Cosa significa provare un programma?

Provare un programma (con un solo test)

- **Algoritmo del testing:**
  - Progettare un **caso di test**
  - Eseguire il caso di test
  - Osservare l'esito del caso di test
  - *SE* il risultato non è quello atteso : **Fallimento** (del programma)
    - RIPETERE
      - Cercare il difetto
      - Correggere il difetto
      - Rieseguire il caso di test
    - FINO A CHE l'esito del test non è quello atteso

Progettare un caso di test

Test case specification

- Minimal set of information able to describe a test case specfication
  - ID Number and Description
  - Preconditions
    - *Hypotheses that must be veriified before the test is executed*
  - Input values
  - Expected output values (the «Oracle»)
  - Expected Postconditions
    - *Hypotheses that must be verified after the test is executed*

ID	Precond	Input	Expected Output	PostCond

Test case execution report

- In addition to the information needed for test case specification:
  - Output values
  - Result
    - *Success if preconditions and postconditions are true and output values are judged equivalent to expected output values*
    - *Failure if preconditions and postconditions are true but output values are not judged equivalent to expected output values*
    - *Not applicable (N/A) if al least a precondition or a postcondition is not true*

ID	Precond	Input	Expected Output	Output	PostCond	Result

Input

- Input can be described as attribue-value pairs …
- … or by a stream
  - For example, they can be represented by a text file
- Input may be described by a sequence of actions
  - For example in GUI based applications, input may be represented by a sequence of user actions
- Input may also be described by a set of signals and by the time of arrival to the system / exit from the system

Input

- Some examples, in order of ascending difficulty:
  - Testing of methods
    - The list of variable is known, the type of values is known
  - Testing of (web) services
    - The list of variables is known, the type of values is unknown/may be changed by the user
  - Testing of Command Line programs
    - The length of the list of variables is unknown/may be changed, the type of values is unknown/may be changed
  - Testing of GUI based programs
    - Input is composed of a list of events and input values, the length of the sequence may assume any possible value
  - Testing of videogames / real time programs
    - The (analogic) time of execution of each event / input insertion is relevant and may assume any possible value
  - Testing of distributed real time programs
    - Multiple analogic inputs may occur at different times and may assume any possible value

Example: method test case specification

- Method prototype:
  - int sum (int x, int y)

- Example of Test case Input Specifications

ID	Precond	Input	Expected Output	PostCond
		x=1 y=2		

- Input is specified as a set of pairs (name, value)

Example: GUI based program test case specification

- Program to test
  - notepad.exe
  - Function to test: search the word «pippo» into the text

- Example of Test case Input Specifications

ID	Precond	Input	Expected Output	PostCond
		Open notepad Click on Modify Click on Find.. Type «pippo» Click on Find Next		

- Input is specified as a sequence of events to be triggered on the GUI

Output

- The output may be specified:
  - As values/objects in method / services testing
  - As a text on screen in command line programs
  - As a graphical screen / object on the screen in GUI program testing
  - As a combination of an output in a given time / interval in a real time system / videogame
  - …

Example: static method test case specification

- Method prototype:
  - static int sum (int x, int y)

- Example of Test case Output Specifications

ID	Precond	Input	Expected Output	PostCond
		x=1 y=2	3	

- Output is specified as a int value

Example: GUI based program test case specification

- Program to test
  - notepad.exe
  - Function to test: search the word «pippo» into the text of the file prova.txt
-

ID	Precond	Input	ExpectedOutput	PostCond
		Open notepad prova.txt Click on Modify Click on Find.. Type «pippo» Click on Find Next	"pippo found" text message	

- Output is specified as a widget shown on the GUI

Preconditions and postconditions

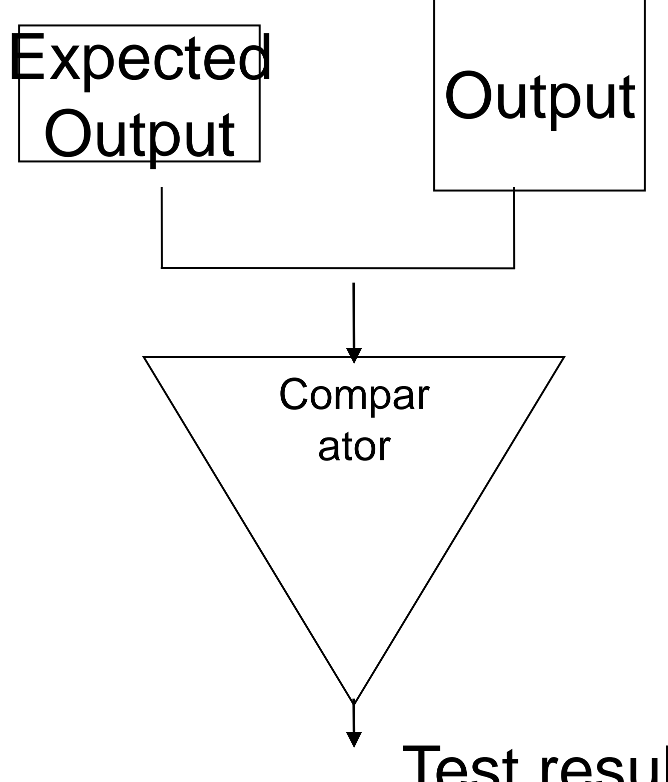
- Preconditions and postconditions may be about:
  - The existence and state of external services or data sources
    - *For example files, databases, services, global variables …*
  - The state of the application before/after the test execution
    - *For example: the correct authentication, the presence or absence of such data in the database, the reaching of a specific interface*
      - ***Usually, if a test is quite complex, then intermediate conditions are added to improve the fault localization ability of the test case***

Output, Expected Output, Test Result

- The (Actual) **Output** obtained by the test execution has to be compared with the **Expected Output** designed in the test case in order to evaluate the test Result

- The **Comparator** is able to compare the oracle expected behavior and the tested software behavior and to evaluate if a failure has occurred

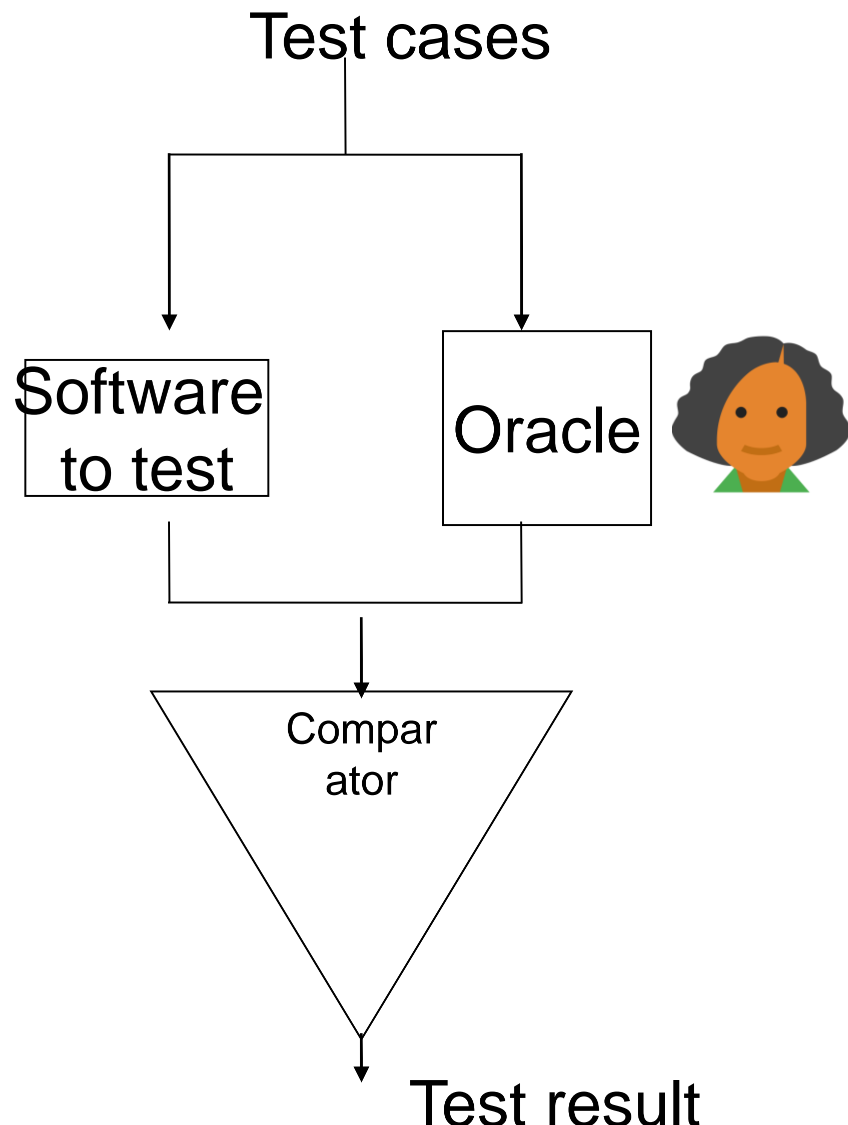
- The Comparator has to evaluate *objectively* if the Expected Output and the Output are equal / identical / equivalent between them



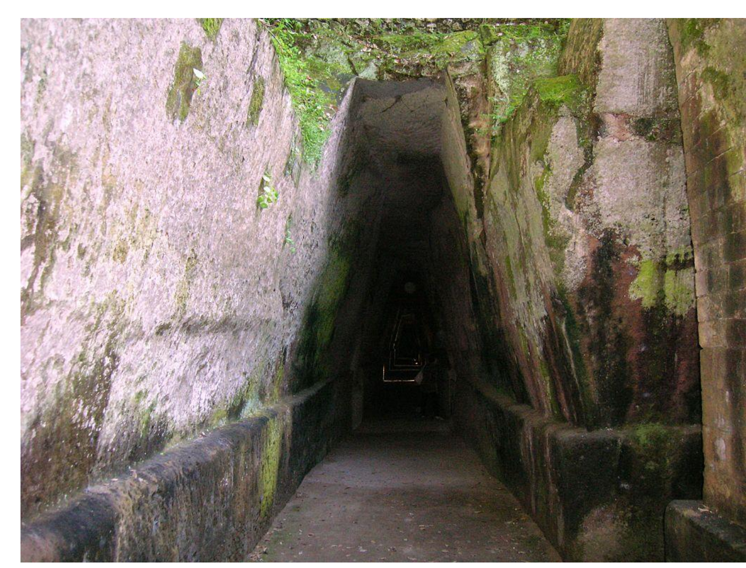
the Oracle

- the **Oracle** knows exactly what are the Expected Outputs of the system under test in response to the each test case

- In the easiest scenario, the Oracle is represented by the column of the Expected Outputs



Who is the Oracle?



- Human oracle
  - He evaluates the success of the test cases on the basis of the requirements and of his personal judgement
    - *For example, in acceptance testing*
- Software oracle
  - An Oracle is another software having the same behavior of the software to be tested
    - *For example, a known version of bubblesort can be used to test a faster quicksort*
    - *A previous version of a software can be the oracle to evaluate the behavior of a newer software offering the same functionality (regression testing)*
- Implicit oracle
  - Testing against crashes, the oracle found a failure in each case a crash occurs
- Formal specification oracle
  - If the requirements are expressed in a formal way, then the oracle can be automatically derived from them

What is the role of the comparator?

- If the oracle provides exact expected values for the output, the comparator has to evaluate a simple bit comparison
- If the oracle is expressed in terms of a set of valid values, the comparator has to evaluate if the obtained result belongs to this set
  - For example, the expected behavior can be expressed as «a positive number», then the comparator has to evaluate a «greater than» condition
- If the oracle is expressed in terms that are not exactly represented by the computer, then the comparator has to perform an approximate evaluation
  - If the expected behavior is the number PI, then the comparator has to compare the difference between the obtained result and an approximate representation of PI
  - If the expected behavior is an image of Naples, the comparator has to compare the output image with a base of Naples images in order to argue if the image is sufficiently similar to a Naples image

Eseguire un caso di test

Come si esegue un caso di test ?

- **A mano**
  - Eseguiamo il programma, scriviamo gli input e guardiamo l'output
- **Automaticamente**
  - Scriviamo un programma che esegue il nostro programma e valuta se l'esecuzione è andata bene oppure no

Test da eseguire a mano

- **Eseguire test a mano è molto faticoso**
  - E' un lavoro ripetitivo
  - Bisogna segnare a parte tutti i test che sono stati fatti e i loro risultati
- **E' facile commettere errori**
  - Bisogna osservare con attenzione il risultato del test
- **E' facile dimenticare di eseguire dei test**

Test automatici

- **Il modo migliore per eseguire automaticamente un test è scrivere un programma che lo esegue**

- **Due regole fondamentali:**
  – I programmi di prova devono essere sufficientemente semplici da non contenere errori
    - Altrimenti chi testa i test?
  – Non bisogna mai manomettere il programma che stiamo provando

JUnit

Unit testing development

- **Erich Gamma and Kent Beck (also known as the authors of design patterns and eXtreme Programming) states:**
  - *Unit-level testing of a methods and classes should be designed and performed by the class developer at the same time as the class development itself*
  -

- **Advantages:**
  - The developer knows exactly the responsibilities of the class he has developed and the results he expects from it
  - The developer knows exactly how to access the class, for example:
  - Preconditions & postconditions

- **Drawbacks:**
  - The developer tends to defend his work ... he will find fewer mistakes than a tester can do!
  - In other words, the developer cannot validate his own work, because he developed on the basis of his own perception of specifications!

A first approach to testing automation: the main method

- **To write a test method ("main") in each class that contains code that can test its behaviors**
- **Problems**
  - The code will also be distributed in the final product, weighing it down
  - How is it possible to structure test cases? How can they be performed overall?
    - *If we have to carry out many test cases, we will have to make many main methods and to compile the project including, from time to time, only one of these main*

Example

```
package calcolatrice;

public class Calcolatrice {

    public int somma(int a, int b) {
     return a+b;
    }
}
```

```
package calcolatrice;
public class Main1 {
   public static void main(String[] args) {
    Calcolatrice c=new Calcolatrice();
    int somma=c.somma(3, 39);
    int atteso=42;
    if (somma==atteso)
        System.out.println("OK");
    else
        System.out.println("Failure");
   }
}

…
```

Limits to overcome from the *main* based approach

- **We are looking for a systematic approach with the aims:**
  - To separate the test code from that the class source code
    - *This problem is partially solved by the main approach*
  - To support the structuring of test cases in test suites
  - To allow the overall (or partial) execution of an entire test suite
  - To separate test outputs from the output of the method under test
  - …

X-Unit family

- **A solution to the previous problems is given by the frameworks of the X-Unit family, including:**
  - JUnit (Java)
    - *He is the progenitor; was originally developed by Erich Gamma and Kent Beck*
  - CppUnit Method (C++)
  - csUnit (C#)
  - NUnit (.NET framework)
  - HttpUnit (Web Application)
  - …

JUnit

JUnit is a framework (in practice it consists of .jar archive containing a collection of classes) that allows the writing of tests in a repeatable way

JUnit includes a runner classes that can be executed as a main method driving test case execution

- **Plug-ins that support the process of writing and running JUnit tests on Java classes are provided by all the most popular IDEs**

JUnit test Components

- **Test Classes**
- **Test methods**
  - **Test classes may contain one or more test methods**

JUnit test Components (version 3)

- **A test class also contains:**
  - A setup() method that is run before each test is run
    - *It is useful for performing preliminary steps necessary to meet the preconditions common to more than one test case*
  - A teardown() method that is run after each test case
    - *It is useful for performing the final steps required to restore the class (and in order to meet the preconditions of the other test cases)*
- **It is essential to always put the application and its resources back in the starting state at the end of each test since Junit never guarantees in what order the tests will be performed**
-

How JUnit works (in a nutshell)

- **JUnit includes is at least one public static method (in the junitrunner class) that can be run from the command line**
- **When JUnit starts, this method**
  1. Query the program looking for test classes and methods in them
     - *The reflection libraries present in the Java language make possible these queries*
  2. Cyclically performs:
     1. *The setup method*
     2. *One of the test methods*
     3. *The teardown method*

  During each of these runs it produces a separate output with the result of the test (the results of the assertions)

Structure of a test method

- **Precondition initialization**
  - Limited to the specific preconditions of the individual test case, whereas the others should be in the setup method
- **Insertion of input values**
  - It can be obtained by calling set methods or by assigning values to public attributes
- **Test code**
  - The method to be tested has to be run by passing parameters related to that test case
  - and collecting its outputs
- **Assessment of assertions**
  - Checking Boolean expressions (assertions) that must be true if the test is successful, that is, if the output data and/or postconditions found are different from the expected ones

Short Tutorial

- **Let's create a new Eclipse project, with a package (calculator)**
- **Within this package we generate a calcolatrice class:**

```
package calcolatrice;

public class calcolatrice {
public calcolatrice(){};

public int somma (int a, int b)
{return a+b;}

}
```

Assertions

Assert

- statement that may be true or false

Expected results correspond to explicit assertions, not with printouts that in any case require expensive visual inspections of the results

Assertions can be:

- true: the test do not detected failures
- false: the test detected a failure since the tested code does not behave as expected

- **Assertions are used to verify both oracles and postconditions**
- **Assertions are used also to verify preconditions: if the precondition fails, the test is not really performed (in this case the test fails, not the program under test)**

Assertions

- **If an assertion is not true the test case fails**
- **`assertNull()`: states that his argument is null**
- **`assertEquals()`: states that its second argument equals() the first one, that is, to the expected value**
  - Many other variations
    - ***`assertNotNull()`***
    - ***`assertTrue()`***
    - ***`assertFalse()`***
    - ***`assertSame()`***
    - *…*
    - ***https://junit.org/junit4/javadoc/4.13/org/junit/Assert.html***

`assertEquals()`

`assertEquals(Object expected, Object actual)`

- **Succeeds if `expected.equals(actual)` and returns `true`**
- **`expected` is the expected result**
- **`actual` is the observed result**

`assertEquals(String message, Object expected, Object actual)`

- **In this variant you specify a message that the runner prints in case of failure of the assertion: it could be very useful to immediately locate the assertion that causes the failure of a test-case and is a possible diagnostic message**

Other assertions

static void assertEquals(boolean expected, boolean actual)
- Asserts that two booleans are equal.

static void assertEquals(int expected, int actual)
- Asserts that two ints are equal.

static void assertEquals(java.lang.String expected, java.lang.String actual)
- Asserts that two Strings are equal.

static void assertFalse(boolean condition)
- Asserts that a condition is false.

static void assertTrue(boolean condition)
- Asserts that a condition is true.

static void assertNull(java.lang.Object object)
- Asserts that an object is null.

static void fail()
- Fails a test with no message.

static void fail(java.lang.String message)
- Fails a test with the given message.

…

Test case writing

- We write a testSomma method that represents a test case for the somma method;
  - since the method belongs to a test class in the same package as the class to be tested, the test method can instantiate objects in the class and access its methods
  -

```
public void testSomma() {

        calcolatrice c=new calcolatrice();
        int a=5,b=7;
        int s=c.somma(a,b);
        assertEquals("Somma non corretta!",12,s);
}
```

The assertEquals method checks whether s (value obtained from the execution of the sum method) is equal to 12 (expected value); otherwise a failure happens and the message error will be outputted

A more general example

The assertion in the setup method corresponds to the precondition: if it is not verified the test is not performed

The assertion in the teardown corresponds to the postcondition: if it is not verified, the test has detected a failure

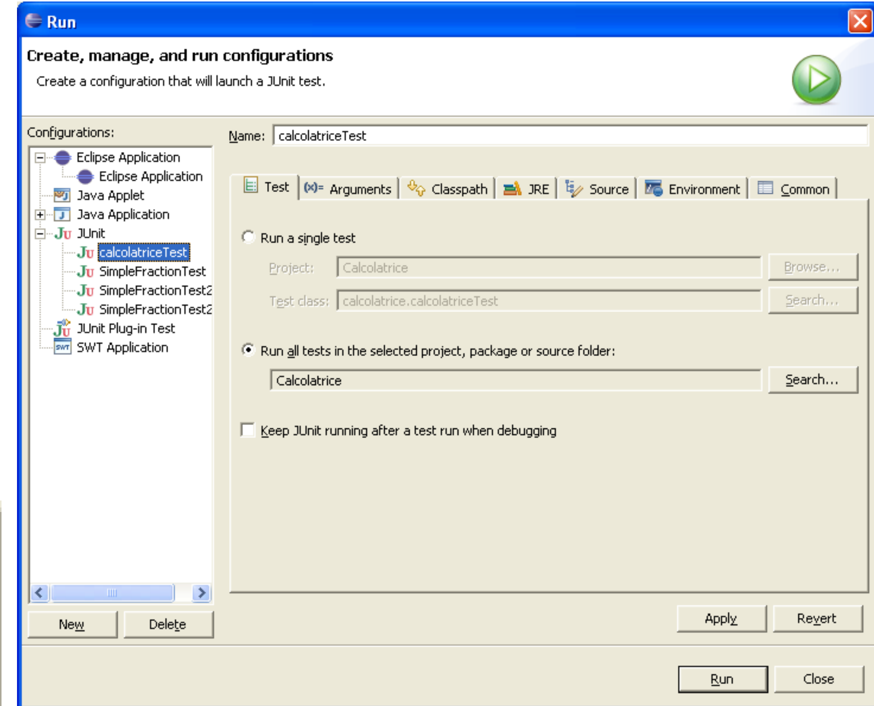
The assertion in the Test corresponds to the oracle: if it is not verified, the test has detected a failure

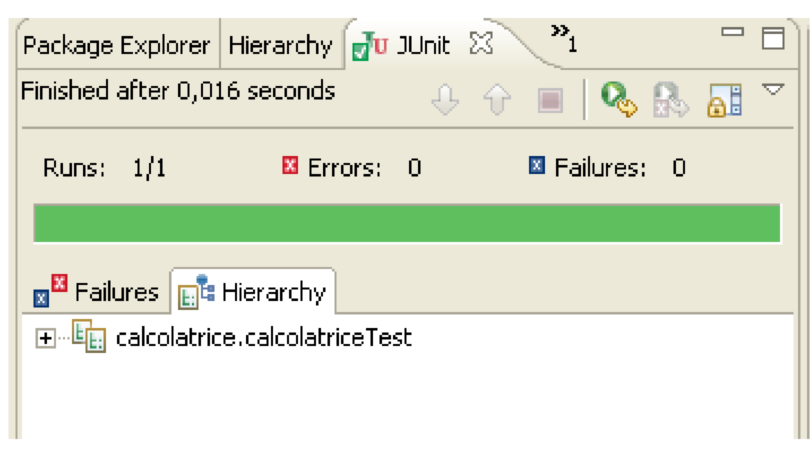
Note that the test is performed only if the precondition is verified, while the tearDown is performed in any case.

```
public class calcolatriceTest {
private Calcolatrice c;

public void setUp() throws Exception {
        c=new Calcolatrice();
        assertNotNull(c);
}

public void tearDown() throws Exception {
        c=null;
        assertNull(c);
}

public void testSomma() {
assertEquals("Somma Sbagliata",12,c.somma(5,7));
}
}
```

Test case execution

…To run your tests, just follow the same procedure used to run applications …

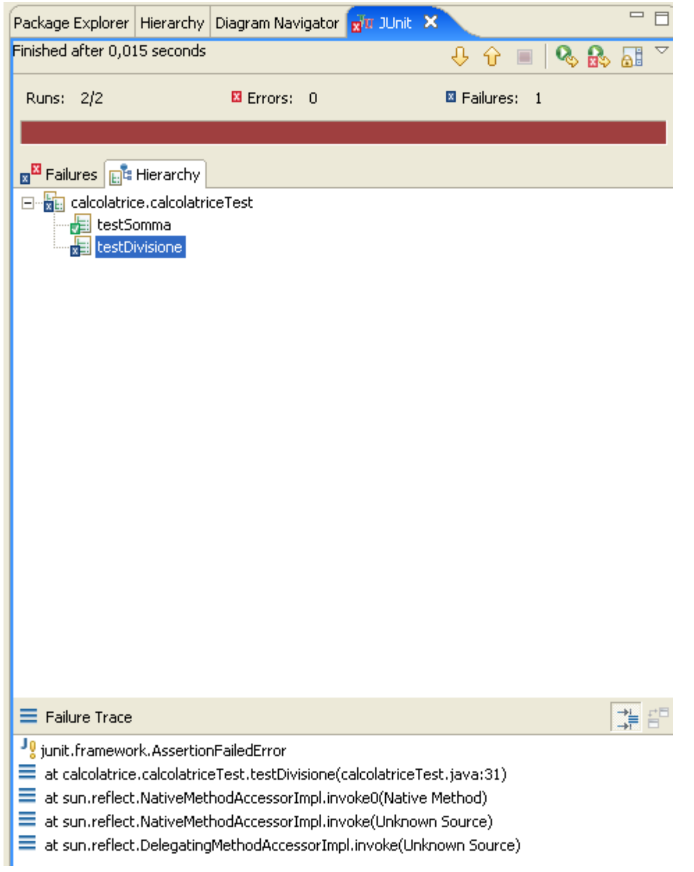




Failed test cases

```
public double divisione (int a, int b)
{return (a+1)/b;}
```

```
public void testDivisione(){

calcolatrice c=new calcolatrice();
int a=15,b=2;
double s=c.divisione(a,b);
assertTrue(s==7.5);
}
```



There is an error in the division method

The assertion fails: we have to correct the divisione method and restart this test

Your application is a special snowflake

Expert

Excuses for Not Writing Unit Tests

O RLY?

@ThePracticalDev

Parameterized test

- **Parameterized tests make it possible to run a test multiple times with different arguments.**
- **They are declared just like regular @Test methods but use the @ParameterizedTest annotation instead.**
- **In addition, you must declare at least one source that will provide the arguments for each invocation and then consume the arguments in the test method.**

Parameterized test example

```
@ParameterizedTest
@CsvSource({"-5,12,7", "0,-4,-4","35,0,35"})
void testSomma(int a, int b, int s) {
    //in order to show the executed tests
    System.out.println("Verifichiamo se la somma di
"+a+ " e "+b+" fa davvero "+s);
    assertEquals("Somma errata",s,c.somma(a,b));
}
```

```
<terminated> calcolatriceTest [JUnit] C:\Program Files\Java\jdk-14.0.2\bin\javaw.exe  (25 set 2022, 15:11:54 – 15:11:55) [pid: 20888]
Verifichiamo se la somma di -5 e 12 fa davvero 7
Verifichiamo se la somma di 0 e -4 fa davvero -4
Verifichiamo se la somma di 35 e 0 fa davvero 35
```

Observation

- **Parameterized tests are the fundamental feature to enable data-driven tests**
  - In data-driven approaches there is a clear separation between test case design and implementation the test values selection strategies
  - In particular, test data can be included in separate files (e.g. csv files)
    - *Many examples of these feature will be proposed in the following*

Come misuriamo se un test è utile?

Effectiveness

- **Un test serve a trovare i difetti**
- **L’Efficacia di una test suite è data dal numero di difetti che trova diviso per il numero di difetti totali**

- **Effectiveness = #Found Fault / #Existing Faults**

Ma sappiamo misurare l'efficacia?

Numero di difetti trovati

- **Numero di difetti trovati**
  - Un test non trova i difetti ma soltanto i fallimenti (failure)
  - Per sapere quale difetto corrisponde a quale fallimento dobbiamo fare debugging
  - Potremmo trovare tantissimi fallimenti ma tutti corrispondenti allo stesso difetto
- **Quindi, alla fine di un test non possiamo misurare il numero di difetti ma solo il numero di fallimenti**

Numero di difetti totali

- **Possiamo conoscere in anticipo quanti difetti ci sono in un programa?**
  - Se lo sapessimo, non staremmo facendo testing!
  - Nemmeno possiamo sapere quanti fallimenti ci sono

Ci sono dei casi in cui misurare i difetti trovati è possibile o utile?

Efficacia relativa

Efficacia relativa

- **Il numero di difetti trovati può servire per capire se una test suite è migliore di un'altra**

- **Efficacia relativa = Efficacia (test 1) / Efficacia (test 2) =**
- **= (#difetti trovati da 1 / ~~#difetti totali~~) / (#difetti trovati da 2 / ~~#difetti totali~~) =**

  = #difetti trovati da 1 / #difetti trovati da 2

Efficacia relativa ai fallimenti

- **Alla fine del test possiamo misurare almeno l'efficacia relativa in termini di fallimenti trovati**

- **Efficacia relativa ai fallimenti =**
- **= #fallimenti trovati da 1 / #fallimenti trovati da 2**

Come possiamo stimare l’efficacia?

Copertura del codice

- **Una misura alternativa che ci può aiutare ad avere una stima dell'efficacia dei test è la copertura del codice**
- **Principio: se i nostri test non eseguono un'istruzione sicuramente non possono accorgersi del fatto che quell'istruzione sia sbagliata**
- **Conseguenza: più istruzioni eseguiamo meglio è**

Misura della copertura del codice

- **Cosa sono le istruzioni nel codice?**
- **La più semplice approssimazione è quella di contare le linee di codice (LOC)**
  - Di solito in un programma c'è una istruzione in ogni riga
- **LOC Coverage = #linee di codice eseguite da almeno un test / #linee di codice totali**

Problemi

- **Tutte le linee di codice sono eseguibili?**
  – Alcune linee contengono commenti, dichiarazioni di variabili oppure sono semplicemente bianche
- **Come facciamo?**
- **Cambiamo la misura**
- **LOC Coverage = #linee di codice eseguibili eseguite da almeno un test / #linee di codice eseguibili**

Problemi

- **In una riga di codice potrebbe esserci più di una istruzione?**
  - Sicuramente!
- **Per avere una misura più precisa facciamo un ragionamento:**
  - Il nostro programma passerà per un compilatore che lo trasformerà in un programma eseguibile
    - Nel caso di java si tratta di un bytecode, in altri linguaggi potrebbe essere un programma in linguaggio macchina
  - Nel linguaggio eseguibile sono rimaste solo le istruzioni eseguibili

Copertura del codice eseguibile

- **Se misuriamo la copertura sul codice eseguibile avremo una misura più semplice**
- **Coverage Codice Eseguibile = #linee di codice eseguibile eseguite da almeno un test / #linee di codice eseguibile**
- **In Eclipse questa copertura è chiamata Instruction Coverage**

Problemi

- **Come facciamo a sapere quali sono le linee di codice eseguibile?**
  - Basta vedere il linguaggio macchina oppure il bytecode
  - Ma riusciremo a capire un codice che è stato scritto dal compilatore?
    - Con maggiore difficoltà

Soluzione

- **Ci sono strumenti di misura che fanno una *proiezione* del codice eseguibile che è stato eseguito sul codice sorgente**

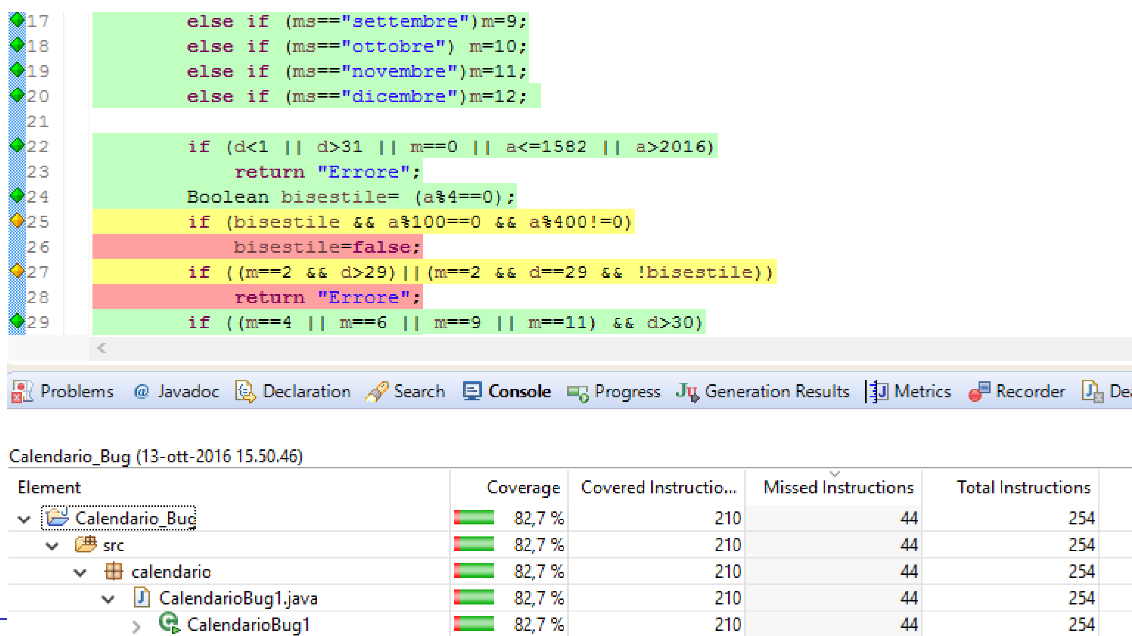
- **In questo caso una linea di codice sorgente può essere coperta, non coperta o anche *parzialmente coperta* :**
    - Copertura parziale Se viene tradotta con più di una linea di codice eseguibile delle quali solo una parte sono state eseguite

Emma / Jacoco behaviour

- **At bytecode level, the coverage of each block is boolean (covered / not covered)**
- **The projection of bytecode coverage on the source code generates fractionary values of coverage for each line of code:**
  - 1: if all the bytecode blocks generated by the line of code have been covered
  - Fractionary, between 0 and 1, the only some of the blocks have been covered
  - 0: if no blocks have been covered
  - Of course, no coverage is measured for all the lines that do not generate bytecode blocks (e.g. comments, blank lines, parentheses, etc.)

Colouring source code

- **Another output is a highlighted version of the source code**
  - Green for lines with coverage = 1
  - Yellow for lines with partial coverage
  - Red for lines not covered
  - White or Gray for lines without blocks to be covered



Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
Calendario_Bug	82,7 %	210	44	254
src	82,7 %	210	44	254
calendario	82,7 %	210	44	254
CalendarioBug1.java	82,7 %	210	44	254
CalendarioBug1	82,7 %	210	44	254

Copertura delle condizioni

- **Tra le tipologie di istruzioni più difficili da coprire ci sono le condizioni logiche**
  - Ad esempio (x>0) && ((y<0) || (x>15))
- **Un espressione come questa viene tradotta in linguaggio macchina e in bytecode con una lunga sequenza di condizioni logiche semplici**
  - Ad esempio x>0, y<0, x>15

Example

- if (x>0 && y>0) …
  - TS1={(x=2, y=-1), (x=-1, y=5)} covers all the conditions but not all the decisions
  - TS2={(x=2, y=1), (x=-1, y=-55)} covers all the conditions and all the decisions but not all the condition combinations
  - TS3={(x=2, y=1), (x=2, y=-1), (x=-2, y=1)(x=-1, y=-55)} covers all the condition combinations (and that implies that it covers all the decisions and all the conditions)

Copertura delle condizioni

- **Un difetto si può nascondere in una qualsiasi delle condizioni**
  - Ad esempio potremmo aver sbagliato una variabile o un valore
- **oppure nella loro combinazione**
  - Ad esempio potremmo aver sbagliato un operatore logico oppure una parentesi
- **La copertura delle istruzioni eseguibile (instruction coverage) tiene conto della maggior parte delle possibili condizioni**

Ci sono dei casi in cui misurare i difetti trovati è possibile o utile?

Mutanti

Mutanti

- **L'unico caso nel quale veramente possiamo misurare quanti difetti sono stato trovati è quello nel quale i difetti ce li abbiamo inseriti direttamente noi!**
  - Oppure li ha inseriti un apposito programma al posto nostro

- **Perché inserire difetti nel codice?**
  - Per sapere se i nostri test sarebbero in grado di trovare veri difetti se ci fossero

To kill (retire) a mutant

- **A mutant is killed (or retired) when:**
  - there is at least one test case whose result differs when run on the original program and on the mutant program
  - The result is generally given by assertion value, but it could also be a regular termination vs crash
- **A mutant survives a test suite if:**
  - For all test cases in the test suite, no distinction is made on the result of any test whether it is run on the original program or on the mutant program

Examples of mutation operators

- **Depending on the code element on which they apply:**
  - Changes on variables and types
  - Changes on arrays (e.g. index) and lists
  - Changes on operators (assignment, arithmetic, logical)
  - Changes to the prototype of a function/method/service
  - Changes of modifiers (eg static, transient, synchronized, final, ...)
  - Changes in inheritance or polymorphism (e.g. casting, super, override, ...)
  - https://pitest.org/quickstart/mutators/

Come ritirare un mutante

- **Un mutante viene ritirato da un test quando l'esito del test è diverso se eseguito sul programma originale e su quello mutante**

- **Il caso più semplice: se eseguo il test sul programma normale non causa un fallimento, mentre se lo eseguo sul programma mutante causa fallimento**
  - Per trovare i mutanti dobbiamo essere anche bravi a scrivere le asserzioni

Mutanti equivalenti

- **Alcuni mutanti sono troppo semplici da trovare**
  - Se il programma mutante non compila allora non c'è bisogno di test per scoprirli → INUTILI !
  - Se il programma mutante viene scoperto da ogni test allora non ci aiuta a capire quali sono i test migliori → INUTILI
- **Alcuni mutanti sono impossibili da trovare**
  - Ad esempio modificano una parte di programma nella quale è impossibile andare → INUTILI
    - Si chiamano Mutanti Equivalenti

Example

Program

```
int add(int x, int y) {
  return x + y;
}
```

Test suite

```
t1: assertEquals(0, add(0, 0))

t2: assertEquals(1, add(1, 0))

t3: assertEquals(2, add(2, 0))
```

Mutants

```
int add(int x, int y) {
  return x - y;
}
```
m_1

```
int add(int x, int y) {
  return x * y;
}
```
m_2

```
int add(int x, int y) {
  return x++ + y;
}
```
m_3

Example

Program

```
int add(int x, int y) {
  return x + y;
}
```

Test suite

```
t1: assertEquals(0, add(0, 0))

t2: assertEquals(1, add(1, 0))

t3: assertEquals(2, add(2, 0))
```

Mutants

```
int add(int x, int y) {
  return x - y;
}
```
m_1

```
int add(int x, int y) {
  return x * y;
}
```
m_2

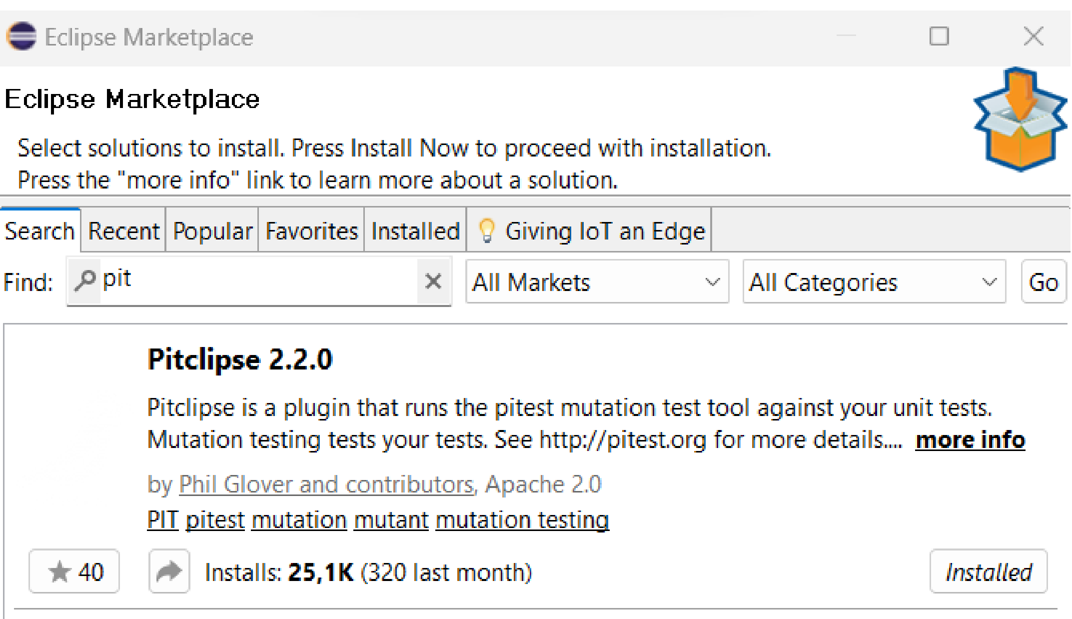
```
int add(int x, int y) {
  return x++ + y;
}
```
m_3

- **T2 kills the mutant M2 because the result of the test is positive on the original program, failure on M2**
- **T3 kills the mutant M2 because the result of the test is positive on the original program, failure on M2**

- **In total, only one of the three mutants was "killed", while the other two survived.**
- **Our test suite is unable to recognize potential faults of m1 and m3**
- **The m3 mutant is called an equivalent mutant because it can be shown that there is no test that can detect it.**
  - In this case x (local variable) is used in addition, then incremented but never reused
- **Our test suite was able to kill half of the mutants that could be killed.**

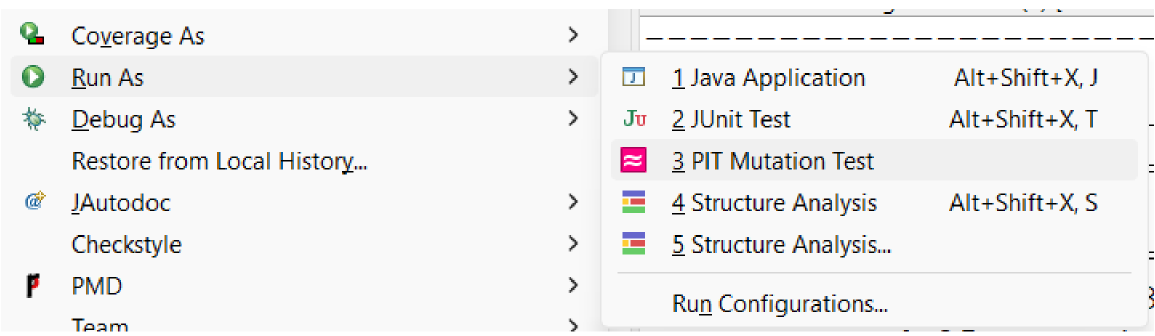
Mutation based testing with pitclipse

- **Pitclipse is an eclipse extension**
- **It allows mutation-based testing in the context of the Eclipse IDE**



Using Pitclipse

- **PIT can be used in Eclipse by the Run as contextual menu**
- **It can be applied on a JUnit test suite without failing test cases**



Pitclipse Output

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	90%	9/10	88%	22/25	88%	22/25

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
default	1	90%	9/10	88%	22/25	88%	22/25

Report generated by PIT 1.6.8

```
================================================================================
- Statistics
================================================================================
>> Line Coverage: 11/13 (85%)
>> Generated 25 mutations Killed 22 (88%)
>> Mutations with no coverage 0. Test strength 88%
>> Ran 287 tests (11.48 tests per mutation)
```

Pitclipse Output

Pit Test Coverage Report

Package Summary

default

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	90%	9/10	88%	22/25	88%	22/25

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
TriangleType.java	90%	9/10	88%	22/25	88%	22/25

Report generated by PIT 1.6.8

Pitclipse Output

Active mutators

- BOOLEAN_FALSE_RETURN
- BOOLEAN_TRUE_RETURN
- CONDITIONALS_BOUNDARY_MUTATOR
- EMPTY_RETURN_VALUES
- INCREMENTS_MUTATOR
- INVERT_NEGS_MUTATOR
- MATH_MUTATOR
- NEGATE_CONDITIONALS_MUTATOR
- NULL_RETURN_VALUES
- PRIMITIVE_RETURN_VALS_MUTATOR
- VOID_METHOD_CALL_MUTATOR

Tests examined

- TriangleTypeTest.[engine:junit-jupiter]/[class:TriangleTypeTest]/[test-template:test3WiseACOC(int, int, int, java.lang.String)]/[test-template-invocation:#15] (8 ms)
- TriangleTypeTest.[engine:junit-jupiter]/[class:TriangleTypeTest]/[test-template:testBCCExtended(int, int, int, java.lang.String)]/[test-template-invocation:#6] (6 ms)
- TriangleTypeTest.[engine:junit-jupiter]/[class:TriangleTypeTest]/[test-template:test3WiseACOC(int, int, int, java.lang.String)]/[test-template-invocation:#19] (8 ms)
- TriangleTypeTest.[engine:junit-jupiter]/[class:TriangleTypeTest]/[test-template:testBCCExtended(int, int, int, java.lang.String)]/[test-template-invocation:#2] (9 ms)

Pitclipse Output

Mutations

Line	Mutations
16	1. changed conditional boundary → SURVIVED 2. changed conditional boundary → SURVIVED 3. changed conditional boundary → SURVIVED 4. negated conditional → KILLED 5. negated conditional → KILLED 6. negated conditional → KILLED
17	1. replaced return value with null for TriangleType::triangle → KILLED
20	1. changed conditional boundary → KILLED 2. changed conditional boundary → KILLED 3. changed conditional boundary → KILLED 4. Replaced integer addition with subtraction → KILLED 5. Replaced integer addition with subtraction → KILLED 6. Replaced integer addition with subtraction → KILLED 7. negated conditional → KILLED 8. negated conditional → KILLED 9. negated conditional → KILLED
21	1. replaced return value with null for TriangleType::triangle → KILLED
24	1. negated conditional → KILLED 2. negated conditional → KILLED
25	1. replaced return value with null for TriangleType::triangle → KILLED
28	1. negated conditional → KILLED 2. negated conditional → KILLED 3. negated conditional → KILLED
29	1. replaced return value with null for TriangleType::triangle → KILLED
31	1. replaced return value with null for TriangleType::triangle → KILLED

Pitclipse Mutation Operators

- https://pitest.org/quickstart/mutators/