**Description of Architecture of Proposed Solution Including block Diagram**

Subsystems:

addFoodSupply(): adds food supply to the Controller. If the item already exists, add 1 to the number.

removeFoodSupply(): removes food supply from the Controller. Makes sure that the number of items is not less than 0

viewSupply(): shows a list of supply

addEquipment(): adds equipment to the Controller. If the equipment already exists, add 1 to the number.

removeEquipment(); removes equipment from the Controller. Makes sure that the number of items is not less than 0.

addEmployee(): adds employee to the Controller. If the employee already exists, add 1 to the number.

removeEmployee(): removes employee from the Controller. Makes sure that the number of employees is not less than 0.

viewEmployees(): shows a list of the employees

assignSchedule(): assigns schedule to an employee

viewSchedule(): shows the schedule

getMonth(): returns a month based off the number given

addMenuItem(): create a new menu item

claimOrder(): select the food

viewPopularityReport(): shows popularity report based off the orders

Discussion:

This FoodTruckManager uses the MVC architectural pattern. This style was applied because there would be a singleton Manager and the Model is dependent on the View which is dependent on the Controller. The same architecture is used in the desktop, mobile and web applications. The only difference is the view based on the platform. Furthermore, this pattern is being applied because the design of this application is very similar to that of assignment 1 and so we decided that this would be the best option. The use of singleton is a plus for this system since we know that only one instance of the manager is needed and the application can easily access the information using the getInstance() method.

All the data is being shared by all the components using the singleton style. This allows every method the capacity to access all the data saved through persistenceXstream. The view is altered by the user and this sends the user interactions to the controller which then executes the appropriate functionality. After that, the controller updates the view and waits for more user input. If an error were to occur the controller would update the view to communicate this. The state of the model is then changed and the umple generated model is used. The state is then updated and the Model then updates the persistence layer if any changes are to be committed. The view then gets its current state from the model to check if any changes were made to the persistence file. For example, is an employee was added, the model state was changed and this new employee has been saved to the xml file. The view needs to access the xml file through the model to give the user the list of current employees if the user wishes to then do something to that employee’s data.

Other architectures styles were not applied for the simple reason that MVC was best suited to this situation. Since the three platforms did not need to be linked or connected to a server, the client-server architecture yields no benefits. Layered architecture would be a pain to implement mainly due to the way that all the data is shared. Therefore, the repository architecture wasn’t applied either. Finally, pipe and filter was not used because the processing of information in this case was simple enough to be handled by MVC. Pipe and filter is useful for more complex systems where several steps of data processing occur and that is not the case here.