IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Introduction to software testing

J.-C. Royer & J.-C. Bach

DCL – Tests – 2025-2026

Progress

Work under Creative Commons BY-SA license

Important notes

- If you do not understand something, please ask your questions. *We cannot answer the questions you do not ask. . .*
- If you disagree with us, please say it (politely)
- People don't learn computer science by only reading few academic slides: practicing is fundamental

Introduction to software testing

- Comparison between program execution and what one thinks to be the right behavior or result
- Generalization to other documents (e.g. specifications)
- Test = a reflex... but its rationalization demands organization
- Rationalizing tests: technical and not always trivial
- Progress in automation (generation, test and coverage)
- Designing tests is a real task in the development process (and therefore *planned*), not an optional one (*if I have the time, I'll write tests*)

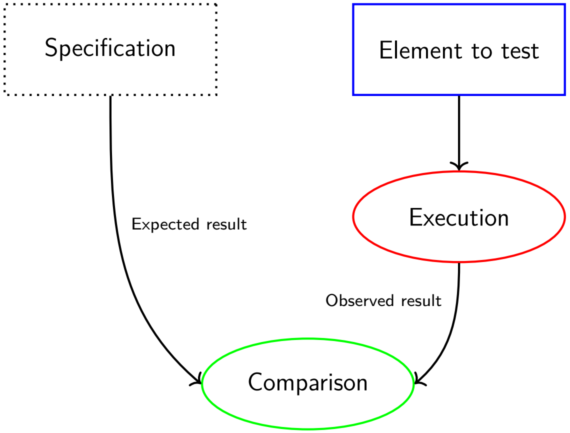
Why testing?

Why testing?

- Detect differences in relation to specifications
- Detect errors
- Increase robustness and confidence
- Determine reliability level
- Evaluate performances
- Evaluate behavior in real-life situations (e.g. ergonomy)
- Target quality criteria
  - functionality, security, integrity, usability, consistency, maintainability, efficiency, robustness, safety

Progress

Test principle: dynamic analysis!



The test characteristics

- What does one want to test? (code, specification, ...)
- When does one want to test it?
  - unit test (component, function, ...)
  - integration test (to the assembly, integration order)
  - system test (versions, at the customer's site, ...)
- Which specification (requirement, comment, UML description, formal property, etc.)?
- Test set and coverage
- Black or white box: specification *vs* specification and implementation (or functional *vs* structural)

Useful vocabulary

- **Nominal test**: test case = valid input data
- **Functional test**: expected behavior
- **Robustness test**: test case = invalid input data
  - Note: nominal tests are done before robustness ones
- **Performance test**: to evaluate various behaviors in relation with the time (response time, stability, load ramp-up, ...)
- **Non-regression test**: to check that fixes do not introduce new problems
- **Unit test**: to test basic code snippets (e.g. functions), simple
- **Integration test**: to test the composition of the basic code snippets
- **Validation test**: to ensure software meets client requirements, before delivery, usually follow scenarios

Testing techniques

- Domain analysis of operations arguments: partitioning, boundary values, ...
- Data flow analysis
- Control flow analysis and graph coverage
- Test generation: random, statistics, combinatoric, algebraic
- Mutation testing
- API-driven
- GUI test
- ...
- Black box (= functional) *vs* White box (= structural)

Partitioning analysis

- Analysis of input and output data
- Define equivalence classes covering the expected behavior of the unit
- Determine a partition of the associated value space
- Work on input or output
- For each class, choose a representative
- Write a test case for each representative

+ intuitive

- modeling, nominal behavior, high combinatorics

Example of equivalence partitioning

- Case: absolute value for floating numbers
  - partitioning the input: negative, zero, positive
  - partitioning the output: null or positive
  - cartesian product of partitions is a partition
  - sort of integer list: 0, 1, 2, 2+ is a possible partition

Boundary value analysis

- Considering input domain of values
- Analysis of partitions by only examining boundary values
- Values before/equal/after the limit

+ Intuitive, simple, useful

- Hard to justify the coverage

- Partition and boundaries can be combined

Finite-state automaton coverage

- Finite-state automaton = finite-state machine (FSM)
- A FSM specifies the behavior
- One tries to cover all arcs, all states or all arcs and states
- Interesting for interactive and communicating systems
- Can be automated, good coverage depending on the specification
- Require a FSM and usually high technical complexity

Model-based generation

- Different types of specifications: FSM, pre/post conditions, algebraic, functional, UML diagrams, etc.
- Simulate the functioning and produce tests from it
- Various techniques: model verification, constraint programming, generating function, . . .
- Test are executed on the tested unit

+ Master precisely the tests coverage
+ Automation

- Models can be really complex with few executable information

- Advanced techniques

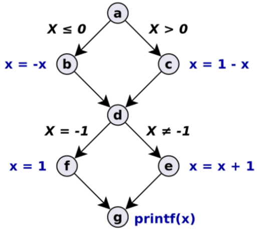
- Tools: QuickCheck (and all its variations and adaptations), . . .

Control flow graph testing

- Control flow graph = description of steps and transitions of the tested unit
- All arcs, all nodes, all paths
- Finding unreachable states or paths

Exemple

```
void foo( int x ) {
  if ( x <= 0 )  x = -x;
  else x = 1 - x;
  if ( x == -1 )  x = 1;
  else x = x + 1;
  printf(" x = %d ", x );
}
```



Data flow analysis

- Analysis of different operations (initializations, affectations) on variables
- df-chains: list definitions and references of a variable
- Can be static or dynamic
- Undefined value when first used: `print(str(X))`
- Redundant definitions: `X:=4; X:=4`
- Useless definitions: `X:=3; X:=4; END`
- Harder in a real program with loops and conditions...

+ Possible automatic analysis

Sets and coverage

- Random test sets (e.g. JCrasher)
- Automatic computing or test generation (TestNG)
- Complete coverage: usually impossible
- Hight cost, real problem
- Model-based generation allows one to master coverage
- Test strategy depending on the system architecture

Progress

Test in practice with Java

From the easiest to the more advanced

- Auto-testing class: add a `main` in the class
- Writing specific `Main` classes to separate tests from the program
- Using `JUnit` (test *framework*)
  - Test cases
  - Test suite
  - Test runner
- Using a coverage tool (e.g. `Cobertura` or `eclemma`)
- Using a test generator (Randoop, EvoSuite, ...)
- Using a consolidation or continuous integration tool

JUnit

- Test help integrated within Eclipse
- Notable improvements and changes
- Based on annotations
- Goal = exploiting Java 1.5 (annotations and generics)
- Currently: JUnit5 ($\geqslant$ Java 8)

Principles and concepts

- Use of annotations to qualify, to select and to execute the test
- Use of assertions pour execute the real work
- TestCase: test case implemented in a test method
  - annotated method with `@Test`, `@RepeatedTest`, `@ParameterizedTest`, `@TestFactory`, `@TestTemplate`
- TestClass: class containing at least one test method
- Tests are grouped in dedicated classes (usual organization in Java)
- TestSuite: list of test classes
- TestRunner: to launch the execution of a test suite

JUnit 5

- What changes compared to JUnit 4
- JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage
- JUnit Platform: basic API to launch tests (can be extended)
- JUnit Jupiter: JUnit annotations for tests writing and `TestEngine` implementation
- JUnit Vintage: compatibility layer with JUnit 3 and 4
- Possible to generate tests *at runtime*!
- Tutorial: `https://howtodoinjava.com/junit-5-tutorial/`

Annotations with JUnit 5

@BeforeEach	execution before each test of the class
@AfterEach	after each test (@After)
@BeforeAll	method before all tests (@BeforeClass)
@AfterAll	method after all the tests (@AfterClass)
@Test	to define a test method
@DisplayName	to name a class or a test
@Disabled	to disable a test
@Nested	to write nested test classes
@Tag	used for documentation and research
@TestFactory	test factory for dynamic tests

Typical example (1/2)

```
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

// squelette de test
public class AppTest {

    @BeforeAll
    static void setup(){
        System.out.println("@BeforeAll executed");
    }

    @BeforeEach
    void setupThis(){
        System.out.println("@BeforeEach executed");
    }

    @Tag("DEV")
    @Test
    void testCalcOne()
    {
        System.out.println("======TEST ONE EXECUTED=======");
        Assertions.assertEquals( 4 , Calculator.add(2, 2));
    }
```

Typical example (2/2)

```
    @Tag("PROD")
    @Disabled
    @Test
    void testCalcTwo()
    {
        System.out.println("======TEST TWO EXECUTED=======");
        Assertions.assertEquals( 6 , Calculator.add(2, 4));
    }

    @AfterEach
    void tearThis(){
        System.out.println("@AfterEach executed");
    }

    @AfterAll
    static void tear(){
        System.out.println("@AfterAll executed");
    }
}
```

Creation of a TestCase

- Add Junit 5 library
- Usual class (preferably in a `tests` package)
- Use (preferably) the *wizard*
- Define initializations and freeing
  - for the class (`@BeforeAll`, `@AfterAll`)
    or
  - for each testcase (`@BeforeEach`, `@AfterEach`)
- Contain test methods `test*` (but naming is free)
- Contain `Assertion.assert*`
- Execution with `run as junittest` menu

Practical considerations

- Tests make you think to what to test
  1. Boundary cases: void, a single element, undefined, null, negative, etc.
  2. Behaviors: exception, return of a precise value, content of variables, ...
  3. Verifying invariants: loops, arrays, properties on data
  4. Verifying pre- and post-conditions of the operations
  5. Operations chaining (sequence diagram)

$\Rightarrow$ better knowledge of the code and of its behavior

- Bring tools and automation

Assertions

- Class `Assertions` with static methods, different profiles et negation
- Overriden for integer, float, string etc.
  - `fail("indeed, it does not work!")`
  - `assertEquals`: semantic equality
  - `assertArrayEquals`: semantic equality
  - `assertNull`: null object
  - `assertSame`: same object in memory (pointer equality)
  - `assertTrue`: condition is true
  - ...
- `https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html`

TestSuite

- Goal: chaining tests because they are usually plenty of tests...
- `@SelectPackages`: test packages selection
- `@SelectClasses`: test classes selection, `@SelectClasses(TestPlace.class, TestTransition.class)`
- `@IncludePackages` and `@ExcludePackages`
- `@IncludeClassNamePatterns` and `@ExcludeClassNamePatterns`
- `@IncludeTags` and `@ExcludeTags`

```
@RunWith(JUnitPlatform.class)
@SelectPackages("com.howtodoinjava.junit5.examples")
public class JUnit5TestSuiteExample {
  ...
}
```

TestSuite

```java
@RunWith(JUnitPlatform.class)
// selects some packages and run the test
// inside and under the subpackages
@SelectPackages({"com.howtodoinjava.junit5.examples",
                 "com.howtodoinjava.junit5.trucs"})

// includes only this subpackage
@IncludePackages("com.howtodoinjava.junit5.examples.packageC")

// all selected but excludes this
@ExcludePackages("com.howtodoinjava.junit5.examples.packageB")

public class JUnit5TestSuiteExample {
  ...
}
```

Parameterized tests

- A test depends often on values
- Some particular values often have a different effect

$\Longrightarrow$ doing a test with a parameter

- A method with parameters
  - use of `@ParameterizedTest` annotation
  - parameters valuess `@ValueSource`
  - a **SINGLE** source for all arguments
- `https://blog.codefx.org/libraries/junit-5-parameterized-tests/`

Example of parameterized tests

```
@ParameterizedTest

// Source of value to use
@ValueSource(ints = { 2, 3, 4, 5, 6 })

void testmult(int arg) {
    Assertions.assertEquals(Calcul.mult(arg), 2*arg);
}
```

Source method

- Argument of `@ValueSource`
- `ints()`, `strings()`, `doubles()`, `longs()` primitives for the corresponding to the primitive types
- There also exists `@EnumSource()`
- `@MethodSource` to define its own source
  - defines a method that produces a stream of arguments
  - `Arguments` is a JUnit interface to aggregate values
  - can return a sequence or an array
  - has to be `static` and can be in another class

Example of a source method

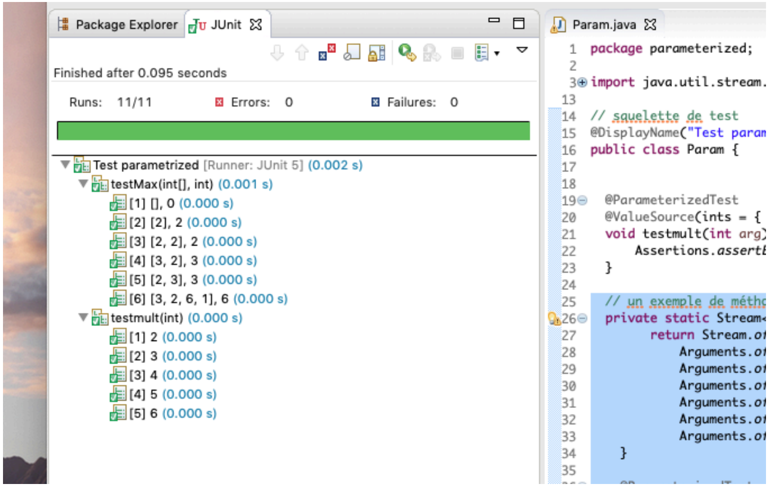
```
// The method args create a stream of pairs
// an int array and an int value
private static Stream<Arguments> args() {
 return Stream.of(
      Arguments.of(new int [] { }, 0),
      Arguments.of(new int [] { 2 }, 2),
      Arguments.of(new int [] { 2, 2 }, 2),
      Arguments.of(new int [] { 3, 2 }, 3),
      Arguments.of(new int [] { 2, 3 }, 3),
      Arguments.of(new int [] { 3, 2, 6, 1 }, 6)
   );
}
```

Example of use

```
@ParameterizedTest

@MethodSource("args")
// each test uses an array of int and an int value
public void testMax(int[] array, int val) {
  Assertions.assertEquals(Calcul.max(array), val);
}
```

JUnit view



```
Finished after 0.095 seconds

Runs:   11/11        Errors:   0         Failures:   0

▼ Test parametrized [Runner: JUnit 5] (0.002 s)
  ▼ testMax(int[], int) (0.001 s)
      [1] [], 0 (0.000 s)
      [2] [2], 2 (0.000 s)
      [3] [2, 2], 2 (0.000 s)
      [4] [3, 2], 3 (0.000 s)
      [5] [2, 3], 3 (0.000 s)
      [6] [3, 2, 6, 1], 6 (0.000 s)
  ▼ testmult(int) (0.000 s)
      [1] 2 (0.000 s)
      [2] 3 (0.000 s)
      [3] 4 (0.000 s)
      [4] 5 (0.000 s)
      [5] 6 (0.000 s)
```

```java
package parameterized;

import java.util.stream.

// squelette de test
@DisplayName("Test param
public class Param {


    @ParameterizedTest
    @ValueSource(ints = {
    void testmult(int arg)
        Assertions.assertE
    }

    // un exemple de métho
    private static Stream<
            return Stream.of
                Arguments.of
                Arguments.of
                Arguments.of
                Arguments.of
                Arguments.of
                Arguments.of
        }

```

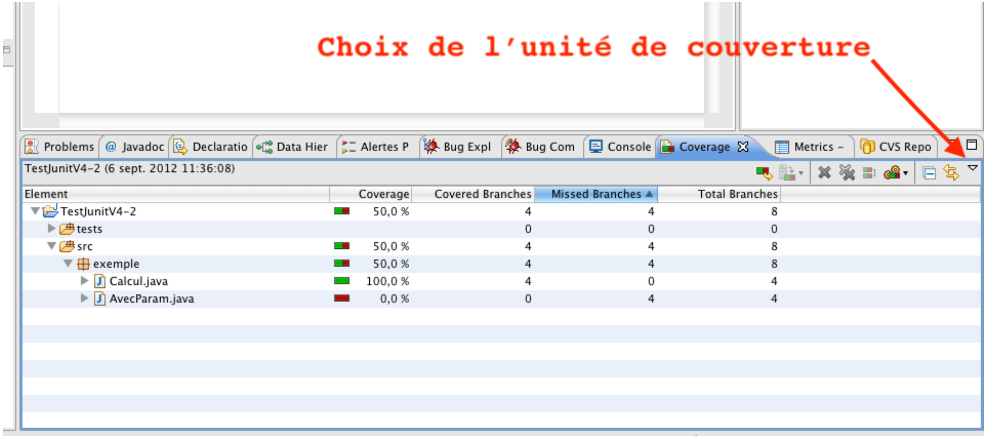
Let's write tests! (demo)

- Tools: Eclipse + JUnit (5) + Eclemma
- Simple example: boolean logic
  - `not: boolean -> boolean`
  - `and: boolean -> boolean -> boolean`
  - `imply: boolean -> boolean -> boolean`
- Scenario
  - create a Java project
  - write/generate `BooleanLogic`
  - generate and write `TestBooleanLogic`
  - run tests, play with JUnit and Eclemma

Coverage example with EclEmma

- `http://www.eclemma.org/index.html`
- Update site: `http://update.eclemma.org`
- Also in the market place
- Open the `Coverage` view and use the action in the contextual menu (or in the tool bar, on the left of the bug)
- The tool shows the coverage depending on different metrics (lines, instruction number, blocks, methods, types)
- Visible in the `Coverage` tab, in the project properties window

Coverage view



TestJunitV4-2 (6 sept. 2012 11:36:08)

Element		Coverage	Covered Branches	Missed Branches ▲	Total Branches
TestJunitV4-2	🟩🟥	50,0 %	4	4	8
tests			0	0	0
src	🟩🟥	50,0 %	4	4	8
exemple	🟩🟥	50,0 %	4	4	8
Calcul.java	🟩	100,0 %	4	0	4
AvecParam.java	🟥	0,0 %	0	4	4

Coverage example with EclEmma

- In the view, one can choose the type of the présentation and the metric
- Choosing the type is insufficient
- Refine the level during the tests, in order to reach the finest level
- The best level is the "branch" one
- The "instruction counter" level is in relation to the bytecode and is not a good coverage metrics
- The "line" level is the least precise level and depends on code formating

Recommendations for unit testing

- Isolate the class to test, has to be unitary (use stub if needed)
- If it is difficult, it means there is a problem of structure
- Test one precise behavior at once
- Write readable and maintainable test
- Identify test steps
- Test in parallel with the development
- Be careful with the chosen code coverage

Test pattern

- Comment the tests
- Use tools to generate test skeletons of a class
- Test pattern for each method of the class
  - instantiate the class to test
  - generate the necessary arguments
  - generate the expected result
  - apply the method and use an assertion
- Iterate the pattern for the different cases
- Do not write complex code within tests (who will test your test code?)

A dependency problem

- A recurring problem: testing a component independently of other components

⇒ Need to mimic a component: *stubs* and *mocks*

- *stubs*
  - are usually trivial snippets of code replacing the component
  - provide canned answers to calls made during the test
  - (usually) not respond at all to anything outside what's programmed in for the test.
- *mocks*
  - simulate the behavior of a component
  - objects pre-programmed with expectations which form a specification of the calls they are expected to receive.
- Further reading for the difference between *stubs* and *mocks*: `https://martinfowler.com/articles/mocksArentStubs.html`
- *mocks* are usually supported by dedicated tools (*e.g.* EasyMock)

Conclusion

- Introduction to testing, with some good practices
- Test $=$ an engineering discipline
- We don't expect you to be Q&A experts during DCL
- We expect you to be able to design and to write unit tests
- From now, we expect you to write tests when you write code (during IDL and MAPD, but also in any other TU)

Gentle reminder

⚠ Important notes

- If you do not understand something, please ask your questions. *We cannot answer the questions you do not ask...*
- If you disagree with us, please say it (politely)
- People don't learn computer science by only reading few academic slides: practicing is fundamental