EECS 3311 lab 06
Converter App

Name: Ozan Isik
Github username: ozan98
Student Id: 215001746

Name: Nofal Rizwan
Github username: NofalRiz26
Student ID:213827720

Name: Enes Bahadir Is
Github username: eJustIS-1
Student ID: 216164337

Name: Ugur Can Avcu
Github username: ugurcanavcu
Student ID: 215814239

Professor Alvine Boayle Belle

Introduction

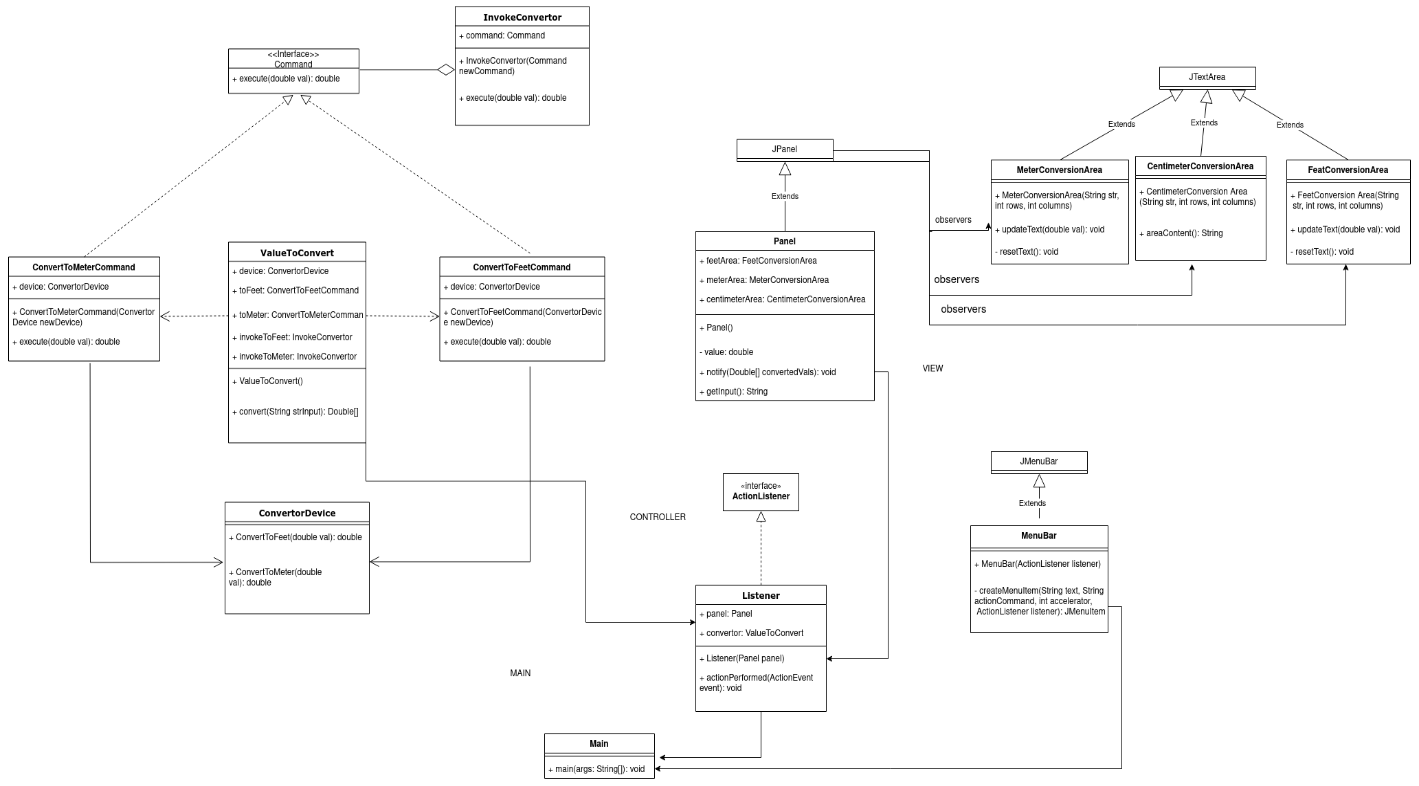
The software project is “Converter Project” which aims to convert the entered value (in cm) and display the entered value in feet and meters. The converter project is written in Java 11 using Java Swing which is a GUI framework for Java. The interface of the game consists of 3 parts. Yellow area is where the user enters the value and rest of the parts are used to display the entered value’s converted versions such as feet and meters. The interface also includes 1 menu which has a button inside it. That button can also be triggered by pressing the “F” key from the local keyboard. To use the project what the user needs to do is to click on the “Update Model” menu and then “Save input centimeters”. What the project does is first stores the entered value, converts it to feet and meter and then displays on green and orange boxes.

The challenge we have faced during the creation of this project because integrating the observer and command design patterns into this project was a little bit difficult for us. Because it was our first time to create a UML and make a plan of the project with observer and command design patterns, we could not figure out how to connect classes. A significant amount of time was spent on this part of the project.

In order to make this project, we have utilized some of the Object Oriented programming concepts, principles, and design patterns. For object oriented programming concepts and principles, polymorphism, inheritance, abstraction and encapsulation design principles are used. Regarding the design patterns we have used, observer, command, and MVC (Model-view-controller) design patterns are used.

The report of the project will comprise four parts which are introduction, design, implementation, and conclusion, respectively.

Design



InvokeConvertor
+ command: Command
+ InvokeConvertor(Command newCommand)
+ execute(double val): double
<<Interface>> Command
+ execute(double val): double
ConvertToMeterCommand
+ device: ConvertorDevice
+ ConvertToMeterCommand(Convertor Device newDevice)
+ execute(double val): double
ValueToConvert
+ device: ConvertorDevice
+ toFeet: ConvertToFeetCommand
+ toMeter: ConvertToMeterComman
+ invokeToFeet: InvokeConvertor
+ invokeToMeter: InvokeConvertor
+ ValueToConvert()
+ convert(String strInput): Double[]
ConvertToFeetCommand
+ device: ConvertorDevice
+ ConvertToFeetCommand(ConvertorDevic e newDevice)
+ execute(double val): double
ConvertorDevice
+ ConvertToFeet(double val): double
+ ConvertToMeter(double val): double
JTextArea
Extends
Extends
Extends
JPanel
Extends
MeterConversionArea
+ MeterConversionArea(String str, int rows, int columns)
+ updateText(double val): void
- resetText(): void
CentimeterConversionArea
+ CentimeterConversion Area (String str, int rows, int columns)
+ areaContent(): String
FeatConversionArea
+ FeetConversion Area(String str, int rows, int columns)
+ updateText(double val): void
- resetText(): void
observers
observers
observers
Panel
+ feetArea: FeetConversionArea
+ meterArea: MeterConversionArea
+ centimeterArea: CentimeterConversionArea
+ Panel()
- value: double
+ notify(Double[] convertedVals): void
+ getInput(): String
VIEW
CONTROLLER
«interface» ActionListener
Listener
+ panel: Panel
+ convertor: ValueToConvert
+ Listener(Panel panel)
+ actionPerformed(ActionEvent event): void
JMenuBar
Extends
MenuBar
+ MenuBar(ActionListener listener)
- createMenuItem(String text, String actionCommand, int accelerator, ActionListener listener): JMenuItem
MAIN
Main
+ main(args: String[]): void

The design patterns that were used for this design are observer and command alongside MVC architecture. Observer design pattern was used to implement the view module of the program. This makes sense because observers change their states once notified by the subject. In this program's case that state is the contents of the areas on the interface. For the observer design pattern the subject is the JPanel and the concrete subject is the Panel. JPanel is the parent of Panel and not the interface because we require the JPanel to create the swing component JPanel. Panel adds observers through the constructor therefore no addObserver method is needed. Panel has the method notify() that broadcasts to all its observers to update their states. This is done by calling the updataText() method of the observers. CentimeterConversionArea, MeterConversionArea, and FeetConversionArea are the concrete observers that inherit from the JTextarea which is the observer. Each observer contains the method called updateText(). This method updates the content of each respective area with the correct converted value. Finally, the client of the pattern is the controller Listener .Command design pattern was used to implement the model module of the program. For the command design pattern the receiver is the ConvertorDevice. ConverterDevice encapsulates all the commands that can be executed.

These are converToFeet() and convertToMeter(). Command is the command of the pattern. Command is the interface that implements all commands. It holds the generic method called execute(). ConvertToFeetCommand and ConvertToMeterCommand are the concrete commands. These implement command interfaces. For these concrete commands, they require the receiver to execute on. Receiver is provided in the constructor. InvokeConvertor is the invoker. This class asks the command to perform the request. It does this by requiring a command which is provided in the constructor. Finally the client is ValueToConvert. ValueToConvert encapsulates the value to be converted and instantiates a respected command and the receiver according to the action that is wanted to take by the controller which in this program's case is to convert.

The oo design principles that were used are inheritance, encapsulation and interfaces. Inheritance is used heavily because both observer and command design pattern require it. For observer inheritance is used between JTextArea and the concrete observers. The method that is mainly used is the update. Inheritance is also used between JPanal and Panel. For the command design pattern, inheritance is used between Command and ConvertToFeetCommand and ConvertToMeterCommand. The main method inherited is execute(). Encapsulation is used in ValueToConvert to encapsulate the input from the user. Encapsulation is also used in ConvertorDevice to store all the commands that are needed like convertToFeer() and converToMeter().

Implementation

Listener:

This is the controller class. This class is also the event handler for the menubar. It handles the event by implementing ActionListener. With ActionListener it is able to listen for the event that occur in the menubar. It is able to do this by picking up events that occur in the actionPerformed. This method then handles the event by calling the model to convert the value and notifies the observers to change their states.

Command:

Interface that represent all command objects. This class declares an interface that handles the execution of an operation through the execute method. The execute method is defined to return a double value that receives an input value. This input value is the input from the user that is asked to be converted in the respected unit.

ConvertorDevice:

Receiver class that encapsulates all the commands of a convertor device. The purpose of this class is to perform the action associated with the request. This class defines 2 commands that are performed when the execute method is run by the invoker. One method is the command to convert to Feet and the other is to convert to Meter. They both take an input value that is provided by the user to be converted to the associated unit.

ConvertToFeetCommand:

This is a concrete command class that executes converting input value to feet command. This class implements execute by invoking the corresponding operation on the receiver. This class needs a receiver before it can make an execution. That is why in the constructor it takes a receiver as a parameter. This is the receiver to execute. In the execute method, it calls the command that is associated with the unit to be converted to. In this class's case that is converting to feet.

ConvertToMeterCommand:

This is a concrete command class that executes converting input value to meter command. This class implements execute by invoking the corresponding operation on the receiver. This class need a receiver before it can make an execution. That is why in the constructor it takes a receiver as a parameter. This is the receiver to make execution to. In the execute method, it calls the command that is associated with the unit to be converted to. In this class's case that is converting to meter.

InvokeConvertor:

The invoker class of the command design pattern. This class asks the command to perform the request that is made by the client. The request is made by first receiving the command. This is done in the constructor. The constructor takes a command as a parameter and this command is to be executed. The command is executed with the convert method. This method is implemented by calling the execute method in the command class that is associated with.

ValueToConvert:

A model class that represents the application to the command design pattern it encapsulates the value to be converted to feet and meter. This class purpose is to encapsulate input value by the user and request to make a convert command to the receiver. Before encapsulating the input value through the convert method, there is a precondition. The precondition is that the input value must be a number representation. This means that the string input that this class receives from the controller must not contain any letters. If the input string contains letters, then and IllegalArgumentExeption is thrown. After the conversion is made the convert method returns an array of size 2. [0] contains the converted to feet value and [1] contains the converted to meters value. This is sent off to the controller after it is called by it.

CentimeterConversionArea:

A view class that represents a JTextarea of the centimeter area. CentimeterConversionArea is the observer of the observer design pattern. CentimeterConversionArea observes the concrete subject class which is Panel. CentimeterConversionArea does not update its state because it does not need to. The function of CentimeterConversionArea is to take input from the user and pass it to the concrete subject so it can then be passed to the model to be converted.

FeetConversionArea:

A view class that represents a JTextarea of the feet area. FeetConversionArea is the observer of the observer design pattern. FeetConversionArea observes the concrete subject class which is Panel. Once the concrete subject has made to operate on notify, FeetConversionArea updates its content with the provided converted value. In this class case it is to update content with the converted Feet value.

MenuBar:

a view class that represent a JMenuBar. The purpose of this Class is to initialize and display the menu bar of the program. It defines a JMenu and adds it to the JMenuBar. It creates the JMenu with one menu and menu item. The menu item is created with the helper method to initialize the parameter of it. It then sets the action command and adds the action listener and as well as adds a keyEvent to check pressed keys.

MeterConversionArea:

A view class that represents a JTextarea of the meter area. MeterConversionArea is the observer of the observer design pattern. MeterConversionArea observes the concrete subject class which is Panel. Once the concrete subject has made to operate on notify, MeterConversionArea updates its content with the provided converted value. In this class case it is to update content with the converted meter value.

Panel:

a view class Panel that represents a JPanel. Panal is the concrete subject class of JPanal which is the subject class. Panel knows its observers which are FeetConversionArea, MeterConversionArea, and CentimeterConversionArea. Panel proposes operation to notify each observer. What the notify operation does is that it broadcasts to all the observers to update their states. This update is displaying the converted values.

The tools that were used in implementation was vsCode ide.

Conclusion

Regarding what went into the creation of this project, the implementation part was mainly easy for our group after we figured out the planning and creating the UML part. As we have mentioned in the introduction part, figuring out how to use observer and command design patterns was our main concern and the most time consuming part. For what went wrong, deciding on and planning what we needed to do with these design patterns took an extreme amount of time and it created a time trouble for us. Also, it affected our availability since the last days of these projects became so stressful. Therefore, we had to be fast and productive enough in a short amount of time. This was also our challenge besides the planning part.

We have learned the importance of analyzing and understanding the project. From our experience, we can easily say that analyzing, understanding and planning the project consumes the most amount of time to make a successful project. Therefore, this project showed the importance of these again. Additionally, it was our first time to use observer and command design patterns. It was a great practice to understand and implement these patterns on a project.

The main advantage of working in a group is to have a chance to discuss ideas because everyone has his/her own idea and considering these different ideas opens all of the group members' minds and creates a different perspective for us. Also, it gives us the opportunity to have an example of teamwork which will be helpful in future when we start working in the software development industry. The main disadvantage of group work is to align the work with your timetable. Because everyone has different schedules and availability, there is always a confusion and discussion about finding a time frame for all of us.

Even though everyone was involved in the project, each group member had an assigned part of the project that needs to be done. Enes worked on the report, specifically the introduction and conclusion parts, whereas Ozan especially worked on the implementation part of the project. Ugur and Nofal were assigned to get UML diagram done. Therefore, each group member was collaborative. Also, if someone finished the assigned work earlier, that person also helped others to finish the project as soon as possible.