JÜLICH
FORSCHUNGSZENTRUM

Software Development in Science

Testing

November 2022 | Online | Wouter Klijn

Contents

- Introduction
- Test types
- Unit tests
- How to start testing
- Testing and refactoring
- Conclusion

Introduction: Testing

- Automatic programs or checklists of software.
  - Meets the design requirements
  - Performance: Speed, Memory, etc.
  - Usable.
  - Installation
  - Stakeholders requirements
- At a higher level:
  - Prevent regressions
  - Document the code's behavior
  - Provide design guidance
  - Supports refactoring

https://en.wikipedia.org/wiki/Software_testing
https://www.devmynd.com/blog/five-factor-testing

Introduction: Testing in science

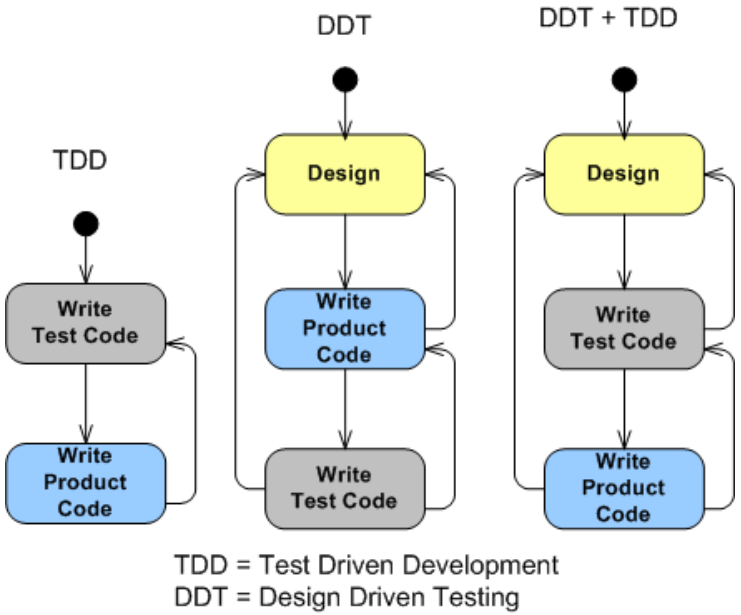
- Testing takes time
  - Science goal: publication
  - Return of investment is later (and for someone else)

- Every extra test:
  - Additional runtime
  - Increased probability ‘flaky’ tests
  - Increased maintenance

http://www.lihaoyi.com/post/PrinciplesofAutomatedTesting.html

Introduction: Testing in science

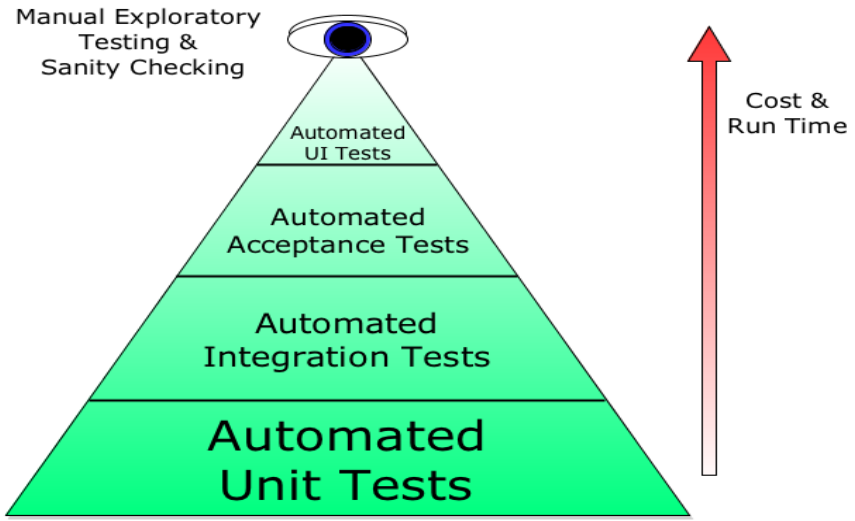
- Increased development speed :
  - Guards against unforeseen code interactions
  - Software is big, tests reduce the code horizon
- Tests as documentation
- Improvements in design result in a loosely coupled design.
  - TDD / DDT formalizes this



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

Test types

- Manual
- Automated UI tests (hard)
- Acceptance tests
  Data driven(delta)
- Integration tests
  Component tests
- Unit testing



http://willhamill.com/2013/08/12/automated-testing-and-the-evils-of-ice-cream

Test types: Manual tests

- Manually running the code

- Can be expanded and become more formalized:
  - 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

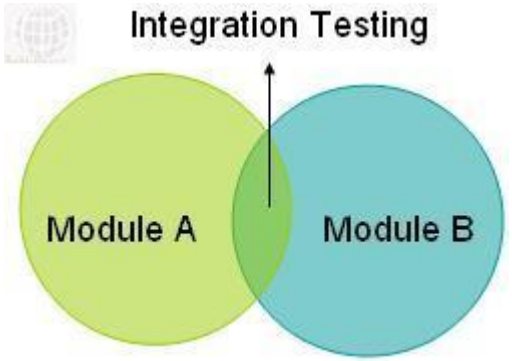
- **Acceptance:** (automated) tests for core functionality
  - Business requirements
  - User story based:
    - As a user I want A thus I do B

- Data driven **delta** test
  - Input and validated output
  - Matches with scientific practice
    - A very good cost trade-off
    - But data changes

Test types: Integration

- Integration of different components
  - Different modules:
    - GUI and backend
    - Server and Client
  - OS and Software
  - Different applications

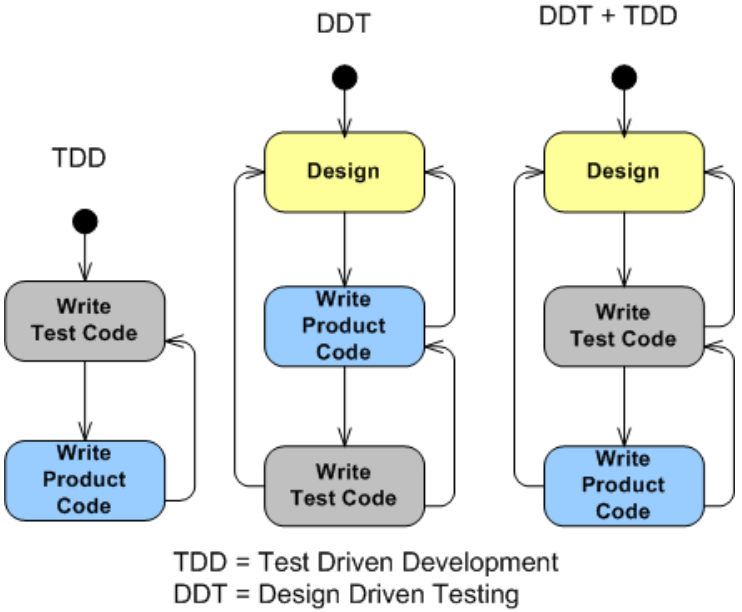
- Performed on the (public) API (Application Programming Interface)



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
  - Run **fast**
- Written during development
- Costly to implement and maintain: long term goal!
- Can be a driver for design and refactoring
  - Test Driven Development



https://en.wikipedia.org/wiki/Unit_testing
http://docs.python-guide.org/en/latest/writing/tests/

Unit tests: xUnit

- De-facto standard for unit testing
  - Implemented in most (all?) programming languages
- test fixture
  - Preparation for on 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 SUT -> teardown

    1. Create files, and dependencies needed to run the component

    2. exercise System Under Test and validate the output :
        - assertxxxxx

    3. Delete used resources

http://pythontesting.net/framework/unittest/unittest-introduction/

Unit tests: Python example

```
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_succes(self):
        self.assertEqual(function(12), 12)

   def teardown(self):
        pass

if __name__ == '__main__':
    unittest.main()
```

```
wouter@WKLIJNWORK:/mnt/c/work$ python3 unittester.py
F.
======================================================================
FAIL: test_something_fail (__main__.TestSomething)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "unittester.py", line 15, in test_fail
    self.assertEqual(function(13), 12)
AssertionError: 13 != 12


----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

Unittest: test assertions

- Functioning is validated using assertions. The most commonly used are:
  - assertEqual(a, b)
  - assertTrue(a)
  - assertIs(a, type)
  - **assertRaises(Exception, function, *args)**
  - assertAlmostEqual(a, b, precision) (float values!!)
  - assertLess(a, b)

Unittests: Practical hints

- Tests are written for other people
  - Also test *obvious* things
- 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
https://blogs.msdn.microsoft.com/micahel/2004/07/07/did-i-remember-to/
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

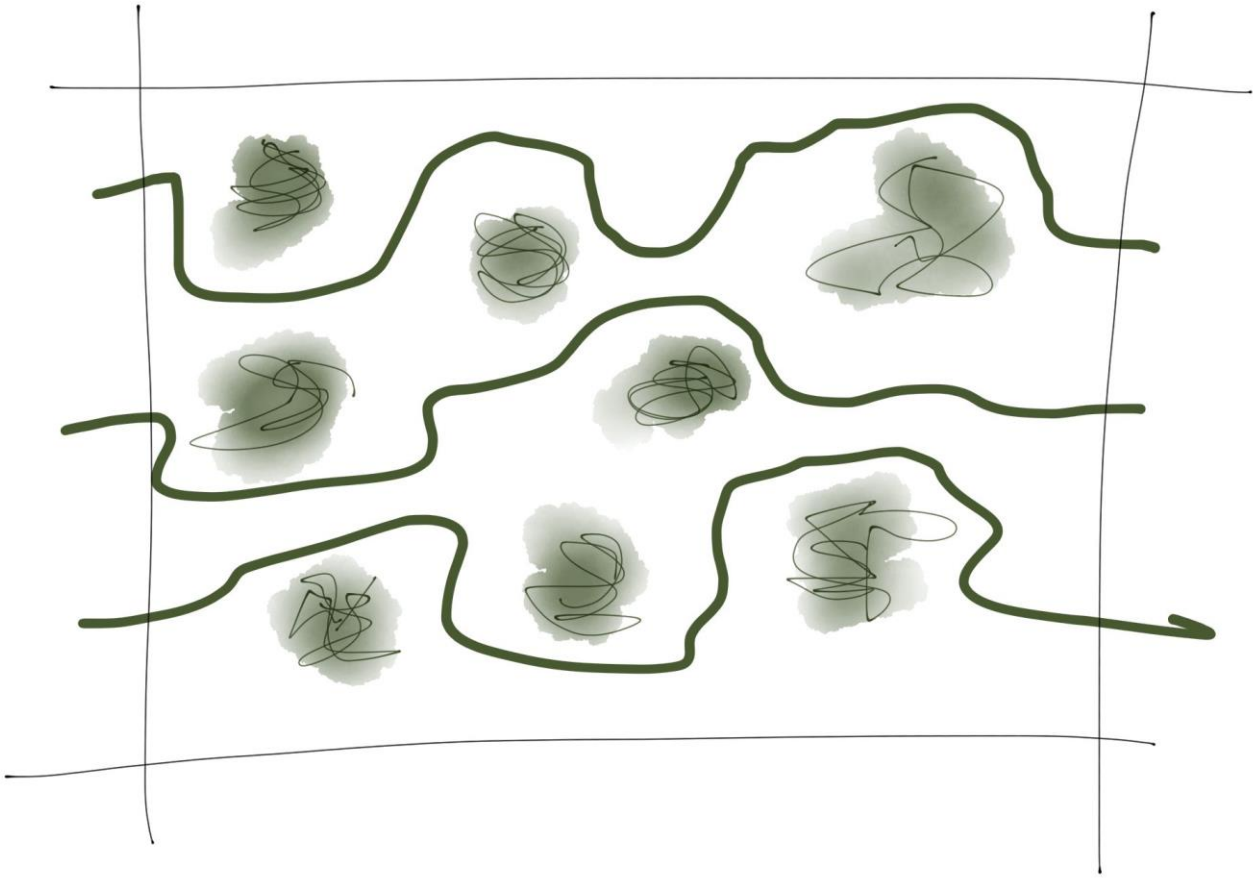
- Pick a single critical **component** and disconnect it from the rest, possible targets:
  - ‘Publishable’ function
  - Often changed code
  - Complicated / scary code
  - Code with lots of errors

- **Unit test** for small parts of the code that do one and only one thing

- Adding test is often leads to refactoring
  - “Legacy software is any code without tests”

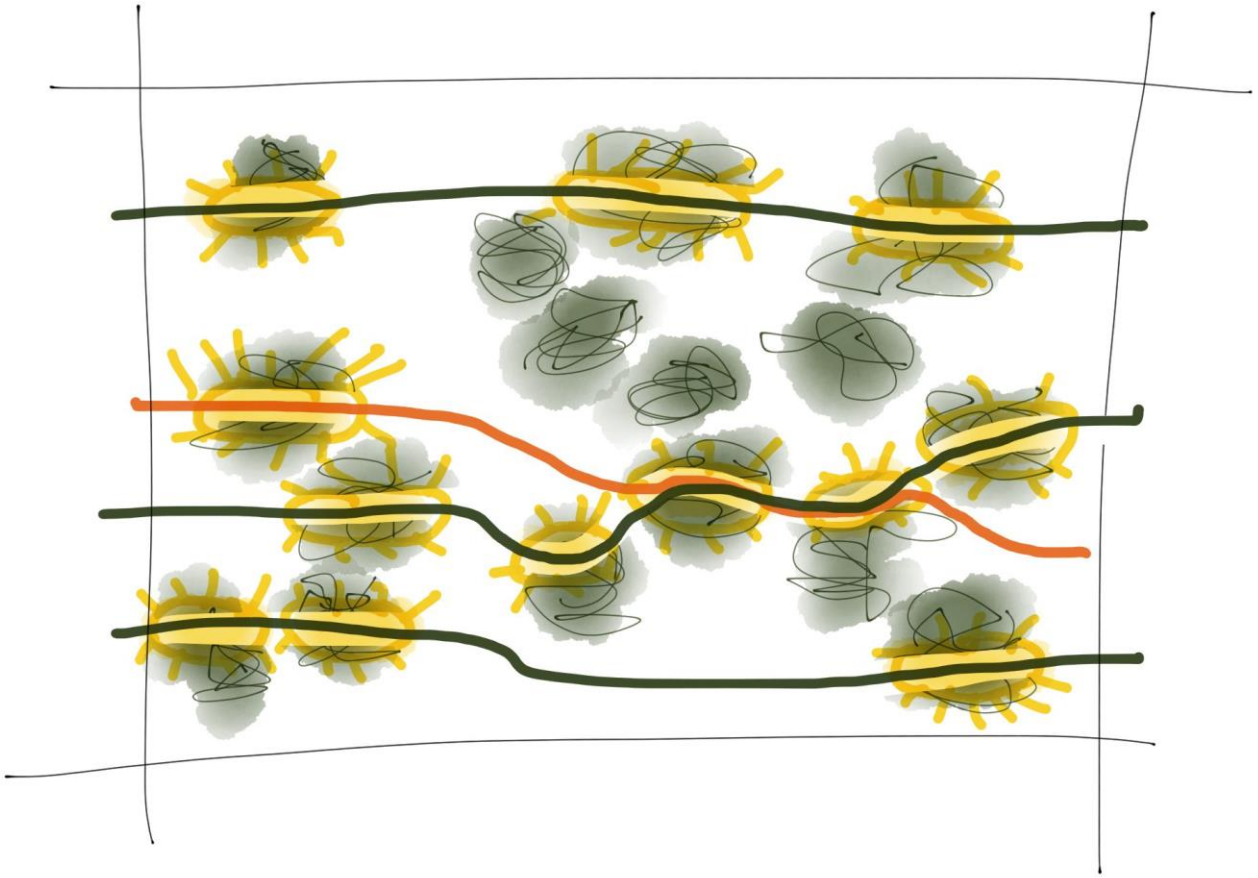
https://www.ethode.com/blog/fixing-spaghetti-how-to-work-with-legacy-code
https://en.wikipedia.org/wiki/Legacy_code

Testing and refactoring



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

Testing and refactoring



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

http://coverage.readthedocs.io/en/latest/

https://www.sonarqube.org/

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