|  |
| --- |
| **(Autonomous Institute Affiliated to VTU)**  **Department of Information Science and Engineering** |
|  |
| A Project Report on |
| **“WUMPUS WORLD”** |
| *Submitted in partial fulfillment of the CIE for the subject*  **Object Oriented Analysis and Design Patterns(IS62B)** |
| by |
| **Vybhav Jain** |
| **1MS16IS144** |
|  |

**Table of Contents:**

1. Problem statement and proposed solution\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_03
2. Problem specification \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_04
3. UML Diagram\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_05
4. Patterns intended and context \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_06
5. Patterns in action\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_07
6. Future modifications\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_12

**1.Problem statement and proposed solution:**

Q) Show the application of various design patterns through the creation of the game –Wumpus World.

Games can provide a variety of sensory experiences for people of all age groups. Additionally, it also actively engages the user’s attention which leads to a better understanding of the matter as compared to the traditional pen and paper approach. Keeping this in mind, the objective of this project is to show a visual representation of the famous “Wumpus World” game which was introduced by Genesereth, and is discussed in Russell-Norvig(2003). The Wumpus World is a simple world (as is the Block World) which is a famous example to represent knowledge and to give clarity involved in logical reasoning approaches. Wumpus world was the basic example to teach an AI system on how to make decision on the basis of mathematical deductions. Here, different methods of the SWING library have been used including but not limited to frames, buttons and panels.

**2.Problem specification:**

The Wumpus world problem is a 4x4 puzzle where the objective of the game is to find the gold. The user must safely navigate their way around bottomless pits of death and evil Wumpus creatures to locate the gold hidden on the board. The goal here is to minimize the number of steps and maximize the points. The project has been implemented in Java and the various components of frames have been used to create a GUI. The various elements in the game are :

1) Wumpus: A deadly monster that lurks in one of the square. This monster emits a pungent odour that can be detected in the adjacent boxes that this monster is present in. When facing a possibility of the presence of this monster, shoot an arrow to kill it or avoid it by taking a different route.

2) Pit: A bottomless hole which will trap you forever. Avoid this at all costs. The pit is so deep, that the breeze that blows because of it can be felt in the adjacent squares, use that as a warning and tread carefully.

3) Gold: The very objective of the game to retrieve this priceless mineral. Identify the square containing the gold by its lustrous glitter which can be seen in the adjacent squares.

4) Actions: There are multiple actions, the most basic being left,right,forward and shoot. Left and right as the names indicate are meant to turn towards their respective directions. Forward will move the player’s position by one square in the direction they are facing.Each forward move costs one point. Lastly, shoot action fires an arrow in the facing direction, and if any wumpus is present, it eliminates it. But, each arrow shot costs 10points.

5) Combo moves: While a single move is good, the goal of the game is to minimise the number of steps. So, here two new actions are introduced: up-right and up-left which as the names indicate, irrespective of the direction, move the player to the diagonal right and diagonal left box respectively at the cost of only 1 point instead of 2.

6) Magic ladder: Present in one of the squares, this ladder will transport the player to a random square. There is no way to find out which square contains the ladder or which square will the ladder lead you to. The only advantage here being the teleportation charge is zero.

**3.UML Diagram:**

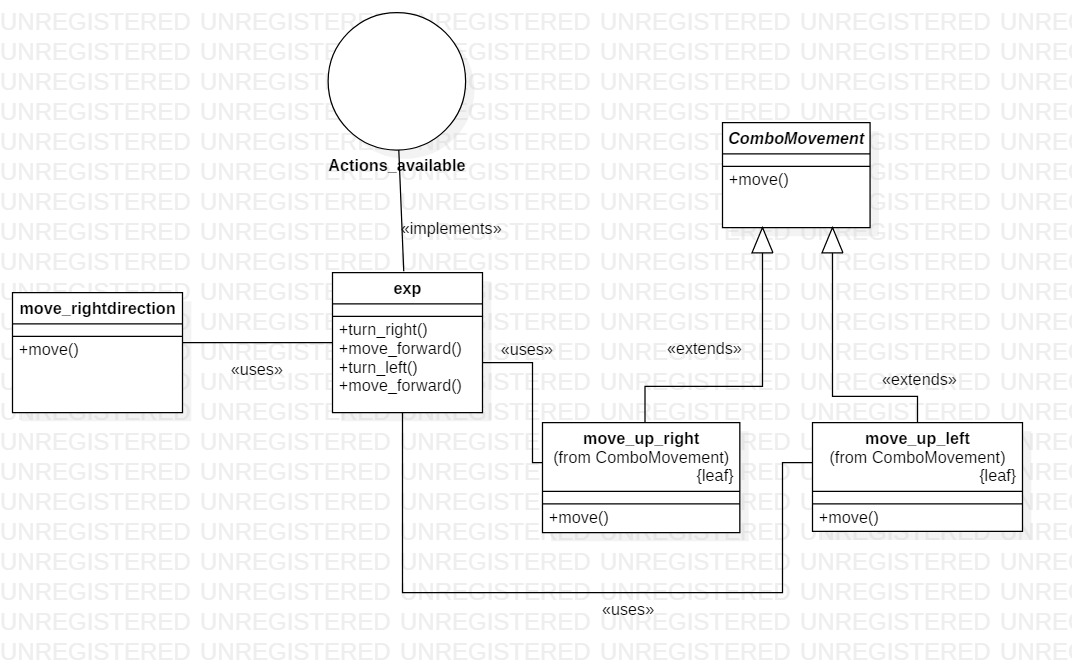
****

Figure1: Shows the various classes and interfaces and the relations between them

Files present:

1. exp.java (Contains the main program)
2. Actions\_available.java(An interface which defines a method)
3. move\_rightdirection.java(Contains a method to turn towards the right direction)
4. ComboMovement.java(An abstract class which contains an abstract method called move which is later to implemented to move in the diagonal direction)
5. Move\_up\_right.java(Inherits ComboMovement.java and writes a method to move in the up-right direction )
6. Move\_up\_left.java(Inherits ComboMovement.java and writes a method to move in the up-right direction )

**4.Patterns intended and context:**

1. **Adapter Pattern:**

An Adapter Pattern is generally used when we want to add a particular method or piece of functionality to our program but instead of coding it manually,we call it from somewhere else. What we want is already present somewhere else, maybe another file or system. But it has a different name and signature.

So in this pattern, we create a private object of the class containing the piece of code we want and we call its method under the name we require. This is like renaming the method and changing the signature based on our requirements.

The advantage of this is that now we don’t need to write the code from scratch, and from the perspective of a game, we will always have modifications in the future most of which would already have been coded by someone else. So here, we not only make our work easier but also future changes and requirements can be handled more easily.

Context:

* The method to turn left has been coded, but the player might also want to turn right.
* We know that some other game has a method to turn right and we do not want to code it ourselves as this would be a waste of time.
* The other game has the method as move() while we need to define it as move\_right() [Due to the interface that is implemented by the exp class]
* So an object of move\_rightdirection[the class containing move] is made and under move\_right() we call the method move.
* Thus, we have successfully coded the method to turn right without manually coding the entire logic.

1. **Strategy Pattern:**

Sometimes to achieve a particular task, there might be more than one way possible. In cases like this, the person or user doesn’t really care which strategy is followed to achieve the goal as long as the goal is achieved. So here, we use the Strategy Pattern which gives the user the option to choose any method at run time to achieve the particular goal.

The main advantage of this pattern is that the during run time, any of the strategies could be followed; this adds a lot of flexibility to the code.

Context:

**C**onsider an action to move in the upward right square. There are two ways to achieve this:

* Move in the right diagonal direction or
* Turn to the right/left direction, take a step forward. Then turn to the left/right direction and take another step forward.
* While each of the ways accomplishes the task, they have different rewards.
* By using this pattern , we give the user the flexibility during run time to choose either of the method to achieve the goal.

**5.Patterns in action:**

1. **Adapter Pattern**

**Module: Actions\_available.java**

package wumpus;

public interface Actions\_available {

public void turn\_right();

}

* We have an interface called Actions\_available which has a method called turn\_right()

**Module: exp.java**

package wumpus;

import java.util.Random;

…

…

public class exp implements Actions\_available {

….

…. // contains the class code

public static void main(String[] args) {

exp t1 = new exp(); //Creates object of exp class

}

}

* We have a class called exp.java which implements Actions\_available and contains the main code as well the void main() code.

**Module: move\_rightdirection.java**

**package** wumpus;

**import** javax.swing.JButton;

**public** **class** move\_rightdirection {

**public** move\_rightdirection()

{

}

**public** **void** move(**int** index,JButton buttonsnew[])

{

String s1 = buttonsnew[index].getText();

String replaceString = "";

**if**(s1.contains("^"))

{

replaceString=s1.replace("^",">");

}

**else** **if**(s1.contains(">"))

{

replaceString=s1.replace(">","v");

}

**else** **if**(s1.contains("<"))

{

replaceString=s1.replace("<","^");

}

**else** **if**(s1.contains("v"))

{

replaceString=s1.replace("v","<");

}

buttonsnew[index].setText(replaceString);

}

}

* We have another class called move\_rightdirection which exists independently on some other system.

**Module: exp.java**

**public** **void** turn\_right() // inherited method from actions\_available to turn right

{

move\_rightdirection mrd = **new** move\_rightdirection();

mrd.move(index,buttonsnew);

}

* While the method present in move\_rightdirection is exactly what we need, we can’t directly use it as the method name is different than what are implementation is(i.e., turn\_right()).So ,we create an object of move\_rightdirection and under the required method name and access its method.

This method of creating a wrapper object for a pre-existing class/interface and calling it under the required method name as our implementation is an example of **Adapter Pattern**.

1. **Strategy Pattern**

**C**onsider an action to move in the upward right square. There are two ways to achieve this:

1. Move in the right diagonal direction as shown in figure2.

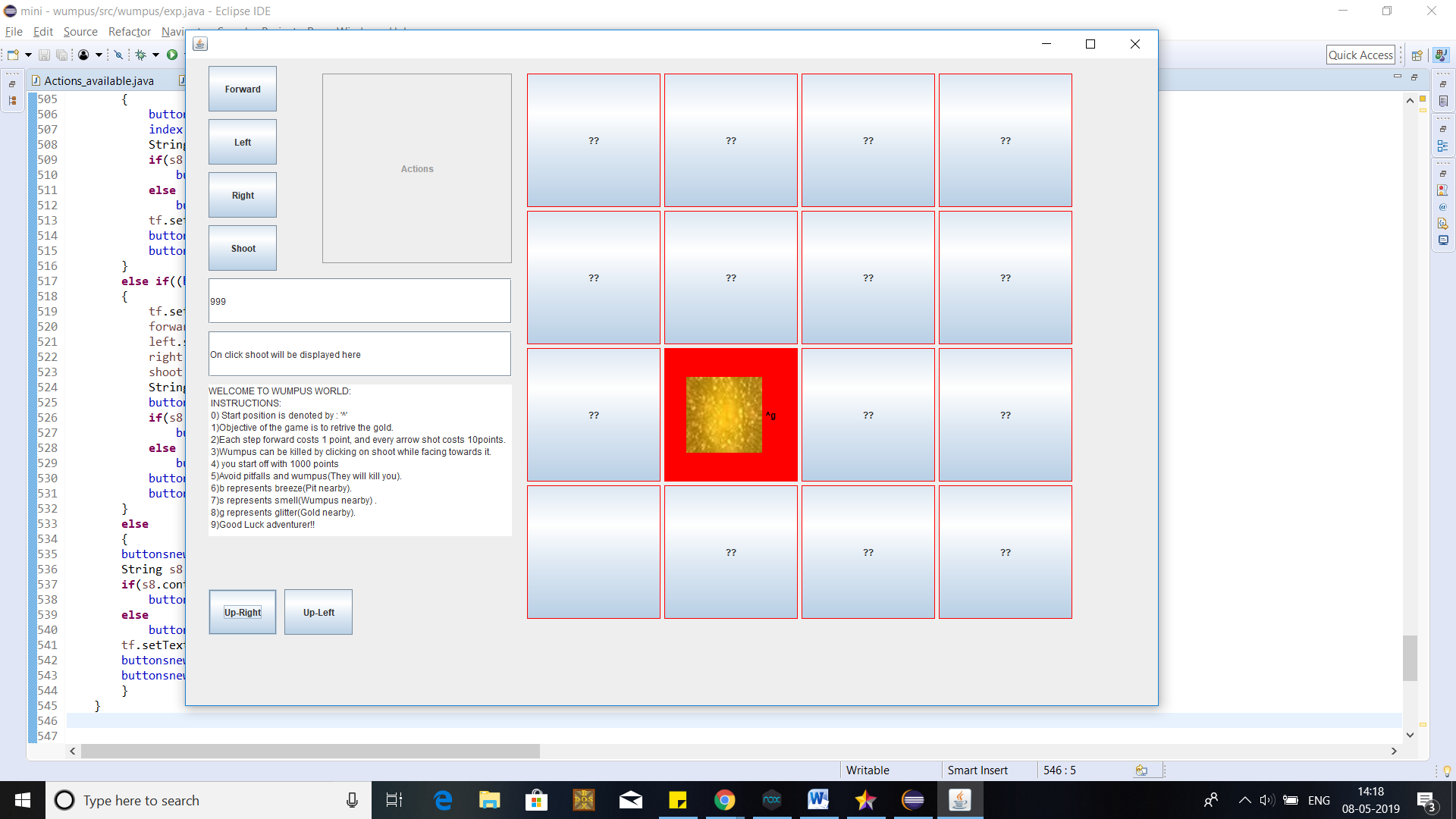
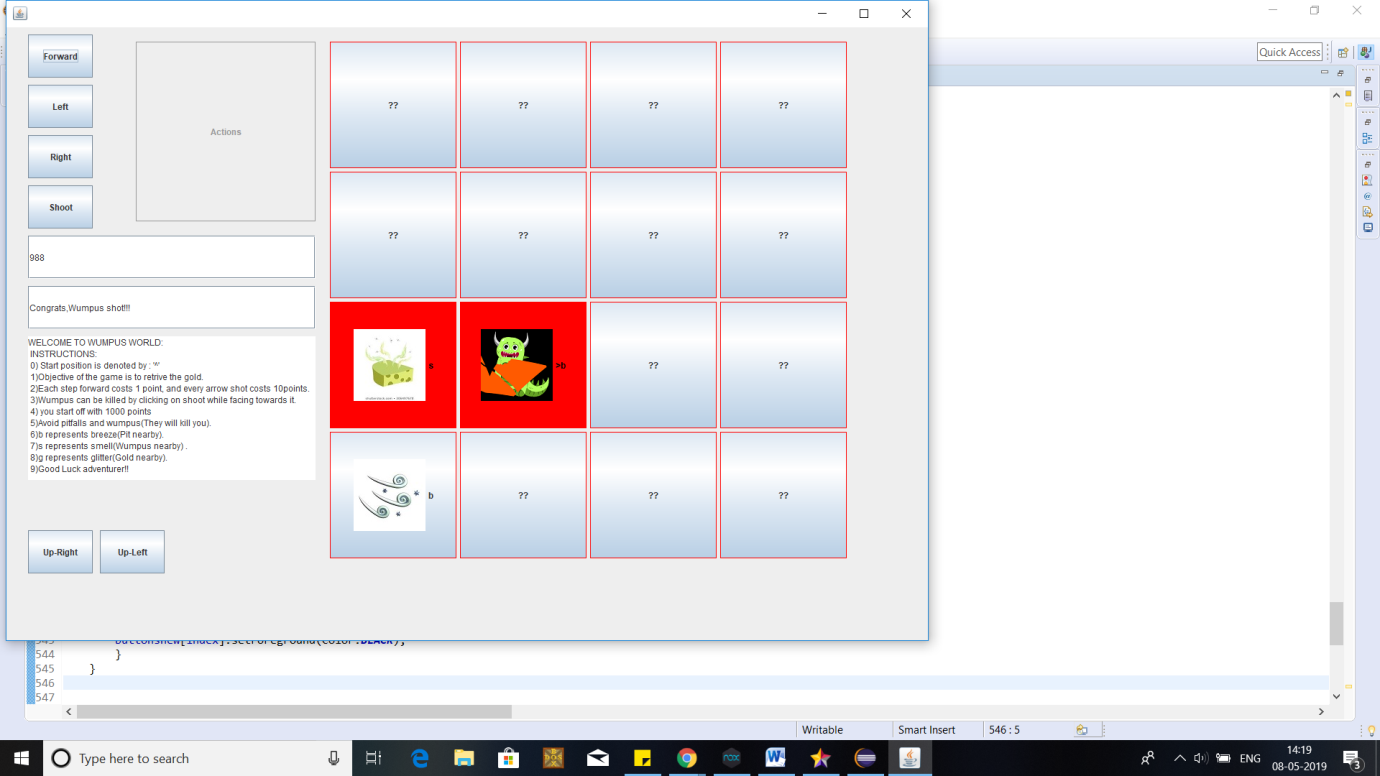
****

Figure2: Shows the right diagonal movement.

1. Turn to the right/left direction, take a step forward. Then turn to the left/right direction and take another step forward as shown in fig3.

**** Figure3: When we move forward, turn right and move forward again.

Below are the code snippets that are used to achieve the above actions:

**Module: ComboMovement.java**

**package** wumpus;

**public** **abstract** **class** ComboMovement {

**public** **abstract** **int** move(**int** index,**int** num,**int** score);

}

* An abstract class which provides an abstract method move which is later defined as various other combination moves.

**Module: ComboMovement.java**

**package** wumpus;

**public** **class** move\_up\_right **extends** ComboMovement {

**public** **int** move(**int** index,**int** num,**int** score)

{

num = index;

**if**(index<=3)

{

index+=4;

}

index-=4;

**if**(index%4 == 3)

{

index-=1;

}

index+=1;

score-=1;

**return** index;

}

}

* A class of ComboMovement which writes the code in the method move for performing the function of diagonal right as illustrated in Figure3.

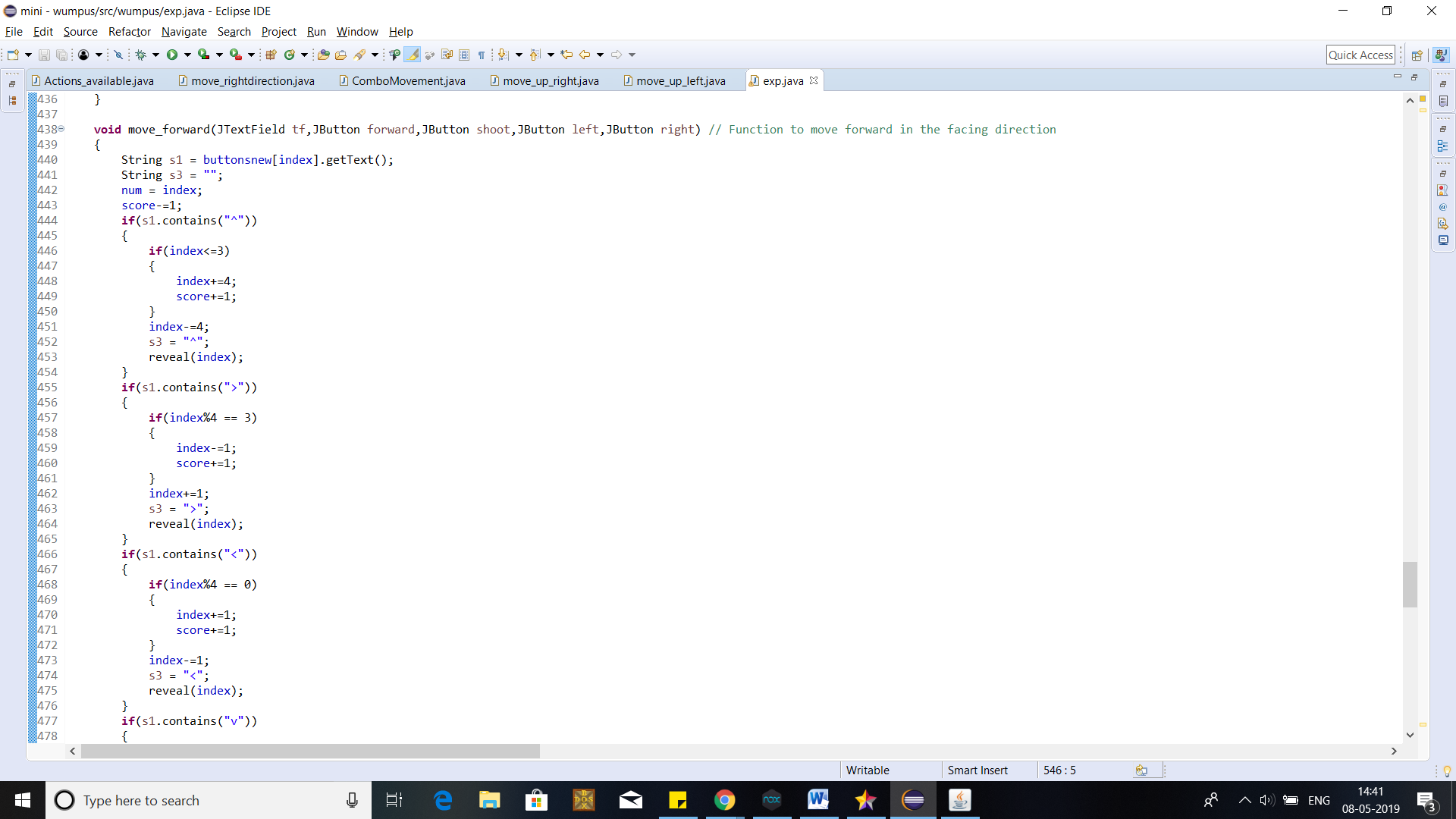


Figure4: The code for moving forward in the facing direction. This in combination turn left and turn right (As shown in Figure3) are used in combination to achieve Figure6.

Thus, we have one or more ways to implement the same task, i.e.., moving in the right diagonal direction. This is an example of **Strategy Pattern.**

**6.Future modifications:**

The major advantages of patterns can be seen in cases of future requirements and modifications. In this section we are going to discuss some possible long term changes and whether or not the implementation of the used patterns can help us or not.

Possible future requirements are:-

* There might be a new method to move from one place to another.
* An extra element such as a monster might be added to the game.
* There might be a requirement to extend the size of the box.
* The number of pits and monsters can be asked to be increased.

Now we will discuss the solutions for the above problems:

1. We already have the traditional method of left/right + forward actions on the GUI to help us to go anywhere. Any new method to achieve the same goal can be considered a new strategy. Since we have implemented the strategy pattern here in run time, adding a new method will be an extremely easy task. Also, when we add a new method, we need not make changes to any other parts of the code. **Thus, for this particular requirement, using the strategy design pattern has definitely helped us and made long term maintenance and additions easy.**
2. When we need to add a new monster, we need to manually hard code the entire specs of the monster. But we know that some game out there will definitely have a monster with similarities to our new addition. Here, we use the adapter pattern to incorporate this new monster into our game with just a few lines of coded**. Hence, for this requirement, using the Adapter pattern has proven advantageous and reduced our work significantly.**
3. For this problem ,we need to manually changes the size of the box being created in the code and not only that, need to change multiple parameters in the forward conditions which is a LOT of work and has HIGH chances to cause a bug if we miss a box. **None of the implemented patterns help us in this problem.**
4. We need to now create another monster or pit or create an array which dynamically adds monsters/pits based on the user requirement and need to make a lot more changes to the code which again is a LOT of work and lot of redundant code. **None of the implemented patterns help us in this problem.**

While problem 3 and 4 where not solved by the implemented patterns, a closer thought into these problems show us that the mistake we made can be that we have given the object creation responsibility as well as what it does responsibility to the same class which causes a lot of confusion. Here, using the Abstract Factory pattern and giving the responsibility of creating the box layout/monster/pits to a factory can help solve problem 3 and 4 to quite some extent.

Thus, Design Patterns help in long term maintenance of the code and help us solve future requirements quite easily if we use it in the right way, know about the different kinds of patterns available and their purpose and have at least the basic knowledge about Object Oriented Principles.