DRAGON THE GAME

GROUP NUMBER:

39

GROUP MEMBERS

Dilrajveer Singh (a1825200)

Amell Julardzija (a1850474)

Swapnil Kumar (a1848855)

USE CASE DESCRIPTION:

Dragon is a game with a twist on the classic snake game. The game is designed to provide users with a fun and engaging experience that challenges their hand-eye coordination and reaction time. The purpose of the game is to control a dragon that moves around the game board, collecting dragon eggs and avoiding fake eggs.

The game's main goal is to score as many points as possible by collecting green and blue dragon eggs, with each green egg adding one point and slowing down the dragon, while each blue egg adds two points. However, there are fake eggs to avoid, with the red eggs causing the dragon to lose two points and the orange eggs causing it to lose one point and move faster. The current score will be displayed at the top of screen whilst playing.

The game is designed to be accessible to users of all skill levels, with easy-to-use controls and simple gameplay mechanics. The game is ideal for users who enjoy casual gaming and want to take a break from their daily routine. The user may pause the game at any time they please and return later. In the main menu, the user can play, adjust the sound, read the rules or exit from the game.

LIST OF POTENTIAL CLASSES:

A list of potential classes include:

- Main Menu
- Player
- Points
- Border Wall
- Eggs
- Dragon
- Dragon egg
- Blue egg
- Green egg
- Fake egg
- Red egg

- Orange egg

CLASS DESCRIPTION:

Game/ Main menu: Player enters into the game where they are introduced to the ‘rules’ and have the option to control the ‘sound/music’ and also have a ‘quit’ option.

Player (Aggregate): The player class will consist the functionality of controlling the game, this includes the 'controls’ such as (W,A,S,D) to move the ‘dragon’. It is also going to be responsible to store the players name and their score.

Points (Aggregate): Stores current score, highest score, update and output highest score

Border Wall: This class will consist of (x,y) coordinates of the border, and will be used to make the grid on which the game will be played on.

Dragon: The dragon class will be the ‘moving object’ in the game. This object will be controlled by the player in order to save the ‘dragon eggs’ and to avoid the ‘fake dragon eggs’. Furthermore, as time goes on the speed of the dragon goes up as well.

Eggs (Abstract): The Eggs class is going to provide the functionality of the ‘Eggs’, when the ‘dragon’ that is being controlled by the player, collides with the ‘eggs’. The score of the player will be updated according to the ‘eggs function’. Furthermore, it is going to have functionality such as introducing ‘eggs ‘at a random ‘(x,y)’ coordinate, and changing its location every time it collides with the dragon.

Dragon egg (Inheritance): Player controls Dragon to pick up ‘dragon eggs’. While the player avoids driving into boundary walls and itself. When a ‘dragon egg’ is rescued, the players score will updated according to the eggs color. Furthermore, it is going to have the functionality such as introducing ‘dragon egg’ on a random ‘(x,y)’ coordinate, and changing its location when it collided with the dragon.

Blue egg (Inheritance):

When the dragon collides with a blue egg the score of the player is increased by 2 point.

Green egg (Inheritance):

When the dragon collides with a green egg the score of the player is increased by 1 point and the speed of the dragon will decrease in order to help the player control the dragon more carefully.

Fake Dragon egg (Inheritance): Similar to the ‘dragon egg’ class, the player will control the dragon to dodge the fake egg rather than collecting it. If the dragon collides with a fake egg, the players score will be deducted by -1 point. Furthermore, it is going to have the functionality such as introducing ‘fake dragon egg’ on a random ‘(x,y)’ coordinate, and changing its location when it collided with the dragon.

Red egg (Inheritance):

When the dragon collides with a red egg the score of the player is decreased by 2 point.

Orange egg (Inheritance):

When the dragon collides with an orange egg the score of the player is decreased by 1 point and the speed of the dragon will increase, making it harder for the player to control the dragon more carefully.

POTENTIAL DATA AND FUNCTION MEMBERS:

Game/ Main menu

State variables:

- Menu

- Rules
- Sound

Behaviours:

- Menu() // constructor (start function)
- display_menu()
- get_rules()
- set_rules()
- Sound()
- ~Menu() // destructor (end function)

Player

State variables:

- Name: String
- Score: Integer
- Controls: Bool

Behaviours:

- Player() // constructor (start function)
- get_name()
- set_name()
- current_score()
- final_score()
- control_choice()
- ~Player()// destructor (end function)

Points

State variables:

- Total Score (Current)
- High Score (Current)
- New Score (High score) // stores values such as eggs, dragon eggs, Fake dragon eggs

Behaviours:

- Points() // constructor (start function)

- get_total_current _score()
- update_total_current_score()
- get_high_current_score()
- set_high_current_score()
- write_txt_score_file()
- read_txt_score_file()
- ~Points() // destructor (end function)

Border Wall

State variables:

- Wall Coordiantes: int
- Dragon Coordinates: int

Behaviours:

- Border() // constructor (start function)
- Wall_x_coord
- Wall_y_coord
- new_dragon_x_coord
- new_dragon_y_coord
- ~Border()// destructor (end function)

Dragon

State variables:

- score
- dragon_length
- num_d_eggs // number of dragon eggs saved
- xD_coordinate [] // random x coordinate of Dragon (On a 10 X 10 grid)
- yD_coordinate [] // random y coordinate of Dragon (On a 10 X 10 grid)

Behaviors:

- Dragon() // constructor (start function)
- dragon_move()
- new_xD_coordinate //updates coordinates of Dragon (On a 10 X 10 grid)
- new_yD_coordinate //updates coordinates of Dragon (On a 10 X 10 grid)

- add_egg()
- dragon_length()
- speed_up()
- update_score
- pause_game()
- game_over()
- ~Dragon()// destructor (end function)

Eggs

State Variables

- Score: integer
- Dragon
- xE_coordinate [] // random x coordinate of Eggs (On a 10 X 10 grid)
- yE_coordinate [] // random y coordinate of Eggs (On a 10 X 10 grid)

Behaviours

- Eggs () // constructor (start function)
- add_score()
- dragon_move()
- add_Eggs()
- pause_game()
- new_xE_coordinate //updates coordinates of Eggs (On a 10 X 10 grid)
- new_yE_coordinate //updates coordinates of Eggs(On a 10 X 10 grid)
- ~Eggs() // destructor (end function)

Dragon egg

State Variables

- Score: integer
- Dragon
- Eggs
- xE_coordinate [] // random x coordinate of Eggs (On a 10 X 10 grid)
- yE_coordinate [] // random y coordinate of Eggs (On a 10 X 10 grid)
- Dragon egg
- xDE_coordinate [] // random x coordinate of Dragon egg (On a 10 X 10 grid)

- yDE_coordinate [] // random y coordinate of Dragon egg (On a 10 X 10 grid)

Behaviours

- DragonEGG() // constructor (start function)
- add_score()
- dragon_move()
- add_eggs()
- pause_game()
- new_xF_coordinate //updates coordinates of Eggs (On a 10 X 10 grid)
- new_yF_coordinate //updates coordinates of Eggs (On a 10 X 10 grid)
- new_xDE_coordinate //updates coordinates of Dragon egg (On a 10 X 10 grid)
- new_yDE_coordinate //updates coordinates of Dragon egg (On a 10 X 10 grid)
- ~DragonEGG() // destructor (end function)

Blue egg

State Variables

- Score: integer
- Dragon
- Dragon egg
- Blue egg
- Plus 2 point
- xDE_coordinate [] // random x coordinate of Dragon egg (On a 10 X 10 grid)
- yDE_coordinate [] // random y coordinate of Dragon egg (On a 10 X 10 grid)
- xBE_coordinate [] // random x coordinate of blue egg (On a 10 X 10 grid)
- yBE_coordinate [] // random y coordinate of blue egg (On a 10 X 10 grid)

Behaviours

- BlueEGG() // constructor (start function)
- add_score()
- dragon_move()
- add_eggs()
- pause_game()
- new_xDE_coordinate //updates coordinates of dragon Eggs (On a 10 X 10 grid)
- new_yDE_coordinate //updates coordinates of dragon Eggs (On a 10 X 10 grid)

- new_xBE_coordinate //updates coordinates of blue egg (On a 10 X 10 grid)
- new_yBE_coordinate //updates coordinates of blue egg (On a 10 X 10 grid)
- plus_two point
- ~BlueEGG() // destructor (end function)

Green egg

State Variables

- Score: integer
- Dragon
- Dragon egg
- Green egg
- Plus 1 point
- Decrease speed
- xDE_coordinate [] // random x coordinate of Dragon egg (On a 10 X 10 grid)
- yDE_coordinate [] // random y coordinate of Dragon egg (On a 10 X 10 grid)
- xGE_coordinate [] // random x coordinate of green egg (On a 10 X 10 grid)
- yGE_coordinate [] // random y coordinate of green egg (On a 10 X 10 grid)

Behaviours

- GreenEGG() // constructor (start function)
- add_score()
- dragon_move()
- add_eggs()
- pause_game()
- new_xDE_coordinate //updates coordinates of dragon Eggs (On a 10 X 10 grid)
- new_yDE_coordinate //updates coordinates of dragon Eggs (On a 10 X 10 grid)
- new_xGE_coordinate //updates coordinates of green egg (On a 10 X 10 grid)
- new_yGE_coordinate //updates coordinates of green egg (On a 10 X 10 grid)
- plus_one_point
- minus_speed
- ~GreenEGG() // destructor (end function)

Fake Dragon egg

State Variables

- Score: integer
- Dragon
- Eggs
- xE_coordinate [] // random x coordinate of Eggs (On a 10 X 10 grid)
- yE_coordinate [] // random y coordinate of Eggs (On a 10 X 10 grid)
- Dragon egg
- xDE_coordinate [] // random x coordinate of Dragon egg (On a 10 X 10 grid)
- yDE_coordinate [] // random y coordinate of Dragon egg (On a 10 X 10 grid)
- Fake Dragon egg
- xFDE_coordinate [] // random x coordinate of Fake Dragon egg (On a 10 X 10 grid)
- yFDE_coordinate [] // random y coordinate of Fake Dragon egg (On a 10 X 10 grid)

Behaviours

- FakeDragonEGG() // constructor (start function)
- add_score()
- dragon_move()
- add_eggs()
- pause_game()
- new_xF_coordinate //updates coordinates of Eggs (On a 10 X 10 grid)
- new_yF_coordinate //updates coordinates of Eggs (On a 10 X 10 grid)
- new_xDE_coordinate //updates coordinates of Dragon egg (On a 10 X 10 grid)
- new_yDE_coordinate //updates coordinates of Dragon egg (On a 10 X 10 grid)
- new_xFDE_coordinate //updates coordinates of Fake Dragon egg (On a 10 X 10 grid)
- new_yFDE_coordinate //updates coordinates of Fake Dragon egg (On a 10 X 10 grid)
- ~FakeDragonEGG() // destructor (end function)

Red egg

State Variables

- Score: integer

- Dragon
- Red egg
- Fake Dragon egg
- Minus 2 point
- xFDE_coordinate [] // random x coordinate of fake Dragon egg (On a 10 X 10 grid)
- yFDE_coordinate [] // random y coordinate of fake Dragon egg (On a 10 X 10 grid)
- xRE_coordinate [] // random x coordinate of red egg (On a 10 X 10 grid)
- yRE_coordinate [] // random y coordinate of red egg (On a 10 X 10 grid)

Behaviours

- RedEGG() // constructor (start function)
- add_score()
- dragon_move()
- add_eggs()
- pause_game()
- new_xFDE_coordinate //updates coordinates of fake dragon Eggs (On a 10 X 10 grid)
- new_yFDE_coordinate //updates coordinates of fake dragon Eggs (On a 10 X 10 grid)
- new_xRE_coordinate //updates coordinates of red egg (On a 10 X 10 grid)
- new_yRE_coordinate //updates coordinates of red egg (On a 10 X 10 grid)
- minus_two point
- ~RedEGG() // destructor (end function)

Orange egg

State Variables

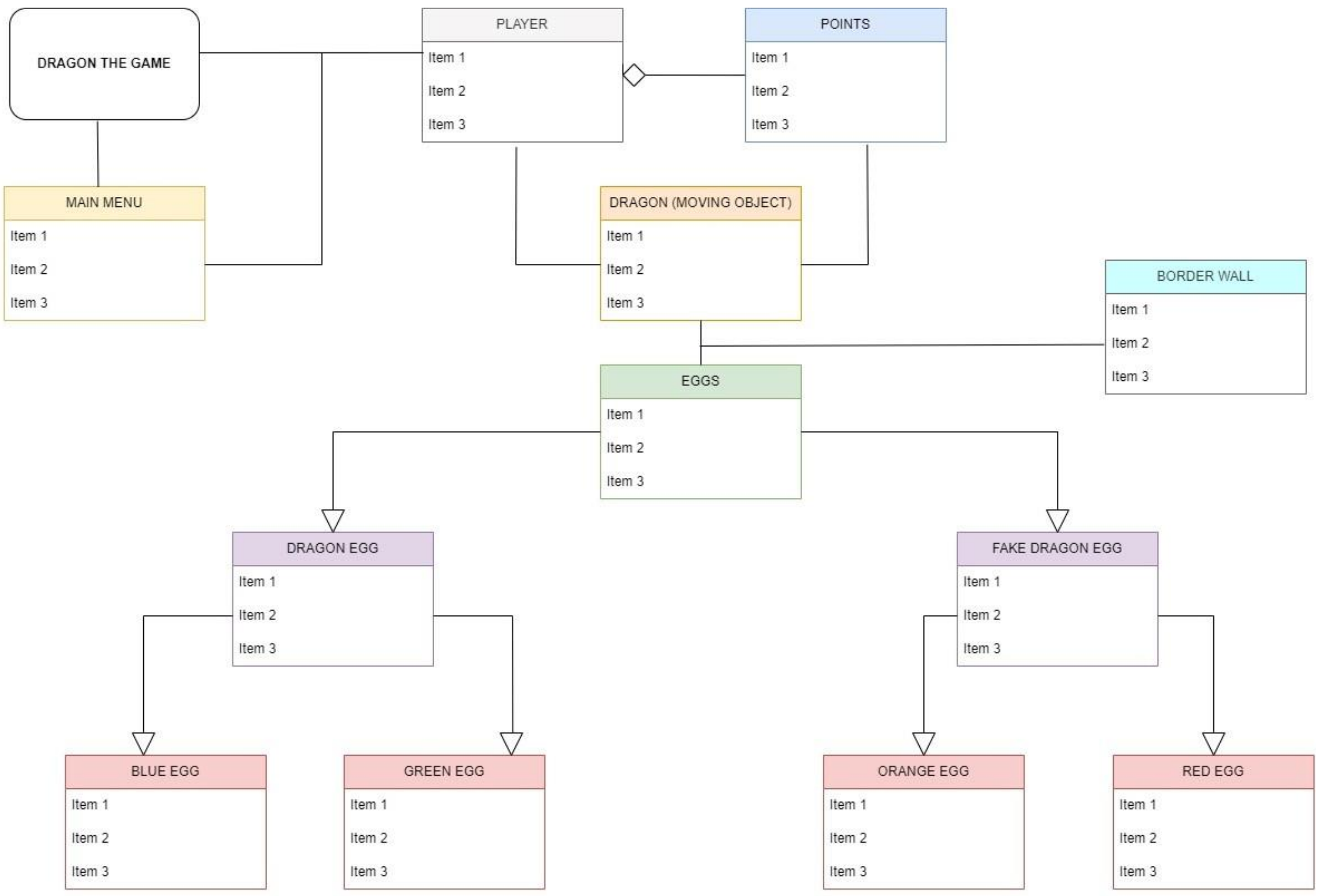
- Score: integer
- Dragon
- Dragon egg
- Orange egg
- Plus 1 point
- Decrease speed
- xFDE_coordinate [] // random x coordinate of fake Dragon egg (On a 10 X 10 grid)
- yFDE_coordinate [] // random y coordinate of fake Dragon egg (On a 10 X 10 grid)
- xOE_coordinate [] // random x coordinate of orange egg (On a 10 X 10 grid)

- yOE_coordinate [] // random y coordinate of orange egg (On a 10 X 10 grid)

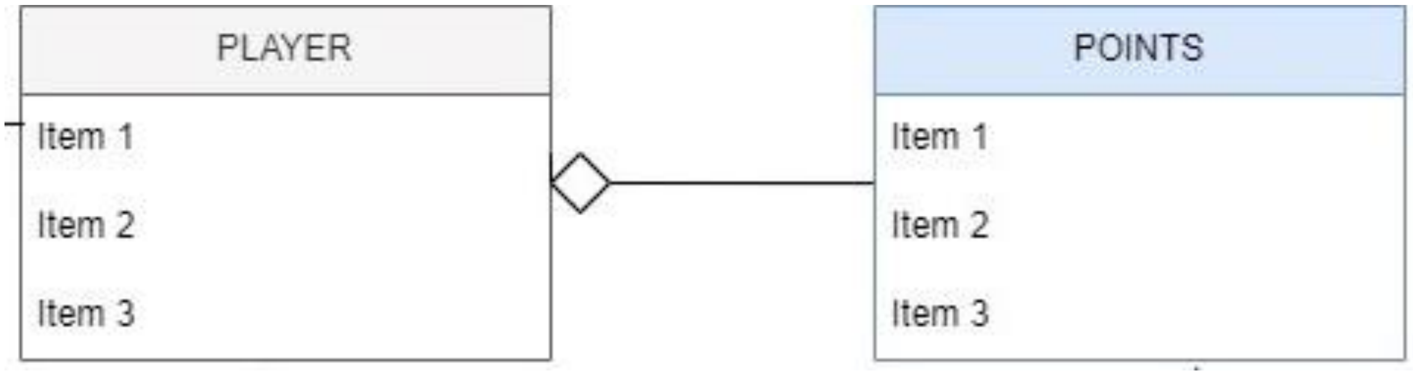
Behaviours

- OrangeEGG() // constructor (start function)
- add_score()
- dragon_move()
- add_eggs()
- pause_game()
- new_xFDE_coordinate //updates coordinates of fake dragon Eggs (On a 10 X 10 grid)
- new_yFDE_coordinate //updates coordinates of fake dragon Eggs (On a 10 X 10 grid)
- new_xOE_coordinate //updates coordinates of orange egg (On a 10 X 10 grid)
- new_yOE_coordinate //updates coordinates of orange egg (On a 10 X 10 grid)
- minus_one_point
- increase_speed
- ~OrangeEGG() // destructor (end function)

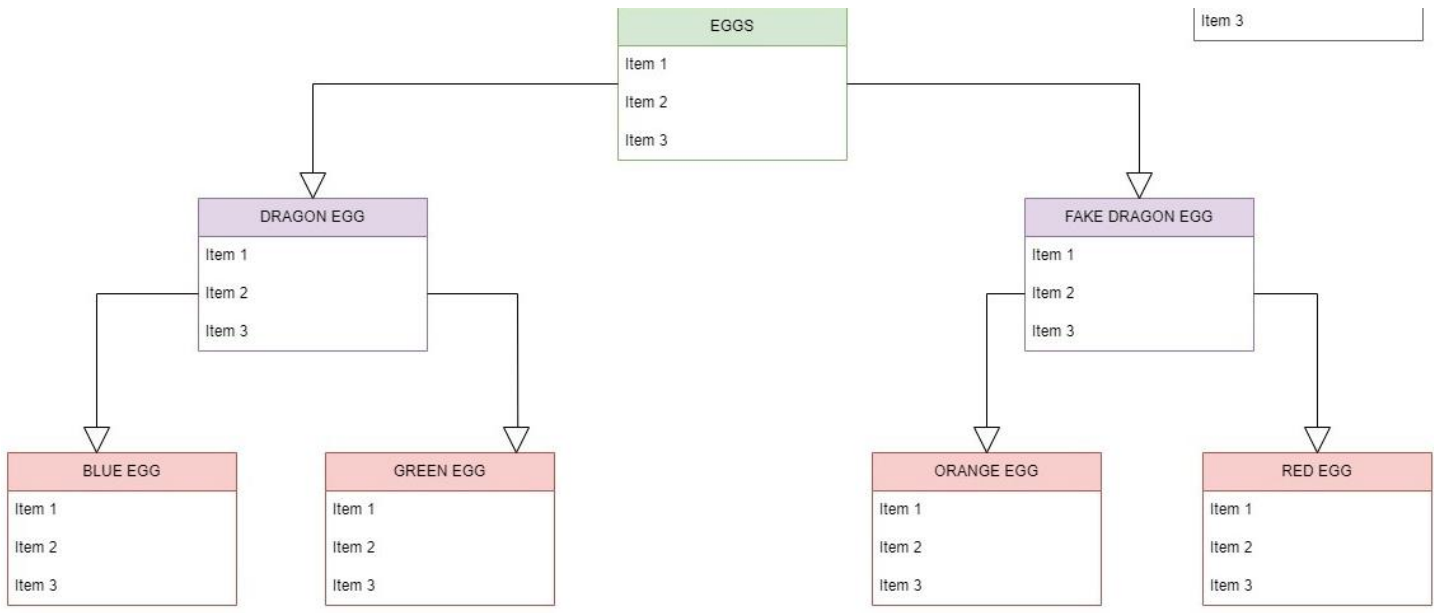
RELATIONSHIPS BETWEEN CLASSES:



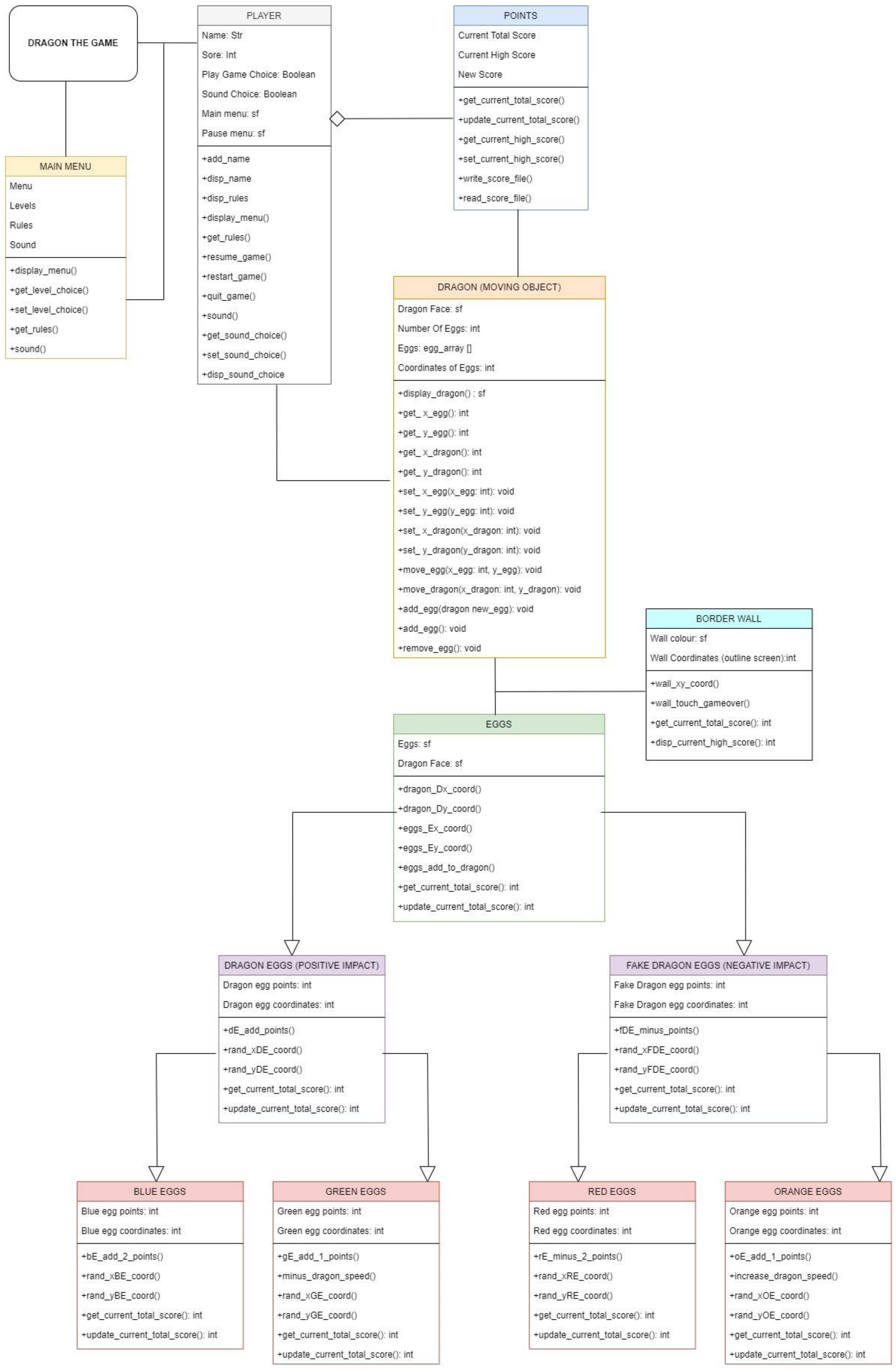
Use of Aggregation:



Use of 3 Level inheritance:


Item 3
EGGS
Item 1
Item 2
Item 3
DRAGON EGG
Item 1
Item 2
Item 3
FAKE DRAGON EGG
Item 1
Item 2
Item 3
BLUE EGG
Item 1
Item 2
Item 3
GREEN EGG
Item 1
Item 2
Item 3
ORANGE EGG
Item 1
Item 2
Item 3
RED EGG
Item 1
Item 2
Item 3

RELATIONSHIPS BETWEEN CLASSES:



PROJECT TASK LIST AND TIMELINE:

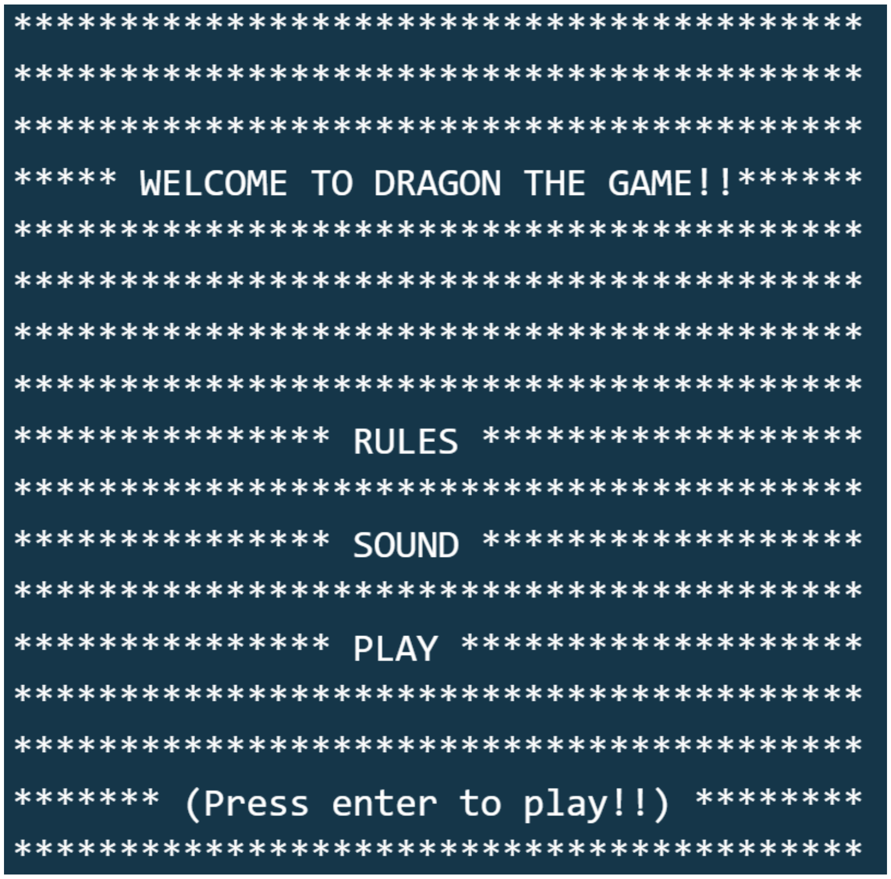
	Day:	Details:	Due:	Objectives:	Completed:	Upcoming Plans:
Week 8	Friday (5/05)	-First meet up		-Make group repository -Begin project plan -Create timeline	-Group repository -Meet up times (Every Tuesday)	-Completing timeline - Record a rough idea of what functions, members and states will be used
Week 9	Tuesday (09/05) Friday (12/05)	-meet up to discuss the progress on the plan and who is doing what. -Allocate who will be coding each section of the game	Project Plan	-Project plan use case description and timeline (Amell) - Complete data functions and member classes and relationship between classes section (Dilrajveer) - Complete debugging and unit testing section (Swapnil)	- Project plan	-Refine project plan fonts, font size, grammar, etc. - Try to finish coding abstract class and inherited classes, dragon-egg, fake-dragon egg and their respective children, over the weekend.
Week 10	Tuesday (09/05) Friday (19/05)	-Meet up to see if anyone needs help on their tasks	Project version 1	- Finish rough version of dragon by practical -Ensure that the game can run without errors and bugs. -Finish implementing our unit testing and debugging methods		-Using tutors' feedback from project version 1, implement changes and refine code further.
Week 11	Tuesday (23/05)		Project Final	-Finish game fully refined without any errors		

USER INTERACTION DESCRIPTION:

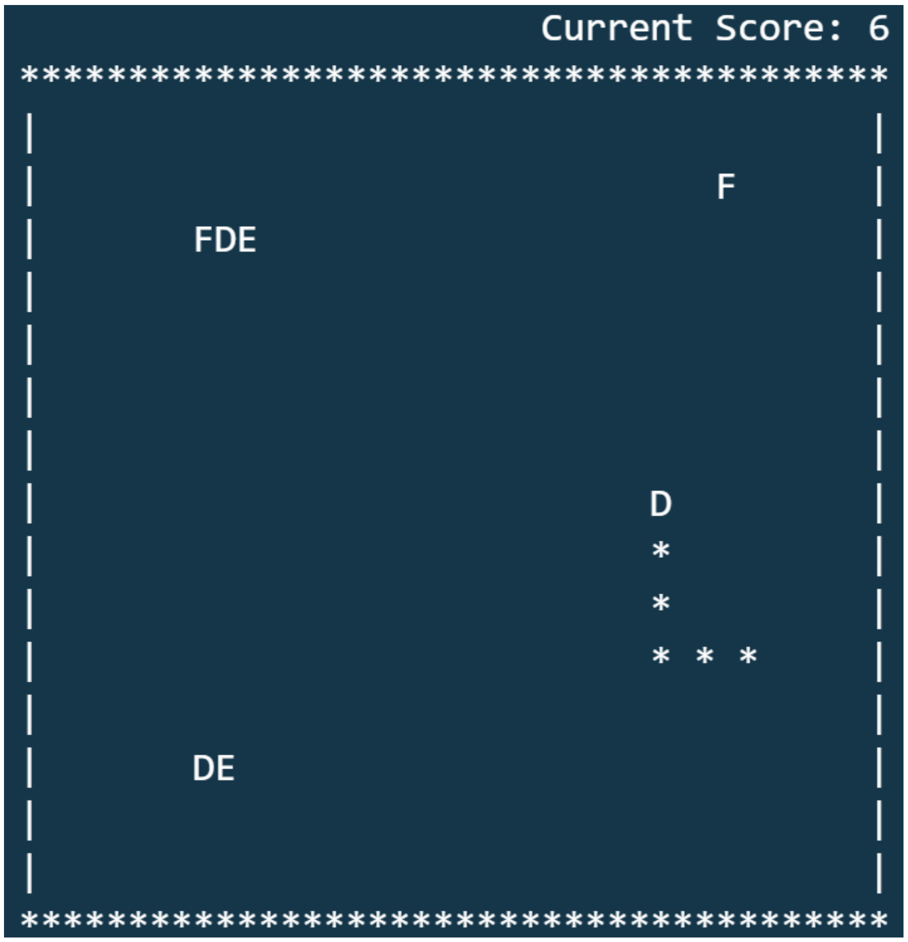
Our game opens into a main menu with four options: **Play**, **Sound**, **Exit**, **Rules**. To play the user can click on **Play** or press the '**enter**' or '**return**' (on Apple products) keys. If the user decides to play, the rules screen will pop up just explaining the rules in case the user does not know them. The player can just press **enter** or **return** to start playing. The game will start, and dragon will move in response to 4 keys: **W** will move dragon up, **D** will move the dragon right, **A** will move the dragon left and **S** will move the dragon down. If the user needs to check the rules again, they can use the pause button **P.** To resume the game from the pause menu, the user can click on the **Resume** option. If the user wants to exit to the menu from the pause screen, they can press the **Exit to menu** option. If they want to exit the entire game, they can click on the **Quit** option.

Built with user friendlessness and easy navigation in mind, this game will have a Home Screen with three main buttons. Those buttons include PLAY GAME, EXIT and SCORES. There will also be an option for the user to mute or unmute the game, depending on their choices. If the user decides to play the game, they will be notified to switch to their keyboard, particularly their arrow keys which will control the snake. It should be noted that the user will have an option of choosing another level if they succeed in getting past the default level. The program's graphics will be quite intuitive, with the game being built in a similar fashion to the snake game. If the user decides to click "scores," they will be shown a list of previous high score holders. The exit button should promptly close the game. Note that the progress will not be saved if this is done in the middle of the game.

GAME PROTOTYPE:



```
****************************************
****************************************
****************************************
***** WELCOME TO DRAGON THE GAME!!******
****************************************
****************************************
****************************************
****************************************
*************** RULES ******************
****************************************
*************** SOUND ******************
****************************************
*************** PLAY *******************
****************************************
****************************************
******* (Press enter to play!!) ********
****************************************
```



```
                        Current Score: 6
****************************************
|                                      |
|                              F       |
|      FDE                             |
|                                      |
|                                      |
|                                      |
|                                      |
|                            D         |
|                            *         |
|                            *         |
|                            * * *     |
|                                      |
|      DE                              |
|                                      |
|                                      |
****************************************
```

UNIT TESTING:

To ensure minimal both manual and automated methods will be used during this project. Manual and automated methods will be used for unit testing, whereas debugging will be reserved for manual methods.

An example of a good manual unit test would be to test user input to the game using the keyboard presses. Keyboard presses will include the 'UP,' 'DOWN', 'LEFT' and 'RIGHT' arrows for the dragon's movement, and keys for menu option selections such as P, R, M, Esc to navigate to pause the game, resume game, return to menu, and quit the game respectively

We plan on testing incrementally. As of right now, we are testing the above functions as they are quite crucial. As the project progresses, increasingly complex tests will help us build this game.

We plan to add all manual test cases to a text file, along with detailed descriptions of the tests just so that they are easy to read. Some test cases that we have come up in the past few days are as follows -

1. SFML Base test - The first test includes running the SFML graphics and windows library to make sure that the library is working. This is a simple test done with the RenderWIndow command and in the best-case scenario a window should pop up.

2. SFML Image testing - The program will use certain images for the snake, the backgrounds and the fruits which need to be imported from the project folder. These images should show up on the SFML display when tested.

3. Switch test - This part will include testing the keyboard inputs. The user will control the game using his arrow keys. Such a program will be written with a simple switch statement and should not yield any errors.

4. Inheritance testing - This is one of the most crucial tests so far. The game will be built in multiple files, consisting of header files, programs and a make File. The game should be able to run without any linking or symbol errors.

5. Boundary testing - The game will include a feature in which the snake will lose health if it touches the boundary. The boundary will be built in SFML shapes and should be built such that the snake cannot go through it and would eventually die after hitting the boundary a few times.

6. Data storage testing - There will be a segment of code in the program built for storing location and data of the SNAKE. This is crucial since the entire game score will be calculated depending on how many fruits the snake has eaten, and its previous

locations will show whether it loses health or not. The program should be able to retrieve previous data including starting location without fail.

7. Score testing - The program should calculate game scores without errors. The scores will include increments and decrements, both of which should run properly as they are crucial to the game.

Important function tests -
The program has some crucial functions that will need to be tested to ensure that the game works fine.
Some of these functions are -

- add_score
- dragon_move
- add_dragon_egg
- add_false_egg

Automated Unit Testing -

The purpose of automated unit testing is to save time, test faster, have a comprehensive range of test cases and to prevent input mistakes. The plan for automated testing will be to set a range of inputs, in the form of text files, to run through the previous sections or combined sections of the program. This plan will focus on manual testing of a program section being developed. Through the continued implementation of this automated testing plan, the final program may be run with all the test file histories.

Since we are building this game using just SFML, most of the automated testing frameworks are not available to us, since they usually work with game engines. However, we have found a few frameworks that we think might be suitable for our program. UnitTest++ is an automated testing framework which uses standard c++. Since it is quite lightweight and easy to set up, this framework would make testing complex cases a lot easier.

Automated testing would not be limited to unit test++. We also plan on testing using test files. These text files would contain a wide range of inputs for the program, such as key controls. After the completion of our game, we plan on using these inputs and running them through combined sections of the code first and then through the entire program.

In case a section of code or merged sections of code fail unit testing, a debugging process will begin.

DEBUGGING:

A debugger is a method used to identify bugs which cause errors within the program. The debugger would highlight the locations of the errors in the program and suggest a cause of the error and method to fix the error. Once the program error is identified, modifications will be made to address the error's cause. To determine the sections of the code that need work, print statements after specified lines of code may be performed to determine where the program breaks or issues regarding programing logic. To check if modifications of towards the program are successful, the program will be run again with test files.
The project team may consider logging sources of compiling errors and methods to resolve these errors to help resolve future programing errors like those previously identified.
Practices to reduce the severity of compiling errors include:
· Keeping the variable names precise and variable styles consistent
· Adding comments to each line or main sections of the code to describe what the code section is performing
· Regression testing:
. Work on the code one step at a time
. Breaking the project idea into sections to make individual short codes for each part which, after testing, are to be run and tested again with other sections of code
. Run and test the codes every time a new change is made within the code