“The Convertor Project”

EECS3311: Software Design

TA: Kazi Mridul

Bryce Cooke – 217346354 – Section A

Farnad Kazem-Zadeh – 214212856 – Section A

Jasleen Chagger - 217546938 - Section A

Ilan Zar - 213025598 - Section A

Part I: Introduction

Explain what the software project is about and what are its goals:

This software project asks us to implement a converter tool which converts values from centimeters to feet and meters respectively. Alongside this conversion we had to provide an interface for the user to input their desired conversion, in centimeters, and also output our returned numbers, which have been converted. This interface also expects us to incorporate colors, to differentiate between the measurements and make it visually easier on the user, this is done using yellow for the user input in centimeters, orange for the measurement in meters and green for the measurement in feet. In terms of how the process of conversion is actually done, our program must implement functionality for storage of the value input by the user via a JMenuBar, which prompts the controller to store the input, in centimeters, in the model. The green and orange boxes observe the model and upon storage of the input by the controller are both updated with their respective measurements.

Explain the challenges associated to the software project:

We anticipate the challenges with this project to arise with the actual implementation of the controller and the model. This is our first time attempting to implement said design pattern and as a result will require us to do some research into how implementation will actually occur. This should prove to be challenging and informative. Alongside this we expect some difficulties when working with JFrame and JPanels.

Explain the concepts (e.g., OOD, OOD principles, design patterns) you will use to carry out the software project:

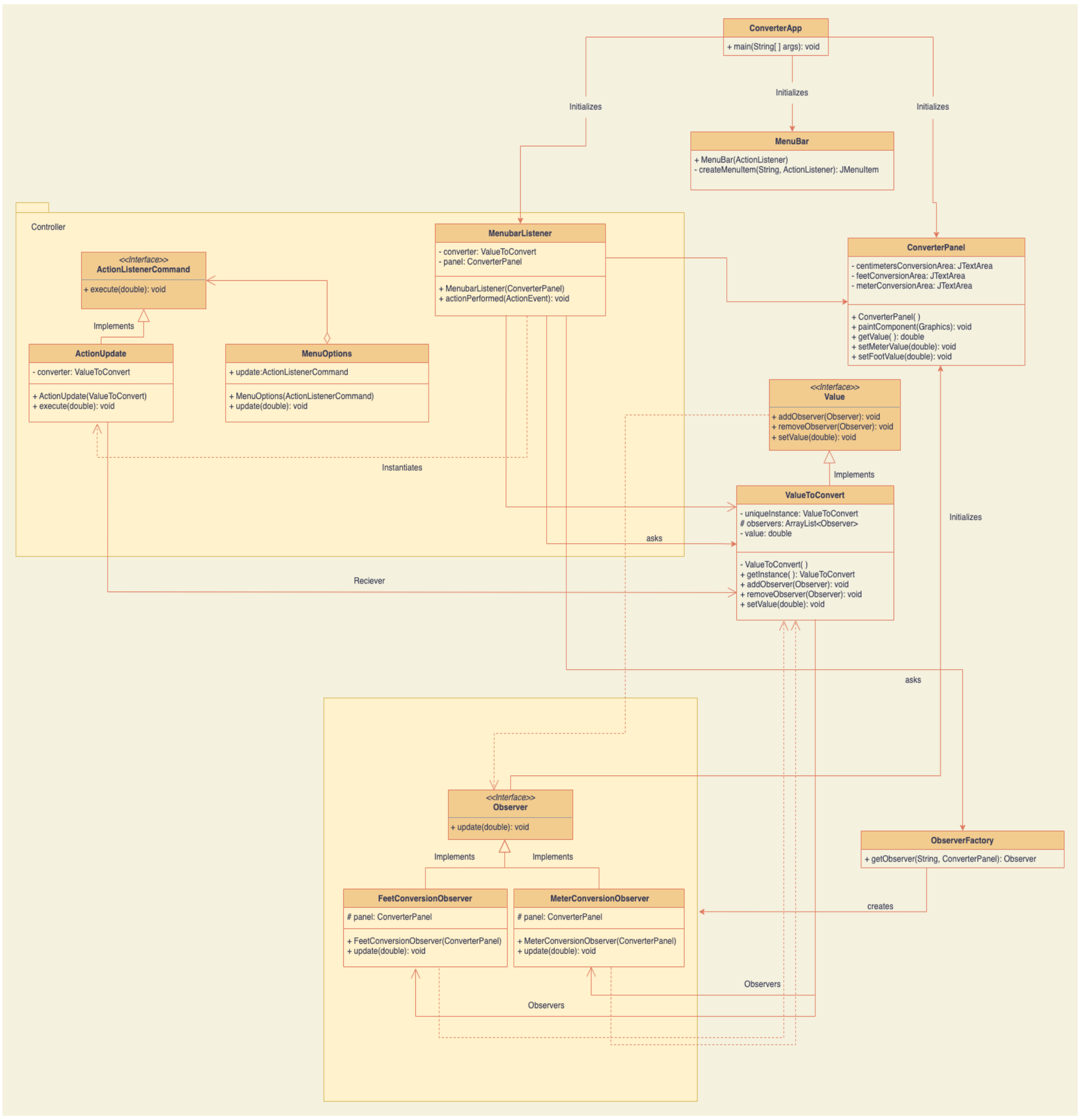
This project will require the usage of many different OOD design principles. Inheritance is an example, where classes will extend others in order to use their functionality. Encapsulation is also used in this project in order to maintain the state of the model. We want to ensure only the objects themselves can modify their states, and this is achieved using the private keyword combined with methods that allow outside classes to request modifications of the objects state, in this instance the controller requesting the model to update the view.. Abstraction is the concept of only sharing what is necessary with other objects which is important to our program as it ensures that we provide functionality to the controller to use methods from the model without knowing directly how it works. The design pattern of model-view-controller will be implemented to help with management of the Graphical User Interface (GUI).

Explain how you are going to structure your report accordingly:

This report will be structured as follows: Firstly we will introduce the design of our proposed solution, using UML diagrams combined with descriptions of our classes, then we will specify how we implemented the program and what tools we used to help with out implementation, then we will end the report with a conclusion summarizing our experiences and what we learnt during this project.

Part II: Design of the solution

Create a first UML class diagram of your system (use at least two design patterns), add the corresponding figure in the report and comment on its elements:


ConverterApp
+ main(String[ ] args): void
Initializes
Initializes
Initializes
MenuBar
+ MenuBar(ActionListener)
- createMenuItem(String, ActionListener): JMenuItem
Controller
MenubarListener
- converter: ValueToConvert
- panel: ConverterPanel
+ MenubarListener(ConverterPanel)
+ actionPerformed(ActionEvent): void
<<Interface>>
ActionListenerCommand
+ execute(double): void
Implements
ActionUpdate
- converter: ValueToConvert
+ ActionUpdate(ValueToConvert)
+ execute(double): void
MenuOptions
+ update:ActionListenerCommand
+ MenuOptions(ActionListenerCommand)
+ update(double): void
Instantiates
ConverterPanel
- centimetersConversionArea: JTextArea
- feetConversionArea: JTextArea
- meterConversionArea: JTextArea
+ ConverterPanel( )
+ paintComponent(Graphics): void
+ getValue( ): double
+ setMeterValue(double): void
+ setFootValue(double): void
<<Interface>>
Value
+ addObserver(Observer): void
+ removeObserver(Observer): void
+ setValue(double): void
Implements
ValueToConvert
- uniqueInstance: ValueToConvert
# observers: ArrayList<Observer>
- value: double
- ValueToConvert( )
+ getInstance( ): ValueToConvert
+ addObserver(Observer): void
+ removeObserver(Observer): void
+ setValue(double): void
asks
Reciever
Initializes
asks
<<Interface>>
Observer
+ update(double): void
Implements
Implements
FeetConversionObserver
# panel: ConverterPanel
+ FeetConversionObserver(ConverterPanel)
+ update(double): void
MeterConversionObserver
# panel: ConverterPanel
+ MeterConversionObserver(ConverterPanel)
+ update(double): void
ObserverFactory
+ getObserver(String, ConverterPanel): Observer
creates
Observers
Observers

Elements of the UML diagram:

Main:

- ConverterApp.java: This class is used to hold the main method. ConverterApp class has no instance variables and main(String[]) is the main method used when running the associated code**.**

Model:

- Observer.java: This is an interface used to create method signatures for the classes that implement it for the Observer pattern. It has no instance variables and has one method named update(double). This method helps to set value in the feetConversionArea in ConverterPanel.
- FeetConversionObserver.java: This is a class that updates the feetConversionArea and acts as a “ConcreteObserver” in the Observer pattern. This class implements the Observer interface. It has an attribute named panel. This class also has a public constructor and overrides the update(double) method.
- MeterConversionObserver.java: This is a class that updates the meterConversionArea and acts as a “ConcreteObserver” in the Observer pattern. This class also implements the Observer interface. Like the FeetConversionObserver class it has an attribute named panel, a constructor and update method overridden from the Observer interface.
- ObserverFactory.java: This is a class that is used to return the specific types of Observers and implements the “Factory” pattern. It doesn’t have any variables but implements a getObserver(String, ConverterPanel) method which returns an Observer type.
- Value.java: This is an interface used as a Subject in the Observer pattern. It has three methods namely, addObserver(Observer), removeObserver(Observer), setValue(double).
- ValueToConvert.java: This is a class that stores the centimeter value and implements the Value interface. It has uniqueInstance, observers and value as attributes. This class also has ValueToConvert( ), getInstance( ) methods and overrides methods from the parent class.

View:

- ConverterPanel.java: This class is used to create the panel that the visuals are displayed on and extends JPanel. It has private variables named as centimetersConversionArea, feetConversionArea, and meterConversionArea. It also has a constructor and other methods.
- MenuBar.java: This class is used to represent what the client will click on. It extends the JPanel class. It has a constructor and createMenuItem method.

Controller:

- ActionListenerCommand.java: This is an interface, and it is used as a part of the “Command” pattern and interface. It serves as a signature for classes implementing it. It does not have instance variables and a method named as execute.

- ActionUpdate.java: This class is used as a ConcreteCommand in the “Command” pattern. It implements the ActionListenerCommand class. It has a private variable named converter and an overridden execute method. It also contains an additional method ActionUpdate(ValueToConvert).
- MenuOptions.java: This class acts as the “Invoker” in the Command Pattern. It has a private attribute, a constructor and update(double) method.
- MenubarListener.java: This class acts as a listener for when the menu bar is clicked. It has two attributes called converter and panel. This class also has a constructor and a method named actionPerformed(ActionEvent).

Explain in your report how you have used design patterns: name the corresponding classes, interfaces, and if possible most relevant operations.

The Command Design Pattern:

- ActionUpdate, ActionListenerCommand, MenuOptions, ValueToConvert, and MenuBarListener make up the command pattern.
- Where, ActionListenerCommand acts as “Command”, ActionUpdate is used as “ConcreteCommand”, ValueToConvert acts as "Receiver", MenuBarListener as "Client", and MenuOption class acts as “Invoker”.

Observer Design Pattern:

- The Value, ValueToConvert, Observer, FeetConversionObserver, and MeterConversionObserver classes make up the Observer pattern and there are two ConcreteObservers instead of one.
- Observer class acts as “Observer”, Value class is used as “Subject”, ValueToConvert acts as “ConcreteSubject” for the Observer pattern. FeetConversionObserver and MeterConversionObserver classes act as “ConcreteObserver” in the Observer pattern.

Singleton Design Pattern:

- To implement the singleton pattern, MenuBarListener class creates a unique instance of ValueToConvert. This restricts the instantiation of ValueToConvert to one "single" instance.
- getInstance() method allows getting the single instance of ValueToConvert class.

Factory Design Pattern:

- The factory pattern is implemented using the ObserverFactory class. The Observer class is extended by two classes i.e., FeetConversionObserver and MeterConversionObserver class. The ObserverFactory class generates instances of FeetConversionObserver and MeterConversionObserver for this project i.e. Observer objects.

- MenubarListener class acts as a "FactoryPatternDemo" for factory pattern.

Use OO design principles in your class diagram. Explain in your report how you have used them: name the corresponding classes, interfaces, and if possible the most relevant operations.

OO design principles:

- Abstraction:
  - Data abstraction can be achieved with either abstract classes or interfaces. There are three interfaces in this project namely Observer, ActionListernerCommand, and Value.
  - Also, we provide functionality to other classes to use methods without knowing directly how the methods actually work.
- Encapsulation:
  - Encapsulation is used to maintain the state of the game. It can be achieved by keeping the attributes of the class private or protected and public getter/ setter methods are provided to manipulate the attributes.
  - All the classes have private attributes in this project to keep the logic inside the classes.
- Polymorphism:
  - When a method needs to change depending on the object using it or on its parameters.
  - A class can inherit from another class and change what the inherited methods did in the parent class.
  - The methods that are inherited from the Observer interface are overridden in FeetConversionObserver and MeterConversionObserver classes. These are implemented according to the requirement. These methods can take many forms as they can be called from Observer but perform the function of the desired class.
- Inheritance:
  - The classes FeetConversionObserver and MeterConversionObserver implements the Observer interface. Also, ActionUpdate implements ActionListenerCommand interface and ValueToConvert implements Value interface. This shows inheritance.
  - These classes have the same functions/methods as the parent class with addition to their own states and operations. The child classes override the methods in parent class.

Part III: Implementation of the solution

<u>Precise documentation</u>

ConverterPanel.java: is a class that is used to create the panel that the visuals are displayed on. This extends JPanel.

- centimetersConversionArea: is a private variable of type JTextArea that represents the centimeter portion and is initialized to zero.
- feetConversionArea: is a private variable of type JTextArea that represents the feet portion and is initialized to zero.
- meterConversionArea: is a private variable of type JTextArea that represents the meter portion and is initialized to zero.
- ConverterPanel(): is a public no-argument constructor. It calls the super classes constructor, JPanel and again for the initial layout. It then sets the three instance variables of this class to the values specified in Lab 6 and adds them using JPanels add method.
- paintComponent(Graphics): is a public method with a return type of void. It passes one argument of type Graphics. The method uses its super class JPanel's paintComponent method and sets the colors and fill the rectangles accordingly.
- getValue(): is a public method with a return type double. It passes no arguments and returns a parsed double version of the centimeter's conversion area's text value.
- setMeterValue(double): is a public method with a return type of void. It passes one argument of type double. It sets the value of meterConversionArea to a text value of the passed in double.
- setFootValue(double): is a public method with a return type void. It passes one argument of type double and sets the text value of feetConversionArea with the double passed in.

MenuBar.java: is a class used to represent what the client will click on. MenuBar.java extends the JPanel class.
MenuBar has no instance variables

- MenuBar(ActionListener): is a public one-argument constructor that passes in an ActionListener. It calls its super classes constructor from JPanel and creates and initializes a new JMenu. Finally it adds the JMenu using the add method from the JPanel class.
- createMenuItem(String, ActionListener): is a private method that passes two arguments with a return type of JMenuItem. One argument is a string and the other is a ActionListener. The method initializes a new JMenuItem using the string passed in and adds the ActionListener passed in to the JMenuItem. Finally the method sets the Accelerator and returns the JMenuItem in question.

FeetConversionObserver.java: is a class that updates the feetConversionArea and acts as a "ConcreteObserver" in the Observer pattern. This class implements the Observer interface.

- panel: is a public variable of type ConverterPanel that holds the information of the panel being worked on.
- FeetConversionObserver(ConverterPanel): is a public one-argument constructor. It passes in a ConveterPanel, uses the super classes constructor and sets the variable panel to the argument being passed in.
- update(double): is a public method of return type void. It passes in one argument of type double and sets the instance variable, panel, foot value to the double passed in.

MeterConversionObserver.java: is a class that updates the meterConversionArea and acts as a "ConcreteObserver" in the Observer pattern. This class implements the Observer interface.

· panel: is a public variable of type ConverterPanel that holds the information of the panel being worked on.
· MeterConversionObserver(ConverterPanel): is a public one-argument constructor. It passes in a ConveterPanel, uses the super classes constructor and sets the variable panel to the argument being passed in.
· update(double): is a public method of return type void. It passes in one argument of type double and sets the instance variable, panel, meter value to the double passed in.

Observer.java: is an interface used to create method signatures for the classes that implement it for the Observer pattern.

Observer.java has no instance variables

· update(double): provides a signature for the update method that will be used by the classes which implement this interface. It is a public method that has a return type void and passes one argument of type double.

ObserverFactory.java: is a class that is used to return the specific types of Observers and implements the "Factory" pattern.

ObserverFactory.java has no instance variables

· getObserver(String, ConverterPanel): is a public static method with a return type Observer. It passes in two arguments of type String and ConverterPanel. It checks to see if the type is meters, feet, or non of the above. Depending on its type it returns a new Conversion Observer of the specified type using the ConverterPanel argument. Otherwise if its not of the specified types it returns null.

Value.java: is an interface used as a Subject in the Observer pattern that provides method signatures for the classes that implement it.

Value.java has no instance variables

· addObserver(Observer): is a public method with a return type void and passes in one argument of type Observer. This method acts as a signature for the classes that implement this interface.
· removeObserver(Observer): is a public method with a return type void and passes in one argument of type Observer. This method acts as a signature for the classes that implement this interface.
· setValue(double): is a public method with a return type void and passes in one argument of type double. This method acts as a signature for the classes that implement this interface.

ValueToConvert.java: is a class that store the centimeter value, acts as a "Receiver" in the Command pattern, acts as a "ConcreteSubject" in the Observer pattern, and implements the Singleton pattern. This class implements the Value interface.

· uniqueInstance: is a private static variable of type ValueToConvert that holds the value for the conversion if it is a unique instance.
· observers: is a protected variable of type List that holds Observer objects in an array list.
· value: is a private variable of type double that holds the value of the Observer.
· ValueToConvert(): is a private no-argument constructor that sets the value of "value" to zero.
· getInstance(): is a public static method with return type ValueToConvert. It has no arguments and checks to see if the variable uniqueInstance is null. If it is, it sets it to a new ValueToConvert. It then returns the variable uniqueInstance.

- addObserver(Observer): is a public synchronized method with a return type of void. It passes in one argument of type Observer. It adds the passed observer to the instance variable list of observers.
- removeObserver(Observer): is a public synchronized method with a return type of void. It passes in one argument of type Observer. It removes the passed observer from the instance variable list of observers.
- setValue(double): is a public method with return type void and passes on argument of type double. The method sets the instance variable value to the passed double and updates the values in the list of observers with the specified value.

ActionListenerCommand.java: is an interface implemented by ActionUpdate.java. It is used as a part of the "Command" pattern and interface. It serves as a signature for classes implementing it.
ActionListenerCommand.java has no instance variables

- execute(double): is a public method with return type void and passes one argument of type double. This method serves as a signature.

ActionUpdate.java: is a class that is used as a ConcreteCommand in the "Command" pattern. It implements the ActionListenerCommand class.

- converter: is a private variable with a return type of ValueToConvert. It holds the ValueToConvert that is being acted on.
- ActionUpdate(ValueToConvert): is a public one-argument constructor. It sets the converter to be the argument being passed in.
- execute(double): is a public method with a return type of void. It passes in one argument of type double and sets the converter value to be the double passed in.

MenubarListener.java: is a class that acts as a listener for when the menu bar is clicked. Regarding the "Command" pattern and the "Observer" pattern this class acts as the "Client". This class also acts as the "FactoryPatternDemo" in the "Factory" pattern.

- converter: is a private final variable with type ValueToConvert. It holds the ValueToConvert that is being acted on.
- panel: is a private final variable with type ConverterPanel. It holds the ConverterPanel that is being acted on.
- MenubarListener(ConverterPanel): is a public one-argument constructor that sets the instance variable panel to the argument passed in. It initializes the converter to a new instance and adds the feet and meter Observers
- actionPerformed(ActionEvent): is a public method with a return type void. It passes in one argument of type ActionEvent. It initializes a double to the panels value, an ActionListenerCommand using the variable converter and a MenuOptions using the ActionListenerCommand. Finally, it updates the menu with the double initialized.

MenuOptions.java: is a class that acts as the "Invoker" in the Command Pattern.

- update: is a package-private variable of type ActionListenerCommand and holds data for updates.
- MenuOptions(ActionListener): is a package-private one-argument constructor. The argument is of type ActionListenerCommand and sets the variable update to its value.
- update(double): is a public method with a return type void. It passes on argument of type double and executes the update using the given double.

ConverterApp.java: is a class that is used to hold the main method.
ConverterApp.java has no instance variables

- main(String[]): is the main method used when running the associated code.

Tools/Libraries specifics used during implementation

Eclipse version “2020-12”

Github

Part IV: Conclusion

What went well in the software project?

This project overall was fairly smooth sailing for our group as we’re significantly more comfortable using the software development life cycle and design patterns due to having prior experience now thanks to our previous projects. The creation of a good, well thought out UML diagram significantly helped our project as we knew where to put every function during development, lowering our time to develop significantly. Github proved to be a useful tool during this project for collaboration purposes as we could upload versions of our work for others to use easily. Our group collaborated well and the overall communication within our group was excellent.

What went wrong in the software project?

While overall this was a smooth experience for us, we did face a few trials and challenges along the way. Unlike our previous lab, during which mostly if not all of our problems arose in the implementation stage, this time around they were distributed evenly between the design and implementation stages.We tried to be as detailed and thorough as possible during the design stage, especially when it came to our UML diagram, and this took some time, as well as many re readings of the requirements. We had some disagreements on where things should be placed but we worked through those as a group and all agreed on the final implementation at the end of it. During the implementation we have some difficulties with the MenuBarListener functions as well as the implementation of the factory design pattern, but as a group we managed to overcome these significantly easier than the last project.

What have you learned from the software project?

This software project lab helped us better our understanding of the software development life cycle in several ways. While we didn’t initially realize it, it helped our requirement analysis comprehension skills as at first we had different views on what they required from us but we managed to all agree on what we thought was the “correct” approach at the end of the design stage. The design stage was one we really learned a lot in because, since we found it very useful in the implementation stage of the last project, we put more importance into it this time around and made sure we did our best with it. Again this stage proved very useful and eased our headaches. Furthermore during the implementation stage we learned the designed patterns used significantly more in depth than before.

What are the advantages and drawbacks of completing the lab in a group?

Group work is a great tool for learning how to work collaboratively as a team. This was a major advantage during this software project as it allowed us to compartmentalize and work on parts of the project individually, before combining them all to create the full project. This not only eased the workload on each individual greatly; but also allowed us to seek out each other for help when confused on how to proceed in our work. Another advantage was having several perspectives on the different approaches we could take and which one we should take, as well as how certain methods should be implemented; which both proved critical to our completion of this software project. However, as with all team work activities, some problems were encountered as well. While our group worked well together and everyone contributed fairly, one problem we consistently ran into was figuring out timing for everyone to have a team meeting. As all of our team members lead busy lives outside of school, it was often difficult finding times for us to meet and collaborate all at once. Another drawback we faced was that while group work provided us with several perspectives as mentioned earlier, we also had to find a middle ground between our ideas to implement, as everyone had a fair say. This proved a little challenging in some aspects but we managed to persevere and work well together.

Add a paragraph to indicate the different tasks of the work that were assigned to each group member, and the portion of the work completed by each group member. Also indicate if each group member was collaborative.

We worked very well together on this project and everyone collaborated equally. As mentioned before we compartmentalized this project and everyone involved in our project went above and beyond to make sure their portions of the work were done to the best of their ability, making changes if group members thought something should be done differently. The requirement analysis step was performed by all group members together. The Design process was performed mainly by Jasleen Chagger and Farnad Kazem-Zadeh however all group members were consulted during this process several times. The implementation portion was handled by Bryce Cooke and Ilan Zar, asking the other members for help when required or stuck. Finally the report was a group effort, everyone writing their own sections they worked on. Overall we would say everyone participated equally in this project and did the work they were assigned fairly.