# SWEN30006 Project 1 - Snakes and Ladders On a Plane

Workshop 15, Team 14
James La Fontaine 1079860
Yun Keng Leong 1133704
(2 member team)

Our goal was to try to preserve much of the original functionality of the program by making minimal modifications. However, we still found it to be appropriate to refactor some of the already existing features while adding our own methods, attributes and classes to accommodate new functionality. The classes which we added include: Strategy (SimpleStrategy), StrategyInitialiser, Statistic (DieValueStatistic & PathTraversalStatistic), and StatisticInitialiser.

## Part 1: Problem Domain to Domain Model

We ensured that we had a sound understanding of the problem before beginning by constructing a partial domain model (see Figure 1) to cover all the aspects of the Snakes and Ladders Game.

## Part 2: Domain Model to Design Model

Our partial design diagram (see Figure 2) depicts the actual implementation alongside the changes that we made, with new methods and attributes added to existing modules being listed below already existing ones. The modifications and justifications for the design will be discussed in more detail below.

### 2.1 - Using multiple dice

Using the principle of information expert, we chose to implement the logic which allows multiple dice to function within the NavigationPane class as this is the class which contains the required information and methods for rolling dice. This allowed us to maintain the same level of cohesion that was already present in the design by continuing to make use of NavigationPane as a class which deals with the rolling of dice - and by extension the handling of turns. We firstly decided to add the attributes nbDice, totalRoll and currentDie to the NavigationPane class to allow the tracking of these values for this task. We also decided to refactor the startMoving method to be called only once at the end of all dice rolls within a single turn in NavigationPane instead of being called for each instance of Die. This allowed us to then simply input our totalRoll attribute into the startMoving method and thus allow the program to still function as it already had and correctly move the player as if multiple dice have been rolled.

Within a new method called rollAllDice(), we implemented a loop to retrieve and subsequently roll multiple die values for each die that was being used and thus were able to utilise the methods which had already been created without having to significantly modify the functionality of NavigationPane. We also reworked the logic for the dieValue array retrieval in getDieValue() to still function correctly for multiple dice without having to greatly modify the functionality of the class. Furthermore we were able to change the roll() method to only remove Die instances at the end of each turn rather than after each roll. Finally, we refactored the information about each roll that was displayed in the console output by the startMoving() method into a displayRollInformation() method which now prints roll information after each roll while accommodating multiple dice and additionally allows for easier reproduction of this particular output if desired at a different point in the game.

## 2.2 - Not traversing down on the minimum roll value

Again utilising the principle of information expert, we deemed it to be most suitable to implement this functionality within the Puppet class as this is where the movement of a particular puppet over connections is handled. Therefore, this also allowed us to maintain the cohesion already existent within the Puppet class while continuing to keep coupling low across the program. We refactored some of the code that handled movement originally in the act() method into separate methods called moveOnConnection() and checkCurrentConnection() to support future reuse of these processes.

We then made an addition to the logic in the checkCurrentConnection() method to also check whether a player rolled the minimum roll when landing on a down path. This was implemented by retrieving the nbDice and totalRoll from NavigationPane and checking whether they are equal. If a player had landed on a downward connection but made the minimum roll, we setActEnabled to false and currentCon to null, then we call the prepareTurn() (previously named prepareRoll()) method in NavigationPane to end the player’s turn as if they hadn’t landed on a connection.

## 2.3 - Pushing the opponent back

Similarly to the previous task, it seemed appropriate to implement functionality for pushing an opponent back within the Puppet class as this class deals with puppet movement. We delegated the responsibility of checking if a player has landed on the same square as their opponent to a new method called checkOpponentSquare(). This method is called after a player has finished moving the appropriate number of places but before their own connection is checked.  We make use of a boolean called conTraversed to track whether the player has reached their final square by using a connection.

This method also uses our new getter in GamePane, getOpponentPuppet(), to retrieve the opponent puppet through use of already existing data and determine whether the opponent is sitting on the same square as where the player has landed without using a connection. If this has happened, then we move the opponent back one cell via using the moveToPreviousCell() method inspired by moveToNextCell(). The movingBack boolean allows us to reuse the code within checkCurrentConnection() and moveOnConnection() and apply it to the opponent puppet without ending the turn by guarding the prepareTurn() calls with a check of movingBack.

After checkCurrentConnection() is called for the opponent puppet, we utilised the setActEnabled() method to animate the opponent puppet via the act() method if they need to traverse a connection after their push back.

## 2.4a - Toggling the direction of connections

We decided to delegate the responsibility of actually switching connections to a new switchConnection() method which, by information expert, should be implemented in the Connection class as this is where the data is stored with which we can infer the direction of a connection. This maintains the high cohesion of the Connection class by continuing to keep it focused on data and manipulation of game connections. This method simply swaps the locEnd and locStart, and the cellStart and cellEnd of a connection instance.

We then implemented a toggleMode() method in NavigationPane (as NavigationPane stores the current status of the toggle) and call switchConnection() for all connections on the board when the player decides to toggle the mode.

## 2.4b - Toggle mode strategy

We decided that strategies should be implemented in their own Strategy class to support high cohesion and low coupling across the design. Through the use of polymorphism, we created a design where future developers can easily create their own implementations of a strategy for toggling the mode by inheriting the abstract Strategy superclass. They must simply implement the logic for checkStrategy() and return a true or false if the relevant conditions are satisfied. Methods were created in the Strategy superclass which provide access to important game data which a strategy may use. These methods provide easier access to this data and also reduce the coupling between Strategy implementations and the other classes in which this data is stored: If a method or functionality for retrieving certain data is altered, then the appropriate change must simply be made in the parent class for correct data retrieval to be maintained across all subclasses.

The Creator pattern was applied to make a StrategyInitialiser class which has the responsibility of initialising the strategy that will be used during a game. This class simultaneously gives developers an easy method of substituting their implemented strategy into the program by simply adding it to the StrategyInitialiser constructor.

The StrategyInitialiser is used by the NavigationPane at the start of the game to initialise the selected strategy in the initialiseStrategy() method called by FrameTutorial. Then the selected strategy's checkStrategy() method is called at the end of every turn in the prepareTurn() method. Our implementation of the specified strategy was achieved by extending the Strategy superclass through the SimpleStrategy subclass and adding appropriate logic to the checkStrategy() method while using the data retrieval methods supplied by Strategy.

## 2.5 - Statistics

Similarly to our solution for implementing strategies, we decided to create a separate Statistic class to again support high cohesion and low coupling. The Statistic class functions very similarly to the Strategy class in that it is an abstract superclass with data retrieval methods which support low coupling across the design. Polymorphism is utilised here again as developers are intended to implement their own statistics as subclasses of Statistic. Any future developer simply has to implement logic for the initialiseStatistic(), trackStatistic() and printStatistic() method in the appropriate manner.

The creator pattern was also applied here to create the StatisticInitialiser class in which a developer constructs their desired statistics and adds them to the statistics array list in the StatisticInitialiser constructor.

NavigationPane again uses the StatisticInitialiser to initialise the statistics at the start of the game by calling each specified statistic's initialiseStatistic() method in the initialiseStatistic() method called by FrameTutorial. Then each statistic in the statistics list has its trackStatistic() method called at the end of every turn. Finally, displayStatistic() is called at the end of the game to print the results to the console. Our implementations of the specified statistics were achieved by extending the Statistic superclass through the DieValueStatistic and PathTraversalStatistic subclass and adding appropriate logic to the initialiseStatistic(), trackStatistic and printStatistic methods while using the data retrieval methods supplied by Statistic.

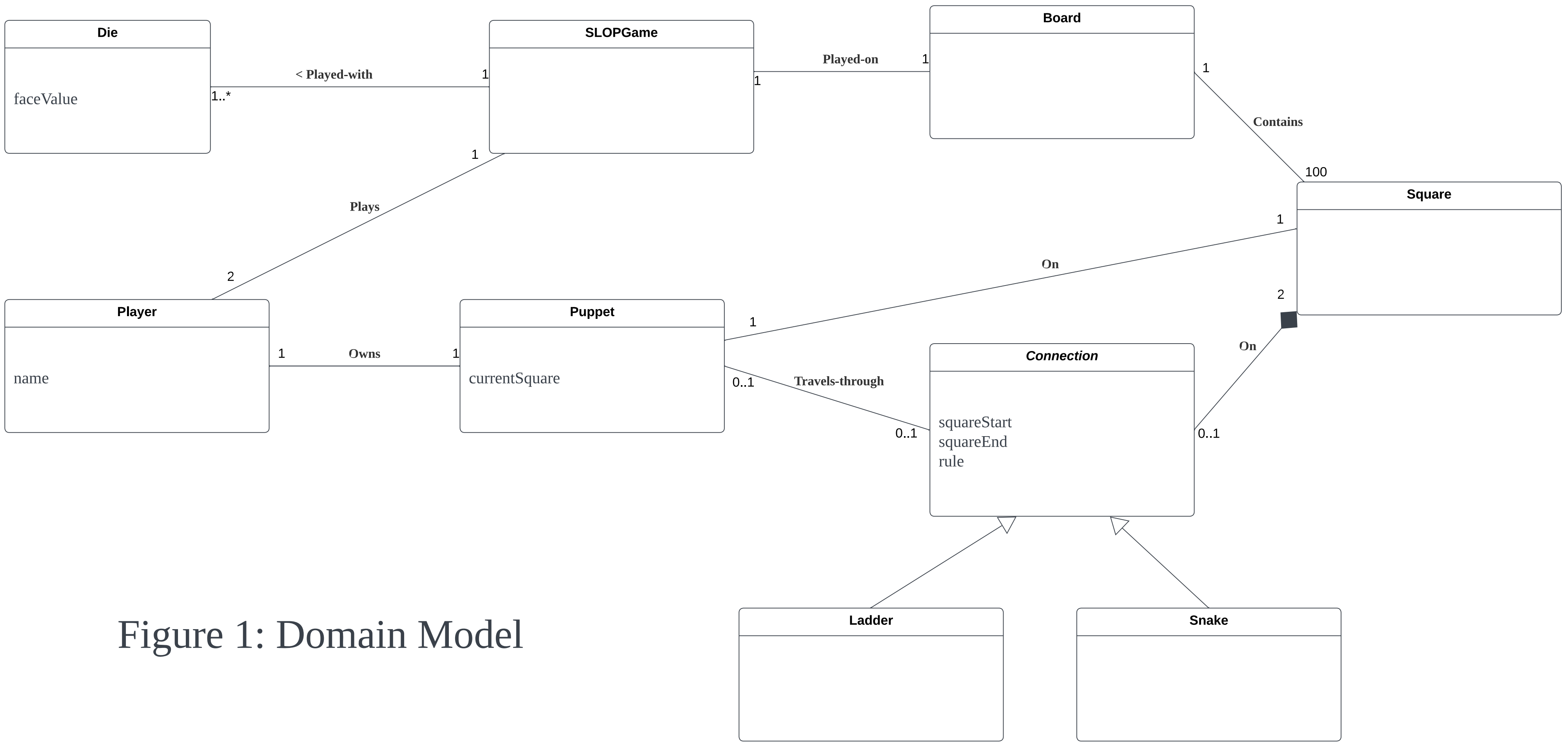

Figure 1: Domain Model

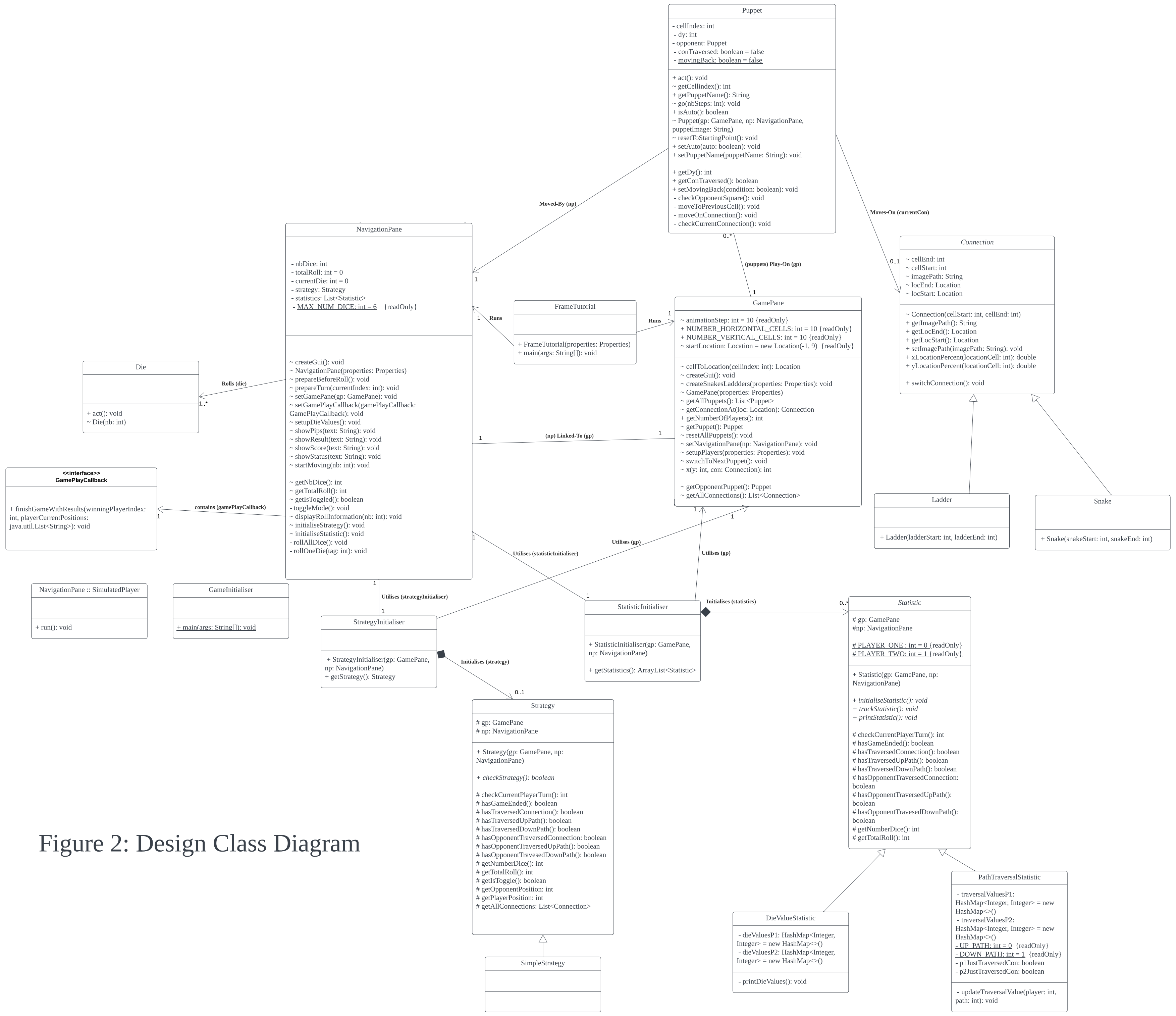
Puppet
- cellIndex: int
- dy: int
- opponent: Puppet
- conTraversed: boolean = false
- movingBack: boolean = false
+ act(): void
~ getCellindex(): int
+ getPuppetName(): String
~ go(nbSteps: int): void
+ isAuto(): boolean
~ Puppet(gp: GamePane, np: NavigationPane, puppetImage: String)
~ resetToStartingPoint(): void
+ setAuto(auto: boolean): void
+ setPuppetName(puppetName: String): void
+ getDy(): int
+ getConTraversed(): boolean
+ setMovingBack(condition: boolean): void
- checkOpponentSquare(): void
- moveToPreviousCell(): void
- moveOnConnection(): void
- checkCurrentConnection(): void
Moved-By (np)
Moves-On (currentCon)
0..*
(puppets) Play-On (gp)
1
0..1
Connection
~ cellEnd: int
~ cellStart: int
~ imagePath: String
~ locEnd: Location
~ locStart: Location
~ Connection(cellStart: int, cellEnd: int)
+ getImagePath(): String
+ getLocEnd(): Location
+ getLocStart(): Location
+ setImagePath(imagePath: String): void
+ xLocationPercent(locationCell: int): double
+ yLocationPercent(locationCell: int): double
+ switchConnection(): void
NavigationPane
- nbDice: int
- totalRoll: int = 0
- currentDie: int = 0
- strategy: Strategy
- statistics: List&lt;Statistic&gt;
- MAX_NUM_DICE: int = 6 {readOnly}
~ createGui(): void
~ NavigationPane(properties: Properties)
~ prepareBeforeRoll(): void
~ prepareTurn(currentIndex: int): void
~ setGamePane(gp: GamePane): void
~ setGamePlayCallback(gamePlayCallback: GamePlayCallback): void
~ setupDieValues(): void
~ showPips(text: String): void
~ showResult(text: String): void
~ showScore(text: String): void
~ showStatus(text: String): void
~ startMoving(nb: int): void
~ getNbDice(): int
~ getTotalRoll(): int
~ getIsToggled(): boolean
- toggleMode(): void
~ displayRollInformation(nb: int): void
~ initialiseStrategy(): void
~ initialiseStatistic(): void
- rollAllDice(): void
- rollOneDie(tag: int): void
1
Runs
1
FrameTutorial
+ FrameTutorial(properties: Properties)
+ main(args: String[]): void
Runs
1
GamePane
~ animationStep: int = 10 {readOnly}
+ NUMBER_HORIZONTAL_CELLS: int = 10 {readOnly}
+ NUMBER_VERTICAL_CELLS: int = 10 {readOnly}
~ startLocation: Location = new Location(-1, 9) {readOnly}
~ cellToLocation(cellindex: int): Location
~ createGui(): void
~ createSnakesLaddders(properties: Properties): void
~ GamePane(properties: Properties)
~ getAllPuppets(): List&lt;Puppet&gt;
~ getConnectionAt(loc: Location): Connection
+ getNumberOfPlayers(): int
~ getPuppet(): Puppet
~ resetAllPuppets(): void
~ setNavigationPane(np: NavigationPane): void
~ setupPlayers(properties: Properties): void
~ switchToNextPuppet(): void
~ x(y: int, con: Connection): int
~ getOpponentPuppet(): Puppet
~ getAllConnections(): List&lt;Connection&gt;
Die
+ act(): void
~ Die(nb: int)
Rolls (die)
1..*
1
(np) Linked-To (gp)
1
&lt;&lt;interface&gt;&gt;
GamePlayCallback
+ finishGameWithResults(winningPlayerIndex: int, playerCurrentPositions: java.util.List&lt;String&gt;): void
contains (gamePlayCallback)
1
Ladder
+ Ladder(ladderStart: int, ladderEnd: int)
Snake
+ Snake(snakeStart: int, snakeEnd: int)
1
1
Utilises (gp)
Utilises (gp)
1
Utilises (statisticInitialiser)
1
Utilises (strategyInitialiser)
1
1
NavigationPane :: SimulatedPlayer
+ run(): void
GameInitialiser
+ main(args: String[]): void
StrategyInitialiser
+ StrategyInitialiser(gp: GamePane, np: NavigationPane)
+ getStrategy(): Strategy
Initialises (strategy)
0..1
StatisticInitialiser
+ StatisticInitialiser(gp: GamePane, np: NavigationPane)
+ getStatistics(): ArrayList&lt;Statistic&gt;
Initialises (statistics)
0..*
Statistic
# gp: GamePane
#np: NavigationPane
# PLAYER_ONE : int = 0 {readOnly}
# PLAYER_TWO: int = 1 {readOnly}
+ Statistic(gp: GamePane, np: NavigationPane)
+ initialiseStatistic(): void
+ trackStatistic(): void
+ printStatistic(): void
# checkCurrentPlayerTurn(): int
# hasGameEnded(): boolean
# hasTraversedConnection(): boolean
# hasTraversedUpPath(): boolean
# hasTraversedDownPath(): boolean
# hasOpponentTraversedConnection: boolean
# hasOpponentTraversedUpPath(): boolean
# hasOpponentTravesedDownPath(): boolean
# getNumberDice(): int
# getTotalRoll(): int
Strategy
# gp: GamePane
# np: NavigationPane
+ Strategy(gp: GamePane, np: NavigationPane)
+ checkStrategy(): boolean
# checkCurrentPlayerTurn(): int
# hasGameEnded(): boolean
# hasTraversedConnection(): boolean
# hasTraversedUpPath(): boolean
# hasTraversedDownPath(): boolean
# hasOpponentTraversedConnection: boolean
# hasOpponentTraversedUpPath(): boolean
# hasOpponentTravesedDownPath(): boolean
# getNumberDice(): int
# getTotalRoll(): int
# getIsToggle(): boolean
# getOpponentPosition: int
# getPlayerPosition: int
# getAllConnections: List&lt;Connection&gt;
SimpleStrategy
DieValueStatistic
- dieValuesP1: HashMap&lt;Integer, Integer&gt; = new HashMap&lt;&gt;()
- dieValuesP2: HashMap&lt;Integer, Integer&gt; = new HashMap&lt;&gt;()
- printDieValues(): void
PathTraversalStatistic
- traversalValuesP1: HashMap&lt;Integer, Integer&gt; = new HashMap&lt;&gt;()
- traversalValuesP2: HashMap&lt;Integer, Integer&gt; = new HashMap&lt;&gt;()
- UP_PATH: int = 0 {readOnly}
- DOWN_PATH: int = 1 {readOnly}
- p1JustTraversedCon: boolean
- p2JustTraversedCon: boolean
- updateTraversalValue(player: int, path: int): void

Figure 2: Design Class Diagram

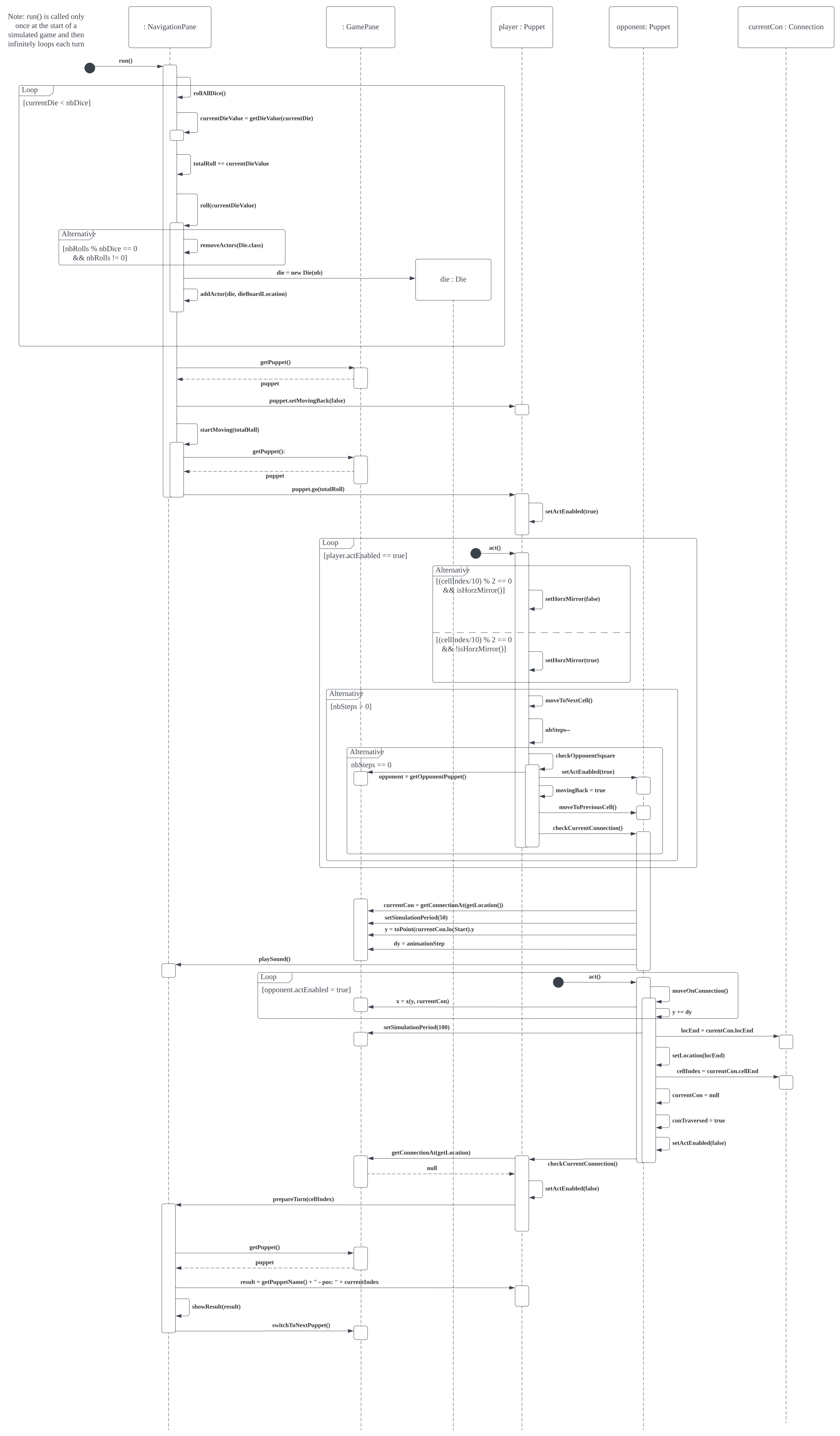

Figure 3: Design Sequence Diagram