Notes for Spring Term project presentation

Title Slide

- Two key ideas:
  - Mutation testing
  - CI
- Project addresses two of the main challenges involved in bringing them together.

What is mutation testing?

- There is a long explanation, as given in my actual project report, which builds up from manual and automated unit and integration testing, through code coverage, and ends up with mutation testing at the top of that hierarchy.
- Too long for this 8 minute presentation.
- Will assume that audience is aware of common automated test ideas.
  - e.g. JUnit, the Python `unittest` module
  - At least some of the above will have been covered in SEPR, industrial placements
- Shorter explanation is as follows...

If we change...

- "If our application code breaks, will our tests detect that?"
- So one way to understand it is as a *stronger* form of code coverage testing:
  - "Which lines of our program are run by our tests?"
- Next few slides give a quick worked example of mutation testing in action, and introduce some terminology.

Original code

- A pretty basic Python method that is supposed to swap the case of ASCII letters passed in...
- ...and leave all other characters alone.
- Fun fact about ASCII, if there's time, is that the binary string on the front of the CS building spells CSB in ASCII.

Tests

- Here we have some mocked-up tests for the code shown previously.
- Obviously, in real-life these would be in actual code, but doing that would just make it less clear here for people unfamiliar with the Python `unittest` syntax.
- This is example is exhaustively testable, but let's pretend it isn't, since the majority of real-world software isn't.

ROR

- We have a mutation operator here.
- All such operators tend to have these rather unfriendly 3-letter codes.
- This one replaces operators as given in the table.

Mutant code

- This is *one of many* possible applications of the ROR operator to our code.
- The code shown is a mutant, a changed version of our original code.
- Note the `<=` towards the end of the 4th line, first comparison block.
- So what we'd do next in the MT process is run our original tests against this mutant.

Tests against mutant

- As you can see all the tests have passed.
- This mutant has survived.
- This is not a good thing, as the mutant introduces an error that is not checked for by the tests.
- To think of it another way, as the test cannot distinguish between the original code and the mutant, they can't give us any confidence that the original code, and not the mutant, is the correct implementation.

MutPy screenshot

- Screenshot shows real MT output when run against the full demo program from which the `flip` method is taken.
- Note that 5 mutants have survived, 14 have been killed (i.e. not survived).
- Mutation score, the main output for MT, is % of mutants killed — 73.7% in this case.

What is continuous integration?

- a.k.a. CI
- The other main pillar, upon which my project stands.

Every time. . .

- "Every time the code changes, run the tests."
- In practice, this means every time code is pushed to a central repository
  - be that through git or subversion or hg or FTPing files like it's 1995
  - a build is then started and the results reported

Lava lamps

- And this is how it is reported, though not always with actual lava lamps, or traffic lights or things like that.
- Most important outcome is whether the latest build of `master` (or other VCS equivalent) has passed or failed. Binary.
- The next two slides show the outcomes of builds in two popular CI systems: Travis and Jenkins.

Travis screenshot

- Note that this build has passed.
- Note the badge (as often seen in GitHub readmes) next to the build title.
- Also note the elapsed time — CI has to be quick to be of any use!

Jenkins

- Note that this build has failed.
- Note that build time was even faster than the previous screenshot.

Benefits

- Giving real-world developers access to an academic tool.

Challenges

- Not really much to say on this slide, other than to lead into the next two.

Speed

- Intuitively, mutation testing is slow:
  - Remember from above that a single test run can take ~2 mins or even more.
  - In the worst-case (all mutants survive), naive MT involves running the whole test suite per mutant.
  - The tiny example app from earlier slides generated 19 mutants.
  - Number of mutants scales (very, very roughly) with the square of the number of lines of code.

Reporting

- MT's output is a % scale, CI needs a binary red or green answer.
- We can't use 100% mutation coverage, because of equivalent mutants.
- So we need a threshold.
- Also, more mundane reporting considerations: How do we surface enough information to be useful without overwhelming the system's users?

My Project

mutiny

multimutiny

mm-benchmark