Software Testing Lecture 7
Property Based Testing

Justin Pearson

2019

When are there enough unit tests?

Lets look at a really simple example.

```
import unittest

def add(x,y):
    return x+y

class TestAddition(unittest.TestCase):
    def test_add_1(self):
        self.assertEqual(add(3,4),7)
```

We could go on writing a lot of test cases. You can try to find edge cases and then try to get some structural coverage of the code.

Random testing of addition

```
def test_add_random_single():
    x = randint(-sys.maxsize, sys.maxsize)
    y = randint(-sys.maxsize, sys.maxsize)
    assert(add(x,y) == x+y)

def test_add_random(max_iterations):
    i = 0;
    while(i<max_iterations):
        test_add_random_single()
        i += 1
        sys.stdout.write('.')
    print("\nRan ", max_iterations," tests.")
```

Random testing of `addition`

- We should not expect any of our tests for addition to fail. Especially, since we implement `addition` with addition.
- But it does illustrate some important concepts.
  - Test oracles: A mechanism for determining if a test has passed or not.
  - Test generator: A mechanism for generating test cases.
- Some thought has to go into both oracles and generators. For generators you have to decide what sort of coverage you going to give. More on this later.

Set intersection

The details are not important. I built a package[1] for representing sets of integers compactly as trees. Operations such as intersection were provided.
Write inefficient but correct code:

```
def intersection(l1,l2):
  new_list = []
  for i in l1:
      if member(i,l2):
          new_list.append(i)
  return(new_list)
```

This is not a very efficient way of doing set intersection, but we can use it as an oracle together with randomly generated sets to generate lots of test cases.

[1] https://bitbucket.org/justinkennethpearson/mdd

Writing Test Oracles

It is the same problem as always; given some inputs how do we know what the correct output is. Often, you use testing as a way to think about the problem and try to understand what the output will be for given inputs.
If you have lots of randomly generated tests then this is obviously not practical. There are a number of techniques available.

- Specification based verification of the outputs.
- Redundant computations
- Consistency checks.

Specification based Verification

If you are lucky, then your software has a specification. If you are less lucky, while trying to test the software and in dialogue with the programmer you write a specification.

Given your specification there are a number of options.

- Use property based testing to derive test cases. More on this later.
- Use techniques from formal methods such as model checking, specification animation tools to generate test cases. This is often much easier than using formal methods to verify the software.
- Treat the specification as a combinatorial problem and use techniques such as Constraint Programming, SAT or SMT to model your problem and derive test cases (Take 1DL448).
- Write code that implements the specification and generates test cases.

Techniques such as these are often referred to as *model based testing*.

Redundant Computations

We have already seen two examples earlier. There are a number of options.

- Write inefficient but correct code, and use this as an oracle. Very similar to *model based testing*, but often more simple. Never underestimate the power of a random number generator and redundant computations. This can be a very good way of finding faults in code.
- Parallel development of code. Very expensive and only used in domains with high reliability requirements (aerospace). Has its own problems. Psychology tells us that people make the same sort of errors.

Consistency Checks

- Not really a test oracle, but often a good way to find errors.
- Make sure that the system satisfies any invariants.
- Make sure that the system ends up in a consistent state after you have done a bunch of operations.

Model Based Testing

Models can come from a variety of sources:

- Finite state machines;
- UML designs;
- State charts;
- Or formal methods.

Either you have to design your own models or adapt existing models so that they can be used to extract tests. There are various tools and techniques out there to extract test cases from models.

Model Based Testing

- Time spent modelling your problem is not wasted time.
- You have a way to generate lots of test cases, and you understand your problem better.

Property Based Testing

Started with QuickCheck[2] which was a combinator library for describing test cases.
Suppose we have a list reversing function then part of the specification is

$$\forall xs : \texttt{list}(\texttt{int}()) \; \mathit{reverse}(\mathit{reverse}(xs)) = xs$$

Then we turn the specification into code (via macros)

```
?FORALL(Xs,list(int()),
            reverse(reverse(Xs)) == Xs)
```

[2] Koen Claessen and John Hughes (2000). "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs"

Property Based Testing

- `?FORALL` takes three arguments: a variable, a generator and a boolean valued expression (the test oracle).
- `?FORALL` uses the generator to generate random instances, the boolean expression then run with the randomly generated instances.
- If it evaluates to false then we have found a failing test case. There is some clever logic to reduce the size of the failing test case.

Commercial and free implementations exist for many programming languages, including Python.

Property Based Testing

This is not a realistic example, but it illustrates the generation strategy.

```
?FORALL((x,y),(int,int),add(x,y)==x+y)
```

This would generate code something like:

```
error_found = false
while (not error_found):
 x = random(int)
 y = random(int)
 error_found = ( add(x,y) == x+y )

print("Error found")
```

Note that `( add(x,y) == x+y )` means run the equality check and return true or false. Again the beauty is that we use executable code for our assertions.

Property Based Testing

Quickcheck has inspired many property testing tools such as:

- PropEr[3] for Erlang
- Hypthoesis[4] for Python.
- Quickcheck has been re-implemented in various languages.

[3] `http://proper.softlab.ntua.gr/`

[4] https://hypothesis.readthedocs.io/en/latest/

Property Based Testing

There are interesting research questions with important practical consequences:

- If you have a counter example, can you find a shorter more human readable one.
- How do you specify and test concurrent behaviour?

Property Based Testing

- Most importantly, writing your tests as properties not only generates you many more test cases, but it forces you to think about the specification of the system.