Bradley Maness

Term Project: Java Chat Application

Test Plan Document

**Table of Contents**

**1 Introduction\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2**

1.1 Purpose and Scope\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2

1.2 Target Audience\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2

1.3 Terms and Definitions\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2

**2 Test Plan Description\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_3**

2.1 Scope of Testing\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_3

2.2 Testing Schedule\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_3

2.3 Release Criteria\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_3

**3 Unit Testing\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_5**

3.1 UserDataBase (Formerly UserData) Class\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_5

3.1.1 Test Case: .txt File Exists\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_5

3.1.2 Test Case: .txt File Doesn’t Exist\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_5

3.2 Conversations (Formerly History) Class\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_5

3.2.1 Test Case: “User\_History” Directory Exists\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_6

3.2.2 Test Case: “User\_History” Directory Doesn’t Exist\_\_\_\_\_\_\_\_\_\_\_\_\_\_6

**4 Integration Testing\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_7**

4.1 Client Input: Create New User\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_7

4.2 Client Input: Sending Message to Server\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_7

**1 Introduction**

**1.1 Purpose and Scope**

The purpose of this document is to provide an outline of specific test cases for important units and subsystems of the project and describe the outputs they should achieve for specific inputs.

**1.2 Target Audience**

The target audience for this document includes the stakeholders of the project(the instructor and any assistants) to illustrate techniques for creating test plans as well as other students who wish to read and understand ideas behind software testing documentation.

**1.3 Terms and Definitions**

This paper does not include any new definitions. It still talks about things in terms of Java. Some objects outlined in the previous document (Design) have been renamed to better reflect their functions as part of a refined design. Each class object that has been renamed will noted throughout the document.

**2 Test Plan Description**

**2.1 Scope of Testing**

The primary focus of the test plan outlined in this document will be on the subsystems and components of the server component as it is responsible for handling and recording data and passing it between different client components. A smaller description for testing the client component will also be provided.

The main areas of the project that will be tested are areas that are responsible for recording and organizing class objects that hold basic(the server class’s UserData class and it’s various sub-data-structures) as well as areas that require direct user input(the Client class). Classes that are responsible for holding basic information but contain no methods for managing data structures(the Message class) will be described, but not be the primary focus. The reason for this focus is to ensure the integrity of the projects fundamental requirement of saving and recalling all data regarding users and message history.

Testing methods that will be used within this document include black-box with specifically chosen test cases(to limit testing time) as well as glass-box methods

**2.2 Testing Schedule**

The testing schedule must be short due to time constraints on the project and must use very simple test cases. First it begin with the simplest base class(Message) and follow a series of integrations for each superclass:

Test Message.

Test Message within Thread.

Test Thread within Conversations.

Test Conversations within UserData.

Test UserData within UserDataBase.

Test UserDataBase within ChatServer.

Once integration and testing is finished within the server component and proper server functionality is verified, we then test the interaction between the server and client components:

Client → Server → Other Client

**2.3 Release Criteria**

For aspects of certain units, specifically units that are responsible for recording and reading data from outside files, there can be no fault tolerance; the execution of these aspects must be air-tight with no faults. Other aspects of the project that handle simple primitive data types have a much higher tolerance for mistakes because this data is ultimately arbitrary as we are mostly testing the data structures that handle them.

**3 Unit Testing**

The purpose of this section is to describe different aspects of the project in terms of units to be tested. Each described unit will consist of a collection of class objects.

**3.1 UserDataBase (Formerly UserData) Class**

The UserDataBase class is a subclass of the server that is responsible for keeping a record of users, their respective message histories, and showing who is currently online as per the project requirements. It reads in a list of users from an external .txt file within the same directory and builds a balanced binary search tree of UserData Class objects from it. Afterwards, whenever a new user is added to the tree the .txt file is appended with the new name.

There are only two cases for this unit: if a proper .txt file exists, or if it does not exist. The minimum criteria for release is perfect reading and writing to the file.

This will be a “Test to Code” scenario and utilize glass-box techniques to ensure that a tree is produced and is balanced.

**3.1.1 Test Case: .txt File Exists**

If a proper .txt file exists, the class first reads a number at the top of the file. Then it uses that number to read lines in a specific order to create UserData class objects and insert them into a binary search tree data structure to produce a balanced tree. To make sure this is the case, the tree will be tested with functions that will check to make sure it is balanced. Any new names added to the tree will be appended to the .txt file from this point on.

**3.1.2 Test Case: .txt File Doesn’t Exist**

If a proper .txt file does NOT exist, then the tree will be empty. Any new user inputed to the BST from it’s empty state must output a new .txt file with a number on it (1 for the first user) and the sole name. All other names inputed into the unit must increase the top number and append each successive name onto the list.

**3.2 Conversations (Formerly History) Class**

The Conversations class is a member of each node contained in the BST data structure of the UserDataBase class. It contains a doubly linked list of Thread class objects which in turn contain a doubly linked list of Message class objects. This system is designed to contain all records of respective user’s message history. These histories are contained within a directory called “User\_History”, which contains subdirectories for each respective user recorded in the .txt file used for UserDataBase. Within these subdirectories are multiple .txt files that correspond to conversations the respective user is involved in. Each line in these files corresponds to a Message class object that gets inserted into a Thread class data structure which in turn gets inserted into the Conversations class data structure included in a respective UserData node.

The test cases for this class include the existence or non-existence of the “User\_History” directory, the directory for a respective user, and the existence of any proper .txt files for inserting Threads into Conversations. The Minimum criteria is similar to that of 3.1 in that only perfect reading, writing, and creation will be acceptable.

This will also use “Test to Code” glass-box methods to ensure it is properly creating the data structures.

**3.2.1 Test Case: “User\_History” Directory Exists**

When ran, Conversations checks for the existence of the “User History” directory. If it exists, it then checks for the existence of the respective user subdirectory it’s associated with.

If this new subdirectory exists, it must then check to see if it has any .txt Thread files associated with it. The final output of this scenario will depend on the existence of any .txt files: Empty for none, and non-empty for one or more.

**3.2.2 Test Case: “User\_History” Directory Doesn’t Exist**

When ran and the “User\_History” directory does NOT exist, then the final output should be the creation of said directory and the subdirectory inside it for the user. The final output should have no user history because no .txt files existed.

**4 Integration Testing**

This final section will discuss test cases for when the two main subsystems of the server described in section 3 are integrated together and receive input from a client program. It will describe the output that a specific input should yield in the final product.

We first begin with the assumption that the subsystems described in section 3 will operate consistently when put together(3.1 loads or creates the user .txt file and 3.2 loads or creates the user history sub-folders and files).

**4.1 Client Input: Create New User**

This scenario has two possible starting positions: “no users” and “one or more”. Since a person’s name, username, and password can be anything, the data is arbitrary.

For “no users” we begin with the assumption that the system is starting off completely clean: No .txt files exist for users and no folders or other files exist for those users histories. The input of a new user from the client should output “users.txt” with a 1 at the top and a single name line. It should also create “User\_History” and an empty sub-folder inside with the name of the newly created user.

For “one or more” we assume all respective folders and files exist and have successfully loaded into our subsystems. Adding a new user should then modify existing files and folders by changing the number of users and append a name on the “users.txt” file as well as create a new empty sub-folder within “User\_History” with the respective name to that folder.

A successful output can be verified by re-running the server and examining the contents of “UserDataBase” for the new name.

**4.2 Client Input: Sending Message to Server**

This test case assumes the client has successfully logged into the server and has created a proper thread to send messages to. It also assumes that all other users addressed in the thread exist. Since a message can be anything, the data inside it is arbitrary.

This presents us with two possibilities to account for when sending a message: the Thread class .txt file exists in all participants “User\_History” sub-folders, or it does not.

If it does not, when the message is sent, the output should be the creation of a new .txt file representing the thread included in each participating user’s “User\_History” folder.

If a thread already exists, then each instance of the .txt file(one for every user involved) in each respective “User\_History” folder must be appended. This can be verified by examining the files after sending and also by reloading the server and examining the Thread class data for any user in the conversation.