SOFTWARE DEVELOPMENT IN SCIENCE

05 TESTING

19/20 NOVEMBER 2019 | WOUTER KLIJN
CONTENTS

• Introduction
• Test types
• Unit tests
• How to start testing
• Testing and refactoring
• Conclusion
INTRODUCTION: TESTING

- Automatic programs or checklists of software
  - Functional requirements
  - Performance: Speed, Memory, etc.
  - Installation
  - Usability
  - Stakeholders / design requirements

- Future / a higher level reasons:
  - Prevent regressions
  - Document the code’s behavior
  - Provide design guidance
  - Supports refactoring
INTRODUCTION: TESTING IN SCIENCE CONS.

- Testing takes time
  - Science goal: publication
  - Return of investment is later (and for someone else)
- Every extra test:
  - Additional runtime
  - Increased probability ‘flaky’ tests
  - Increased maintenance
INTRODUCTION: TESTING IN SCIENCE PROS

- Increased development speed:
  - Guards against unforeseen code interactions
  - Software is big, tests reduce the code horizon
- Tests as documentation
- Improvements in design
  - Loosely coupled code
  - Formalize in Test Driven Development (TDD)

TDD = Test Driven Development
DDT = Design Driven Testing

https://bulldozer00.com/2015/01/18/beware-of-micro-fragmentation/
TEST TYPES

Credit: https://www.symbio.com/solutions/quality-assurance/test-automation/
TEST TYPES: MANUAL TESTS

- Manually running the code
  - Larger software projects
  - Graphical User Interfaces (hard to test)
- Dedicated testers:
  - Scripted / checklist
  - Exploratory tests

https://www.leaseweb.com/labs/2013/12/testing-techniques-better-manual-testing
TEST TYPES: ACCEPTANCE

- **Acceptance**: (automated) tests for core functionality
  - Business requirements
  - User story based:
    - “As a user I want A thus I do B”
- Data driven **delta** test
  - Input and validated output
  - Matches with scientific practice
    - A very good cost trade-off
    - Fragile: data changes with code changes
TEST TYPES: INTEGRATION

- Integration of different components
  - Different modules:
    - GUI and backend
    - Server and Client
  - OS and Software
  - Different applications
- Performed on the API (Application Programming Interface)

http://istqbexamcertification.com/what-is-integration-testing/
http://softwaretestingfundamentals.com/integration-testing/
TEST TYPES: UNIT TESTS

• Test of (smallest) non trivial units of functionality
  • Test **one thing** and proof its **correct**
  • **Independent** of other tests
  • **Fast** execution
• Written during development
• Costly to implement and maintain: long term goal!
• Can be a driver for design and refactoring
  • Test Driven Development

TDD = Test Driven Development
DDT = Design Driven Testing
UNIT TESTS: XUNIT

- De-facto standard for unit testing
  - Implemented in most (all?) programming languages

- **test fixture**
  - Preparation for one or more tests

- **test case**
  - The smallest unit of testing

- **test suite**
  - Collection of test cases executed together

- **test runner**
  - Executes the test

http://pythontesting.net/framework/unittest/unittest-introduction/
https://docs.python.org/2/library/unittest.html
UNIT TESTS: XUNIT

• Setup -> exercise System Under Test (SUT) -> teardown
  1. Create files, and dependencies needed to run the component
  2. Exercise SUT and validate the output:
     • assertxxxxx
  3. Delete used resources
import unittest

def function(parameter):
    return parameter

class TestSomething(unittest.TestCase):
    def setUp(self):
        pass

def test_fail(self):
    self.assertEqual(function(13), 12)

def test_succeed(self):
    self.assertEqual(function(12), 12)

def teardown(self):
    pass

if __name__ == '__main__':
    unittest.main()
UNITTEST: TEST ASSERTIONS

• Validation with assertions. The most commonly used are:
  • `assertEqual(a, b)`
  • `assertTrue(a)`

• `assertIs(a, type)`
• `assertRaises(Exception, function, *args)`

• `assertAlmostEqual(a, b, precision)`  # to test float values
• `assertLess(a, b)`
UNITTESTS: PRACTICAL HINTS

• Tests are written for other people
  • Also test things *obvious* for you
• Add to new or about to change code
• Test corner cases:
  • Negative, zero, one, two, three, many, max
  • Test correct and incorrect behavior (exceptions)
• Bigger project: *speed*
  • Test suite with subsets of the tests
• Use a *checklist*?

http://aurisc4.blogspot.de/2015/01/basic-rules-of-automated-software.html
http://www.thebraidytester.com/downloads/YouAreNotDoneYet.pdf
HOW TO START TESTING

• Writing down the **manual tests** you already do
  • Doubles as documentation

• Create an **data driven delta** test
  • Create test data
  • Forces you to think about ‘user’ interactions
  • Doubles as introductory how-to
HOW TO START TESTING

• Start with a single critical component, isolate.
  • ‘Publishable’ function
  • Often changed code
  • Complicated / scary code
  • Code with lots of errors
• Unit test for small parts of the code that do one and only one thing
  • "Legacy software is any code without tests"
• Adding test is often leads to refactoring
TESTING AND REFACTORIZING

https://ronjeffries.com/xprog/articles/refactoring-not-on-the-backlog/
CONCLUSION

• Test have costs and benefits

• At a minimum write down your manual tests and automate your data driven delta test

• Use a test framework (xUnit)

• Add tests for code that you are changing
Questions?
TOWARDS CONTINUES INTEGRATION


- https://www.sonarqube.org/

- http://www.aviransplace.com/2013/03/16/the-roard-to-continues-delivery-part-1/