Software testing fundaments

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

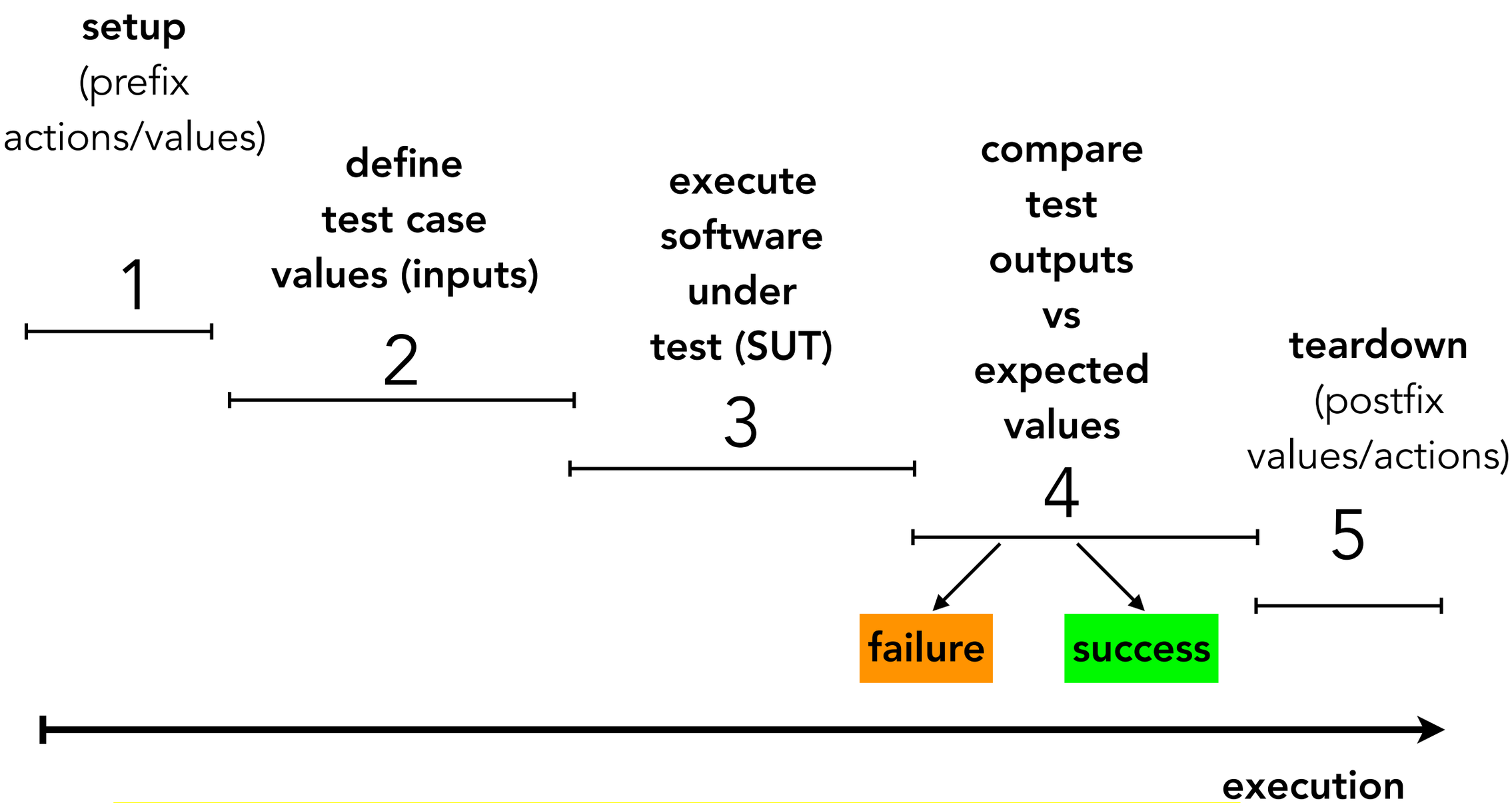
Outline

- Software testing
  - quick revision of our overview early in the semester
- Programming tests — xUnit style
- Base concepts & terminology in software testing
  - Fault, Error, Failure
  - The RIP conditions for test failure
  - Criteria for test requirements — simple examples
  - Test coverage
  - Criteria subsumption

Software testing

- We discussed the importance of software testing during the SDLC, but not the fundaments or how tests are actually programmed.
- **Recall … (from 2nd class)**
  - **Testing:** Observe if software meets the expected behavior when executed.
  - ***Does not guarantee absence of bugs, in fact it seeks to expose them.***
  - General goal anyway is to establish a degree of confidence in software.
- **Recall:** How is the execution of a single test case specified? We call it a test case with the following ingredients:
  - The **subject-under-test** (SUT), e.g. function, class, component …
  - The **inputs for the test** and the **expected outputs**.
  - The necessary **prefix/post-fix actions & values** to setup/teardown the SUT state.

Test case — execution



Check separate slides - "JUnit - An introduction"
to understand how tests can be programmed in xUnit-style

Test case

- **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 != 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.

Programming tests

xUnit-style

Writing xUnit-style tests

- JUnit: the mostly widely used Java unit testing library.
- Conventions in JUnit are employed in other xUnit frameworks for other languagens.
- We provide a basic overview of JUnit 4 features:
  - Anatomy of basic JUnit test methods
  - JUnit assertions and other misc. features
  - Associated patterns for test programming using xunitpatterns.com as reference, the site for “xUnit Testing Patterns”, a book by G. Meszaros.
- Complementary material:
  - A good online tutorial: Unit Testing with JUnit @ vogella.com

JUnit - test classes

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

public class ArrayOperationsNumZeroTest {

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

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

imports

test class

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

another test
method/case

JUnit/xUnit - conventions

- Testcase class pattern
  - Group related test methods in a single test class.
- Test naming conventions (xunitpattterns.com) — *“names of the test package, the Testcase Class and Test Method to convey at least the following information:*
  - *The name of the SUT class.*
  - *The name of the method or feature being exercise.*
  - *The important characteristics of any input values related to the exercising of the SUT.*
  - *Anything relevant about the state of the SUT or its dependencies.”*
- Examples of common conventions:
  - Test class names end with `Test` and provide an indication of the SUT `ArrayOperationsTest, MyClassTest`
  - Test method names should start with `test` and provide indication for the test-case intent like `testNumZeroWithEmptyArray, testNumZeroWithArrayWithoutZeros`

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
`assertEquals(msg,expected,value)` `assertNotEquals(msg,expected,value)`	`value.equals(expected)` `!value.equals(expected)`
`assertTrue(msg, value)` `assertFalse(msg, value)`	`value == true` `value == false`
`assertNull(msg, expression)` `assertNotNull(msg, expression)`	`value == null` `value != null`
`assertArrayEquals(msg, vexp, vVal)`	Arrays `vExp` and `vVal` have the same contents.
`assertSame(msg, expected, value)`	`value == expected` (exactly the same object reference)

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

Other JUnit features: exceptions as expected results

```
@Test (expected = NullPointerException.class)
public void testNumZeroWithNullArgument() {
   int[] x = null;
  numZero(x);
}
```

equivalent to

```
@Test
public void testNumZeroWithNullArgument1() {
   int[] x = null;
   try {
      numZero(x);
      fail("expected NullPointerException");
   } catch (NullPointerException e) { }
}
```

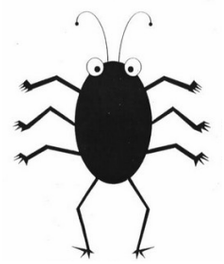
Note: the 2nd pattern is more verbose and unnecessary in this case. It is useful in situations when we wish to perform other assertions beyond the expected exception behavior.

Other JUnit features: prefix/postfix actions

```
@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
  ...
}
```

Base concepts & terminology in software testing

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 (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)] >`
  - `4: < x={0,1,2,0}, count=0,i=1,PC=[if(x[i]==0) (l3)] >`
  - `...`
  - `<x={0,1,2,0}, count=1, PC=[return count;(l5)] >`

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

- **R**eachability
  - The fault in the source code is reached during execution.
- **I**nfection
  - The program state enters in an error state, affected by the execution of the faulty code.
- **P**ropagation
  - The errors in program state are propagated to the ouputs.

`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**: **`<x={0,1,2,0}, i=1, count=0, PC= if ...,l3>`** deviates from expected internal state **`<x={1,0,2,0}, i=0,count=1,PC =[if ..., l3]>`**
  - **And failure: 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**: **`<x={1,0,2,0}, i=1, count=0, PC= if ...,13>`** deviates from expected internal state **`<x={1,0,2,0}, i=0,count=1,PC =[if ..., 13]>`**
  - **No Failure**! **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%.

Simple 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) = { 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.