Kendall Helms

AS2 – Project Writeup

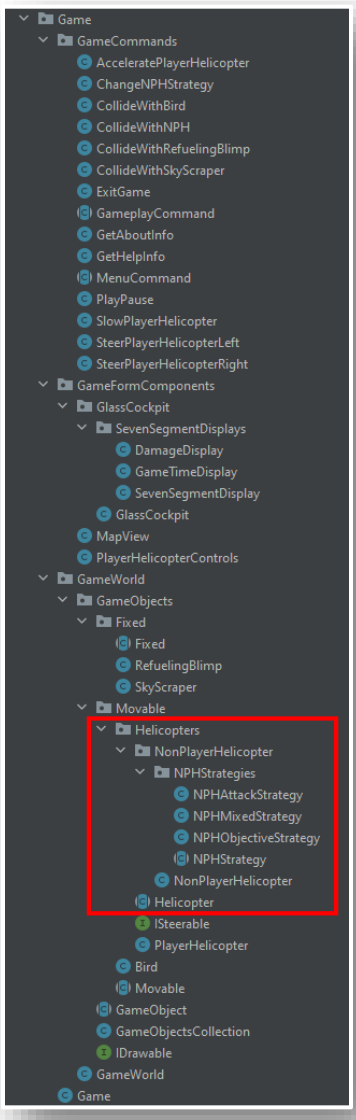
CSC 133 – 04

4/19/21

SkyMail 3000 ver. 0.2.0

Assignment 2 Project Writeup

Project Structure – This project is organized in a hierarchical structure. Packages are used to break the program into component parts, which are then further broken up into smaller parts as necessary. For example, the package, *Helicopters*, contains the abstract parent class, *Helicopter*, as well as all of its extended classes. One of these extended classes, *NonPlayerHelicopter*, utilizes the Strategy design pattern, and therefore implements objects of the *NPHStrategy* class. Since *NPHStrategy* and its extended classes specifically interact with *NonPlayerHelicopters*, they are held in a *NPHStrategies* package, nested within an overarching *NonPlayerHelicopter* package which includes the class of the same name. This keeps things tidy and accurately represents how the pieces fit together. The image to the right shows the structure of the entire project, as well as the example above outlined in red.



The main components of this project are *Game*, *GameWorld*, *GameFormComponents*, and *GameCommands*. *Game* is a Codename One *Form* GUI component, which serves dual purpose as the container for all of the game's visual components, as well as the game's controller class. *GameWorld* is the game's model, containing all of the data and methods which describe the game. *GameFormComponents*' title is self-descriptive. It contains all of the visual components of the game. Finally, *GameCommands* holds all of the various commands the player will use to interact with the game.

Game Component – *Game*, the game's controller, is a single class, which all other components of the game are slave to. This class instantiates each of these components within its scope, and controls them either directly, or indirectly through the objects it does directly control. This importantly includes the game's model, *GameWorld*. After instantiation, *Game* interacts with its components through their public methods. This design ensures that while each disparate part works independently of one another, they can be synchronized by *Game*.

Game implements the *Runnable* interface, utilizing a *UITimer* which ticks every 20ms. Each tick of the timer checks *GameWorld* for a win or lose condition, then moves the game model forward an increment of time if appropriate. It also contains a *pause()* method, which stops the *UITimer* and invokes the necessary methods of its components to stop the game where it is. A *play()* method reverses this condition, allowing the player to resume where they left off. In the case of either a win or lose condition during gameplay, *Game* pauses the game, informs the player, and offers to reset the game and start from the beginning.

This class is also implemented as a Codename One *Form* GUI component, displaying all of the visual components of the game, including the *GlassCockpit*, *MapView*, and *PlayerHelicopterControls,* all of which are contained within the *GameFormComponents* package. The first two of these act as observers of *GameWorld*, taking information from *GameWorld* and displaying it graphically. This information sharing between components is facilitated by *Game*, using a call to the *GameWorld:init()* method to pass in the *MapView and GlassCockpit* objects which *Game* owns. This call is made on the first tick of the *UITimer* (a flag is flipped at the call), ensuring that *MapView* has been laid out to its final size on the screen ahead of time.

Finally, in order to allow the player to control the game, *Game* uses *Command* objects in conjunction with the on-screen buttons implemented in *PlayerHelicopterControls*, the Codename One *Side Menu,* and key-bindings. To simplify the code for these commands, and to ensure that they are always targeting the correct instance of *Game*, *Game* is written in the singleton pattern.

GameWorld Component – *GameWorld* is the game's model. It contains and manipulates all of the important data which represent the game. This includes all of the game's objects (i.e., *Helicopters, Birds, RefuelingBlimps, and SkyScrapers*), as well as game state information like the number of lives remaining.

GameWorld, like *Game*, is written in the singleton design pattern mostly for the ease of writing commands, while also guaranteeing that only one instance of *GameWorld* exists at a given time. This means that resetting the game is largely painless, and is a failsafe against any mistakes in this process.

This class has a two-step initialization process, in which the singleton object is first instantiated, and summarily initialized with a call to the *init()* method. This ensures proper timing, the use of which is described above in *Game.* The *init()* method first saves the *MapView* and *GlassCockpit* objects passed to it by *Game*, then sets the size of the area which *GameWorld* represents based on the internal on-screen size of the *MapView*. An internal class, *Area*, is used for this purpose. Next, it spawns the game's objects into their proper locations (using the internal *Location* class), in the proper order, and at the proper scale based on the size of the display. Finally, it passes the relevant objects into the *MapView* and *GlassCockpit* in order to provide the data for them to display.

```
public void init(MapView mapView, GlassCockpit glassCockpit) {
    this.mapView = mapView;
    this.glassCockpit = glassCockpit;

    worldArea = new Area(mapView.getInnerWidth(), mapView.getInnerHeight());

    spawnSkyScrapers( n: 5);
    spawnRefuelingBlimps( n: 3);
    spawnBirds( n: 5);
    spawnPlayerHelicopter();
    spawnNonPlayerHelicopters( n: 4);

    mapView.setGameObjectsCollection(gameObjectsCollection);
    glassCockpit.setPlayerHelicopter(getPlayerHelicopter());
}
```

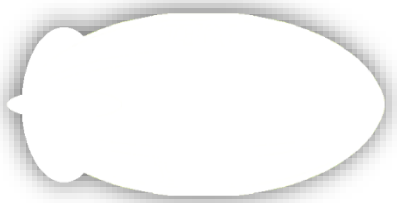
- ***GameObjectsCollection* –** The spawned objects are saved in a collection of type *GameObjectsCollection*. This class is an extension of Java's *AbstractCollection*, which allows for easy implementation of custom collection types. In this case, *GameObjectsCollection* will accept objects of any type which extends the abstract *GameObject*. It operates in all the same ways as a standard Java collection, but also contains methods which return subsets of the total collection based on object type. It also provides a method which returns the location of the player's spawn.

```
for(SkyScraper s : gameObjectsCollection.getSkyScrapers()){...}
```

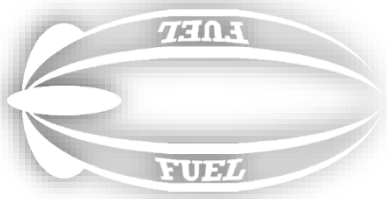
The game's objects are described by a hierarchy of classes, which describe the general class of objects, objects which are fixed in place, such as *SkyScrapers*, and movable objects, such as *Birds*. The general *GameObject* class implements an *IDrawable* interface, which contains only a single *draw()* method. This method is invoked by *MapView* to draw a representation of the *GameObject* on the screen. Each subclass overrides this method in order to uniquely represent the type graphically. Below are a list of interesting design choices surrounding *GameObjects*.

- ***GameObject* –** *GameObject:draw()* doesn't do any operations on the graphics. Instead, it updates the *MapXY[]* and *ImageXY[]* fields, which represent the coordinates of the object's location on the *MapView's* graphics plane. These fields can be accessed by all subclasses through getter methods.
- ***Fixed Objects* –** On spawn, fixed objects are placed in random locations, taking into account their proximity both to the edges of the game area, and to other fixed game objects.

- ***RefuelingBlimps –*** These objects, while fixed in-game, represent real-world movable objects. As such, the game draws them facing in any one of the four cardinal directions. Also of note, these objects are represented by two colors which may be changed independently of one another; these are the main body color, which is held in the parent class' *color* field, as well as a secondary *color2* field held in the *RefuelingBlimp* class which represents the accent color, as well as the text showing reading the blimp's



GameObject:color target



color2 target

capacity value. These colors are used to recolor the two images used to represent the *RefuelingBlimp*, shown below.

- ***Bird –*** This class uses a filmstrip type animation to represent the *Bird's* flapping wings. The speed of this animation is determined by the speed of the *Bird*.

- ***Helicopter –*** *Described below.*

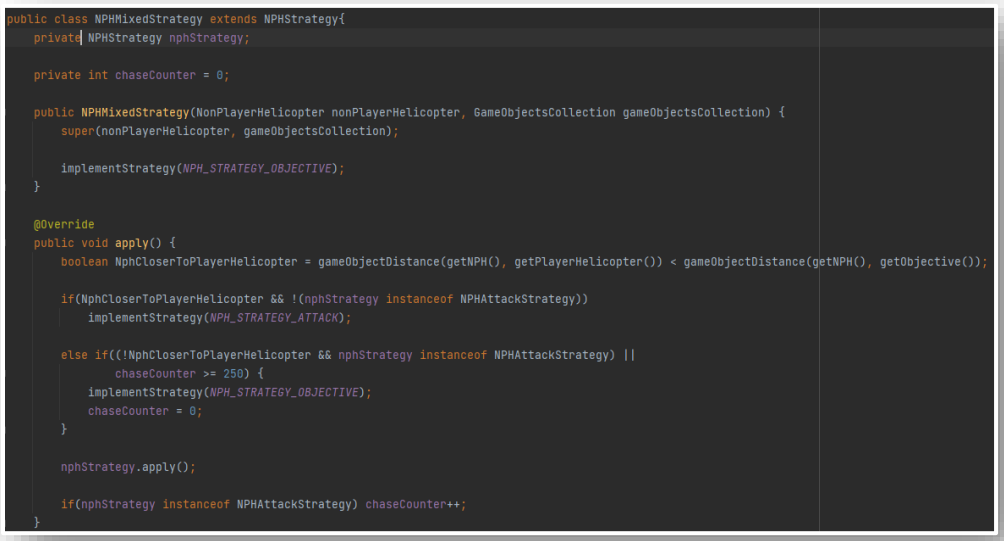
The *GameWorld* class implements all of the methods which outline gameplay. Of great importance is the *tick()* method which increments the model forward in time, as referenced in *Game*. This method's first priority is to invoke the *move()* method of each movable object held within the *GameObjectsCollection*. Next it checks to see if any of the relevant objects need to be despawned and does so. Then, it checks to see if the *PlayerHelicopter* has become immobilized due to taking too much damage, or running out of fuel. If that's the case, and there are lives remaining, the *GameWorld* reinitializes with one less life. Finally, it invokes the update methods of the two observers so that they display current information.

Also among these methods are those which are invoked by *Game's* commands. These are what allow the player to control the *PlayerHelicopter*, as well as simulate collision events, and change the *NonPlayerHelicopters'* strategies.

Helicopter – This *GameObject* requires its own section. For assignment 2, this class was revised to become an abstract class. The two subclasses, *PlayerHelicopter* and *NonPlayerHelicopter* are new for this iteration. *PlayerHelicopter* is more or less a straight extension of *Helicopter*, but implemented in the singleton pattern. More interesting is the *NonPlayerHelicopter.*

This class' implementation again is very straight forward, with only a few extra, simple methods which implement the strategy design pattern. This means that *NonPlayerHelicopters* can apply *NPHStrategy* methods to guide their movement.

In this implementation, *NPHStrategy* is an abstract class, which describes how a *NonPlayerHelicopter* should move in relation to a target object, and the classes which extend implement different methods for choosing that target. The three of these subclasses are *NPHAttackStrategy, NPHObjectiveStrategy*, and *NPHMixedStrategy*. Simplest of these is *NPHAttackStrategy*, which sets the target as the *PlayerHelicopter*, and follows it without interruption. *NPHObjectiveStrategy* sets the target as the next SkyScraper in the sequence. More interestingly, *NPHMixedStrategy* applies one of the other two strategies within its scope, based upon which of the two target objects is further away.



```
public class NPHMixedStrategy extends NPHStrategy{
    private NPHStrategy nphStrategy;

    private int chaseCounter = 0;

    public NPHMixedStrategy(NonPlayerHelicopter nonPlayerHelicopter, GameObjectsCollection gameObjectsCollection) {
        super(nonPlayerHelicopter, gameObjectsCollection);

        implementStrategy(NPH_STRATEGY_OBJECTIVE);
    }

    @Override
    public void apply() {
        boolean NphCloserToPlayerHelicopter = gameObjectDistance(getNPH(), getPlayerHelicopter()) < gameObjectDistance(getNPH(), getObjective());

        if(NphCloserToPlayerHelicopter && !(nphStrategy instanceof NPHAttackStrategy))
            implementStrategy(NPH_STRATEGY_ATTACK);

        else if((!NphCloserToPlayerHelicopter && nphStrategy instanceof NPHAttackStrategy) ||
                chaseCounter >= 250) {
            implementStrategy(NPH_STRATEGY_OBJECTIVE);
            chaseCounter = 0;
        }

        nphStrategy.apply();

        if(nphStrategy instanceof NPHAttackStrategy) chaseCounter++;
    }
```

UML Diagram –