Homework 5
Computer Science II: Data Structures
CS:2230

Name: Nadav Kohen

1 Playing the Game

Simply run the main method in the file, Interface.java, to run the game (do not run the Battle.java, it is an older version). The user must choose how many Fighters each Team will have, my personal computer could handle no more than four due to memory restrictions so I recommend games of four Fighters per Team. Next the user must set up the game by selecting the Teams in order. Finally, the GUI will appear and the user can take turns with the computer by clicking the buttons at the top of the Interface for the Fighter's moves. Specific instructions for the game will appear if the user clicks the Instructions button.

2 The Fighters (For full details see javadoc)

2.1 The Abstract Fighter Class

In order to create a new type of Fighter, one must extend the Fighter class. This allows for concrete Fighters to interact with any other Fighter regardless of their class. The Fighter's fields are hp, mrate, prate, and a list of Effects called effects (see next subsection). The Fighter class implements methods that take damage, and activate effects, as well as drawing a Fighter (with no accessory) using a Graphics object and also an abstract copy method that must be implemented by all Fighters which returns a copy of the Fighter by value.

2.2 Effects

An Effect can is constructed by simply giving it as parameters, the duration, damage to be dealt, and a name of the Effect. This effect should then be added to a Fighter's effects field by using addEffect. Any Fighter can then use activateEffects to activate all of their Effects and discard those whose duration has ended.

2.3 Teams

The Team class simply stores an ordered list of Fighters that is meant to represent a Team where the first Fighter is the active one, as well as a reference to the enemy Team. This functionally gives access to all information of the game through any Team object as all Fighters can be accessed. This object also allows for easy changes in which Fighter is active, as well as calculation of total hp and also has a method called activateEffects that calls on all of its Fighters activateEffects methods.

2.4 Knights

The Knight class extends the Fighter class and implements the copy method accordingly. The Knight has three moves: heal, melee, and sacrifice as described in the problem description. There is also a drawKnight method that will call on drawFighter for this Knight but will also draw a sword in one of the Knight's hands.

2.5 Wizards

The Wizard class extends the Fighter class and implements the copy method accordingly. The Wizard has four moves: flame, fireworks, groupHeal, and fumes as described in the problem description where fumes creates an Effect named "fumes". There is also a drawWizard method that will call on drawFighter for this Wizard but will also draw a hat on the Wizard's hat with a kind of star on it.

2.6 The Move Enum

The Move enum contains a constant for every move of every Fighter in existence. This allows for easy-to-read code and more uniformity in method parameters and return types. This enum also contains methods for each Fighter that return a Move array of all of that Fighters Moves.

3 AI Engine

3.1 The Node Class

This class implements a tree which stores all possible futures for up to eight turns, and then assigns values to the leaves signifying how favorable these positions are and then reasons what Move the Team at hand should make. Node is an abstract class which encodes only the tree structure but not the metric to be used to evaluate the strength of a position, thus the setEndMetric method which should initialize metric for the leaves of this tree must be overridden by any subclass and this method encodes the actual strategy to be used. The addToFuture method must also be overridden to ensure that what is added to future is of the subtype and not of type Node so that the correct setEndMetric will be called when leaves are reached.

3.2 The AI Class

This class extends the Node class and can be used for any setEndMetric desirable as it uses a public static field called strat of type Strategy (see subsection below) which determines what is done in setEndMetric. To use AI, one must simply initialize it with a Team right before that Team's first move and then access AI.move and AI.switched as necessary. Then, after each user Move, simply call on AI.updateTree.
The following are the currently implemented strategies.
Delta HP: This maximizes the difference in total hp between the two teams.
HP Ratio: This maximizes the ratio between the two teams of total hp.
Survival: This maximizes the AI's total hp.
Kill Opponent: This minimizes the user's total hp.
Weighted HP: This maximizes the difference in total hp between the two teams but with hp over 100 from any Fighter weighted at $\frac{1}{4}$ and hp from any Fighter below ten not counted at all. The rationale is that a Fighter with 200 hp is much worse than two Fighters each with 100 as a single Sacrifice move from a Knight could wipe them out. Likewise, three Fighters with ten hp each is much worse than one fighter with thirty hp as the three fighters can be wiped out by a single Fireworks move from a Wizard.
Random: This sets metric randomly.

3.3 The Strategy Enum

The Strategy enum contains a constant for every case of setEndMetric that has been implemented by AI. This allows for easy usability of AI as a single class as opposed to needing to implement a new class subtyping Node for every new Strategy.

4 The Process for Creating New Fighters/Strategies

4.1 Extending Fighter

1. Create two constructors, one being default that simply calls on Fighter's constructor giving it the desired prate and mrate; the other must ask for an int and an ArrayList¡Effect¿ and then simply call on Fighter's constructor giving it the same prate and mrate as well as the int and the ArrayList. The latter constructor is for use by copy, but could be used anywhere.

2. Code the moves of this type of Fighter.

3. Override the abstract copy method of Fighter, this should look exactly the same as the copy methods of Knight and Wizard but with your return type and return a new Fighter of your type.

4. Create a draw method that takes a Graphics object and then calls on drawFighter and then adds some sort of accessory to your Fighter to distinguish them from other Fighters on a GUI.

5. Override toString with the single line: return "[Your type here]: " + super.toString().

6. Add all of this Fighter's moves as constants in the Move enum, then create a method that returns an array of all of this type of Fighter's Moves and also add these moves to the allMoves method of Move.

7. Finally, add the following code to the setFuture method of the Node class:
```
//Create [Your type here] Move children
if(t.active() instanceof [Your type here])
{
for(Move m: Move.all[Your type here]Moves())
addToFuture(m,null);
}
```

4.2 Extending Node

There are two ways to do this:

1. Create a new class that extends Node.

2. Write the following code:
```
/**
* Constructs this Node and makes a decision for t.
* @param t The computer's Team to make a move.
*/
public [Your type here](Team t){
super(t);
}

private [Your type here](Team t, [Your type here] parent, Move m, boolean last, Fighter switched){
super(t,parent,m,last,switched);
}

@Override
protected void addToFuture(Move m,Fighter s){
Team t = copyT();
boolean last = Battle.turn(t,m,s);
future.add(new [Your type here](t.enemyTeam,this,m,last,s));
}
```

3. Finally, override the setEndMetric method of the Node class and initialize metric for the leaves of the tree, keep in mind you must check if (where % 2 == 1) as if this is true, t is the computer's Team, otherwise t.enemyTeam is the computer's Team.

An easier alternative:

1. Add a case to the switch on strat in the setEndMetric method of the AI class and give it a name and implement a method.

2. Then simply add this name to the Strategy enum.

5 The Graphical User Interface

Originally a JDialog-based interface was implemented to test the code, this was in Battle.java, but this is now only used for its static methods that simulate the game, although this interface is still fully functional. In place now is a JFrame called Interface which contains buttons for all the Moves possible to the user's

active Fighter as well as buttons to display the details of all Fighters, and to display the instructions of the game. This JFrame also has a Panel called GamePanel which does the work of drawing out both Teams and spacing the Fighters accordingly, as well as displaying a correct end screen if the game has ended.
IMPORTANT NOTE: If new Fighter types are created, this Interface may not be flexible enough to have a simple integration i.e. if this Fighter has more than four moves as Interface was made with only the game as it is in mind, but an integration should not be overly challenging. Implementing new strategies via AI, however, should pose no problem whatsoever, simply set AI.strat equal to your new type of Strategy.

6 Notes on Current Gameplay

1. This game is not particularly enjoyable but the AI (for many strategies) does play pretty well, especially if the user is unaware of what Strategy is being implemented.

2. Battles between Fighters of the same type is somewhat broken. For example, when a Knight faces another Knight, they can deal only 20 damage with Melee but can Heal themselves for 30; this inevitably ends in a Sacrifice.

3. Gameplay is largely affected by the types of Fighters ad more specifically their ordering on each Team. With some initial positions, it is near impossible to lose and likewise with some initial positions it is almost impossible to win.