1.0 Unit Test Plan:

Class Testing:

The classes that do not need to be tested are the classes that are associated with the view of the application. The rational for this is quite simple. The view class always starts up in the same state and loads up information that it receives from the persistence layer and therefor testing should be focus on persistence and not on the view. Furthermore, the view is used to display errors and successes that it receives from the controller classes. All the information that it must display is focuses in other classes and this class is merely just a way of communicating that information to the user and therefore if the other classes are tested to the point of acceptability, then the view would be acceptable as well. The view will be tested to the extent of just checking if it loads the proper information.

The only classes that must be unit tested are the controllers and the persistence class. The reason that the controllers must be unit tested is because we are inputting information that must be validated before continuing to the next sequence where the information is added into the FoodTruckManager. The application has many controller classes; a supply controller with two sub-controllers called FoodSupplyController and EquipmentController, a employee controller and a menu item controller. The supply controller can add, remove and view supply. The ability to add and remove must be unit tested because these items being added and removed are user input. The unit test will test that the user input is valid and it will test that the expected output is the correct output. The employee controller controls the addition and removal of an employee, assigning an employee a schedule and viewing and employee list and schedule. The same tests will be conducted for this controller as well. One thing to note is that the view methods of the classes will not be tested using unit tests because if the other methods are working or acceptable, than the viewing methods merely access the information and display it and this introduces little space for error and it will be a waste of time to test. The menu item controller can add a menu item, can claim an order (input sales of an item) and view the popularity report of the menu. As stated above, the classes with user input must be tested; add menu item and claim order. The view popularity report will not be tested for the same reason as the other viewing methods. For all the classes, we must make sure that the controller is correctly throwing an error/exception when needed.

The reason that the persistence class should be unit tested is because we must make sure that information from the FoodTruckManager is written to the XML files. The persistence layer must only be called if no exception or error occurs in the controllers, so the unit testing for this must be tested after the testing of the controllers or by adding a driver and by assuming that the input is inputted properly.

General Description of Unit Tests:

The way that we are producing the unit tests is simple, whenever a system requires user input, it will be tested to see if it has proper functionality. Anything that writes to persistence must also be tested and the persistence itself must be tested. Each class that can produce an error will also be tested. We will also analyse the ranges of input that are possible. For example, many methods should not allow negative numbers to be added and it should allow any number that is positive to be added. This will allow us to come up with good test cases. Partition style testing will be used in the identification of the values to test.

For example, addFoodSupply must produce an error for negative values. We will make a case where we add a negative number of a food supply and this unit test will check if an error is thrown. If the thrown error matches what is expected, then the unit test was passed. Then we will add a certain number to check whether it gets added. Then we will add an additional amount to make sure that the two numbers are added together.

Our testing goal is 70% - 100% testing coverage. Since we can determine the ranges if input, we can test specific situations to increase our testing coverage. As shown in the above example, the testing coverage for that portion will fall within the 70% - 100% range because we covered negative numbers, positive numbers and zero. The only numbers that haven't been tested are very far off edge cases that deal with input of numbers that are greater than the storage capacity

testing coverage for that portion will fall within the 70% - 100% range because we covered negative numbers, positive numbers and zero. The only numbers that haven't been tested are very far off edge cases that deal with input of numbers that are greater than the storage capacity of our variable. We are assuming that the user will not enter a number that exceeds the size of an integer type for example since that case is highly unlikely.

Test Execution:

Our tests will be executed after we finish an iteration. This is to make sure that we have completed the iteration and we will be able to see that it functions. Testing after each iteration will also allow us to make sure that this iteration didn't break any functionality in the previous iteration. It will allow us to observe the effects of the changes on the system. The Desktop and Mobile Applications will use JUnit Testing because they are both coded in Java, and the Web Application will use PHPUnit, which is specific to PHP. Because PHP uses its own persistence and has its own 'language' when writing to the txt file, its persistence does not have to be unit tested. We must unserialize the data using something like TextEdit to manually see if the data has been written.

2.0 Integration Strategy:

The chosen integration strategy was top down integration. This was the chosen method because we are able to test the higher levels before the lower levels. For example, we first create the view and open it to make sure it looks as planned and make sure it can take in simple input. The functionality for accepting the input and checking its validity is initially stubbed out. After the view is looking acceptable, we then move on to writing the classes that will handle the inputted information. We write the Supply controller and then test that it accepts input and handles exception but this time we stub out the persistence later and just update the view without persistence. We do the same for the other controllers and when all of those are tested, we add the persistence layer and perform the necessary unit tests described in the unit test plan to determine its functionality.

The order of the testing will be as follows:

1. Begin by creating the overall view and make sure that input can be taken in. Stub out all the controller and persistence classes.
2. Begin by implementing the controllers to test the input that is received from the view and handle any errors. Stub out the persistence layer. Make sure that the view is being updated.
3. Add the layer of persistence and make sure that data is being saved and accessed by the view. Test to check if only proper data is added.

Order of class testing including subsystems:

1. View
2. Supply Controller
    a. Food Supply Controller
        i. Add food supply
        ii. Remove food supply
    b. Equipment Controller
        i. Add equipment
        ii. Remove equipment
3. Employee Controller
    a. Add employee
    b. Remove employee
    c. Assign schedule
        i. View schedule
    d. View employee list
4. Menu Item Controller
    a. Add menu item
    b. Claim order
    c. View popularity report
5. Persistence

Component testing will be used in addition to the unit tests. The components will be the controllers shown above. Once a component is completed, it will be tested. This is a good method to use since all the controllers are independent of each other so not much stubbing is going to be needed to accomplish the tests. The only class that will need to be stubbed when testing the controllers will be the persistence layer. This will help us determine whether that

method to use since all the controllers are independent of each other so not much stubbing is going to be needed to accomplish the tests. The only class that will need to be stubbed when testing the controllers will be the persistence layer. This will help us determine whether that component has any defects and whether it is functioning. Using this method will allow us to see that the controller is updating the view as needed and producing errors as needed as well. By testing a whole component, we will be able to see how the different methods interact with each other and see if they cause any problems when assembled. For example, the add and remove methods might work perfectly when they are alone but may introduce problems when used together. For example, the remove method might not see that the add method increased the amount of supply and this will yield an issue. So, they must be tested together to see if they work together as expected. Once a component is completed, they will be tested.

Testing goal:

The testing goal is 70% - 100%; similar to the unit testing cases. The reason for this is due to unforeseen circumstances such as type overflow. Using component testing, we will be able to achieve this. The unit tests tested the component's subsystems alone but this will test the components. Using the same rational as in the unit tests for determining which inputs to test, we can reach this testing goal. The component tests will use the unit tests but combine them to test a whole component. For example, if a unit test tested the adding of supply and another one, tested the removal of supply, we can combine those tests into one component test. In that test we will add and then remove to see if the expected result occurs rather than doing them individually. This will yield many test cases. We can have a set of inputs that add and then remove and then another set of inputs that remove and then add but we will be using a partition style to get a wider coverage. The differences in the testing are the same as the unit test plan. Since the java application and the android application both use java, they can be tested with Junit while the PHP application can be testing with PHPUnit. The main difference between the unit test and the integration test is that the Junit and PHPUnit tests are being run on the individual subsystems or methods and for the integration test, the Junit and PHPUnit tests are being run on whole components.

System Test Plan

Test Situations:

1. View a staff member's schedule by selecting a staff member
    b. Check for error when not selecting a staff member
2. View staff members in system
3. Add a staff member to the system by entering a name and role
    a. Check for error when not specifying a name
    b. Check for error when not specifying a role
4. Remove a staff member from the system by selecting a staff member
    a. Check for error when not selecting a staff member
5. Assign a schedule to a staff member by selecting a staff member in the system, a date, a start time and an end time
    a. Check for error when not selecting a staff member
    b. Check for error when not selecting a date
    c. Check for error when not selecting a start time
    d. Check for error when not selecting an end time
    e. Check for error if date is before current date
    f. Check for error if start time is before current time
    g. Check for error if end time is before start time
6. View the food supply in the system
7. View the equipment supply in the system
8. Add to the food supply by indicating a name and an amount
    a. Check for error if no name is provided
    b. Check for error if no amount is provided
    c. Check for error if negative amount is provided
9. Remove from the food supply by indicating a name and an amount
    a. Check for error if no name is provided
    b. Check for error if name does not match an existing food supply's name
    c. Check for error if no amount is provided
    d. Check for error if negative amount is provided
10. Add to the equipment supply by indicating a name and an amount
    a. Check for error if no name is provided
    b. Check for error if no amount is provided
    c. Check for error if negative amount is provided
11. Remove from the equipment supply by indicating a name and an amount
    a. Check for error if no name is provided
    b. Check for error if name does not match an existing equipment supply's name
    c. Check for error if no amount is provided

equipment supply's name
    c. Check for error if no amount is provided
    d. Check for error if negative amount is provided
12. Add a menu item to the system by specifying an item name
    a. Check for error if no name is provided
13. Claim that a menu item was ordered by specifying the menu item that was ordered and the amount that was ordered
    a. Check for error if no menu item is provided
    b. Check for error if no amount is provided
    c. Check for error if negative amount is provided
14. View the popularity report displaying the most popular menu items and their corresponding amount of times ordered

Rationale:

Our test situations chosen correspond to each of the use cases in the requirements use-case diagram. Thus, we are testing every feature of our system that a user can manipulate. Moreover, our test situations include testing each use case's failure scenarios. Since we are testing all use cases and alternative flows, we have 100% test coverage.

Detailed Test Case #1:

Title: Assign schedule to a staff member
Actors: User
Requirements: 1003
Main Scenario:

1. User selects a staff member from the drop-down menu
2. User selects a date from the date picker
3. User inputs a start time
4. User inputs an end time
5. User clicks on assign schedule button
6. Schedule is assigned to selected staff member

Alternatives:
1a: User does not select a staff member from the drop-down menu
1a1: User clicks on assign schedule button and views error message; use case ends
2a: User does not select a date
2a1: User clicks on assign schedule button and views error message; use case ends
2b: User selects a date prior to current date
2b1: User clicks on assign schedule button and views error message; use case ends
3a: User does not input start time
3a1: User clicks on assign schedule button and views error message; use case ends
3b: User inputs start time prior to current time
3b1: User clicks on assign schedule button and views error message; use case ends
4a: User does not input end time
4a1: User clicks on assign schedule button and views error message; use case ends

4a1: User clicks on assign schedule button and views error message; use case ends
4b: User inputs end time prior to start time
4b1: User clicks on assign schedule button and views error message; use case ends

Test Situations:

1) User inputs all data successfully
2) User doesn't select staff member but inputs other data correctly
3) User doesn't select a date but inputs other data correctly
4) User selects a date prior to current date but inputs other data correctly
5) User doesn't input start time but inputs other data correctly
6) User selects today's date and inputs a start time prior to current time and all other data correctly
7) User doesn't input an end time but inputs other data correctly
8) User inputs an end time prior to the inputted start time and all other data correctly

Test Coverage:

Base: number of main and alternative scenarios = 8
Test situations cover all 8 cases. 100% coverage of use case

Detailed Test Case #2:

Title: Add an employee to the system
Actors: User
Requirements: 1002
Main Scenario:

1. User inputs the name of the employee
2. User inputs the role of the employee
3. User clicks on hire employee button
4. System adds employee

Alternatives:

1a: User does not input a name
1a1: User clicks on hire employee button and error message is displayed; use case ends
2a: User does not input a role
2a1: User clicks on hire employee button and error message is displayed; use case ends

Test Situations:

1) User inputs all data successfully
2) User does not input a name but does input a role
3) User does not input a role but does input a name

Test Coverage:

Base: number of main and alternative scenarios = 3
Test situations cover all 3 cases. 100% coverage of use case

Group 12

Work Plan for Next Iterations

Ecse – 321

Note: Everything that is highlighted in green is complete.

Note: Everything that is in italics indicates a change in the work plan

Below will list the remaining functionality to be added to the project along with the tentative date of delivery.

Note that all the programming for iterations up to and not including 8 have been completed for the java portion of the project. Only the android and php versions remain.

1ST Iteration:

- Add supply
    - Completion Date: First deliverable
    - Effort required: Medium

2ND Iteration:

- Remove supply
- Add/Remove Equipment
    - Completion Date: October 19th 2016
    - Effort required: Medium

3rd Iteration:

- Add Employee
- *Remove Employee*
- Add Schedule
    - Completion Date: October 25th 2016
    - Effort required: A lot

4th Iteration:

- Interface Implementation (Create a better view)
- Testing high priority items
    - Completion Date: October 29th 2016
    - Effort required: medium

5th Iteration:

- Display Supply
- Display Equipment

    - Completion Date: November 13th 2016
    - Effort required: A lot

6th Iteration:

- Check Staff Schedule
- View all staff members
    - Delivery Date: November 20th 2016
    - Effort Require: A lot

7th Iteration:

- Update supply
- Keep Track of the Sales of Menu Items
- Popularity Report
    - Delivery Date: November 30th 2016
    - Effort Required: A Lot

8th and beyond (If necessary)

- Bug Fixes
- Error Fixes
- Unforeseen Issues
- Further Testing
    - Delivery Date: TBD
    - Effort Required: TBD