Testgetriebene Entwicklung – JUnit

Seminar Datenverarbeitung
Wintersemester 2006/07

vorgelegt von

Markus Lausberg
und
Robin Ziehe

Betreuer:
Dipl. Inform. Bodo Kraft, MaTA

8. Dezember 2006

Fachbereich Maschinenbau und Mechatronik
Goethestraße 1
52064 Aachen

Dienstgebäude Seffenter Weg 23
52074 Aachen

Inhaltsverzeichnis:

1 Einleitung

Diese Ausarbeitung soll eine Einführung in den Softwareentwicklungsprozess der Testgetriebenen Entwicklung geben. Dabei werden sowohl allgemeine Testgrundlagen erläutert, die Prinzipien von Unit-Tests erklärt, als auch eine Einführung in das Unit-Testing-Framework JUnit gegeben.

1.1 Motivation

Testgetriebene Entwicklung gehört zu der Gruppe der agilen Softwareentwicklungsmethoden, welche sich zum Ziel setzen, den Softwareentwicklungsprozess zu verbessern. Der dabei verwendete Ansatz unterscheidet sich sehr von dem der herkömmlichen Entwicklungsprozesse. Der Prozess der Softwareentwicklung soll leichtgewichtiger und flexibler werden, so dass der bürokratische Aufwand reduziert wird.

Das Testen spielt auch in den herkömmlichen Entwicklungsprozessen eine wichtige Rolle. Die Wichtigkeit des Testens wird bei der testgetriebenen Entwicklung allerdings noch stärker betont. Das Testen ist nicht eine bestimmte Phase, die zu einem Zeitpunkt durchlaufen wird, sondern eine Aktivität, die von Anfang an mitläuft. Dadurch sollen immer wieder auftretende Probleme, vermieden werden.

Dazu gehört zum Beispiel die Frage nach dem Zustand einer in Entwicklung befindlichen Software. Es ist schwer, eine Aussage über die Qualität zu treffen, weil die Anzahl der noch enthaltenen Fehler unbekannt ist. Im Allgemeinen wird anerkannt, dass ein fehlerfreies Programm nicht möglich ist, da jedes nicht triviale Programm Fehler hat. Mit steigender Komplexität eines Programms steigt die Fehlerwahrscheinlichkeit exponentiell an.

Selbst Tests können nicht die Abwesenheit von allen Fehlern zeigen, sondern nur die Anwesenheit. Eine große Sammlung von Tests kann einem aber die Sicherheit geben, dass wenigstens die getesteten Fehler nicht auftreten. Da diese Tests jederzeit ausgeführt werden können, kann man auch zu jedem Zeitpunkt einen Überblick über den Zustand der Software erhalten.

Ein anderes hinreichend bekanntes Problem ist, dass Debugging viel Zeit kostet. Debugging dient dem Auffinden von Stellen, an denen ein Fehler auftritt. Beim Debugging wird die Fehlerregion zunächst grob eingegrenzt und danach schrittweise Programmcode analysiert, wobei es nicht unüblich ist, dass es sich bei der groben Eingrenzung um mehrere Klassen und Unmengen an Quellcodezeilen handelt. Dazu werden entweder Debugging-Ausgaben oder spezielle Programme (Debugger) verwendet. Dieser Vorgang ist deshalb so zeitaufwändig, weil sich Änderungen an einer Stelle im Quellcode oft auf scheinbar völlig unabhängige Systemteile auswirken können und dies auf Anhieb nicht nachvollziehbar ist. Ein großes Repertoire von Tests kann helfen, die betroffenen Stellen sehr schnell einzugrenzen.

Ein weiteres Problem betrifft Softwareprojekte, die sich über einen längeren Zeitraum erstrecken. Bei diesen Projekten wird es nach einiger Zeit immer schwieriger, neue Funktionalität zu implementieren. Während der Programmierungsphase stellt sich heraus, dass vom ursprünglich geplanten Design abgewichen werden muss. Es ist in dieser Phase sehr schwer, schon vorhandenen Code zu ändern, da die Angst besteht dadurch alte Funktionalität zu zerstören.

Aus diesen Gründen ist es nur sehr schwer möglich, auf geänderte Anforderungen des Kunden oder der Programmierabteilung einzugehen bzw. diese umzusetzen.
Bei testgetriebener Entwicklung besteht ein Ziel darin, dass Software zu jeder Zeit mit möglichst geringem Aufwand änderbar ist.

1.2 Was ist testgetriebene Entwicklung?

Bei der testgetriebenen Entwicklung (engl.: „test-driven development“ oder „test first development“) werden zu erfüllende Anforderungen in Form von Tests formuliert. Der ursprüngliche Ablauf der Programmierphasen wird umgekehrt und das Testen befindet sich nicht mehr am Ende des Programmierens, sondern ist das Erste, was ein Programmierer macht. Zunächst existiert der Test; bevor das Ziel verfolgt wird, diesen Test erfolgreich zu bestehen. Daher schlägt dieser immer erst einmal fehl, da noch kein Programmcode existiert.

Dadurch, dass Tests auf verschiedenen Ebenen der Softwareentwicklung durchgeführt werden können, können sie auch genutzt werden, um die Entwicklung auf verschiedenen Ebenen voranzutreiben. Daraus ergibt sich, dass Tests auch von verschiedenen Personen, die an Entwicklungsprozessen beteiligt sind geschrieben werden können.

So können zum Beispiel Anforderungen, die ein Kunde an eine Software stellt, in Form von Akzeptanztests festgehalten werden. Im Idealfall schreibt der Kunde konkrete Benutzungsbeispiele auf, die dann als Akzeptanztests betrachtet werden und automatisiert durchgeführt werden. Dem Programmierer können diese Tests bei der Entwicklung zeigen, wie weit das Programm schon die Kundenanforderungen erfüllt und welche Funktionalität noch implementiert werden muss. Laufen alle Tests erfolgreich durch, sind die Kundenanforderungen erfüllt. Wenn die gesamte Anforderungsdefinition in Form von Tests formuliert wurde, kann man das Programm dann als fertig betrachten.

Tests, die direkt vom Programmierer ausgeführt werden, sind Komponententests. Diese testen jeweils eine einzelne Einheit unabhängige vom Gesamtsystem. Dabei können diese Einheiten verschieden groß sein. Es können Klassen, Ketten von Klassen, oder sogar Module sein. Zunächst wird eine Klasse isoliert gegen ihre öffentliche Schnittstelle getestet. Wenn diese Tests erfolgreich sind, wird die Klasse mit ihrer direkten Umgebung betrachte. Dabei betrachtet man die mit ihr assoziierten Klassen und erhält so Klassen-Ketten, die als nächstes getestet werden. Nachdem mehrere solcher Kettentests durchgeführt wurden, betrachtet man die Klassen die zusammen ein Modul bilden als eine zu testende Einheit. (siehe Abbildung 1)

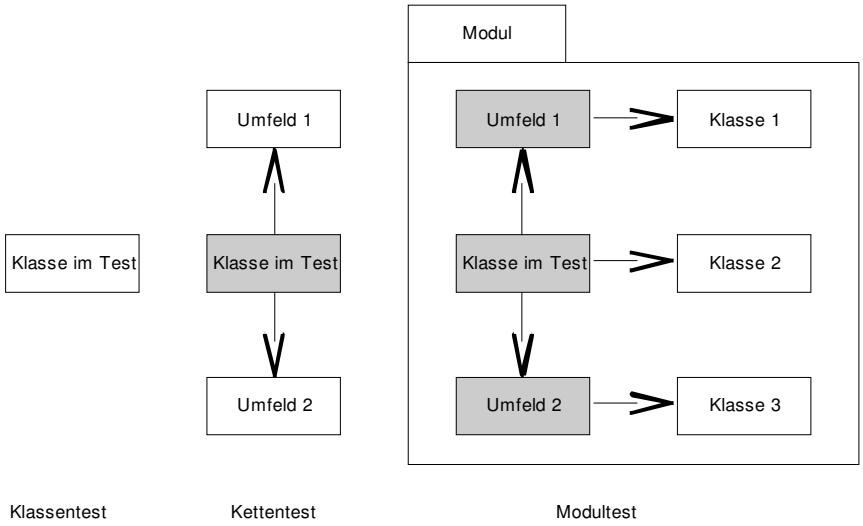


Abbildung 1: Komponententest Einheiten

2 Einführendes Beispiel

Um das Konzept und die Ideen der Testgetriebenen Entwicklung und vor allem die Umsetzung in Java mit dem Framework JUnit 3.8 zu verdeutlichen, hält sich das Skript an ein Beispiel, um die Anforderungen an den Programmierer und die Umsetzungen in die Praxis transparenter zu machen.

In der Java-Klasse Bisektion.java wird mit Hilfe des Bisektionsverfahrens eine Nullstellen Näherung der Sinusfunktion in einem Intervall bestimmt, welches der Benutzer vorgibt.

Damit das Bisektionsverfahren ein brauchbares Ergebnis liefern kann, müssen die von dem Benutzer vorgegebenen Intervallpunkte a,b die Bedingung

$$f(a) * f(b) < 0$$

erfüllen.

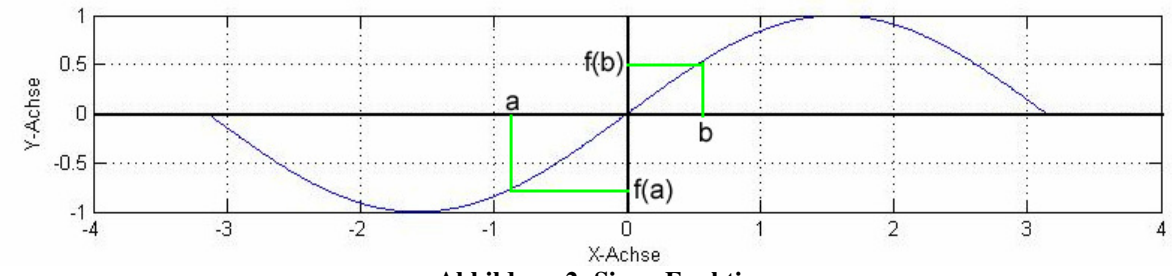


Abbildung 2: Sinus-Funktion

Das Bisektionsverfahren berechnet die Nullstelle einer Funktion, indem das Intervall in geeigneter Form halbiert wird, bis die Nullstelle näherungsweise gefunden wurde.

Die Vorschrift, nach der das Intervall halbiert und die neuen Intervallgrenzen berechnet werden, wird wie folgt beschrieben:

$$a < b \cap f(a) * f(b) < 0$$

$$f(\frac{b-a}{2}) < 0 \cap f(a) < 0 \Rightarrow a = \frac{b-a}{2}$$
$$f(\frac{b-a}{2}) < 0 \cap f(a) > 0 \Rightarrow b = \frac{b-a}{2}$$
$$f(\frac{b-a}{2}) > 0 \cap f(a) < 0 \Rightarrow b = \frac{b-a}{2}$$
$$f(\frac{b-a}{2}) > 0 \cap f(a) > 0 \Rightarrow a = \frac{b-a}{2}$$

Es gibt zwei Abbruchkriterien für das Bisektionsverfahren, welche der Benutzer bestimmen kann. Zum einen die Anzahl der Intervallteilungen und zum anderen der Abstand der beiden Intervallgrenzen a, b. Das Programm belegt die Anzahl der Intervallteilungen standardmäßig mit 10 und den Abstand der Intervallgrenzen mit 0.00000000001.

Die Funktion $f(x)$ ist in diesem Fall eine Näherung der Sinusfunktion

$$f(x) = \sin(x)$$

Zur Bestimmung des Funktionswert von $\sin(x)$ wird nicht die in Java vorgegebene Methode, sondern eine Näherung verwendet, die in der Java-Klasse Sinus.java implementiert wurde. Der Benutzer erhält die Option, die obere Grenze selbstständig setzen zu können.

$$\sin(x) = \sum_{i=0}^{n} (-1)^i \frac{x^{(2i+1)}}{(2i+1)!} \ , \ n \in \Re^+$$

Lässt der Benutzer diese Möglichkeit aus, so wird für die obere Grenze der Standardwert 10 durch das Programm gesetzt.

Zu den Java-Klassen Bisektion.java und Sinus.java gibt es die jeweiligen Testklassen BisektionTest.java und SinusTest.java in denen verschiedenste Testfallkriterien implementiert wurden.

Die durch die Testklassen berechnete Nullstelle wird in einer Datei gespeichert und mit Hilfe der Klasse Datenvergleich.java mit dem exakten Wert verglichen.

Die Testklassen wurden in der TestSuite-Klasse TestAll.java zusammengeführt, so dass durch den Aufruf der Klasse TestAll.java die Tests SinusTest.java und BisektionTest.java nacheinander ausgeführt werden.

3 Testgrundlagen

Die in diesem Kapitel beschriebenen Grundlagen beschränken sich nicht auf die Testgetriebene Entwicklung. Sie beschreiben allgemeingültige Vorgehensweisen zum erzeugen von Testsfällen. Da die Testgetriebene Entwicklung das Vorgehen beim Testen beschreibt und nicht die Art der Testfall Findung, sind diese Grundlagen Voraussetzung um gute Tests zu schreiben. Testgetriebene Entwicklung bringt erst mit guten Tests Vorteile.

3.1 White-Box-Test

Der White-Box-Test ist ein Testverfahren, bei dem die Funktionalität auf Grundlage der Kenntnisse der konkreten Implementierung einer Klasse geprüft wird. Der Tester benötigt dazu also den Quellcode der zu entwickelnden Software. Meistens ist es der Programmierer selbst, der diesen Test schreibt, da dieser einen sehr guten Überblick über den Quellcode hat. Die Kenntnisse der Implementierung wird genutzt, um möglicherweise auftretende Fehler aufzuzeigen.

3.1.1 Codeabdeckung

Ein Maß um die Qualität einer Testsammlung zu beurteilen ist die Menge des Programmcodes der von ihr getestet wird. Wenn man die Codeabdeckung betrachtet, muss zwischen folgenden Arten unterschieden werden:

- Anweisungsüberdeckung
- Zweigüberdeckung
- Pfadüberdeckung

Bei der Anweisungsüberdeckung wird nur das Durchlaufen jeder Anweisung betrachtet. Dabei wird zum Beispiel bei einer if-Anweisung mit leerem else-Zweig nicht verlangt, dass dieser durchlaufen wird. Das ist jedoch bei der Zweigüberdeckung notwendig. Jede mögliche Verzweigung muss dann durchlaufen werden. Die Pfadüberdeckung betrachtet jeden möglichen Pfad durch ein Programm als verschieden. Das bedeutet, dass eine Schleife, die bei jedem Durchlauf verlassen werden kann, jeweils einen neuen Pfad darstellt abhängig davon wie oft sie durchlaufen wurde.

Die Anzahl der möglichen Pfade ist also unüberschaubar groß, weshalb ein Testen auf Ebene der Pfadüberdeckung nicht praktikabel ist.

```java
public class Beispiel {
    public void ueberdeckung(int a, int b, int c) {
        if (a < b) {
            System.out.println("a ist kleiner als b");
        }
        if (b < c) {
            System.out.println("b ist kleiner als c");
        }
    }
}
```

Listing 1: Beispiel Codeabdeckung

In diesem Beispiel würde ein Aufruf der Funktion ueberdeckung() mit den Parametern a=1, b=2 und c=3 für die Anweisungsüberdeckung ausreichen.

Für die Zweigüberdeckung müssen auch die nicht vorhandenen else-Zweige durchlaufen werden. Deshalb müsste ein zweiter Aufruf mit a=3, b=2 und c=1 durchgeführt werden.

Für die Pfadüberdeckung würde selbst das nicht genügen, denn dafür muss jede mögliche Kombination von Zweigen durchlaufen werden. Es müssen also noch zwei zusätzliche Aufrufe getestet werden, wie zum Beispiel a=2, b=1 und c=3 und a=1, b=3 und c=2.

Die Abbildung 3 zeigt welche Durchläufe für welche Art der Überdeckung notwendig sind. Der jeweils durchlaufene Pfad ist dabei in rot gekennzeichnet.

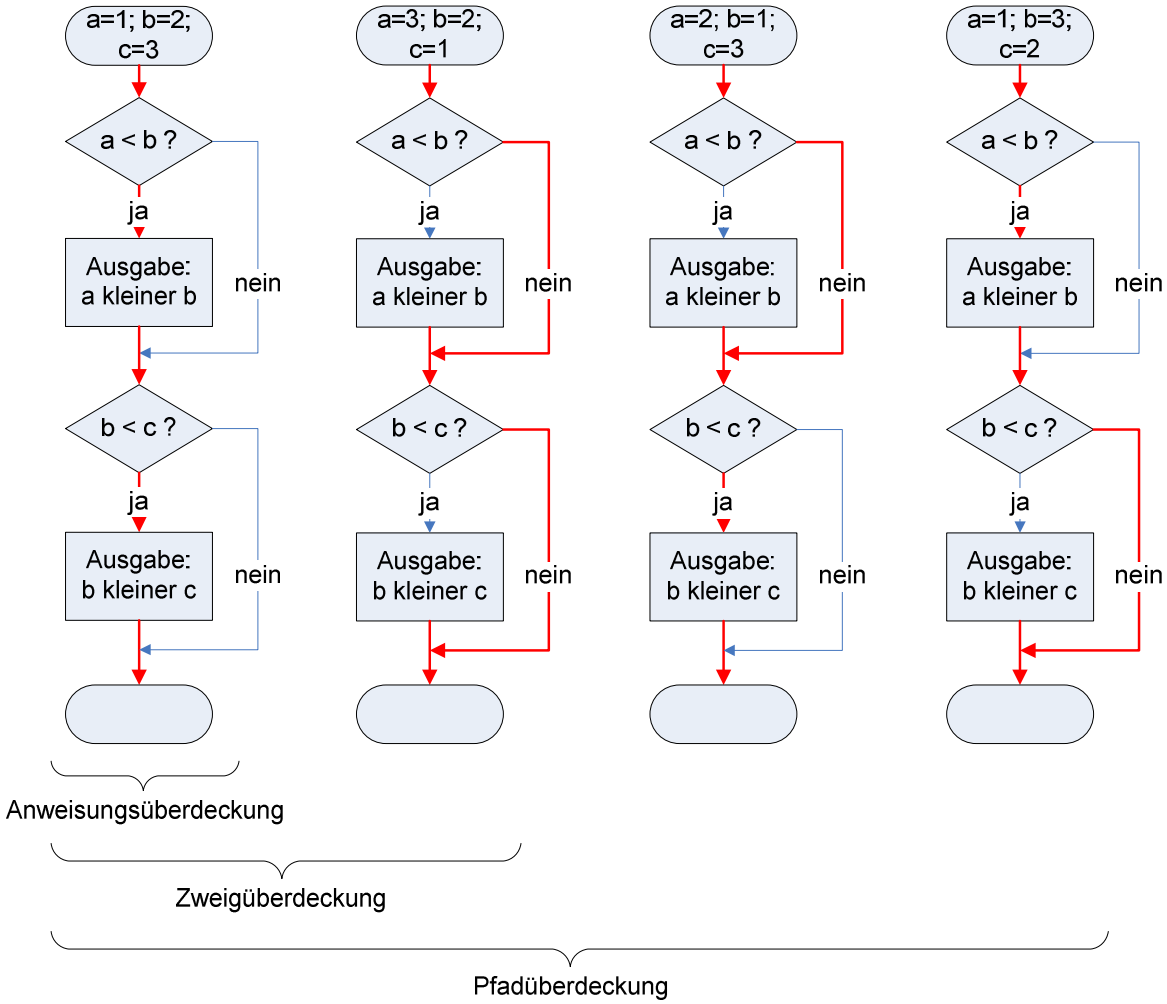


Abbildung 3: Flussdiagramm Codeabdeckung

Für Schleifen gibt es ein weiteres vereinfachendes Verfahren, dass mit hoher Wahrscheinlichkeit ausreicht, um Fehler darin zu finden. Dazu wird nicht jede mögliche Anzahl von Durchläufen getestet, sondern nur eine Auswahl von fünf Fällen.

1. Kein Eintritt in die Schleife
2. Genau ein Durchlauf durch die Schleife
3. Genau zwei Durchläufe durch die Schleife
4. Eine typische Anzahl von Durchläufen
5. Die maximale Anzahl von Durchläufen

Um die Codeabdeckung der eigenen Testsammlung zu messen, gibt es Programme, die den Anteil von getestetem Programmcode bestimmen. Die Ausgabe eines solchen Coverage-Analyzers ist in Abbildung 4 zu sehen. Hier wird die Codeabdeckung des gesamten Projekts, der einzelnen Pakete und jeder einzelnen Datei in Prozent angezeigt. Diese Ausgabe kann einem helfen, Programmteile zu finden die noch nicht ausreichend getestet werden.

Diesem Ergebnis sollte jedoch nicht zu viel Bedeutung zugemessen werden. Es ist nicht notwendig eine 100-Prozentige Codeabdeckung zu erreichen. Methoden, die keine eigene Logik enthalten, wie zum Beispiel get- und set-Methoden, brauchen nicht getestet werden. Im Allgemeinen kann man die Regel anwenden: „Es muss nur getestet werden, was auch schief gehen kann“. Außerdem kann ein, von einem Test erfolgreich durchlaufener Programmteil, dennoch Fehler enthalten. Dann würde einem die hohe Codeabdeckung eine trügerische Sicherheit geben.

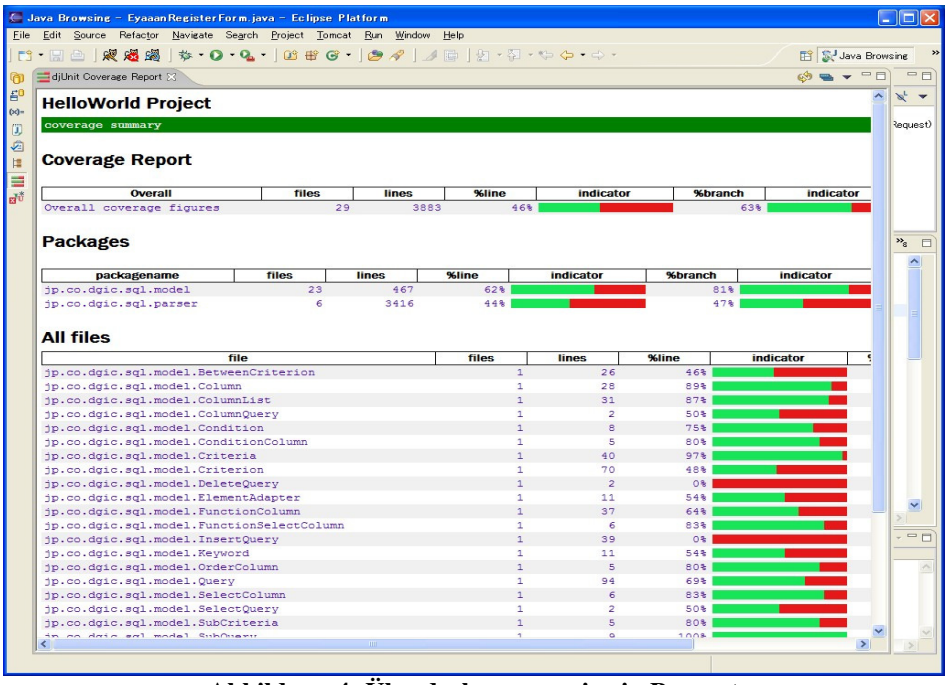


Abbildung 4: Überdeckungsanzeige in Prozent

3.2 Black-Box-Test

Bei dem Black-Box-Test ist nur die Schnittstelle eines Objektes bekannt. Es werden keine Annahmen über die Implementierung der Komponenten gemacht, sondern nur anhand der Spezifikation getestet. Diese Tests kann und sollte von jemandem geschrieben, der den Quellcode nicht geschrieben hat, um zu vermeiden das Annahmen über die Implementierung gemacht werden.

3.2.1 Grenz- und Extremwerte

Eine große Anzahl von Testfällen alleine reicht nicht aus, um aussagekräftige Ergebnisse zu erhalten. Gute Tests zeichnen sich dadurch aus, dass sie möglichst viele Fälle abdecken, die auftreten können. Es ist hingegen nicht notwendig und erst recht nicht effizient, Testfälle zu wählen, die von der Software gleich behandelt werden.

Besonders aussagekräftig sind Testfälle, die Grenz- und Extremwerte benutzen. Grenzwerte sind Werte die sich möglichst nahe beieinander liegen, aber ein unterschiedliches Verhalten in der Software auslösen.

Extremwerte sind Werte an den Rändern von Definitionsbereichen. Sie werden oft gesondert behandelt und sollten deshalb auf jeden Fall getestet werden, da Programmabstürze oft mit diesen Werten zusammenhängen und dies dringend vermieden werden muss.

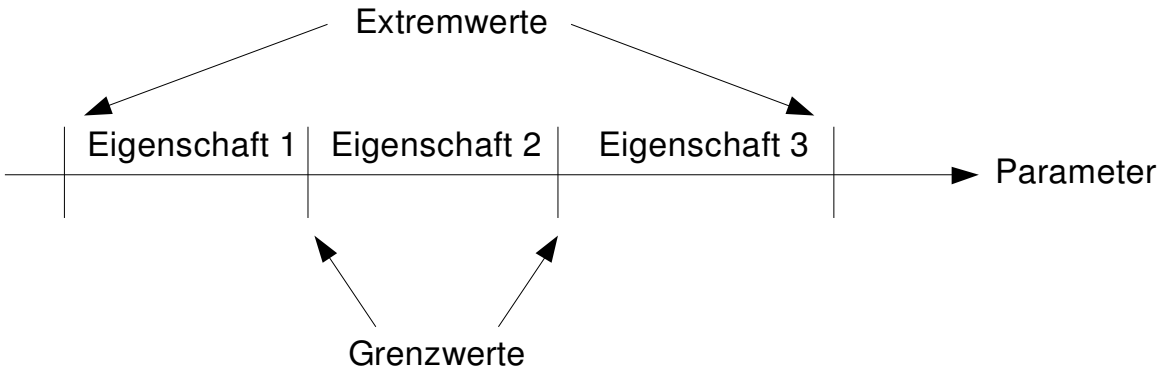


Abbildung 5: Grenz- und Extremwerte

3.2.2 Äquivalenzklassen

Um nicht eine unüberschaubare Menge von Tests zu erhalten und deshalb möglicherweise wichtige zu übersehen, ist es sinnvoll so genannte Äquivalenzklassen zu bilden.

Äquivalenzklassen sind Gruppen von Eingaben, die von der zu testenden Komponente gleich behandelt werden. Aus jeder dieser Klassen wird nur ein Testfall gebraucht, weil alle Elemente dieser Klasse das gleiche Verhalten hervorrufen. Es können aber auch mehrere Werte aus einer Äquivalenzklasse getestet werden, nämlich dann wenn es sinnvoll scheint, die Ränder, also Extremwerte, zu berücksichtigen.

Fast immer gibt es die Äquivalenzklassen der gültigen und der ungültigen Eingaben. Meistens müssen diese jedoch noch weiter unterteilt werden, da gültige Eingaben unterschiedlich verarbeitet werden können und ungültige Eingaben auch unterschiedliche Fehler hervorrufen können..

3.3 Grey-Box-Test

Grey-Box-Tests sind eine Mischung der beiden vorherigen Testarten. Sie treten häufig bei Komponententests auf. Diese Tests sind zunächst Black-Box-Tests, da sie vor der Implementierung des eigentlichen Quellcodes geschrieben werden und zu diesem Zeitpunkt nur die öffentlichen Schnittstellen bekannt sind. Hier ist die Implementierung noch unklar und kann sich auch noch mehrmals ändern. Wenn dann grundlegender Code existiert, sollten die weiteren Komponententests White-Box-Tests sein. Da die Implementierung nun bekannt ist, können die Methoden des White-Box-Tests angewandt werden um noch fehlende Aspekte der konkreten Implementierung aufzuzeigen. Die zuvor geschriebene Black-Box-Testklasse wird so erweitert, dass verschiedene Programmanweisungen durchlaufen und getestet werden.

4 Unit-Tests

In der Vergangenheit wurden Tests und Software, während der Erstellung durch den Programmierer, als ein gemeinsames Packet angesehen, wobei Tests als zeitraubend, aber notwendig, galten. Eine neue Idee der Softwareerstellung gibt vor, dass Tests von der eigentlichen Software isoliert behandelt werden. In diesem Kapitel wird der Isolierte Test beschrieben und erläutert, warum der bisher gängige Ablauf, den Test nach dem Quellcode zu schreiben, verworfen werden kann und eine Alternative vorgestellt.

4.1 Isolierte Tests

Die grundsätzliche Idee ist, Produktions-Code, welcher zur verkaufsfertigen Software gehört, und Test-Code, welcher während der Erstellung der Software generiert wird und lediglich zur Überprüfung des Produktions-Codes benötigt wird, voneinander zu trennen. Bei Unit-Tests werden kleine Programmteile wie Klassen, Komponenten oder Module isoliert von dem Rest des Programms getestet. Die Ausführung der Tests kann direkt nach der Implementierung von neuem Produktions-Code geschehen. Die Tests laufen eigenständig, da sie unabhängig von dem Produktions-Code kompiliert und ausgeführt werden können. Es ist nicht notwendig, das ganze Projekt, mit allen vorhandenen Klassen des Produktions- und Test-Codes, zu kompilieren, um einen Test neu zu Erstellen. Die Tests erhalten ihr eigenes Framework. Für die Programmiersprache Java bietet sich das Framework JUnit (Seite 15) an, welches umfangreiche Funktionalitäten bietet, um Tests automatisiert und selbstständig ablaufen zu lassen.
Unit-Tests können nach jeder Modifikation ausgeführt werden. Für jede Programmklasse sollte es mindestens eine Testklasse geben. In einer Testklasse sollten nur Methoden vorhanden sein, welche eine gemeinsame Klasse Testen. Werden in einer Testklasse zwei Programm-Code Klassen getestet, so empfiehlt es sich, diese Testklasse in zwei neue Testklassen zu spalten, da sonst der Überblick über die Testklassen verloren geht und ein komplexes effizientes zusammenstellen der Test verhindert wird.

4.2 Testgetriebenes Programmieren

Testgetriebenes Programmieren (auch testgesteuerte Programmierung, engl. Test-First Development oder Test-Driven Development, Abkürzung TDD) bedeutet, dass der Programmierer Testklassen bereits vor den zu testenden Klassen schreibt und somit die ursprüngliche Reihenfolge des Entwicklungsablaufs vertauscht wird. Testen findet nicht mehr nach oder während der Erzeugung des Produktions-Codes, sondern vor der Erzeugung des Produktions-Codes statt.

Der Ablauf dieser Art zu programmieren kann in drei Teile gegliedert werden:

1. Es wird ein Test geschrieben, welcher den zu schreibenden Programmcode effektiv testet, jedoch zunächst fehlschlägt, da die dazu gehörenden Programmzeilen noch fehlen.
2. Der Programmcode wird so weit generiert, dass wenigstens kompiliert werden kann, jedoch schlägt der Test immer noch fehl.
3. Der Programmcode wird so weit vorangetrieben, bis der dazu passende Test nicht mehr fehlschlägt und davon ausgegangen werden kann, dass der Programmcode keine Fehler enthält.

Der Vorteil dieses Ablaufs liegt in dem direkten Feedback des geschriebenen Codes. Zu jeder Zeit kann geprüft werden, ob der geschriebene Code funktioniert oder ob durch Veränderungen an beliebigen Stellen zuvor fehlerfreier Code auf einmal fehlschlägt. Die Auswirkung einer im Nachhinein veränderten Basisklasse, kann mit Hilfe eines Durchlaufs aller Testklassen erkannt und behoben werden. Nach Beendigung eines wichtigen Programmteils wie einer Schleife, Bedingung, Methode oder Klasse kann der Unit-Test Aufschluss darüber geben, ob korrekt programmiert wurde, ohne darauf zu warten, dass der gesamte Programmcode geschrieben wurde.
Werden die Tests nach der Implementierung des Produktions-Codes entwickelt, kann das Problem auftreten, dass die Software nur schwer testbar ist, da während der Programmierung keine Rücksicht auf Tests genommen wurde. Der Grund liegt meist an einer fehlerhaften Strukturierung oder an der mangelnden Anzahl öffentlicher Methoden, die es dem Programmierer an verschiedenen Stellen einer Klasse erlauben, einen Statusbericht oder ein zwischen Ergebnis zu erfragen. Dieses Problem entfällt bei der Testgetriebenen Entwicklung, da die Testklassen bereits existieren, bevor der Programmierer mit der Erzeugung des Programm-Codes beginnt und somit jede Klasse testbar ist.

Während des gesamten Programmiervorgangs wird in möglichst kleinen Schritten vorgegangen, so dass Fehler sofort und ohne lange Suche lokalisiert und behoben werden können und fehlerfreier Produktions-Code die Folge ist. Das zeitaufwendige Suchen von Fehlern in unübersichtlichen Quellcodezeilen wird auf ein Minimum reduziert, was die Freude am Programmieren erhält und das Testen nicht mehr zu einem notwendigen übel werden lässt. Wie in dem Zitat von Frank Westphal nachzulesen, gibt ein erfolgreicher Testlauf in der Endphase eines großen Projektes dem Programmierer die Sicherheit die er braucht, um das Projekt zu verbessern und zu erweitern.

„*Sie werden sich später ganz sicher mal dabei ertappen, die Tests zwei- oder dreimal nacheinander auszuführen, nur wegen des zusätzlichen Vertrauens, wenn hunderte von Tests ablaufen und der Fortschrittbalken dabei langsam mit Grün voll läuft*.“

[1]

Die Gefahr besteht jedoch, dass den Testklassen zu viel Vertrauen geschenkt wird und somit während der Produktions-Code Erzeugung nicht gewissenhaft und möglichst fehlerfrei programmiert wird.

4.3 Testplan

Zu Beginn eines Projektes sollte über die Zusammensetzung der Software diskutiert werden. Welche Klassen werden mit welchen Methoden benötigt? Der Vorteil von Test-First Entwicklungen ist, dass die Diskussion über Schnittstellen oder die Frage nach genügend public-Methoden zum Testen der Software entfallen und dadurch Zeit gespart wird.

Zwei wichtige Fragen sind:
- wie ist die Reihenfolge der Tests in objektorientierten Programmen aufzubauen
- wie muss ich diese Programme aufbauen, damit die Tests erfolgreich laufen können

Zu Beginn des Projektes, noch vor dem Schreiben von Testfällen und Produktions-Code, wird über Grenzfälle und Extremfälle diskutiert, jedoch auch über Sinn oder Unsinn von Testmethoden und ob schon am Anfang Testfälle zusammengefasst werden können. Es wird ein genauer Plan erstellt, wie die Testklassen auszusehen haben und welchen Inhalt die Testmethoden haben müssen. Ebenfalls wird darüber diskutiert, ob die Testklasse das Programm oder eine Spezifikation, wie das Eingabe- oder Berechnungsmodul, testen soll.

Ein wichtiger Punkt, ist die Beschaffung von geeigneten Testdaten. Um eine Klasse zu testen, muss nicht nur die Eingabe, sondern auch das Ergebnis zuvor bekannt sein. Testdaten umfassen somit die Ein- und Ausgabedaten einer Testklasse. Wenn, wie im Beispiel zu sehen, eine mathematische Näherung berechnet wird, so muss ein Vergleichwert existieren, um die Genauigkeit des Ergebnisses zu klassifizieren. Die Schwierigkeit liegt in dem Finden solcher Vergleichswerte, da manche mathematischen Ergebnisdaten wiederum nur Näherungen sind. Zudem ist das Design der Testdaten zu diskutieren. Werden zum Beispiel die Testdaten direkt in die Klasse der Testsoftware geschrieben oder etwa in separate Testdateien ausgelagert und bei Bedarf eingelesen? Wenn die Testdaten ausgelagert werden, wie muss die Datei aufgebaut sein?

Eine sinnvolle Arbeit ist das Erstellen von Testdokumenten. Testdokumentationen beinhalten eine Vielzahl hilfreicher Informationen, damit Außenstehende einen direkten Einstieg in die Testklasse und somit auch in das Programm haben.

Eine gekürzte Zusammenfassung der Testdokumentation nach ANSI/IEEE 829:
- Testplan
  - Eindeutige Testfall ID
  - Zu testende und nicht zu testende Komponenten und Funktionen
  - Vorgehensweise
  - Testtätigkeiten und Testumgebung
- Testskript
  - Testziel
  - Spezielle Anforderungen
  - Einzelschritte
- Testfall
  - Eingaben und Ausgaben
  - Besonderheiten
  - Abhängigkeiten

Diese Abarbeitung der Dokumentation kann bei späteren Problemen schneller zu einer Lösung führen.

Sind all diese Fragen geklärt, bleibt die Frage, welche Hilfsmittel für das Projekt verwendet werden sollen, denn das Java Framework JUnit ist nur eines von vielen möglichen Werkzeugen. Hilfsmittel für Codeabdeckung und Refactoring sowie einige mehr sind nützliche Tools, welche die Effektivität der Tests steigern können.

5 JUnit als Java Framework für Unit-Tests

JUnit ist ein Java Open-Source-Framework, mit dem automatisierte Tests geschrieben und ausgeführt werden können. Es wurde von Kent Beck und Erich Gamma entwickelt. Die Tests werden in Java codiert und sind wiederholbar. Zu JUnit gehört eine Bedienungsoberfläche, womit die Ausführung der Tests vereinfacht wird. JUnit überprüft die Tests selbstständig auf Ausführbarkeit, da die Oberfläche und Laufzeitmethoden bereits im Framework lauffähig existieren und der Programmierer lediglich Testmethoden hinzufügt. Das Framework ist nicht nur für Java erhältlich, sondern ebenfalls für andere Programmiersprachen wie C, C++, Delphi, C#, Visual Basic .NET, Visual J# und viele mehr.

Hier wird JUnit Version 3.8 vorgestellt [3].

Um den Einstieg in JUnit zu vereinfachen, wird die Theorie an einfachen praktischen Beispiel Dateien erklärt. Die Dateien sind diesem Skript beigelegt und umfassen:

- Bisektion.java
- BisektionTest.java
- Dateivergleich.java
- IntervallException.java
- Sinus.java
- SinusTest.java
- TestAll.java

5.1 Inhalt des Frameworks

Um die Vorteile von JUnit zu nutzen, wird, wie in Listing 2 verdeutlicht, die zu schreibende Testklasse von der JUnit Framework-Basisklasse TestCase abgeleitet.

```
1 import junit.framework.TestCase;
2
3 public class SinusTest extends TestCase
4 {
5   public SinusTest(String name){
6     super(name);
7   }
8 }
```

Listing 2: Ableitungsvorschrift für Testklassen

Die in dieser Klasse vorhandenen Testmethoden müssen, wie in Listing 3 hervorgehoben, mit dem Wort „test“ beginnen, da das Framework die Testmethoden automatisch erkennt und anspricht.

```
public void test...(){
}
```

Listing 3: Vorgabe für Testmethoden

5.1.1 Überprüfungen mittels Assert-Methoden

Die eigentlichen Vergleiche bzw. Tests, werden mittels Assert-Methode durchgeführt, die jede Testklasse durch die Klasse TestCase geerbt hat. Diese Methoden erlauben es, Behauptungen über die zu testende Klasse aufzustellen und zu beweisen.

Es gibt 7 Assert-Methoden, wobei manche Methoden durch das Framework überladen wurden, damit der Programmierer sie über unterschiedliche Parameter aufrufen kann.

- assertTrue(boolean)
- assertFalse(boolean)
- assertEquals(Objekt, Objekt), aber auch andere Parameter wie *String*, *int*, *double*, *float*, *boolean*, *byte*, *char* oder *short* sind zugelassen.
- assertNull(Objekt)
- assertNotNull(Objekt)
- assertSame(Objekt, Objekt)
- assertNotSame(Objekt, Objekt)

Mit Hilfe dieser Methoden kann beliebig viel getestet werden. Es ist möglich, zu berechnende Ergebnisse mit dem exakten Wert zu vergleichen (Listing 4, Zeile 5).

```
1 public static void testNstBerechnung()throws Exception
2 {
3   Bisektion bisektion = new Bisektion(-1,1);
4
5   assertEquals(0,bisektion.getNullstelle());
6 }
```

Listing 4: Beispiel für assertEquals

Assert-Methoden können auch als Zusicherung bezeichnet werden, da, wie in Beispiel für Zusicherungen beschrieben, Angaben über die Zustände von Variablen getroffen werden können.

```
1 public final void testImplementierung() throws Exception
2 {
3
4   // Gültige Sinus Konstruktor Aufrufe
5   Sinus sin = null;
6   assertNull(sin);
7
8   sin = new Sinus(0);
9   assertNotNull(sin);
10  sin = null;
11 }
```

Listing 5: Beispiel für Zusicherungen

Sollte eine Assert Methode nicht zu dem erwarteten Ergebnis führen, so bricht JUnit den Testfall ab und wirft den Fehler AssertionFailedError mit entsprechender Fehlermeldung (Abbildung 6: JUnit Fehlermeldung).

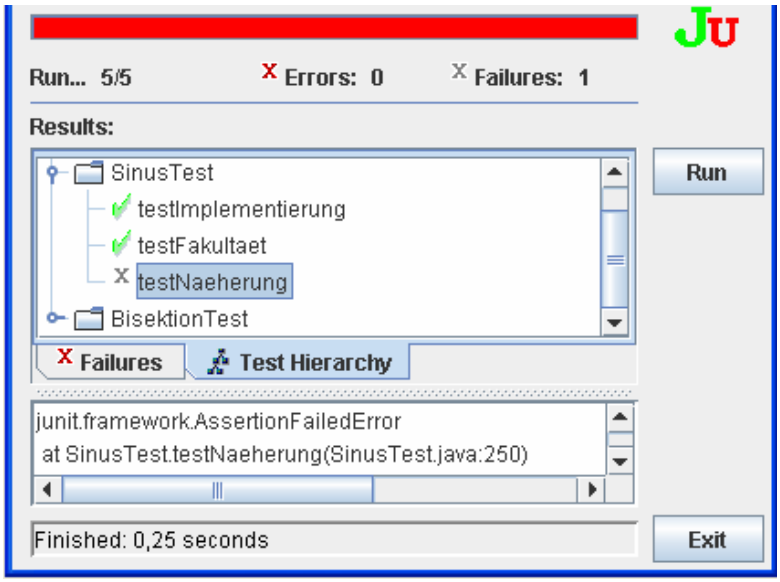


Abbildung 6: JUnit Fehlermeldung

Um mehr Informationen aus der Fehlermeldung zu erhalten, kann jeder Assert-Methode ein *String* übergeben werden, so dass jede Fehlermeldung eindeutig zu zuordnen ist. Die Parameter der Assert-Methode bleiben gleich, jedoch ist der erste Parameter ein *String Objekt* und die üblichen folgen wie gewohnt.

Beispiel:
assertEquals(byte, byte) bzw. assertEquals(String, byte, byte)

5.1.2 Exception Handling in JUnit

Es gibt in JUnit nicht nur die Möglichkeit, bei richtiger Eingabe Ergebnisse miteinander zu vergleichen, sondern auch, erwartete Exceptions, welche bei falscher Eingabe oder Berechnung von Klassen geworfen werden sollen, zu testen.

Eine übliche Vorgehensweise bei zu erwartenden Exceptions ist die in Listing 6 gezeigte. Fakultäten können nur von Positiven Zahlen berechnet werden und somit muss der Aufruf in Listing 6 Zeile 8 zu einem Fehler führen. Wird der Fehler durch die Methode nicht erkannt und keine Exception geworfen, so wird der catch-Block in Zeile 11 nicht angesprochen. Die fail-Methode in Zeile 10 wird von JUnit erkannt und führt zu einem Abbruch der Methode testFakultaet mit entsprechender Fehlermeldung.

```
1  public final void testFakultaet() throws Exception
2  {
3     /*
4     * Eine negative Fakultät kann nicht berechnet werden und
5     * muss von der Methode erkannt werden.
6     */
7    try{
8      Sinus.fakultaet(-2);
9      // Wenn keine Exception geworfen wurde, so ist dies ein Fehler
10     fail(this.getName()+" fehlgeschlagen");
11   }catch(Exception expected){
12     // Wird dieser Teil angesprochen, wurde die Exception geworfen
13     // und der Fehler erkannt
14   }
15 }
```

Listing 6: Exception Handling in JUnit

Eine Exception, die nicht durch den Programmierer, wie in Listing 6 gezeigt, abgefangen wird ist unerwartet und wird daher unerwartete Exception genannt.

Bei unerwarteten Exceptions wird die Testmethode abgebrochen und der Fehler von JUnit protokolliert, wobei JUnit bei einem Abbruch generell zwischen Failure und Error unterscheidet.

- Failure: eine Zusicherung, wie eine Assert-Methode, wurde während des Tests verletzt bzw. nicht erfüllt. In der Ausgabe wird die, wie in Abbildung 7 zu sehen, Grau gekennzeichnet.
- Error: eine unerwartete Exception, wie eine ArrayIndexOutOfBoundsException (Abbildung 7), ist aufgetreten und wurde durch den Programmierer nicht abgefangen. Dies wird dem Programmierer durch ein rotes Kreuz kenntlich gemacht.

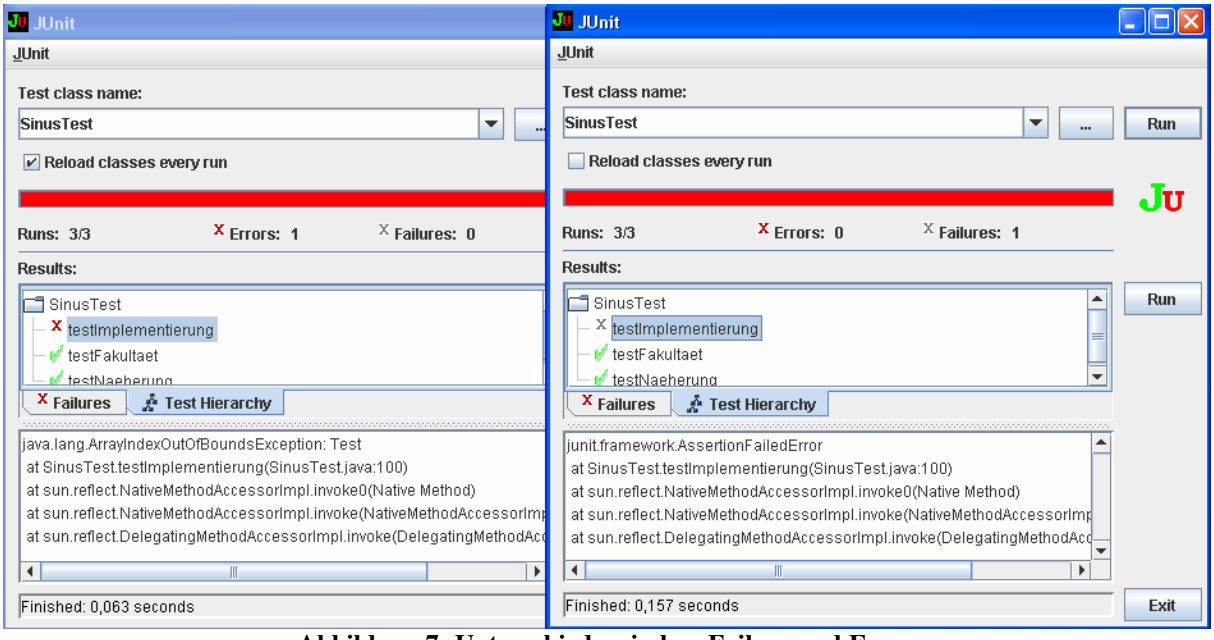


Abbildung 7: Unterschied zwischen Failure und Error

5.1.3 Ausgabe

Jede Testklasse besitzt, wie in Listing 7 zu sehen, eine main-Methode, in welcher der TestRunner run-Methode das aktuelle Testklassen class-Objekt übergeben wird (Zeile 8).

```
1  import junit.framework.TestCase;
2
3  public class SinusTest extends TestCase
4  {
5
6     public static void main(String args[])
7     {
8        junit.swingui.TestRunner.run(SinusTest.class)
9     }
10
11 }
```

Listing 7: Main-Methode der Klasse TestCase

Durch Aufruf der main-Methode erzeugt der TestRunner (Listing 7 Zeile 8) eine grafische Oberfläche, welche auf Java Swing aufbaut.

Der Entwickler wird über Erfolg oder Misserfolg von Tests über diese JUnit Oberfläche informiert, welche beim Start einer Testklasse automatisch aktiviert und fokussiert wird. In dieser Oberfläche hat der Entwickler die Möglichkeit, den Erfolg oder Misserfolg jedes einzelnen durchlaufenen Tests zu erfahren und falls notwendig, jede einzelne Fehlermeldung zu lesen. Führen alle durchlaufenden Tests zu dem erwarteten Ergebnis, so wird der Verlaufsbalken grün dargestellt. Sollte nur ein Test nicht das erwartete Ergebnis liefern, färbt sich der Balken rot. Zudem wird die Reihenfolge der durchlaufenen Tests angezeigt. Die Oberfläche kann nach Beendigung der Tests entweder mit dem Button „Exit“ geschlossen oder alle Tests über den Button „Run“ neu gestartet werden.

Es gibt drei Alternativen für die Testoberfläche:
- junit.swingui.TestRunner mit grafischer Swing Oberfläche
- junit.awtui.TestRunner auf einfacher Basis der AWT
- junit.textui.TestRunner als Batch Job auf der Textkonsole

Alle Alternativen bieten den Fortschrittsbalken, jedoch ist die Swing Oberfläche die angenehmste und übersichtlichste.

5.2 Vorteile von TestCase

In einer Testklasse gibt es zumeist mehrere Methoden, welche verschiedene Eigenschaften einer Klasse testen. Diese Methoden werden nacheinander, und unabhängig voneinander ausgeführt.

In einer Testklasse werden oft Variablen benutzt, um Ergebnisse zu vergleichen und zu überprüfen oder Dateien geöffnet, um sie auszulesen. Sollten Variablen mit gleichen Werten in mehreren Testmethoden benötigt werden, so sollten diese Variablen aus den Testmethoden herausgenommen und global als Klassenvariablen deklariert werden. In JUnit gibt es für diesen Fall die Methoden setUp und tearDown, in denen Klassenvariablen initialisiert und Dateien geöffnet, sowie geschlossen werden können.

Diese Methoden, mit den dazugehörigen Variablen, werden Fixture genannt. Jede Testmethode einer Testklasse erhält sein eigenes Fixture und nur die eigene Testmethode kann auf sein Fixture zugreifen. Sind in einer Testklasse Methoden vorhanden, die nicht auf das Fixture zugreifen, so ist es ratsam, diese Methoden aus der Testklasse zu entnehmen und in eine neue Testklasse auszulagern [1].

> *„Generell sollten Testklassen um die Fixture organisiert werden, nicht um die getestete Klasse.“*
>
> [1]

Somit ist es nicht unüblich, dass zu einer Klasse mehrere Testklassen existieren.

Jede einzelne Testmethode durchläuft denselben Zyklus und existiert nur in diesem, d.h. für jede Testmethode wird ein spezielles Test Objekt erzeugt (siehe Abbildung 8). In diesem Testfall Objekt sind alle Klassenvariablen und die in Abbildung 8 gezeigten Methoden enthalten.

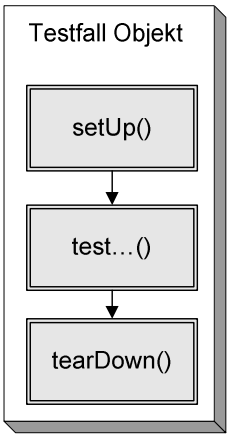


Abbildung 8: Aufbau eines Testfall Objektes

Möchte der Entwickler nach der Ausführung eines Testfalles Speicher freigeben, Dateien schließen oder ähnliches, so kann er dies in der tearDown-Methode programmieren. Der Ablauf ist stets derselbe, da zuerst die setUp-Methode ausgeführt wird, in der Variablen initialisiert werden können. Danach wird die Testmethode ausgeführt und zum Schluss wird die tearDown-Methode ausgeführt in der Speicherplatz freigegeben kann oder Dateien geschlossen werden können.

Die Testklasse SinusTest.java enthält drei verschiedene Testmethoden (siehe Abbildung 9).

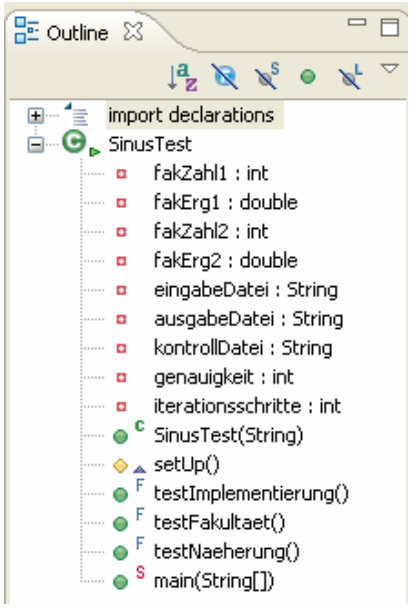


Abbildung 9: Klasse SinusTest

JUnit erzeugt, nach dem Aufruf der Klasse SinusTest, für jede Testmethode ein neues Testfall Objekt. Somit wird sichergestellt, dass jeder Test eigenständig und unabhängig von der Reihenfolge der Methoden ablaufen kann. Jeder Test baut sich, wie in Abbildung 10 zu sehen, selber auf und auch wieder ab.

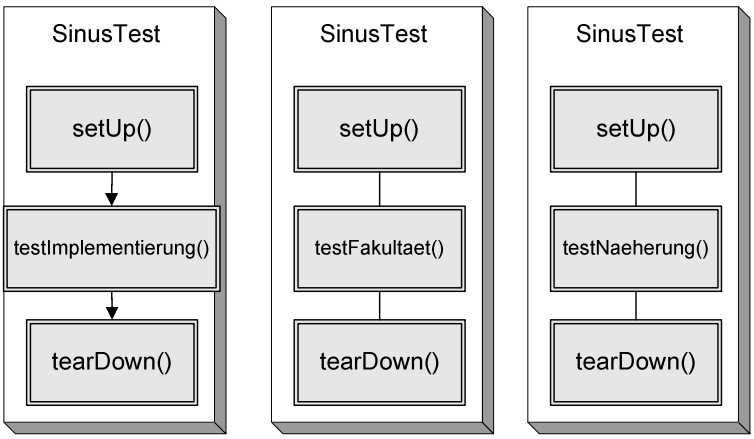


Abbildung 10: Ablauf eines Testfall Objektes

Tritt eine unerwartete Exception (5.1.2 Exception Handling in JUnit) in der setUp-Methode auf, so führt JUnit keine weiteren Methoden aus und beendet sofort das jeweilige Test Objekt. Sollte die unerwartete Exception jedoch erst in der zu testenden Methode geworfen werden, so beendet JUnit zwar den Test, führt zuvor jedoch noch die tearDown-Methode aus und bereinigt damit das komplette Test Objekt.

5.3 Zusammenfassen von Testklassen durch TestSuite

Sind nun mehrere Tests geschrieben worden, sollte es möglich sein, diese Tests zu automatisieren und mehrere Klassen, wenn nicht sogar alle, auf einmal zu testen. Bisher muss jede Testklasse separat gestartet werden und erhält ihr eigenes Ergebnisfenster, was bei vielen Tests sehr schnell unübersichtlich und mühselig wird.
In der statischen Methode suite wird dem Entwickler die Möglichkeit geboten mehrere TestCase-Klassen nacheinander in eine TestSuite-Klasse zu integrieren, sodass alle darin enthaltenen Tests nacheinander gestartet werden und das Ergebnis in einem einzigen Ergebnisfenster dargestellt wird.

Einer TestSuite-Klasse können TestCase-Klassen, aber auch TestSuite-Klassen hinzugefügt werden, wobei die Möglichkeit existiert, pro Package eine TestSuite-Klasse zu entwerfen und alle Package-TestSuite-Klassen in eine TestSuite-Klasse zu integrieren.

Um eine solche Klasse zu erzeugen, müssen folgende Eigenschaften (Listing 8: Beispiel für suite-Methode) eingehalten werden:

- Es muss eine öffentliche statische main() Methode vorhanden sein, welche der run() Methode des TestRunner Objektes die aktuelle Klasse übergibt (Listing 8, Zeile 31-34).
- Eine öffentliche statische Methode suite(), wie in Listing 8 Zeile 36-41, muss in der Klasse vorhanden sein, welche ein Test Objekt zurückliefert.
- In der suite() Methode werden alle zu testenden Klassen einem TestSuite Objekt übergeben und dem Aufrufenden Programm zurückgeliefert. In dem Beispiel aus Listing 8: Beispiel für suite-Methode werden dem suite Objekt die Klassen SinusTest.class Zeile 38 und BisektionTest.class Zeile 39 übergeben.

```
28 public class TestAll
29 {
30
31   public static void main(String[] args)
32   {
33     junit.swingui.TestRunner.run(TestAll.class);
34   }
35
36   public static Test suite() {
37     TestSuite suite = new TestSuite();
38     suite.addTestSuite(SinusTest.class);
39     suite.addTestSuite(BisektionTest.class);
40     return suite;
41   }
42
43 }
```

Listing 8: Beispiel für suite-Methode

Die suite-Methode gibt ein Test Objekt zurück (Beispiel für suite-Methode, Zeile 40), da sowohl TestCase, als auch TestSuite Objekte enthalten sein können, welche beide die Klasse Test als Interface nutzen (siehe Abbildung 11).

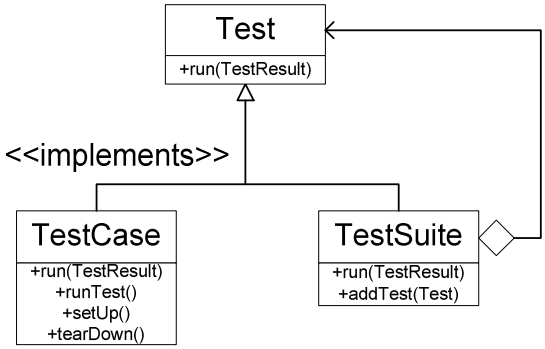


Abbildung 11: Implementierungshierarchie von Test

Da sowohl TestCase also auch TestSuite eine Run-Methode enthalten, welche durch das Interface Test vorgegeben wird, können beide Klassen dem TestRunner Objekt übergeben werden. Die suite-Methode muss daher nicht unterscheiden welche Klasse sie zurückliefern muss und wählt die Klasse Test. Diese Eigenschaft ermöglicht es nicht nur TestCase Klassen, sondern auch mehrere TestSuite Klassen zusammen zu fassen.

5.4 JUnit in Eclipse

Seit Eclipse 3.0 ist JUnit ein Bestandteil von Eclipse und muss nicht gesondert installiert werden. Das Starten von Klassen mit Hilfe von JUnit kann über zwei Optionen geschehen.

1. „Run As… - JUnit Test“
2. „Run As… - Java Application“

Bei der ersten Option wird die JUnit Oberfläche in Eclipse integriert dargestellt (siehe Integrierte JUnit Oberfläche in Eclipse).

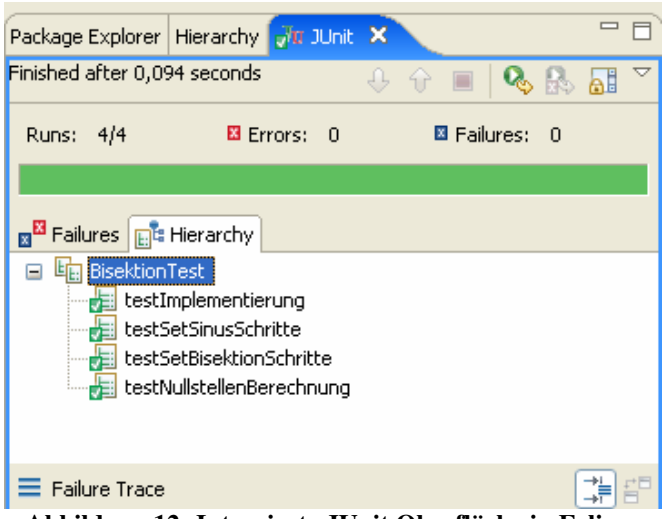


Abbildung 12: Integrierte JUnit Oberfläche in Eclipse

Die zweite Option öffnet ein separates neues Fenster außerhalb von Eclipse (siehe Abbildung 13).

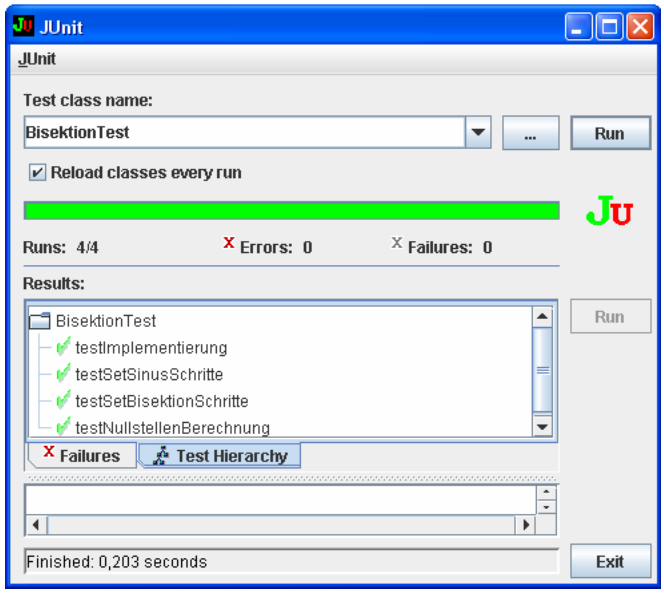


Abbildung 13: Externe JUnit Oberfläche in Eclipse

Die Erfahrung zeigt, dass die zweite Option die angenehmere ist, da so in Eclipse nicht zwischen „Package Explorer“ und „JUnit“ gewechselt werden muss.

Um den Test über die Konsole aufzurufen müssen im Classpath folgende Elemente enthalten sein:

- JUnit Class-Dateien
- Die eigenen Class-Dateien, einschließlich der JUnit Tests
- Die Bibliotheken von denen die Klassen abhängen

Windows Beispiel:

```
Set CLASSPATH=%JUNIT_HOME%\junit.jar;c:\myproject\classes;c:\myproject\lib\
something.jar
```

Unix (bash) Beispiel:

```
export CLASSPATH=$JUNIT_HOME/junit.jar:/myproject/classes:/myproject/lib/
something.jar
```

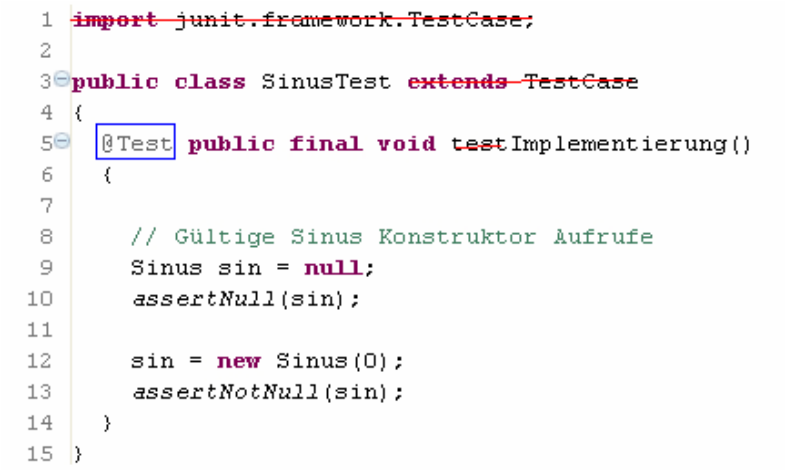
Danach wird der Runner aufgerufen:

```
java org.junit.runner.JUnitCore <test class name>
```

5.5 Ein Einblick in JUnit 4.0

Es sei zum Schluss erwähnt, dass JUnit 4.0 Mitte 2006 erschienen ist und auf einer völlig neuen API aufbaut.

Der Grundgedanke und die eigentliche Funktion von JUnit ist gleich geblieben, jedoch muss eine Testklasse nicht mehr von TestCase abgeleitet werden und Testmethoden werden nicht mehr durch ihre Namenskonvention test…() erkannt, sondern, wie in Abbildung: Testmethoden in JUnit 4.0 gezeigt wird, durch die Annotation @Test (Zeile 5) zu Beginn der Methode. setUp und tearDown Methoden werden, in Gleicherweise, durch die Annotationen @Before und @After ersetzt.



```
1  import junit.framework.TestCase;
2
3  public class SinusTest extends TestCase
4  {
5    @Test public final void testImplementierung()
6    {
7
8      // Gültige Sinus Konstruktor Aufrufe
9      Sinus sin = null;
10     assertNull(sin);
11
12     sin = new Sinus(0);
13     assertNotNull(sin);
14   }
15 }
```

Abbildung 14: Testmethoden in JUnit 4.0

Da die Testklasse nicht mehr von TestCase abgeleitet wird enthält sie keine Assert-Methoden mehr und der Aufruf der Assert-Methoden gestaltet sich aufwändiger. Der Aufwand wird jedoch dank Java 5.0 eliminiert, da ein statischer Import von org.junit.Assert dazu führt, dass die Assert-Methoden wie in JUnit 3.8 benutzt werden können.

Zudem gibt es in JUnit Version 4.0 zwei neue Assert-Methoden, mit denen Arrays verglichen werden können.

- public static void assertEquals(String message, Object[] expecteds, Object[] actuals);
- public static void assertEquals(Object[] expecteds, Object[] actuals);

Durch die Annotation @Ignore besteht die Möglichkeit Tests zu ignorieren, so dass der Test zwar in der Methode existiert, jedoch nicht ausgeführt wird.

Dies ist nur eine sehr kurze Einführung in die Änderungen von 4.0, jedoch zeigen die obigen Beispiele bereits, dass JUnit sich stetig weiterentwickelt und die Funktionalität neuer Java Versionen integriert.

6 Was noch zur Testgetriebenen Entwicklung gehört

Auch wenn bei testgetriebener Entwicklung zuerst an testgetriebenes Programmieren gedacht wird, so ist dies nur eine der drei Säulen von testgetriebener Entwicklung. Desweiteren gehören zur testgetriebenen Entwicklung auch das Refactoring und die Beständige Integration (engl.: „continuous integration“).

6.1 Refactoring

Refactoring bedeutet, das Design eines Programms zu ändern, ohne seine Funktionalität zu verändern. Das Ziel dabei ist, zu vereinfachen und redundanten Code zu eliminieren. Dadurch wird ein Programm leichter zu warten und besser zu erweitern, weil der Quellcode leichter lesbar wird. Gängige Vorgehensweisen sind dabei zum Beispiel das Umbenennen von Attributen oder Methoden, das verschieben von Attributen oder Methoden in andere Klassen, oder das Aufteilen oder Zusammenführen von Methoden. Da Refactoring alleine ein großes Gebiet ist, möchten wir an dieser Stelle auf das Buch von Martin Fowler [4] verweisen.

Der Vorteil der testgetriebenen Programmierung liegt darin, dass nach jedem Refactoring-Schritt direkt getestet werden kann, ob die Software noch wie erwartet funktioniert.

6.2 Beständige Integration

Bei herkömmlichen Softwareentwicklungsprozessen ist die Integration eines Gesamtsystems meist erst sehr spät vorgesehen oder sie wird sehr lange herausgezögert. Was dann passiert, wird auch Big-Bang-Integration genannt. Dabei werden erst die einzelnen Komponenten des Gesamtsystems gebaut und einzeln getestet. Dann wird das Gesamtsystem gebaut und gestartet. Wenn dann ein Fehler auftritt, ist es meist sehr schwierig, ihn zu lokalisieren.

Beständige Integration bedeutet von Anfang an die Integration einzuplanen und durchzuführen. Aus dem Quellcode kann jederzeit ein funktionierendes Gesamtsystem erzeugt werden, das auf dem aktuellen Entwicklungsstand ist. Dieser Vorgang wird vollständig automatisiert. Deshalb kann er auch problemlos mehrmals täglich durchgeführt werden.

Zu solch einer Integration gehören das vollständige Kompilieren des Quellcodes, ein Lauf der Unit Tests, das Erzeugen der Dokumentation und der Konfigurationsdateien und das Erzeugen eines installierbaren Programms.

Da dieser Vorgang vollständig automatisiert stattfinden soll, sind einige Werkzeuge dafür notwendig:

- Versionsverwaltung
- Built-Skript
- Integrationsserver

Das Built-Skript sollte alle Aufgaben, die zur vollständigen Integration gehören, ohne weitere Eingriffe durchführen können.

Für größere Projekte ist es sinnvoll, einen Integrationsserver zu benutzen, der zu festgelegten Zeitpunkten eine Integration durchführt. In einer solchen Umgebung sehen die Schritte, die der Entwickler durchzuführen hat, folgendermaßen aus:

- Synchronisieren der eigenen Quellen mit der Versionsverwaltung
- Erzeugen eines lokalen Builds zum Erkennen von Konflikten oder Fehlern
- Versionisieren des Lokalen Standes

Danach wird der Integrationsprozess auf dem Integrationsserver angestoßen und bei Erfolg ein neuer Build versioniert. Bei Misserfolg muss dies sofort den beteiligten Personen gemeldet werden, um die Ursache für das Fehlschlagen zu beheben. So wird sichergestellt, dass die aktuelle Version der Software voll funktionsfähig ist.

6.3 Abschließende Bewertung

Die Testgetriebene Entwicklung kann nur dann Vorteile Bringen, wenn man es schafft qualitativ gute Testfälle zu schreiben. Das Vorhandensein und regelmäßige ausführen von Tests alleine nutzt überhaupt nicht. Dadurch würde eine Trügerische Sicherheit aufgebaut, die über den wahren Zustand der Software täuscht. Deshalb ist es besonders notwendig die Verfahren der Testfallfindung zu kennen und anwenden zu können.

Des Weiteren ist es schwierig, bereits vorhandenen Quellcode nachträglich mit Unit Tests zu versehen. Wenn nicht schon bei der Entwicklung der Klassen auf Testbarkeit geachtet wurde müssen oft spezielle Techniken angewandt werden, um sie nachträglich testbar zu machen. Zum Erlernen von testgetriebener Entwicklung ist es also sinnvoll mit einem neuen Projekt zu beginnen.

7 Literaturverweis

[1] Westphal, Frank (2005):
Testgetriebene Entwicklung mit JUnit & FIT
Wie Software änderbar bleibt
1. Auflage. Heidelberg: dpunkt.verlag

[2] Link, Johannes (2005):
Softwaretests mit JUnit
2. Auflage. Heidelberg: dpunkt.verlag

[3] JUnit Download:
http://www.junit.org

[4] Fowler, Martin (1999)
Refactoring
Improving the Design of Existing Code
1. Auflage. Addison-Wesley Professional

[5] Vigenschow, Uwe (2004)
Objektorientiertes Testen und Testautomatisierung in der Praxis
Konzepte, Techniken und Verfahren
1. Auflage. Heidelberg: dpunkt.verlag