McGill University

ECSE 321 – Group Project Deliverable 2

Design Specification

Prepared by:

Brent Coutts (260617550)

Rony Azrak (260606812)

Jamie Day (260665720)

Max Johnson (260498322)

October 31st, 2016

1 Description of Architecture of Proposed Solution Including Block Diagram

This food truck management system was created using the model view controller (MVC) architectural pattern. This pattern breaks the full food truck system into three subsystems, the model, the view, and the controller. This is a successful architectural pattern, because it separates the three main concerns of a common application: application data representation, application data manipulation, and user interaction with the application.

It is sensible to separate the full system into subsystems that each address a separate concern. This compartmentalization means that updates to one of the subsystems should not have an impact on the other concerns. Program maintenance is simplified as it is possible to update one subsystem separately from the other two. This is a much more desirable alternative than having to update the whole system, when it is only required to update one concern of the application. For example, with the MVC architectural pattern, it is possible to update the application view without requiring the rest of the system to be changed.

The MVC architecture pattern contains three subsystems, each of which directly and specifically relates to the previously stated three main program concerns. The model component handles the application data representation concern. The system data and the operations performed on this data are handled by the model subsystem. The logic and associated rules present in the application are also contained within the model. This subsystem is the central component of the entire system as it dictates the core behaviour of the program. For example, the model of the food truck management system contains the behaviour that an order can be placed for a certain menu item (along with many more class attributes and class associations).

In order to produce the model, it is efficient to use domain modelling instead of writing the code in its entirety. Umple online was the program of choice used to produce the model for this food truck management system. This domain modelling software allows for relatively quick and simple model development. It functions by taking the text input of a simple coding language and producing an output of a class diagram. The visualization of the class diagram greatly aided the production of the model. This class diagram provided a clear representation the attributes contained in each class, as well as how each class is associated with other classes in the model. Once the class diagram was satisfactory for the problem statement of the food truck management system, Umple was used to generate Java and PHP code. This was all of the coding that was required by the model subsystem.

User interaction with the application is handled by the view subsystem. In order for the user to be able to interact with the system, it is required for the system to have a view component. This subsystem dictates how the data is presented to the user. In the case of the desktop and Android implementations, the data is presented in an application window. The PHP implementation has a view that dictates the data presentation in a web browser window. It is clear that the user requires a view of the system in order to know how to interact with the program, as they would otherwise not know what functionalities the program offers.

Finally, the concern of manipulating the application data is represented by the controller subsystem in the MVC architecture. User interaction with the program, such as typing into text fields and clicking buttons, is manipulated within this component of the architecture. The controller takes an input and converts it into commands which are understood by the rest of the subsystems.

This MVC architectural pattern was chosen because it has the most relevant advantages with regards to the program requirements. A major requirement of this food truck management system is that it is implemented on desktop, mobile, and web platforms. The MVC architecture is an excellent pattern for when multiple platforms are required, because the model component is reusable for any of the platforms. The potential for subsystem reuse decreases the time spent on subsequent platform implementations. Furthermore, the view and controller subsystems have a tendency to not require as much code to perform their functionalities. Thus, the capability of reusing the model is a huge benefit to this architectural pattern.

Another advantage of choosing this architectural pattern is that each subsystem can be updated individually. As program requirements are updated and functionalities are implemented or removed, the pertinent subsystems can be changed to reflect the changes without changing the rest of the subsystems. It is possible to have multiple interpretations of the provided requirements specification. This architecture is useful when the future requirements for interaction are not necessarily known, as changes to the subsystems can be isolated. The main disadvantage of this architectural style is that it is overly cumbersome when the interactions of the program are simple. However, as the food truck management system requires a large number of different interactions, this architectural pattern is an efficient method for the production for the system.

Other architectural patterns such as: client-server and pipe-and-filter architectures were also explored before choosing the MVC pattern. The client-server pattern offered the advantage of allowing updates to the data in the system to be seen across all platforms. However, it was not required that the data alterations be shared between each of the platforms. Also, it must be considered that food trucks would not necessarily have access to an internet connection in order to interact with the services provided within the architecture.

The pipe-and-filter architecture offers a very logical flow of data through each of the components. This pattern is logical for business operations (i.e. managing a food truck) and is easy to understand. However, this pattern requires the system to be very rigid as the data must be communicated between components in a very specific way. If it was realized during the coding process that a mistake was made in the flow of a pipe, the code refactoring required could potentially be major. Changing the flow of a pipe would require all of the filters along the pipe to also be changed. The program requirements are relatively ambiguous, potentially leading to many different interpretations of the system. The pipe-and-filter architecture is simple when the data flows through the system are simple and rigid. It is not as simple in the pipe-and-filter pattern to have constant changes in the system interactions. The potential for changes in the system interactions could cause large amounts of code rewriting; whereas, it is simpler in the MVC pattern to update the model without as much risk of affecting the other subsystems.

Now that the function of each subsystem is understood, the interactions between the subsystems can be explored. The relationships between each component in the MVC architecture are visualized in the block diagram present in Figure 1 below. The user sees a view of the application, showing the functionalities that the program is capable of. Depending on what they require from the program, the user interacts with the application. This interaction can be in the form of text field inputs, clicking buttons, or selecting options from a spinner. Each of these actions is converted into a command understood by the system using the controller. The controller sends the commands to the model. The model stores the input data and performs the necessary operations based upon the commands sent by the controller. Once the model has manipulated the data, it sends the updated data to the view subsystem. The view is updated, allowing the user to see the change that their input made to the program.

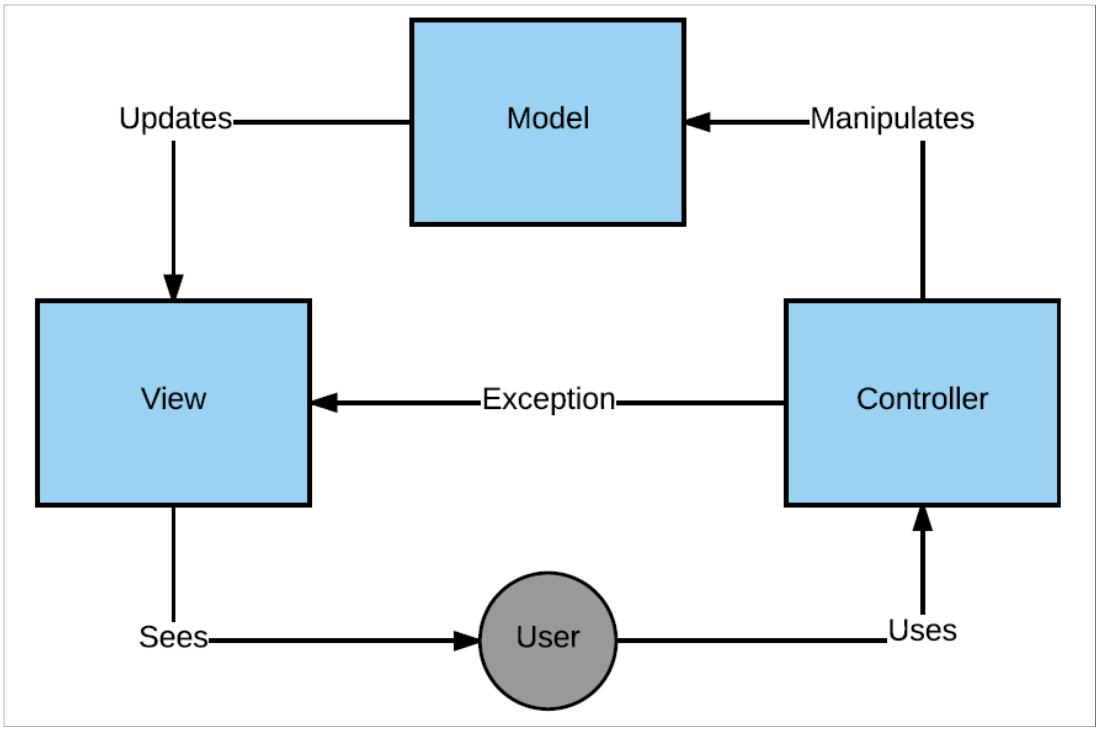

Figure 1: Block Diagram of MVC Architecture Pattern

The user input into the application is taken by the controller. If the data input is invalid, and unable to be understood by the model, the controller will throw an exception to the view. The view will take this exception and print an error message for the user. If the controller receives valid input, it is transferred to the model. The model performs operations on the data, such as updating the quantity of a food supply. The model may or may not pass data back to the view. If the information updated is present in a spinner of the view, the model will pass the data to the view so the user can see the new option. Though, if the data input by the user is not present in a spinner, it is not necessary for the model to pass the data to the view as the data is not used in this subsystem. An example of this is if the user inputs a new employee schedule. The data will be stored, but it will not be passed to the view unless a further command is applied to retrieve the schedule information.

As the program should function in the same way regardless of the platform, the model does not need to be changed on any platform. The controller interacts with the model in a different manner for the web application and the view subsystem is clearly different for each of the different platforms. The interactions between each of the subsystems, however, remains the same. Though the view or controller components may need to be updated for the new platform, their functionality is very similar and their interactions in the MVC architecture are the same.

The same architecture pattern was used for all three application platforms (desktop/mobile/web). One of the large advantages of the MVC pattern is that its structure allows for simple reuse of the model between different platforms. It would therefore be a waste of effort to produce this architectural pattern for only on or two of the platforms as the other platforms would need to be created from nothing. If the desktop Java code is produced first, it can be exported to the Android Studio Project. This allows both the model and the controller to be reused for the mobile Android application.

Also, programming in the same pattern causes the programmer to become more adept at working with that architecture. Changing from working on one platform to another requires almost no thought, other than changing the programming language that is being coded in. When the same architectural pattern is used, collaboration between the platforms is maximized. Solutions to issues encountered in one platform implementation can be transferred to other platforms, reducing the time spent debugging the code. Reusing the MVC architectural pattern for each of the platforms was considered to be the most efficient course of action for producing the food truck management system.

2 Description of Detailed Design of Proposed Solution Including Class Diagram

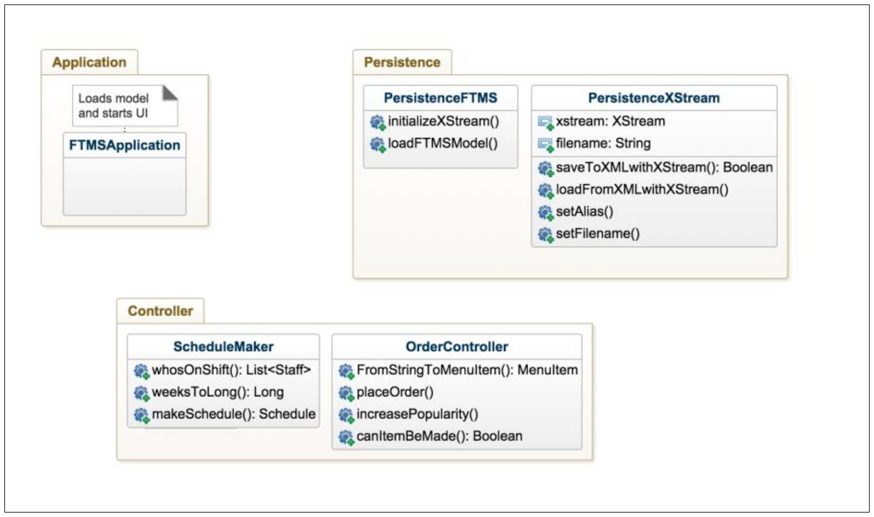

Figure 2: Class Diagram for Controller Subsystem and Persistence

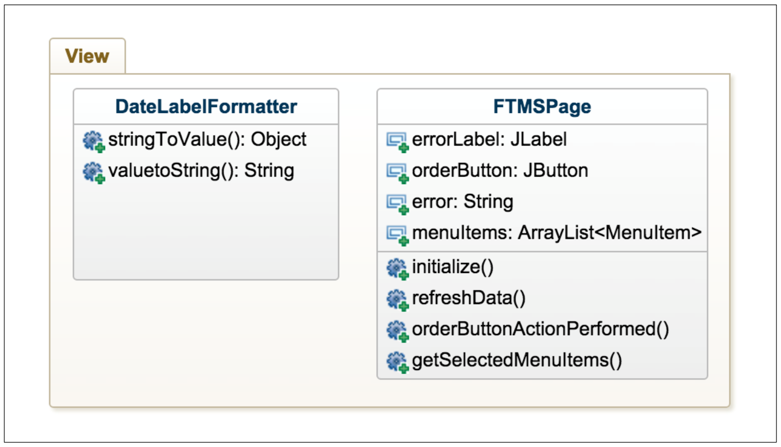

Figure 3: Class Diagram for View Subsystem

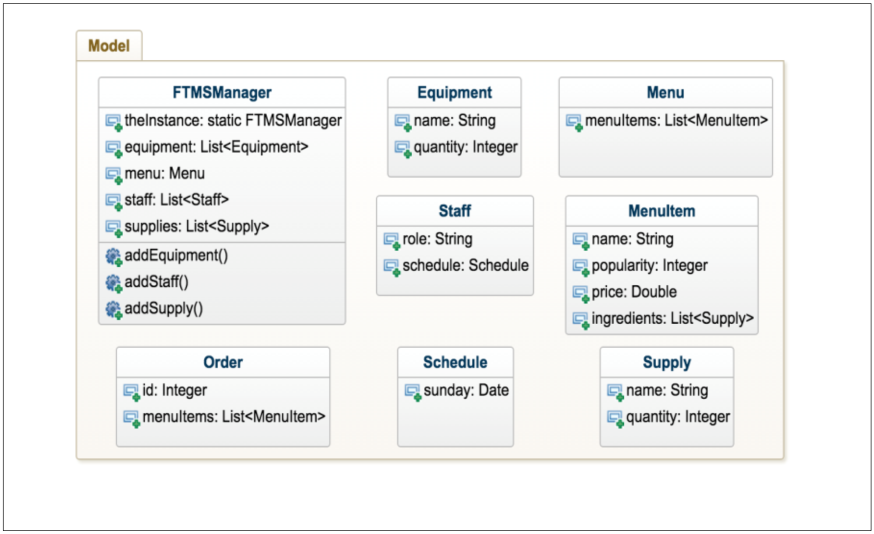

Figure 4: Class Diagram for Model Subsystem

FTMSManager – We are using the singleton design pattern. The FTMSManager has a static instance that manages all the other classes in Model.

FTMSApplication – This class simply serves as the entry point to the application. It calls upon the PersistenceFTMS subsystem to load persistence data, and starts the UI.

FTMSPage – This class is the desktop view. It calls the Java Swing & AWT APIs in order to display appropriate content retrieved via the controllers. It also fires events, which call controller methods.

OrderController – This is a controller class containing all the methods necessary for handling an order. Currently it is called statically from events fired by the View.

ScheduleMaker – This is a controller class containing all methods for interacting with staff schedules. It contains all the necessary functions and utilities for schedule creation and display. Called from the View.

3 Updated Work Plan

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

3.1 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 food truck management system 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.

3.2 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.

3.3 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, which should not take more than an hour. Some functions will use many simple functions inside of them to perform something bigger and more important in relation to our program. We will create component tests for those, which should also take around an hour. Finally, we will create 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.

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 #12 while also providing physical evidence that all of the functional requirements #1-#6 have been properly met.

These should be done by November 14th. As this deliverable is due within the next two weeks, the group members are aware of their schedules. It has been decided that a meeting will 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 is expected that each program will take one person 6 hours to complete.

3.4 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.

The pipeline plan should be done by November 21st.

3.5 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.

3.6 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.

4 Reference System Requirements

4.1 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.

4.2 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.