

Introduction to Software Engineering

Software Testing & Unit Tests

- Resources
 - *Ian Sommerville*
Software Engineering 8th Edition
Addison Wesley 2007
 - *Robert v. Binder*
Testing Object-Oriented Systems - Models, Patterns, and Tools
Addison Wesley 2000
 - *Peter Liggesmeyer*
Software-Qualität
Spektrum 2002



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Software Testing



TECHNISCHE
UNIVERSITÄT
DARMSTADT

|| Validation

“Are we *building the right product?*”

Verification

“Are we *building the product right?*”

Ian Sommerville

Software Engineering 8th Edition; Addison Wesley 2007

Two *complementary* approaches for *verification* and *validation* (V&V) can be distinguished.

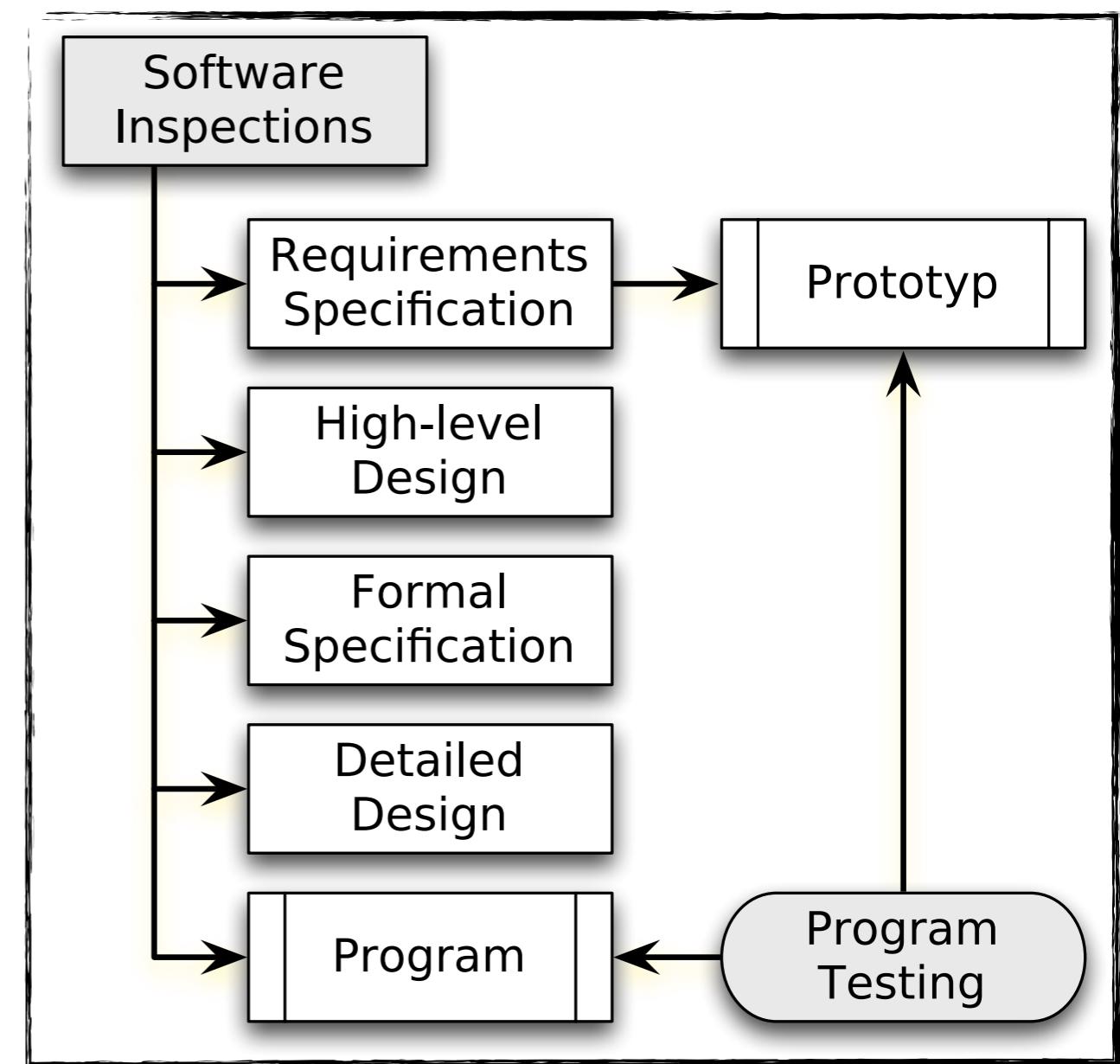
- **Software Inspections or Peer Reviews**

(Static Technique)

“Software inspections” can be done at all stages of the process.

- **Software Testing**

(Dynamic Technique)



Software inspections check the correspondence between a program and its specification.

Software Inspections - Static Technique | 5

- Some techniques
 - **Program inspections**
The goal is to [find program defects, standards violations, poor code](#) rather than to consider broader design issues; it is usually carried out by a team and the members systematically analyze the code.
An inspection is usually driven by checklists.
(Studies have shown that an inspection of roughly 100LoC takes about one person-day of effort.)
- ...

Software inspections check the correspondence between a program and its specification.

Software Inspections - Static Technique | 6

- Some techniques
- ...
- **Automated source code analysis**
 - Includes - among others - control flow analysis, data use / flow analysis, information flow analysis and path analysis.
 - Static analyses draw attention to anomalies.
- ...

FindBugs™ – Find Bugs in Java Programs

http://findbugs.sourceforge.net/index.html

Google

FindBugs™ because it's easy

Docs and Info

- Demo and data
- Users and supporters
- FindBugs blog
- Fact sheet
- Manual
- Manual(ja/日本語)
- FAQ
- Bug descriptions
- Mailing lists
- Documents and Publications
- Links

Downloads

FindBugs Swag

Development

- Open bugs
- Reporting bugs
- Contributing
- Dev team
- API [no frames]
- Change log

FindBugs™ - Find Bugs in Java Programs

This is the web page for FindBugs, a program which uses static analysis to look for bugs in Java code. It is free software, distributed under the terms of the [Lesser GNU Public License](#). The name FindBugs™ and the [FindBugs logo](#) are trademarked by [The University of Maryland](#). As of July, 2008, FindBugs has been downloaded more than 700,000 times.

FindBugs requires JRE (or JDK) 1.5.0 or later to run. However, it can analyze programs compiled for any version of Java. The current version of FindBugs is 1.3.9, released on 20:11:47 EDT, 21 August, 2009. [We are very interested in getting feedback on how to improve FindBugs.](#)

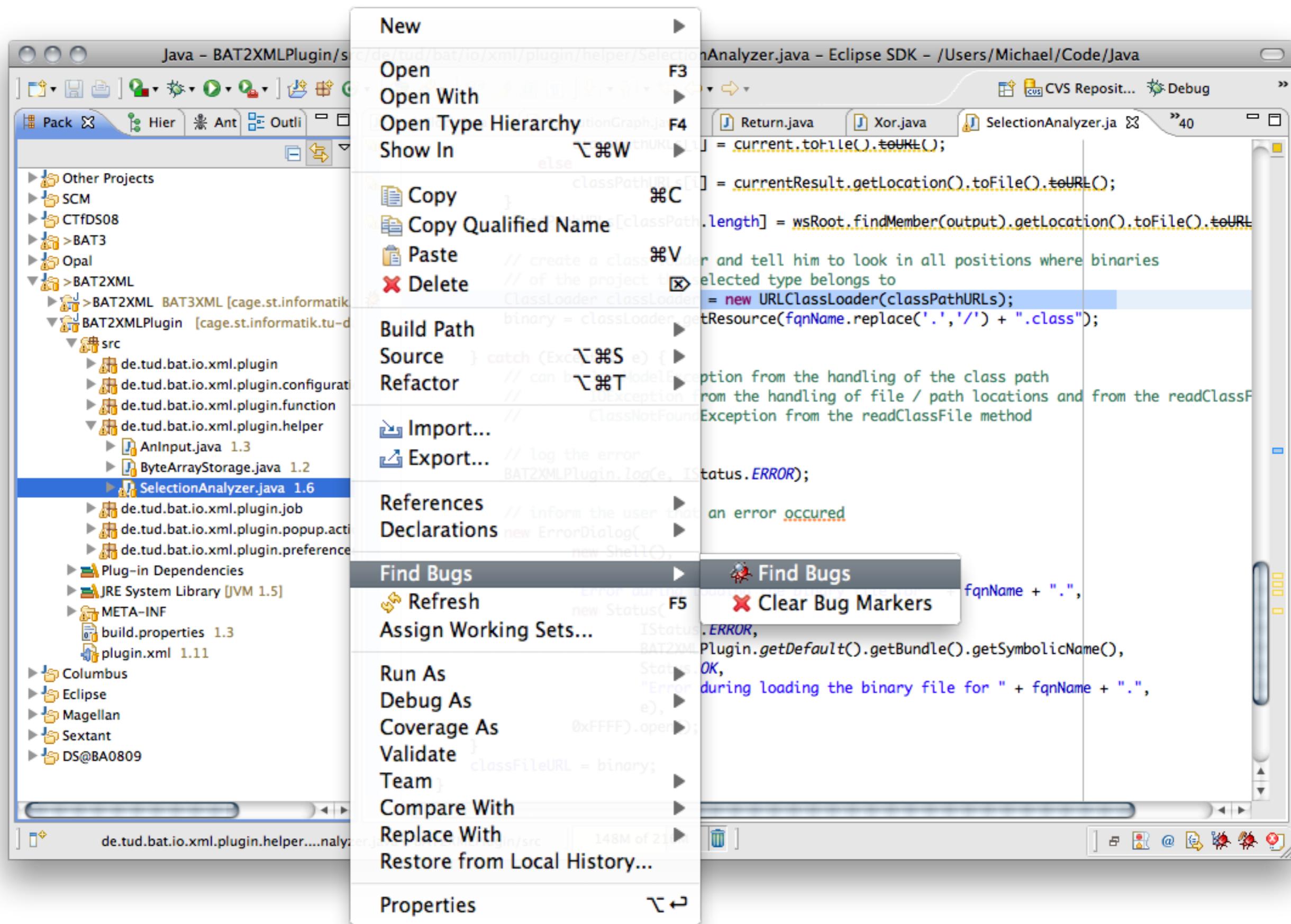
[Changes](#) | [Talks](#) | [Papers](#) | [Sponsors](#) | [Support](#)

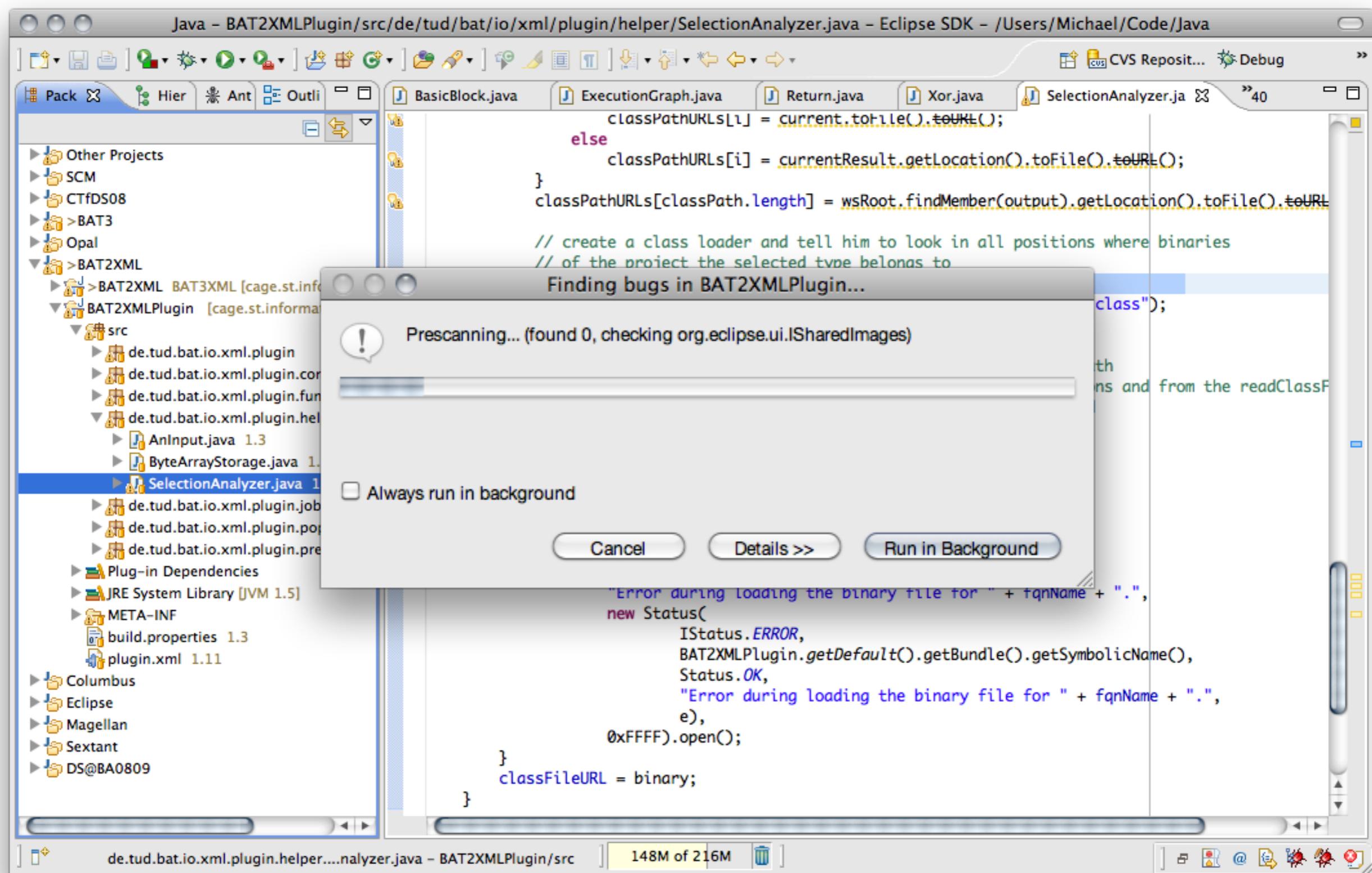
New

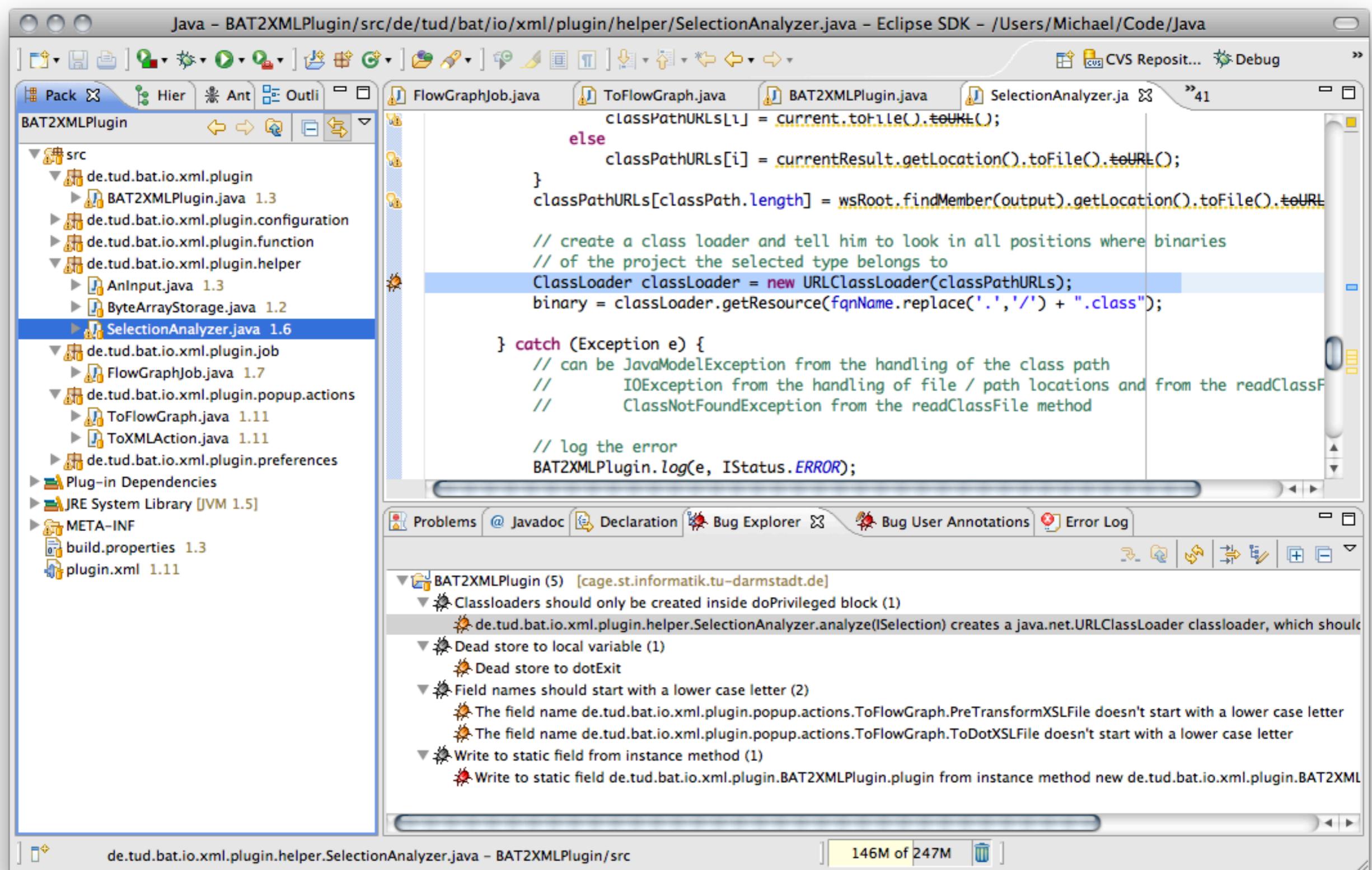
- **JavaOne talk:** [Slides](#) from my JavaOne talk, Mistakes That Matter.
- **FindBugs community review:** We are previewing FindBugs community review, in which anyone can review issues in open source projects (i.e., mark issues as "must fix" or "mostly harmless"), and those reviews are automatically shared with other reviewers.

This is a pre-beta release, not ready for deployment. The implementation will be undergoing significant changes before general availability.

Initially, we are posting results for:







Software inspections check the correspondence between a program and its specification.

Software Inspections - Static Technique | 11

- Some techniques
 - ...
- **Formal verification**
Formal verification can guarantee *the absence of specific bugs*. E.g., to guarantee that a program does not contain dead locks, race conditions or buffer overflows.

Software inspections check the correspondence between a program and its specification.

Software Inspections - Static Technique | 12

Software inspections do not demonstrate that the software is useful.

Software testing refers to running an implementation of the software with test data to discover program defects.

Software Testing - Dynamic Testing | 13

- **Validation testing**

Intended to show that the software is what the customer wants

(Basically, there should be a test case for every requirement.)

- **Defect testing**

Intended to reveal defects

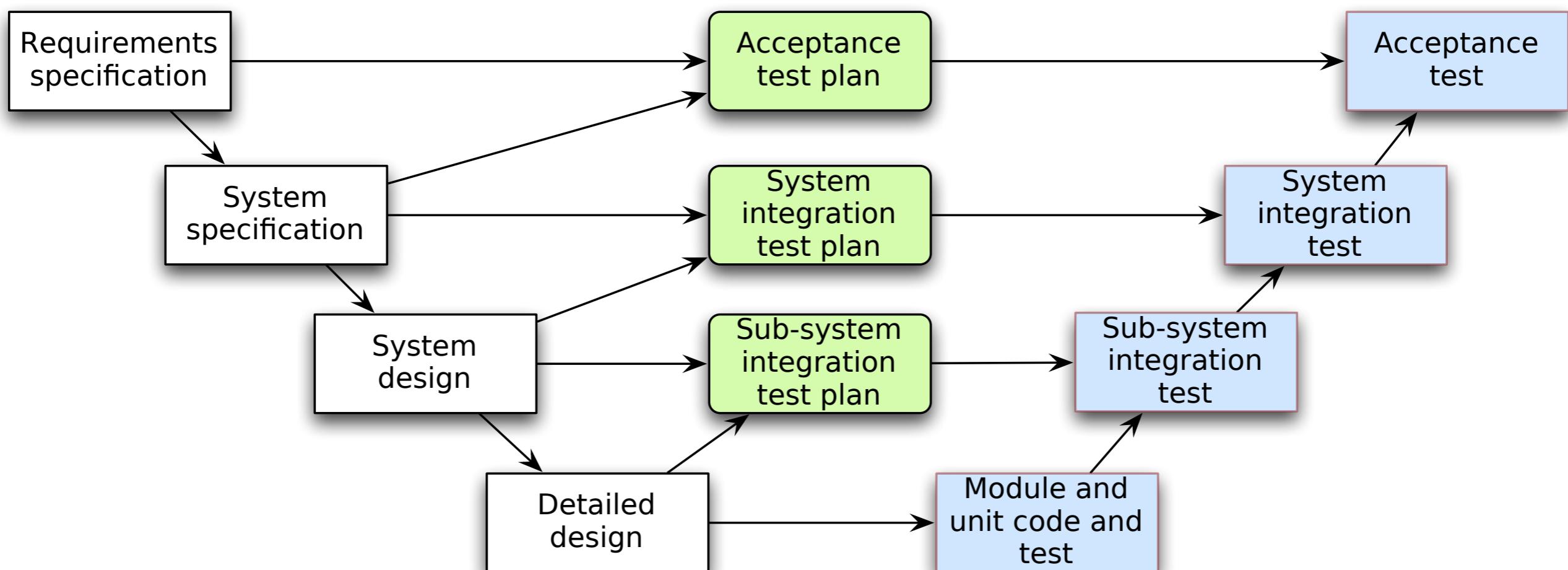
- **(Defect) Testing is...**

- **fault directed** when the intent is to reveal faults
- **conformance directed** when the intent is to demonstrate conformance to required capabilities

No Strict Separation

Test plans set out the testing schedule and procedures; they establish standards for the testing process. They evolve during the development process.

- V&V is expensive; sometimes half of the development budget is spent on V&V



The scope of a test is the collection of software components to be verified.

Software Testing - Scope of Tests | 15

- **Unit tests**

(dt. Modultest)

Comprises a relatively small executable; e.g., a single object

- **Integration test**

Complete (sub)system. Interfaces among units are exercised to show that the units are collectively operable

- **System test**

A complete integrated application. Categorized by the kind of conformance they seek to establish: functional, performance, stress or load

“Testing can only show the presence of errors, not their absence.



E. Dijkstra

The design of tests is a multi-step process.

Software Testing - Test Design | 17

1. Identify, model and analyze the responsibilities of the system under test (SUT)
(E.g., use pre- and postconditions identified in use cases as input.)
2. Design test cases based on this external perspective
3. Add test cases based on code analysis, suspicions, and heuristics
4. Develop expected results for each test case or choose an approach to evaluate the pass / no pass status of each test case

After the test design a test automation system (TAS) needs to be developed.

Software Testing - Test Automation System | 18

A test automation system will...

- start the implementation under test (IUT)
- set up its environment
- bring it to the required pretest state
- apply the test inputs
- evaluate the resulting output and state

The goal of the test execution is to establish that the implementation under test (IUT) is minimally operational by exercising the interfaces between its parts.

Software Testing - Goal of Test Execution | 19

To establish the goal...

1. execute the test suite; the result of each test is evaluated as pass or no pass
2. use a coverage tool to instrument the implementation under test; rerun the test suite and evaluate the reported coverage
3. if necessary, develop additional tests to exercise uncovered code
4. stop testing when the coverage goal is met; all tests pass ("Exhaustive" testing is not possible!)

Test Point

(dt. Testdatum (Prüfpunkt))

Test Point | 20

- A test point is a specific value for...
 - test case input
 - a state variable
- The test point is selected from a domain; the domain is the set of values that input or state variables may take
- Heuristics for test point selection:
 - Equivalence Classes
 - Boundary Value Analysis
 - Special Values Testing

Test Case

(dt. Testfall)

- Test cases specify:
 - pretest state of the implementation under test (IUT)
 - test inputs / conditions
 - expected results

Test Suite

Software Testing - Terminology | 22

- A test suite is a collection of test cases

Test Run

(dt. Testlauf)

- A test run is the execution (with results) of a test suite
- The IUT produces actual results when a test case is applied to it; a test whose actual results are the same as the expected results is said to pass

Test Driver & Test Harness/Automated Test Framework

Software Testing - Terminology | 24

- Test driver is a class or utility program that applies test cases to an IUT
- Test harness is a system of test drivers and other tools to support test execution

Failures, Errors & Bugs

Failure = dt. Defekt(, Fehlschlag)

Fault = dt. Mangel

Error = dt. Fehler

- A **failure** is the (manifested) inability of a system or component to perform a required function within specified limits
- A **software fault** is missing or incorrect code
- An **error** is a human action that produces a software fault
- Bug: error or fault.

Test Plan

- A document prepared for human use that explains a testing approach:
 - the work plan,
 - general procedures,
 - explanation of the test design,
 - ...

Testing must be based on a **fault model**.

Because the number of tests is (for practical purposes) infinite, we have to make an assumption about **where** faults are likely to be found!

Testing must be based on a **fault model**.

Two general fault models and corresponding testing strategies exist:

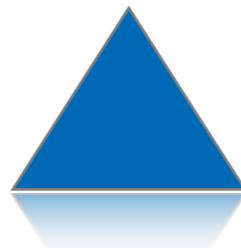
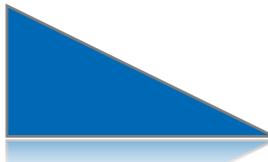
- Conformance-directed testing
 - Fault-directed testing
-



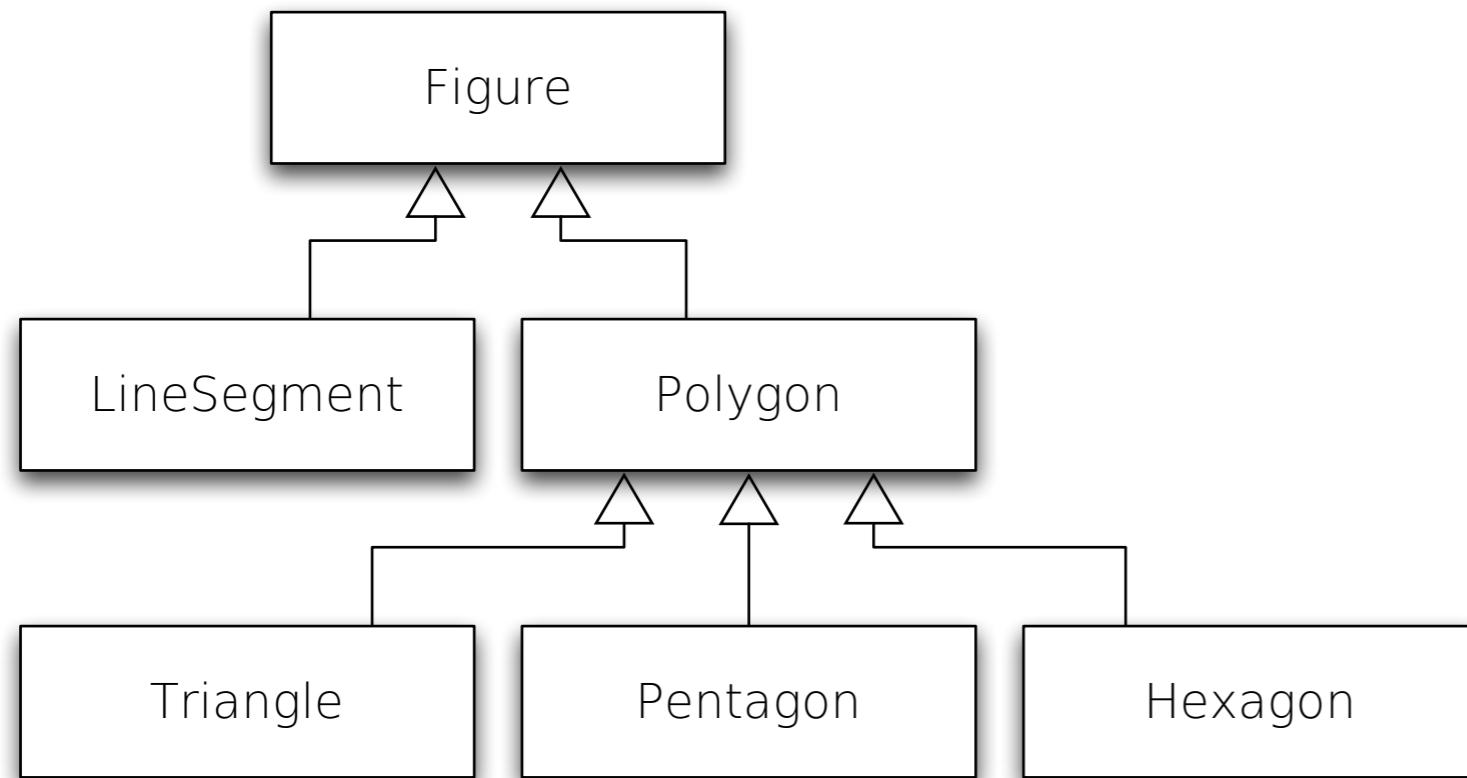
Testing has to be efficient.

Developing a Test Plan

- Devise a test plan for a program that:
 - reads three integer values,
 - which are interpreted as the length of the sides of a triangle
 - The program states whether the triangle is
 - scalene (dt. *schief*),
 - isosceles (dt. *gleichschenklig*), or
 - equilateral (dt. *gleichseitig*)
- A valid triangle must meet two conditions:
 - No side may have a length of zero
 - Each side must be shorter than the sum of all sides divided by 2



An Implementation of a Triangle



```
class Polygon extends Figure {
    abstract void draw(...);
    abstract float area();
}

class Triangle extends Polygon {
    public Triangle(...);
    public void setA(LineSegment a);
    public void setB(LineSegment b);
    public void setC(LineSegment c);
    public boolean isIsosceles();
    public boolean isScalene();
    public boolean isEquilateral();
}
```

Test Descriptions

Description	A	B	C	Expected Output
Valid scalene triangle	5	3	4	Scalene
Valid isosceles triangle	3	3	4	Isosceles
Valid equilateral triangle	3	3	3	Equilateral
First perm. of two equal sides	50	50	25	Isosceles
(Permutations of previous test case)	Isosceles
One side zero	1000	1000	0	Invalid
First perm. of two equal sides	10	5	5	Invalid
Sec. perm. of two equal sides	5	10	5	Invalid
Third perm. of two equal sides	5	5	10	Invalid
Three sides greater than zero, sum of two smallest less than the largest	8	5	2	Invalid

Test Descriptions

Description	A	B	C	Expected Output
(Permutations of previous test case)	Invalid
All sides zero	0	0	0	Invalid
One side equals the sum of the other	12	5	7	Invalid
(Permutations of previous test case)	Invalid
Three sides at maximum possible value	MAX	MAX	MAX	Equilateral
Two sides at maximum possible value	MAX	MAX	1	Isosceles
One side at maximum value	1	1	MAX	Invalid
+ Further OO related tests w.r.t. the type hierarchy etc. (e.g. are the line segments connected.)				

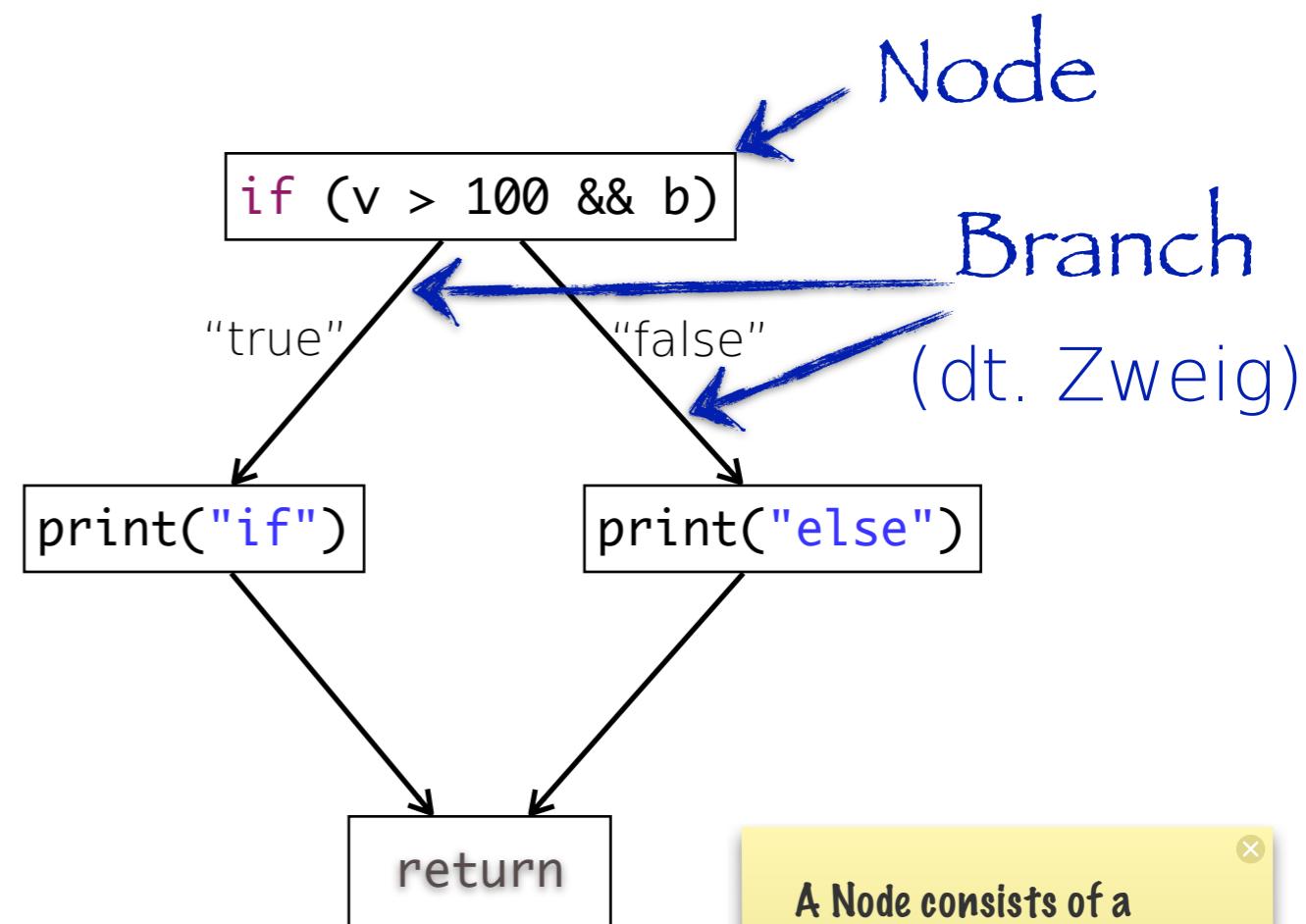
Coverage

Coverage = dt. Abdeckung

- The completeness of a test suite w.r.t. a particular test case design method is measured by coverage
- Coverage is the percentage of elements required by a test strategy

The Control-flow Graph of a Method

```
static void doThat(int v, boolean b) {  
  
    if (v > 100 && b) {  
        print("if");  
    }  
  
    else {  
        print("else");  
    }  
  
    return;  
}
```



A Node consists of a sequence of statements without any branches in or out (except of the last statement).

A branch describes a possible control-flow.

Common Method Scope Code Coverage Models

- **Statement Coverage** is achieved when all statements in a method have been executed at least once
- **Branch Coverage** is achieved when every path from a node is executed at least once by a test suite; compound predicates are treated as a single statement
- **Simple Condition Coverage** requires that each simple condition be evaluated as true and false at least once (Hence, it does not require testing all possible branches.)
- **Condition Coverage** =
Simple Condition Coverage + Branch Coverage
- **Multiple-condition Coverage** requires that all true-false combinations of simple conditions be exercised at least once

branch =dt. Verzweigung; condition =dt. Bedingung;

branch coverage =dt. Zweigüberdeckung

simple condition coverage =dt. einfache Bedingungsüberdeckung

Conditions - Exemplified

```
static void doThat(int v, boolean b) {  
    simple/atomic condition(s)  
    :  
    if (v > 100 && b) {  
        print("if");  
    }  
    else {  
        print("else");  
    }  
}
```

Here, "v > 100" is the first condition and "b" is the second condition.

In Java, simple/atomic conditions are separated by "&&" / "&" or "||"/"|" operators.

Compound Predicates - Exemplified

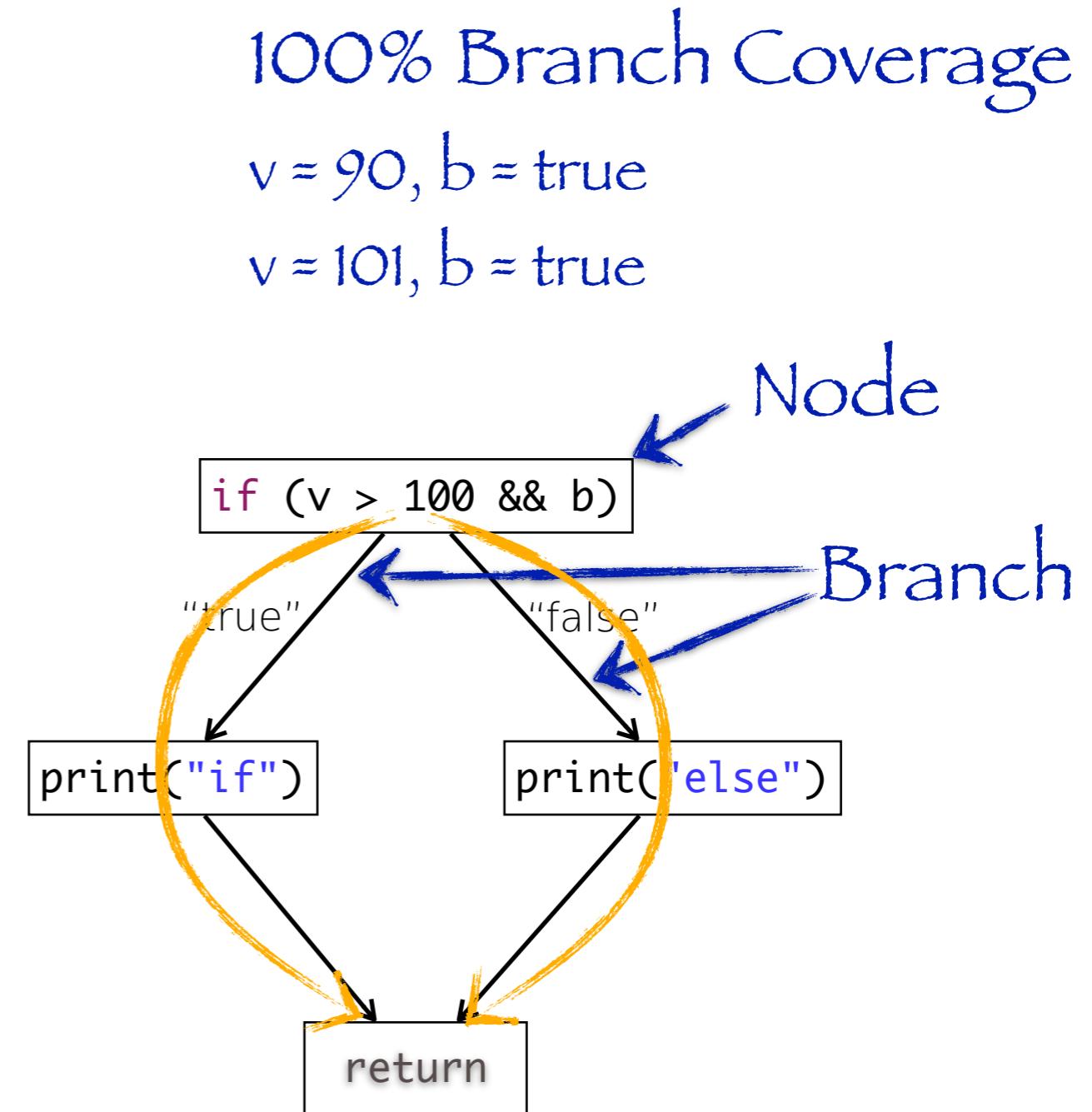
```
static void doThat(int v, boolean b) {  
    if (v > 100 && b) {  
        print("if");  
    }  
    else {  
        print("else");  
    }  
}
```

(compound) predicate (expression)

Here, "v > 100 && b" is called a predicate resp. a compound predicate. This compound predicate consists of two "simple" conditions.

Branch Coverage Exemplified

```
static void doThat(int v, boolean b) {  
    if (v > 100 && b) {  
        print("if");  
    }  
    else {  
        print("else");  
    }  
}
```



Simple Condition Coverage Exemplified

Recall: The condition is an expression that evaluates to true or false. I.e., an expression such as $\neg b$ (not b) is the condition.

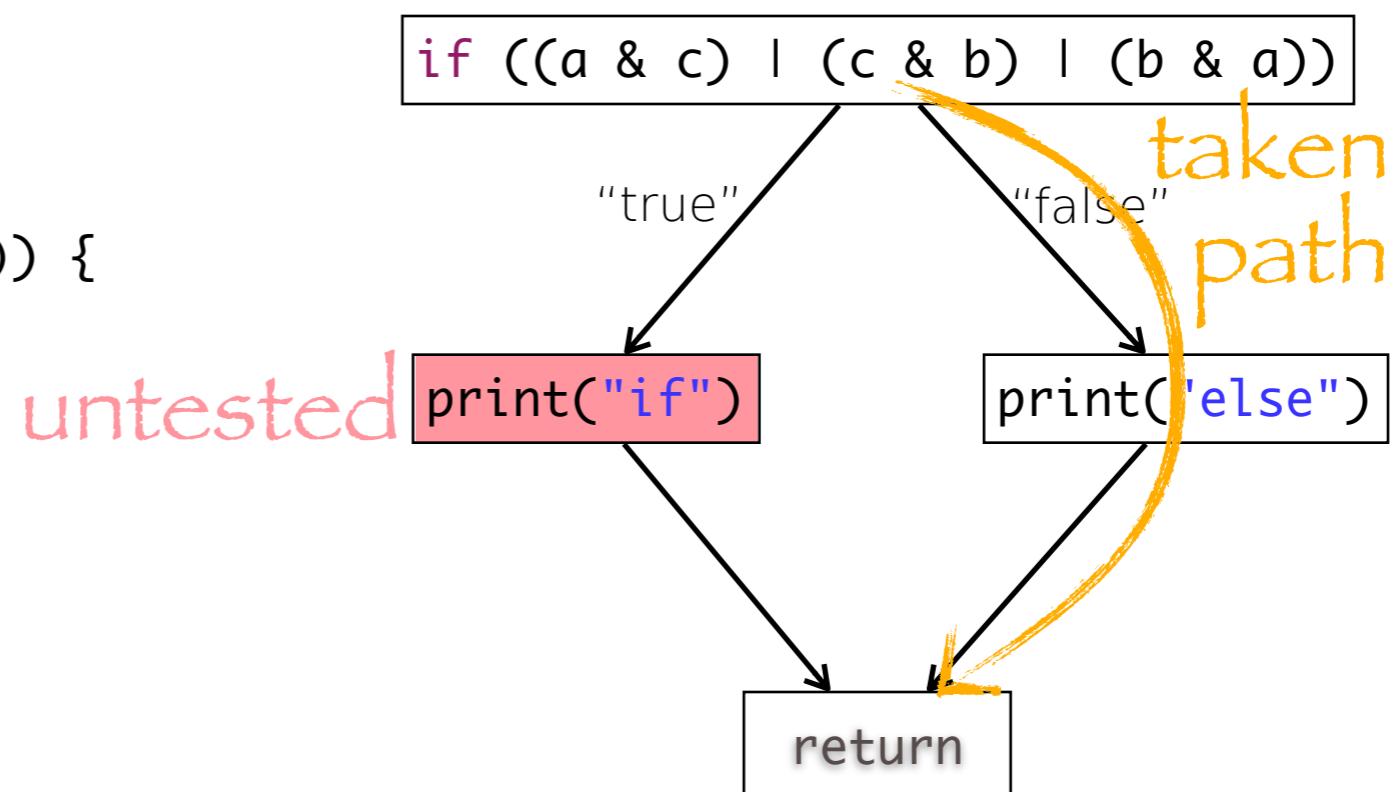
```
static void doThat(  
    boolean a,  
    boolean b,  
    boolean c) {  
  
    if ((a & c) | (c & b) | (b & a)) {  
        print("if");  
    }  
    else {  
        print("else");  
    }  
}
```

100% Simple Condition Coverage

$a = \text{true}, b = \text{false}, c = \text{false}$

$a = \text{false}, b = \text{true}, c = \text{false}$

$a = \text{false}, b = \text{false}, c = \text{true}$



(Simple) Condition Coverage Exemplified

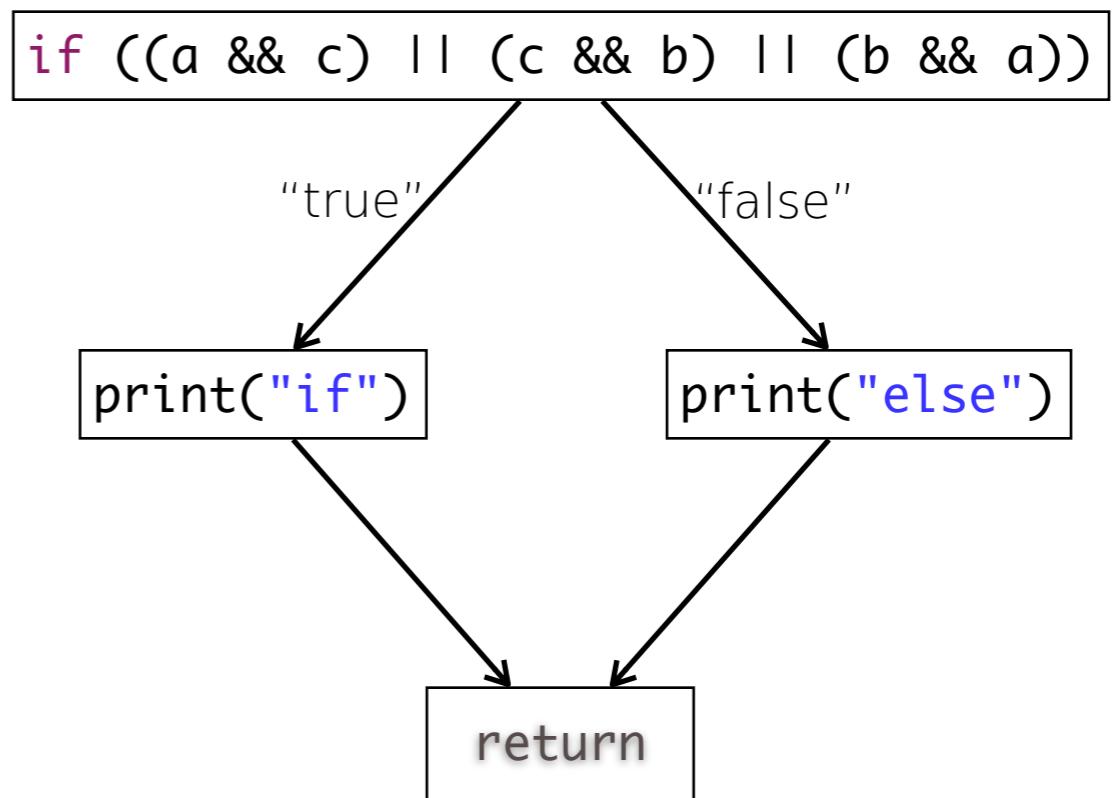
100% (Simple) Condition Coverage

a = true, c = true (b is not relevant)

a = false, c = true, b = true

a = false, c = false, b = false

```
static void doThat(  
    boolean a,  
    boolean b,  
    boolean c) {  
  
    if ((a && c) || (c && b) || (b && a)) {  
        print("if");  
    }  
    else {  
        print("else");  
    }  
}
```



Recall, if we have shortcut evaluation, simple condition coverage implies branch coverage!

Basic Block Coverage

- A **basic block** is a sequence of consecutive instructions in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end
- **Basic block coverage** is achieved if all basic blocks of a method are executed
(~~Sometimes “statement coverage” is used as a synonym for “basic block coverage” - however, we do not use these terms synonymously.~~)
(Basic blocks are sometimes called segments.)

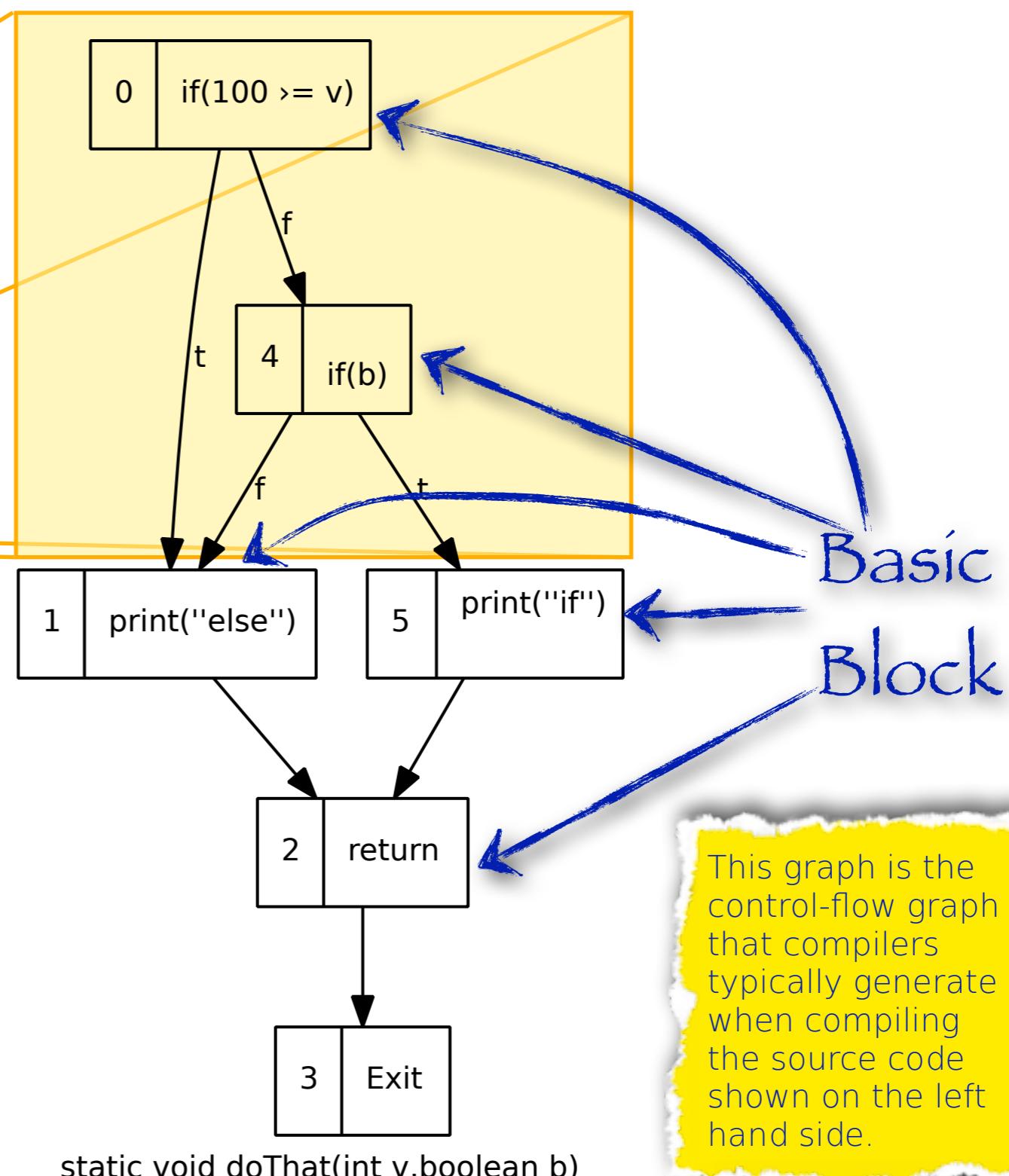
Basic Block Coverage Exemplified

100% Basic Block Coverage

$v = 90, b = \text{"not relevant"}$

$v = 101, b = \text{true}$

```
static void doThat(int v, boolean b) {  
    if (v > 100 && b) {  
        print("if");  
    } else {  
        print("else");  
    }  
}
```



This graph is the control-flow graph that compilers typically generate when compiling the source code shown on the left hand side.

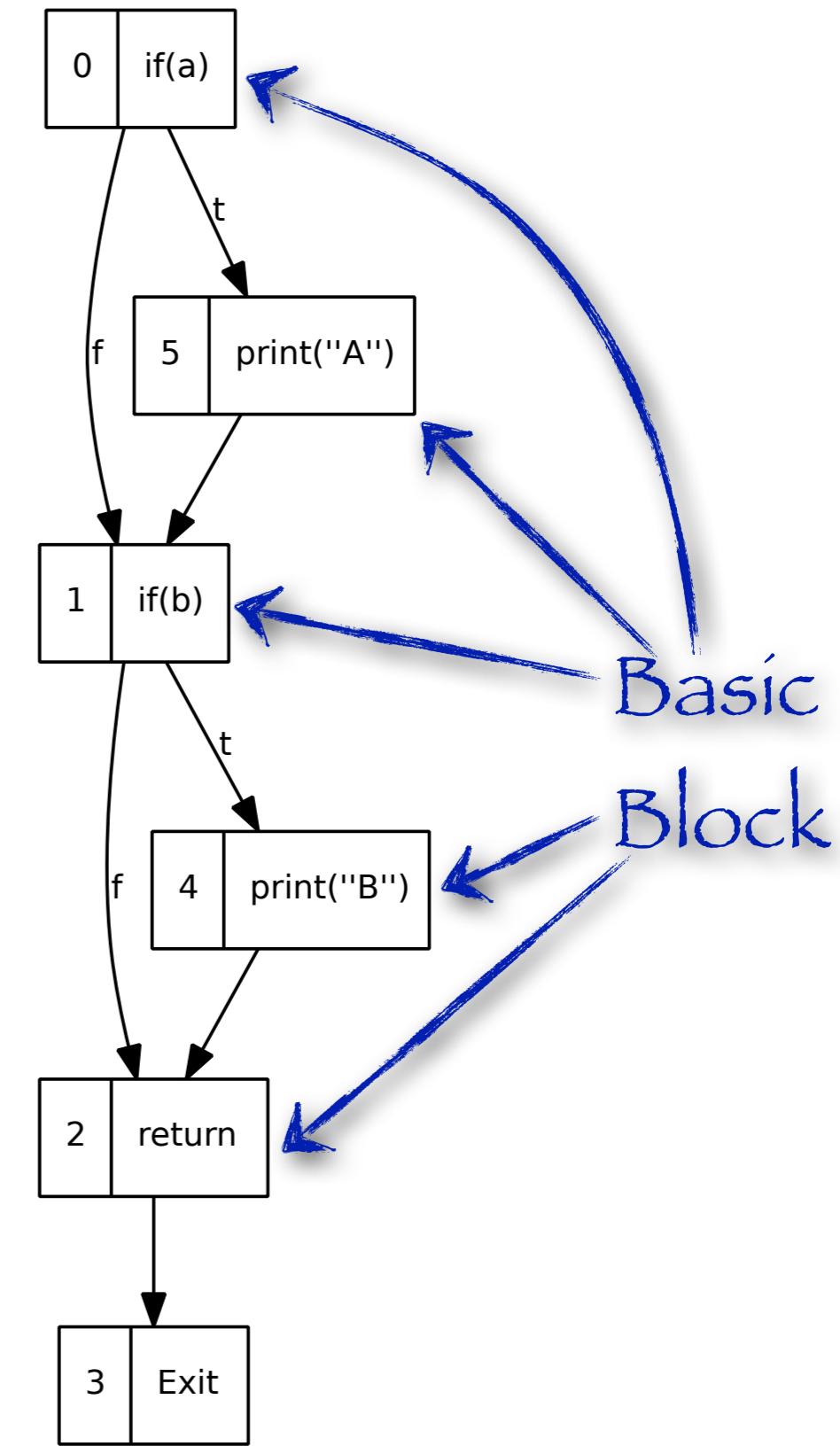
Control-flow Graph

```
static void doThis(boolean a, boolean b) {
    if (a) {
        print("A");
    }
    if (b) {
        print("B");
    }
}
```

	a	b	
Statement Coverage	TRUE	TRUE	Minimal Number of Tests to Achieve ... Coverage
Basic Block Coverage	TRUE	TRUE	
(Simple) Condition Coverage	TRUE	TRUE	
	FALSE	FALSE	
	FALSE	FALSE	
Branch Coverage	TRUE	TRUE	

Here, condition coverage can also be achieved using other test cases.(E.g. a=false; b=true and a=true; b=false.)

No case covers all possible paths!



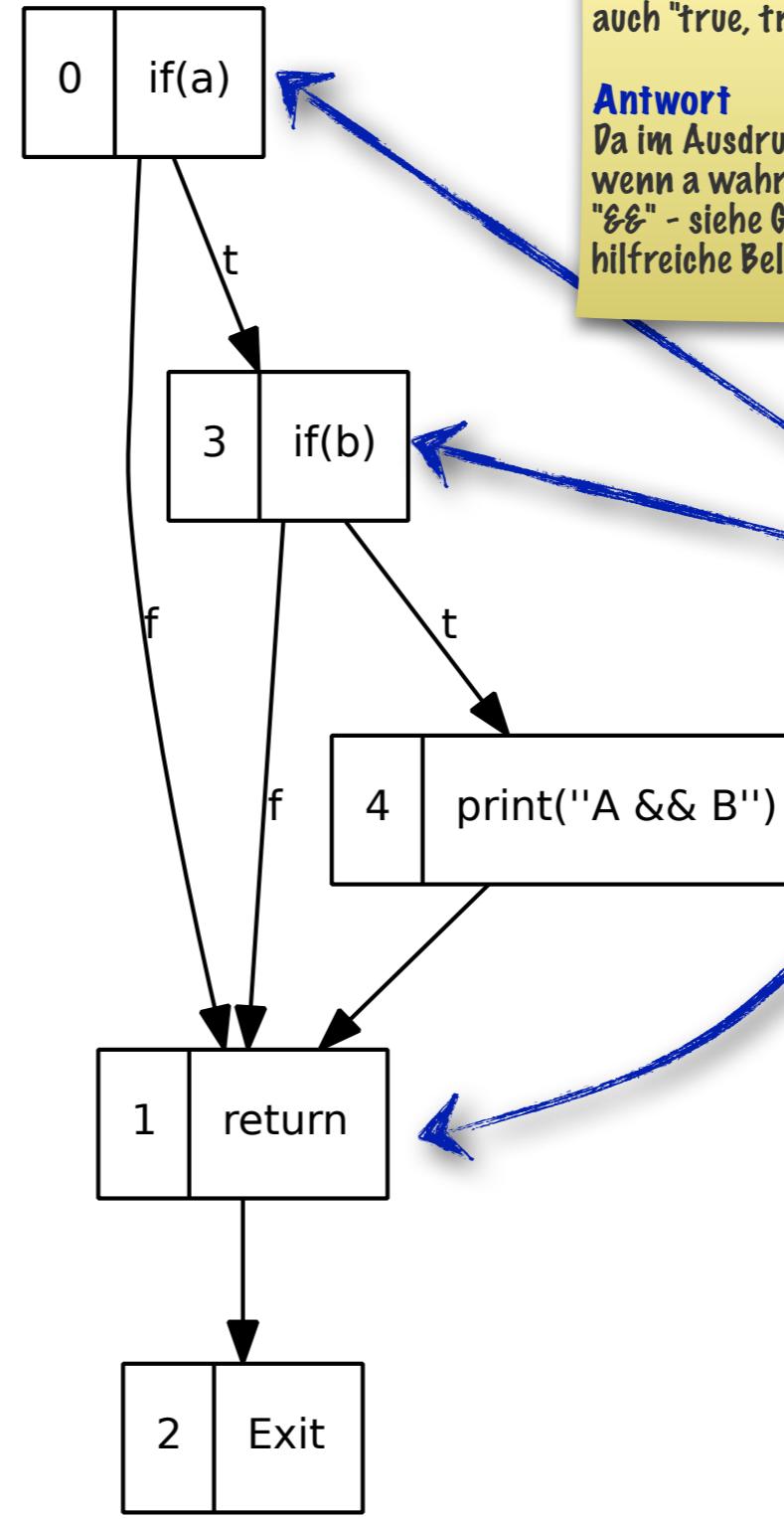
static void doThis(boolean a,boolean b)

Control-flow Graph

```
static void doThis(boolean a, boolean b) {
    if (a && b) {
        print("A && B");
    }
}
```

	a	b	
Statement Coverage	TRUE	TRUE	
Basic Block Coverage	TRUE	TRUE	
(Simple) Condition Coverage	TRUE	TRUE	
	TRUE	FALSE	
	FALSE	/	
Branch Coverage (w.r.t. the given source code)	TRUE	TRUE	
	FALSE	/	
	TRUE	TRUE	
	TRUE	FALSE	
	FALSE	/	
Multiple Condition Coverage			

Minimal Number of Tests to Achieve ... Coverage



static void doThis(boolean a, boolean b)

Frage

Würde im Falle von Condition Coverage nicht auch "true, true" und "false, false" ausreichen?

Antwort

Da im Ausdruck "a && b", "b" nur evaluiert wird wenn a wahr ist (Short-cut Evaluation von "&&" - siehe Graph) - ist "false / false" keine hilfreiche Belegung der Parameter.

Basic
Block

Frage / Antwort:

Wäre der Code:

```
if (a) {
    if (b)
        print("A && B")
    else
        print("Hello!")
}
return;
```

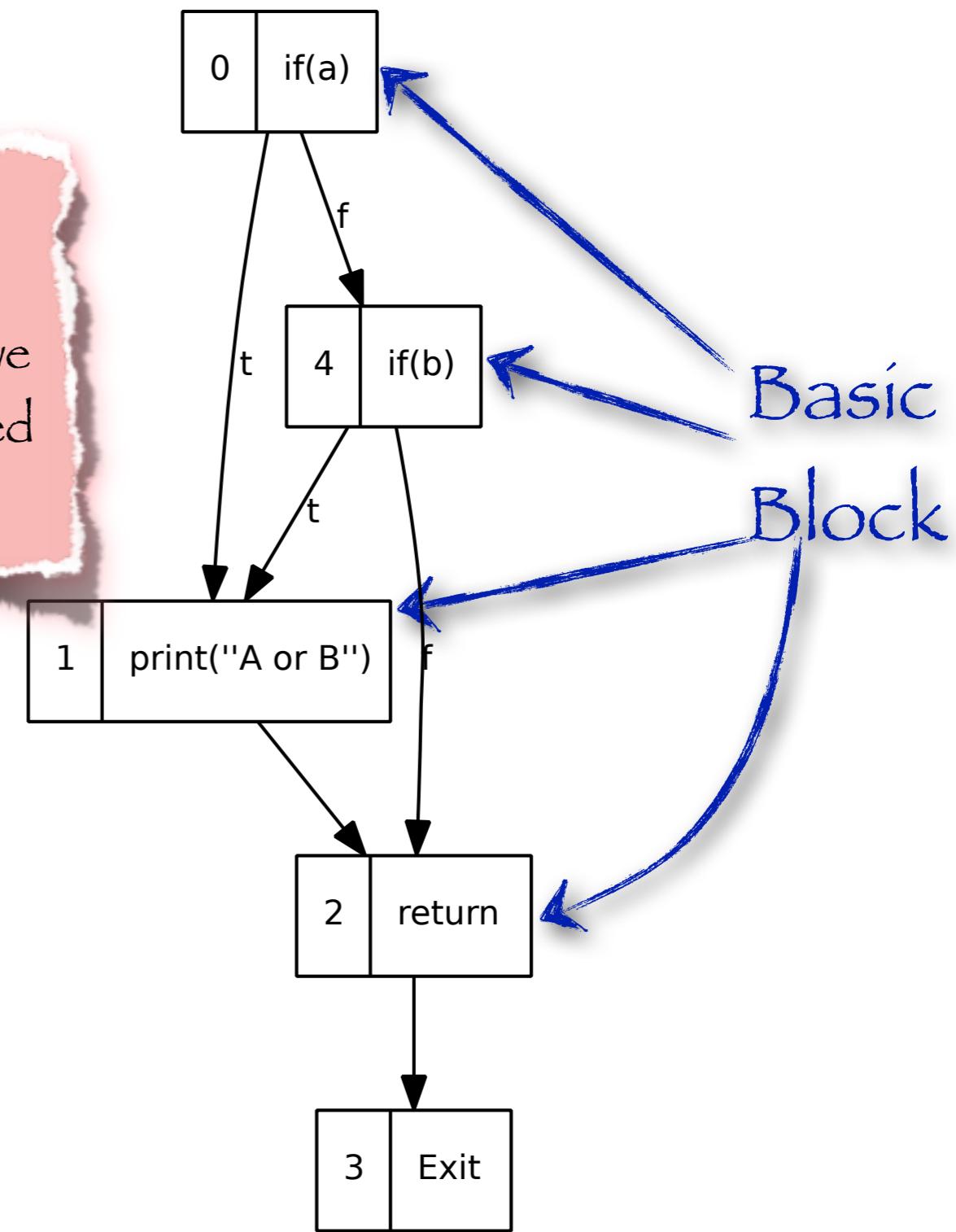
dann wäre für "Statement Coverage" folgende Testfälle notwendig: a=true; b=false und a=true; b=true. (Ebenso für Basic Block Coverage)

Control-flow Graph

```
static void doThis(boolean a, boolean b) {
    if (a || b) {
        print("A or B");
    }
}
```

We have achieved 100% statement coverage, though we have never evaluated the condition b.

	a	b	Minimal Number of Tests to Achieve ... Coverage
Statement Coverage	TRUE	/	
Basic Block Coverage	FALSE	TRUE	
(Simple) Condition Coverage	FALSE	TRUE	
	FALSE	FALSE	
Branch Coverage (w.r.t. the source code)	TRUE	/	
	TRUE	/	
	FALSE	FALSE	



static void doThis(boolean a, boolean b)

```
static long process(String[] args) throws IllegalArgumentException {
```

```
    Stack values = new Stack();
    for (int i = 0; i < args.length; i++) {
        String arg = args[i];
        try {
            long value = Long.parseLong(arg);
            values.push(value);
        } catch (NumberFormatException nfe) {
            // there is no method to test if a string is a number
            if (values.size() > 1) {
                long r = values.pop();
                long l = values.pop();
                if (arg.equals("+")) {
                    values.push(l + r);
                    continue;
                }
                if (arg.equals("*")) {
                    values.push(l * r);
                    continue;
                }
            }
            throw new IllegalArgumentException("Too few operands or operator unknown.");
        }
    }
    if (values.size() == 1) return values.pop();
    else throw new IllegalArgumentException("Too few (0) or too many (>1) operands.");
}
```

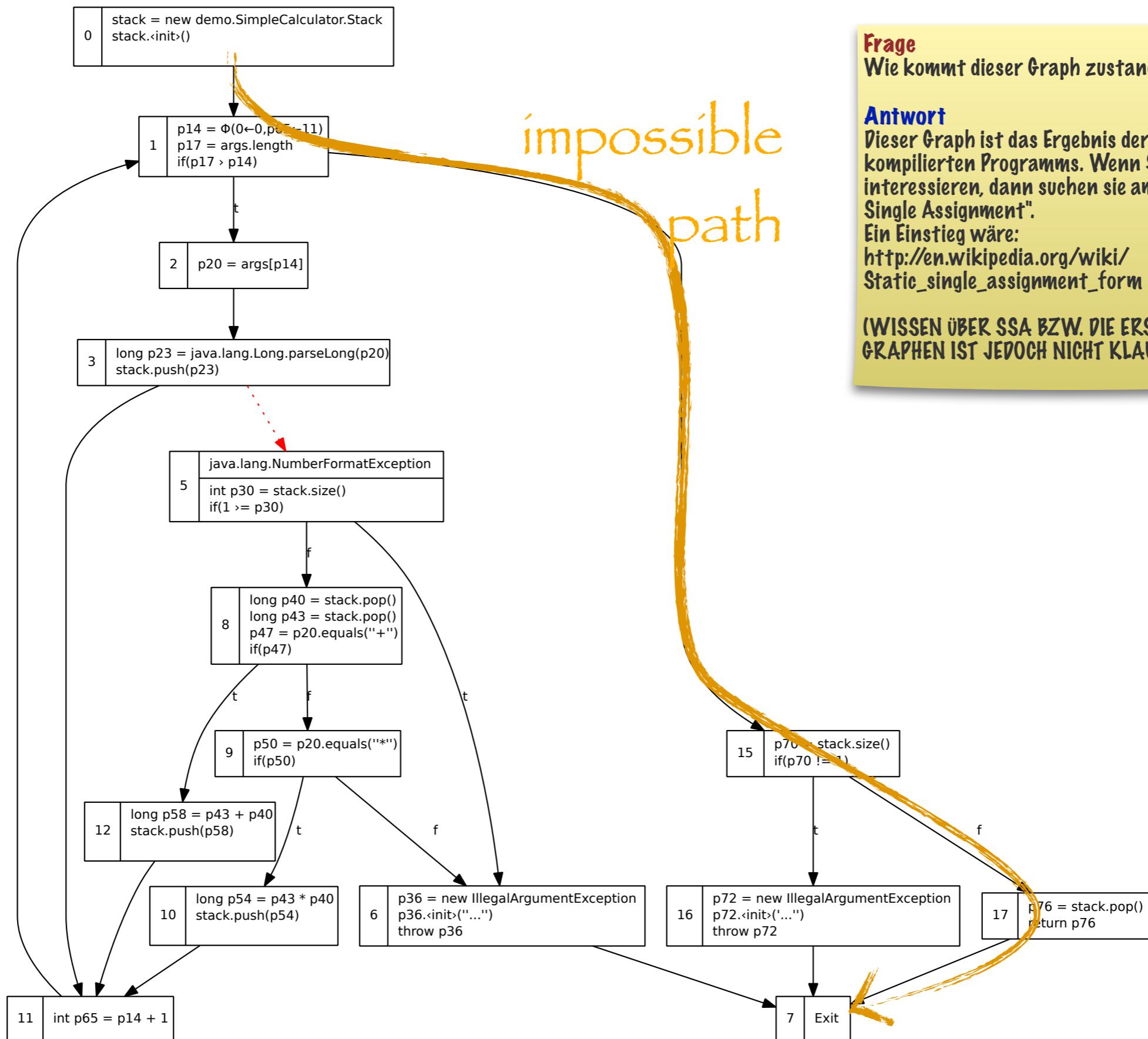
Calculating the result of
an arithmetic expression
in postfix notation:

$4 \ 5 + 5 * 3 \ 4 * * = ?$

$\cancel{4} \ 2 + 2 * 3 \cancel{4} * * = ?$

Basic Blocks of long process(String[] args)

static long process(java.lang.String[] args) | 47



Frage

Wie kommt dieser Graph zustande?

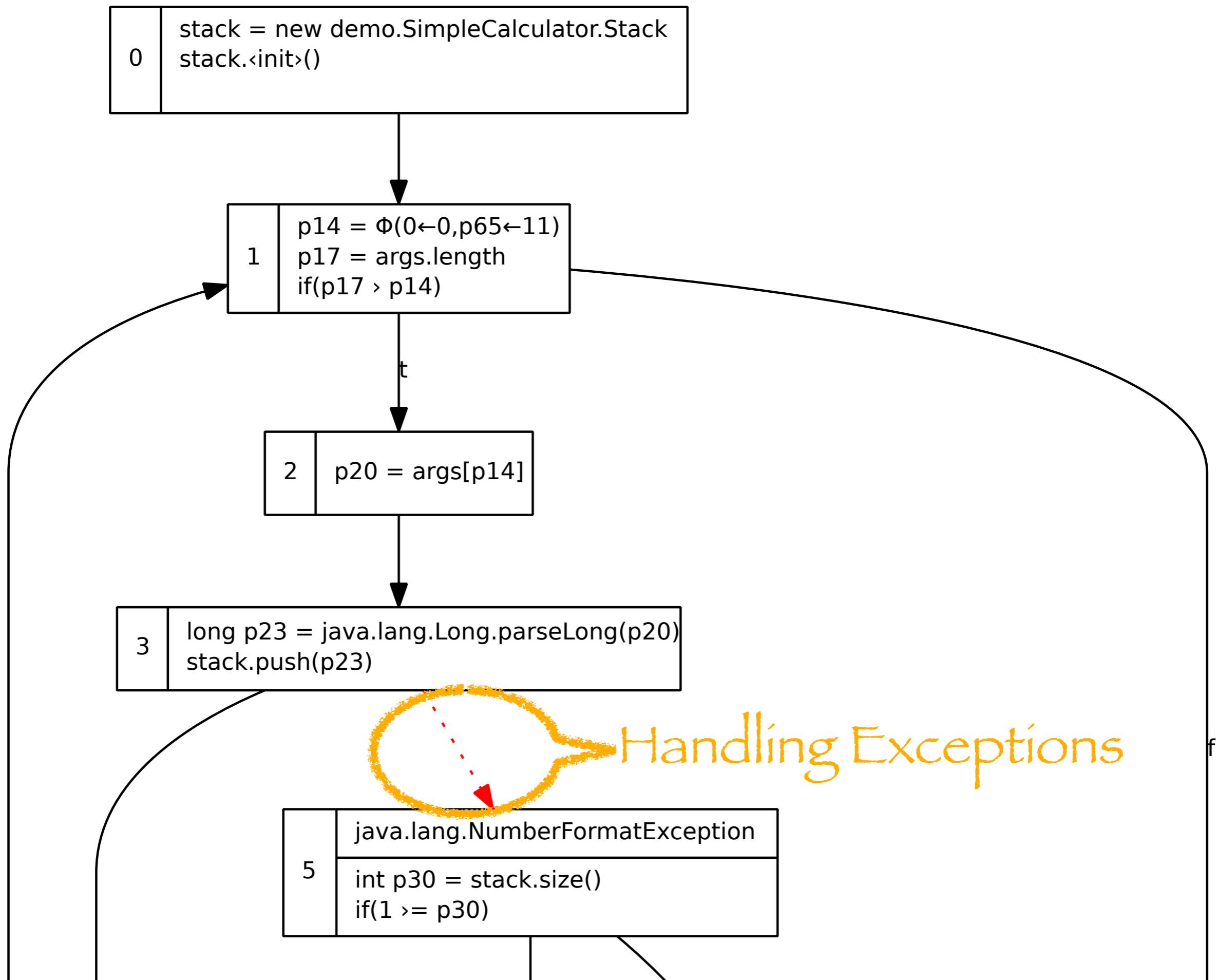
Antwort

Dieser Graph ist das Ergebnis der Repräsentation des kompilierten Programms. Wenn Sie Details dazu interessieren, dann suchen sie am Besten nach "Static Single Assignment".

Ein Einstieg wäre:

http://en.wikipedia.org/wiki/Static_single_assignment_form

(WISSEN ÜBER SSA BZW. DIE ERSTELLUNG SOLCHER GRAPHER IST JEDOCH NICHT KLAUSURRELEVANT!)



|| *Do not use a code coverage model as a test model.*

Do not rely on code coverage models to devise test suites. Test from responsibility models and use coverage reports to analyze test suite adequacy.



Covering some aspect of a method [...] is never a guarantee of bug-free software.

Robert V. Bender

Testing Object-Oriented Systems
Addison Wesley 2000

Steve Cornett

<http://www.bullseye.com/coverage.html>

Software Testing - Code Coverage | 50

- Recommended Reading

The screenshot shows a web browser window titled "Code Coverage Analysis". The page content is as follows:

Code Coverage Analysis

This paper gives a complete description of code coverage analysis (test coverage analysis), a software testing technique.

By Steve Cornett. Copyright © [Bullseye Testing Technology](#) 1996-2008. All rights reserved.
Redistribution in whole or in part is prohibited without permission.

Contents

- [Introduction](#)
- [Structural Testing and Functional Testing](#)
- [The Premise](#)
- [Basic Metrics](#)
 - [Statement Coverage](#)
 - [Decision Coverage](#)
 - [Condition Coverage](#)
 - [Multiple Condition Coverage](#)
 - [Condition/Decision Coverage](#)
 - [Modified Condition/Decision Coverage](#)
 - [Path Coverage](#)
- [Other Metrics](#)
 - [Function Coverage](#)
 - [Call Coverage](#)
 - [Linear Code Sequence and Jump \(LCSAJ\) Coverage](#)
 - [Data Flow Coverage](#)
 - [Object Code Branch Coverage](#)
 - [Loop Coverage](#)
 - [Race Coverage](#)
 - [Relational Operator Coverage](#)
 - [Weak Mutation Coverage](#)
 - [Table Coverage](#)
- [Comparing Metrics](#)
- [Coverage Goal for Release](#)
- [Intermediate Coverage Goals](#)
- [Summary](#)
- [References](#)

Limits of Testing



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Limits of Testing

The number of input and output combinations for trivial programs is already (very) large.

Assume that we limit points to integers between 1 and 10; there are 10^4 possible ways to draw (a single) line.

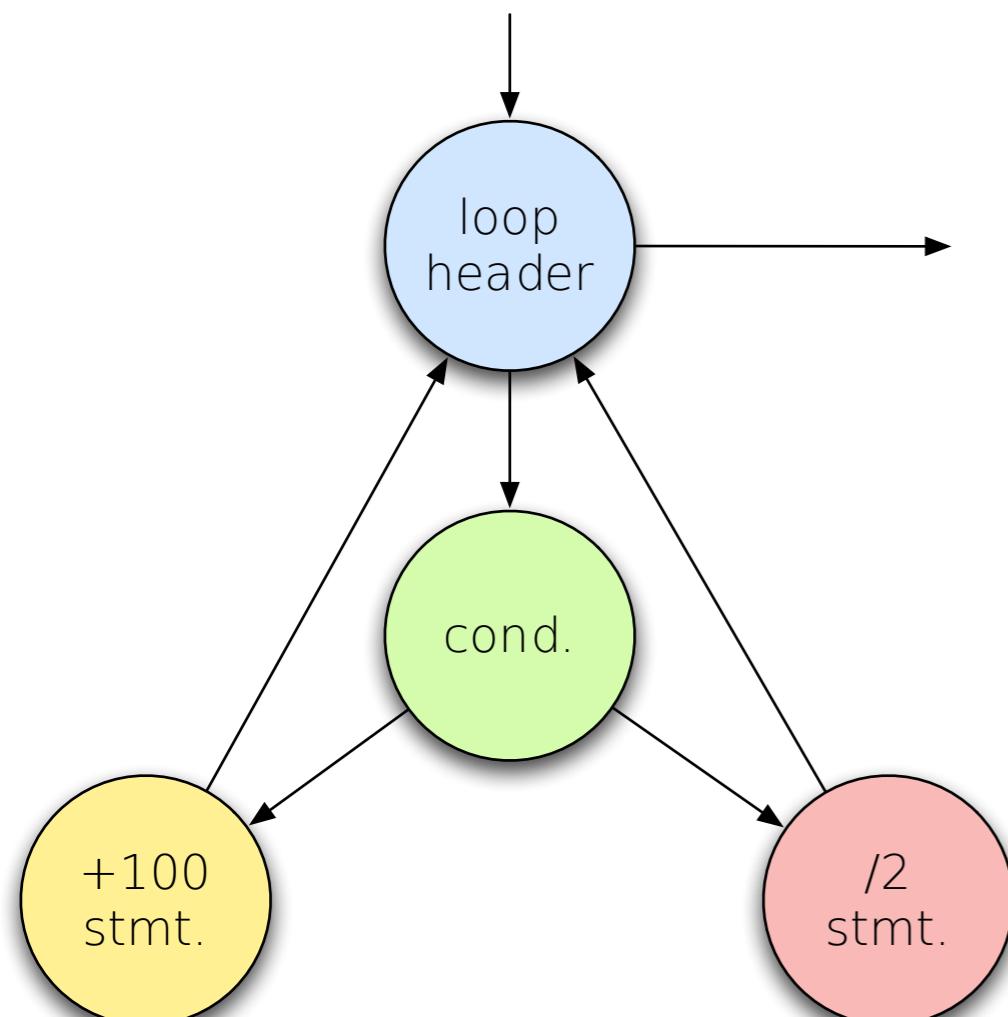
Since a triangle has three lines we have $10^4 \times 10^4 \times 10^4$ possible inputs of three lines (including invalid combinations).

We can never test all inputs, states, or outputs.

Limits of Testing

Branching and dynamic binding result in a very large number of unique execution sequences. Simple iteration increases the number of possible sequences to astronomical proportions.

Software Testing - Limits | 53



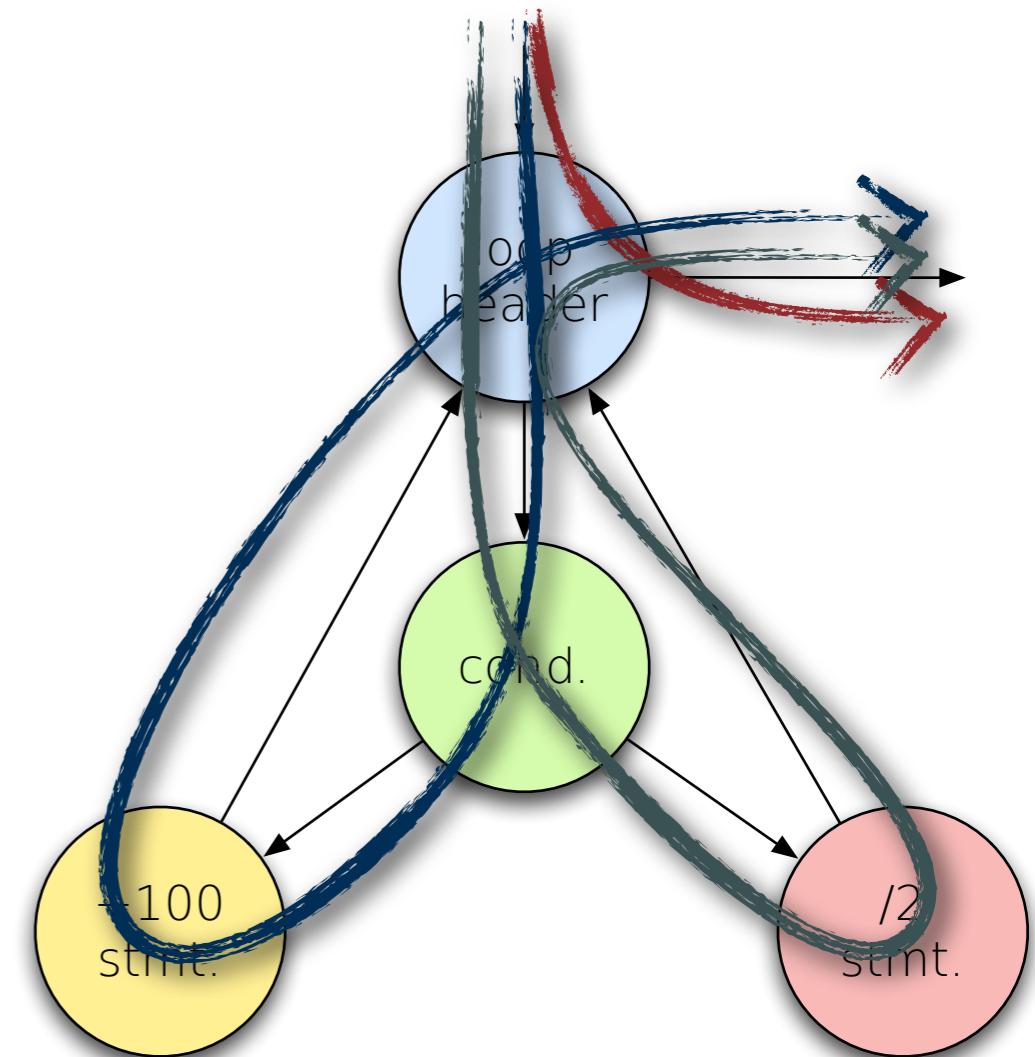
```
for (
    int i = 0;
    i < n;
    ++i
)
{
    if (a.get(i) == b.get(i))
        x[i] = x[i]+100;
    else
        x[i] = x[i] / 2;
}
```

Limits of Testing

Branching and dynamic binding result in a very large number of unique execution sequences.

If we count **entry-exit paths without regarding iteration** there are only three paths:

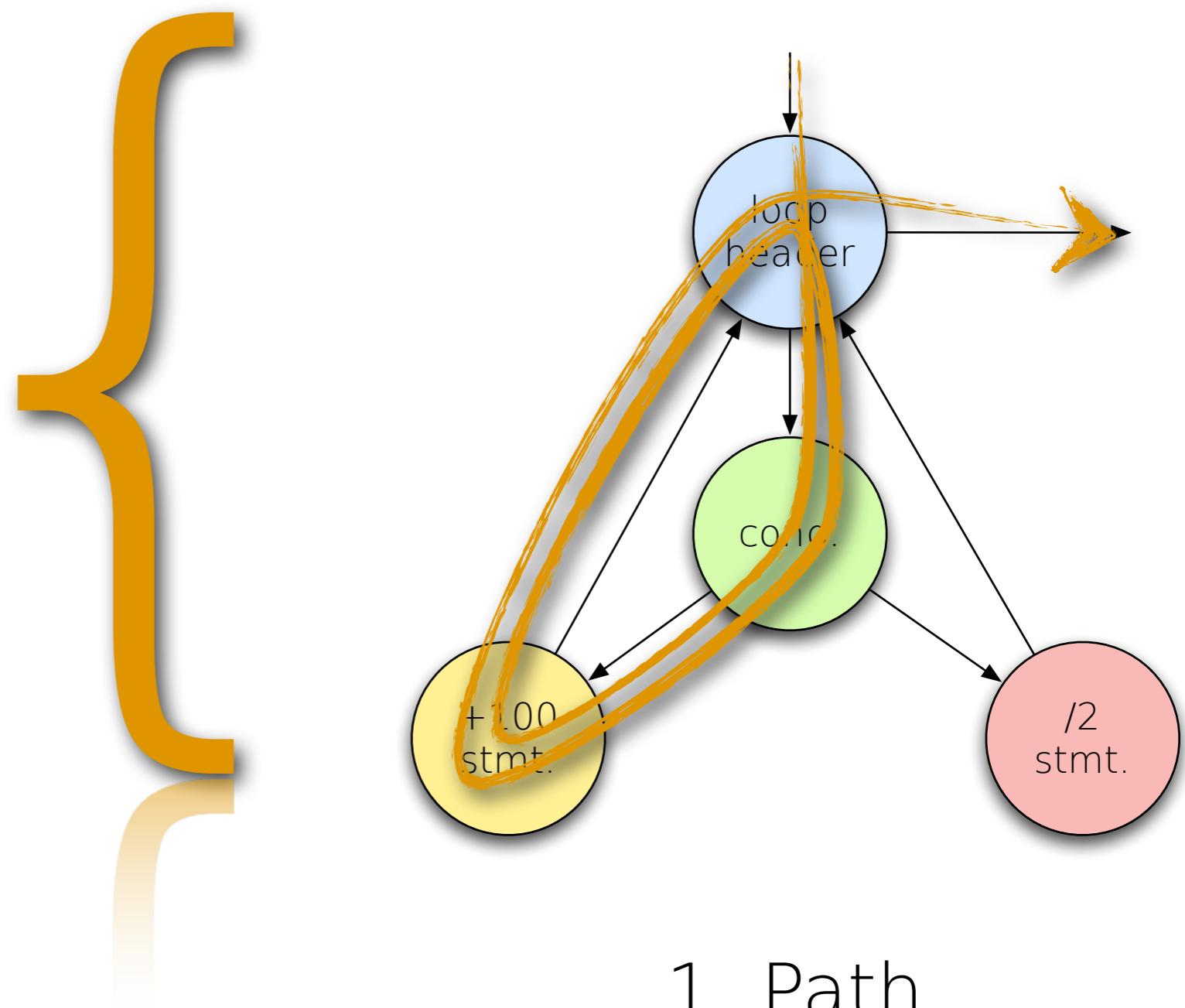
1. **loop header, exit**
2. **loop header, cond., +100**
3. **loop header, cond., /2**



Limits of Testing

Branching and dynamic binding result in a very large number of unique execution sequences. *Simple iteration increases the number of possible sequences to astronomical proportions.*

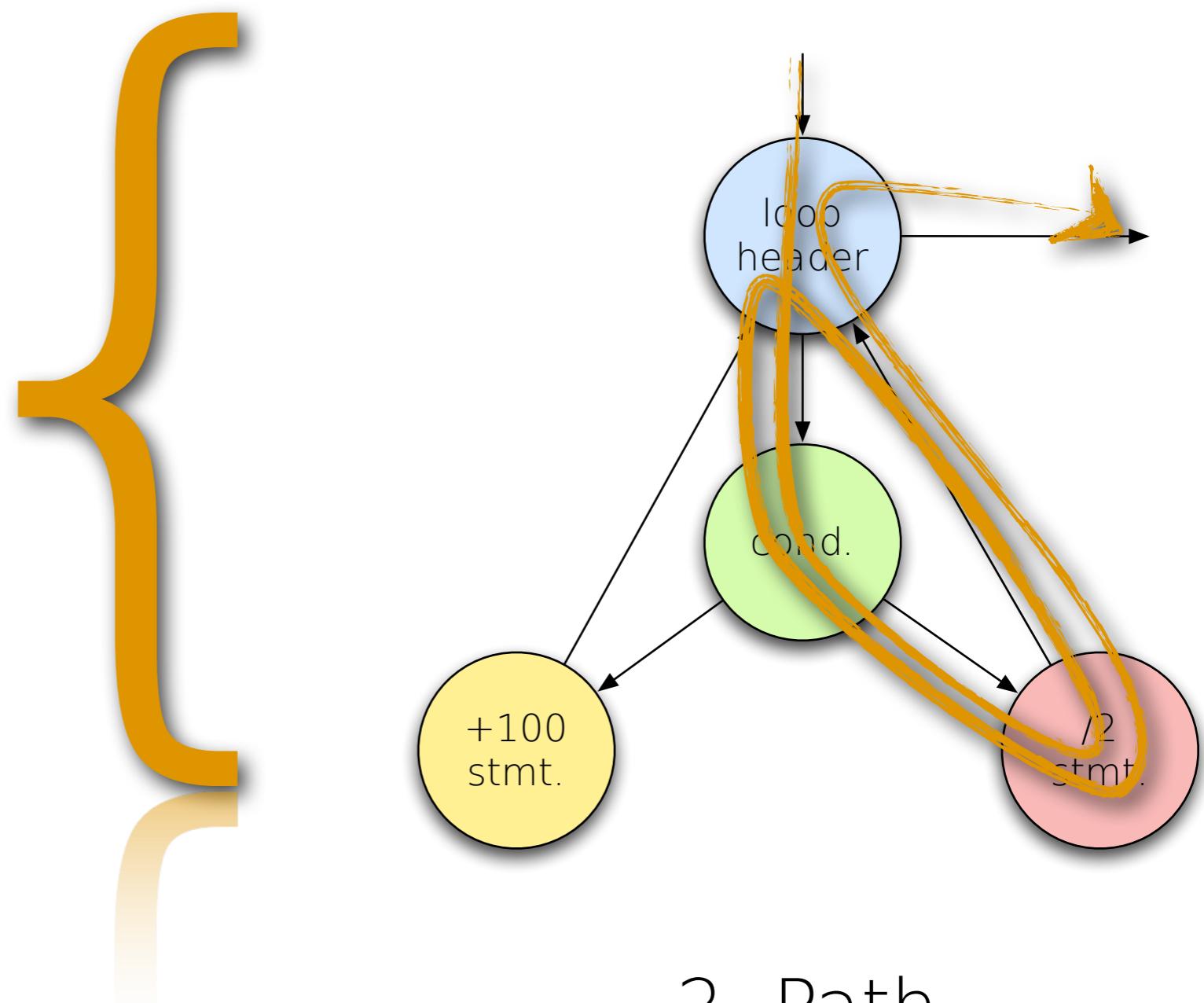
Number of iterations	Number of paths
1	$2^1 + 1 = 3$
2	$2^2 + 1 = 5$
3	$2^3 + 1 = 9$
10	1.025
20	1.048.577



Limits of Testing

Branching and dynamic binding result in a very large number of unique execution sequences. *Simple iteration increases the number of possible sequences to astronomical proportions.*

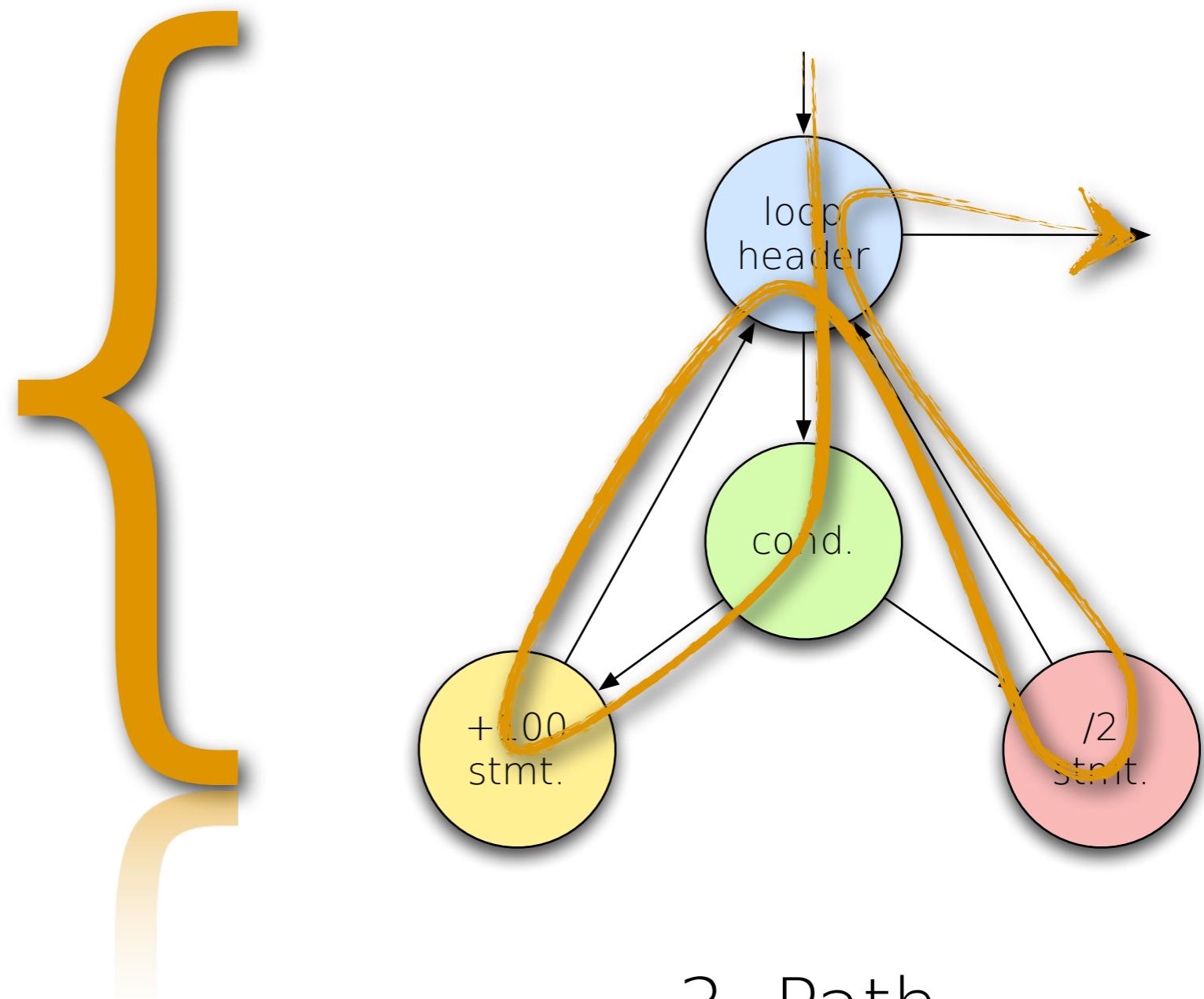
Number of iterations	Number of paths
1	$2^1 + 1 = 3$
2	$2^2 + 1 = 5$
3	$2^3 + 1 = 9$
10	1.025
20	1.048.577



Limits of Testing

Branching and dynamic binding result in a very large number of unique execution sequences. *Simple iteration increases the number of possible sequences to astronomical proportions.*

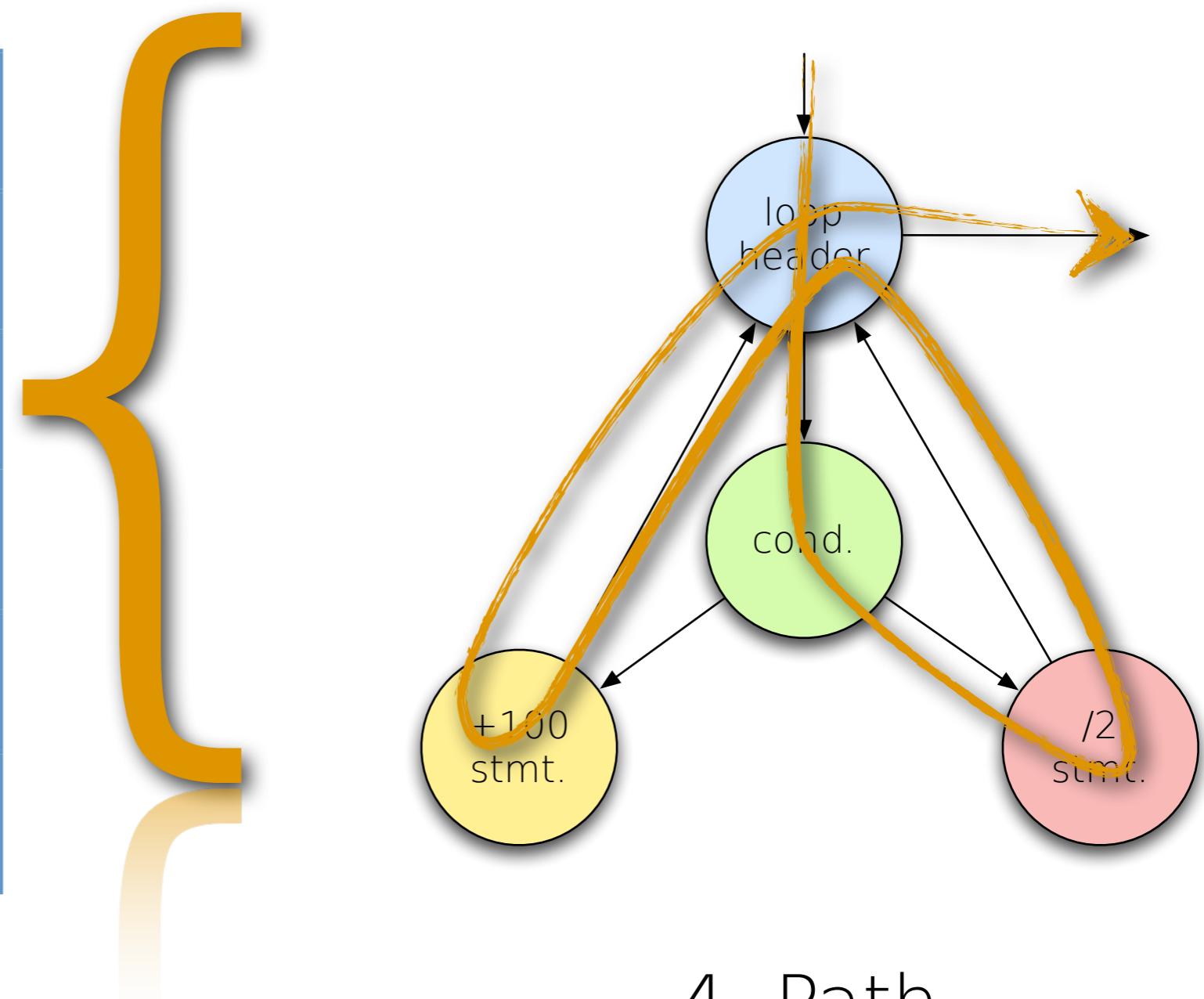
Number of iterations	Number of paths
1	$2^1 + 1 = 3$
2	$2^2 + 1 = 5$
3	$2^3 + 1 = 9$
10	1.025
20	1.048.577



Limits of Testing

Branching and dynamic binding result in a very large number of unique execution sequences. *Simple iteration increases the number of possible sequences to astronomical proportions.*

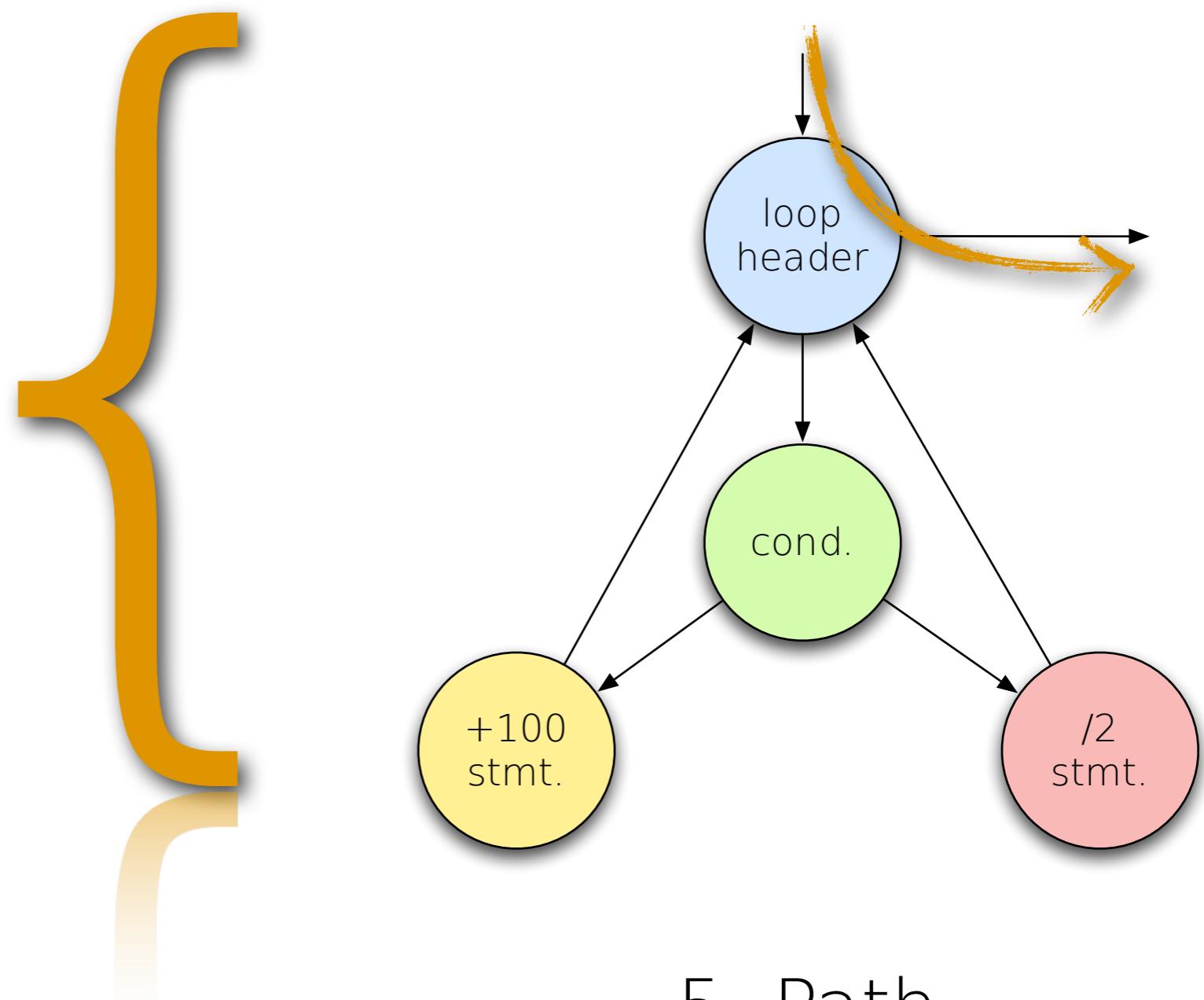
Number of iterations	Number of paths
1	$2^1 + 1 = 3$
2	$2^2 + 1 = 5$
3	$2^3 + 1 = 9$
10	1.025
20	1.048.577



Limits of Testing

Branching and dynamic binding result in a very large number of unique execution sequences. *Simple iteration increases the number of possible sequences to astronomical proportions.*

Number of iterations	Number of paths
1	$2^1 + 1 = 3$
2	$2^2 + 1 = 5$
3	$2^3 + 1 = 9$
10	1.025
20	1.048.577



5. Path

The ability of code to **hide faults** from a test suite is called its **fault sensitivity**.

Software Testing - Limits | 60

Coincidental correctness is obtained when buggy code can produce correct results for some inputs.

E.g. assuming that the correct code would be:

$$x = x + x$$

but you wrote

$$x = x * x$$

If $x = 2$ is tested the code hides the bug: it produces a correct result from buggy code. However, this bug is easily identified.

Implementing Tests

- A Very First Glimpse



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
static long process(String[] args) throws IllegalArgumentException {
```

```
    Stack values = new Stack();
    for (int i = 0; i < args.length; i++) {
        String arg = args[i];
        try {
            long value = Long.parseLong(arg);
            values.push(value);
        } catch (NumberFormatException nfe) {
            // there is no method to test if a string is a number
            if (values.size() > 1) {
                long r = values.pop();
                long l = values.pop();
                if (arg.equals("+")) {
                    values.push(l + r);
                    continue;
                }
                if (arg.equals("*")) {
                    values.push(l * r);
                    continue;
                }
            }
            throw new IllegalArgumentException("Too few operands or operator unknown.");
        }
    }
    if (values.size() == 1) return values.pop();
    else throw new IllegalArgumentException("Too few (0) or too many (>1) operands.");
}
```

Calculating the result of
an arithmetic expression
in postfix notation:

$4 \ 5 + 5 * 3 \ 4 * * = ?$

$\cancel{4} \ 2 + 2 * 3 \cancel{4} * * = ?$

A Test Plan That Achieves Basic Block Coverage

static long process(java.lang.String[] args) | 63

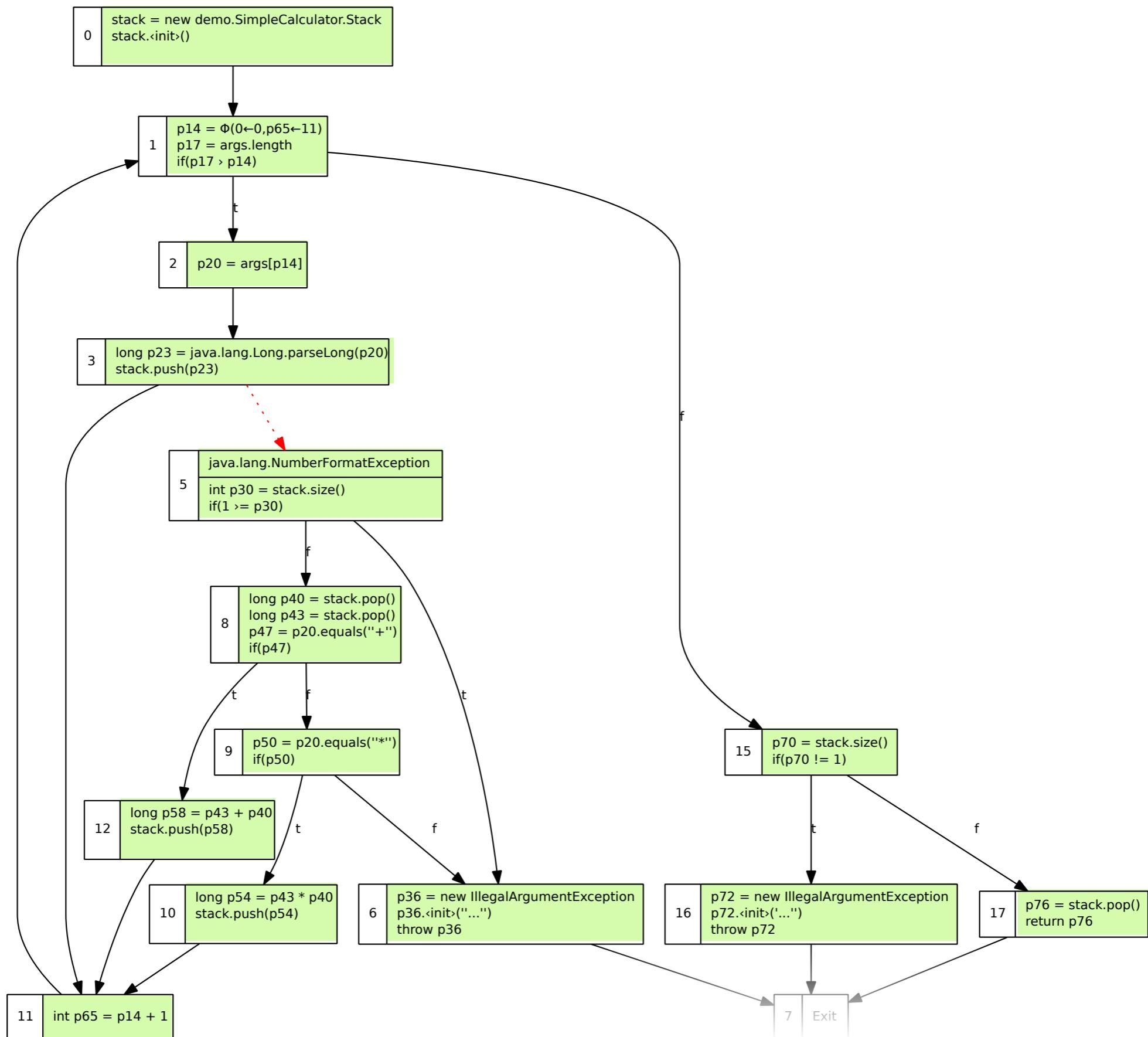
A Test Plan That Achieves Basic Block Coverage

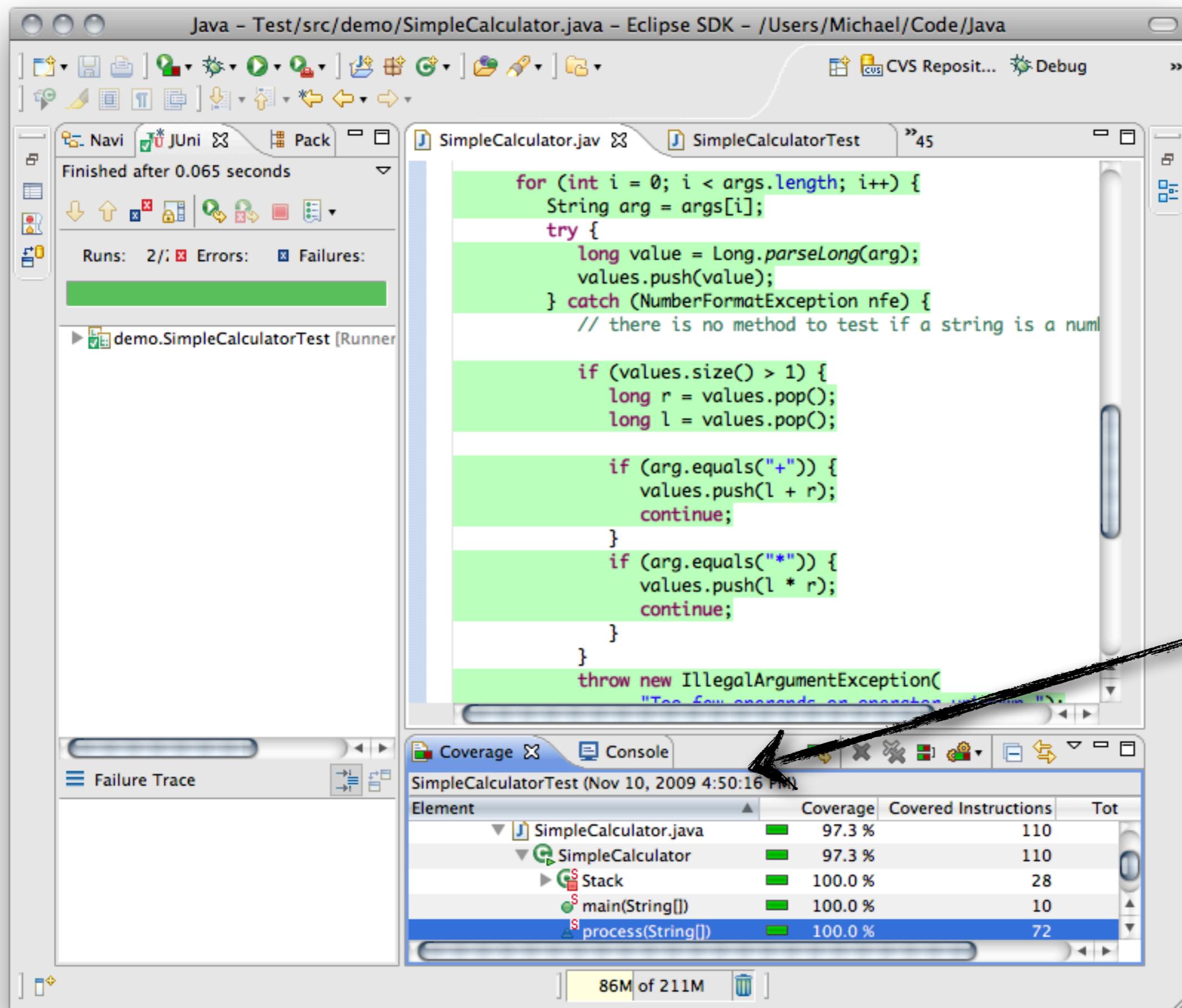
```
static long process(java.lang.String[] args) | 64
```

Description	Input	Expected Output
Test calculation of the correct result	{"4", "5", "+", "7", "*"} {"4", "5", "+", "*", "7"}	63 <i>Exception: "Too few operands or operator unknown."</i>
Test that too few operands leads to the corresponding exception	{"4", "5", "+", "*"} {"4", "5", "+", "*", "7"}	<i>Exception: "Too few operands or operator unknown."</i>
Test that an illegal operand throws the corresponding exception	{}	<i>Exception: "Too few (0) or too many (>1) operands left."</i>
Test that an expression throws the corresponding exception	{"4", "5"}	<i>Exception: "Too few (0) or too many (>1) operands left."</i>

Basic Blocks of long process(String[] args)

static long process(java.lang.String[] args) | 65





ECL
Emma
(Eclipse
Plug-in)

Writing a Test Case using JUnit (4)

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.fail;

import java.util.Arrays;

import org.junit.Test;

public class SimpleCalculatorTest {

    @Test
    public void testProcess() {
        String[] term = new String[] {
            "4", "5", "+", "7", "*"
        };
        long result = SimpleCalculator.process(term);
        assertEquals(Arrays.toString(term), 63, result);
    }
}
```

Writing a Test Case using JUnit (3)

Testing Exception Handling

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.fail;

import java.util.Arrays;

import org.junit.Test;

public class SimpleCalculatorTest extends ... {

    public void testProcess() {

        try {
            SimpleCalculator.process(new String[0]);
            fail();
        } catch (IllegalArgumentException iae) {
            assertEquals(
                "Too few (0) or too many (>1) operands.",
                iae.getMessage());
        }
    }
}
```

Writing a Test Case using JUnit (4)

- Testing Exception Handling

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.fail;

import java.util.Arrays;

import org.junit.Test;

public class SimpleCalculatorTest {
    @Test(expected=IllegalArgumentException.class)
    public void testProcess() {
        SimpleCalculator.process(new String[0]);
    }
}
```

Behavior-Driven Development

The goal is that developers define the behavioral intent of the system that they are developing.

<http://behaviour-driven.org/>

Software Testing - Behavior-Driven Development | 70

Using ScalaSpec 1.5: <http://code.google.com/p/specs/>

```
import org.specs.runner._  
import org.specs._  
  
object SimpleCalculatorSpec extends Specification {  
  
  "The Simple Calculator" should {  
    "return the value 36 for the input {"6","6","*"}" in {  
      SimpleCalculator.process(Array("6", "6", "*")) must_== 36  
    }  
  }  
}  
}
```

Implemented in Scala

(Method-) Stub

- A stub is a partial, temporary implementation of a component (e.g., a placeholder for an incomplete component)
- Stubs are often required to simulate complex systems; to make parts of complex systems testable in isolation

An alternative is to use a Mock object that mimics the original object in its behavior and facilitates testing.

Testing comprises the efforts to find defects.

Debugging is the process of locating and correcting defects.

(Hence, debugging is not testing, and testing is not debugging.)

Summary

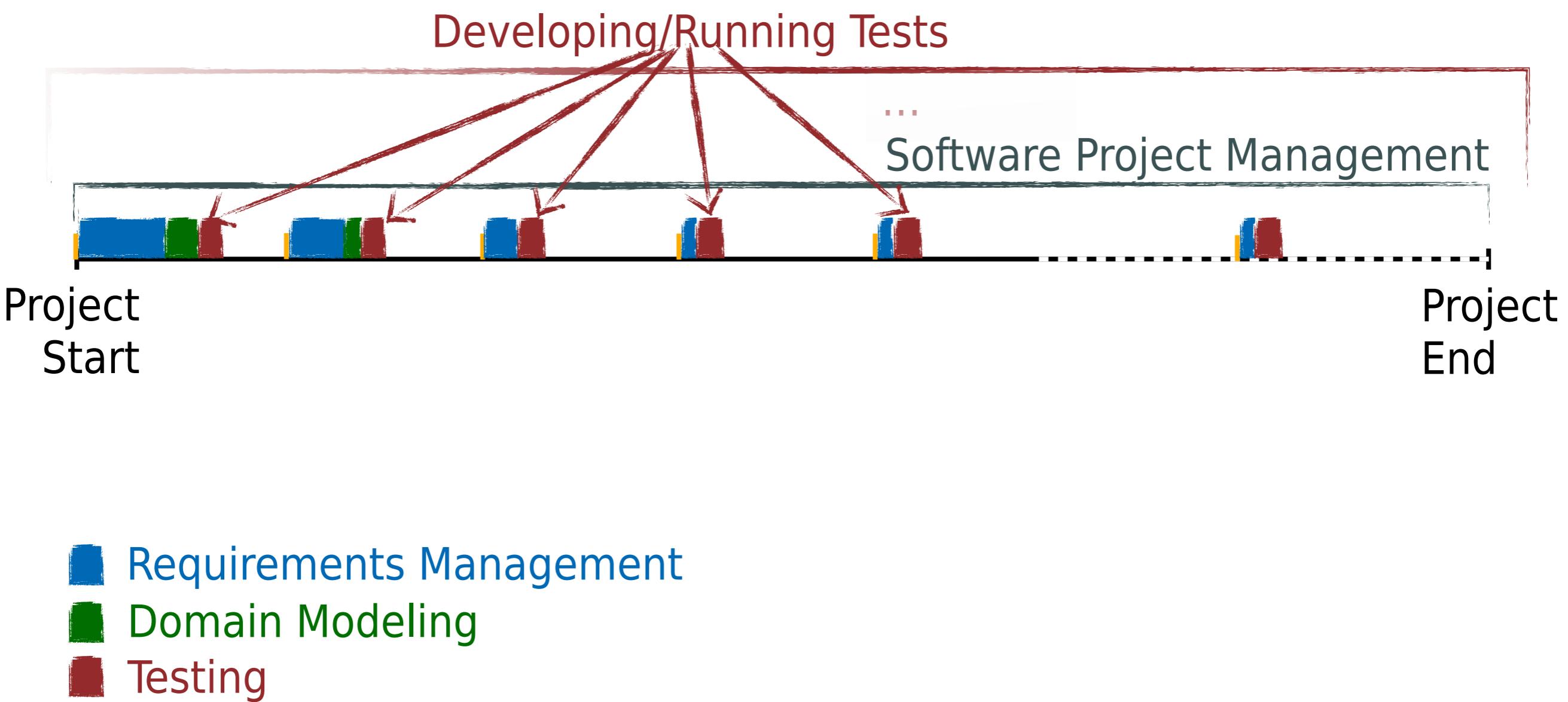


TECHNISCHE
UNIVERSITÄT
DARMSTADT

The goal of this lecture is to enable you to systematically carry out small(er) software projects that produce quality software.

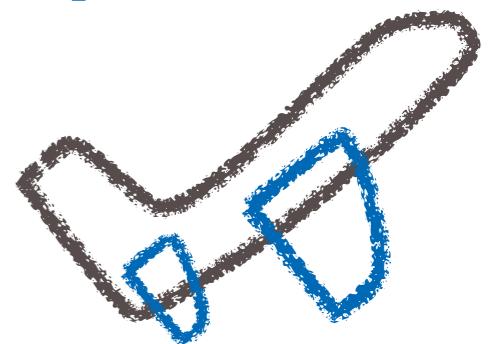
-
- Testing has to be done systematically; exhaustive testing is not possible.
 - Test coverage models help you to assess the quality of your test suite; however, “just” satisfying a test coverage goal is usually by no means sufficient.
 - Do take an “external” perspective when you develop your test suite.

The goal of this lecture is to enable you to systematically carry out small(er) commercial or open-source projects.



|| A Tester's Courage

The Director of a software company proudly announced that a flight software developed by the company was installed in an airplane and the airline was offering free first flights to the members of the company. “Who are interested?” the Director asked. Nobody came forward. Finally, one person volunteered. The brave Software Tester stated, ‘I will do it. I know that the airplane will not be able to take off.’



Unknown Author

<http://www.softwaretestingfundamentals.com>