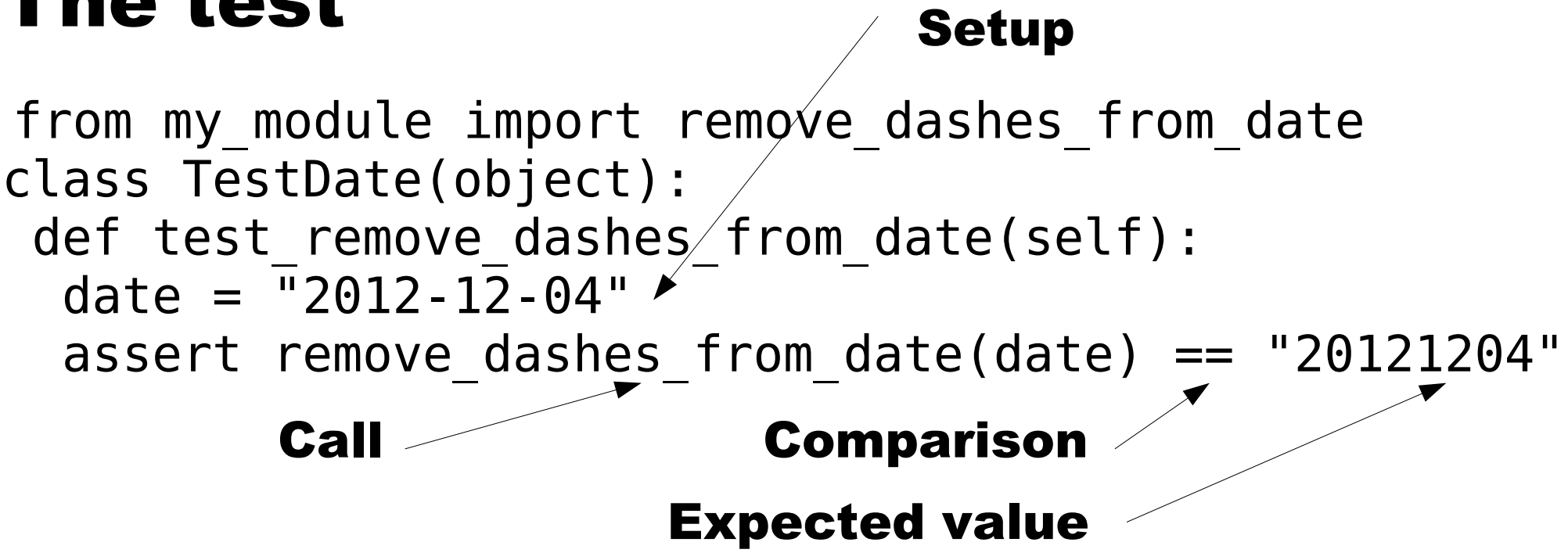
Openend

Jacob Hallén

Testing for Beginners

Europython 2013

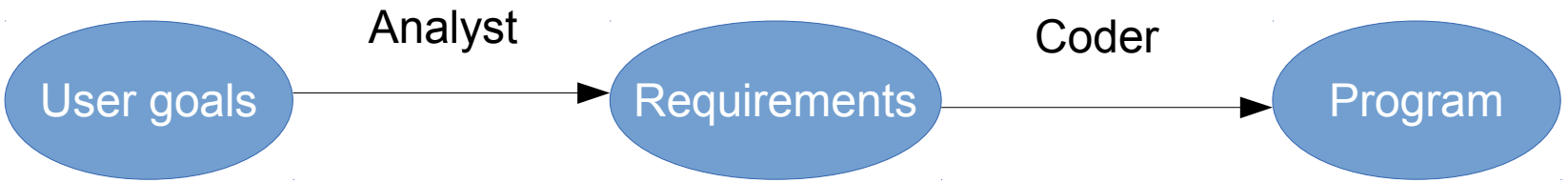
The test



The code

```
def remove_dashes_from_date(date):
    date_list = date.split("-")
    return ''.join(date_list)
```

Development process



Unit tests
Integration tests

System tests

Acceptance tests
Deployment tests

Unit testing

- Also known as ***component testing***, refers to tests that verify the functionality of a specific section of code, usually at the function level. In an object-oriented environment, this is usually at the class level, and the minimal unit tests include the constructors and destructors.

Integration testing

- Any type of software testing that seeks to verify the ***interfaces between components*** against a software design.
- We put two or more components together and test that they interoperate as intended.
- Two schools – Big Bang and Incremental

System testing

- Tests a completely integrated system to verify that it meets its requirements.
- There are testing frameworks, but often you have to build your own testing harness.

Acceptance testing

- Determines if the software fulfills the user goals.
  - May be manual
  - May be a set of tests agreed on before development started
  - May include tests for regulatory compliance

Deployment testing

- Ensures that the software works in its intended production environment
  - Availability of all prerequsites
  - Compatibility with library/OS versions
  - Resource consumption
  - Collision with other components

Tools for unit testing

- PyUnit – part of Cpython
  - Simple, explicit

http://docs.python.org/2/library/unittest.html

- py.test – from merlinux, free software
  - Advanced gathering of test suites
  - More support for complex setups
  - Better diagnostics

http://pytest.org/

- nosetest
  - Same as py.test, but different

https://nose.readthedocs.org/

PyUnit

```
import unittest

from my_module import remove_dashes_from_date

class TestDate(unittest.TestCase):
 def test_remove_dashes_from_date(self):
  date = "2012-12-04"
  self.assertEqual(remove_dashes_from_date(date),
  "20121204")
```

Why are we (unit) testing?

Why are we testing?

- To formulate an idea of what the unit should do
- To check that we do what the tests asks
  - While we develop
  - Always in the future – prevent regressions
- To make the program testable
  - Changes coding style!

Why are we testing?

- To eliminate bug sources
  - Code coverage
  - With enough tests, all bugs are shallow
- To help us understand our code at a later stage
  - Refactoring
  - Without tests, you are not refactoring. You are mucking about in your code.
- To know when we are done
  - When your tests pass, you are fulfilling the specifications, as they are understood right now.

When do we run our tests?

When do we run our tests?

- To see if the code we are working on passes
  - During development – may be partial
- Before check-in
  - Full test suite, if possible
- Atomatically on each push
  - Full test suite
- Nightly
  - Full test suite

What are we testing?

What are we testing?

- Storing information
  - **Test:** Store some information, retreive it and check that it is the same
- Retreiving information
  - **Test:** Store some information, retreive it and check that it is the same
- Calculating stuff
  - **Test:** Make the calculation and compare with the expected result

Continued

The test

Storing

```
import pytest
class TestX(object):
    def test_setter(self):
        val = "A text"
        obj = X(val)
        assert obj.value == val
```

The code

```
class X(object):
    def __init__(self, value):
        self.value = value
    def get_value(self):
        return self.value
```

Retreiving

The test

```
import pytest
class TestX(object):
    def test_getter(self):
        val = "A text"
         obj = X(val)
        assert obj.get_value() == val
```

The code

```
class X(object):
    def __init__(self, value):
        self.value = value
    def get_value(self):
        return self.value
```

The test

Calculating

```
import pytest
class TestDate(object):
 def test_remove_dashes_from_date(self):
  date = "2012-12-04"
  assert remove_dashes_from_date(date) == "20121204"
```

The code

```
def remove_dashes_from_date(date):
    date_list = date.split("-")
    return ''.join(date_list)
```

What are we testing?

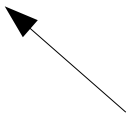
- Input
  - This includes network connections, databases, user access and input from 3rd party libraries
  - **Test**: Put some known input on the channel, Check that it gets into the program in the expected form.
- Output
  - Network connections, user output, 3rd party libraries
  - **Test:** Check that your output is what you expect. This is often hard!

```
import pytest, sqlite3
from stocks import get_stock

@pytest.fixture
def db():
  conn = sqlite3.connect('example.db')
  c = conn.cursor()
  c.execute('''CREATE TABLE stocks
    (name text, date text, trans text, ...)''')
  c.execute('''INSERT INTO stocks VALUES
    ('MSFT', '2006-01-05','BUY', ...)''')
  conn.commit()
  yield conn
  # Teardown code goes here

class TestDB(object):
  def test_get_stock(self, db):
    assert get_stock(db, 'MSFT')['date'] == '2006-01-05'
```

Input



Known data

Output

0. Write expected result by hand
1. Make call to generate file
2. Open file
3. Open file with expected result
4. Compare contents
5. If equal, test passed
6. If not equal, show diff

If the expected result changes, we have to modify the expected results

Combinations

- Retreive – store
  - **Test**: Store something, run code, check store
- Retreive – calculate – store
  - **Test:** Store something, run code, check store
- Input – calculate – store
  - **Test:** Known data, run code, check store
- Retreive – calculate – output
  - **Test:** Store something, run code, check output
  - Doubly hard

Avoiding combinations

- Refactor your code
  - Break out calculations
  - Split out input/output
  - Separate retreiving and storing
- Test output with simple, unchanging data
  - Test the data that goes to the output before it gets there

Dealing with things that are hard to test

Mocking input

```
class MockDB(object):
    def cursor(self):
        return self

    def execute(self, sql_str):
        return [['MSFT',
                 '2006-01-05','BUY', ...]]
```

Mocking output

```
class MockSocket(object):
    def connect(self, *args):
        self.buffer = StringIO()
        return self

    def write(self, arg):
        self.buffer.write(arg)
```

You can now use Mocksocket.buffer.getvalue() to check what you sent to the socket.

Tests are samples

- A general case
- Edge cases – all of them
- Exceptions – all explicit ones, important implicit ones

```
with pytest.raises(KeyError):
    my_func('xyzzy')
```

How much do I need to test?

- Untested code is a technical debt
  - Technical debt accrues interest
- The amount of interest depends on many factors
  - Longevity of the code
  - Need for change
  - Criticality of the code
  - Number of users

The testing technical debt is all the extra time and costs you have to spend in the future because you didn't write tests when you wrote the code.

If you can't do full coverage, test the most primitive parts of your system.

jacob@openend.se
http://www.openend.se

We can help you getting started with testing.

Mentoring, peer programming, code review, tools, infrastructure.