DESIGN TESTS: WHITE/GLASS BOX TESTING

Goal: To design *selective tests* to ensure that we have executed/ exercised all:

1. independent paths in the code at least once.

   ☞ **Basis Path Testing**

2. logical decisions on their true and false sides.

   ☞ **Condition Testing**

3. loops at their boundaries and within their bounds.

   ☞ **Loop Testing**

4. internal data structures to ensure their validity.

   ☞ **Data Flow Testing**

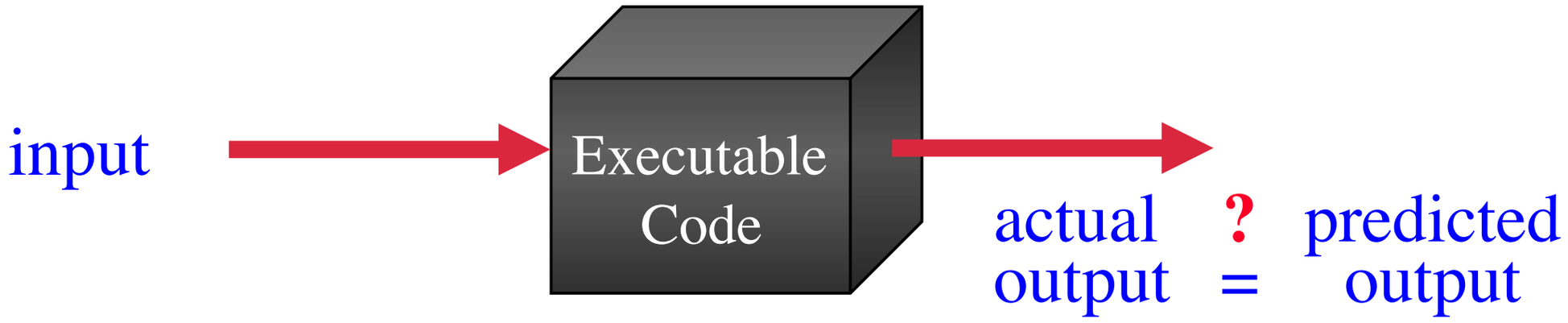
Basis path testing (4 steps)

1. From the code, draw a corresponding flow graph
2. Determine the Cyclomatic complexity
   2.1. Why?
3. Identify a basis set of linearly independent paths
   3.1. based on what complexity?
4. Prepare test inputs to execute each path in the basis set

Today

- Blackbox testing
- Testing in practice
  - JUnit
- Test coverage

DESIGN TESTS: BLACK BOX TESTING



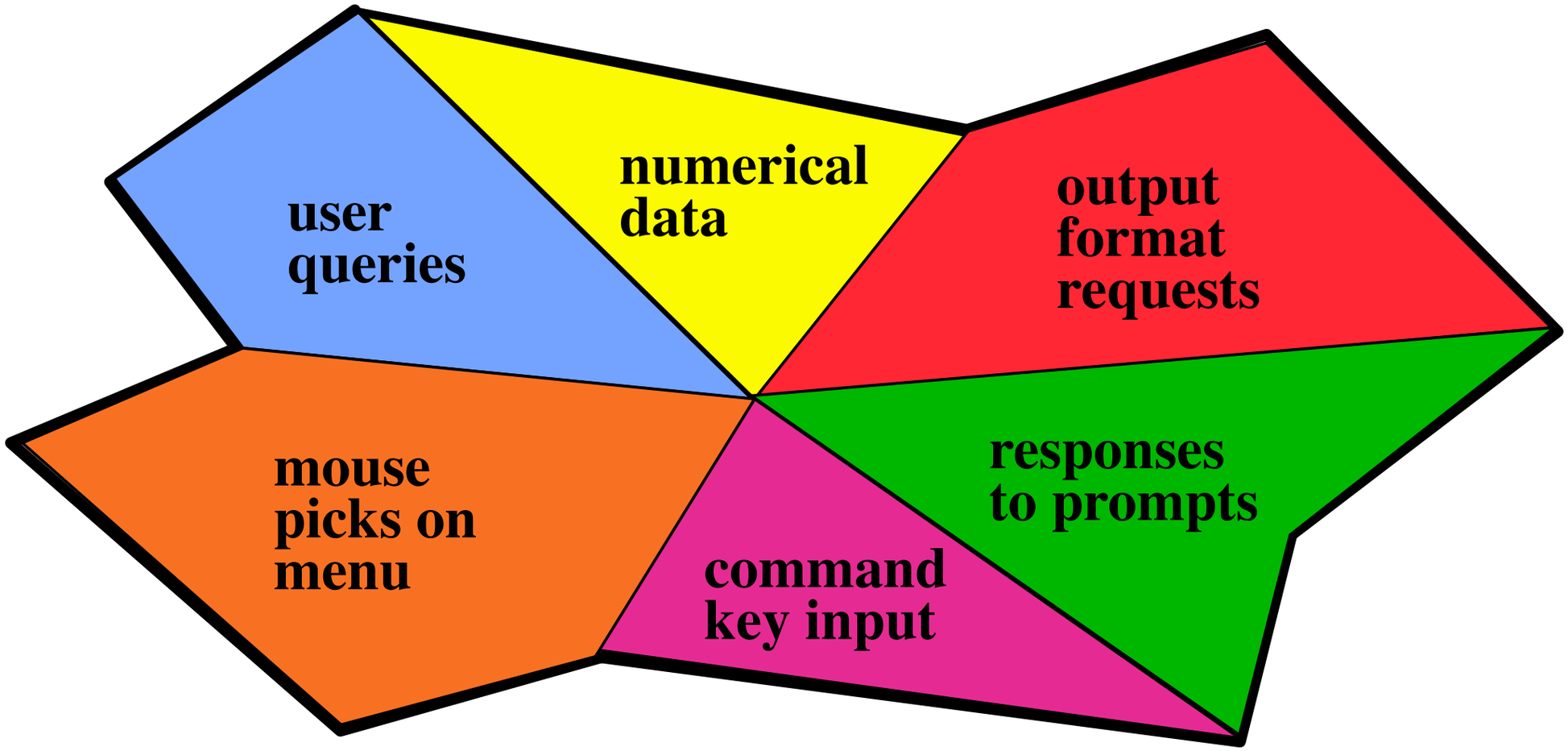
Black box tests attempt to find:

- incorrect or missing functions
- interface incompatibility errors
- data structure or external database access errors
- performance errors
- initialization and termination errors

To achieve reasonable testing, a black box test case should cover a range of input or output values, not just a single value. That is, it should tell us something about the presence or absence of a class of errors (e.g., all character data is correctly/incorrectly processed).

BLACK BOX TESTING: EQUIVALENCE PARTITIONING

Equivalence partitioning **groups inputs and outputs by type to create thorough test coverage of a class of errors.**



BLACK BOX TESTING: EQUIVALENCE PARTITIONING

IDEA: The input space is very large, the program is small.

- A program's behaviour is the "same" for "equivalent" sets of inputs.

- Ideal test suite:
  - Identify sets with the same behaviour.
  - Try one input from each set.

- Two problems
  1. The notion of the same behaviour is subtle.
     - Naïve approach: execution equivalence
     - Better approach: revealing subdomains
  2. Discovering the sets requires perfect knowledge.
     - Use heuristics to approximate the sets cheaply.

NAÏVE APPROACH: Execution EQUIVALENCE

```
int abs (int x) {
// returns:   x < 0        => returns -x
//            otherwise    => returns x

if (x < 0)    return -x;
else          return x;
}
```

- All **x** < 0 are execution equivalent.
  - The program *takes the same sequence of steps* for any **x** < 0.

- All **x** ≥ 0 are execution equivalent.
  - The program *takes the same sequence of steps* for any **x** ≥ 0.

☞ **This suggests, for example, that {-3, 3} is a good test suite.**

NAÏVE APPROACH: Execution EQUIVALENCE

- Consider the following buggy code.

```
int abs (int x) {
// returns:   x < 0        => returns -x
//            otherwise    => returns x

if (x < -2)   return -x;
else          return x;
}
```

- Two executions:

  `x` < -2 `x` ≥ -2

- Three behaviours:

  `x` < -2 (**OK**) `x` = -2 or -1 (**BAD**) `x` ≥ 0 (**OK**)

☞ **{-3, 3} does not reveal the error!**

DESIGN TESTS: BLACK BOX TESTING

1. Selecting Subdomains

2. Selecting Boundary Values

REVEALING SUBDOMAIN APPROACH

- We say that a program has the "same behaviour" for two inputs if it
  1. gives a correct result on both, or
  2. gives an incorrect result on both.

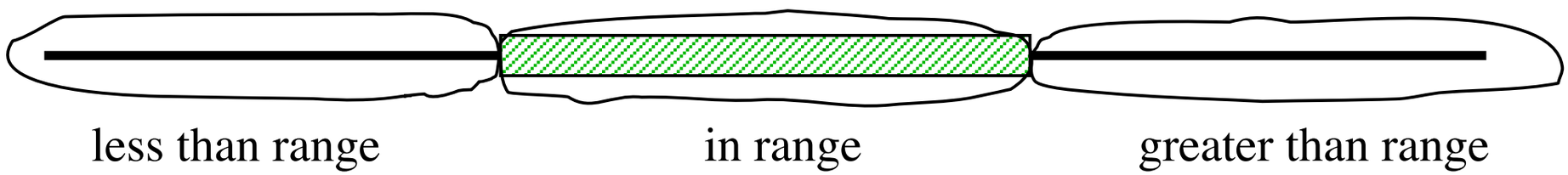
- A subdomain is a subset of possible inputs.

- A subdomain is revealing for an error, E,
  1. if each element in the subdomain has the same behaviour.
  2. if the program has error E, then it is revealed by the test.

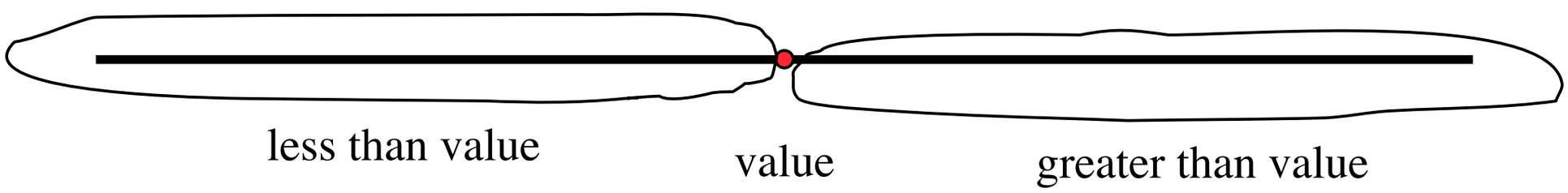
- The trick is to divide the inputs into sets of revealing subdomains for various errors.

HEURISTICS FOR SELECTING SUBDOMAINS

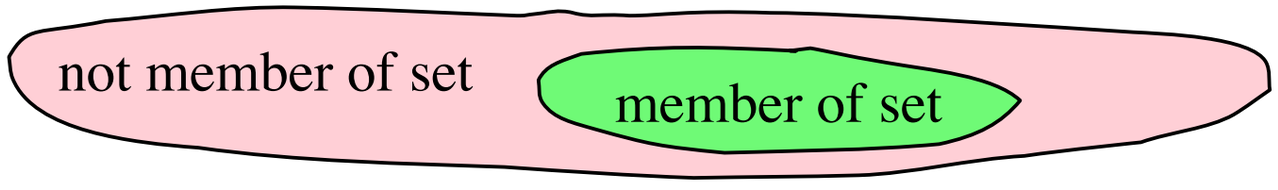
1. If the input is a range → one valid and two invalid subdomains:



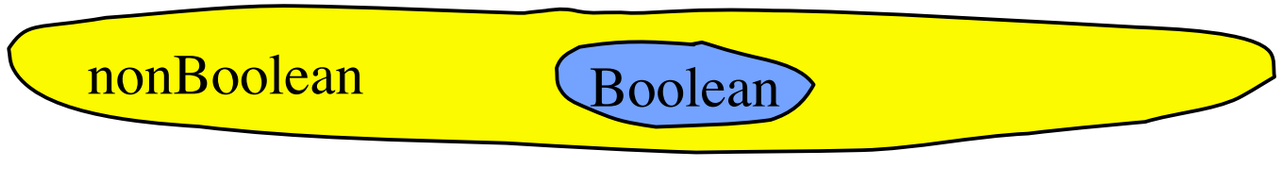
2. If the input is a specific value → one valid and two invalid subdomains:



3. If the input is a set of related values → one valid and one invalid subdomain:



4. If the input is Boolean → one valid and one invalid subdomain:



BOUNDARY TESTING

More errors occur at the "boundaries" of a subdomain than in the "center".

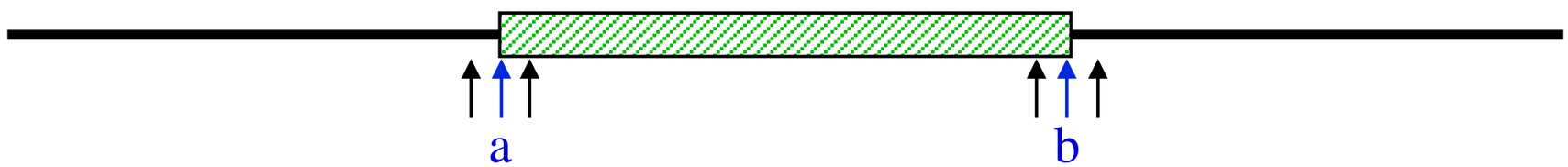
- **Why?**
  - off-by-one bugs
  - forget to handle empty container
  - overflow errors in arithmetic
  - program does not handle aliasing of objects

- Small subdomains at the "boundaries" of the main subdomains have a high probability of revealing these common errors.

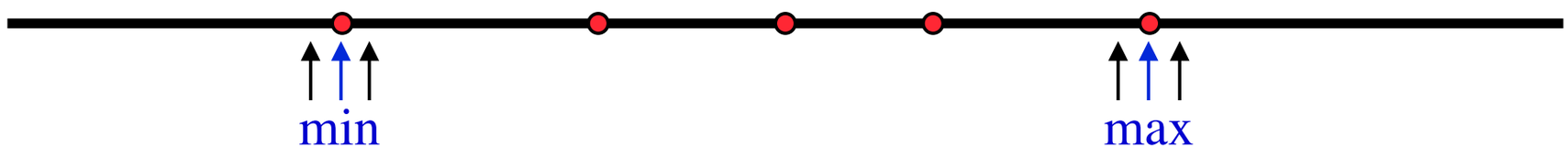
For testing we select:
1. "typical" values "inside" a subdomain.
2. values at the "boundaries" of a subdomain.

HEURISTICS FOR SELECTING BOUNDARY VALUES

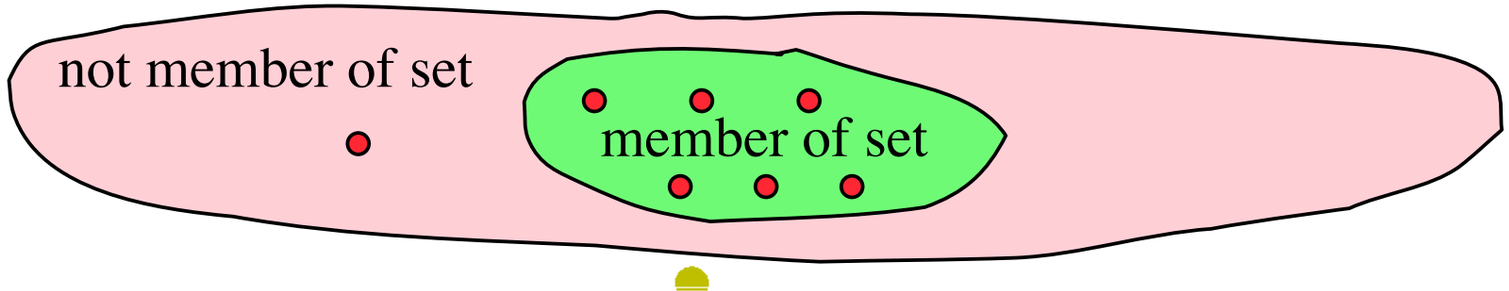
1. If the input is a range and is bounded by **a** and **b**, then use **a**, **b**, and values just above and just below **a** and **b**, respectively.



2. If the input is a number of discrete values, use the **minimum** and the **maximum** of the values and values just above and just below them, respectively. (Can also be applied to a single specific input value.)

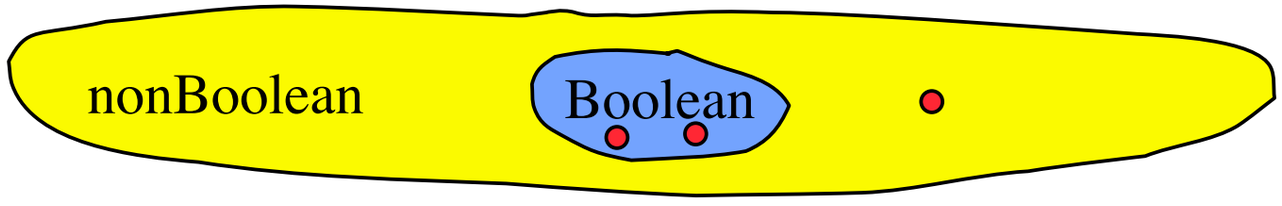


3. If the input is a set of values, test all values in the set (if possible) and one value outside the set.



HEURISTICS FOR SELECTING BOUNDARY VALUES

4. If the input is a Boolean, test for both Boolean values (T, F) and for a non-Boolean value.



5. Apply guidelines 1 and 2 to create **output values** at the minimum and maximum expected values.

   E.g., if the output is a table, create a minimum size table (1 row) and a maximum size table.

6. If data structures have boundaries, test these boundary values and values just above and just below them, respectively.

   E.g., for an array with bounds 1 to 10 → test array index = 0, 1, 2, 9, 10, 11.

BOUNDARY TESTING

Other boundary cases

Arithmetic

- Smallest/largest values
- Zero

Objects

- Null
- Circular
- Same object passed to multiple arguments (aliasing)

BOUNDARY TESTING: ARITHMETIC OVERFLOW

```
public int abs (int x)
// returns: |x|
```

- Tests for **abs**

  What are some values or ranges of x that might be worth testing?

- How about …

```
int x = -2147483648; // This is Integer.MIN_VALUE
System.out.println (x<0);              // true
System.out.println(Math.abs(x)<0); // also true!
```

From Javadoc for Math.abs:

> Note that if the argument is equal to the value of Integer.MIN_VALUE, the most negative representable int value, the result is that same value, which is negative.

BOUNDARY TESTING: DUPLICATE AND ALIASES

```
<E> void appendList(List<E> source, List<E>
                                destination) {
// modifies:  source, destination
// effects:   removes all elements of source and appends
//            them in reverse order to the end of
//            destination

        while (source.size()>0) {
            E element = source.remove(source.size()-1);
            destination.add(element)
        }
}
```

☞ **What happens if source and destination refer to the same thing?**

Testing for aliasing (shared references) is often overlooked!

ASU Course Registration System

Black Box Testing

Selecting Boundary Values and Designing Test Cases

EXAMPLE ASU BLACKBOX TESTING

Procedure: Student Record Search

In the ASU System, the *Maintain Student Information* use case uses a procedure that searches the database for the record of a given student. The input to the search procedure is a student ID. The output is either a success indication and the student record containing the given student ID or a failure indication and a message indicating the nature of the failure. The valid range of student IDs is from 1000000 to 9999999.

For the following test categories:
(a) typical
(b) boundary
(c) other
what specific test values and test cases should be used to test whether the search procedure works correctly and accepts only valid student IDs?

EXAMPLE ASU BLACKBOX TESTING:
SELECTING BOUNDARY VALUES

Typical values

Test Case	Test Value
1. Mid-range—valid student ID; existing student record	5000000
2. Mid-range—valid student ID; non-existent student record	5000001
3. Below lower range—invalid student ID	500000
4. Above upper range—invalid student ID	50000000

Boundary values

Test Case	Test Value
5. Minimum range—valid student ID	1000000
6. Minimum range—valid student ID	1000001
7. Minimum range—invalid student ID	999999
8. Maximum range—valid student ID	9999999
9. Maximum range—valid student ID	9999998
10. Maximum range—invalid student ID	10000000

Other values

Test Case	Test Value
11. Nonnumeric value—invalid student ID	5647ABC

EXAMPLE ASU BLACKBOX TESTING:
DESIGNING TEST CASES

Test case procedures: typical values

Test case 1:	Mid-range—valid student ID; existing student record
Setup:	Ensure that a record with student ID 5000000 exists in the data base
Test value:	5000000
Verification:	Successful retrieval of student record with ID 5000000

Test case 2:	Mid-range—valid student ID; non-existent student record
Setup:	Ensure that no record with student ID 5000001 exists in the data base
Test value:	5000001
Verification:	Unsuccessful retrieval and no such record error message

Test case 3:	Below lower range—invalid student ID
Setup:	None
Test value:	500000
Verification:	Invalid student ID error message

Test case 4:	Above upper range—invalid student ID
Setup:	None
Test value:	50000000
Verification:	Invalid student ID error message

EXAMPLE ASU BLACKBOX TESTING:
DESIGNING TEST CASES

Test case procedures: boundary values

Test case 5:	Minimum range—valid student ID
Setup:	Ensure that a record with student ID 1000000 exists in the data base
Test value:	1000000
Verification:	Successful retrieval of student record with ID 1000000

Test case 6:	Minimum range—valid student ID
Setup:	Ensure that a record with student ID 1000001 exists in the data base
Test value:	1000001
Verification:	Successful retrieval of student record with ID 1000000

Test case 7:	Minimum range—invalid student ID
Setup:	None
Test value:	999999
Verification:	Invalid student ID error message

Test case 8:	Maximum range—valid student ID
Setup:	Ensure that a record with student ID 9999999 exists in the data base
Test value:	9999999
Verification:	Successful retrieval of student record with ID 9999999

EXAMPLE ASU BLACKBOX TESTING:
DESIGNING TEST CASES

Test case 9:	Maximum range—valid student ID
Setup:	Ensure that a record with student ID 9999998 exists in the data base
Test value:	9999998
Verification:	Successful retrieval of student record with ID 9999998

Test case 10:	Maximum range—invalid student ID
Setup:	None
Test value:	10000000
Verification:	Invalid student ID error message

Test case procedures: other values

Test case 11:	Nonnumeric value—invalid student ID
Setup:	None
Test value:	5647ABC
Verification:	Invalid student ID error message

BLACKBOX TESTING EXAMPLE: VIDEO SALES

Procedure: validateRating

In the video sales and rental shop system, there is a procedure to validate the review rating input by the user. Recall that a review can be given a rating, which is an integer value, from 1 to 5.

For each of the following test categories:

(a) typical

(b) boundary

(c) other

what specific test values would you use to test whether the validateRating procedure works correctly and accepts only valid rating values?

BLACKBOX TESTING EXAMPLE: SOLUTION

We need to decide whether to treat the input values as a range or as a set of values. Since the number of allowed values is small, five in total, we can treat them as a set of values and test all of them. However, since the input values are also integers, we should also apply the tests for a range of values. We choose to do both and use the following test cases and values.

Member of set/typical values

Test Case	Test Value
1. Set member/in range—valid rating	1
2. Set member/in range—valid rating	2
3. Set member/in range—valid rating	3
4. Set member/in range—valid rating	4
5. Set member/in range—valid rating	5
6. Below range—invalid rating	-5
7. Above range—invalid rating	9

BLACKBOX TESTING EXAMPLE: SOLUTION

Boundary values

Test Case	Test Value
8. Minimum range—invalid rating	0
9. Maximum range—invalid rating	6

Other values

Test Case	Test Value
10. Numeric decimal—invalid rating	2.5
11. Non-numeric value—invalid rating	P
12. No input—invalid rating	
.	
.	
.	

Today

- Blackbox testing
- Testing in practice
  - JUnit
- Test coverage

Unit (object) testing

- Isolation
- Single object and test it by itself
- Check each object behavior

Rhythm of an Object test

- Create an object
- Invoke a method
- Check the result

Requirements for unit testing framework

- Must be automated
- Must verify themselves
- Must be easy to run simultaneously

JUnit Framework

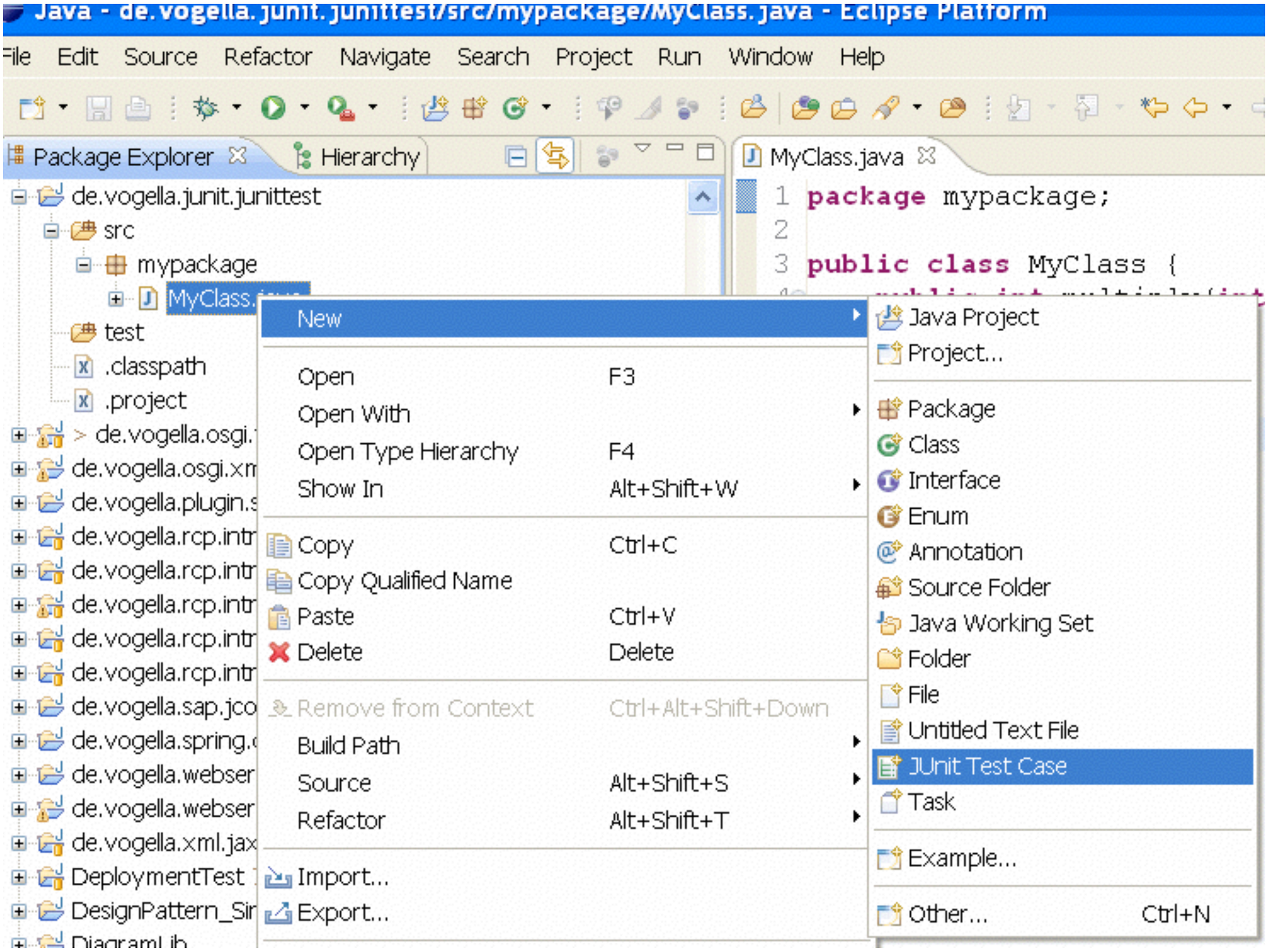
```
package testingTest;
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class MyFirstJUnitTest {
  @Test
  public void simpleAdd() {
    int result = 1;
    int expected = 1;
    assertEquals(result, expected);
  }
}
```

http://www.vogella.de/articles/JUnit/article.html

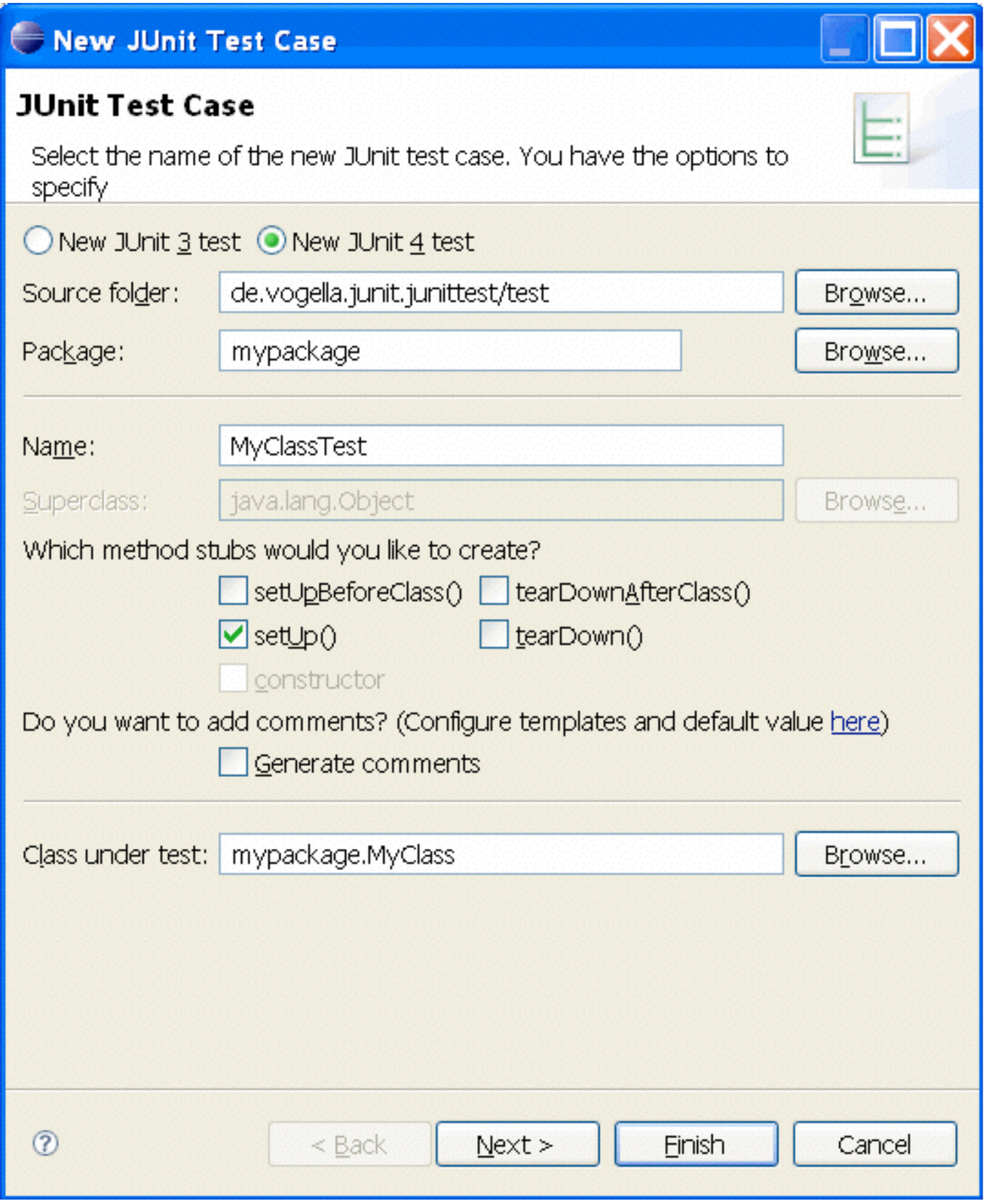
JUnit with Eclipse

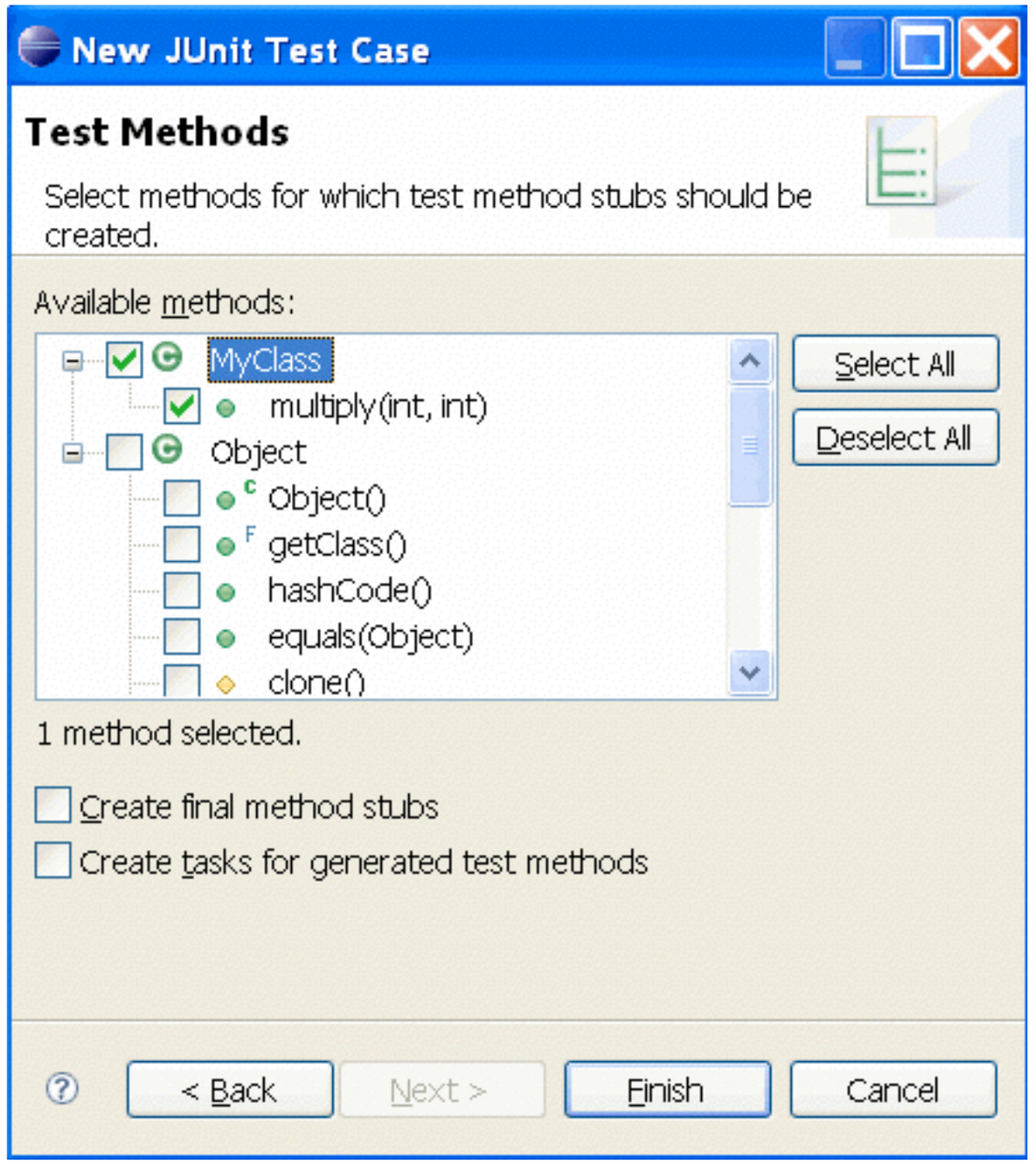
```
package mypackage;

public class MyClass {
  public int multiply(int x, int y) {
    return x / y;
  }
}
```

Create a test case



Select methods to test



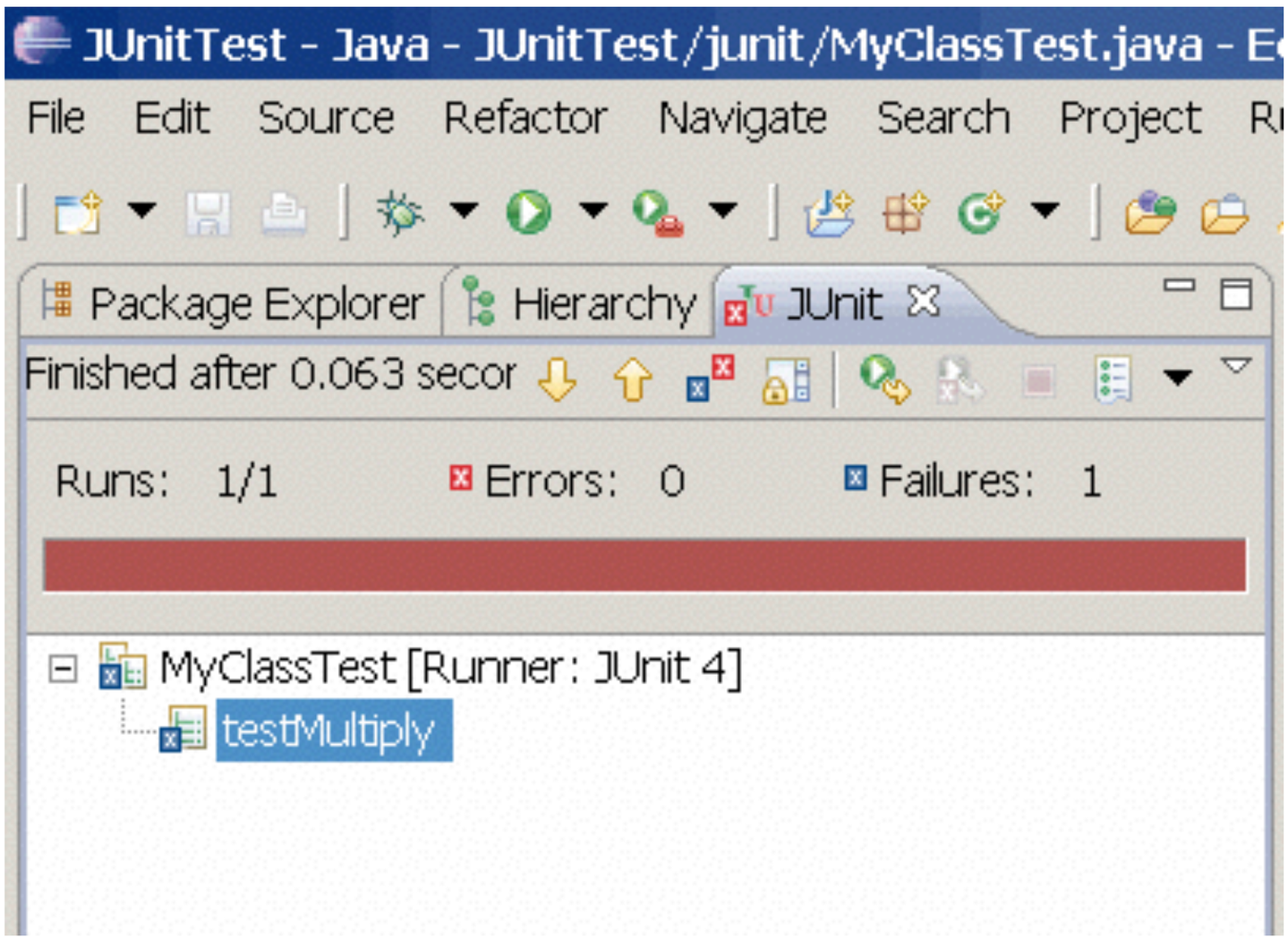


Edit the test case

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;
public class MyClassTest {
   @Test
   public void testMultiply() {
      MyClass tester = new MyClass();
      assertEquals("Result", 50, tester.multiply(10, 5));
   }
}
```

Run the test

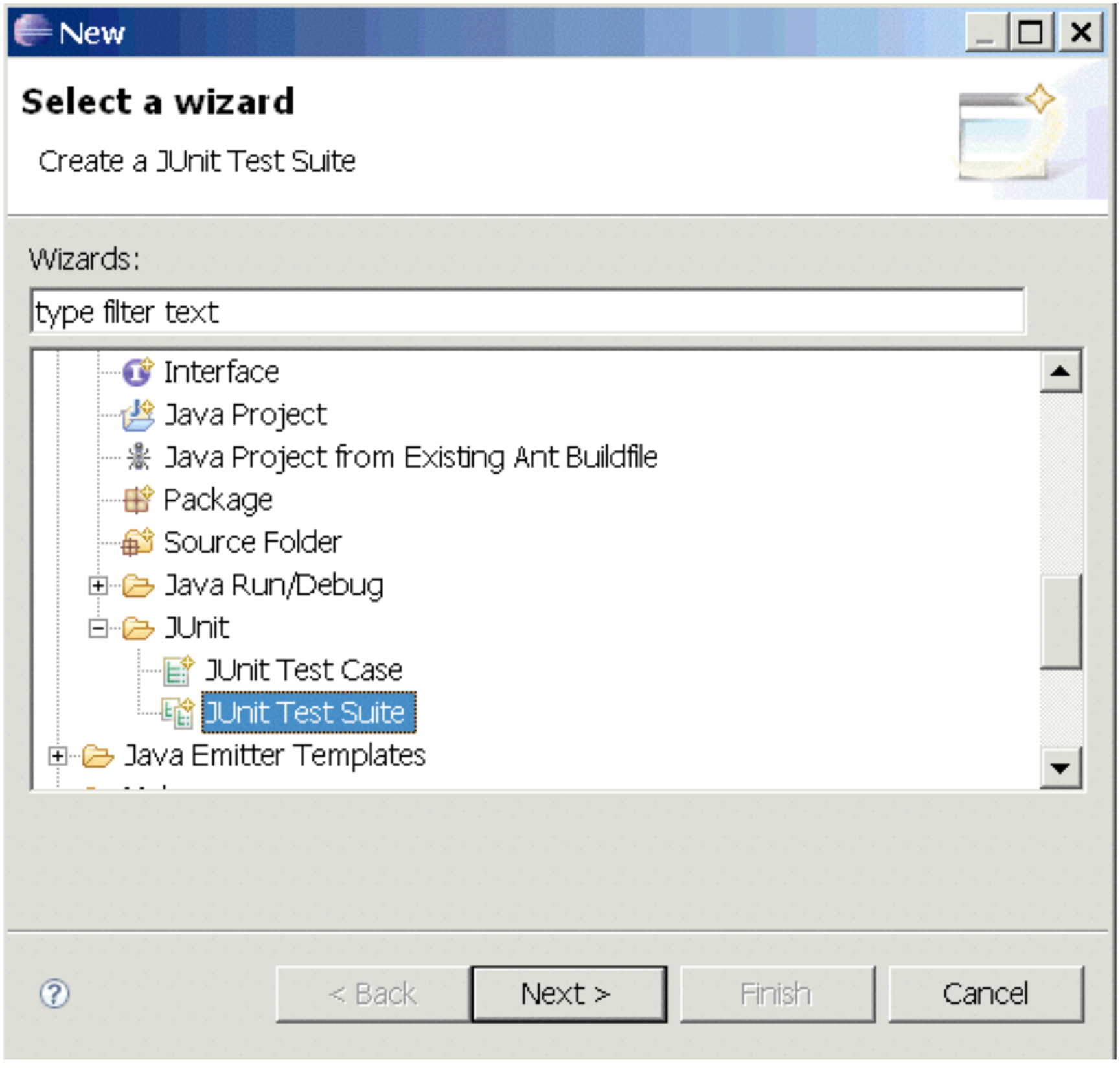

Refresh F5
Assign Working Sets...
Run As
Debug As
Profile As
Validate
1 Run on Server Alt+Shift+X, R
2 JUnit Test Alt+Shift+X, T
Run Configurations...


JUnitTest - Java - JUnitTest/junit/MyClassTest.java - E
File Edit Source Refactor Navigate Search Project R
Package Explorer
Hierarchy
JUnit
Finished after 0.063 secor
Runs: 1/1
Errors: 0
Failures: 1
MyClassTest [Runner: JUnit 4]
testMultiply

Test case and suite

- Test case: single test
- Test suite: a set of test cases
  - Want to run multiple tests at once

Create a test suite



Change the code

```
package mypackage;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses( { MyClassTest.class })
public class AllTests {
}
```

Annotations (Junit 4)

Annotation
@Test public void method()
@Before public void method()
@After public void method()
@BeforeClass public void method()
@AfterClass public void method()
@Ignore
@Test(expected=IllegalArgumentException.class)
@Test(timeout=100)

http://javarevisited.blogspot.hk/2012/06/junit4-annotations-test-examples-and.html

Test methods/Assertions

Statement
fail(String)
assertTrue(true);
assertsEquals([String message], expected, actual)
assertsEquals([String message], expected, actual, tolerance)
assertNull([message], object)
assertNotNull([message], object)
assertSame([String], expected, actual)
assertNotSame([String], expected, actual)
assertTrue([message], boolean condition)
try {a.shouldThroughException(); fail("Failed")} catch (RuntimeException e) {asserttrue(true);}

Naming conventions

- Test class names
  - Ends with “Test”
  - AccoutTest for Account
- Test method names
  - Starts with “test”
  - Name should *reveal intent*
    - *testWithdrawWhenOverdrawn()*

Test behavior, not methods

- Consider testing a stack
  - push, pop, peek

- Popping an empty stack fails somehow
- Peeking at an empty stack shows nothing
- Push an object onto the stack, then peek at it, expecting the same object you pushed
- Push an object onto the stack, then pop it, expecting the same object you pushed

Today

- Blackbox testing
- Testing in practice
  - JUnit
- Test coverage

Testing does not ...

- Guarantee the absence of bugs
- It shows bugs if they are

- When should we stop testing?

Because we usually have limited ...

Evaluating your tests

- How could we evaluate your tests?
  - Good/bad/good enough?
- How could we test your tests?

Test Coverage

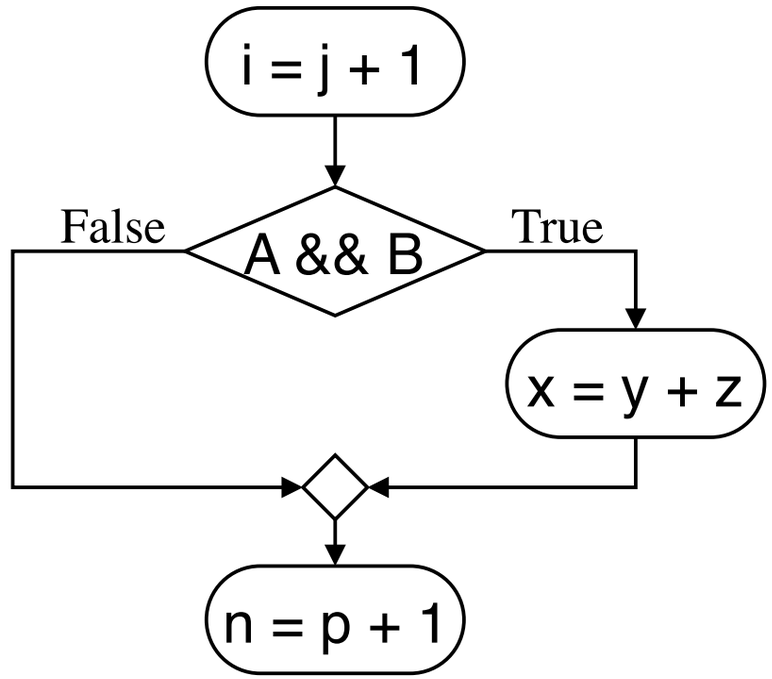
- Statement coverage
- Branch/Decision coverage
- Path coverage

Statement coverage

- Which statements have been executed?
- 100% statement coverage is not enough
  - Why?

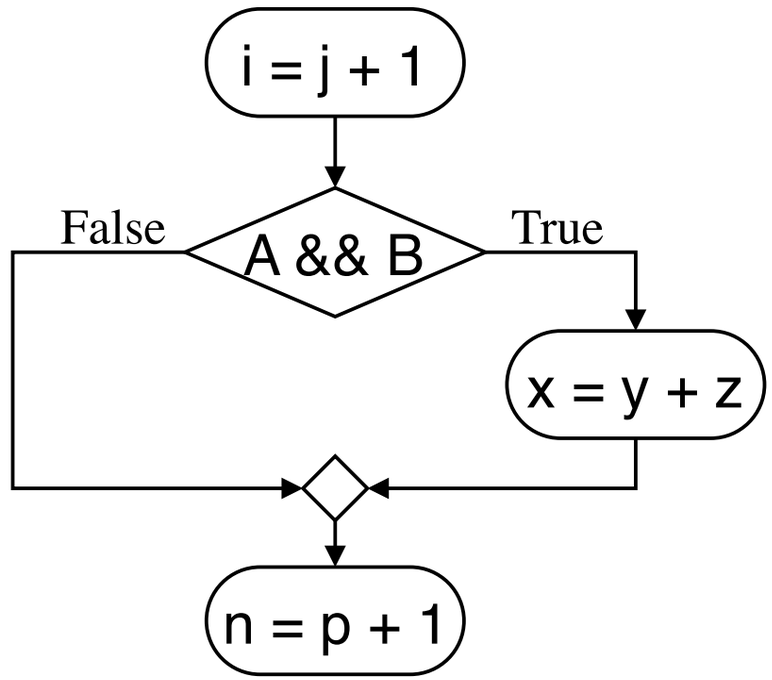
IS STATEMENT TESTING SUFFICIENT?

```
i = j + 1
if (A && B) {
   x = y + z;
   }
n = p + 1;
```



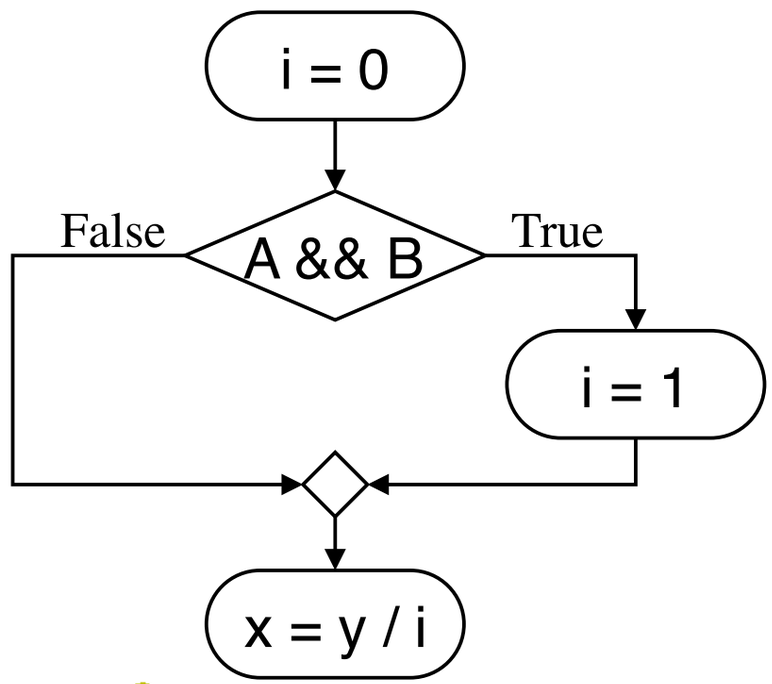
IS STATEMENT TESTING SUFFICIENT?

```
i = j + 1
if (A && B) {
   x = y + z;
   }
n = p + 1;
```



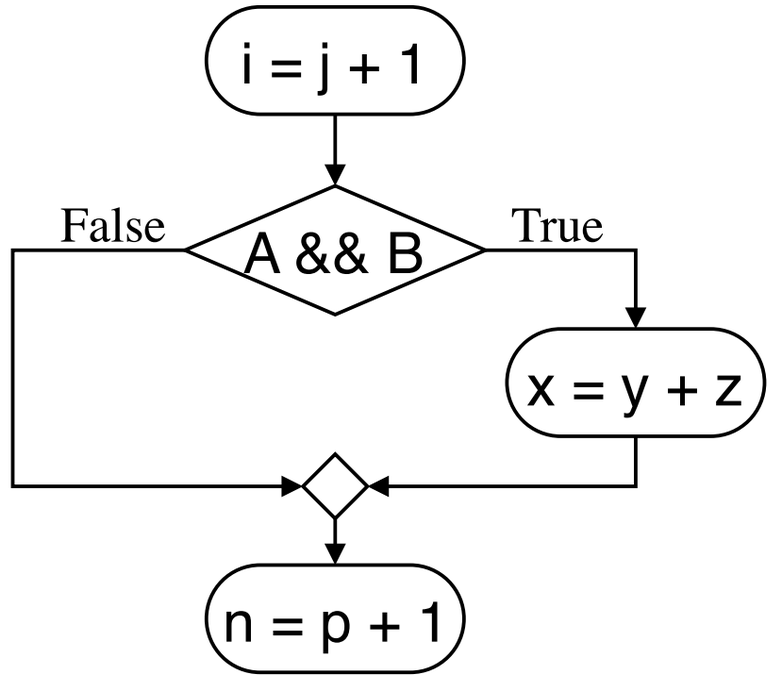
Suppose we modify the program to:

```
i = 0
if (A && B) {
   i = 1;
}
x = y / i;
```



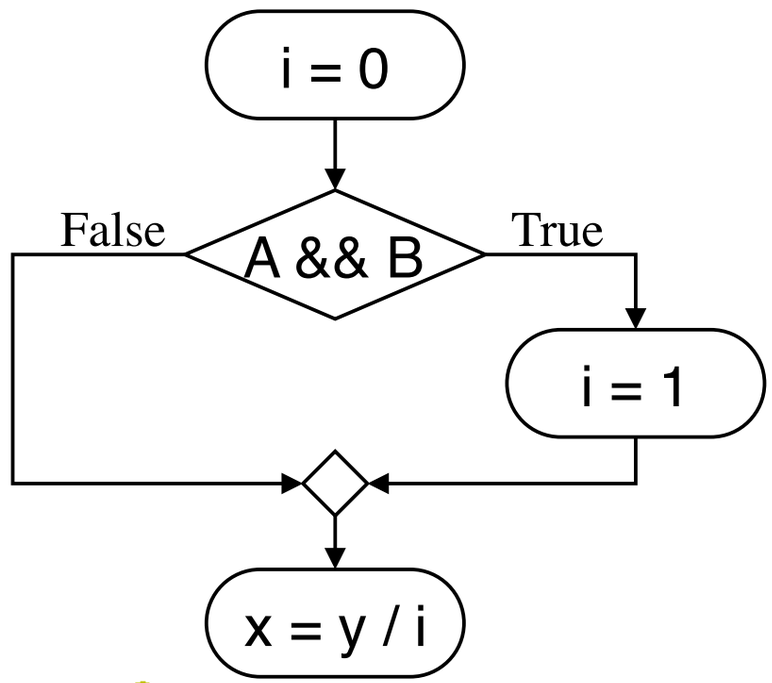
IS STATEMENT TESTING SUFFICIENT?

```
i = j + 1
if (A && B) {
   x = y + z;
   }
n = p + 1;
```



Suppose we modify the program to:

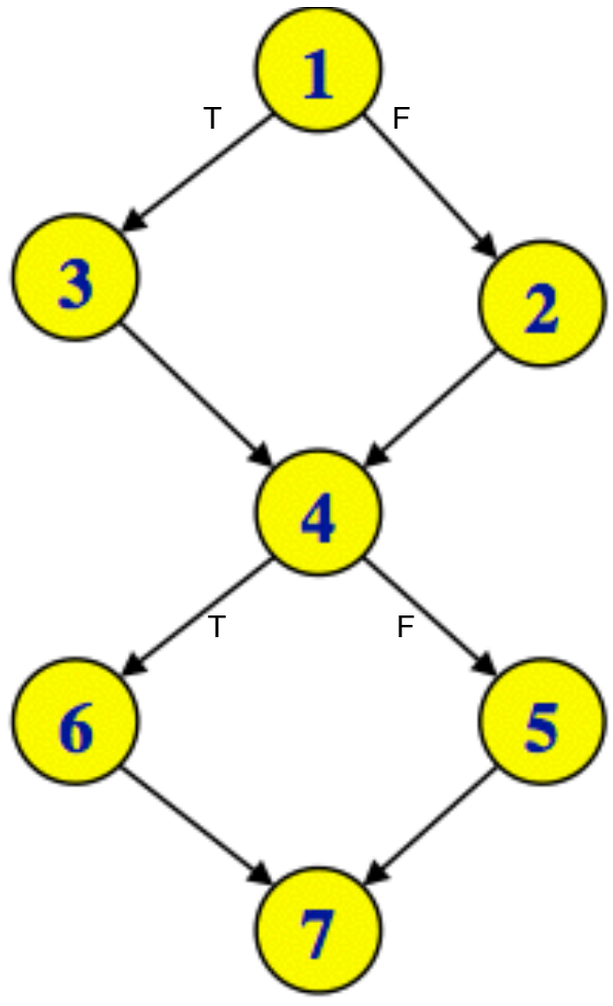
```
i = 0
if (A && B) {
   i = 1;
}
x = y / i;
```



Even though the false branch executes no additional statements, it needs to be tested!

Branch coverage

- How many branches are covered?
- Better than the statement coverage
- But 100 % branch coverage does not mean we explorer all possible cases
  - Why?



Path coverage

Path coverage

- In this the test case is executed in such a way that every path is executed at least once.
- All possible control paths taken, including all loop paths taken zero, once, and multiple (ideally, maximum) items in path coverage technique.
- Flow Graph, Cyclomatic Complexity and Graph Metrics are used to arrive at basis path.

https://sites.google.com/site/swtestingconcepts/home/test-design-techniques/for-white-box/statement-branch-and-path-coverage

Path coverage

- All possible paths
- Stronger than branch coverage
  - Why?