3311 Software Design - Lab 6

Amandeep Chhina, 210-866-192, Section A

Yazan Armoush, 217-188-921, Section A

Amir Mohamad, 217-387-762, Section B

Mohamed Meghawry, 212-861-217, Section A

November 12, 2021

I Introduction

I.I Explain What The Software Project About and What Are Its Goals

The purpose of this software project is to thoroughly comprehend the Software Design to Software Implementation process, including the various design patterns utilized such as Command and Observer pattern, the benefits and costs associated with each design pattern, and the ability to convert software description to software implementation. In particular the deliverable object of this project is an application that displays an interface with three panels. The yellow panel is where the user can enter a value in centimeters. The green and orange panel then display the corresponding centimeters entered as feet (green) and meters (yellow) after being converted. There are two separate ways to do conversions, one is to manually use the "Save the input parameters" menu bar item or the other to "Enable auto conversion" in which case the conversions happen immediately when a value is entered into the centimeters panel.

I.II Explain The Challenges Associated To The Software Project

Some of the challenges associated with this Software project was implementing the Observer design pattern. This was challenging because the update of the dependents made it so that the states had to be updated for all dependents, and as such they had to be notified and subscribed to these state change events. This was more challenging to implement then simply using a global variable that houses the state, and then each dependent would just read and update its own state as needed. However, the advantages of using the Observer pattern was that instead of events having to be polled at regular intervals they can be received and updated immediately upon a state change.

I.III Explain The Concepts (e.g., Object Oriented Design, Object Oriented Design Principles, Design Patterns) You Will Use to Carry Out The Software Project

Abstraction, *Encapsulation*, *Polymorphism*, and *Inheritance* are the four core design concepts of OOD. *Abstraction* refers to concealing a class's non-essential information from other classes and only exposing methods that are needed to communicate with other classes. This project's *AbstractJTextArea* class is an abstract class. *Encapsulation* refers to retaining an object's logic within a class. In most classes, getter and setter methods are utilised. Inheritance enables a child class to utilise the parent's state and methods without having to change them, and is used within *CentimeterTextArea* when it inherits from *AbstractJTextArea* for example. *Polymorphism* refers to a method's ability to take on several forms. In this project, *Polymorphism* allows *CentimeterTextArea*, *FeetTextArea*, and *MeterTextArea* to be added to a list of dependents as *JTextArea*, so we can abstract away the specifics of *CentimeterTextArea*, *FeetTextArea*, and *MeterTextArea* when we are handling *JTextArea* objects. We'll also use OOD ideas like the *Single Responsibility Principle* (SRP), which states that each class should only be responsible for one thing. Such as *MeasurementType* class which helps us organize the different type of measurements and provides the associated suffix.

We'll also leverage OOD Patterns like *Observer* pattern, in which we house all dependents subscribe them to updates and send update notifications whenever another dependents state changes. This will be used to produce the various commands, and it will be an excellent method to isolate the commands from the class's fundamental responsibilities when handling the different commands. Another OOD Pattern will be the *Command* Pattern which we use with *Command*, *SaveCommand*, and *EnableAutoCommand* classes to implement. This allows us to decouple the specifics of the command we will use with the particular Save and Auto Enable command we will use to house the various commands.

I.IV Explain How You Are Going To Structure You Report Accordingly

As a result, the report will be prepared to emphasise the above-mentioned OOD Principles and Patterns. In addition, we will describe the rationale for selecting such patterns, as well as any alternatives to the patterns we select when we believe they are appropriate. In addition to the framework, there are four elements to this project report: introduction, solution design, solution implementation, and conclusion. The report will be organised around the lab requirements prerequisite questions stated in the corresponding slides, which we will answer and explain in order.

II Design of The Solution

II.I UML Class Diagram of Your System

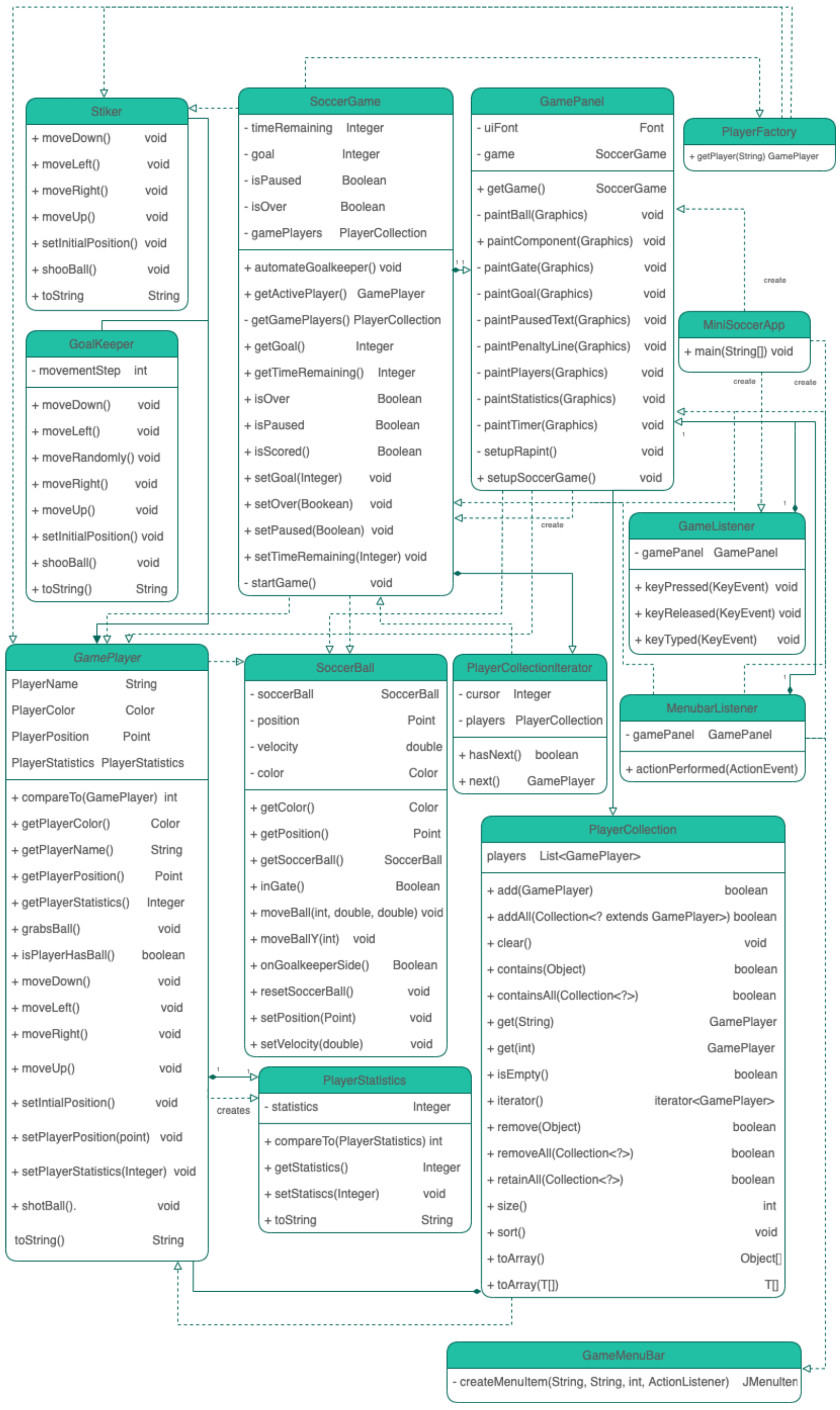
Stiker
+ moveDown() void
+ moveLeft() void
+ moveRight() void
+ moveUp() void
+ setInitialPosition() void
+ shooBall() void
+ toString String
GoalKeeper
- movementStep int
+ moveDown() void
+ moveLeft() void
+ moveRandomly() void
+ moveRight() void
+ moveUp() void
+ setInitialPosition() void
+ shooBall() void
+ toString() String
SoccerGame
- timeRemaining Integer
- goal Integer
- isPaused Boolean
- isOver Boolean
- gamePlayers PlayerCollection
+ automateGoalkeeper() void
+ getActivePlayer() GamePlayer
- getGamePlayers() PlayerCollection
+ getGoal() Integer
+ getTimeRemaining() Integer
+ isOver Boolean
+ isPaused Boolean
+ isScored() Boolean
+ setGoal(Integer) void
+ setOver(Bookean) void
+ setPaused(Boolean) void
+ setTimeRemaining(Integer) void
- startGame() void
GamePanel
- uiFont Font
- game SoccerGame
+ getGame() SoccerGame
- paintBall(Graphics) void
+ paintComponent(Graphics) void
- paintGate(Graphics) void
- paintGoal(Graphics) void
- paintPausedText(Graphics) void
- paintPenaltyLine(Graphics) void
- paintPlayers(Graphics) void
- paintStatistics(Graphics) void
- paintTimer(Graphics) void
- setupRapint() void
+ setupSoccerGame() void
PlayerFactory
+ getPlayer(String) GamePlayer
create
MiniSoccerApp
+ main(String[]) void
create
create
create
GameListener
- gamePanel GamePanel
+ keyPressed(KeyEvent) void
+ keyReleased(KeyEvent) void
+ keyTyped(KeyEvent) void
GamePlayer
PlayerName String
PlayerColor Color
PlayerPosition Point
PlayerStatistics PlayerStatistics
+ compareTo(GamePlayer) int
+ getPlayerColor() Color
+ getPlayerName() String
+ getPlayerPosition() Point
+ getPlayerStatistics() Integer
+ grabsBall() void
+ isPlayerHasBall() boolean
+ moveDown() void
+ moveLeft() void
+ moveRight() void
+ moveUp() void
+ setIntialPosition() void
+ setPlayerPosition(point) void
+ setPlayerStatistics(Integer) void
+ shotBall(). void
toString() String
SoccerBall
- soccerBall SoccerBall
- position Point
- velocity double
- color Color
+ getColor() Color
+ getPosition() Point
+ getSoccerBall() SoccerBall
+ inGate() Boolean
+ moveBall(int, double, double) void
+ moveBallY(int) void
+ onGoalkeeperSide() Boolean
+ resetSoccerBall() void
+ setPosition(Point) void
+ setVelocity(double) void
PlayerCollectionIterator
- cursor Integer
- players PlayerCollection
+ hasNext() boolean
+ next() GamePlayer
MenubarListener
- gamePanel GamePanel
+ actionPerformed(ActionEvent)
PlayerStatistics
- statistics Integer
+ compareTo(PlayerStatistics) int
+ getStatistics() Integer
+ setStatiscs(Integer) void
+ toString String
creates
PlayerCollection
players List<GamePlayer>
+ add(GamePlayer) boolean
+ addAll(Collection&lt;? extends GamePlayer&gt;) boolean
+ clear() void
+ contains(Object) boolean
+ containsAll(Collection&lt;?&gt;) boolean
+ get(String) GamePlayer
+ get(int) GamePlayer
+ isEmpty() boolean
+ iterator() iterator<GamePlayer>
+ remove(Object) boolean
+ removeAll(Collection&lt;?&gt;) boolean
+ retainAll(Collection&lt;?&gt;) boolean
+ size() int
+ sort() void
+ toArray() Object[]
+ toArray(T[]) T[]
GameMenuBar
- createMenuItem(String, String, int, ActionListener) JMenuItem

Figure 1: UML Diagram

II.I.1 *Controller*

The *Controller* class is the intermediary between the display and the logic used for the various conversions. As such it has the associated methods *updateCentimeterText()*, *updateFeetText()*, and *updateMeterText()* which allows the model to speak to the display and update the text area for the separate conversions.

II.I.2 *MenubarListner*

The *MenubarListener* class implements *ActionListener*. It has an attribute *commands* which holds all the valid commands that can be run. When an action event is triggered, it can do the necessary action.

II.I.3 *Main*

The *Main* class is the entry point of our application from where it can be accessed and run. It has a list of arguments that can be entered, however these arguments are currently not passed to the application.

II.I.4 *Command*

The *Command* interface is used to implement the *Command Design Pattern*, and as such ensures an execute() method is created for all classes that implement it, so it is a valid executable command.

II.I.5 *EnableAutoCommand*

The *EnableAutoCommand* class allows for meters and feet panel to be automatically be updated when the centimeters panel is changed. It has the implemented *execute()* method required by the *Command* interface which provides the listeners for the three panels so they can be automatically updated.

II.I.6 *SaveCommand*

The *SaveCommand* class allows for meters and feet panel to be updated to the corresponding conversion of the centimeters panel when it is triggered. It has the implemented *execute()* method required by the *Command* interface which executes the converters for the other panels to make the corresponding conversion from centimeters to feet and meters.

II.I.7 *MeasurementType*

The *MeasurementType* enum helps us organize the different type of measurements we have. It has a *MeasurementType()* constructor which adds the provided suffix, and the overridden *toString()* method to provided the appropriate suffix for each corresponding measurement.

II.I.8 ***Observer***

The *Observer* interface is used to give the template for our *Observer Pattern*. It asks to implement *update()*, which allows updates to occur to the particular dependent or to all dependents that are subscribed when another dependent is updated.

II.I.9 ***Subject***

The *Subject* interface is used within the *Observer Pattern* to update all the dependents to a new dependent or a state change. It has the corresponding methods *addSubscriber()*, and *notifySubscribers()* to provide the addition of dependents as well as communicate state updates.

II.I.10 ***ValueToConvert***

The *ValueToConvert* class takes a dependent and converts it from the dependents type to the corresponding types of every other dependent. To do these conversions it has the corresponding *setMeasurement()*, *getType()*, *getValue()*, *addSubscriber()* as well as *notifiy-Subscribers()* to propagate after it performs the conversion.

II.I.11 ***AbstractTextArea***

The *AbstractTextArea* is an abstraction of a *JTextArea* to allow us to create different types of measurement views. This class implements Observer so we can manage subscribers for a general text area view.

II.I.12 ***CentimeterConversionArea***

The *CentimeterConversionArea* is an extension of a *JTextArea* and is additionally an *Observer*. This class is used to display and input centimeter measurement.

II.I.13 ***FeetConversionArea***

The *FeetConversionArea* is an extension of a *JTextArea* and is additionally an *Observer*. This class is used to display and input feet measurement.

II.I.14 ***MeterConversionArea***

The *MeterConversionArea* is an extension of a *JTextArea* and is additionally an *Observer*. This class is used to display and input meter measurement.

II.I.15 ***MainView***

The *MainView* class is used to initialize our applications main frame. It adds all the *AbstractJTextArea*s on into its composed frame and sets all its related properties. All the initialization happens in the constructor and is finally set to be visible once the *MainView* is instantiated.

II.II Use Object Oriented Design Principles in Your Class Diagram

II.II.1 Inheritance

Inheritance is used in this project mainly to create concrete classes of the abstract class *AbstractJTextArea*. The classes *AbstractJTextArea*, *CentimetersConversionArea*, *FeetConversionArea*, and *MeterConversionArea* extend *AbstractJTextArea* and inherit all the properties. Additionally to make sure that all implementations are consistent, *AbstractJTextArea* sets the *JTextArea* dimensions to 250 by 250, and sets the foreground color.

II.II.2 Abstraction

Abstraction is used all throughout project to hide all the implementation details of certain functionality. For example, *Command* and the concrete commands all abstract their command functions and are executed through their entry points, *execute()* and used in the *MenuBarListener*.

II.II.3 Encapsulation

Encapsulation is used throughout the project to reduce the scope of the necessary attributes and implementation to the object level. This makes sure that we can only modify objects with getters and setters. For example, *ValueToConvert* provides *getValue()* and *getType()* to access the attributes of the measurement type and its value. Similarly, the *MainView* provides *getCentimeterTextArea()*, *getFeetTextArea()*, and *getMeterTextArea()* to provide the Controller access to the *AbstractJTextArea* components.

II.II.4 Polymorphism

Polymorphism complements inheritance in this project by allowing the different concrete classes of *AbstractJTextArea* to have a different background color and to have different behavior when updating their states based on the state of the subject they are observing, *ValueToConvert*.

II.III Explain in your report how you have used design patterns

1. **Observer pattern**: This project uses the Observer design pattern to manage update the states of all three concrete implementations of *AbstractJTextArea*, *CentimetersConversionArea*, *FeetConversionArea*, and *MeterConversionArea*. The program needs to keep track of the current value to convert and this is done using the *ValueToConvert* class. In the observer design pattern there are two main components: the observer and the subject. To implement this design pattern we created two interface classes, *Subject* and *Observer*. The *Subject* has a list of subscribers that it manages and alternatively, the observer is an entity that "subscribes" to a *Subject*. The key behavior that makes this design pattern function is the ability for a *Subject* to notify its list of subscribers. This is done through the overriding the *notifySubscribers(ValueToConvert)* method of a *Subject*. In our implementation, the the *AbstractJTextArea* implements *Observer* and *ValueToConvert* implements *Subject*. The *ValueToConvert* handles the subscriptions of other observers, which are the concrete implementations of *AbstractJTextArea*. When the state of *ValueToConvert* changes, it notifies all the *AbstractJTextArea* and updates their instances with the current *ValueToConvert*. When a *Subject* notifies its subscribers, the *ValueToConvert* is passed down so the text in the *JTextArea* can be overridden with the measurement conversion accordingly.

2. **Command pattern**: This project uses the Command design pattern to manage the different action commands of the menu bar in the *MainView* class. The program has two commands, the *SaveCommand* and the *EnableAutoCommand*. These two commands implements the *Command* interface to provide delegation of command execution to the concrete commands. The *MenuBarListner* uses a *HashMap* that hashes all the commands and once an action is performed (shortcut/key press) on the respective *JMenuItem*, the corresponding *Command*'s execute function is called. In this manner we can abstract the action of the user interacting with the menu bar and create the respective *Command* implementation. The different participating components of the Command pattern include the *Command* interface, the concrete implementations (*SaveCommand* and *EnableAutoCommand*), the invoker, the client, and the receiver. In our program the invoker is the *Controller* class. This class executes these commands based on the *actionPerformed()* event. The client is the *MainView* which instantiates the concrete commands and hashes them into the command *HashMap*. Lastly, the receiver is the *Controller* which controls the three main text area components. All these different components of the Command design pattern enable us to easily give our receiver a method to execute a commonly used function.

III Implementation of The Solution

III.I Describe how you have implemented and compiled all the classes of your class diagram in Java

The completion of this project was fairly quick and followed a methodical procedure that depended on the teamwork of all the members. We began by creating the required packages adhering to the MVC (Model View Controller architecture). This enabled us to create a separation of concerns of all the required components of the program. This enabled us to delegate different tasks within our team. For example, a member could be working on the view while another could work on the model for the program. After defining the structure of the program, we implemented the main required classes, *ValueToConvert*, *CentimeterConversionArea*, *FeetConversionArea*, and *MeterConversionArea*, as specified in the instructions. After understanding the requirements, we next applied the two required design patterns, Observer pattern and Command Pattern. We constructed UML diagrams of our initial design which included some abstractions which ended up being unnecessarily complex for this simple project (we will discuss this later). The next step after understanding that *ValueToConvert* is the subject and the text areas are the observers, we implemented, from the UML, our initial design following the design pattern. Along side the Observer design pattern, we used the Command design pattern by abstracting the different menu bar actions of the user into commands that would execute. For example, the "Save input centimeters" menu option has a corresponding *Command* implementation, *SaveCommand* using the Command pattern (similarly for *EnableAutoCommand*. The method of storing the menu *Command*s was by using a *HashMap* which used the *actionCommand* of the *JMenuItem* as a key. The technical implementations to allow the program to function as expected included adding *CaretListener*s to the different text area views. This allows us to use the *Controller* to update the other views with converted measurements when they gain the focus of the mouse.

III.II Specify the tools you have used during the implementation

1. **Editor** - IntelliJ IDEA 2021.2 (Ultimate Edition)
2. **JDK** - Java 16
3. **Build Automation Tool** - Gradle 7.2
4. **Latex Compiler** - Cloud Based Editor Overleaf
5. **UML** - Draw.io Flowchart Generator

III.III Create a short video

The demonstration video is available to view within the *README.md*

IV Conclusion

IV.I What went well in the software project?

Implementation of the project went very well due to the collaboration between all the group members. We managed to finish the project in a very short amount of time due to our teamwork and constructive feedback. We took an iterative approach to continuously refine the project after our initial implementation to get closer to the requirements. The combination of our teamwork and ability to provide and accept constructive feedback was a successful aspect of this project.

IV.II What went wrong in the software project?

Our team attempted to implement the project using high level of abstraction that was unnecessary for the requirements. The benefit of this approach was to abstract the project, but it also meant that we didn't . More specifically, there was a measurement model package to handle measurement related functionality. This complicated the project and we decided to take a more simple approach for the simple requirements of this project. This was inefficient since we had to refactor some code so this was a little downside to implementing this project.

IV.III What have you learned from the software project?

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

Working in a group has its benefits, such as the capacity divide up work and make sure that everyone gets to contribute. This can lessen the burden on an individual member and make the team more productive. Group work also enhanced the communication and efficiency of the team. We were able to have clear division of work and clarity when the work load is great. Members were able to support each other and help each other understand all the requirements for the project.
The disadvantages of working in a group is mainly scheduling a time for everyone in the group to work together. Some group members work and have to attend to other courses so it is difficult to maintain a schedule for everyone to work together.

IV.V Add a list to indicate the different tasks of the work that were assigned to each group member.

Setup: Amir, Amandeep

Documentation: Amandeep

Uml: Yazan, Mohamed

Report: Amandeep, Mohamed, Yazan, Amir