**Software Test Plan (STP) Template**

Items that are intended to stay in as part of your document are in **bold**; explanatory comments are in *italic* text. Plain text is used where you might insert wording about your project.

This document is an annotated outline for a Software Test Plan, adapted from the IEEE Standard for Software Test Documentation (Std 829-1998).

Tailor as appropriate. Where you decide to omit a section, you might keep the header, but insert a comment saying why you omit the element.

**Smith-Day**

****

**image from:** [**https://en.wikipedia.org/wiki/FreeCol**](https://en.wikipedia.org/wiki/FreeCol)

**FreeCol Final Project**

**Software Quality Assurance Plan**

**Version: 1.7 Date: (05/17/2018)**

**Document History and Distribution**

1. Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Revision #** | **Revision Date** | **Description of Change** | **Author** |
| 1 | 4/11/2018 | Introduction and some of 3 &4 | Noah Day |
| 2 | 4/19/2018 | 5. Approach section, testing plans | Jeff Smith |
| 3 | 4/20/ 2018 | finishing 3 and 4, features to test | Noah Day |
| 4 | 4/23/2018 | 6 Criteria and 7 Testing process | Noah Day |
| 5 | 5/4/2018 | verifying current information | Both |
| 6 | 5/15/2018 | 8. Environment reqs | Both |
| 7 | 5/17/2018 | 9 change procedures | Jeff Smith |

**TABLE OF CONTENTS**

**1.** **Introduction 1**

**2.** **Test Items 2**

**3. Features To Be Tested 3**

**4. Features Not To Be Tested 3**

**5. Approach 3**

**6. Pass / Fail Criteria 5**

7. **Testing Process 5**

**8. Environmental Requirements 6**

**9. Change Management Procedures 7**

# INTRODUCTION

**The Software Test Plan (STP) is designed to prescribe the scope, approach, resources, and schedule of all testing activities. The plan must identify the items to be tested, the features to be tested, the types of testing to be performed, the personnel responsible for testing, the resources and schedule required to complete testing, and the risks associated with the plan**.

**1.1 Objectives**

The approach that we are planning on taking is more of a high level approach to the software. We are looking more at the parent classes which control more of the software than each individual classes, which could result in extensive testing.

**1.2 Testing Strategy**

**Testing is the process of analyzing a software item to detect the differences between existing and required conditions and to evaluate the features of the software item.**

The general level of testing our team decided to do was a broad, parent/controller class testing. This allows us to provide a more general level of testing, which in turn should allow us to make overall changes to the program, without having to test each individual item within the code. In regards to the general classes, we are planning on testing the general controller class, as well as the ones for the server, the GUI and the client. We are using the JUnit tests that were provided with the code a our benchmark to make sure that the code still operates as expected. We understand that this does not entirely check to make sure that the code is correct, but it will give us a definite marker for how the code should be operating.

**1.3 Scope**

**Testing will be performed at several points in the life cycle as the product is constructed. Testing is a very 'dependent' activity. As a result, test planning is a continuing activity performed throughout the system development life cycle. Test plans must be developed for each level of product testing.**

Our tests will be the control classes that control a large majority of the code. In this sense, it should allow us to test a large scope of the code rather than testing each individual class. Version control will be done from GitHub.

**1.5 Definitions and Acronyms**

*(Specify definitions of all terms and agency acronyms required to properly interpret the Software Test Plan. Reference may be made to the Glossary of Terms on the IRMC web page.)*

# TEST ITEMS

In this document, we will include:

* Requirements specification,
* Design specification,
* Features (availability, response time),
* Defect removal procedures, and
* Verification and validation plans.

**2.1 Program Modules**

*(Outline testing to be performed by the developer for each module being built.)*

**2.2 User Procedures**

*(Describe the testing to be performed on all user documentation to ensure that it is correct, complete, and comprehensive.)*

# 3. FEATURES TO BE TESTED

Software features to be tested include GUI classes, initially just the full GUI.java file. We are going to test in game controller and handler classes, as this is the biggest thing that affects users experiences. We also plan to test a lot of server side code to make sure everything works efficiently on the front end and the back end. In addition, we want to test the modeling classes in the SERVER files and the COMMON files. However, we do not plan to test all of these files, mostly parent classes or integration of said classes.

# 4. FEATURES NOT TO BE TESTED

Any features that originated from methods that are not explicitly controlling more than one method will not be directly tested. They will be tested with that of the parent methods. We also do not plan to test all subclasses in model files, mostly their parent classes.

# 5. APPROACH

The main goal we want to keep in mind while testing is to make sure we do regression testing. With each new test and code edit we want to make sure we did not cause any new faults. Firstly, we plan to run JUnit tests (mostly provided in code) to verify they work. Then we plan to do more component/ integration testing with static analysis tools, such as PMD, an extension that can be used in Eclipse. This test does not take long, but will point out bad design as well as inefficient code. Evaluating these test result should consist of refactoring methods and classes, as well as potential design changes to make sure the code is efficient.

It is important that we also do coverage tests of our test cases. It would not make sense to run these tests over and over again if they are not sufficient. The integration of EclEmma with Eclipse coverage tools makes EclEmma an attractive tool. We want to evaluate how much of the code we are covering with our test cases to determine if more needs to be done. Evaluating the coverage results shouldn’t be difficult with this tool, and it would be easy to see if more tests need to be added.

Next we plan to use a FindBugs tool to look at integration, security and performance testing. This tool will point out where there are concerns about the integration of all the pieces of the code and how they interact. It should also tell us if there are any potential memory issues by analyzing things like null pointers and dereferenced variables. This will point out many ‘bugs’ or potential code faults that need to be fixed. To verify we did not change the functionality, it would be imperative to also rerun our component testing after our FindBugs test.

Google’s CodePro is tool we plan to use for auditing the code, similar to PMD. this tool will help us spot and fix bugs faster to reduce time costs. CodePro would also be useful to generate test cases, however we already have component tests written. Creating these tests and running our tools on them may prove that they are indeed more efficient than what we have. This could take quite some time if it requires re-writing tests that we already have, but theoretically in a larger team, that time would be well spent.

**5.1 Component Testing**

We plan to do a lot of component tests throughout this process. Initially, we want to run the JUnit unit tests that we have to verify they all work. Then we want to run tools like PMD and EclEmma to determine if the source code is efficient and if the test code is sufficient. Inputs for EclEmma would be our test cases, and it would analyze how much of the code is actually being covered with these tests.

**5.2 Integration Testing**

We do not have a concrete integration testing tool that we plan to use. However, using SpotBugs to find potential faults and security risks should be sufficient to determine if the code is working well as an entity. Using PMD for design flaws and inefficient code will also help to determine if the software as a unit is operating as efficiently as possible.

**5.3 Interface Testing**

*No explicit interface testing done.*

**5.4 Security Testing**

SpotBugs will be run on the entire system, and if any code in the controller classes have bugs, those will be a main priority, as well as anything else that is a highest concern level risk.

**5.5 Performance Testing**

Performance testing in the sense of user side efficiency is not in the plan, however we do plan to test the performance and efficiency of the code. This should in term create a faster and better experience for the user.

**5.6 Regression Testing**

Regression testing is going to be a main focus with all of these tests. After different tools are used, we plan to rerun tests like PMD, SpotBugs and EclEmma to verify that the changes we are making are not affecting the design and efficiency, the security and system integration, or the coverage of our test cases.

**5.7 Acceptance Testing**

Acceptance testing is not a major part of this project in our mind. As this is open source software and we have set guidelines, it is difficult to seek customer approval. The only customers would be anyone we want to play our game (as users) and Dehlinger (as a “client”). We do not plan to do explicit acceptance testing throughout this process.

**5.8 Beta Testing**

*As above.*

# 6. PASS / FAIL CRITERIA

*(Specify the criteria to be used to determine whether each item has passed or failed testing.)*

**6.1 Suspension Criteria**

Suspension of all testing will commence when the code is not functioning properly. Using version control, the team will revert back to the last working product and resume testing

**6.2 Resumption Criteria**

Suspension will resume after a working version of the code has been restored.

**6.3 Approval Criteria**

Approval of the changes will be done after the code has been checked with the unit testing to make sure that it runs completely, and without any failures.

# 7. TESTING PROCESS

**7.1 Test Deliverables**

The provided ANT build commands export an XML file containing the results of the Unit tests. In addition, PMD provides markers within the code that show where areas of concern. We will also provide a document stating what bugs, identified by SpotBugs, were fixed and where they were in the code.

**7.2 Testing Tasks**

Each team member will run tests according to their addins to Eclipse, and then modify the code according the results of the testing. It would be efficient for each member to rerun their tests after the other has tested the code with a different tool, to maintain the progress made before. For example, after one member runs SpotBugs and fixes code, the other should run PMD on the edited code to make sure no design or efficiency flaws were created.

**7.3 Responsibilities**

Each group member is responsible for their respective testing features, in addition to making sure that the code is pushed to GitHub, in addition to having the code run correctly.

**7.4 Resources**

Resources for testing include, but are not limited to: Eclipse marketplace, developers website, ReadMe guide, and information from tests.

**7.5 Schedule**

April 1-14: Test plan document, review of code files, familiarization with software.

April 15-29: Individual team member testing

May 1-9: Regression testing (and retesting)

May 10-15: Final git commits, finish documentation, pray.

# 8. ENVIRONMENTAL REQUIREMENTS

**8.1 Hardware**

Computer 1:

* AMD A6-4455M APU, 2.10 GHz Processor
* HP Pavilion ca. 2014
* 4 GB RAM

Computer2:

* i7 Processor
* Nvidia GeoForce GTX 1060 Graphics Card
* 16 GB RAM

**8.2 Software**

* Eclipse
* GitHub

**8.3 Security**

The testing environment is done completely within Eclipse and the plugins used in Section 8.2. The updated code is then uploaded to GitHub to be merged between the testing team.

**8.4 Tools**

Tools being used in Eclipse are as follows:

* + CodePro
  + PMD
  + SpotBugs
  + Eclemma
  + Hansel
    - Did not import into eclipse

Each tool was used according to its’ respective user guide and support. All came from the Eclipse marketplace, or were installed through Eclipse.

**8.5 Risks and Assumptions**

There are several risks and assumptions that are made when taking on a task of this size. Not every resource/item is easily accessible to testing, and due to the time constraints, the tests may not completely cover every item or situation; therefore not all scenarios are completely tested for. The assumption though, is that the pre established tests cover all the important features of the game. Each risk is to be dealt with accordingly to how the team sees fit. In the case of any bugs, this would be to provide more tests cases in order to try and fix any errors that would arise from the test cases being absent. These test cases will be modeled after the user’s guide for each subsequent testing tool, and will be modified until they pass the unit testing. This would become more constrained the close to the due date that this occurs.

# 9. CHANGE MANAGEMENT PROCEDURES

The change in the testing process shall occur in the event that any given tools does not perform as expecting. In the case that this occurs, testing shall resume as normal. The initiation of the change shall occur at first from the team member, who will then confirm it with the team. From there, the team will confer about the new tool, and agree on the new implementation process based off of the user’s guide. If the new tool performs a similar task to that of the previous tool, it should be accepted. We should not be using several tools that all do the same exact thing as this boosts our time and efficiency costs.