

Software testing

Questões de Segurança em Engenharia de Software (QSES)

Mestrado em Segurança Informática

Departamento de Ciência de Computadores

Faculdade de Ciências da Universidade do Porto

Eduardo R. B. Marques, edrdo@dcc.fc.up.pt



Software testing



You are a lucky bug. I'm seeing that you'll be shipped with the next five releases.

■ Testing

- Observe if software meets the expected behavior when **executed**.
- Does *not guarantee* absence of bugs, in fact it seeks to *expose them*.

How important is testing?

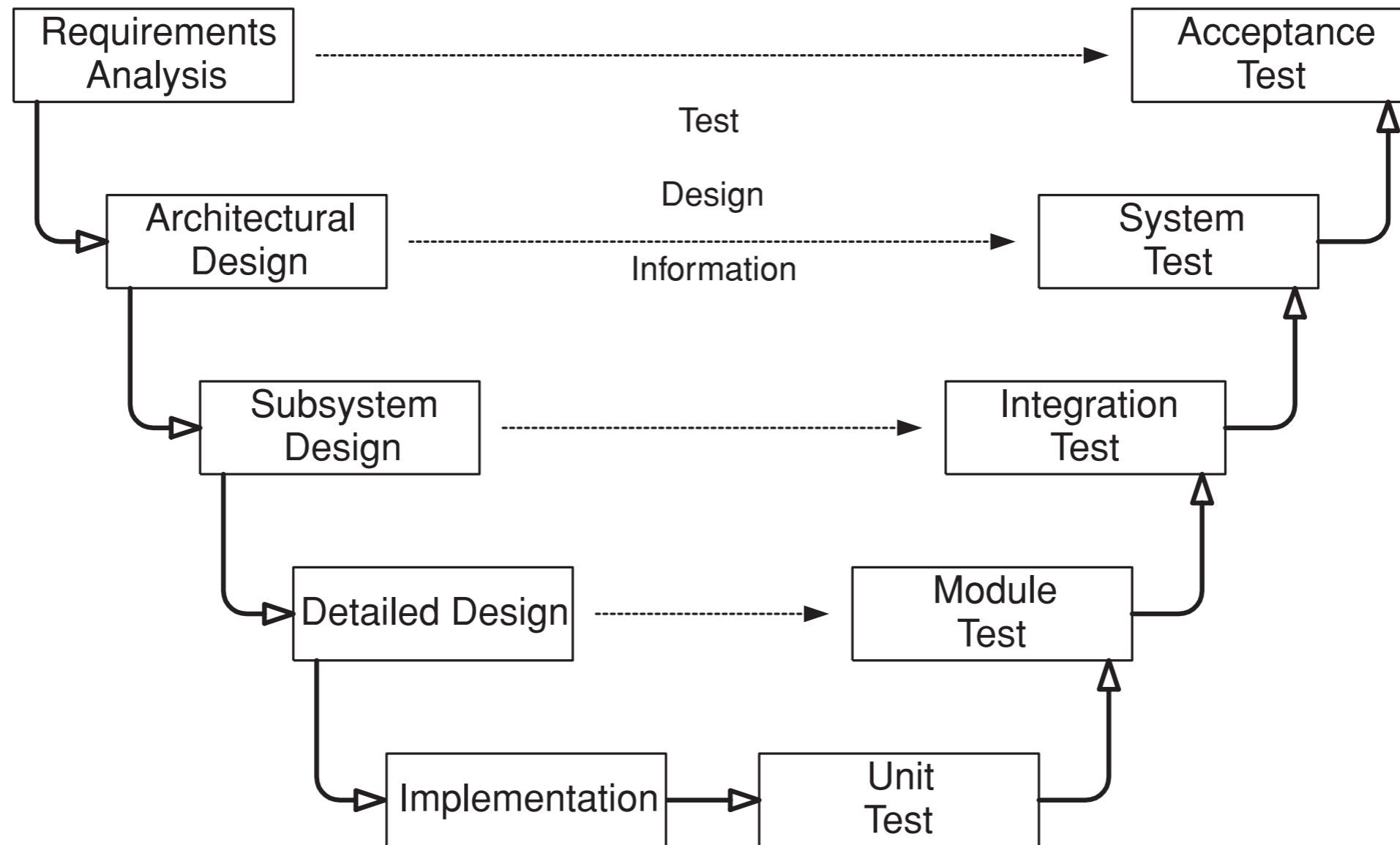
History of Software Testing



Image source: [History of software testing](#), blog article, Ashish Singh, 2012

- Testing is the standard approach for ensuring that software has a high level of reliability.
- No (serious) SDLC process goes without testing.

Testing levels & the SDLC



From: "Introduction to Software Testing", Amman & Offutt

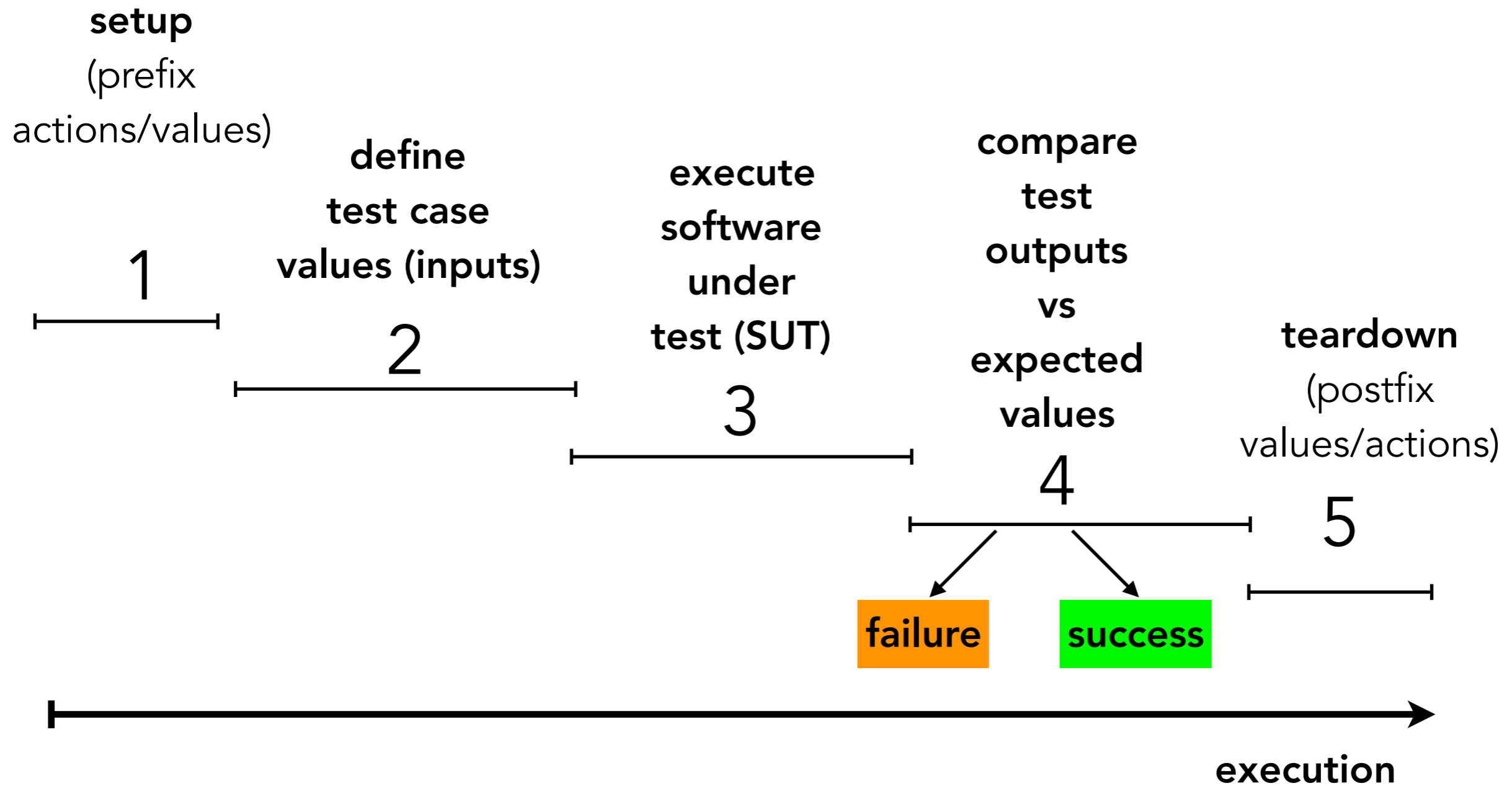
How “mature” should testing be?

■ Beizer’s scale for test process maturity

- **Level 0:** “There’s no difference between testing and debugging.”
 - ◆ **Question: What is debugging?**
- **Level 1:** “The purpose of testing is to show that the software works.”
- **Level 2:** “The purpose of testing is to show that the software doesn’t work.”
- **Level 3:** “The purpose of testing is not to prove anything specific, but to reduce the risk of using the software.”
- **Level 4:** “Testing is a mental discipline that helps all IT professionals develop higher quality software.”

Base concepts & terminology in software testing

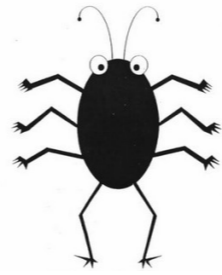
Test case



Test case (cont.)

- **Test case inputs:** the input values necessary to complete a particular execution of the SUT.
 - The data supplied by to the SUT (e.g. method arguments).
 - The pre-state (starting state) of the SUT (if stateful).
- **Expected outputs:** the expected values for the test case if and only if the program satisfies its intended behavior.
 - The data produced by the SUT in response to the input (e.g. function return values).
 - The post-state of the SUT (if stateful).
- **Test failure:** expected outputs \neq observed outputs
- **Prefix values/actions:** inputs/commands necessary to put the SUT or its environment into the appropriate state before execution e.g. database setup.
- **Postfix values/actions :** inputs/commands necessary to reset the SUT or its environment after execution e.g. database teardown.

A simple bug



```
public static int numZero(int[] x) {  
    // Effects: if x == null throw NullPointerException  
    // else return the number of occurrences of 0 in x  
1    int count = 0;  
2    for (int i = 1; i < x.length; i++)  
3        if (x[i] == 0)  
4            count++;  
5    return count;  
}
```

- There is a simple “bug” in numZero...
 - **Where is the bug location** in the source code ? How would you fix it?
 - If the bug is location is reached, **how does it corrupt program state**? Does it always corrupt program state ?
 - **If program state is corrupted, does numZero fail** ? How?
- **The term “bug” is ambiguous however ...** are we referring to the source code or to outcome of a failed execution ? We need clear terminology.

Example test cases for numZero

```
public static int numZero(int[] x) {  
    // Effects: if x == null throw NullPointerException  
    // else return the number of occurrences of 0 in x  
1    int count = 0;  
2    for (int i = 1; i < x.length; i++)  
3        if (x[i] == 0)  
4            count++;  
5    return count;  
}
```

test case	test case values (x)	expected values	actual execution	failure?
1	null	NullPointerException	NullPointerException	No
2	{ }	0	0	No
3	{ 1, 2, 3 }	0	0	No
4	{ 1, 0, 1, 0 }	2	2	No
5	{ 0, 1, 2, 0 }	2	1	Yes

Fault, Error, Failure [Falta, Erro, Falha]

```
public static int numZero(int[] x) {  
    // Effects: if x == null throw NullPointerException  
    // else return the number of occurrences of 0 in x  
1    int count = 0;  
2    for (int i = 1; i < x.length; i++)  
3        if (x[i] == 0)  
4            count++;  
5    return count;  
}
```

- **Fault:** a defect in source code [**the location of the bug**]
 - `i = 1` in the code [should be fixed to `i = 0`]
- **Error:** erroneous program state caused by execution of the defect [**semantic effect of the bug**]
 - `i` becomes 1 (array entry 0 is not ever read)
- **Failure:** propagation of erroneous state to the program outputs [**manifestation of the bug**]
 - The output value for `x = { 0, 1, 0 }` is 1 instead of the expected value 2.
 - Failure happens as long as `x.length > 0 && x[0] = 0`

State representation - convention

- We will represent program states using the notation `<var=v1, ..., varN=vN, PC=program counter>`
- Example sequence of states in the execution of `numZero({0,1,2,0})`
 - 1: `< x={0,1,2,0}, PC=[int count=0 (11)] >`
 - 2: `< x={0,1,2,0}, count=0, PC=[i=1 (12)] >`
 - 3: `< x={0,1,2,0}, count=0, i=1, PC=[i<x.length (12)] >`
 - 4: `< x={0,1,2,0}, count=0, i=1, PC=[if(x[i]==0) (13)] >`
 - ...
 - `<x={0,1,2,0}, count=1, PC=[return count;(15)] >`

Error state - convention

- We will use the convention: an error state is the first different state in execution in comparison to an execution to the state sequence of what would be the correct program.
- If we had `i=0` the execution of `numZero({0,1,2,0})` would begin with states:
 - 1: `< x={0,1,2,0}, PC=[int count=0 (l1)] >`
 - 2: `< x={0,1,2,0}, count=0, PC=[i=0 (l2)] >`
 - 3: `< x={0,1,2,0}, count=0, i=0, PC=[i<x.length (l2)] > ...`
- Instead we have
 - 1: `< x={0,1,2,0}, PC=[int count=0 (l1)] >`
 - 2: `< x={0,1,2,0}, count=0, PC=[i=1 (l2)] >`
 - 3: `< x={0,1,2,0}, count=0, i=1, PC=[i<x.length (l2)] > ...`
- The first error state is 2: `< x={0,1,2,0}, count=0, PC=[i=1 (l2)] >`

The **RIP** Conditions for test failure

■ **Reachability**

- The fault in the source code is reached during execution.

■ **Infection**

- The program state enters in an error state, affected by the execution of the faulty code.

■ **Propagation**

- The errors in program state are propagated to the outputs.

numZero: execution w/error and failure reachability + infection + propagation

```
public static int numZero(int[] x) {  
    // Effects: if x == null throw NullPointerException  
    // else return the number of occurrences of 0 in x  
1    int count = 0;  
2    for (int i = 1; i < x.length; i++)  
3        if (x[i] == 0)  
4            count++;  
5    return count;  
}
```

- Considering an execution where $x = \{0, 1, 2, 0\}$
 - **Error:** $\langle x=\{0,1,2,0\}, i=1, count=0, PC= \text{if } \dots, 13 \rangle$ deviates from expected internal state $\langle x=\{1,0,2,0\}, i=0, count=1, PC = [\text{if } \dots, 13] \rangle$
 - **And failure:** $\text{numZero}(\{0,1,2,0\})$ will return 1 rather than 2.

numZero: execution w/error **but no failure**
reachability + infection **but no propagation**

```
public static int numZero(int[] x) {  
    // Effects: if x == null throw NullPointerException  
    // else return the number of occurrences of 0 in x  
1    int count = 0;  
2    for (int i = 1; i < x.length; i++)  
3        if (x[i] == 0)  
4            count++;  
5    return count;  
}
```

- Considering an execution where $x = \{1, 0, 2, 0\}$
 - **Error:** $\langle x=\{1,0,2,0\}, i=1, count=0, PC= \text{if } \dots, 13 \rangle$ deviates from expected internal state $\langle x=\{1,0,2,0\}, i=0, count=1, PC = [\text{if } \dots, 13] \rangle$
 - **No Failure!** $\text{numZero}(\{1,0,2,0\})$ will return 2 as expected.

More terminology

- **Test set:** a set of test cases. We will use notation **T** for a test set.
- **Test requirement :** requirement that should be satisfied by a test set. Test requirements normally come in sets. We use notation **TR** for the set of test requirements.
- **Coverage criterion:** A coverage criterion **C** is a rule or collection of rules that define a set of test requirements **TR(C)** to be satisfied by a test set.
- **Coverage level:** the percentage of test requirements that are satisfied by a test set. We say **T** satisfies **C** if the coverage level of **TR(C)** by **T** is 100 %.
- **Infeasible requirement:** requirement that cannot be satisfied by any test case. If there are infeasible test requirements, the coverage level will never be 100%.

Structural coverage criteria

```
public static int numZero(int[] x) {  
    // Effects: if x == null throw NullPointerException  
    // else return the number of occurrences of 0 in x  
1    int count = 0; /* I1 */  
2    for (int i = 1 /* I2 */; i < x.length /* I3,B1 */; i++ /* I4 */)   
3        if (x[i] == 0) /* I5,B2 */  
4            count++; /* I6 */  
5    return count; /* I7 */  
}
```

- **Line coverage (LC):** cover every line in the SUT.
 - $TR(LC) = \{ \text{line 1, line 2, line 3, line 4, line 5} \}$
- **Instruction coverage (IC):** cover every instruction in the SUT.
 - $TR(IC) = \{ I1, I2, I3, I4, I5, I6, I7 \}$
- **Branch coverage (BC):** cover every instruction, and including all cases at choice points (if, switch-case, etc).
 - $TR(BC) = \{ NPE-B1, B1, !B1, B2, !B2 \}$

LC, IC, BC for numZero

```

public static int numZero(int[] x) {
    // Effects: if x == null throw NullPointerException
    // else return the number of occurrences of 0 in x
1   int count = 0; /* I1 */
2   for (int i = 1 /* I2 */; i < x.length /* I3,B1 */; i++ /* I4 */)
3       if (x[i] == 0) /* I5,B2 */
4           count++; /* I6 */
5   return count; /* I7 */
}

```

test case	test case values (x)	expected values	exec. result	test fails?	LC	IC	BC
t1	null	NPE	NPE	no	1 2	i1 i2 i3	NPE-B1
t2	{ }	0	0	no	1 2 5	i1 i2 i3 i7	!B1
t3	{1,2}	0	0	no	1 2 3 5	All except i6	B1, !B1, !B2
t4	{0,0 }	2	1	yes	All	All	B1, !B1, B2
t5	{1,1,0}	1	0	no	All	All	B1, !B1, B2, !B2

LC, IC, BC for numZero

T (test set)	LC level	IC level	BC level
{ t1 }	40 % (2/5)	42 % (3/7)	20 % (1/5)
{ t1, t2 }	60 % (3/5)	57 % (4/7)	40 % (2/5)
{ t2, t3 }	80 % (4/5)	85 % (6/7)	60 % (3/5)
{ t4 }	100 % (5/5)	100 % (7/7)	60 % (3/5)
{ t1, t5 }	100 % (5/5)	100 % (7/7)	100 % (5/5)

test case	test case values (x)	expected values	exec. result	test fails?	LC	IC	BC
t1	null	NPE	NPE	no	1 2	i1 i2 i3	NPE-B1
t2	{ }	0	0	no	1 2 5	i1 i2 i3 i7	!B1
t3	{1,2}	0	0	no	1 2 3 5	All except i6	B1, !B1, !B2
t4	{0,0}	2	1	yes	All	All	B1, !B1, B2
t5	{1,1,0}	1	0	no	All	All	B1, !B1, B2, !B2

LC, IC, BC for numZero

T (test set)	LC level	IC level	BC level
{ t1 }		(3/7)	20 % (1/5)
{ t1, t2 }		(4/7)	40 % (2/5)
{ t2, t3 }		(6/7)	60 % (3/5)
{ t4 }	100 % (5/5)	100 % (7/7)	60 % (3/5)
{ t1, t5 }	100 % (5/5)	100 % (7/7)	100 % (7/7)

**100 % coverage for all criteria
but bug is not exposed!!!
t1 and t5 do not fail**

test case	test case values (x)	expected values	exec. result	test fails?	LC	IC	BC
t1	null	NPE	NPE	no	1 2	i1 i2 i3	NPE-B1
t2	{ }	0	0	no	1 2 5	i1 i2 i3 i7	!B1
t3	{1,2}	0	0	no	1 2 3 5	All except i6	B1, !B1, !B2
t4	{0,0}	2	1	yes	All	All	B1, !B1, B2
t5	{1,1,0}	1	0	no	All	All	B1, !B1, B2, !B2

Criteria subsumption

- **Criteria Subsumption:** A coverage criterion **C1** subsumes **C2** if and only if every test set that satisfies criterion **C1** also satisfies **C2**.
- For instance:
 - instruction coverage subsumes line coverage
 - branch coverage subsumes instruction coverage
- The inverse is not true. In the previous slide:
 - If `count++` appeared in the same line as `if (x[i] == 0)`, test case `t3` would cover all lines but not all instructions (instruction `i6` is not be executed by `t3`).
 - Test `t4` covers all instructions, but not all branches.

xUnit testing

xUnit — historical perspective

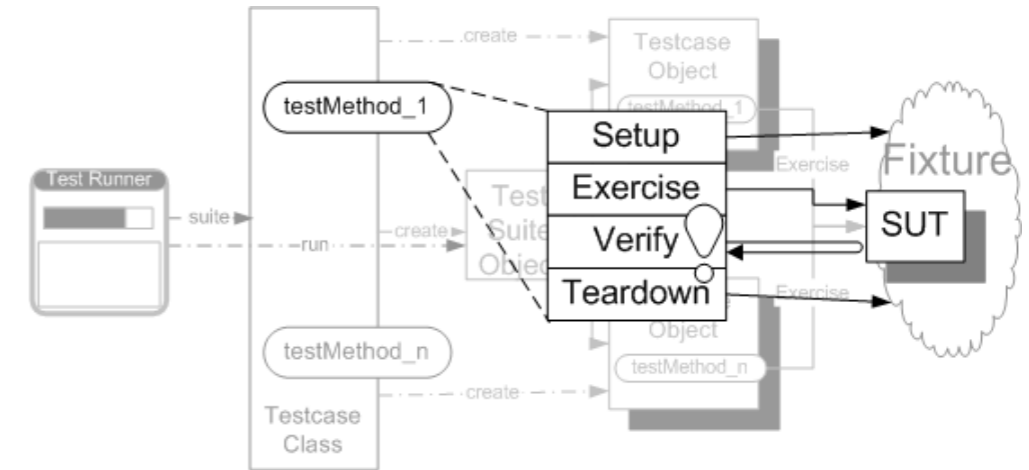
- **xUnit**: the general designation of a variety of unit testing frameworks, with an overall structure and functionality inspired by [SUnit](#).

SUnit

The mother of all unit testing frameworks

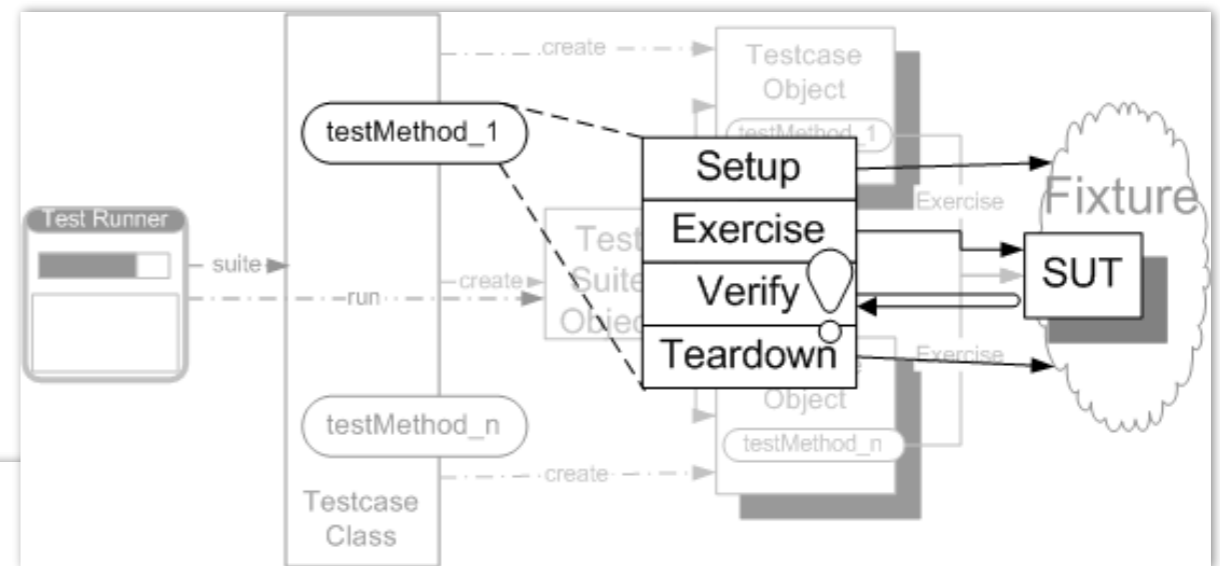
- **SUnit** is a testing library for the SmallTalk language developed by Kent Beck in 1998 (Kent Beck is one of the authors of the influential “[Manifesto for Agile Software Development](#)”.)
- Kent Beck and Erich Gamma (one of the author of the “[Gang of Four](#)” book) applied the same concepts to a Java library called **JUnit**, that to this day is the most popular testing library for Java.
- The success of JUnit inspired several xUnit frameworks for other languages, e.g., just to name a few: [Google Test](#) (C / C++), [NUnit](#) (C#), [unittest](#) (Python), [PHPUnit](#) (PHP). Many others exist, for languages and/or frameworks, for instance see Wikipedia’s “[List of unit testing frameworks](#)”.

xUnit — concepts



- Test (case)
 - Specification of an individual test.
 - “[Four-phase test](#)” is the common test programming pattern: setup > exercise SUT > verify > teardown.
- Test execution
 - Execution of a single test.
- Test fixture
 - Setup and teardown actions, necessary to setup the initial state of a SUT (before a test runs) and tear-it down (after a test runs).
- Test suite
 - Set of tests that can run in any order and share the same test fixture.
- Test runner
 - Program that runs test suites.

Four-phase test pattern



How It Works

We design each test to have four distinct phases that are executed in sequence. The four parts are **fixture setup**, **exercise SUT**, **result verification** and **fixture teardown**.

- In the first phase, we set up the **test fixture** (the "before" picture) that is required for the SUT to exhibit the expected behavior as well as anything you need to put in place to be able to observe the actual outcome (such as using a **Test Double** (page X).)
- In the second phase, we interact with the SUT.
- In the third phase, we do whatever is necessary to determine whether the expected outcome has been obtained.
- In the fourth phase, we tear down the **test fixture** to put the world back into the state in which you found it.

Source: xunitpatterns.com — companion site to “XUnit test patterns - Refactoring Test Code” book by G. Meszaros (Addison Wesley Signature Series curated by Martin Fowler)

JUnit — example

```
import static org.junit.Assert.*;
import static qses.ArrayOperations.numZero;
import org.junit.Test;
```

imports

```
public class ArrayOperationsNumZeroTest {
```

test class

```
@Test
```

```
public void testNumZeroEmptyArray() {
    int x[] = {}; // zero-sized array
    int n = numZero(x);
    assertEquals("0 zeros", 0, n);
}
```

test method
has @Test annotation
- one per test case -

```
@Test
```

```
public void testNumZeroArrayWithNoZeros() {
    int[] x = { 1, 2, 3 };
    int n = numZero(x);
    assertEquals("0 zeros in an array with no zeros", 0, n);
}
```

another test
method/case

```
...
```

JUnit - test methods

```
@Test
public void testNumZeroArrayWithNoZeros() {
    int[] x = { 1, 2, 3 };
    int n = numZero(x);
    assertEquals("0 zeros in an array with no zeros", 0, n);
}
...
```

1) setup test case values

2) execute SUT

exec. output

3) assert expected vs. test outputs

expected

■ Test design atterns

- Setup + execute SUT + verify expected results (+ teardown)
 - Use assertion methods provided by JUnit to verify expected results
 - Use assertion messages together with assertion methods to give an indication of what went worn
- [“Four-phase test”](#), [“Assertion Method”](#), and [“Assertion Message”](#) patterns [see also G. Meszaros, pages 358-372]

JUnit assertions

Method	Checked condition
<code>assertEquals(msg, expected, value)</code> <code>assertNotEquals(msg, expected, value)</code>	<code>value.equals(expected)</code> <code>!value.equals(expected)</code>
<code>assertTrue(msg, value)</code> <code>assertFalse(msg, value)</code>	<code>value == true</code> <code>value == false</code>
<code>assertNull(msg, expression)</code> <code>assertNotNull(msg, expression)</code>	<code>value == null</code> <code>value != null</code>
<code>assertArrayEquals(msg, vExp, vVal)</code>	Arrays <code>vExp</code> and <code>vVal</code> have the same contents.
<code>assertSame(msg, expected, value)</code>	<code>value == expected</code> (exactly the same object reference)

- Full list at <http://junit.org/junit4/javadoc/latest/index.html>

JUnit: setup/teardown

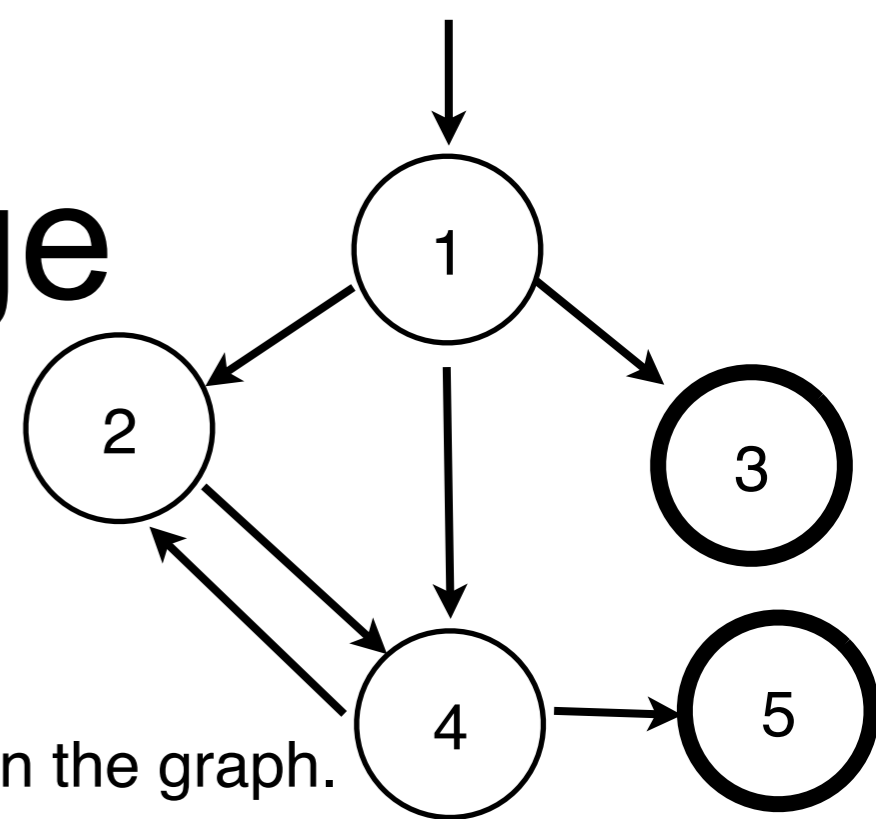
```
@BeforeClass
public static void globalSetup() {
    // executed once before all test
    ...
}
@AfterClass
public static void globalTeardown() {
    // executed once after all tests
    ...
}
@Before
public void perTestSetup() {
    // executed every time before each test
    ...
}
@After
public static void perTestTeardown() {
    // executed every time after each test
    ...
}
```

**Beyond basic
coverage criteria**

Testing approaches

- Related questions :
 - What are good tests?
 - What are meaningful inputs?
 - What (coverage) criteria should be used to derive them?
- Line/instruction/branch coverage
 - Easy to understand and measure through program instrumentation,
 - The most common metrics for coverage assessment in practice.
 - Fragile however in the sense of possibly conveying a false notion regarding the quality of inputs/tests and their ability to expose bugs.
- We will briefly look at a few approaches that go further ...
 - Graph-based coverage, input space partitioning, mutation testing, property-based testing

Graph-based coverage



■ Basic approach

- Model the SUT as a graph.
- The execution of a test case corresponds to a path in the graph.
- Coverage criteria specify requirements as sets of paths that must be covered by test paths.

■ Graphs as models for:

- individual procedures — control flow graphs (discussed next)
- interacting units — call graphs
- finite-state machine abstractions of software

■ Structural vs. data-flow based coverage

- Structural: takes into account only structure of the graph (example application next)
- Data-flow based: also account for data usage in association to nodes/edges (we won't cover this)

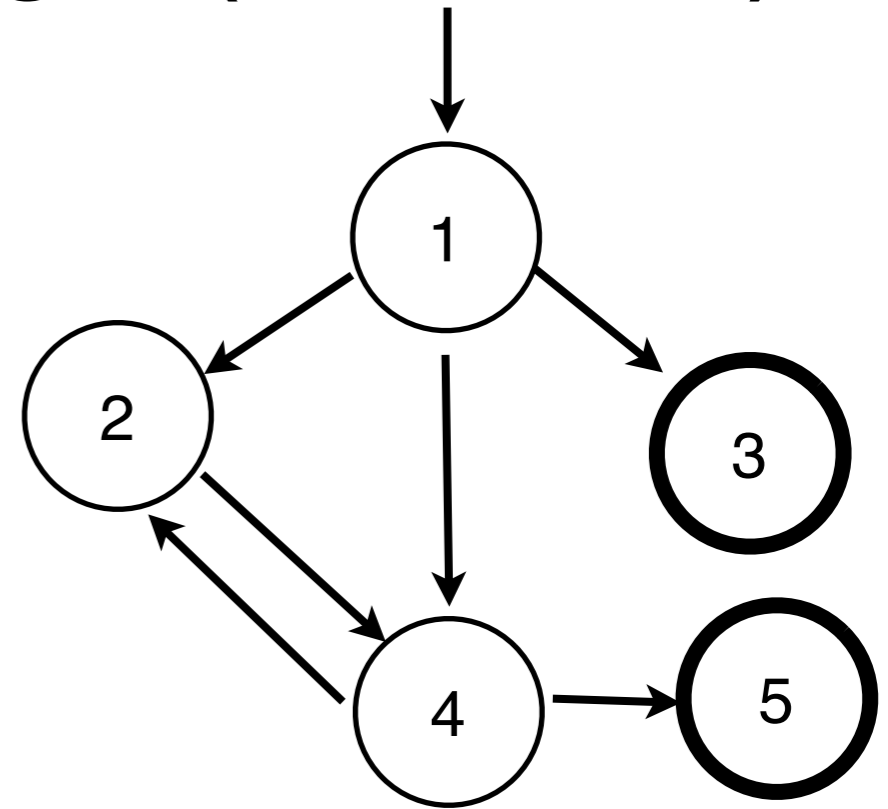
Node and edge coverage (NC, EC)

$TR(NC) = \{ [1], [2], [3], [4], [5] \}$

$TR(EC) = \{ [1,2], [1,3], [1,4], [2,4], [4,2], [4,5] \}$

$T1 = \{ [1,3], [1,2,4,5] \}$ satisfies NC, but not EC

$T2 = \{ [1,3], [1,2,4,5], [1,4,2,4,5] \}$
satisfies both NC and EC



■ Node coverage (NC)

- Test requirements: cover every node (all graph paths up of length 0)
- $TR(NC)$ = set of nodes in the graph

■ Edge coverage (EC): cover every edge.

- Test requirements: cover every edge (all paths up to length 1)
- $TR(EC)$ = set of edges in the graph

■ EC subsumes NC. Why?

Control flow graph (CFG)

- A **control flow graph (CFG)** can be used to represent the control flow of a piece of (imperative) source code.
 - **Nodes** represent **basic blocks** - sequences of instructions that always execute together in sequence.
 - **Edges** represent **control flow** between basic blocks.
 - The **entry node** corresponds to a method's entry point.
 - **Final nodes** correspond to exit points, e.g. in Java: `return` or `throw` instructions.
 - **Decision nodes** represent choices in control flow - e.g. in Java: due to `if`, `switch-case` blocks or condition tests for loops.

Example

```
public static int occurrences(char[] v, char c) {  
    if (v == null) {  
        throw new IllegalArgumentException();  
    }  
    int n = 0;  
    for (int i=0; i < v.length; i++) {  
        if (v[i] == c) {  
            n++;  
        }  
    }  
    return n;  
}
```

➔ Basic blocks (nodes)

- ➔ 1: if (v == null)
- ➔ 2: throw ...;
- ➔ 3: n=0; i=0;
- ➔ 4: i < v.length;
- ➔ 5: v[i] == c;
- ➔ 6: n++;
- ➔ 7: i++;

➔ Entry node

- ➔ 1

➔ Decision nodes

- ➔ 1, 4, 5

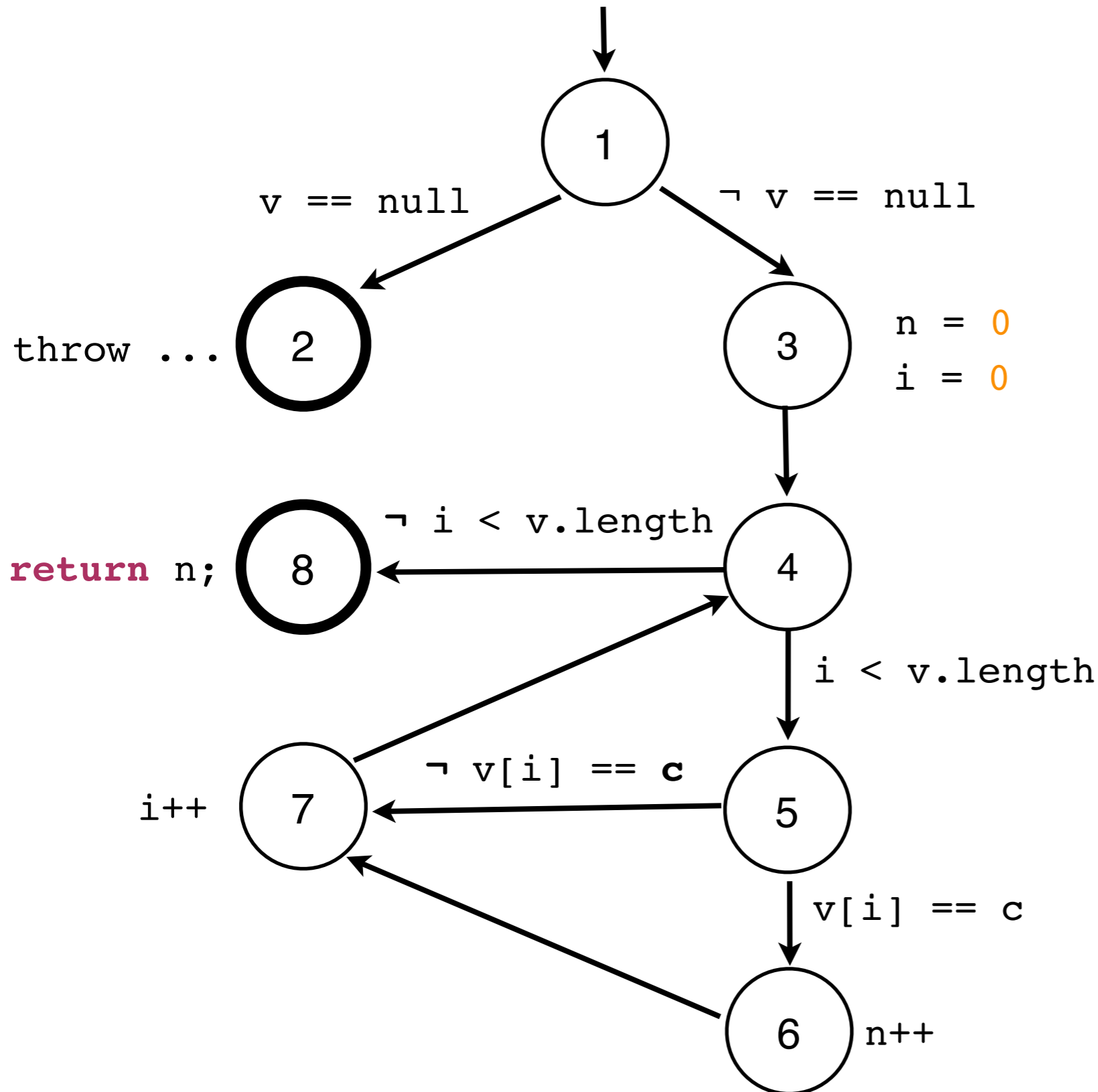
➔ Exit nodes

- ➔ 2, 8

➔ Control flow (edges)

- ➔ 1 → 2, 1 → 3
- ➔ 3 → 4
- ➔ 4 → 5, 4 → 8
- ➔ 5 → 6, 5 → 7
- ➔ 6 → 7
- ➔ 7 → 4

CFG for occurrences ()

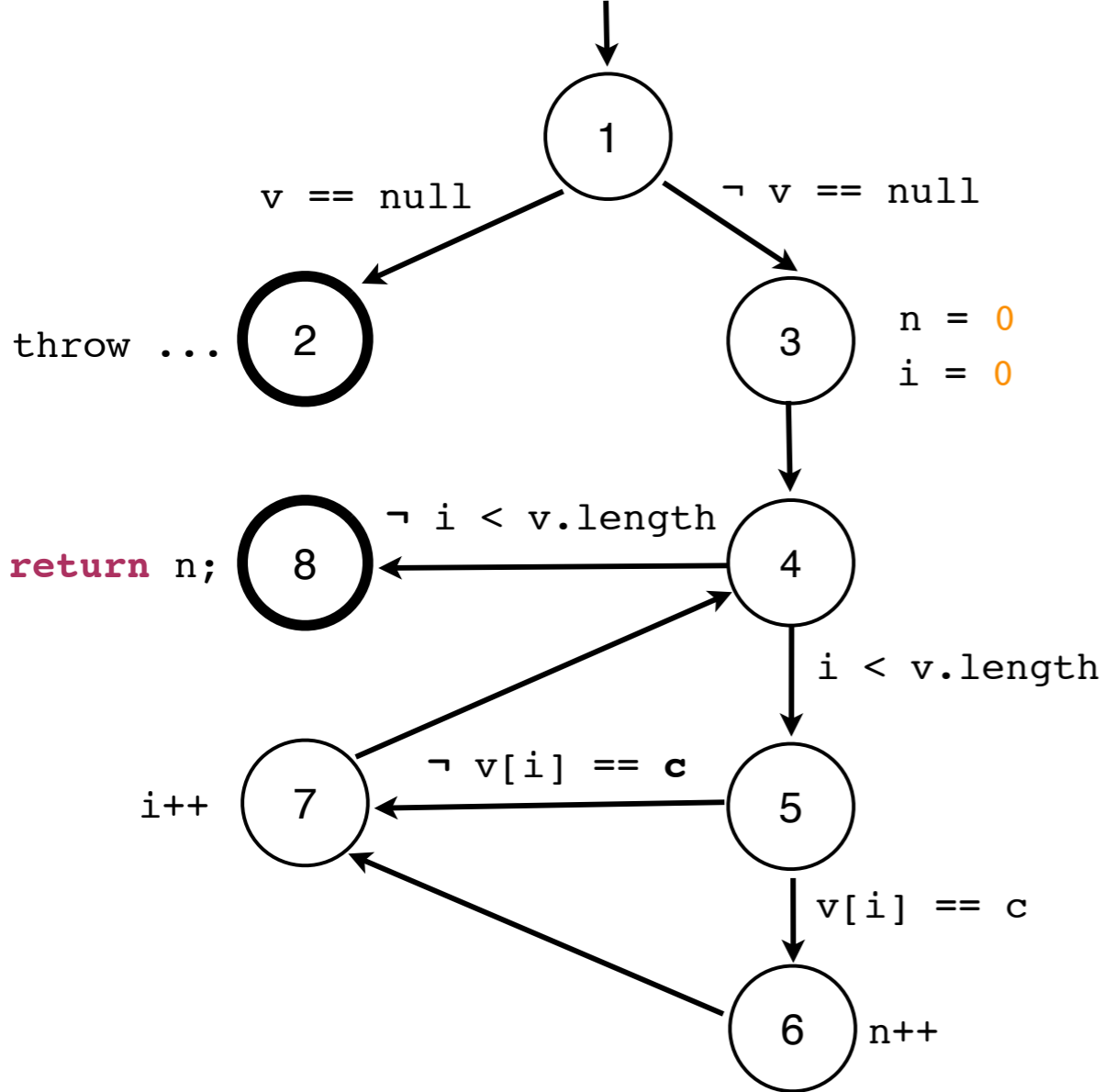


➔ Basic blocks (nodes)

- ➔ 1: `if (v == null)`
- ➔ 2: `throw ...;`
- ➔ 3: `n=0; i=0;`
- ➔ 4: `i < v.length;`
- ➔ 5: `v[i] == c;`

➔ Control flow (edges)

- ➔ 1 → 2, 1 → 3
- ➔ 3 → 4
- ➔ 4 → 5, 4 → 8
- ➔ 5 → 6, 5 → 7
- ➔ 6 → 7
- ➔ 7 → 4



Node coverage

TR(NC) = { [1],[2],[3],[4],[5],[6],[7],[8] }

NC satisfied by { t1, t2 } or {t1, t3}

Edge coverage

TR(EC) = TR(NC) U {
 [1,2],[1,3],[3,4],[4,5],[4,8], [5,6],[5,7],[6,7],[7,4]
 }

EC satisfied by { t1, t3 } but not by {t1,t2}.

t	test case values (v,c)	exp. values	test path	covered nodes	covered edges
t1	(null, 'a')	IAE	[1,2]	1 2	[1,2]
t2	({'a'}, 'a')	1	[1,3,4,5,6,7,4,8]	1 3 4 5 6 7 8	[1,3][3,4][4,5][5,6] [6,7][7,4][4,8]
t3	({'x','a'}, 'a')	1	[1,3,4,5,7,4,5,6,7,8]	1 3 4 5 6 7 8	[1,3][3,4][4,5][5,6] [6,7][7,4][5,7][4,8]

Beyond node/edge coverage

- **Edge-pair coverage (EPC)** - cover all paths up to length 2
 - **EPC** subsumes **NC** and **EC**
- **NC, EC, EPC** are instances of the general criterion: cover all paths up to length k
 - **NC** for $k=0$; **EC** for $k=1$; **EPC** for $k=2$;
- As we increase k we approximate ... **Complete-Path-Coverage (CPC)**
 - **CPC**: Cover all possible paths.
 - The number of paths may be infinite or very large e.g., code with loops (CFGs with cycles) - CPC generally not applicable.
 - In practice, instead of “increasing k ”, we should try to pick a subset of “relevant” paths in the graph, e.g., criteria like Prime Path Coverage [Amman & Offutt].

Mutation testing

```
public static int numZero(int[] x) {  
    int count = 0;  
    for (int i = 0; i < x.length; i++)  
        if (x[i] == 0)  
            count++;  
    return count;  
}
```

Introduce "faults"
by mutating the code.
What's the point?

```
public static int numZero(int[] x) {  
    int count = 0;  
    for (int i = 1; i < x.length; i++)  
        if (x[i] == 0)  
            count++;  
    return count;  
}
```


The premise for mutation testing

- **Fundamental premise of mutation testing**

- ✦ *“if the software contains a fault, there will usually be a set of mutants that can only be killed by a test case that also detects the fault”* [provided we consider a rich set of mutation operators], Ammann and Offutt

sensitivity to mutations (killing mutants)

\approx

sensitivity to faults (exposing failures)

“Testing the tests”

- Suppose you have a test set T for program P (maybe derived applying some coverage criteria C , manually or automatically).
- Program-based mutation testing helps answering the following key question:
 - *How “good” is T (and C)?*
- For $m \in M$ (the set of all mutants), if T is “good” then a test in T should kill m .
- If no test in T kills a mutant m , then T should be reformulated (one may also question the choice of C)...
- **Program-based mutation is many times taken as the “golden standard” of coverage criteria, given its potential to subsume other testing criteria.**

Killing the mutants ...

```
public static int numZero(int[] x) {  
    int count = 0;  
    for (int i = 1; i < x.length; i++)  
        if (x[i] == 0)  
            count++;  
    return count;  
}
```

- **i = 1** is a **mutation** of **i = 0** ; the code obtained by changing **i=0** to **i=1** is called a **mutant** of numZero.
- We say a test **kills the mutant** if the mutant yields different outputs from the original code.
 - Considering **x={1,0,0}** the mutant is **not killed**; **2** is the return value of the method for both the original code and the mutant.
 - Considering **x={0,1,0}** the mutant is **killed**; the result is **1** rather than **2**.

```

public static
int min(int x, int y) {
    int v;
    if (x < y)
        v = x;
    else
        v = y;
    return v;
}

```

original code

Example 2

Which mutants will be killed by tests:

(t1) $(x,y) = (0,0)$

(t2) $(x,y) = (0,1)$

(t3) $(x,y) = (2,1)$

Observe that **m2** can not be killed. Why not?

```

public static
int min(int x, int y) {
    int v;
    if (x >= y)
        v = x;
    else
        v = y;
    return v;
}

```

m1

```

public static
int min(int x, int y) {
    int v;
    if (x <= y)
        v = x;
    else
        v = y;
    return v;
}

```

m2

```

public static
int min(int x, int y) {
    int v;
    if (x < y)
        v = x;
    else
        v = -y;
    return v;
}

```

m3

mutants

	x	y	min	m1	m2	m3
t1	0	0	0	0	0	0
t2	0	1	0	1	0	0
t3	2	1	1	2	1	-1

t1 kills none of the mutants.

t2 kills m1.

t3 kills m1 and m3.

Observe that m2 will always yield the same result as the original code. Thus it cannot be killed. It is a **functionally equivalent mutant**.

```
public static
int min(int x, int y) {
    int v;
    if (x >= y)
        v = x;
    else
        v = y;
    return v;
}
```

m1

```
public static
int min(int x, int y) {
    int v;
    if (x <= y)
        v = x;
    else
        v = y;
    return v;
}
```

m2

```
public static
int min(int x, int y) {
    int v;
    if (x < y)
        v = x;
    else
        v = -y;
    return v;
}
```

m3

Mutation operators from PIT

Math Mutator (MATH)

Active by default

The math mutator replaces binary arithmetic operations for either integer or floating-point arithmetic with another operation. The replacements will be selected according to the table below.

Original operation	Mutated operation
+	-
-	+
*	/
/	*
%	*
&	
	&
^	&
<<	>>
>>	<<
>>>	<<<

For example

```
int a = b + c;
```

will be mutated to

```
int a = b - c;
```

<http://pitest.org>

Mutation operators from PIT (2)

Conditionals Boundary Mutator (CONDITIONALS_BOUNDARY)

Active by default

The conditionals boundary mutator replaces the relational operators `<`, `<=`, `>`, `>=`

with their boundary counterpart as per the table below.

Original conditional	Mutated conditional
<code><</code>	<code><=</code>
<code><=</code>	<code><</code>
<code>></code>	<code>>=</code>
<code>>=</code>	<code>></code>

For example

```
if (a < b) {  
    // do something  
}
```

will be mutated to

```
if (a <= b) {  
    // do something  
}
```

<http://pitest.org>

Mutation operators and effectiveness

■ Mutants to avoid ...

- **stillborn mutant** (i.e., dead at birth): mutant is not syntactically valid
- **functionally-equivalent mutant**: no test can kill it
- **trivial mutant**: almost every test can kill it

■ For effectiveness, a mutation operator should:

- always define a syntactically valid transformation (generate no stillborn mutants)
- generate functionally-equivalent and trivial mutants with low probability
- mimic typical programmer mistakes
- not be subsumed by another operator i.e., tests that kill mutants created by the other operator also kill the ones generated by this one (or a large fraction of them)

Mutation testing - coverage

- **Mutation operator** o : takes a program P and yields a set of mutants of p , $o(p)$.
- Let O be the set of mutation operators and M be the set of all mutants generated using O i.e., $M = \{ m \mid m \in o(p), o \in O \}$
- **Killing mutants**
 - We say a test t **kills** $m \in M$ iff the output of t for m differs from the output of t for P .
- **Mutation coverage** = percentage of mutants in M killed by at least one test.

Mutation testing tools - basics

- A MT tool has a built-in set of mutation operators. The set of mutants for the SUT is generated in automated manner according to the mutation operators.
- A test set in context is ran against the mutants. As soon as a mutant from the set is killed, it is typically not exercised by further tests.
- If the mutation coverage is not satisfactory, the test set is typically revised and/or increased with further test cases.
 - **Obs:** The strategies for both mutant generation and test selection/execution can be quite elaborate in technical terms.

Property-based testing

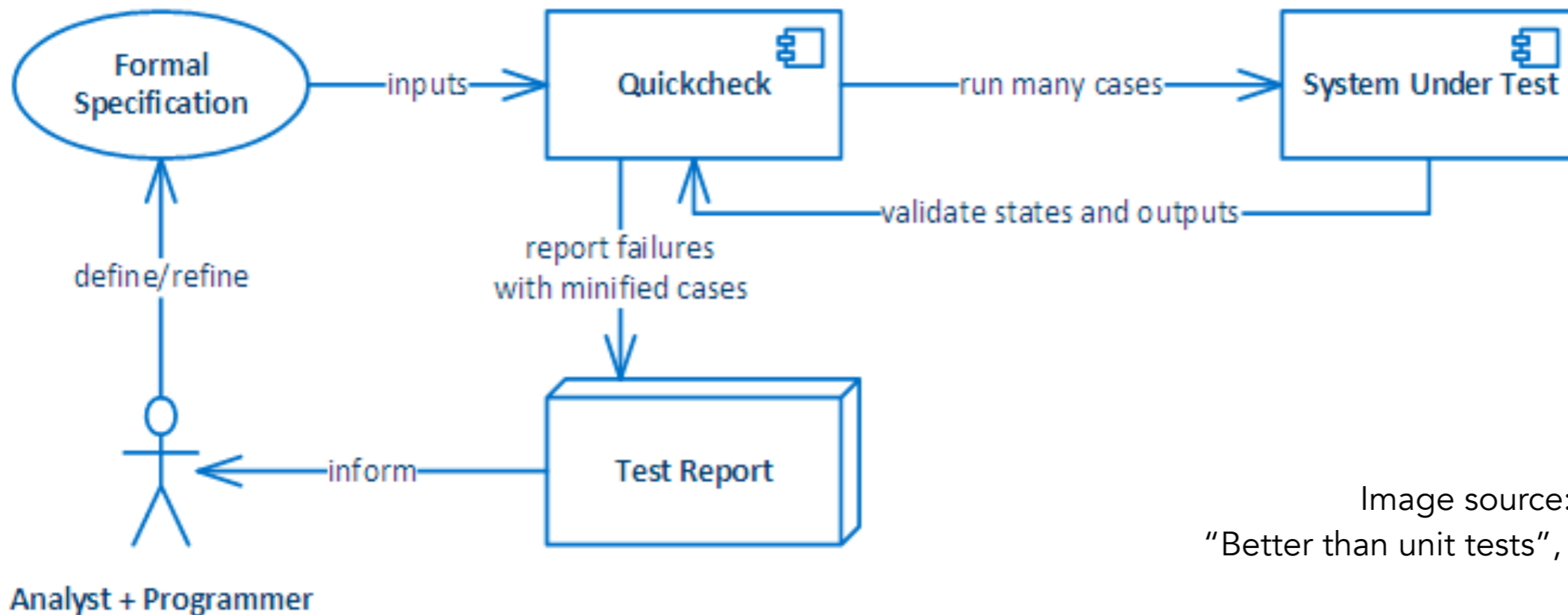


Image source:
"Better than unit tests", M. Nygard

■ Approach:

- Specify **properties to check** instead of inputs !
- A great number of test cases are generated **automatically**, in line with the specification.
- If a property fails for a certain input, try to find the **minimal input** that violates the property, a process designated as **shrinking**.
- Original formulation: "[QuickCheck: a lightweight tool for random testing of Haskell programs](#)", Koen Claessen and John Hughes, Proc. ICFP, 2000. Other property-based testing frameworks: [scalacheck](#) (Scala/Java), [Hypothesis](#) (Python), Java (QuickTheories), ...

Validation of a Tiny Encryption Algorithm (TEA) [implementation](#) using [QuickTheories](#) (for Java 8)

```
@Test
public void testTEAWithFixedKey() {
    TEA obj = new TEA("0123456789ABCDEF".getBytes());
    qt()
    .forAll(byteArrays(integers().between(1,256),
        bytes(Byte.MIN_VALUE, Byte.MAX_VALUE, (byte) 0)))
    .describedAs(data -> Arrays.toString(data))
    .check( data -> Arrays.equals(data, obj.decrypt(obj.encrypt(data))));
}
```

fixed encryption key, but generator used for data (random byte array with length between 1 and 256)

Property: $\forall \text{data}, \text{decrypt}(\text{encrypt}(\text{data})) = \text{data}$

```
@Test
public void testForAnyKey() {
    Gen<Byte> anyByte = bytes(Byte.MIN_VALUE, Byte.MAX_VALUE, (byte) 0);
    Gen<byte[]> keyGen = byteArrays(constant(16), anyValue)
        .describedAs(Arrays::toString);
    Gen<byte[]> dataGen = byteArrays(integers().between(1, 100),
anyValue).describedAs(Arrays::toString);

    qt()
    .forAll(keyGen, dataGen)
    .check( (key, data) -> {
        TEA tea = new TEA(key);
        return Arrays.equals( tea.decrypt(tea.encrypt(data)), data);
    });
}
```

variable key also

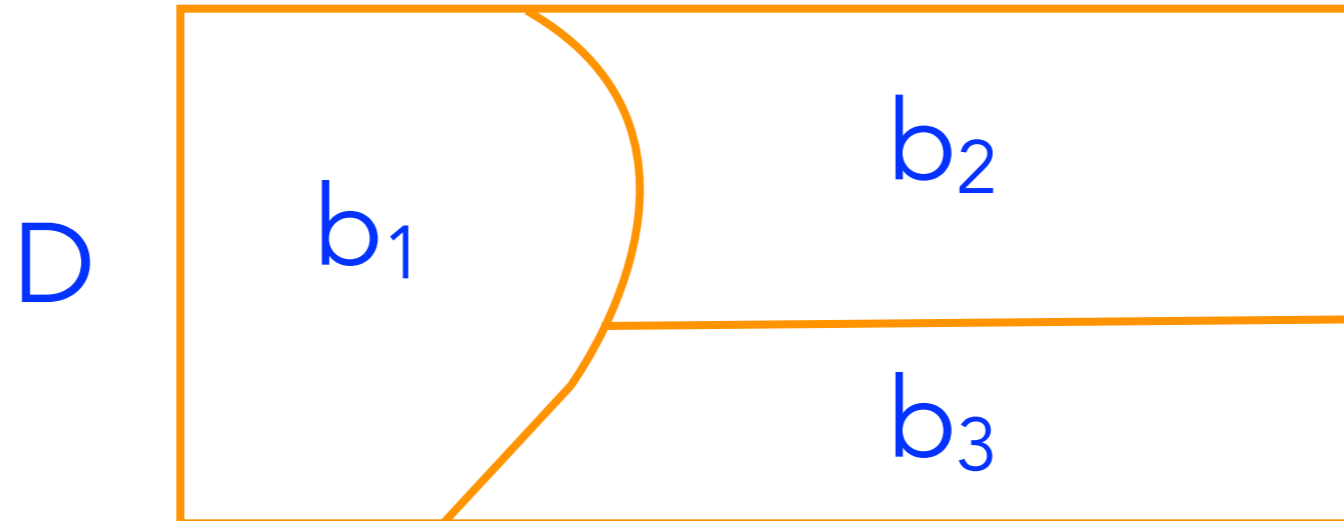
QuickTheories (example 2)

```
@Test
public void testValidPasswordNoPunct() {
    Gen<Byte> lo = bytes( (byte)'a', (byte)'z', (byte)'a' );
    Gen<Byte> up = bytes( (byte)'A', (byte)'Z', (byte)'A' );
    Gen<Byte> digit = bytes( (byte)'0', (byte)'9', (byte)'0' );
    Gen<Byte> combined = lo.mix(up,50).mix(digit,25);
    Gen<byte[]> arrGen = byteArrays(integers().between(10, 20), combined);
    Gen<String> strGen = arrGen.map(ba -> new String(ba));
    qt()
        .withFixedSeed(0)
        .forall(strGen)
        .assuming(s -> s.chars().anyMatch(Character::isLowerCase))
        .assuming(s -> s.chars().anyMatch(Character::isUpperCase))
        .assuming(s -> s.chars().anyMatch(Character::isDigit))
        .check(CHECKER::isPasswordOK);
}
```

Input space partitioning (ISP)

- **Base approach:** identify relevant classes of input values and derive test cases from it.
- **Step 1. Identify the input parameters** for the SUT.
- **Step 2. Model the input domain** by defining one or more characteristics in the input domain. Each characteristic defines blocks that partition the input space.
- **Step 3. Apply some criterion** over characteristic of the input domain, defining a set of test requirements.
- **Step 4.** Derive test inputs (test cases).
- Also known as equivalence partitioning.

ISP - definitions



- **Input domain (D):** the set of possible values for the input parameters.
- A **characteristic** q for D is a partition of D . It defines **blocks** b_1 , \dots , b_n such that :
 - $\forall i, j : i \neq j \quad b_i \cap b_j = \emptyset$ **(blocks are disjoint)**
 - $D = b_1 \cup \dots \cup b_n$ **(blocks cover the entire input domain)**
- **Q** : the set of characteristics we consider to derive test requirements.

ISP - guidelines

- **Meaningful characteristics:** Each characteristic should represent a meaningful feature for the input domain.
- **Distinctive blocks:** blocks of a characteristic should be reasonably aligned with distinctive values for it, e.g., consider:
 - “common use” values
 - boundary values
 - “invalid use” values
 - relevant relations between input parameters
- **Subdomains:** if necessary break down domain into sub-domains
 - E.g. first partition into “valid” and “invalid” values, then define characteristics for each of these domains, or sub-partition them further if convenient.

isPasswordOK example

```
/**  
 * Test if password is OK.  
 * @param password The password  
 * @return <code>>true</code> is password is OK.  
 */  
boolean isPasswordOK(String password);
```

We wish to implement a password setting policy scheme, where a password is considered "OK" if all of the following conditions are met:

1. The length is between 10 and 20 characters.
2. It contains at least one upper-case character.
3. It contains at least one lower-case character.
4. It contains at least one decimal digit.
5. It may (but need not) contain at most one punctuation symbol, i.e., one of the following . : ? ! , ;.
6. It does not contain any character that does not fit in one of the above categories.

isPasswordOK example (2)

- Null vs non-null characteristic
 - Breaks domain into “null” sub-domain and “non-null” subdomain
- For the “non-null” subdomain we may consider:
 - l = Length of password
 - U = # upper-case characters
 - L = # lower-case characters
 - D = # digits
 - P = # punctuation characters
 - I = # invalid symbols

isPasswordOK example (3)

- Possible blocks for the length characteristic (l)
 - $l < 10$, $10 \leq l \leq 20$, $l > 20$ (3 blocks)
 - The blocks must define a partition. Thus, the block values do not intersect and we cannot rule out any possible value of L.
 - A more fine-grained choice could consider $l=10$ and $l=20$ blocks to force testing of boundary values for length.
- A possible choice of blocks for the $x = U, L, D$, and I characteristics
 - $x=0$, $x > 0$ (2 blocks each)
- Finally, for P (the punctuation characters)
 - $P = 0$, $P = 1$, $P > 1$

isPasswordOK example (4)

- Input “Ab1234567890” fits in the following blocks:
 - $10 \leq l \leq 20$ (the length is 12)
 - $U > 0$
 - $L > 0$
 - $D > 0$
 - $P = 0$
 - $I = 0$
- What are the blocks for “ABxy12!\$?” ?

ISP coverage criteria

■ *t*-wise coverage (TWC)

- Cover *t* blocks of different characteristics by at least one test case.

■ Each Choice Coverage (ECC) [*t*=1]

- Cover each block of each characteristic at least once.

■ Pair-wise Coverage (ECC) [*t*=2]

- Cover each block pair of two different characteristic at least once.

■ All-Combinations Coverage (ECC) [*t* = number of characteristics]

- Cover each combinations of blocks of different characteristic at least once.

ECC coverage for isPasswordOK

- A few tests are enough, for instance:
 - "Ab1234567890" covers blocks $10 \leq l \leq 20$, $U > 0$, $L > 0$, $D > 0$, $P = 0$, $I = 0$
 - "!@" covers blocks $1 < l < 10$, $U = 0$, $L = 0$, $D = 0$, $P = 1$, $I > 0$
 - "!!" covers blocks $l > 20$, $U = 0$, $L = 0$, $D = 0$, $P > 1$, $I = 0$

PWC coverage for isPasswordOK

- “Ab1234567890” will cover 15 block pairs (5 + 4 + 3 + 2 + 1)
 - (10 <= 1 <= 20, U > 0), (10 <= 1 <= 20, L > 0),
(10 <= 1 <= 20, D > 0) (10 <= 1 <= 20, P = 0),
(10 <= 1 <= 20, I = 0)
 - (U > 0, L > 0), (U > 0, D > 0), (U > 0, P = 0),
(U > 0, I = 0)
 - (L > 0, D > 0), (L > 0, P = 0), (L > 0, I = 0),
 - (D > 0, P = 0), (D > 0, I = 0)
 - (P = 0, I = 0)
- Covering all block pairs will require more test cases.

ISP - test effort vs coverage

■ ECC

- $\sum_{i=1, \dots, |Q|} |B_i|$ test requirements, at least $\max_{i=1, \dots, |Q|} |B_i|$ tests.
- **isPasswordOK: ≥ 3 tests**

■ PWC

- $\sum_{i,j=1, \dots, |Q|, i \neq j} |B_i| \cdot |B_j|$ requirements, at least M^2 tests for $M = \max_{i=1, \dots, |Q|} |B_i|$
- **isPasswordOK: $\sim 3 \times 3 = 9$ tests**

■ ACoC

- $\prod_{i=1, \dots, |Q|} |B_i|$ test requirements and as many tests required
- **isPasswordOK: $3 \times 2 \times 2 \times 2 \times 2 \times 2 \times 3 = 144$ tests**

Security-oriented testing

Security testing

- How does security testing differ from standard program testing?
 - **Security features** must be tested with regard to possible adversarial actions => guided by the requirements posed to security-minded code.
 - Other features must be tested in respect to “**unintended behavior**” => guided by the possibility of common vulnerabilities in standard code
- What are general recommended practices? How can standard or specific testing approaches help?
- As an introductory discussion, let's have a look at some of the activities in the [BSIMM Security Testing touchpoint](#).

BSIMM - Security Testing

- *[ST1.1: 100] “Ensure QA supports **edge/boundary value condition testing**”*
 - *“The QA team goes beyond functional testing to perform basic adversarial tests and probe simple **edge cases and boundary conditions**, no attacker skills required.”*
- *[ST1.3: 88] “Drive tests with **security requirements and security features**.”*
 - *“For the most part, security features can be tested in a fashion similar to other software features.”*
- **Standard software testing practices apply. We will see a few important approaches in this sense.**

BSIMM - Security Testing (cont.)

- “[ST3.4: 3] **Leverage coverage analysis**”
 - *“Testers measure the code coverage of their security tests. Code coverage analysis drives increased security testing depth.”*
- “[ST2.5: 12] **Include security tests in QA automation**”
 - *“Security tests run alongside functional tests as part of **automated regression testing**. In fact, the same automation framework houses both, and security testing is part of the routine.”*
- **Again, standard software testing practices apply.**
- [ST2.5: 12] mentions regression testing ? **Q:** Are you familiar with it?

A note on regression testing

- **Regression testing:** testing if updated software still behaves the same (in regard to unchanged requirements) after a change in its code (bug fix, new feature) or after integration with an updated version of an external component.
- Ideally, a regression test suite is composed of a minimal set of “core tests” that always run in automated fashion whenever an update takes place.
- In a large code base, determining which tests should be part of (or removed from) the regression suite is not straightforward. Running regression test suites (if too big) for instance on every build may also be quite costly.

BSIMM - Security Testing (cont.)

- *[ST2.1: 30] “Integrate **black-box security tools** into the QA process.”*
 - *“The organization uses one or more black-box security testing tools as part of the QA process. Such tools are valuable because they encapsulate an **attacker’s perspective**, albeit generically”.*
 - Some commercial frameworks are mentioned in the text combining static/dynamic analysis, pen-testing and **fuzz testing**.
- *[ST2.6: 13] Perform **fuzz testing customized to application APIs**.*
 - *“Test automation engineers or agile team members customize a **fuzzing framework** to the organization’s APIs. The fuzzing framework has a built-in understanding of the application interfaces it calls into.”*
- **Fuzz testing? A common practice in security-oriented testing, but not in standard program testing. We will see what it means.**

BSIMM - Security Testing (cont.)

- “[ST3.3: 4] Drive tests with **risk analysis** results.”
 - “Testers use architecture analysis results to direct their work [...] Adversarial tests like these can be developed according to risk profile, with high-risk flaws at the top of the list.”
- “[ST3.5: 3] Begin to build and **apply adversarial security tests (abuse cases)**.”
 - “Testing begins to incorporate test cases based on abuse cases”
- These are important aspects mentioned earlier in the course. They require expertise and overall understanding of:
 - the software architecture and design
 - the inherent attack surface
 - the SDLC process in place

BSIMM testing activities — state of practice (2018)

SECURITY TESTING (ST)		
ACTIVITY DESCRIPTION	ACTIVITY	PARTICIPANT %
LEVEL 1		
Ensure QA supports edge/boundary value condition testing.	ST1.1	83.3
Drive tests with security requirements and security features.	ST1.3	73.3
LEVEL 2		
Integrate black-box security tools into the QA process.	ST2.1	25.0
Share security results with QA.	ST2.4	11.7
Include security tests in QA automation.	ST2.5	10.0
Perform fuzz testing customized to application APIs.	ST2.6	10.8
LEVEL 3		
Drive tests with risk analysis results.	ST3.3	3.3
Leverage coverage analysis.	ST3.4	2.5
Begin to build and apply adversarial security tests (abuse cases).	ST3.5	2.5

- Source: [BSIMM 9](#) - Gary McGraw, Ph.D., Sammy Miguez, and Jacob West
- “[the] result of a multiyear study of real-world software security initiatives We present the BSIMM8 model as built directly out of data observed in 109 software security initiatives”

Fuzzing (fuzz testing)

Fuzzing

■ What is fuzzing ?

- Testing software with invalid and possibly malicious data, usually generated in semi-automatic manner.

■ What is the goal of fuzzing?

- Evaluate program response to invalid input, rather than “common case” inputs used for plain functional testing.

■ Optimal response to invalid inputs:

- a graceful failure — in line with the “Fail Safely” design principle. Nothing “unintended” or “bad” happens!

■ Vulnerable responses to invalid input may include (possibly a combination of):

- program crashes, memory corruption (e.g. buffer overflows). failure to detect the error in input

Deriving inputs

■ Deriving inputs — essential techniques

- **Randomisation:** generate random inputs, or introduce randomness during generation:
- **Mutation:** mutate given inputs according to some criteria
- **Grammar-based generation:** use a grammar to generate inputs
- Hybrid approaches combining these are common.

■ Fuzz-testing process

- **Black-box:** generate inputs and monitor execution result, blindly.
- **White-box:** guide input generation according to feedback from execution + information regarding program structure.

Random input

```
$ head -c 15 /dev/urandom | xargs ping  
ping: cannot resolve ?c?D?\fN\016?=?;?: Unknown host
```

- No context of the software at stake or the type of input.
- Easy to implement, but will typically expose only shallow bugs

Mutation-based input generation

- Start from valid inputs, mutate them according to some strategy for instance:
 - Applying randomisation, e.g., random bit flips.
 - More generally, applying mutation rules
 - Mutation fragments may be domain-specific, e.g., contain shell-code, SQL injection, etc.
- Ability to expose bugs: dependent on starting inputs and mutation expressiveness for the context at stake.
- Example tools next: radamsa, ZAP fuzzer, zzuf

Example tools — radamsa

```
$ echo 192.168.106.103 | radamsa --count 10 --seed 0
-107.167.106.103
192.168.8407971865571866.-9[?]5154737306362663942413194069
191.1A1.1A1.106.1
192.129.18.106.103
192.168.0.103
192.170141183460.106.1802311213346089.104
-3402823669209.106.168.106.16.103
192093846346337460765704.192.65704.-1.?-18446744073709518847
192.106.0
191.168.106.103
$ echo 192.168.106.103 | radamsa --count 1 --seed 0 | xargs ping
ping: invalid option -- 1
```

- [Radamsa](#): a mutation-based input generator
- Mutates given inputs, randomly applying pre-defined mutation rules and patterns.

Example tools — radamsa (2)

```
$ ./radamsa --list
Mutations (-m)
...
bd: drop a byte
bf: flip one bit
bi: insert a random byte
...
sr: repeat a sequence of bytes
sd: delete a sequence of bytes
ld: delete a line
...
ls: swap two lines
...
num: try to modify a textual number
xp: try to parse XML and mutate it
...
Mutation patterns (-p)
od: Mutate once
nd: Mutate possibly many times
bu: Make several mutations closeby once
```

- Example mutations and mutation patterns (listed with `radamsa --list`)

ZAP fuzzer

Select part of the input to “fuzz with”, in this case the “1” value that is part of the HTTP request header

```
GET
http://localhost:8081/vulnerabilities/sqli/?id=1&Submit=Submit HTTP/1.1
Proxy-Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.0
```

Select “fuzz set” of replacements for the chosen input, in this case strings likely to trigger SQLi, if a vulnerability of this kind exists

Several test cases will be considered for execution, each replacing ‘1’ by potentially malicious input

Files: Find Next Find Prev

- Active SQL Injection
- MS SQL Injection i
- MS SQL Ninja Injection (Blind)
- MySQL Injection (Blind)
- MySQL Injection 101
- MySQL/MS SQL Common Injection
- Oracle SQL Injection
- Passive SQL Injection
- SQL Injection
- URI Exploits
- User Agents

State	Payloads
☀ Reflected	' or '1'='1
☀ Reflected	' or '1'='1
☀ Reflected	' or '7659'='7659
☀ Reflected	' or '7659'='7659
☀ Reflected	' or 'a'='a
☀ Reflected	' or 1=1--
☀ Reflected	1;(load_file(char...
☀ Reflected	1' or '1'='1
☀ Reflected	1

Example programs - zzuf

```
zzuf -r 0.02 -s 1:3 cat ./silly_program.c

J'a|cl}de <st?i?.h>

inu`main(int avgc, char*? argw) {
    int l = 0;
    whidE("fgfgets*buf,sizeof(Buf-, f) != NULL- {
        pryntf(bt?;
    } dclose(f);
    retezn 0;J}

#include |stdio.h

i|t main(int aRfc, ch`r** argv) {
    ahar buf[128];
```

- **zzuf** automates the fuzzing process by **transparently fuzzing read from files or from the network.**
 - Mutations are introduced randomly according to a specified bit fuzzing ratio.
 - The target program runs in batch mode for a specified number of trials / seeds.
 - It has been successful in [uncovering bugs in real-world programs.](#)

Example programs - zzuf (2)

- In this case zzuf transparently mutates data from the network (use of the `-n` switch).

```
$ zzuf -r 0.02 -s 1 -n curl http://www.dcc.fc.up.pt/~edrdo/aulas/qses/index.html
% Total      % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Total     Spent    Left     Speed
100    328      0   328      0      0      60      0  --:--:--   0:00:05  --:--:--    0
HTTP/1.1 200 OK
Date: Wed, 13 Dec 2018 12:42:36 GMT
Server: Apache/2.4.18 (Ubuntu)
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block
X-Content-Type-Options: nosniff
Expires: Wed, 12 Dec 2007 05:40:54 GMT
Etag: "07-57bd?86197e5a"
Accept-Ranges: bytes
Content-Length: 71
Content-Type: text/html
```

```
<html>?<body>
```

“Fuzzed” execution

```
ZZUF! |est(resource
```

```
</body>
</html>
```

```

% Total      % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Total     Spent    Left     Speed
100    71    100    71      0      0     220      0  --:--:--  --:--:--  --:--:--  1145
```

```
<html>
<body>
```

```
ZZUF test resource -- QSES 2018/2019
```

```
</body>
</html>
```

Normal execution


Grammar-based input generation

- Generate inputs using a grammar.
 - Grammar rules may express possible deviations.
 - Combination with mutation: alternatively, valid inputs may be generated using a grammar, and then mutated.
 - This approach can be more systematic, is potentially able to generate more relevant inputs, and account for complex combinations of input fragments.
- Example tool illustrated next: [blab](#)
 - A few others of the same kind: [ABNFuzzer](#) [gramfuzz](#)

Example tools - blab

`ip_address.blab`

```
output = ip_address "\n"  
ip_address = octet "." octet "." octet "." octet  
octet = [0-9] | [1-9][0-9] | "1" [0-9][0-9] | "2" [0-4][0-9] | "25" [0-5]
```



```
$ blab ip_address.blab -n 10 -s 0  
4.4.4.104  
5.148.205.94  
0.237.230.95  
0.140.232.252  
178.81.250.6  
252.252.252.8  
135.159.123.250  
204.5.172.8  
177.188.21.213  
0.78.204.240
```

- **Blab**: a grammar-based black-box fuzzer
- Inputs generated according to grammar. In this example the grammar generates only valid IP addresses.

Example tools - blab (2)

`fuzzed_ip_address.blab`

```
output = fuzzed_ip_address "\n"
fuzzed_ip_address = octet "." octet "." octet "." octet
octet = normal_octet | fuzzed_octet
normal_octet = [0-9] | [1-9][0-9] | "1" [0-9][0-9] | "2" [0-4][0-9] | "25" [0-5]
fuzzed_octet = [0-9]{3}
```



```
$ blab fuzzed_ip_address.blab -n 10 -s 0
40.4.40.40
143.696.528.100
137.013.61.242
7.433.5.522
113.277.743.145
123.6.119.235
740.810.87.801
221.077.43.319
079.737.507.518
947.479.245.947
```

- In this variation we allow the possibility of malformed IP IP addresses.

Generate, then mutate

```
$ blab fuzzed_ip_address.blab -n 5 -s 0 | tee generated.txt  
40.4.40.40  
143.696.528.100  
137.013.61.242  
7.433.5.522  
113.277.743.145  
$ radamsa --count 1 --seed 22 generated.txt -p nd=10  
3321759348573678331568.4.40.40  
143.696.528.100  
1.013.61.0  
7.65535.9223372036854775803.522  
113.280.743.145
```

- Generation and mutation can be combined, e.g., blab + radamsa.

Black-box fuzzing

- **Simplest approach — “black box” fuzzing**
 - Repeatedly feed the program with fuzzed inputs, without consideration for the program structure.
 - Observe program responses and assert that program fails gracefully / nothing “bad” happens (crashes, memory corruption etc).
- **Looking for bugs — possible strategies**
 - Instrument the program with runtime sanitizers to monitor abnormal program execution (undefined behavior, buffer overflows, etc)
 - Inspect exit codes (e.g. SIGSEV = 139 — segmentation fault), program output, etc

White-box fuzzing

■ Idea

- Monitor (instrumented) program state during execution and observe which changes to input cause new program states to be explored.
- The information is used to generate new inputs, trying to avoid inputs that repeat the same program paths.

■ The goal is to explore the state-space of the program as extensively as possible / increase code coverage.

- The execution is automatic, but can be time-consuming given that many executions of the program under test will be triggered.
- Tools can derive inputs randomly or (with better results) through mutations of a pre-defined set of inputs that are accepted by the program.

■ Example tools:

- [AFL](#), [libFuzzer](#), [SAGE](#)

libFuzzer / AFL

■ libFuzzer, AFL

- The fuzzers are employed by [ClusterFuzz](#) and [Google's OSS-Fuzz project](#) (“continuous fuzzing of open source software”)
- Employ program instrumentation/monitoring coupled with input mutation techniques that are guided by runtime coverage information.
- The fuzzers are quite more effective if supplied with an initial corpus of input samples that are representative of the program execution.
- Normally used in combination with sanitizers to uncover bugs, e.g. AddressSanitizer.

libFuzzer — example test

```
extern char* escapeHTML(char* input);

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    if (size > 0) {
        // Ensure null termination
        char* buf = (char*) malloc(size+1);
        memcpy(buf, data, size);
        buf[size] = 0;

        // Call function to test
        char* edata = escapeHTML(buf);

        // Clean-up
        free(buf);
        free(edata);
    }
    return 0;
}
```

- **LLVMFuzzerTestOneInput**: test entry point
- **data**: fuzzed input from initial corpus or randomly generated

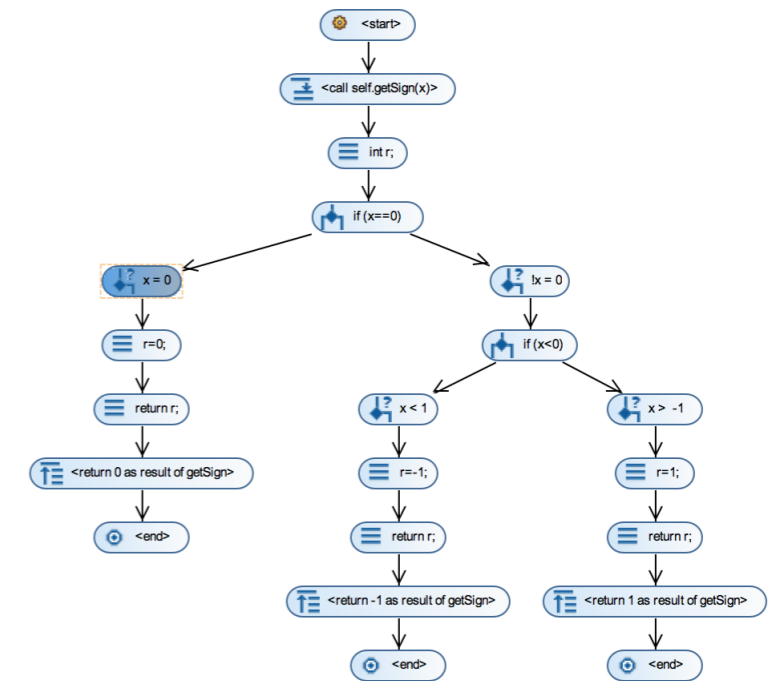
libFuzzer — example test (2)

```
$ clang -g -fsanitize=address,fuzzer escapeHtml.c fuzzTest.c -o fuzzTest
$ ./fuzzTest corpus seeds
INFO: Seed: 1148001052
INFO: Loaded 1 modules (13 inline 8-bit counters): 13 [0x5a5f70,
0x5a5f7d),
...
INFO: -max_len is not provided; libFuzzer will not generate inputs larger
than 4096 bytes
INFO: seed corpus: files: 1 min: 73b max: 73b total: 73b rss: 27Mb
#2 INITED cov: 12 ft: 13 corp: 1/73b exec/s: 0 rss: 27Mb
...
=====
==18004==ERROR: AddressSanitizer: heap-buffer-overflow on address
0x602000004dd5 at pc 0x00000054f5ac bp 0x7ffc444e1990 sp 0x7ffc444e1988
...
artifact_prefix='./'; Test unit written to ./
crash-0ed93db39ee78666320ef6bf9145483c206743d4
Base64:
JgAAALa2q6urq6urq21sJjwAAAAMPSYAJiY8AAAAAAAAAACs8AAAAJiYmJiYmJiYmJiYmJiYma
AAAAAAAAAAAAAAAAAACs8JnRtDw==
```

- Execution takes initial input samples (**seeds** dir) and derives more inputs (**corpus** dir). Bug is then eventually detected by **AddressSanitizer**.

Symbolic execution

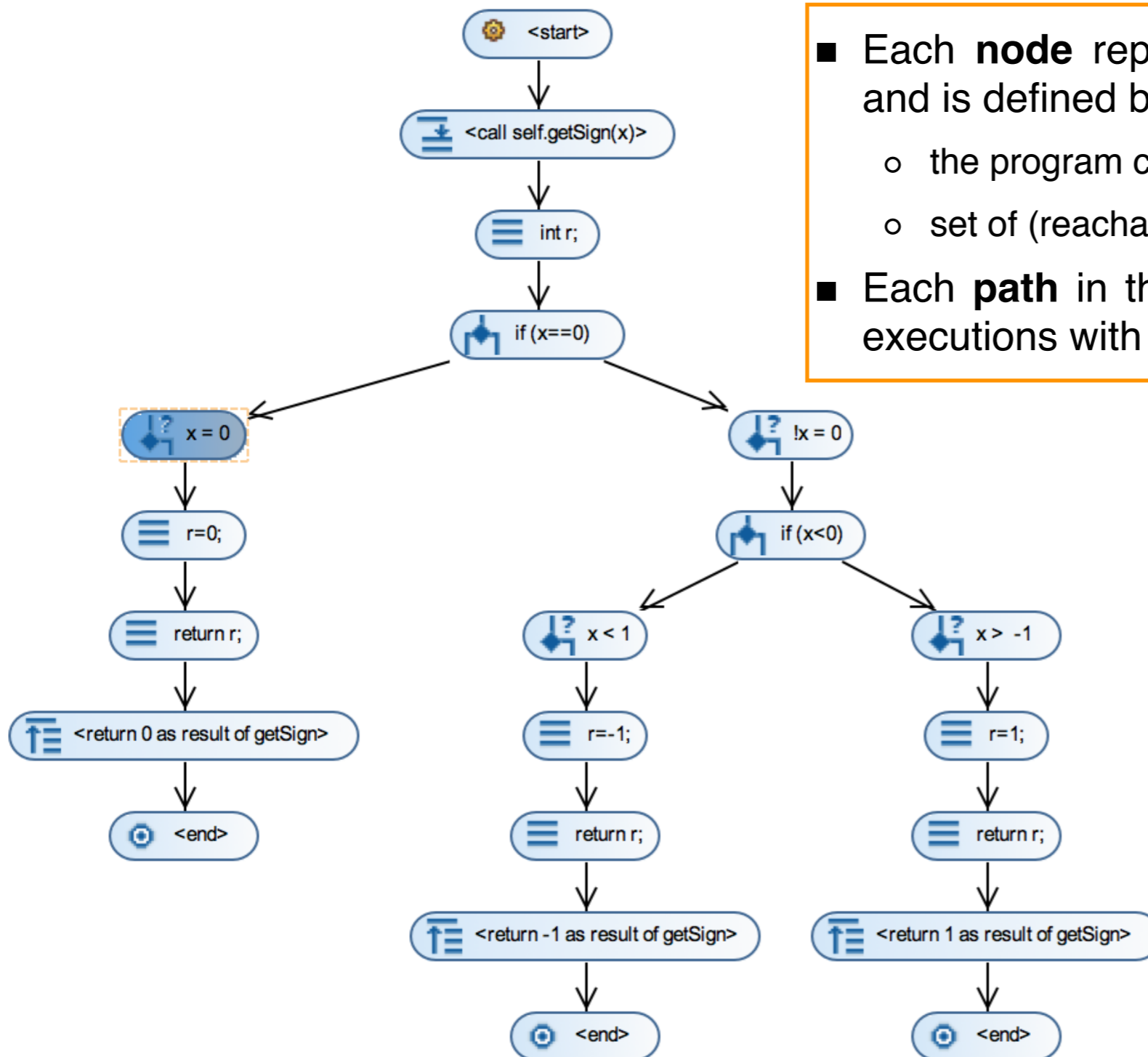
Symbolic execution



■ Approach:

- Execute a program, treating some or all of the inputs as **symbolic** — actual values need not be specified.
- When a branch condition that depends on symbolics input, follow each branch leading to a **symbolic execution tree**.

Symbolic execution tree



- Each **node** represents a symbolic execution state and is defined by:
 - the program counter (PC)
 - set of (reachability) conditions over the symbolic inputs
- Each **path** in the tree represents a set of possible executions with the same reachability conditions.

```
int getSign(int x) {  
    int r ;  
    if (x == 0)  
        r = 0;  
    else if (x < 0)  
        r = -1;  
    else  
        r = 1;  
    return r;  
}
```

[screenshot obtained using the [KeY Symbolic Execution Debugger](#)]

Major aspects

■ Automated verification

- no need to specify concrete inputs as in testing — in fact we can use symbolic execution to generate test cases instead!
- coverage can be guided by precise (boundary) conditions, unlike fuzz-testing

■ Path explosion ...

- The number of execution paths grows exponentially with respect to the number of inputs, control branches, ...
- Mitigations: coverage criteria constraints, special techniques (e.g. partial order reduction to identify/merge equivalent states, exploration heuristics)

■ Environment interface (e.g., OS system calls, multi-threading):

- Native execution (simple), but leads to non-deterministic side effects being propagated down the execution tree (global execution may be not accurate / repeatable)
- Approach: implement a set of primitives of interest through a symbolic execution layer

A few symbolic execution tools

■ C / LLVM

- [Klee](#) (discussed next)
- [Cloud9](#)
- [S2E](#)

■ x86

- [angr](#)
- [Triton](#)

■ Java

- [Java Path Finder \(JPF\) - symbc module](#)
- [KeY symbolic execution debugger](#)

Klee

■ Features:

- Symbolic execution of LLVM bytecode
- Bindings for POSIX runtime, libc, and uclibc (library calls can be executed symbolically)
- Program arguments and standard input can be symbolic, there is also support for symbolic files

■ Fast start:

- Online: <http://klee.doc.ic.ac.uk/>
- Docker image: <http://klee.github.io/docker/> (used in class)

pany.c

```
char pass[MAX_PASSWORD_LEN+1];  
klee_make_symbolic(pass, sizeof(pass), "pass");  
isPasswordOK(pass);
```

passwd.c

```
int isPasswordOK(const char* pass) {  
    ...  
    int len = strlen(password);  
    ...  
}
```

compilation

```
$ clang -emit-llvm -g -I $KLEE_SRC_ROOT/include -c testAnyString.c passwd.c  
$ llvm-link testAnyString.bc passwd.bc -o testAnyString
```

Klee execution

```
$ klee --max-time=60 --libc=uclibc ./testAnyString
```

...

```
KLEE: ERROR: /home/klee/klee_build/klee-uclibc/libc/string/strlen.c:22: memory  
error: out of bound pointer
```

```
KLEE: done: generated tests = 9
```

Klee test case inspection (for the error)

```
$ ktest-tool --write-ints --trim-zeros klee-last/test000002.ktest
```

```
ktest file : 'klee-last/test000002.ktest'
```

```
args      : ['testAnyString']
```

```
num objects: 1
```

```
object 0: name: b'pass'
```

```
object 0: size: 13
```

```
object 0: data: b'\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01'
```

passwd.c

```
char pass[MAX_PASSWORD_LEN+1];
klee_make_symbolic(pass, sizeof(pass), "pass");
isPasswordOK(pass);
```

pass is marked as symbolic

passwd.c

strlen vulnerable to buffer overflow

```
...
int len = strlen(password);
...
}
```

char* pass) {

compilation

```
$ clang -emit-llvm -g -I $KLEE_SRC_ROOT/include -c testAnyString.c passwd.c
$ llvm-link testAnyString.bc passwd.o testAnyString.o
```

Klee execution

```
$ klee --max-time=60 --libc=uclibc ./testAnyString
```

klee execution detects buffer overflow / program crash

```
...
KLEE: ERROR: /home/klee/klee_build/klee-uclibc/libc/string/strlen.c:22: memory error: out of bound pointer
KLEE: done: generated tests = 9
```

Klee test case inspection (for the error)

```
$ ktest-tool --trim-zeros klee-last/test000002.ktest
ktest file : 'klee-last/test000002.ktest'
args      : ['testAnyString']
num objects: 1
object 0: name: b'pass'
object 0: size: 13
object 0: data: b'\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01'
```

this input value for pass is not null-terminated

pnultermstring.c

```
char pass[MAX_PASSWORD_LEN+1];  
klee_make_symbolic(pass, sizeof(pass), "pass");  
int len = klee_range(0, sizeof(pass)-1, "len");  
klee_assume(pass[len] == 0);  
isPasswordOK(pass);
```

pass is marked
as symbolic

len is also symbolic — an integer
with range between 0 and
sizeof(pass)-1

Klee executes under the
assumption that
pass[len] == 0
i.e. that pass is null-terminated

- In this case, klee finds no buffer overflow errors
- In fact, no errors at all, we can add more assumptions of interest but also assertions

```

#define LET_POS(var, cond) \
int var = klee_range(0, len-1, #var); klee_assume(cond);

char pass[MAX_PASSWORD_LEN+1];
klee_make_symbolic(pass, sizeof(pass), "pass");
int len = klee_range(MIN_PASSWORD_LEN, MAX_PASSWORD_LEN, "len");
LET_POS(lpos, IS_LOWER(pass[lpos]));
LET_POS(upos, IS_UPPER(pass[upos]) & (upos != lpos));
LET_POS(dpos, IS_DIGIT(pass[dpos]) & (dpos != lpos) & (dpos != upos));
klee_assume(pass[len] == 0);
for (int i=0; i < len; i++) {
    if (i != dpos && i != lpos && i != upos) {
        char c = pass[i];
        klee_assume( IS_DIGIT(c)
                    | IS_UPPER(pass[i])
                    | IS_LOWER(pass[i]));
    }
}

int ok = isPasswordOK(pass);
klee_assert(ok);

```

pvalidLUD.c

- Assumptions:

- Password has valid length, and contains at least one lower case letter, one upper case letter, and one digit

- Assertions

- isPasswordOK should return 1.

A few research highlights

■ Overview papers

- [Symbolic execution for software testing in practice: preliminary assessment](#), Cadar et al., ICSE'11
- [All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution \(but Might Have Been Afraid to Ask\)](#) , Schwartz et al., SP'10

■ A few example applications / tools

- [KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs](#), OSDI'08
- [SAGE: Whitebox Fuzzing for Security Testing](#), Godefroid et al, CACM'12
- [Automatic Exploit Generation](#), Avgerinos et al., CACM, 2014 -
- [Checking Interaction-Based Declassification Policies for Android Using Symbolic Execution](#) , Micinski et al., ESORICS'15
- [SAFELI – SQL Injection Scanner Using Symbolic Execution](#), Fu & Xian, TAV-WEB '08