MONASH University

FIT4004 – System Validation & Verification, Quality and Standards

ASSIGNMENT 3

AUTOMATED UNIT TESTING AND CONTINUOUS INTEGRATION

TEST ANALYSIS

ARVIN WIYONO

awiy1@student.monash.edu

24282588

1. INTRODUCTION

The purpose of this report is to communicate the results and analysis of automated unit testing and continuous integration during the implementation of a simple mail merge functionality with Python 3.3. The purpose was to produce a high-quality and functional module that conformed to the given requirements. Moreover, the team agreed to adopt a disciplined Test-Driven Development (TDD) process.

While Python unittest was the major testing framework, the team also made use of other development tools:

- **Nosetests** – Python test runner
- **Coverage.py** – Code coverage report generator
- **Drone.io** – Continuous integration pipeline

2. RESULTS AND ANALYSIS

2.1. DISCOVERED BUGS

Most of the bugs discovered by the unit tests were functional and regression bugs. To be more specific, these bugs were likely to occur within the aggregated function, such as `fill_template()`, that encompasses all required methods to achieve the desirable functionality. However, the source of the defects could be easily identified as tests were continuously executed along with the implemented changes.

With the applied TDD practice, this is logically true as we always strived to validate whether the module exhibited the correct behaviors and confirmed to the functional requirements in a repeated series of *test-code-refactor* cycles. This indeed served as a very strong foundation for the module construction.

2.2. COVERAGE AND CONTINUOUS INTEGRATION TOOLS

Among other tools, **coverage.py** was definitely my favorite. With our white-box testing strategy (i.e. statement and branch coverage) and tests cases that were devised to aim a minimum of 90% code coverage, coverage.py provided us with very useful insight regarding which path that had not been tested yet. For instance, when we experienced a missing test for the Exception branch:

```
83          t = Template(input_string)
84          try:
85                  result = t.substitute(word_hash)
86                  return result
87          except KeyError as e:
88                  raise MacroNotDefined(str(e))
```

Not only the module's coverage, but it also checked which test cases within the test suite that were not successfully run. This situation could occur since all Python unit tests must be named differently. Two or more tests having the same name would lead the other ones to be ignored.

In addition to the coverage tool, having **Drone CI** setup is very similar to having a security protocol that helps assure every build is in a sustainable correctness state. This is useful because sometimes developers forget to run the tests before pushing new commits to the remote repository.

2.3. WHO IS IN CHARGE: DEVELOPER OR TESTER?

With the Agile trend that revolutionizes the modern SDLC, I think it is far better for someone that writes the code to simultaneously write the tests. The justification is simply because unit tests are very implementation related (otherwise, why do we need to think of statement, branch and code coverage?) and therefore developers possess more complete knowledge regarding the state of the system.

Additionally, separating these responsibilities can potentially lead to some pitfalls either within the system or the team due to several reasons:

- Defects cannot be identified quickly.
- Code needs to wait for being tested. Thus, testers can be the major blockers.
- Developers and testers can blame each other for being responsible for the defects, etc.

2.4. THE *MOCKING* PYTHON

Mocking in Python was fun! Although we needed to spend some time to learn, its application had been proven to support the main purpose of the unit test: testing code in isolation. In our case, mock was thoroughly used to test the functions `send_mail()` and `mailmerge()` within the MailMerge class. The two benefits that we gained the most were:

A. **Faster execution time** – all the tests could be executed in an instant. Without mocking, the function `send_mail()` would take some time to send an email to a recipient every time unit testing is run.
B. **Exception testing** – The returned values of the external and internal modules could be defined, thus testing both happy and exception paths were made easier.

3. IMPROVEMENTS AND CONCLUSION

Some improvements that I would like to take into actions in the future are:

- Write more testable code by breaking it into functions with an appropriate size.
- Plan which portions of the code that need to be mocked and document the reasons.
- Write test cases that are more self-documented (test cases need refactor too).
- Spike our test strategy! Experiment and observe whether the strategy can achieve the testing goal.

To conclude, having automated unit testing in place and achieving a great percentage of code coverage is crucial to the quality aspect of software projects. Although it takes more efforts and some scenarios can be missed at some time, it provides us with confidence that our code functions correctly! It is the very first step for achieving continuous integration, continuous delivery and lastly, continuous deployment in the Agile world.