# COSC 603 - Final Project

# FreeCol Refactoring

## Software Quality Assurance Plan

**Version: (1.0)** **Date: (05/16/2016)**

**Jessica Rudolph**

**Faye Vincent**

**Matthew Herrmann**

**<u>Table of Contents</u>**

## 1. Introduction

The aim of our software test plan is to improve the quality of a large, open-source software project. For testing purposes, we chose to improve the software quality of the game FreeCol. This game is a turn-based strategy game based on the old game Colonization. Our software testing strategies discussed in the assurance plan herein will attest the knowledge we have gained in software testing and maintenance this semester. Further, the plan specifies the items that were tested and the type of testing that was performed.

### 1.1 Objectives

The scope of the testing activities extends to many components of this application. Components range from core debugging facilities, GUI, Core system features, high level game logic, and low level game logic. The approach used was for static inspection of the code, and reviewing its architecture. The team reviewed the components and identified areas which we thought could potentially be tip of the icebergs for bugs - so our selection was stochastic but methodical in its objective. The major work items which will be delivered in the milestones are our identified classes, our two tools, and a thoughtfully tested solution loaded on Github which the relevant artifacts attached. Weekly meetings will held to discuss major issues and possible blockers for team members. Our team will work independently, but freely engage in regular conversation in an attempt to share knowledge and learn from each other's experiences.

### 1.2 Scope

The scope of the testing activities extends to many components of this application. Components range from core debugging facilities, GUI, Core system features, high level game logic, and low level game logic. The approach used was for static inspection of the code, and reviewing its architecture. The team reviewed the components and identified areas which we thought could potentially be tip of the icebergs for bugs - so our selection was stochastic but methodical in its objective. The major work items which will be delivered in the milestones are our identified classes, our two tools, and a thoughtfully tested solution loaded on Github which the relevant artifacts attached. The code we post should be improved, able to build, bootable, and functioning. Weekly meetings will held to discuss major issues and possible blockers for team members. Our team will work independently, but freely engage in regular conversation in an attempt to share knowledge and learn from each other's experiences.

## 2. Features To Be Tested

Each of the software tester's of the project herein chose their own, unique method to prioritize the features to be tested. Each test group in the following sections represent a set of features that one software tester chose to refactor and test for the purposes of the project.

### 2.1 Test Group 1 [Jess Rudolph]

The set of features to be tested included in this test group includes the following classes: DebugUtils.java, Game.java, ResourceManager.java, ResourceFactory.java, and CollectionUtils.java. These classes were chosen based on a combination of factors, such as how frequently the classes were used by users of the system, the number of lines of code, the computed metrics, and the number of lines of dead code. There is a good amount of documentation about FreeCol that was utilized to determine the frequency and importance the classes chosen for this specific test group. Then, metrics were ran on the code to provide a brief overview and synopsis of the quality of the code.

Once the above data was collected, the classes were narrowed down by those that were in the most need of refactoring and restructuring the code. The class DebugUtils.java class was chosen because it had a high cyclomatic complexity of 3.96 and contained 4 methods that were determined to be dead code with about 1,300 lines of code. The class Game.java was chosen not only becuase of its importance for the game to operate, but also because the class required some major refactoring and had a number of document warnings. Additionally, the Game. java class had a cyclomatic complexity of 1.86 and 2 methods containing dead code, with more than 1,400 lines of code. ResourceManager.java was elected as a class to conduct testing on because it had a cyclomatic complexity of 3.90 and several document warnings. It had fewer lines of code than the other classes with only about 500 lines of code, which presented a new challenge. The final class chosen in this test group was CollectionUtils.java because it had a cyclomatic complexity of 1.20 and an 10 methods containing dead code with over 1,400 lines of code to test and refactor.

| Feature | Package Name | Class Name | Avg. Cyclomatic Complexity | # Methods of Dead Code | # Lines of Code |
|---|---|---|---|---|---|
| Jess | | | | | |
| 1 | net.sf.freecol.common.debug | DebugUtils.java | 3.96 | 4 methods | 1,300 lines |
| 2 | net.sf.freecol.common.resources | ResourceManager.java | 3.90 | 4 methods | 500 lines |

| | | | | | |
|---|---|---|---|---|---|
| 3 | net.sf.freecol.common.resources | ResourceFactory.java | 3.90 | 0 methods | 300 lines |
| 4 | net.sf.freecol.common.model | Game.java | 1.86 | 2 methods | 1,400 lines |
| 5 | net.sf.freecol.common.util | CollectionUtils.java | 1.20 | 10 methods | 1.450 lines |

### 2.2 Test Group 2 [Faye Vincent]

The set of features to be analyzed included in this test group includes the below classes. Theses classes were chosen based upon FreeCol problem/bug issue #2959. This issue is concerning the "Strange Negotiation with Europeans". The problem is reproducible but inconsistent. The problem scenario is during negotiation with the French gold is demand and the play responds by offering the gold (#103). The gold is offered but is not accept until it is offered several times. Once the gold is accepted the French declare war on the Player. I choose these classes based on their relationship to the issue.

For testing the code was built, but I was unable to run the program using the debugger. To assist me in finding the bug in the code I used several tools. I use the CodePro Tools to determine the Cyclomatic Complexity of the code and Find Bugs to determine if there were bugs in the classes. I found that the packages net.sf.freecol.common.networking and net.sf.freecol.common.model did contain bugs but not in any of the classes below.

| **Feature** | **Package Name** | **Class Name** | **Avg. Cyclomatic Complexity** | **# Lines of Code** |
|---|---|---|---|---|
| Faye | | | | |
| 1 | net.sf.freecol.common.model | DiplomaticTrade.java | 53 | 544 lines |
| 2 | net.sf.freecol.common.model | Stance.java | 51 | 195 lines |
| 3 | net.sf.freecol.common.model | StanceTradeItem.java | 28 | 210 lines |
| 4 | net.sf.freecol.common.networking | DOMMessage.java | 121 | 945 lines |
| 5 | net.sf.freecol.common.networking | DiplomaticMessage.java | 66 | 411 lines |

### 2.3 Test Group 3 [Matthew Herrmann]

The classes below were chosen because of the interconnection with the greater software system. As a game with a rich graphical user interface it was essential to develop tests as this it main mode of interaction with most users - and is essential to the game experience. If a regression is introduced it can cause serious system issues, and quickly render the application unusable.

The classes were determined based off a combination of direct user interaction, the ability to add large regression inadvertently, and core system logic. If a bug were introduced to these classes the impact could be catastrophic to the performance of the game.

Using consistent conventions while coding helps to ensure ease in reading, navigation, maintenance, and development of code. A tool called Checkstyle which is a Eclipse plugin offers static analysis features which help to identify troublesome code which deviates too much from some set parameters. Code that is deemed to have deviated too much is flagged and there are some quick fixes which can be applied to help to restructure the code without changing its meaning. This tool was run against all five identified classes, and changes were recommended and applied against **`Direction.java`** and **`UnitAnimationAction.java`**. These two sets of changes made adjustments to the code, the changes didn't change the logic of the code, simply changed the spacing to make it more readable (for the UnitAnimationAction.java file). When Checkstyle was run against **`Direction.java`** it flagged the use of the **`this`** keyword. It also rearranged the use of the **`final`** keyword.

From the classes below the average cyclomatic complexity was also reduced in **`UnitAttackAnimation`** from the original cyclomatic complexity of 4.5 and was reduced to 2.39.

PMD was run against all of the classes below and and the output of those changes can be found in /doc/logs/PMD_Output. PMD was used to help identify issues in the code that require correction, however there is no expected performance increase from these changes.

| **Feature** | **Package Name** | **Class Name** | **Avg. Cyclomatic Complexity** | **# Lines of Code** |
|---|---|---|---|---|
| Matt | | | | |
| 1 | net.sf.freecol.client.gui | GUIMessage | 1 | 80 lines |
| 2 | net.sf.freecol.client.gui | ChatDisplay | 2 | 147 lines |
| 3 | net.sf.freecol.client.gui | ChoiceItem | 1.31 | 216 lines |
| 4 | net.sf.freecol.client.FreeColClient | UnitAttackAnimation | 4.5 (Improved to 2.39) | 156 lines |
| 5 | net.sf.freecol.common.model | Direction | 1.84 | 253 lines |

## 3. Features Not To Be Tested

Features not tested in this effort, due to time constraints are performance testing, full spectrum end-to-end testing, and complex configuration integration testing. Performance testing when done needs to be done comprehensively to make it useful in most cases - and we simply didn't have the time to perform that. For end-to-end testing the issue was resources - some of the testing would actually require extensive game play and complex changes to the system in order to determine if the code is being properly exercised.

Complex configuration testing - for example if over a network, has lots of security implications as well as performance implications. It was considered outside the scope of this assignment and we only focused on the standalone features and use.

## 4. Approach

Our approach to testing will start in the selection of the classes within this project. These diverse of classes represent a variety of components within the greater system. Due to resource constraints, full end-to-end testing is nearly impossible. To maximize the coverage of the system we've identified this wide variety of classes. within the project. We hope, as we said in the Objective section, to this will reveal the tip of an iceberg and find the larger, and wide ranging system issues. The core types of tools we will use are automated code refactoring tools and bug detectors - in most cases static analyzer to identify troublesome areas where we can continue to search for bugs if we had additional resources and time.

We will also attempt to identify certain hotspots in the code which may be too complex for the purposes they are servicing and need to be refactored. We will use these tools in combination to improve the overall quality of the code. We will refactor areas in need of attention to help improve the maintainability of this code as well. We will develop unit tests to help detect regressions as this system continues in it's test cycle. Additionally we will maintain our core artifacts in Github, and use Google Docs to collaborate on documents.

## 5 Component Testing

### 5.1 Specific Components

#### 5.1.1 Test Group 1 [Jess Rudolph]

To execute component testing, the metrics were analyzed to determine the extent of testing and refactoring the code for each of the features in Test Group 1. The first step was refactoring the code to reduce the cyclomatic complexity of each class. To do this, the "Extract Method" option was used. Next, the dead code was removed from the feature. Each of these changes had to be made with care to ensure that removal and refactoring of code within a specific feature didn't affect other classes. After each change, it was ensured that no new errors were created in any of the other classes. The next step was using the AutoRefactor tool and the Spartan Refactoring tool to refactor the code and reduce the cyclomatic complexity even further. If the cyclomatic complexity was increased while utilizing these refactoring tools, the code was manually refactored to reduce the cyclomatic complexity. The final step was utilizing CodePro's "Audit Code" function to correct specific parts of the code. All of the medium and high risk items were 100% resolved in the methods for Test Group 1 when auditing for CodePro Core, Potential Errors and Refactorings, and Security risks. Refactoring the code was complete when there were no more errors in the class, the cyclomatic complexity was reduced, and the medium and high risk auditing flags were removed.

#### 5.1.3 Test Group 3 [Matthew Herrmann]

Componont testing for the classes ChatDisplay.java, ChoiceItem.java, GUIMessage.java, UnitAttackAnimation.java, and Direction.java was execuited after analysis using code auditing tools - specifically CodePro. CodePro had the greatest facilities to identifying static issues - it also allowed for the generation and execution of unit tests which can help identify regressions and prevent major issues from creeping into the codebase. Classes were reviewed based off the number of flagged issues, the greatest concern was the cyclomatic complexity. The class that required immediate attention for cyclomatic complexity was UnitAttackAnimation. This class was particularly in need because initially it had a cyclomatic complexity of 4.5.

The tools that were used beyond WindowTester Pro which required that the code receive a customization in the main allowing it to boot within Eclipse. The modification was how the Jar URL was handled. WindowsTest Pro is outdated - and would crash out for a variety of reasons with the same input from the user. There is a good chance this was a resource allocation issue due to the machine being developed on being a VM, the JVM needed to run Eclipse, FreeCol, and WindowsTest Pro - it was extremely resource intensive. A crash log was included in our main deliverable for future triaging.

Checkstyle was used to help remove the random deviations from the coding style. The default configuration was used to do the static analysis. Directions.java and UnitAttackAnimation were flagged and had fixes applied. The other three classes ChatDisplay.java, ChoiceItem.java, and GUIMessages.java were checked - however there were no issues which required fixing.

PDM was used to try to discover major issues that could potentially be a problem in this large scale testing and QA effort. The issues were identified were in some cases considered urgent issues. These issues will be resolved with large scale refactoring later - however other static analysis skipped them in most cases.

AutoFactory, while selected as a tool by another team member, was also used in this process, along with the native refactoring capabilities that are already in the Eclipse. These refactorings will be used to go beyond the currently modified sections of code to include more classes across components to hopefully identify trouble spots.

## 5.2 Integration Testing

Given unlimited time and resources this application should be tested in a variety of configurations and uses. Integration Testing took place on a very small scale - however if this application were a commercial endeavor or if this application were going to be placed in a much larger production environment it would require extensive and regular integration testing. Certain changes which were made over the lifecycle of this project had to be rolled out, because they failed some basic smoke testing - the system couldn't boot and run.

## 5.3 Interface Testing

WindowTester Pro as recommended was attempted to be used - however we were unsuccessful. This tool was developed for a much older version of Eclipse. It also had difficulty capturing changes within the game - and then the game would simply freeze on the menu screen when they would both attempt to be run. There are some possible issues for this - on hypothesis which were unable to test due to time constraints was the a configuration issue with the host machine (using a Ubuntu 16.10 version which is hosted on a virtual machine itself could be to blame for the failed mouse capture). Additionally, the game was being run on a JVM that was on a VM with low RAM allocated to it. With the increased RAM there were still issues with - and the game consistently would crash out, or WindowTester Pro would crash. An example log is located in /doc/logs which contains a JVM dump of the error.

The ability to boot the application and run it within Eclipse was added to the FreeCol.java file which is the main for running this project. This allows for certain plugins to be run against the code. We developed this change when trying to use WindowTest Pro - but we were unsuccessful due to the reasons mentioned above.

## 5.4 Security Testing

### 5.4.1 Test Group 1 [Jess Rudolph]

CodePro Tools Auditing was used to resolve the Security warnings in the features/methods for Test Group 1.

### 5.4.3 Test Group 3 [Matthew Herrmann]

CodePro Tools Auditing was used to across ChatDisplay.java, ChoiceItem.java, GUIMessage.java, UnitAttackAnimation.java, and Direction.java classes. This tool was used to identify the areas where there may be potential issues with the code logic which requires a change. The output from these reports can be viewed in the doc/AuditResults.

## 5.5 Performance Testing

After implementing our changes there were no noticeable performance issues to the code when the application booted. The goal for our time horizon was not to introducing any major obvious performance problems. Although we reduced the complexity of some of the methods with the application, on our time constraints we did not spend time doing extensive optimizations. An area that could probably be improved in the memory usage - this is due to the overserve behavior of the system abruptly crashing out for different reasons when attempting to perform certain specific tasks - instead of crashing out for the same method each time. Given more time, we would investigate this in by attaching a profiler to the JVM to look for spikes under load. We would also use a GUI automation tool (WindowTest Pro would crash out) to help simulate intense user interaction on the system.

## 5.6 Regression Testing

Unit tests were developed to help reduce the chance of regressions being introduced into the codebase. The CodePro suite by Google was used to develop unit tests for ChatDisplay, ChoiceItem, GUIMessage, UnitAttackAnimation, and Direction classes.These tests themselves need additional refactoring and expansion to remove bugs and to ensure the maximum amount of code is being covered but the tests. This helps to ensure that in the future less bugs are potentially introduced. The results of these tests are located under the doc directory in the subdirectory of test. Additionally unit tests will be expanded to other classes to help ensure that they meet some nonzero amount of coverage so that they don't break in the future should another developer introduce changes, and the bugs that could come with those changes.

### 5.7 Acceptance Testing

For the purposes of this assignment we believe that making our changes and having the system successfully boot and run with some simple operations within the game (ex., being able to make position changes) is our level of acceptable performance. If this game were going into a production environment or being sold there would need to be extensive testing of all major functions for both performing their expected function - and that there are no major performance issues in multi user environments. We would also request specifications for the platform that the game is run on to help minimize the chance of performance issues bogging the system down.

## 6. Pass / Fail Criteria

The pass/fail criteria for project is for all members to be able to analyze their code, make changes if needed - and have those changes improve the code and not introduce obvious stability issues. Essentially, can the game be built, booted, and run with changes to the codebase. This will be a success if we can make improvements to the codebase, learn more about it - particularly its weak points where bugs could creep in sometime in the future. The improvements and analysis from this project will help throughout the lifecycle of the FreeCol game. The artifacts within this document and within the /docs directory will aid in leveraging this for future work. We will deem this project a failure if we are unable to learn or to improve the overall codebase. If resources of people and time were not constraining use we would develop large scale unit tests for this application. We would also develop custom integration test suites to test the application in ways we cannot with the allotted time. The application would need to be tested not just in it's standalone mode - but also following the client model which is available.

## 7 Test Deliverables

### 7.1 High Level Group Breakdown

### 7.1.1 Test Group 1 [Jess Rudolph]

| Feature | Package Name | Class Name | Avg. Cyclomatic Complexity | # Methods of Dead Code | # Lines of Code |
|---|---|---|---|---|---|
| Jess | | | | | |
| 1 | net.sf.freecol.common.debug | DebugUtils.java | 3.12 | 0 methods | 904 lines |
| 2 | net.sf.freecol.common.resources | ResourceManager.java | 1.94 | 0  methods | 302 lines |
| 3 | net.sf.freecol.common.resources | ResourceFactory.java | 3.63 | 0 methods | 162 lines |
| 4 | net.sf.freecol.common.model | Game.java | 1.82 | 0 methods | 644 lines |
| 5 | net.sf.freecol.common.util | CollectionUtils.java | 1.20 | 8 methods | 1,300 lines |

### 7.1.2 Test Group 3 [Matthew Herrmann]

The researched and modified classes below will be included in the repository. Audit logs, reports, metric results, crash logs will be located under the /doc directory on Github for ChatDisplay.java, ChoiceItem.java, GUIMessage.java, UnitAttackAnimation.java, and Direction.java. Generated test cases will also included in the code and will only be removed if they contain auto generated artifacts that cause issues but aren't relevant to the code.

| Feature | Package Name | Class Name | Avg. Cyclomatic Complexity | # Lines of Code |
|---|---|---|---|---|
| Matt | | | | |
| 1 | net.sf.freecol.client.gui | GUIMessage | 1 | 80 lines |
| 2 | net.sf.freecol.client.gui | ChatDisplay | 2 | 147 lines |
| 3 | net.sf.freecol.client.gui | ChoiceItem | 1.31 | 216 lines |
| 4 | net.sf.freecol.client.FreeColClient | UnitAttackAnimation | 4.5 (Improved to 2.39) | 156 lines |
| 5 | net.sf.freecol.common.model | Direction | 1.84 | 253 lines |

## 7.2 Testing Tasks

The testing tasks were to use automated tools to review the code to help speed up the identification process. Detected issues should be refactored when feasible to do so, and these refactorings must survive a build, boot, and run of the system. Our hope is that our diverse selection of classes will help us visit many different facets of the system instead of a more focused effort which could leave components completely untouched.

## 7.3 Responsibilities

The group members in this task are responsible for working together to research the FreeCol application. Members will meet regularly, preferably weekly. Members will be responsive to questions directed to them over email. Members are encouraged to do personal research to resolve issues that they are experiencing. Members are expected to know the collaborative tools and do due diligence to become competent at using the tools prior to engaging in development.

## 7.4 Schedule

***Week 1:*** Team members should familiarize themselves with the assignment
***Week 2:*** Team members will be available with for a call to discuss the assignment/next steps
***Week 3:*** Team members should identify five classes, and two tools they will use
***Week 4:*** Team members should be prepared with their code ready researched, improved, and integrated into a functioning build. Members should contribute equally to the documentation and development of the collaborative artifacts.

## 8. Environmental Needs

Building and execution of this application took place on a VM running Ubuntu 16.10. This environment was hosted on OSX Yosemite. We believe that the performance issues experienced in some of the testing were attributable to the resource intensive nature of running the game, eclipse, additional debugging plugins running on their own instance of Eclipse (ex. WindowTest Pro). This type of configuration hogged resources and caused lots of stability issues. The VM was running with 2 gb of RAM on a machine with two virtual cores. The host machine which was also being used in tandem could have potentially introduced some of these instabilities however in the past the host machine use wasn't enough to significantly hog resources from the VM. Another team member was able to both build the fresh code, and run it from command line.

## 8.1 Tools

There were several Eclipse plug-ins and tools utilized in improving the quality of the software.

- **AutoRefactor [Selected by Jess Rudolph; Used by Jess Rudolph and Matthew Herrmann]**
  - AutoRefactor is an Eclipse plugin to automatically refactor Java code bases. The aim is to fix language/API usage in order to deliver smaller, more maintainable and more expressive code bases.
- **Spartan Refactoring [Selected by Jess Rudolph; Used by Jess Rudolph]**
  - Automatically find and correct fragments of code to make your Java's source code more efficient, shorter and more readable.
- **Google CodePro Tools [Used by Matthew Herrmann, Jess Rudolph, and Faye Vincent]**
  - CodePro AnalytiX is the premier Java software testing tool for Eclipse developers who want to be active participants in improving the quality and security of the code they produce. CodePro AnalytiX seamlessly integrates into the Eclipse environment, using automated source code analysis to pinpoint quality issues and security vulnerabilities before code reaches QA, or worse, production.
- **Checkstyle [Selected by Matthew Herrmann; Used by Matthew Herrmann]**
  - Checkstyle is a heuristic based Eclipse static analysis plugin which can identify and flag specific issues in the code, such as helps maintain consistent coding practices throughout the code. It helps enforce coding standards, which when applied consistently can make code far easier to understand, navigate, and maintain. Coding standards are not new, but helpful tools like Checkstyle are and their application is widespread in industry. Custom styles can also be used in the system, and are defined in configurations with Eclipse.
- **WindowTester Pro [Selected by Matthew Herrmann; Used by Matthew Herrmann]**
  - This is a GUI automation tool which is available to produce unit cases and has capture and replay capability. It is designed to integrate into Eclipse to help test software with GUI. This is helpful to speed up the development of test suites, and also speed up their execution. The only real alternatives are scripting or manual testing. WindowTester Pro is out of date of Eclipse. Encountered serious compatiability issues, and had to modifiy FreeCol.java **`main[]`** to get Eclipse to boot both the application (FreeCol) and the capture/replay recorded for WindowTester Pro.
- **PMD [Selected by Matthew Herrmann; Used by Matthew Herrmann]**
  - This is a static analysis based utility which helps identify major issues in terms of coding style which tend to both lead to and introduce bugs in the application. This was run against code to look for find areas where there could be improvement, and to log these changes for future work.
- **Find Bugs [Selected by Faye Vincent; Used by Faye Vincent and Matthew Herrmann]**
  - This is a static analyzing tool for Java which uses heuristics to identify possible poor coding practices. It uses rules and filters in the static analysis, and it is a powerful tool

which can help identify trends in a codebase, and becomes even more useful the larger the codebase is.

### 8.2 Risks and Assumptions

A risk of this process is having tools available that are compatible with the modern iterations of Eclipse (ex. Luna), and can easily perform analysis on the codebase, which although under active development is still an old codebase. While there are many tools available - many require extensive changes to the code to introduce the proper debugging functionality to the application. While remote debugging is possible, this wouldn't have been the appropriate application for such a tool. These tools accelerate the testing process - however on our timeline we'd need to reverse engineer code to build it in, which is a strain on our resource of time.

## 9. Change Management Procedures

To help control change we will have change management management procedures in place. Our change management procedures will start with a master codebase uploaded to get Github. We will use GitHub collaboratively. After our master code base has been uploaded we will each make sure that no other team member is working on a given branch and if so create our own branch. Our core objective is to make sure that our work does not impact the whether the application can be built or other team members work. If a team member is going to do high risk work, it is asked that they create their own branch - make their change there, use automated merging tools or git features to properly integrate past changes. Manual integration on a small scale is also an option. Versioning must be enabled at all times, and not member is allowed to force a change the repository without first consulting the other members, and have in a backup which will remain untouched should something go wrong the team has the opportunity to recover their work - and the work history.