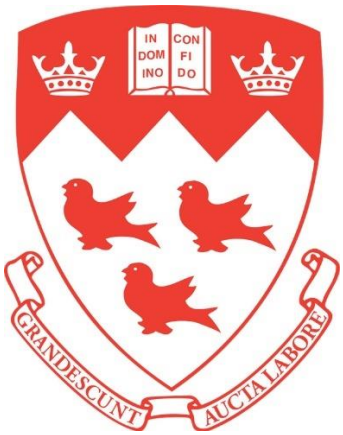
McGill University



ECSE 321 – Group Project Deliverable 3

Quality Assurance Plan

Prepared by:

Rony Azrak (260606812)

Brent Coutts (260617550)

Max Johnson (260498322)

November 14th, 2016

1 Unit Test Plan

We will indicate which classes need to be tested and which do not. Some methods like getters and setters are too trivial and with a big program like this it is better to leave them untested. We will not test code that deals with UI either because unit tests are back-end tests.

In the application package, the FTMSApplication class calls upon the PersistenceFTMS subsystem to load persistence data, and starts the UI, so we will not have unit tests for it.

In the controller package, we will test 3 of the 4 methods of the OrderController class: FromStringToMenuItem, canItemBeMade and increasePopularity. placeOrder uses some of these methods so it does not belong in the unit tests.

We will test all methods of the ScheduleMaker class.

In the model package, we will test almost all the methods of the FTMSManager class: numberOfEquipment, hasEquipment, indexOfEquipment, hasMenu, numberOfStaff, hasStaff, indexOfStaff, numberOfSupplies, hasSupplies, indexOfSupplies, addEquipment, removeEquipment, addEquipmentAt, addOrMoveEquipmentAt, addStaff, removeStaff, addStaffAt, addOrMoveStaffAt, addSupply, removeSupply, addSupplyAt, addOrMoveSupplyAt, delete.

We will test 2 methods of the equipment class: delete and toString because the rest of the methods are getters and setters.

We will test all of the methods of the Menu class except for getters and setters and trivial methods (like minimumNumberOfMenuItems which only returns an int of value 0): numberOfMenuItems, hasMenuItems, indexOfMenuItem, addMenuItem, removeMenuItem, addMenuItemAt, addOrMoveMenuItemAt, delete.

We will do the same thing with MenuItem class: indexOfIngredient, addIngredient, removeIngredient, addIngredientAt, addOrMoveIngredientAt, delete, toString.

We will test the same methods for the Order class as we tested for the MenuItem class except that instead of dealing with ingredients we deal with menu items.

We will test 3 methods of the Schedule class: setSunday, delete and toString.

Aside from getters and setters, the Staff and Supply classes only have 2 methods: delete and toString which we will unit test.

For the persistence package, there is no particular method to test, but while unit testing other methods, we should add assertions and evaluate the persistence file to test if that works well.

Finally, the View package display the content retrieved via the controllers so there is no unit test we can create for it.

We will use different tools to unit test our different applications. JUnit for the desktop app and PHPUnit for the web app. Since our android app uses the jar from the java desktop app, we don't need to test the same methods again. These tests are run on simple methods that should rarely be modified, so we can afford to run tests every time we do modify some. Our goal is to have a 75% test coverage of all our code.

2 Integration Strategy

Integration testing is performed in order to verify the functional performance and reliability requirements are met for major design sections. These sections are logical groupings of components which have previously been unit tested. This method employs a black box technique, as it is more general than unit testing. It is only of interest to see that the functionality grouping accepts an input and produces a valid output. This clearly requires that all of the separate components are interacting in the proper manner.

The integration strategy that was chosen for implementation is the top down approach. In this approach the high-level units are created/tested first, followed by the lower-level units. The highest-level component is tested on its own first, then the next level below is grouped with the highest-level component causing this group of components to be tested and so on.

As the tests are performed from the top down, stubs of lower-level units can be required if these units are not created yet. Stubs are a simplistic function/component that acts in the same way as the lower-level component would. However, at this point in the food truck management system (FTMS) design, all of the components have been created and it is not necessary to use stubs during the testing.

Components within a system interact in various methods. The components can pass parameters, share memory, interface, or pass messages. The use of the program is largely centered around components passing parameters through method calls.

The common errors that need to be considered through the integration testing are misuse, misunderstanding, and timing. The misuse error stems from the caller creating an error by using an interface incorrectly, for example mistyping a command. Misunderstanding refers to when a person calling the program attempts to use the interface in a method that the program was not created to handle. Finally, timing refers to the interface being used in the correct way, but at the wrong time. Misuse and misunderstanding are the most pertinent errors that will be dealt with.

The top-down integration strategy was chosen to be applied because the programmers felt that, if necessary, stubs would be easier to implement than drivers. Also, working in a forward manner through each of the integrations is easier to logically understand as this is the path the computer follows when the program is executed.

During the production of the FTMS, the program was created in a backwards ‘bottom-up’ manner. Therefore, the lower-level components have undergone more code review as they were created near the beginning of the software production. This suggests to the programmers that these are the most refined components and problems are more likely to be present in the highest up components. When this is the case, it is wise to implement a top-down testing strategy. This is the case because it is desirable to encounter issues in the beginning of the integration testing for a top-down approach, as there are fewer components grouped together in these first integration testing stages. It is thus easier to make isolate the issue, make updates to the code or tests, and continue with the top-down integration testing.

The program consists of four main use cases. These cases are: ordering, adding a menu item, modifying equipment or inventory, and modifying staff schedules/roles. Each of these use cases will be tested as a separate integration. The specific components discussed relate to the Java platform; however, each of the different platforms is highly similar. A list of the testing order is provided as follows:

1) EquipmentController
    i. FTMS Manager
        i. Equipment
            1. Supply
2) MenuController
    i. MenuItem
        i. Supply
3) OrderController
    i. FTMS Manager
        i. Menu
            1. MenuItem
                a. Supply
4) ScheduleMaker
    i. FTMS Manager
        i. Schedule
            1. Staff

The tests are run in the order listed above because it is simpler to test the smaller integrations first. This is done in order to ensure these functionalities pass before testing larger functionalities which also encompass some of these previously involved components.

For the Java platform, JUnit was used to run the integration tests. For PHP and android respectively, PHPUnit Test and gradle. These are tools that the programmers are comfortable using due to their usage in previous projects. It is also expected that some previously created tests and test structures will be able to be refactored for use in the FTMS testing.

The tests that will be written in addition to the unit tests, involve testing each of the 4 use cases of the FTMS system. Black box tests will be written that test each use case can receive an input and produce the required output. The unit tests previously conducted focussed on white box testing of each component. Each component will have been deemed functional on its own, leading to the integration tests to determine if the most related components function together. This does not require knowledge of the code, hence the black box approach.

Test coverage is determined using Equation 1. As the program currently stands, the total number of items present is 15. This value is taken from the class diagram in the previous deliverable. Based upon the previous list of integration testing order, 11 different items will be tested. This leads to the planned integration testing coverage of 73%.

Equation 1: Test Coverage

$$Coverage = \frac{Number\ of\ Items\ Excercised}{Total\ Number\ of\ Items} \times 100\%$$

The differences between the integration testing for each of the three platforms is relatively minor. The order that the classes and subsystems are tested has already been described. For the PHP platform, a few extra classes and subsystems are involved just by the nature of the programming language. One of the extra subsystems necessary to have is the code that handles input for one of the use cases of interest. For example, the addmenuitem code is required as an intermediate that takes the input from the view and provides it to the controller. This code will be involved in the integration testing, in addition to the subsystems and components present in the java code, as it is required for the program to function.

The Android code is heavily based upon the Java code, as it uses the jar file from the Java code as a basic. Thus, the testing for this platform is essentially identical to the testing for the Java code aside from the different tools used to conduct the tests.

Once the integration tests had been performed, the system can be tested as a whole. System testing is the last testing stage for the FTMS. After running each of these different hierarchies of tests, though neither exhaustive testing nor 100% coverage were performed, it can be said with a high degree of confidence that the program is functional and ready for release.

3 System Test Plan

System Testing will occur after each component and integration test is finished and the system is first assembled. After that, every time a new update/upgrade to a unit goes up though the stack and eventually gets merged into master the system tests will be run again. These tests can be grouped together into deliverables or patches, although this is discouraged due to ease of testability constraints. If you will group updates together, please provide justification for the packaging; be specific.

As the system is developed with quite a few private and overwritten functions it is best to test the system by running it with the Java UI (simply executing the program). By running it as such we can see exactly how the implementation is incumbent upon the system as a whole. Since security is not a vital parameter, this is a valid form of testing. Since the program is also only used by 1 person at a time, load tests are vestigial.

When performing the system test please perform the following actions:

-Request menu items that don't exist

-Request menu items that can't be made

-Request menu items that can be made

-Access the menu when there is nothing there

-Access the menu when it is populated

-Populate the menu

Since this is done mostly via the GUI, it is hard to use a testing platform like JUnit to test the states of the GUI. Especially, with private encapsulation. While it may be a reasonable choice to test each different test in the suite by editing the persistence XML, since verifying the GUI was accurate would be too complex it is considered out of the way. By using this black-box technique, we also allow for a much faster testing cycle.

4 Updated Work Plan

The blue colour text shows the updates that were made to the work plan during the previous stage in the design process. A list of the functional and non-functional system requirements is present after the work plan for reference.

4.1 Deliverable 1:

We created the functional and non-functional system requirements, which took around an hour. We got together as a group and created the domain model in 30 minutes. The Use cases, requirements-level sequence diagrams and implementation-level sequence diagram each took one hour. We separated the work of the source code for the 3 different prototypes and it took a total of 5 hours.

The FTMS was written in Java, Android, and PHP in order to meet non-functional requirement #9. The other non-functional requirement that was addressed in this deliverable was #8 related to the persistence of the system. The produced code only contained the use case implementation for ordering. Thus, the functional requirements of #3, #4, #5 and #6 were addressed. The functional requirements #1 and #2 relating to the use case of scheduling staff, were left to be completed in a later deliverable. It was also not of particular concern to address functional requirements #11 and #12 as this was only a prototype program.

All of this was done by October 17th.

4.2 Deliverable 2:

The block diagram shows the overall architecture of our software. It shows the relationships and dependencies between the main systems. We only show the systems on a global level, without writing details about those so it took around 4 hours to create.

The class diagram is a more detailed design of how the different classes interact with each other and the methods they use. We need the domain model and use cases to help create the class diagram. Having these in hand, this took around 2 hours. We created the class diagram for the desktop application. These were done by October 31th.

4.3 Deliverable 3:

Once we know what functions we want to have in place and what they should perform, we will create unit tests for all the simple functions we create. This process took an hour and a half and was completed the 14th of November.

Some functions will use a grouping of small functions inside to perform something bigger and more important in relation to our program. This is referred to as the integration testing. The purpose of integration testing is to test the functionality for each of the created use cases. The method of integration testing applied was the top-down method. Implementing tests in this manner took 2 hours. The documentation produced on the integration tests was completed within 2 hours. Both the tests and the documentation were completed on November 13th.

Finally, we created tests on a global scale of the software to see if our main functionalities work and if we reach our goals regarding the program. The systems tests should be faster to create than the component tests, and even faster than unit tests because we are working on the most global scale and we have fewer things to test. These tests stack up to each other. If the unit tests are not successful, then the component and system tests will not be either. The system tests were finished by November 12th.

Performance/stress testing will be done after all tests pass. We will try to run our program with large amounts of objects and see how well it performs. This test should take the least amount of time, so around 30 minutes.

This deliverable largely concerns itself with creating tests for the program. Testing relates to the non-functional requirement #10 while also providing physical evidence that all of the functional requirements #1-#6 have been properly met.

These were done by November 14th. As this deliverable was to be done in two weeks, the group members were aware of their schedules. It had been decided that a meeting would be held on the weekend of November 6th-7th to begin discussing the deliverable and allocating tasks for each of the members. Each member has mostly focused on one platform implementation until this point and this production method will continue for the testing stage. Allowing for potential issues during the testing stage, it was initially expected that each program will take one person 6 hours to complete.

4.4 Deliverable 4:

The release pipeline will require collaboration between all of our team members as it will integrate all the different tools of our software and release it to the user. It should show all the different stages of handing the program from the team to the user. It shows which stages trigger others automatically or manually. For example, once all the tests and builds are successful, the test stage will manually trigger the release stage. An expected time of 3-5 hours will be dedicated for this per group member.

This stage in the FTMS project concerns itself with collaborating the software processes into a pipeline. Through this pipeline changes can be quickly propagated and considered for release. The changes are not released instantaneously, but are delivered frequently in small bursts of updates.

This is a shorter time frame than previous deliverables. The pipeline plan will be done in one week by November 21st.

4.5 Deliverable 5:

Once the pipeline plan is complete and we have all other documents and systems created for our software, we will be ready to start preparing our presentation. We need to have a clear understanding of our program in order to present it to others and we need to encapsulate every part of it. This is expected to take the longest amount time so we will dedicate 7-10 hours for it.

The presentation material should be done by November 27th so we have time to get prepared for the actual presentation.

4.6 Deliverable 6:

After the presentation is complete, we will take as many critics into account to improve the last bits of our program before submitting the full implementation. We will use an estimate of 2-3 hours for this part.

The final source code should be done by December 15th.

5 Reference System Requirements

5.1 Functional System Requirements

The system shall:

#1 - High Priority: Keep track of staff (i.e., their schedules and roles).

#2 - High Priority: Include functionality to manage staff (add, modify, remove).

#3 - High Priority: Keep track of available equipment (grills, fryers, etc.)

#4 - High Priority: Keep track of available ingredients (meat, lettuce, cheese, etc.)

#5 - High Priority: Keep track of menu items and their popularity.

#6 - High Priority: Handle customer orders if there are enough ingredients for the selected menu items and automatically update remaining ingredients in the case of a successful order and increase the recorded popularity of items ordered.

5.2 Non-functional System Requirements

The system shall:

#7 - High Priority: Support laptop/desktop machines, mobile devices, and web browsers.

#8 - Medium Priority: Support persistence, with changes from previous sessions being stored locally.

#9 - Medium Priority: Perform operations swiftly in a manner which minimizes user frustration.

#10 - High Priority: Ensure that the system is trustworthy in such a way that no data can be lost in the case of software malfunction.