Fachhochschule
Münster University of
Applied Sciences

Software Qualität
JUnit und Testüberdeckung

Systematisches Testen mit JUnit und Code-Coverage Werkzeugen zur Qualitätsverbesserung

Prof. Dr. Nikolaus Wulff

Nichfunktionale Anforderungen

- Fehlerfreie Software wird als selbstverständlich erachtet. Gute Entwickler wissen dem ist nicht so!
- Die Komplexität von Software wächst **nicht linear** mit der Anzahl an Codezeilen und damit steigt auch die Zahl an Fehlermöglichkeiten überproportional.
- Software muss daher *ausreichend getestet* werden.
- Wichtig ist es nicht nur zu spezifizieren, was die Soft- und Hardware im *Normalfall* leisten, sondern auch wie sie sich im *Fehlerfall* verhalten soll.
- Tests müssen „weh tun“ und sollen Fehler provozieren, um die Software „hart“ zu machen.

Nichfunktionale Anforderungen II

- Entwicklung und Test eines sequentiell ablaufenden Programms ist schwierig – die Entwicklung einer parallelen Anwendung ist um vieles schwieriger, da ganz neue Klassen von Fehlern hinzukommen:
- ***Race Conditions***, ***Deadlocks*** und ***Starvation***.
- Diese Fehler gibt es in einem seriellen Programm nicht, obwohl es schon genug Fehlerquellen bietet.
- Vor der Parallelisierung eines Algorithmus sollte daher sichergestellt sein das die serielle Variante eine ausreichende Testüberdeckung besitzt.

(M)ein Standpunkt

- Nicht zu Testen ist grob fahrlässig oder Dummheit.
- Nicht ausreichend zu Testen ist unprofessionell.
- Test müssen automatisierbar sein, da händisches Testen mit zunehmender Codebasis immer umständlicher und aufwändig wird.
- Werden Fehler gemeldet, werden genau hierzu Testfälle entwickelt, die „rot“ sind. So wächst die Testüberdeckung dort wo es „weh tut“ und nicht sinnlos wo es nichts bringt oder einfach ist.
- Testfälle werden nie entfernt, so ist gewährleistet das sich alte Fehler nie wieder einschleichen!

Testüberdeckung

Die Qualität von Tests wird meist klassifiziert nach dem Grad C_x der getesteten Codeabdeckung:

- C_0 – Jede Codezeile wird ausgeführt.
- C_1 – Jede Alternative wird ausgeführt
- C_2 – Alle möglichen Pfade werden ausgeführt.
- C_3 – Jede kombinatorische Bedingung einer Verzweigung wird ausgeführt.

C_3 erfordert erhebliche Anstrengungen und ist meist nur für sicherheitskritsche Anwendungen bezahlbar.

Anweisungsüberdeckung

- C_0 kennzeichnet den Bruchteil an getesteten Anweisungen (*statement coverage*):

$$C_0 = \frac{getestete\ Anweisungen}{alle\ Anweisungen}$$

- C_0 ist die einfachste zu vermessende Metrik und kennzeichnet im Wesentlichen den Bruchteil an durch Tests geprüfte Codezeilen. Eine hohe C_0 Überdeckung sollte immer erreicht werden.

Zweigüberdeckung

- C_1 misst den Anteil der getesteten Verzweigungen im Programmfluß (*branch coverage*):

$$C_1 = \frac{getestete\ Verzweigungen}{alle\ Verzweigungen}$$

- Eine hohes C_1 bedingt meistens auch ein höheres C_0 (solange in allen Verzweigungen die selbe Größenordnung an Anweisungen erfolgt), bzw. falls $C_1=1$ dann gilt auch $C_0=1$.

Pfadüberdeckung

- C_2 misst den Anteil der getesteten Pfade durch das Programm (*path coverage*):

$$C_2 = \frac{getestete\ Pfade}{alle\ Pfade}$$

- C_2 und C_1 sind nicht identisch, gehören aber eng zusammen, da meist ein Pfad in Abhängigkeit von einer Bedingung gewählt wird.

Bedingungsüberdeckung

- C_3 misst den Anteil der getesteten Terme (*komplexe if-else* Bedingungen) (*condition coverage*):

$$C_3 = \frac{getestete\ Bedingungen}{alle\ möglichen\ Bedingungen}$$

- Alle Terme einer Bedingung müssen sowohl mit *true* als auch mit *false* getestet werden. Hierbei gilt es besonders auch auf Seiteneffekte der „Kurzschluß Operatoren && und || zu achten.
  - Es ist möglich eine Verzweigung/einen Pfad einzuleiten ohne alle Bedingungen getestet zu haben...
- Ein hohes C_3 erfordert exponentiellen Aufwand.

Wer testet den Tester?

- Gute Tests zu entwickeln ist nicht einfach und erfordert viel Gespür und Erfahrung, gerade wenn es um eine C_2 – C_3 Abdeckung geht.
- Jede funktionale und nichtfunktionale Anforderung sollte systematisch getestet sein.
- Die automatisierten Tests müssen selber getestet sein. Hierzu mag es helfen mit *Mock*-*Objekten* zu arbeiten. Die Tests müssen dann fehlschlagen und zeigen, dass sie auf Fehler der Mocks anspringen.

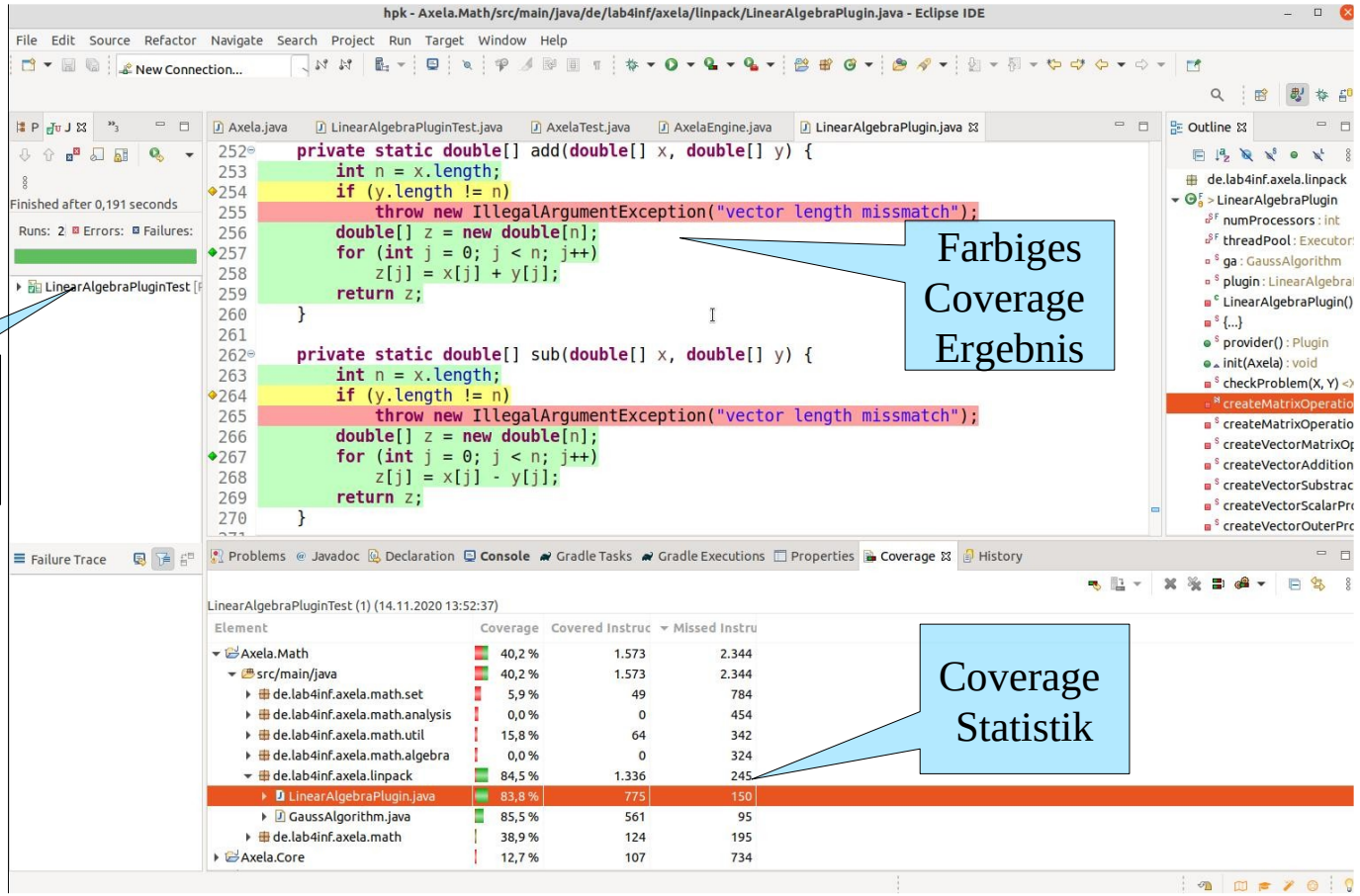
Eine nicht testbare Software ist ein klares Indiz für ein schlechtes Design!

Mock: engl. Attrape

Testüberdeckung messen

- Alle professionellen IDEs bieten inzwischen für die meisten Programmiersprachen Plugin um die Testüberdeckung zu messen und zu visualisieren.
- In der Eclipse wird z.B. die C_0 statement coverage durch grünen oder roten Hintergrund angezeigt.
- In gelber Farbe und mit Raute werden Bedingungen und Verzweigungen farblich kenntlich gemacht.
- Zusätzlich wird eine Statistik für das gesamte Projekt, jedes Paket und jede Klasse erstellt.
- Code Coverage gibt gezielte Hinweise wo noch weitere Tests notwendig sind.

Eclipse Test Coverage



Grün getestet, rot nicht, gelb nicht alle Bedingungen.

Grün Coverage sagt nichts über den Erfolg des Tests!

Einige JUnit Assertions

assert Funktionen um „Wahrheiten“ zu überprüfen:

- assertEquals(a,b) a.equals(b) Object
- assertEquals(j,k) $|j - k|=0$ Nur! Ganzzahl
- assertEquals(x, y, ε) $|x - y|< \varepsilon$ Gleitkommazahl
- assertTrue/False Bewerte logischen Ausdruck
- assert(Not)Null(o) Erwarte (k)eine Referenz
- assertThrows(...) Erwarte eine Exception
- fail(msg) Fehlermeldung

Alle Methoden liegen in einer überladenen Variante mit einem zusätzlichen String als Nachricht vor...

JUnit Pragmatik

- Um sinnvoll zu testen muss vorab „die Wahrheit“ bekannt sein nach einem Verfahren unabhängig vom zu testendem Objekt.
- Der **wahre Wert** gegen den verglichen wird ist immer der(das) **ganz links** stehende Wert(Objekt).
- Viele kleine Testmethoden sind besser als eine Große, die alle möglichen Szenarien enthält. Die Testüberdeckung und verdeckte Fehler sind so leichter zu erkennen.
- Es gilt sowohl „erlaubte“ als auch „verbotene“ Belegungen von Aufrufen gezielt zu testen.

Test Design

- White-Box Tests befinden sich im selben Paket wie das zu untersuchende Testobjekt – allerdings in einem dezidierten Sourcepfad nur für die Tests!
- Tests haben dann Zugriff auf protected und/oder package Scope Methoden.
- Durch geschicktes Refactoring lassen sich Klassenhierarchien systematisch durch „parallel“ verlaufende Testhierarchien effizient testen.
- „SuperTests“ definieren häufig abstrakte Fabrikmethoden zum Erzeugen des @Test setUp.
- und enthalten spezielle assert*XYZ* Methoden...

double, Vector, Matrix Test

- Absolute Differenz zweier double Zahlen x,y

  $\delta_a = |x - y| \leq \varepsilon$ => assertEquals(x, y, ε)

- Relative Differenz zweier double Zahlen $x \neq 0$

  $\delta_r = |1 - y/x| \leq \varepsilon$ => assertRelativeEquals(x, y, ε)

- Maximale Differenz zweier Vektoren

  $\|z - x\| \leq \varepsilon$ => assertArrayEquals(x, y, ε)

- Maximale Differenz zweier Matrizen

  $\|A - B\| \leq \varepsilon$ => assertMatrixEquals($\boldsymbol{A}$, $\boldsymbol{B}$, ε)

- assertRelativeEquals und assertMatrixEquals müssen selber entwickelt werden da keine JUnit Methoden. Beachten Sie die Fehlertoleranz ε !

bottom-up Teststrategien

- Entwicklung von Teststrategien am Beispiel von Matrizen **A, B** und Vektoren *x*,*y*,*z*.
- Ist die Inverse $\mathbf{A}^{-1}$ bekannt gilt $\mathbf{A}*\mathbf{A}^{-1} = \mathbf{E}$. Damit lässt sich die Matrizenmultiplikation testen.
- Soll zu einem Problem $y = \mathbf{A}*x$ bei gegebenem y und **A** der Vektor x per „solve“ berechnet werden, so lässt sich der Solver mit Zufallszahlen testen:

  Generiere **A**, x random, berechne $y = \mathbf{A}*x$ und lasse den Solver z=solve(**A**, y) berechnen. Dann muss in jeder Vektornorm gelten $\|z-x\| \leq \varepsilon$

  D.h. in der Max-Norm für alle k Elemente $|z_k - x_k| \leq \varepsilon$

Teststrategien cont.

- Sobald der Solver einen Vektor *x* zu *y*=**A***x* berechnet lässt sich mit ihm auch das Problem **B**=solve(**A**,**E**) zur Einheitsmatrix **E** lösen.
- Dann muss gelten $\mathbf{B} \equiv \mathbf{A}^{-1}$ ist die Inverse zu **A**! Auf diese Weise lassen sich aufeinander aufbauend immer weitere Testfälle **mit Zufallszahlen** testen.
- Jedes dieser Testszenarien beruht auf einem zuvor bereits getesteten „niederwertigen“ Testszenario, auf das es sich abstützen kann.
- So lassen sich immer kompliziertere Algorithmen testen. Dies ist notwendig vor der Parallelisieung!

Zusammenfassung

- Automatisierte Tests sind ein effektives Mittel um Software zu härten.
- Tests erfordern ein Durchdringen des Problembereichs was nie schaden kann ;-) und erzwingen zugleich ein klareres Design der Lösung und Schnittstellen und verändern nachhaltig den Programmierstil zum Guten.
- Vor dem Parallelisieren von Algorithmen müssen sehr gute Testfälle für die sequentiellen Varianten vorliegen, die auch parallel noch wahr sein müssen.
- Gute Testüberdeckung ist hierfür unerlässlich!