

# Introduction to tests

DLR-IDL 2023-2024



#### You are free:

- to use, to copy, to distribute and to transmit this creation to the public;
- to adapt this work.

#### Under the following terms:

- Attribution (BY): you must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- ShareAlike (SA): if you modify, transform, alter, adapt or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

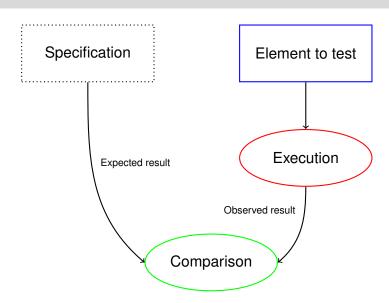
<sup>&</sup>lt;sup>1</sup>http://creativecommons.org/licenses/by-sa/2.0/.

- 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 (we follow Crocker's rules<sup>2</sup>)
- people don't learn computer science by only reading few academic slides: practicing is fundamental

<sup>&</sup>lt;sup>2</sup>http://sl4.org/crocker.html

- 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)

- 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

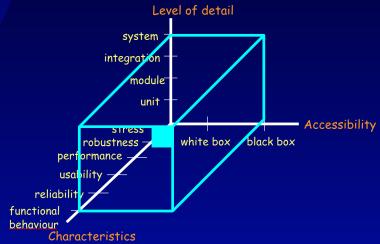


#### The test characteristics

- 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)



# Types of Testing



- 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 / 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

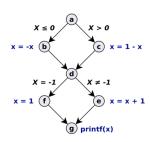
- Considering input domain of values
- Analysis of partitions by only examining boundary values
- Values before/equal/after the limit
- + Intuitive, simple, useful
- Hard to justify the coverage
- Partition and boundaries can be combined

## 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

- 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
- Tooling: QuickCheck (for Haskell, but it has been declined in several languages), . . .

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



#### 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

- 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

#### 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 20 / 47

- Test help integrated within Eclipse
- Notable improvements and changes
- Based on annotations
- Goal = exploiting Java 1.5 (annotations and generics)
- Currently: JUnit5 (≥ 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 22 / 47

- 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/

@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.apj.AfterAll:
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.apj.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));
```

```
@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

```
// Attention needs Executable
// asserts that executable throws an exception of the expected type
public static <T extends Throwable> T
     assertThrows(Class<T> expectedType. Executable executable)
// Executable is a functional interface that can be used to implement
// any generic block of code that potentially throws a Throwable.
// Needs a Lambda expression
@Test
@DisplayName("An example of exception testing")
void exceptionTesting() {
 Throwable exception = assertThrows (ConstructionFailure.class,
        () -> new Variable (""));
                // to check some information
    assertEquals("Variable: name is empty", exception.getMessage());
```

Tests if the exception is raised, then some informations of the exception are tested

If there is no exception or if an other than the expected one (inheritance is taken in account) is
raised, then there is an error

#### Also uses the notion of Lambda expression in Java

```
@Test
void timeoutExceeded() {
// The following assertion fails
// with an error message similar to:
// execution exceeded timeout of 10 ms by 91 ms
    assertTimeout(ofMillis(10), () -> {
        // Simulate task that takes more than 10 ms.
        Thread.sleep(100);
    });
}
```

#### 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)
- better knowledge of the code and of its behavior
  - Bring tooling and automation

Assertions 30 / 47

 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

- Goal: chaining tests because they are usually plenty of tests...
- @SelectPackages: test pacakages 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 32 / 47

```
@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 {
```

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

```
@ParameterizedTest
// Source of value to use
@ValueSource(ints = \{ 2, 3, 4, 5, 6 \})
void testmult(int arg) {
   Assertions.assertEquals(Calcul.mult(arg),
                            2*arg);
```

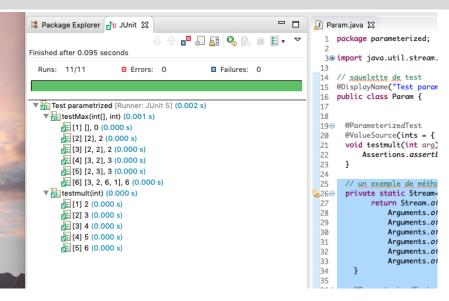
- Argument of @ValueSource
- ints(), strings(), doubles(), longs() primitives for the corresponding to the primitive types
- There also exists @EnumSource()
- @MethodSource to define est own source
- Define a method that produces a stream of arguments
- Arguments is a JUnit interface to aggregate values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

```
// 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)
```

```
@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 39 / 47



## 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

- 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



# 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

- 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 recurring problem: testing a component independently of other components
- → Need to mimic a component
  - 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: https://www.easymock.org)

#### 

- 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 (we follow Crocker's rules<sup>3</sup>)
- people don't learn computer science by only reading few academic slides: practicing is fundamental

<sup>3</sup>http://sl4.org/crocker.html