Course-Software-Testing

MarDATA

Enno Prigge Willi Rath Dilip Hiremath Sven Gundlach

Kiel University

18th November 2020

New features: „Naive“ Workflow

Typical „unstructured “ approach:

- Implementation of new features
- Call and test of features from main method, output with **System.out.println**
- Manual check whether results are correct (or at least plausible)
- Test code in Main method is not used later

New features: „Naive“ Workflow

Typical „unstructured “ approach:

- Implementation of new features
- Call and test of features from main method, output with **System.out.println**
- Manual check whether results are correct (or at least plausible)
- Test code in Main method is not used later

Cons?

- Code in Main method not „belonging there“
- Test code is discarded, although it is still useful later
- Tests are no longer executed

$\longrightarrow$ Regressions are not recognized

Continuous integration

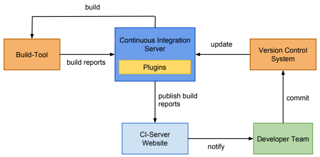
Why?

- Integration errors are continuously identified
- Early warnings for non-matching components
- Regression tests identify errors
- The developers are "‘trained"’ to be responsible
- Tool example:
  - Jenkins `http://jenkins-ci.org/`

See also the lectures on continuous integration for Kieker/ExplorViz and at PPI.

Continuous Integration Environment

Continuous Integration in Zeiten agiler Programmierung https://heise.de/-1427092



Automated Testing

Automatic tests:

- Test code in a specially designated area of the project
- Execution like „Main methods“, but tool support for:
    - Comparison with expected behavior
    - automatic execution of all tests
    - Statistics on completed and failed tests

Automated Testing

Automatic tests:

- Test code in a specially designated area of the project
- Execution like „Main methods“, but tool support for:
  - Comparison with expected behavior
  - automatic execution of all tests
  - Statistics on completed and failed tests

Pros:

- Separation of functionality and tests
- Tests are retained
- Tests can be run continuously, automatically
  $\rightarrow$ Regressions are identified
- Tests as „Qquality gateway“ for integration of new code into the master branch or master repository

Automated Testing: JUnit

JUnit

- First published 2000
- eclipse-support since 2002
- Integration e.g. with Maven

Automated Testing: JUnit

JUnit

- First published 2000
- eclipse-support since 2002
- Integration e.g. with Maven

JUnit in Softwaretechnik:

- Structure and example in lecture
- Details in exercise
- Your submission: JUnit-Tests required (Quality-Gateway for correction)

An Demo-function

that should be tested . . .

```
package com.example.project;

public class Calculator {

        public int add(int a, int b) {
                return a - b;
        }

}
```

Take a look at this in eclipse.

An Demo-function

```
package com.example.project;

import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

class MyTest {

        @Test
        @DisplayName("1␣+␣1␣=␣2")
        void addsTwoNumbers() {
            Calculator calculator = new Calculator();
            assertEquals(2, calculator.add(1, 1),
                                          "1␣+␣1␣should␣equal␣2");
        }

}
```

Automated Testing: Summary

Evaluation:

- Automatic tests cause few additional work!
- Help to recognize quality problems early.

Automated Testing: Summary

Evaluation:

- Automatic tests cause few additional work!
- Help to recognize quality problems early.

Questions:

- How many tests should I write?
- Which parts of the software should be tested?
- And what sample data?
- From where do I know the „expected“ results?
- ...

Automated Testing: Summary

Evaluation:

- Automatic tests cause few additional work!
- Help to recognize quality problems early.

Questions:

- How many tests should I write?
- Which parts of the software should be tested?
- And what sample data?
- From where do I know the „expected“ results?
- ...

More on this in the section „Quality control“.

Mocking: Motivation I

Unit-Testing:

- Code uses other objects of the system
- Objects must be accessible for testing

Issues:

- Unit-Testing! No complete system run!
- Other objects / classes may not be available yet!
- Performance
- Difficult to provide real object:
  - Specific parameters (time, sensor values)
  - Error states (network error, . . .)

Mocking: Motivation II

- Temperature-Sensor
- Network services

Mocking is particularly relevant for „Test Driven Development“.

Approach: Mock objects

Alternative: Mock objects (also: Test Doubles):

- Objects that resemble real behavior
- Can be used in test instead of the real object.

Approach: Mock objects

Alternative: Mock objects (also: Test Doubles):

- Objects that resemble real behavior
- Can be used in test instead of the real object.

Interface:

- Must be usable instead of the „real object“
  $\rightarrow$ Same interface (or subinterface) as real object

Approach: Mock objects

Alternative: Mock objects (also: Test Doubles):

- Objects that resemble real behavior
- Can be used in test instead of the real object.

Interface:

- Must be usable instead of the „real object“
  $\rightarrow$ Same interface (or subinterface) as real object

Behaviour:

- „Based on“ real object
  $\rightarrow$ for code to be tested: like real object
- Explicit triggering of values, states, . . .
  $\rightarrow$ for Test-Code:
  configurable!

Mock objects: Differentiations

Most simple version: Stub object

- Always returns fixed results to requests
- Results can possibly be set by the test code

. . .

Mock objects: Differentiations

Most simple version: Stub object

- Always returns fixed results to requests
- Results can possibly be set by the test code

. . .

Most complex version: Fake object

- Working implementation as real object
- Take shortcuts and behave in a much simpler way

Mock objects: Categories

https://martinfowler.com/articles/mocksArentStubs.html

Dummy objects	are passed around but never actually used. Usually they are just used to fill parameter lists.

Mock objects: Categories

https://martinfowler.com/articles/mocksArentStubs.html

Dummy objects	are passed around but never actually used. Usually they are just used to fill parameter lists.
Fake objects	actually have working implementations, but usually take some shortcut which makes them not suitable for production (an in-memory database is a good example).

Mock objects: Categories

https://martinfowler.com/articles/mocksArentStubs.html

Dummy objects	are passed around but never actually used. Usually they are just used to fill parameter lists.
Fake objects	actually have working implementations, but usually take some shortcut which makes them not suitable for production (an in-memory database is a good example).
Stubs	provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.

Mock objects: Categories

https://martinfowler.com/articles/mocksArentStubs.html

Dummy objects	are passed around but never actually used. Usually they are just used to fill parameter lists.
Fake objects	actually have working implementations, but usually take some shortcut which makes them not suitable for production (an in-memory database is a good example).
Stubs	provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.
Spies	are stubs that also record some information based on how they were called.

Mock objects: Categories

https://martinfowler.com/articles/mocksArentStubs.html

Dummy objects	are passed around but never actually used. Usually they are just used to fill parameter lists.
Fake objects	actually have working implementations, but usually take some shortcut which makes them not suitable for production (an in-memory database is a good example).
Stubs	provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.
Spies	are stubs that also record some information based on how they were called.
Mocks	are [...] objects pre-programmed with expectations which form a specification of the calls they are expected to receive.

Framework: Mockito

Übersicht:

- `https://site.mockito.org/`
- Java-Framework for creating Mock objects
- First release: 2008

If you use Mockito in tests you typically:

1. Mock away external dependencies and insert the mocks into the code under test
2. Execute the code under test
3. Verify that the code executed correctly

For example, you can verify that a method has been called with certain parameters. This kind of testing is sometimes called behavior testing. Behavior testing does not check the result of a method call, but it checks that a method is called with the right parameters.

Mockito with JUnit I

```
import static org.mockito.Mockito.*;

public class MockitoTest  {

  @Mock
  MyDatabase databaseMock;                          // 1

  @Rule public MockitoRule mockitoRule
                        = MockitoJUnit.rule();      // 2

  @Test
  public void testQuery()  {
    ClassToTest t = new ClassToTest(databaseMock);  // 3
    boolean check = t.query("*␣from␣t");            // 4
    assertTrue(check);                              // 5
    verify(databaseMock).query("*␣from␣t");         // 6
  }
```

Mockito with JUnit II

```
}
```

Mockito with JUnit III

1. Tells Mockito to mock the databaseMock instance
2. Tells Mockito to create the mocks based on the `@Mock` annotation
3. Instantiates the class under test using the created mock
4. Executes some code of the class under test
5. Asserts that the method call returned true
6. Verify that the query method was called on the `MyDatabase` mock

Siehe auch `https://www.vogella.com/tutorials/Mockito/article.html`

Mocking: Summary

Motivation: „Replace“complex objects for test:

- Simulation
- Replace with dummies
- Collect additional debug information

Connection to Unit-Testing:

- Extends assertions by Verification: Testing of interactions with mock objects

Verification:

- Mock objects log function calls. This allows to check properties of the *stack trace (call history)* of mock objects.
- **Note**: Do not mistake for formal verification!

Unit Tests *vs.* Mock-Verification?

- Addition, not replacement: Check different aspects!
- **More details and examples: practice!**

Quality control: Overview

Quality control for software:

- Known: Software errors occur in practice.
- Goal: Prevent errors **at least** in live software.

Options?

Quality control: Overview

Quality control for software:

- Known: Software errors occur in practice.
- Goal: Prevent errors **at least** in live software.

Options?

- Code Review
  - four-eyes principle
  - Team Review, if applicable via pair programming

Quality control: Overview

Quality control for software:

- Known: Software errors occur in practice.
- Goal: Prevent errors **at least** in live software.

Options?

- Code Review
  - four-eyes principle
  - Team Review, if applicable via pair programming
- Tests
  - Manually
  - Automatically

Quality control: Overview

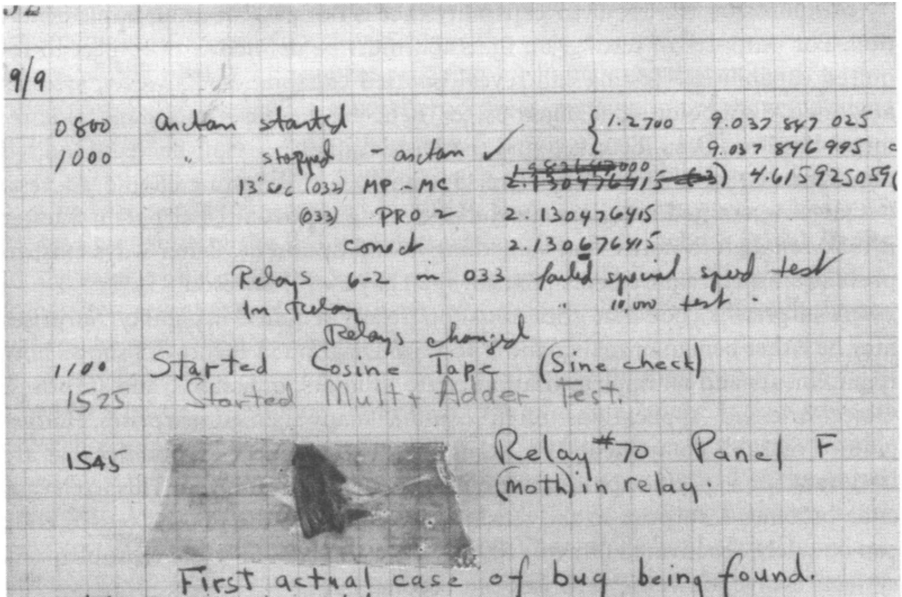
Quality control for software:

- Known: Software errors occur in practice.
- Goal: Prevent errors **at least** in live software.

Options?

- Code Review
  - four-eyes principle
  - Team Review, if applicable via pair programming
- Tests
  - Manually
  - Automatically
- Formal methods
  - Formal verification
  - Proofs of Program Correctness

Things we do not want: Bugs



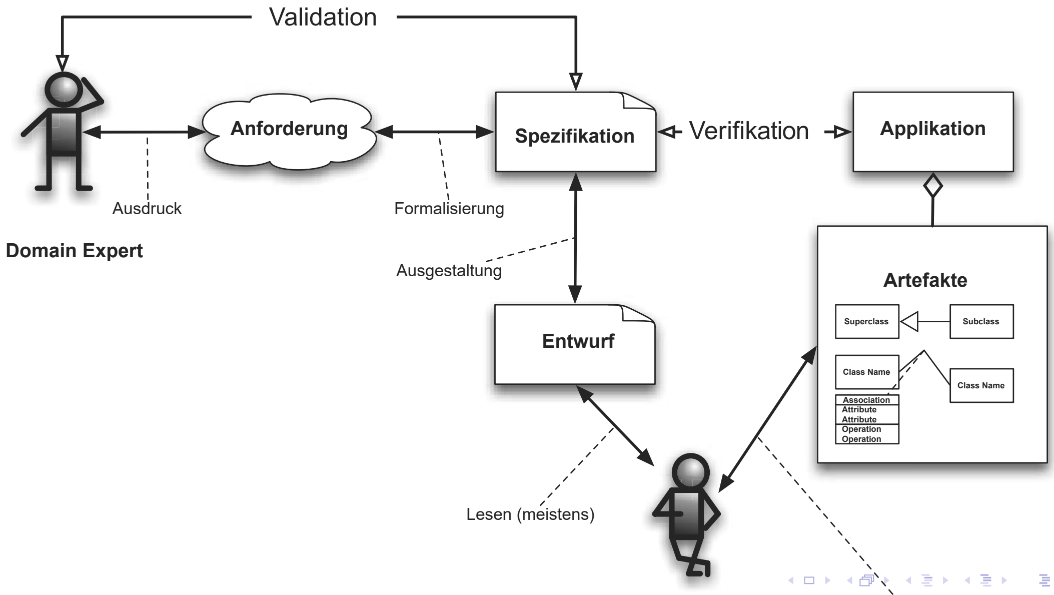
9/9

0800 Anctan started {1.2700 9.037 847 025
1000 " stopped - anctan ✓ 9.037 846 995
13" sec (032) MP - MC ~~2.130476415~~ ~~(23)~~ 4.615925059
(033) PRO 2 2.130476415
const 2.130676415
Relays 6-2 in 033 failed special speed test
In relay " 10,000 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult+ Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.

Validation and verification of software

- Validation and verification of the implemented software system:
  - "‘Do we build the system right?"’ (Verification)
    Check that the system meets the requirements
  - "‘Do we build the right system?"’ (Validation)
    Check whether the *actual* (user) requirements have been realized.
- Validation can be supported e.g. by prototyping (see LE **??**)
- Occasionally these terms are also used differently:
  - e.g. verification for formal proofs and validation for *running*.

Validation and Verification: Roles



Verification Objectives

- Proof of the correct functioning of all functions of a system in relation to a given specification
  - Identify errors
  - Absence of errors
- The verification process itself must also be verified:
  - Are the test conditions correct?
  - Is a formal proof correct?
- Technical (non-functional) requirements must also be checked:
  - Efficiency, portability, modifiability, etc.
- The result is not always simply *correct* or *incorrect*.
  - Reliability requirements and subjectivity require more fine-grained metrics, e.g. *good enough* or 99% availability.

Classification of verification techniques

Symbolic Execution

The behavior of the system is tested with respect to the expected behavior.

- Structure-oriented methods
  white-box, with knowledge of the implementation
- Function-oriented methods
  black-box, without knowledge of the implementation

Static Analysis

The behavior of a system is analyzed to derive whether the behavior is correct.

- Static proofs of correctness
- Reviews, inspections, walkthroughs, etc.

Testing for verification

Unit-Test	Single functions / components are tested.
Integration test	Interdependent components are tested together. Focus on interface testing. Components are integrated into subsystems and tested together.
System test	Test of the complete system.

Objective of a test planning

Select test cases in such a way that the probability of finding errors or ruling them out is high.

The V-Model

Use especially by military and governmental authorities.

images/Qualitaetssicherung/abbildungen/VModel

Verification and validation of the sub-products are essential components of the V-Model.

Problems in software testing

- Test cases often describe the behavior more detailed than the specification:
  $\longrightarrow$ Develop test cases before implementation
- Basic problem:
  "Program testing can be used to show the presence of bugs, but never to show their absence."
- Questions to answer:
  - How to determine suitable test cases and test plans?
  - How to check the correctness of the test implementation?
  - Who is the tests authority?

Challenges of the steps in the test process

- Identification of test scenarios (use cases)
  - What is to be tested?
- Establishment of concrete test scenarios
  - How is the test candidate executed?
  - What is the correct behavior?
- Formalization of tests
  - How is the test specified?
- Carrying out tests
  - Automation!
- Maintenance of tests
  - Synchronized with the changes in the business software, the tests must (mostly) be adapted and modified.

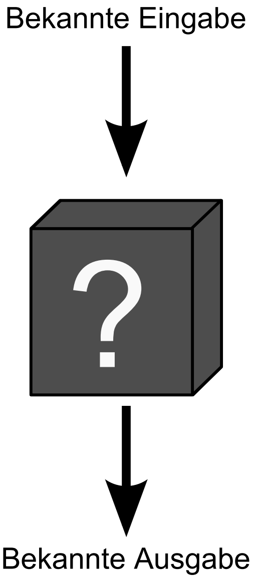
Objectives for software testing

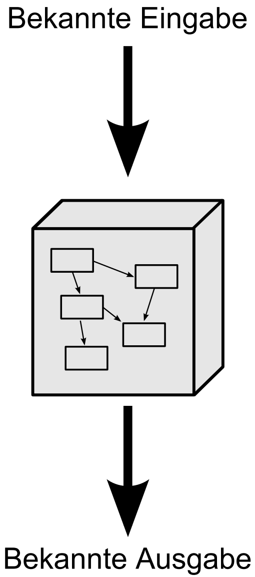
- It must be clear what results are expected when testing.
- Use of *systematic* methods:
  - Selection of test cases if possible not (only) intuitive or random.
- The results should be fully reproducible.
  - This is a major problem especially when testing parallel programs.
- Errors should not only be detected, but also localized and fixed.
  - ‚Known bugs‘ are not a sign of high quality software.
- Quality requirements must be checked with the highest accuracy possible.

Pragmatic view on testing

- Testing allows conclusions about the quality of a software system
- Quantitative metrics support conclusions
  - Test coverage etc.
- Automated regression tests provide protection against side effects of modifications
  - A regression test is the repetition of test cases to ensure that modifications in already tested parts of the software do not cause new errors (“regressions”).

Black-Box vs. White-Box Testing





White-box Testing of Modules

(testing in the small)

C0: Statement Coverage: All executable statements in the source code are executed at least once (Statement coverage criterion).

C1: Edge coverage: Each edge of the control graph being traversed at least once (Edge-coverage criterion).

C2: Path (Branch) Coverage: All paths leading from the initial to the final node of the control graph being traversed (Path-coverage criterion).

C3: (Compound) Condition Coverage: Each possible combination of conditions must be executed at least once (Condition coverage criterion).

Problems with Statement coverage

C0: Statement coverage

- Is a program sufficiently tested when each statement has been executed at least once?
- How to determine a minimum test quantity?
- Is it useful to cover even empty statements (e.g. missing `else`)?
- The structure of the program is not taken into account
- Basically:
  Coverage is not decisive!

Coverage of all edges of a control flow graph

C1: Edge-coverage

Precondition is the construction of a control flow graph for an (imperative) program:

- Every single statement, which does not contain any further statements, is represented as a node.
- Conditional statements are represented as branches.
- Loops are represented as cycles and
- Sequences by edges between nodes.

Construction of control flow graphs

images/Qualitaetssicherung/abbildungen/ControlFlowGraphs

Example: Program for faculty computation

```
public long faculty
              (int n) {
  if (n < 0) {
    throw new
      Exception();
  }
  long fac = 1;

  while (n > 0) {
    fac *= n;
    n--;
   }
  return fac;
}
```

Example: Program for faculty computation

```
public long faculty
              (int n) {
  if (n < 0) {
    throw new
      Exception();
  }
  long fac = 1;

  while (n > 0) {
    fac *= n;
    n--;
   }
  return fac;
}
```

images/Qualitaetssicherung/abbildungen/FacultyCalculation

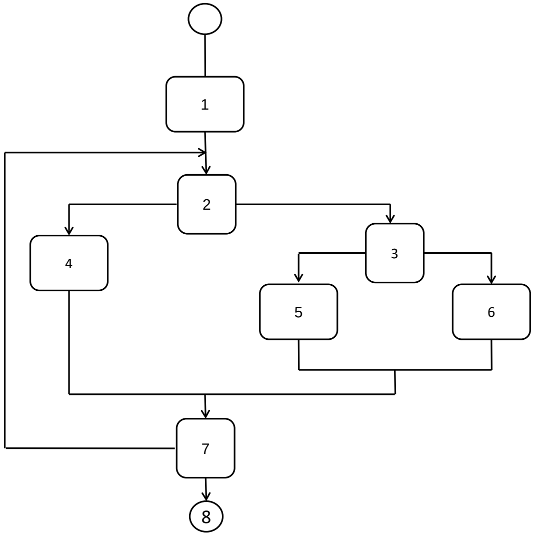
Test cases for Edge-coverage

Nr	Class	Test case description	Expected Results	Exemplary input data
1	Normal	Input parameter is valid	Faculty of the input parameter	42
	Checked edges: 2, 3, 4, 5, 6, 7			
2	Error	Input parameter is invalid	Exception	-1
	Checked edges: 1			

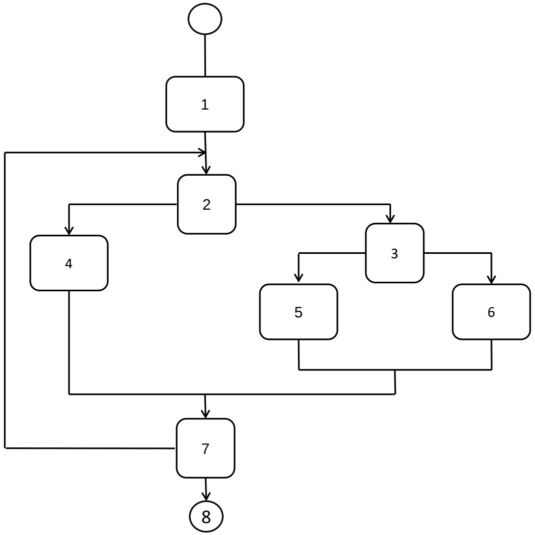
Cyclomatic Complexity

- Behind this software metric by McCabe is the idea that above a certain complexity a module is no longer comprehensible to humans.
- The cyclomatic complexity is defined as the number of independent paths on the control flow graph of a module.
- The cyclomatic complexity indicates how many test cases are needed to achieve Edge-coverage.
- The complexity measure according to McCabe is equal to the number of binary branches plus 1.
- According to McCabe, the cyclomatic number of a self-contained subprogram should not exceed 10, otherwise the program is too complex and too difficult to test.

Cyclomatic Complexity: Example

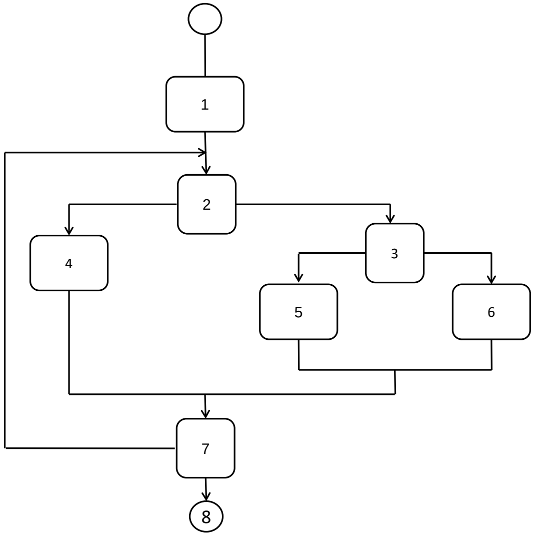


Cyclomatic Complexity: Example



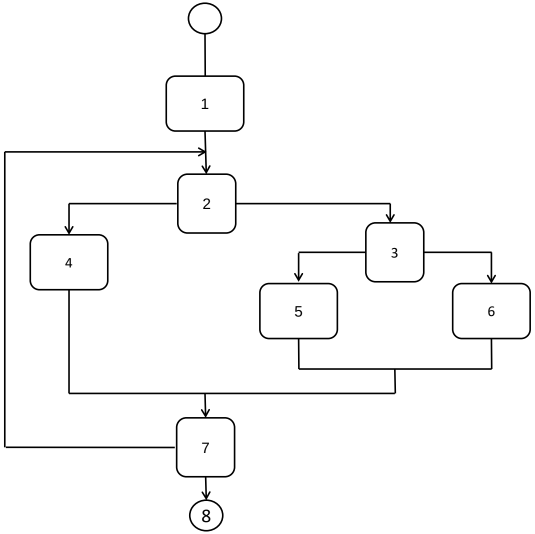
Path 1: 1,2,3,6,7,8

Cyclomatic Complexity: Example



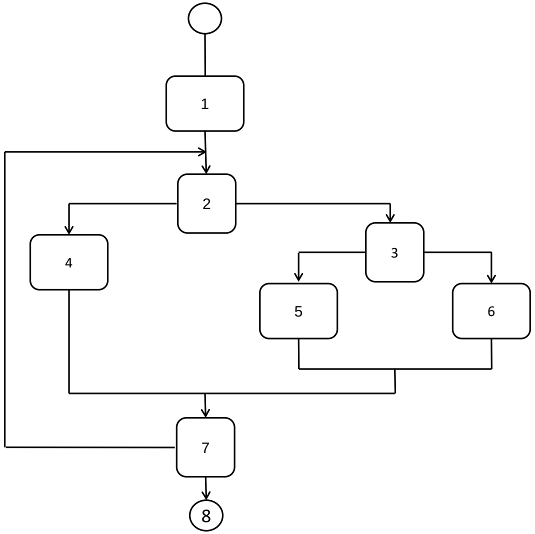
Path 1: 1,2,3,6,7,8
Path 2: 1,2,3,5,7,8

Cyclomatic Complexity: Example



Path 1: 1,2,3,6,7,8
Path 2: 1,2,3,5,7,8
Path 3: 1,2,4,7,8

Cyclomatic Complexity: Example



Path 1: 1,2,3,6,7,8
Path 2: 1,2,3,5,7,8
Path 3: 1,2,4,7,8
Path 4: 1,2,4,7,2,4,...7,8

Cyclomatic Complexity = 4

Notes on the Edge-coverage criterion

- It is enforced, that every condition is passed with true and false.
- The edge coverage criterion is thus *stronger* than the statement coverage criterion.
- Occasionally however not strong enough:
  - Compound conditions are not sufficiently considered.
- Insufficient for testing loops

Path (Branch) Coverage

C2: Path (Branch) Coverage

Even under simple conditions, the Edge-coverage can still be insufficient: Testing of loops

```
if (x != 0) {
    y = 5;
}
else {
    y = 6;
}
if (z > 1) {
    z = z / x;
}
else {
    z = 0;
}
```

Path (Branch) Coverage

C2: Path (Branch) Coverage

Even under simple conditions, the Edge-coverage can still be insufficient: Testing of loops

```
if (x != 0) {
    y = 5;
}
else {
    y = 6;
}
if (z > 1) {
    z = z / x;
}
else {
    z = 0;
}
```

- The test set {(x=0, z=1), (x=1, z=3)} meets the Edge-coverage criterion, but does not recognize the division by 0.
- Dependencies remain undetected!

Edge-coverage (continued)

- Die Testmenge
  {(x=0, z=1), (x=1, z=3), (x=0, z=3), (x=1, z=1)}
  meets the Path-coverage criterion.
- Problem: The number of paths grows exponentially with the length of the program code, so testing with the Path-coverage criterion is unrealistic.
- Another problem: Number of paths for loops..
  Solution: Heuristics.
  Example heuristics for the number of loop cycles:
  - 0 times
  - *moderately* often
  - maximum count

  Boundary values are important!

Condition coverage criterion

C3

Here e.g.

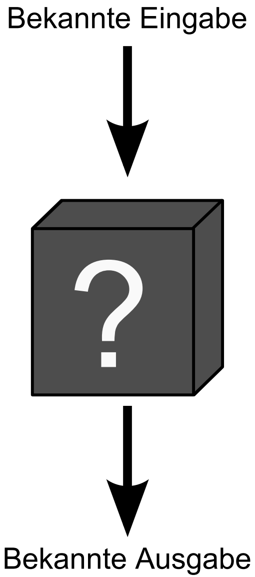
```
if (C1 and C2) ST;
  else SF;
```

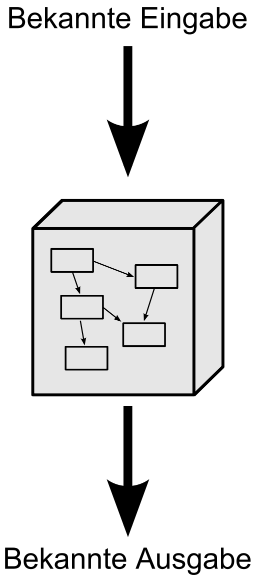
is transformed into the following statement:

```
if (C1)
  if (C2) ST
  else SF
else SF
```

- The new control flow graph covers all basic (sub)conditions and thus makes *hidden* edges visible.
- The condition coverage criterion may therefore result in even stronger test sets.

Black Box vs. White Box Testing





Blackbox Testing of modules

- Idea: A part of a program is tested without knowledge of the internal implementation.
- Problem: How to define the test set?
- Defining the test set can only rely on the specification.
- Formal specifications are then a prerequisite for the systematic generation of test sets.

Medieval Example

Solutions for equations of the form:

$$x^3 + ax^2 + bx + c = 0$$

- 1535: Tartaglia finds a solutions but keeps it to himself.
- Fior poses 30 problems to test this.

Medieval Black Box Text

$$\begin{tikzcd}[column sep=huge]
\text{Tartaglia} & & \text{Fior} \\
& & \text{solves } (x-r)(x-s)(x-t) \\
{} & & {} \arrow[ll, "x^3 + ax^2 + bx + c = 0"'] \\
{} \arrow[rr, "\text{r', s', t'}"] & & {} \\
& & \text{verifies } r = r', s = s', t = t'
\end{tikzcd}$$

Black Box Test methods

- Derive test cases from the program specification.
- Disregard program structure.
- Comprehensive but low-redundancy testing of the specified functionality.
- The key factor is the functional coverage
- Defining a test case:
  - Functional equivalence class formation (equivalence class partitioning)
  - Boundary value analysis
  - Special value testing

Equivalence Class Partitioning

Aim

Divide input parameter and output ranges into equivalence classes. Devide the definition ranges of input parameters and value ranges of output parameters into equivalence classes from which test cases can be derived.

Assumption

When processing a representative from an equivalence class, a program behaves the same way as with all other values from this equivalence class.

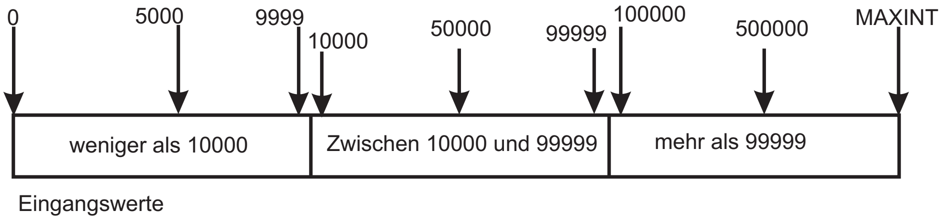
Representative value for a test case

Pick any representative value from the class.

Boundary Value Analysis

- Test cases that cover the boundary values of equivalence classes or that are in the immediate vicinity of the boundaries uncover errors very often.
- Not any element from the equivalence class is valid as representative for all boundaries.
- Pick one or more elements such that every boundary of the equivalence class is tested.

Equivalence classes with boundary values



Specification for a search function

```
procedure Search (Key : ELEM ; T: ELEM_ARRAY;
      Found : in out BOOLEAN; L: in out ELEM_INDEX) ;
```

Pre-condition

– the array has at least one element

T'FIRST <= T'LAST

Post-condition

– the element is found and is referenced by L

(Found and T (L) = Key)

or

– the element is not in the array

(**not** Found **and**

not (**exists** i, T'FIRST >= i <= T'LAST, T (i) = Key))

Equivalence Class Formation

based on the specification

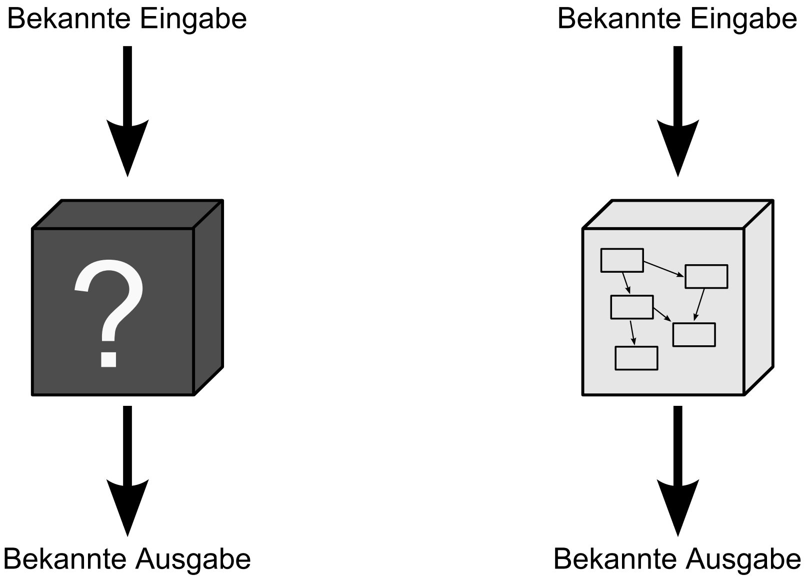
- Inputs that fulfill the pre-condition
- Inputs that fail the pre-condition
- Inputs for which the key element is in the array
- Inputs for which the key element is not in the array
- Input sets
  - With 0 elements
  - With 1 element
  - With multiple elements

Equivalence classes for the search function

Array	**Element**
Single value	In sequence
Single value	Not in sequence
More than 1 value	First element in sequence
More than 1 value	Last element in sequence
More than 1 value	Middle element in sequence
More than 1 value	Not in sequence

Input sequence (T)	**Key**	**Output** (Found, L)
17	17	true, 1
17	0	false, *
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 7
17, 18, 21, 23, 29, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, *

Black Box and White Box Testing

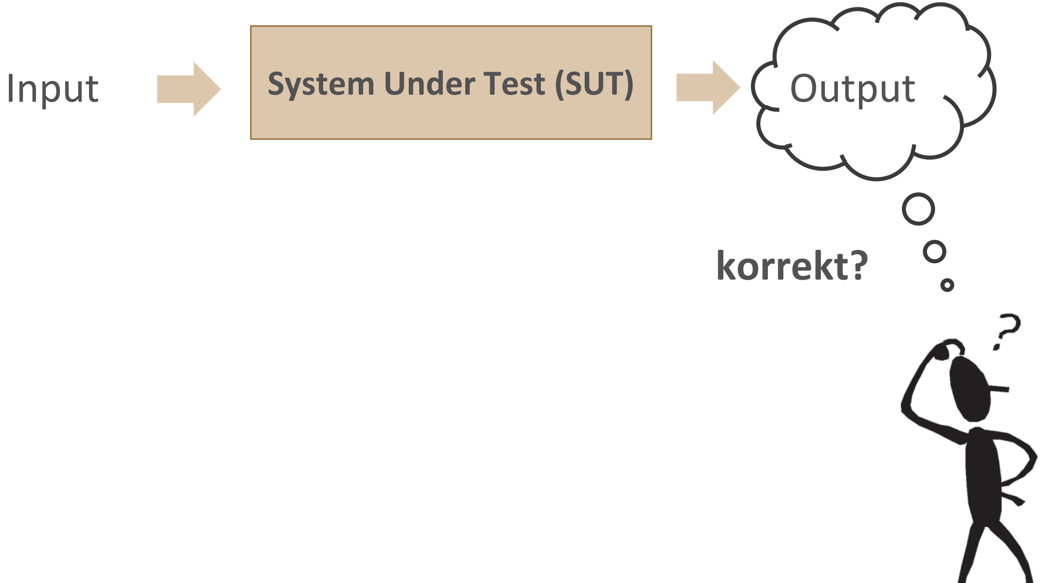


Fuzzy Testing

Fuzzing:

- Unlike unit tests, fuzzy testing does not define test cases manually but generates them randomly based on statistic functions.
- By reaching a large number of generated tests, the software is tested for unusual values as well.
    - This is especially interesting for uncovering security relevant vulnerabilities.
- Fuzzing is usually applied in black box testing to understand error affinity of new software and to detect vulnerabilities.
- If a software reproducible generates a problem (e.g. crashes) under fuzzy testing, white box tests can be used to find the specific cause.

Test Oracle

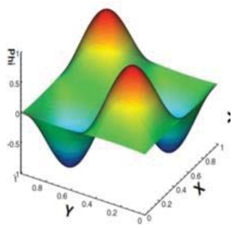


Are there types of software that do not have a test oracle

`https://monti.com`

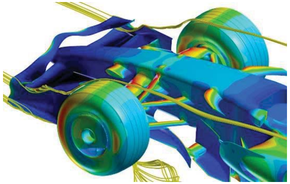
The oracle problem

It is not always possible to define a test oracle



Scientific calculations

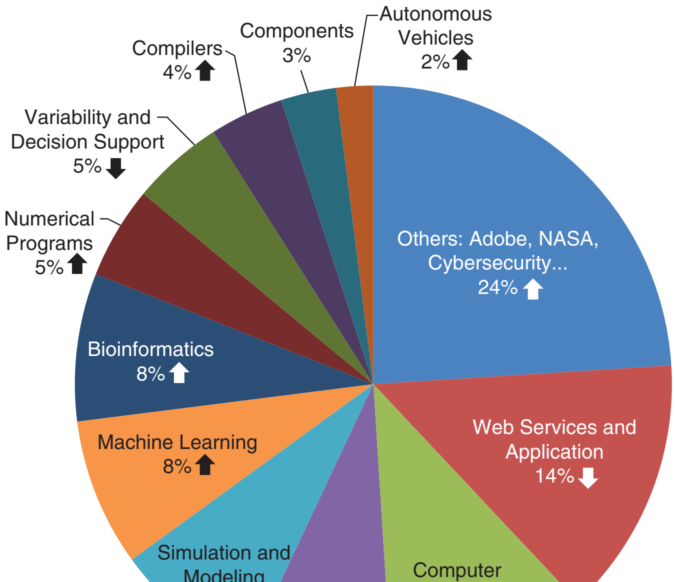
Artificial intelligence



Simulation and modelling

Was tun?

Application of metamorphic testing



Integration testing

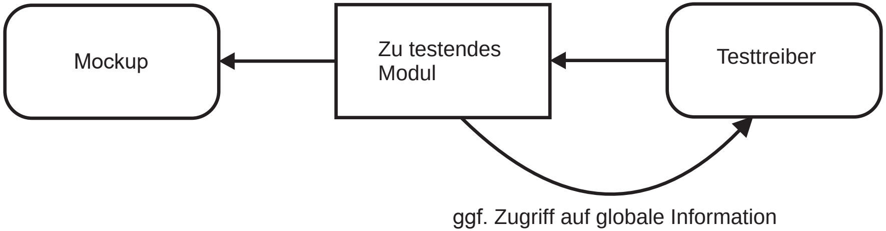
Testing in the large

- Testing individual modules separately does not guarantee a correct inter-operation of multiple modules.
- Many modules cannot be tested in isolation.
- Additionally: Steps for testing complex systems (modular testing):
  - Modular tests (a.k.a. Unit Tests & Components Tests)
    Testing a component without context.
  - Incremental integration test of multiple components in their context.
  - System test: Test of the whole system in the application environment (end-to-end).
- Acceptance tests (a.k.a. Funktional tests)
  Focus on testing of ‘cross cutting’ functionality.

Integration test (contd.)

- For embedded real time systems, the interoperation of hardware and software needs to be tested.
  - Digital Twins
- Incremental testing should be preferred over ‚Big-Bang'testing, which directly goes for the full system right after unit testing.
- A clear separation of interface and implementation makes integration testing easy.
  - Easier swapping of ‚mock-ups' for ‚real' modules.
    See, e.g., Mockito `https://github.com/mockito/`

Incremental testing



"‘Test framework"’

- Incremental tests can be bottom-up, top-down (or even jo-jo), with respect to the hierarchy of composition.
- A hierarchical architecture is very effective for this purpose.

Regression testing

- A test tool saves all test cases that have been done.
- This allows for an automatic repretition of all past tests after the software under test was changed.
- Executes nominal-actual comparison:
    - Input data for the test object
    - Expected outputs or reactions of test object (passing result)

Test driven development and refactoring

- Test driven development combines two techniques:
  - Test-driven programming for external evolution,
  - refactoring for internal evolution.
- Refactoring
  Changing a program with the aim of improving the internal structure without changing the (external) functionality.
- Refactoring is valid if all tests are passing.
- Here, tests are a form of software requirements specification (SRS).

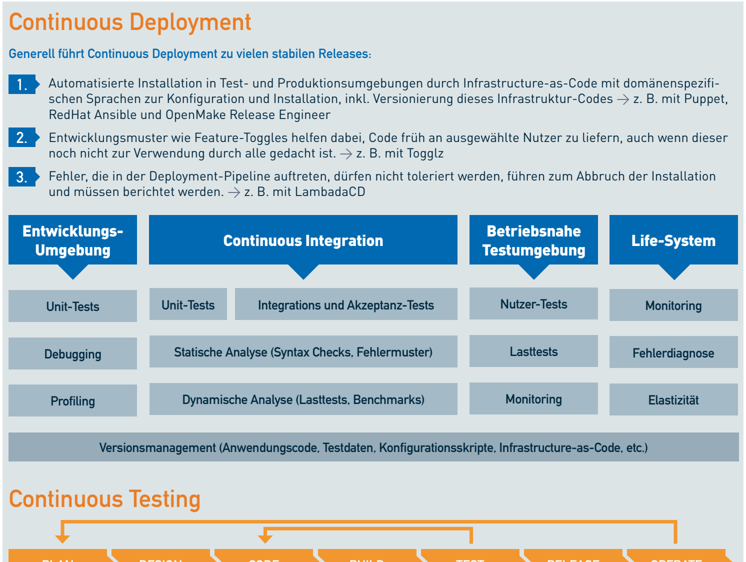
Continuous Integration

Task, i. a..:

- Execution of static analyses (e.g., SpotBugs, Checkstyle)
- Building the systems (e.g., Gradle)
- Automated testing (e.g. JUnit, Selenium)
- Checking of log and monitoring data (e.g. Kieker)

DevOps: Development and operations

Continuous Testing, Delivery & Deployment



Continuous Deployment

Generell führt Continuous Deployment zu vielen stabilen Releases:

1. Automatisierte Installation in Test- und Produktionsumgebungen durch Infrastructure-as-Code mit domänenspezifischen Sprachen zur Konfiguration und Installation, inkl. Versionierung dieses Infrastruktur-Codes → z. B. mit Puppet, RedHat Ansible und OpenMake Release Engineer
2. Entwicklungsmuster wie Feature-Toggles helfen dabei, Code früh an ausgewählte Nutzer zu liefern, auch wenn dieser noch nicht zur Verwendung durch alle gedacht ist. → z. B. mit Togglz
3. Fehler, die in der Deployment-Pipeline auftreten, dürfen nicht toleriert werden, führen zum Abbruch der Installation und müssen berichtet werden. → z. B. mit LambdaCD

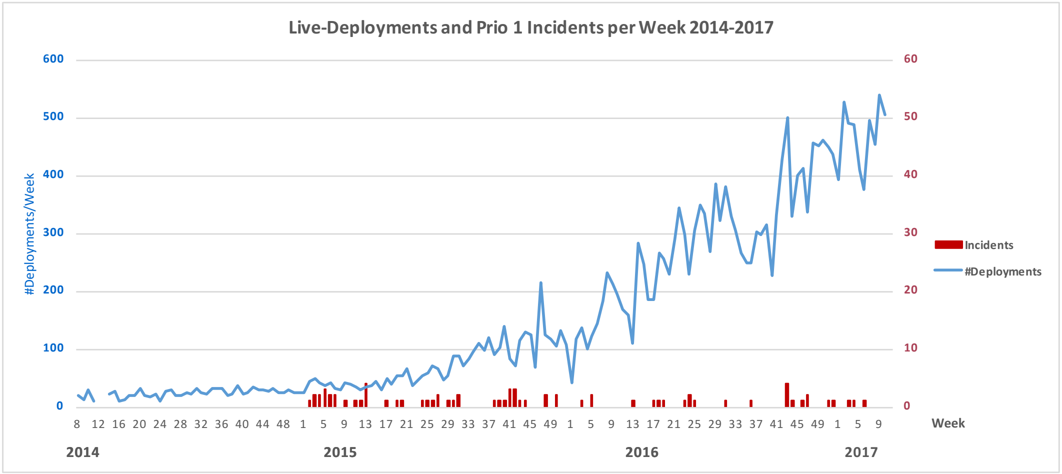
Entwicklungs-Umgebung	Continuous Integration		Betriebsnahe Testumgebung	Life-System
Unit-Tests	Unit-Tests	Integrations und Akzeptanz-Tests	Nutzer-Tests	Monitoring
Debugging	Statische Analyse (Syntax Checks, Fehlermuster)		Lasttests	Fehlerdiagnose
Profiling	Dynamische Analyse (Lasttests, Benchmarks)		Monitoring	Elastizität

Versionsmanagement (Anwendungscode, Testdaten, Konfigurationsskripte, Infrastructure-as-Code, etc.)

Continuous Testing

Agility and reliability

By Continuous Testing, Example otto.de


Live-Deployments and Prio 1 Incidents per Week 2014-2017
#Deployments/Week
Incidents
#Deployments
Week