  
  
  
  
CS 319 - Object-Oriented Software Engineering  
Project Final Report  
  
  
Animal Uprising  
Group 3-A  
Bora Ecer

Ata Gün Öğün

Albjon Gjuzi

Tanay Akgül

04.11.2017

**1. Introduction:**

The final report includes implementation process, user guide, deviation from Design Report, what is left, and conclusion part. In the implementation process the implementations we have done will be explained. In the user guide the system will be explained briefly parallel to the Analysis and Design Report. In the deviation from Design Report, the changes from the Design Report of the project will be explained and in the “What is left” part the non implemented parts of the system will be given.

1. **Implementation:**

The implementation of the program started right immediately after the Design Report, because we had an idea of the classes that were needed and the relation between them. The most difficult part was trying to understand and learn how the 2D graphics work in Java.

Basically our implementation of GUI will resemble the MVC pattern. This can be understood from the class relations that we have in our Design Report. The Model, View and Controller are divided in subsystems there. The Model subsystem is based on the GameObject class which is then extended by every other object that our program will have such as Hero, Allies, Enemies and Castle. The View will be based on the GameEngine class which will create the main frame in which the game will be played. These two main components if we can call them like that, will then be connected to the Controller class which is GameManager, and as the name suggests this class will manage everything that happens in our application. Furthermore, we will give a detailed information about each component of the MVC pattern and each class of each subsystem.

**UIManagement Subsystem: (The View)**

* GameEngine Class: GameEngine class has already been implemented. The GameEngine class sets the JFrame and Canvas attributes and a method createView() that initializes the frame and the Canvas. It also has the getFrame() and getCanvas() methods to be able to use these attributes in other classes where GameEngine instance object will be created.

**GameControl Subsystem:** **(The Controller)**

* GameManager Class: GameManager is the façade class of the game, it provides connection between the UIManagement Subsystem and GameModel Subsystem as previously explained. GameManager is a Runnable class and an instance of this class starts working with the method start() is called. Start() method first calls init() which initializes the gameEngine, states of the game and sets the currentState which is GameState by default for now. Also, the GameManager has the game loop in it which keeps iterating until the end of the game and calls update() and render() methods of currentState of the game. The loop is determinded by an attribute called running which is set to true when the game is played, and will be set to false when the game is over.
* ImageManager Class: ImageManager class is reading the images from the Resources file and setting them as Static BufferedImages. For the moment we have implemented only the main objects of the game that are the hero, the castle, two allies and one enemy.
* InputManager Class: InputManager class is determining which of the keys from the keyboard can be used to play the game and it implements KeyListener class in order to tell the program which button is pressed and released. We are using A-D to move the hero left and right and the numbers to summon allies.
* States Class: States class is an abstract class that will be later implemented from the three other state classes. It contains a void method called setState() that will be used to change the state of the game throughout the execution time of the program, a static method called getState() that simply return the current state of the game which is marked by a variable called currentState. It also has void update() and void render(Graphics graphics) abstract methods that will be implemented from the subclasses mentioned.
* GameState Class: GameState class will extend the States class and therefore it has to implement the two abstract methods that States class had. The update() method updates the coordinates of each object in the screen or it starts the cool down by calling startCooldown() method for the object that is going out from the screen.

**GameModel Subsystem:** **(The Model)**

* ObjectManager Class: ObjectManager class is nothing more than an ArrayList of GameObjects. It keeps a count for the number of objects and a method called addObject( GameObject x) which adds the object x in the ArrayList and increments the count for the objects. We also need to decrease the object count so we have a method called decreaseObjectCount() which will occur when one of the allies or enemies dies.
* CharacterObject Class: CharacterObject class is an abstract class that has the main attributes needed for every object in the game such as: movementSpeed, maxHealth, attackDamage. The default width and height are also defined in this class as final attributes and cannot be later modified from the subclasses of this class. Other than these, this class has all the setters and getters for each of the attributes mentioned.
* CastleObject: CastleObject class is the only object class that does not extend CharacterObject class but it extends directly the GameObject class, because it has different attributes and methods from the soldiers. This class has an update() method that will update the health of the enemies castle and will be used to summon enemy soldiers from time to time.
* All other object classes simply implement the interfaces of their current type of enemy or ally and extend the character object. Then they have their own different attributes such as health points and attack points.

1. **User Guide**

Animal Uprising is a strategy/adventure game in which the player will be able to control a hero rather than having a castle. During the game, in order to destroy the enemy castle, the player will have to carve a path to the enemy castle by defeating the enemy soldiers. The player can use the keyboard buttons A and D to move left and right respectively.

The player can summon ally soldiers in order to have a bigger striking force towards enemy soldiers. At the moment the summoning can be made from the keyboard numbers, but we might change it to a mouse click on the ally picture that will be displayed at the bottom of the screen while the game is being played.

When the user strikes, he will have to wait for a certain amount of time until he can unleash another attack on the enemy soldier. The same thing is valuable for summoning the ally. If one ally is called, the player has to wait for a certain amount of time called cooldown time until he can summon another ally.

Once the player and allied soldiers reach to the enemy castle, they can attack it to tear it down. The player hero will be able to cast both defensive and offensive skills which either help the allies or harm the enemies. These functionalities are not implemented yet, but we are on the right track. The game has five different difficulty levels and in each level, the player has to face more and harder types of enemy soldiers.

Upon finishing each level, and destroying all enemy soldiers the player will gain coin which will allow the player to buy items, and upgrade skills or soldiers. The player will be able to pause the game and continue whenever they want to.

The user can also go to the shop menu while the game is paused, and he can buy upgrades for the skills of the hero or for the allies. Also, the player will be able to replay completed levels to gain more coins. There will also be an option to reset the whole game from scratch in order to set a higher score.

**4. Deviation from Design Report**

There were not much deviations from the Design Report that we have uploaded, but as every project in the world we had to change some things in order to make a better application and form a better hierarchy of the classes.

We have created a package of classes to GameControl that consists of one abstract class called States and three subclasses called GameState,MenuState,PauseState that extend the abstract class and are connected to GameManager. Those four classes are explained in detail in the implementation part since we did not have them before.

We have also added a class called ImageManger that controls all the images of the program and loads them as BufferedImages. The code in the methods start(), run() and stop () of the class GameManager will be used too many times in our program as it is essential to run() when needed, and stop() when needed.

We also have removed the StationaryObject class that we had thought as the parent of the CastleObject class, but since it would have the same attributes as its parent class, then we thought to move CastleObject class one level up in the hierarchy and get rid of its “unnecessary parent”.

Besides these changes, we have also changed a lot of attributes for most of the classes. Actually we have added some that we did not think of during the time we were preparing our design report.

1. **What is left?**

We are focusing on the game play for now, but the Main Menu and the Pause Game states classes will be also be implemented with their attributes and methods. They will provide different views from the GameState class and different options for the user. The ShopMenu is also needed, and we have not implemented it yet because we are still deciding on what will currently be inside the shop.

Also, the main mechanic of the game is still missing, which is the combat. We are going to implement the combat system. Also, the skills and skill animations for hero object.

The amount of money the user gains for each dead enemy or level won are still to be determined. To be determined are the prices of the objects in shop too as soon as we finish defining what they will be.

We will also replace the images of the objects with animations to make the game seem more real, and the controls for the user might change from keyboard to keyboard + mouse.

The **settings** class will be also implemented in order to play sounds and music, or even change the controls according to user’s preferences.

1. **Conclusion**

The purpose of the final report is to document the implementation process, deviation from design report and the incomplete parts. A user guide was also provided. In implementation part the Model, View and Controller are divided in subsystems. The Model subsystem is based on the GameObject class. The View will be based on the GameEngine class and controller class which is GameManager. In the deviation from design report the changes what we have done is explained. Finally in what is left part the settings will be implemented in order to play sounds and music, or even change the controls according to user’s preferences.

We chose this project in order to learn more about 2D programming in Java and have a closer look at complex systems like the one we are making. By the time we finish our game, we will have learnt a lot of things about Object Oriented Software engineering. The idea was also to understand how to cooperate in a team-work, share ideas and choose together what is the best solution for a given problem that can occur. The implementation with the MVC is showing us its pros and its cons with the other types of implementations that exist nowadays, and in overall the project was the best chance we had to improve our OOP skills. We have all benefited from this project.