Tutorial: Software testing in scientific programming

David W.H. Swenson
E-CAM Extended Software Development Workshop
Lyon, France; 3 April 2019

Talk materials: https://gitlab.e-cam2020.eu/dwhswenson/software_testing

Funded by the EU (Project n°676531)

Outline

- Overview
- Terminology
- xUnit-style testing
- Hands-on with the pytest package and coverage

Outline

- **Overview**
- Terminology
- xUnit-style testing
- Hands-on with the pytest package and coverage

Why should you write tests?

- Um, so your code has fewer bugs? (Duh)
- Testable code is better-designed code
- Don't break things that do work (see regression testing)
- The tests *are* the API

Why don't you write tests?

- Because you're lazy

Why don't you write tests?

- Because you're in a rush to get to publication, and writing tests takes time
- Because scientific software is complicated, and so it is hard to write simple tests for it

Outline

- Overview
- **Terminology**
- xUnit-style testing
- Hands-on with the pytest package and coverage

What we'll cover

- **dynamic testing**: runs the underlying code
- **functional testing**: check that inputs/outputs are as expected
- **unit (component) testing**, **integration testing**, **system testing**
- **black box** and **white box** (**structure-based**) approaches
- xUnit-style testing (with pytest)

What we won't cover

- **static testing**: e.g., reviews of code quality
- **non-functional testing**: e.g., performance
- **acceptance testing**
- details of designing test cases
- test management / planning
- practical aspects on non-xUnit approaches

Regression Testing and Continuous Integration

- "**Regression tests**:" Run tests after every change. Goal: Don't break anything.

- "**Continuous integration**:" Merge branches to master after every change. Goal: Don't let code diverge too far.

Closely related concepts. Once you open a PR, we attempt to merge your changes to master (CI). If that is successful, we run our test suite (RT).

Example Travis-CI Script

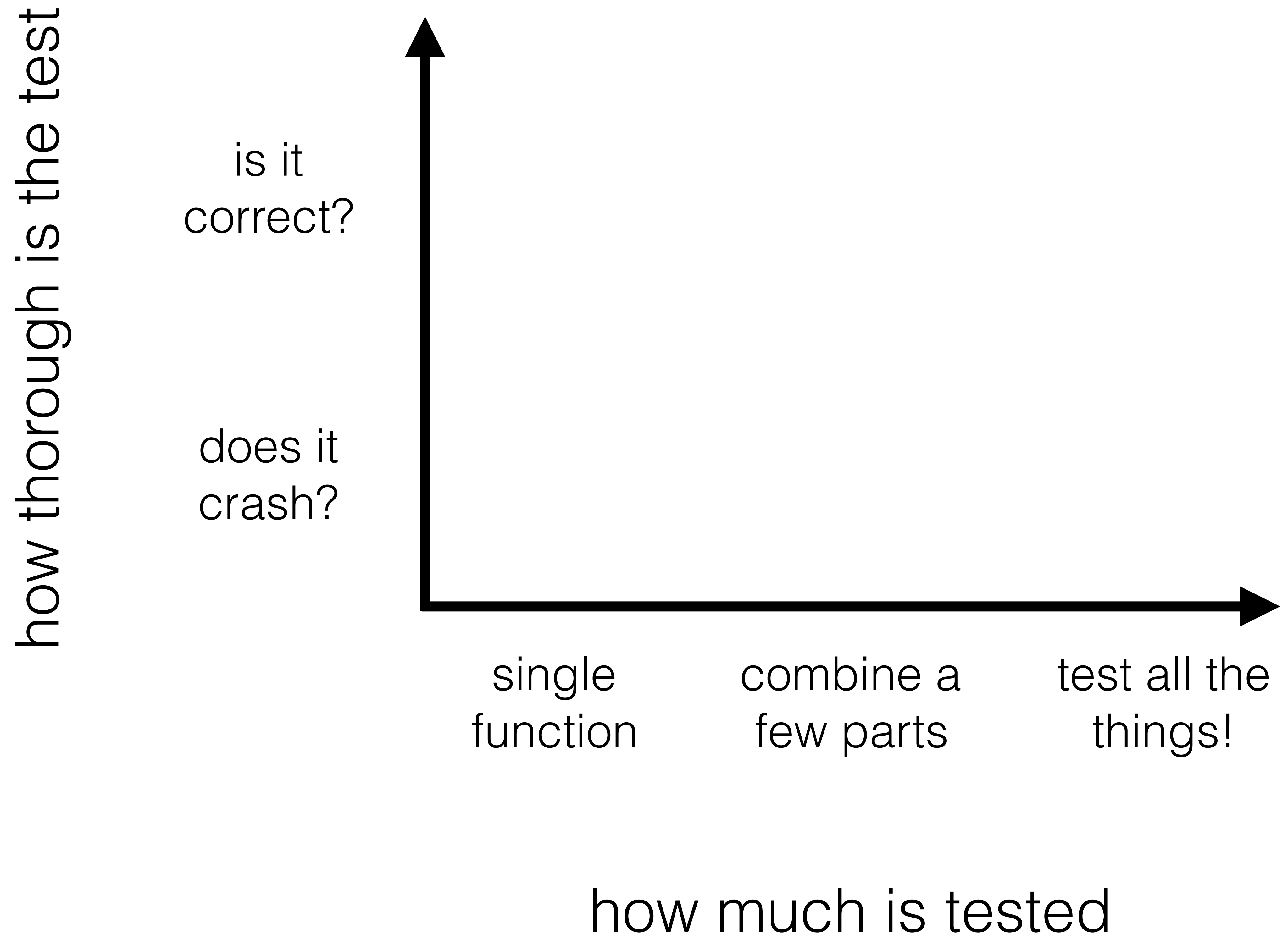
```yaml
language: python

env:
  matrix:
    - CONDA_PY=2.7
    - CONDA_PY=3.7

before_install:
  - deactivate
  - source ci/miniconda_install.sh  # uses $CONDA_PY

install:
  - conda install -y -c conda-forge --file requirements.txt
  - python setup.py install

script:
  - which python
  - export MPLBACKEND=PS
  - python -c "import my_package"
  - conda install -y -c conda-forge --file test_requirements.txt
  - py.test -vv --cov=my_package --cov-report xml:cov.xml

after_success:
  - coveralls
```

Test Driven Development

*Write the tests **FIRST**!*

1. Add a test
2. Run tests to see if new test fails
3. Write the code
4. Run tests
5. Refactor

Terminology



Terminology

how thorough is the test

	unit test	integration test	system test
test	do this!	do this!	impossible (do partially)
smoke test	lame	do this! (e.g., plots)	as much as possible

how much is tested

True Unit Tests

- Test only a small part of the code
- Each test should be independent!

```
def square_it(number):
    return number * number

def first_n_squares(n):
    return [square_it(i+1)
            for i in range(n)]
```

```
def test_square_it():
    assert_equal(square_it(1), 1)
    assert_equal(square_it(2), 4)
    assert_equal(square_it(8), 64)

def test_first_n_squares():
    # without running square_it?
```

In practice, often use small integration tests (not independent) in place of true unit tests

Mocks / Stubs

- How do you write a unit test, if your function depends on other functions?

- Interfacing with an MD engine: How to test your code, without running the real MD engine?

- Answer: create something that acts like the engine, but is faster/predictable! A "mock engine"

- Run a *separate* test specifically to check engine interface

- There are fancy tools for mocks, but toy subclasses also work

Outline

- Overview
- Terminology
- **xUnit-style testing**
- Hands-on with the pytest package and coverage

xUnit-Style Testing

- Frameworks in many languages (SUnit, JUnit, RUnit, etc.)
- Test runner (identifies tests and runs them)
- Test fixtures (setup and teardown)
- Use assertions to verify correctness

xUnit frameworks for common scientific programming languages	
	Python: pytest, unittest (standard library)
	C++: Google Test, Boost Test Library
	Fortran: pFUnit, FRUIT

Autodiscovery of Tests

Preface stuff with `test` or `Test` and `pytest` will automatically find the tests and run them.

```
# in test_module.py

def test_function():
    # bare test function
    pass

class TestClass(object):  # can inherit from object or UnitTest
    def test_method(self):
        # test method within class
        pass
```

Test Structure Mirrors Code

```python
# in file module.py
# this is the main code


def some_function():
    pass

class MyClass(object):
    def method(self):
        pass
```

```python
# in file test_module.py
# tests for module.py
from module import *

def test_some_function():
    pass

class TestMyClass(object):
    def test_method(self):
        pass
```

ISTQB: “bi-directional traceability between the tests and the test basis”

Fixtures

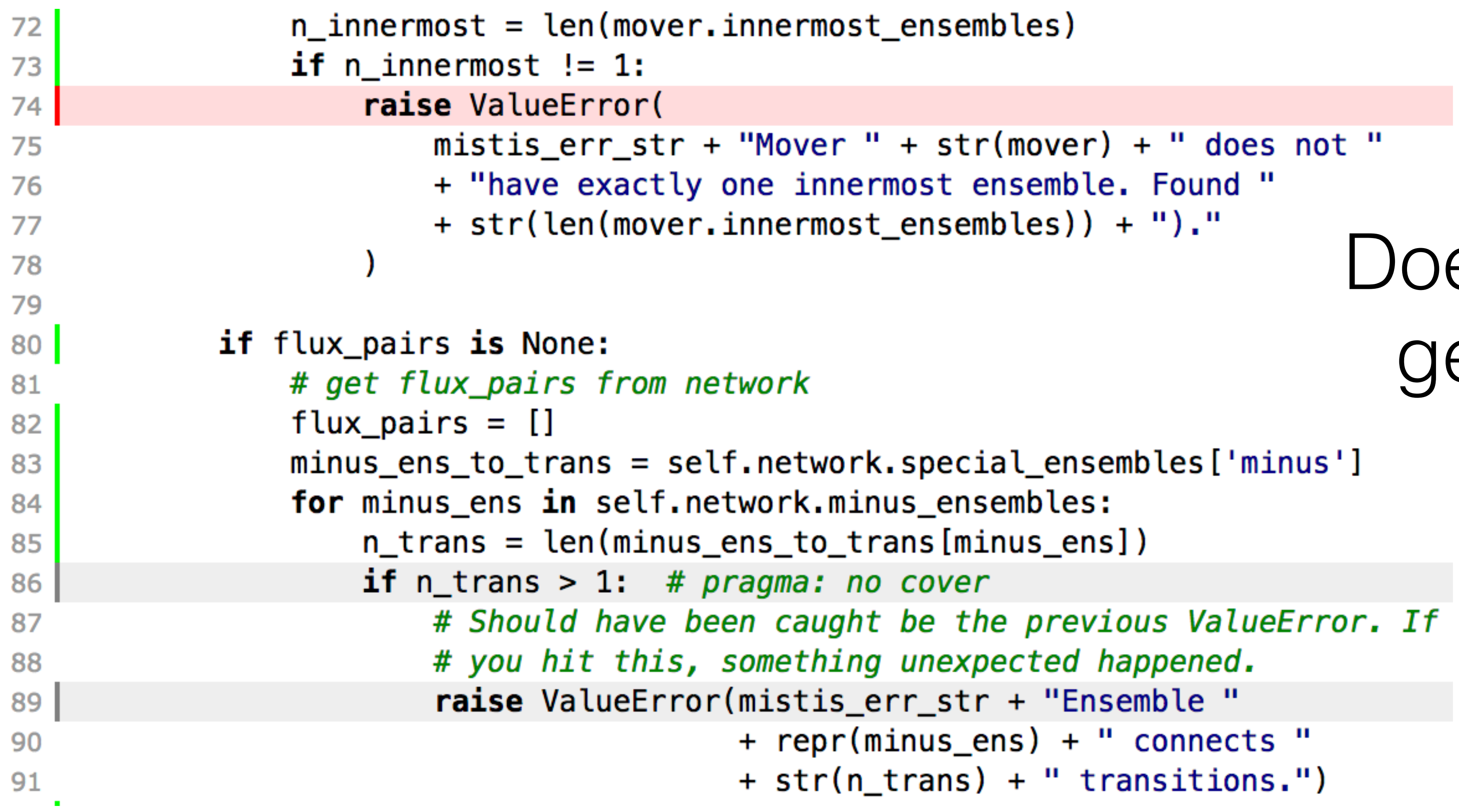
Some parts of tests (especially setup) are often repeated

Fixtures allow you to group this boilerplate code together

```python
def setup_module():
    print "Module-level setup"

def teardown_module():
    print "Module-level teardown"

class TestClass(object):
    def setup(self):
        print "  Class-level setup"

    def teardown(self):
        print "  Class-level teardown"
```

Coverage

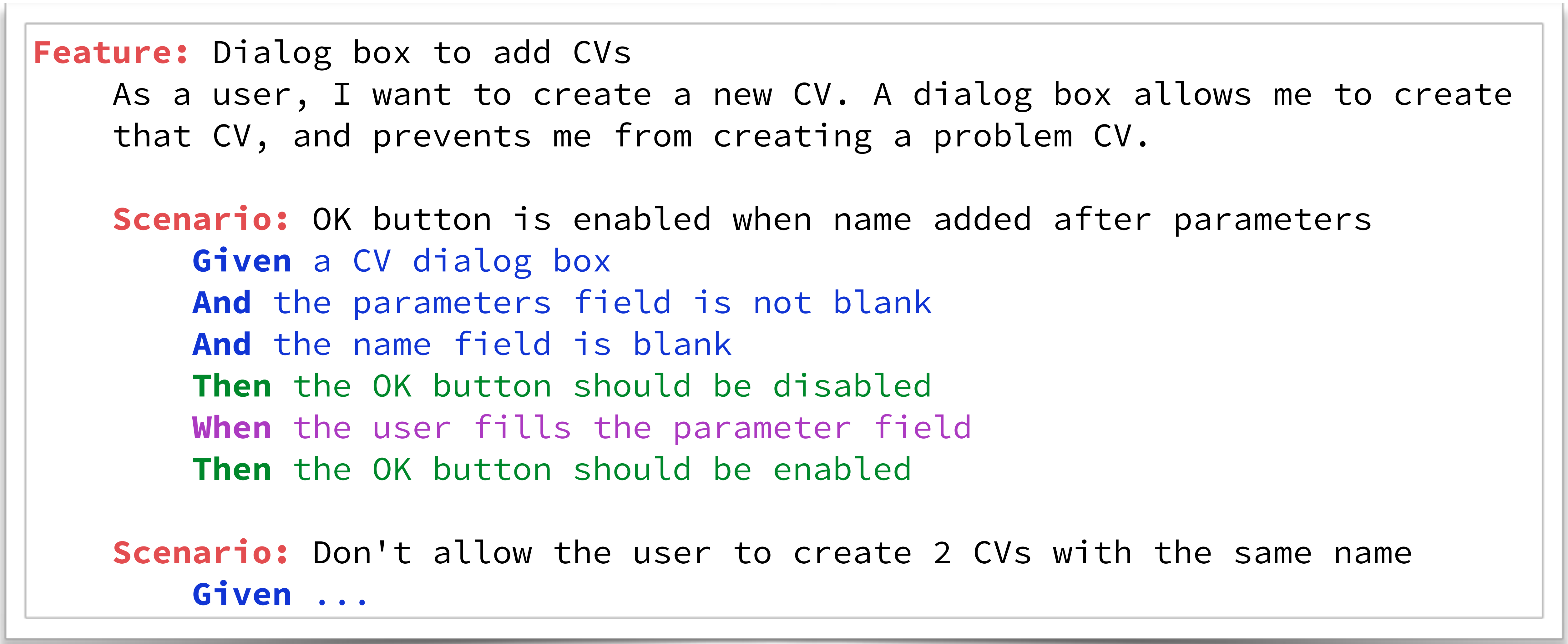
Good testing requires quality (good tests)
and quantity (test all the code)



```
        n_innermost = len(mover.innermost_ensembles)
        if n_innermost != 1:
            raise ValueError(
                mistis_err_str + "Mover " + str(mover) + " does not "
                + "have exactly one innermost ensemble. Found "
                + str(len(mover.innermost_ensembles)) + ")."
            )

    if flux_pairs is None:
        # get flux_pairs from network
        flux_pairs = []
        minus_ens_to_trans = self.network.special_ensembles['minus']
        for minus_ens in self.network.minus_ensembles:
            n_trans = len(minus_ens_to_trans[minus_ens])
            if n_trans > 1:  # pragma: no cover
                # Should have been caught be the previous ValueError. If
                # you hit this, something unexpected happened.
                raise ValueError(mistis_err_str + "Ensemble "
                                 + repr(minus_ens) + " connects "
                                 + str(n_trans) + " transitions.")
```

Minimal check:
Does every line of code
get run during tests?

Aside: Behavior-Driven Development



```
Feature: Dialog box to add CVs
    As a user, I want to create a new CV. A dialog box allows me to create
    that CV, and prevents me from creating a problem CV.

    Scenario: OK button is enabled when name added after parameters
        Given a CV dialog box
        And the parameters field is not blank
        And the name field is blank
        Then the OK button should be disabled
        When the user fills the parameter field
        Then the OK button should be enabled

    Scenario: Don't allow the user to create 2 CVs with the same name
        Given ...
```

- Human-readable tests!
- Designed for large teams (spec not written by devs)
- Neat, but not necessarily best for science

Outline

- Overview
- Terminology
- xUnit-style testing
- **Hands-on with the pytest package and coverage**

Live Demo!

- Raising exceptions (Jupyter notebook)
- Running pytest
- Handling exceptions with pytest
- Tests and coverage (simple example)
- Tests and coverage (you do it)

Talk materials: https://gitlab.e-cam2020.eu/dwhswenson/software_testing

Funded by the EU (Project n°676531)