Software Testing 2022/23 Portfolio

Matthew Davidson

January 27, 2023

1 Outline of the Software Being Tested

The software I will be testing is from my Informatics Large Practical project. This software (called PizzaDronz) is an online pizza ordering system that uses drones to deliver pizza to students who are in Appleton Tower. The path the drone takes must avoid designated no fly zones which are usually open areas in which students tend to gather. The purpose of these no fly zones is to prevent collisions (with people) if the drone's hardware fails or the pizza boxes weren't correctly attached to the drone. The data for the no fly zones, restaurants and orders is retrieved from a central server which provides `.json` and `.geojson` files. The program then uses this data to construct a visibility graph of all of the restaurant locations and Appleton Tower. This graph is used to path find from Appleton Tower to each of the restaurants. Each of these paths is then approximated as the drone is only allowed to move a certain distance in each of the cardinal directions.

2 Learning Outcomes

1. **Analyze requirements to determine appropriate testing strategies** [20%]

   (a) **Range of requirements, functional requirements, measurable quality attributes, qualitative requirements**

   See `requirements.pdf`

   (b) **Level of requirements, system, integration, unit**

   See `requirements.pdf`

   (c) **Identifying test approach for chosen attributes**

   See `requirements.pdf` for how I will test each requirement. Overall I will performing the testing activities after I have completed implementation. This is due to the time constraints of the project. The project is due by the 14th of December, so completion of implementation takes priority. Whatever time I have remaining after the implementation stage will be used for testing. In an ideal world I would have taken a test driven development or DevOps approach, however due to an insufficient time budget, I couldn't take the time to write the tests beforehand or set up an automated pipeline for CI/CD.

   (d) **Assess the appropriateness of your chosen testing approach**

   Ultimately this was an appropriate testing approach, as the specification had already been given to me and it was just a case of choosing how I wanted to structure the system and implement it. The chosen approach worked sufficiently however I think that choosing a test driven development approach may have allowed me to be more productive as it allows me to measure my progress more readily and set deadlines accordingly.

   However due to the constrained time I had on this project, it may not have been feasible to write tests beforehand, as finishing the implementation was a higher priority than the overall correctness. Writing tests after completion of the implementation allowed me to prioritise the most important functionality of the system and test that first, then move on to the less important parts once I was happy.

2. **Design and implement comprehensive test plans with instrumented code** [25%]

(a) **Construction of the test plan**

(b) **Evaluation of the quality of the test plan**

(c) **Instrumentation of the code**

(d) **Evaluation of the instrumentation**

See `test-planning.pdf` for all of LO2

3. **Apply a wide variety of testing techniques and compute test coverage and yield according to a variety of criteria** see `code/tests` and `code/src` for the tests and source code respectively. The program will likely not run, as it has been pulled directly from my ILP project and I'm not sure if we're allowed to make our solutions public yet. [20%]

   (a) **Range of techniques**

   (b) **Evaluation criteria for the adequacy of the testing**

   These tests use a simplified version of the card validation system, without dealing with the order objects. The tests are mostly optimistic. Issues in my code may avoid detection to the non-exaustive nature of my testing, ideally I would test every combination of valid and invalid card numbers, CVVs and expiry dates.

   (c) **Results of testing**

   After running 194 tests, 100% passed. Also see the `test-log.pdf`.

   (d) **Evaluation of the results**

   These results are useful as it validates the logic of the card validation system. Despite being simplifed (by avoiding the use of Order objects, etc.) it still demonstrates that the core logic is correct.

4. **Evaluate the limitations of a given testing process, using statistical methods where appropriate, and summarise outcomes** [15%]

   (a) **Identifying gaps and omissions in the testing process**

   A lack of extensive testing for the whole order validation class is a significant omission from the testing process, as ideally each step of the process from taking an order to handing the result of the validation to other parts of the system should be tested. To remedy this, we need to identify the inputs to each of the methods in the order validation class and work out what the correct outputs should look like. Furthermore, we should gather some sample data for orders in order to get a clear picture of the type of data users will be entering, whether is be valid or invalid. Randomly generated data is sufficient for quick and dirty testing but users are unpredictable creatures and so real world inputs are better to ensure long-term reliability.

   A lack of development resource has meant that considerably less time has been spent of testing and evaluation activities than intended. To remedy this, a more concrete plan along with setting goals for developers would allow more simple progress checking. Furthermore identifying more sound requirements and deliverables would allow developers to plan the structure of the system better, and write more conscise code.

   (b) **Identifying target coverage/performance levels for the different testing procedures**

   Ideally I am looking for 100% line/branch coverage in the functionality I am testing, as these are relatively simple functions with few dependencies. For the qualitative requirements identified, I am mainly focussed on meeting Q1, as this has the highest priority. The limit of 2 minutes runtime was set as this is one of the requirements identified in the course specification. This will be measured by timing how long it takes from execution of the program to program exiting.

   (c) **Discussing how the testing carried out compares with the target levels**

   Upon testing F2/F7-9, I found that 100% line coverage was met for the functions related to these requirements.

   For testing Q1, I found that the execution time of the program never really breached 1 minute.

(d) **Discussion of what would be necessary to achieve the target level**

- Q3: monitor the system's status and ensure that the system is up 95% of the time. If this requirement is failing to be met, investigate what is causing the system to crash and implement functionality to improve the ability to recover from a degraded or failing state.
- Q4: monitor the system's RAM usage, if this figure ever goes over 4GB then we could profile the code to see which data structures/operations are using the most memory. Alternative data structures could then be considered, perhaps reducing the memory usage at the cost of speed. A balance would have to be struck between memory usage and execution time.

5. **Conduct reviews, inspections, and design and implement automated testing processes** [20%]

(a) **Identify and apply review criteria to selected parts of the code and identify issues in the code**

- Meaningful variable names: throughout the codebase meaningful variable names have been used as often as possible in order to maximise readability for markers and reviewers.
- Data errors considered: There are a fundemental lack of tests for some parts of the codebase, however for the parts that do have tests there are wide reaching tests.
- Exceptions handled: Throughout the codebase exceptions are handled as often as possible in order to prevent hard crashes which may confuse or frustrate users.
- Consistent use of types: Various types are introduced throughout the codebase and these are named as conventionally as possible
- Proper formatting: the code has been formatted consitently throughout in order to maintain readability and allow other developers to edit the code with minimal confusion.

(b) **Construct an appropriate CI pipeline for the software**

For this system an appropriate CI pipeline would consist of automated integration testing of the feature branch that is being worked on. After commiting changes to the branch, the changes will initally be subject to a set of appropriate unit tests. If these tests are passed to a satisfactory level, then the changes reach the integration stage, where a set of integration tests are performed. If these too pass a satisfactory number of tests, then the changes reach the review stage, where a reviewer manually reviews the changes and ensures they are approptiate. If these changes are suitable to be merged, then they will be merged into the main branch. Otherwise the reviewer will communicate with the programmer the changes that are required and what was wrong with what was submitted.

(c) **Automate some aspects of the testing**

Testing should be embedded throughout the pipeline to ensure that buggy code isn't reaching production. Suitable tests would consist of thorough unit tests for the feature being worked on, as well as relevant integration and system level tests. If the changes are relatively minor, it may not be necessary to perform wide reaching system level tests as this would be a waste of computational resources.

(d) **Demonstrate the CI pipeline functions as expected**

The proposed CI pipeline would identify errors that are caught by the tests written, however if the quality of the tests isn't sufficient then the system doesn't really work. If system level tests are frequently failing then this suggests insufficient unit and integration level tests. However if unit tests are failing and the system level tests are passing then this suggests that the system level tests need improvement. Ideally if the unit tests are correct and submitted code is incorrect then the pipeline would notify the programmer that their submission had failed the tests and provide some information as to which tests failed and potentially the cause of the fault, allowing them to quickly identify the problem and fix with minimal stress.