# 

# Concurrent Programming: Assignment II – Snake Game.

Unisa |

Group 5

ErkHan Atesh, Ke zhang, Jason Lay

2019

Table of Contents

[Concurrent Programming: Assignment II – Snake Game. 0](file:///C:\Users\MetaBoi\Desktop\Final%20Report%20.docx#_Toc11015167)

[0](#_Toc11015168)

[-Rules and Logic 1](#_Toc11015169)

[*Rules:* 1](#_Toc11015170)

[- *Details:* 1](#_Toc11015171)

[*Logic:* 1](#_Toc11015172)

[- *Login and server population:* 1](#_Toc11015173)

[- *Server and Hosting:* 1](#_Toc11015174)

[- *Snake and movement:* 1](#_Toc11015175)

[- *Border Map and Snake Location:* 2](#_Toc11015176)

[- *Game State, UI and Player Snakes:* 2](#_Toc11015177)

[-Diagrams: 2](#_Toc11015178)

[Design Class Diagram: 2](#_Toc11015179)

[LTSA 3](#_Toc11015180)

[-Responsibilities: 4](#_Toc11015181)

[Erkhan Atesh 4](#_Toc11015182)

[Ke Zhang 4](#_Toc11015183)

[Jason Lay 4](#_Toc11015184)

[-Conclusion: 5](#_Toc11015185)

# -Rules and Logic

## *Rules:*

This game doesn’t have any fruit, and the whole game revolves around survival. Snake’s objective is to block other snakes with it’s body -therefore killing them- and try to stay alive for as long as possible. Last snake standing wins.

### *Details:*

* If snake hits another snake, it dies
* If snake hits the border, it dies
* If snake gets hit by another snake, it’s body grows by 1
* If there’s 2 snakes left, one of them has to die for the game to end.

## *Logic:*

### *Login and server population:*

Four classes are involved in this process: *Client, LoginBuffer, ServerHelperClass, Server.*

**LoginBuffer** – is responsible for inserting and extracting *InfoStore* object class instances from Stack collection when appropriate. Plays a role of a buffer.

**Client** – is responsible for calling a method *loginRequest() -*that belongs to *LoginBuffer* class-, it’s responsible for inserting a passed instance of an *InfoStore* class, that contains all the essential information, to the Stack that plays a role of a buffer called *loginQueue.*

**ServerHelperClass** – is responsible for obtaining information from the *LoginBuffer* stack instance by calling a method *processLoginRequest(). processLoginRequest()* method belongs to LoginBuffer and is primarily responsible for extracting an InfoStore object instance, from the Stack *loginQueue,* and then returning it. Once *InfoStore* instance has been obtained from the *LoginBuffer*, *ServerHelperClass* uses data from obtained instance to instance and pass a snake object instance to the HashMap that is stored on the *Server* class.

**Server –** is responsible for storing instantiates Snake objects [in context of *Login and Server population* logic].

### *Server and Hosting:*

Server is primarily responsible for hosting all the processes, such as: *Login, UI* and *Snakes.* It doesn’t hold logic of these processes, all it does is simply host other processes. In it’s run method it demonstrates Executor implementation. Reasons for executor are mainly due to the need for snakes to be logged in first before starting to render the game window and to make execution and joining of the processes easier.

### *Snake and movement:*

Snake accommodates logic of the following: *Movement, Collision detection, self-destruction, collection editing.* Snake holds all of the listed above functionality primarily due to it being safer in terms of concurrency. In other words, Snake is responsible for it’s own state and state of it’s variables and collections, hence eliminating a risk of concurrency errors. Movement method is responsible for collision detection, growth of the collided snake, general movement [it generates a random direction].

### *Border Map and Snake Location:*

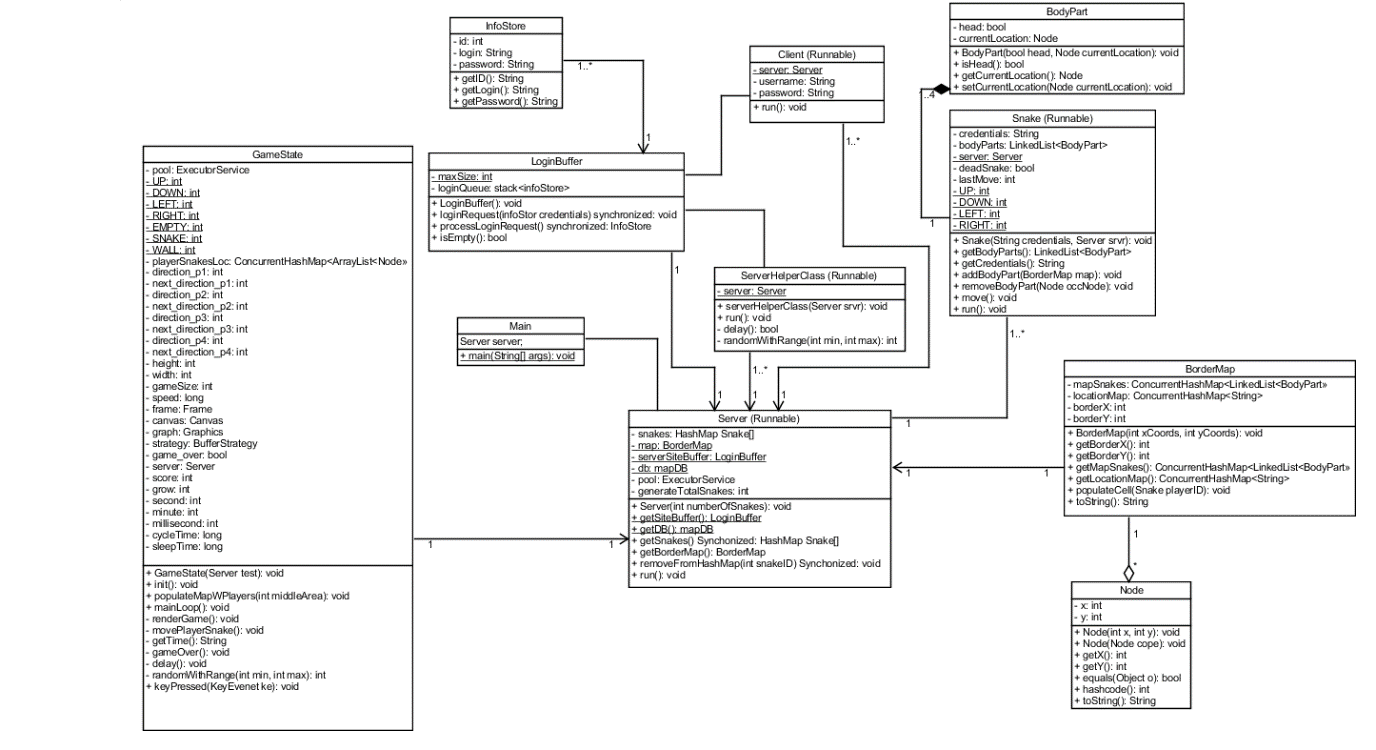
BorderMap class is primarily responsible for holding information about snakes’ locations. It has collections that contain essential data for snakes’ movement and UI display of said movement. It also holds a method *populateCell(…)* that is responsible for placing a snake at a random location on a map [also ensures that cell doesn’t have other snakes residing there].

### *Game State, UI and Player Snakes:*

GameState class is responsible for UI representation of snake interaction between one another and accommodation of player snake functionality. GameState uses *BorderMap’s* collections to obtain coordinates to then establish what exactly is painted where [either it’d be a border or a snake]. Player movement method is pretty similar to Snake movement methods, with an exception of movements being inputted through a keyboard instead of being randomly generated.

# -Diagrams:

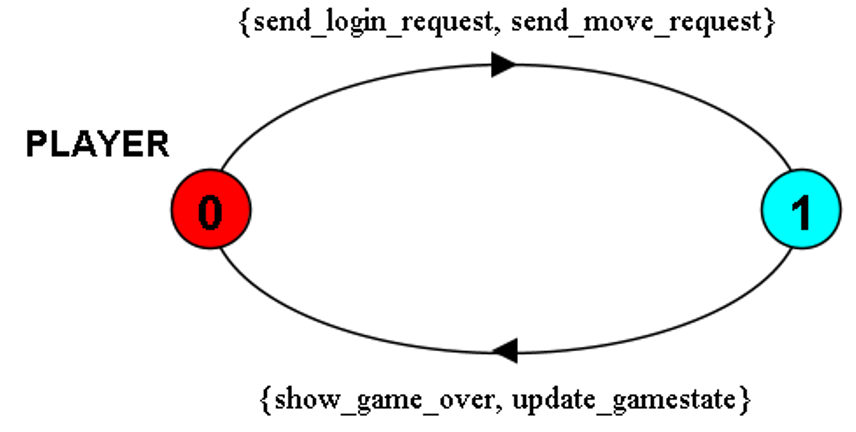
## Design Class Diagram:



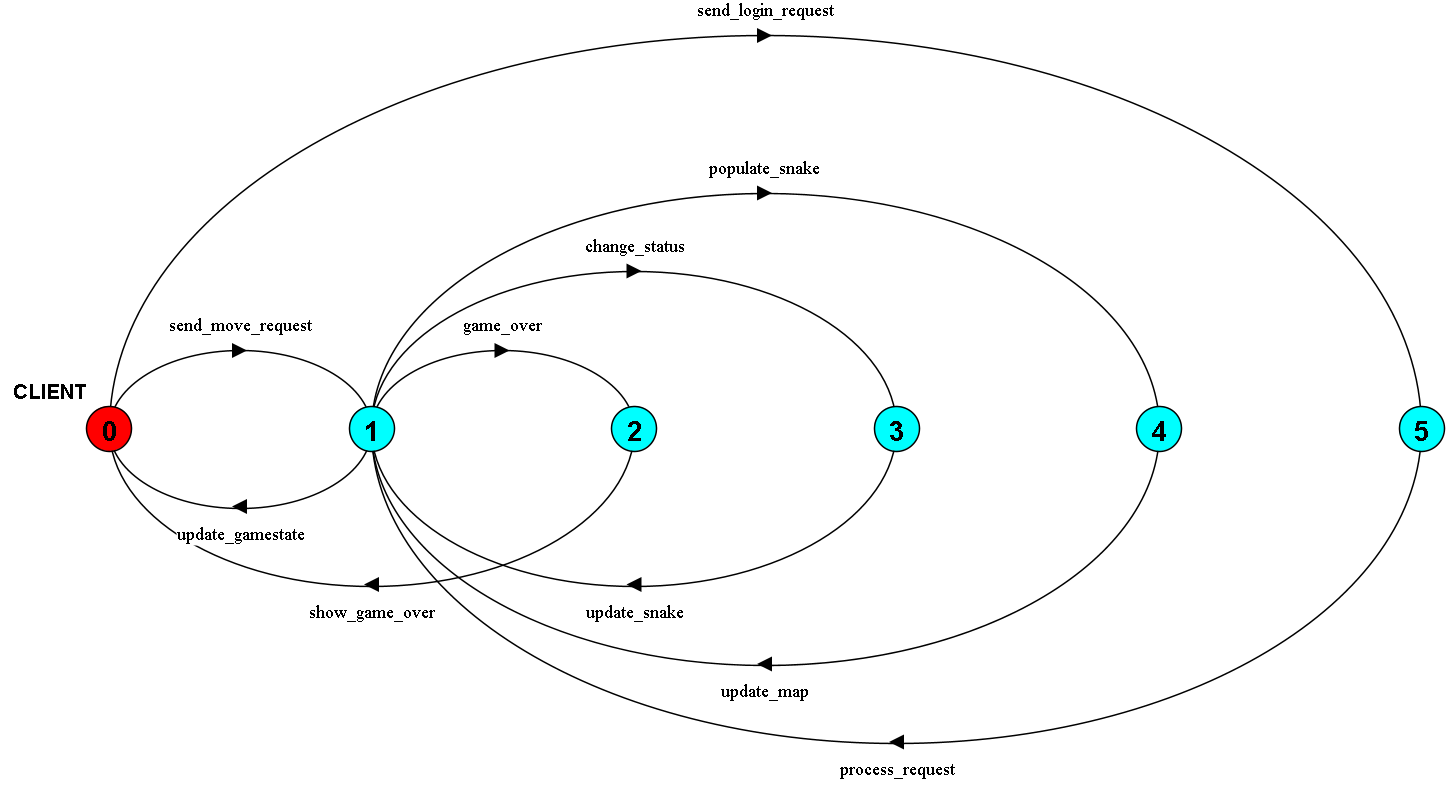
This diagram contains 1 aggregate, 1 composite and the rest are either direct or simple relations.

* To justify aggregate relationship, we need to take into account the fact that a snake practically doesn’t exist without *BodyPart*s object instances. *BodyPart* class is used to draw a snake and to demonstrate how long a snake is, therefore body parts form a snake. If all *BodyPart*s are removed from the Snake’s collection of *BodyPart*s then it would be considered dead.
* To justify composite relationship, we need to consider that border map is formed thanks to nodes. Nodes hold x,y coordinates that allow for identification of a snake on a map. However, the reason why it’s composite rather than an aggregate relationship is because it technically can exist without nodes.
* To justify simple and direct relations holding the majority, we need to take into consideration the fact that a lot of the classes do not directly derive from one another [or are subclasses of one another] and secondly, classes are primarily only interact with each-other and are stored inside one another, which does not necessarily indicate anything more than a simply or direct relationship.

## LTSA



As we can see on a model PLAYER, we can clearly identify a set of actions that a player model demonstrates. It shows that a client can send login request as well as send requests for movement, also, lose a game and update a game state – meaning, changing location or dying.



Composition of Server and Player results in LTSA above. Player will send login request to the buffer in order to login to the server. The buffer will process the login request and connect player to the server. Then the player can send move request to the server, the server will process these request to change snake’s status and its growth, map gets updated every render cycle. Once the snake dies, it will lead to the game over state for said user.

# -Responsibilities:

## Erkhan Atesh

* Server
  + PopulateCell
  + Constructor
  + Run method
* Client
  + Whole Class
* GameState
  + Render Code
* Snake
  + Skeleton of the Snake Class
  + Collision Logic and some movement Logic
* BorderMap
  + Whole class
* ServerHelperClass
  + Whole Class
* Main
* LoginBuffer
  + Whole Class

## Ke Zhang

* Server
  + Some logic of populate Cell
* Snake
  + Movement Logic
* BodyPart
  + Whole class
* GameState
  + Render Code
  + Player snake
* Main

## Jason Lay

* Server
  + MapDB
  + MapDB population
* Node
  + Whole Class
* InfoStore
  + Whole Class
* Main
* Tests
  + Mockito
  + Junit

# -Conclusion:

Project can handle up to 350 snakes running at the same time till the moment of the last snake standing. This project demonstrates concurrency and concurrency principals and data structures such as Producer Consumer.