Software Engineering (Tests)

Lina YE

CentraleSupélec

https://www.lri.fr/~linaye/GL.html
lina.ye@centralesupelec.fr

Sequence 3, 2017-2018

Plan

Software Testing

Beizer's Test Philosophy

Level 0

- No difference between testing and debugging
- Adopted by many undergraduate CS majors
  - get their programs to compile
  - debug the programs with a few inputs

Beizer's Test Philosophy

Level 0

- No difference between testing and debugging
- Adopted by many undergraduate CS majors
  - get their programs to compile
  - debug the programs with a few inputs
- A program's incorrect behavior (validation) cannot be distinguished from a mistake within the program (verification)
- Not very useful to develop reliable or safe software

Beizer's Test Philosophy

Level 1

- The purpose of testing is to show correctness
- Run a collection of tests without finding failures

Beizer's Test Philosophy

Level 1

- The purpose of testing is to show correctness
- Run a collection of tests without finding failures
- Cannot demonstrate that
  - Is it a good software?
  - Are the set of tests good?

Beizer's Test Philosophy

Level 1

- The purpose of testing is to show correctness
- Run a collection of tests without finding failures
- Cannot demonstrate that
  - Is it a good software?
  - Are the set of tests good?
- How much testing remains to be done?
- No way to quantitatively express or evaluate the tests done

Beizer's Test Philosophy

Level 2

- The purpose of testing is to show failures
- Create testing professions (test engineers)
  - Put testers and developers into an adversarial relationship (not good for team morale)
  - What to do if no failures are found?

Beizer's Test Philosophy

Level 2

- The purpose of testing is to show failures
- Create testing professions (test engineers)
  - Put testers and developers into an adversarial relationship (not good for team morale)
  - What to do if no failures are found?
- Persistent problems: run a set of tests without failures
  - Is our software very good?
  - Is the testing weak?

Beizer's Test Philosophy

Level 3: good

- The purpose of testing is not to prove anything specific, but to reduce the risk of using the software
- Testing can show the presence of failures but not their absence.
- There is always some risk whenever we use software

Beizer's Test Philosophy

Level 3: good

- The purpose of testing is not to prove anything specific, but to reduce the risk of using the software
- Testing can show the presence of failures but not their absence.
- There is always some risk whenever we use software
- Collaborative work (positive): work together to reduce risk

Beizer's Test Philosophy

Level 3: good

- The purpose of testing is not to prove anything specific, but to reduce the risk of using the software
- Testing can show the presence of failures but not their absence.
- There is always some risk whenever we use software
- Collaborative work (positive): work together to reduce risk
- Level 3 $\rightarrow$ Level 4 (mental discipline that increases quality; testers train developers)

Criteria for classifying testing

1. Goal of testing:
   - performance, security, robustness, etc.

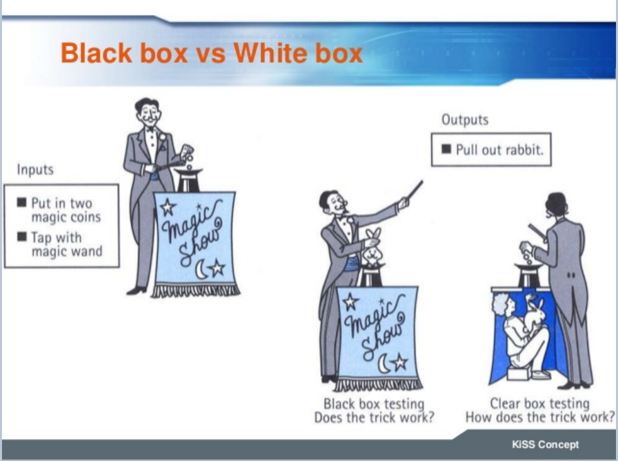
Criteria for classifying testing

1. Goal of testing:
   - performance, security, robustness, etc.
2. Testing levels (the cycle V)
   - Unit test
   - Integration test
   - Acceptance test

Criteria for classifying testing

1. Goal of testing:
   - performance, security, robustness, etc.
2. Testing levels (the cycle V)
   - Unit test
   - Integration test
   - Acceptance test
3. System nature under testing:
   - Black box: functional testing
   - White box: structural testing

Black and white box testing



Functional testing (black box)

- Do not take into account the implementation
  - Testing is based only on the inputs/outputs
  - Data coverage

Functional testing (black box)

- Do not take into account the implementation
  - Testing is based only on the inputs/outputs
  - Data coverage
- Generate test cases from the specification
  - Pre-condition: generate inputs
  - Post-condition: generate outputs

Functional testing (black box)

- Do not take into account the implementation
  - Testing is based only on the inputs/outputs
  - Data coverage
- Generate test cases from the specification
  - Pre-condition: generate inputs
  - Post-condition: generate outputs
- Two common techniques
  - Random testing: generate arbitrarily inputs
  - Testing by partitioning input space

Functional testing: example

- Example of triangle
  - input: three integers a, b and c
  - output: right-angled, isoceles, equilateral, invalid

Functional testing: example

- Example of triangle
  - input: three integers a, b and c
  - output: right-angled, isoceles, equilateral, invalid
- Lots of test cases required
  - right-angled triangle, isoceles triangle, equilateral triangle, invalid
  - all permutations of two equal sides
  - all permutations of $a+b<c$
  - all permutations of $a+b=c$
  - all permutations of $a=b$ and $a+b=c$
  - values in MAXINT
  - non-integer inputs

Random testing

- Pick possible inputs uniformly by treating all inputs as equally valuable
- But: defects are not distributed uniformly
- Assume Roots applies quadratic equation $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$, which fails if $b^2 - 4ac = 0$ and $a = 0$
- Random sampling is unlikely to choose a=0 and b=0
- Many defects are related to specific inputs
- Input space partitioning

Input space partitioning

- Impossible to test all
  - an integer of 32-bit as input $\rightarrow$ 4,294,967,296 values
  - one hour on the recent machine
  - 3 integers of 32-bit, $(2^{32})^3 \approx 10^{28}$ legal inputs: 2.5 billion years with 10^{12} tests/s

Input space partitioning

- Impossible to test all
  - an integer of 32-bit as input $\rightarrow$ 4,294,967,296 values
  - one hour on the recent machine
  - 3 integers of 32-bit, $(2^{32})^3 \approx 10^{28}$ legal inputs: 2.5 billion years with 10^{12} tests/s
- partition input space into equivalent classes
  - The values in the same equivalent class have the same behaviors from the specification point of view

Input space partitioning

- Impossible to test all
  - an integer of 32-bit as input $\rightarrow$ 4,294,967,296 values
  - one hour on the recent machine
  - 3 integers of 32-bit, $(2^{32})^3 \approx 10^{28}$ legal inputs: 2.5 billion years with 10^{12} tests/s
- partition input space into equivalent classes
  - The values in the same equivalent class have the same behaviors from the specification point of view
- Test cases: for each equivalent class
  - a value of the limit
  - a value just before the limit
  - a value in the middle
  - a value just after the limit (robustness test)

Partition: examples

- Example 1: absolute value
  - 2 equivalent classes: value$\leq$0 and value$\geq$0
  - 9 test cases: MinInt, MinInt+1, -10, -1, 0, 1, 5, MaxInt-1, MaxInt

Partition: examples

- Example 1: absolute value
  - 2 equivalent classes: value$\leq$0 and value$\geq$0
  - 9 test cases: MinInt, MinInt+1, -10, -1, 0, 1, 5, MaxInt-1, MaxInt
- Example 2: insert to a list (size is 20)
  - List level:
    - empty list or 1 element
    - full list 19 or 20 elements
    - list of 10 elements (middle)

Partition: examples

- Example 1: absolute value
  - 2 equivalent classes: value$\leq$0 and value$\geq$0
  - 9 test cases: MinInt, MinInt+1, -10, -1, 0, 1, 5, MaxInt-1, MaxInt
- Example 2: insert to a list (size is 20)
  - List level:
    - empty list or 1 element
    - full list 19 or 20 elements
    - list of 10 elements (middle)
  - Insertion level
    - just before and after the first element
    - just before and after the last element
    - in the middle of the list

Structurel testing (white box)

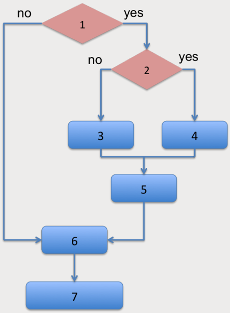
- Internal structure of software as criteria to generate test cases

Structurel testing (white box)

- Internal structure of software as criteria to generate test cases
- Coverage criteria
  - Block/Instruction coverage
    - each instruction should be covered by at least one test case
  - Branch/Decision coverage
    - each branch should be covered by at least one test case
    - implies the block coverage
  - Path coverage
    - each execution path should be covered by at least one test case
    - implies branch coverage

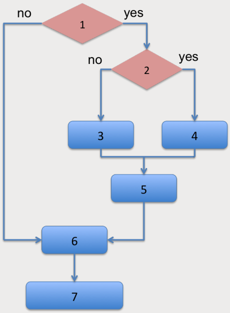
Structurel testing: example

- Coverage of instruction blocks
- Coverage of branches
- Coverage of paths



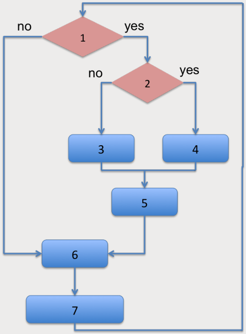
Structurel testing: example

- Coverage of instruction blocks
  - 2 paths suffice
- Coverage of branches
  - 3 paths required
- Coverage of paths
  - 3 paths required



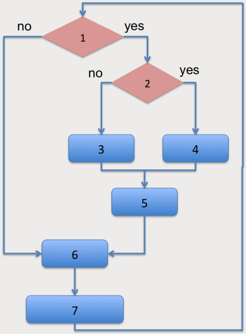
Structurel testing: example

- Coverage of instruction blocks
  - 2 paths suffice
- Coverage of branches
  - 3 paths required
- coverage of paths
  - 3 paths?
  - how many iterations needed? N?

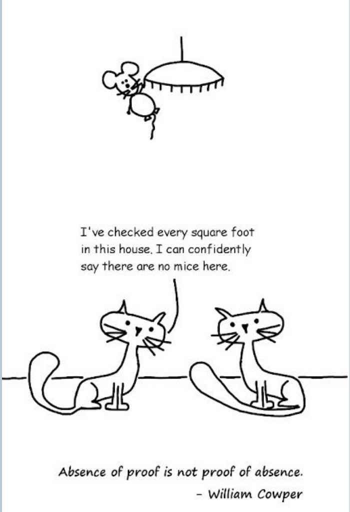


Structurel testing: example

- Coverage of instruction blocks
  - 2 paths suffice
- Coverage of branches
  - 3 paths required
- coverage of paths
  - 3 paths?
  - how many iterations needed? N?
  - number of paths 3^N exponential with the number of iterations



Really all covered?



Cyclomatic complexity

- A source code complexity measurement that determines the number of linearly independent path (not a sub-path of another path)
- It is calculated by developing a Control Flow Graph of the code
- Lower the Program's cyclomatic complexity, lower the risk to modify and easier to understand

Cyclomatic complexity

- A source code complexity measurement that determines the number of linearly independent path (not a sub-path of another path)
- It is calculated by developing a Control Flow Graph of the code
- Lower the Program's cyclomatic complexity, lower the risk to modify and easier to understand
- Calculate cyclomatic complexity: $CC = E - N + 2*P$
  - E = number of edges in the flow graph
  - N = number of nodes in the flow graph
  - P = number of nodes that have exit points

Control flow graph

- The control structure of a program can be represented by the control flow graph of the program.
- The control flow graph G = (N, E) of a program consists of a set of nodes N and a set of edge E.

Control flow graph

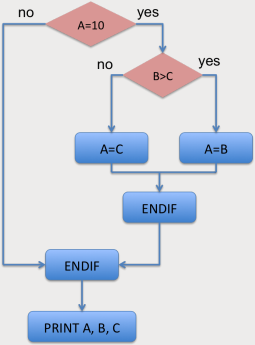
- The control structure of a program can be represented by the control flow graph of the program.
- The control flow graph G = (N, E) of a program consists of a set of nodes N and a set of edge E.
  - A statement node contains a sequence of statements. The control must enter from the first statement and exit from the last statement.
  - A decision node contains a conditional statement that creates 2 or more control branches.
  - A merge node usually does not contain any statement and is used to represent a program point where multiple control branches merge.

Control flow graph

- The control structure of a program can be represented by the control flow graph of the program.
- The control flow graph G = (N, E) of a program consists of a set of nodes N and a set of edge E.
  - A statement node contains a sequence of statements. The control must enter from the first statement and exit from the last statement.
  - A decision node contains a conditional statement that creates 2 or more control branches.
  - A merge node usually does not contain any statement and is used to represent a program point where multiple control branches merge.
  - There is an edge from node n_1 to node n_2 if the control may flow from the last statement in n1 to the first statement in n2.

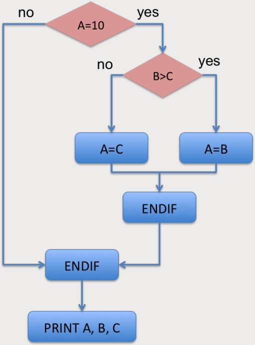
Example

```
IF A == 10 THEN
    IF B > C THEN
        A = B
    ELSE
        A = C
    ENDIF
ENDIF
Print A
Print B
Print C
```



Example

```
IF A == 10 THEN
    IF B > C THEN
        A = B
    ELSE
        A = C
    ENDIF
ENDIF
Print A
Print B
Print C
```



E=8; N=7; P=1

CC=E-N+ 2*P =8-7+2=3

Functional vs. structural

- The structural approaches can find program errors more easily (verification)
- The functional approaches can find incorrect behaviors more easily (validation)
- They are complemented:
  - missing functionality defects: functional testing
  - decision defects: structural testing

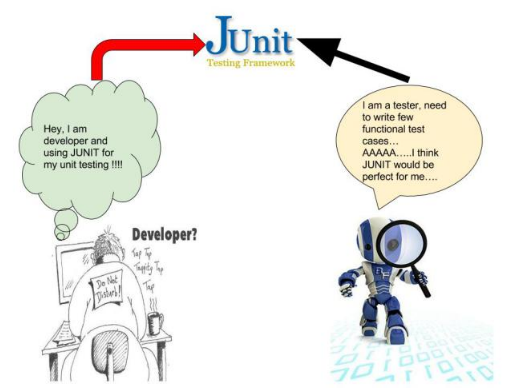
JUnit

- JUnit is a unit testing framework designed for the Java programming
  - Authors: Erich Gamma, Kent Beck

Objective

If the test cases are easy to be created and executed, then the developers would be required to do this.

JUnit


JUnit
Testing Framework
Hey, I am developer and using JUNIT for my unit testing !!!!
I am a tester, need to write few functional test cases… AAAAA.....I think JUNIT would be perfect for me….
Developer?
Do Not Disturb!

Test Automation

- A test script to define:
  - the actions sent to the System Under Test (SUT)
  - the responses expected of SUT
  - the way to determinate whether a test fails or not
- Test execution system
  - read and execute the scripts on the SUT
  - save test results

What is a JUnit test

- A test script is just a set of Java methods
  - The idea is to create and use the objects before verifying whether these objects have the good properties.
- Assertions
  - A package containing the functions that allow the verification of different properties:
    - equality between objects
    - reference identity
    - reference null/non-null
  - The assertions are used to determinate the verdict of a test: Pass or Fail

A JUnit test case

```
/* Test of setName() method, class Value */

@Test

      public void createAndSetName(){
              Value v1=new Value();

              v1.setName("Y");

              String expected="Y";
              String actual=v1.getName();

              Assert.assertEquals(expected, actual);
        }
```

A JUnit test case

```
/* Test of setName() method, class Value */

@Test

        public void createAndSetName(){
                Value v1=new Value();

                v1.setName("Y");

                String expected="Y";
                String actual=v1.getName();

                Assert.assertEquals(expected, actual);
        }
```

define this method as a test

A JUnit test case

```
/* Test of setName() method, class Value */

@Test

      public void createAndSetName(){
              Value v1=new Value();

              v1.setName("Y");

              String expected="Y";
              String actual=v1.getName();

              Assert.assertEquals(expected, actual);
      }
```

confirm that setName saves the name of v1

A JUnit test case

```
/* Test of setName() method, class Value */

@Test

      public void createAndSetName(){
              Value v1=new Value();

              v1.setName("Y");

              String expected="Y";
              String actual=v1.getName();

              Assert.assertEquals(expected, actual);
      }
```

verify the name of v1

A JUnit test case

```
/* Test of setName() method, class Value */

@Test

      public void createAndSetName(){
              Value v1=new Value();

              v1.setName("Y");

              String expected="Y";
              String actual=v1.getName();

              Assert.assertEquals(expected, actual);
        }
```

expected and actual should be the same

Test verdicts

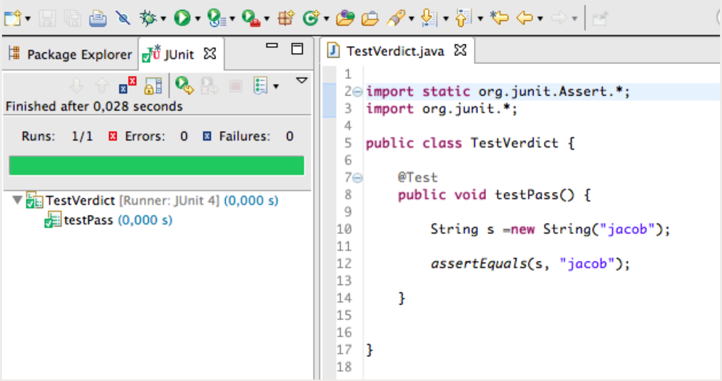
A verdict is the execution result of one test.

- Pass: the test has been correctly executed and the software has the expected behavior.
- Fail: the test has been correctly executed and the software has the unexpected behavior.
- Error: the test has not been correctly executed, which may due to
  - unexpected event during the test
  - the test cannot be initialized correctly

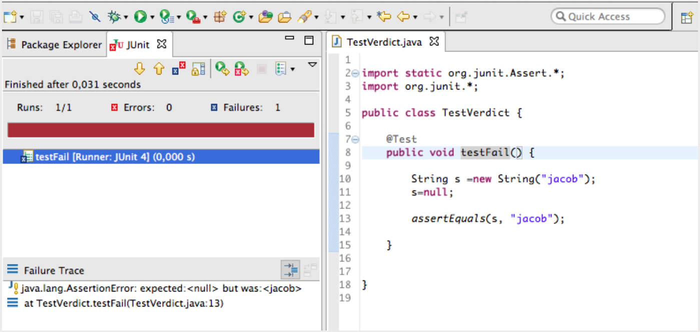
Test verdicts

```
@Test
   public void testErrorVsTestFailure() {

      String s =new String("jacob");
      s=null;

      assertEquals('j', s.charAt(0) );
      /*above line throws test error as you are trying to
      access charAt() method on null reference*/

      assertEquals(s, "jacob"));
      /*above line throws Test failure as the actual
      value null is not equal to "jacob"*/
      }
```

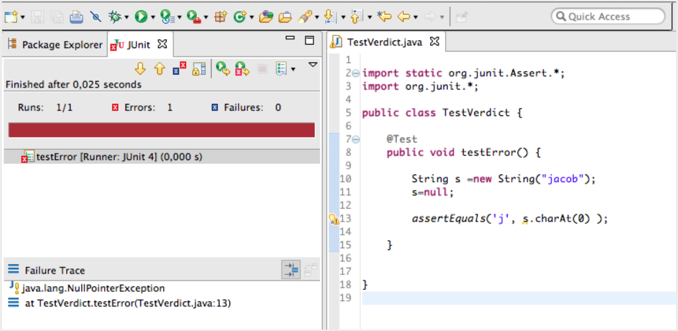
Eclipse interface



Eclipse interface



Eclipse interface



Eclipse

- To create a new test class from an existing class: click right on the class → New → JUnit Test Case
- To execute the set of tests
  - use the same arrow for program execution
  - the test result shown to the left

Organization of JUnit tests

- Each method corresponds to a test with its own verdict (pass, error, fail).
- Conventionally, all tests for the same class are collected in the same test class
  - naming convention:
    - Class to be tested: NameClass
    - Class containing tests: NameClassTest
- demo

Execute JUnit tests

- There is no graphic interface of JUnit to run the tests, but an API is available to be used.
- Eclipse uses the API of JUnit to provide graphic interface to run tests.

Execute JUnit tests

- There is no graphic interface of JUnit to run the tests, but an API is available to be used.
- Eclipse uses the API of JUnit to provide graphic interface to run tests.
- When a test class is executed, all test methods are executed.
- The order to execute these methods is not predefined.
- It is necessary to write the tests whose result is independent of the execution order.

Assertions

- The assertions are defined in the class Assert:
  - if an assertion is true, then the execution continues
  - if an assertion is false, then the execution terminates and the test result is fail
  - if any other exception is generated, the test result is error
  - if no assertion is false in the method, the test result is pass
- All assertion methods are static.

Assertion methods

- Test boolean condition (true or false)
  - assertTrue(condition)
  - assertFalse(condition)
- Test if an object is null or non-null
  - assertNull(object)
  - assertNotNull(object)
- Test if two objects are identical (i.e., two references to the same object)
  - assertSame(expected, actual): true if expected==actual
  - assertNotSame(expected, actual)

Assertion methods

- Test equality between two objects
  - assertEquals(expected, actual): valid if expected.equals(actual)
- Test equality between two arrays
  - assertArrayEquals(expected, actual)
    - The arrays should have the same size
    - for all correct values of i, test according to the cases: assertEquals(expected[i], actual[i]) or assertArrayEquals(expected[i], actual[i])
- There exists also an assertion that always fails: fail()

The parameters of assertion methods

- If an assertion method has two parameters, the first is the expected value and the second is the actual value
  - This has no impact on the test result but is used to send the message to users
- All assertion methods can have an extra parameter whose type is String, which is on the first place. This parameter will be included in the error message if the assertion fails.
  - Examples:
    fail(message)
    assertEquals(message, expected, actual)

Equality assertions

- assertEquals(a, b) is based on the method equals() of the class that is tested
  - This assertion is to evaluate a.equals(b)
  - Recall: if the method equals is not defined in the class, then it is inherited from the parent class Object
- If a and b are the primitive types like int, boolean, ..., then the following behavior is implemented for assertEquals(a, b):
  - a and b with the equivalence of their object type: (Integer, Boolean, ...), and then a.equals(b) is evaluated.

Assertions for non-integer number

- When one compares the non-integer number (double or float), there is an extra parameter that is necessary: delta
- The assertion evaluate:
  Math.abs(expected-actual)$\leq$delta
  This is done to avoid the rounding errors
- Example:
  assertEquals(aDouble, anotherDouble, 0.0001)

Test fixture

The purpose of a test fixture is to ensure that there is a well known and fixed environment in which tests are run so that results are repeatable.

- The fixtures are composed of
  - The objects and the resources used for tests
  - The initialization (setup) and deallocation (teardown) of these objects and resources.

Setup

- The set of tasks effectuated before each test.
  - Example: create the interesting objects, based on which one works, open a connection network, etc...
- Use the key word @Before before the methods
- All methods with this key word will be executed before each test, but with any possible order.

Teardown

- The set of tasks effectuated after each test.
  - Example: be sure that the resources are liberated, reset the system in the good state for the following tests
- With the key word @after before the methods
- All methods with this key word will be executed after each test, but with any possible order.
- The methods are executed even when the test fails

Setup and Teardown: Example

```
public class OutputTest {
    private File output;
    @Before
        public void createOutputFile () {
            output=new file (...);
        }
    @After
        public void deleteOutputFile () {
            output.delete();
        }
    @Test
        public void test1WithFile () {
            /** code for test case objective */
        }
    @Test
        public void test2WithFile () {
            /** code for test case objective */
        }
}
```

Execution order

1. createOutputFile()
2. test1WithFile()
3. deleteOutputFile()
4. createOutputFile()
5. test2WithFile()
6. deleteOutputFile()

- Remark: test1WithFile can be executed after test2WithFile

Once-only Setup

- The set of tasks effectuated only one time before the set of tests
  - Example: restart a server
- With the key word @BeforeClass before the methods
- Can be used for a static method

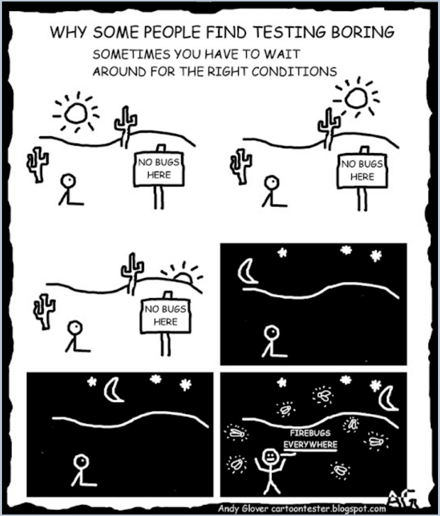
```
@BeforeClass
        public static void anyNameHere () {
                /** class setup code here */
        }
```

Once-only Teardown

- The set of tasks effectuated only one time after the set of tests
  - Example: stop a server
- With the key word @AfterClass before the methods
- Can be used for a static method

```
@AfterClass
        public static void anyNameHere () {
                /** class cleanup code here */
            }
```

Boring test?


WHY SOME PEOPLE FIND TESTING BORING
SOMETIMES YOU HAVE TO WAIT AROUND FOR THE RIGHT CONDITIONS
NO BUGS HERE
NO BUGS HERE
NO BUGS HERE
FIREBUGS EVERYWHERE
Andy Glover cartoontester.blogspot.com

What is mutation testing

- Mutation testing involves modifying a program in small ways.
  - Such modifications model small defects that may appear during the development.
- Mutation testing is a form of white-box testing.
  - estimate/improve the efficiency of test suites
  - find out the problems in the SUT.

Principles

Let Prog be a program and Tests be a set of tests:

- Apply the mutations on the program Prog
  - Each mutant is created by applying one mutation on Prog
  - A set of mutants $Prog_1, Prog_2, \ldots, Prog_n$
- Run the set of tests Tests on each mutant
  - We say that Tests kills the mutant $Prog_i$ if an error is detected
- If Tests kills k mutants on n
  - The mutation coverage of Tests is calculated by k/n
  - Tests is considered as perfect when $k = n$

Mutation testing is totally automatic.

Mutation equivalence

- Tests is not perfect when: $(k/n) < 1$
- In practice: some mutants are not different from original program
  - Such mutants are called equivalent mutations

Mutation equivalence

- Tests is not perfect when: $(k/n) < 1$
- In practice: some mutants are not different from original program
  - Such mutants are called equivalent mutations

```
int i=2;
if (i>=1) {
      return "foo";
}
...

int i=2;
if (i>1) {
      return "foo";
}
```

Mutation types

- Mutation of values: modify the values of constants or of parameters
  - Example: the bound for cycles, the initial value, etc.
- Mutation of decisions: modify the conditions
  - Example: replace the comparison $>$ by $>=$ or $<$.
- Mutation of declarations: delete or inverse the order of code lines.
  - Example: delete the variable incrementation in a cycle.

Mutation generation

- Mutation of source code
  - The mutations are effectuated by modifying the source code that is then recompiled.
- Mutation of assembly language
  - The mutations are effectuated by modifying the assembly code.

Mutation generation

- Mutation of source code
  - The mutations are effectuated by modifying the source code that is then recompiled.
- Mutation of assembly language
  - The mutations are effectuated by modifying the assembly code.
- Advantage of source code
  - A great quantity of mutations can be effectuated
  - The mutations are similar to the errors that may generated by a programmer
  - The mutations can easily be understood
- Advantage of assembly code
  - Mutation generation is quicker.

PIT: mutation testing for Java

- PIT is a system of mutation testing for Java based on the source code.
- Two methods
  - mutation coverage: measure the efficiency of the tests
  - line coverage: a coverage of detailed code (line by line)
- More information: http://pitest.org/

PIT

```
        // Verify for a ".." component at next iter
        if ((newcomponents.get(i)).length() > 0 
        {
            newcomponents.remove(i);
            newcomponents.remove(i);
            i = i - 2;
            if (i < -1)
            {
                i = -1;
            }
        }
    }
```

Formal methods

Principles

- Mathematic proofs are used on the program to demonstrate that it holds some properties
- Difficult to be used
- The only method with the guarantee
- For some critical applications (e.g., Meteor)

Formal methods

Principles

- Mathematic proofs are used on the program to demonstrate that it holds some properties
- Difficult to be used
- The only method with the guarantee
- For some critical applications (e.g., Meteor)

Tool examples

- B method
- Isabelle, Coq