Folie 1

Die folgenden Folien beschäftigen sich mit der Testphase; dazu gehören Hintergründe zu Softwarefehlern, verschiedene Testverfahren und der Einsatz von Junit. Zu jeder Folie sind Notizenseiten erfasst.

Verbesserungsvorschläge und Fehlerhinweise können Sie gerne an die Adresse mail@mielebacher.de senden.

Folie 2

Eckpunkte der letzten Veranstaltung

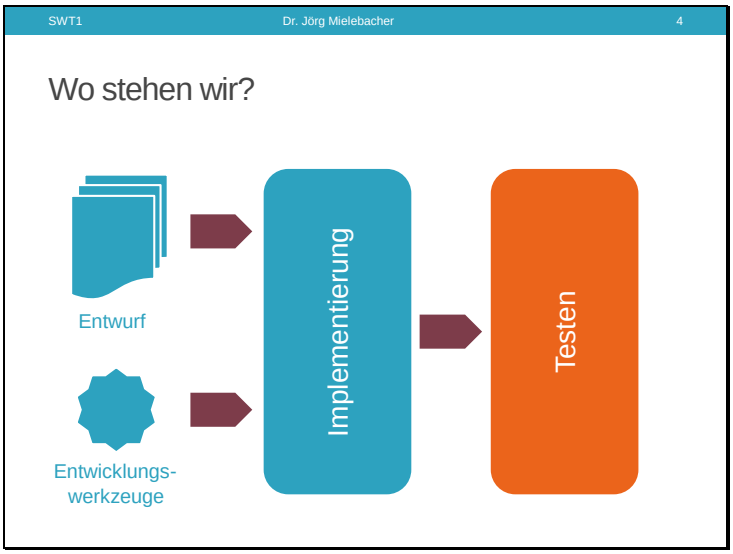
- Der Quellcode und damit verbundene Ergebnisse werden auf Grundlage des Entwurfs erstellt.
- Eine IDE bietet Editorfunktionalität sowie die Anbindung der relevanten Werkzeuge.
- Styleguides beschreiben den Aufbau des Quellcodes.
- Eine Versionsverwaltung speichert die Änderungen an allen Quellcodedateien und steuert den Zugriff durch mehrere Entwickler.
- Quellcode sollte selbsterklärend sein, relevante Dokumentation enthalten und für Dokumentations-generatoren vorbereitet sein.

Folie 3

Literaturempfehlungen

- Meyer: Seven Principles of Software Testing
  http://se.ethz.ch/~meyer/publications/testing/principles.pdf
  - Praxisnahe Auseinandersetzung mit Softwaretests
- Pilone, Miles: Softwareentwicklung von Kopf bis Fuß
  - Testen kurz aber anschaulich beschrieben, auch sonst gut (z.B. Versionsverwaltung)
- Grechening et al: Softwaretechnik
  - Gutes Testkapitel, sonst eher mittelmäßig; aber: Fallbeispiele
- http://www.istqb.org
  (International Software Testing Qualifications Board)
  - Zertifizierung von Testern; Glossar mit Testbegriffen
- http://www.junit.org
  - Informationen zu JUnit, Downloads, Tutorials

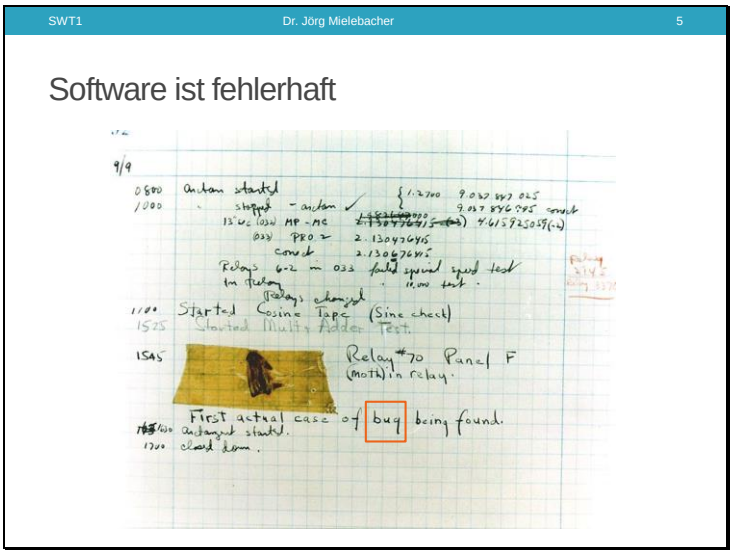
Folie 4



Die zurückliegenden Veranstaltungen beschäftigten sich mit der Analyse von Anforderungen und anschließend mit dem Entwerfen der Software. Auf dieser Grundlage wurde in der Implementierungsphase der Quellcode erstellt.

Die Testphase muss nun prüfen, ob das entwickelte Programm sich korrekt verhält und seine Spezifikation erfüllt.

Folie 5

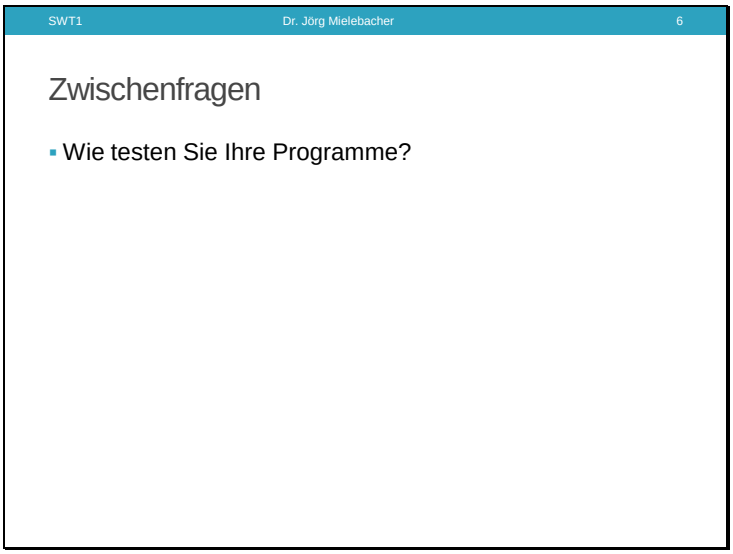


Software ist fehlerhaft. Man geht heutzutage davon aus, dass Software 2-3 Fehler pro 1000 Zeilen Code enthält. Manche Autoren sehen dieses Verhältnis sogar nur bei „guter“ Software erfüllt und sprechen von durchschnittlich bis zu 10 Fehlern pro 1000 Zeilen.

Softwarefehler begleiten die Programmierung von Computern seit ihren Anfängen. Dazu zeugt beispielsweise der hier gezeigte Auszug aus einem Log-Buch des Harvard Mark II vom 09.09.1947. Die aufgeklebte Motte hatte in den Relais des Computers einen Fehler verursacht. Gefunden und dokumentiert wurde dies von Grace Hopper, die entscheidend am Aufbau der ersten Großrechner mitwirkte. Dass sie den „Bug“ gefunden und beseitigt hatte, brachte sie in Verbindung mit dem Begriff „Debugging“.

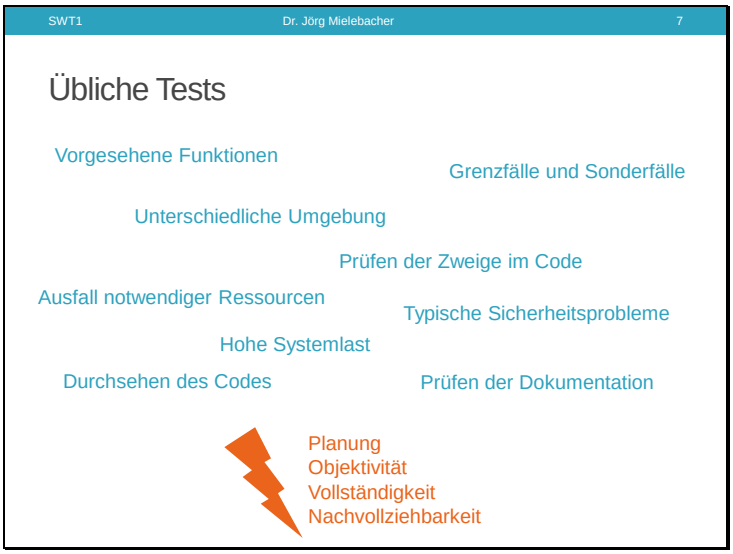
Hopper steht allerdings auch in Zusammenhang mit einem anderen Softwarefehler: dem Y2K-Bug. So hatte sie in ihren Programmen Jahreszahlen stets zweistellig gespeichert, um den Speicherbedarf zu reduzieren.

Folie 6



Wie testen Sie Ihre Programme? Wie gehen Sie vor? Was dokumentieren Sie? Wann führen Sie die Tests durch?

Folie 7

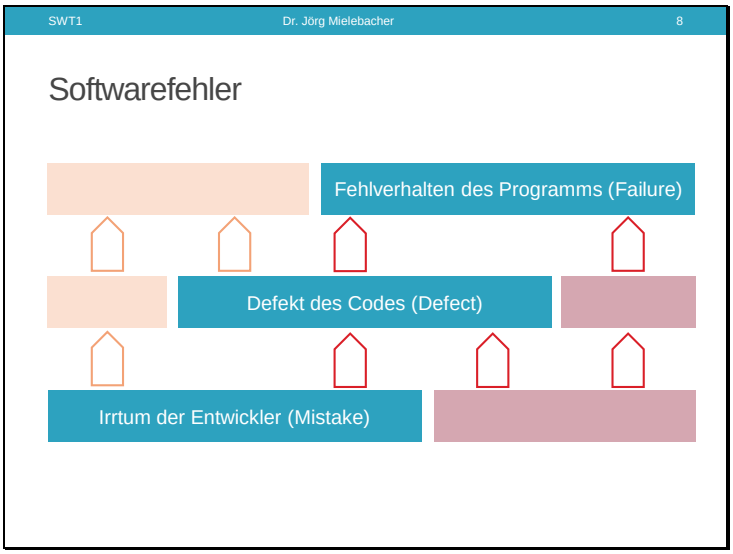


Software wird getestet, um Fehler in ihr aufzuspüren. Dabei testet man üblicherweise im Hinblick auf folgende Aspekte:

- Vorgesehene Funktionen (d.h. sind alle spezifizierten Anforderungen erfüllt)
- Verhalten der Software bei Grenz- und Sonderfällen (z.B. Eingabe unerlaubter Werte, Sonderfälle bei Berechnungen, Division durch 0)
- Unterschiedliche Systemumgebungen (z.B. Betriebssystem, Hardware, Browser)
- Prüfen der einzelnen Verzweigungen (z.B. if, switch-case)
- Verhalten bei Ausfall notwendiger Ressourcen (z.B. Netzwerkausfall, Speicher voll)
- Typische Sicherheitsprobleme (z.B. SQL Injection)
- Verhalten unter Last (d.h. viele gleichzeitige Zugriffe, grosse Datenvolumina)
- Durchsehen des Codes (d.h. Suche nach schlechtem Stil, offenkundigen Fehlern, Formatabweichungen)
- Prüfen der Dokumentation (d.h. Kontrolle der Vollständigkeit, Übereinstimmung von Dokumentation und Code, Verständlichkeit usw.)
- U.v.m.

Bei all diesen Tests wird oft gegen vier grundlegende Anforderungen verstoßen:
1.) Planung: Tests müssen in geeigneter Weise geplant werden.
2.) Objektivität: Tests müssen frei von persönlicher Einschätzung durchgeführt werden und sollen nicht dazu dienen, Entwickler zu verurteilen – die Qualität des Produkts soll im Vordergrund stehen. Auch darf die Durchführung von Tests nicht durch die Angst vor dem Entdecken eigener Fehler behindert werden.
3.) Vollständigkeit: Die notwendigen Tests müssen berücksichtigt und durchgeführt werden. Sie dürfen z.B. nicht aus Zeitgründen übersprungen werden.
4.) Nachvollziehbarkeit: Alle Schritte des Testens müssen auch später nachvollziehbar sein. Hierzu gehört eine zweckmäßige und vollständige Dokumentation aller Schritte von der Planung bis zur Auswertung.

Folie 8

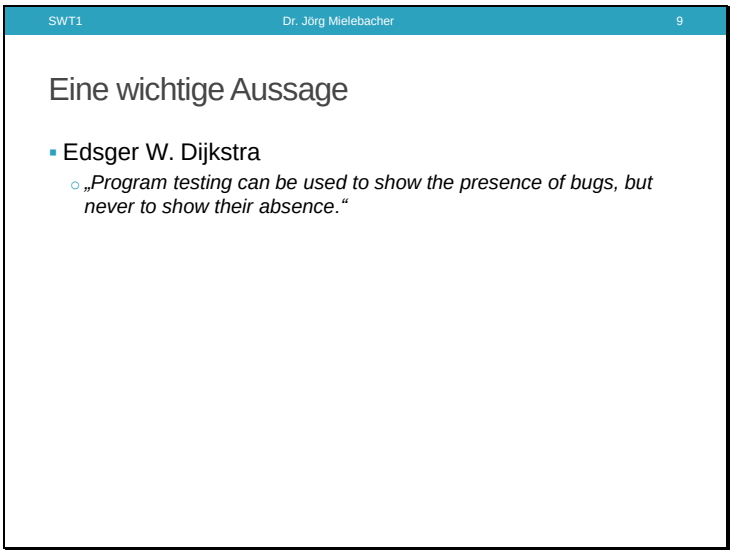


Es gibt unterschiedliche Arten von Softwarefehlern (Logikfehler, Anforderungsfehler, Bedienfehler usw.); dementsprechend gibt es auch unterschiedliche – auf diese Fehlerarten ausgerichtete – Testverfahren, dazu später mehr. Allgemein verbindet man mit Softwarefehler und mit dem Testen die folgende Kausalkette: Ein Irrtum des Entwicklers kann einen Defekt des Codes verursachen. Der Defekt wiederum kann ein Fehlverhalten des Programms zur Folge haben.

Durch Testen des Programms versucht man, Fehlverhalten des Programms hervorzurufen und damit Defekte aufzuspüren. Idealerweise ist mit der Erkennung und Beseitigung des Defekts auch ein Lerneffekt verbunden, um gleichartige Irrtümer der Entwickler künftig auszuschließen.

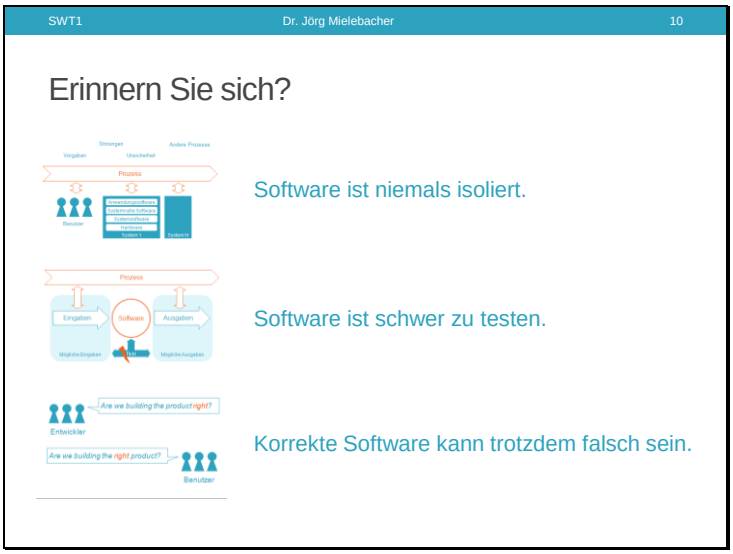
Allerdings: Nicht jeder Defekt wird als Fehlverhalten erkannt - das ist ein grundlegendes Problem des Testens.

Folie 9



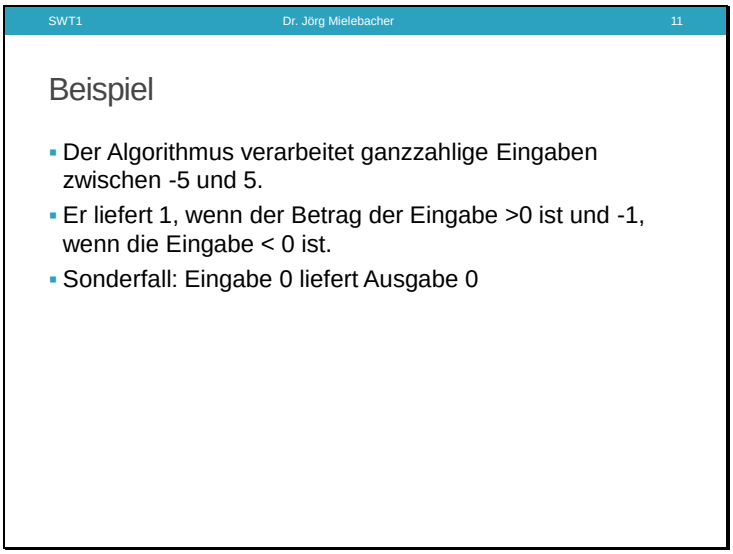
Edsger W. Dijstra gilt als Wegbereiter der strukturierten Programmierung. Neben zahlreichen wichtigen Algorithmen (z.B. der nach ihm benannte Dijkstra-Algorithmus zur Bestimmung von kürzesten Wegen in Graphen) und Konzepten (z.B. Semaphoren zur Synchronisation von Threads) war er maßgeblich an der Entwicklung der Programmiersprache Algol 60 beteiligt.
Dass Tests stets nur die Anwesenheit von Fehlern zeigen, nicht aber ihre Abwesenheit, ist eine zentrale Feststellung, wenn man sich mit der Entwicklung von Software beschäftigt.

Folie 10



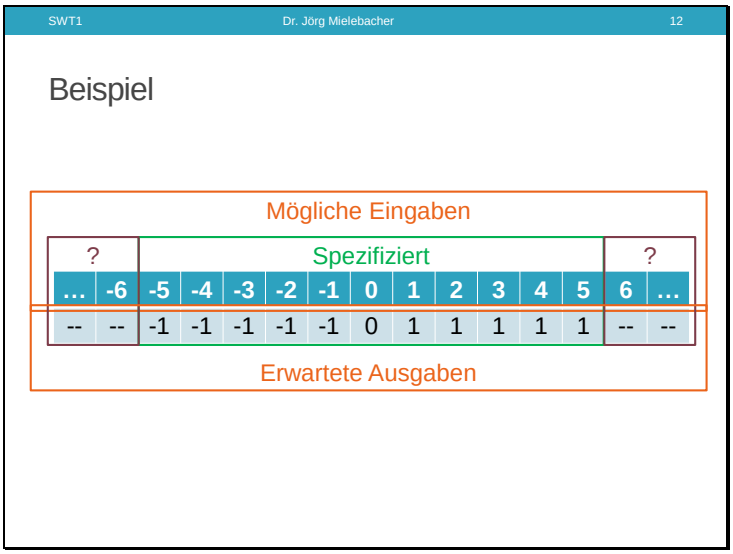
Eine Erinnerung an das erste Kapitel:
1.) Software ist niemals isoliert; sie ist stets eingebettet in ein Umfeld von Benutzern, anderen Systemen usw. Also kann auch dieses Umfeld Fehler verursachen oder zumindest begünstigen.
2.) Software ist schwer zu testen - wegen der Komplexität der möglichen Ein- und Ausgaben sowie Programmzustände einerseits und wegen der Abhängigkeit von der Systemumgebung (s.o.) andererseits.
3.) Eine Software ist korrekt, wenn sie ihre Spezifikation erfüllt. Erfüllt die Spezifikation aber nicht die tatsächlichen Anforderungen der Benutzer, ist die Software für den Benutzer dennoch nicht oder nicht vollständig geeignet.

Folie 11



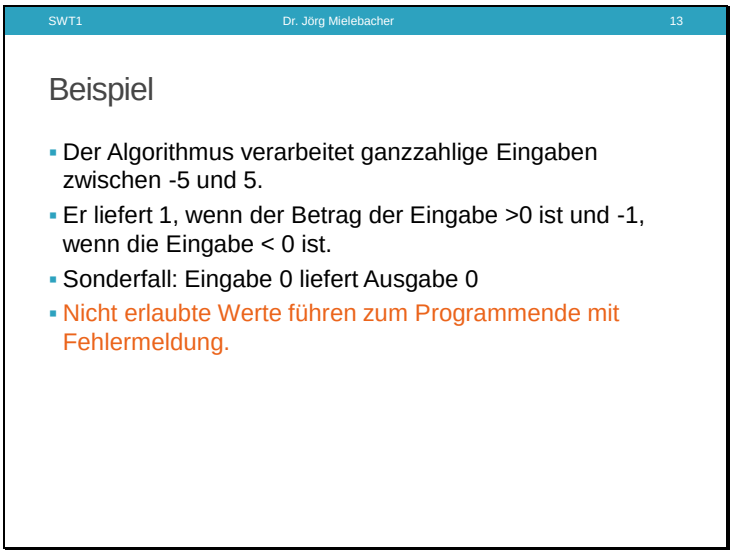
Der hier beschriebene Algorithmus soll getestet werden. Hier sehen Sie die Anforderungen an den Algorithmus.

Folie 12



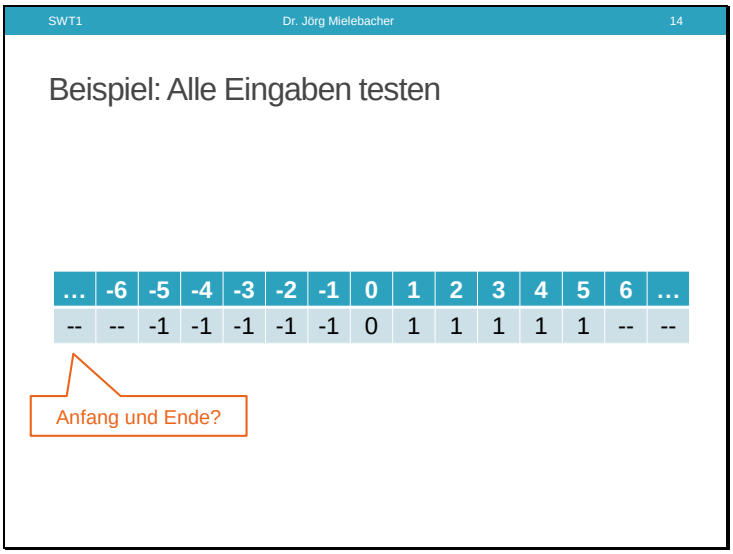
Listet man nun alle möglichen Eingaben auf, so kann man jeder Eingabe die erwartete Ausgabe gegenüberstellen. Dabei zeigt sich schnell, dass nur manche der möglichen Eingaben spezifiziert sind. In diesem Fall muss man entweder die Spezifikation anpassen oder entsprechende Annahmen treffen und dokumentieren.

Folie 13



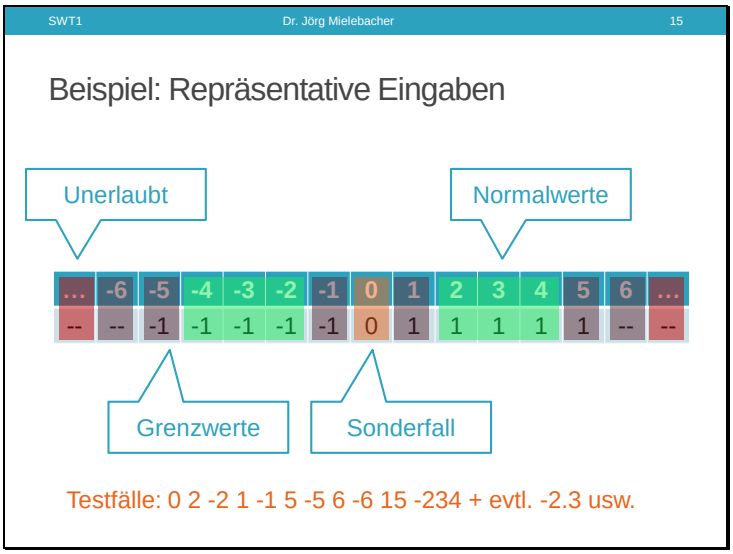
Hier fügen wir die zuvor nicht spezifierten Eingaben hinzu und beschreiben, wie sich das Programm in diesem Fall verhält.

Folie 14



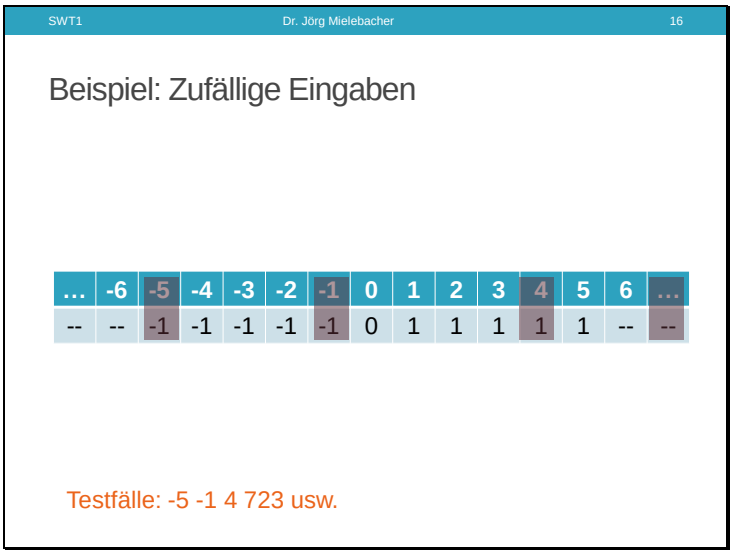
Alle Eingaben zu testen, wäre die umfassende Teststrategie, hierdurch aber auch sehr aufwendig. Ist die Eingabe z.B. ein Integer, müsste man rund 4 Milliarden Werte ausprobieren. Oder aber man geht davon aus, dass bei Wert 1329471923 dasselbe Ergebnis zu erwarten ist wie bei Wert 8372873 usw. Dann hätte man aber nicht mehr alle möglichen Eingaben getestet.

Folie 15



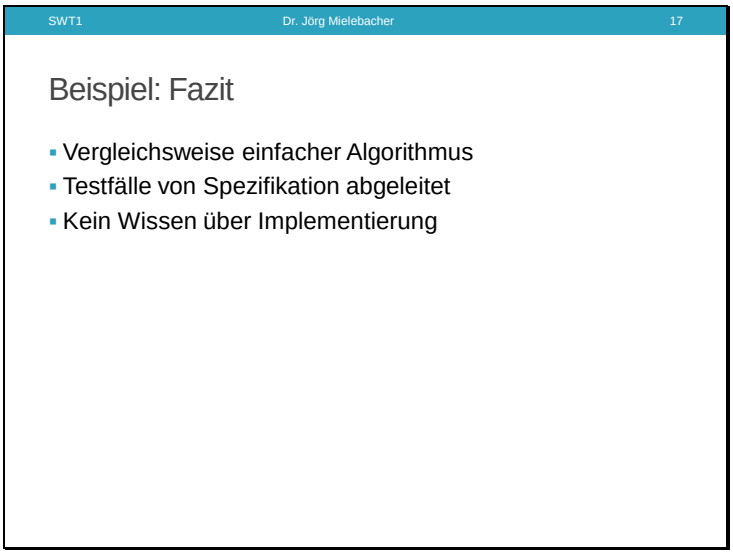
Im Allgemeinen wird man versuchen Klassen ähnlicher Werte zu identifizieren und zu jeder Klasse repräsentative Werte zu testen; üblicherweise alle Grenzwerte und alle Sonderfälle (sofern es sich um einzelne Werte handelt), alle Vertreter alle Klassen von Normalwerten und Vertreter aller Klassen von unerlaubten Werten.

Folie 16



Man könnte aber auch schlichtweg eine Menge zufälliger Eingaben generieren und den Algorithmus damit automatisiert testen. Dies bezeichnet man als Fuzzing bzw. Fuzz testing. Der Ansatz wirkt zwar weniger systematisch, kann aber auch unerwartete Schwachstellen aufdecken.

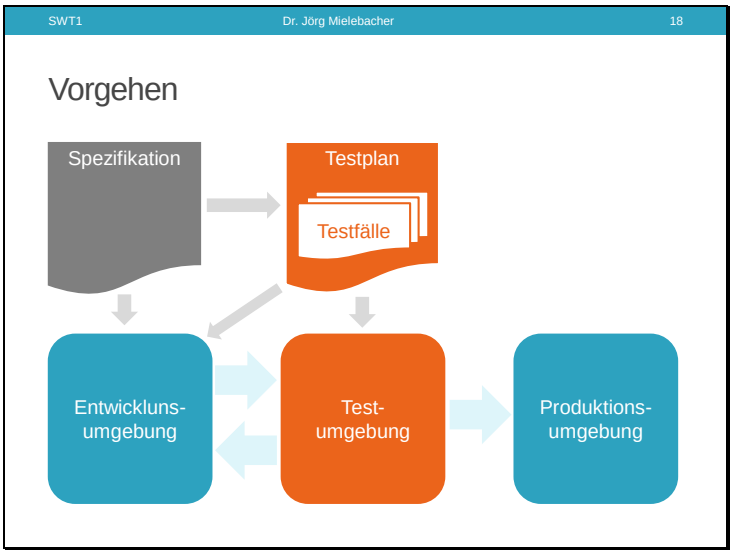
Folie 17



Das hier betrachtete Beispiel basiert auf einem sehr einfachen Algorithmus. Komplexe Abläufe, Zustände, Parallelität usw. spielen hier zunächst keine Rolle, können aber Softwaretests erheblich erschweren.

Das Vorgehen war hier stark geprägt von den spezifizierten Anforderungen (sog. Black-Box-Test), d.h. nur was spezifiziert ist, wird auch getestet. Verbergen sich aber in der Implementierung Schwachstellen (z.B. ungünstiger Entwurf, unsichere Programmkonstrukte usw.) bleibt dies u.U. unerkannt oder kann nur indirekt entdeckt werden.

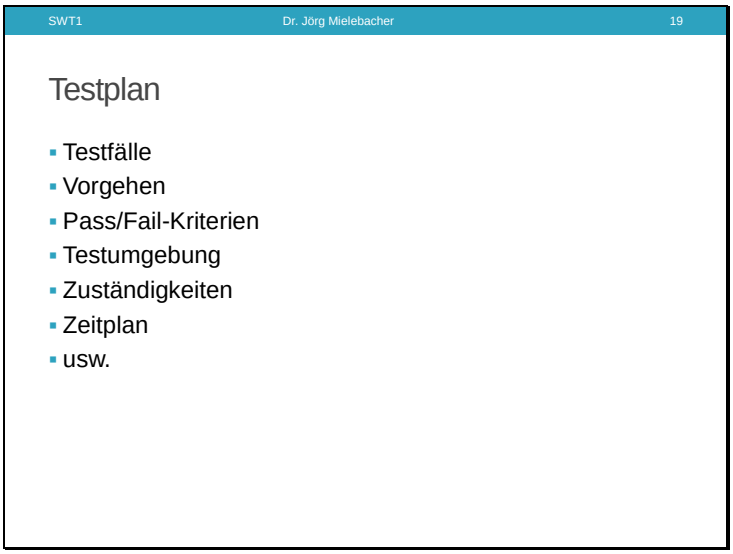
Folie 18



Software wird typischweise in einer eigens dafür vorgesehenen Testumgebung (oder auch in mehrere Testumgebungen) getestet. Zu dieser Umgebung gehören definierte Hard- und Software, darunter spezielle Testwerkzeuge, beispielsweise zur Testautomatisierung (Ausführen von Tests, „Anklicken“ von GUI-Elementen, Generierung von Testdaten usw.). Die Testumgebung ist von der Entwicklungsumgebung getrennt, d.h. erst nach Freigabe des Quellcodes für die Testphase wird dieser Stand der Software in die Testumgebung übernommen. Eine dritte Umgebung ist die sog. Produktionsumgebung – die tatsächliche Einsatzumgebung der Software (z.B. der Server des Auftraggebers). Daneben gibt es häufig noch spezielle Trainingsumgebungen und dergleichen.

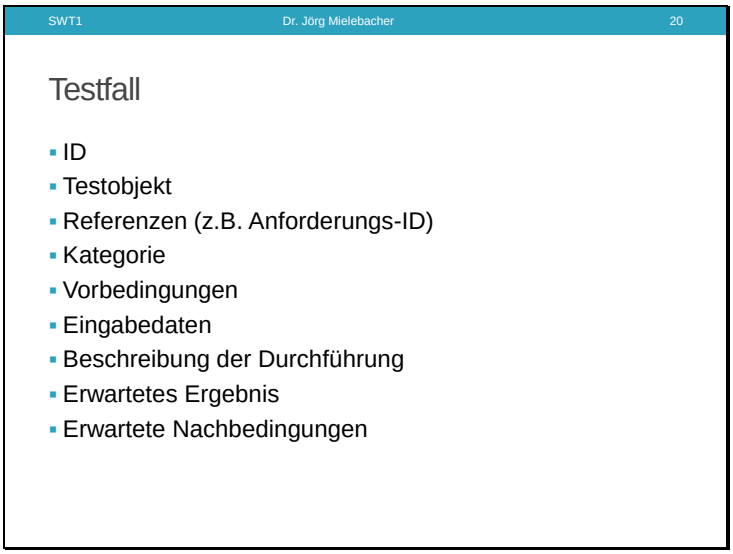
Die Spezifikation der Software ist die Grundlage der Implementierung, darüber hinaus ist sie eine wesentliche Grundlage des sog. Testplans. Dieser beschreibt, auf welche Weise die Software getestet wird und wie die Ergebnisse ausgewertet werden (z.B. Test bestanden oder nicht bestanden). Insbesondere bezieht sich der Testplan auf die einzelnen Testfälle.

Folie 19



Der Testplan ist ein zentrales Dokument der Testphase. Er beschreibt, was getestet (Testobjekt) und wie diese Tests zu erfolgen haben. Z.B. sei das Login-Formular einer Anwendung das Testobjekt. Hierin testet man nun u.a. die Authentifizierung des Benutzers (Test item), wobei man dieses Item für unterschiedliche Bedingungen (gültiges Passwort, ungültiges Passwort, leeres Passwort usw.) prüft. Jeweils vergleicht man erwartetes und beobachtetes Verhalten.
Die Pass-/Fail-Kriterien legen fest, unter welchen Umständen der Test bestanden oder nicht bestanden ist. Außerdem beschreibt er die zu verwendende Testumgebungen (Hardware, Software usw.), Zuständigkeiten (z.B. für Vorbereitung, Genehmigung, Ausführung und Auswertung), Zeitpläne u.v.m.

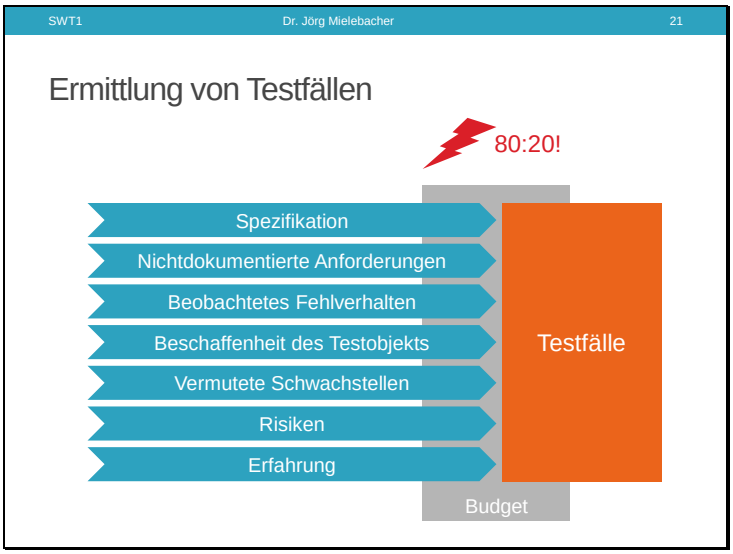
Folie 20



Testfall werden eindeutig durch eine ID bezeichnet. Neben administrativen Angaben wie Autor, Version usw. gehört zu jedem Testfall:

- das Testobjekt (d.h. was wird getestet)
- die einschlägigen Referenzen, z.B. Verweise auf Spezifikationsversionen, Anforderungs-IDs usw.
- ggf. eine Testkategorie (z.B. Benutzeroberfläche, Sicherheit)
- die für die Durchführung des Tests notwendigen Vorbedingungen (z.B. „Benutzer mit Rolle xy ist im System angemeldet.“).
- die zu verwendenden Eingabedaten (Tastatureingaben, zu verwendende Dateien usw.)
- die Beschreibung, wie vorzugehen ist (z.B. „Klicke im Datei-Menü den Eintrag Abmelden an.“)
- das erwartete Ergebnis (bzw. die erzeugten Ausgaben), wenn die vorgesehenen Aktionen mit den vorgesehenen Eingaben erfolgt sind, sowie die damit zusammenhängenden Nachbedingungen (z.B. „Benutzer ist abgemeldet“).

Folie 21

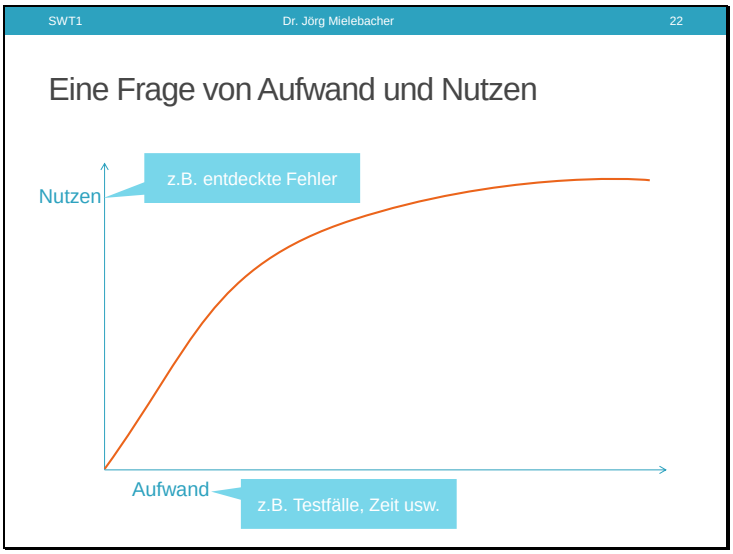


Die Testfälle ermittelt man anhand mehrerer Quellen:

- Spezifikation (d.h. die zu erfüllenden Anforderungen)
- Nichtdokumentierte, aber dennoch wichtige Anforderungen
- Bereits beobachtetes Fehlverhalten des Programms (sog. Regressionstests)
- Beschaffenheit des Testobjekts (Kontrollfluss, Schnittstellen usw.)
- Vermutete Schwachstellen (d.h. Eingabefilterung, Nullpointer usw.)
- Risiken (gemessen an Auftretenswahrscheinlichkeit, Schadensausmaß und Entdeckungswahrscheinlichkeit)
- Erfahrungen mit vergleichbaren Projekten und mit den Teammitgliedern

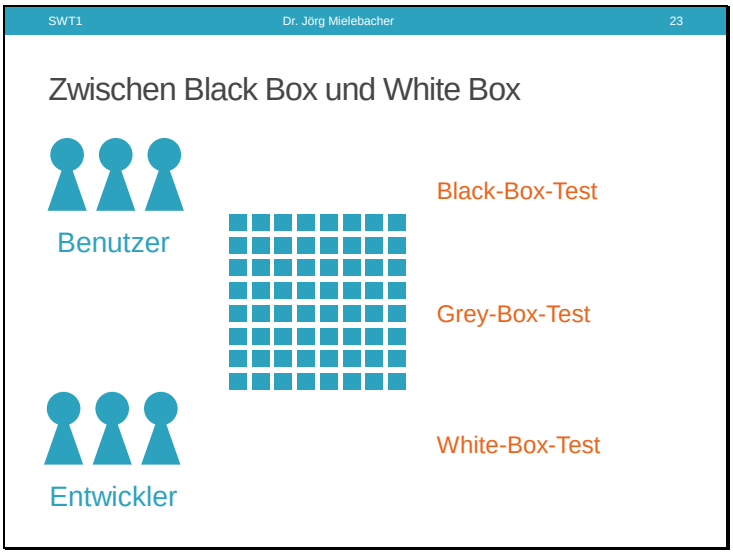
Jeder Testfall erhöht allerdings auch die Dauer (und damit die Kosten) des Tests. Deshalb wird man immer auch die Wirksamkeit dieser Testfälle sowie das Verhältnis von Aufwand und Nutzen (80-20-Regel, Pareto-Prinzip).

Folie 22



Gerade das eben angesprochene 80:20-Prinzip sollte man bei Softwaretests immer gut im Auge behalten: Mit vertretbarem Aufwand (d.h. mit sinnvoll gewählten Testfällen) lässt sich bereits ein angemessener Nutzen erreichen. Es geht also darum, die wirklich kritischen Fehler zu erkennen und zu beseitigen.

Folie 23



Bei Tests unterscheidet man im Wesentlichen drei Herangehensweisen, die den Sichtweisen der jeweiligen Akteure entsprechen:

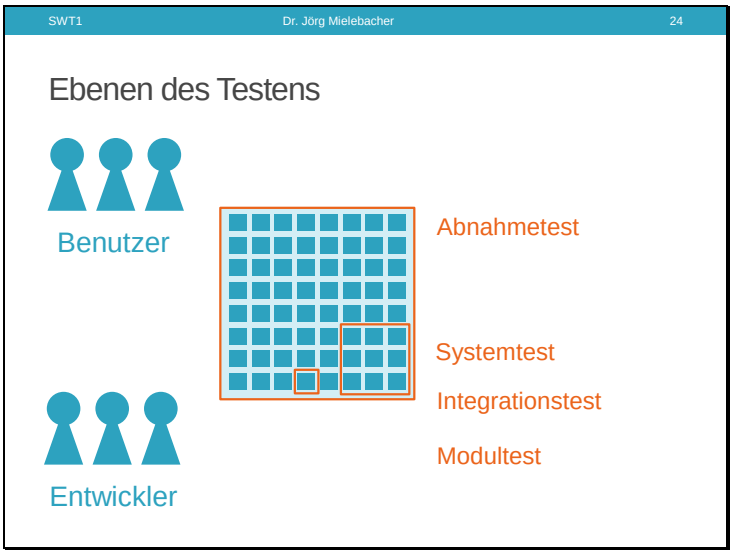
Benutzer sehen nur die Software als solche, haben aber keinen Zugriff auf deren Interna. Die Software ist also eine Black Box, die sich nur auf Einhaltung der Spezifikation überprüfen lässt. Ein Black-Box-Test soll also klären, ob alle verlangten Funktionen wie spezifiziert vorhanden sind. Hierzu gehören:
- Benutzerschnittstelle (Aufbau, Beschriftungen usw.)
- Ablauf der Programmfunktionen
- Benutzereingaben (Überprüfung, Aufbau usw.)
- Programmausgaben (Formate, Genauigkeit usw.)
- Umgang mit Sonderfällen, Grenzfällen, nicht erlaubten bzw. nicht spezifierten Bedienschritten

Im Gegensatz dazu kennen die Entwickler alle Interna der Software; sie haben Zugriff auf den Quellcode und können diesen für die Fehlersuche verwenden. Mit der Kenntnis des Quellcodes lassen sich gezielt Schwachstellen suchen. Hierzu gehören:
- Überprüfung der verschiedenen Ablaufzweige (z.B. bei Verzweigungen)
- Überprüfung von Systemressourcen (z.B. Speicher)
- Test der Fehlerbehandlung
- Prüfen der Übereinstimmung von Quellcode und Dokumentation

(Ein Teil der) Tester hat außerdem Zugriff auf Teile der Interna; beispielsweise kennen Sie Schnittstellen, haben Zugriff auf Logs usw. Dies bezeichnet man als Gray Box. Grey-Box-Tests ähneln daher Black-Box-Tests, erlauben aber weitergehende Untersuchungen, z.B.:
- Überprüfung des Loggings
- Überprüfung von Schnittstellen
- Sicherheitstests

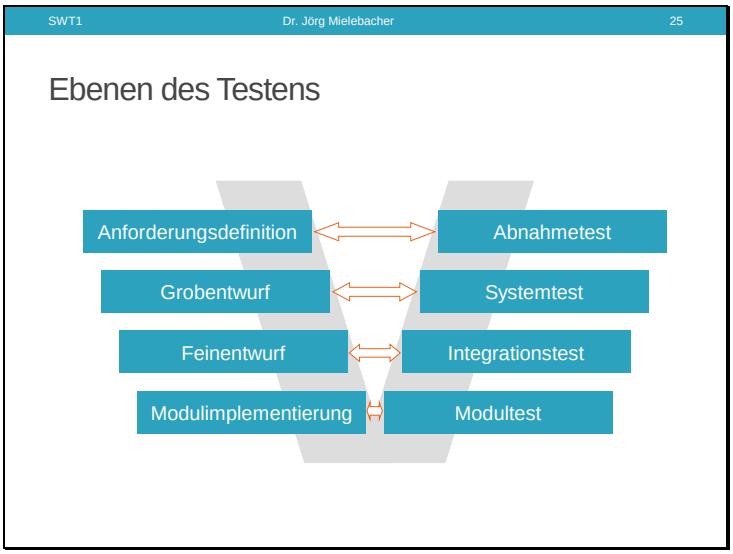
Unter diesen drei Ansätzen gibt es nicht den einen richtigen Ansatz. Vielmehr ergänzen sich diese Ansätze gegenseitig. Beispielsweise sind Black-Box-Tests geeignet, die Erfüllung der Spezifikation zu überprüfen. Sie sind jedoch „blind“ für Fehler, die sich nicht als Abweichung von der Spezifikation ausdrücken.

Folie 24



Tests erfolgen auf unterschiedlichen Ebenen, üblicherweise vom Kleinen zum Großen: Modultests prüfen die einzelnen Softwaremodule. Integrationstests prüfen das Zusammenwirken der Module. Der Systemtest prüft schließlich das Gesamtsystem. Modul-, Integrations- und Systemtests werden üblicherweise von den Entwicklern (bzw. den zugeordneten Testern) durchgeführt, also durch den Auftragnehmer. Im Zuge der Auslieferung der Software führt der Auftraggeber den sog. Abnahmetest durch, auf dessen Grundlage er sich entscheidet, ob die gelieferte Software seiner Bestellung (also seinen Anforderungen) entspricht.

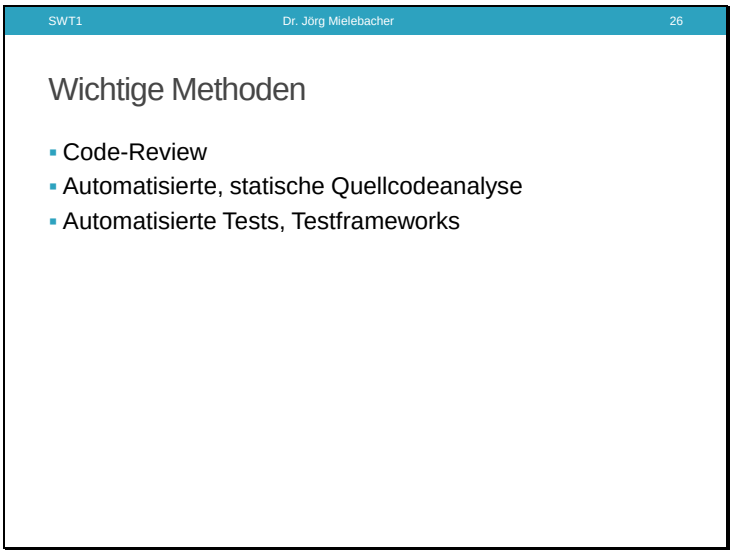
Folie 25



Die genannten Ebenen des Testens entsprechen den einzelnen Phase der Entwicklung. Das gezeigte V zeigt auf der linken Seite Spezifikation, Entwurf und Implementierung hin zu den einzelnen Modulen. Jedem dieser Schritte ist auf der rechten Seite eine Testebene zugeordnet.

Das hier gezeigte Vorgehen bezeichnet man im Kontext der Vorgehensmodelle auch als das sog. V-Modell, dazu mehr im folgenden Kapitel.

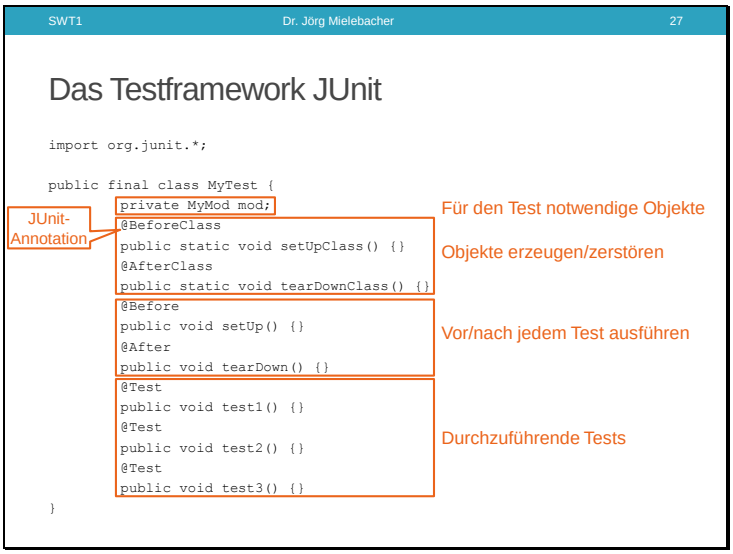
Folie 26



Folgende Methoden sind im Rahmen der Testphase wichtig:

- Code-Review: Im Rahmen eines Code-Reviews wird der Quellcode (oder relevante Teile davon) durchgesehen und auf offenkundige Fehler oder schlechten Programmierstil hin überprüft. Auch die Einhaltung des Styleguides wird dabei überprüft.
- Automatisierte, statische Quellcodeanalyse: Dabei wird zu prüfende Software nicht ausgeführt. Der Quellcode wird auf übliche Fehlerquellen hin untersucht, z.B. nicht initialisierte Variablen, nicht verwendete Variablen, Zugriff auf nicht vorhandene Feldindizes, unerreichbarer Code, falsche Parameterzuordnungen, nicht verwendeter Code. Beispiele solcher Programme sind Lint und Checkstyle.
- Automatisierte Tests (meist durch sog. Testframeworks, wie JUnit) erlauben die automatische Ausführung zuvor programmierter Tests. Die Automatisierung beschleunigt die Testausführung und erzeugt automatische einen erheblichen Teil der notwendigen Dokumentation. Auch sind automatisierte Tests frei von subjektiver Einschätzung und Ermüdung. Meist werden solche Tests in Versionsverwaltungssysteme integriert und automatisch nach einem Commit ausgeführt.

Folie 27



JUnit ist ein Framework für Modultests in Java (www.junit.org). Das Framework ist Open-Source und z.B. in Eclipse bereits integriert. Für andere Sprachen gibt es ähnliche Frameworks (z.B. CppUnit). Die einzelnen Testfälle werden in Quellcode formuliert und lassen sich zu sog. Testsuiten zusammenfassen.

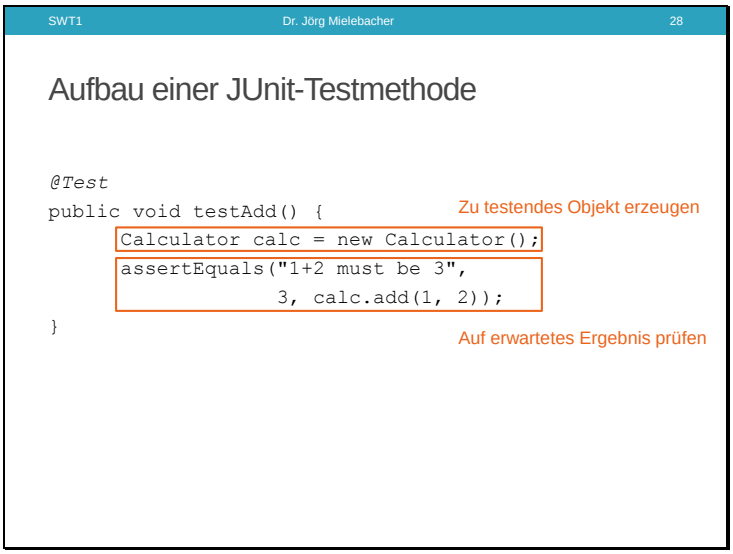
Für jeden Testfall erstellt man eine Klasse (hier: MyTest). Diese enthält die einzelnen durchzuführenden Tests (hier: test1-3) in Form von Methoden. Wie Sie man im Quellcode sehen kann, enthält diese Klasse Annotationen, die mit @ beginnen (z.B. @Test), die Annotationen sind wichtig für JUnit, da sie die Ausführung des Testfalls steuern.

Diejenigen Objekte, die über den gesamten Testfall hinweg benötigt werden, legt man als Attribute des Testfalls an (hier: mod). Diese Objekte lassen sich mit den als @BeforeClass bzw. @AfterClass annotierten Methoden erzeugen bzw. zerstören. Dies gilt auch für alle anderen Aktivitäten, die zur Vorbereitung bzw. am Ende des Testfalls ausgeführt werden sollen.

Während die @BeforeClass und @AfterClass Methoden nur einmal ausgeführt werden, führt JUnit die mit @Before und @After annotierten Methoden vor bzw. nach jedem Test aus (beispielsweise um jeweils vergleichbare Ausgangsbedingungen zu schaffen).

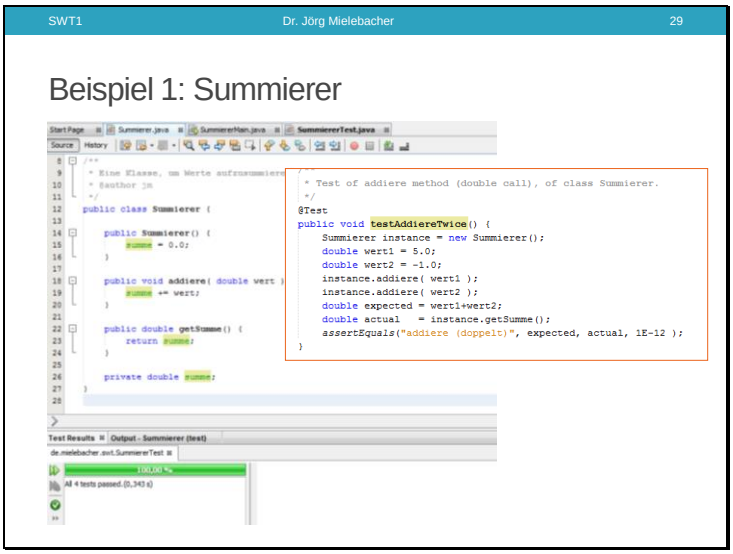
Im obigen Beispielen würde Junit somit folgende Methoden aufrufen:
setUpClass, setUp, test1, tearDown, setUp, test2, tearDown, setUp, test3, tearDown, tearDownClass

Folie 28



Ein Test-Methode ohne weiteren Inhalt würde immer als bestanden gewertet werden. Auf Fehlschlag kann man (unter anderem) durch verschiedene Assertions prüfen. Im obigen Beispiel würde der Test fehlschlagen, wenn die Methode add ein anderes Ergebnis als 3 liefert.

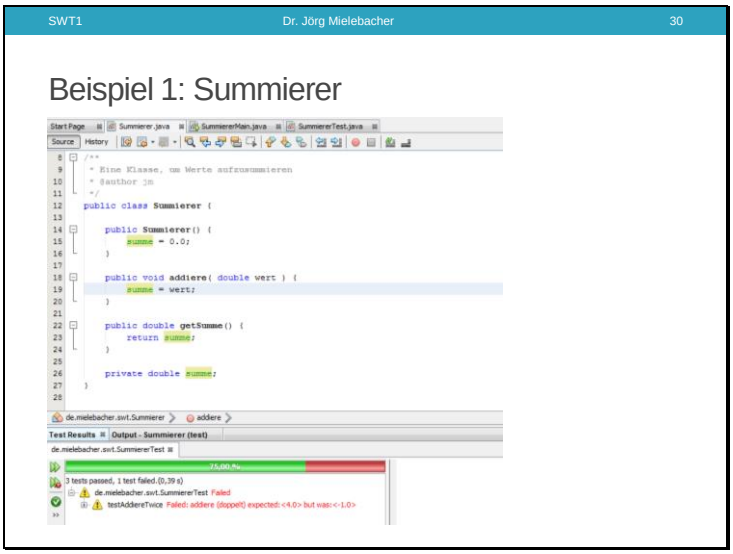
Folie 29



Die hier gezeigte Klasse Summierer hat die Aufgabe, eine Summe zu bilden, d.h. mit jedem Aufruf von „addiere“ wird die Summe um den übergebenen Wert erhöht.

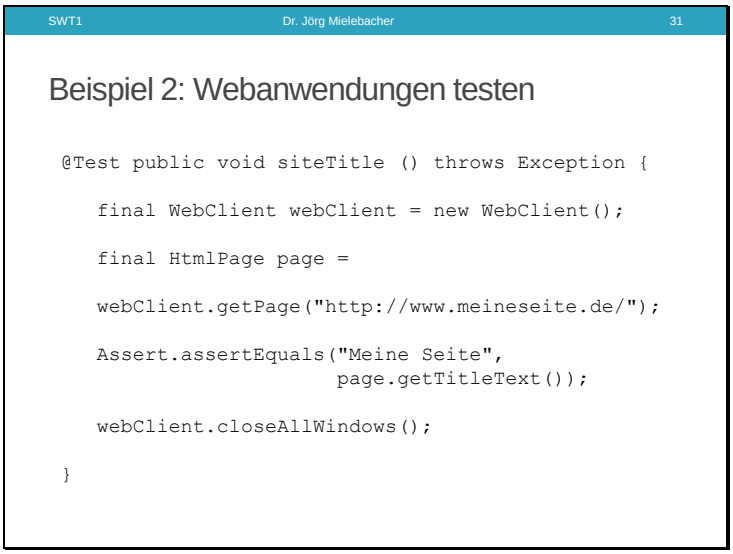
Für diese Klasse wurden Junit-Tests geschrieben. Eine Besonderheit hierbei ist die spezielle Form von assertEquals für double-Werte: Sie erhält als Parameter die Mitteilung, den erwarteten Wert, den tatsächlichen Wert sowie als letzten Parameter (hier: 1E-12) die Toleranz bei der Prüfung auf Gleichheit, d.h. Gleichheit liegt vor wenn $|\text{erwartet-beobachtet}| \leq 10^{-12}$

Folie 30



Hier sieht man, wie durch einen kleinen Fehler in addiere (= statt +=), der zweite Test fehlschlägt. Bei einem einfachen Aufruf von addiere wäre das Ergebnis korrekt gewesen, bei einem mehrfachen Aufruf enthält der Summierer stets den zuletzt addierten Wert, deshalb schlägt dieser Test fehl.

Folie 31

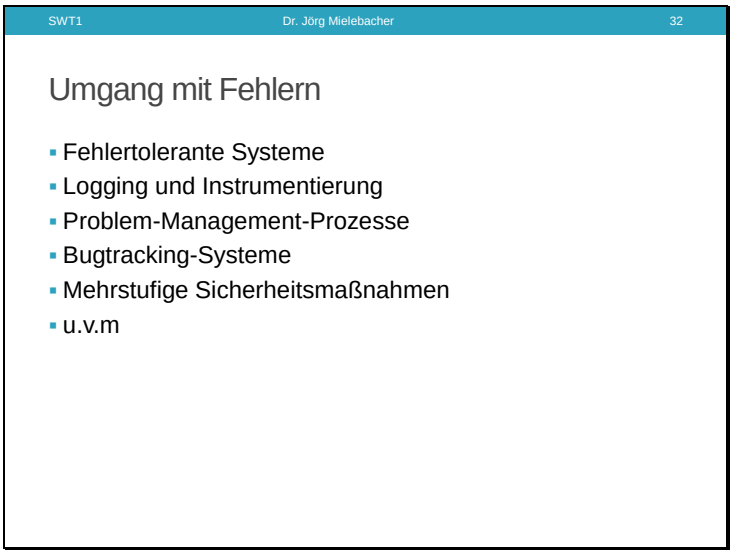


Für das Testen von Webanwendungen gibt es unterschiedliche Ansätze:

- HtmlUnit imitiert unterschiedliche Browser und erlaubt es, diese aus Junit heraus fernzusteuern und Inhalte auszulesen.
- Selenium erlaubt das Aufzeichnen und erneute Abspielen von Bedienschritten in einer Webanwendung.

Das hier gezeigte Beispiel zeigt, wie man mit HtmlUnit prüfen kann, ob der Titel einer Webseite korrekt ist.

Folie 32



Wenn also durch Softwaretests nie die Abwesenheit von Fehlern nachgewiesen wird, sind zusätzliche Maßnahmen notwendig, wenn die Software in den Betrieb übergeht, hierzu gehören unter anderem:

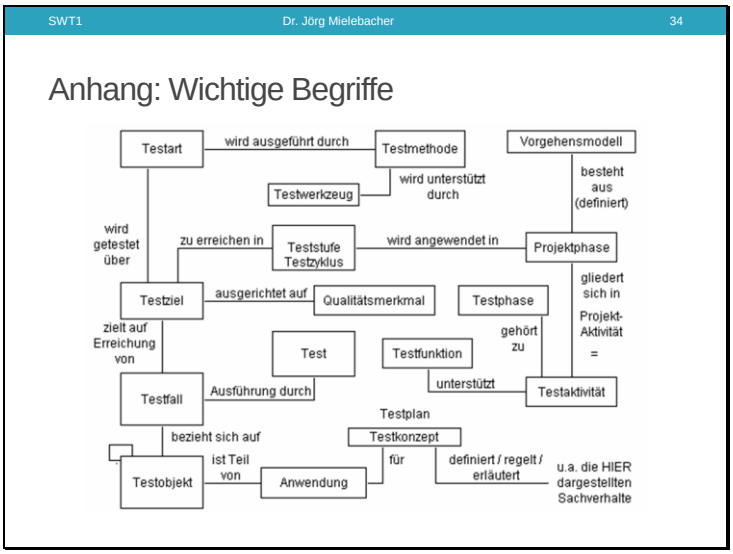
- System sollten stets fehlertolerant entworfen werden, d.h. man nimmt Fehler als vorhanden an und versucht deren Folgen zu minimieren. Ausserdem sieht man häufig redundante Komponenten vor oder zumindest vorkonfigurierte Reserversysteme (z.B. bei Hardwareausfall).
- Logging und Instrumentierung helfen während der Ausführung, Probleme zu erkennen oder Informationen für die Fehlersuche zu erhalten.
- Für die Nutzung der Systeme werden sog. Problem-Management-Prozesse definiert. Diese beschreiben, wer im Fall von Vorkommnissen (Fehler usw.) diese erfasst und wie sie nachverfolgt werden. Üblicherweise stellt der Softwareanbieter einen Helpdesk bereit, dessen Mitglieder Anfragen der Kunden bearbeiten (z.B. Fehlermeldungen, besondere Bedienschritte usw.). Teilweise werden solche Helpdesks auch von den Unternehmen eingerichtet, die eine bestimmte Software verwenden, um schneller auf solche Probleme reagieren zu können.
- Bugtracking-Systeme (z.B. Bugzilla, Trac, Mantis) mit denen sich Fehler berichten und nachverfolgen lassen. Die Fehler werden entweder vom sog. Helpdesk (also den Support-Mitarbeitern, die die Anfragen der Kunden bearbeiten) oder aber von den Nutzern selbst eingetragen. Fehler lassen sich auf diese Weise systematisch erfassen und auswerten; außerdem lassen sich damit Aufgaben für die Entwickler definieren. Bugtracking-Systeme sind häufig mit Versionsverwaltung und Build-Management gekoppelt.
- Da auch sicherheitsrelevante Komponenten fehlerhaft (und damit angreifbar) sein können; setzt man meist mehrstufige Sicherheitsmaßnahmen um, z.B. eine sog. Two-Factor-Authentifizierung (z.B. Passwort und TAN) oder aber die Verzahnung technischer und organisatorischer Maßnahmen (z.B. Sicherheitsschulungen).

Folie 33

Zusammenfassung

- Irrtümer der Entwickler können zu Defekten des Codes führen, diese zu Fehlverhalten des Programms.
- Tests haben zum Ziel Fehlverhalten aufzudecken.
- Tests müssen geplant, objektiv, vollständig und nachvollziehbar sein.
- White-, Gray- und Black-Box-Testen sind drei wichtige Testansätze.
- Automatisierte Tests beschleunigen das Testen und machen es zuverlässiger.
- JUnit ist ein verbreitetes Testframework.
- Tests können immer nur die Anwesenheit von Fehlern zeigen, nie deren Abwesenheit.
- Entwickler müssen Fehlern stets erwarten.

Folie 34



Diese Übersicht stellt einige wichtige Begriffe und ihre Zusammenhänge dar.

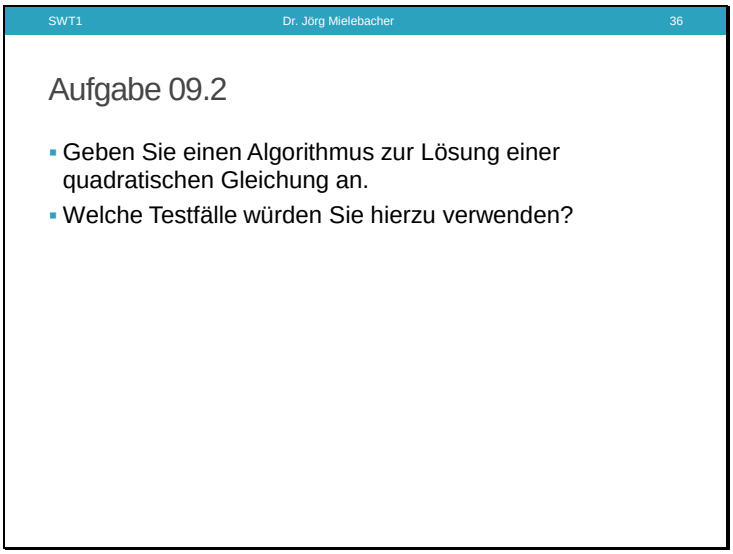
Folie 35

Aufgabe 09.1

- Was versteht man jeweils unter Black-Box-, Gray-Box- und White-Box-Test? Was wird dabei getestet?
- Welche Ebenen von Tests verwendet man meist und wer führt diese Tests durch?
- Welche Informationen beschreiben einen Testfall?
- Was versteht man unter Verifikation? Was unter Validierung?
- Erklären Sie, weshalb Dijkstras These (Tests können nur die Anwesenheit, nie die Abwesenheit von Fehlern zeigen), offensichtlich zutrifft.

Bitte bearbeiten Sie die Aufgaben.

Folie 36



Bitte bearbeiten Sie die Aufgabe.

Folie 37

Aufgabe 09.3

- Eine Webanwendung verwendet eine Authentifizierung mithilfe von Benutzername (nur Kleinbuchstaben und Ziffern) und Passwort (A-Z,a-z,0-9,#?!+-*). Nach Eingabe von Benutzername und Passwort sowie Anklicken des „Login“-Buttons soll der Benutzer angemeldet werden und die Startseite angezeigt werden. Bei einem Fehlschlag soll eine Fehlermeldung erscheinen und ein Eintrag im System-Log vorgenommen werden.
- Welche Tests würden Sie durchführen?

Bitte bearbeiten Sie die Aufgabe.