Approval Testing Workshop

In this workshop we will use the tool 'TextTest' to test some simple programs. The instructions below assume you are using the virtual machine provided. If you prefer to use your own machine, there are some instructions in the section 'using your own machine'. The idea is that if you use the virtual machine provided, it should be faster to get going with the exercise.

Start the virtual box

Unzip the virtual box image and then open it with Virtual Box. Start the machine (it is named 'approval-testing').

Use the username '**demo**' and password '**codingdojo**'

To do the exercises, you will need to use a 'Terminal', i.e. a command prompt.

- Click on the 'trefoil' (start) button in the bottom left hand corner. Select 'System Tools' then 'XTerm'

Keyboard layout

By default the machine uses a 'us' keyboard layout. To change this:

```
$ setxkbmap se
```

'se' is the code for the Swedish keyboard layout. You could try 'fi' for a finnish layout.

Yatzy Exercise

Yatzy is a simple dice game that you may have played. This code is designed to work out how many points you get for a given roll of five 6-sided dice, when scored in a given category. Categories are things like 'Threes' where you score the total of all the dice reading a '3', (there is a summary of the rules included with the code). I have set it up so you can write approval tests with TextTest, to check the code implements the rules correctly. The thing is, I can tell you now, the code is buggy. In fact, there are three different implementations, each with a its own selection of bugs for you to find.

```
$ cd /home/demo/workspace/Yatzy-Approval-Kata
$ tt
```

I have started to write a first test case, but there are no 'approved' results to compare against, so the test fails. You should start by working out whether the result should be approved, by checking it against the Yatzy rules outlined in the README file. (If you have any questions about the Yatzy rules I can act as product owner and explain them).

Note you can switch the version you're testing by opening the config file and changing the entry for 'executable'. (You can find the texttest config file under python/texttests/config.gr, or via the texttest GUI, on the 'config' tab.) The version 'yatzy1.py' has the least bugs, 'yatzy2.py' has more, and 'yatzy3.py' has many bugs.

Add more test cases. If you find any bugs, don't approve those results, rather create a failing test that shows the actual output you believe should be given instead. Note down on a piece of paper which tests fail against which versions of the code.

Discussion points

- If you have found any bugs in the code, the next step would be to report them and have a developer fix them. What additional information (if any) would you need to give the developer beyond the failing test case(s) you have created?
- How much effort went into creating these tests, and how many bugs will they find, compared with other approaches you could have taken? For example:
  - unit tests with a framework like JUnit
  - 'Quickcheck'-style functional test, where you have a function to generate dice rolls, and a function to check the score is always within allowable ranges for each category.
  - an excel spreadsheet where you specify several rolls of 5 dice, together with the expected answer for each category. Get someone (with a cheap hourly price) to manually check all of the rolls and categories against the actual output from the program.
  - 'Cucumber' style given-when-then specifications for each category
  - Use actual dice, an actual scoresheet, and take pictures of each roll in an actual game. Write an image-recognition program to parse the dice rolled and the score written on the sheet and compare them against the program output.
- Do take a look at the 'sample solution' and compare it with your own. How is it different and why?

```
$ cd /home/demo/solutions/Yatzy-Approval-Kata-sample-solution
$ tt
```

GildedRose Exercise

This is a version of the refactoring kata invented by Terry Hughes. The idea of the exercise is that this code is in production, and is working fine. So in this case, the user is happy and you are not trying to find bugs in it. The trouble is they want a new feature, and before that is going to be possible to add, some refactoring is needed. Before any refactoring can be done, the developers need some regression tests to lean on. I have set it up so you can write approval tests with TextTest, to use as these regression tests. This code is here:

```
/home/demo/workspace/GildedRose-Approval-Kata
```

You can open the TextTests like this:

```
$ cd /home/demo/workspace/GildedRose-Approval-Kata
$ tt
```

At the start of the exercise, the texttest fails, because it has not yet been defined properly, there is no approved 'golden master' to compare against.

The test uses a class called "texttest_fixture.py" to adapt the production code for approval testing. You can take a look at this code, open it in an editor:

```
$ subl python/texttest_fixture.py
```

What it does, is read in a csv file 'items.csv' to populate a list that it then passes to the 'updateQuality' method. Then it prints out the new state of all the items in the list. It will do this once by default, or if you have passed a command line argument, it will execute it that many times. Note that you should not need to change this code, just understand what it does.

You should run the test case that is provided and examine the result. Look at the description of the business rules included in 'README.md'. Once you understand what is going on, you can approve the result. You may also want to rename the test to something more descriptive. As I said before, this code is not considered buggy, it just lacks tests.

You should create some more test cases for interesting cases. You can use the description of the code in the README, and the code itself to help you to discover these. (I am not expecting you to do any refactoring in the production code)

Discussion points

- Are your test cases a better description of what the system does than the initial requirements document that you started with? In what way?
- If the business rules changed, how easy would it be to update the tests to reflect this? For example in the following scenarios:
  - A new kind of item 'Conjured', which degrades twice as fast as normal items.
  - The maximum quality was changed to be 55 instead of 50
  - A new 'Legendary' item was introduced, 'Brimstone Hand', that always has a quality of 95
  - A new item 'concert ticket' that varies according to how many are in stock - the total quality of all the tickets together adds up to 100 at all times, until the SellIn date, when they all drop to 0.
- Can you think of any new requirements that would be really hard to accommodate in these tests?
- Do take a look at the 'sample solution' and compare it with your own. How is it different and why?

```
$ cd /home/demo/solutions/GildedRose-Approval-Kata-sample-solution
$ tt
```

Minesweeper Exercise

The minesweeper exercise is found under the 'workspace' folder. Open it in sublime text so you can browse and edit the files:

```
$ cd /home/demo/workspace/Minesweeper-Approval-Kata
$ subl .
```

The actual code is under the 'python' subdirectory. The 'minesweeper.py' script takes input on standard input, for example 'ATInput.txt' or 'demo.txt', which is a text file showing where the mines are placed on a minefield. This scripts prints to standard output the 'clues' that you would get in a minesweeper game. Run the script like this:

```
$ ./python/minesweeper.py < ATInput.txt
```

You should see it output the 'clues' onto the console. Look at this output carefully. It is not correct, there is a bug in the minesweeper script. It also creates a log file 'minesweeper.log'. In the log file you can see it's not finding all the mines in the input.

Write a texttest that exposes this bug. Start texttest like this:

```
$ tt
```

1. Set up an empty test suite for this script

Enter the application name, 'Minesweeper', a suitable file extension, for example 'ms', and a subdirectory, for example 'texttests'. (This directory will be created under the 'python' folder). For the 'executable' navigate in the file browser and select /home/demo/Minesweeper-Approval-Kata/python/minesweeper.py, which is the program you want to test.

2. Create a simple test case exposing the bug

You should now get a test browser window with one test suite on the left hand side, which is currently empty. Select the test suite and right click to add a test case. The name of the test case should not contain spaces, and should describe the purpose of the test case, without being too

long. Fill in a fuller description below if the succinct description is not enough. The 'minesweeper' script doesn't require command line arguments, so you don't need to fill in any.

In the newly created test case, you will want to specify a file to be sent to standard input. On the right-hand side of the screen, right click on '**Definition files**' and select '**Create/Import**'. The type of file you want is 'stdin', i.e. standard input. You can either create a new file or use the file browser to copy an existing file, for example 'ATInput.txt'.

Now your test is defined enough that you can run it. Select the test case (or test suite) and press '**Run**'. The test run will open in a new window. Since you have not yet approved a 'golden master' for this test case, it will fail. You actually want to create a failing test that exposes the bug. Inspect the output that has been gathered on standard output. You might want to create more tests with different input. If if you find input that actually works, you might want to approve it. In this case click '**Approve**'.

You will want to fix the bug in the 'minesweeper.py' script. Take a look at the line commented 'this is the bug'. Try removing this line. When you have edited the script, run your tests again, until you are happy with the output and are willing to approve it. Now you have a test case.

3. Include the 'minesweeper.log' in the test cases

Now you have a way to regression test the basic functionality of the 'minesweeper' script with different kinds of input. This script also produces a log file which might be interesting to include as one of the 'golden master' files as part of the test case. This log file includes a datestamp on every line that you will need to filter out.

Open the config file for this application (this file is listed on the right hand side of the test browser, on a tab called 'config', double click on the minesweeper config file to open it for editing). This file controls the settings for all the test cases.

You will need to add an entry for '**collate_file**' to pick up the log file, and '**run_dependent_text'** to filter the timestamp. Refer to the configuration file reference documentation for more information:

http://texttest.sourceforge.net/index.php?page=documentation_3_27&n=configfile_default

Hint: to match this string:

```
123
```

You can use a regex '\d+' to match one or more digits:

```
\d+
```

You can also use the '^' symbol to tell it these digits are at the start of the line.

If you're stuck, try this:

```
[collate_file]
log:minesweeper.log
[end]
[run_dependent_text]
log:^\d+{REPLACE <timestamp>}
[end]
```

4. Compare your solution with the example solution

I have already made a working solution to this exercise, you can open it like this:

```
$ cd /home/demo/solutions/Minesweeper-Approval-Kata-sample-solution
$ subl .
$ tt
```

See how it compares with what you have done.

GOCD Dashboard demo

This is a simple web application that displays information about failing pipelines in a GO Continuous Delivery server (GOCD). It is available open source on github, we develop it and use it at Pagero.

git clone https://github.com/magnus-lycka/gocddash

Refer to the README files in that project for how to get it running. I had intended to provide a working installation on this virtual box but didn't have time. You can at least view the test cases using this command:

```
$ texttest -d texttest -c ${PWD}
```

Take a look at the test cases. Work out what is being tested, and, perhaps more importantly, what is not being tested. Consider the following questions:

- what inputs are varied in order to provoke different behaviours in each test case?
- what information is included in the golden master in every test case?
- which pages or parts of pages are not being tested?

Using your own machine instead of the virtual box

You will need to install some prerequisites. Below are some useful commands for a debian/ubuntu machine.

You will need to have python, and texttest. For example:

```
$ sudo apt-get install python-pip
$ sudo pip install texttest
```

For more detailed instructions, and for different platforms see http://texttest.readthedocs.io/en/latest/installation.html

You need to have an editor and a diff tool configured for texttest to use. I recommend sublime text and meld. Install them like this:

```
$ sudo add-apt-repository ppa:webupd8team/sublime-text-3
$ sudo apt-get update
$ sudo apt-get install sublime-text-installer
$ sudo apt-get install meld
```

Then you need to configure texttest to use them:

```
$ cd
$ mkdir .texttest
$ touch .texttest/config
$ subl .texttest/config
```

Enter the following in that file, and save:

```
[view_program]
default:subl
[end]
[diff_program]
default:meld
[end]
```

For convenience, I also like to create an alias ‘tt’ for starting texttest:

```
$ cd
$ subl .bashrc

alias tt=‘texttest -d python -c .’
```

The exercises are all available on Github. For example:

```
$ sudo apt-get install git
$ mkdir workspace
$ git clone https://github.com/emilybache/Minesweeper-Approval-Kata
```

Similarly, you should clone:

- Minesweeper-Approval-Kata-sample-solution
- GildedRose-Approval-Kata
- GildedRose-Approval-Kata-sample-solution
- Yatzy-Approval-Kata
- Yatzy-Approval-Kata-sample-solution