Testing, JUnit Extensions, TDD

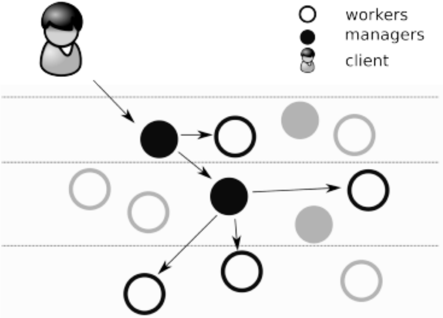
PV260 Software Quality

Stanislav Chren, Václav Hála

15. 4. 2021

Developers' Tests

- Unit tests
- Integration tests
- End-to-end tests



- Tomek Kaczanowski, *Practical Unit Testing with...*

Unit Tests

Unit test. . .

- focuses on single class
- makes sure that YOUR code works
- controls context
- knows nothing about the users of the tested system
- is unaware of layers, external systems and resources
- runs very quickly, is executed frequently

- Tomek Kaczanowski, *Practical Unit Testing with...*

Unit Tests

Unit test DOES NOT...

- talk to the database
- communicate across the network
- touch the file system
- misbehave when run in parallel with any other unit tests
- require special things done to your environment to run

- Michael Feathers, *A Set Of Unit Testing Rules*

Anatomy of a Unit Test

AAA

- Arrange
- Act
- Assert

BDD

- Given
- When
- Then

xUnit

- Setup
- Exercise
- Verify
- Teardown

http://c2.com/cgi/wiki?ArrangeActAssert
http://martinfowler.com/bliki/GivenWhenThen.html
http://xunitpatterns.com/Four%20Phase%20Test.html

JUnit extensions

- JUnit is an extremely powerful tool and virtually anything can be done using only the pure JUnit core functionality
- In some cases however we might benefit from using extensions of the basic functionality, syntactic sugar . . .
- These allow us to work faster, reduce the boilerplate code which brings no value, and make the test suite easier to maintain
- For most common needs both third party libraries and native JUnit extensions (some only in experimental branch) exist

JUnit extensions

- Fluent API for assertions
  - Hamcrest http://hamcrest.org/JavaHamcrest/
  - AssertJ https://assertj.github.io/doc/
- Parametrized /Data-Driven tests
  - JUnit 5 Parametrized https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests
  - Zohhak runner http://piotrturski.github.io/zohhak/

JUnit extensions - cont

- Property testing using randomized input
  - JUnit Theories `http://junit.org/apidocs/org/junit/experimental/theories/Theories.html`
  - junit-quickcheck `https://github.com/pholser/junit-quickcheck`
- And many others
  - Unitils `http://www.unitils.org/summary.html`
  - catch-exception `https://github.com/Codearte/catch-exception`
  - tempus-fugit `http://tempusfugitlibrary.org/`

AssertJ

http://joel-costigliola.github.io/assertj/

- Rich DSL, specific for many types - Collections, Strings, numbers, Exceptions, Time . . .
- Really helpful error messages
- Soft Assertions - show all errors, not just the first
- Extractors and Tuples
- Many extensions exist to test Database, Swing, Guava...

see homepage for extensive showcase of features

AssertJ

example: AssertJTest class

catch-exception

https://github.com/Codearte/catch-exception

- Catch and verify exceptions in a single line of code
- The test is more concise and easier to read.
- The test cannot be corrupted by a missing assertion.
- A single test can verify more than one thrown exception.
- The test can verify the properties of the thrown exception after the exception is caught.
- The test can specify by which method call the exception must be thrown.

- *Javadoc of CatchException class*

catch-exception

- Java 8+ syntax (version 2.0.0)

  ```
  MyObject myObject = new MyObject();
  catchException(() -> myObject.doStuff(1));
  Exception caught = caughtException();
  assertThat(caught).is...
  ```

catch-exception

example: CatchExceptionTest class

Zohhak

https://code.google.com/p/zohhak/

Allows us to run one test on many sets of data, provided in annotation next to the testcase

```
@TestWith({
    "1,2,3",
    "-19,7,-12"
})
public void testAdd(int a, int b, int expected) {
    Calculator calc = new Calculator();
    int result = calc.add(a,b);
    assertEquals(expected, result);
}
```

Zohhak - Data

- The Strings inside the `@TestWith({...})` each represent one test input
- Inside each of these input Strings individual arguments for the test are separated by commas (',')
- Types of the arguments are infered from the parameters of the test method and the arguments are coerced to these types before being passed to the test
  - Coercion of basic primitive types comes out-of-th-box
  - Custom coercion for any type can be written

Zohhak - Coercions

For more complex types we have to teach zohhak how to convert from String (the String in data annotation) to our type

```java
@Coercion
public Person toPerson(String input) {
    String[] split = input.split(";");
    Person person = new Person(split[0], split[1]);
    return person;
}
```

We can then use Person in our tests

```java
@TestWith({
    "John;Doe",
    "Frank;Perceval"
})
public void testWithPerson(Person person){
```

Zohhak

example: Vector2DTest class

JUnit 5 Parametrized

https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests

- Same purpose as Zohhak
- Offers more options but can be more cumbersome as well
- Can read data from file, e.g. CSV
- Instead of `@Test` annotation, use `@ParametrizedTest` followed by another annotation for data source

example: JUnit5Parametrised.java

junit-quickcheck

http://pholser.github.io/junit-quickcheck/site/0.7/index.html

- We don't test concrete inputs but properties of code
- Input is generated randomly
- The test is a specification of what the code should do
- If error is found QuickCheck tries to 'Shrink' it to 'smallest' possible value which causes the same error
- Inspired by QuickCheck for Haskell `https://hackage.haskell.org/package/QuickCheck`

junit-quickcheck

```java
@RunWith(JUnitQuickcheck.class)
public class SymmetricKeyCryptographyProperties {
    @Property
    public void decryptReversesEncrypt(
            byte[] plaintext, Key key){
        Crypto crypto = new Crypto(key);
        byte[] ciphertext = crypto.encrypt(plaintext);
        assertEquals(plaintext,
                     crypto.decrypt(ciphertext));
    }
```

junit-quickcheck

Task 1

- Try to run QuickcheckTest, it should fail
- The test is correct, implementation is broken
- Find what is wrong with the current implementation
- Implement StringSplitter so that the test passes

junit-quickcheck

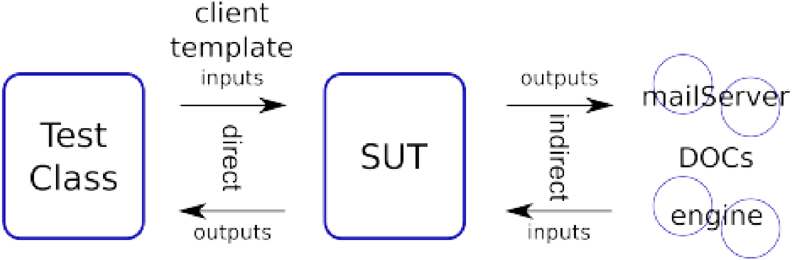
Task 2

- Come up with at least 3 properties of a sorting algorithm
- Work with the Sorter interface
- Write quickcheck test for each of these properties
- Implements the Sorter using algorithm of your choice

Behavior Verification

State Verification

Behavior Verification



Mocking in Unit Testing

- Unit testing is simple for classes with no dependencies
- How do we test an object which depends on many other things (many of which might not even be implemented yet) ?
- We create stand-in objects which share interface with the required dependency
- Inside, instead of some complex behavior, these are hard-wired to work in the one particular test case
- We can create these substitutes either by hand or use a mocking framework

Mockito

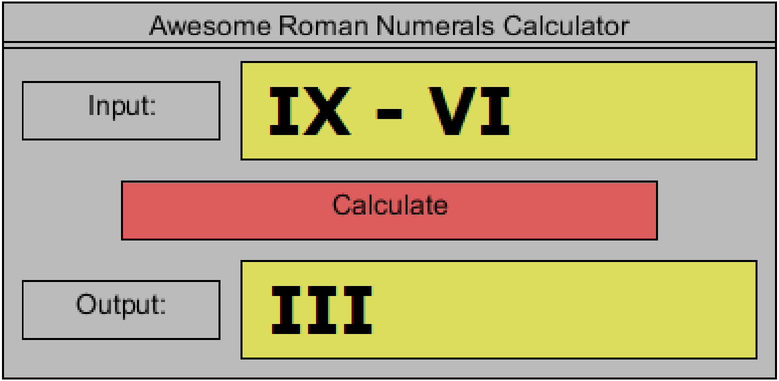
http://mockito.org/

> *We decided during the main conference that we should use JUnit 4 and Mockito because we think they are the future of TDD and mocking in Java.*
> *(Dan North - author of BDD)*

- Interaction verification
- Input stubbing (data, exceptions. . .)
- Test Spy wrappers
- Mock both classes and interfaces
- Lightweight API

Working Example

- Model for an app doing basic math on Roman numerals
- We only care about the inner logic, the UI doesn't concern us



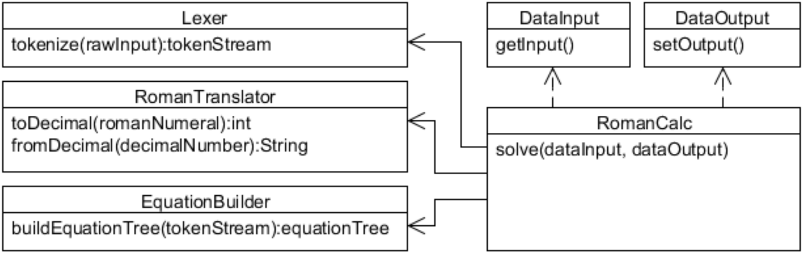
Awesome Roman Numerals Calculator

Input: IX - VI

Output: III

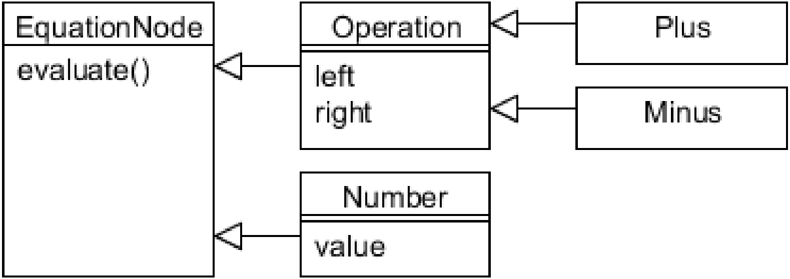
Working Example - Structure

- We already have the design done, all interfaces are prepared
- DataInput and DataOutput represent the textboxes
- Clicking the Calculate button calls the solve method


Lexer
tokenize(rawInput):tokenStream
RomanTranslator
toDecimal(romanNumeral):int
fromDecimal(decimalNumber):String
EquationBuilder
buildEquationTree(tokenStream):equationTree
DataInput
getInput()
DataOutput
setOutput()
RomanCalc
solve(dataInput, dataOutput)

Working Example - Structure

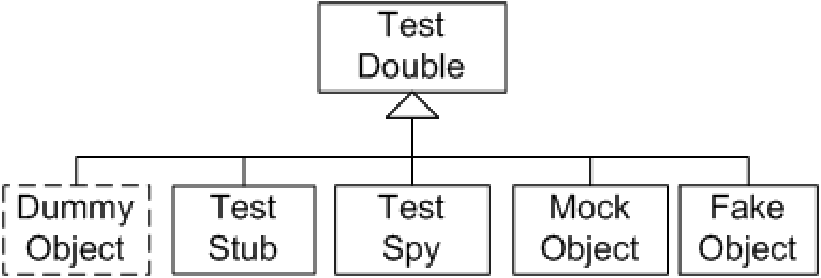
- Lexer tokenizes the raw input
- Number tokens are translated by the RomanTranslator and sent to EquationBuilder
- Tree representation of the equation is assembled
- The decimal result is translated to Roman numerals
- Formated result is sent back to output



Test Doubles Hierarchy

http://xunitpatterns.com/Test%20Double.html

- There are many types of stand-in objects used in testing
- Each plays a different role, the simplest type possible should be used (That is dont use a Mock if all you need is a Dummy)



Dummy Object

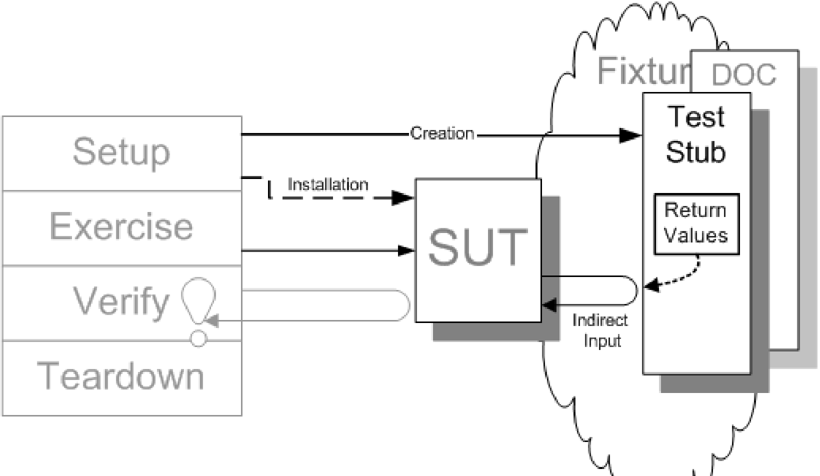
RomanCalculatorTest#testExceptionFromInput

- We need to provide real object (that is not *null*), but at the same time we know it will never be used during the test
- Even better, we pass *null* to the test which helps readability as we are clearly signalling that the value is not used
- This is of course not possible with null-checks in constructors, so we have to use dummies instead.
- To Assert or Not To Assert
  `http://misko.hevery.com/page/5/`

Test Stub

RomanTranslatingTokenStream#testConvertsToDecimalTokens

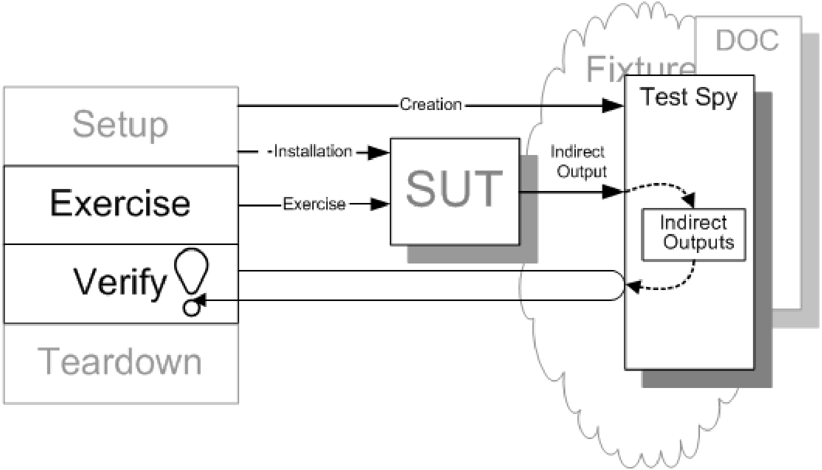
- We want one of SUT's dependencies to provide specific input to the SUT when queried



Test Spy

RomanTranslatingTokenStream#testRecognizesRomanNumeral

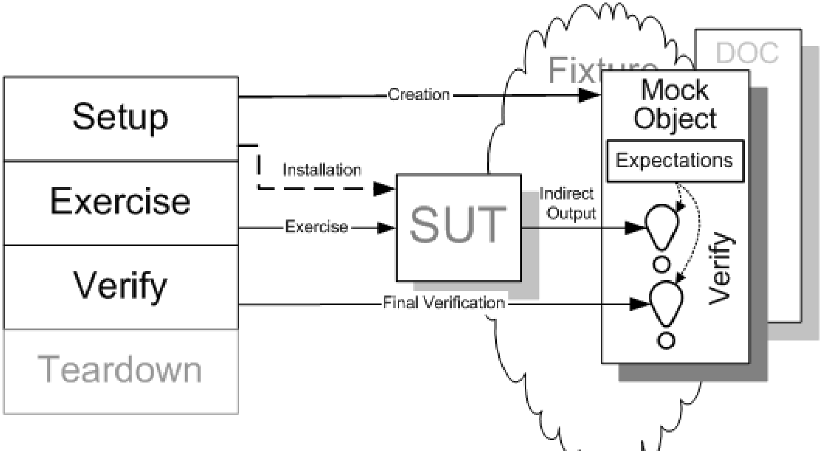
- We want to know SUT's interacts with one of its dependencies
- The spy only records the interaction, it is checked manually



Mock Object

RomanTranslatingTokenStream#testCorrectInputSingleOperator
RomanTranslatingTokenStream#testConvertsToDecimalTokens

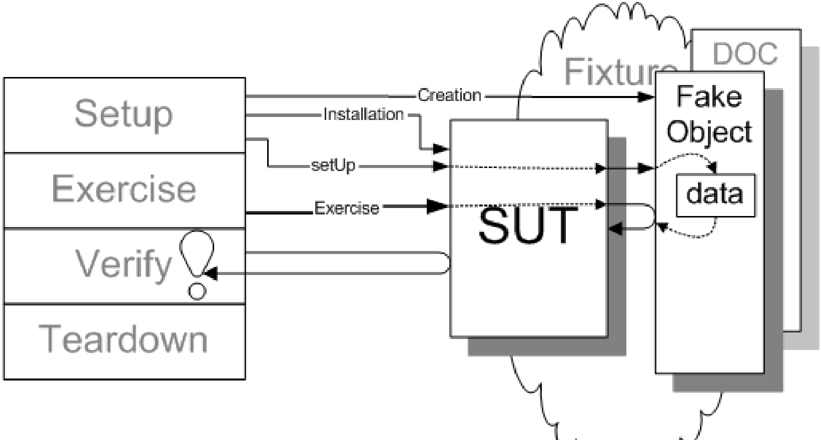
- Similar task as Test Spy, but checks the validity of SUT's interaction with the mock on the fly



Fake Object

No Example

- Has the same functionality as its real counterpart, but implements it in a more test friendly way
- e.g. an in-memory database instead of disk-based one



Test Spy vs. Mock

Test Spy

Mockito

- Arrange → Act → Assert
- Whole test runs
- Nice
- Verification always in caller

Mock

EasyMock

- Record → Exercise → Verify
- Stop on first error
- Strict
- Might be suppressed by environment

Test Doubles Exercise

Task 1

- Implement all tests in CustomerAnalysisTest

Test Coverage

In computer science, test coverage is a measure used to describe the degree to which the source code of a program is tested by a particular test suite.

- High coverage does not necasarilly mean that your project has quality tests (there could be tests with no assertions, hardly maintainable tests . . .)
- However, low coverage can point to parts of insufficiently tested code which has a high chance of containing all kinds of bugs and other problems

Types of Coverage

Consider this code:

```
public int doIt(boolean c1, boolean c2, boolean c3) {
    int x = 0;
    if (c1)
        x++;
    if (c2)
        x--;
    if (c3)
        x+=3;
    return x;
}
```

Types of Coverage

- Statement coverage
  - Check that all statements in the code are executed
  - For 100% coverage single test input required *(true, true, true)*
- Branch coverage
  - Check that all possible results of conditions occur
  - For 100% coverage two test inputs required *(true, true, true)*, *(false, false, false)* or any other combination with both true and false for all conditionals
- Path coverage
  - Every possible path through the code is executed
  - For 100% coverage all possible combinations of inputs (and values for member attributes if there were any) must be used, thats 8 cases for this example

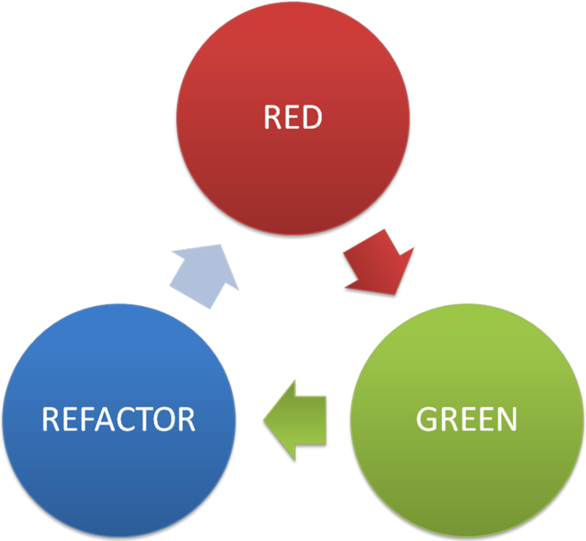
TDD - Overview

Test Driven Development: By Example, Kent Beck

> *Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards.*

- Quickly add a test.
- Run all tests and see the new one fail.
- Make a little change.
- Run all tests and see them all succeed.
- Refactor to remove duplication.
- Repeat . . .

Red Green Refactor



The Three Laws of TDD

Professionalism and Test-Driven Development, Robert C. Martin

1. You may not write production code until you have written failing unit test.
2. You may not write more of a unit test than is sufficient to fail, and not compiling is failing.
3. You may not write more production code than is sufficient to pass the currently failing test.

Tennis Game Kata - Scoring

- Each player starts with 0 points
- The scoring then goes like this 0 → 15 → 30 → 40
- If `A` has 40 and scores, and `B` doesn't have 40, `A` wins
- If both have 40 and `A` scores, `A` has Advantage
- If `A` has Advantage and scores, they win
- If `A` has Advantage, `B` has 40 and scores, both are at 40 again
- Scores are written in the format 'A - B', e.g. '30 - 15'
- When `A` has Advantage, the score is written as 'A - 40'
- If scores are equal, e.g. both have 30, it is called '30 all'
- If both players have 40 points, it is called 'deuce'

Tennis Game Kata - Task

- Try to not skip ahead and always have passing tests for existing functionality before moving forward
- We want to create a TennisGame which has `scoredA()`, `scoredB()` and `showScore()`
- The show method should return score in format defined above, if there is a winner it gives 'winner: A/B'
- Also if there is a winner already and either `scoredA()` or `scoredB()` is called, exception should be thrown