Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 4 — January 14, 2019

Patrick Lam *version 1*

Last time. We saw two examples: `findLast`, where we did an example about identifying faults, errors, and failures, and finding distinguishing test cases; and the line intersection example, where we informally reasoned about the correctness of that algorithm and demonstrated it using testing. The reasoning is a lot more ad-hoc than what we've seen in algorithms courses.

About Testing

We can look at testing statically or dynamically.

Static Testing (ahead-of-time): this includes static analysis, which is typically automated and runs at compile time (or, say, nightly), as well human-driven static testing—typically code review.

Dynamic Testing (at run-time): observe program behaviour by executing it; includes black-box testing (not looking at code) and white-box testing (looking at code to develop tests).

Usually the word "testing" means *dynamic testing.* Unit Tests

Naughty words. People like to talk about "complete testing", "exhaustive testing", and "full coverage". However, for many systems, the number of potential inputs is infinite. It's therefore impossible to completely test a nontrivial system, i.e. run it on all possible inputs. There are both practical limitations (time and cost) and theoretical limitations (i.e. the halting problem).

The first part of the course is about defining test suites and aims to give you tools to answer the question "when should I stop testing?" Reiterating the syllabus, I might stop:

- *When I run out of time.* Open-ended exploratory testing; automatic input generation.
- *When I am close enough to being exhaustive.* Coverage of: (enough) statements, branches, program states, use cases.

I'll again put in a shout-out to mutation as being another way to validate whether one's test suite is good enough.

Test cases

As we've seen in the last two lectures, a *test case* contains:

- what you feed to software; and
- what the software should output in response.

Our test cases have been easy to generate so far, but that's not always the case.

Definition 1 Observability *is how easy it is to observe the system's behaviour, e.g. its outputs, effects on the environment, hardware and software.*

Definition 2 Controlability *is how easy it is to provide the system with needed inputs and to get the system into the right state.*

Anatomy of a Test Case

Consider testing a cellphone from the "off" state:

$$\underbrace{\langle \text{ on } \rangle}_{\text{prefix values}} \quad \underbrace{1\ 519\ 888\ 4567}_{\text{test case values}} \quad \overbrace{\underbrace{\langle \text{ talk } \rangle}_{\text{verification values}} \quad \underbrace{\langle \text{ end } \rangle}_{\text{exit codes}}}^{\text{postfix values}}$$

Definition 3

- Test Case Values*: input values necessary to complete some execution of the software. (often called the test case itself)*
- Expected Results*: result to be produced iff program satisfies intended behaviour on a test case.*
- Prefix Values*: inputs to prepare software for test case values.*
- Postfix Values*: inputs for software after test case values;*
  - verification values*: inputs to show results of test case values;*
  - exit commands*: inputs to terminate program or to return it to initial state.*

Definition 4

- Test Case*: test case values, expected results, prefix values, and postfix values necessary to evaluate software under test.*
- Test Set*: set of test cases.*
- Executable Test Script*: test case prepared in a form to be executable automatically and which generates a report.*

On Coverage

Ideally, we'd run the program on the whole input space and find bugs. Unfortunately, such a plan is usually infeasible: there are too many potential inputs.

Key Idea: Coverage. Find a reduced space and cover that space.

We hope that covering the reduced space is going to be more exhaustive than arbitrarily creating test cases. It at least tells us when we can plausibly stop testing.

The following definition helps us evaluate coverage.

Definition 5 *A* test requirement *is a specific element of a (software) artifact that a test case must satisfy or cover.*

Working with TRs:
- formulate a set of TRs (depending on the system)
- ensure that some test case meets each of the TRs

We write TR for a set of test requirements; a test set may cover a set of TRs.

Two software examples:

- cover all decisions in a program (branch coverage); each decision gives two test requirements: branch is true; branch is false.
- each method must be called at least once; each method gives one test requirement.

Infeasible Test Requirements. Sometimes, no test case will satisfy a test requirement. For instance, dead code can make statement coverage infeasible, e.g.:

```
if (false)
  unreachableCall();
```

or, a real example from the Linux kernel:

```
while (0)
  {local_irq_disable();}
```

Hence, a criterion which says "test every statement" is going to be infeasible for many programs.

Quantifying Coverage. How good is a test set? It's great if it covers everything, but sometimes that's impossible. We can instead assign a number.

Definition 6 *(Coverage Level). Given a set of test requirements TR and a test set T, the coverage level is the ratio of the number of test requirements satisfied by T to the size of TR.*

e.g. My test suite achieves 80% statement coverage on this program.