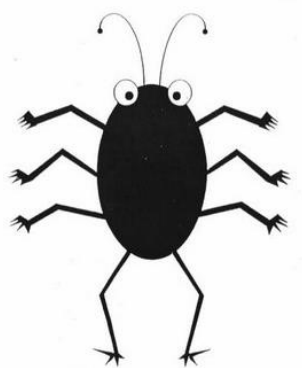


# Software Testing

## - Introduction -

- Bugs? What is testing? The RIP model.
- How important is testing ?
- Test case, test requirement, coverage criteria, coverage level, criteria subsumption.
- An introduction to Junit, EclEmma, and SpotBugs.
- Testing and the software development process. Testing activities. Testing levels.



*“**Bug**—as such little faults and difficulties are called—show themselves, and months of anxious watching, study, and labor are requisite before commercial success—or failure—is certainly reached.” [Thomas Edison, 1878]*



9/9

0800 Antan started  
 1000 " stopped - antan ✓  
 1300 (032) MP - MC ~~1.982147000~~  
 (033) PRO 2 2.130476415  
 conch 2.130676415

Relays 6-2 in 033 failed special speed test  
 in relay " 10.000 test.

Relay  
 2145  
 Relay 3370

1100 Started Cosine Tape (Sine check)  
 1525 Started Mult + Adder Test.

1545



Relay #70 Panel F  
 (moth) in relay.

First actual case of bug being found.

1630 Antan started.  
 1700 closed down.



“First actual case of bug being found” [!!!]

Note in Harvard Mark II logbook by Grace Hopper,  
 1947 [actual moth (bug) part of the logbook], U.S.  
 Naval Historical Center Online Library Photograph



## Other famous “bugs”



Pentium  
FDIV  
bug  
(1994)

$$\frac{4195835.0}{3145727.0} = 1.33374 \neq 1.33382$$



### Ariane 5 bug (1996)

“It took the European Space Agency **10 years and \$7 billion to produce Ariane 5 [...]**.

**All it took to explode** that rocket less than a minute into its maiden voyage last June, scattering fiery rubble across the mangrove swamps of French Guiana, **was a small computer program trying to stuff a 64-bit number into a 16-bit space.**”

From: “A bug and a Crash”, James Gleick,

<http://www.around.com/ariane.html>



Apple Maps, 2012



### Página temporariamente indisponível

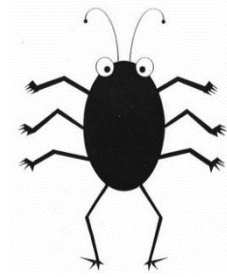
Vimos por este meio informar que por motivos de realização de intervenções técnicas imprescindíveis, a plataforma CITIUS, estará indisponível durante o período da manhã de hoje.

Agradecemos a vossa compreensão e pedimos desculpa por possíveis incómodos causados.

Citius, Ministério da Justiça, 2014

Other examples: “[History’s Worst Software Bugs](#)”, Wired [\[link\]](#)

# A simple bug



```
public static int numZeros (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's location is reached, how does execution corrupt the program state? Does it always corrupt the program state?
  - ✱ If program state is corrupted, does numZero fail? How?
- The term “bug” is ambiguous ... are we referring to the source code or to outcome of a failed execution? We need clear terminology.

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

```
public static int numZeros (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 [ $\sim$  the location of the bug]
  - ✱  $i = 1$  in the code [should be  $i = 0$ ]
- **Error:** erroneous program state caused by execution of the defect
  - ✱ if  $x[0] == 0$ , **count** becomes 0 (and should be 1)
- **Failure:** propagation of an erroneous state to the program output
  - ✱ The output value for  $x = \{ 0, 1, 0 \}$  is 1, while the expected value is 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 (line 1)]>`
  - \* 2: `<x={0,1,2,0}, count=0, PC=[i=1 (2)]>`
  - \* 3: `<x={0,1,2,0}, count=0, i=1, PC=[i<x.length (2)]>`
  - \* 4: `<x={0,1,2,0}, count=0, i=1, PC=[if(x[i]==0) (3)]>`
  - \* ...
  - \* `<x={0,1,2,0}, count=1, PC=[return count; (5)]>`

# Error state - convention

- Error state is the first state in an execution different from that in an execution of a correct program.
- If we had  $i=0$ , the execution of `numZero({0,1,2,0})` would start with states:
  - ✱ 1: `<x={0,1,2,0}, PC=[int count=0 (1)] >`
  - ✱ 2: `<x={0,1,2,0}, count=0, PC=[i=0 (2)]>`
  - ✱ 3: `<x={0,1,2,0}, count=0, i=0, PC=[i<x.length (2)]>`
- Instead we have:
  - ✱ 1: `<x={0,1,2,0}, PC=[int count=0 (1)]>`
  - ✱ 2: `<x={0,1,2,0}, count=0, PC=[i=1 (2)]>`
  - ✱ 3: `<x={0,1,2,0}, count=0, i=1, PC=[i<x.length (2)]> ...`



# Basic terminology

- Recall:
  - ✱ Fault: A static defect in the software's source code (defined by one or more source code locations)
  - ✱ Error: An incorrect internal state that is the manifestation of some fault
  - ✱ Failure: External, incorrect behaviour wrt the requirements or other description of the expected behaviour
- **Testing:** evaluating software by observing its execution.
- **Test failure:** a test execution that results in a failure.
- **Debugging:** the process of finding a fault given a failure.

# Test failure - RIP model

- Reachability
  - ✱ The fault is reached.
- Infection
  - ✱ Execution of the fault leads to an error.
- Propagation
  - ✱ Errors are propagated to the program output (failure)
- If the RIP conditions are met, test failures can occur.

# Reach, infect, no propagate: error but no failure

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

- Consider an execution where  $x=\{1,0,2,0\}$ 
  - \* Error: State  $\langle x=\{1,0,2,0\}, i=1, \text{count}=0, \text{PC}=\text{if } \dots \rangle$  deviates from expected state  $\langle x=\{1,0,2,0\}, i=0, \text{count}=1, \text{PC}=\text{if } \dots \rangle$
  - \* No Failure:  $\text{numZero}(\{1,0,2,0\})$  returns 2 as expected.

# Reach, infect, and propagate: error with failure

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

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



## Reach, no infect: fault but no failure

```
//@ ensures: if x == null throws NullPointerException
//@ else returns the number of non-negative numbers in x
public static int numNonNegatives (int[] x) {
1   int count = 0;
2   for (int i = 0; i < x.length; i++)
3       if (x[i] > 0)
4           count++;
5   return count;
}
```

- Consider an execution where  $x=\{1\}$ 
  - ✱ No error: State  $\langle x=\{1\}, i=0, \text{count}=1, \text{PC}=\text{for}...\rangle$  at the second entry to the loop is still valid
  - ✱ Hence no Failure: `numNonNegatives({1})` returns 1 as expected.

# Some faults might be very hard to find

## Fault sensitivity

- The ability of code to hide faults from a test suite
- Executing buggy code does not always expose the bug

## Coincidental correctness

- Faulty code can appear to work correctly by coincidence.

```
public static int scale(int x) {  
1   int j = j - 1;    // should have been j + 1  
2   j = j / 3000;  
3   return j;  
}
```

- Only 40 out of 65,536 values produce an incorrect result. None of them on a boundary for j.
- Testing with 99.9% of all possible numbers might not find the problem.
- Domain knowledge is critical in these cases

# How “mature” is your testing?

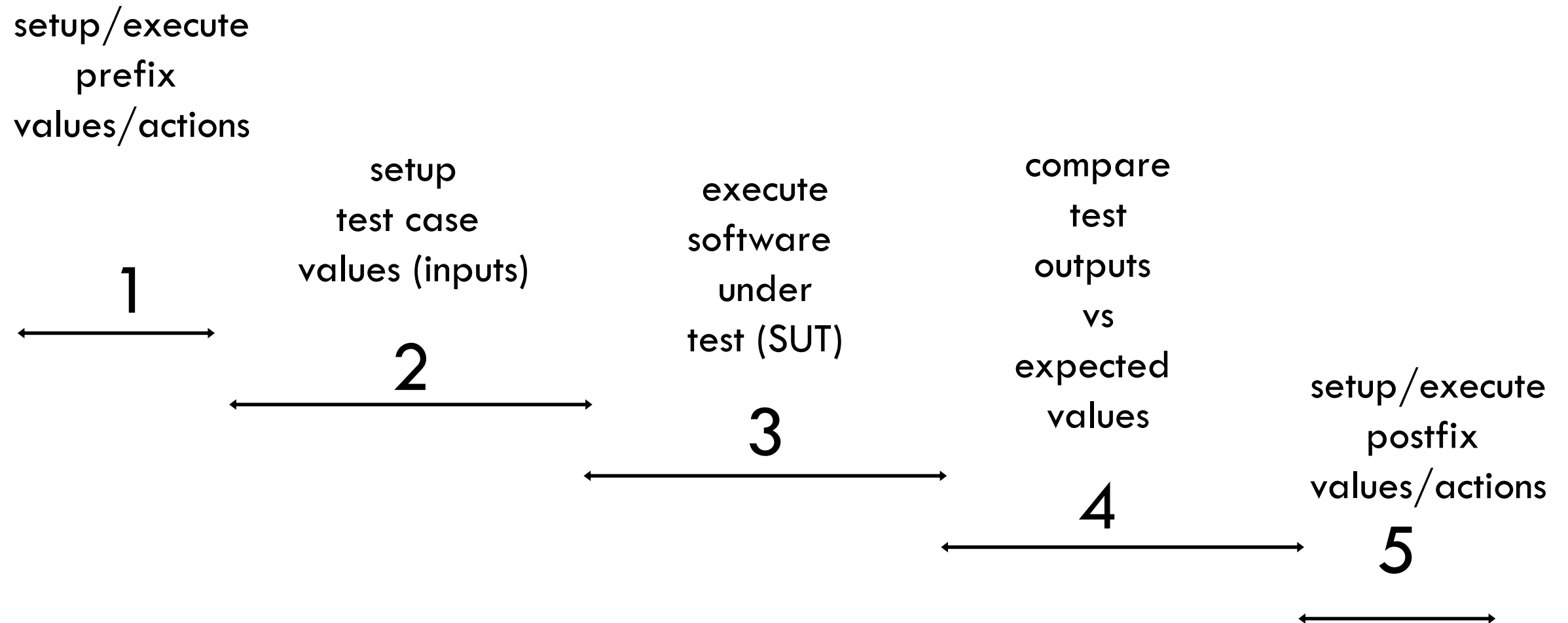
- Beizer’s scale for test process maturity [A & O, page 8]
  - ✱ Level 0: “There’s no difference between testing and debugging.”
  - ✱ Level 1: “The purpose of testing is to show that the software works.”
  - ✱ Level 2: “The purpose of testing is to show that the software doesn’t work.”
  - ✱ Level 3: “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.”

# More terminology - test case

- **Test case:** A test case is composed by the test case values, expected results, prefix values, and postfix values necessary for a complete execution and evaluation of the software under test (SUT)
  - **Test case values:** the input values necessary to complete some execution of the software under test
  - **Expected results:** the expected result that will be produced by the test case if and only if the program satisfies its intended behavior
  - **Prefix values/actions:** any inputs/commands necessary to put the software into the appropriate state to receive the test case values
  - **Postfix values/actions:** any inputs/commands that need to be sent to the software after the test case values are sent



# Test case execution



- We'll focus today only on testing that requires steps 2-4.

# Example test cases for numZero

```
public static int numZeros (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

# More terminology

- **Test set:** a set of test cases. We denote  $T$  for a test set.
- **Test requirement:** a requirement that should be satisfied by the test set. Test requirements normally come in sets. We denote  $TR$  for a set of test requirements.
- **Coverage criterion:** A coverage criterion  $C$  is a rule or collection of rules that define a set of test requirements  $TR(C)$ .
- **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%.

# Basic coverage criteria

```
public static int numZeros (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).
  - ✱  $TR(BC) = \{ \text{NPE-B1, B1, !B1, B2, !B2} \}$



# LC, IC, BC for numZero

```

public static int numZeros (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	1	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	1	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)	43% (3/7)	20% (1/5)
{t1, t2}	80% (4/5)	86% (6/7)	40% (2/5)
{t2, t3}	100% (5/5)	100% (7/7)	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  
yet bug 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	1	no	All	All	B1, !B1, B2, !B2

# Criteria subsumption

- Criteria Subsumption: A coverage criterion C1 subsumes criterion 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 eg:
  - ✱ 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.



# Side effects of measures

- Earlier in testing, the pressure is to increase bug counts. In response, testers will:
  - Run tests of features known to be broken or incomplete.
  - Run multiple related tests to find multiple related bugs.
  - Look for easy bugs in high quantities rather than hard bugs.
  - Less emphasis on infrastructure, automation architecture, tools and more emphasis on bug finding. (short-term payoff but long-term inefficiency.)

# Side effects of measures

- Later in testing, the pressure is to decrease the new bug rate:
  - Run lots of already-run regression tests.
  - Don't look as hard for new bugs.
  - Classify unrelated bugs as duplicates.
  - Classify related bugs as duplicates (and closed), hiding key data about the symptoms / causes of the problem.
  - Report bugs informally, keeping them out of the tracking system.

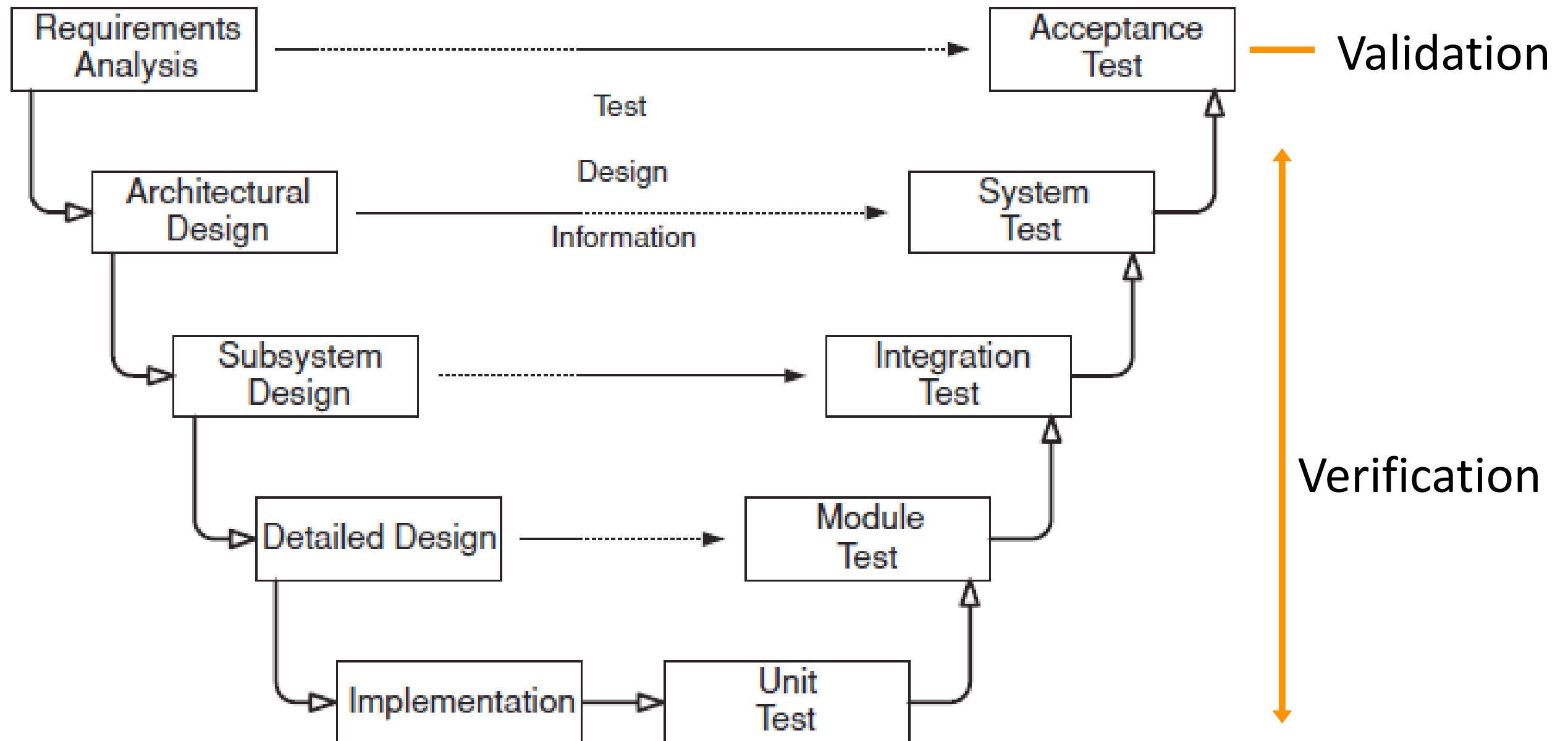
# Test engineering - terminology

- Verification [A & O, p.11]
  - ✱ “The process of determining whether the products of a given phase of the software development process fulfils the requirements established during the previous phase.”
  - ✱ Requires technical background on the software, normally done by developers at the various stages of development
- Validation [A & O, p.11]
  - ✱ “The process of evaluating software at the end of the software development to ensure compliance with intended usage.”
  - ✱ Not done by developers, but by experts in the intended usage of the software

# Test engineering - terminology

- Verification and validation are both about assessing the quality of a system. However, they do have a subtle difference, which can be quickly described by a single question:
  - Verification: “Have we built the software right?”
    - Is the SUT in accordance with the specifications?
  - Validation: “Have we built the right software?”
    - Does the SUT satisfy the customer? Is it useful?
- Both verification and validation are fundamental for ensuring the delivery of high-quality software systems.

# Testing and the SW development lifecycle



[A & O, p 6]

# What are the benefits?

- **Cost-Effective:** testing any IT project helps you to save money for the long term. If bugs are caught at an earlier stage, it will cost less to fix.
- **Security:** the most vulnerable and sensitive benefit of software testing. People are looking for trusted products. It helps in removing risks and problems earlier.
- **Product quality:** an essential requirement of any software product. Testing ensures a quality product is delivered to customers.
- **Customer Satisfaction:** the main aim of any product is to give satisfaction to their customers. UI/UX Testing ensures the best user experience.
- However, developing tests is a time-consuming task subject to incompleteness and further errors. It's not possible to perform exhaustive testing.

# Unit testing

- In some situations, the goal is to test a single feature of the software, purposefully ignoring the other units of the systems.
- When we test software components in isolation, we are doing what is called **unit testing**.
- Defining a 'unit' is dependent on the context. A unit can be just one method or can consist of multiple classes.
- Here is a definition for unit testing by Roy Osherove
  - “A unit test is an automated piece of code that invokes a unit of work in the system. And a unit of work can span a single method, a whole class or multiple classes working together to achieve one single logical purpose that can be verified.”

Roy Osherove, “The Art of Unit Testing”, 2nd edition, 2013



# Unit testing

👍 Unit tests are fast. Fast tests make it possible to test huge portions of the system in a small amount of time allowing constant feedback.

👍 Unit tests are easy to control. A unit test tests the software by giving certain parameters to a method and then comparing the return value to the expected result.

👍 Unit tests are easy to write and do not require a complicated setup. A single unit is often cohesive and small, making the job of the tester easier.

🗨️ The large number of classes in a system and their interaction cause the system to behave differently in its real application than in unit tests. Therefore, unit tests do not perfectly represent the real execution of a software system.

🗨️ Some types of bugs cannot be caught at unit test level. They only happen in the integration of the different components (which are not exercised in unit tests).

# Integration testing

- Integration testing is the test level we use when we need something more integrated (or less isolated) than a unit test but without the need of exercising the entire system.
- The goal of integration testing is to test multiple components of a system together, focusing on the interactions between them instead of testing the system as a whole.
- Are they communicating correctly? What happens if component A sends message X to component B? Do they still present correct behavior?

# Integration testing

👍 The advantage of integration tests is that, while not fully isolated, devising tests just for a specific integration is easier than devising tests for all the components together. Therefore, the effort of writing such tests is a little more than the effort required for unit tests but less than the effort for testing the entire system.

🗨️ Note that the more integrated our tests are, the more difficult they are to write.

🗨️ Usually more complex setups and teardown stages are needed.

# System testing

- To get a more realistic view of the software, and thus perform more realistic tests, we should run the entire software system, with all its databases, front-end apps, and any other components it has.
- When we test the system in its entirety, we are doing what is called system testing. In practice, instead of testing small parts of the system in isolation, system tests exercise the system as a whole.

# System testing

👍 The obvious advantage of system testing is how realistic the tests are. After all, the more realistic the tests are, the greater the chance that the system works when released.

👍 System tests also capture the user's perspective better than other tests. System tests are a better simulation of how the final user interacts with the system

🗨️ System tests are often slow when compared to unit tests. Tests must start and run the whole system with all its components. Contrary to unit tests, there are lots of dependencies that must be satisfied.

🗨️ System tests are harder to write. Some components (e.g., databases) might require complex setup before being used in a testing scenario. This implies additional code for automating the tests.

🗨️ System tests tend to become flaky. A flaky test is one that presents an erratic behavior: if you run it, it might pass or it might fail for the same configuration or unexpected state. Flaky tests are an important problem for software development teams.

# Introduction to JUnit

- JUnit is a testing framework for Java
  - the most widely used framework
- Eclipse includes JUnit and a GUI interface for it
- We'll use JUnit 5 with Eclipse
- Today:
  - ✱ JUnit test classes
  - ✱ Anatomy of basic JUnit test methods
  - ✱ JUnit assertions and other miscellaneous features
  - ✱ Associated patterns for test programming

# JUnit – test classes

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class TestArrayOperationsNumZero {

    @Test
    public void testNumZerosWithEmptyArray() {
        int[] x = {}; // zero-sized array
        int n = numZero(x);
        assertEquals(0, n, "count of zeros");
    }

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

    ...
}
```

imports

test class

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

another test  
method/case



# JUnit/xUnit - conventions

- “TestClass class pattern” [G.Meszaros page 373]
  - ✱ Group related test methods in a single test class.
- The name of test packages/classes/methods should at least transmit:
  - ✱ The name of the SUT class (e.g., `TestArrayOperationsNumZero`)
  - ✱ The name of the method or feature being tested (e.g., `testArrayOperationsNumZero`)
  - ✱ The relevant characteristics of the test case values per each test method (e.g., `testNumZeroEmptyArray`)
  - ✱ See G. Meszaros, “xUnit Testing patterns”, page 158
- It is also common to prefix or suffix test classes with “Test” and prefix/suffix test methods with “test”.

# JUnit - test methods

```
@Test
public void testNumZerosWithArrayWithoutZeros() {

    int[] x = {1, 2, 3};

    int n = ArrayOperations.numZero(x);

    assertEquals(0, n, "0 zeros in an array with no zeros");
}
...
```

1) setup test case values

2) execute SUT

3) assert expected vs. test outputs

expected

output

## o Test design patterns

- ✿ 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 wrong

# JUnit assertions

Method	Condition checked
<code>assertEquals(expected, actual, msg)</code> <code>assertNotEquals(expected, actual, msg)</code>	<code>actual.equals(expected)</code> <code>!actual.equals(expected)</code>
<code>assertTrue(actual, msg)</code> <code>assertFalse(actual, msg)</code>	<code>actual == true</code> <code>actual == false</code>
<code>assertNull(actual, msg)</code> <code>assertNotNull(actual, msg)</code>	<code>actual == null</code> <code>actual != null</code>
<code>assertArrayEquals(vExp, vAct, msg)</code>	<b>Arrays <code>vExp</code> and <code>vAct</code> have the same contents.</b>
<code>assertSame(expected, actual, msg)</code>	<code>actual == expected</code> <b>(exactly the same object reference)</b>

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

## exceptions as expected results

```
@Test
public void testNumZerosWithNullArgument() {
    int[] x = null;
    assertThrows(NullPointerException.class, () -> {
        ArrayOperations.numZeros(x);
    });
}
```

equivalent to

```
@Test
public void testNumZerosWithNullArgument() {
    int[] x = null;
    try {
        ArrayOperations.numZero(x);
        fail("expected NullPointerException");
    } catch (NullPointerException e) { }
}
```

Note: the 2nd pattern is more verbose and unnecessary in this case. It is useful in situations when we wish to perform other assertions beyond the expected exception behaviour.

# Other JUnit features: prefix/postfix actions

```
@BeforeAll
public static void globalSetup() {
    // prefix actions executed once before any test
    ...
}
@AfterAll
public static void globalTeardown() {
    // prefix actions executed once after all tests
    ...
}
@BeforeEach
public void perTestSetup() {
    // prefix actions executed before each test
    ...
}
@AfterEach
public static void perTestTeardown() {
    // actions executed after each test
    ...
}
```

# Other JUnit features: parameterized tests

```
public class Calculator {  
  
    public int evaluate(String expression) {  
        int sum = 0;  
        for (String summand: expression.split("[+()]"))  
            sum += Integer.valueOf(summand);  
        return sum;  
    }  
  
}
```

- Simply parse the given string by tokens, to sum them all

# Other JUnit features: parameterized tests

```
@ParameterizedTest(name = "check if {0} equals {1}")
@CsvSource({
    "200,      200",
    "1+2,      3",
    "0+0,      0",
    "00+000,   0",
    "0+-1+0,   -1",
    "100+-100, 0",
    "1+1+1+1+2+3+4+5+1+2+3+4+5, 33"
})
public void expressionTest(String expression, int expected) {
    int actual = calculator.evaluate(expression);
    assertEquals(expected, actual);
}
```




## Other JUnit features: parameterized tests

```
@ParameterizedTest(name = "check if {0} equals {1}")
@CsvSource({
    "200,      200",
    "1+2,      3",
    "0+0,      0",
    "00+000,   0",
    "0+-1+0,   -1",
    "100+-100, 0",
    "1+1+1+1+2+3+4+5+1+2"
})
```

Runs: 20/20 ✖ Errors: 0

---



<=

Testing Simple Calcula

simpleTest() (0,067 s)

```
public void expressionTest(String expression) {
    int actual = calculator.evaluate(expression);
    assertEquals("Expected " + expected + " but was " + actual, expected, actual);
}
```

Runs: 20/20    Errors: 0    Failures: 0

```
<= Testing Simple Calculator => [Runner: JUnit 5]
  simpleTest() (0,067 s)
  expressionTest(String, int) (0,024 s)
    check if 200 equals 200 (0,024 s)
    check if 1+2 equals 3 (0,003 s)
    check if 0+0 equals 0 (0,002 s)
    check if 00+000 equals 0 (0,028 s)
    check if 0+ -1+0 equals -1 (0,000 s)
    check if 100+ -100 equals 0 (0,000 s)
    check if 1+1+1+1+2+3+4+5+1+2+3+4+5 equals 35 (0,000 s)
  timingABigTest() (0,029 s)
```

# Other JUnit features: repeated tests

```
private Random r = new Random();

@RepeatedTest(value=10)
public void randomTest() {
    int a = r.nextInt(1000)-500;
    int b = r.nextInt(1000);

    int expected = a+b;
    int actual    = calculator.evaluate(a+"+"+b);
    assertEquals(expected, actual, a+"+"+b);
}
```

- Useful for production of random tests
- Another possibility is to produce and test random expression sentences

# Other JUnit features: tests with time-out

```
@Test
public void timingABigTest() {
    int a = 123456;
    int b = 900000;

    int expected = a+b;
    int actual    = assertTimeoutPreemptively(
        Duration.of(1500, ChronoUnit.MICROS),
        () -> calculator.evaluate(a+"+"+b)
    );
    assertEquals(expected, actual, a+"+"+b);
}
```

# Other JUnit features: assumptions

```
@Test
public void checkTitleWithOnlyOneBookTest() {

    // Testing pre-conditions
    assertTrue(shelf.books().isEmpty());

    shelf.add(new Book("Software Patterns"));
    assertTrue(shelf.contains("Software Patterns"));
}
```

- A failing assumption does not mean a test is failing, but simply that the test won't provide any relevant information, so it doesn't need to run.
- See more eggs in maven project `vvs_junit`

# Introduction to Static Testing

- Static Testing checks faults in a software application without executing its source code in order to discover faults.
  - Early detection of faults prior to test execution.
  - Early warning about suspicious aspects of the code or design.
  - Improved maintainability of code and design.
  - Prevention of faults, if lessons are learned in development.

# Introduction to Static Testing

- Two categories of static testing
  - Reviews – manual examinations of code
  - Automated tools – use of software applications to analyze source code

# Reviews

- A review in static testing is a meeting conducted to find causes of failures, i.e., faults in the program, rather than the failures themselves.
- By reviewing the program all team members get to know about the progress of the project and sometimes the diversity of thoughts may result in excellent suggestions.
  - Costs: 5% to 15% of development effort
  - Benefits: reduce faults by a factor of 10, testing cost reduced 50%-80%
  - note that testing and debugging usually costs 50%



# Automated Tools

- Check for violations of standards.
- Check for things which may be faulty.
- Can find unreachable code, undeclared variables, parameter type mismatches, uncalled functions, etc.
- The objective of static analysis is to find faults in software source code and software models. Static analysis is performed without executing the software being examined by the tool.
- As with reviews, static analysis finds faults rather than failures.

# Automated Tools

- Types of tools:
  - Lexical: Words, strings, and regexes.
  - Syntactic: Tree of program structure.
  - Control flow graph (eg: checks for unreachable executions paths)
  - Data flow graph (checks uses of program's variables)
- Here are some examples of static analysis tools:
  - SpotBugs <https://spotbugs.github.io> [we'll try this one this week]
  - Checkstyle <https://checkstyle.sourceforge.io>
  - PMD <https://pmd.github.io>
  - Google's Error Prone <https://errorprone.info>
  - SourceMeter <https://www.sourcemeter.com>
  - Checkmarx <https://www.checkmarx.com>

# Exercise 1

[Adapted A&O pg.16, ex.3; maven project vvs\_01]

- For each method `findLast()`, `countPositive()`, `lastZero()`, and `oddOrPos()`
  - ✱ (a) Identify the fault and what would be the correct code.
  - ✱ (b) If possible, identify a test case that does not execute the fault (reachability condition not met).
  - ✱ (c) Identify a test case that leads to failure (reachability, infection and propagation conditions all met). Identify the first error state.
  - ✱ (d) If possible, identify a test case that results in an error, but not a failure (propagation condition not met). Identify the first error state.
  - ✱ (e) If possible, identify a test case that executes the fault but does not lead to an error state (infection condition not met).

# Exercise 2

- Write a JUnit test class for each method in exercise 1:
  - ✱ Start by writing tests that satisfy Line Coverage
  - ✱ Enhance the tests (if necessary) to satisfy Instruction Coverage
  - ✱ Enhance the tests (if necessary) to satisfy Branch Coverage
- Assess the coverage of your tests using Eclemma
  - ✱ Update at Eclipse Marketplace
  - ✱ See <http://eclemma.org> for instructions

# Exercise 3

- Perform a static analysis with SpotBugs
  - ✱ Install/Update at Eclipse Marketplace
  - ✱ R-click ArrayOperations.java | SpotBugs | FindBugs
  - ✱ Read the report and understand what's the problem