**Software Test Plan Document**

**Date:** April 18, 2015

**Version:** v1.3

**Team: Team A**

**1.0INTRODUCTION**

*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 primary objective of testing an application in software development is to determine that the application works as intended by the specifications of the requirements and the design. When testing the HealthNet application, we will deploy our tests to this effect. By the end of development, unit tests and integration tests should be developed, operational, and passing, and acceptance tests will be determined.

**1.2 Testing Approach**

Specific individual testing approaches for each form of testing being used are documented in Clauses 4.0 to 4.3.

**1.3 Responsibilities**

All members of the team should be involved in the testing process. The testing coordinator is primarily responsible for the design and implementation of test cases; however, all members of the team who contribute to the code base will need to regularly review, add to, and execute test implementations based on what is needed during development. The test coordinator will also be responsible for producing preliminary acceptance tests. Test result logs will be built into the project and updated whenever tests are run. The testing coordinator may request assistance with test development from the other members of the team.

**1.4 Resources**

Testing design and implementation will be done by the testing coordinator and at least one other member of the team. Performing and validating tests will be done by those contributing to the application code base. We will use the unittest framework included with Django (see Clause 6.1 of the Requirements) to implement any unit, integration, and regression tests. The coverage package from the PyPI Python package repository (found at the following URL: https://pypi.python.org/pypi/coverage/4.0a2) will be used to analyze the amount of code under test.

**1.5 Risks and Assumptions**

Test implementations will be available as they are completed.

**2.0 FEATURES TO BE TESTED**

During testing, we will prepare tests for components that will be implemented per release. Prior to Release 1, we will aim to test the following system requirements: REQ-001, REQ-002, REQ-003, and REQ-008. Prior to Release 2, we will aim to test the following system requirements: REQ-004, REQ-005, REQ-006, REQ-007. In both releases, we will also examine the associated use cases. For more information on the specific system requirements and use cases to be tested per release, consult Clauses 5.0 and 4.0 respectively in the Requirements document. These features to be tested encompass all requirements that are specified, partitioned per release. This is because a failure to examine whether the software meets the requirements can potentially lead to software that does not meet the requirements.

**3.0 FEATURES NOT TO BE TESTED**

As stated in Clause 2.0, in the interest of thorough examination of our application, we will test every feature of the software that we build. However, it should be noted that any third-party frameworks used to build the software will not need to be tested beyond the way we use those frameworks.

With this said, this document reflects our goals for testing within the project. As such, although we will work towards maximizing the amount of features we test, in the interest of time, it will be unlikely that we complete testing of each and every required feature by each release. It should also be said that testing implementations may rely heavily on the full design and development of the object or objects they interface with; those tests may not be reflected in the code in those situations, although they can still be contained within the Test Case Tracker.

**4.0 SPECIFIC APPROACHES AND STRATEGIES**

During testing, we will use coverage utilities (specified in Clause 1.4) to provide a surface-level understanding of the percentage of code being tested. In order to track each test case, we will make use of the Test Case Tracker document for all fields of testing.

**4.1 Unit Testing - Component Testing**

The purpose of unit testing and component testing is to verify that the functionality of all components meet their design requirements. All unit tests will be tracked in the Test Case Tracker document organized per design component. All team members will eventually contribute to unit tests as they develop code, but the testing coordinator will be assigned the responsibility of assessing the completeness of the unit tests before running integration tests.

Unit-testing is, by necessity, a white-box testing method. In other words, our unit tests should be developed based on the internal unit structure. When writing a unit test, the developer should work on identifying all potential paths through the function they are testing. There are various ways to do this; formal UML 2.0 activity diagrams are preferred, although merely listing all possible situations on a piece of paper would take less time and be simpler for the developer. After determining all possible paths through the code, the developer should recognize certain equivalence classes of tests; that is, the developer should find all distinct testing scenarios that will walk through the program. Next, the developer should find inputs that fulfill each testing scenario and the output he or she expects. The developer should then implement the tests of the function that have been developed, clearly assigning each input value to a particular variable, accepting those variables as input into the function, and asserting that the result is what he or she expects. If a function has not been developed yet in the unit itself, the developer should implement the function in what is called a stub, which is effectively a placeholder for the function. Lastly, the test developer should run the test.

After creating the unit tests for a particular unit, the developer should record the test and its result in the Test Case Tracker document.

Evaluating our unit tests will be accomplished using a coverage analysis tool specified in Clause 1.4. As stated before, we are using white-box testing. We expect the unit tests to execute as many lines of the code as possible, so if our coverage tool identifies we are missing some segments of code, we know our tests are incomplete or have suffered regression. In this way, testers can test the completeness of their unit tests. However, this method does **not** test whether the unit tests include all invalid inputs possible; this is why re-examination of the tests by the test coordinator and by the developer is still necessary.

For information on specific unit test cases, refer to the Unit Tests sheet in the Test Case Tracker document.

**4.2 Integration Testing**

Integration tests will be implemented in order to determine whether system components can collaborate in the required ways. Integration tests will be conducted *if and only if* all of the components involved have passed their respective unit tests. We will be using a bottom-up approach wherein smaller integration “clusters” will be tested before larger “clusters”.

Integration tests should be created by first analyzing which classes interact with each other in each use case. The smallest interactions between classes should be identified, and developed, produced, and operated prior to larger collaborations with more components within the system. For each interaction, the test developer should create a collaboration diagram to ease identification of test cases. For each collaboration, the tester should use the diagram to develop test cases between the operating components. The test cases should then be documented in the Test Case Tracker and implemented in an integration testing folder within the project.

For information on specific integration test cases, refer to the Integration Tests sheet in the Test Case Tracker document.

**4.3 Acceptance Testing**

Acceptance tests are written to verify that the final product fulfills all of the system requirements and corresponding use cases to a high degree of satisfaction. In particular, acceptance tests will walk the customer through some representative operations of the program that match provided and derived use cases in the requirements document. These tests will be conducted at the end of the project with the customer, but should also be conducted by the team separately prior to submission. The testing coordinator will be responsible for writing acceptance tests; the entire team will be responsible for verifying their accuracy.

When writing acceptance tests, little to no references should be made directly to the design document. Because the acceptance tests examine whether use cases are fulfilled, all acceptance tests should be written to the requirements document. Acceptance tests need to be specific in what inputs are provided and where, and they should contain a few examples of alternative use case flows.

Acceptance test execution should focus on the “happy path” of the entire program; that is, the use case that would be executed most frequently by the users of the system. This “happy path” has the secondary requirements of being as error-free as possible; exceptions, errors, etc. should never occur. From the happy path, execution priority should then be given to the next most-used use case, and so on. Some examples of error-handling need to be included in the acceptance tests that are executed with the customer.

Acceptance tests will be based off of the Use Case Descriptions contained in Clause 4.2 of the Requirements Document. While developing acceptance test cases, an activity diagram will be developed that walks through each step of the use case execution. As stated in the previous paragraph, priority of test case development will then be assigned to the designated main execution path, with alternative paths defined and tested afterwards. After creating a test that examines the features of main and alternative execution paths, the test will be recorded in the Test Case Tracker document.

For information on specific acceptance test cases, refer to the Acceptance Tests sheet in the Test Case Tracker document.

**5.0 DOCUMENT CHANGE LOG**

|  |  |  |  |
| --- | --- | --- | --- |
| ***Document Version Number*** | ***Revision Date*** | ***Summary of Changes*** | ***Author(s)*** |
| *Initial / v0.0* | *02/28/2015* | *Initial revision.* | *Brian Mejorado* |
| *v0.1* | *03/06/2015* | *Clarified where exactly unit tests should be implemented in Clause 4.1. Fleshed out integration test operations in Clause 4.2.* | *Brian Mejorado* |
| *v0.2* | *03/07/2015* | *Preliminary introduction created in Clause 1.0. Updated terminology in Clause 2.0 for more correct references. Added caveat to Clause 3.0. References to Test Case Tracker in Clauses 4.1 & 4.2 updated for precision. Added reference to Test Case Tracker in Clause 4.3.*  *Removed all blue text in preparation for later addition to the repository.* | *Brian Mejorado* |
| *R1 / v1.0* | *03/08/2015* | *Updated approach to acceptance testing in Clause 4.3 to reflect the current course of the project.* | *Brian Mejorado* |
| *v1.1* | *03/12/2015* | *Updated Clause 4.1 to reflect new insights on unit testing and to be more specific on how developers should conduct tests. Applied increased knowledge of acceptance testing to Clause 4.3* | *Brian Mejorado* |
| *v1.2* | *04/05/2015* | *Added information to Clause 1.2.* | *Brian Mejorado* |
| *v1.3* | *04/18/2015* | *Updated Clause 4.3 to reflect recent applications of acceptance testing processes. Fixed a typo in Clause 5.0.* | *Brian Mejorado* |