

Hinweis:

Diese Druckversion der Lerneinheit stellt aufgrund der Beschaffenheit des Mediums eine im Funktionsumfang stark eingeschränkte Variante des Lernmaterials dar. Um alle Funktionen, insbesondere Animationen und Interaktionen, nutzen zu können, benötigen Sie die On- oder Offlineversion. Die Inhalte sind urheberrechtlich geschützt.
©2018 Beuth Hochschule für Technik Berlin

TST - Objektorientiertes Testen und Test Driven Development



Lernziele und Überblick

Eines der wichtigsten Themen der Softwaretechnik ist die Qualitätssicherung der zu entwickelnden Software. Eine Testkultur in größeren Softwareprojekten ist unverzichtbar. Automatisierte Testläufe sind ein fester Bestandteil des **continuous integration** geworden. Es wird unabhängig vom Entwickler automatisiert und asynchron getestet.



Lernziele

Ziel dieser Lerneinheit ist es, sowohl die Denkweise des Testens näher zu bringen, als auch die entsprechenden Werkzeuge kennenzulernen, mit denen Testen bisweilen auch **Spaß** machen kann.

Es ist wichtig, dass Sie die Testwerkzeuge selber ausprobieren, sich diese zugleich zu eigen machen und in der **Praxis** anwenden! Zwingen Sie sich dazu nach **Test-First** zu programmieren. Dies ist eine wichtige Erfahrung, die jeder Softwaretechniker/in gemacht haben muss.



Zeitbedarf und Umfang

Zum Durcharbeiten dieser Lerneinheit und der Übungen benötigen sie ca. 3,5 Stunden.



Film

Webkonferenz zur Lerneinheit TST

1.4 Testebenen

Im Folgenden wird etwas ausführlicher erläutert, welche Testgranularitäten es geben kann. Neben den reinen Softwaretests auf Methodenebene sind dabei natürlich auch die Tests der eigentlichen Anforderungen zu testen. Bei der Planung von Softwaretests ist es wichtig, die Ebenen zu betrachten, die getestet werden können.

Testebene	Testverfahren
Modul, Klasse, Methode	Modultest, Klassentest, Unit-Tests
Komponente	Komponententests
Schicht (z. B. MVC)	Test der Subsysteme
Systemweit	Integrationstest

Komponententests sind oft schwer durchzuführen, weil Komponenten Abhängigkeiten haben. Hier kann deshalb mit Mock-Objekten **Kapitel 4** gearbeitet werden. Als Beispiel sind EJB-Komponenten der Version 2 zu nennen. Diese brauchen einen Container um ausgeführt zu werden. Dieser kann jedoch oftmals nur schwer in einen Test eingebunden werden.

Aber auch einfachere Komponenten können Netzwerk- oder Datenbankverbindungen benötigen, die oft nicht zur Verfügung stehen oder teuer sind. Diese können dann aber unter Umständen mit Mock-Objekten simuliert werden.

© Beuth Hochschule Berlin - Dauer: 21:52 Min. - Streaming Media 24.4 MB

Die Hinweise auf klausurrelevante Teile beziehen sich möglicherweise nicht auf Ihren Kurs. Stimmen Sie sich darüber bitte mit ihrer Kursbetreuung ab.


1 Einführung

Über dem objektorientierten Testen steht die Softwarequalität. Diese wird nach **BALZERT** folgendermaßen definiert:



Definition

Softwarequalität

„Unter Softwarequalität versteht man die Gesamtheit der Merkmale und Merkmalswerte eines Softwareprodukts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernissen zu erfüllen.“  [Bal98]

Softwarequalität

Softwarequalität ist für ein konkretes Projekt schwer zu fassen und muss individuell definiert werden. Entscheidend sind Qualitätsmerkmale (Kriterium), die es festzulegen gilt. Diese definieren, was das Produkt erfüllen muss, damit eine bestimmte Qualität sichergestellt ist. Der Grad der Erfüllung des Qualitätsmerkmals ist eine Metrik für einen bestimmten Indikator.



Beispiel

Beispiele für Softwarequalität

A. Ein GUI-Abbruchknopf



B. Vorgesehene Softwaretests

1.1 Mechanismen in Vorgehensmodellen

Alle modernen Vorgehensmodelle berücksichtigen die Softwarequalität. So ist dem Leser bestimmt noch bekannt, wie in den ersten Lerneinheiten in den Vorgehensmodellen RUP, V oder XP Softwarequalität adressiert wird. Neben der eigentlichen Softwarequalität wird dort aber hauptsächlich auch die Prozessqualität angegangen, die neben der reinen Softwarequalität ebenfalls eine große Bedeutung hat.

Schlüsselement Softwaretests

Beispiele für die Sicherstellung der Qualität im Prozess für die Software ist eine **iterative Softwareentwicklung**, ein **Refactoring** des Codes, **Codereviews** oder das **Pair-Programming**. Schlüsselement dieser Lerneinheit ist aber der **Softwaretest**, der im Code Testmethoden implementiert ist.

Zusätzlich gibt es unabhängige Qualitätsmodelle wie z. B. die des ISO/IEC 9126

[www !\[\]\(aff7c69c44a5e015f18c35867ef3f5c3_img.jpg\) http://de.wikipedia.org/wiki/ISO/IEC_9126](http://de.wikipedia.org/wiki/ISO/IEC_9126)

Sehr empfehlenswert zu diesem Thema ist das Buch Test Driven Development von **KENT BECK**

 [Be03]

1.2 Softwaretests



Definition

Softwaretest

Unter Softwaretests bezeichnet man ein automatisches oder manuelles Verfahren zur Verifikation und Validierung eines Softwareprogrammes.

Ziele des Testens

Ziel der Softwaretests ist es auf der einen Seite, die Qualität des Programmes sicherzustellen und zu erhöhen, wodurch das **Risiko** von Fehlern minimiert werden soll.

Auf der anderen Seite dient der konkrete Softwaretest dazu, spezifische Fehler im Code aufzudecken. Diese müssen dann folglich **dokumentiert** werden. Anschließend sollten diese **jedoch reproduziert** werden, um einen konkreten **Testfall** für diesen Fehler erstellen zu können. Üblicherweise wird der Fehler dann erst im Anschluss daran behoben.

Dies geschieht üblicherweise in Form eines „**Bug Fixes**“ die in Systemen wie Jira, Bugzilla oder JavaForge verwaltet werden können. In einem Prozessmodell, das Fehler- und Qualitätsmanagement berücksichtigt, müssen Fehler normalerweise freigegeben oder im Falle von kritischen Fehlern, eskaliert werden.

Freigeben bedeutet in diesem Fall, dass der Fehler erst behoben und dann von einer unabhängigen Person überprüft und als gelöst „freigegeben“ wird. Der Sinn ist, dass nicht der Autor des Codes den Fehler „weghackt“, sondern dass dieser sauber dokumentiert, reproduziert und getestet wird.

Kritische Fehler (vielleicht in Analogie zum Vattenfall Atomkraftwerk Krümmel) müssen „eskaliert“ werden. Dies bedeutet, dass Vorgesetzte informiert werden müssen und ein Regelwerk in Kraft tritt, das ein Fehlermanagement beinhaltet.

1.3 Testplan

Jeden Test, jedes Testvorgehen und jedes Testkonzept muss **geplant** werden. Dabei muss man für die Tests ausreichend Zeit einplanen! Dies wird häufig nicht berücksichtigt, kann sich aber rechnen, da das Entfernen von Fehlern in späteren Phasen sehr viel teurer sein kann.




Dem Tester sollte weiterhin bewusst sein, dass man nur die Fehler finden kann, die man auch sucht. Dies bedeutet, die Fehlersuche sollte sehr breit weit ausgelegt sein und eine Menge von **Testfällen** enthalten.

1.4 Testebenen

Im Folgenden wird etwas ausführlicher erläutert, welche Testgranularitäten es geben kann. Neben den reinen Softwaretests auf Methodenebene sind dabei natürlich auch die Tests der eigentlichen **Anforderungen** zu testen. Bei der Planung von Softwaretests ist es wichtig, die Ebenen zu betrachten, die getestet werden können.

Tab.: Testebenen und -verfahren

Testebene	Testverfahren
Modul, Klasse, Methode	Modultest, Klassentest, Unit-Tests
Komponente	Komponententests
Schicht (z. B. MVC)	Test der Subsysteme
Systemwelt	Integrationstest

Komponententests sind oft schwer durchzuführen, weil Komponenten Abhängigkeiten haben. Hier kann deshalb mit Mock-Objekten  Kapitel 4 gearbeitet werden. Als Beispiel sind EJB Komponenten der Version 2 zu nennen. Diese brauchen einen Container um ausgeführt zu werden. Dieser kann jedoch oftmals nur schwer in einen Test eingebunden werden.

Aber auch einfachere Komponenten können Netzwerk- oder Datenbankverbindungen benötigen, die oft nicht zur Verfügung stehen oder teuer sind. Diese können dann aber unter Umständen mit Mock Objekten simuliert werden.

1.5 Testverfahren für bestimmte Testebenen

Testverfahren	Beschreibung
Abnahmetests Akzeptanztests Funktionale Tests Oberflächentests	Hier wird oft der Auftraggeber gebeten zu prüfen, ob die <u>Software</u> seinen <u>Anforderungen</u> entspricht. Die Software wird mehr oder weniger „live“ ausprobiert.
Unit-Tests	Hier wird die kleinste Einheit (Unit) des <u>objektorientierten Programmes</u> (in der Regel die Signatur der <u>Methoden</u>) getestet.
Grenzwert-Tests (Boundary-Value Tests)	Hier werden die Unit-Tests so gestaltet, dass diese Extremwerte bekommen (wie z. B. null , 0 , -MAX_INT , +MAX_INT , etc.).
Destruktionstests Crashtests	Diese Tests versuchen einen unangemessenen Umgang mit der Software zu simulieren. Beim Crashtest wird versucht das <u>System</u> zum Absturz zu bringen.
Stresstests Lasttests	Auch hier werden Ausnahmesituationen simuliert. Üblicherweise ist dies z. B. der von Programmen simulierte Zugriff von zehntausenden <u>Anwendern</u> gleichzeitig auf das Programm oder die Verarbeitung einer großen Menge von Transaktionen.
Installationstests Integrationstests	Hier wird geprüft, ob die Installationen funktionieren oder ob es <u>Abhängigkeiten</u> gibt, die diese verhindern. Arbeiten die <u>Komponenten</u> wirklich richtig zusammen?
Regressionstests	Hierbei werden Tests gefahren, die versuchen, bereits behobene Fehler wieder zu reproduzieren. Dies ist bei der Weiterentwicklung* der Software sehr wichtig.
Zufallstests	Ein <u>Test</u> , der auf der Verwendung von Zufallsdaten basiert.

Tab.: Testverfahren

* Evolution von Software bedeutet auch Evolution der Qualitätssicherung. Daher müssen Fehler nicht nur behoben werden, sondern auch dokumentiert und in Testfällen getestet werden können. Es gibt Fälle, wo der Fehler behoben wird, dann aber in Abwandlung später wieder (anders) auftritt wenn die Software weiterentwickelt wird. Daher ist das Einfangen der Fehler in Testfällen hier besonders wichtig.

1.6 Wissen der testenden Person über die Komponente

Beschreibung	Wissen
Black-Box-Tests	Diese <u>Tests</u> werden von Entwicklern geschrieben, die kein Wissen über den Inhalt, des zu testenden <u>Moduls</u> haben. Das zu testende <u>System</u> ist eine „schwarze Kiste“, die nur von außen betrachtet werden kann. Sichtbar ist dann z. B. nur eine Methodensignatur oder eine GUI. Üblicherweise finden Black-Box Tests viele Fehler, da der Testentwickler „anders“ denkt, als die Person, die den zu testenden <u>Code</u> erstellt hat.
White-Box-Tests	Ein White-Box-Test hat die Kenntnis über den internen Ablauf der zu testenden <u>Komponente</u> . Üblicherweise wird dieser Test von der gleichen Person geschrieben. Ein bekanntes Beispiel ist der Test auf eine bestimmte Exception. Hier wird also ein Test geschrieben, ob die Komponente beispielsweise bei Übergabe eines Null-oder besonderen Wertes auch die richtige Exception liefert.
Grey-Box-Tests	Dieser Begriff stammt aus der testgetriebenen Entwicklung. Es wird dabei davon ausgegangen, dass zuerst der Test entwickelt wird. Die testschreibende Person hat also nur eine vage Vorstellung davon, was die spätere Komponente einmal tun wird.

Tab.: Box-Tests

Abschließend sei nochmals darauf hingewiesen, dass die Automatisierung des Testens eine immer größere Rolle spielt. Dies gilt sogar für alle Bereiche: der Testspezifikation, des Testlaufes selbst, der Testauswertung und der Testdokumentation. Tests können und müssen heute vielfach generiert werden (oder zumindest die Testrumpfe). Die Tests selbst werden im Buildmanagement automatisiert ausgeführt. Die Testberichte werden dann automatisch erstellt und soweit aggregiert, dass eine Testauswertung nur minimalen Aufwand erfordert.

Sie können jetzt entscheiden, ob Sie mit dem Kapitel 2 Werkzeuge fortfahren oder erst einmal im letzten Kapitel zu Test Driven Development etwas über die Philosophie des Testens lesen möchten.

2 Werkzeuge für Unit Tests/Testwerkzeuge

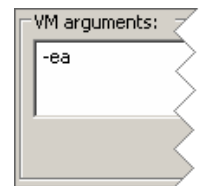
Der Modultest (engl. **unit** test) ist ein Teil des Softwareprozesses (Extreme Programming). Als Voraussetzung für Refactoring kommt ihm eine besondere Bedeutung zu. Nach jeder Änderung sollte durch Ablaufen lassen aller Testfälle nach Programmfehlern gesucht werden.

Ein Modul kann eine **Klasse** sein, deren Signaturen ihrer **öffentlichen** oder protected **Methoden** hier getestet werden sollen. Sie testen also, ob für allen Input der Methoden, ein korrekter Output zurückgegeben wird.

Prinzipiell können – mit geeigneten Werkzeugen – auch **private** Methoden getestet werden. Es wird jedoch **kontrovers** gestritten, ob dies sinnvoll ist. Der Modulautor ist ein Verfechter des Einbeziehs von privaten Methoden in den Test (wo es sinnvoll ist). Viele namenhafte Softwaretechniker halten dies aber nicht für sinnvoll.

2.1 Assertions

Assertion = Zusicherung `assert boolean;`
oder `assert boolean:value`



Java 7:

<http://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html>

Java 6:

<http://docs.oracle.com/javase/6/docs/technotes/guides/language/assert.html>



Quellcode

Klasse Becks

```
public class Becks{

    public Object converter(String s, int age){
        assert s != null;           // Pre-condition 1
        assert age > 0:age;         // Pre-Condition 2
        // heftiges Codieren
        assert s != null;           // Invariante
        // heftiges Codieren
        // z. B. anhaengen der Zahl an den String
        assert s != null;
        return s;                   // Post Condition
    }
}
```

2.2 JUnit

JUnit ist aus dem Gedanken des eXtreme Programming entstanden und wurde von **ERICH GAMMA** und **KENT BECK** entwickelt.

Das Programm soll hier nicht näher beschrieben werden. Nützliche Informationen zu JUnit finden Sie unter <http://www.junit.org>

Es ist das meistgenutzte Testwerkzeug, das in nahezu alle Sprachen portiert wurde. Es gibt sogenannte Testrunner für Text, Awt und Swing. Es gibt allerdings einige Beschränkungen bei JUnit Versionen < 4:

1. Testklasse **XXXTest** leitet von `junit.framework.TestCase` ab
2. Die Testmethode muss **testYYY** heißen

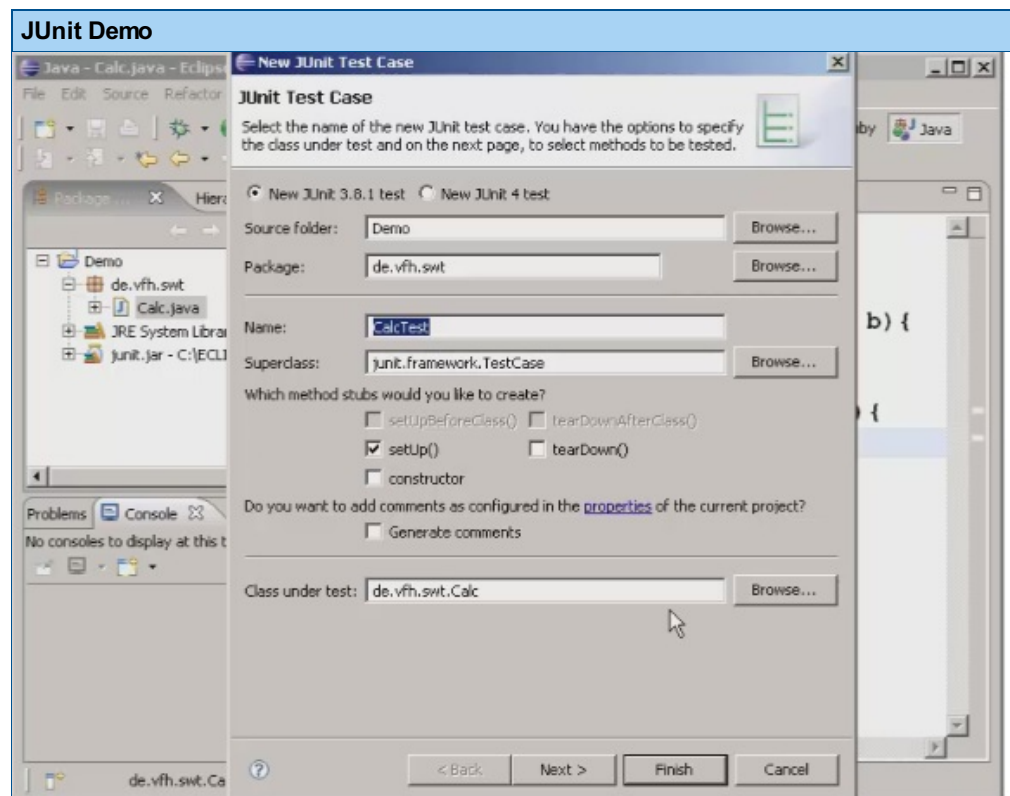
Die Einschränkungen bei der Namensgebung gibt es bei JUnit 4 nicht mehr.

2.3 Live Demo! JUnit 3.8.*

In der folgenden Animation können Sie sich JUnit im Einsatz anschauen. Die Animation ist vertont, schalten Sie daher auch Ihre Kopfhörer oder Lautsprecher an.



Film



© Beuth Hochschule Berlin - Dauer: 05:03 Min. - Streaming Media 7.56 MB

2.4 Testpositionierung und Testsuiten

Für kleine einfache Projekte können die Testklassen im gleichen Package liegen. In umfangreicheren Projekten wird ein Package mit dem Namen `*.test` angelegt. Letzteres hat den Vorteil, dass dann auch protected Methoden getestet werden können

Mehrere Tests lassen sich nun einfach durch eine Testsuite zusammenfassen. Dies funktioniert analog zu dem Composite Pattern.

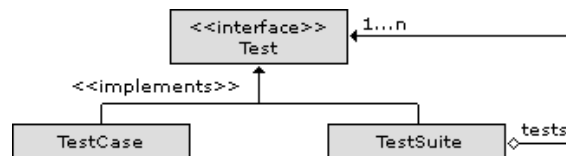
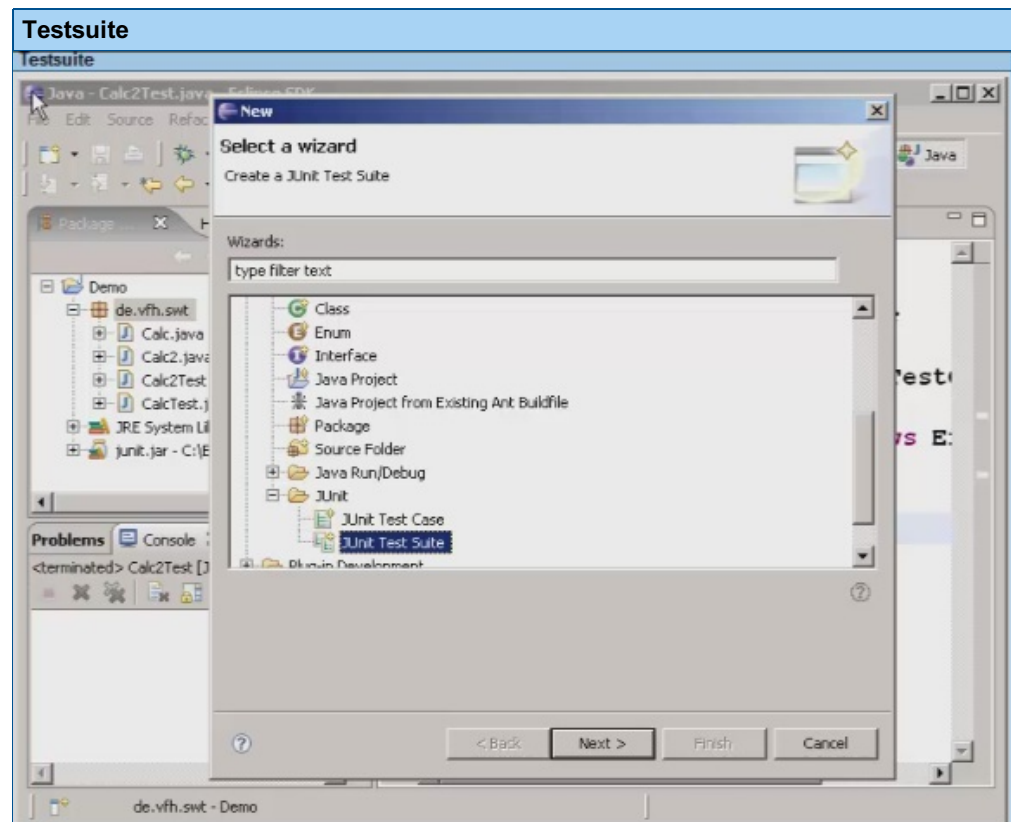


Abb.: TestSuite
Quelle: Simon Willnauer



Film



© Beuth Hochschule Berlin - Dauer: 01:04 Min. - Streaming Media 1.68 MB

2.5 Testen mit JUnit 4

Ab Java 5 sind **Annotations** möglich, teilweise auch schon unter Java 1.4
`@SuppressWarnings („unchecked“)`.

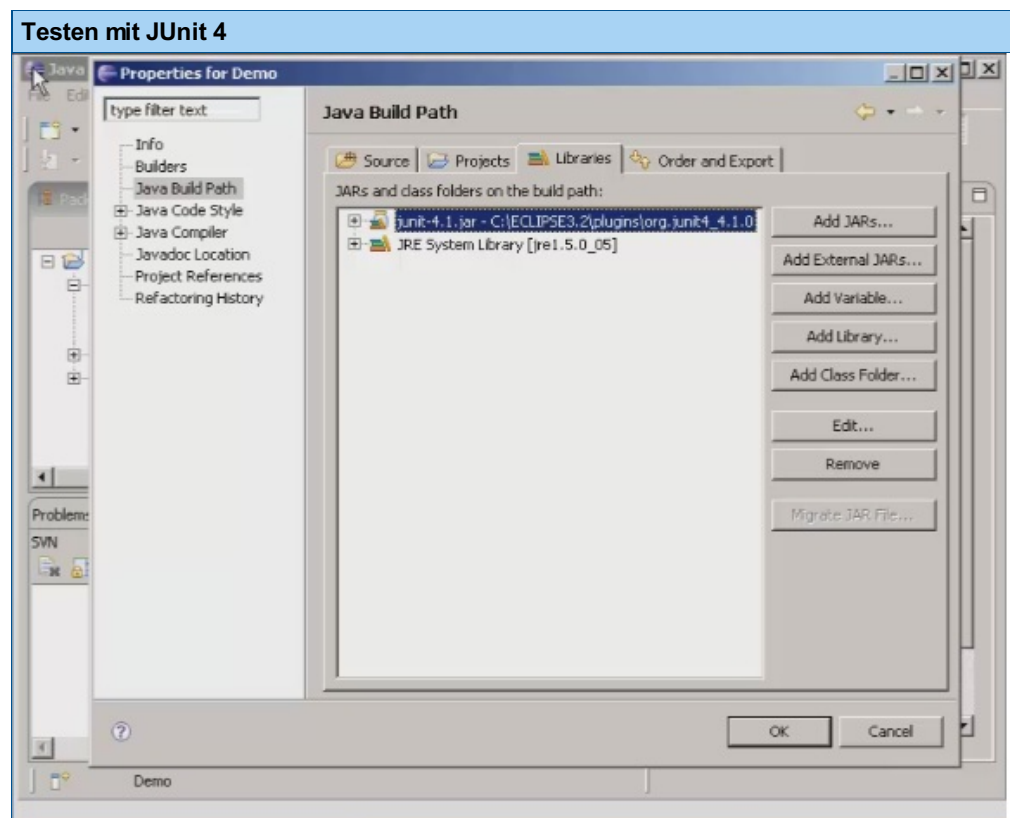
Dies ähnelt den **Custom Attributes** unter C#. Mit dem Symbol „@“ können Metadaten in den Code mit eingebunden werden. Dies war vorher nur über javadoc Kommentare möglich und musste umständlich mit z. B. `xDoclet` weiterverarbeitet werden.

Zu jeder Annotation – wie z. B. `@test` – muss es ein Interface geben. Dann kann der Compiler die Annotation prüfen. Danach kann mit Tools wie APT oder direkt aus dem Programmcode heraus auf die Annotation zugegriffen werden.

Dies kann man nur bei Testtools ausnutzen. Wie sieht das also in der Praxis unter Eclipse aus?



Film



© Beuth Hochschule Berlin - Dauer: 03:07 Min. - Streaming Media 4.93 MB

2.6 Sprachenuniversell

Welche generellen Prinzipien haben wir gelernt?

1. Tests zu schreiben ist **einfach**!
2. Es können viele Tests auf einmal in einer **Suite** ausgeführt werden
3. Es gibt Methoden um den Test vorzubereiten und zu beenden (**setUp**, **tearDown**, **@Before**, **@After**)
4. Es gibt entweder Namenskonventionen um Testmethoden auszuzeichnen oder diese werden **annotiert**
5. Man bekommt einfaches **Feedback** für jeden Testlauf. Entweder es hat geklappt, dann passiert nicht viel oder der Test scheiterte und wir bekommen den StackTrace.

Ist das jetzt nur für Java so oder gilt das fast **universell** für alle Programmiersprachen ?
Machen wir den Test und schauen über den Tellerrand: Eine Quick-Demo mit Ruby!

2.7 QuickDemo in Ruby



Film

Demo in Ruby

```

Notepad++ - 3\CAPTIVATE\LE14_TESTEN\MATERIAL\test_calc.rb
File Edit Search View Format Language Settings Macro Run Plugins ?

Calcib new3 test_calc.rb

require 'calc.rb'
require 'test/unit'

class CalcTest < Test::Unit::TestCase
  def setup
    @c = Calc.new
  end
  def test_add
    assert_equal 12, @c.add(7, 5)
  end
  def test_sub
    assert_equal 7, @c.sub(10, 3)
  end
end
  
```

Ruby file | rb chr : 430 | Ln : 12 Col : 26 Sel : 0 | Dos/Windows ANSI | JMS

© Beuth Hochschule Berlin - Dauer: 02:01 Min. - Streaming Media 2.48 MB

2.8 Abschluss: Wie machen es die Profis unter Java?

Die Profis unter den Entwicklern verwenden TestNG von CEDRIC BEUST

<http://www.testng.org>

Die Gründe dafür werden im folgenden Text näher erläutert. TestNG geht schon im Ansatz klar über einfache Unit-Tests hinaus und ist für Integrationstests viel besser geeignet. Mit TestNG kann man viel besser dynamische Inhalte in die Testklassen transportieren (**DataProvider**, **TestFactories**). Es gibt extrem flexible Möglichkeiten den Test zu konfigurieren z. B. um Test-Gruppen zu bilden, was in Enterprise Umgebungen unerlässlich ist. Auch eine Gruppierung von Gruppen ist möglich.

TestNG liefert einen ausführlichen HTML-Report, was ebenfalls sehr gut zu einem Ant-Build passt. Die **@Before** und **@After** Semantik ist noch feiner. TestNG kann mit der **BeanShell** beliebig erweitert werden. Es kann die erwartete Exception definiert werden. Für Tests kann eine Erfolgswahrscheinlichkeit angegeben werden.



Film

TestNG Beispiel

```

package de.vfh.test;
import org.testng.annotations.*;

public class PruefRechner {
    Calc c;

    @BeforeClass
    public void aufsetzen() {
        c = new Calc();
    }

    @Test(groups={"TestsAbisM"})
    public void checkeAdd() {
        assert 10 == c.add(7,3); //
    }

    @Test(groups={"TestsNbisZ"})
    public void checkeSub() {
        assert 5 == c.sub(8, 3);
    }
}
  
```

© Beuth Hochschule Berlin - Dauer: 03:59 Min. - Streaming Media 5.99 MB

3 Code Coverage

Unter Code Coverage versteht man die Testabdeckung durch Test-Cases. Des Weiteren zeigt die Code Coverage die ungetesteten Codestellen.

In der Praxis ist eine Testabdeckung von 80% ein erstrebenswertes Ziel. Eine hohe Testabdeckung ist mittlerweile ein wichtiges Marketinginstrument und spielt bei der Auftragsvergabe und -abnahme eine wichtige Rolle. Es gibt mittlerweile hervorragende (auch freie Werkzeuge) die den Prozess der Code Coverage Analyse automatisieren und recht umfassend durchführen können.



Formulieren

Übung TST-01

Viel hilft viel?

Eine hohe Test-Abdeckung weist auf eine hohe Code-Qualität hin!

Würden Sie dieser Aussage zustimmen? Notieren Sie sich kurz Argumente, die sowohl für als auch gegen diese Aussage sprechen.

Halten Sie Ihre Ausführungen für das nächste synchrone Meeting bereit.

Bearbeitungszeit: 15 Minuten

Eine gute und vertiefende Einführung in das Thema Code Coverage finden Sie im Internet unter:

 <http://www.bullseye.com/coverage.html>

3.1 Code Coverage Tools

Es gibt zahlreiche Code Coverage Werkzeuge für Java. Im Internet finden Sie diese beispielsweise unter der Adresse:

 <http://java-source.net/open-source/code-coverage>

Einige der wichtigsten Werkzeuge sind:

1. Clover (kommerziell), Testversion möglich
2. Emma (emma.sourceforge.net), Open-Source
3. Cobertura (cobertura.sourceforge.net/), Open-Source

Weitere Werkzeuge sind:

- Quilt
- NoUnit
- InsEct
- Hansel
- Jester
- JVMDI Code Coverage Analyse
- Grobo
- jcoverage
- JBlanket

3.2 Beispiel Clover

3 Arten der Messung:

1. Statement Coverage
Anzahl der durchlaufenden Statements einer Klasse
2. Conditionals Coverage
Wie konditionelle Konstrukte wie **if/else** durchlaufen werden. Der Test sollte im Idealfall alles abdecken
3. Method Coverage
Wie viel Methoden einer Klasse aufgerufen werden



Formel

Die gesamte Coverage ergibt sich aus einer komplexeren Formel:

$$TPC = (CT + CF + SC + MC) / (2 * C + S + M)$$

CT = conditionals that evaluated to „true“ at least once

CF = conditionals that evaluated to „false“ at least once

SC = statements covered MC = methods entered

3.3 Beispielreport von Clover

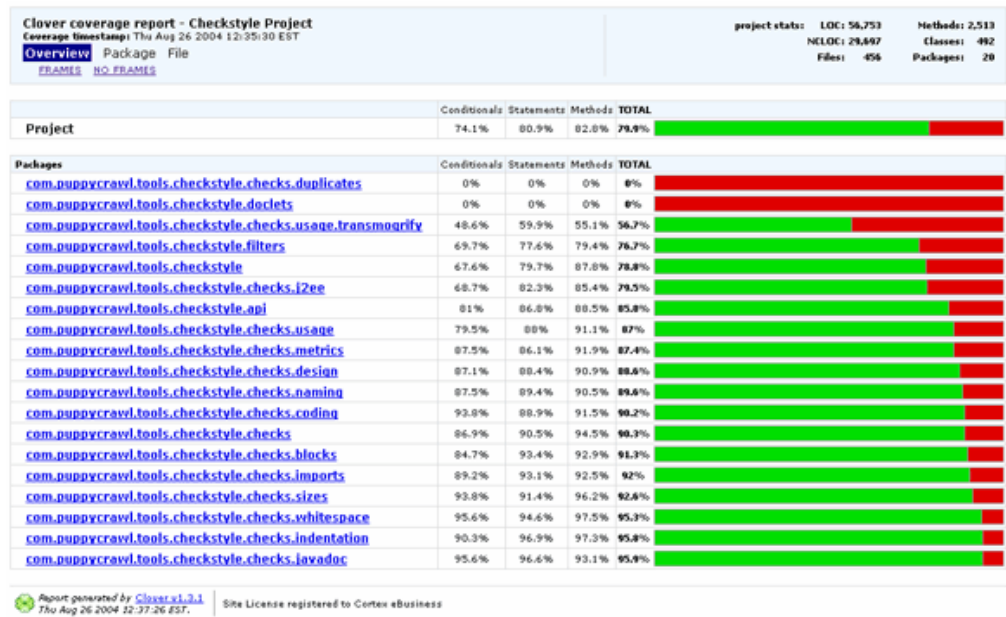


Abb.: Beispielreport von Clover

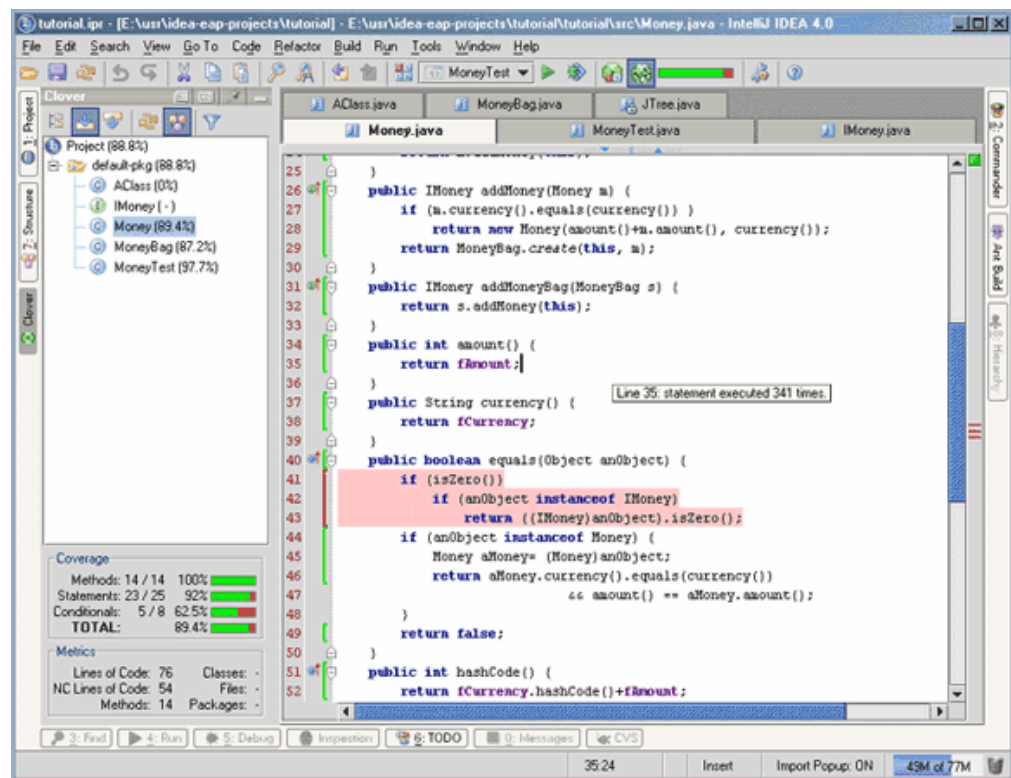


Abb.: Beispielreport von Clover

4 Mock Objekte

Übersetzungsprogramme liefern folgende Erklärung für Mock-Objekte: Attrappe, Fälschung, Nachahmung, Pseudo-, Schein-, Attrappe und täuschen.

In zu vielen Fällen kann nicht so einfach eine Unit getestet werden, weil die Komponente zu viele Abhängigkeiten hat. Wünschenswert für die Tests wären daher Testobjekte, die einer zu testenden Komponente eine reale Umgebung vortäuscht. Diese Objekte nennt man Mock-Objekte. Beispielsweise ein Objekt, das so tut, als wäre es eine Datenbank. Diese würde dann irgendwelche Result-Sets zurückliefern, was jedoch für den Test nicht von Bedeutung ist.

Tools wie www.jmock.org oder www.easymock.org erleichtern das Schreiben von Mock-Objekten! Weiterführende Informationen finden Sie im Internet unter:

www.mockobjects.com

Im Folgenden werden Mock-Objekte auch MOs genannt.

4.1 Die Problematik der Abhängigkeiten

Die Anwendung eines Tests oder einer Testklasse auf einzelne Dienst-Komponenten birgt ein Problem. Dienst-Komponenten sind in den meisten Fällen von weiteren Komponenten oder Diensten abhängig. (z. B. K1, K2, Datenbankdiensten oder Netzwerkdiensten)

Mock-Objekte hingegen können die zu testende Komponente isolieren und eine reale Umgebung simulieren.

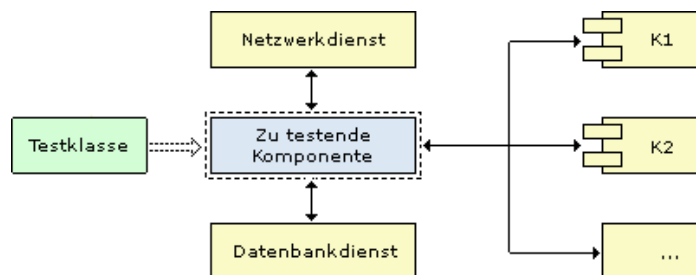


Abb.: Die Problematik der Abhängigkeiten

4.2 Quelle & Idee

Es gibt verschiedene Quellen und Ideen zum Thema Mock-Objekte. Diese finden Sie im unten stehenden Text.

1. Paper von TIM MACKINNON, STEVE FREEMAN, PHILIP CRAIG
Endo-Testing: Unit Testing with Mock Objects
2. Konferenz: eXtreme Programming and Flexible Processes
in Software Engineering - XP2000
3. Download z. B. unter: www.connextra.com

"Once," said the Mock Turtle at last, with a deep sigh, "I was a real Turtle."

(Alice in Wonderland, Lewis Carroll)

4.3 Technik und Features


Seit Java 1.4 gibt es bessere Möglichkeiten: Mit `java.lang.reflect.Proxy` kann man dynamische Proxies erschaffen, die auf Interfaces basieren. Fast alle Tools haben Add-Ons, die auch Proxies für Klassen generieren können.


Einfach nur leere Mock-Objekte, die still ihren Dienst tun, ist aber noch zu wenig. dynamische Mock-Objekte unterstützen zusätzlich:

1. Vorgegebene Return-Values
2. Exceptions Result Chaining
3. Parameter Tracing
4. Aufrufanzahl und Reihenfolge mitzählen und checken
5. Defaultvalues generieren

Unterschieden wird zwischen statischen Mock-Objekten (können z. B. als statischer Code generiert werden) und dynamischen Mock-Objekten - mit Proxies oder aspektorientierten Ansätzen.

Typische Beispiele sind

 <http://www.easymock.org> und

 <http://www.jmock.org>

4.4 Zustandsautomaten von dynamischen Mock Objekten

Dynamische Mock-Objekte haben ein ähnliches Funktionsschema, wie Kassettenrecorder.

Nach einem Reset kann eine Aufzeichnung gestartet werden. Von Preparing zu Working mit Replay, dann wird mit Verify überprüft, ob das was aufgezeichnet wurde, auch wirklich wieder abgespielt wird. Das geht in beide Richtungen.

Das heißt, die Mock-Objekte können nicht nur die Signatur von Methodenkomponenten emulieren, sie besitzen auch Intelligenz, sodass man sehen kann ob beim Test auch wirklich die richtige Methode - mit den richtigen Argumenten und den richtigen Returnvalues - vorher eingespielt und danach im Test abgespielt worden ist.

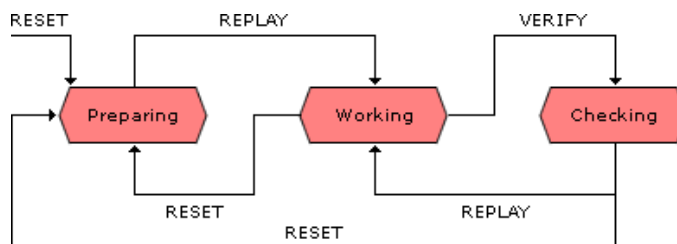


Abb.:
Zustandsautomaten von
dynamischen Mock Objekten

4.5 Ein einfaches Beispiel

Die zu testende Klasse **BusinessClass** ist die Klasse, die z. B. mit **JUnit** getestet werden soll. Sie ist aber abhängig von einer weiteren Klasse die **Collaborator** heißt und auch ein solches Interface besitzt. **BusinessClassTest** heißt infolgedessen die Testklasse, die die **BusinessClass** testet.



Im folgenden einfachen Beispiel werden die benötigten Komponenten erläutert. Des Weiteren werden unter anderem die Implementierung der zu testenden Klasse **BusinessClass** und die Testklasse **BusinessClassTest** beschrieben.



Film

Beispiel Mock-Objekt

Benötigte Komponenten

```

package de.vfh.doit;

public interface Collaborator {
    void storeDocument(String title);
}
  
```

- **storeDocument** ist die Methode eines Repositories, die unsere Business-Methode benötigt.

© Beuth Hochschule Berlin - Dauer: 7:33 Min. - Streaming Media 8.79 MB

Fazit

Mock-Objekte sind Hilfsmittel während der Entwicklungsphase oder für echte Tests.

Der Programmierstil erfordert, dass das zu testende Objekt seine abhängige Komponente dynamisch erhält. Per Setter, per Factory, Aspekten oder per IoC-Framework! Dies fördert eben nicht nur gutes Design sondern auch gute JDepend Metriken. Das sind jedoch Themen für die andere Lerneinheiten.

4.6 Ein weiteres Beispiel mit Setter Injection



Beispiel



Quellcode

```
public class BusinessClass2 {

    private Collaborator coll = null;

    public void setMock(Collaborator c){
        this.coll = c; //Entweder mock oder etwas reales von Aussen
    }

    public int userClicksSaveDoc (String title){
        if (coll==null) // Wenn nichts da ist, dann default...
            coll = new CollaboratorClass();
        coll.storeDocument(title);
        return title.length(); // Wir nehmen an, dass der Test OK ist,
                                // wenn die Abhaengigkeiten verfügbar sind
    }

    // Hier mal schnell zur Demonstration als inner class
    class CollaboratorClass implements Collaborator {
        public int storeDocument(String title){
            return title.length();
        }
    }
}
```

5 TDD – Test Driven Development

Die Idee des Test Driven Development besteht darin, zuerst den Testfall und danach die Komponente zu schreiben, deren Test dann schon existiert!

Diese Idee des TDD entstammt ebenfalls aus dem **Extreme Programming** und wurde seit der Jahrtausendwende populär. Begonnen wird also mit einem roten Balken oder einem negativen Report, aber dies ist positiv zu sehen. Man hat sofort einen klaren Auftrag

Im Extremfall gibt es nur wenig Spezifikationen und Tests weisen sogar den Weg zu dem, was eigentlich implementiert werden soll.

Der erste Test schlägt beim Test Driven Development also immer fehl. Man hat mit TDD eine inkrementelle Entwicklung mit dem Feedback von ganz konkreten Tests.

5.1 Der TDD Zyklus

Beim Test Driven Development Zyklus wird damit begonnen einen Test zu schreiben. Anschließend soll der Test automatisiert ablaufen. Da jedoch in der ersten Runde noch kein Code vorhanden ist, wird der Test scheitern.

Als nächstes wird einfacher Code geschrieben, der lediglich den Test erfüllt. Nach einem weiteren automatisierten Durchlauf sollte der Test funktionieren. Der Code ist noch nicht optimal. Er wird refaktoriert und perfektioniert. Daraufhin beginnt der Zyklus wieder mit dem Schreiben weiterer Tests.

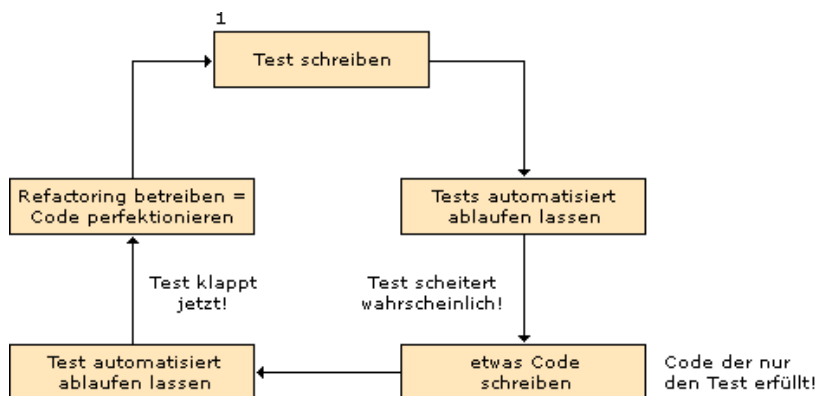


Abb.: Der TDD Zyklus

5.2 Vorteile und Eigenschaften

TDD (Test Driven Development) fördern Tests, die zeitnah zur Programmierung sind. Diese Tests sind automatisiert und beliebig oft wiederholbar und können mit den **Build** (ant, rake, etc.) eingebaut werden.

Das Testen mit den Werkzeugen wie **JUnit** oder **TestNG** kann sogar Spass machen. Das frühe Beheben und Finden von Bugs ist oft viel kosteneffizienter, als die Fehler später zu finden.



Achtung

Beim Entwickler findet ein anderer Denkprozess statt, sobald dieser die Codierung der Tests voran stellt. Dies hat Auswirkungen auf die Erstellung des Codes, der dann zumeist sauberer und fehlerfreier durch den Entwickler codiert wird, als zuvor.

Zusammenfassung

- Für die Softwarequalität ist es entscheidend das Qualitätsmerkmale festgelegt werden, an denen der Grad der Erfüllung gemessen werden kann.
 - Softwaretests sind automatische oder manuelle Verfahren zur Verifikation und Validierung eines Softwareprogrammes.
 - In einem Prozessmodell, das Fehler- und Qualitätsmanagement berücksichtigt, müssen Fehler üblicherweise freigegeben - oder im Falle von kritischen Fehlern, eskaliert werden.
 - Es existieren verschiedene Testverfahren für bestimmte Testebenen.
 - Evolution von Software bedeutet auch Evolution der Qualitätssicherung.
 - Die Automatisierung des Testens spielt eine immer größere Rolle, dies gilt für alle Bereiche - Spezifikation, Testlauf, Auswertung und Dokumentation.
 - Code Coverage bezeichnet die Testabdeckung durch Test-Cases. In der Praxis ist eine Testabdeckung von 80% erstrebenswert.
 - Testobjekte, die einer zu testenden Komponente eine reale Umgebung vortäuschen werden Mock-Objekte genannt. Diese können die zu testende Komponente isolieren und eine reale Umgebung simulieren.
 - Unterschieden werden statische und dynamische Mock-Objekte. Mock-Objekte sind Hilfsmittel während der Entwicklungsphase oder für echte Tests.
 - Beim Test Driven Development schlägt der erste Test immer fehl. TDD ist eine inkrementelle Entwicklung resultierend aus dem Feedback von ganz konkreten Tests.
-

Wissensüberprüfung

Damit sind wir am Ende der Lerneinheit angelangt. Es folgen noch die Übungen die Sie bearbeiten können. Für Sie zum Reflektieren und zum Implementieren.



Formulieren

Übung TST-02

Softwaretest, Testwerkzeuge

1. Was versteht man unter Testen? Wieso testet man Software?
2. Wie würden sie für ein Softwarepaket ihrer Wahl, Softwarequalität definieren?
3. Was tun sie, wenn sie bei der Arbeit mit ihrer Software einen Fehler gefunden haben?
4. Warum sind Tests oft schwer durchzuführen?
5. Welche Testebenen gibt es?
6. Was versteht man unter Black-Box-/ White-Box- und Grey-Box-Tests?
7. Was sind Assertions und welche Rolle spielen sie in Testwerkzeugen?
8. Testen sie ihre Software oder eine Demoklasse mit **JUnit 3.8.*!**
9. Wozu braucht man die Methoden **setUp()** und **tearDown()**?
10. Was sind Testsuiten?
11. Was ist neu an modernen Testwerkzeugen wie **JUnit 4** oder **TestNG**?
12. Was kann **TestNG** mehr?
13. Experimentieren sie mit **JUnit 4** und **TestNG**! Senden sie die Ergebnisse an den Betreuer!

Bearbeitungszeit: 20 Minuten



Formulieren

Übung TST-03

Code Coverage

1. Was versteht man unter Code Coverage?
2. Ist Code Coverage wichtig? Warum? Was sagt eine hohe Zahl aus?
3. Was decken moderne Werkzeuge mehr ab, als nur die Zahl der getesteten Methoden zu testen?
4. Laden sie ein Code Coverage herunter und lassen sie sich einen Report erstellen.
5. Testen sie Clover im PlugIn Modus und lassen sie sich auch die Abdeckung von Statements und Conditionals aufzeigen!

Bearbeitungszeit: 40 Minuten



Formulieren

Übung TST-04

TDD (Test Driven Development)

1. Was versteht man unter TDD (Test Driven Development)?
2. Wie läuft der Zyklus im TDD ab?
3. Warum kann TDD von Vorteil sein?

Bearbeitungszeit: 15 Minuten



Formulieren

Übung TST-05

Mock-Objekte

1. Was sind Mock-Objekte?
2. Bei welcher Problematik helfen Mock-Objekte?
3. Was sind statische und dynamische Mocks?
4. Testen sie ein abhängige Komponente praktisch z. B. mit **jmock** oder **easymock**!
Benutzen sie zuerst nur den proxy.
5. Versuchen sie im zweiten Schritt einmal, die mock-Verwendung vorzuprogrammieren und dann mit (z. B. **replay/verify**) zu überprüfen!
6. Versuchen sie, die **Return-Values** voreinzustellen und dann in der konkreten Nutzung auch auszugeben.
7. Welche Architektur der Klassen implizieren Mock-Objekte?
8. Ist es immer einfach oder sinnvoll, diese Architektur zu übernehmen?

Bearbeitungszeit: 60 Minuten



Multiple Choice

Übung TST-06**Testverfahren zuordnen**

Ordnen Sie den verschiedenen Testverfahren eine oder mehrere der folgenden Aussagen zu!

1. Der Auftraggeber wird gebeten zu prüfen, ob die Software seinen Anforderungen entspricht. Die Software wird mehr oder weniger „live“ ausprobiert.
2. Die kleinste Einheit des objektorientierten Programmes wird getestet.
3. Hier werden Unit-Tests so gestaltet, dass diese Extremwerte bekommen.
4. Simulation eines unangemessenen Umgangs mit der Software.
5. Hier werden Ausnahmesituationen simuliert.
6. Hier wird geprüft, ob die Einrichtung einer Software erfolgreich war oder ob es Abhängigkeiten gibt, die diese verhindern.
7. Es wird versucht, bereits behobene Fehler zu reproduzieren.
8. Basiert auf der Verwendung von nicht festgelegten Daten.

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	
Integrationstests	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Lasttests	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Zufallstests	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Stresstests	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Regressionstests	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Crashtests	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Destruktionstests	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Grenzwert-Tests	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Oberflächentests	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Akzeptanztests	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Installationstests	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Abnahmetests	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Unit-Tests	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Funktionale Tests	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

[?](#) [Test wiederholen](#) [Test auswerten](#)