Fayngersh / Saint-Amour

Final Project

Software Quality Assurance Plan

Version: (1.0.4) **Date: (05/14/2016)**

Document History and Distribution

1. Revision History

Revision #	Revision Date	Description of Change	Author
1.0.0	April 25	Initial Plan	The team
1.0.1	May 2	Agreed on a package	Firmin
1.0.1	May 2	Agreed on a set of tools to use	Fima
1.0.2	May 4	Agreed on approach	Fima
1.0.3	May 8	Agreed on pass/fail criteria	Firmin
1.0.4	May 14	Completed plan	The team

TABLE OF CONTENTS

1. INTRODUCTION

Our project of choice is FreeCol. FreeCol is a large open-source game which can be found on sourceforge. Since it is a large project and we have a limited amount of time, we plan on using code metric tools to focus our testing efforts on one package (common.model). Once we have determined that package, we will improve the quality of the code (through various tools), and add to the existing unit tests as needed.

1.1 Objectives

Since this is a large project, and we are a small team with a short amount of time, our goal is to focus on one package, the common.model package. The code quality of the package will be improved using various tools. Finally, units tests will be added and the existing tests for that package will be enhanced to achieve an optimal amount of code coverage.

1.2 Testing Strategy

Our testing will mostly stay at the unit / component level. Testing will be focused on the common.model package. The package will be split in two parts: classes from A-M and N-Z. The code in these parts will be refactored and improved via code improvement tools such as CheckStyle and UCD. CodePro will be used to generate test cases for untested classes in the package. We will then look at the coverage of the existing tests and add additional tests as needed.

1.3 Scope

Our test plan will change as testing is performed. When new tools or methods are used, the plan will be updated accordingly. Once we have decided on where to focus our efforts the plan will be updated accordingly.

1.5 Definitions and Acronyms

UCD: Unnecessary Code Detector

Coverage: The degree to which the source code is tested by a particular test suite

2. TEST ITEMS

2.1 Program Modules

Unit Testing and refactoring will be performed on the common.model package.

2.2 User Procedures

Our documentation primarily involves reports that were generated from our metrics and improvement tools. Consequently, no testing will be performed on these reports, as they are all highly specialized and as correct as the tool can possibly make them.

3. FEATURES TO BE TESTED

Classes belonging to the common.model package are to be tested. We are testing these classes in particular based on analytics provided by the JDepend and CodePro tools. We are only testing these classes due to a lack of time/resources.

4. FEATURES NOT TO BE TESTED

GUI: The common.model package does not involve the GUI, and we did not have the time or the manpower to test everything.

Server: The common.model package does not involve the Server, and we did not have the time or the manpower to test everything.

Gameplay: Our aim was not to test or change gameplay, but to improve the quality of the code and tests within the common.model package.

Interfaces: Our aim was not to test the interfaces but to improve and test the code of the common.model package

5. APPROACH

Since this is a large application, our team will look to test one package in particular. We have decided to focus our efforts on the common.model package of the FreeCol project. The JDepend tool was used in order to determine which package had the most afferent couplings. CodePro Analytix tool was used in order to view average cyclomatic complexity of the packages, which also played a role in choosing a package.

Prior to actual testing, our team would divide the common.model package in half, with one member focused on classes from A-M and the other N-Z. Each member would use their respective tools in order to refactor the package to the best of their abilities. The member working on A-M would use UCD in order to detect unnecessary code and give solutions to them, as well as Robusta in order to detect and correct code smells, or indicators of poor design. The member working on N-Z would use the PMD and CheckStyle tools in order to adjust the code in order to adhere to a set of coding standards. This includes removing unused imports, renaming variables that are too long or too short, adding curly braces to if/else/for statements, removing unused modifiers, and correcting various data flow anomalies.

The overall approach to testing this package consists of comparing the existing test directory to the common.model package to see which, if any classes were yet to be tested. If such a case exists, we would use CodePro to generate a Junit test for that class. In addition, we would use the EclEmma tool to view the coverage of the test class over the class it is testing. We would then manually revise the test class and add further tests in order to achieve best possible coverage, rerunning the EclEmma tool to check if our changes had any effect.

5.1 Component Testing

CodePro will be used as a starting point for component testing. Once a test suite has been generated via CodePro, we will work to enhance those tests to to achieve maximum or almost-maximum coverage of the classes they are testing. The coverage will be measured using the EclEmma tool.

5.2 Integration Testing

We have decided to forego integration testing due in part to time constraints, as well as the fact that we are focusing primarily on one package. We lack the resources and knowledge of the entire project to perform effective integration testing.

5.3 Interface Testing

We have decided to forego interface testing. See section 5.2 for reasoning.

5.4 Security Testing

We have decided to forego security testing. See section 5.2 for reasoning.

5.5 Performance Testing

We have decided to forego performance testing. See section 5.2 for reasoning.

5.6 Regression Testing

Our team performed regression testing in an unconventional manner. Our version of such testing consisted of frequently running the “testall” option under the ant build in order to ensure that none of our changes would cause the tests to fail.

5.7 Acceptance Testing

We have decided to forego acceptance testing. See section 5.2 for reasoning. Also, we didn’t have clients with acceptance criteria to judge if our project was acceptable.

5.8 Beta Testing

We have decided to forego beta testing. This is due in part to not having customers to test the application.

6. PASS / FAIL CRITERIA

6.1 Suspension Criteria

When no more faults / bugs are found testing will be suspended

6.2 Resumption Criteria

Testing will resume when any errors / bugs are introduced.

6.3 Approval Criteria

Code coverage of 90% and the absence of bad code smells make up the approval criteria.

7. TESTING PROCESS

7.1 Test Deliverables

Metric reports
Coverage reports
Code improvement reports

7.2 Testing Tasks

Enhancing existing tests: Knowledge of Junit testing needed
Adding new tests as needed: Knowledge of CodePro and Junit testing needed
Regression testing (after refactoring): Knowledge of ant build with “testall” needed

7.3 Responsibilities

Management: Firmin and Fima
Designing tests: Firmin and Fima
Checking tests: Firmin and Fima
Developers: Firmin and Fima

7.4 Resources

Refactoring A-M: Firmin, using UCD and Robusta
Refactoring N-Z: Fima, using PMD and CheckStyle
Unit Tests: Test generation using CodePro. Test coverage using EclEmma

7.5 Schedule

Refactoring A-M: 12 hours

Refactoring N-Z: 12 hours

Unit tests: 4 hours

8. ENVIRONMENTAL REQUIREMENTS

8.1 Hardware

Any computer capable of running Eclipse Juno with at least 2 gigabytes of memory should suffice. Besides that, an internet connection is required for collaboration through our repository.

8.2 Software

The software requirements for this project are to possess the Eclipse IDE for java development, as well as the necessary plugins, such as CodePro and EclEmma. Each member should also have installed the plugins necessary for their individual refactoring/testing requirements.

8.3 Security

No such requirements exist for our project.

8.4 Tools

Eclipse: IDE of choice

JDepend: Metrics tool, was used to focus our efforts

UCD: Unnecessary Code Detector, finds dead code

Robusta: Bad code smells detector

PMD: Source code analyzer, finds common programming flaws

CheckStyle: Coding standard tool, automates the process of checking code to enforce a coding standard.

CodePro: Metrics, unit testing, and test generation tool

EclEmma: Coverage tool, used for analyzing code coverage of unit tests.

8.5 Risks and Assumptions

Constraints on testing:

- Time constraints
- Team of only two members

Risks:

- Working in different environments: We each had different computers, different operating systems. That often results in an aspect if a project working on one machine but not the other. The contingency plan is that if an issue comes up, research a solution and get it working. Consult your teammates for advice/assistance with the issue if applicable.
- Schedule: There may not be enough time to do significant testing. There is no way to circumvent this issue besides simply doing as much as possible with the time we have.
- Code Repository: There may be conflicting code when multiple people are working on the same project. To circumvent this issue, the merge tool will be used so that no one's changes are lost in the development process.

9. CHANGE MANAGEMENT PROCEDURES

The use of any new tools or any new testing methods will initiate a change. The changes will be added to the plan by one team member and the other will review those changes. The addition of a tool will also be discussed within the team until a concise decision is made as to the applicability and effectiveness of the tool.