Essential Software Engineering for Computational Science and Data Science

Hans Fangohr

2019-06-12

Outline

Software Engineering introduction

Software processes and methodologies

Testing

Test Driven Development

Practial aspects of test driven development

Tools

Summary

Literature

Software Engineering introduction

Software Engineering introduction

Software engineering

is the study and an application of engineering to the design, development and maintenance of software

(`https://en.wikipedia.org/wiki/Software_engineering`)

Including

- requirements capture
- design of software
- implementation
- testing
- verification and validation
- delivery
- maintenance

Software engineering helps to develop working software

Famous accidents I

Ariane Rocket Goes Boom (1996)

Cost: $500 million

Disaster: Ariane 5, Europe's newest unmanned rocket, was intentionally destroyed seconds after launch on its maiden flight. Also destroyed was its cargo of four scientific satellites to study how the Earth's magnetic field interacts with solar winds.

Cause: Shutdown occurred when the guidance computer tried to convert the sideways rocket velocity from 64-bits to a 16-bit format. The number was too big, and an overflow error resulted. When the guidance system shut down, control passed to an identical redundant unit, which also failed because it was running the same algorithm.

`http://www.devtopics.com/20-famous-software-disasters-part-2/`

Famous accidents II

Mars Climate Crasher (1998)

Cost: $125 million

Disaster: After a 286-day journey from Earth, the Mars Climate Orbiter fired its engines to push into orbit around Mars. The engines fired, but the spacecraft fell too far into the planet's atmosphere, likely causing it to crash on Mars.

Cause: The software that controlled the Orbiter thrusters used imperial units (pounds of force), rather than metric units (Newtons) as specified by NASA.

`http://www.devtopics.com/20-famous-software-disasters-part-3/`

Famous accidents III

Therac 25 Accident ($\sim$1985)

Cost: ?

Accident: The Therac-25 was a radiation therapy machine which was involved in at least six accidents between 1985 and 1987, in which patients were given massive overdoses of radiation.

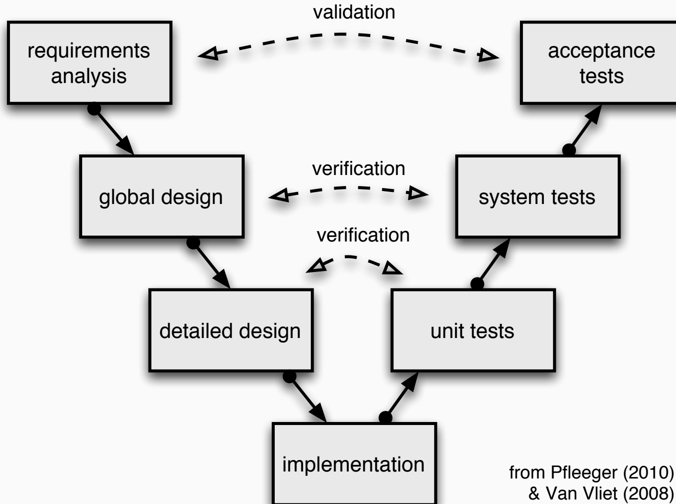
Cause: Concurrent programming errors, combined with integer overflow, lead to sometimes giving patients radiation doses that were thousands of times greater than normal, resulting in death or serious injury.

`http://sunnyday.mit.edu/papers/therac.pdf`

Software quality in Computational Science and Engineering

- errors
  - lead to wrong science
  - not good use of research funding
  - can be dangerous if results are used
  - Some case studies in Mike Croucher on ”*Is your research software correct?*” at `http://mikecroucher.github.io/MLPM_talk/`
- code maintainability and robustness allows
  - re-use by future students/researchers
  - re-use by other groups
  - reproducibility
  - better value for investment into coding

Software engineering overview, V-model



from Pfleeger (2010)
& Van Vliet (2008)

Verification and Validation (terminology)

Validation

Have we got the right product?

(Does the product fulfill the requirements of the customer?)

Verification

Have we got the product right?

(Does the code do what the specification says?)

Planned versus agile software engineering

Plan-driven methods

- separates planning, design, implementation as distinct activities
- integrate customer at beginning and end

Agile methods

- see design and implementation as key activities
- iterative refinement
- integrate all activities and require customer involvement throughout the process

Agile methods

- iterative process
- customer closely involved
- deliver software protoype regularly
- reduce functionality if not sufficient time
- adjust flexibly to customer input
- tests-driven development

Works very well for small teams ($\approx$ 10 people).

(But new evidence that also works well for larger organisations.)

Software processes and methodologies

Methodology and Process

Having a process makes the difference

- between Surgery and Cutting people open
- between Engineering and Tinkering
- between Software Engineering and Programming

Software process

- Requirements
- Hardware
- Language/environment
- Architecture of programs
- Conventions and standards
- Version control
- Continuous integration
- Coordination techniques (planned, agile, etc)
- *Testing*
- *Test Driven Development (TDD)* - critical part of modern process

Testing

Different levels of testing

- unit testing
- integration testing
- systems testing
- regression testing
- acceptance testing

Ways to test

- execute code manually, testing different inputs and outputs
- write and run test code
- test the application by running through particular use cases
- test the application by deploying it
- using dedicated test teams
- white box (glass box) testing & black box testing
- bug seeding (estimate effectiveness of testing)
- weakness oriented testing
- risk oriented testing
- representative testing

Test code

Executable tests

- provide documentation and example use of code,
- provide living form of documentation
- catch future errors
- provide long term time savings
- allow us to change the code easily and *embrace change*

(see ”continuous integration”)

First test code example (1/2)

```python
def f(n):
    s = 0
    # Loop from 0 to n:
    for i in range(1, n + 1):
        s = s + i
    return s

def test_f():
    assert f(3) == 0 + 1 + 2 + 3
    assert f(5) == 15
    assert f(10) == 55
```

First test code example (2/2)

Test by running `py.test` on the source file

```
cd code && py.test -v example1.py
```

```
Wed  9 Dec 2015 15:50:32 GMT
============================= test session starts ============
platform darwin -- Python 3.4.3 -- py-1.4.27 -- pytest-2.7.1 -
rootdir: /Users/fangohr/gitdocs/teaching-software-engineering/
plugins: hypothesis, cov
collecting ... collected 1 items

example1.py::test_f PASSED

=========================== 1 passed in 0.01 seconds =========
```

Test design strategies

- experience
- guidelines
- partitioning
  - identify classes of parameters
  - test within each class
  - test at class boundaries (border cases)

Second test code example (1/3)

```
def sum_custom(n):
    """
    Given an integer n:
      - return the sum from 0 to n if n >= 0
      - return -1 for n <  0
      - raise a TypeError if a is not of type int
    """
    if type(n) is not int:
        raise TypeError("f(n) expects integer, not {}"
                        .format(type(n)))

    if n >= 0:
            s = 0
            for i in range(1, n + 1):
                s = s + i
            return s
    else:
        return -1
```

Second test code example (2/3)

```
from sum_custom import sum_custom as f

def test_positive():          # partitioning n
    assert f(2) == 0 + 1 + 2
    assert f(3) == 0 + 1 + 2 + 3
    assert f(5) == 15
    assert f(10) == 55

def test_negative():
    assert f(-1) == -1
    assert f(-10) == -1

def test_border_case():
    assert f(1) == 1
    assert f(0) == 0
    assert f(-1) == -1
    assert f(-2) == -1
```

Second test code example (3/3)

```
def test_raises_exception():
    with pytest.raises(TypeError):
        f(1.0)
    with pytest.raises(TypeError):
        f("This is a string")
```

Run tests automatically (py.test)

Here we use py.test to execute the tests automatically:

```
cd code && py.test -v test_sum_custom.py
```

```
============================= test session starts =====================
platform darwin -- Python 3.5.1, pytest-2.8.1, py-1.4.30, pluggy-0.3.1
cachedir: .cache
rootdir: /Users/fangohr/gitdocs/teaching-software-engineering/slides/c
collecting ... collected 4 items

test_sum_custom.py::test_positive PASSED
test_sum_custom.py::test_negative PASSED
test_sum_custom.py::test_border_case PASSED
test_sum_custom.py::test_raises_exception PASSED

=========================== 4 passed in 0.01 seconds ==================
```

Test automation

Test automation

Wherever possible, create executable tests so that they can be run automatically.

Effectiveness of testing

Testing effectiveness

Testing can only show the presence of errors, not their absence

(Dijkstra et al, 1972)

- Exhaustive testing generally impossible

Testing tools

Python

- `py.test`/ nose - third party test tools
- PyUnit (object oriented) - inbuilt XUnit style framework
- hypothesis (Python) - automatic test case generation
- coverage - how many lines of code are covered by tests?
- radon Complexity - how complicated is the code?

Other

- JUnit (Java)
- XUnit (many languages)

C

- CUnit, ... (C)

py.test: stdout is hidden if tests pass

Standard output is filtered out for all tests that pass:

```
def f(n):
    print("in f(n), n={}".format(n))
    r = 1 / (n - 1)
    print("return value = {}".format(r))
    return r

def test_f():
    assert f(2.) == 1
```

```
cd code && py.test -v example_output.py
```

```
============================= test session starts ==============================
platform darwin -- Python 3.5.1, pytest-2.8.1, py-1.4.30, pluggy-0.3.1 -- //anaconda/bin/pyth
cachedir: .cache
rootdir: /Users/fangohr/gitdocs/teaching-software-engineering/slides/code, inifile:
collecting ... collected 1 items

example_output.py::test_f PASSED
```

py.test: stdout is displayed if test fails (1/2)

Standard output is displayed for tests that fail:

```
def f(n):
    print("in f(n), n={}".format(n))
    r = 1 / (n - 1)
    print("return value = {}".format(r))
    return r

def test_f():
    assert f(2.) == "provoke error"
```

py.test: stdout is displayed if test fails (2/2)

```
============================= test session starts ==============================
platform darwin -- Python 3.5.1, pytest-2.8.1, py-1.4.30, pluggy-0.3.1 -- //anaconda/bin/pytho
cachedir: .cache
rootdir: /Users/fangohr/gitdocs/teaching-software-engineering/slides/code, inifile:
collecting ... collected 1 items

example_output_fail.py::test_f FAILED

=================================== FAILURES ===================================
____________________________________ test_f ____________________________________

    def test_f():
>       assert f(2.) == "provoke error"
E       assert 1.0 == 'provoke error'
E        +  where 1.0 = f(2.0)

example_output_fail.py:8: AssertionError
----------------------------- Captured stdout call -----------------------------
in f(n), n=2.0
return value = 1.0
=========================== 1 failed in 0.01 seconds ===========================
```

py.test: use -s to not capture stdout

A switch is available to supress capturing of standard output:

-s ⇔ --capture=no

```
def f(n):
    print("in f(n), n={}".format(n))
    r = 1 / (n - 1)
    print("return value = {}".format(r))
    return r

def test_f():
    assert f(2.) == 1
```

```
cd code && py.test -v -s example_output.py
```

```
============================= test session starts ==============================
platform darwin -- Python 3.5.1, pytest-2.8.1, py-1.4.30, pluggy-0.3.1 -- //anaconda/bin/pyth
cachedir: .cache
rootdir: /Users/fangohr/gitdocs/teaching-software-engineering/slides/code, inifile:
collecting ... collected 1 items

example_output.py::test_f in f(n), n=2.0
return value = 1.0
PASSED
```

`py.test`: use `-l` to show local variables (1/2)

A switch is available to show local variables in failing context

`-l` $\Longleftrightarrow$ `--showlocals`

```
def f(n):
    r = (2 * n - 2) * (4 - n ** 2)
    t = 1 / r
    return t

def test_f():
    assert f(2) == 1
```

py.test: use -l to show local variables (2/2)

```
cd code && py.test -q -l example_localvars.py
true
```

```
F
=================================== FAILURES ===================================
____________________________________ test_f ____________________________________

    def test_f():
>       assert f(2) == 1


example_localvars.py:7:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

n = 2

    def f(n):
        r = (2 * n - 2) * (4 - n ** 2)
>       t = 1/r
E       ZeroDivisionError: division by zero

n          = 2
r          = 0

example_localvars.py:3: ZeroDivisionError
1 failed in 0.01 seconds
```

`py.test`: test that exception is raised

- Use `pytest` context to ensure exceptions are raised:

  Example code (`example_exception.py`):

```
import pytest

def f(x):
    if x is None:
        raise ValueError("Called with x==None")

def test_f_exception():
    with pytest.raises(ValueError):
        f(None)
```

- Test will pass only if `f(None)` raises `ValueError`.

`py.test`: Running tests selectively (1/2)

- sometimes, we only want to run one particular test.
- Can select using `-k NAME`, where `NAME` is a substring of the test name(s) to be run.

Example code (`example_select.py`):

```python
def f(x):
    return 2 * x

def test_number():
    assert f(2) == 4

def test_str():
    assert f("fish") == "fishfish"

def test_list():
    assert f([42]) == [42, 42]
```

py.test: Running tests selectively (2/2)

```
cd code && py.test -v -k str example_select.py
```

```
============================== test session starts ============
platform darwin -- Python 3.5.1, pytest-2.8.1, py-1.4.30, plug
cachedir: .cache
rootdir: /Users/fangohr/gitdocs/teaching-software-engineering/
collecting ... collected 3 items

example_select.py::test_str PASSED

========================= 2 tests deselected by '-kstr' =======
===================== 1 passed, 2 deselected in 0.00 seconds ==
```

`py.test`: how to execute all tests

- `py.test MYFILE.PY`
  searches `MYFILE.PY` for functions called `test_*`
- `py.test PATH`
  searches for files called `test_*.py` and `*_test.py` in directory `PATH` and all subdirectories
- `py.test --collect-only`
  shows which test cases can be found (without executing them)

`py.test`: where to put the test code

Many options, including

- combine test functions and actual code in the same file
- gather tests for `lib.py` in `test_lib.py` or `lib_test.py`
- gather test code in separate `tests` subdirectory

py.test: fixtures (1/2)

- If many tests need the same object, create this through a ”fixture” function.
- use Python decorator to make fixture
- use name of fixture as input argument in test functions

```
import pytest

@pytest.fixture
def db():
    # some complicated operation
    print(" == Setting up database == ")
    db = {}     # Imagine this is a data base
    return db  # provide the fixture value

def test_1(db):
    db['key1'] = 42
    assert db['key1'] == 42

def test_2(db):
    assert 'key1' not in db
    assert len(db) == 0
```

py.test: fixtures (2/2)

Fixture function called (=object created) for every test:

```
cd code && py.test -v -s example_fixture.py
```

```
============================ test session starts ===========
platform darwin -- Python 3.4.3 -- py-1.4.27 -- pytest-2.7.1 -
rootdir: /Users/fangohr/gitdocs/teaching-software-engineering/
plugins: hypothesis, cov
collecting ... collected 2 items

example_fixture.py::test_1  == Setting up database ==
PASSED
example_fixture.py::test_2  == Setting up database ==
PASSED

=========================== 2 passed in 0.01 seconds =========
```

`py.test`: setup and teardown fixture (1/2)

- some fixtures need to be shutdown after test
- in XUnit terms: ”setup” and ”teardown”
  - setup is called to create the object required for testing
  - teardown to shut it down after test

```
import pytest

@pytest.fixture
def mydb(request):                # setup
    # some complicated operation
    print(" == setup database == ")
    db = {}
    db['active'] = True
    def myteardown():          # teardown
        print (" == teardown database == ")
        db['active'] = False
    request.addfinalizer(myteardown)
    return db

def test_1(mydb):
    assert len(mydb) == 1

def test_2(mydb):
    assert isinstance(mydb, dict)
```

py.test: setup and teardown fixture (2/2)

```
cd code && py.test -v -s example_setup_teardown.py
```

```
============================= test session starts ============
platform darwin -- Python 3.4.3 -- py-1.4.27 -- pytest-2.7.1 -
rootdir: /Users/fangohr/gitdocs/teaching-software-engineering/
plugins: hypothesis, cov
collecting ... collected 2 items

example_setup_teardown.py::test_1  == setup database ==
PASSED == teardown database ==

example_setup_teardown.py::test_2  == setup database ==
PASSED == teardown database ==


=========================== 2 passed in 0.01 seconds =========
```

py.test: Runtime – keep the tests fast 1/2

- Testing is most efficient if you can run the (automated) tests often and quickly
- Useful to find slowest tests
- Use `py.test --durations=N` to show the N slowest tests

Example code (`example_slow.py`)

```
def f(n):
    if n == 1 or n == 2:
        return 1
    else:
        return f(n - 1) + f(n - 2)

def test_basics():
    assert f(1) == 1

def test_basics2():
    assert f(2) == 1

def test_medium1():
    assert f(30) == 832040

def test_medium2():
    assert f(33) == 3524578

def test_long1():
    assert f(35) == 9227465

def test_long2():
    assert f(36) == 14930352
```

py.test: Runtime – keep the tests fast 2/2

```
cd code &&  py.test --durations=5 example_slow.py
```

```
============================ test session starts ============
platform darwin -- Python 3.4.3 -- py-1.4.27 -- pytest-2.7.1
rootdir: /Users/fangohr/gitdocs/teaching-software-engineering/
plugins: hypothesis, cov
collected 6 items

example_slow.py ......

=========================== slowest 5 test durations ========
4.27s call     example_slow.py::test_long2
2.58s call     example_slow.py::test_long1
0.99s call     example_slow.py::test_medium2
0.23s call     example_slow.py::test_medium1
0.00s teardown example_slow.py::test_basics
=========================== 6 passed in 8.08 seconds ========
```

py.test: Marking tests: expected to fail (xfail)

Can mark tests as eXpected to FAIL where we know about a bug/problem but want to keep the test

```
import pytest

@pytest.mark.xfail
def test_write_theses_function():
    write_thesis(topic="Quantum Physics",
                 length="90 pages")
```

```
cd code &&  py.test -v example_xfail.py
```

```
============================= test session starts ==============================
platform darwin -- Python 3.5.1, pytest-2.8.1, py-1.4.30, pluggy-0.3.1 -- //anaconda/bin/pyth
cachedir: .cache
rootdir: /Users/fangohr/gitdocs/teaching-software-engineering/slides/code, inifile:
collecting ... collected 1 items

example_xfail.py::test_write_theses_function xfail
```

py.test: Marking tests: Skipping tests (1/3)

- Sometimes, tests need to be conditionally skipped
- use `pytest.mark.skipif` decorator

Example code (`example_skipif.py`):

```
import sys
import pytest

@pytest.mark.skipif(sys.version_info[0] >= 3,
                    reason="not python3 compatible")
def test_integer_division():
    assert 1/2 == 0        # only valid in Python <= 2
```

py.test: Marking tests: Skipping tests (2/3)

Running this test with Python 3, will skip the test:

```
cd code &&  py.test example_skipif.py
```

```
============================== test session starts ============
platform darwin -- Python 3.5.1, pytest-2.8.1, py-1.4.30, plug
rootdir: /Users/fangohr/gitdocs/teaching-software-engineering/
collected 1 items

example_skipif.py s

=========================== 1 skipped in 0.00 seconds =========
```

py.test: Marking tests: Skipping tests (3/3)

Switch `-rs` Reports Skipped tests

```
cd code &&  py.test -rs example_skipif.py
```

```
============================= test session starts ============
platform darwin -- Python 3.5.1, pytest-2.8.1, py-1.4.30, plug
rootdir: /Users/fangohr/gitdocs/teaching-software-engineering/
collected 1 items

example_skipif.py s
=========================== short test summary info ==========
SKIP [1] example_skipif.py:3: not python3 compatible

========================== 1 skipped in 0.00 seconds =========
```

XUnit style testing (unittest / PyUnit) (1/3)

```python
import unittest                           # standard Python library

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()
```

XUnit style testing (unittest / PyUnit) (2/3)

```
cd code && python example_pyunit.py
```

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

`py.test` can run `unittest` test cases (3/3)

```
cd code && py.test -v example_pyunit.py
```

```
============================= test session starts =============
platform darwin -- Python 3.5.1, pytest-2.8.1, py-1.4.30, plug
cachedir: .cache
rootdir: /Users/fangohr/gitdocs/teaching-software-engineering/
collecting ... collected 3 items

example_pyunit.py::TestStringMethods::test_isupper PASSED
example_pyunit.py::TestStringMethods::test_split PASSED
example_pyunit.py::TestStringMethods::test_upper PASSED

=========================== 3 passed in 0.03 seconds =========
```

`py.test` switch summary

short	name	meaning
`-v`	`--verbose`	output more detail
`-q`	`--quiet`	output less detail
`-l`	`--showlocals`	show local variables in failing context
`-rs`	`--report=skip`	show skipped tests and reasons
`-x`	`--exitfirst`	exit instantly on first fail
`-k EXP`		run only test with name matching EXP
`-s`	`--capture=no`	show stdout from running tests
	`--durations=N`	show the `N` slowest tests
	`--collect-only`	collect and report tests to run

`py.test --help` shows all options

`py.test` – other features

- `py.test` is big $\rightarrow$ `http://pytest.org`
- `py.test` has many plugins extending its capabilities (`https://docs.pytest.org/en/latest/plugins.html`, `http://plugincompat.herokuapp.com`)
- can provide and clean up temporary files and directories
- provides fixtures per class and module
- ...

Testing of computational science code

How do we test computational codes for correctness as there are no exact solutions to compare against?

- a lot of code is not concerned with the key equation/model, and can be tested as behaviour is deterministic and known

Simulation results can be tested by

- comparison with analytic special cases (often not exploiting the full simulation capabilities)
- comparison with results obtained using a different method
- comparison with results from other simulation codes
- comparison with results from earlier versions of the same software

Software engineering challenges in computational research

Other issues with research software engineering

- computing hardware changes
- unexpected changes in requirements (it's research)
- reproducibility (would be good)
- fast execution competes with readable and maintainable code
- metrics don't reward good software / sustainability
- high turnaround of people (often PhD students)
- lack of training in programming
- lack of training in software engineering (version control, testing)

Should I really be writing test code?

Yes

Good practice:

- Part of any repository
- Ideally part of distribution
  Example: `scipy.test()`
- Ideally run after every code change ($\rightarrow$ continuous integration)

The role of testing

Tests are a tool.

The Process is: *Test Driven Development*

Test Driven Development

Motivation: reduce fear

- Test-driven development (TDD) is a way of managing fear during programming. [from Kent Beck, 2002]
- Fear in the ”*this is a hard problem and I can’t see the end from the beginning*” sense
- Fear
  - makes you tentative
  - makes you grumpy
  - makes you want to communicate less
  - makes you avoid feedback
  - freezes creativity (stops you from exploring new ideas)

Need to *have confidence in our code* to reduce fear.

(Also culture of respect and team spirit in software development teams.)

Test driven development (TDD) basics

Step 1: adding feature

- decide what the new code is going to do
- write a test that will pass when the feature is implemented
- run the new test code, ensure that it fails (”red”)
- write the code until the test passes (”greed”)

Step 2: Refactor

- simplify code
- avoid duplication
- add design decision one at a time

Example (exercise/tdd-units/todo.org)

```
Need a class that can represent distances in mm, metre, cm and
km. Possible design is to carry a value and a unit (='mm', 'm',
'cm' and 'km') around.

Desirable features / use cases:



- Have 'Distance' object that stores values and units
- Convert object to distance in metres
  10 km --> 10,000
  1cm --> 1e-2
  2.5mm --> 2.5e-3
- Convert object to float (always in metres)
- Convert distance to other units
1 km in mm -> 1,000,000
- Add inches to set of known units
  1 in in metres == 0.0254
- Allow addition of Distance objects
  1m + 1m = 2m
- Add distance objects with different units
  1m + 1inch = 1.0254
```

TDD strategy: Fake it till you make it (1/3)

Fake It: return a constant

and gradually replace constants with variables until you have the real code

- okay to make the test pass somehow (initially)
  - commit all coding crimes under the sun if necessary
  - but don’t forget to refactor and tidy up later

TDD strategy: Obvious implementation (2/3)

Obvious Implementation

Type in the real implementation

TDD strategy: triangulation (3/3)

Triangulation

Only generalize code when we have two or more examples. When the second example demands a more general solution, then and only then do we generalize.

Strategy:

- implement case one trivially
- implement case two trivially
- at this point, ’triangulate’ and combine the two special cases into a generic algorithm (and avoid duplication)

Why ”triangulation”? (from Kent Beck’s book)

If two receiving stations at a known distance from each other can both measure the direction of a radio signal, there is enough information to calculate the range and bearing of the signal (if you remember more trigonometry than I do, anyway.) This calculation is called triangulation.

What do we gain from test driven development ?

By writing the tests first, we

- design (the interface) before we code
  - reduces complexity of task as you can focus on the design without having to worry about the implementation
- document our design
  - each test is an example use case
- proof that code implements design
- encourage design of testable code (!)
- achieve test coverage of code automatically
- make refactoring easier (possible)
- benefit from the tests when debugging

Testable code

- looks a lot like good code
- modular
- decoupled design
- methods/functions of limited scope
- reduces cyclomatic complexity

Result of Test Driven Development (TDD)

Better code in less time

…but you will have to use TDD do this for a while before you will be faster

Practial aspects of test driven development

Reminder: TDD development (”red-green-refactor”)

1. Pick a feature
2. write the test first
    - make sure it fails
3. implement some code to somehow make the test pass (without breaking the other tests)
4. refactor existing code and tests (growing code base must be cleaned up regularly during test-driven development)
5. Go back to 1.

By continually re-running the test cases throughout each refactoring phase, the developer can be confident that process is not altering any existing functionality.

What happens in the refactor step?

- New code can be moved from where it was convenient for passing a test to where it more logically belongs
- Duplication must be removed
- Object, class, module, variable and method names should clearly represent their current purpose and use
- As features are added, method bodies can get longer and other objects larger. They benefit from being split and their parts carefully named to improve readability and maintainability.
- Inheritance hierarchies may be rearranged to be more logical and helpful, and perhaps to benefit from recognised design patterns.
- There are specific and general guidelines for refactoring and for creating clean code.

[See also `https://en.wikipedia.org/wiki/Test-driven_development`]

What do we mean by duplication

- The concept of removing duplication is an important aspect of any software design.
- For TDD, it also applies to the removal of any duplication between the test code and the production code — for example magic numbers or strings repeated in both to make the test pass (initially)

Development styles and terminology

- TDD is sometimes describe as ”red/green/refactor”
- Principles associated with TDD:
  - ”keep it simple, stupid” (KISS)
  - ”You aren’t gonna need it” (YAGNI)
  - ”Fake it till you make it” (Beck, 2002)

Which feature to implement first?

- Have a todo list with all features required ("backlog")
    - this will grow and change over time
- pick a feature to work on next that that
    - you feel confident about
    - is realistic to complete quickly
    - will teach you something
- some features may have higher priority (from the customer/requirements)

How long to work on each test?

In other words: how much functionality should one test cover?

- generally: a short time (20 minutes, an hour, …)
- some people try to make the test pass before taking a break

Do I have to test *everything*?

Use judgement:

- some things are too hard to test
- in particular integration with external tools
- some tests are too trivial
- overtesting is possible: try to test each thing once
- exploratory coding without tests is okay

If you have to do back testing

If you have to add tests to existing code:

- write the test
- see it pass
- break the code
- see the test fail (to double check test works)
- [then fix the code again (by going back to original version).]

If you have to work with/extend/maintain code that has no tests

Prioritise your time and write tests for the parts you will be working on

This way:

- you will have tests for new code
- you will have tests for the fast-changing parts of the code
- ’static’ parts of the code may have no tests, but if they do not require change, they are probably bug free

Adding regression tests (if you discover a bug)

- create a test that fails because of the bug
- fix the bug (i.e. make the test pass)
- the process adds a new test to your test suite

Learning opportunity

Try to understand why you did not have this test in the first place

Good practice for writing tests

- Separate common set-up and teardown logic into test support services utilized by the appropriate test cases ('fixtures')
- Treat your test code with the same respect as your production code.
- Get together with your team and review your tests and test practices to share effective techniques and catch bad habits.

Things to avoid when writing tests

- Having test cases depend on system state manipulated from previously executed test cases.
- Dependencies between test cases. A test suite where test cases are dependent upon each other is brittle and complex.
- Execution order should not be presumed.
- Testing precise execution behavior timing or performance.
- Building “all-knowing oracles.” An oracle that inspects more than necessary is more expensive and brittle over time.
- Testing implementation details.
- Slow running tests.

Summary TDD

Key things to remember

TDD 1

- Red/Green/Refactor

TDD 2

- Don’t write a line of new code unless you first have a failing automated test.
- Eliminate duplication.

Tools

Version Control

Repository software we can run locally or on our own servers

- git
- mercurial

Web hosted services to serve repositories (generally free for open source code)

- github (git only)
- bitbucket (git and mercurial)

Testing

- `py.test`, `nose`
- JUnit
- XUnit (https://en.wikipedia.org/wiki/XUnit)
- and more (https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks)
- coverage (https://pypi.python.org/pypi/coverage)

Test coverage

Given some Python code:

```
def sum_custom(n):
    if type(n) is not int:
        raise TypeError("f(n) expects integer, not {}".format(type(n)
    if n >= 0:
            s = 0
            for i in range(1, n + 1):  # Loop from 0 to n
                s = s + i
            return s
    else:
        return -1
```

And some tests

```
from example_partial_coverage import sum_custom as f

def test_sum_custom():          # partitioning n
    assert f(3) == 0 + 1 + 2 + 3
    assert f(5) == 15
    assert f(10) == 55
```

`coverage` example, `terminal` output

- Using the `pytest-cov` plugin, we can ask: how many lines of the source are 'covered' by the tests?
- Command: `py.test --cov --cov-report=FORMAT TESTSTORUN`
- Example

```
cd code && py.test --cov --cov-report=term test_example_partial_coverage.py
```

```
============================ test session starts ===============================
platform darwin -- Python 3.4.3 -- py-1.4.27 -- pytest-2.7.1
rootdir: /Users/fangohr/gitdocs/teaching-software-engineering/slides, inifile: pytest.ini
plugins: hypothesis, cov
collected 1 items

test_example_partial_coverage.py .
--------------- coverage: platform darwin, python 3.4.3-final-0 ----------------
Name                                   Stmts   Miss  Cover
------------------------------------------------------
example_partial_coverage.py                9      1    89%
test_example_partial_coverage.py           7      0   100%

========================== 1 passed in 0.01 seconds ===========================
```

`coverage` example, `html` output

Using `--cov-report=html`, we can create a nice html representation of coverage:

Coverage for **example_partial_coverage** : 89%

9 statements | 8 run | 1 missing | 0 excluded

```
1  def sum_custom(n):
2      if type(n) is not int:
3          raise TypeError("f(n) expects integer, not {}".format(type(n)))
4      if n >= 0:
5          s = 0
6          for i in range(1, n + 1):  # Loop from 0 to n
7              s = s + i
8          return s
9      else:
10         return -1
```

« index   coverage.py v3.7.1

Containers

- Container contains a (virtual) operating system environment
    - typically linux
- popular: Docker and Singularity
- Useful for
    - complicated installations
    - multiple conflicting software environments on the same computer
    - reproducible software environments
- Basic introduction: `https://github.com/fangohr/containers-for-science/blob/master/README.md`

Continuous Integration (CI)

Key idea:

Execute tests automatically when the code changes

In more detail detail

- Continuous integration tool watches repository
- if repository has new commits, run all the tests
- test suites write machine readable test result file
- continuous integration tool emails committer if tests fail
    - also email line manager

- Can also build executables, documentation, release versions
- Run tests on multiple environments (hardware, OS, libraries, ...)

Jenkins CI (`https://jenkins-ci.org`)

- Jenkins (free, flexible, needs server [=hardware])

Jenkins

An extensible open source continuous integration server

BLOG | CONNECT | BUG TRACKER | WIKI | CI | DONATE

Learn more about how
Jenkins + Docker work together

Meet Jenkins
Find out what Jenkins is and get started.

Use Jenkins
See how to get more out of your Jenkins.

Customize Jenkins
Choose from **1122** plugins to customize Jenkins exactly to your needs.

Download Jenkins

Release | Long-Term Support Release

Java Web Archive (.war)
Latest and greatest (1.643)
changelog | past releases

Native packages

Windows
Ubuntu/Debian
Red Hat/Fedora/CentOS
Mac OS X
openSUSE
FreeBSD

Travis CI (`http://travis-ci.org`)

- Cloud hosted service
- commercial provider but test open source code free
- connects with github and bitbucket

Travis CI `.travis.yml`

- Example: `http://github.com/fangohr/travisci`
- Instruct travis via `.travis.yml`

```yaml
language: python
python:
  - "2.7"
  - "3.4"

cache: pip

install:
  - pip install hypothesis

before_script:
#   - sudo apt-get install libsundials-serial-dev libfftw3-dev
#   - ls /usr/lib/x86_64-linux-gnu/

# command to install dependencies
# install: "pip install -r requirements.txt"
# command to run tests
script: make test
```

Other CI services

Many similar Cloud hosted services to Travis CI, including

- Cirle CI
- Snap CI
- ...

Other software:

- Buildbot (`http://buildbot.net`)
  - similar to Jenkins but Python based and more light-weight?

Summary

Summary Software Engineering for Computational Science

- Software Engineering is no exact science
- Best practice for Computational Research
    - version control is essential
    - and having tests is crucial
    - continuous integration should be a must
    - test driven development if you can
    - use containers to set up the software environment
    - automate everything: Computers are good at repetitive things — we must exploit that.
- Choose methods you enjoy and that increase your (long term) productivity. Choose responsibly.

Literature

Literature

Kent Beck: Test Driven Development by Example

Addison-Wesley Signature Series, Paperback – 8 Nov 2002

http://www.eecs.yorku.ca/course_archive/2003-04/W/3311/sectionM/case_studies/money/KentBeck_TDD_byexample.pdf

Talk by Evan Dorn, Los Angeles Ruby Conference

https://www.youtube.com/watch?t=15&v=HhwElTL-mdI

Discussion on TDD

http://martinfowler.com/articles/is-tdd-dead/