Design Manual

CSCI 205 Final Project Team 3

Introduction

The creative snake games system utilized the networking for the multiplayer games and the MVC design principle for the basic snake game. Here, the snake pane is separated into three parts: model, view, and controller. Each part equally contributes to a fully functional snake game. The multiplayer game option is implemented via the JavaFX API and Java networking and server socket API, which can hold up to 4 players in a server room. The server is hosted on the player's computer, which will have an IP and a port.

We implemented the multiplayer networking so that a player can either join a game hosted by other players via entering host IP and port or host a new game by entering the port and number of players. These are achieved via the Java server socket.

Completed user stories

- As a player, I want to be able to control my character with keyboard inputs
- As a player, I want to see the game board
- As a developer, I want to write code in MVC pattern
- As a player, I want a feature where I will die when I touch my own snake body
- As a player, I expect a feature that running into the wall will end the game
- I expect that eating food will increase my length and score
- As a developer, I want to create a presentation that walks through the creation of this project
- As a user, I want a realistic-looking snake movement
- As a player, I want to see my snake

Incomplete user stories

- As a user, I want to be able to see my score at the bottom of the screen
- As a player, I would like the High score to be saved.
- As a player, when I die I want to have a final score
- As a player, I expect that speed and turn radius can be changed for the room to increase or decrease the difficulty of the game.
- As a player, I expect playing with other players and cutting them off will increase my score

Object-Oriented Design Explanation

Our object-oriented design of the snake game is based on three major parts, which are described below. The order of our program is from high-level design (where the program first calls) to low-level design (where all game objects are implemented)

1. [**High-level**] JavaFX MVC design: Model, View, Controller

2. [**Medium-level**] Multiplayer network: Created from the model, handles the server room and the game environment. This includes all game objects.

3. [**Low-level**] Game objects: All game objects are circles. The snake, items, and walls are extended from assets, which are extended from JavaFX circles.

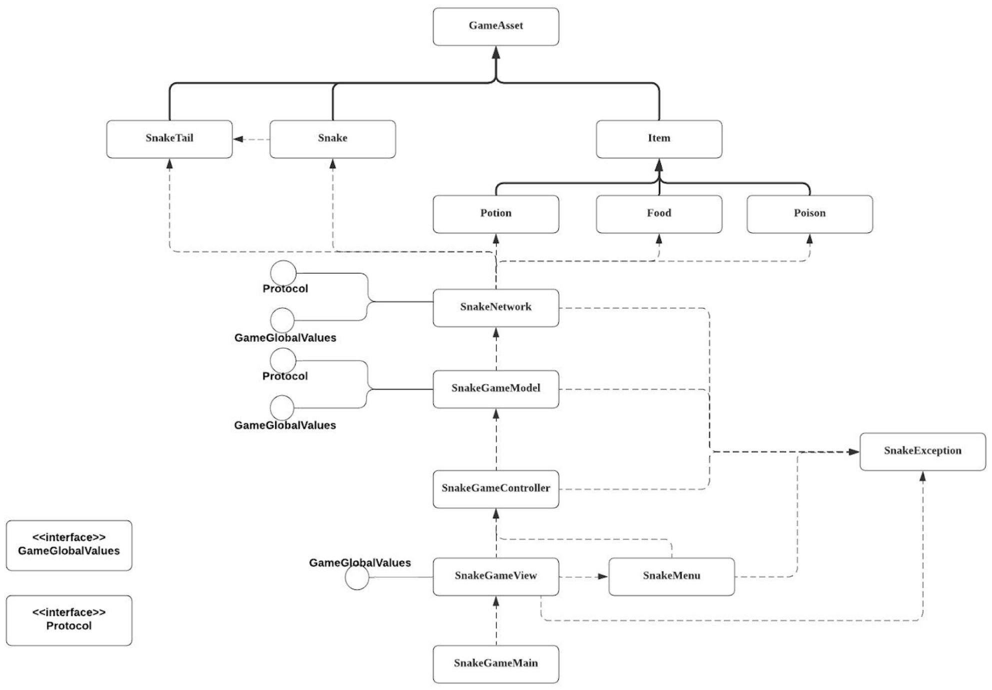


Figure*: the simple basic uml diagram showing a clear overall OOAD structure*

View: The snake game view creates a snake menu for the game and a controller. The view will reference the game controller to update the view of the snake, which will be reflected on the screen. The view is also responsible for initializing the snakes, items, and walls in the game, which is implemented by calling the controller then calling the model then calling the snake game network. The view will be updated when it receives orders from the controller. The snake view is the most basic class and will call all other classes when the program begins. The view is extended from the JavaFX application.

Class: SnakeGameView	
Responsibilities	**Collaborators**
● Displaying the state of the game to the user ● Allowing the user to input keystrokes to control his character ● Extends application and uses an animation timer to update the visual game based on information contained in the controller. ● It will have Pane as the root, and an ArrayList of GameAssets which will represent the sections of the snake. This will also take the player's input if the button is pushed it will update the controller so the model can reflect the change.	● SnakeGameController

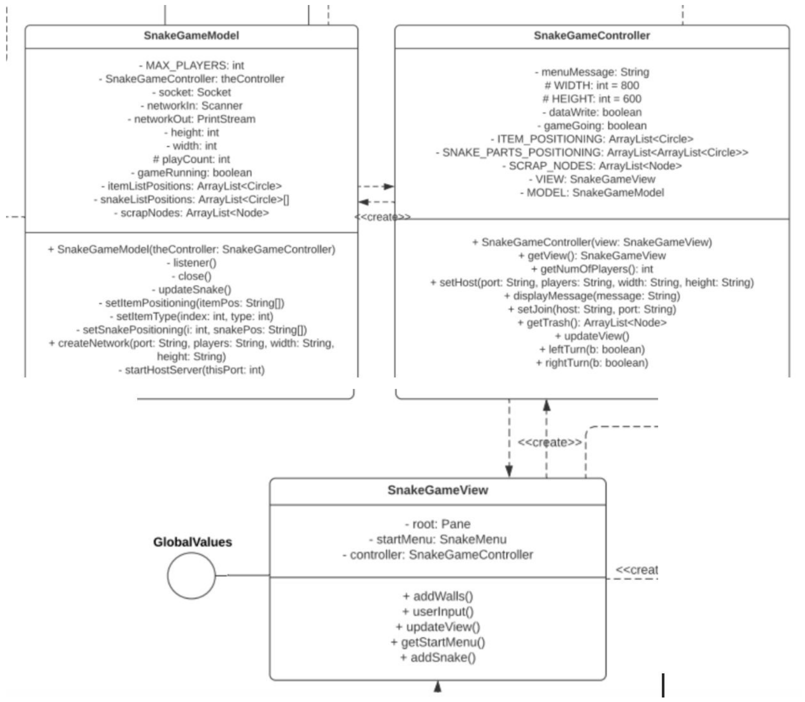
Controller: The snake game controller creates the snake model. The controller is responsible for displaying messages, getting items and the positioning of snake parts, and updating the view for the snake game view. The controller controls the height and width of the window, which is implemented by setting a final constant, which will call the snake game view for adjusting the window pane. The controller is also responsible for setting the host and for calls to the snake game model for creating the network.

Class: SnakeGameController	
Responsibilities	**Collaborators**
● Retaining and updating the games information. ● Communicated with the model to keep its information updated. ● Use calculated information from the model and update itself to the status of the player's snake, its state in the game (alive or dead), its current state in movement (boolean, left, right, or if both are false strait), and an ArrayList of Strings that can be broken down with split() that will represent a snake objects sections by positions rotation and sprite.	● SnakeGameModel ● SnakeGameView

Model: The snake game model creates the snake game network and sets the item and snake parts positioning. This is where we make the snake shapes and create and host the server. It is thus responsible for listening to the client and updating the snake server based on the information sent by the client. In addition, the snake game model will raise any snake exceptions and display the error messages for debugging if errors are encountered. The model implements protocol and game index positioning interfaces that help control the server room from the snake game network.

Class: SnakeGameModel	
Responsibilities	**Collaborators**
• **Using an animation timer to update the state of the game** • **Keeping track of a player's actions, state, and collision detection.** • **A fully functional JavaFX app that will handle the calculations for positions and collisions.** • **During its animation updates, it will use in-game ticks to calculate the next position of the player and update each piece of the player to be one tick behind. It is dependent on the Snake class for calculations**	• **SnakeGameController** • **SnakeNetwork** • **SnakeException** • **<<interface>> Protocol** • **<<interface>> GameIndexPositioning**

This screenshot of the UML shows the implementation of our MVC design. It shows the dependencies and how the controller creates the model, and the view creates the controller.



```
SnakeGameModel
- MAX_PLAYERS: int
- SnakeGameController: theController
- socket: Socket
- networkIn: Scanner
- networkOut: PrintStream
- height: int
- width: int
# playCount: int
- gameRunning: boolean
- itemListPositions: ArrayList<Circle>
- snakeListPositions: ArrayList<Circle>[]
- scrapNodes: ArrayList<Node>
+ SnakeGameModel(theController: SnakeGameController)
- listener()
- close()
- updateSnake()
- setItemPositioning(itemPos: String[])
- setItemType(index: int, type: int)
- setSnakePositioning(i: int, snakePos: String[])
+ createNetwork(port: String, players: String, width: String,
  height: String)
- startHostServer(thisPort: int)

SnakeGameController
- menuMessage: String
# WIDTH: int = 800
# HEIGHT: int = 600
- dataWrite: boolean
- gameGoing: boolean
- ITEM_POSITIONING: ArrayList<Circle>
- SNAKE_PARTS_POSITIONING: ArrayList<ArrayList<Circle>>
- SCRAP_NODES: ArrayList<Node>
- VIEW: SnakeGameView
- MODEL: SnakeGameModel
+ SnakeGameController(view: SnakeGameView)
+ getView(): SnakeGameView
+ getNumOfPlayers(): int
+ setHost(port: String, players: String, width: String, height: String)
+ displayMessage(message: String)
+ setJoin(host: String, port: String)
+ getTrash(): ArrayList<Node>
+ updateView()
+ leftTurn(b: boolean)
+ rightTurn(b: boolean)

SnakeGameView
- root: Pane
- startMenu: SnakeMenu
- controller: SnakeGameController
+ addWalls()
+ userInput()
+ updateView()
+ getStartMenu()
+ addSnake()
```

SnakeNetwork: The main view for the players. The snake network creates all game objects in the server game room. This includes snakes, all game items (food), and walls. The network handles all the game objects that the player will interact with, and will handle all the collisions with the items, walls, and other snakes in the server. The network implements a protocol interface that will be shared between the client and the server so that each will implement this so the protocol is standardized.

Class: SnakeNetwork	
Responsibilities	**Collaborators**
● **The main view for the player implementing the multiplayer network.** ● **Shows a game of snake happening in real-time, updates movements 60 times a second** ● **Players can change the angle of the snake by using their arrow keys**	● **Item** ● **Snake** ● **SnakeTail** ● **GameAsset** ● **<<interface>> Protocol**

Lastly, we will talk about the lowest level of our snake game design, which is all the game objects in this game. All game objects are circles.

Snake and SnakeTail: The snake represents a list of snake tail objects that are assembled as a fully functional snake controlled by the player. A list of snake tails is a list of GameAsset objects as circles that will increase or decrease based on the snake's interaction of the items. The snake tails are also for getting their position and turning angle based on the parent node (parent snake tail).

Class: Snake	
Responsibilities	**Collaborators**
● **Represents a single snake object that will be controlled by the player** ● **Head will turn according to user's control (via pressing keyboard)** ● **Record the current length of the snake body** ● **It contains a list of SnakeTail objects representing a complete snake object.**	● **SnakeTail** ● **SnakeNetwork** ● **GameAsset** ● **Item**

Class: SnakeTail	
Responsibilities	**Collaborators**
• **Represents the actual visible piece of the snake** • **Keeps track of its own and its parent's positions and always positions itself one tick behind its parent** • **After a certain point will be able to collide with the head object ending the game** • **The most important aspect is each piece will have a reference to its parent piece (the next piece inline), and on an update will update its x y and rotation to be one tick behind its parent piece.** • **Inherits GameAssets.**	• **Snake** • **GameAsset**

Item: Poison, Food, and Potion are three items that can be found in the snake game, which will interact with the snake objects.

Abstract Class: Item	
Responsibilities	**Collaborators**
• **A GameAsset that if collided with will disappear and add to the player who ate its tail.** • **An item will randomly generate in the snake pane. When that happens, any player can eat an item, and there will be a limit to how many can appear at a once**	• **Snake** • **SnakeTail** **Subclasses:** • **Poison** • **Food** • **Potion**

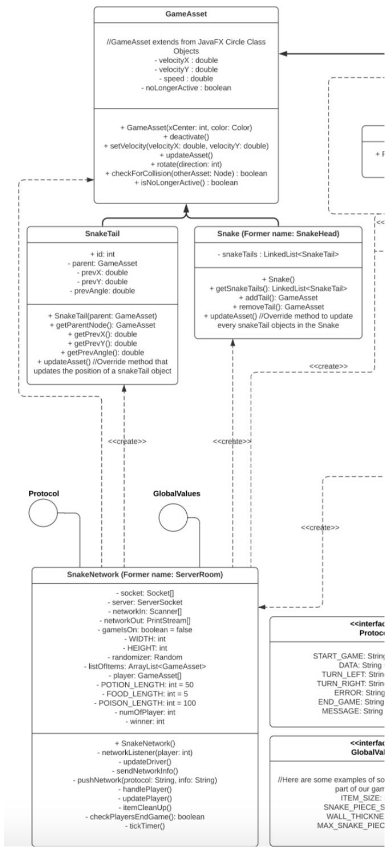
Class: Poison	
Responsibilities	**Collaborators**
● Represents the visible piece of the poison	● Item
● Inherited from the class Item	● Snake
● When a snake eats it, the snake shrink by 10 blocks	● SnakeTail
● If the snake is less than 10 blocks, then the snake will be just 1 block.	

Class: Food	
Responsibilities	**Collaborators**
● Represents the visible piece of the food	● Item
● Inherited from the class Item	● Snake
● When a snake eats it, the body length will increase by 1 block	● SnakeTail

Class: Potion	
Responsibilities	**Collaborators**
● Represents the visible piece of the potion	● Item
● Inherited from the class Item	● Snake
● When a snake eats it, the body length will immediately increase by 10 blocks	● SnakeTail

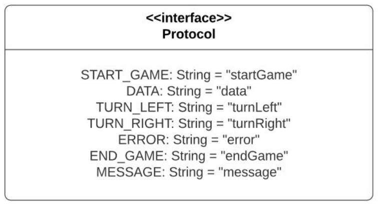
GameAsset: All the game objects listed above are based on this abstract class, the game asset, which is also inherited from the JavaFX Circle. This reflects our design of all the game objects in the game. All game objects are circles.

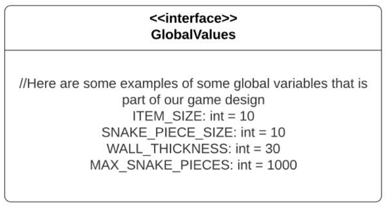
Abstract class: GameAsset	
Responsibilities	**Collaborators**
● **An abstract class that will have some of the basic principles all assets will require.** ● **A node root, which can be constructed into any shape requires, x, y, and rotate positioning, to the object to be manipulated around the pan; and will all have an update method, which when called will update the Node in the pain to be in the correct position.** ● **Key for polymorphism to simplify the act of treating objects as game objects.**	● **SnakeTail** ● **Item** ● **Snake** ● **SnakeNetwork**


GameAsset
//GameAsset extends from JavaFX Circle Class Objects
- velocityX : double
- velocityY : double
- speed : double
- noLongerActive : boolean
+ GameAsset(xCenter: int, color: Color)
+ deactivate()
+ setVelocity(velocityX: double, velocityY: double)
+ updateAsset()
+ rotate(direction: int)
+ checkForCollision(otherAsset: Node) : boolean
+ isNoLongerActive() : boolean
SnakeTail
+ id: int
- parent: GameAsset
- prevX: double
- prevY: double
- prevAngle: double
+ SnakeTail(parent: GameAsset)
+ getParentNode(): GameAsset
+ getPrevX(): double
+ getPrevY(): double
+ getPrevAngle(): double
+ updateAsset() //Override method that updates the position of a snakeTail object
Snake (Former name: SnakeHead)
- snakeTails : LinkedList<SnakeTail>
+ Snake()
+ getSnakeTails(): LinkedList<SnakeTail>
+ addTail(): GameAsset
+ removeTail(): GameAsset
+ updateAsset() //Override method to update every snakeTail objects in the Snake
<<create>>
<<create>>
<<create>>
<<create>>
Protocol
GlobalValues
SnakeNetwork (Former name: ServerRoom)
- socket: Socket[]
- server: ServerSocket
- networkIn: Scanner[]
- networkOut: PrintStream[]
- gameIsOn: boolean = false
- WIDTH: int
- HEIGHT: int
- randomizer: Random
- listOfItems: ArrayList<GameAsset>
- player: GameAsset[]
- POTION_LENGTH: int = 50
- FOOD_LENGTH: int = 5
- POISON_LENGTH: int = 100
- numOfPlayer: int
- winner: int
+ SnakeNetwork()
- networkListener(player: int)
- updateDriver()
- sendNetworkInfo()
- pushNetwork(protocol: String, info: String)
- handlePlayer()
- updatePlayer()
- itemCleanUp()
- checkPlayersEndGame(): boolean
- tickTimer()

This image of the UML diagram to the right shows the dependencies of the SnakeNetwork, Snake and SnakeTail, and GameAsset. The snake network creates the Snake and SnakeTail in the network, their properties are inherited from the GameAsset class, which creates the objects from the javafx circle.

As can be seen from the diagram, the snake network class implements two interfaces, game global values, and protocol. Game global values is an interface that stores all the numbers that we used in designing our game. Protocol is the string word that will be communicated between the server and the client. These protocols will decide the running of the game. (Some keywords shown below)


<<interface>>
Protocol
START_GAME: String = "startGame"
DATA: String = "data"
TURN_LEFT: String = "turnLeft"
TURN_RIGHT: String = "turnRight"
ERROR: String = "error"
END_GAME: String = "endGame"
MESSAGE: String = "message"


<<interface>>
GlobalValues
//Here are some examples of some global variables that is part of our game design
ITEM_SIZE: int = 10
SNAKE_PIECE_SIZE: int = 10
WALL_THICKNESS: int = 30
MAX_SNAKE_PIECES: int = 1000