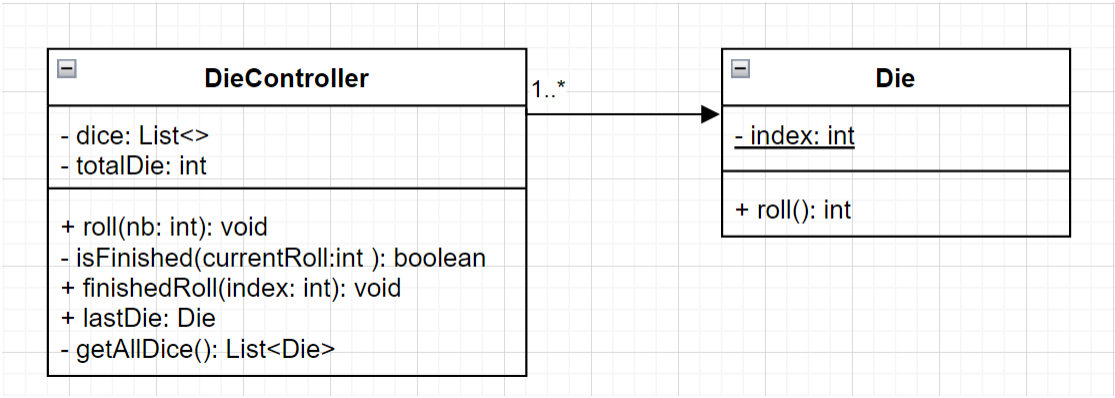
SWEN30006 Project 1 Report

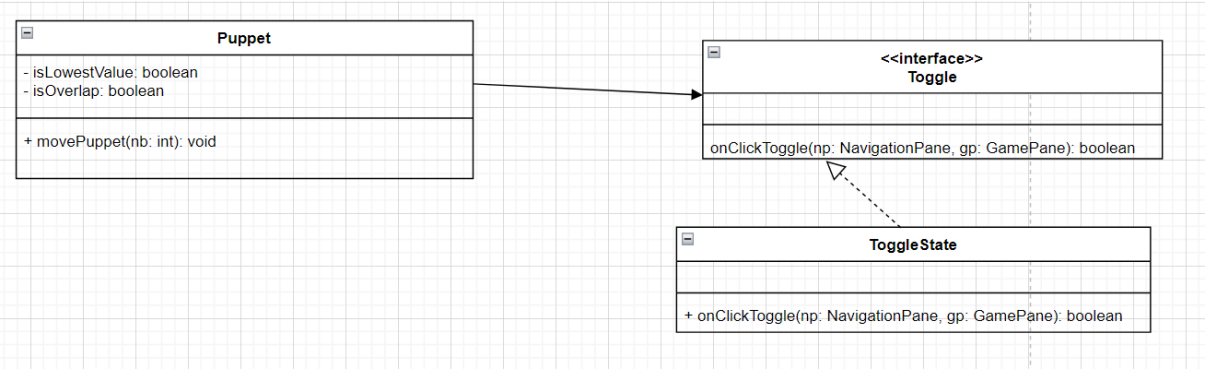
Wednesday 2pm Team 1
Zheren Huang 1050041
Chaowen Zeng 1079279
Cheng Yin Loh 1179055

Before the design change for task 1, the NavigationPane is tasked to roll the dice. While it works and is acceptable, it leads to low cohesion as NavigationPane might be too cluttered which affects comprehensibility. Therefore, an intermediary artificial class DieController is added to take over the responsibility of dice rolling from NavigationPane. This makes use of the Pure Fabrication principle as the class does not represent a concept in the problem domain and is specifically created to support high cohesion and low coupling. In addition, by the creator principle, since NavigationPane "contains" and closely uses DieController, it is responsible to create the DieController object. NavigationPane's tasks can be delegated, and classes would be more task independent, reducing the impact of change. This design would support any pre-specified number of dice (at least one) being rolled, by just changing the properties file. The dice's information, such as animations, location, size, does not need to be changed to accommodate for when additional dice are used as only one die is displayed on screen. For example, in a game where two dice are used, instead of rolling two dice at once, the player would roll twice, and the total face value would be added up.

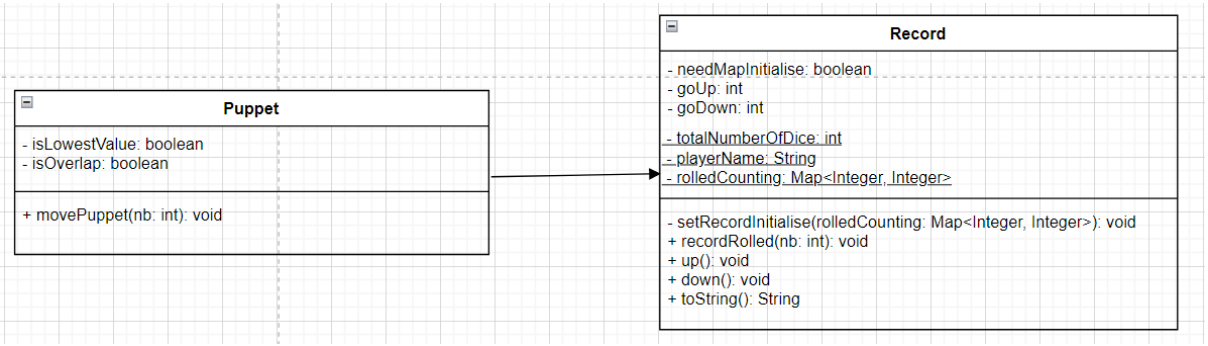


In task 4, the protected variation principle inspired the current design using interfaces. A class ToggleState is created, which implements an interface Toggle, to document the algorithm used to determine if an auto player should reverse the direction of all snakes and ladders. The Puppet class would be responsible for creating the ToggleState object since by the creator, Puppet uses the algorithm for determining whether to change directions if it's an auto player. To add on, if NERDI wants to employ other strategies for auto players in the future, extra classes implementing the Toggle interface can be effortlessly created and implemented. Variations in the algorithm can be organised in separate classes and files and can be tested to seek the best algorithm for auto players'. The GamePane would oversee flipping the start and end points of connections since it stores the list of all connections on the board, and NavigationPane would be responsible to call that method in the GamePane after performing

checks upon the toggle button on the user interface. This design would protect other classes from the variations of the algorithm to toggle connections by wrapping it in an interface and utilising polymorphism to implement the variations through the interface, hence achieving protected variations.

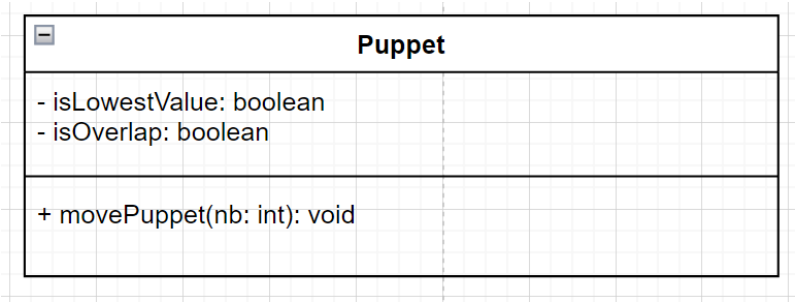


To record statistics in task 5, a class named Record is used to store details of rolls and traversing connections. By the creator principle, Record is created by Puppet as a player contains information of their gameplay, hence closely uses the Record class. Pure fabrication is also utilised since Record is purely a class created for convenience of recording statistics of players. This design attests to low coupling and high cohesion as both classes are independent of each other with different functionalities and responsibilities. Without utilising the Record class, Puppet would be doing extra work to store statistics that could be simply delegated away. Even though recording statistics isn't unrelated to Puppet class, the statistics are not part of the game and are only used solely to track players' gameplay. Through this design, additional statistics can be easily added or removed since new methods can be added in the Record class to record said statistics and called whenever the puppet acts. This leads to a cleaner design where the Puppet class can focus on its tasks.



On the other hand, there weren't significant design changes for task 2 and 3. In task 2, since we need to check the condition for task2 before the go() method in Puppet to proceed , so we decide put this condition in puppet class in order to let puppet class know that whether it dies the lowest number, and puppet act as an information expert. In order to achieve that, we create a method call isLowestValue() to check this condition. The reason why we treat the puppet class as an information expert is that no other class use the function isLowestValue(). It will ensure low coupling between those classes.

Similarly in task 3, as per the design sequence diagram, a for loop is used to determine if any puppet has overlap, and if so, get that puppet to move back a square by utilising additional if conditions in the Puppet's act function, and a function movePuppet() that caters for moving forwards and backwards that replaces the existing moveToNextCell() function. And at same time, we decide to let Puppet act as an information expert again, to lower the coupling in this system. These changes might lead to higher cohesion and lower coupling between the classes. For instance, Puppet contains variables which aren't directly related, and NavigationPane consists of extra work. While it might be harder to implement changes in the future, this minimises the number of classes created and used which reduces complications in the overall design.



Overall, the creator principle is used most often as it is one of the main methods of determining responsibility for creating instances of classes. Low coupling and high cohesion are crucial in design as they allow code to be more maintainable and easier to support future design changes. Pure fabrication is utilised a couple of times to assign responsibilities to different classes, hence leading to low coupling.

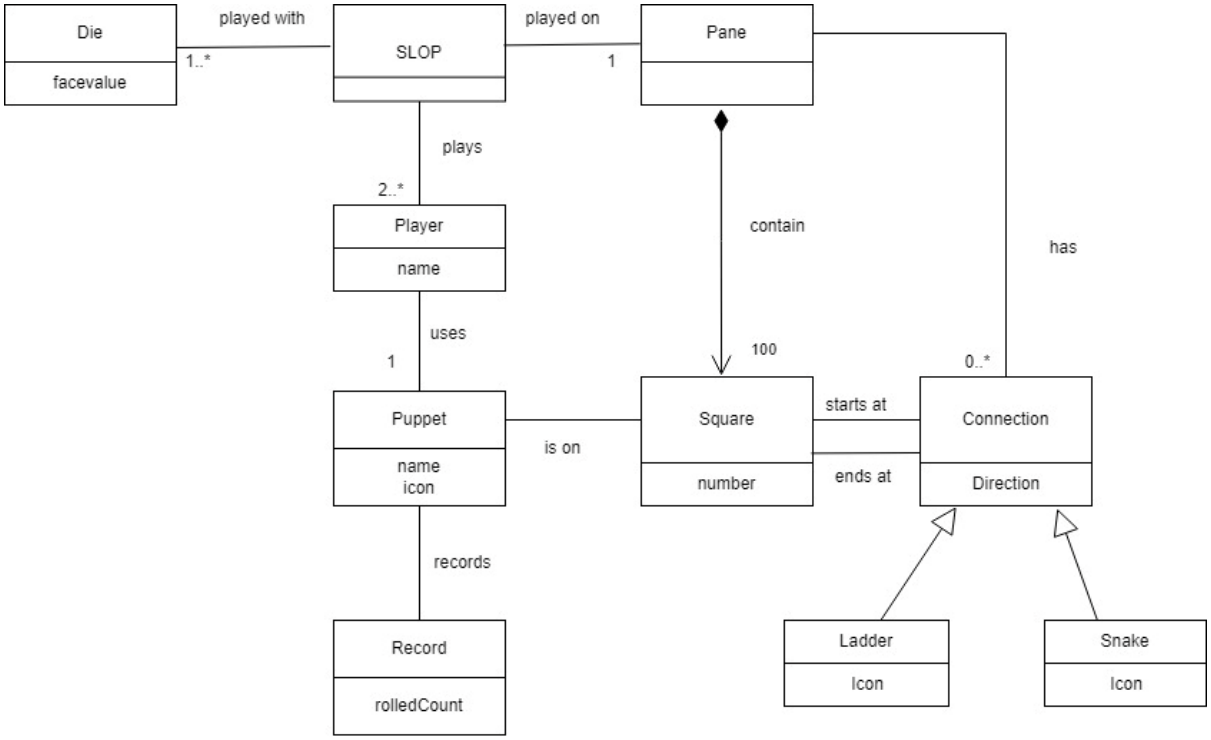


Figure 1: Domain Model

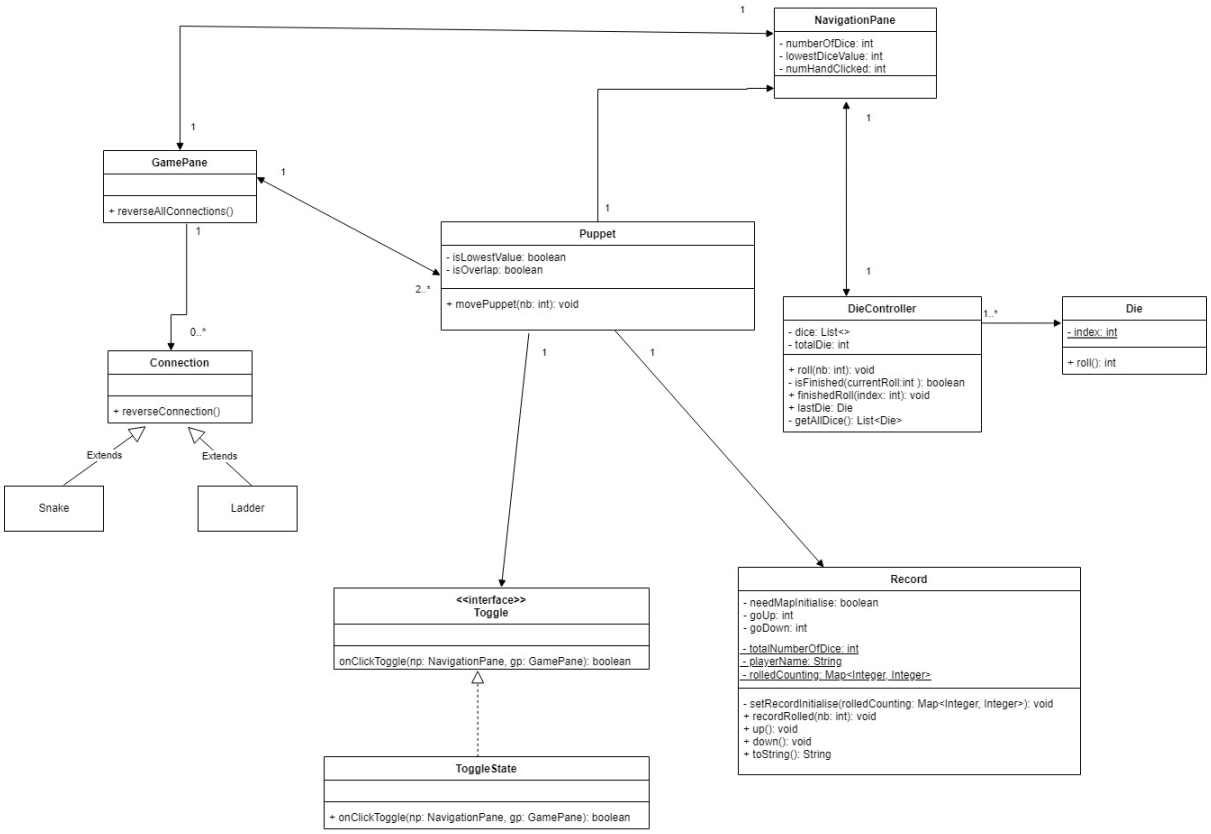


Figure 2: Design Class Diagram

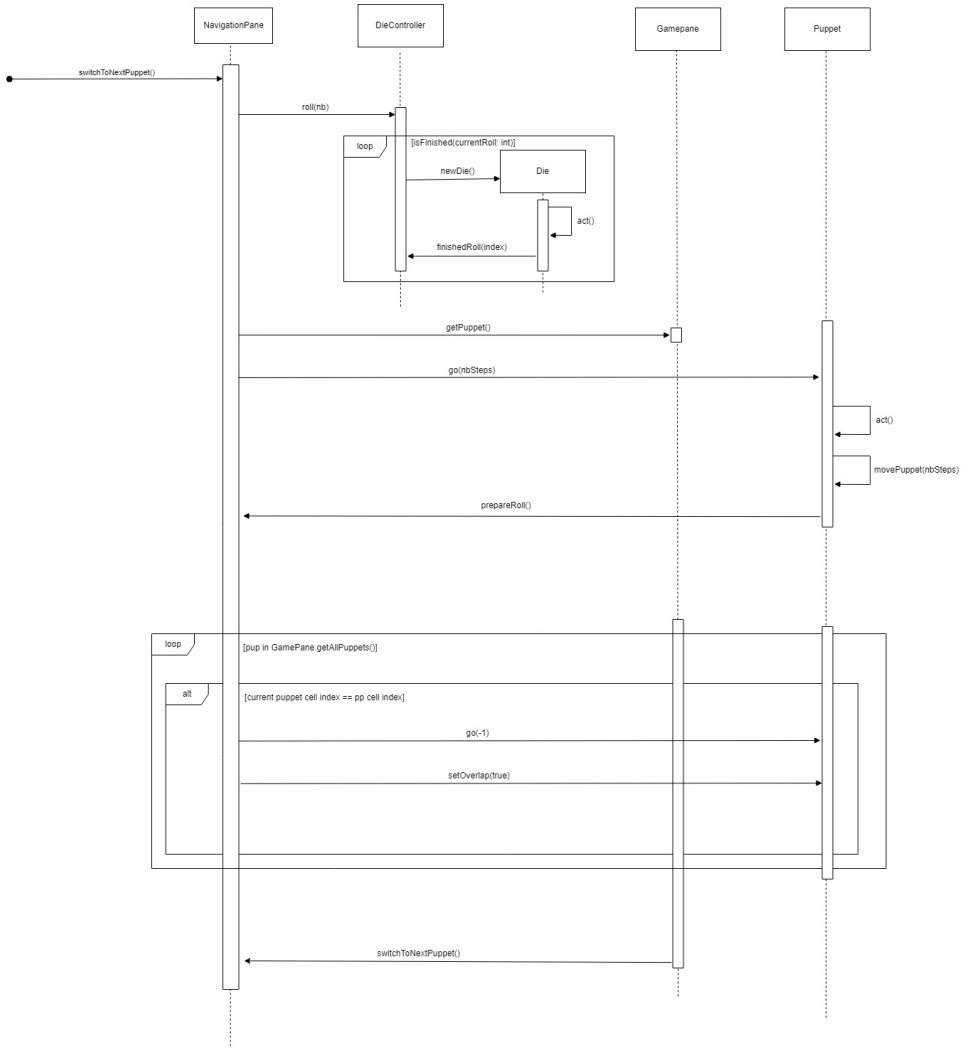


Figure 3: Design Sequence Diagram