Outline

1. Introduction
2. What to test?
3. Types of tests
4. Automated testing
Verification and Validation:

- Validation ensures that the software fulfills the requirements.
- Verification ensures that the software meets the specification, three approaches:
  - Prove correctness by formal verification: costly, do not prevent from bugs in the specification.
  - Code inspection by peer reviews.
  - Testing.
Running the program on all possible inputs is impossible for complex problems:
  - exploration space might be insanely large (or worse infinite)

Test on a subset of inputs:
  - Partition inputs in significant classes maximizing the coverage of all the possible cases.
  - To do this choose particular inputs for your tests:
    - inputs that tests all the control branches of your code
    - boundary cases (detect overflow and off by one bugs)
    - duplicate, null or invalid inputs.
Example of partitionning (1/2)

specification:
int compare (int a, int b);

The function compare returns:
  0 if a is equal than b
  1 if a is strictly superior to b
-1 if a is strictly inferior to b

Q : What inputs would you test ?
Example of partitionning (2/2)

```java
int compare (int a, int b) {
    int c = a-b;
    if (c == 0) return 0;
    else if (c<0) return -1;
    else return 1;
}
```
Example of partitionning (2/2)

```java
int compare (int a, int b) {
    int c = a-b;
    if (c == 0) return 0;
    else if (c<0) return -1;
    else return 1;
}

System.out.println(compare(10,10)); -> 0
System.out.println(compare(10,5)); -> 1
System.out.println(compare(-10,-5)); -> -1
System.out.println(compare(-2147483648,1)); -> 1
```
Black box and White box testing

Black box testing
- Generate test cases from the specification only.
- Do not make the same assumptions than the programmer.
- Tests are independent of the implementation.

White box testing
- Generate test cases from the source code.
- Improves coverage: we know the different control paths in the code.
Unit tests

- A unit is the smallest testable part of an application.
- Test a single functionality in the code.
- Usually tests a single method.
- Unit tests allow to isolate the parts of the system and show they are correct.
- Most useful during the implementation phase.
Functional tests

- Functional tests verify the program as a whole.
- Centered in functionality which may be distributed among many classes and functions.
- Important during the integration phase.
Regressions tests

- Each time a bug is detected, a test that catches it must be written.
- If later on code is changed, the test ensures that if the bug appears again, it will be caught.
JUnit

- Allows automazing tests.
- Helps during regression testing.
- [http://www.junit.org/](http://www.junit.org/)
Example (1/2)

```java
import junit.framework.*;

public class TestCompare extends TestCase {
    CompareClass comp;
    protected void setUp() {
        comp = new CompareClass();
    }
    public void testPositive() {
        int compare = CompareClass.compare(10, 5);
        assertEquals(compare, 1);
    }
    public void testBoundaries() {
        int compare = CompareClass.compare(-2147483648, 1);
        assertEquals(compare, -1);
    }
}
```
Example (2/2)

```bash
$ javac -cp junit-4.5.jar:. TestCompare.java
$ java -cp junit-4.5.jar:. junit.textui.TestRunner TestCompare
..F
Time: 0.003
There was 1 failure:
1) testBoundaries(TestIt) junit.framework.AssertionFailedError: expected:<1> but was:<-1>
at TestCompare.testBoundaries(TestIt.java:11)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
FAILURES!!!
Tests run: 2, Failures: 1, Errors: 0
```
This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 Unported License.