Ben Reichert

# Term Project: *ChocAn*

Test Plan Document

## Table of Contents

### Table of Contents

# 1 Introduction

This document is describing the Chocoholics Anonymous project, specifically the test plan for the ChocAn software. This document will outline the test plan organization, timeline, and details of software testing. The document will describe how unit and integration tests will be performed on the ChocAn software, and testing requirements.

## 1.1 Purpose and Scope

The purpose of this document is to supplement the ChocAn Requirements document and the ChocAn Design document. It is intended to detail the testing that will be performed for the ChocAn software system. This document will cover an outline of the testing to be performed, but will leave the specific details of testing and writing said tests to the developers of Ben Reichert Software, LLC.

## 1.2 Target Audience

This document is written for the developers, and stakeholders of the ChocAn Software system.

## 1.3 Terms and Definitions

### 1.3.1 EFT

Electronic Funds Transfers: used to keep track of payments to be made between providers and ChocAn.

### 1.3.2 DC

ChocAn Data Center, responsible for keeping records and service transactions.

### 1.3.3 Terminal

The specially designed ChocAn terminal, which will include a PC along with a magnetic card reader for reading member cards. This hardware design is not the responsibility of Ben Reichert Software, LLC, and thus will not be discussed. The software that runs on this Terminal is within the scope of this project and document.

### 1.3.4 Provider

A provider is an institution registered with Chocoholics Anonymous that provides treatments and consultations with health care professionals, dietitians, internist, and exercise specialists.

### 1.3.5 MC

MC: Member Card: A plastic card embossed with the ChocAn member's name and member ID. This card has a magnetic strip with the previous information encoded on it for reading through the Terminal.

### 1.3.6 System

The ChocAn system as a whole, in all of it's functionality.

### 1.3.7 MID

Member IDentification number, a 9 digit sequential number.

# 2 Test Plan Description

The purpose of this section of the testing document is to outline the scope of the test plan, testing schedule, and release criteria pertaining to accepted tolerances. This section is to let the reader know what the planned scope of testing is, when testing will be done, and the criteria of the testing of the ChocAn system software.

## 2.1 Scope of Testing

The test plan scope includes the front end terminal simulator and file handling. The plan will not currently include the database storage as time is limited to when we would like to deploy the application. The database testing layer will be added at a later date after the initial product delivery. The front end terminal simulator will include test inputs and control flow to test 80% of methods and their control structures (if, switch blocks) to get 80% code coverage for testing. The file handling methods will be tested to ensure 80% code coverage, and will generate sample inputs running the according methods. Unit tests for all current methods will be implemented, as well as integration tests for the main layers of the application.

## 2.2 Testing Schedule

November 25th – Document presented to clients
November 28th - unit and integration tests started
November 30th – writing tests completed
December 1st – begin software testing process
December 5th – ship tested code to production servers

## 2.3 Release Criteria

Functionality and security tests must pass before the software is deployed. Tests for performance, or usability are not required to pass for deployment. As long as the critical services are testing properly, the system can be deployed. Non-functional requirements are not expected to pass tests.

# 3 Unit Testing

Describe the purpose of this section and outline its contents. Name the units you will be testing and describe their functionality.

## 3.1 User Interactions

The User interactions menu will be implemented through a Python class structure. The class will have methods for storing user inputed data, and displaying output data to a pseudo-terminal. This will be the main class that has submenus, that will have their own class objects that are part of the UINT(user interaction) object. This way there is one hierarchical structure to deal with that has many submodules. The submodules will be dynamically loaded at runtime for submenu components. This will have a UNIX shell like structure for menus and prompts. Each submenu will have commands to run that will prompt the user for input, and then generate appropriate output.

80% code coverage is expected. The unit tests will test given input and output as this is a user interaction based part of the software. It's purpose is to provide a front end user experience for the terminal operators that will have terminals in the provider offices. This unit of the system is responsible for handling user input, and producing coherent output.

### 3.1.1 Managing Member Accounts

Managing member accounts will be two separate menus, one for deletion of members, and one for addition of members. Member data will be stored as temporary strings in a dictionary object, and then be written to the DB. This is the case for both reading and writing data. If the user account needs to be updated the data will be loaded from the DB to a dict (a method will handle this) and then modified as a dictionary, and then loaded back into the DB. If a member needs to be added, a dictionary will be created and then used the dict_save() method to write the dict to the DB.

#### 3.1.1.1 Adding Member

The Provider Terminal Operator will drop into the adding member submenu and be prompted for each requirement as specified in the requirements document for adding a user. This submenu will be a set of prompt methods, that ultimately call a DB set function to add the user to the database system. Information will be stored as a dictionary.

We will know if the unit test passes if the output from the terminal matches the following table:

| *Input* | *Expected Output* |
|---|---|
| <valid member information> | "Member successfully added to System." |
| <invalid member information> | "Member information invalid. Please enter valid member information." |

#### 3.1.1.2 Deleting Member

The Provider Terminal Operator will drop into the deleting member submenu and be prompted for each requirement as specified in the requirements document for deleting a user. This submenu will be a set of prompt methods, that ultimately call a DB set function to delete the user to the database system. Verification functions will be called to verify the Provider Members ID, as well as the MID and Members name and personal information for verification purposes. One method will be responsible for input and output, while another does checking against the DB to ensure authentication is valid. Once valid, a set function will remove the member from the database, and only can be added again through the adding members submenu.

We will know if the unit test passes if the output from the terminal matches the following table:

| *Input* | *Expected Output* |
|---|---|
| <valid member information> | "Member removed from System." |
| <invalid member information> | "Member could not be removed due to invalid information provided. Please try again." |

#### 3.1.1.3 Look-Up Member

When the Provider Terminal Operator scans a MID, they enter this menu which will pull up member information and submenus based on the MID. From here, they can edit a member's information. The Members information will be loaded into a temporary Python dictionary which can be edited, and then passed to a function that will create a SQLite query to update the database accordingly from the given dictionary.

We will know if the unit test passes if the output from the terminal matches the following table:

| *Input* | *Expected Output* |
|---|---|
| <valid MID> | <member information> |
| <invalid MID> | "Member could not be found due to invalid Member ID number. Please try again." |

#### 3.1.1.4 Displaying Fees

The Provider Terminal Operator will follow the terminal prompts for input once the display fees menu is selected after a user is looked up in the database system. This is a submenu to the look-up member system. The fee system will be a class object, that will have handler functions for user IO. Since it only needs read abilities, there is no need to write to the database, but a reading method will need to query the database to display fees.

We will know if the unit test passes if the output from the terminal matches the following table:

| *Input* | *Expected Output* |
|---|---|
| <valid MID> | <member fee information> |
| <invalid MID> | "Member could not be found due to invalid Member ID number. Please try again." |

## 3.2 Data Center Automated Processes

These processes are ones that are either time schedule based (ie, it runs on Friday night at 9pm) or automated processes that run on cron times every X minutes or hours or days. These processes need little to no user interaction, other than the occasional "button click" to start the process. Examples may include a manager starting the email system, or asking for a generated report. In this case, they make an initial request, and the automated software does the rest and eventually reports back to them when the process is complete. These systems will not be based on the terminal system, and will be implemented in a separate set of classes and objects that are created at runtime of the application. Some user input will be necessary from the terminal system to "kick start" these systems, but they will largely required no user input as they are automated.

80% code coverage is expected. The unit tests will test given input and output as this is a user interaction based part of the software. The purpose the DC processes are to automatically do tasks and generate reports on specific intervals.

### 3.2.1 Member Reports

Member reports will be generate daily through an automated process requiring no user interaction. The reports will be saved to disk as specified in the requirements document. A part of the reports class will be a file IO handler to deal with reading and writing files to and from disk. There will need to be a DB handler as well to get information to generate the reports from. Once the reports are created, the application will wait until the specified time to send them to the email system, at which time the reports will be sent via email to members. This will all be contained in a python class object, with helper functions that are based off of the Time module for automated run times. User interaction is not required.

We will know if the unit test passes if the output from the terminal matches the following table:

| *Input* | *Expected Output* |
|---|---|
| This process is run at at Midnight on Fridays. | Member reports to file, emails |

### 3.2.2 Transaction Logs

The system will keep an automated log saved to disk of every interaction with the system. This will be an automated process requiring no user input. If an individual wishes to see the log, they must contact the DC support team to obtain the secure log as necessary. Every action a terminal based user does, will call a specific log function in this object that records their submissions through the application. This will be a relatively simple class with one or two methods that are just responsible for taking a user input value (as a string) and giving that to a file handler to append the string to an existing log file.

We will know if the unit test passes if the output from the terminal matches the following table:

| *Input* | *Expected Output* |
|---|---|
| <any program command> | Line appended to log file |

### 3.2.3 EFT Data

Every transaction requiring transfer of funds from a member to ChocAn or ChocAn to a provider will automatically be appended to a EFT Data log every day, and send to ACME to be processes nightly. This process requires no user input. This class will require a function handler that deals with file IO, as well as DB interactions to keep track of what is stored in the database for EFT data. Each line of the EFT file will be appended by a write function. Another function will handle the format string to prepare to write the appropriate EFT data to the file.

We will know if the unit test passes if the output from the terminal matches the following table:

| *Input* | *Expected Output* |
|---|---|
| Every evening at 9PM the ACME Accounting Services will sync the EFT data to their system. | Line appended to log file stating that EFT data was transferred to ACME. EX: "EFT data send to ACME at November-24-2014-0800" |

# 4 Integration Testing

This section is to describe the integration testing plan for the ChocAn software system. The integration tests are combined into two groups, User Input and the Data Center Automated Processes since they are the two main parts of the system software.

| *ID* | *Integration Test* | *Paragraphs* |
|---|---|---|
| 1 | Terminal → DC | 3.1.1.1, 3.1.1.2, 3.1.1.3, 3.1.1.4 |
| 2 | Automated Processes → disk IO | 3.2.1, 3.2.2, 3.2.3 |
| 3 | Automated Processes → email | 3.2.1 |
| 4 | Automated Processes → database | 3.2.1, 3.1.1.1, 3.1.1.2, 3.1.1.3, 3.1.1.4 |
| 5 | Terminal → DB | 3.1.1.1, 3.1.1.2, 3.1.1.3, 3.1.1.4 |