EECS 3311 Lab 6 Report

Andrew Hocking, 215752835
Adrian Koduah, 216793887
Yun Lin, 214563449
Saad Shahid, 216178477

Part 1: Introduction

This software project is an application that behaves like a unit converter. It converts a given value of one unit to other units, specifically it converts centimeters to meters and feet. The goal of this project is to design and implement the Converter application using the principles and patterns of Object Oriented Design (OOD).

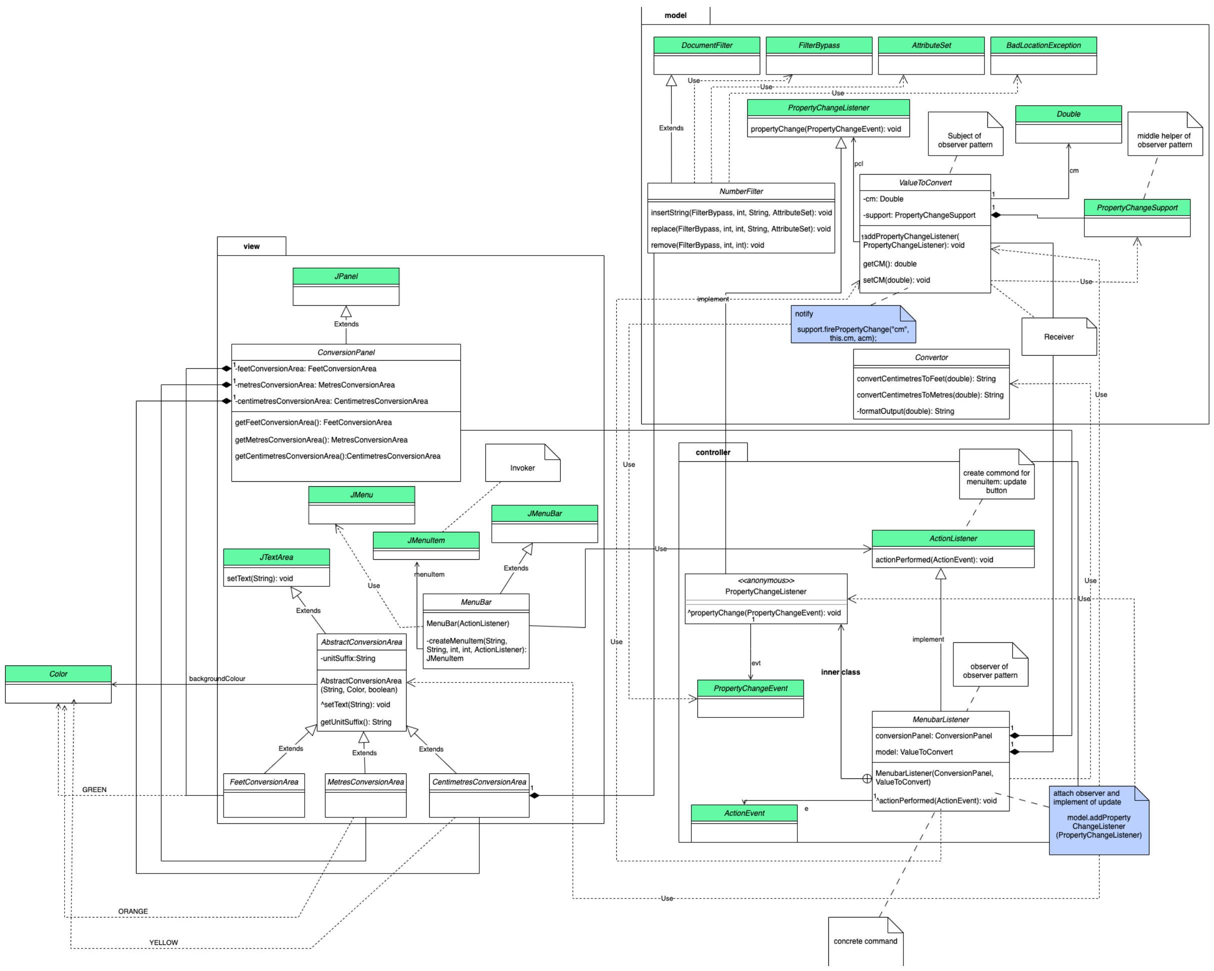
The primary challenge for this project was to fully implement the application using the Model/View/Controller design pattern from scratch, where no partially-implemented code was given. In previous projects, at least some code was already implemented, and we simply needed to build on top of it. Starting from scratch in the project meant we needed to apply the knowledge we learned about design in order to come up with a working solution that we could translate into code.

This project uses all four OOD Principles: *Inheritance, Polymorphism, Encapsulation,* and *Abstraction.* Inheritance is the process where a class can inherit methods and attributes of another class through the relationship of parent class and child class. Polymorphism occurs when methods in a child class override the functionality of methods in the parent class. Encapsulation is the process of privatizing certain methods/attributes that are specific to a class and don't pertain to any external classes, preventing direct access to them from the external classes. Abstraction is the overarching process of hiding internal information from other components that are not important to them, only showing what is important to them.

This report will be structured in the following order: 1. Introduction, 2. Design of the Solution, 3. Implementation of the Solution, and 4. Conclusion.

Part 2: Design of the Solution

Class diagram:


model
DocumentFilter
FilterBypass
AttributeSet
BadLocationException
Use
Use
Use
Extends
PropertyChangeListener
propertyChange(PropertyChangeEvent): void
Subject of observer pattern
Double
middle helper of observer pattern
pcl
cm
NumberFilter
insertString(FilterBypass, int, String, AttributeSet): void
replace(FilterBypass, int, int, String, AttributeSet): void
remove(FilterBypass, int, int): void
ValueToConvert
-cm: Double
-support: PropertyChangeSupport
addPropertyChangeListener(PropertyChangeListener): void
getCM(): double
setCM(double): void
PropertyChangeSupport
1
1
1
Use
implement
notify
support.firePropertyChange("cm", this.cm, acm);
Receiver
Convertor
convertCentimetresToFeet(double): String
convertCentimetresToMetres(double): String
-formatOutput(double): String
Use
view
JPanel
Extends
ConversionPanel
1 -feetConversionArea: FeetConversionArea
1 -metresConversionArea: MetresConversionArea
1 -centimetresConversionArea: CentimetresConversionArea
getFeetConversionArea(): FeetConversionArea
getMetresConversionArea(): MetresConversionArea
getCentimetresConversionArea():CentimetresConversionArea
Invoker
JMenu
JMenuItem
JMenuBar
JTextArea
setText(String): void
Use
menuItem
Extends
Extends
MenuBar
MenuBar(ActionListener)
-createMenuItem(String, String, int, int, ActionListener): JMenuItem
AbstractConversionArea
-unitSuffix:String
AbstractConversionArea (String, Color, boolean)
^setText(String): void
getUnitSuffix(): String
Color
backgroundColour
Extends
Extends
Extends
FeetConversionArea
MetresConversionArea
CentimetresConversionArea
1
GREEN
ORANGE
YELLOW
Use
Use
controller
create commond for menuitem: update button
ActionListener
actionPerformed(ActionEvent): void
Use
<<anonymous>>
PropertyChangeListener
^propertyChange(PropertyChangeEvent): void
1
evt
PropertyChangeEvent
inner class
implement
observer of observer pattern
Use
Use
MenubarListener
conversionPanel: ConversionPanel
model: ValueToConvert
MenubarListener(ConversionPanel, ValueToConvert)
1 ^actionPerformed(ActionEvent): void
1
1
e
ActionEvent
attach observer and implement of update
model.addPropertyChangeListener (PropertyChangeListener)
Use
concrete command

Design Patterns Used

Command Pattern

Invoker class: MenuBar class a container with menuItem.
Receiver class: ValueToConvert class.
Command class: ActionListener Interface   it declares an interface that handles the execution of an operation through the *actionPerformed* method.
Concrete Command class: MenubarListener class   it implements *actionPerformed* by invoking the *setCM* method on ValueToConvert.

Observer Pattern

Concrete subject class: ValueToConvert class   the method *addPropertyChangeListener* to attach observer objects.

Concrete observer class: MenubarListener class an anonymous inner class implementation of PropertyChangeListener interface implements *propertyChange* method as update method of observer class.
Helper class: PropertyChangeSupport instance it helps send notifications to observers when a specified attribute of the subject class has changed.
The implementation of the notifying method of the subject class in the ValueToConvert class is named *setCM*. This method gets help from the helper class by calling the *firePropertyChange* method from the instance of PropertyChangeSupport.
At first, during the initialization step of the MenubarListener controller class, this step will call the subject instance's attach method (*addPropertyChangeListener*) and attach its instance as a listener to listener maps of PropertyChangeSupport.
Later when MenubarListener calls the *setCM* method, the PropertyChangeSupport will notify the observer objects by searching though its listener map and then pass a PropertyChangeEvent to the observer object. After that, the observer object changes the view containers based on the event.

Design Principles Used

Inheritance

Inheritance simplifies the coding process by allowing classes to inherit and access attributes and methods that were already created by parent-classes. That's why inheritance is apparent throughout the project. For example, the MenuBar class extends the JMenuBar class, inheriting attributes from the preexisting JMenuBar class. The ConversionPanel class extends the JPanel class, inheriting attributes from JPanel. CentimetresConversionArea, FeetConversionArea and MetreConversionArea classes are all subclasses of the superclass AbstractConversionArea abstract class.

Polymorphism

An example of polymorphism in this project can be seen in the AbstractConversionArea class, which extends the JTextArea class. AbstractConversionArea overrides JTextArea's *setText* method with its own version, which ensures that the text always has the unit string at the end and is never empty (if it is, it replaces it with "0"). Now, every time one of the children of AbstractConversionArea uses the *setText* method, it is using the AbstractConversionArea version instead of the JTextArea version.

Abstraction

The AbstractConversionArea abstract class defines the structure of a ConversionArea object, which is a View component that visually constructs a square on the panel. The specific details for how the square is visually constructed is contained within the AbstractConversionArea class, and is not necessary information for its subclasses (CentimetresConversionArea, FeetConversionArea and MetreConversionArea). The subclasses only require basic information such as the unit the square represents, the colour of the square, and whether or not the user can edit it, all of which are all passed as parameters to the superclass' constructor method and they are the variables unitSuffix, backgroundColour and isEditable respectively. By this means, the subclasses are reduced down to the abstract concept of a square view because only the necessary characteristics of the superclass are shown to the subclasses.

Encapsulation

There are multiple examples of encapsulation in this project. In the MenuBar class, for example, the helper method *createMenuItem* is private. Since this method is only needed to create menu bar items, it is not needed anywhere else in the code. So, it makes logical sense to keep this method private. Encapsulation can also be seen in the AbstractConversionArea class. Here, the unitSuffix attribute is private. It can be read, using the public *getUnitSuffix* method, but it cannot be set from anywhere outside the class, including its subclasses. This ensures that the unit suffix is set in stone and cannot be changed after the fact, which otherwise might lead to confusion should it be changed.

Part 3: Implementation of the Solution

The tools used to complete the coding portion of this software project were the following:

- OS: Windows 10
  Eclipse Build ID: 20210910-1417
  JRE: org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_16.0.2.v20210721-1149
- OS: macOS 11.2
  Eclipse Build ID: 20211212-1212
  JRE: Java SE 8 [1.8.0_171]

Click here to see the Javadoc for a detailed implementation explanation.
Click here to see a video demonstration of how to launch and use the application.

These links are also available in the README.md file on the project's GitHub repository, which can be accessed by clicking here.

Part 4: Conclusion

Overall the implementation of classes within the MVC structure went well. The classes ended up working the way they should, each contributing to their respective roles of the Model, View and Controller. Although achieving the end result was somewhat challenging. Unlike the previous software projects, this was the first one where we had to start the implementation from scratch. The challenge is that there is more room for errors. For example, getting the View to match exactly with the image provided took longer time than anticipated because there were no dimensions given. Additionally, external research was necessary to get the layout working using Swing. Starting the project from scratch can also be seen as an advantage because it was a perfect hands-on approach for practicing and strengthening our understanding of Object Oriented design principles and patterns.

Working together as a group for this project was beneficial in many ways. Within the scope of the course, group work made the completion of the project less stressful because we were able to consolidate together in the design and implementation process by sharing our ideas and knowledge with each other. Outside the scope of the course, group work gave us exposure to a realistic workplace environment. We got experience in collaborating with unknown members, each with a different set of skills to achieve a common goal. Just like the previous projects, we have learned that following Object Oriented design principles and patterns makes the completion of a software project more simple, no matter how complex the project is. Starting with the design phase by planning out the structure and the goals, and implementing the code modularly, which in turn simplifies retesting and refactoring code.

Disadvantage to working in groups is assigning work to group members through online communication. In theory, communication via online is readily accessible and simple for collaborating. On the contrary it can be challenging when everyone has different schedules and availability. It made it difficult to set small deadlines for different parts of the project, so instead the project was passively worked on during the time given.

Three ways that would ease the completion of the software project are first communicating with members as soon as possible. Delegating tasks to group members earlier gives more time for completing the tasks, and gives more room for any unexpected delays. Second would be to ensure that everyone has a good understanding of the requirements of the project. Clearing out any misunderstanding before makes it simple. Finally to keep everything organized and work faster, we should adopt the SCRUM model taught in class for simplifying the process, and sticking to it. Having a team leader that checks up on everyone is essential to keep the group together so that there is a mutual understanding between everyone and eases the workflow. Assigning smaller tasks to members with smaller deadlines ensures that everything gets completed in a timely manner.

Tasks Completed			
Andrew Hocking	**Adrian Koduah**	**Yun Lin**	**Saad Shahid**
Implemented app view		Implemented app controller	Report: Introduction
Assisted development of app model		Lead development of app model	Report: Design Principles (Inheritance, Abstraction)
Wrote all Javadoc comments and hosted Javadoc on website		Report: Design Patterns	Report: Conclusion
Report: Design Principles (Polymorphism, Encapsulation)		Video demo	
Created GitHub repo and Discord server for collaboration		UML Diagram	
Created interactive To-Do list on GitHub			