ChocAn

Test Plan Document

Authors: Daniel Mendez, Alexander Salazar, Alexander Dupree,
Arman Alauizadeh, Kyle Zalewski, Dominique Moore

Table of Contents

1. Introduction **2**
1.1 Purpose and Scope 4
1.2 Target Audience 5
1.3 Terms and Definitions 5

2. Test Plan Description **7**
2.1 Scope of Testing 7
2.2 Testing Schedule 8
2.3 Release Criteria 8

3. Unit Testing **9**
3.1 Strategy 9

4. Smoke Testing **13**
4.1 GCC with Debug Configuration 13
4.2 GCC with Release Configuration 14
4.3 Clang with Debug Configuration 15
4.4 Clang with Release Configuration 15

5. System Testing **17**
5.1 Test Cases 17

1. Introduction

This document will cover how the ChocAn software will be tested, and verified. The ChocAn software stores the information about ChocAn members, providers, and managers. Depending on who logs in to the system, the functionality of the software will change. When a provider runs the software, the provider is asked to enter his or her provider number. They will then run a transaction using the ChocAn software with the member's number. If the member's number is invalid, then an error message is displayed, the type of error is also displayed. Once the member's ID has been validated the provider records the transaction information. Afterwards, the software looks up the fee to be paid for the service provided and displays it on the terminal. The provider can request the software for a Provider Directory, an alphabetically ordered list of service names and corresponding service codes and fees, this directory is sent to the provider as an email attachment.

Every Friday the week's file of services provided is read, and prints a number of reports. A ChocAn manager can request that a report be individually ran at any time. Each member who has consulted a ChocAn provider during that week receives a list of services provided to that member, sorted in order of service date. Each provider who has billed ChocAn during that week receives a report, sent as an email attachment, containing the list of services they provided to ChocAn members. Managers receive a summary report of all providers who were paid that week, the number of consultations each had, and their total fee for that week.

During the day the ChocAn software allows operators to add new members to ChocAn, delete members who have resigned, and to update member records. Similarly, provider records are also added, deleted, and updated.

Each member report will be written to its own file, and the name of the file should begin with the member name followed by the date of the report. The provider reports should be handled in the same way. The Provider Directory must also be created as a file.

This section will provide an overview of what tests will be covered, why testing is important, who this document is targeted towards, and definitions for some of the terms that will be used.

1.1 Purpose and Scope

In order to ensure that the software is functional and meets the specified requirements many tests will be implemented. There are a lot of factors to consider when testing software, in fact it’s too much to count, and there for account for. This test plan covers potential common cases that we expect may cause the software to malfunction. The intent of this document is to provide a detailed description of what tests are being used, why they’re being used, how they are implemented, and how the tests show that the software does in fact meet the requirement the client requested. The kinds of test that this document will cover are:

- Unit testing
- Smoke testing
- System testing

1.2 Target Audience

The audience for this document are the software developers working on the product, the ChocAn managers, providers, and members.

1.3 Terms and Definitions

EFT - Electronic Funds Transfer

ChocAn - Chocoholics Anonymous

Manager - Administrator for the Chocoholics Anonymous organization

Member - A member of the ChocAn medical service network

Provider - Health care professional that provides ChocAn services for members

Record - Shorthand for either a Member or Provider record which stores the information relevant to the record holder

Catch2 - Free and Open-Source multi-paradigm test framework for C/C++ https://github.com/catchorg/Catch2

CircleCI - Continuous Integration service that builds the software on multiple platforms. https://circleci.com/

CodeCov - Code Coverage reporting service that performs in depth analysis of code execution patterns. https://codecov.io/

Codacy - Automated code review service that performs an in-depth static analysis of the code with a suite of analysis tools. https://www.codacy.com

Unit Test - Test assertions for individual program units, generally functions

Mocks - Object that simulates the behavior of a complex dependency, and supplies deterministic values for testing.

Stubs - Similar to a Mocked object, stubs simulate behavior of software components

2. Test Plan Description

In this section describes the overall plan for testing the ChocAn system. This includes the scope of what our test cover, the schedule for the testing, and the criteria for what we consider acceptable.

2.1 Scope of Testing

The scope of our testing shall include unit and integration tests on any code being written. Code coverage reports via *CodeCov* will be used as a metric for determining the effectiveness of the tests being performed.

Unit test will be written in conjunction with the code to be tested, using the *Catch2* testing framework. All unit tests must pass before new tests or code can be written.This will ensure all code being used is being tested first, and mitigates regressions. Should any bugs occur afterwards, a specific test case will be written that replicates the bug. Any action that could result in an exception will also be covered in the unit test. All methods will be tested to ensure they throw exceptions if required. This ensures the scope for our unit tests includes most, if not all, behavior exhibited by that specific unit.

Integration testing will be done using *CircleCi* anytime a push is made to our repo. *CircleCi* will attempt to build our ChocAn system on Ubuntu 16.04 using GCC with C++17, with our provided build script. All current unit test will be run every time our build script is executed. This ensures the scope of our integration tests covers all unit tests and system specific requirements.

User testing will be done as components of the system become available to use. A user test shall involve a person interacting with the ChocAn system as they see fit, while making note of bugs or UI/UX improvement opportunities.

We will not be testing for compatibility on platforms other than Ubuntu 16.04. We will also omit testing performance metrics, execution speed and memory required. Performance considerations will be made during code reviews

prior to merging changes. We also exclude EFT account processing and email notifications from our testing scope, as these lie outside the boundaries of the system

2.2 Testing Schedule

Unit/Integration:	Continuous test driven development
User Testing :	November 24 - Release

2.3 Release Criteria

We must have 100% passing unit and integration tests to consider release. Code coverage shall be at least 90% and must include code needed for use cases. Only minor issues regarding UI or UX may persist into release.

3. Unit Testing

This section outlines the strategy, methods, and tools that will be used to test each unit within the ChocAn application. The test suite accompanying the ChocAn software will provide confidence in the dependability and functionality of the system. Furthermore, the expansive test coverage will quick detection and patching of regressions.

3.1 Strategy

Testing is an integral part of the design of our system. All classes and components are designed in such a way that dependencies can be injected with mocks or stubs and deterministic testing can be accomplished. To ensure testability among our code, every function, class, and component is reviewed and refactored specifically to reduce coupling and allow for structured unit testing.

Primarily, our testing strategy is to provide test cases that show that, when given expected input, units perform as expected. Depending on the components this will involve either a guideline based strategy, or a partitioned input strategy. At a high level, this means that every function will have test cases that evaluate acceptable input as well as what happens when invalid input is given. Each class will have a test case specifically for the construction of the object when given a variety of acceptable and invalid inputs. And finally, every component will have test cases demonstrating the proper and improper behavior of the component. This means that every source file in the project will be mirrored with a test file that will define all the tests related to that component. Within the test file, there will be multiple test cases that examine each layer of functionality the component provides.

The initial burden of designing, writing, and executing unit tests will be performed by the developer of the code that is being written. However, to enforce a strict adherence to disciplined testing, all code undergoes a review process that must pass a suite of checks before it is merged into the master branch. These checks include: Continuous Integration, passing of all automated tests, acceptable code coverage reports, static code analysis, and a review by at least one other project developer.

3.1.1 Tooling

To facilitate this Unit testing strategy, we take advantage of multiple open source tools and software. First, the testing framework used for our project is *Catch2***.** *Catch2* is a free and open-source test framework that licensed under the permissive Boost Software License 1.0. This test framework provides a number of advantages, namely test automation and a variety of well defined assertion statements that make testing specific expressions easy.

To validate what code is actually being tested, we use Gcov for profiling and analysis of the code. These Gcov reports are compiled when the test binary is executed and are sent to *Codecov.io* servers where browser friendly reports are generated and further analysis is conducted. Furthermore, *Codecov* is hooked into the remote repository and provides us with up-to-date coverage reports during the code review process. These reports are used to determine the quality of a pull request, and evaluate whether the current tests are comprehensive enough for each unit. Specifically, we will look for functions with multiple branching paths and ensure that each path is executed in a unit test. Even though code coverage requirements are generally arbitrary, we will strive for at least a 90% coverage report. Our primary goal is to ensure that each functional piece of code has a unit test, and worry less about transitive code or functions that simply print to the screen; these will be covered in our integration and system tests.

Unit tests will be run locally on the developers machine before it is pushed to the remote repository. When the new code is checked in however, *CircleCI* our continuous integration service, will create a build on one of their machines. Once the code is compiled in the debug and release configurations, CircleCI will execute all unit tests. If any test fails, then the entire build will be labelled as failed. If the build has failed, the code will not be accepted into the master branch of the repository until fixes have been made.

3.1.2 - Test Cases

Instead of providing an exhaustive (and probably incomplete) list of all test cases in the system, the following section will contain a minimal summary of what types of test cases each object will possess. If code requires testing that is not summarized in the below test cases, we will generate specific tests for the task of testing that unit.

Constructors - Every object will have a test demonstrating proper and improper construction of the object. Constructors that take in a set a parameters and initialize internal values will have those attributes interrogated to ensure initialization is done correctly. Any constructor that can throw an exception will have tests that inject inputs for each type of exception that may be thrown.

Functions - Every function will be accompanied with a test. Functions with branching paths will be tested with inputs that cause each branch of code to be executed. Any function that accepts a pointer as a parameter will be tested with a null pointer. Functions that manipulate data structures will have test cases with inputs partitioned for the state of the data structure. I.E. fixed size arrays with tests for when the array is empty, partially filled, and full. Any function that can throw an exception will have a test with inputs that will cause this exception to be

thrown. Functions that change the state of an internal variable will have that variable inspected in a test. Any function that performs computation on a numeric type will be tested with inputs that force the computation result to overflow or underflow.

Operators - Operator overloads will be tested in much the same way as regular functions. However, classes that define comparison and equality operators will have each operator tested for expected results. Specifically, these operators will be tested with inputs that causes the function to return true and inputs that return false.

Exceptions - If an exception can be thrown, then there will be a test that causes the exception to be thrown. Furthermore, caught exceptions will be verified that type the exception matches the expected type.

4. Smoke Testing

Smoke testing will give an overall impression of the basic functionality of our code from both development and user-side perspectives. We will ensure with these tests that code compiles properly without errors, and, once the executable has been created, that it runs as expected through the various user-level functionalities. The test will follow a structure that tests each possible configuration and fails overall upon failure at any subtest. The smoke testing protocol will be run every time code is pushed to the remote repository, with the assumption that if the test fails, the build is considered failed as a whole. A successful smoke test is necessary for a successful build.

Note that the debug configuration means that the code will be compiled with debug symbols, and profiling flags set to on. Also, the debug configuration will utilize a Mocked in-memory internal database. The release build is compiled with all optimizations flags on and is connected to the actual sqlite3 database.

4.1 GCC with Debug Configuration

This test case will determine basic functionality of the program using GCC to compile with the debug configuration set. If any of these subtests fail, the build has failed. There is no need to continue the smoke test in upon failure of this test case.

4.1.1 Compile Software with GCC Using Debug Configuration

This subtest will compile the project using GCC with the debug configuration set. Upon successful compilation (no errors), move on to the next subtest. If the compilation produces errors, this subtest and therefore the entire smoke test is considered failed.

4.1.2 Log in as Manager with Software Run Using Debug Binaries

This subtest will run after successful GCC compilation with the debug configuration set. The subtest will be considered a pass if login completes with manager credentials entered. If this subtest passes, move to the next subtest. Otherwise, the entire test fails.

4.1.3 Log in as Provider with Software Run Using Debug Binaries

This subtest will run after successful subtest 4.1.2. The subtest will be considered a pass if login completes with provider credentials entered. If this subtest passes, move to the next test (4.2). Otherwise, the entire test fails.

4.2 GCC with Release Configuration

This test case will determine basic functionality of the program using GCC to compile with the release configuration set. If any of these subtests fail, the build has failed. There is no need to continue the smoke test in upon failure of this test case.

4.2.1 Compile Software with GCC Using Release Configuration

This subtest will compile the project using GCC with the release configuration set. Upon successful compilation (no errors), move on to the next subtest. If the compilation produces errors, this subtest and therefore the entire smoke test is considered failed.

4.2.2 Log in as Manager with Software Run Using Release Binaries

This subtest will run after successful GCC compilation with the release configuration set. The subtest will be considered a pass if login completes with manager credentials entered. If this test passes, move to the next subtest. Otherwise, the entire test fails.

4.2.3 Log in as Provider with Software Run Using Release Binaries

This subtest will run after successful subtest 4.2.2. The subtest will be considered a pass if login completes with provider credentials entered. If

this subtest passes, move to the next test (4.3). Otherwise, the entire test fails.

4.3 Clang with Debug Configuration

This test case will determine basic functionality of the program using Clang to compile with the debug configuration set. If any of these subtests fail, the build has failed. There is no need to continue the smoke test in upon failure of this test case.

4.3.1 Compile Software with Clang Using Debug Configuration

This subtest will compile the project using Clang with the debug configuration set. Upon successful compilation (no errors), move on to the next subtest. If the compilation produces errors, this subtest and therefore the entire smoke test is considered failed.

4.3.2 Log in as Manager with Software Run Using Debug Binaries

This subtest will run after successful Clang compilation with the debug configuration set. The subtest will be considered a pass if login completes with manager credentials entered. If this test passes, move to the next subtest. Otherwise, the entire test fails.

4.3.3 Log in as Provider with Software Run Using Debug Binaries

This subtest will run after successful subtest 4.3.2. The subtest will be considered a pass if login completes with provider credentials entered. If this subtesttest passes, move to the next test (4.4). Otherwise, the entire test fails.

4.4 Clang with Release Configuration

This test case will determine basic functionality of the program using Clang to compile with the release configuration set. If any of these subtests fail, the build has failed. Failure of this test case indicates failure of the smoke test. If this test case succeeds, the smoke test is considered a success.

4.4.1 Compile Software with Clang Using Release Configuration

This subtest will compile the project using Clang with the release configuration set. Upon successful compilation (no errors), move on to the next subtest. If the compilation produces errors, this subtest and therefore the entire smoke test is considered failed.

4.4.2 Log in as Manager with Software Run Using Release Binaries

This subtest will run after successful Clang compilation with the release configuration set. The subtest will be considered a pass if login completes with manager credentials entered. If this test passes, move to the next subtest. Otherwise, the entire test fails.

4.4.3 Log in as Provider with Software Run Using Release Binaries

This subtest will run after successful subtest 4.4.3. The test will be considered a pass if login completes with provider credentials entered. If this subtest passes, the entire smoke test is considered successful. Otherwise, the entire test fails.

5. System Testing

The purpose of System Testing is to ensure that all of the system components are successfully integrated to create a complete system. Testing here is responsible for finding any unexpected interaction errors between the components within the system. It is also responsible for proving that the functional and non-functional requirements of the system are met. If requirements are met, then system components have successfully been integrated.

5.1 Test Cases

5.1.1 - User can log in with Manager credentials

The system shall transition to the Manager menu when given proper Manager credentials.

5.1.2 - User can log in with Provider credentials

The system shall transition to the Provider menu when given proper Provider credentials.

5.1.3 - User cannot log in with improper credentials

The system shall not transition from the login screen when given invalid credentials.

5.1.4 - Provider Menu transitions to View Account

The purpose of this test case is to ensure system successfully transitions to View Account screen from the Provider Menu when the View Account option is selected.

5.1.5 - View Account transitions back to previous Menu

The purpose of this test case is to ensure system successfully returns to the previous menu after the user has finished viewing their Account

5.1.6 - Manager Menu transitions to View Account

The purpose of this test case is to ensure system successfully transitions to View Account screen from the Manager Menu when the View Account option is selected.

5.1.7 - Provider Menu transitions to Add Transaction.

The purpose of this test is to ensure that the system transitions to Add Transaction screen from Provider Menu when the Add Service Transaction option is selected from the Menu

5.1.8 - Add Service Transaction records the transaction data in the Database

The purpose of this test is to ensure that the system successfully adds a service transaction to the database when a service transaction is made. Furthermore, we will verify that the Provider information and cost owed is stored correctly in the EFT file.

5.1.9 - Invalid inputs during the Add Service Transaction

The purpose of this test is to ensure that the system displays an error message when invalid inputs are given while adding a service transaction and allows the user to re-enter a valid input.

5.1.10 - Provider Menu transitions to Update Member Account

The purpose of this test is to ensure that the system transitions to Update Member Account screen from the Provider Menu when Update Member Account option is selected from the menu

5.1.11 - Member account is updated

The purpose of this test is to ensure that the system successfully updates a member’s account data when the user enters the new data.

5.1.12 - Invalid inputs given when updating member account

The purpose of this test is to ensure that the system displays an error message when given an invalid input while updating a members account and allows the user to re-enter a valid input.

5.1.13 - Manager Menu transitions to Create Member/Provider Account

The purpose of this test is to ensure that the system transitions to Create Member/Provider Account screen from Manager Menu when this option is selected from the menu.

5.1.14 - New Provider/Member account created

The purpose of this test is to ensure that the system successfully creates a provider/member account when given valid data inputs and the new account is stored securely in the database.

5.1.15 - Invalid inputs given when creating provider account

The purpose of this test is to ensure that the system displays an error message when given an invalid input while creating a provider/member account and allows the user to re-enter valid inputs.

5.1.16 - Manager Menu transitions to Delete Provider/Member Account

The purpose of this test is to ensure that the system transitions to Delete Provider/Member Account from Manager Menu when given the correct input for Delete Provider/Member Account.

5.1.17 - Account deleted

The purpose of this test is to ensure that the system successfully deletes a provider/member account when given valid inputs, which could be a valid account number.

5.1.18 - Invalid inputs given when deleting account

The purpose of this test is to ensure that the system displays an error message when given an invalid input while trying to delete a provider or member account and allow user to re-enter valid inputs.

5.1.19 - Deleting non-existent account

The purpose of this test is to ensure that the system returns to the Manager Menu from Delete Provider/Menu and displays an error message when a non-existent account is inputted to be deleted.

5.1.20 - Manager Menu transitions to Generate Summary Reports

The purpose of this test is to ensure that the system transitions to Generate Summary Reports from Manager Menu when given the correct input for Generate Summary Reports.

5.1.21 - Summary Reports Generated

The purpose of this test is to ensure that the system successfully generates the summary reports when at Generate Summary Reports state.

5.1.22 - Manager Menu receives invalid input

The purpose of this test is to ensure that the system displays an error message when receiving an invalid input at the Manager Menu and allows the user to re-enter a valid input.

5.1.23 - Provider Menu receives invalid input

The purpose of this test is to ensure that the system displays an error message when receiving an invalid input at the Provider Menu and allows the user to re-enter a valid input.