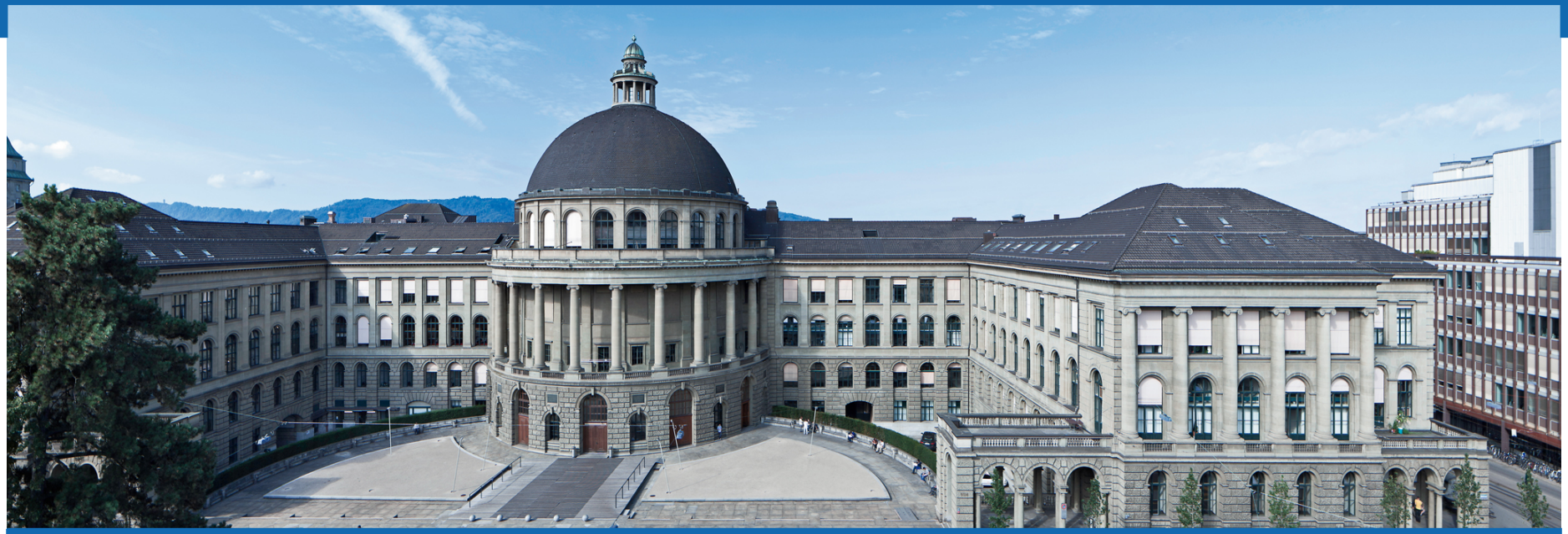
ETH zürich



Unit Testing

Manuel Weberndorfer

Introduction

Test Taxonomy

Tests have different scope.

- Unit Testing
- Integration Testing
- …

Test Taxonomy II

Tests have different goals.

- Acceptance Testing
- Regression Testing
- …

Unit Testing

- Testing by developers for developers
- Code that tests code
- Automatic checks of what you know about code

Example Code

solvers.py:

```
def solve_linear_equation(k, d):
    """Solves k * x + d = 0 for x."""
    return -d / k
```

Example Test

test_solve_linear_equation.py:

```
from solvers import solve_linear_equation


def test_solves_general_linear_equation():
    """Checks solution for k = 2 and d = 5."""
    assert solve_linear_equation(k=2, d=5) == -2.5
```

Run the Test

No output, everything OK (Python 3).

```
>>> import test_solve_linear_equation
>>> test_solve_linear_equation.test_(...)_equation()
>>>
```

Run the Test Again

Not OK (Python 2.7).

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "test_solve_linear_equation.py", line 5, (...)
    assert solve_linear_equation(k=2, d=5) == -2.5
AssertionError
```

Lesson Learned

Software development is complex.

- Requirements change
  - Code used differently
  - Used code changes
- Rechecking manually not feasible

Beyoncé Rule

If you liked it then you should have put a test on it.

How to Unit Test

xUnit Frameworks

- Nicer output
- More checks
  - Float comparison
  - Expected exceptions
  - …

Test Runners

- Find tests (based on pattern)
- Execute them
- Create a report (e.g. in XML)

Remember Solver

solvers.py:

```
def solve_linear_equation(k, d):
  """Solves k * x + d = 0 for x."""
  return -d / k
```

TestCase

test_solve_linear_equation.py:

```
from unittest import TestCase
from solvers import solve_linear_equation


class SolversTest(TestCase):

  def test_solves_general_linear_equation(self):
    """Checks solution for k = 2 and d = 5."""
    result = solve_linear_equation(k=2, d=5)
    self.assertEqual(result, -2.5)
```

Run Tests

Start runner: `python -m unittest discover`

```
======================================================
FAIL: test_solves_general_linear_equation (...)
Checks solution for k = 2 and d = 5.
------------------------------------------------------
Traceback (most recent call last):
  File "(...)", line 8, in test_(...)_equation
    self.assertEqual(result, -2.5)
AssertionError: -3 != -2.5
------------------------------------------------------
Ran 1 test in 0.000s
```

Fix Tests

Adapt for floating point numbers.

```
def test_solves_general_linear_equation(self):
  """Checks solution for k = 2 and d = 5."""
  result = solve_linear_equation(k=2., d=5.)
  self.assertAlmostEqual(result, -2.5)
```

Check Result

Restart runner: `python -m unittest discover`

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Special Cases

Add test to `TestCase`.

```
def test_throws_for_gradient_zero(self):
  """Solve throws exception for gradient zero."""
  with self.assertRaises(ZeroDivisionError):
    solve_linear_equation(k=0, d=5)
```

Check Result

Restart runner: `python -m unittest discover`

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Next Steps

- Fix behaviour for integers
- What happens when `k=0` and `d=0`?
- …

Result

Verified Documentation.

```
class SolversTest(TestCase):

  def test_solves_general_linear_equation(self):
    (...)
    result = solve_linear_equation(k = 2., d = 5.)
    (...)

  def test_throws_for_gradient_zero(self):
    (...)

  def test_throws_for_gradient_and_offset_zero(self):
    (...)
```

Know the Framework

Use suitable assertions.

- `assertEqual` instead of `assertTrue(a == b)`
- `assertAlmostEquals` for floats
- many more... (see documentation)

Best Practices

Good Tests

Write your tests *first*.

- *F* ast
- *I* solated
- *R* epeatable
- *S* elf-validating
- *T* imely

Fast

Prepare for 1000s of tests.

- Maximum: few milliseconds
- You should run all the tests often
- You will write many of them

Isolated

Minimal maintenance.

- Independent of each other
- Independent of behaviour tested elsewhere
- Independent of machine
- Independent of external data/network

Repeatable

Rerun to locate bugs.

- Same result every time you run it
- No random input
- Independent of environment (network, etc.)

Self-Validating

Do the work once.

- Test determines if result was expected
- No manual work involved
- Remember, you want to run them often

Timely

Write the test as soon as possible.

- You still know why you wrote the code that way
- You know the special cases
- You have (manual) test cases ready

Pattern

Given-When-Then Pattern.

- Given: Setup of environment
- When: Execute what you want to test
- Then: Check result
- (if required): Cleanup

Recall Test

test_solve_linear_equation.py:

```
class SolversTest(TestCase):

  def test_solves_general_linear_equation(self):
    """Checks solution for k = 2 and d = 5."""
    result = solve_linear_equation(k=2., d=5.)   # WHEN
    self.assertAlmostEqual(result, -2.5)         # THEN
```

Help the Reader

That's why I chose `5.` and `2.`.

```
def test_solves_general_linear_equation(self):
  """Checks solution for nonzero gradient, offset."""
  nonzero_gradient = 2.
  nonzero_offset = 5.

  result = solve_linear_equation(nonzero_gradient,
                                 nonzero_offset)

  self.assertAlmostEqual(result, -2.5)
```

Repeat

We could do the same thing here.

```
def test_throws_for_gradient_zero(self):
  """Solve throws exception for gradient zero."""
  with self.assertRaises(ZeroDivisionError):
    solve_linear_equation(k=0, d=5)
```

Fixtures

We use a nontrivial test fixture.

```
class SolversTest(TestCase):

  def setUp(self):
    self.nonzero_gradient = 2.
    self.nonzero_offset = 5.

  def test_solves_general_linear_equation(self):
    (...)
```

Rollout

No need to copy and paste.

```
def test_throws_for_gradient_zero(self):
  """Solve throws exception for gradient zero."""
  with self.assertRaises(ZeroDivisionError):
    solve_linear_equation(k=0, d=self.nonzero_offset)
```

Set the Stage

Use Fixtures.

- Provide data for the tests
- Initialize classes with nontrivial constructors
- Provide convenience methods
- Name a collection of tests

Real-World Testing

Challenges

You will be facing common challenges.

- I can’t test hidden/private code.
- I can’t test every combination of inputs.
- I don’t understand what a function does.
- The function has 20 parameters, so 2ˆ20 tests?
- The function needs to read a file.

Private Code

I can't test hidden/private code.

No problem, everybody else can't, either.

- Test what users can do with your code
- Extract code if this makes testing difficult

Lean Testing

I can't test every combination of inputs.

Test until you are confident that your code works.

- Test the uses cases the unit was designed for
- Test 'boundaries' and special cases
- Add tests when you discover bugs
- 'Code coverage' tools can help

No Documentation

I don't understand what a function does.

No problem, use unit testing.

- Write a test that checks any assumption
- Look at the actual output
- Fix the test
- Repeat

Complex Interface

The function has 20 parameters.

Unit testing tests atomic units.

- Add regression tests first
- Try to extract atomic units and test them
  - 20 one-boolean-parameter-functions: 40 tests
  - One 20-boolean-parameter-function: ~1M tests
- Add integration tests for the (important) use cases

Interoperation

The function needs to read a file.

```
def bad_function():
  with open("data.txt", "r") as file:
    x = file.read()
    return x + x
```

No problem, there are tools for that: ‘Mocking’

Mocking

Choose a return value yourself (Python 3).

```
from unittest import TestCase, mock
from bad_function import bad_function


class MockTest(TestCase):

  def test_returns_read_string_twice(self):
    """bad_function returns read string twice."""
    with mock.patch('builtins.open',
                    mock.mock_open(read_data='abc')):
      self.assertEqual(bad_function(), 'abcabc')
```

Other languages

Unit Testing is not language-specific.

- Principles apply to software engineering in general
- Unit testing frameworks (e.g.)
  - Python: `unittest`
  - R: `test_that`
  - C++: `googletest`
  - Java: `JUnit`

Conclusion

Benefits

Correctness and more.

- Check new code for bugs
- Check old code for new bugs after modifications
- Documentation
- Tested/able code is often good code
  - Small units (simpler, reuseable)
  - Special cases covered
  - Easy to improve later

Questions?

Acknowledgements

- This talk is based on a talk by Uwe Schmitt.