**Project Report:**

* Design Rationale.
* Task Implementations.
* Design Visualisations.

-

-

-

-

Task 1:

*“Allow for the game to use a pre-set number of (six-sided) dice, defaulting to one. The property file already contains the setting, but it is not currently used. Each dice should be rolled (randomly generated) in turn.”*

*C class*

Through our study of the base code for the program, it was understood that the navigation pane was central in that it interacts with both the actor and controllers. Our work in task-1 aimed to enable multiple dice functions where if more than one die were set, all dice must be rolled before an actor may move on the board. To achieve this, a new class ‘Cup’ was implemented which worked as a bridge between the ‘Die’ and ‘NavigationPane’ classes. It sums all values of the rolled dice and outputs the value to ‘NavigationPane’. Following this, by using ‘AddActors’ (line 384 in ‘NavigationPane’), the corresponding actor is worked on and moved to the specified location on the board. Our reasoning for implementing a new class for task 1 was that it was the simplest way to tie that functionality into the code and at the same time improved the modularity and maintainability of our program with low-coupling.

Task 2:

*“Players do not travel down a path symbol if they rolled the lowest possible roll (a 1 if using only one dice, a 3 for three dice and so on) to land on the symbol square.”*

*Low coupling*

To ensure the game adhered to these specifications, we created a new Boolean type called ‘isLowest’. The Boolean ‘isLowest’ is able to determine whether or not the lowest possible number were rolled by comparing the rolled value with the number of dice, (ie rolled value is equal to number of dice). So, in the case where ‘isLowest’ is true and the player lands on a position which exhibits a connection, the connection will not be triggered. Our ‘isLowest’ variable was incorporated into the Puppet class (go method, line 81) with an if statement. To complete task 2, we implemented the ‘isLowest’ Boolean as described above and altered some if statements in the ‘act’ class (line 128 in Puppet.java) to check if the player is on a connection or not which combined to achieve the desired functionality described in task 2. The main idea behind our approach to task 2 was that it was again, simple to implement, both in terms of the complexity of the changes made and the actual amount of code that needed to be written and or altered. So not only did our solution effectively solve the problem at hand, it was time efficient meaning we could invest more energy into other parts of our project.

Task 3:

*If a player lands on the same square as their opponent when moving after rolling the dice (only), the opponent moves one square backwards and must follow the rules of the landing square.*

*Low coupling*

To develop the functionality specified in task three we implemented a Boolean type ‘isBack’ within the Puppet class. ‘isBack’ works by getting all puppets from the Puppet class and comparing the current players index with all the other puppets’ indexes. Therefore, if there exists a match between the indexes of the current player and any other puppets, ‘isBack’ returns true and the matched puppet has its index reduced by one. What we noticed when discussing alternatives to solve for task 3 was that the information required to develop a solution was largely already available in the skeleton code. We were able to take advantage of this by simply pulling information from the ‘Puppet’ class and comparing it via our new Boolean ‘is Back’. This appealed to the simplicity we desired in our solutions and also avoided us having to implement further code to track the puppet states which would essentially mean unnecessarily duplicating information in the code.

Task 4:

Why create Strategy interface: protected variation and indirection

Creation of Strategy object inside Puppet class: Creator

4.a,b

An isReversed variable is added to the Connection class which indicates whether the Connection is reversed or not, when isReversed=true, the starting location and ending location is exchanged comparing to the initialize value, the reverse action is achieved by reverse() method inside Connection class, calling this method will set isReversed to the opposite value(when it is true, set to false and vice versa), and do the location exchange. There is also a setReversed() method that achieve the same functionality, but instead of changing isReversed back-and-forth, it can assign isRevered directly to true or false, this method is used for toggling strategy button.

4.c

In NavigationPane class, the basic framework of toggle button is already set up. We adjust its calling toggleCheck() to update with isReversed variable in startMoving() and buttonChecked() method.

4.d

The reason we decided to create a Strategy interface is that in task 4.d, it is noted that NERDI games will change strategy in the future, which requires our design of implementations of task 4 to be robust with changes of the implementation of future coming strategies, that is, we need to protect the functionality of the existing system from the incoming new changes of Strategy class. Thus, we decided to use Protected variations pattern and create a Strategy interface to handle future coming classes. In this way, the coupling of the system and the strategy is reduced, and applies indirection. When the developers decided to implement a new strategy or change the current strategy, they only need to create a new class which implements the Strategy interface or changing the methods in the implementation class. The rest of the system only cares about methods declared in the interface, which protecting the rest of system from changing in order to adapt to new strategies

We decided to put the creation responsibility of Strategy object to Puppet class by applying Creator Pattern, because it makes sense to let each puppet have one strategy, and the strategy and puppet are closely related (puppet needs to use instance of Strategy class), therefore in the future the developers want to apply different strategy to different puppets, our design still works.

For calling of doStrategy(), we add that within act() method in puppet so that the puppet will apply strategy after its moving action.

We also added two more methods inside GamePane class to help with the giving strategy implementations, getNextPuppet() will get the next puppet and reverseAllConnection() will do all connections reversion in one go, the position of these methods’ implementation apply Information Expert pattern.

Task 5:

*In order to track game play, the game should keep some basic statistics: for each player, we want to know how many times that player rolled each possible value of the dice, and how many path symbols they traversed up and how many path symbols they traversed down. NERDI will want to add additional statistics in the future.*

*a. An example of the required format for output of dice roll counts is: “Player 2 rolled: 2-7, 3-11, 4-0, 5-0, 6-2, 7-3, 8-1, 9-0, 10-1, 11-1, 12-2”*

*b. An example of the required format of output for path symbol traversals is: “Player 1 traversed: up-3, down-7”*

*why create Statistic class: high cohesion, low coupling*

*create Statistics class inside Puppet class : Creator*

*methods inside Statistics class: information expert*

We interpreted task 5 as an extension of the current game which enables the puppet movement recording function. We had to implement an efficient approach so the function could record movements both standard dice rolls and path symbol traversals. We achieved this by creating the class ‘statistics’ which makes a HashMap for record storing and then prints out a copy of the current record after a player’s turn has ended. All the information records were stored in a string format and all of these records were drawn from the ‘Puppet’ class, more precisely the individual actions of each puppet. We chose to implement this solution for task five as it demonstrated low-coupling and high cohesion through the creation of the ‘statistics’ class. The inner workings of ‘statistics’ aren’t concerned with other parts of the code, aside from requiring correct input, this way our solution to task 5 was modular and also ergonomic. If we knew the input to ‘statistics’ was correct, but its output was buggy, it narrowed down our search for the error in our code to a single class, it was effective both in terms of execution, but also modularity and time efficiency.