TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik, Institut für Software- und Multimediatechnik, Lehrstuhl für Softwaretechnologie

11. Validation

Prof. Dr. U. Aßmann
Technische Universität Dresden
Institut für Software- und Multimediatechnik
Gruppe Softwaretechnologie

Version WS11-1.0, 05.11.11
http://st.inf.tu-dresden.de

1. **Defensive Programming**
   1. **Contracts**
   2. **Reviews**
2. **Tests**
   1. **Test Processes**
   2. **Regression Tests**
   3. **Junit**
   4. **FIT**
   5. **Acceptance Tests**
   6. **Model-Based Tests**
   7. **Eclipse-Based Test Tools**

Obligatory Reading

- Alternatively
  - Maciaszek Chapter 12 (is not enough)
  - Zuser Kapitel 5, 9, 12
  - Brügge Kap. 11
- Adrion, W. R., Branstad, M. A., and Cherniavsky, J. C. 1982. Validation, Verification, and Testing of Computer Software. *ACM Comput. Surv.* 14, 2 (Jun. 1982), 159-192. DOI= http://doi.acm.org/10.1145/356876.356879
- SWEBOK 2004
  - Kap. 5 Testen http://www.computer.org/portal/web/swebok/html/contentsch5#ch5
  - Kap. 11 Quality http://www.computer.org/portal/web/swebok/html/ch11

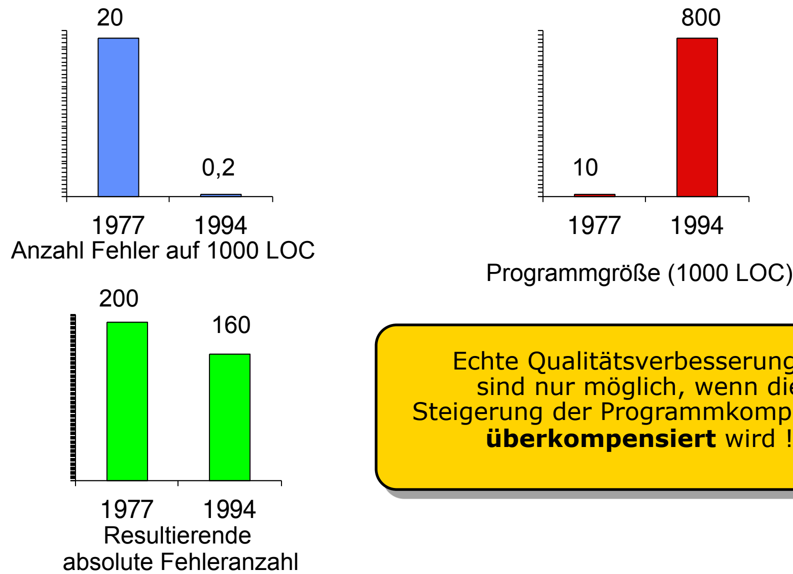
Recommended Reading

- Uwe Vigenschow. Objektorientiertes Testen und Testautomatisierung in der Praxis. Konzepte, Techniken und Verfahren. dpunkt-Verlag Heidelberg, 2005 Very good practical book on testing. Recommended!
- Axel Stollfuß und Christoph Gies. Raus aus dem Tal der Tränen. Hyades: Eclipse als Automated Software Quality Plattform. Eclipse Magazin 2/05. www.eclipse-magazin.de
- Bernhard Rumpe. Agile Modellierung mit UML. Springer, 2004. Chapter 5 Grundlagen des Testens, Chapter 6 Modellbasierte Tests
- Robert Binder. Testing Object-Oriented Systems. AWL.
- Frank Westphal. Unit Testing mit JUnit und FIT. Dpunkt 2005.
- Liggesmeyer P., Software-Qualität, Heidelberg: Spektrum-Verlag 2002, 523 S., ISBN 3-8274-1118-1
- http://www.opensourcetesting.org/
- http://www.junit.org
- http://de.wikipedia.org/wiki/Liste_von_Modultest-Software
- Kommerzielle Tools http://www.imbus.de/english/test-tool-list/
- Web Test Tools http://www.softwareqatest.com/qatweb1.html

Verification and Validation

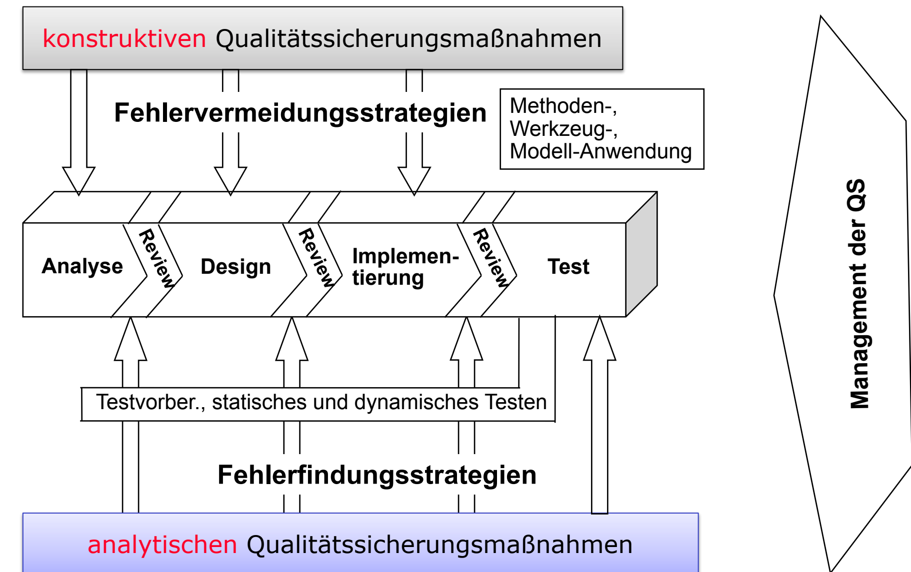
- **Verification** of correctness:
  - Proof that the implementation conforms to the specification (correctness proof)
  - Without specification no proof
  - "building the product right"
  - **Formal verification**: Mathematical proof
  - **Formal Method**: a software development method that enables formal verification
- **Validation**:
  - Plausibility checks about correct behavior (defensive programming, such as reviewing, tests, Code Coverage Analysis)
- **Test**:
  - Validation of runs of the system under test (SUT) under well-known conditions, with the goal to find bugs
- **Defensive Programming:**
  - Programming such that less errors occur

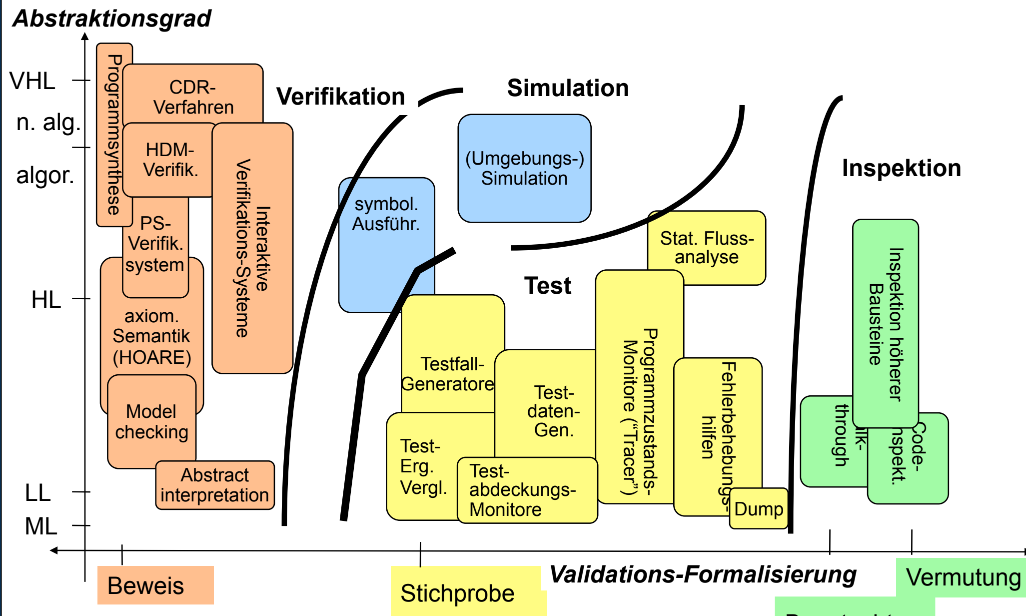
> Testing shows the presence of bugs, but never their absence (Dijkstra)



Echte Qualitätsverbesserungen sind nur möglich, wenn die Steigerung der Programmkomplexität **überkompensiert** wird !

Maßnahmen der Software-Qualitätssicherung werden differenziert nach:





Quelle: Hesse, W.: Methoden und Werkzeuge zur Software-Entwicklung: Einordnung und Überblick; Informatik-Fachberichte Bd. 43 Springer Verlag 1981

Constructive quality management:
Reduce errors by safer programming...

11.1 DEFENSIVE PROGRAMMING

11.1.1 Contract Checking with Layers Around Procedures

- Assertions in procedures can be used to specify tests (*contract checks*). Usually, these are layered around the core functionality of the procedure
  - Programming style of "analyse-and-stop": analyze the arguments, the surrounding of the arguments, and stop processing with exception if error occurs
  - Some programming languages, e.g., Eiffel, provide contract checks in the language
- Validating preconditions (assumptions):
  - Single parameter contract checks
    - Polymorphism layer: analysis of finer types
    - Null check, Exception value check
    - Range checks on integer and real values
    - State analysis: which state should we be in?
    - Condition analysis: invariants fulfilled?
    - Cross-relation of parameters
      - Object web checks (null checks in neighbors)
  - Invariant checks
  - Postcondition checks (guarantee)

Example: Contract Language in Eiffel

```
set_hour (h: INTEGER) is
      -- Set the hour from 'h'
    require
      valid_h: 0 <= h and h <= 23
    do
      hour := h
    ensure
      hour_set: hour = h
      minute_unchanged: minute = old minute
      second_unchanged: second = old second
    end
```

Precondition

Postcondition

Invariant Checks: Ex.: Triangle Invariant

- In a triangle, the sum of two sides must be larger than the third [Vigenschow]

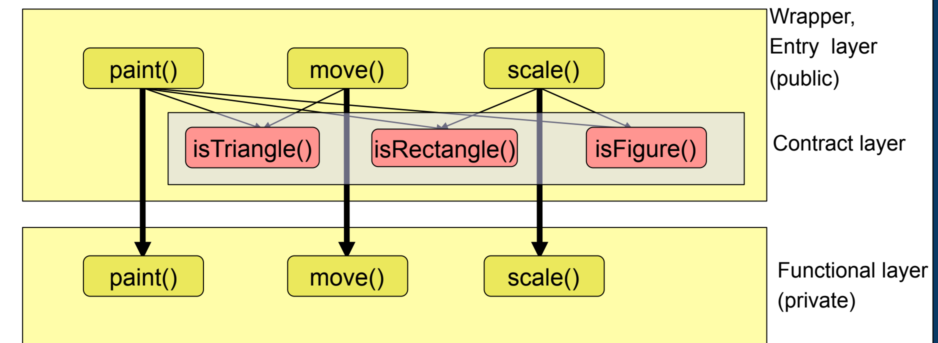
```
public boolean isTriangle(double s1, double s2, double s3){
   return ((a+b > c) && (a+c > b) && b+c > a));
}
```

- In a triangle-manipulating program, this is an invariant:

```
public void paintTriangle(Triangle t) {
   // preconditions
   assertTrue(t != null);
   assertTrue(t->s1 > 0 && t->s2 > 0 && t->s3 > 0);
   // invariant check
   assertTrue(isTriangle(t->s1, t->s2, t->s3));
   // now paint.
   ....
   // invariant check again
   assertTrue(isTriangle(t->s1, t->s2, t->s3));
   .. postconditions...
}
```


Implementation Pattern: Contract Layers

- Contract checks should be programmed in special check-procedures so that they can be reused as *contract layers*
- Advantage: entry layers can check contracts once, other internal layers need no longer check



- Are specified in OCL (object constraint language), referring to an UML class diagram:

```
context Person.paySalary(String name)
pre P1:  salary >= 0 &&
     exists Company company: company.enrolled.name == name
post P2: salary = salary@pre + 100 &&
     exists Company company:
            company.enrolled.name == name &&
            company.budget = company.budget@pre - 100
```

- More in Chapter "Validation with OCL"
- These contracts can be generated to contract code for methods (*contract instrumentation*)
  - Contract checker generation is task of *Model-driven testing (MDT)*
  - More in special lecture

Constructive QM with specific development processes

11.1.2 VALIDATION WITH INSPECTION AND REVIEWS

- Project managers should put up a bunch of *checklists* for the most important tasks of their project members and themselves
- [Pilots use checklists to start their airplanes]
- *Question lists* are a specific form of checklists to help during brainstorming and quality assurance
- http://www.rspa.com/spi/chklst.html#Anchor-49575

Example from Bazman http://bazman.tripod.com/

1. Does a failure of validation on every field cause a sensible user error message?
2. Is the user required to fix entries which have failed validation tests?
3. Have any fields got multiple validation rules and if so are all rules being applied?
4. If the user enters an invalid value and clicks on the OK button (i.e. does not TAB off the field) is the invalid entry identified and highlighted correctly with an error message.?
5. Is validation consistently applied at screen level unless specifically required at field level?
6. For all numeric fields check whether negative numbers can and should be able to be entered.
7. For all numeric fields check the minimum and maximum values and also some mid-range values allowable?
8. For all character/alphanumeric fields check the field to ensure that there is a character limit specified and that this limit is exactly correct for the specified database size?
9. ……….

Internal Reviews

- **Inspection**: A colleague reads the programmer's code
  - Inspection according to a predefined checklist
  - Programmer explains the code:
  - Check programming conventions, clarity of code, use of design patterns
  - Detect problems, but don't solve them
  - Often needs a moderator
- **Walkthrough**: going through the code with test data and simulate it by hand
  - A project leader should group her people to walkthrough or inspect in pairs
- **Review** from another group
  - More formal, With explicit review meeting, duration: 30-90 minutes
  - Protocol: Email or formal document
    - Participants, time, duration
    - Name, version, variant of code sources inspected
    - Review issue list
    - Actions determined
  - A review issue database is also nice (similar to a bug tracking or requirements management system)
  - Bug Tracking Database http://www.mantisbt.org/

Pair Programming: Permanent inspection

- Programming in pairs
  - A programmer
  - An inspector (reviewer)
- Change roles after some time
- Psychology: Not everybody likes to program in pairs
  - Egoless programming is desired

Pair programming is permanent inspection

External Reviews

- More formal:
  - An unrelated colleague from another department, or an unrelated team reviews the code
- Special review meeting
  - Prepare meeting by distributing all relevant documents
  - A review leader (moderator) guides through the meeting
- Formal protocol:
  - Review form is often standardized for a company
- Specifications can also be reviewed (requirements specs, design specs)
  - Find out inconsistencies with source code

Reviews contribute to quality

Audits

- Most formal kind of external review
- Professional auditors (quality assurance personel) from QA departments, or even external companies
  - Producer may be absent, auditing can be done from documents alone
- Audits take longer than reviews
  - Planning phase
  - Audits contain several reviews
- Audits can also check the
  - financial budgets: Auditors check how the money was spent (time sheets, travel, labor cost, ...)
  - planning
  - conformance to law (compliance)

A General Heuristic: Tight Feedback Loops

- Software processes are highly dynamic. It is hard to pre-plan them.
- Install process guidelines that lead to tight feedback loops:
  - Feedback early, often, frequently.
  - Better early light feedback than late thorough feedback
- For reviews, this means: review early, review often.

Analytic QM

11.2. VALIDATION WITH TESTS

The Problem

- Programmers program under time pressure (on-demand)
- Programmers program on-demand, because programming is hard
  - Domain problems
  - Special cases are forgotten
  - The effect of users is underestimated
  - [The demo effect]
  - A writer never finds his own bugs (Betriebsblindheit)
- Tests have destructive, negative nature
  - It is not easy to convince oneself to be negative!
  - Quality assurance people can ensure this, ensuring a reasonable software process

Wieviel sieht der Test vom SUT?

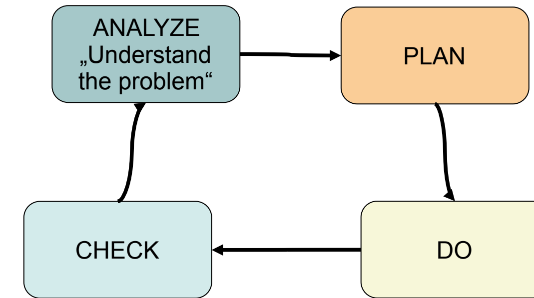
- **Statische Programmanalyse**, ohne dass das Programm ausgeführt wird
- **Dynamische Programmanalyse** durch Ausführung (oder Simulation) in einer geeigneten Testumgebung mit ausgesuchten Testdaten

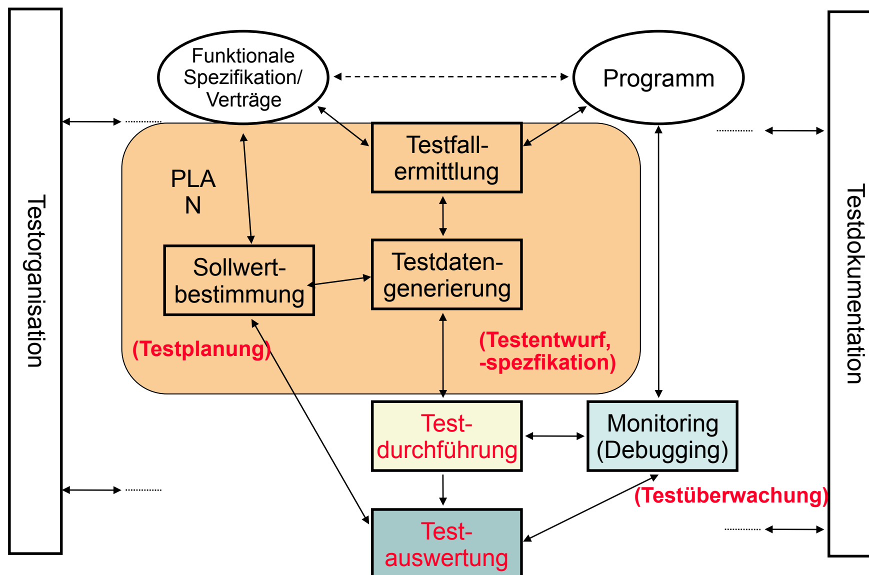
"Black-Box" Test	"Grey-Box" Test	"White-Box" Test
Funktionsabdeckung Äquivalenzklassenanalyse Grenzwertanalyse intuitive Testfallermittlung Zufallstest Fehlererwartung	"Back-to-Back"-Test Test spezieller Werte zustandsbasierter Test Zweigüberdeckung Entscheidungsüberd. Weg-Überdeckung	**Struktur**abdeckung Codeüberdeckung Anweisungsüberdeckg.

- **Strukturorientierter** Test (meist Greybox oder Whitebox)
  - Kontrollflussorientiert (Maß für die Überdeckung des Kontrollflusses)
    - Anweisungs-, Zweig-, Bedingungs- und Pfadüberdeckungstests
  - Datenflussorientiert (Maß für die Überdeckung des Datenflusses)
    - Defs-/Uses Kriterien, Required k-Tupels-Test, Datenkontext-Überdeckung
- **Funktionsorientierter** Test (Test gegen eine Spezifikation, meist Blackbox)
  - Äquivalenzklassenbildung, Zustandsbasierter Test, Ursache-Wirkung-Analyse z. B. mittels Ursache-Wirkungs-Diagramm, Transaktionsflussbasierter Test, Test auf Basis von Entscheidungstabellen
- **Diversifizierender** Test (Vergleich der Testergebnisse mehrerer Versionen)
  - Regressionstest, Back-To-Back-Test, Mutationen-Test
- Sonstige Mischformen
  - Bereichstest bzw. Domain Testing (Verallgemeinerung der Äquivalenzklassenbildung), Error guessing, Grenzwertanalyse, Zusicherungstechniken

http://de.wikipedia.org/wiki/Softwaretest [Liggesmeyer]

- George Polya. How to Solve It (1945).



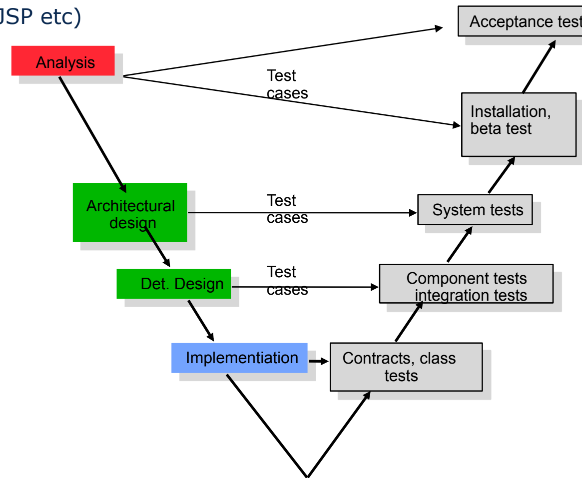


Quelle: Müllerburg, M. u.a.(Hrsg.): Test, Analyse und Verifikation von Software; GMD-Bericht Nr. 260, R. Oldenbourg Verlag 1996 S.115

2.2.1 TEST PROCESSES

Standard Test Process as Right Wing of the V Model

- Tests should be done *bottom-up*
  - Defensive programming (contracts)
  - Then unit tests (class tests)
  - Then component tests (EJB, JSP etc)
  - Then the system
  - Then the beta test
  - Then acceptance test



The Cleanroom Method

- The Cleanroom method divides the project members into programmers and testers.
- Developer must deliver a result almost without bugs
  - Testing forbidden!
- Incremental process
- Experience with Cleanroom Method
  - Selby tested CleanRoom with 15 Student Teams, 10 Cleanroom/5 non-Cleanroom
  - Cleanroom-Teams produce simpler code with more comments
  - 81% want to use it again
  - All Cleanroom teams manage milestones, 3 of 5 non-Cleanroom teams not.
  - But: programmers do not have the satisfaction to run their code themselves
- Only the problems with formal specification

CleanRoom in the NASA

- In 1987, the NASA developed a 40KLOC control program for satellites with Cleanroom
- Distribution of project time:
  - 4% Training staff in Cleanroom,
  - 33% design
  - 18% Implementation (45% writing, 55% reviewing),
  - 27% Testing,
  - 22% Other things (e.g., meetings)
- Increase in productivity 69%.
- Reduction of error rate 45%.
- Resource reduction 60-80%.
- Developers, prohibited to test their code, read intensively. This catches many bugs (~30 bugs for 1KLOC).

Microsoft's Software Process "Synchronize and Stabilize"

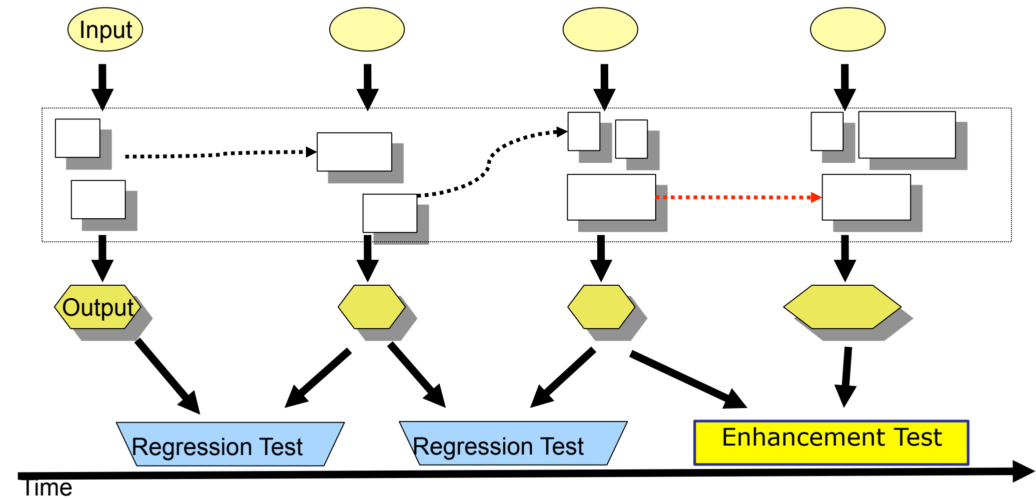
- .. is a CleanRoom process:
- Microsoft builds software until 12:00 (synchronization)
- In the afternoon, test suites are run by the test teams, i.e., separation of programmer and tester
- Programmers get feedback the next day
- [IBM tests in China]

How to be sure that a change did not introduce errors...
Diversifying tests

11.2.2 REGRESSION TESTS

Regression Tests as Diffs on Outputs of Subsequent Versions

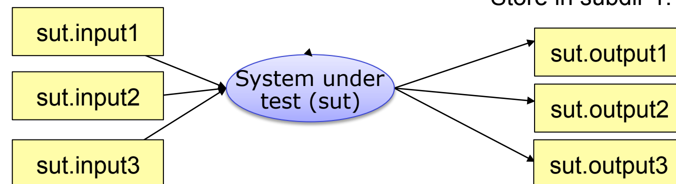
- Regression tests are operators that check semantic identity between versions that have similar input/output relation
- Enhancement tests test enhanced functionality

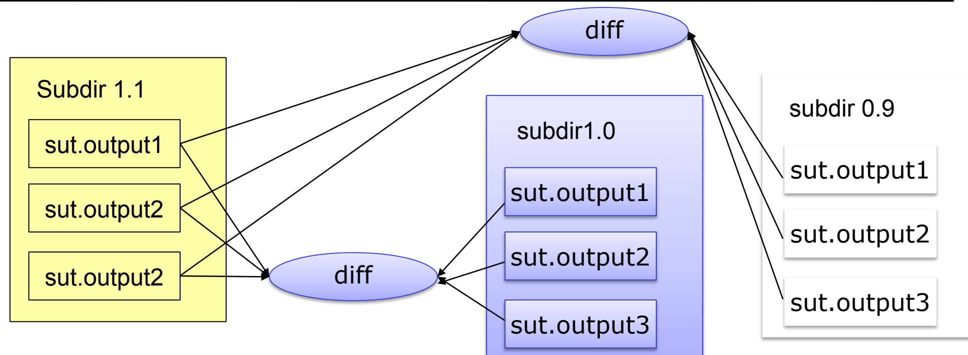


A Poor Man's Regression Testing Environment

- The UNIX tool diff is able to textually compare files and directories (recursively)

Store in subdir 1.1





subdir 0.9
sut.output1
sut.output2
sut.output3

Diff Listings for Regression Tests

- Diff shows lines that have been removed from first file (<) "went out" and added to the second (>) "came in"

```
diff file~1.1 file~1.0

< if (threshold < 0.9) stopPowerPlant();
> if (threshold > 0.9) stopPowerPlant();

-- compares entire directory to subdirectory
   1.0
diff -rq . 1.0
./subdir/f.c:
< if (threshold < 0.9) stopPowerPlant();
> if (threshold > 0.9) stopPowerPlant();
```

- Diff invocations are wired together for test suites with shell scripts or makefiles
- On windows, use cygwin shell (www.cygwin.org)

Variants of Regression Tests

- [Binder] distinguishes 5 *coverage patterns* for regression tests:
- All (exhaustive): best
- Risky use cases
- Re-test profile: profile code and re execute tests on most executed code
- Changed code (code that changed between versions)
- Changed code and all dependent code

Importance of Regression tests

- Regression tests are *the most important mechanism* to ensure quality if a product appears in subsequent versions
  - Without regression test, no quality
- Companies sell test data suites for regression tests
  - Validation suites for compilers (e.g., Ada or XML)
  - Validation suites for databases
  - Test data generators that generate test data suites from grammars
  - Test case generators
- The more elaborated your regression test method is, the better your product will be

GUI Regression Test with Capture/Replay Tools

- A **capture tool** allows for recording user actions at a GUI
  - Recording in macros or scripts
- A **replay tool** reads the scripts and generates events that are fed into the system
  - The replay tool can be started in batch, i.e., can be integrated into a regression test suite
  - Hence, the GUI can be regression tested
- Capture/replay tools can record the most important workflows how systems are used
  - Opening documents, closing, saving
  - Exception situations
  - Even big office suites seem not to be tested with capture/replay tools
- Examples:
  - Mercury Interactive WinRunner www.mercuryinteractive.de
  - Rational Robot www-306.ibm.com/software/rational
  - Abbot - http://abbot.sourceforge.net/doc/overview.shtml
  - Jellytools is a JUnit-derivative for test of Swing-GUI
  - web2test from Leipzig http://www.saxxess.com/content/14615.htm

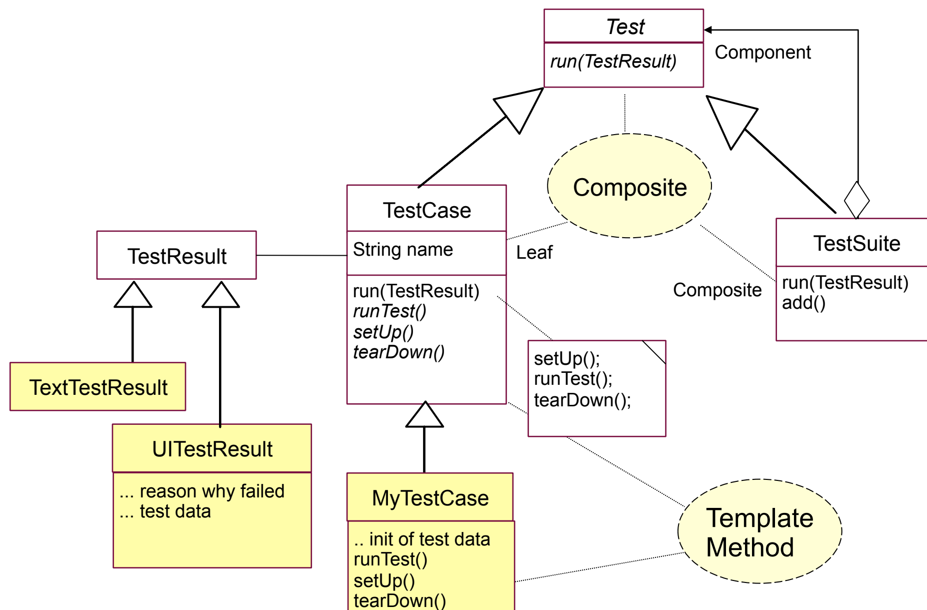
(Repetition from Softwaretechnologie)

11.2.3 THE JUNIT REGRESSION-TEST FRAMEWORK

The JUnit Regression-Test Framework

- JUnit www.JUnit.org is a simple framework for regression tests of classes and components
  - Run test cases and test suites
  - Plugins for IDE available (Eclipse, JBuilder, ..)
  - Available for several languages
    - Nunit for .NET
    - CppUnit for C++
- Test result classification:
  - Failure (Zusicherung wird zur Laufzeit verletzt)
  - Error (Unvorhergesehenes Ereignis, z.B. Absturz)
  - Ok
- JUnit supports permanent regression testing during development:
  - All test cases are collected and repeated over and over again

JUnit Framework



Example: Test of Dates

```
// A class for standard representation of dates.
public class Date {
  public int day; public int month; public int
year;
  public Date(String date) {
    day = parseDay(date);
    month = parseMonth(date);
    year = parseYear(date);
  }
  public int equals(Date d) {
    return day == d.day &&
           year == d.year &&
           month == d.month;
  }
}
```

New TestCase

- TestCases are methods, start with prefix "test"
- Test cases contain test data in a *fixture*
- Problem: test data is *integrated* with test case

```
public class DateTestCase extends TestCase {
    Date d1;
    Date d2;
    Date d3;
    int length = 42;
  protected int setUp() {
    d1 = new Date(„1. Januar 2006");
    d2 = new Date(„01/01/2006");
    d3 = new Date(„January 1st, 2006");
  }
  public int testDate1() {
    assert(d1.equals(d2));
    assert(d2.equals(d3));
    assert(d3.equals(d1));
    .... more to say here ....
  }
  public int testDate2() {
    .... more to say here ....
  }
}
```

Example: Test Case in Nunit (.Net) and JUnit 4.0

- Tags are used to indicate test case classes, set up, tear down methods
  - The tags are metadata that convey additional semantics for the items

Use of Test Case Classes

- A test case is created by calling the constructor
- The constructor finds the method of the given method name and prepares it for call (reflection)
- *The run()* method starts the test case with the fixture and returns a test result
- In case of a failure, an exception is raised

```
public class TestApplication {
   ...
   TestCase tc = new DateTestCase(„testDate1");
   TestResult tr = tc.run();
}
```

Test Suites

- A *test suite* is a collection of test cases (pattern Composite)

```
public class TestApplication {
   ...
   TestCase tc = new DateTestCase(„testDate1");
   TestCase tc2 = new DateTestCase(„testDate2");
   TestSuite suite = new TestSuite();
   suite.addTest(tc);
   suite.addTest(tc2);
   TestResult tr = suite.run();
   // Nested test suites
   TestSuite subsuite = new TestSuite();
   ... fill subsuite ...
   suite.addTest(subsuite);
   TestResult tr = suite.run();
}
```

TestRunner GUI

- The classes JUnit.awtui.TestRunner, JUnit.swingui.TestRunner are simple GUIs to present test results
- Test suites can be given the class object of a test case, and it finds the test case methods on its own (reflection)

```
public class TestApplication {
  public static Test doSuite() {
    // Abbreviation to create all TestCase objects
    // in a suite
    TestSuite suite = new TestSuite(DateTestCase.class);
  }
  // Starte the GUI with the doSuite suite
  public static main () {
    JUnit.awtui.TestRunner.run(doSuite());
  }
}
```


11.2.4 FIT Testing Framework

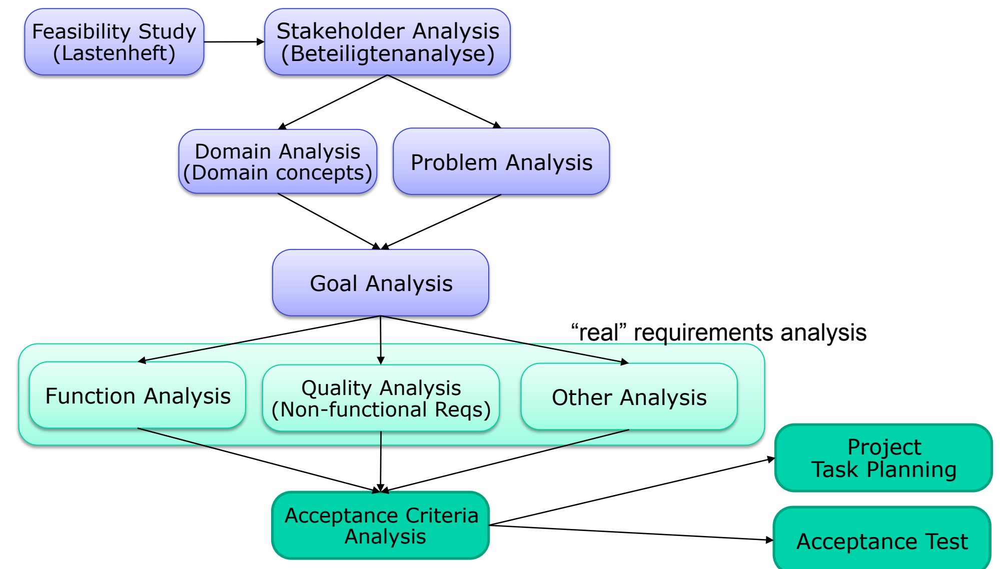
- FIT is an acceptance and regression testing framework
- A software testing tool designed for customers with limited IT knowledge
- Test cases can be specified in tables
  - Wiki
  - Excel
  - HTML
  - DOC
  - ....
- Fit test tables are easy to be read and written by customer

FIT Testing Framework

- Story-based tests
  - Stored in test tables
- Parse input and invoke methods throught reflection
- FitRunner to start the test (Command line)
- Can be combined with GUI robots like Abbot

fit.ActionFixture		
start	calculator2003.CalculatorGuiFixture	
enter	delay	2000
check	value	0
press	five	
press	three	
press	plus	
press	five	
press	equals	
check	value	58 *expected* / 48 *actual*
press	minus	
press	two	
press	equals	
check	value	56 *expected* / 46 *actual*

11.2.5 Acceptance Tests

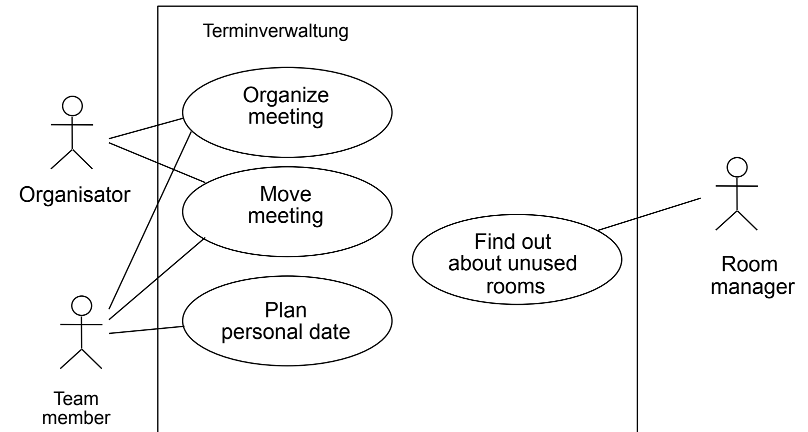


Acceptance Test

- *Acceptance test cases* are part of the SRS
  - Are checked by the customer for fulfillment of the contract
  - Without passing, no money!
- Acceptance tests are *system tests*
  - Run after system deployment
  - Test entire system under load
  - Test also non-functional qualities
- After every evolution step, all acceptance test cases have to be repeated
- Regression test:
  - Should-Be-outputs are compared with actual outputs
  - Consists of a set of test cases (a test suite)

Deriving Test Cases from Functional Specifications

- Most often, acceptance tests are derived from use cases, function trees, or business models
- Every use case yields at least one acceptance test case
  - For every test case, a test driver is written

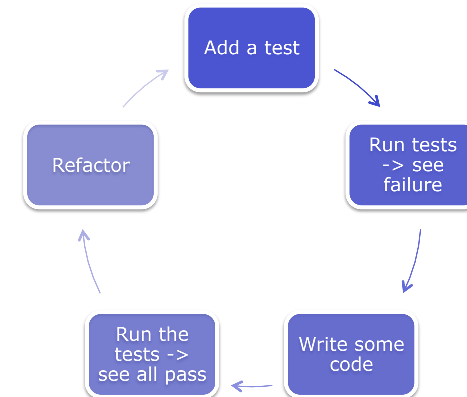


Tests and Tutorials

- Some test cases can be written in a user-friendly style (*tutorial test cases).*
- If they are enriched with explanations, *tutorial threads* result
- Hence, sort out some test cases for tutorial test cases
- [Java documentation]

Test-First Development (Test-Driven Development)

- Iterate:
  - First, fix the interface of a method
  - Second, write a test case against the interface
  - Third, program method.
  - Fourth, Run test case. If test case works, add it to the current test suite



- Advantages
  - Permanent regression test (test data integrated)
  - Stable extension of the code: no big bang test, collection of test cases always running
  - Functionality so far can always be demonstrated
- TDD is like *automating the reviewer:* the test case plays the role of the criticizing colleague!

- Given a function f under test with y = f(x).
- x and y can be values or object graphs [Rumpe04].

Possible patterns for test cases:

- Equality tests x == y
- Difference predicate: Predicate(x, f(x))
- Feature tests: Predicate(f(x))
- Equivalence class test: f(x) === e from equivalence class
- Abstraction test: Abstraction(f(x)) = Abstraction(z) with a fix z
- Identity test: $f^{-1}(f(x)) = x$
- Oracle function: f(x) = oracle(x)

- Instead of fixing the test data in a fixture, the test data can be separated from the application.
- Advantages:
  - Test data can be specified symbolically, instead of using constructor expressions
  - Test data can be persistent in files or in databases
  - Test data can be shared with other products in the product line
- Disadvantages:
  - Database must be maintained together with code (versioning)
- Example:
  - Big compiler test suites (e.g., Ada)
  - Database test suites

- **Stub**: Empty implementations behind the interface
- **Dummy**: Simple, restricted simulation of the interface functionality
- **Mock:** Dummy that also checks the protocol of the class-under-test (to mock, „etwas vortäuschen“)
  - Often statecharts (Steuerungsmaschinen)

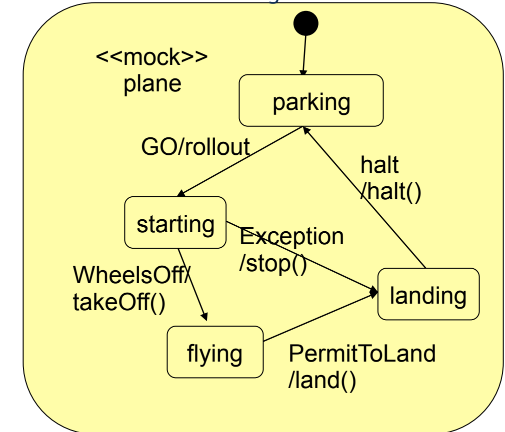
http://de.wikipedia.org/wiki/Mock-Objekt

Modality of Classes

- **Non-modal class (stateless)**
  - no call protocol over method set
- **Uni-modal class (stateful)**
  - call protocol exists (described by a Chomsky language: state chart, context free language, context-sensitive, Turing-complete)
- **Quasi-modal class (stateful with restrictions)**
  - The class has additional semantic restrictions (e.g., limited buffer size)
- **Modal class**
  - business rules describe the call protocol. For instance, some accounts cannot be overcharged, others until a certain credit limit

Mock Object for (Uni-)Modal Classes

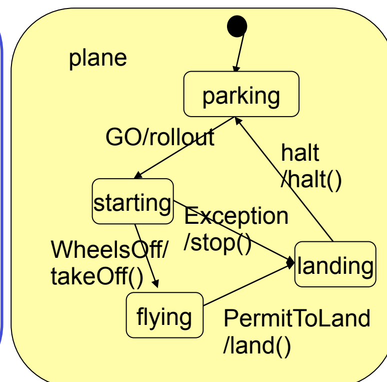
- A **mock object** simulates a modal class-under-test, implementing the life-cycle protocol
- If the CUT is a unimodal class, with an underlying state chart, the test driver should test all paths in the state chart
  - The mock must check whether all state transitions are done right
  - Test case tests:
    - Path 1: parking->starting->flying->landing->parking
    - Path 2: parking->starting->landing->parking (emergency path)
  - Driver checks that after each method that is called for a transition, the right state is reached
  - Mock object implements state transitions



Mock Object

```
Public class PlaneMock extends MockObject {
  int state;
  public enum { parking, starting, flying, landing };
  public PlaneMock() {
    state = parking;
  }
```

```
public class PlaneTestCase extends TestCase {
  pMock = new PlaneMock();
  public void setUp() { .. }
  public void tearDown() { .. }
  public void testPath1(){
    pMock.rollout();
    assertEquals(pMock.starting,pMock.getState());
    pMock.takeOff();
    assertEquals(pMock.flying,pMock.getState());
    pMock.land();
    assertEquals(pMock.landing,pMock.getState());
    pMock.halt();
    assertEquals(pMock.parking,pMock.getState());
   public void testPath2() {..}
}
```



Framework EasyMock

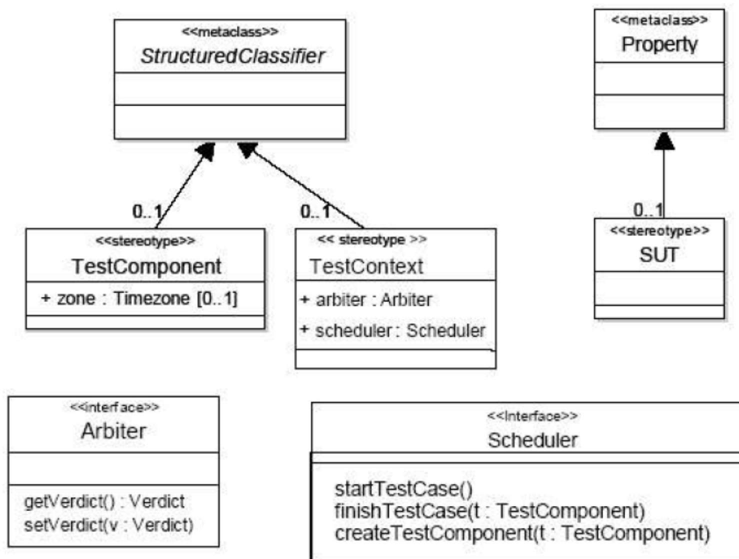
- http://de.wikipedia.org/wiki/Easymock
- http://www.easymock.org
- EasyMock automates the creation of mock objects by generating mock objects as proxy objects
- An **easymock** object is a proxy to an empty real object, with two modes:
  - Recording mode: In this mode, the easymock learns how it should be used
  - Replay mode: In this mode, it tests whether it has been used correctly
- A **strict easymock** learns also the order of calls

11.2.6 MODEL-BASED TESTING

The UML Testing Profile UTP

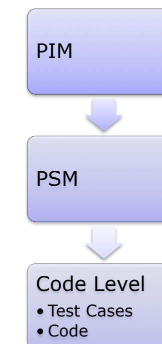
- P. Baker et. Al. The UML 2.0 Testing Profile (2008)
  - http://en.scientificcommons.org/43308184
- The concepts of JUnit can be modeled in a class diagram
- The OMG has standardized a UML profile (an extension of the UML metamodel) providing the concepts of JUnit
  - UTP is a collection of stereotypes that can mark up class diagrams
  - <<SUT>>, <<TextComponent>>, <<TestContext>>
  - and of tagged values (Tagged values are the same concept as C# attributes or Java Xdoclets)
  - <<startTestCase>>, <<finishTestCase>>, <<createTestComponent>>
- UTP tags are *translated to* target programming languages
  - Tests are platform-independent
- Test cases and suites can be specified while modelling
  - Code of test cases can be generated from a CASE tool

UTP Profile (Metamodel)



UTP

- Realized testing profile as:
  - UML 2.0 meta model
  - MOF (Meta Object Facility) meta model
- Mapping to existing test infrastructures
  - Test Control Notation TTCN-3
  - JUnit
- Usage in MDA Stack



11.2.7 ECLIPSE-BASED TEST PLATFORMS

Eclipse Supports Many Test Platform Plugins

- Hyades www.eclipse.org/test-and-performance
  - Test Capture-Replay framework for web and other applications
    - Http requests can be recorded, generated into JUnit test case classes, afterwards replayed
    - Uses UTP to specify test cases
  - A Remove-Access-Controller mediates between Eclipse and the SUT
  - Test data can be stored in data pools
  - Log-file analysis based on the Common-Base-Event format of IBM
- Solex http proxy logger www.sf.net/projects/solex
- Scapa stress test www.scapatech.com
- HttpUnit, htmlUnit extensions of JUnit for test of web applications
  - httpunit.sf.net
  - htmlunit.sf.net

TPTP

- TPTP Platform Project http://www.eclipse.org/tptp/
  - Covers the common infrastructure in the areas of user interface, EMF based data models, data collection and communications control, remote execution environments and extension points
- TPTP Monitoring Tools Project
  - Collects, analyzes, aggregates and visualizes data that can be captured in the log and statistical models
- TPTP Testing Tools Project
  - Provides specializations of the platform for testing and extensible tools for specific testing environments
  - 3 test environments: JUnit, manual and URL testing
- TPTP Tracing and Profiling Tools Project
  - Extends the platform with specific data collection for Java and distributed applications that populate the common trace mode, also viewers and analysis services

TPTP Profiling Tool

Profiling and Logging - ProductCatalog.java - Eclipse SDK

File Edit Source Refactor Navigate Search Project Run Window Help

Working Sets | Profiling and L... | Java

Console | Execution Statistics | Method Invocation Details

Execution Statistics - com.sample.product.Product at popescu011 [PID: 396] (Filter: No filter)

Method	Class	Package	Base Time (sec...	<Cumulative Ti...	Calls
main(java.lang.String[]) void	Product	com.sample.p...	0.07%	44.68%	0.02%
readData(java.lang.String) void	ProductCatalog	com.sample.p...	0.05%	41.08%	0.02%
parseContent(java.io.File, javax.	ProductCatalog	com.sample.p...	0.09%	21.30%	0.39%
createParser() javax.xml.parsers	ProductCatalog	com.sample.p...	0.02%	19.34%	0.39%
parse(java.io.InputStream, org.x	SAXParser	javax.xml.pa...	0.07%	19.23%	0.39%
parse(org.xml.sax.InputSource,	SAXParser	javax.xml.pa...	0.15%	19.11%	0.39%
parse(org.xml.sax.InputSource)	AbstractSAXP...	org.apache.x...	13.02%	18.18%	0.39%
newInstance() javax.xml.parsers	SAXParserFa...	javax.xml.pa...	0.04%	12.64%	0.02%
find(java.lang.String, java.lang.S	FactoryFinder	javax.xml.pa...	0.04%	12.53%	0.02%
findJarServiceProvider(java.lang.	FactoryFinder	javax.xml.pa...	12.35%	12.35%	0.02%
newSAXParser() javax.xml.parse	SAXParserFa...	org.apache.x...	0.07%	6.64%	0.02%
SAXParserImpl(javax.xml.parsers	SAXParserImpl	org.apache.x...	0.32%	6.57%	0.02%
SAXParser()	SAXParser	org.apache.x...	6.22%	6.22%	0.02%
startElement(java.lang.String, ja	ProductCatalog	com.sample.p...	1.28%	5.17%	0.78%
println(java.lang.String) void	ConsolePrintS...	com.ibm.jvm.io	0.01%	3.18%	0.02%
println(java.lang.String) void	PrintStream	java.io	0.00%	3.00%	0.02%
newLine() void	PrintStream	java.io	2.89%	2.89%	0.02%
append(java.lang.String) java.lar	StringBuffer	java.lang	1.40%	2.08%	12.23%
FileInputStream(java.io.File)	FileInputStream	java.io	0.13%	1.90%	0.39%
open(java.lang.String) void	FileInputStream	java.io	1.73%	1.73%	0.39%
StringBuffer(java.lang.String)	StringBuffer	java.lang	0.41%	0.95%	2.27%

21M of 40M

- Separation of reviewer and producer is important
- Defensive programming is good
- Test-first development produces *stable* products
- Without regression tests, no quality
- Mock classes simulate classes-under-test, realizing their life-cycle protocol
- Test tools, e.g., on the Eclipse platform, help to automate testing of applications, also web applications