Implementing Test Cases

TestNG - A testing framework

Test case implementation

- Popular unit-testing frameworks in Java
  - JUnit
  - TestNG

- TestNG was inspired on Junit
- It provides some distinctive functionalities
  - Gap reduced with JUnit 5
- It works for functional and higher levels of testing

What is TestNG?

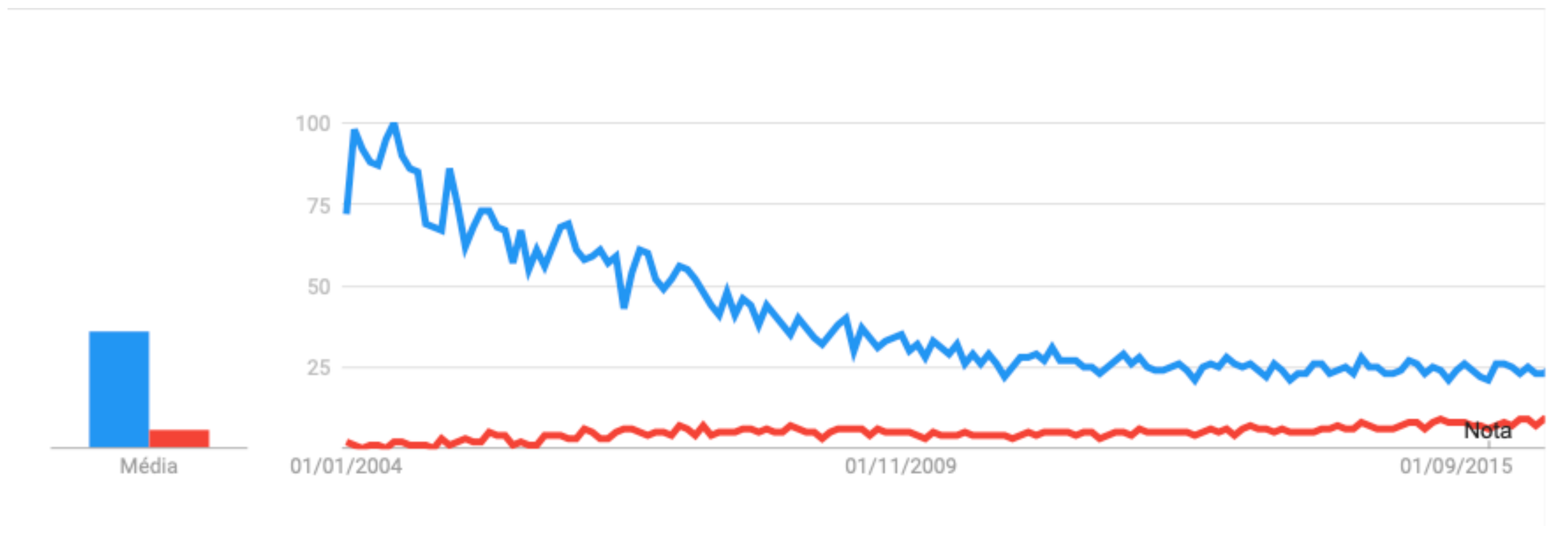
- Automated testing framework
- NG = Next Generation
- Similar to JUnit (especially JUnit 4)
- Not a JUnit extension (but inspired by JUnit)
- Designed to be better than JUnit, especially for higher levels of testing
- Created by Dr. Cédric Beust (of Google)
- Open source (http://testng.org)

History

- **sUnit** for Smalltalk (~1998, Kent Beck)
- **JUnit** (~2000, Kent Beck & Erich Gamma)
  - Latest 3.x: 3.8.2 (March 2006)
- **TestNG** (~2004)
  - Latest: 7.1.0 (2019)
- **JUnit 4.x** (2006)
  - Latest: 4.13 (2019)
- **JUnit 5** (September 2017)
  - Latest: 5.6.0 (2020)

Why TestNG?

- Interest over time using Google Trends
  - 1-1-2004 up to 8-3-2021
  - Terms: JUnit and TestNG



Why TestNG?

- However,
- Number of emails in the first 2 months of 2016
  - Junit mailing list: 6 mails
  - TestNG mailing list: 63 threads and 268 mails
- Number of releases from 2009 to 2016:
  - Junit: 6
  - TestNG: 26

Implementation of a Test case

- Test Case = a test of something
- In order to implement a test case you need to know its specification:
  - The input
    - Includes the parameters, initial state of the object being invoked and maybe some other global variables
  - The method to test
  - The expected output
    - Includes returned value (if any), expected result state of the invoked object, and (maybe) expected state of parameters, …

Properties of an implemented test case

- Test a single condition of the IUT
  - Do not try to exercise the same method several times to save implementation time
  - Follow the AAA pattern
- Independent
  - Should not depend on the outcome of the previous test case
- Self-cleaning
  - Returns the system's state to initial state
- Documented
  - Test goal should be clear and understandable
    - Document the test method or
    - Use a good test method name

Properties of an implemented test case - 2

- Accurate
  - Agrees with documentation
- Reasonable probability of catching a defect
- Repeatable
  - Can be used to perform the test over and over
  - It is completely automated
- Simple and clear to understand
  - It should be small
    - No more than 10-15 LOC excluding setup/tear down
- Fast
- …

AAA Pattern

- Implementation of a test case should follow the **Arrange-Act-Assert** pattern
- Each testing method should group its code into three functional sections (separated by blank lines):
  - **Arrange** all necessary preconditions and inputs
    - Instantiate object under test and set up test data
  - **Act** on the method under test
    - Invoke method under test on object under test
  - **Assert** that the expected results have occurred
    - Check that result after invocation is equal to expected resul

- Application of this pattern is orthogonal to the testing framework

AAA - Advantages

- Makes the test code easier to read
- Clearly separates what is being tested from the setup and verification steps
- Avoids some test errors
  - Assertions intermixed with "Act" code.
  - Test methods that try to test too many different things at once.

Unit Test Scenario – The Three A's

- No language/framework support for the specification of the several sections
  - Use comments

```
@Test
public void testWithdraw() {
    // Arrange
    AccountImpl account = new AccountImpl("1234", 2000);
     int amount = 300;

     // Act
    account.withdraw(amount);

   // Assert
   assertEquals(1700, account.balance());
}
```

Test class/method in TestNG

- `test case = method`
- Use Java annotations to setup and configure tests

```
import org.testng.annotations.Test;

public class MyTestClass {
  @Test
  public void aTestMethod() throws ... { ... }
}

or

import org.testng.annotations.Test;
@Test
public class MyTestClass {
  public void aTestMethod() throws ... { ... }
}
```

  - All public methods MyTestClass are test methods
  - Can still use **@Test** in methods for specifying other properties

TestNG Assertions - 1

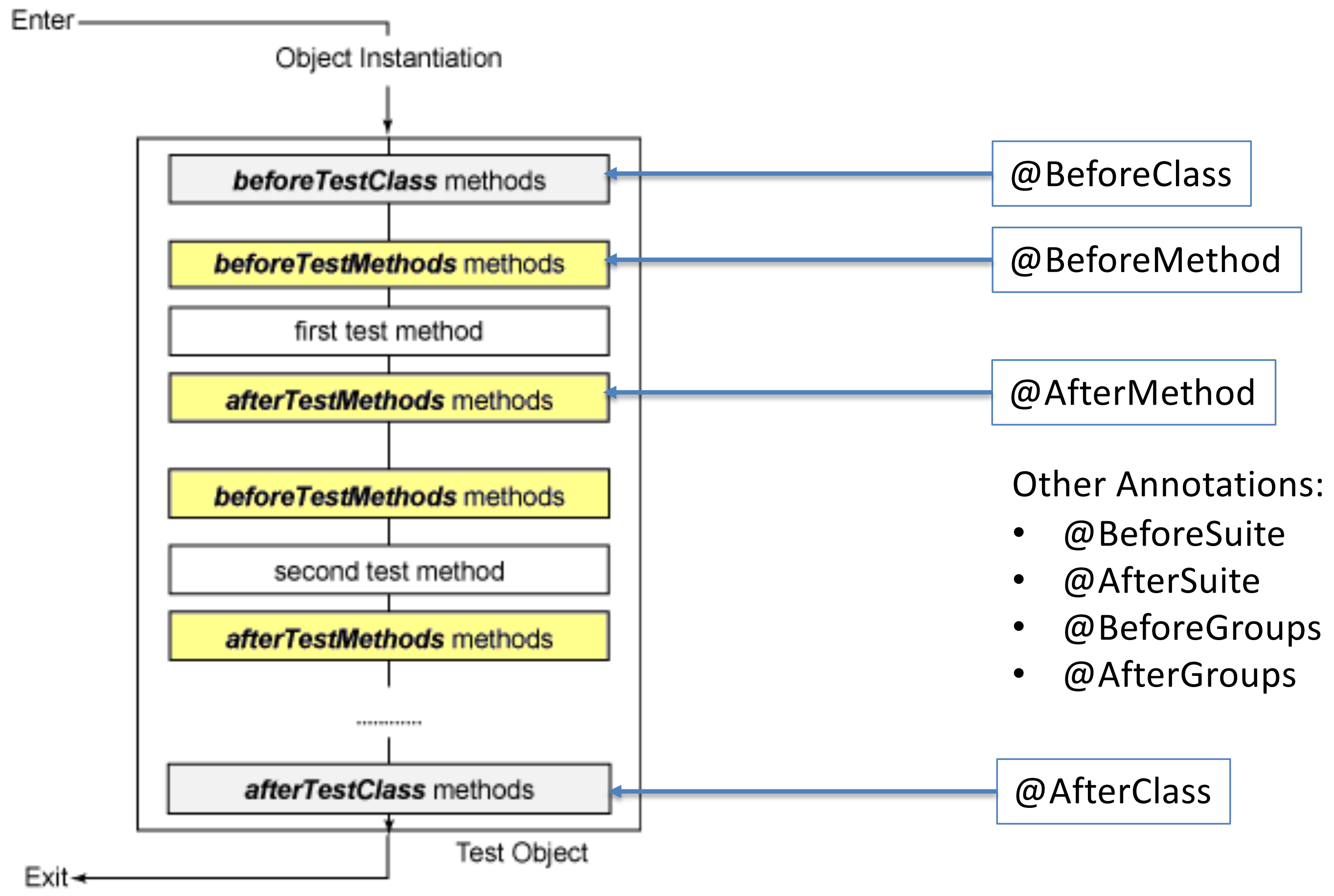
```
import static org.testng.Assert.*;
import org.testng.annotations.Test;

public class MyTest {
  @Test
  public void myTestMethod() {
    // Arrange
    ...
    // Act
    ...
    // Assert
    assertTrue(boolExpression);
    // ... more assertions
  }
}
```

TestNG Assertions - 2

- TestNG: similar to JUnit 4
  - assertEquals and assertNotEquals
  - assertNull and assertNotNull
  - assertSame and assertNotSame
  - assertTrue and assertFalse
  - fail

- TestNG assertEquals(…) assertion signatures are different than JUnit's:
  - **`JUnit:  [msg,]  expected, actual`**
  - **`TestNG: actual, expected  [, msg]`**

TestNG Engine



Enter
Object Instantiation
beforeTestClass methods
beforeTestMethods methods
first test method
afterTestMethods methods
beforeTestMethods methods
second test method
afterTestMethods methods
....
afterTestClass methods
Test Object
Exit
@BeforeClass
@BeforeMethod
@AfterMethod
@AfterClass

Other Annotations:

- @BeforeSuite
- @AfterSuite
- @BeforeGroups
- @AfterGroups

Example

```java
package com.asjava;
import org.testng.annotations.*;

public class TestNGTest {
    @BeforeMethod public void beforeMethod() {
        System.out.println(”@Method");
    }
    @BeforeClass public void beforeClass() {
        System.out.println("@BeforeClass");
    }
    @Test public void test1() {
        System.out.println("test1");
    }
    @Test public void test2() {
        System.out.println("test2");
    }
    @AfterClass public void afterClass() {
        System.out.println("@AfterClass");
    }
    @AfterMethod public void afterMethod() {
        System.out.println("@AfterMethod");
    }
}
```

Result of execution

```
@BeforeClass
@BeforeMethod
test1
@AfterMethod
@BeforeMethod
test2
@AfterMethod
@AfterClass
================================
Custom suite
Total tests run: 2, Failures: 0, Skips: 0
================================
```

Handle Expected Exception - 1

- **How to test a method that can throw an exception?**

```java
package com.asjava;

import static org.testng.Assert.*;
import org.testng.annotations.Test;
import java.util.List;

public class TestNGExpectedExceptionTest {
  @Test public void testNullPointerException() {
    try {
      List list = null;
      int size = list.size();
      fail("The test should have failed");
    } catch (NullPointerException e) {
      // success, do nothing: the test will pass
      // may add some asserts to check if object/system was modified
    }
  }
}
```

Fails to follow AAA pattern

Handle Expected Exception - 2

- **A conciser way:**
  - Use expectedExceptions attribute of @Test

```
import static org.testng.Assert.*;
import org.testng.annotations.Test;
import java.util.List;

public class TestNGExpectedExceptionTest {
 @Test(expectedExceptions = NullPointerException.class)
 public void testNullPointerException() {
   // Arrange
   List list = null;
   // Act
   int size = list.size();
 }
}
```

- If exception does not occur, test is marked as a failure
- Makes tests more concise, making the test case more readable and understandable
- Can accept mote than one exception:
  - **@Test(expectedExceptions = { T1.class, ... })**
- **However**, not checking
  - state of system/out after exception
  - error message of the exception object
- How to handle the invocation of a method that should not throw an exception?
  - Place fail() *inside* catch block

Handle Expected Exception - 3

- There is an alternative way using lambda expressions and the *assertThrows* method
  - static <T extends java.lang.Throwable> void assertThrows(java.lang.Class<T> throwableClass, Assert.ThrowingRunnable runnable)
  - We can now have the Assertion region

```
import static org.testng.Assert.*;
import org.testng.annotations.Test;
import java.util.List;

public class TestNGExpectedExceptionTest {
  @Test public void testNullPointerException() {
     // Arrange
     List list = null;

     // Act
     assertThrows(NullPointerException.class, () -> { list.size(); } );

    // Assert
    …
  }
}
```

Handle Expected Exception - 4

- We can even check the exception thrown with **expectThrows**
  - `static <T extends java.lang.Throwable> T expectThrows(java.lang.Class<T> throwableClass, Assert.ThrowingRunnable runnable)`

```
import static org.testng.Assert.*;
import org.testng.annotations.Test;

public class TestNGExpectedExceptionTest {
  @Testpublic void testNullPointerException() {
    // Arrange
    NullPointerException exc;
    java.util.List list = null;

    // Act
    exc = expectThrows(NullPointerException.class, () -> { list.size(); } );

    // Assert
    assertTrue(exc.getMessage().contains(”…"));
  }
}
```

- Remark that act section also has a part of assert

Parameterized Tests

- Sometimes, there are several similar test cases, each having a different combination of the input parameters and corresponding expected output
- How to implement these test cases?
- Fist Solution: copy/paste approach
  - Not ideal
- Best solution: the testing framework should make this simple

Parameterized Tests in TestNG

- Based on data providers
  - It is a method that returns Object[][]
    - First dimension´size is the number of times the test method will be invoked
    - Each row must be compatible with the parameter types of the test method
  - Or returns Iterator<Object[]>
    - A lazy version of the previous approach
  - Use @DataProvider(name = ”nameOfDataProvider”)
  - Data provider method can know the name of test method
- Assign a data provider to a test method
  - @Test(dataProvider = “nameOfDataProvider”)

Example

```
@DataProvider
private Object[][] getMoney(){
   return new Object[][] {
      {new Money(4, "USD"), new Money(3, "USD"), 7},
      {new Money(1, "EUR"), new Money(4, "EUR"),  5},
      {new Money(1234, "CHF"), new Money(234, "CHF"), 1468}};
}

@Test(dataProvider = "getMoney")
public void shouldAddSameCurrencies(Money a, Money b, int expectedResult) {
    // Act
   Money result = a.add(b);

   // Assert
   assertEquals(result.getAmount(), expectedResult);
}
```

Test Groups

- A group contains any number of test methods.
  - Groups can span classes
- Each test method can be tagged with any number of groups:
  - @Test // no groups
  - @Test (groups = “group1”)
  - @Test (groups = { “g1”, “g2”, ... })
- Groups can also be externally defined (TestNG xml configuration file)
- It is possible to have a group of groups
- A group is identified by a unique string (don’t use white space)
  - E.g., “slow”, “fast”, “gui”, “check-in”, “week-end” “unit”,“regression”,“integration”,“broken.unknownReason”

Groups -2

- TestNG community suggests hierarchical names from more general to less
  - database.table.CUSTOMER or alarm.severity.cleared
  - Design group names so that you can select them with prefix patterns
- Example:

```
@Test(groups = { “goldenRegression" })
public class All {
  @Test(groups = { “regression" })
   public void method1() { }

   public void method2() { ... }
}
```

Groups - 3

- Execute test cases belonging to a group

```
Code:
package example1;
public class Test1 {
  @Test(groups = {"func", "check"})
  public void testMethod1() {
  }

  @Test(groups = {"func", "check"} )
  public void testMethod2() {
  }

  @Test(groups = { "func" })
  public void testMethod3() {
  }
}
```

```
Configuration file:

<test name=”E1">
  <groups>
    <run>
      <include name="func"/>
    </run>
  </groups>
  <classes>
    <class name="example1.Test1"/>
  </classes>
</test>
```

Dependency testing in TestNG

- Make sure that execution of a test case is made only if a given test cases was executed with success before
- TestNG uses *dependOnMethods* or *dependsOnGroups* to implement the dependency testing
  - If the dependent method fails, all the subsequent test methods will be skipped, not marked as failed
  - Imposes a test execution order
- Usually, bad practice for unit testing
- But very important for system and integration testing
  - Fail fast
    - Run full system tests only if smoke test passed

Dependency Testing in TestNG

- Important in final report
- Test methods not executed due to a dependency:
  - Marked as SKIP
  - Not as Failed

Dependency Testing - Examples

```
import org.testng.anotations.*;
public class TestNGTestDependency {
  @Test
  public void serverStartedOk() {}

  @Test(dependsOnMethods = { "serverStartedOk" })
   public void method1() {}
}
```

```
import org.testng.anotations.*;
public class TestNGTestDependency {
  @Test(groups = { "init" })
  public void serverStartedOk() {}

  @Test(groups = { "init" })
  public void initEnvironment() {}

  @Test(dependsOnGroups = { "init*" })
  public void method1() {}
}
```

Concurrency

- Execute a test method several times using one or more threads
- Example:

```
@Test(threadPoolSize = 3, invocationCount = 20)
public void concurrencyTest() {
   System.out.print(" " + Thread.currentThread().getId());
}
```

- Result:
  - 13 12 13 11 12 13 11 12 13 12 11 13 12 11 11 13 12 11 12 13

Ignored Test Cases

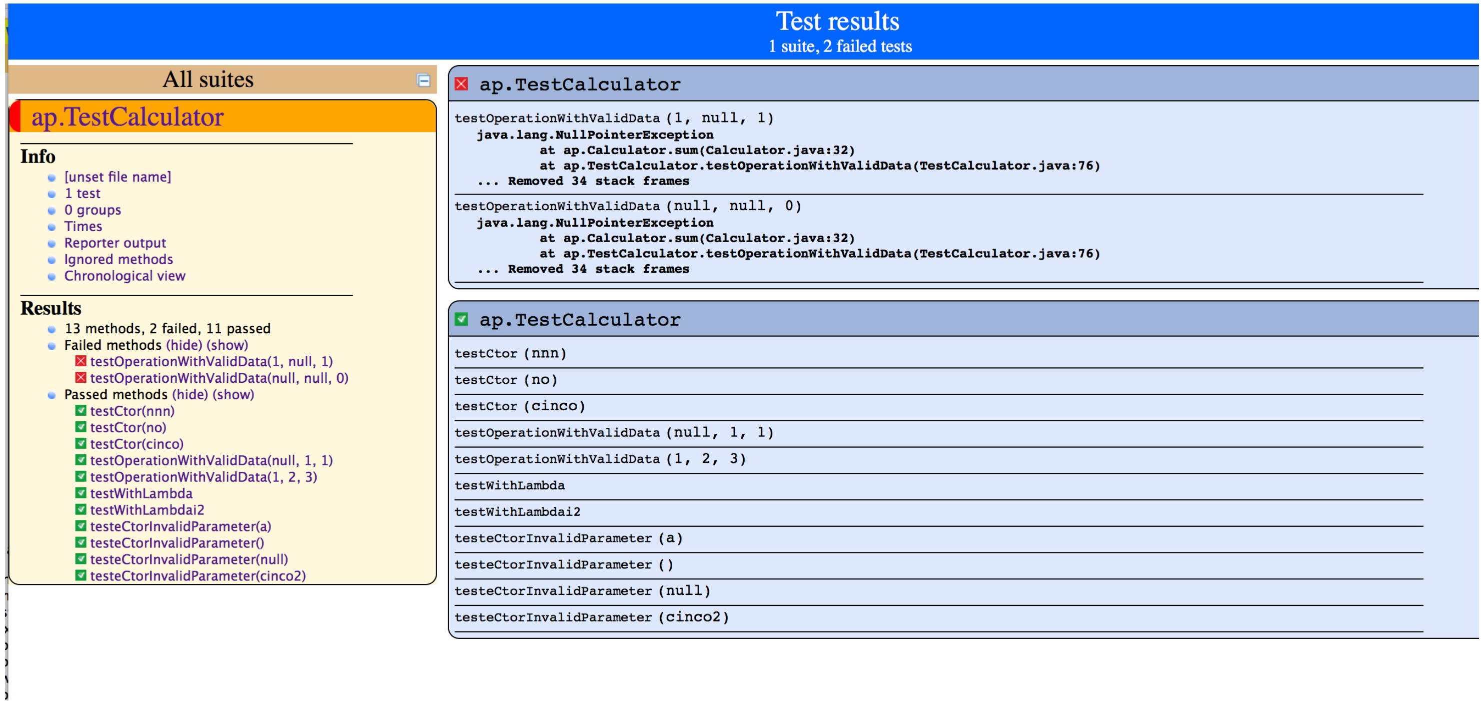
- Enable or disable test cases
  - @Test(enabled = false)
  - Or use @Ignore
  - Add to group that is excluded
  - Very useful when you have a test case that is broken

Running Failed Tests

- TestNG creates a testng-failed.xml in output directory
  - Contains failed methods
  - Allows to re-run the failed tests
  - Can reproduce the failures and verify fixes quickly

TestNG Report

- With Maven
  - Report stored in target/surefire-reports/index.html



Test results
1 suite, 2 failed tests

All suites

ap.TestCalculator

Info
- [unset file name]
- 1 test
- 0 groups
- Times
- Reporter output
- Ignored methods
- Chronological view

Results
- 13 methods, 2 failed, 11 passed
- Failed methods (hide) (show)
  - testOperationWithValidData(1, null, 1)
  - testOperationWithValidData(null, null, 0)
- Passed methods (hide) (show)
  - testCtor(nnn)
  - testCtor(no)
  - testCtor(cinco)
  - testOperationWithValidData(null, 1, 1)
  - testOperationWithValidData(1, 2, 3)
  - testWithLambda
  - testWithLambdai2
  - testeCtorInvalidParameter(a)
  - testeCtorInvalidParameter()
  - testeCtorInvalidParameter(null)
  - testeCtorInvalidParameter(cinco2)

ap.TestCalculator

```
testOperationWithValidData (1, null, 1)
   java.lang.NullPointerException
           at ap.Calculator.sum(Calculator.java:32)
           at ap.TestCalculator.testOperationWithValidData(TestCalculator.java:76)
   ... Removed 34 stack frames
testOperationWithValidData (null, null, 0)
   java.lang.NullPointerException
           at ap.Calculator.sum(Calculator.java:32)
           at ap.TestCalculator.testOperationWithValidData(TestCalculator.java:76)
   ... Removed 34 stack frames
```

ap.TestCalculator

```
testCtor (nnn)
testCtor (no)
testCtor (cinco)
testOperationWithValidData (null, 1, 1)
testOperationWithValidData (1, 2, 3)
testWithLambda
testWithLambdai2
testeCtorInvalidParameter (a)
testeCtorInvalidParameter ()
testeCtorInvalidParameter (null)
testeCtorInvalidParameter (cinco2)
```

More Information

- Detailed comparison between JUnit and TestNG
  – https://www.baeldung.com/junit-vs-testng