Merge : A Dating Website for Software Developers (Testing)

1. Introduction

MMG (Manchester Match Group) are an British internet company that owns and operates several online dating web sites. They have asked MMU corp to design and build a dating website aimed at software developers.

Subsequent to an earlier progress report[1], this document provides a description and evaluation of the testing procedures carried out during the initial development phase.

2. Black Box Testing

While 100% test coverage of an application may not be feasible, or even desirable, testing should be as comprehensive as possible to ensure high quality software. This requires examining the application from multiple perspectives. Modelling testing as a 'box', tests types fall into two main categories: 'white box' testing, where tests can see 'inside' the box, and have knowledge of the application code; and 'black box' testing, where no knowledge of the inner workings of the application is required.

Black box testing examines a software application from the outside and can help to identify issues with functionality and useability. It can pass input to the 'box' and get output from the 'box', but it does not require knowledge of what the source code itself is doing. Testing involves performing actions on the user interface (UI) and verifying that the results of each action are as expected. They can include manual testing and UI automation testing.

2.2. GUI Testing Strategy

In choosing which scenario to test at the graphical user interface (GUI) level, we worked with the MMG to first identify their highest priority features. These currently constitute their minimum 'must have' requirements.

We aimed to push testing to the lowest possible level (i.e. mostly unit tests, fewer integration tests, fewest number of GUI tests) as lower-level tests are quicker to write and run, and need to be executed on a frequent basis. Therefore we only tested at the GUI level for scenarios where there were outcomes that could only be tested in the GUI.

In the current implementation of the Merge application, the only data that the user is required to input is their name, email address and location. The only additional condition on each of

[1] https://drive.google.com/open?id=1aiDFjH8dAqrv2eygQ-q_vdN8Og01IcnO2D6xKTZKElo

these is that the relevant field is populated in the GUI. We have no existing website users so the MMG provided us with some test user data (Table 1).

ID	Name	Email Address	Location
001	Romeo Montague	r.montague@gmail.com	Verona
002	Juliet Capulet	j.capulet@hotmail.com	Verona
003	Elizabeth Bennett	e.bennett@longbourn.ac.uk	Meryton
004	Fitzwilliam Darcy	f.darcy@pemberley.ac.uk	Derbyshire

Table 1: Model user data provided by MMG for GUI testing.

2.3. Initial Manual Testing

If automated testing is not possible, GUI testing can be carried out manually. In this project, brief manual testing was carried out to highlight any major issues, before running automated tests. This testing process was defined using a set of ‘manual test scripts’. This will allow tests to be repeated at a later date, and provides documentation of expected behaviours. While documentation is generally discouraged in Agile, in this case it provides an easy method of communication with our external stakeholders MMG . Following testing, the ‘actual outcome’ of each test and its ‘status’ (i.e whether it passed or failed) was recorded. An example of one of the manual test scripts used is provided in Table 2. Other types of manual tests include exploratory testing and useability testing.

ID	Scenario	Test Case	Precondition	Steps	Test Data	Expected Outcome	Actual Outcome	Status
001	View Singleton Profiles	When a singleton creates a profile, they can view a list of other singleton profiles.	Singleton has provided a name, email address and location.	1. Navigate to the homepage and click ‘Add Profile’.. 2. Enter a name, email address and location. 3. Save Profile.	Singleton name Singleton email address Singleton location	A list of singleton profiles is shown on the web page.	A list of singleton profiles is shown on the web page.	PASS

Table 2: Example test scripts used in the manual testing of the Merge GUI.

2.4. Exploratory Testing and Useability Testing

‘Exploratory’ testing involves the unguided exploration of a software product. As the current implementation of the Merge application is fairly limited, so too is our exploration ‘space’. As a basic test, we tried navigating backwards and forwards throughout the application. The current Merge implementation could be considered to have ‘passed’ this round of exploratory testing as we were unable to break it.

The useability of the Merge application could be measured quantitatively (Selenium IDE, for example, can measure the time taken to carry out specific actions, which could be compared for different versions of a feature) or qualitatively (e.g. observations may suggest that users find a particular version of a feature easier to navigate). A qualitative assessment of useability could be obtained by recruiting a cross-section of participants (no more than 5 is likely to be necessary) to carry out particular website tasks (presented as detailed scenarios), either remotely or in person, while observations are made. A sample scenario for the Merge application is shown in Figure 1.

Participant feedback could also be gathered after testing. Regular useability testing may help us to improve a current version of the application, and inform the future design process (e.g. by ensuring consistency and clarity in the labelling of elements, intuitiveness of use, or inclusion of all pertinent business information). A source or potential improvements could also be gathered by comparing observations of the use our website with those of other dating websites.

Figure 1: A sample scenario to provide to participants in useability testing.

2.5. Automated GUI Testing

The goal of our GUI testing is to simulate real user behaviour. In the case of the Merge application, our users will interact with a GUI (i.e. a web page). Automated GUI testing can be used to simulate the user interactions (e.g. keyboard input, mouse clicks etc.).

We used ‘Selenium’ for our automated GUI testing. This is a popular open source Java library which offers several tools for web automation. For initial testing we generated replayable GUI tests by recording our interactions with the website using the ‘Selenium IDE’ extension for Google Chrome.

We were also able to configure additional assertions as part of our Selenium IDE tests (Figure 2). We found that the easiest way to do this was to add unique ids to the elements we wanted to ‘assert’ e.g.

.

Going forward, we will implement a ‘locator’ strategy when writing the code for our views. This involves assigning a unique id to each element on a page.

Selenium IDE also offers the facility to export tests as Java JUnit tests. While recording our tests meant we able to avoid writing tests in code (which can often prove arduous), our recorded tests may be difficult to maintain if our testing requirements change in the future. We therefore exported our recorded tests and added them to the rest of the unit tests for the Merge application. This will make our test easier to maintain as our application grows.

For the same reasons we will develop future automated GUI tests in Selenium WebDriver. This will also allow us to develop our tests directly from business requirements using a ‘business driven development’ process.

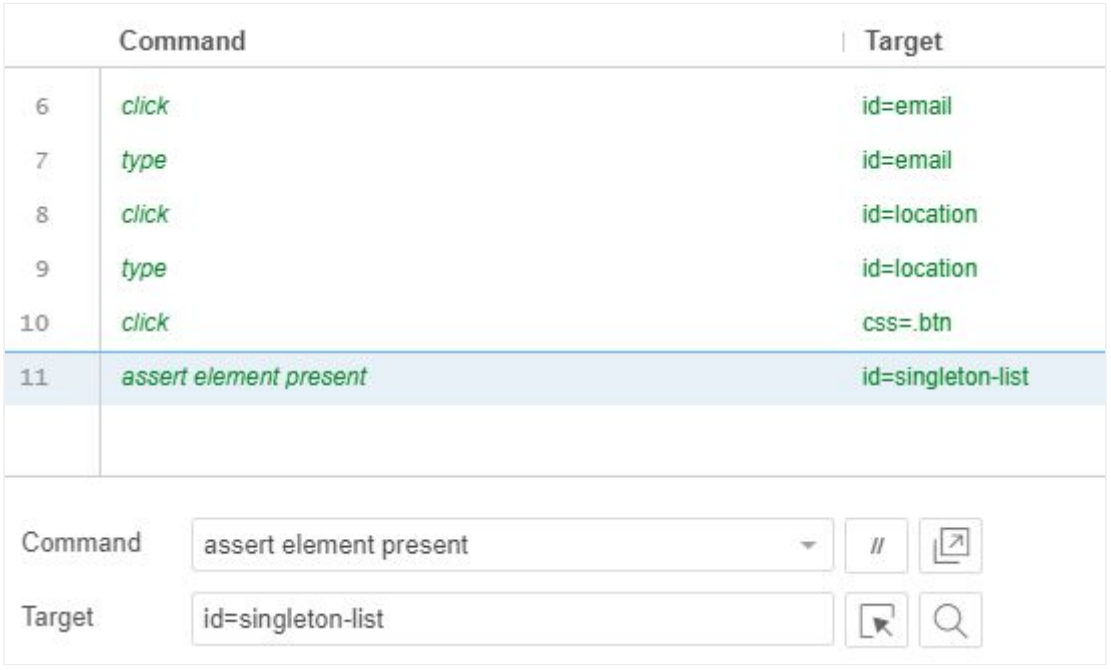

Figure 2: Configuring additional assertions in Selenium IDE.

3. Behaviour Driven Development

Behaviour Driven Development (BDD) is a development approach which emphasises communication, collaboration and providing value to the product owner. BDD is therefore highly compatible with an overall agile development philosophy. As in test driven development (TDD), BDD tests are written before the code but they are written in business-friendly language, and focus on the behaviour of the system being developed. BDD is used to write tests that can then form the basis of TDD at the implementation stage.

BDD starts at the requirements gathering phase of the software lifecycle. In collaboration with the client ‘user stories’ are developed, which provide descriptions of required functionality from the perspective of the system user. For each user story, a set of ‘acceptance criteria’ is also defined, which formalises each feature or requirement and maps out all possible scenarios for that requirement. This process helps to avoid miscommunication of requirements between the client and the development team, and provides a method to verify when a requirement has been fulfilled. An example of a user story obtained during the initial development of the Merge application is shown in Figure 3. To support TDD at the initial implementation stage of this project, user stories and acceptance criteria were translated into executable tests that were used to verify system behaviour.

BDD is driven by business needs and is therefore more likely to result in software products that the user (and therefore the product owner) actually want. BDD (in conjunction with TDD and automated testing) also supports continuous delivery (in line with agile principles) and ensures high quality code. These factors may ultimately increase the confidence of developers with respect to their code, and lower costs by minimising risks and reducing the cost of maintaining the code.

Figure 3: An example of a ‘user story’ developed during an initial workshop with MMG.

3.1. Acceptance Tests

Gherkin is a specification language that can be used to develop executable tests for the BDD tool Cucumber. Gherkin uses plain language, which is easily understandable by technical and non-technical stakeholders. The aim is to improve communication, and allow all stakeholders to be involved in developing system tests.Gherkin makes use of a number of keywords including:

- ‘Given’- the precondition for the scenario.
- ‘When’ - the action taken by a system user.
- ‘Then’ - the expected behaviour of the system for this scenario.

Tests written in Gherkin can also act as ‘living’ documentation for the system being developed.

Based on the user stories gathered in collaboration with MMG we wrote acceptance tests for our application using Gherkin. Selenium Webdriver will allow our acceptance tests to be converted into unit tests. Figure 16 in an example of an acceptance test for a typical successful user path through the application (the ‘happy path’). We will use this happy path scenario as the starting point for our GUI testing in WebDriver.

Figure 16: A user acceptance test written in Gherkin.

4. White Box Testing

‘White box’ testing focuses on what is happening in a software application at the code level. Tests (e.g unit tests) are written for existing code, and are used to test its functionality by verifying the expected execution result for specific inputs. To ensure the testing is comprehensive, test cases (e.g. sets of input values and the output they are expected to generate) should be chosen to maximise the coverage of the set of all possible code execution paths through the application.

4.1. Test Cases

One method that can be used to generate test cases for white box testing is to create and examine ‘control-flow’ graphs for the application. These aim is to identify all possible execution paths. Using an example from this project, a simple control flow diagram can be used to represent part of the ‘addUser’ method in the UserController class (Figure 5). Here ‘result’ is bound to an instance of the ‘User’ class which has three required fields : ‘name’, ‘email’ and ‘location’. These are input by the user in the GUI and are obtained from the ‘add-user’ view. If any of the relevant fields are left empty ‘result.hasErrors()’ = ‘true’.

Figure 4: An extract from the Merge code. The 'addUser' method in the UserController class.

The control flow diagram (Figure 5) indicates two possible execution paths, and so we define two test cases:

- Case 1: result = ("John", "", "Manchester") should return the 'add-user' view (add-user.html);
- Case 2:result = ("John", "john@domain.com", "Manchester") should return the 'index' view (index.html).

Once the test cases have been generated they can be used as the basis for writing unit tests (e.g. see Figure 6).

The coverage of control flow testing can be limited by the fact that, where there is interaction or overlapping functionality between code blocks, testing every statement in a block of code will not necessarily mean all execution paths are tested. This could result in bugs in the code being missed. While path coverage is important for comprehensive testing, as the code for our application continues to be developed, and the number of decision points increases, the number of paths will also increase exponentially. This may make it infeasible to test all possible execution paths.

One way to circumvent this issue is via the structured testing of all code 'branches', allowing the evaluation of all possible decision outcomes. This method is called 'basis path' testing, and the set of identified branches is called the 'basis set'. The number of basis sets can be significantly lower than the number of paths in a large code base, reducing the time and effort required for testing. For example, in this project, there are three decision points in the 'add-user' view that lead to 8 possible execution paths, but only 4 possible branches in the basis set.

Some limitations of the white-box testing methods described above are that they can't account for 'impossible' paths (e.g. those with interacting or contradictory decision points) or paths that are 'missing' (i.e. required paths that have not been implemented).

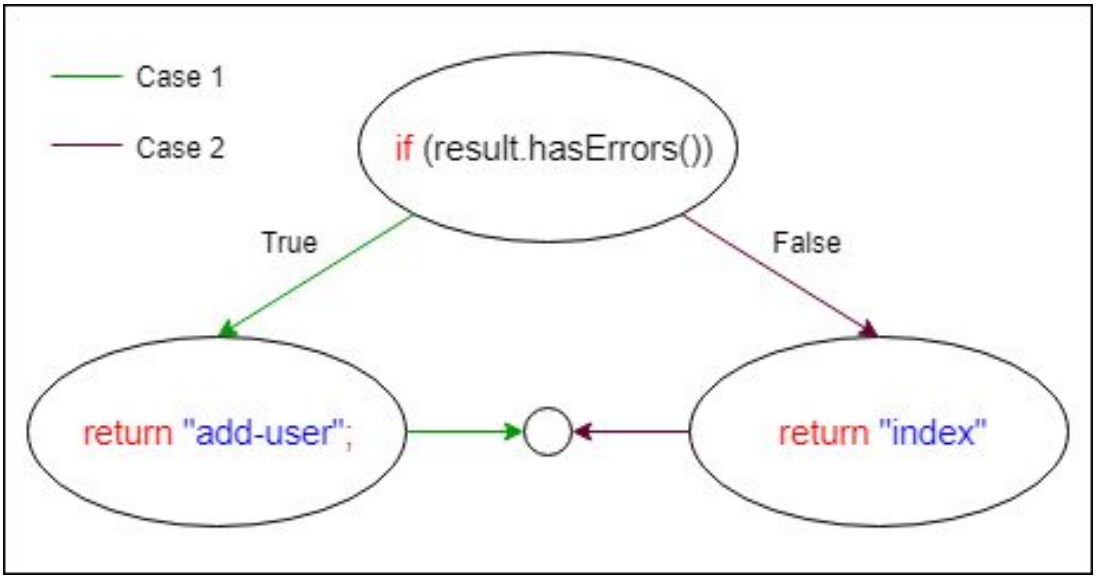

Figure 5: A control-flow diagram for the add-user method.

```
@Test
public void whenCalledaddUserAndValidUser_thenCorrect() {
    User user = new User( name: "John",  email: "john@domain.com",  location: "Manchester");

    when(mockedBindingResult.hasErrors())
            .thenReturn(false);

    assertThat(userController.addUser(user, mockedBindingResult, mockedModel))
            .isEqualTo("index");
}

@Test
public void whenCalledaddUserAndInValidUser_thenCorrect() {
    User user = new User( name: "John",  email: "john@domain.com",  location: "Manchester");

    when(mockedBindingResult.hasErrors())
            .thenReturn(true);
```

Figure 6: Examples of unit tests written for the 'UserController' class.

5. GitHub: Bug Management and Code Reviews

In this project we are using Git as our source control tool[2]. To manage the project repository we use GitHub - a hosting service for git repositories that provides a GUI for source control management, and a number of other collaboration features.

5.1. Bug Management

'Bugs are managed in GitHub as 'issues'. The process for creating issues in GitHub is shown in Figures 7a- 7c. Issues can labelled, or assigned to a specific team member (Figure 9). Each issue is also associated with own discussion forum, where team members can collaborate. Issues can be closed by clicking on 'Close Issue' in the GitHub 'Issues' tab (Figure 8), or by a commit that references a GitHub keyword[3] and the issue reference number.

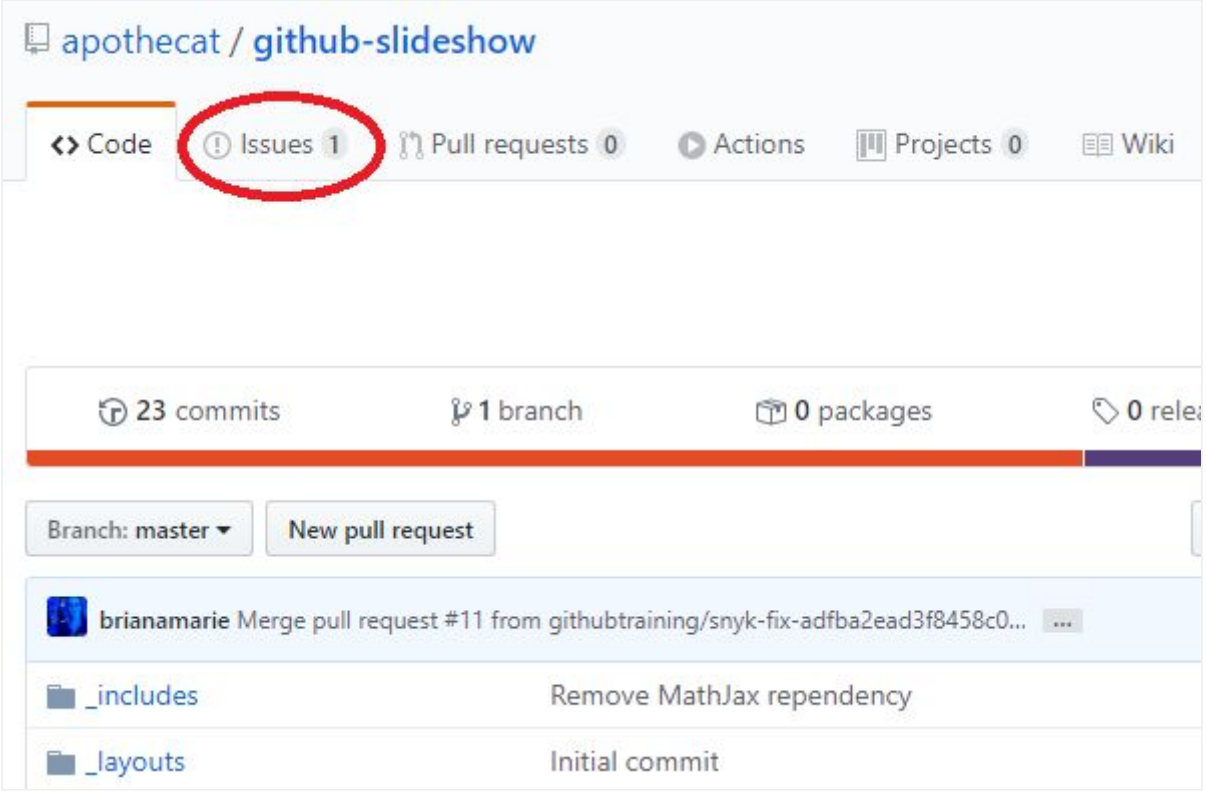

Figure 7a: Creating an issue in GitHub (The GitHub Training Team, 2020). Go to GitHub and navigate to the main page of the project repository. Click on the 'issues' tab.

[2] https://drive.google.com/open?id=1aiDFjH8dAqrv2eygQ-q_vdN8Og01IcnO2D6xKTZKEIo

[3] Close, closes, closed, fix, fixes, fixed, resolve, resolves, resolved (GitHub Help 2000a)

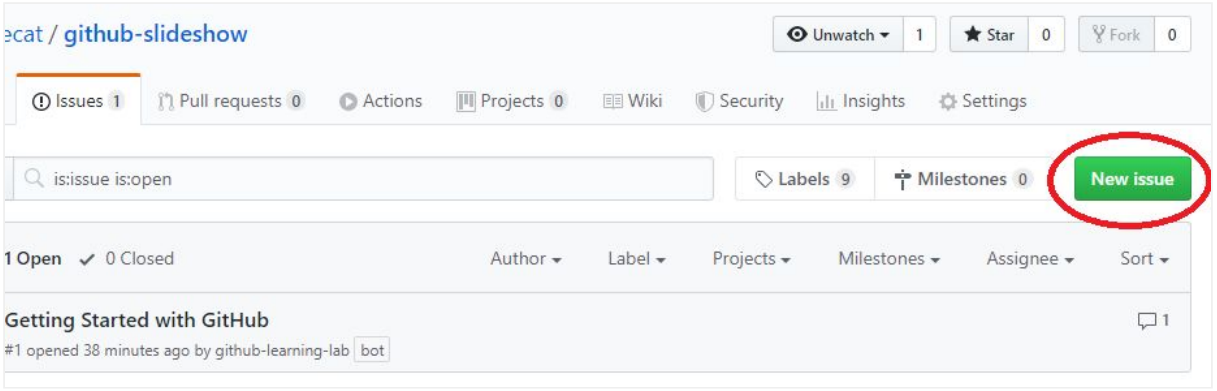

Figure 7b: Click on 'New Issue'.

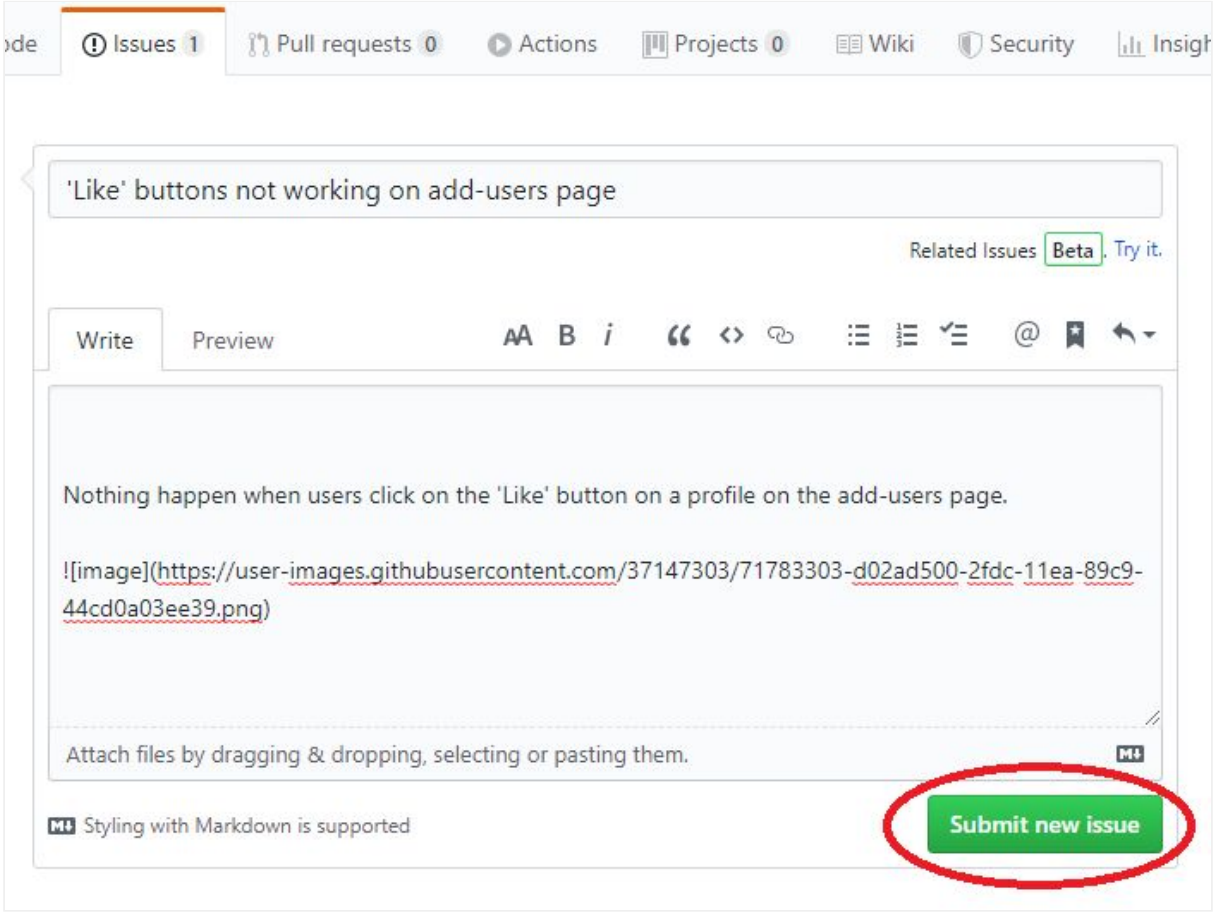

Figure 7c: Add a title and description for the issue, then click 'Submit new issue'.

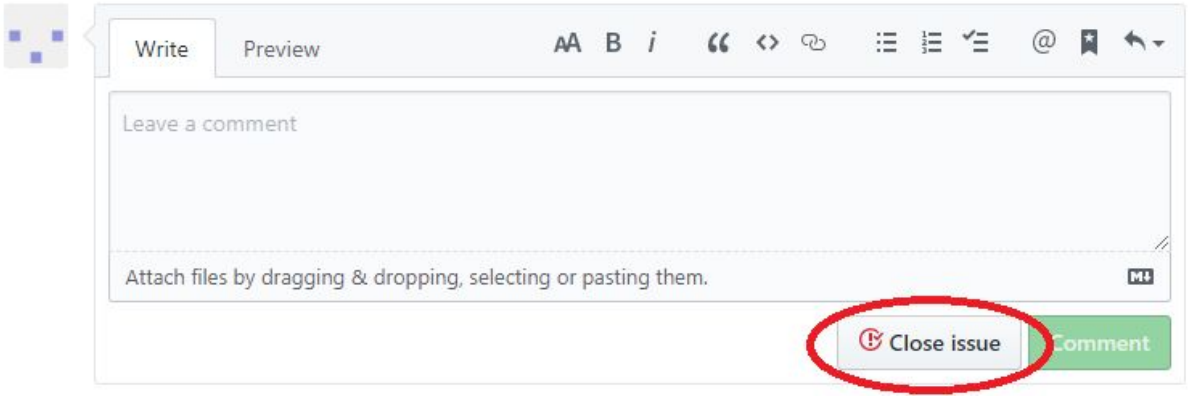

Figure 8: Closing an issue in GitHib (The GitHub Training Team, 2020).

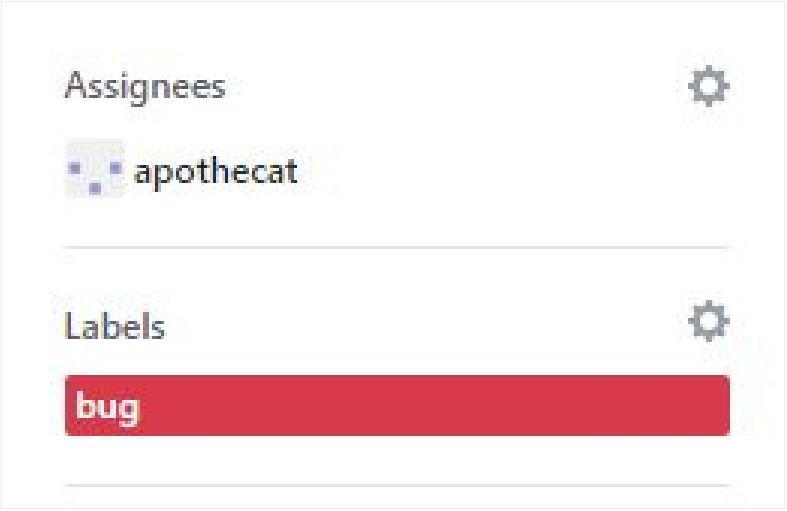

Figure 9: Assignees and labels in GitHub issues (The GitHub Training Team, 2020).

5.2. Code Reviews

In this project we use ‘GitHub Flow’ - a workflow based on the popular ‘GitFlow’ branching model (Atlassian, no date). In GitFlow, developers must create a new branch for each new feature under development, branching from, and merging back into a ‘develop’ branch. Changes are added to feature branches in the form of ‘commits’, which document the changes that have been made and why.

GitHub Flow builds on GitFlow by introducing the concept of ‘pull requests’. A pull request is made when a developer is ready to merge a feature branch back into the main development branch. This notifies an assigned collaborator, who can then review the developer’s commits, and provide feedback so that any necessary improvements can be made before a merge is executed (Figure 10). The process for making a pull request in GitHub is shown in

Figures 11a-11c. The process for reviewing a pull request in GitHub is shown in Figures 12a-12d. New GitHub issues can be opened for code associated with a pull request (GitHub Help, 2020b), or closed by referencing a GitHub keyword[4] and the issue reference number in the request description.

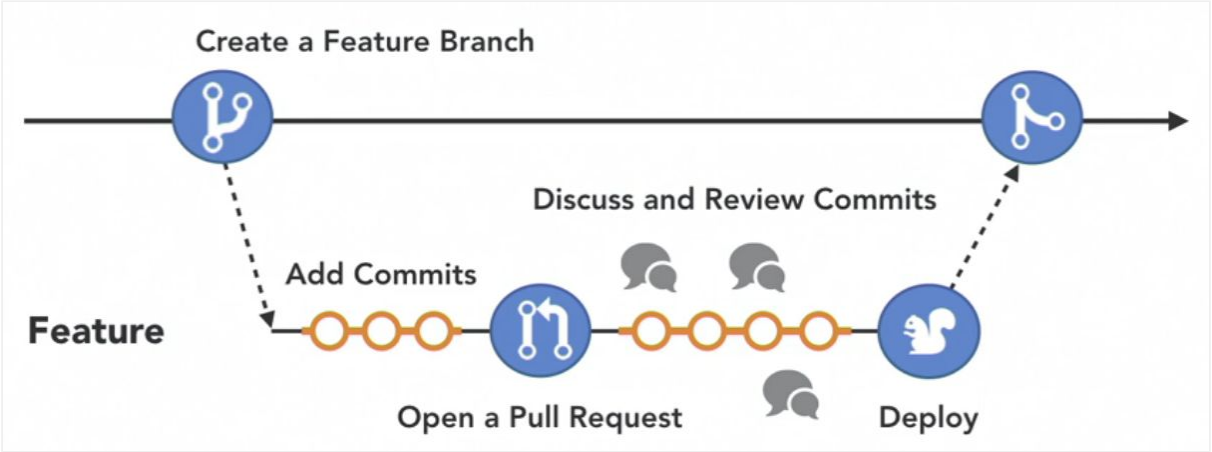

Figure 10: GitHub Flow (LinkedIn Learning, 2019).

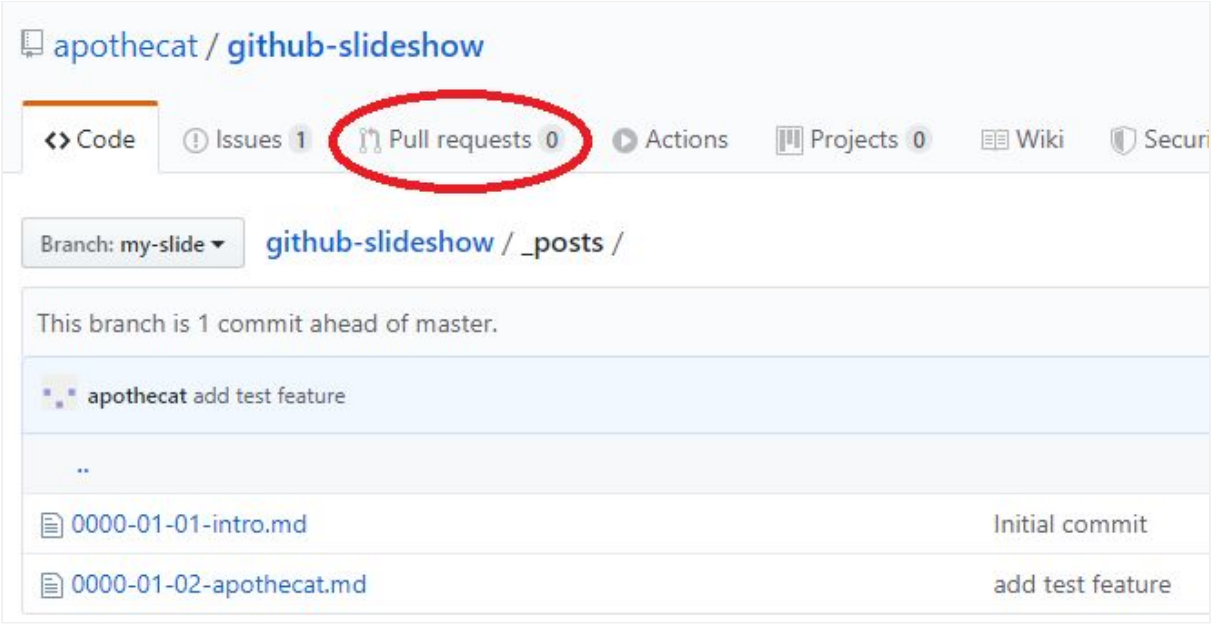

Figure 11a: Making a pull request in GitHub (The GitHub Training Team, 2020).. Go to GitHub and navigate to the main page of the project repository. Click on the "pull requests" tab.

[4] Close, closes, closed, fix, fixes, fixed, resolve, resolves, resolved (GitHub Help, 2020a)

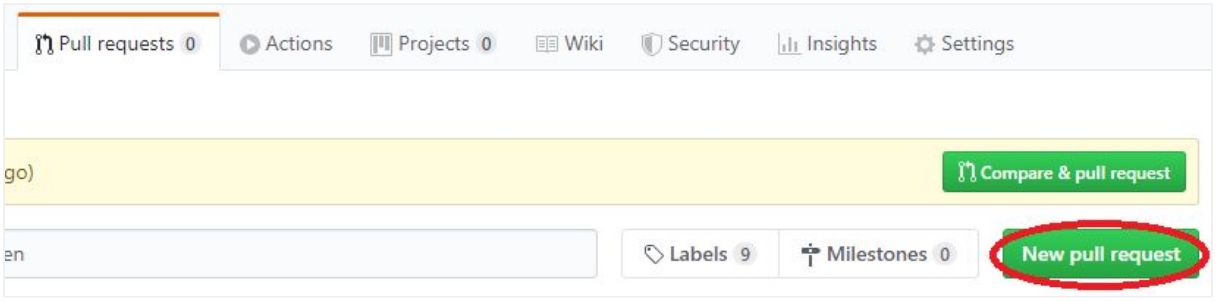

Figure 11b: Click New pull request.

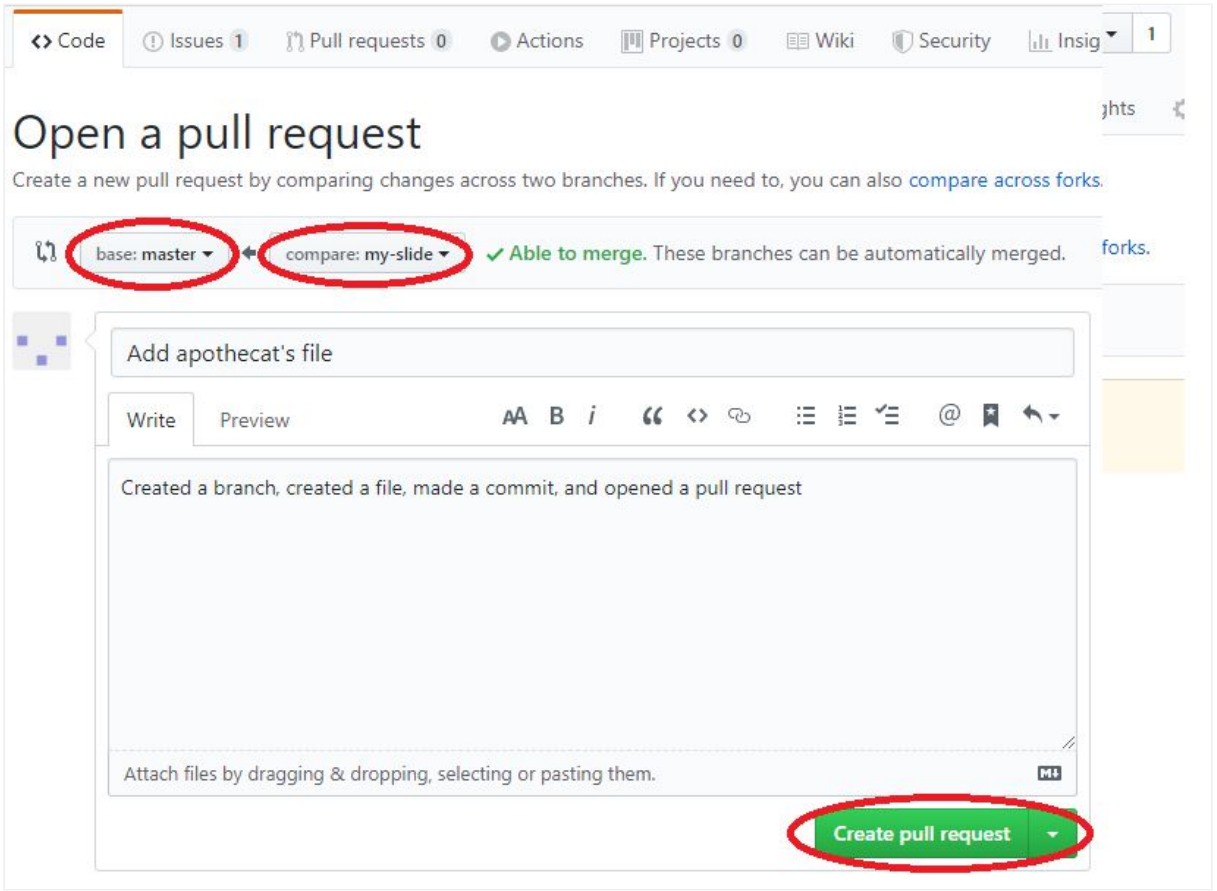

Figure 11c: In the 'code' tab. In the 'base' drop down menu. select the branch you want to merge to. In the 'compare' drop-down menu, select the name of the feature branch you want to merge. Enter a title for your pull request, and a description of the changes. Click create pull request'.

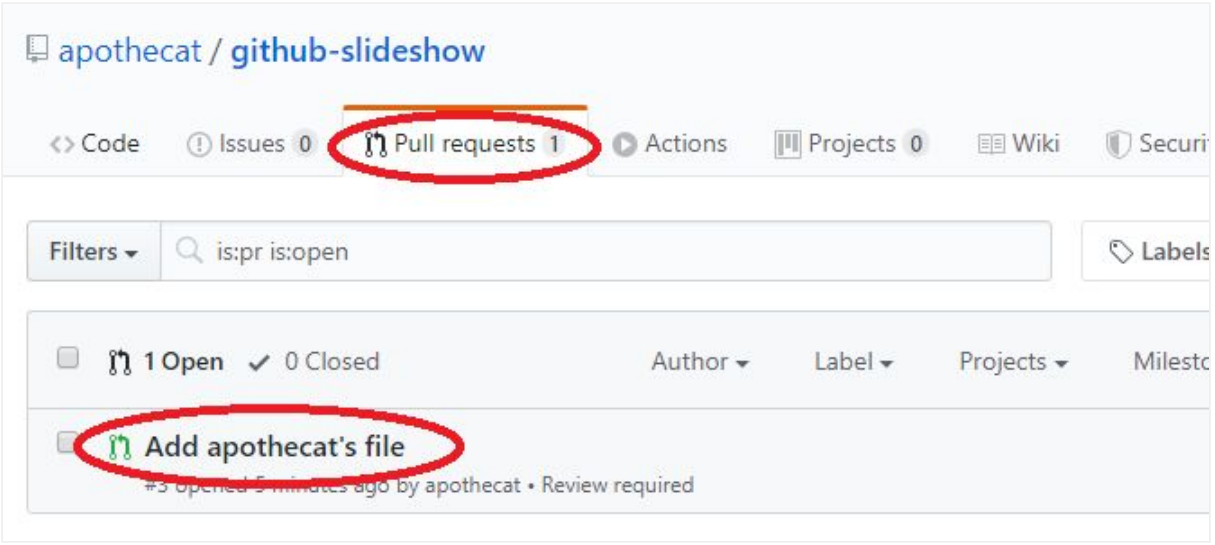

Figure 12a : Reviewing a pull request in GitHub (The GitHub Training Team, 2020). In the 'pull requests' tab click on the request you would like to review

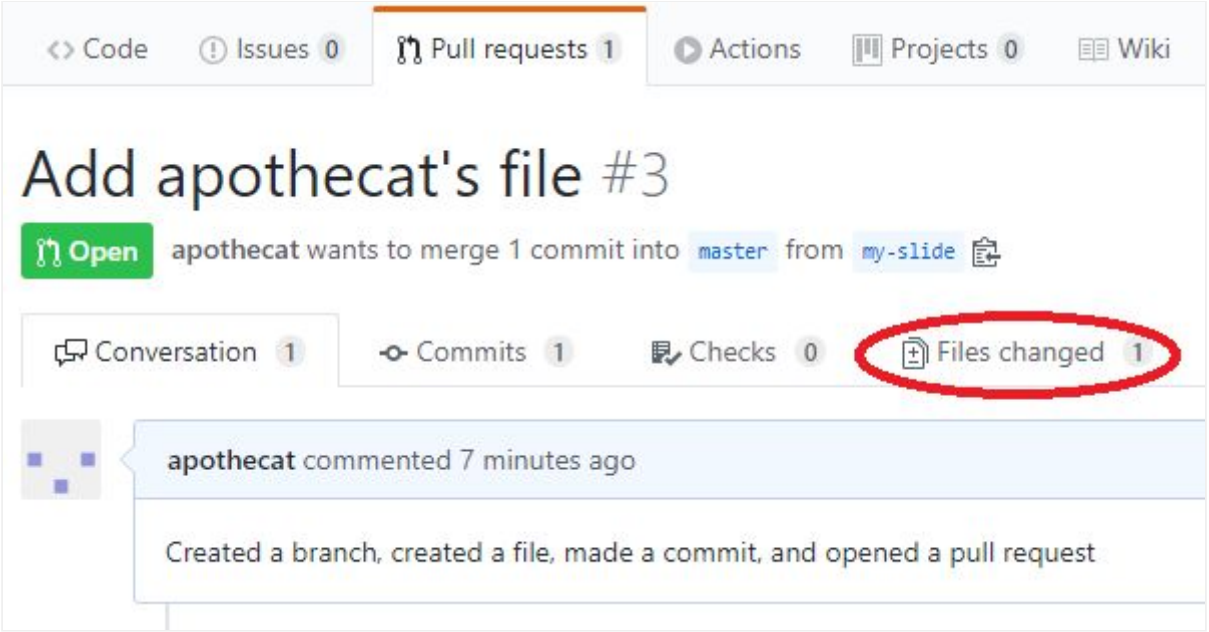

Figure 12b: In the Pull request, open the 'Files changed' tab to review the changes.

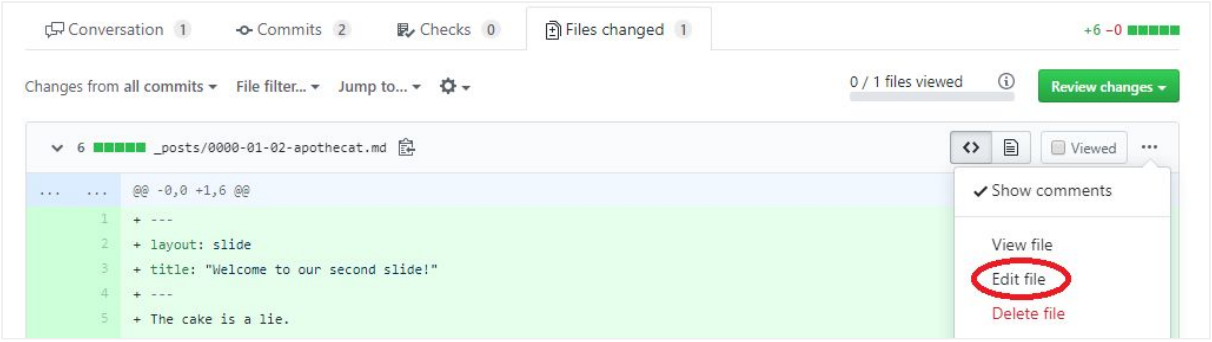

Figure 12c: Changes can be made to files in the GUI by clicking the 'Edit file' in the drop down menu on the right.

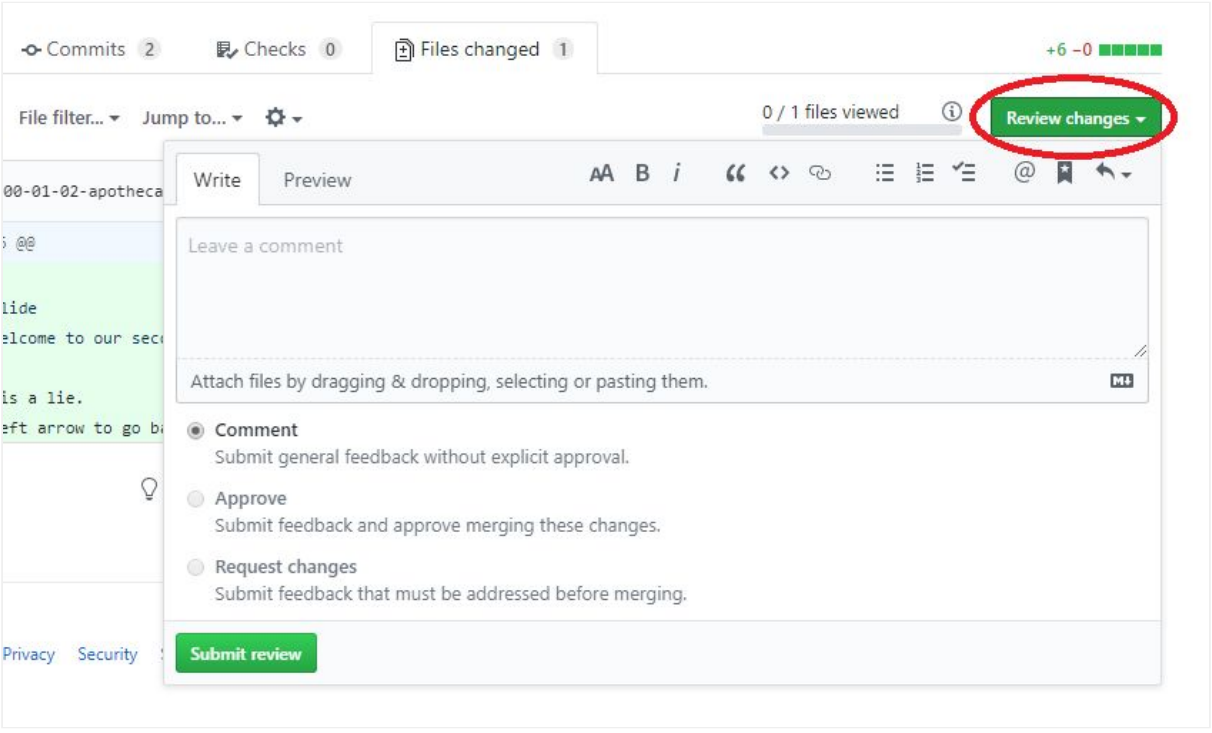

Figure 12d: Reviewers can provide feedback, request changes, and approve pull request using options in the 'review changes' drop down menu.

6. Static Analysis

Static analysis is the examination of source code, without the need to execute the code. A number of static analysis tools exist that can be used to automatically review code and look for flaws. The choice of tool generally depends on the programming language being used. As this project in written in Java, using the IntelliJ IDE, the following tools were used to carry

out a static analysis on code for this project via the suite of 'QAPlug' plugins for intelliJ (Soldevelo, 2019):

- Checkstyle https://checkstyle.org/ which checks Java code for its adherence to coding standards.
- PMD https://pmd.github.io/ an analysis tool which supports Java, and looks for common programming flaws and security vulnerabilities.
- FindBugs https://spotbugs.github.io/ which is designed to search for bugs in Java code.

A summary of the static analysis is shown in Appendix 1. The full static analysis report can be viewed in the 'analysis results' window in IntelliJ (Figure 13)..[5]

The process of using the static analysis tools to carry out a review of the project code was quick and simple. Manually reviewing the code would take much longer, even for the initial, basic implementation of the Merge site. One can imagine that on a larger project, the time (and therefore cost) of using an automated tool over manually checking the code could be significant. The speed and simplicity of use of analysis tools could also allow for more frequent testing of the code.

In addition, the analysis tools picked up on several issues that I had missed in my initial manual review of the project code (e.g. an unused import java.util.Optional in UserService.java). As I am new to Java programming, the tools were also able to identify coding and design issues that I did not have the required level of knowledge to recognise (e.g. the incorrect use of hidden fields and public / default constructors). The manual review process could be improved by passing the code on to a human expert for review. However, it is possible that their knowledge may still be subject to 'unknown unknowns'. Experts can also be expensive to hire.

While the use static analysis tools may be quicker, cheaper, more comprehensive, and less error prone than manual code reviewers, they may not be as useful in detecting design flaws. One could imagine that they could also become intrusive if the defined analysis parameters are too stringent . In addition, as automated tools rely on particular signatures for their analyses, there is a possibility that some problems may be missed. The generation of 'false-negatives' could then lead to a false sense of security regarding code quality if automated tools are relied on too heavily. A combination of the use of automated analysis tools and manual checks may therefore be the best approach to reviewing our application code.

[5] Open the 'merge' project in IntelliJ. Right-click on the 'main' directory and select 'Analyse' then 'Analyse Code'. This will open the 'analysis results' window displaying the full static analysis report.

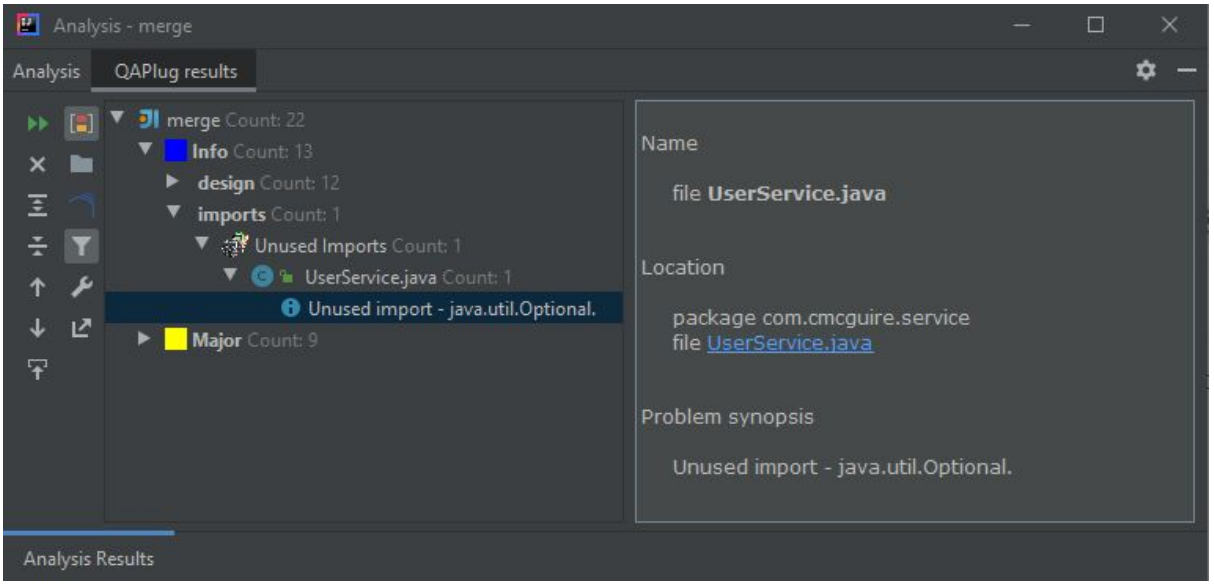

Figure 13: The QAPlug 'analysis results' window, displaying the results of the static analysis of the project code using the 'PMD' 'Checkstyle' and 'FindBugs' tools.

7. Formal Methods

Formal methods are an approach to creating 'error free' code by applying formal mathematical logic in the design, development and verification of software systems (Sommerville, 2010). Mathematical analysis is used in many areas of engineering to improve design reliability. It has previously been proposed that the application of formal methods could achieve this in the software engineering domain. However, historically, the use of formal methods has not been widespread, owing to the perception that they are time consuming, expensive, and require specialist skills, and are therefore not a worthwhile investment for most software projects (Finney, 1996; Holloway, 1997; Sommerville, 2010).

The use of formal methods has therefore typically been reserved for critical systems in industries such as defence, medicine and avionics, where software errors can have a devastating impact, and confidence in code precision is vital (Leveson and Turner, 1993; SolarWinds, 2009; Rose 1994, Tassey, 2002). However, as society is increasingly relying on software in all aspects of daily life, and as technological advancement move towards more automated systems (e.g. self-driving cars), there is an increasing focus on creating highly reliable software, and many large companies have started to turn towards the use of formal methods (e.g. AWS, Microsoft, Facebook, Dropbox, MongoDB, Elasticsearch).

There are two main areas of study in formal methods 'formal specification' and 'formal verification'.

6.1. Formal Specification

Formal specification involves creating a high level description of a system's properties and behaviours, written in a formal specification language (with defined vocabulary, syntax and semantics) that allows mathematical deductions to be made about it (Sommerville, 2010). There are a number of languages used for formal specification including VDM, Z notation, TLA+ and PlusCal.

TLA+ is a high-level specification language which is especially useful in testing concurrent and distributed systems (Lamport, 2018). PlusCal is an algorithm language that is designed to be easier for programmers to understand (Wayne, no date). Figure 14 shows an example of a PlusCAL algorithm which models a simple banking application. Using the TLA+ Toolbox IDE, PlusCAL algorithms can be transpiled into TLA+ providing a more user-friendly way to create formal specifications (Lamport, 2019). Figure 15 shows TLA+ specification for the algorithm shown in Figure 14.

Multi-thread processes can be particularly problematic as they can give rise to race conditions. The above method could be used in our application to model and specify any proposed multi-thread process e.g. if we were to implement a requirement to approve new user profiles before they are published, on the submission of a new user profile, the application might need to return an HTML response to the user, while simultaneously instigating an approval process.

Formal specifications also provide documentation of system requirements and functionality, and the act of writing them can be useful in itself to uncover issues in the early stages of a project (Lamport, 2002; Newcombe et al., 2015). However, using a formal specification to verify the 'correctness' of a subsequent software design also relies on the assumption that the specification itself is 'correct' (Gaudel, 1994; Lamsweerde, 2000; Méry and Singh, 2010). In addition, creating a comprehensive formal specification at the start of a project does fit with the agile philosophy, although some have argued that if implemented appropriately, formal methods can enrich agile processes (Ghezzi, 2018; Nummenmaa et al., 2011). Some aspects of software functionality, such as user experience, can also be difficult to quantify using formal specification.

```
---- MODULE transfer ----
EXTENDS Naturals, TLC

(* --algorithm transfer
variables alice_account = 10, bob_account = 10, money = 5;

begin
A: alice_account := alice_account - money;
B: bob_account := bob_account + money;

end algorithm *)
====
```

Figure 14: A PlusCAL algorithm for a simple banking application. Alice and Bob each have some amount of money in their accounts. Alice would like to send some money to Bob (Wayne, no date).

```
EXTENDS Naturals, TLC

(* --algorithm basic
variables alice_account = 10, bob_account = 10, money = 5;

begin
A: alice_account := alice_account - money;
B: bob_account := bob_account + money;

end algorithm *)
\* BEGIN TRANSLATION
VARIABLES alice_account, bob_account, money, pc

vars == << alice_account, bob_account, money, pc >>

Init == (* Global variables *)
        /\ alice_account = 10
        /\ bob_account = 10
        /\ money = 5
        /\ pc = "A"

A == /\ pc = "A"
     /\ alice_account' = alice_account - money
     /\ pc' = "B"
     /\ UNCHANGED << bob_account, money >>

B == /\ pc = "B"
     /\ bob_account' = bob_account + money
     /\ pc' = "Done"
     /\ UNCHANGED << alice_account, money >>

Next == A \/ B
           \/ (* Disjunct to prevent deadlock on termination *)
              (pc = "Done" /\ UNCHANGED vars)

Spec == Init /\ [][Next]_vars

Termination == <>(pc = "Done")

\* END TRANSLATION
```

Figure 15: An example of a formal specification, written in TLA+, transpiled from the PlusCal algorithm shown in Figure 14.

6.2. Formal Verification

Formal verification is the process of verifying the ‘correctness’ of a system’s code against its formal specification using mathematical methods

This can include deductive methods - using ‘theorem provers’ to producing formal mathematical proofs of code correctness; ‘model-checking’ - verifying code properties by exploring all possible resulting states of program execution; or static analysis - the verification of specification satisfaction by abstracting the program execution.

However, writing mathematically ‘correct’ programs, and providing the required mathematical proofs is often difficult. One alternative approach often used is ‘design-by-contract’. This focuses on the precise description of method syntax and behaviour, and specifies the interactions between the communicating components of a system. Assertions are embedded in the code itself. When combined with model checking, this has proven to be a powerful method to detect subtle bugs and generate innovative designs (Newcombe et al., 2015).

References

Atlassian. (no date) *Gitflow Workflow*. [Online] [Accessed on 4th December 2019] https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

Finney, K. (1996) 'Mathematical Notation in Formal Specification: Too Difficult for the Masses?' *IEEE Transactions on Software Engineering*, 22(1), pp. 158-159.

Gaudel M. (1994) 'Formal specification techniques'. *Proceedings of the 16th international conference on Software engineering*, Washington:IEEE Computer Society Press, pp. 223–227.

Ghezzi, C. (2018) 'Formal Methods and Agile Development: Towards a Happy Marriage.' In Gruhn V., Striemer R. (ed.) The Essence of Software Engineering. Berlin:Springer

GitHub Help. (2020a) *Closing issues using keywords.* [Online] [Accessed on 30th December 2019]

GitHub Help. (2020b) *Opening an issue from code.* [Online] [Accessed on 28th December 2019] https://help.github.com/en/github/managing-your-work-on-github/opening-an-issue-from-code

Holloway, C. 'Why Engineers Should Consider Formal Methods.' *Proceedings of the 16th Digital Avionics Systems Conference.*

Lamport, L. (2002). *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Boston: Addison-Wesley.

Lamport, L. (2018) The TLA+ Home Page [Online] [Accessed on 3rd January 2020] https://lamport.azurewebsites.net/tla/tla.html

Lamport, L. (2019) *The TLA+ Toolbox.* [Online] [Accessed on 4th January 2020] https://lamport.azurewebsites.net/tla/toolbox.html

Lamsweerde, A.V. 'Formal specification: A roadmap.' *ICSE 2000: Proceedings of the Conference on The Future of Software Engineering*, New York:ACM USA, pp. 147–159

Leveson, N. G. and Turner, C. S. (1993) 'An Investigation of the Therac-25 Accidents.' *Computer*, 26(7), pp. 18–41.

LinkedIn Learning (2019) *The basic GitHub workflow*. [Online] [Accessed on 27th December 2019]

https://www.linkedin.com/learning/github-essential-training/the-basic-github-workflow?u=36102708

Méry D. and Singh N.K. (2010) 'Trustable Formal Specification for Software Certification.' In: Margaria T., Steffen B. (ed.) Leveraging Applications of Formal Methods, Verification, and Validation. Berlin: Springer.

Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M.and Deardeuff, M. (2015) 'How Amazon web services uses formal methods.' *Commun. ACM,* 58(4) pp. 66–73.

Nummenmaa, T., Tiensuu, A., Berki, E., Mikkonen, T., Kuittinen, J. and Kultima, A. (2011) 'Supporting agile development by facilitating natural user interaction with executable formal specifications.' *ACM SIGSOFT Software Engineering Notes*, 36(4), pp. 1-10.

Rose, B. (1994) *FATAL DOSE Radiation Deaths linked to AECL Computer Errors*. [Online] [Accessed on 4th January 2020] http://www.ccnr.org/fatal_dose.html

Solarwinds. (2009) *10 Historical Software Bugs With Extreme Consequences.* [Online] [Accessed on 1st January 2020] https://royal.pingdom.com/10-historical-software-bugs-with-extreme-consequences/

Soldevelo. (2019) *QA Plug quality assurance plugin.* [Online] [Accessed on 31st December 2019] https://qaplug.com/ https://qaplug.com/

Sommerville, I. (2010). *Software Engineering.* 9th. ed., Boston:Addison-Wesley.

Tassey, G. (2002) *The economic impacts of inadequate infrastructure for software testing*. National Institute of Standards and Technology.

The GitHub Training Team. (2020) *Introduction to GitHub.* [Online] [Accessed on 29th December 2019] https://lab.github.com/githubtraining/introduction-to-github

Wayne, H. (no date) *PlusCal.* [Online] [Accessed on 2nd January 2020] https://learntla.com/pluscal/

Appendix 1: Summary of Static Analysis

From ‘qaplug_result.html’

Project:

merge

Scope:

Directory '...\src\main'

Profile:

Default

Results:

Enabled coding rules: 460

- FindBugs: 346
- Checkstyle: 31
- PMD: 83

Problems found: 22

- FindBugs: 0
- Checkstyle: 22
- PMD: 0

Time statistics:

- Started: Thu Jan 02 20:43:47 GMT 2020
    - Initialization: 00:00:00.016
    - Checkstyle: 00:00:00.500
    - FindBugs: 00:00:01.015
    - PMD: 00:00:01.664
    - Gathering results: 00:00:00.031
- Finished: Thu Jan 02 20:43:50 GMT 2020