Quinn Roemer & Logan Hollmer

Dr. Daryl Posnett

CSC 133 MWF 9-9:50am

March 1, 2020

Assignment 1 – Documentation

In this assignment, we were tasked with refactoring the Snake game in the Packt book using several design patterns we discussed in class. To do this, we were tasked with creating a GameObject class that we would extend in several of the required classes (Snake, Body, Apple, etc). The GameObject that we created for this assignment includes a Point object to hold the location of the GameObject on the screen and also a Bitmap type to hold the sprite the object is associated with. In addition, this class has a concrete Draw method that uses the guaranteed variables (Point & Bitmap) to draw the object on the screen. This Draw method is used in every object except Snake, as this is overridden due to some necessary features required for that object.

In the project documentation provided to us, we were guided to implement a new Apple class inside the code using a builder strategy. The intent with this strategy was to allow for the creation of different types of objects using different colored sprites to represent apples of varying point values. In our code, this was implemented by creating a class called MyApple. In this class, we have one private integer to hold the point value, a couple of getters for the location and point value of the apple, and lastly a private constructor. This private constructor prevents the object from being instantiated unless you go through the builder class contained in the same file called AppleBuilder. This class has public methods for setting the apple color and associated Bitmap (using a passed String), setting the location of the apple, and setting the point value using a passed integer. Once all the options are selected the build method is called which returns a new MyApple. This activates the private constructor and the contents of the apple are automatically set.

Another requirement outlined in the project documentation is the odds of spawning apples of certain colors (with associated point values) and the score multiples which determines how many apples are on screen at once. This was implemented using an apple container class called AppleBasket that holds a HashMap of all the current active apples. Please note, a Hashmap was used to simplify the collision detection of the Snake Head hitting the Apple. Instead of having to search through a list of all of the spawned apples, we just check if there is an apple there using the Point object as a key. Inside the AppleBasket class, we have a constructor that creates the first apple which is always guaranteed to be a standard, one point, red apple. A method for removing an apple from the HashMap using a passed Point object, getter methods for both the AppleMap and the point value of the apple at a passed location. In addition, we have a spawn method which creates and adds apples (the number of apples on screen increases by one every five points up to a maximum of ten). This method calls the createNewApple method which randomly selects what type of apple to create using the required percentages.

Once the apple is selected, it is built using the AppleBuilder class and returned to the spawn method. Lastly, we have a draw method that steps through the AppleMap and calls each apples draw method that they inherited from GameObject.

Next, we created a new GameSound object using the Strategy design pattern to handle the game audio. This GameSound object has a constructor that selects which strategy to use (pre/post Lollipop or silent) depending on the host devices Android version and a passed boolean. If the boolean is true, the silent strategy is always chosen. Else, the constructor selects the pre or post Lollipop strategy. This strategy is then selected and loaded as a private strategy object. Each strategy implements the ISound interface which requires the strategies to have methods for playGetApple and playSnakeDeath. Each strategy has a constructor that sets up the sounds and loads the files in addition to these required methods. When a GameSound object is created, and the play methods are called. The GameSound object calls the selected strategies associated play methods.

To help fulfill the requirement of having a composite pattern for the snake in the game, we wrote an abstract class called Movable that inherited from GameObject. Movable has a concrete method called follow which allows the object to go to a specified point. This allows objects to follow each other as they can pass their location to another object who could then move there. The other method inside this class is the abstract move method. We wanted classes that extend this class to have a move method with implementations unique to themselves.

The class Head extends Movable and is used to represent the head of the snake. The head has an extra variable that is used to track its current direction of travel. Now, a possibly better way of doing this would have been to make Head into a state pattern. However, we ran out of time to do this, so instead, the head simply has a variable that controls the heading. Head also has a method called updateHeading, this is a form of encapsulation and is a very important method for two reasons. First, it updates the direction of travel. It does so with a switch statement that has been encapsulated in a private method. It also changes the direction the head graphic is facing. Since our goal was that every object would have only one graphic, we let the head rotate its graphic based on the direction the head is turning. This is much simpler than having four different head graphics and switching which one is currently being displayed. One other thing of note, the head implements the move method to move based on the direction the head is facing.

The class Body extends Movable and is used to represent a standard body segment of the snake. The class is pretty minimal so there is not much to explain here. The only thing of interest, that we decided on is that we did not want the body segment to have a unique move method. Thus, when implementing move, we had it simply print to the terminal that the body does not have an independent move method.

The class Snake also extends Movable and is used to represent the whole snake. This class is the composite part of our composite pattern. The composite pattern is built around the Movable class and thus snake can have several Movable components. The snake has a variable and an ArrayList that is of type Movable. Usually, the ArrayList would be populated with body objects. However, the head object or a full snake could be

put in there also if one wishes to have a snake follow another snake. Most of the methods in the snake class are there for encapsulation. Examples of this include the addBody, loseBody, reset, and detectDeath. All of these are self-explanatory and have ample documentation inside the source code. The detectDeath method has further encapsulation for each of the ways of dying. With the implementation of the bad apples, we now have three ways that the snake could die. It could leave the screen, it could eat its own tail, or it could eat enough bad apples that it died from starvation (negative points). Each of these ways of dying has its own method that is called by the general detectDeath method. The snake also has its own unique move method implementation. When the snake moves, it moves its head and anything that is in its body list. Finally, the snake has a method for checking apples. We can pass the snake the appleBasket which has a map of all the apples in play. We then check the map for an apple at the current head location using the Point object as the key. Please note, we saw a noticeable increase in performance when using a HashMap has opposed to the ArrayList.

Next, we want to make a note about the interaction between the points, the length of the body, and the number of apples on the screen. The game starts by spawning one apple at a time, for every five points the player has, another apple will spawn. So, at five points two apples will spawn at a time and at ten points there will be three apples on the screen. The snake's body will grow and shrink to match the number of points a player has. If a player has seven points, then they will have seven body segments. However, if they were to hit a bad apple, they would lose two points and have only five points total, the snake's body would be updated to reflect this and the snake would only have five body segments.

One change that we carried over from our Pong assignment was the ScreenInfo object. This class holds information on the screen. In this case, the number of blocks high and wide, the segment size, and the colors used for the screen. This object is instantiated in SnakeActivity and is passed into SnakeGame when the game starts. This class was created to encapsulate and abstract data. Instead of having to pass around individual variables, this allowed us to pass around a general-purpose object that holds all the required information for the screen.

Finally, the last change we made to the code was a general clean-up of the code. Adding further documentation to areas where we felt additional explanation was needed while reformatting large portions of the code so as to enforce a general standard and enhance readability.

Screenshot of Snake game running in the emulator