Principles of Good Unit Tests

Dr. Rodrigo Spínola

Context

- Testing is a paramount activity in ensuring software quality
- **Automated tests are safety nets when modifying production code** WHY?
  - however their quality is usually taken for granted or overlooked
- Two main goals:
  - Understand the importance behind writing good unit tests.
  - Know good practices that can be applied to improve the quality of unit tests.

Today

Introduction to Unit Tests	Principles of Good Unit Testing


How Good Are My Tests?

Auxiliary Material: D. Bowes, T. Hall, J. Petric, T. Shippey and B. Turhan, "How Good Are My Tests?," 2017 IEEE/ACM 8th Workshop on Emerging Trends in Software Metrics (WETSoM), 2017, pp. 9-14, doi: 10.1109/WETSoM.2017.2.

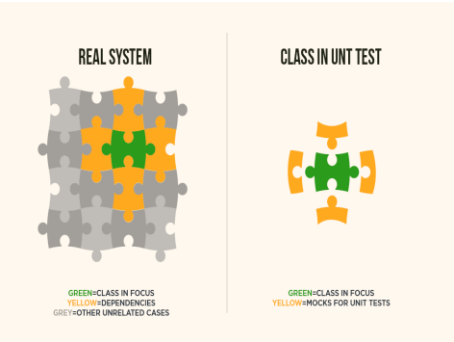
Unit Testing

class, methods

Unit testing is a software testing method by which individual units of source code are tested to determine whether they are fit for use (Kolawa and Huizinga, 2007).

Unit testing is an important process for improving the quality of software systems.

It helps ensure that production code is robust under many usage conditions.



A simple example in Python

- Before you dive into writing tests, you'll want to first decide:
  - What do you want to test?
- Then the structure of a test should follow this workflow:
  1. Create your inputs
  2. Execute the code being tested, capturing the output
  3. Compare the output with an expected result

5

A simple example in Python

- **We're testing sum()**
- There are many behaviors in sum() we could check:
  - Can it sum a list of whole numbers (integers)?
  - Can it sum a tuple or set?
  - Can it sum a list of floats?
  - What happens when we provide it with a bad value, such as a single integer or a string?
  - What happens when one of the values is negative?

6

A simple example in Python

- The simplest test would be a list of integers

```python
import unittest

from my_sum import sum


class TestSum(unittest.TestCase):
    def test_list_int(self):
        """
        Test that it can sum a list of integers
        """
        data = [1, 2, 3]
        result = sum(data)
        self.assertEqual(result, 6)

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

Defines a new test case class called TestSum, which inherits from unittest.TestCase

Defines a test method, .test_list_int(), to test a list of integers. The method .test_list_int() will:

- Declare a variable data with a list of numbers (1, 2, 3)
- Assign the result of my_sum.sum(data) to a result variable
- Assert that the value of result equals 6 by using the .assertEqual() method on the unittest.TestCase class

Defines a command-line entry point, which runs the unittest test-runner .main()

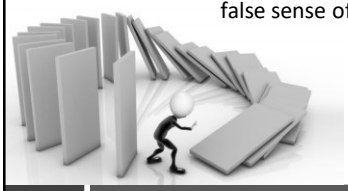
Importance behind Writing Good Unit Tests

- Writing effective tests is as challenging as writing good production code
- Like production code, test code must be maintained and evolved

- As testing has become more commonplace, improving the quality of test code can help improve the quality of the associated production code…

Importance behind Writing Good Unit Tests

- There is evidence that test code is not always of high quality (Zaidman et al., 2008)(Athanasiou et al., 2014)
  - Vahabzadeh and Mesbah (2015) showed that half of the projects they investigated had bugs in the test code
    - Creating false alarms that can waste developer time;
    - Causing tests to miss important bugs in production code, creating a false sense of security.



Importance behind Writing Good Unit Tests

- It is common reviewers ask questions to better understand test code:
  - "Why do you need this for?"
  - "What does this variable name mean?"
- Test code requires improvements, including:
  - suggestions to use better coding practices;
  - fix typos;
  - write better Java-docs;
  - improve code readability;
  - test coverage.
- Current bug detection tools are also not tailored to detect test bugs, making the role of writing good unit tests even more critical.

Pursuing Good Unit Tests

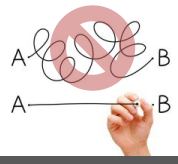
Reviewing test code	**Following principles of good unit test code**
Testing test code	Others...

Knowing them in the first place is important to avoid several common mistakes when writing unit tests.

Simplicity

- As we depend on tests as a safety net, we must rely on their validity and ensure their maintainability through simplicity
  - There is less chance of making mistakes and it is easier to maintain test base when we avoid complexity.

- Violation of this principle may be observed in test code in terms of the:
  - size of test code
  - number of assertions per test case, and
  - conditional test logic.

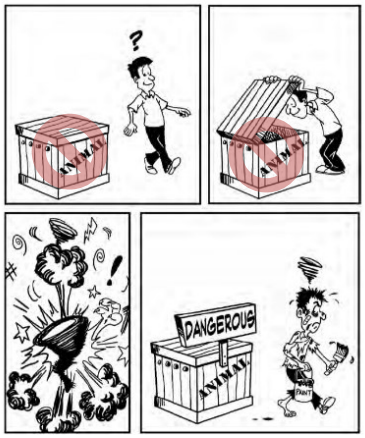


Keep it simple, keep it safe.

13

Readability and Comprehension

- Unit tests are self-documenting instruments;
  - the expectation is to have a clear understanding of the intentions of the test code.
- Besides being simple, expressiveness of test is fundamental;
- Using magic numbers, branching, and inexpressive naming conventions would disrupt this principle.



14

Single Responsibility

- A test should have only a single reason to fail
  - when a test fails, one should be able to locate the root cause of the problem.
- Calls to multiple methods in class under test indicate that we are testing more than we should in a single test case.

Example

```
@Test
public void test1()
{
    String[] countries = { "Canada", "AUSTRALIA", "spain", "Chad", "JaPaN" };
    Sorter sorter = new Sorter();
    CaseModifier caseModifier = new CaseModifier();
    String[] expected = { "AUSTRALIA", "CANADA", "CHAD", "JAPAN", "SPAIN" };

    String[] result = sorter.sort(caseModifier.toUpper(countries));

    Assert.assertArrayEquals(expected, result);
}
```

Readability and Comprehension

Single Responsibility

- There are two problems with this unit testing:
  - The name is too vague – what is the unit test actually about?
  - What specifically is being tested – the sorter or the case modifier?

Example

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

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

- Too many assertions!

Maintainability

- As production code evolves, it is inevitable that test code will evolve with it;
- Tests should be easily maintainable
  - For example: duplication of test code is a common practice to have a fast start when writing a new test case. However, it would make maintainability harder in the long term, unless test code is refactored.

Happy vs Sad Tests

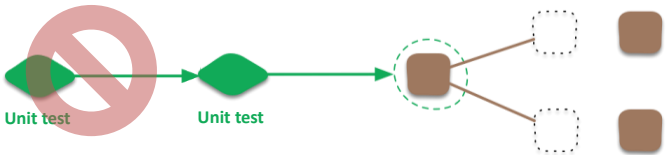
Due to confirmation bias, it is common to see testers' tendency to confirm behavior rather than to break it.

To verify the system (happy tests)
vs
To break the system (sad tests)

Considering testing strategies such as partitioning strategy and boundary case analysis, we expect at least as many sad test cases as happy test cases.

Test (in)dependency

- We should be able to run our tests in any order and in isolation;
- Tests should not rely on each other in anyway;
- This allows us to add new test cases without considering any dependencies or any effects they might have on existing test cases.



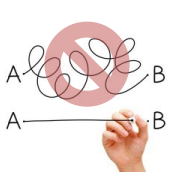
Tests should not Dictate the Code

- Keep test logic out of production code!
- We should not make any modifications in the production code for the purpose of testing
  - for example: to include a production code method that is only called from within the test code to access internal states of the class under test

21

Summary

Simplicity	Readability and Comprehension	Single Responsibility	Maintainability	Happy vs Sad Tests	Test (in)dependency	Tests should not Dictate the Code



TDresearchteam
Technical Debt Research Team

VCU Computer Science
College of Engineering

Additional Reading

- D. Bowes, T. Hall, J. Petric, T. Shippey and B. Turhan, "How Good Are My Tests?," 2017 IEEE/ACM 8th Workshop on Emerging Trends in Software Metrics, 2017, pp. 9-14.
- Ayaan M. Kazerouni, James C. Davis, Arinjoy Basak, Clifford A. Shaffer, Francisco Servant, Stephen H. Edwards. Fast and accurate incremental feedback for students' software tests using selective mutation analysis, Journal of Systems and Software, Volume 175, 2021, 110905, ISSN 0164-1212.
- Lucinéia Souza, Sávio Freire, Verusca Rocha, Nicolli Rios, Rodrigo Spínola and Manoel Mendonça. 2020. Using Surveys to Build up Empirical Evidence on Test Related Technical Debt. In XXXIV Brazilian Symposium on Software Engineering, 2020, Natal, Brazil.



Principles of Good Unit Tests

Dr. Rodrigo Spínola

> Lecture 18 - Principles of Good Unit Tests

TDresearchteam
Technical Debt Research Team

VCU Computer Science College of Engineering