Vorlesung "Softwaretechnik"

Analytische Qualitätssicherung 2

Lutz Prechelt

Freie Universität Berlin, Institut für Informatik

- Testautomatisierung
  - Testframeworks
  - Test-Modularisierung
  - Aufnahme/Wiederg.-Werkzeuge
- Sonstiges über Testen
  - Heuristiken zur Defektortung
  - Leistungs-, Benutzbarkeits-, Abnahmetests
- Manuelle statische Prüfung
  - Durchsichten und Inspektionen
  - Perspektiven-basiertes Lesen
  - Empirische Ergebnisse
- Automatische statische Prüfung
  - Modellprüfung
  - Quelltextanalyse

Lernziele

- Diverse Ansätze und Werkzeuge zur Testautomatisierung kennen lernen

- Diverse Ansätze für Durchsichten kennen lernen
- Vorteile von Durchsichten gegenüber Tests verstehen

- Einige Ansätze für automatische statische Prüfungen grob kennen lernen

Wo sind wir?: Taxonomie "Die Welt der Softwaretechnik"

Freie Universität Berlin

Welt der Problemstellungen:

- Produkt (Komplexitätsprob.)
  - Anforderungen (Problemraum)
  - **Entwurf (Lösungsraum)**
- Prozess (psycho-soziale P.)
  - Kognitive Beschränkungen
  - Mängel der Urteilskraft
  - Kommunikation, Koordination
  - Gruppendynamik
  - Verborgene Ziele
  - **Fehler**

Welt der Lösungsansätze:

- Technische Ansätze ("hart")
  - Abstraktion
  - Wiederverwendung
  - **Automatisierung**
- Methodische Ansätze ("weich")
  - Anforderungsermittlung
  - Entwurf
  - **Qualitätssicherung**
  - Projektmanagement

Wo sind wir?: Qualitätssicherung

- Einsicht: Man macht beim Bau von SW zahlreiche Fehler
  - die häufig zu schwerwiegenden Mängeln führen

- Prinzipien:
  - **Konstruktive Qualitätssicherung**: Ergreife vorbeugende Maßnahmen, um zu *vermeiden*, dass etwas falsch gemacht wird (Qualitätsmanagement, Prozessmanagement)
  - ***Analytische Qualitätssicherung***: Verwende prüfende Maßnahmen, die entstandene Mängel aufdecken
  - ***Softwaretest***: dynamische Prüfung
  - ***Durchsichten***: manuelle statische Prüfung

Übersicht

Freie Universität Berlin

Analytische QS:

- **Dynamische Verfahren**
  - **Defekttest**
    - Wie wählt man Zustände und Eingaben aus?
    - Wer wählt Zustände und Eingaben aus?
    - Wie wählt man Testgegenstände aus?
    - Wie ermittelt man das erwartete Verhalten?
    - Wann wiederholt man Tests?
    - **Wann/wie kann und sollte man Tests automatisieren?**
  - Benutzbarkeitstest
  - Lasttest etc.
  - Akzeptanztest
- Statische Verfahren
  - ...

Konstruktive QS:

- Test- und Durchsichtsmgmt.
- Prozessmanagement
- Projektmanagement, Risikomanagement

Testwerkzeuge: Arten

- Unittest-Frameworks
  - f. programmatische Tests
- Aufnahme/Wiedergabe-Werkzeuge (auch genannt: Record/Playback, Capture/Replay etc.)
  - vor allem für GUI-Programme, aber auch f. z.B. Web-APIs
- Lasttest-Werkzeuge
- Leistungstest-Werkzeuge: Profiler, Speicherprüfer, ...
- (u.a.)

- Wichtigste Hersteller:
  - Open Source Selenium (Web), JUnit, pytest, phpUnit, ...
    www.opensourcetesting.org
  - HP Enterprise Mercury testing tools
    IBM Rational testing tools
    Borland Silk testing tools

Ein generisches Testframework: JUnit

Freie Universität Berlin

JUnit

- Generisches Testframework für Java
    - doofer Name: Hilfreich für die meisten Arten von Tests, nicht nur Unit-Tests (Modultests)
  - Verwaltungsrahmen für anwendungsspezifische Testfälle
  - In der modernen SW-Entwicklung allgegenwärtig
    - Es gibt ähnliche Frameworks für viele Sprachen: https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks
- Grundideen:
  1. Testfälle liegen in einer sep. Klasse mit annotierten Methoden
     - @Test Testfall; Ergebnisprüfungen mit *assertEquals* etc.
     - Durch diese Vorgaben werden automatisierte Tests gleichmäßig und übersichtlich strukturiert
  2. JUnit (1) ruft viele solche Tests auf, (2) fängt Ausnahmen, (3) zählt Versagen, (4) trägt Ergebnisse zusammen

Eine zu testende Klasse (Triviales Beispiel)

Freie Universität Berlin

```
public class Calculator {
    private static int result;

    public void add(int n) {
      result = result + n;
    }
    public void substract(int n) {
      result = result - 1; //Defect!
    }
    public void multiply(int n) {
      // Not implemented yet!
    }
    public void divide(int n) {
      result = result / n;
    }
    public void square(int n) {
      result = n * n;
    }
    public void squareRoot(int n) {
      for (;;) ; //infinite loop: Defect!
    }
    public void clear() {
      result = 0;
    }
    public void switchOn() {
      result = 0; // etc.
    }
    public void switchOff() { ... }

    public int getResult() {
      return result;
    }
}
```

Source: http://www.devx.com/Java/Article/31983/0/

Zugehörige JUnit-Testklasse

Freie Universität Berlin

```
public class CalculatorTest
        extends TestCase {
    private static Calculator
      calculator = new Calculator();

    @BeforeEach
    public void clearCalculatr() {
      calculator.clear();
    }

    @Test
    public void subtract() {
      calculator.add(8);
      calculator.subtract(3);
      assertEquals(
        calculator.getResult(), 5);
    }
    // also add, divide, squareRoot

    @Test(expected =
    ArithmeticException.class)
    public void divideByZero() {
      calculator.add(1);
      calculator.divide(0);
    }

    @Disabled("not ready yet")
    @Test
    public void multiply() {
      calculator.add(8);
      calculator.multiply(10);
      assertEquals(
        calculator.getResult(), 80);
    }
}
```

JUnit-Testablauf in Pseudocode

Freie Universität Berlin

Zur Ausführung einer Testklasse oder Test-Suite:

FOR each method annotated **@Test**:

- JUnit calls the method annotated **@BeforeEach** to set up the class-specific testing environment
- JUnit calls the method annotated **@Test** to perform the individual test
- JUnit catches any exceptions that are thrown
    - (1) those thrown by '*assert*' indicate explicitly detected failures
    - (2) others indicate other failures
        - (3) except that those announced by *expected=SomeException.class* etc. *must* happen (and will then be ignored)
- JUnit calls the method annotated **@AfterEach** to deconstruct the class-specific testing environment

END FOR

- JUnit reports the number and nature of problems found

JUnit Beispielausgabe

Freie Universität Berlin

java –ea org.junit.runner.JUnitCore CalculatorTest

JUnit version 4.1
E..E.I

There were 2 failures:

1) **subtract**(CalculatorTest)
   java.lang.AssertionError: expected:<5> but was:<7>
   at org.junit.Assert.fail(Assert.java:69)

2) **squareRoot**(CalculatorTest) java.lang.Exception:
   test timed out after 1000 milliseconds at
   TestMethodRunner.runWithTimeout

FAILURES!!! Tests run: 6, Failures: 2

Was ist das Problem, wenn man solche Timeouts benutzt?

JUnit: Sonstiges

- Test-Suites: **@Suite.SuiteClasses**({A.class, B.class})
- Tabellen-gesteuerte Tests:
  - Annotate test class with **@RunWith(Parameterized.class)**
  - Include a method **@Parameters**
    public static Collection data()
    that returns the test data table
  - The methods **@Test** have an entry of the Collection as parameter and will be called once for each of them
- Test-Parallelisierung
- Test-Auswahl und -Reihenfolge
  - Test-Gruppierung mit @tag, Aufruf für einzelne Testgruppen
  - Zuerst-Ausführen von neuen/schnellen/geradefehlgeschlagenen Tests
- IDE-Integration:
  - Ein Plugin erleichtert Aufrufe und verschönert Ausgaben
- u.v.a.m.

Test-Modularisierung

Freie Universität Berlin

- Komplexe automatisierte Tests brauchen Modularisierung!
  - Trenne Testinhalt (Anwendungslogik) von Testmechanik (Test vorbereiten, Teile aufrufen, Ergebniswerte abholen)
- Vor allem für drei Arten von Tests wichtig:
  - Integrationstests, die komplexen Zustand herstellen müssen
  - Tests, die getrennte Systemteile aufrufen
    - ggf. über Technologiegrenzen hinweg
  - GUI-Tests

Schwieriges Thema!

Welche 2 Hauptvorteile bekommt man durch gute Modularisierung bei Tests?

Modularisierung von Web-GUI-Tests: Selenium

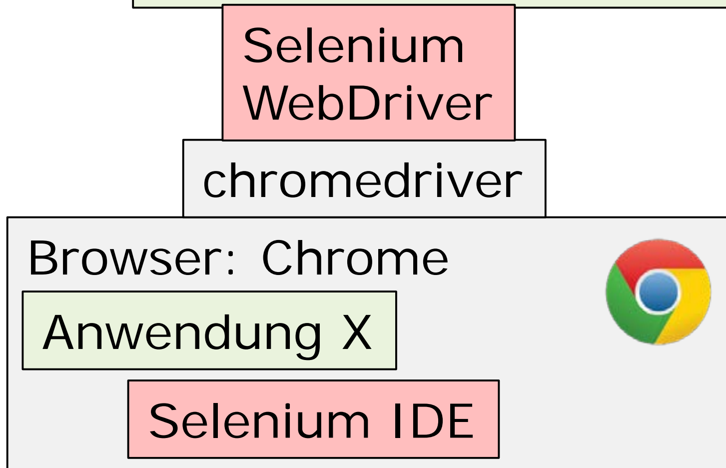
Freie Universität Berlin

(Schlecht modularisiert!)

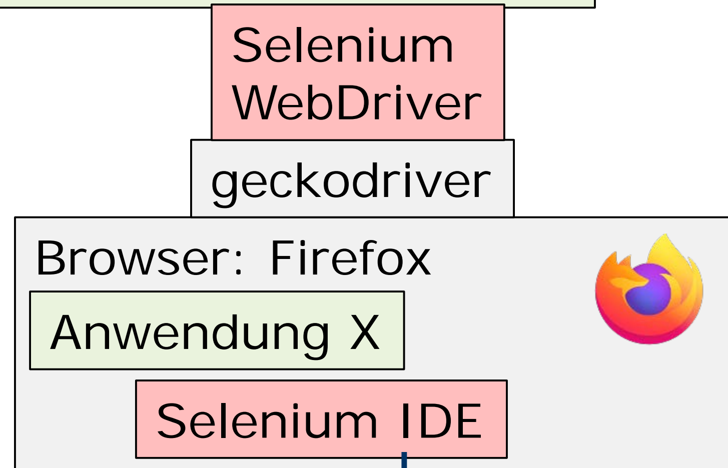
```
TestfallFuerX.java:
import org.openqa.selenium.WebDriver
...
driver.findElement(By.id("price")).sendKeys("10");
...
String result = driver.findElement(...
assertEquals(result, "11.90");
```

Legende
eigener Code
Selenium-Framework
Browser-Hersteller

Selenium WebDriver — chromedriver



Selenium WebDriver — geckodriver



...

WebDriver exists for C#, Java, JS, Python, Ruby and for Chrome/Edge/FF/Opera/Safari browsers

Selenium IDE: Records interactions, exports them for WebDriver in C#, Java, Python, Ruby

Defektortung (Systemebene): 9 Heuristiken

Freie Universität Berlin

1. Verstehe das System
   - (auch wenn's schwerfällt)
2. Reproduziere das Versagen
   - (auch wenn's schwerfällt)
3. Nicht denken, hingucken!
   - selbst wenn Du glaubst Du weisst was los ist
4. Teile und herrsche
   - Keine voreiligen Schlüsse bitte
5. Ändere immer nur eine Sache
   - selbst wenn sie trivial erscheint
6. Mach Notizen was passiert
   - schriftlich!
7. Prüfe Selbstverständlichkeiten
   - zumindest nach einiger Zeit vergeblicher Suche
8. Frag Außenseiter um Rat
   - zumindest nach einiger Zeit vergeblicher Suche
9. Wenn Du es nicht repariert hast, ist es auch nicht repariert
   - also repariere es und prüfe dann nochmal nach

Wo ist Debugging am schwersten: Bei Modul-, Integrations- oder Systemtests?

Mehr dazu im **Kurs "Debugging"** (zuletzt im SoSe 2007)

Übersicht

Analytische QS:

- **Dynamische Verfahren**
  - Defekttest
    - Wie wählt man Zustände und Eingaben aus?
    - Wer wählt Zustände und Eingaben aus?
    - Wie wählt man Testgegenstände aus?
    - Wie ermittelt man das erwartete Verhalten?
    - Wann wiederholt man Tests?
    - Wann/wie kann und sollte man Tests automatisieren?
  - **Benutzbarkeitstest**
  - **Lasttest**
  - **Akzeptanztest**
- Statische Verfahren
  - ...

Konstruktive QS:

- Test- und Durchsichtsmgmt.
- Prozessmanagement
- Projektmanagement, Risikomanagement

Benutzbarkeitstest

- Teil des *Usability Engineering* (Benutzbarkeits-Gestaltung)
  - Idealerweise eingebettet in Prozess mit *User-Centered Design*
- Prüft, ob echte Benutzer_innen in der Lage sind, die Funktionen des Systems zu nutzen
  - und wo dabei Schwierigkeiten auftreten
- Verfahren ist meist die Beobachtung solcher Benutzer_innen
  - bei freier Benutzung oder
  - beim Lösen vorgegebener Aufgaben
- anschließende Verbesserung der Software
- und erneute Beobachtung

Sehr wichtig und wertvoll,
aber komplexes separates Thema

https://www.usability.gov

Qualitätsmerkmale von Software (ganz grob)

Freie Universität Berlin

Externe Qualitätseigenschaften (aus Benutzersicht)

- Benutzbarkeit
  - Bedienbarkeit, Erlernbarkeit, Robustheit, ...
- Verlässlichkeit
  - Zuverlässigkeit, Verfügbarkeit, Sicherheit, Schutz
- Brauchbarkeit
  - Angemessenheit, Geschwindigkeit, Skalierbarkeit, Pflege, ...
- ...

(Man kann diese Listen auch ganz anders machen.)

Interne Qualitätseigenschaften

- Zuverlässigkeit
  - Korrektheit, Robustheit, Verfügbarkeit, ...
- Wartbarkeit
  - Verstehbarkeit, Änderbarkeit, Testbarkeit, Korrektheit, Robustheit
    - dafür nötig sind z.B. Dokumentiertheit, Flexibilität, etc.
- Effizienz
  - Speichereffizienz, Laufzeiteffizienz, Skalierbarkeit
- ...

Interne Q. sind Mittel zum Zweck!

Ziel von Benutzbarkeitstests

Ziel von Last-/Stresstests

Lasttest, Leistungstest, Stresstest

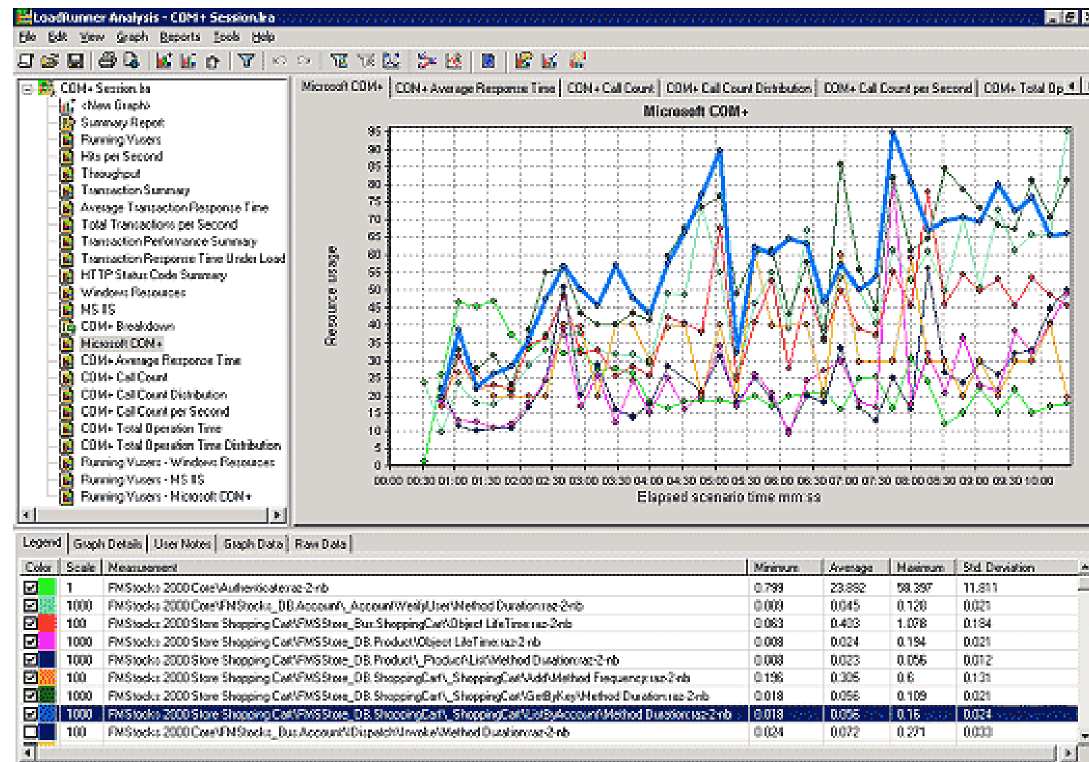
Schicken verschiedene Arten massenhafter Eingaben

Fragen:

- **Lasttest**: Kann das System genügend viele Benutzer/innen zugleich schnell genug bedienen?
  - Wichtig insbesondere im Web
- **Leistungstest**: Worauf wie die Antwortzeit verwandt?
  - Grundlage für Leistungsoptimierungen: Die lohnen nur an wenigen Stellen
- **Stresstest**: Überlebt das System auch Überlasten, massenhaft unsinnige Eingaben und ähnliches?
  - Wichtig bei lebens- und geschäftskritischen Systemen
  - Deckt sogar Defekte in Betriebssystemkernen auf („fuzzing"), siehe "Month of the Kernel Bugs" (Nov. 2006) http://projects.info-pull.com/mokb/

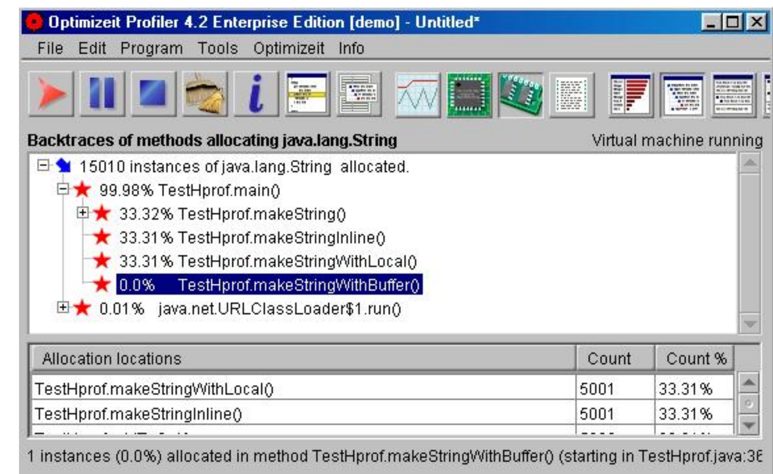
Werkzeuge für Lasttests

- Simulieren die Aktionen zahlreicher Benutzer/Aufrufer
- Werkzeuge verfügbar
  - Kommerziell: Rational Performance Tester, LoadRunner, SilkRunner
  - Open Source: JMeter, OpenSTA, The Grinder
- Für unterschiedl. Plattformen, vor allem für verteilte Systeme
  - http/https
  - RDBMs
  - EJB
  - SAP R/3
  - CORBA, COM+
  - etc.
- Noch ein komplexes, separates Thema
  - Hardwareaufwand
  - Entwurf von Szenarios



Werkzeuge für Leistungstest

- Leistungstests dienen insbesondere als Grundlage für die Optimierung von Entwürfen und Implementierungen
  - Laufzeitverhalten; Speicherbedarf
- Hilfswerkzeuge dabei sind sogenannte Profilierer (profiler)
  - z.B. perf/hperf; http://java-source.net/open-source/profilers
- Sie erzeugen z.B. Ausgaben folgender Art:
  - Laufzeitprofilierer: *"Die Methode A wurde 126 745 mal aufgerufen, verbrauchte netto 22,6% der Laufzeit und brutto (also mit Unteraufrufen) 71,2% der Laufzeit."*
    - und das für alle Methoden
    - sowie Informationen über Threads
  - Speicherprofilierer: *"Von Klasse B wurden 1.793.233 Exemplare erzeugt (700.482 KB). Davon existieren noch 287 Exemplare (115 KB)."*



Akzeptanztest

- Dient dazu, dem Auftraggeber zu demonstrieren, dass das Produkt nun tauglich ist
- Beachte den Wechsel des Ziels:
  - Defekttests und Benutzbarkeitstests sind erfolgreich, wenn sie Mängel aufdecken
    - denn das ist ihr Zweck
  - Akzeptanztests sind hingegen erfolgreich, wenn Sie keine (oder nur geringe) Mängel aufdecken
- Akzeptanztests sollten direkt aus z.B. Use Cases hergeleitet werden
  - Wer macht das idealerweise: Entwickler_innen, Tester_innen oder Auftraggeber_innen?

Gemeinsame Schwäche aller Testverfahren

- Alle testenden Verfahren führen zunächst nur zu Versagen
- Das Versagen muss dann auf einen Mangel zurückgeführt werden (bei Defekttest genannt **Debugging**)
  - Siehe oben: Heuristiken für die Defektlokalisierung
- Das kann sehr **aufwändig** sein:
  1. Das in Frage kommende Codevolumen ist evtl. sehr groß
  2. Evtl. spielen mehrere Mängel zusammen
  3. Oft wirkt ein Mangel zeitlich lange bevor man das Versagen sieht
- In dieser Hinsicht sind **statische und konstruktive Verfahren günstiger**:
  - Die Aufdeckung des Mangels geschieht hier meist direkt am Mangel (insbesondere: Defekt)
  - Die Lokalisierungsphase entfällt deshalb

Übersicht

Analytische QS:

- Dynamische Verfahren (Test)
  - Defekttest
  - Benutzbarkeitstest
  - Lasttest
  - Akzeptanztest

- **Statische Verfahren**
  - **Manuelle Verfahren**
    - **Durchsichten, Inspektionen**
  - Automatische Verfahren
    - Modellprüfung
    - Quelltextanalyse

Konstruktive QS:

- Test- und Durchsichtsmgmt.
- Prozessmanagement
- Projektmanagement, Risikomanagement

Manuelle statische Verfahren

- Bei manuellen statischen Verfahren werden SW-Artefakte von Menschen **gelesen** mit dem Ziel Mängel aufzudecken

1. Arbeitsergebnisse werden gründlich auf Mängel hin durchgesehen (oft separat von zwei Personen)
   - Mängel können sein:
     - Defekte
     - Verletzungen von Standards
     - Verbesserungswürdige Lösungen
2. Mängel werden dokumentiert und ggf. diskutiert
3. Mängel werden beseitigt
   - evtl. nicht alle (wg. Kosten/Nutzen-Abwägung)
4. Nach heiklen Korrekturen evtl. erneute Prüfung

Vorteile manueller statischer Verfahren

Freie Universität Berlin

1. Im Gegensatz zum Test benötigen solche Verfahren keinen ausführbaren Code
   - sondern sind anwendbar auf **alle Arten von Dokumenten**: Anforderungen, Entwürfe, Code, Testfälle, Dokumentationen
     - und sogar auf Prozessdokumente wie Projektpläne u.ä.
2. Dadurch werden Mängel **früher aufgedeckt**, was viel Aufwand spart
3. Außerdem haben die Verfahren **Zusatznutzen** neben der Aufdeckung von Mängeln:
   - **Kommunikation**: Verbreitung von Wissen über die Artefakte (und damit verbundene Anforderungen und Entwurfsideen) im Team
   - **Ausbildung**: Wissenstransfer über Technologie und gute Strukturen/Stil

Begriffe

Freie Universität Berlin

- Manuelle Prüfmethoden laufen unter vielen verschiedenen Bezeichnungen:
  - Durchsicht (review)
  - Kollegendurchsicht (peer review)
  - Inspektion (inspection)
  - Durchgang (walkthrough)
  - Formelle Technische Durchsicht (formal technical review, FTR)
  - und anderen
- Bei manchen Leuten haben manche oder alle dieser Begriffe eine recht genau festgelegte Bedeutung
- Leider nicht überall die gleiche

- Vorschlag: Meistens von "Durchsicht" sprechen
  - Und bei Bedarf präzisieren, was man damit meint

Beispiel 1: Kleine Codedurchsicht (halbformell)

Freie Universität Berlin

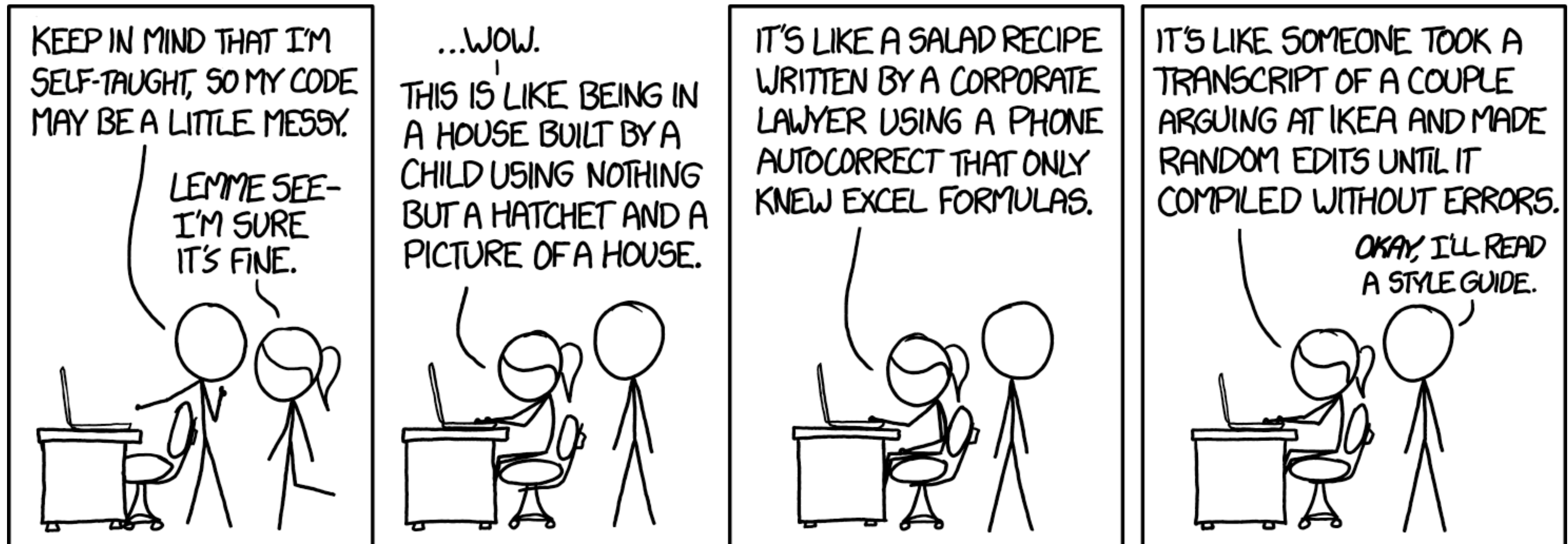
Peer Review:

- Entwickler A hat 4 zusammengehörige Klassen fertig gestellt
  - samt automatisierter Modultests ("unit tests")
- Bittet Entwicklerin B, die 4 Klassen zu begutachten
    - insgesamt 600 Zeilen Code
  - B kennt die Anforderungs- und Entwurfsüberlegungen, aus denen sich ergibt, was die Klassen leisten sollten
- B nimmt sich dafür 3 Stunden Zeit und meldet dann entdeckte Mängel an A zurück:
  - 2 vergessene Funktionen
  - 2 Zweifel an Bedeutung von Anforderungen
  - 4 Fehler in Steuerlogik
  - 1 Fehler in Testfall
  - 5 übersehene Fehlerfälle
  - 4 Vorschläge zur Verbesserung der Robustheit
  - 3 Verstöße gegen Entwurfs-/Kodier-/Kommentierrichtlinien

Werkzeugunterstützung z.B. GitHub pull requests oder Gerrit

Mehrwert von Codedurchsichten gegenüber Defekttests?

- Auch Prüfung von Entwurfs- und Kodierstil, nicht nur von Programmlogik
- Entdeckung fehlender Überprüfungen in automatisierten Tests u.a.



https://www.xkcd.com/1513/

Beispiel 2: Fagan-Inspektion (hoch formell)

Freie Universität Berlin

Vorgeschlagen von M. Fagan und H. Mills bei IBM ca. 1975

- Ziel: Möglichst viele Schwächen im Dokument finden
- Sehr aufwändiger Teamprozess:
  1. Planung: Team zusammenstellen (Autor_in, Moderator_in, Schriftführer_in, 2 bis 5 Gutachter_innen)
  2. Einführungstreffen: Autor_in und Moderator_in erklären Produkt und Inspektionsziele
  3. Lesephase: Gutachter_innen sehen Produkt durch und machen sich damit vertraut
  4. Inspektionstreffen: Team sucht gemeinsam nach Mängeln, Schriftführer_in protokolliert Ergebnisse, Autor_in fragt ggf. nach Klarstellung
  5. Autor_in korrigiert Produkt, Moderator_in kontrolliert
  6. Sammlung statistischer Daten (Größe, Aufwand, Defekte)
  7. Evtl. neue Inspektion des korrigierten Dokuments

Lohnt am ehesten für Anforderungen oder Architekturentwürfe

Eine Lesetechnik: Perspektiven-basiertes Lesen

Freie Universität Berlin

Perspective-based Reading (PBR):

- Besonders geeignet für
  - natürlichsprachliche Dokumente in frühen Phasen (Anforderungen, Architektur, Grobentwurf, GUI-Entwurf)
  - Code
- Es gibt mehrere Gutachter_innen
- Jede_r verwendet zur Analyse eine andere Perspektive, z.B.:
  - Endbenutzer_in (evtl. verschiedene Benutzergruppen)
  - Entwerfer_in, Implementierer_in
  - Tester_in etc.
- Zu jeder Perspektive gehören andere Fragen und Schwerpunkte und andere Arten von möglichen Mängeln
  - Jede Gutachter_in bekommt eine Beschreibung, die ihre Perspektive erklärt (inkl. Checklisten)
- Erhöht Effektivität durch Senken der Überlappung

Andere Lesetechniken

- Object-Oriented Reading Techniques (OORTs)
  - Familie von Lesetechniken für Architektur- und Grobentwürfe
- Use-Based Reading (UBR)
  - Perspective-based Reading angewandt auf Entwürfe von Bedienschnittstellen
- Defect-Based Reading (DBR)
  - Spezialverfahren für Anforderungsdokumente in Form von Zustandsautomaten

Beispiel 3: Selbst-Begutachtung (sehr informell)

Freie Universität Berlin

- Anderes Extrem: Autor_in sieht ihr Produkt allein durch
- Kann sehr effektiv sein
  - Reales Beispiel:
    - 4 Monate Arbeit, dann 2 Tage Durchsicht,
    - 45% Defekt-Entdeckungsquote,
    - Rest durch 2 Monate Test
      - Hat also wohl weitere 1-2 Monate Testzeit eingespart!
  - Effektiv vor allem, wenn typische persönliche Defektarten bekannt sind
  - und die Begutachtungsgeschwindigkeit empirisch optimiert wird
- Nutzen hängt stark von Einstellung ab
  - "Ich will viele Defekte aufdecken"
  - Ein paar Tage Abstand nach der Entwicklung sind empfehlenswert

Beispiel 4: Feinkörnige, werkzeuggestützte Code Reviews

Freie Universität Berlin

Heute weitest verbreiteter Typ von Durchsichten

- Begutachtet wird eine kleine Menge zusammengehöriger Änderungen an der Software
  - Umfang: "Je kleiner, desto besser"
- Oft formalisiert und routinemäßig
  - z.B. "jede Änderung braucht zwei 'OKs' aus Durchsichten"
  - erzwungen über Versionsverwaltungswerkzeuge
- Auch anwendbar auf Dokumentation, Konfiguration, Testfälle, etc.

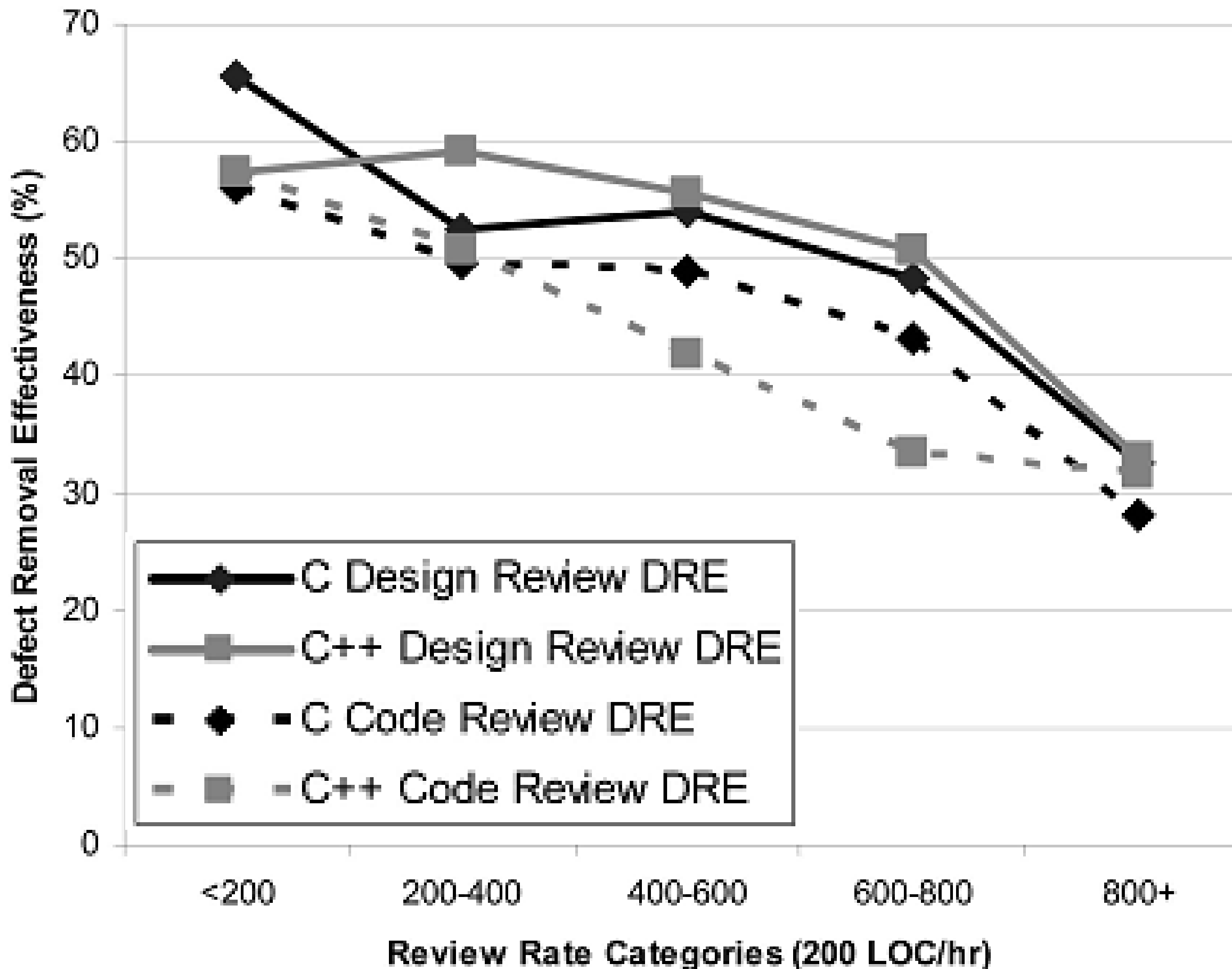
- Vorteile?
  - Verwaltungsaufwand gering
  - Reviewer können das "nebenbei" machen
- Nachteile?
  - Überblick geht leicht verloren

Gängig in agilen Prozessen

Empirische Resultate über Durchsichten

- Codedurchsichten finden mehr Defekte pro Stunde als Test
- Codedurchsichten finden andere Defekte als Test
- Wichtigste Steuergröße: Begutachtungsgeschwindigkeit
  - für Code: günstig sind ca. 50–300 Zeilen pro Stunde
- Fagan-Codeinspektionen sind ineffizient
  - große Überschneidung der Resultate verschiedener Gutachter
  - Kein Mehrwert durch Treffen: Separate Durchsicht reicht
  - Lange Laufzeit (Kalendertage): Prozessverzögerung
- Selbstbegutachtung kann sehr effektiv sein
  - Verlangt jedoch den Willen und die Geduld
- Durchsichten in frühen Phasen sparen besonders viel Kosten
- Perspektiven-basiertes Lesen:
  - Effektiver als wenn jeder alles sucht
  - Effizient durch hohe Gesamtausbeute

Begutachtungsgeschwindigkeit (Bsp.)

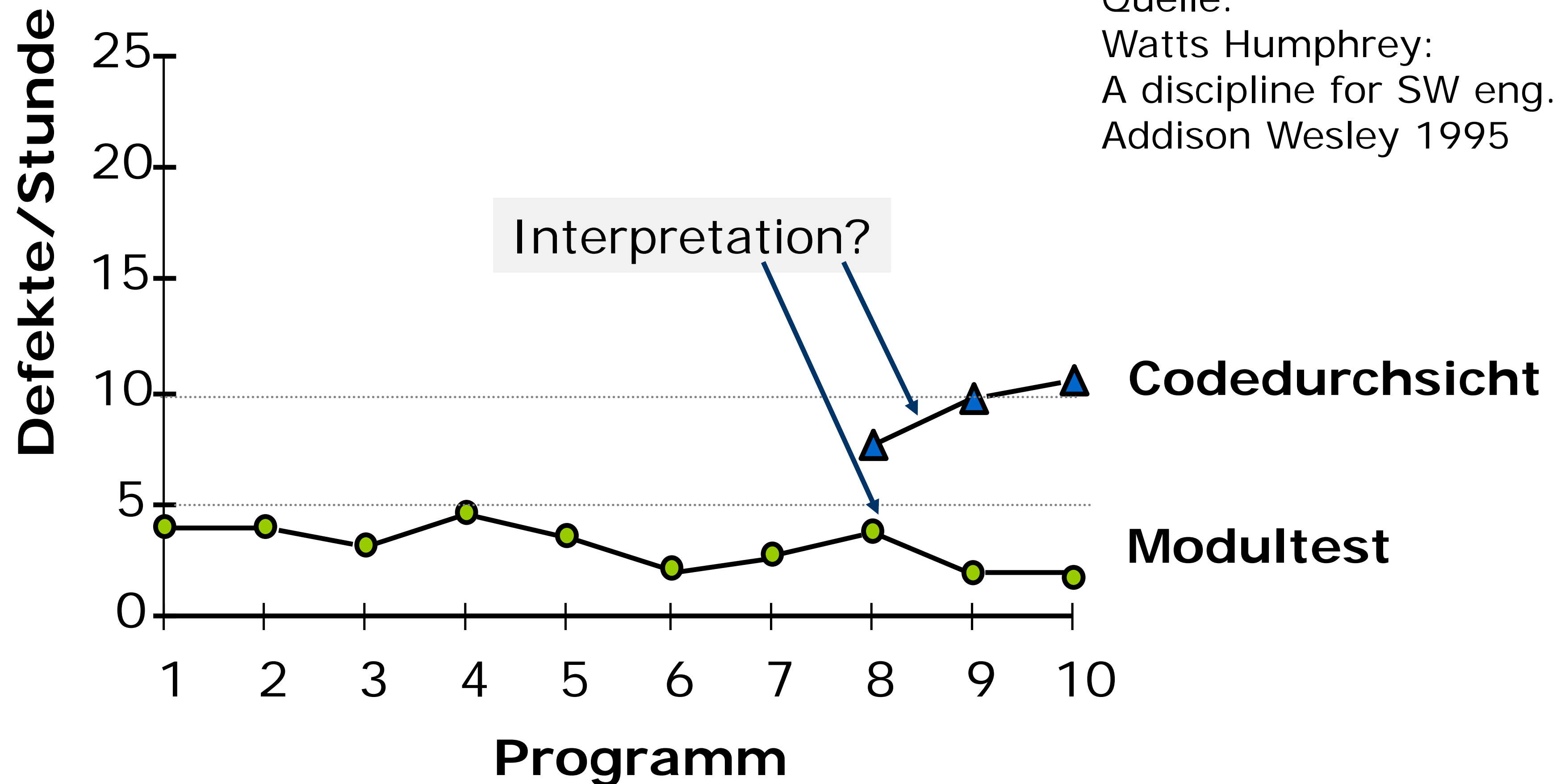


Quelle:
Kemerer, Paulk,
IEEE Trans. on SE
35(4), July 2009

Jeder Punkt repräsentiert Daten von >100 Durchsichten (von vielen Programmierern)

Effizienz
Durchsicht im Vergleich zu Test (Bsp.)

- Beispiel: 20 Studenten, je 10 kleine selbstgeschr. Progr., erst Codedurchsicht (ab Programm 8), dann Test



Übersicht

Analytische QS:

- Dynamische Verfahren (Test)
  - Defekttest
  - Benutzbarkeitstest
  - Lasttest
  - Akzeptanztest
- **Statische Verfahren**
  - Manuelle Verfahren
    - Durchsichten, Inspektionen
  - **Automatische Verfahren**
    - **Modellprüfung**
    - **Quelltextanalyse**

Konstruktive QS:

- Test- und Durchsichtsmgmt.
- Prozessmanagement
- Projektmanagement, Risikomanagement

Modellprüfung: Zustandsautomaten

- Für voll spezifizierte (Zustands)Automaten kann der gesamte Zustandsraum untersucht werden
  - Prüfung von Sicherheitseigenschaften, d.h. ob der Automat etwas tun kann, was er nicht tun soll
  - d.h. z.B. Beantwortung von Fragen der Art "Kann eine Abfolge A, B auftreten?"
    - Naives Bsp. für automatische Steuerung eines Containerkrans: "Kann ANHEBEN, LOSLASSEN geschehen?" (also ohne ABSENKEN dazwischen)

- Es gibt Modellprüfer-Software, die selbst große Zustandsräume (z.B. 10^{20} Zustände) effizient prüfen kann
  - Forschungsgebiet "model checking"
  - u.U. auch für unendliche Zustandsräume
  - ggf. mit stochastischen Überlegungen

Modellprüfung: Allg. Spezifikationsprüfung

- Auch andere Arten strukturierter Spezifikationen können auf bestimmte Eigenschaften automatisch geprüft werden:
  - Manche Arten von Inkonsistenz (innere Widersprüche)
  - Manche Arten von Unvollständigkeit
    - z.B. bei Fallunterscheidungen über Aufzählungstypen

```
switch(opera)
    case '+':

    case '-':

    case '*':

    case '/':

    default: p
```

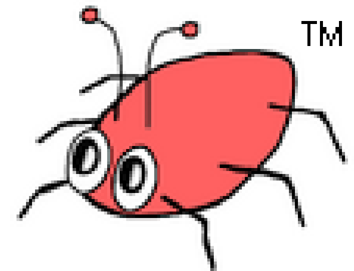
Wichtige Spezialfälle:

- Natürlichsprachliche Anforderungsbeschreibungen:
  - Vollständigkeit von Fallunterscheidungen oft überprüfbar
  - und <u>bei entsprechender Dokumentstruktur</u> automatisierbar

- Programmcode in typisierten Programmiersprachen:
  - Typprüfungen zur Übersetzungs- und Ladezeit sind eine Form der statischen Konsistenzprüfung

Quelltextanalyse

- Für manche Programmiersprachen gibt es Werkzeuge, die Quellcode auf bestimmte Schwächen abklopfen
- z.B. für Java
  - dubiose *catch*-Konstrukte, fehlende *finally*-Klauseln, verdächtige Verwendungen von *wait()/notify()* etc.
  - siehe z.B. "FindBugs" http://findbugs.sf.net/
- z.B. für C/C++
  - Verdächtige Verwendung von *malloc()/free()*
  - Verdächtige Verwendung von lock()/release() bei Nebenläufigkeit
- z.B. mit Bezug auf Sicherheitslücken etc.

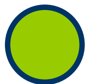


- Große Unterschiede in der Leistungsfähigkeit der Werkzeuge
  - offenes Forschungsgebiet "static analysis"
  - ein Problem sind z.B. die falschen Alarme

Übersicht

Analytische QS:

- Dynamische Verfahren (Test)
  - Defekttest
  - Testautomatisierung
  - Benutzbarkeitstest
  - Lasttest
  - Akzeptanztest
- Statische Verfahren
  - Manuelle Verfahren
    - Durchsichten, Inspektionen
  - Automatische Verfahren
    - Modellprüfung
    - Quelltextanalyse



Sie befinden sich hier

Konstruktive QS:

- Test- und Durchsichtsmgmt.
- Prozessmanagement
- Projektmanagement, Risikomanagement

Danke!

So einen falschen Font kann man in Software viel leichter mit einer Durchsicht entdecken als mit Tests.