**Test Plan**

**Introduction**

Our system under test (SUT), Fat Free CRM is an open source customer relationship management (CRM) platform built using Ruby on Rails. The system features group collaboration, campaign and lead management, contact lists, and opportunity tracking.

**Objectives and Tasks**

1. **Objectives**

Our main objective was to develop and execute Software and Quality Assurance test plan and interpret the results.

1. **Tasks**

We tested the selected modules from the SUT using various tools for static and dynamic testing. We also analyzed the bottlenecks in the application and the areas of code that need improvement, using a couple of QA metrics.

We have performed manual inspection on each of the selected module and compared the results with the results generated from a static analyzer, Rubocop. We have also found the code coverage of the SUT using SimpleCov, a code coverage analysis tool. Static

Later, we performed dynamic testing to test the user interface of the SUT using RSpec test cases and also automated the web browser using Capybara and Selenium.

Moreover, we used QA metrics like Rack-mini-profiler and GitStats to find the choke-points in the code and to analyze the commit history of the repository.

**Scope**

Since the product has varied functionalities, we chose to work on the following modules – Leads, Tasks, Users and Dashboard.

We have analyzed the reports generated at every level of the test plan, but we have not changed the source code to improve the quality of the code. Our analysis might help the authors improve the quality of their code.

**System Overview and Key Features**

Fat Free CRM is an open source customer relationship management platform. It can be operated on different machines for example windows, Mac and Linux. It comes with trouble free installation process. Moreover, the program comes with a number of comforts that provide users with gobs of options. Fat Free CRM provides abundant benefits for web-based management. It allows users to send mass emails, create tickets and manage meetings all in one program. The software also provides customized access depending on each user’s role.

**Test Strategy**

**Static Testing:**

1. **Manual Inspection:**

* **Definition**

An inspection is one of the most common sorts of review practices found in software projects. The goal of the inspection is for all of the inspectors to reach consensus on a work product and approve it for use in the project.

* **Participants**

Manual Inspection was performed by the following team members:

1. Sayali Pendharkar
2. Sanjay Kalla

* **Methodology**

Our team has manually inspected four different modules of our SUT and generated a report describing different test report criteria like coding standards, comments, logic and error handling.

We considered the files related to our modules and analyzed the code manually playing the role of a moderator and recorder.

* **Results**

|  |  |
| --- | --- |
| **Manual Inspection Summary** |  |
| Code Statistics | 4 modules |
| **Dashboard Module** |  |
| Lines Covered | 156 |
| Number of Bugs Reported | 6 |
| **Leads Module** |  |
| Lines Covered | Controller- 256 ; Model - 197 |
| Number of Bugs Reported | Controller- 2 ; Model – 1 |
| **Tasks Module** |  |
| Lines Covered | Controller- 219 ; Model - 284 |
| Number of Bugs Reported | Controller- 2 ; Model – 4 |
| **Users Module** |  |
| Lines Covered | Controller- 159 ; Model - 184 |
| Number of Bugs Reported | Controller- 2 ; Model - 2 |

Our results were stationed on four distinct criteria:

* + 1. Coding Standards (indentation, magic nos.)
    2. Comments (needless redundant etc.)
    3. Logic (array index)
    4. Error handling (error messages).

The results of manual inspection are recorded in an excel sheet named ‘manual\_testing\_reports’ in the Test Support folder of the repository.

* **What can we reflect from our learning in this entire course** -

First thing first which is 100% automation testing is not possible. The pivotal Goal of Manual Testing is to make sure that the system under test is defect free and works according to aspired requirements.

Testers has to play role of an end-user and discern unexpected error /bug. Moreover, find ways to eradicate those bugs in all possible and unique ways. Manual testing concepts are so extensive, it also includes exploratory testing as a subset, wherein testers explore the software to identify errors in it.

So from our manual inspection experience we were able to make following notes: High priority was not that common and was visible for just a few cases - Refactoring of code, error messages and useless assignments to variable. Low Priority - Long classes, long methods, indentation, needless comments.

1. **Static testing using a static analyzer (Rubocop):**

* **Definition**

Static testing is a software testing method that involves examination of the program's code and its associated documentation but does not require the program be executed. Rubocop is a static code analyzer for Ruby.

* **Participants**

The following tasks were performed by the team member:

1. Sayali Pendharkar – Install and execute Rubocop and generate reports.
2. Sanjay Kalla – Analyze the reports and compare them with the results from manual testing.

* **Methodology**

In advance to what we had done, we used a static analyzer named Rubocop and generated and analyzed the report with the existing manual inspection report.

* **Results**

|  |  |
| --- | --- |
| **Rubocop Summary** |  |
| Code Statistics | 4 modules |
| **Dashboard Module** |  |
| Lines Covered | 156 |
| Number of Bugs Reported | 5 |
| **Leads Module** |  |
| Lines Covered | Controller- 256 ; Model - 197 |
| Number of Bugs Reported | Controller- 3 ; Model – 1 |
| **Tasks Module** |  |
| Lines Covered | Controller- 219 ; Model - 284 |
| Number of Bugs Reported | Controller- 3 ; Model – 5 |
| **Users Module** |  |
| Lines Covered | Controller- 159 ; Model - 184 |
| Number of Bugs Reported | Controller- 4 ; Model - 2 |

The results generated by Rubocop are recorded in an excel sheet named ‘manual\_testing\_reports’ in the Test Support folder of the repository where a comparison of the reports is possible.

Installation and Usage of the tool is mentioned in detailed in ‘Installation\_Usage\_Instructions’ file in the Tool Support folder of the repository.

* **What we learned out of this observation?**

We discovered many new criteria like levels of block testing using tool that were not considered while manual inspection. Comments given were different and in more comprehensive than manual inspection comments. New coding standards depicted more testing was required than manual inspection. Also, sometimes we have reported bugs to be present, however tool didn’t consider them as bugs.

1. **Code coverage:**

* **Definition**

Code coverage is a measure used to describe the degree to which the source code of a program is tested by a particular test suite. A program with high code coverage has been more thoroughly tested and has a lower chance of containing software bugs than a program with low code coverage.

* **Participants**

1. Sayali Pendharkar – Installing and executing the tool.
2. Sanjay Kalla – Analyzing the reports.

* **Methodology**

We have used SimpleCov to find the code coverage of the SUT.

* **Results**

The report generated states that 44.22% of the code was covered - 5921 lines were covered and 7470 lines were missed. The analysis shows that the coverage for the SUT is low. It should be increased by adding more test cases and covering more lines of the code.

From the reports, mostly branches are not covered, some methods are not covered entirely since Test cases for all the methods are not written.

|  |  |
| --- | --- |
| **Code Coverage Summary** | 44.22% coverage |
| Code Statistics | 354 files;13391 relevant lines |
| **Dashboard Module** |  |
| Coverage | 35.44% |
| Lines Covered | 28 |
| **Leads Module** |  |
| Coverage | Controller-58.62% ; Model - 18.49% |
| Lines Covered | Controller- 22 ; Model - 51 |
| **Tasks Module** |  |
| Coverage | Controller-18.69% ; Model – 53.72% |
| Lines Covered | Controller- 20 ; Model - 65 |
| **Users Module** |  |
| Coverage | Controller-31.15% ; Model – 61.33% |
| Lines Covered | Controller- 19 ; Model - 46 |

Installation and Usage of the tool is mentioned in detailed in ‘Installation\_Usage\_Instructions’ file in the Tool Support folder of the repository.

* **Reflections from lessons learned -**

Code coverage is a white box testing methodology.

Types of code coverage:

* Statement Coverage,
* Path Coverage,
* Branch Coverage.

In general, statement coverage should be high; branch and path coverage will be comparatively low. However achieving high code coverage does not guarantee high optimality product; it merely improves the quality of the product.

1. **Metric Fu:**

* **Definition**

Metric\_fu is a set of rake tasks that make it easy to generate metrics reports. It uses Saikuro, Flog, Flay, Reek, Roodi, and Rails built-in stats task to create a series of reports.

* **Participants**

1. Sayali Pendharkar – Installation and execution of the tool

* **Methodology**

It finds its usage in number of applications like:

1. Flog – Measures code complexity
2. Saikuro – Measures cyclomatic complexity
3. Flay – Finds duplication (both copy/paste and structural)
4. Reek – Spots code smells
5. Roodi – Finds lots of general problems
6. Churn – Identifies files that change too often
7. Rails best practices– Code matric tool
8. Cane– Code quality threshold violations
9. HotSpot– Meta analysis of your metrics to find hotspots in the code

* **Results**

Flog shows you the most torturous code you wrote. The higher the score, the harder it is to test. Average score for our SUT is 13.0.

Saikuro is a Ruby cyclomatic complexity analyzer. It is a quantitative measure of the number of linearly independent paths through a program's source code.

|  |  |  |
| --- | --- | --- |
| Module | Total Lines | Total Complexity |
| Dashboard | 150 | 34 |
| Tasks | Controller- 213; Model – 257 | Controller- 48 ; Model - 58 |
| Leads | Controller- 250 ; Model - 159 | Controller- 56 ; Model – 34 |
| Users | Controller- 153; Model - 144 | Controller- 23; Model - 25 |

**Results from Saikuro**

Flay analyzes ruby code for structural similarities. Differences in literal values, names, whitespace, and programming style are all ignored. Code that flay reports as similar is a good candidate for refactoring. The total score of the entire app is 2438, and will reduce if duplicate code is refactored. Churn finds the files that change frequently. Changing a file often is a bad sign. However, in our SUT, only user model is changed frequently, which is a good sign.

After executing the tool, the report is generated in the web browser. The results can be found in the Test Support folder under the folder metric\_fu.

Installation and Usage of the tool is mentioned in detailed in ‘Installation\_Usage\_Instructions’ file in the Tool Support folder of the repository.

* **Reflections from lessons learned -**

The files that we have considered showed high flog score - which means difficult to test, which then justifies the low code coverage. Results from Flay are utilitarian because they find out the duplicate code which further helps in refactoring.

We missed this in manual testing, hence such tools should be used. Reek finds out the common code smells in ruby code - which is not possible by manual inspection. Reek examines the code and cautions you about the design issues.

According to their checks, a method is too long if there are more than 20 lines of code. Moreover, the cyclomatic complexity of the method should equal to or less than 8.

**Dynamic Testing:**

* **Definition**

Dynamic testing, the other main category of software testing methods, involves interaction with the program while it runs.

* **Participants**

1. Sayali Pendharkar
2. Sanjay Kalla

Our team wanted to explore grounds of User Interface Testing and desired to include that in our project. This gives an edge to our project. Firstly, we would like to introduce UI testing.

This all has been done using capybara and selenium which is our next topic in hand.

We have used an application called Capybara, which basically is Acceptance Test framework for web applications which helps us test web applications by simulating how a real user would interact with the app. Furthermore, capybara uses different web drivers for example capybara-webkit, poltergeist, selenium etc.

For our project we have chosen Selenium due to its high popularity and demand. Selenium automates browsers. The above analysis has been orderly and concisely presented in form of html reports which include following details:

1. Number of test cases passed

2. Number of test cases failed

3. Time taken in execution.

For the Rspec part of the analysis we engaged time to run the test cases and individually analyze pass or fail conditions. These reports are very instructive and self-explanatory.

* **Results**

We have ensured to take as many scenarios and tasks as possible in the limited time constraint. We have taken in account scenarios related to each module in our code, like for say, dashboard module had two test cases which helps us to check all the tabs and also links the task opportunity accounts, We were able to produce 100% test case results. For Task module we were able to produce five different test cases relating to task assigned to signed user, task for other user, view task, edit task, delete task and again successfully procured 100% test case results.

For our next module i.e. Leads we proposed six test cases relating to View leads, create leads, edit leads, delete leads, remember the comments written in the leads and finally search leads. In this one of our proposed test case failed as a result we were not able to achieve 100% test results. For the user module we generated a single test case relating to admin which can create a user this was easily executed and resulted in “pass”. The basic tasks that are performed by the users are tested and almost all the test cases pass.

Installation and Usage of the tool is mentioned in detailed in ‘Installation\_Usage\_Instructions’ file in the Tool Support folder of the repository.

* **Reflection from lessons learned**

There are various levels of Dynamic Testing Techniques which are as follows:

* Unit Testing
* Integration Testing
* System Testing
* Acceptance Testing

The reason for choosing selenium was that monotonous web-based administration tasks should be automated as well as test cases cover most of the tasks.

**Quality Assurance**

1. **Rack-mini-profiler:**

* **Definition**

Rack: A Rack provides a minimal, modular and adaptable interface for developing web applications in Ruby. By wrapping HTTP requests and responses in the simplest way possible, it unifies and distills the API for web servers, web frameworks, and software in between (the so-called middleware) into a single m.

Mini profiler: In a mini profiler, page loading speed displayed on every page. Optimize while you develop, performance is a feature.

* **Participants**

1. Sayali Pendharkar – Installing and executing the tool.

* **Methodology**

Rack mini-profiler is a middleware that displays speed badge for every html page. Designed to work both in production and in development.

* **Results**

After successful installation and execution of the tool, at every page load we come to know the time taken by each query to load the page. This time is displayed in milliseconds. This information is displayed at the top left corner of the page.

The output is the time taken to load page in milliseconds. On clicking the displayed value, we get the breakdown of how much time was spent in each part of rails application. If we click the number of SQL queries executed, we get to know what SQL queries were performed and its stack trace.

Installation and Usage of the tool is mentioned in detailed in ‘Installation\_Usage\_Instructions’ file in the Tool Support folder of the repository.

* **Reflection from lessons learned**

Rack mini profiler is a performance tool and measures the performance with respect to the SQL queries, memory allocation and CPU time. Rack-Mini-Profiler gives you a constant reminder of how long each page takes to load. That helps you learn more about how your application is performing. You’ll build an intuitive sense of which pages are slow, and which ones are fast. You’ll start to notice when a page takes surprisingly long to render. And you can start fixing it right away, while it’s still on your mind. This helps in improving the performance of the application while developing it, rather than measuring the performance of the app at the end after development.

1. **GitStats:**

* **Definition**

It is a statistics generator for git (a distributed revision control system) repositories. It examines the repository and produces some interesting statistics from the history of it. Currently HTML is the only output format.

* **Participants**

1. Sayali Pendharkar
2. Sanjay Kalla

* **Methodology**

After executing the tool, the output is generated in a web browser. The reports generated are based on the criteria of activities performed, authors, lines of code changed etc.

* **Results**

The reports were generated in 7 seconds. There were a total 4394 commits, 4.3 commits per day. Among 111 authors involved, there were 39.6 commits per author. Maximum commits were made during the mid-week. The application was mostly developed during 2010-2012. Time slots where maximum commit history is seen are between 23.00 hrs to 6.00 hrs.

Total lines of code committed is approximately 64312 and 1024 files were committed, most of which were Ruby files (.rb) followed by haml, js, yml and coffee. Maximum commits by Mike Dvorkin.

After executing the tool, the report is generated in the web browser. The results can be found in the Test Support folder under the folder gitstats.

Installation and Usage of the tool is mentioned in detailed in ‘Installation\_Usage\_Instructions’ file in the Tool Support folder of the repository.

* **Reflection from lessons learned**

Considering the time slots and different geographical positions of the collaborators of the code we got to understand the complexity of globalizing software engineering some of challenges included:

* + 1. The communication due to different geographical locations.
    2. Testing and maintenance.

However, as a part of our learning we also got to know the assets of being globalized which were:

* + 1. Access to more resources at affordable costs
    2. Knowledge sharing
    3. Shorter lifecycle.

These were the key learnings out of this project.

**Hardware and Environment Requirements**

The SUT can be installed on Linux, MAC OS and Windows. The master branch is now on Rails 4.2. However, there is a Rails 3.2 branch available if you still need to use it.

Ruby 2.2 (master branch / rails 4 versions only)

MySQL v4.1.1 or later (v5+ is recommended), SQLite v3.4 or later, or Postgres 8.4.8 or later.

**Test Schedule:**

Our group has agreed on dividing and managing our goals and setting respective benchmarks in a following manner:

For our first release, we agreed upon running through our code and performing manual inspection for four different modules of our SUT.

For our second release, we consented upon doing multiple things

Firstly, comparing our manual inspection report with results obtained after running our code through a static analyzer tool namely Rubocop. Secondly, to find the Code coverage of the SUT.

For our third release, we tested the SUT by automating the web browsers using Selenium

And also run RSpec test cases for our modules and presenting highly precise, informative and meaningful reports for the same.

Our final release focused mostly on the Quality Assurance part of our project. The focus was on finding bottlenecks of the application by using a tool, Rack-Mini-Profiler. We also liked to touch upon the Software Configuration Management aspect, which analyses the Github commits.Along with this, we also used Metric Fu, a compilation of several tools to help find areas of code that could be improved.

**Tools and Approval:**

|  |  |
| --- | --- |
| **Tools** | **Approved By** |
| Rubocop | Vignesh Subbian |
| SimpleCov | Vignesh Subbian |
| Selenium | Vignesh Subbian |
| Rack-mini-profiler | Vignesh Subbian |
| GitStats | Vignesh Subbian |
| Metric Fu | Vignesh Subbian |