Software Testing

Artyom Semyonov

СПБгУ, СП

30 апреля 2020 г.

Software testing

What is software testing?

Software testing is an investigation conducted to provide stakeholders with information about the quality of the software product or service under test.

Manual vs Automated

Who does testing?

- **Manual testing** – perfomed by a *human*.
- **Automated testing** – perfomed by a *program*.

Manual vs Automated

Who does testing?

- **Manual testing** – perfomed by a *human*.
- **Automated testing** – perfomed by a *program*.

Why does manual testing still exist?

Manual vs Automated

Who does testing?

- **Manual testing** – perfomed by a *human*.
- **Automated testing** – perfomed by a *program*.

Why does manual testing still exist?

- t – time to manually perform some test scenario once
- T – time to automate the scenario
- n – count of the scenario repeat
- $T >> t$ (in most cases)
- if $T > n * t$ then manual testing is better.

Manual vs Automated

Who does testing?

- **Manual testing** – perfomed by a *human*.
- **Automated testing** – perfomed by a *program*.

Why does manual testing still exist?

- t – time to manually perform some test scenario once
- T – time to automate the scenario
- n – count of the scenario repeat
- $T >> t$ (in most cases)
- if $T > n * t$ then manual testing is better.

This lecture is about **Automated testing** only.

Functional vs non-functional

Software testing is about an application behaviour.

- **Functional testing** – check if *functionality* is correct.
- **Non-functional testing** – other checks.
    - Scalability testing
    - Performance testing
    - Security testing
    - Usability testing
    - . . .

Functional vs non-functional

Software testing is about an application behaviour.

- **Functional testing** – check if *functionality* is correct.
- **Non-functional testing** – other checks.
    - Scalability testing
    - Performance testing
    - Security testing
    - Usability testing
    - . . .

This lecture is about **Functional testing** only.

Testing levels (1)

We can perform testing of the application on different levels.

- **Unit testing**
  - Example: functions, classses
  - Unfair testing
  - But fast, simple and clear, stable

Testing levels (1)

We can perform testing of the application on different levels.

- **Unit testing**
  - Example: functions, classses
  - Unfair testing
  - But fast, simple and clear, stable
- **Integration testing**
  - Example: interaction with other system
  - More fair testing
  - Slower, more complicated, not so stable

Testing levels (1)

We can perform testing of the application on different levels.

- **Unit testing**
  - Example: functions, classses
  - Unfair testing
  - But fast, simple and clear, stable
- **Integration testing**
  - Example: interaction with other system
  - More fair testing
  - Slower, more complicated, not so stable
- **System testing**
  - Example: use the server API from a *different machine*
  - The fairest testing
  - Slow, complex, unstable

Testing levels (2)

Testing levels are sometimes vague.

- Testing of backend API – Integration? System?
- Testing of DAO – Unit? Integration?

Testing levels (2)

Testing levels are sometimes vague.

- Testing of backend API – Integration? System?
- Testing of DAO – Unit? Integration?

What you should know:

- Which test is fair and which is less fair.
- Best testing practices which are universal for any testing level. We will discuss them.

The «box» approach

- **Black-box testing** – The tester knows *nothing* about the implementation
- **White-box testing** – The tester knows *everything* about the implementation
- **Grey-box testing** – The tester knows *something* about the implmenetation

The «box» approach

- **Black-box testing** – The tester knows *nothing* about the implementation
- **White-box testing** – The tester knows *everything* about the implementation
- **Grey-box testing** – The tester knows *something* about the implmenetation

«Knows» means «As if he knows». Prefer the **Black-box** approach by following the encapsulation principle.

Terminology

«Test» has a vague meaning. It's better to use more precise terms:

- **Test-case** – a test scenario. Usually a method
- **Test-suite** – a set of test-cases. Usually a class

The test-case workflow

How do tests work?

- Test-case started
- Test-case finished. Result: *success* or *failure*
- If result is a *failure* then it generates a test-report

The purpose of the tests

Why do we need tests?

- Check the functionality. **Not the main purpose!**
- Fix the functionality. **The main purpose!!**
  - To Simplify a refactoring
  - Tests are also code specification
- Unit-tests force you to write more modular code (according to TDD)

Quality tests

Features of the quality test:

- Problem localiztion
- Stability
- Readable reports
- No duplicates **More tests $\neq$ better!**
- Tests should be simple. **We don't want to test the test-cases!**

Quality tests

Features of the quality test:

- Problem localiztion
- Stability
- Readable reports
- No duplicates **More tests $\neq$ better!**
- Tests should be simple. **We don't want to test the test-cases!**
  - Low *cyclomatic complexity* of the test code
  - Of course, other good coding practices

Kinds of test-cases

- Simple one-assertion
- General one-assertion
- Multi-assertion
- Property-base
- Parameterized

Simple one-assertion test-case

- **Actual result** – the real result returning by the program
- **Expected result** – the result you are expecting

```
val actualResult: R = getActualResult()
val expectedResult: R = getExpecatedResult()
actualResult ==? expectedResult // is equal to?
```

- The most simple kind of test-cases
- Prefer this kind if you can

General one-assertion test-case

```
    val actualResult: R = getActual()
    val expectedPredicate: (R => Boolean) = getExpecated()
    expectedPredicate(actualResult) ==? true
```

- The generalization of the *Simple one-assertion test-case*
- Use Matcher pattern to improve the quality of the test.

Multi-assertion test-case

```
    // ...
    actualResult1 ==? expectedResult1
    // ...
    actualResult2 ==? expectedResult3
    // ...
```

- The generalization of the *General one-assertion test-case*
- Avoid this kind of test-cases if you can.
- Use SoftAssert pattern to imporve the quality of the test.

Property-base test-case

- **Invariant** – the predicate that always must be true: $\forall x \in X : \textit{predicate}(x)$
- This kind of test-cases checks the invariants
- There are some situations where this type of test is better than a usual test
- Use Generator pattern to implement test-cases of this type.

Parameterized test-case

```
def testCase(params: P) = {
  // ... scenario
}
```

- Before test-case was a function *without* parameters
- Now test-case is function with *with* parameters
- The generalization of *any* kind of test-case.

Summary

What we have learnt:

- Tests should be as simple as possible
- Prefer Balck or Grey-box approach in most situations
- Choose the most appropriate kind of test-case
- Use test patterns like *Matcher*, *SoftAssert*, *Generator*

For self-study

- **Cyclomatic complexity**: use search engine
- **Test-driven-development (TDD)**: use search engine
- Mock pattern
  - Libraries: *Mockito*, *ScalaMock*
  - Don't abuse it. Use it if needed only!

Questions

Questions?