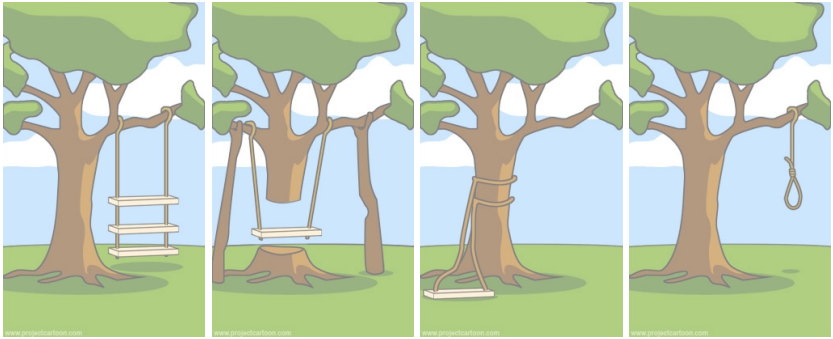
Software Engineering

8. Testing | Thomas Thüm | January 14, 2021

Software Testing



how the customer explained it | how the analyst designed it | how the programmer implemented it | what the beta testers received

Lecture Overview

1. Quality Assurance
2. White-Box Testing
3. Black-Box Testing

Quality Assurance

Andy Hunt

"No one in the brief history of computing has ever written a piece of perfect software. It's unlikely that you'll be the first."

Donald Trump (May 2020)

"If we didn't do any testing, we would have very few cases."

Software Quality

Quality

Quality is the entirety of properties and characteristics of a product or process that indicate adequacy with respect to given requirements. [Ludewig and Lichter]

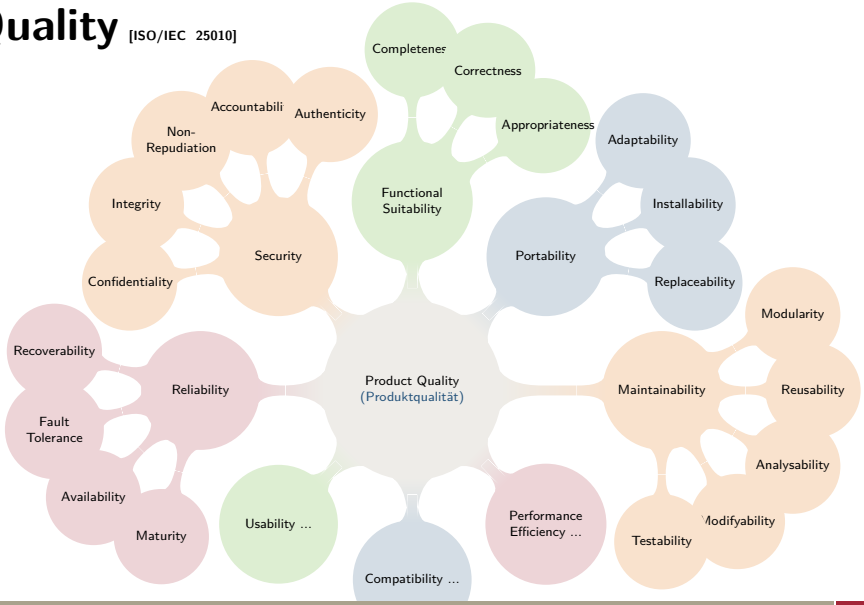
Quality Assurance (Qualitätssicherung)

Quality assurance are all activities with the goal to improve the quality. [Ludewig and Lichter]

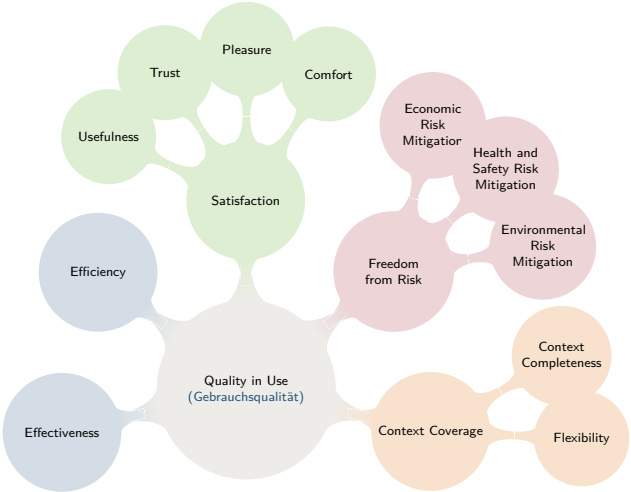
Expectations on Quality [Sommerville]

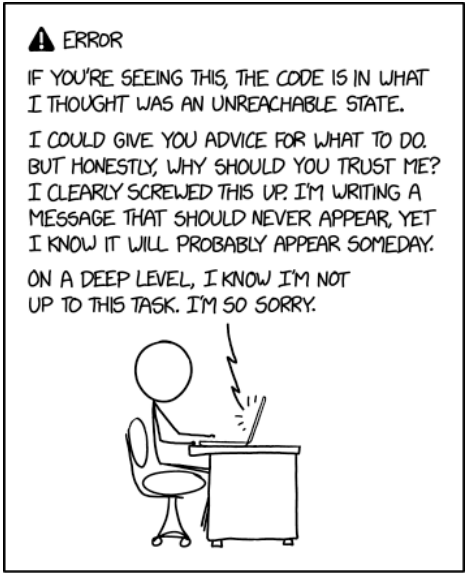
"Because of their previous experiences with buggy, unreliable software, users sometimes have low expectations of software quality. They are not surprised when their software fails. When a new system is installed, users may tolerate failures because the benefits of use outweigh the costs of failure recovery. However, as a software product becomes more established, users expect it to become more reliable. [...] If a software product or app is very cheap, users may be willing to tolerate a lower level of reliability. [...] Customers may be willing to accept the software, irrespective of problems, because the costs of not using the software are greater than the costs of working around the problems."

Product Quality [ISO/IEC 25010]



Quality in Use [ISO/IEC 25010]





NEVER WRITE ERROR MESSAGES TIRED.

Software Testing

Software Testing **[Sommerville]**

"Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use."

Validation Testing **[Sommerville]**

"Demonstrate to the developer and the customer that the software meets its requirements."

Defect Testing **[Sommerville]**

"Find inputs or input sequences where the behavior of the software is incorrect, undesirable, or does not conform to its specification."

V&V **[Boehm 1979]**

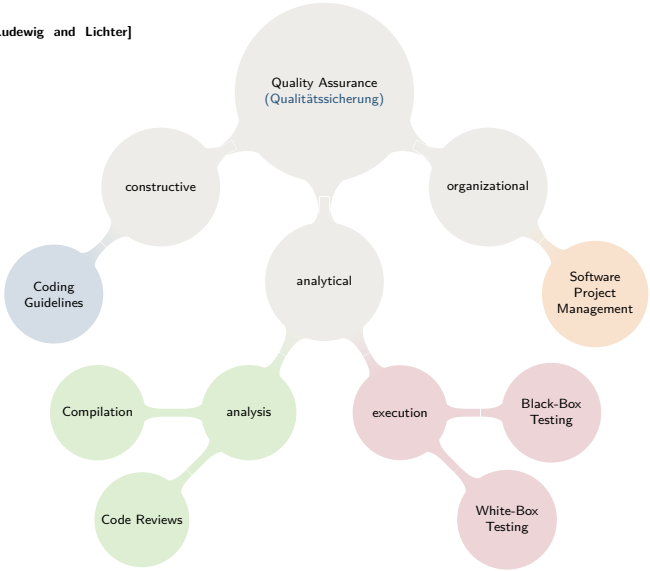
"**Validation**: Are we building the right product?
Verification: Are we building the product right?"

Stages of Testing **[Sommerville]**

1. "**Development testing**, where the system is tested during development to discover bugs and defects"
2. "**Release testing**, where a separate testing team tests a complete version of the system before it is released to users"
3. "**User testing**, where users or potential users of a system test the system in their own environment"

"In **manual testing**, a tester runs the program with some test data and compares the results to their expectations. [...] In **automated testing**, the tests are encoded in a program that is run each time the system under development is to be tested." [Sommerville]

Quality Assurance [Ludewig and Lichter]



Code Reviews

- Idea: improve quality by asking other programmers for feedback
- Typically applied with quality checklist
- Quality criteria: functionality, comprehensibility, maintainability, coding guidelines, design patterns, ...
- Reviewer selection: based on familiarity with code, availability, expertise

- Cannot be done by yourself
- Reviewers need programming experience and knowledge of the code (mutual feedback)
- Feedback should be timely and constructive
- Only changes reviewed, not too many

Code Reviews on Github

Collaborative code review.

Code review is an essential part of the GitHub workflow. After creating a branch and making one or more commits, a Pull Request starts the conversation around the proposed changes. Additional commits are commonly added based on feedback before merging the branch.

Modulo in Different Programming Languages

Overview by Torsten Curdt:

Language	13 mod 3	-13 mod 3	13 mod -3	-13 mod -3
C	1	-1	1	-1
Go	1	-1	1	-1
PHP	1	-1	1	-1
Rust	1	-1	1	-1
Scala	1	-1	1	-1
Java	1	-1	1	-1
Javascript	1	-1	1	-1
Ruby	1	2	-2	-1
Python	1	2	-2	-1

Perform Example Code Review

```java
/**
 * Computes the remainder of the
 * Euclidean devision of a by b. In
 * contrast to the Java version a % b,
 * the output will always be positive.
 * Throws ArithmeticException when b is
 * equal to 0.
 *
 * @param a dividend
 * @param b divisor != 0
 * @return remainder r with 0 <= r < b
 */
public static int modulo(int a, int b) {
    if (b < 0) {                // 1
        b *= -1;                // 2
    }
    int m = a;                  // 3
    while (m < 0 || m > b) {    // 4
        m += m < 0 ? b : -b;    // 5
    }
    return m;                   // 6
}
```

Quality Assurance

Lessons Learned

- What is quality, quality assurance, product quality, quality in use?
- Terms: software testing, validation/defect testing, validation/verification, development/release/user testing, manual/automated testing, constructive/analytical/organizational quality assurance
- Code reviews
- Further Reading: Sommerville, Chapter 8 Software Testing
  Ludewig and Lichter, Chapter 5 (Software-Qualität) and Chapter 13 (Software-Qualitätssicherung und -Prüfung)

Practice

- Think of a change to the modulo method (cf. previous slide) that would be classified as error or warning by the Java compiler
- Do a code review of the modulo method (i.e., comment on possible improvements)
- Submit your results in Moodle and inspect other submissions:
  `https://moodle.uni-ulm.de/mod/moodleoverflow/discussion.php?d=2127`

Lecture Contents

White-Box Testing

Edsger W. Dijkstra (1972)

"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence."

Test Cases (Testfälle)

Systematic Test **[Ludewig and Lichter]**

A systematic test is a test, in which
1. the setup is defined,
2. the inputs are chosen systematically,
3. the results are documented and evaluated by criteria being defined prior to the test.

Test Case **[Ludewig and Lichter]**

In a test, a number of test cases are executed, whereas each test case consists **input values** for a single execution and **expected outputs**. An **exhaustive test** refers a test in which the test cases exercise all the possible inputs.

Exhaustive Testing in Practice?

```
boolean a, b, c;
int i, j;
```

`bla(a,b,c)` has $2^3 = 8$ possible inputs
`blub(i,j)` has $(2^{32})^2 = 2^{64} \approx 10^{19}$ inputs

- assuming 10^9 test cases can be executed in 1 second (cf. CPU with more than 1 GHz)
- exhaustive test of `blub` takes $\approx$ 585 years
- testing for a day would cover less than 0.0005 % of the inputs

How to test thousands of such methods several times a day?

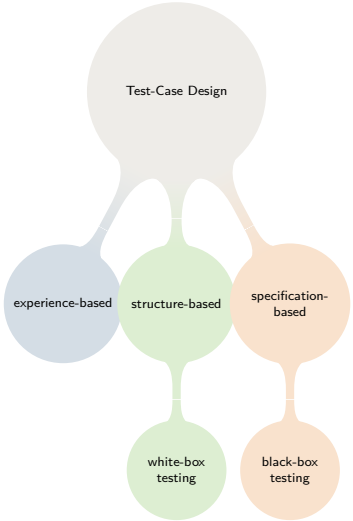
Test-Case Design (Testfallentwurf)

Goal **[Ludewig and Lichter]**

Detect a large number of failures with a low number of test cases. A test case (execution) is **positive**, if it detects a failure, and **successful** if it detects an unknown failure.

An ideal test case is ... **[Ludewig and Lichter]**

- representative: represents a large number of feasible test cases
- failure sensitive: has a high probability to detect a failure
- non-redundant: does not check what other test cases already check

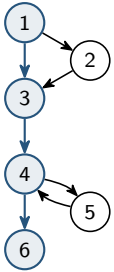


White-Box Testing (Strukturtest)

White-Box Testing **[Ludewig and Lichter]**

- inner structure of test object is used
- idea: coverage of structural elements
- code translated into control flow graph
- specific test case (concrete inputs)
  derived from logical test case (conditions)
  derived from path in control flow graph

```
public static int modulo(int a, int b) {
    if (b < 0) {                // 1
        b *= -1;                // 2
    }
    int m = a;                  // 3
    while (m < 0 || m > b) {    // 4
        m += m < 0 ? b : -b;    // 5
    }
    return m;                   // 6
}
```



Coverage Criteria (Überdeckungskriterien)

Coverage Criteria **[Ludewig and Lichter]**

1. statement coverage (Anweisungsüberdeck.): all statements are executed for at least one test case
2. branching coverage (Zweigüberdeckung): statement coverage and for each branching statement all branches have been exercised
3. term coverage (Termüberdeckung): branching coverage and terms (n) used in a branching statement are combined exhaustively (2^n) (simplified)

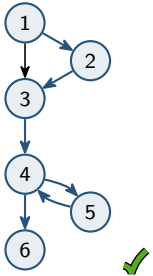
In Practice

100% statement coverage not feasible in presence of dead code or some unreachable error handling

100% term coverage not feasible for certain dependencies between choices: `even() || odd()`

Statement Coverage for Modulo Example

```
public static int modulo(int a, int b) {
    if (b < 0) {                    // 1
        b *= -1;                    // 2
    }
    int m = a;                      // 3
    while (m < 0 || m > b) {        // 4
        m += m < 0 ? b : -b;        // 5
    }
    return m;                       // 6
}
```

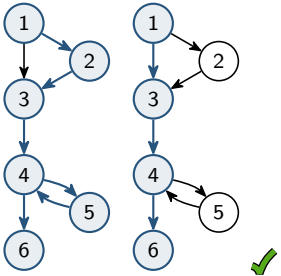


First Test Case

path:	1, 2, 3, 4, 5, 4, 6
logical test case:	$b < 0 \wedge a > -b \wedge a + b \leq -b$
specific test case:	$a = 5$, $b = -3$
expected result:	$m = 2$

Branching Coverage for Modulo Example

```
public static int modulo(int a, int b) {
    if (b < 0) {                    // 1
        b *= -1;                    // 2
    }
    int m = a;                      // 3
    while (m < 0 || m > b) {        // 4
        m += m < 0 ? b : -b;        // 5
    }
    return m;                       // 6
}
```



First Test Case

path: 1, 2, 3, 4, 5, 4, 6

logical test case: $b < 0 \wedge a > -b \wedge a + b \leq -b$

specific test case: $a = 5$, $b = -3$ and $m = 2$

Second Test Case

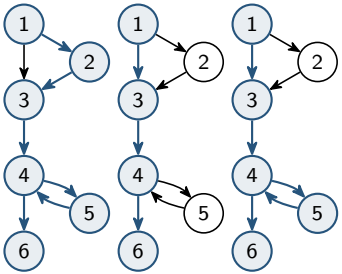
path: 1, 3, 4, 6

logical test case: $b \geq 0 \wedge 0 \leq a \leq b$

specific test case: $a = 0$, $b = 5$ and $m = 0$

Term Coverage for Modulo Example

```
public static int modulo(int a, int b) {
    if (b < 0) {                    // 1
        b *= -1;                    // 2
    }
    int m = a;                      // 3
    while (m < 0 || m > b) {        // 4
        m += m < 0 ? b : -b;        // 5
    }
    return m;                       // 6
}
```



logical test case: $b < 0 \wedge a > -b \wedge a + b \leq -b$
specific test case: $a = 5$, $b = -3$ and $m = 2$ ✓

$\neg(m < 0)$ and $m > b$ ✓

logical test case: $b \geq 0 \wedge 0 \leq a \leq b$
specific test case: $a = 0$, $b = 5$ and $m = 0$ ✓

$\neg(m < 0)$ and $\neg(m > b)$ ✓

$m < 0$ and $m > b$ impossible* ✓

path: 1, 3, 4, 5, 4, 6
logical test case: $b \geq 0 \wedge a < 0 \wedge 0 \leq a + b \leq b$
specific test case: $a = -2$, $b = 5$ and $m = 3$ ✓

$m < 0$ and $\neg(m > b)$ ✓

*see third part of the lecture

White-Box Testing

Lessons Learned

- Systematic test, exhaustive testing
- Test case: representative, failure sensitive, non-redundant
- Test-case design: experience-/structure-/specification-based
- Coverage in white-box testing: statement/branching/term coverage
- Further Reading: Ludewig and Lichter, Chapter 19 (Programmtest)

Practice

- Watch screencast on how to do white-box testing with JUnit
- Implement a subtraction in the calculator and update the white-box tests accordingly: `https://github.com/tthuem/2020WS-SWT-Calculator/tree/testinglecturev1`
- Submit your code changes and tests in Moodle: `https://moodle.uni-ulm.de/mod/moodleoverflow/discussion.php?d=2128`

Lecture Contents

Black-Box Testing

Black-Box Testing [Ludewig and Lichter]

Motivation

- source code not always available (e.g., outsourced components, obfuscated code)
- specific test cases derived from logical ones using arbitrary values
- specification not incorporated so far (only for expected results)
- invalid inputs not tested
- errors are not equally distributed

Black-Box Testing (Funktionstest)

- test-case design based on specification
- source code and its inner structure is ignored (assumed as a black-box)

1. Equivalence Class Testing

- idea: classify inputs and outputs into equivalence classes
- assumption: equivalent test cases detect the same faults, one test case is sufficient

2. Boundary Testing

- extension of equivalence class testing
- goal: use experience (e.g., off-by-one errors)
- for every equivalence class: consider smallest, typical, and largest value

In Practice

- often combinations of white-box and black-box testing
- more techniques with requirements or design

Equivalence Class Testing

JavaDoc Specification for Modulo Example

```
/**
 * Computes the remainder of the
 * Euclidean devision of a by b. In
 * contrast to the Java version a % b,
 * the output will always be positive.
 * Throws ArithmeticException when b is
 * equal to 0.
 *
 * @param a dividend
 * @param b divisor != 0
 * @return remainder r with 0 <= r < b
 */
public static int modulo(int a, int b) {
```

Equivalence Classes

- input a: $a < 0$, $a \geq 0$
- input b: $b < 0$, $b \geq 0$
- output: $m = 0$, $m > 0$, exception

Test Cases

	TC1	TC2	TC3
$a < 0$	X		
$a \geq 0$		X	X
$b < 0$	X		
$b > 0$		X	
$b = 0$			X
$m = 0$	X		
$m > 0$		X	
exception			X
input a	-3	1	2
input b	-3	2	0
expected output	0	1	exception
result	0 ✓	1 ✓	timeout ✗

Boundary Testing

Test Cases

	TC1	TC2	TC3	TC4	TC5	TC6	TC7
$a < 0$	X			min	max		
$a \geq 0$		X	X			min	max
$b < 0$	X			max		min	
$b > 0$		X			max		min
$b = 0$			X				
$m = 0$	X			X		X	X
$m > 0$		min			max		
exception			X				
input a	-3	1	2	minInt	-1	0	maxInt
input b	-3	2	0	-1	maxInt	minInt	1
expected output	0	1	exception	0	maxInt-1	0	0
result	0 ✓	1 ✓	timeout ✗	0 ✓	maxInt-1 ✓	timeout ✗	1 ✗

Detected Faults in Modulo Example

```
public static int modulo(int a, int b) {
    if (b < 0) {                // 1
        b *= -1;                // 2
    }
    int m = a;                  // 3
    while (m < 0 || m > b) {    // 4
        m += m < 0 ? b : -b;    // 5
    }
    return m;                   // 6
}
```

✖ TC3: infinite loop for $b = 0$, missing exception compared to JavaDoc

✖ TC6: b remains negative as `-Integer.MIN_VALUE == Integer.MIN_VALUE` and the loop condition is fulfilled for any integer

✖ TC7: indicates that $m > b$ in the loop condition should be fixed to $m \geq b$

Improved Modulo Example

```
/**
 * Computes the remainder of the
 * Euclidean devision of a by b. In
 * contrast to the Java version a % b,
 * the output will always be positive.
 * Throws ArithmeticException when b is
 * equal to 0.
 *
 * @param a dividend
 * @param b divisor != 0
 * @return remainder r with 0 <= r < b
 */
public static int modulo(int a, int b) {
    int m = a % b;
    return m < 0 ? m + b : m;
}
```

Passes All Test Cases

✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓

Reasons for Positive Test Cases

Reasons for Positive Test Cases

- actual fault
- wrong test case (input and expected results do not match)
- interaction with other programs/libraries
- fault in the compiler
- fault in the operating system / device drivers
- fault in the hardware or hardware defect
- not enough memory
- does not halt (cf. undecidability of the halting problem)
- bitflip due to cosmic ray
- ...

Brian Kernighan (1978)

"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?"

Black-Box Testing

Lessons Learned

- Black-box testing: equivalence class testing and boundary testing
- Even systematic testing cannot ensure finding all faults
- Further Reading: Ludewig and Lichter, Chapter 19 (Programmtest)

Practice

- Watch screencast on how to do black-box testing with JUnit
- Extend the black-box tests of the calculator for the subtraction (or another operation): `https://github.com/tthuem/2020WS-SWT-Calculator/tree/testinglecturev2`
- Submit your changed code and tests in Moodle: `https://moodle.uni-ulm.de/mod/moodleoverflow/discussion.php?d=2129`

Lecture Contents