**Design Rationale:**

Our approach to achieving the specified functionalities centred on simplicity and general adherence to GRASP principles. As part of this, we set out to introduce alterations and additions to the code only when necessary and in such a way that it limited the responsibilities of the original classes. When studying the skeleton code, we understood that a lot of the functionalities we needed to develop could be achieved with very minimal adjustments to the code and as we worked on our design models and developed our understanding of the assignment itself, the ways in which we could solve the problems became clearer and our ability to work with GRASP principles improved. This is reflected throughout our work and discussed in further detail below.

**Task 1:**

**GRASP patterns applied:**

*Cup class creation: pure fabrication.*

*Methods inside Cup class: information expert.*

Through our study of the base code for the program, it was understood that the ***NavigationPane*** 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 with pure fabrication and at the same time improved the modularity and cohesion of our program.

**Task 2:**

**GRASP patterns applied:**

*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 (i.e. rolled value is equal to number of dice). 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) 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. The consequences of this were that our solution effectively solved the tasks requirements and worked higher cohesion into the code.

**Task 3:**

**GRASP patterns applied:**

*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 position 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 in other words we took advantage of the polymorphism of this part of the code.

**Task 4:**

**GRASP patterns applied:**

*Why create* ***Strategy*** *interface: Protected variation and indirection.*

*Creation of* ***Strategy*** *object inside Puppet class: Creator.*

**a, b:**

An ‘isReversed’variable is added to the ***Connection*** class which indicates whether the connection is reversed or not, when ‘isReversed’ is true, the starting location and ending location are exchanged compared 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 achieves the same functionality, but instead of changing ‘isReversed’ back-and-forth, it can assign ‘isReversed’ directly to true or false, this method is used for the toggling strategy button.

**c:**

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

**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 in 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 the 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 that implements the Strategy interface or change the methods in the implementation class. The rest of the system only cares about methods declared in the interface, which protects the rest of the 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 instances of ***Strategy*** class), therefore in the future if the developers want to apply a different strategy to different puppets, our design still works.

For calling of ***doStrategy*** method, we call that at the end of ***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 applies information expert pattern.

**Task 5:**

**GRASP patterns applied:**

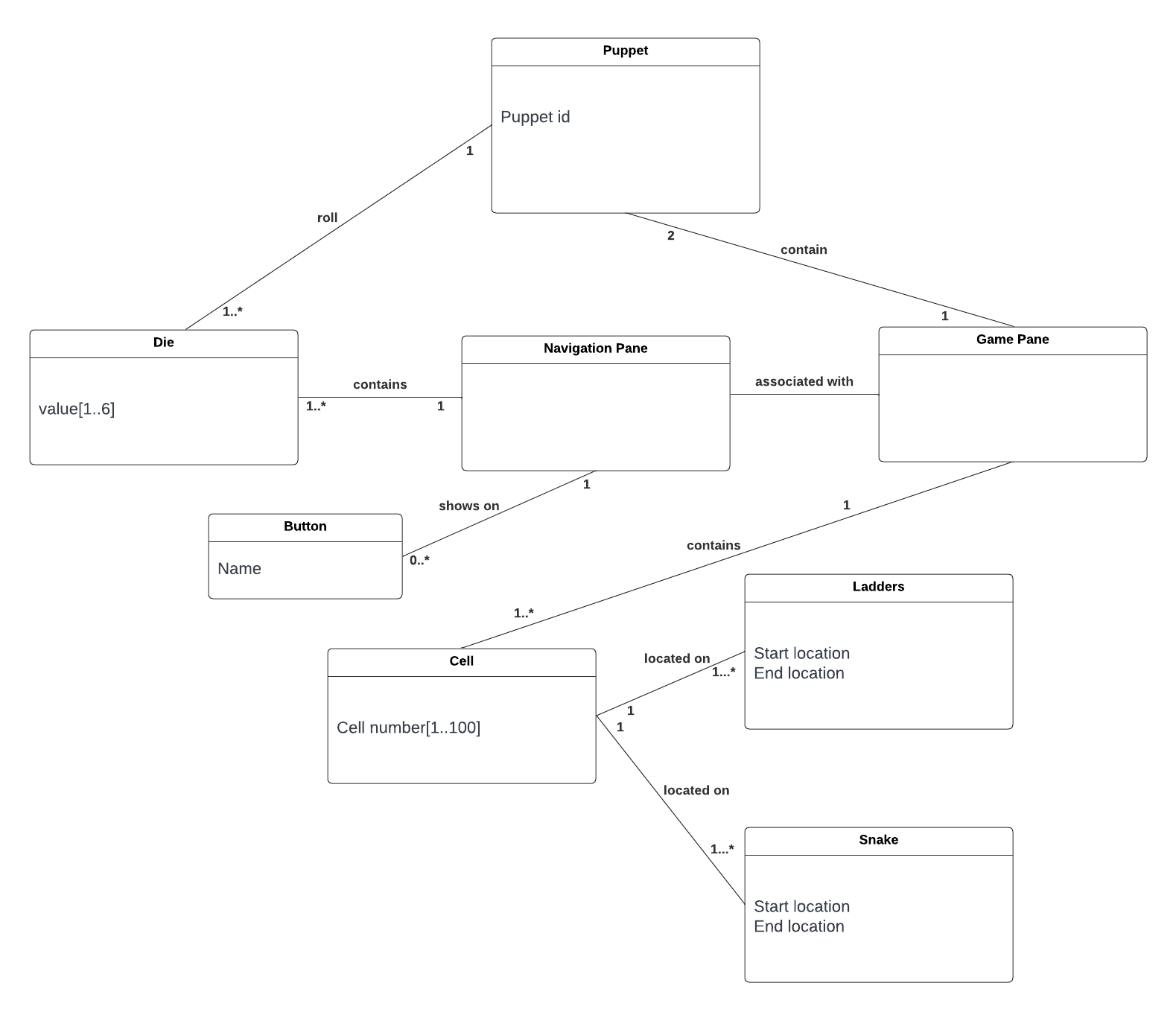
*Why create* ***Statistics*** *class: high cohesion, low coupling.*

*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 S***tatistics*** 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*** are not 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.

**Domain model**



**Design Class Model**

