11. Validation

Lecturer: Dr. Sebastian Götz

Prof. Dr. U. Aßmann
Technische Universität Dresden
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie
http://st.inf.tu-dresden.de/teaching/swt2

WS 2017, 06.11.2017

1. Defensive Programming
   1. Contracts
2. Reviews
3. Tests
   1. Test Processes
   2. Regression Tests
   3. FIT
   4. Stubs and Mocks
   5. Acceptance Tests

Obligatory Reading

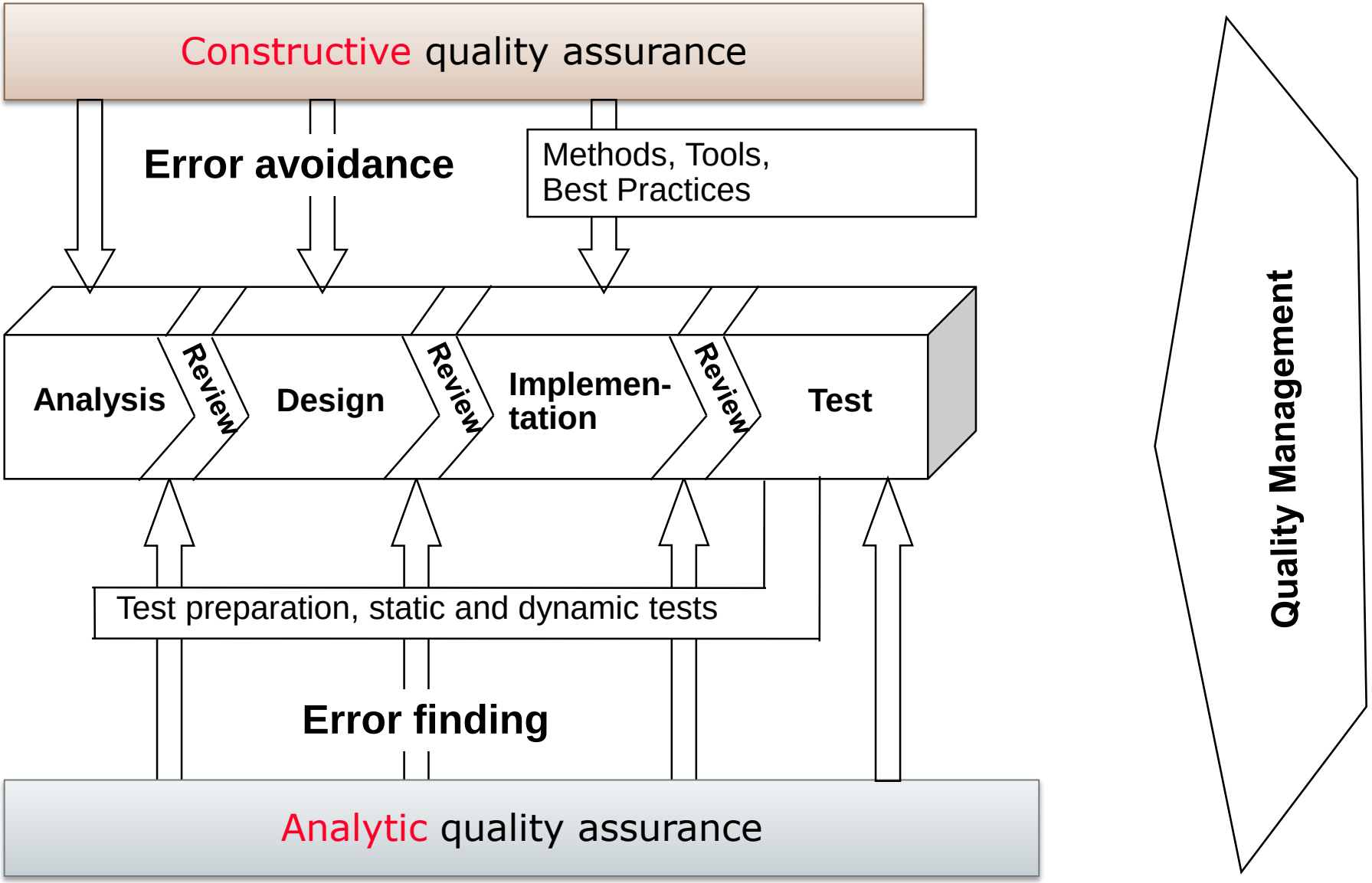
- Balzert Kap. 1 (LE 2), Kap 2 (LE 4)
- Maciaszek Chap 6-8
- ZOPP from GTZ www.gtz.de:
  - Ziel-orientierte Projektplanung. GTZ (Gesellschaft für technische Zusammenarbeit). GTZ is a German society for development. ZOPP is a general-purpose project planning and requirements analysis method. Google for it.....
  - http://portals.wi.wur.nl/files/docs/ppme/ZOPP_project_planning.pdf
  - ZOPP is part of Project Cycle Management (PCM), a more general methodology for project management
    - http://baobab-ct.org/learning/pcm.html
  - http://en.wikipedia.org/wiki/Logical_framework_approach
  - The ZOPP-88 version, on which this material is basedhttp://pmkb.com.br/wp-content/uploads/2013/08/GTZ-ZOPP.pdf

Verification and Validation

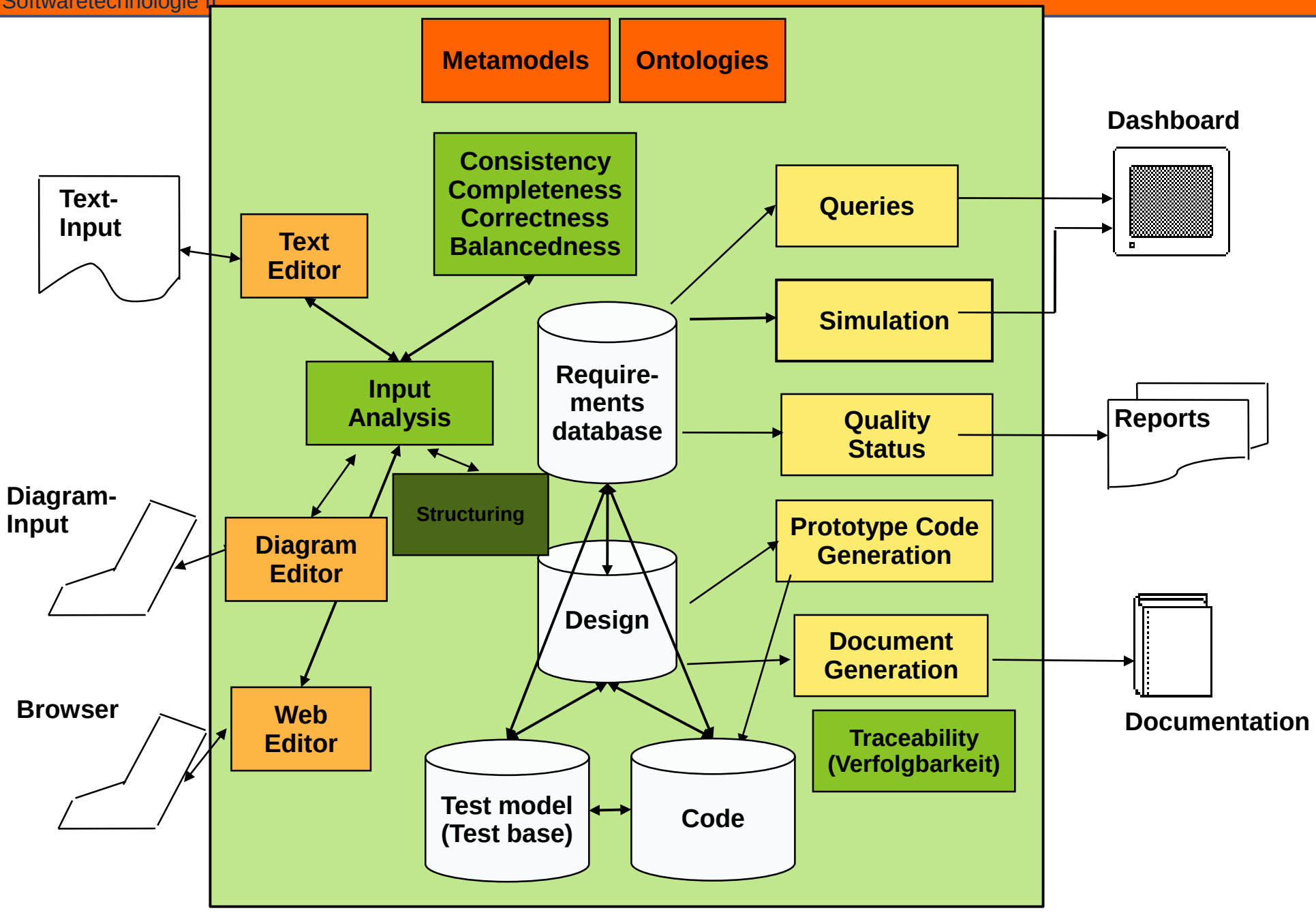
- **Verification** of correctness:
  - Proof that the implementation conforms to the specification (correctness proof)
  - Without specification no proof
  - "building the product right"
  - **Formal verification**: Mathematical proof
  - **Formal Method**: a software development method that enables 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)

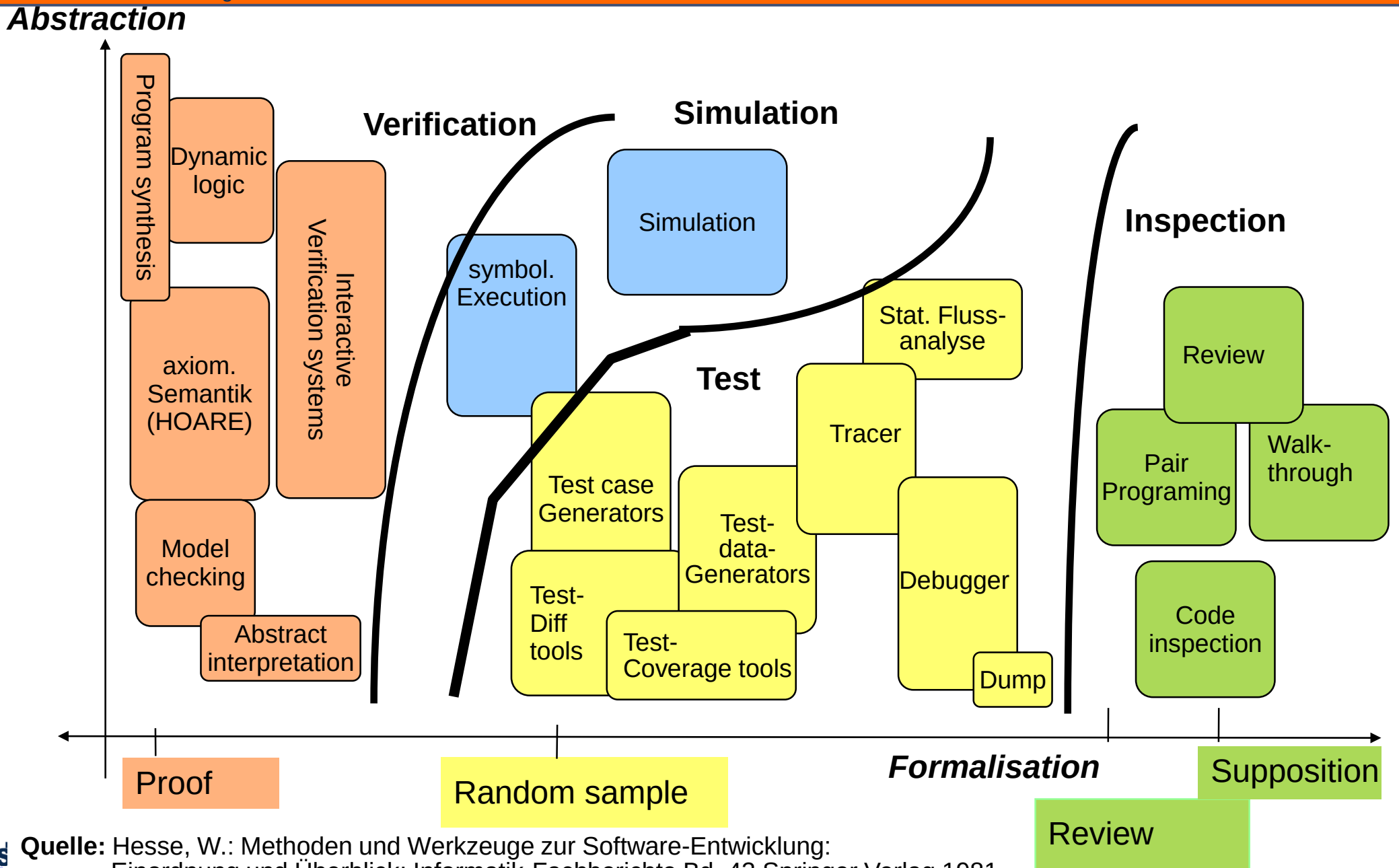
Quality Management (QM, Quality Assurance, QA)



QM with Traceability between Tests, Requirements, Design, and Code



Verification and Validation Techniques



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
- **Precondition checks (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**
  - Form of data structures, i.e., different forms of graphs
- **Postcondition checks (guarantee)**

Example: Contract Language in Eiffel

- http://www.ecma-international.org/publications/standards/Ecma-367.htm
- http://sourceforge.net/projects/tecomp/

```
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...
}
```

Invariant Checks: Ex.: Triangle Invariant

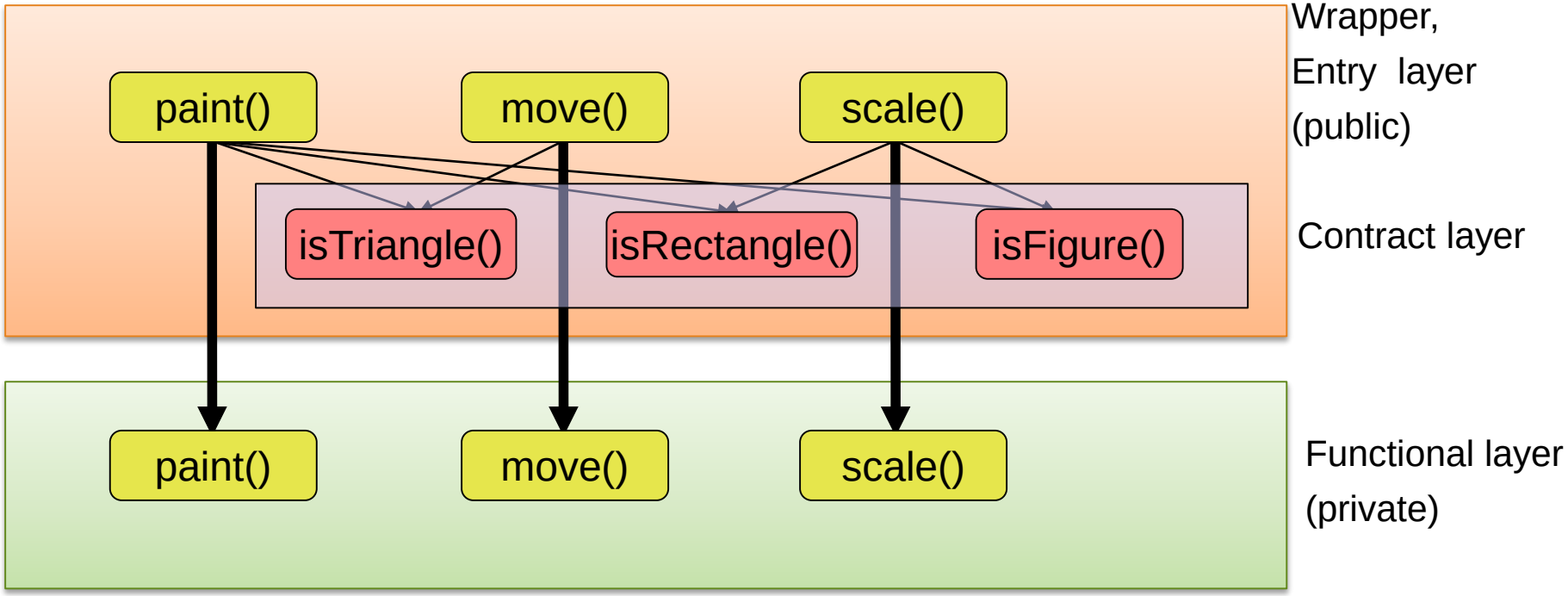
- Contract check can be encapsulated into procedures in a **contract layer**

```
public boolean preconditionCheck(Triangle t){
   assertTrue(t != null);
   assertTrue(t->s1 > 0 && t->s2 > 0 && t->s3 > 0);
}
public void invariantCheck(Triangle t) {
   assertTrue(isTriangle(t->s1, t->s2, t->s3));
}
public boolean postconditionCheck(Triangle t){
   assertTrue(t->s1 > t->LIMITLENGTH);
}
```

```
public void scale(Triangle t, double scaling) {
   preconditionCheck(t);
   // now scale.
   t->s1*=scaling; t->s2*=scaling; t->s3*=scaling;
   invariantCheck(t);
   postconditionCheck(t);
}
```

Implementation Pattern: Contract Wrapper Layers

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



Inner Functional and Outer Wrapper Layer


```
// Outer wrapper layer with contract checks (slow)
public void scale(Triangle t, double scaling) {
   preconditionCheck(t);
   // now scale.
   scalePrivate(t);
   invariantCheck(t);
   postconditionCheck(t);
}
```

```
// Inner (functional) layer without contract checks (fast)
public void scalePrivate(Triangle t, double scaling) {
   // Precondition is assumed to be true
   // now scale.
   t->s1*=scaling; t->s2*=scaling; t->s3*=scaling;
   // Invariant is assumed to be true
   // Precondition is assumed to be true
}
```

Model-Based Contracts

- A **model-based contract** is usually 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

Checklists

- 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

- Examples:
  - http://www.rspa.com/spi/chklst.html#Anchor-49575

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 a colleague; use test data to simulate it by hand
  - A project leader should group her people to walkthrough or inspect in pairs
- **Review** from another group
  - More formal
  - Review preparation: send all documents (code, requirement specification, design specification, test cases, documentation) to the reviewers
  - Explicit review meeting, duration: 30-90 minutes
  - Protocol: Email or formal document to the reviewed group and the management
    - Participants, time, duration
    - Name, version, variant of code sources inspected
    - **Review issue list**
    - Actions determined
  - Review followup – working on the issue list
  - 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
- Review preparation phase
- 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
  - **Review issue list**
- Review follow-up

Reviews contribute to quality

Audits

- Most formal kind of external review
- Professional auditors (quality assurance personnel) 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

Static Test (Static Analysis)

- **Static program analysis** without executing the program
- Required: a tool for static analysis

Test Method (Analyse)	Purpose
Symbolic Execution	Execution with symbolic values
Traceability	Tracing code to requirments by code-requirement-mappings
Abstract Interpretation	Execution with equivalence classes
Data-flow analysis (Value flow analysis)	Abstract Interpretation with focus on flow of data (values)
Control-flow analysis	Abstract Interpretation with focus on flow of control conditions
Metrics	Counting methods for static features of programs

Dynamic Test

- **Dynamic program analysis** by Test

Test Method (Analyse)	Purpose
Simulation	Execution with concrete values on a simulator (software in the loop, SIL)
Functional test	Black-box test against interfaces
Structure-oriented test	Coverage analysis of control flow paths Coverage analysis of data flow
Other tests	Fault injection, regression test

[Liggesmeyer]

http://de.wikipedia.org/wiki/Softwaretest

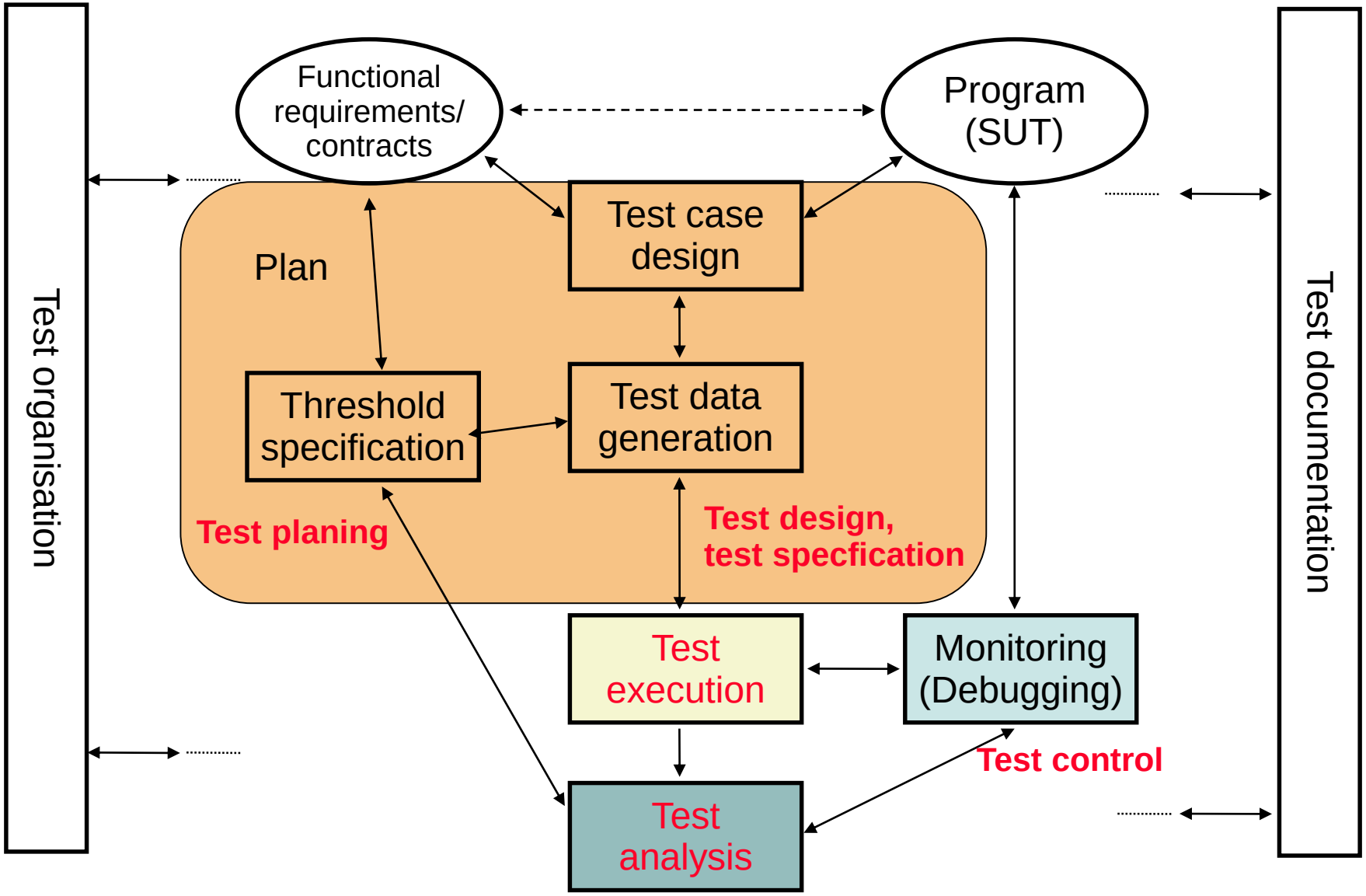
The Problems with Testing

- 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

11.2.2.1 TEST PROCESSES

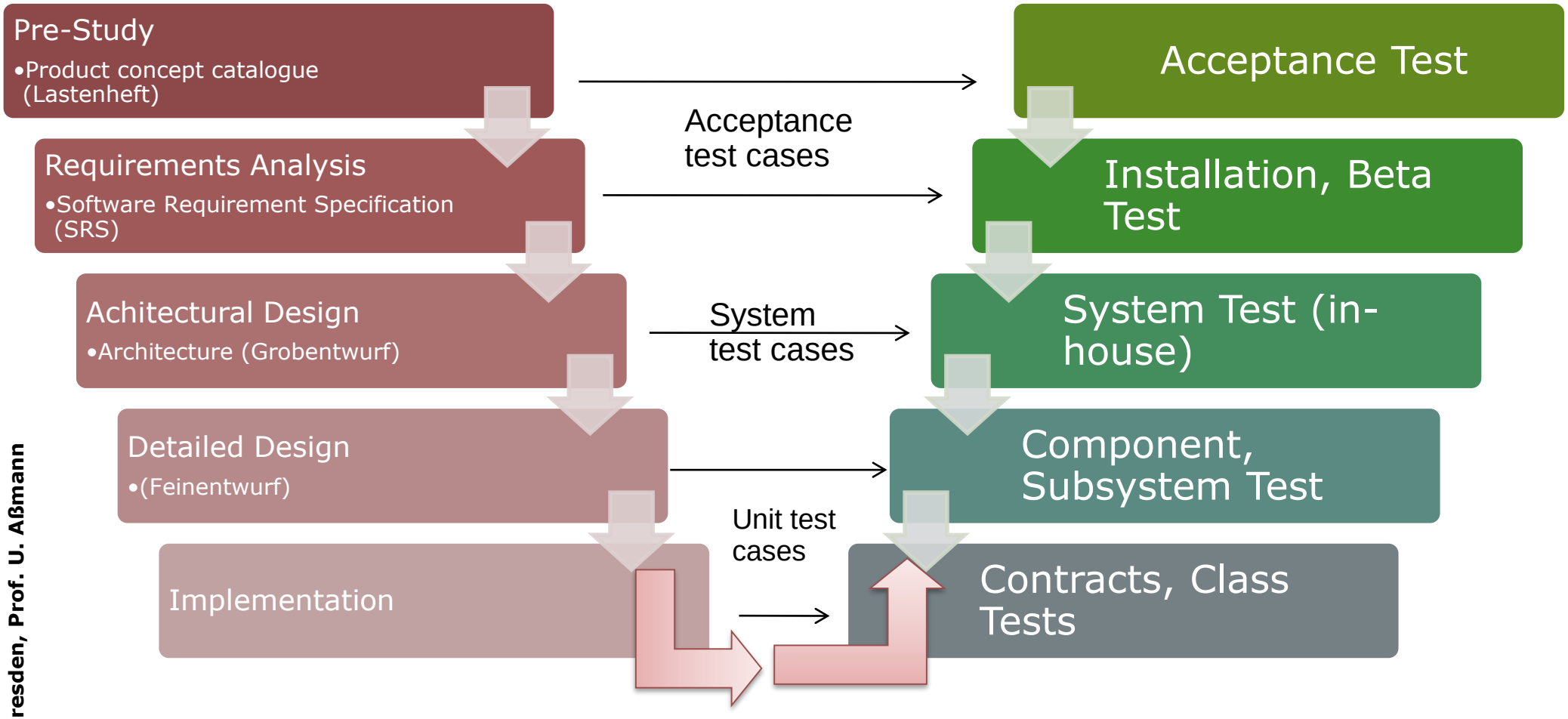
Test Management



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

Standard Test Process as Right Wing of the V Model

- Tests should be done *bottom-up*



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 specific 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]

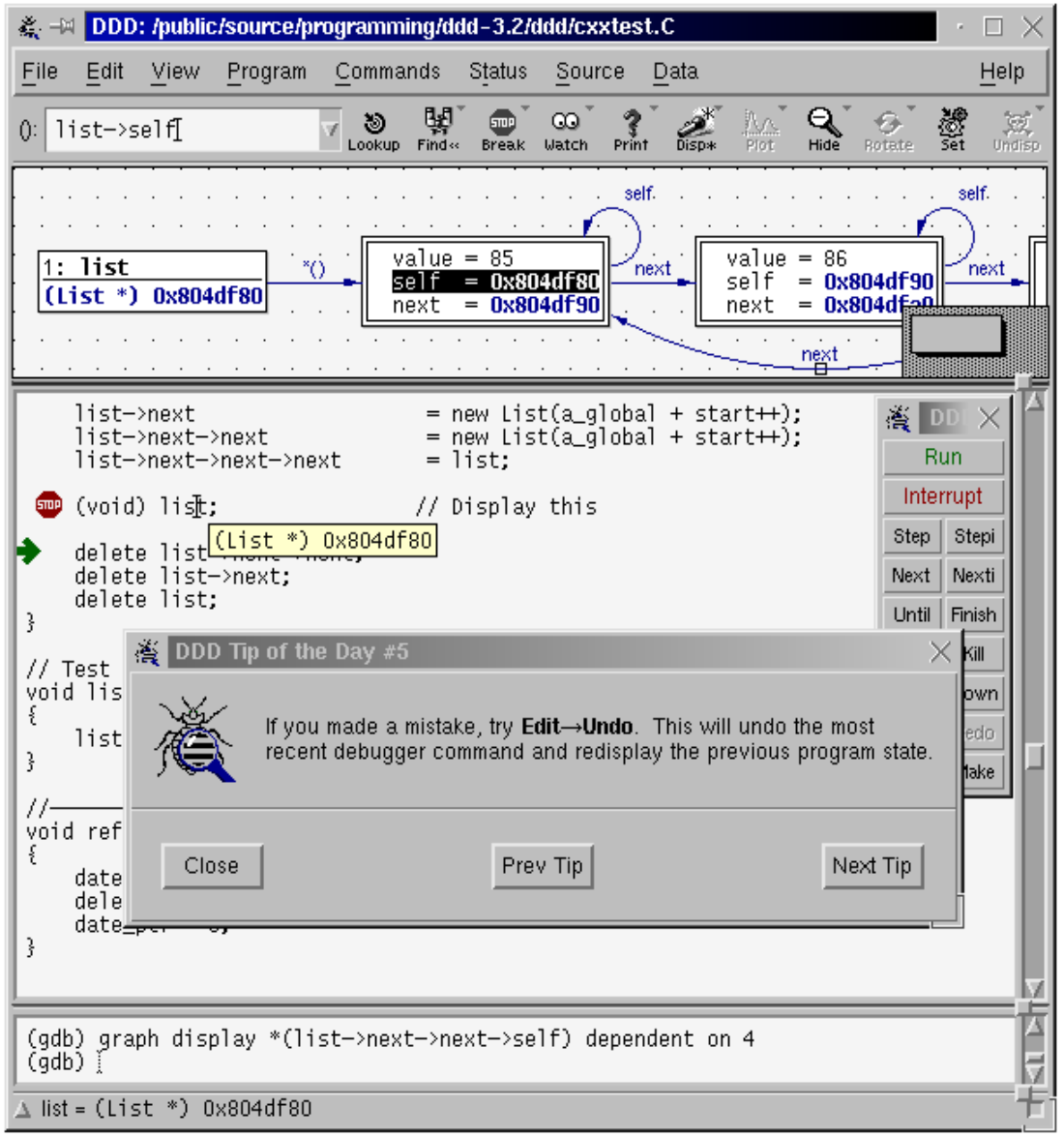
11.3 IN-VITRO-TEST RUNS WITH DEBUGGERS

Debugger (Entwanzer)

- A ***Debugger*** runs a program *in-vitro* and can stop it at any time
  - ***Breakpoints****:* line numbers to stop the execution
  - ***Watchpoints***: Events changing the value of a variable
  - ***State monitoring:*** Display of all values of variables, registers, stack, heap values
  - ***State change:*** modify the values of the state
- Good debugger work with several threads so that race conditions in parallel programs can be found

Dynamic Display Debugger (DDD)

- ddd is a visualization front-end for other text-based debuggers
  - C/C++: GDB, DBX, WDB
  - Java: JDB
  - Perl: Perl debugger
  - bash: bashdb
  - make: remake
  - Python: pydb
- ddd visualizes data structures in the heap

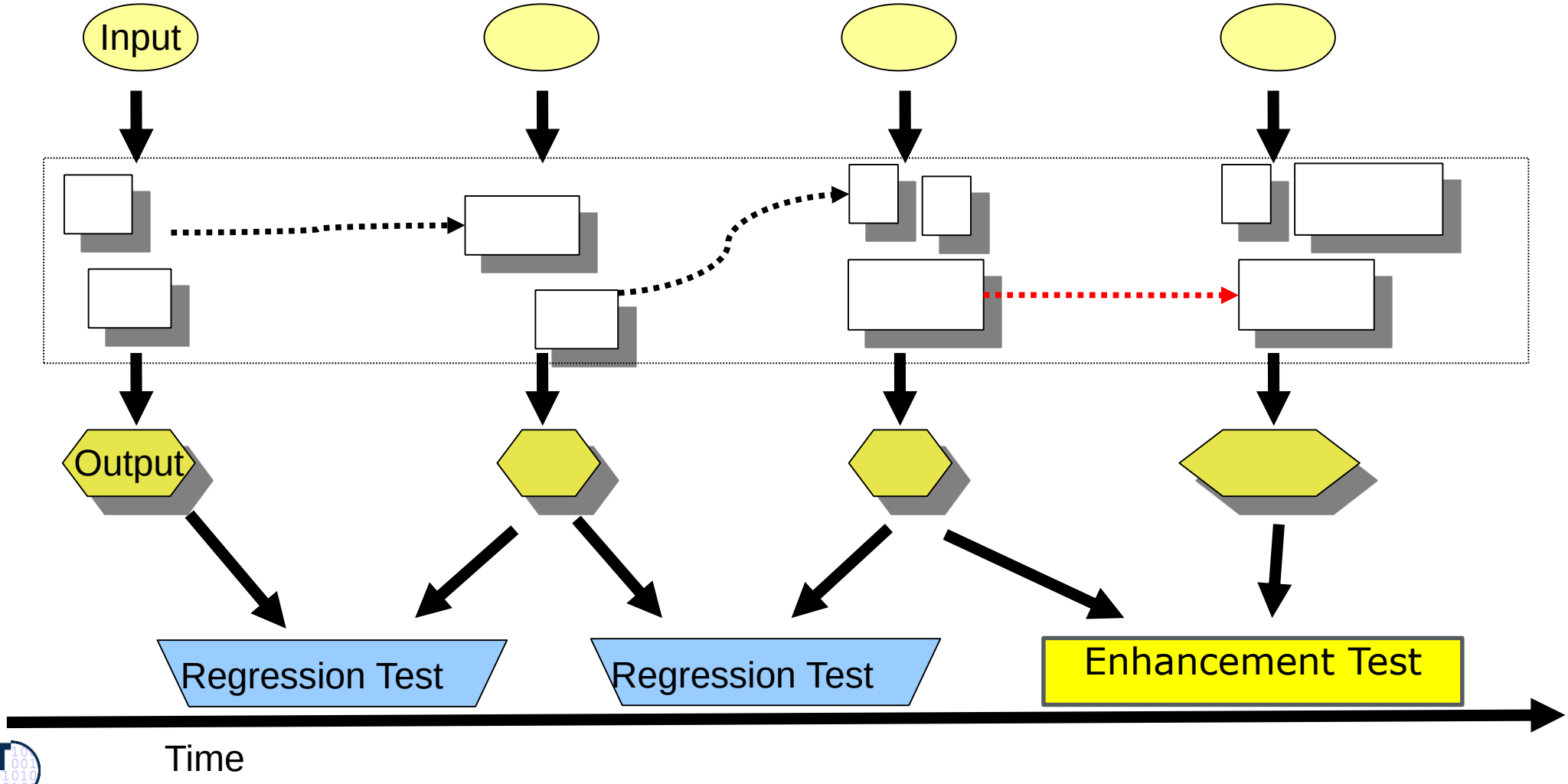


How to be sure that a change did not introduce errors...
Diversifying tests

11.4 REGRESSION TESTS

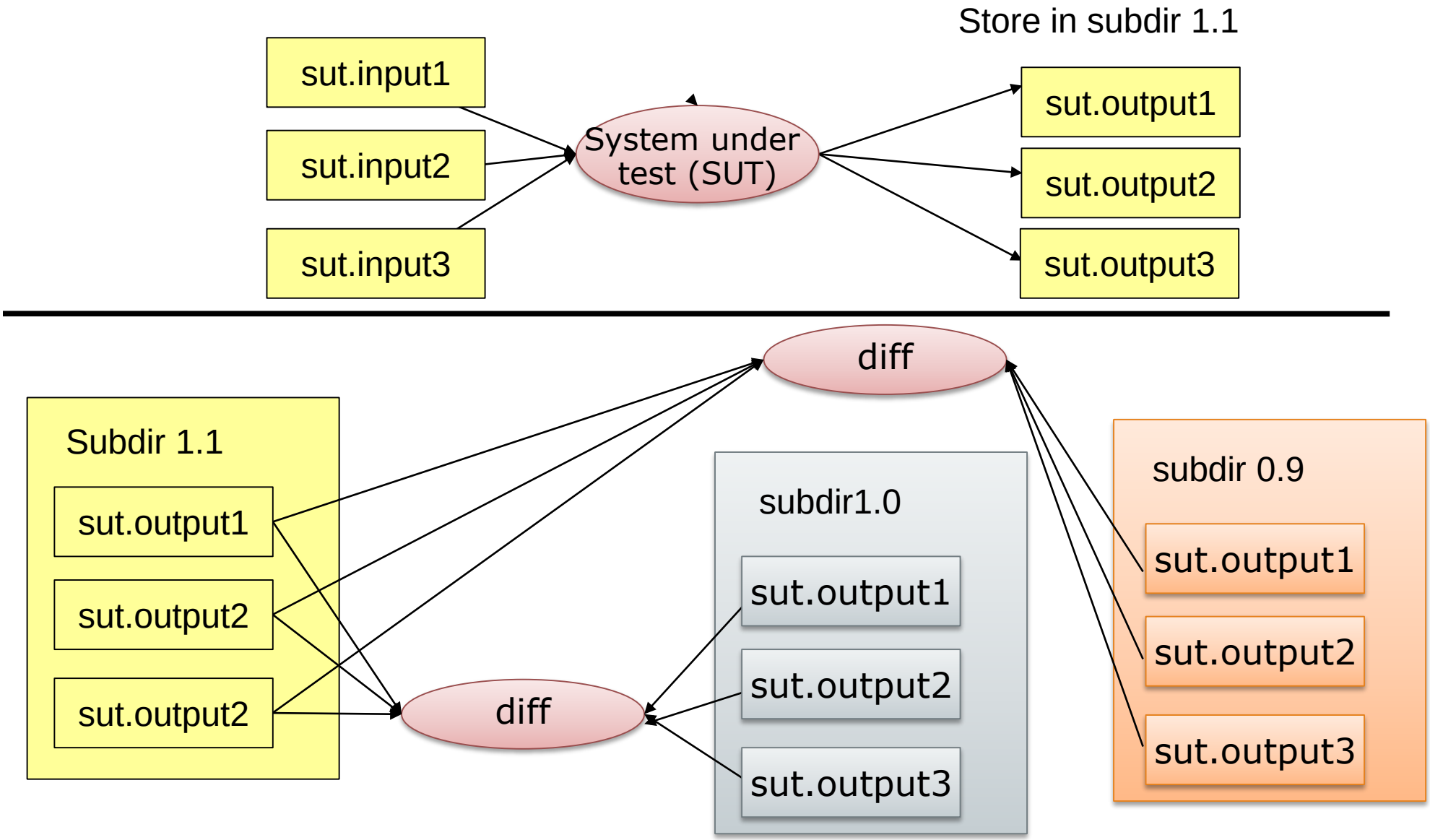
Regression Tests as Diffs on Outputs of Subsequent Versions

- 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)



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
- Colordiff uses colors
- On windows, use cygwin shell (www.cygwin.org)

Coverage Patterns of Regression Tests

- [Binder] distinguishes 5 *coverage patterns* for *redoing* regression tests:

1. Re-test all test data (exhaustive): that is the best
2. Re-test risky use cases
3. Re-test profile: profile code and re execute tests on most executed code
4. Re-test changed code (code that changed between versions)
5. Re-test 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

Without excellent regression test suite – there is no product.

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
  - Drag-and-drop situations
  - 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

11.4.2 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 through 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*

Classification of Test Cases

- 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)

Separation of Test Data and Test Cases

- 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

11.4.3 Stubs and Co

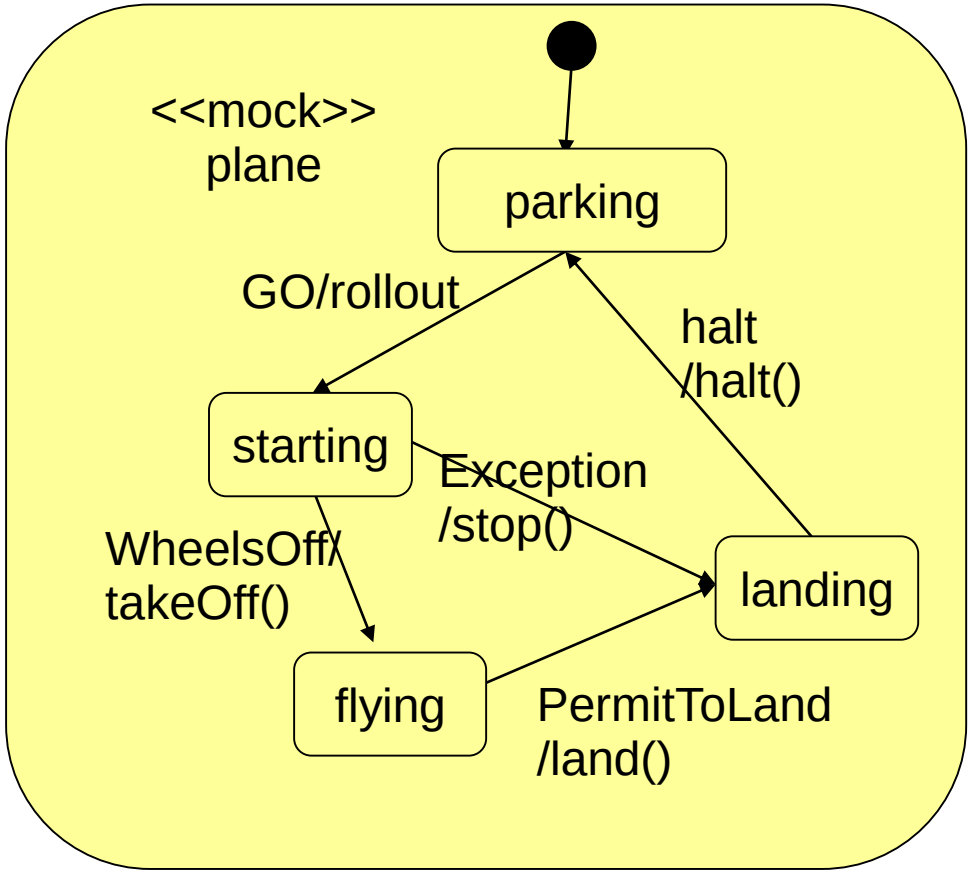
- **Stub**: Empty implementations of the class-under-test (CUT) behind its interface
- **Dummy**: Simple, restricted simulation of the interface functionality
- **Mock:** Dummy that also checks the protocol of the class-under-test (to mock, „etwas vortäuschen“)
  - Often statecharts (Steuerungsmaschinen)

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

Mock Object for (Uni-)Modal Classes

- A **mock object** simulates a class-under-test, implementing the life-cycle protocol
- If the CUT has 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

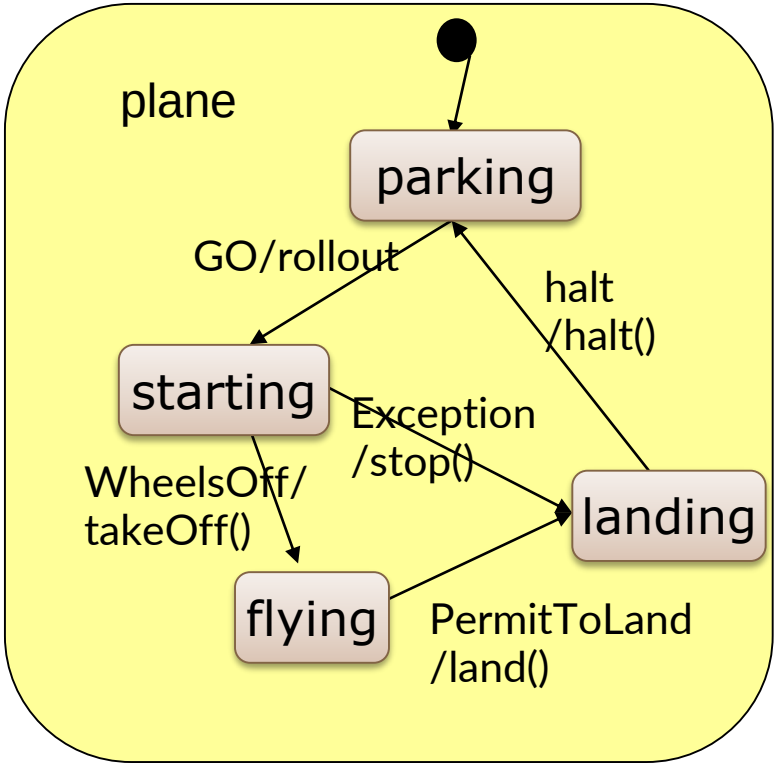
- Test case tests:
  - Path 1: parking->starting->flying->landing->parking
  - Path 2: parking->starting->landing->parking (emergency path)
- Driver checks that after each method that is called for a transition, the right state is reached
- Mock object implements state transitions



Mock Object


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

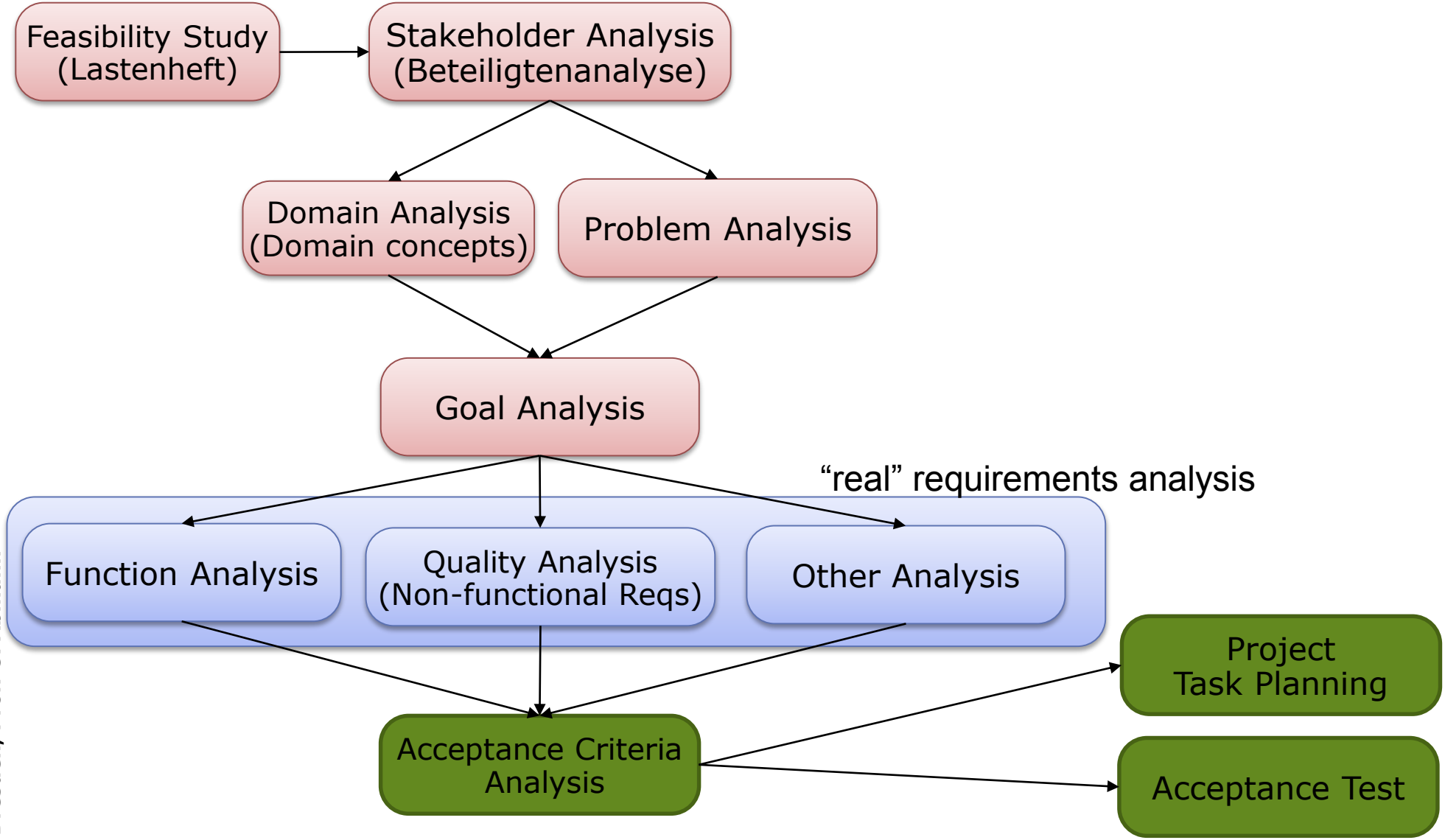
```
public class PlaneTestCase extends TestCase {
  pMock = new PlaneMock();
  public void setUp() { .. }
  public void tearDown() { .. }
  public void testPath1(){
    pMock.rollout();
    assertEquals(pMock.starting,pMock.getState());
    pMock.takeOff();
    assertEquals(pMock.flying,pMock.getState());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

11.4.4 Acceptance Tests

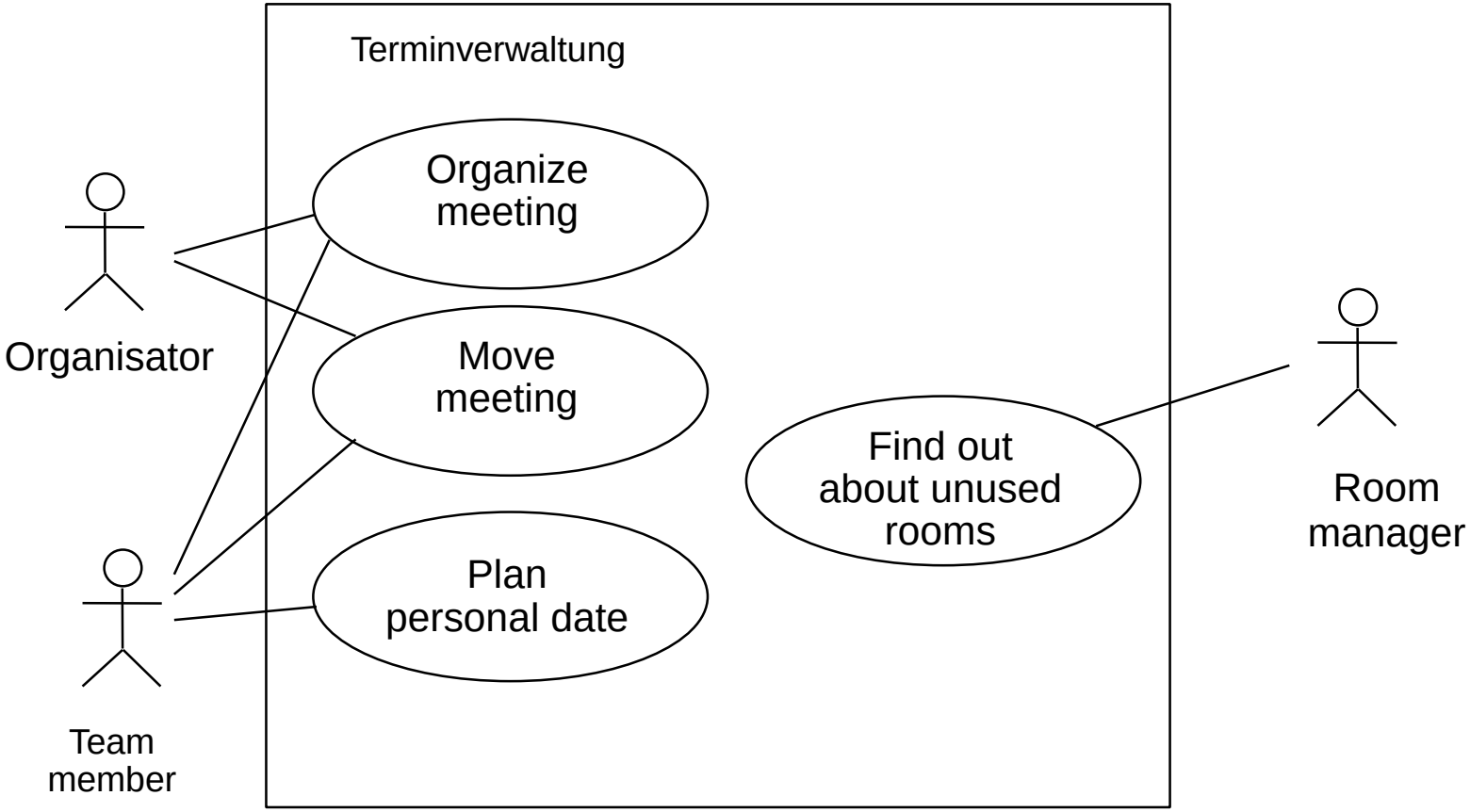


Acceptance Test

- *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

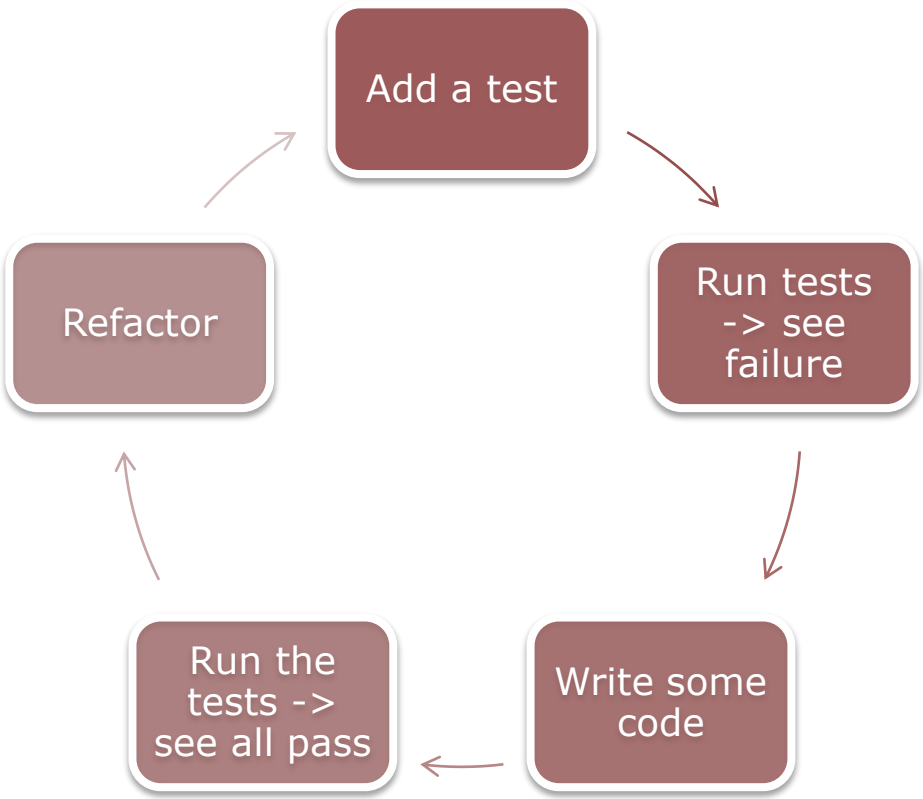


Tests and Tutorials

- 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



Test-First Development (Test-Driven Development)

- 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!

What Have We Learned?

- 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

The End