Test Driven Development, Unit tests & Mocking

...

Adam Kučera

03/2018

What is testing?

Manual testing

Automatic testing

Black-box testing

White-box testing

Unit testing

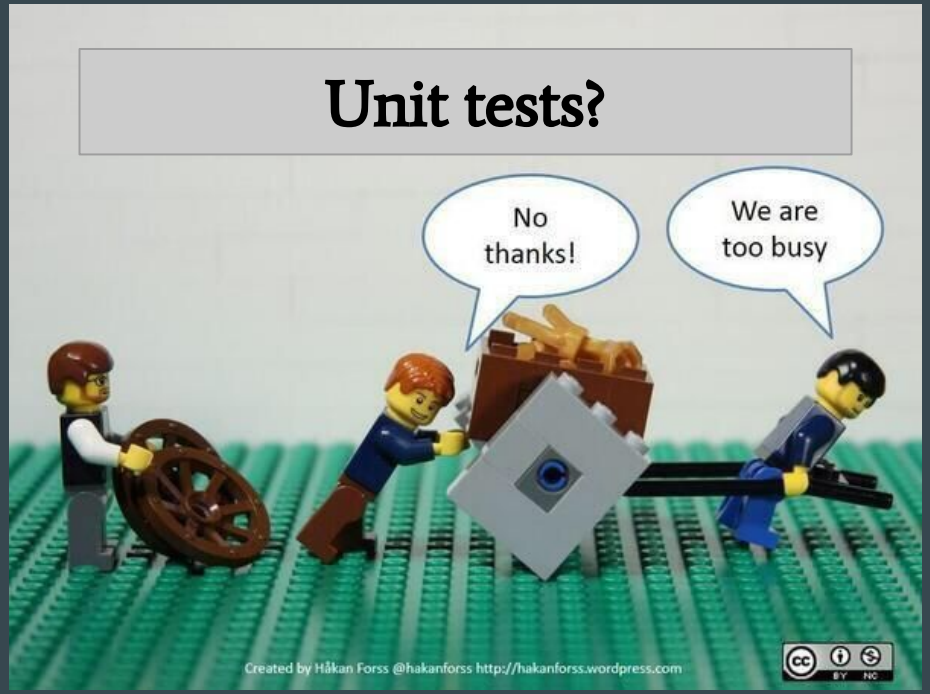
Integration testing

System testing

Integral part of programmer's life

Why unit testing?

- Verify *units* of code -> functions, classes, modules?
- Why? When you want to...
  - find **bugs**
  - check that new feature works, **quickly**
  - make **documentation** of a change
  - check that changing a feature doesn't break the code - **refactoring**
  - have **clean code** and architecture
  - save **money**
  - HAVE **CONFIDENCE**



How does a test look like? Arrange, Act, Assert

- **Arrange**
  - get system into desired state (sometimes no actions necessary)
- **Act**
  - perform action you want to test (usually call one method)
- **Assert**
  - verify that the result is what you expected
  - verify that interactions between objects happened
- **(After)**
  - clean up after your test

Arrange, Act, Assert

```java
@Test
public void testSum() {
  // Arrange
  ArithmeticCollection collection = new ArithmeticCollection();
  collection.add(3);
  collection.add(6);
  // Act
  int actual = collection.sum();
  // Assert
  assertThat(actual, is(9));
}
```

Always structure your tests in this manner. (without the comments)

What to test?

Read the code


HAPPY PATHS

Sad paths

Code coverage plugins

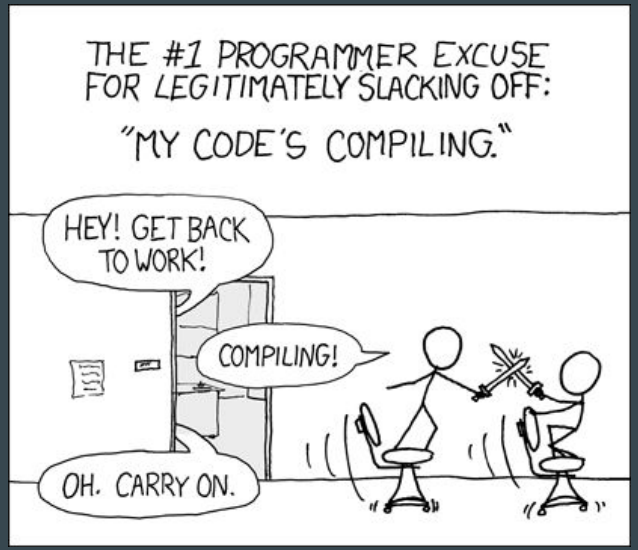
Sumador.wsdl × | Welcome × | index.html × | Main.java × | MainTest.java ×

```java
    /** Creates a new instance of Main */
    public Main() {
    }

    public void doThis(){
        System.out.println("This is covered");
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        System.out.println("Code Coverage Plugin is Cool");
        if (5<6){
            System.out.println("This will be covered");
        } else{
            System.err.println("This will not be covered");
        }
    }

}
```

F.I.R.S.T. - what makes a good test?

Fast

- Databases, network is slow
- Use mocking
- Keep performance in mind, when deciding what to tests to run
- Trade-off between confidence and speed



#2: “I am running tests...”

Isolated

- Tests should always behave the same
- They can run in any order, at any time
- Test only one thing in one test
- One reason to break the test

Repeatable

- You will always get the same result, when running the test on same codebase

Why does man print “gimme gimme gimme” at 00:30?

1502

We've noticed that some of our automatic tests fail when they run at 00:30 but work fine the rest of the day. They fail with the message "gimme gimme gimme" in stderr, which wasn't expected. Why are we getting this output?

/ date / man

share improve this question

463

edited Nov 20 '17 at 18:01
Jeff Schaller
25.4k 8 40 88

asked Nov 20 '17 at 14:19
Jaroslav Kucera
3,980 4 5 20

Self-validating

- Tests can be run automatically
- You don’t need to evaluate them manually

Timely

- You should always write tests - TDD
- You should run tests often
- Don’t necessarily add tests to old static and working code

Mocking - handling dependencies

Mocking

- Objects have dependencies and it is hard to test a unit while taking care of these dependencies.
- Dependencies like database, REST calls, disk reads are even a bigger problem.
- **Mocking** comes to the rescue.
- You wanted to test an **unit** in **isolation** anyway.

- Mocking = creating a fake dependency object, which behaves in a controlled manner
- You need **dependency injection** to enable Mocks.

Mocking frameworks

- Create mock
- Describe expectations:
  - WHEN some method is called with some parameters
  - THEN RETURN some object or THROW some exception
- Inject our mock to the tested class
- Do actual testing against the Mock
- Verify calls:
  - VERIFY that there a method call, exactly n TIMES

```
//create mock
EmployeesService mockService = mock(EmployeesService.class);
//set expectations
expect(mockService.getEmployeeByName("Adam")).andReturn(new Employee("Adam"));
//inject mock to our class
PaycheckService paycheckService = new PaycheckService(mockService);
//do actual testing
assertThat(paycheckService.raiseEmployeesSalary("Adam", 5000), is(true));
//verify that mock was called
verify(mockService, times(1)).getEmployeeByName("Adam");
```

Using mocks

- You should follow same principles for tests as before, when using Mocks
- Make sure, that you are testing against mock and not real service (e.g. change your data in test and see it fail)
- Mocks create a *gap* in tests => you need *integration* tests too!

TDD - Test Driven Development

Test Driven Development

- A brilliant technique which will lead to better quality code with good testability and test coverage

1. Write a test that fails. RED
2. Get the test to pass. GREEN
3. Clean up any code added or changed in the prior two steps. REFACTOR

Write test that fails

```
int sum(int first, int second) {
    return 0;
}


@Test
void testSumSumsFirstAndSecond() {
    assertThat(sum(2,3), is( value: 5));
}
```

1 test failed - 8ms

Make it green

```
int sum(int first, int second) {
    return 5;
}


@Test
public void testSumSumsFirstAndSecond() {
    assertThat(sum(2,3), is( value: 5));
}
```

1 test passed - 10ms

Refactor / write another test

```
int sum(int first, int second) {
    return 5;
}

@Test
public void testSumSumsFirstAndSecond() {
    assertThat(sum(2,3), is( value: 5));
}

@Test
public void testSumSumsZeros() {
    assertThat(sum(0,0), is( value: 0));
}
```

testSumSumsZeros	68ms
testSumSumsFirstAndSecond	0ms

Make it green...

```
int sum(int first, int second) {
    return first + second;
}

@Test
public void testSumSumsFirstAndSecond() {
    assertThat(sum(2,3), is( value: 5));
}

@Test
public void testSumSumsZeros() {
    assertThat(sum(0,0), is( value: 0));
}
```

testSumSumsZeros	7ms
testSumSumsFirstAndSecond	0ms

Golden rule

Always see your tests fail!

That’s the only way to be sure that you are testing what you want to test.

Sources

- Pragmatic Unit Testing in Java 8 with Junit
  - Available on Safari books (ask your manager to get you access, it is really good for beginners and quite short)
- The four levels of software testing