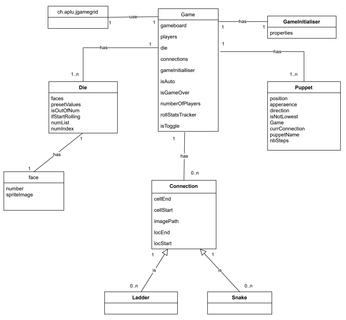
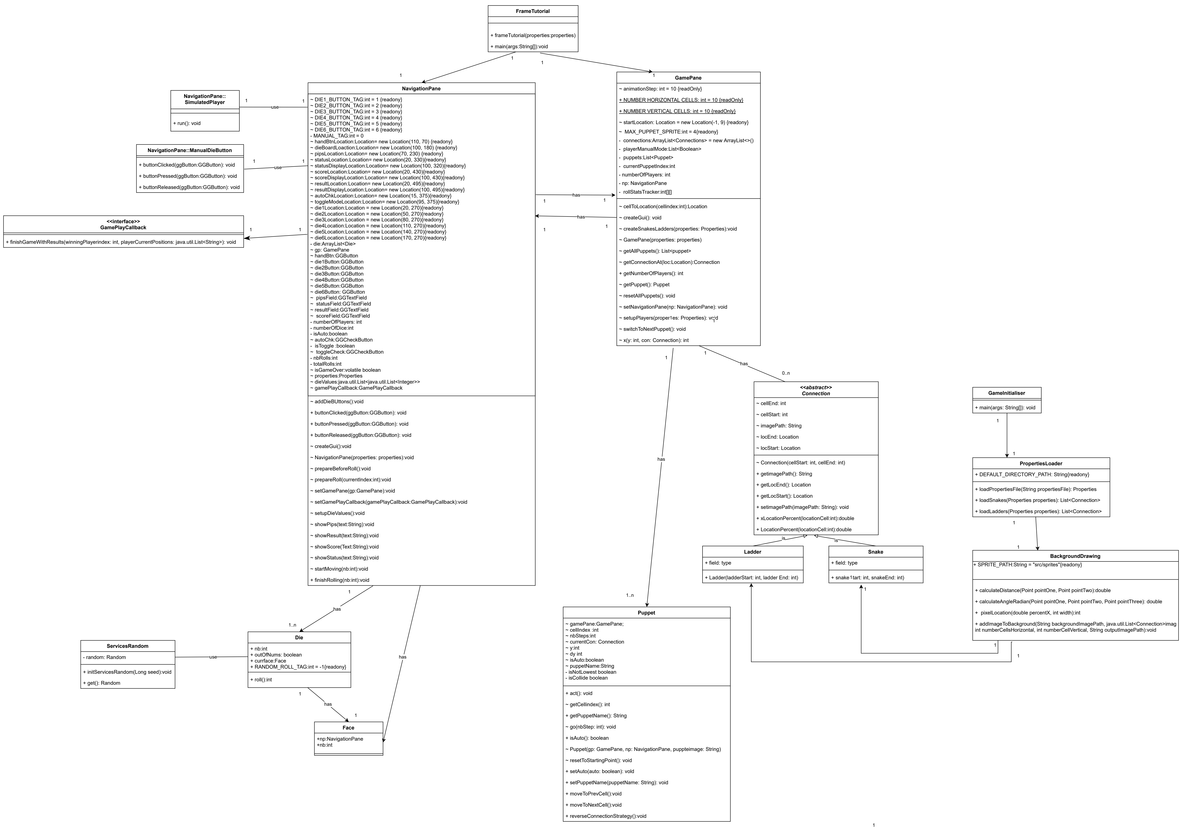
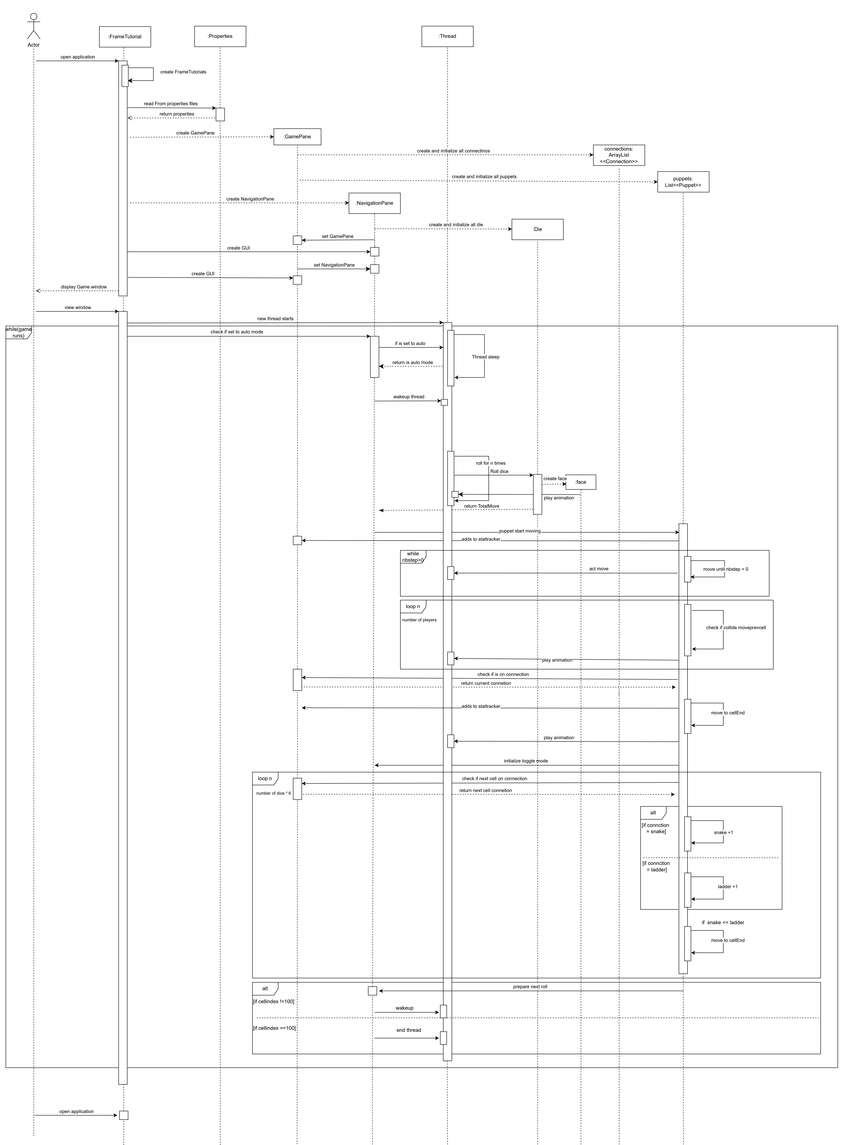
Domain class diagram



Design class diagram



Design sequence diagram



Current build

A video game based on a popular board game called Snakes and Ladders which is developed by The Games division of New Exciting Realtime Discoveries Inc. (NERDI). This is a game based almost entirely on luck, the players start rolling the dice from the starting point and the first to the finish wins, there may be some path symbols which provide paths between two squares on the board to guide the players up or down. To enhance the game and make SLOP more interesting, our task is to improve the design and implementation of the game according to design principles (GRASP) and add some rules to make the game more reasonable and playable.

One of the most obvious advantages of the current version of the SLOP provided by the project is Polymorphism. The use of inheritance could reduce the interface and interface between modules by enhancing consistency, which can increase the maintainability of the program. Similarly, it provides program reuse functionality, reduces code and data redundancy, and increases program reusability. It could also simplify the complexity of the code, clearly reflect the hierarchical relationship between related classes, and make the conditioning of the code clearer. In addition, the provided code has a nice code style that is easy to understand, and a clever game design that makes the game run successfully in certain modes.

However, there are some bad designs in the current build of this game. These designs and the lack of documentation lower the efficiency to implement new features and maintain the code. A very obvious characteristic of the current build is high coupling between classes, almost every class has an attribute storing other classes. GamePane, puppet and die all have an attribute storing navigation pane, which can be avoided because puppet can access through its gamepane attribute, we improved this through the process of implementing task 2~5. Classes are also low cohesion, with each class doing either more or less than it is supposed to. In the current build, the navigation pane is responsible for rolling a dice, as well as tracking the preset die value in properties files, this is an example of low cohesion, because die should be responsible for rolling a value, we improved this in task 1.

Improvements

Task 1

The goal of task 1 is to implement the function of having multiple die in a single game, however, before starting to implement this function, there are a few improvements to make. Currently, the workflow of a dice rolling is: NavigationPane has an attribute named dieValues, this attribute keeps track of all the preset die values in a 2D ArrayList, Navigation pane uses a complex formula to decide which players' ArrayList and which exact dice number to use, then return this value to the roll() function, that creates a die instance (extend of the actor), sets its sprite animation then set its setActEnable to true. The die instance starts to play the sprite animation, and when that animation finishes, the die calls the startmoving() function in the navigation pane which allows the puppet to start moving. This workflow works reasonably well with one die, but it is very low cohesive. In this application, navigation is a use case controller, responsible for getting user inputs and coordinating dice, but it should not be responsible for rolling the dice, a dice should roll itself, navigation pane only needs to initiate a roll. To solve this problem, we used the concept of pure fabrication, by moving the roll() function to the die class and adding an ArrayList inside the die to store a specific players' die values and another arraylist in

NavigationPane to store the number of dice based on the number of players. Now, to roll a dice, navigationpane no longer creates a new dice, but gets a die from the ArrayList using the player index and calls the roll() function in that dice. But this raises another problem, due to the nature of actors, we can not change the sprite animation set in the constructor when instances were created, so we made up a class called face to handle animation for a specific number. Each time roll is called, a new instance of face containing the current number is added to the actors, and when the face animation finishes, it calls the finishRolling() function in NavigationPane which then detects whether the number of rolls equals the number of die, if it is true, then it calls startMoving() function, else, it starts rolling again.

Task 2

In the second task, we need access to multiple information/classes across the program to accomplish the design - Navigation Pane, Puppet, Connections and Number of Die are all essentials to detect players' input and implement additional rules to the avatars' movements.

All information related to the navigation pane, puppet, connections can be accessed by the Gamepane class. Information which is stored indirectly in the navigation pane such as the term "number of die" could also be accessed by the Gamepane.

By accomplishing/ identifying the design above, the Gamepane class would now have a wide range of information access which makes Gamepane an efficient information expert in our design. To achieve task 2, we took advantage of Gamepane's nature.

Since tasks will be fulfilled by objects' own information, The information encapsulation will thus be maintained. Which means, lower coupling is expected as a result of implementation of information expert principle and this design.

Task 3

The goal of task 2 is to detect whether players will collide with one another. If a puppet steps on a puppet after moving, the puppet being stepped on should move one step backward and follow the rules on that landing square.

To achieve this specific requirement, we changed GamePane to an information expert. After the implementation, the GamePane class can then hold the total number of squares that each puppet has moved during the game which was stored in the puppet class. Additionally, with the help of the information expert, the class will also have access to all other puppets' location information.

By having access to the information of avatars' locations and movements, we could implement the new rule quite easily.

By turning gamePane into an information expert, we could also achieve low coupling and lower dependency between the classes due to the same reason above.

Additionally, to implement the "moving one step back" feature correctly , after multiple tests, we have realized that a separate method which moves the avatar one step back needs to be present, since the provided "one step back" code in the original version of the program is regardless of the

vertical space, which means a play avatar will be pushed out of the grid boundary when it lands on another players avatar which sits on the right side of the map.

Task 4

In task 4, we aim to implement a new feature to the current game - a toggle option which gives each player an opportunity to reverse the current roles of the path symbols at the end of their round, Making the snakes go up and ladders go down. Additionally, a toggle button which allows interactive setting, an indicator which displays the current operation status of the path symbols are both needed to be implemented in our design.

Since the GamePane class has access to the information about all connections, thus we decide to make it an Information expert which could be used to manipulate the directions of all connections.

We approached this task taking advantage of polymorphism, by having the ability to provide different implementations of a method, we could replace all snakes with ladders, and all ladders to snakes.

Information experts are also implemented to achieve the “simple strategy” of AI players which would be employed in the automated testing mode. Since the game pane knows all players’ positions and all connections, it is easier for us to develop a strategy which heavily relies on observation and evaluation of other avatars’ position and future possible connection points down the path.

Additionally, we have become aware that since the changes made in the orientation of the connection points are permanent which means the effect of each “toggle” will last until the game’s termination, the past orientation status of the connection points do need to be taken into consideration in future rounds.

Task 5

In task 5, we aim to track the game play by keeping the following in-game statistics: The number of times that each player has rolled each possible value on the dice, the number of path symbols that the avatars have traversed up and the number of path symbols that the avatars have traversed down.

Due to the similar reasons mentioned in previous tasks, since GamePane class has access to the information of whether a player has moved on a connection point or not and the total movement number of each player, we have made GamePane class the information expert.

In order to track and store all the dice rolls values that each player has tossed, we have decided to implement a simple hashmap.

Additionally, since the game rounds are based on time duration, we have discovered that in the default code if the player's avatar climbing/ descending actions take too long, it could force the system to skip the current round. To avoid such behavior of the program, we have manually implemented a faster climbing/ descending time for the avatars.