Genauigkeit der Berechnung ist ein Qualitätsmerkmal der meisten wissenschaftlichen Softwareprojekte. Numerische Analyse, numerische Methoden und Berechnungen mit Gleitkommazahlen stellen Schwerpunkte für Verifikation dar.

5.4 Größe des Projektes

Wissenschaftliche Software wird anders als die meiste andere Software in kleinen Gruppen oder alleine entwickelt. In einer Umfrage [10] antworteten 58%, dass sie die Software alleine Entwickeln während nur 9% mit mehr als 5 Leuten zusammen arbeiten. Unter diesem Aspekt betrachtet wird klar, dass Abstriche gemacht werden *müssen* (vgl. Abschnitt 4.3). Auch fällt es unter Umständen schwerer Code zu kommentieren, wenn der Entwickler davon ausgeht, dass er der einzige ist, der den Code je liest. Das dies nicht ratsam ist spiegelt sich in einem Umfrageergebnis aus [10] wieder. Code kommentieren steht an vorletzter Stelle der Dinge, die bei der Softwareentwicklung in der Wissenschaft am meisten Zeit benötigen, während gleichzeit schlecht kommentierter Code für 40% die „nervigste“ Sache an wissenschaftlichem Code ist (für 80% ist es in den Top 3 der schlechten Dinge in wissenschaftlichem Code). Einer der in [2, S. 5] befragten Wissenschaftler rät seinen Studenten sogar Code komplett neu zu entwickeln – dies sei einfacher als sich in fremden Code einzuarbeiten (wegen der schlechten Dokumentation).

6 Zusammenfassung

Softwaretests sind ein umfangreiches Thema – gutes Testen verringert nicht nur die Fehlerzahl und erhöht das Vertrauen in die Software, es trägt auch zur Wartbarkeit bei. Neue Versionen lassen sich schnell und automatisiert verifizieren. Zahlreiche Tools erleichtern Entwicklern das Testen ihrer Software. Wissenschaftler sind keine Experten in der Softwareentwicklung und getestet wird dementsprechend. Wissenschaftlich behandelt die Software Themen auf allerhöchstem Niveau während die Software technisch gesehen eher auf niedrigstem Niveau rangiert. Die Ursache für dieses Problem ist an verschiedenen Stellen zu suchen. In der Wissenschaft ist der Zeitdruck oft sehr hoch, Zeit um die Software zu Pflegen ist nicht vorhanden. Die Software ist nur das Mittel zum Zweck und wird nicht von Experten entwickelt. Da wissenschaftliche Software oft von nur einer Person entwickelt wird, fehlt die Akzeptanz für Verfahren, die eigentlich der Kollaboration dienen. Zudem sinkt die Hemmschwelle etwas „quick & dirty“ zu machen, wenn es in nächster Zeit außer dem Entwickler selbst niemandem vorgelegt wird. Arbeiten Wissenschaftler und Softwareentwickler zusammen, wird dies oft durch Defizite der Softwareentwickler im Verständnis für die zugrundeliegenden Modelle erschwert.

Literatur

[1] *Code coverage*. März 2011. URL: http://en.wikipedia.org/wiki/Code_coverage.

[2] Rebecca Sanders und Diane Kelly. „The Challenge of Testing Scientific Software“. In: *Proceedings of the 3rd annual Conference of the Association for Software Testing (CAST 2008)*. 2008.

[3] John Paul Elfriede Dustin Jeff Rashka. *Automated Software Testing - Introduction, Management and Performance*. Massachusetts: Addison Wesley Longman, Inc., 2008.

[4] C. Greenough L.S. Chin D.J. Worth. *A survey of software testing tools for computational science*. Techn. Ber. CCLRC ePublication Archive [http://epubs.cclrc.ac.uk/oai] (United Kingdom), 2007.

[5] Mauro Pezze und Michal Young. *Software testing and analysis - process, principles and techniques*. John Wiley & Sons, Inc, 2008.

[6] Judith Segal. „Models of scientific software development“. In: *First International Workshop on Software Engineering in Computational Science and Engineerin (SECSE 08)*. 2008.

[7] *Software testing*. Jan. 2011. URL: http://en.wikipedia.org/w/index.php?title=Software_testing&oldid=410699177.

[8] *Testabdeckung*. März 2011. URL: http://de.wikipedia.org/wiki/Testabdeckung.

[9] *Verification and validation*. Jan. 2011. URL: http://en.wikipedia.org/w/index.php?title=Verification_and_validation&oldid=400943822.

[10] Gregory Wilson. „How Do Scientists Really Use Computers?“ In: *American Scientist* (2009).