EECS3311 Lab 6 Report

Hui Xu - 215637226 - Section A - TA: Kazi Mridul
Mashhood Anwar - 216370124 - Section B - TA: Naeiji Alireza
Wei Bin Zhang - 215678444 - Section B - TA: Naeiji Alireza
Ying Zhang – 217392986 - Section A - TA: Kazi Mridul

Part I: Introduction

- **The software project and its goal**:

  This software project is an application that displays an interface with three different colored textfields. When entering a number to the yellow view that represents the value in centimeters, after clicking the save input centimeters button on the update model menu bar, the centimeter value will convert to feet and meter values and show on the green and orange views accordingly. The goals of this project are using the MVC model and implementing Observer and Command patterns to complete it.

- **Challenges**:

  We started this software project from scratch. The first challenge is to build the overall structure of this project and make sure we use the MVC model correctly and places all the classes accordingly. The second challenge is to understand the Observer pattern and implement it. The third one is to understand the command pattern and implement it. Another challenge is to make the teamwork consistent so that every group member understands what we are doing and keeps the project style uniform.

- **OOD, OOD principles, design patterns**

  OOD specifies software objects and the way they collaborate to satisfy. It defines objects states, behaviors and the way these objects interact to achieve the desired requirements.

  OOD main design principles include Abstraction, Encapsulation, Polymorphism, and Inheritance. Abstraction implies hiding the unnecessary details of a class to other classes, only exposing methods that are relevant to interact with other classes. In our project, Command, Observer, and Subject interface are used to achieve abstraction. Encapsulation implies keeping the logic (state and methods) of an object inside a class. Getter methods and Setter methods are used in most classes. Inheritance implies making the child class reuse the parent state and methods without changing them. ConverterPanel class is a subclass of JPanel, ConverterMenuBar is a subclass of JMenuBar. Polymorphism implies the ability of a method to take different shapes. Polymorphism allows out execution process can be applied to read and write commands.

  Design patterns:

  - **Observer pattern** is a software design pattern in which an object, named the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes. To implement observer pattern, we have Observer interface, and CentimetersConversionArea, FeetConversionArea, MeterConversionArea classes as concreteObserver classes that implements Observer. We have the Subject interface, and ValueToConvert class as concreteObserverable that implements Subject.

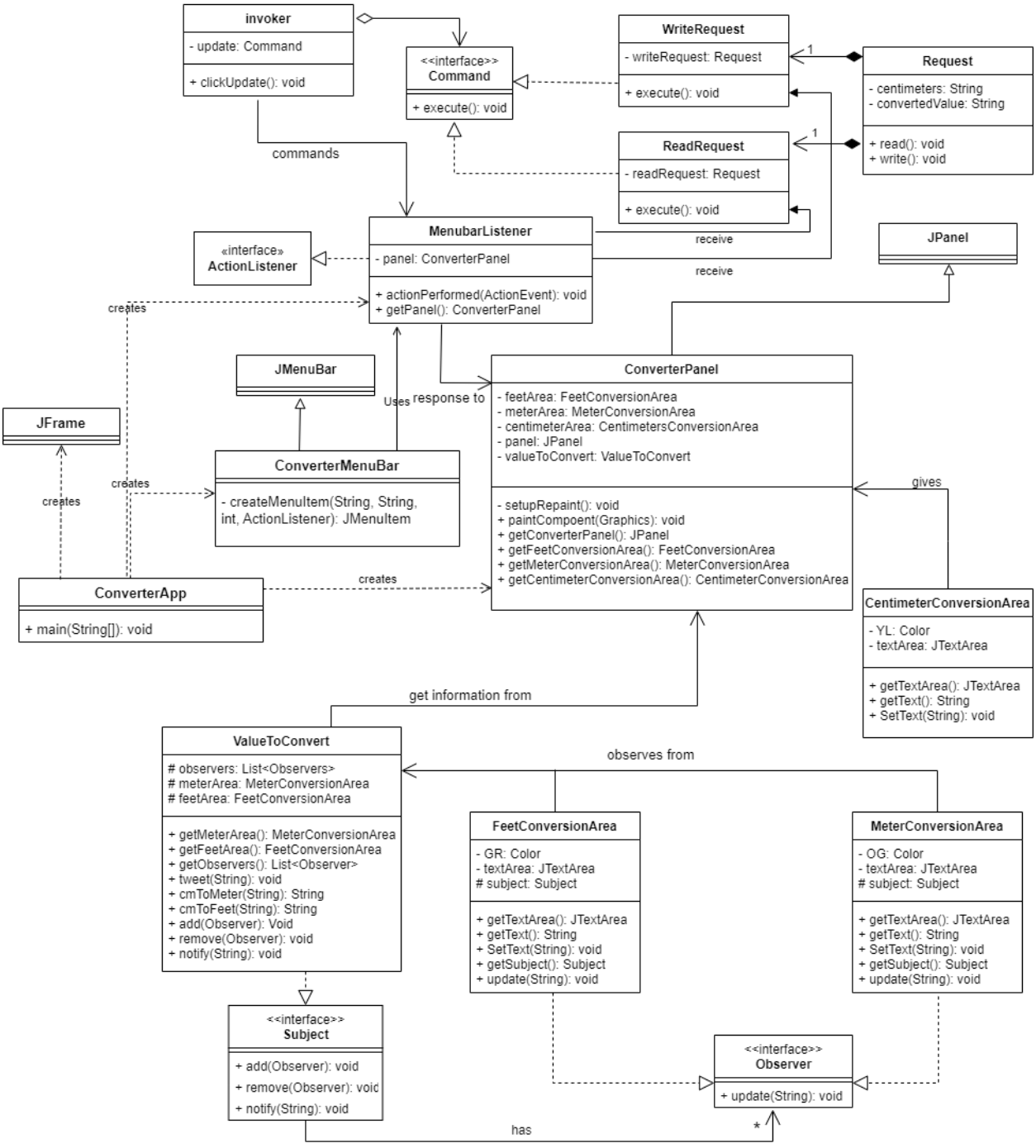
- **Command pattern** is a behavioral design pattern in which an object is used to encapsulate all information needed to perform an action or trigger an event at a later time. To implement command pattern, we have Command interface, Invoker class that set the command, ReadRequest and WriteRequest classes as concreteCommand that implement Command interface. We also have a Request class as the Receiver that does the actions read and write.
-

- **How we structure the report**

  There are four parts of this project report: introduction, design of the solution, implementation of the solution, and conclusion. The report will be structured based on the requirement questions from the lab slides and we will answer and explain them one by one.

Part II: Design of the solution

- **UML diagram**



- **UML diagram comments** (descriptions for all the classes)

    - **Classes for controller package**:
        - **<interface>Command**: An interface that only contains execute() method
        - **Invoker**: takes an command and make it executable from menu bar
        - **MenubarListener**: takes action from the converterPanel, response and execute corresponding command base on the input
        - **Request**: takes centimeter and convert, has two method: read() and write()

        - **ReadRequest**: implements Command, and contains Request, making itself a read command for values to be taken
        - **WriteRequest**: implements Command, and contains Request, making itself a write command for values to be written

    - **Classes for model package**:
        - **<interface>Observer**: An interface that only contains update() method that takes information from the subject
        - **<interface>Subject**: An interface that contains add(), remove() and notify() for all its observers
        - **ValueToConvert**: implements Subject, this class contains a list of the observers(class below), able to return any of its observers, convert the value provided and notify to each observer in the list. This class is also able to add or remove observers
        - **CentimeterConversionArea**: this class sets up the Area related to centimeter and provides its value to ConverterPanel Then the value is given to the subject class ValueToConvert
        - **FeetConversionArea**: implements Observer, this class sets up the Area related to centimeter and ready to update its value from the subject class ValueToConvert
        - **MeterConversionArea**: implements Observer, this class sets up the Area related to centimeter and ready to update its value from the subject class ValueToConvert

    - **Classes for view package**:
        - **ConverterMenuBar**: This class creates the menu bar for the menu bar listener and for the panel
        - **ConverterPanel**: This class creates the panel and all the updated corresponding areas in valueToConvert. It also holds the controller like the the read and write requests from the menu bar listener

    - **Class for main**:
        - ConverterApp: This will be the main method for this lab, it uses The converterPanel, converterManuBar and JFrame for to output result and take in value, it is basically the main view and control of the program

- **How we used design patterns** (name the corresponding classes, interfaces, and most relevant operations)

    As stated above, this lab will be using two design pattern (Command pattern and Observer pattern), and they will be the main focus for this ConverterApp program

    For command pattern, an interface called Command is used, this interface only contains the method execute(). Then two concrete commands called WriteRequest and ReadRequest will implement the Command interface that is responsible for writing and reading as a request. We then have an Invoker class that contains one of the concrete commands above, the only thing that Invoker

class has is an update method that runs the execute() method that is inside the command classes. And this all provides feedback or updates to MenuBarListener which responses to the converterPanel

For Observer pattern, two interfaces called Observer and Subject are used, Observer interface only contains an update() method that changes its value provided by Subject. Subject interface contains add() and remove() to manage its observers, and notify() method to provide values to all its observers. We have the two classes (FeetConversionArea, MeterConversionArea) that implement Observer. These classes are responsible for creating the corresponding area in the converter panel, but most importantly, their value is provided by the Subject, which is the ValueToConvert class that implements Subject. ValueToConvert class manage the list of the three observers above, and is able to notify() or provide information to all of its observers

- **OO design principles in the UML** (name the corresponding classes, interfaces, and the most relevant operations)
  ValueToConvert encapsulates the conversion behavior of FeetConversionArea and MeterConversionArea. It also implements the Subject interface. FeetConversionArea and MeterConversionArea implements the Observer interface. The WriteRequest class and the ReadRequest class implements the command interface which both classes have the same states but different behaviors. For example, both classes have a method called execute() but their jobs are different. In WriteRequest, execute() does the job for executing write request; in ReadRequest, execute() does the job for executing read request. Furthermore, MenubarListener implements ActionListener which allows MenubarListener to use the feature of ActionListener. We applied encapsulation, abstraction and polymorphism while we were implementing the observer pattern and command pattern.

Part III: Implementation of the solution

- **Coding:** see java implementation
- **Javadoc** (comments, invariants, preconditions, and postconditions): see doc folder
- **Tools using:**
  - Intellij: IDEA 2021.2 (Ultimate Edition)
  - Eclipse: 2021-09 (4.21.0), Build id: 20210910-1417. JRE System Library JavaSE-16 is used
  - UML: diagram.net
- **Short video:** on GitHub

Part IV: Conclusion

- **What went well:**
  Tasks are divided easily and quickly. Each member chose the part he/she wanted to work on. We coded very fast to complete the whole project and make it run. Report progress is quite influenced for us. Observer pattern and command pattern are also implemented very quickly.

- **What went wrong:**
    - We coded very quickly without implementing patterns at the very beginning. After pattern implementation, we think everything should be working as desired; however, the observer pattern doesn't do notify and update action. With continuous testing and debugging, we found out that the observers list should contain two objects that is actually empty in our notify method. The corresponding update method does not print the update message as expected. We found out that we didn't have a getter method for Subject attribute in observer classes and we didn't access the attribute correctly, then we fixed the update message. Later on, we found out that we didn't access the right observer list object in the notify method which leads to us always having an empty observer list. We fixed this problem by accessing the MeterConversionArea Subject so that it has two followers.

- **What we have learned:**
    - We learned how to add text field with colors in java. We learned how to use observer pattern and command pattern for this software project. We coded everything as soon as possible and made it run, then later on we practiced modifying the code with observer pattern and command pattern. We also practiced working as a team to do software practice.

- **Group work advantages and drawbacks:**
    - **Advantages to work in group**: group members can collaborate and divide work into pieces, each member will get fewer work tasks comparing to if it was an individual software project. Group members make a synergy to increase work efficiency. We learn more through group discussion since different people will have different problems and challenges.
    - **Drawbacks to work in group**: it's hard to divide works for individuals. The quantity of workload is hard to remain fair to each team member. Team collaboration takes more time than individual work since the project divides into pieces, team members need to discuss, explain and understand what other members did. It also takes time to wait for group members' responses.

- **Task table and collaboration**

Group Member	Tasks	Contribute	Collaboration
Hui Xu	Report - Part II UML drawing Report - Part II UML class commenting Report - Part II How you used design patterns	100%	Responsible & Collaborative

Mashhood Anwar	Coding - Command Pattern Report - Part III JavaDoc - model package Report - Part III JavaDoc - control package	100%	Responsible & Collaborative
Wei Bin Zhang	Coding - Observer Pattern Report - Part II OO design principles Report - Part III video and other stuff if needed	100%	Responsible & Collaborative
Ying Zhang	Coding - Starter code Report - Part I intro (the whole thing) Report - Part III JavaDoc - view package + main package Report - Part IV (the whole thing)	100%	Responsible & Collaborative