FIT3077
Software Architecture & Design

Composite

Basic Architecture

Architecture Design

In the context of JavaFX's Model-View-Controller (MVC) architecture, the Model encapsulates both the application data and business logic. The View is accountable for the graphical user interface (GUI) that exhibits the application's current state. On the other hand, the Controller, which has access to both the Model and the View, manages user inputs, updates the Model, and mirrors these changes in the View concurrently.

Our Nine Men’s Morris game is structured around this MVC architecture. This choice was motivated by the MVC architecture's clear separation between the game's logic, GUI design, and user interactions. This clear separation allows us to replace or extend any component independently, enhancing the overall maintainability and extensibility of the system.

IntelliJ IDEA Ultimate, our chosen integrated development environment (IDE), natively supports Scene Builder, a tool we use to construct our FXML file (main.fxml), which outlines the game’s View. The game's Model is housed within the edu.monash.game package. The Controller component is represented by classes outside this package, specifically those in the edu.monash package.

The game Model operates independently of the ViewController's implementation. However, the ViewController supervises the game's lifecycle and employs it for the game's business logic. This is the reason that the ViewController lies outside of the game engine’s package.

The structural design of our game is similar to those found in native mobile application frameworks like Android. These frameworks typically include Views, typically delineated as XML files, and Controllers, represented by ViewControllers. The ViewControllers are responsible for processing user inputs and updating the GUI. Simultaneously, the Model embodies the data and the methods essential for manipulating this data. Often, this data is kept in sync with cloud databases. In the context of our game, the game engine, located within the edu.monash.game package, fulfils this role.

Even though the architectural design hasn't changed since the previous sprint, we received feedback in Sprint 2 that suggested we needed to keep the Model, View and Controller as distinct packages. This feedback underlined the need for us to explicitly justify our architectural decisions.

Player Phase Transitioning

In Nine Men's Morris, gameplay unfolds through four independent phases that each player progresses through *independently*. They are each represented by a concrete implementation of PlayerPhase.

1. **Placement Phase (PlacePhase)**: The game starts with both players taking turns to place their pieces on the game board.
2. **Slide Phase (SlidePhase)**: After all pieces have been placed, players take turns moving their pieces along the lines of the board.
3. **Jump Phase (JumpPhase)**: When a player has exactly 3 pieces left on the board, they are allowed to move their pieces freely without restrictions.
4. **Terminal Phase**: The game concludes when a player has fewer than three pieces left on the board, resulting in their loss. More on this phase later.

It is important to note that players can be in different phases simultaneously. For example, one player could be in the SlidePhase after placing their pieces on the board, while the other is in the JumpPhase because they have exactly three pieces remaining. Therefore, we have each Player know their current PlayerPhase as a composite relationship, instead of having the game know the current "GamePhase".

Each of the above represents a unique state of the player, reminiscent of state transitions in a Finite Automaton (FA), though only sequential. However, the design must also account for potential non-linear phase transitions in future iterations. Utilising explicit case analysis for such additions would result in an exponentially growing number of cases, which would consequently make the method impossible to close for modification. To circumvent this, we adopted the State Pattern in designing the PlayerPhase interface. This pattern emulates FA's state transitions (including non-sequential transitions), enabling the Player class to remain closed for modifications, while new PlayerPhases can be extended as needed, adhering to the Open-Closed Principle (OCP).

Additionally, at any point of time, if a player forms a mill, they can remove one of their opponent's pieces from the board, regardless of the phase they are in. However, do notice that this behaviour is agnostic of the player's current phase. Therefore, it is not embedded within PlayerPhase.

A package named edu.monash.game.players contains the Player and PlayerPhase classes. External entities only need to know that a Player can determine if they are permitted to make a specific Move at any point in time, through the Player.canPerformMove(Board board, Move move) method. They do not and should not need to be aware of the existence of PlayerPhase.

Two critical methods in each PlayerPhase are validate(Board board, Move move) and transition(). The validate method checks if a move is possible based on the current state of the board, while transition updates the player's PlayerPhase under certain conditions. These methods are only accessible within the edu.monash.game.players package. The

Player.canPerformMove method exposes the validation logic for entities outside of the package to test for Move validity.

In the previous iteration, the lack of a PlayerPhase.transition method made it impossible to transition between different PlayerPhases. Additionally, the prior design didn't adequately account for the terminal phase, signifying a player's loss. To rectify these issues in the current iteration, we introduced a terminal phase represented as null. When a player no longer possesses a PlayerPhase, it indicates they have lost the game. For instance, when a player has fewer than three pieces left, the transition method of JumpPhase will update the associated player's PlayerPhase to null.

View Controller & Actions

The Controller component of the MVC architecture is represented by the ViewController, its related classes, and the Actions in our setup.

The ViewController is connected to the Game, which holds a reference to the Model component (an instance of Game), and is responsible for updating the GUI, or the View component defined by the FXML file. The ViewController only interacts with the Game when the user interface receives an input that changes the game state. Such inputs could be placing a piece on the board (via PlaceAction), moving pieces along the game lines (MoveAction), or removing an opponent's pieces (RemoveAction).

When such an action occurs, the relevant Action is generated and then executed on the game by calling the game.execute(action) method. This method returns a boolean value indicating the success or failure of the Action's execution. With this information, the ViewController updates the GUI accordingly. The key aspect here is that the ViewController doesn't need to know how the Game engine operates or how the Actions are executed. Its sole responsibility is to create the appropriate Actions based on GUI events, and update the GUI based on the success of these Actions.

If needed, the Action interface provides a method isValid that tells whether the Action will be executed or not when game.execute(action) is called. This allows the ViewController to make preparations for the GUI updates.

In future iterations, this strategy will be extended to game loading and saving mechanisms through the implementation of LoadAction and SaveAction. These will be aided by a privately managed Deserializer and Serializer, respectively.

ViewController

Originally, the design was to have the ViewController manage all the logic related to the Controller component of the MVC architecture. However, this approach risked creating a "god class" that is responsible for too many things. This directly violates the Single Responsibility Principle (SRP), making any future extensions to the class difficult. It also indirectly breached the Open-Closed Principle (OCP), as the ViewController would always be open to modifications.

To address this, we decided to divide the responsibilities of the ViewController among three classes during this sprint: GameBoardGridPane, PlayerHandGridPane, and ViewController. GameBoardGridPane represents the game board, while PlayerHandGridPane stands for the players' hands displayed on the sides of the GameBoardGridPane. Both of these classes are subclasses of the JavaFX GridPane class. The remaining logic, like starting a new game or closing the application via the menu bar, is handled by the ViewController. Doing so helps achieve distributed intelligence.

While there is shared logic between GameBoardGridPane and PlayerHandGridPane, and having one subclass the other might seem to uphold the Don't Repeat Yourself (DRY) principle to the fullest extent, we believe it would be an overreach. It introduces a significant flaw - pieces can be dragged from a player's hand, but they cannot be dragged to a player's hand. This forces the PlayerHandGridPane to adopt event handling logic from the GameBoardGridPane and vice versa, which breaks the Liskov Substitution Principle (LSP) and leads to unwanted behaviour in the subclasses.

Our current design, despite minor violations of the DRY principle, ensures a clear segregation of logic between the GridPanes. Additional logic can be incorporated into the custom classes as methods, and they can be independently replaced, enhancing flexibility and maintainability.

Usability

Usability is of utmost importance in game design. A game lacking in usability will lose players to competing games. Thus, usability is the key to retaining players and fostering their continued engagement.

In JavaFX, the drag-and-drop functionality is represented through a sequence of events, each with its own event handler. These handlers can be set on the Nodes using the methods setOnDragDetected, setOnDragOver, setOnDragDropped and setOnDragDone. During our team meetings, we debated the use of Actions in the ViewController.

In our first design, the onDragOver method was set to accept any event, ensuring it would continue propagating to the next method in the sequence, onDragDropped. Here, the user releases the mouse click, triggering the creation of an appropriate Action which is then executed via game.execute(action). If the Action is invalid, the game won't execute it and will return false. However, this design has a usability flaw. Since onDragOver accepted the event regardless of its validity, if the Action failed to execute in onDragDropped, there would be no animation to indicate the invalidity of the move to the user. The piece would simply snap back to its original position, potentially leading to player confusion.

Given that we implemented Actions using the Command Pattern, allowing for their serialisation and deserialization, we decided to incorporate a validity check in the onDragOver method using Action.isValid. Now, every time a player drags a piece to a potential drop location, an appropriate Action is created and validated against the current game state. If the Action is invalid, the event will not be accepted and its propagation will be stopped. This way, if the player attempts to drop the piece, an animation will smoothly return it to its original position, clearly showing the move's invalidity. If the Action is valid, it will be serialised and stored in the JavaFX DragEvent Dragboard and be passed to the onDragDropped handler. The onDragDropped event handler will deserialize it and execute it. If the onDragDropped handler is invoked, it means the Action is valid, and it will execute it with game.execute(action).

The downside of this method is the added cost of serialisation and deserialisation of Actions, and the validity test on hovering over each droppable Node. However, the application is very lightweight and is designed to run on modern computers, so it is not a huge compromise.

Our team has chosen to implement the advanced requirement of game state saving and loading. This decision means our version of Nine Men’s Morris won't have a comprehensive tutorial, only a written guide, as the assignment specifications only allow for one advanced requirement. Despite this, we aim to ensure that the game rules are intuitive, enabling players to learn and progress through the game naturally, reducing the potential for confusion.

Dependency Management & Project Building

The motto of Java, "Write Once, Run Anywhere" (WORA), illustrates its capacity to be developed and deployed on any platform, thanks to the Java Virtual Machine (JVM) which functions as an abstraction layer above the underlying system architecture, making it architecture-independent.

Even though our team uses diverse operating systems, such as macOS running on ARM-based RISC CPUs and 64-bit Windows running on CISC architecture, we need not worry about the compatibility of external packages across different platforms due to Java's platform-independent nature.

Nevertheless, for the sake of reproducibility and to reduce the potential debugging effort within our limited project timeframe, we need to ensure consistent usage of the same versions of external libraries. This is where Gradle proves invaluable. Gradle enables us to precisely define the versions of the external packages we use. It retrieves the necessary packages from a designated repository, in our case, Maven Central, tailored for the target platform. This capability relieves us of debugging stress related to dependency version discrepancies, allowing us to focus on resolving issues within our code. This reproducibility aspect is particularly crucial given our asynchronous coding setup amidst other academic commitments.

Moreover, should we need to incorporate more packages into our Nine Men’s Morris project, Gradle simplifies this process. By specifying the package name and its version, Gradle takes care of the rest. Dependency extensibility is achieved.

By incorporating Gradle as our dependency management tool and build system, we have satisfied the non-functional requirements: reproducibility and extensibility.

Lastly, when our project reaches completion at the end of Sprint 4, we will need to compile it into a standalone desktop application for distribution to end-users. Gradle provides us with the means to accomplish this without the need for additional tools in our development workflow. It's capable of building and packaging the application, making it an all-encompassing solution for our project needs.

Human Values

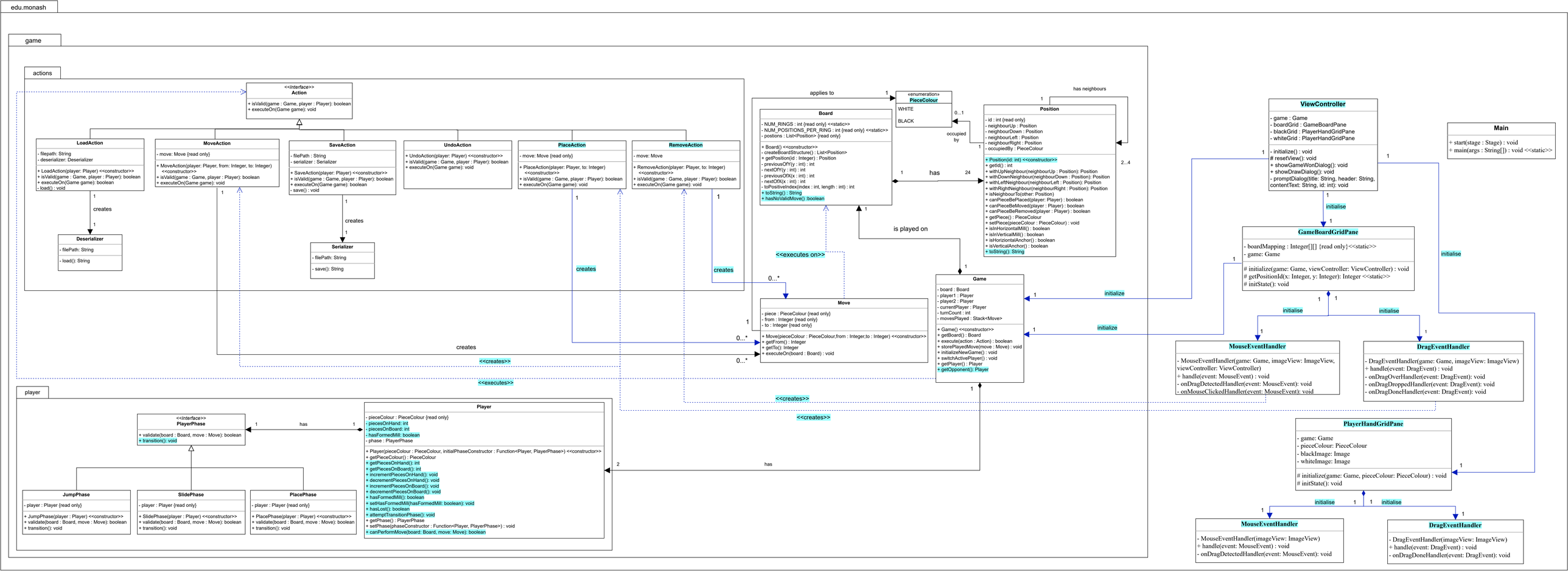
We aligned our design philosophy with Schwartz's theory of human values in designing the Nine Men’s Morris game, focusing on the value of "achievement". This value represents an individual's quest for success, expertise, and demonstration of proficiency. It encompasses setting and achieving goals, mastering skills, and garnering recognition for one's successes.

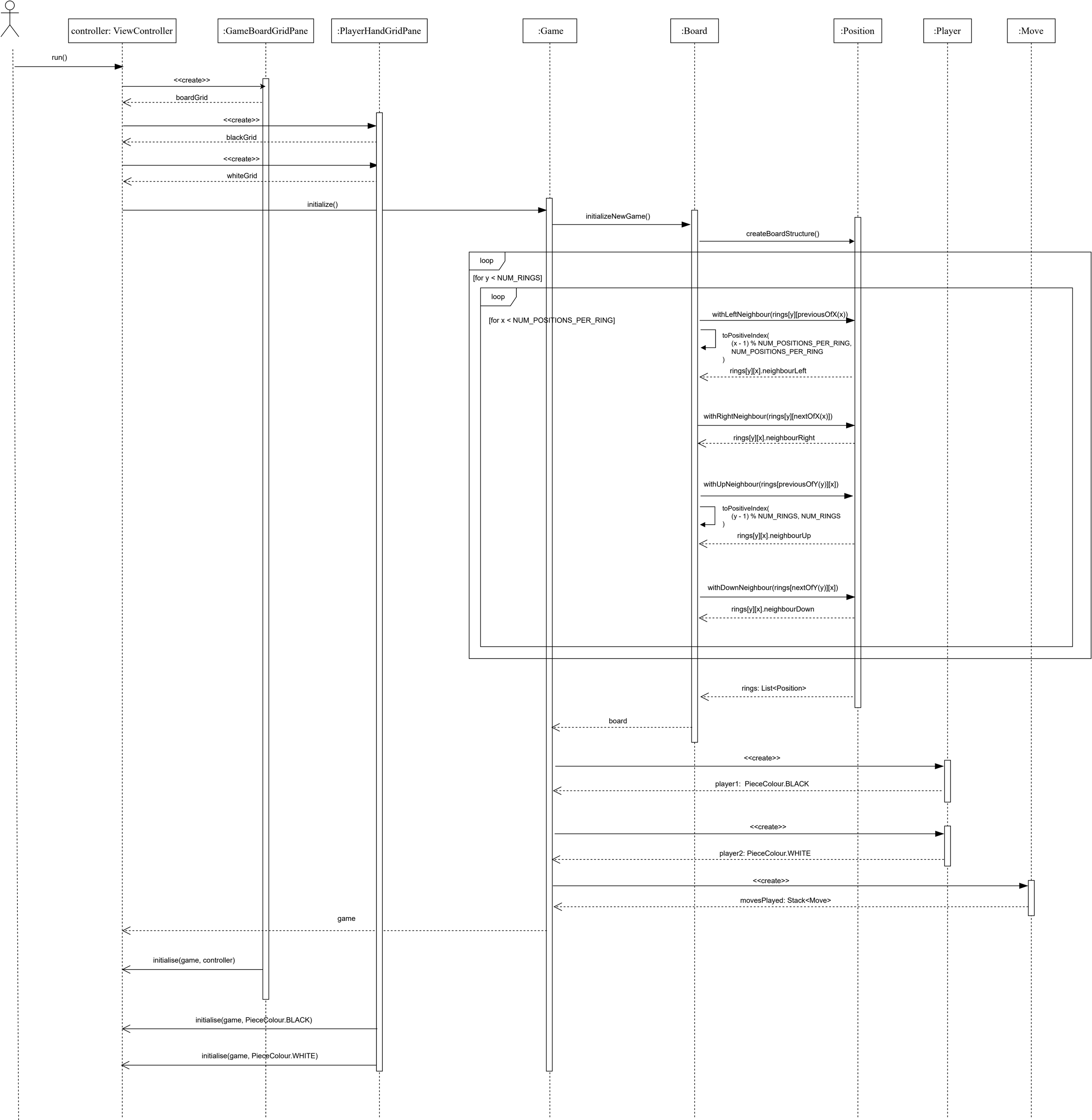
Our design incorporates this value, enhancing the player's sense of advancement, challenge, and gratification. We have embedded the value of achievement into the game's design through:

1. **Goal-driven and competitive gameplay**: Each player has a clear objective - to leave the opponent with less than 3 pieces on the board, thus achieving victory. This objective encourages players to strategize their every move and serves as a constant motivator, driving players towards success and completion.

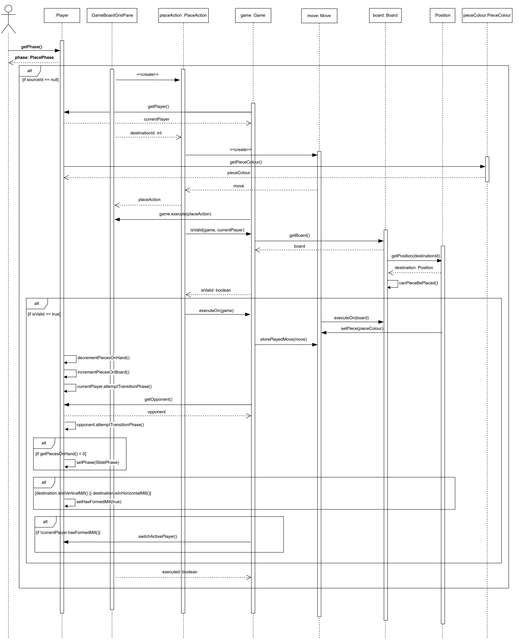
2. **Skill honing**: Our game presents players with opportunities to cultivate and refine their skills. As they engage with the game, players have the chance to improve their strategic thinking, problem-solving capabilities, reflexes, and decision-making skills. By facilitating such growth, the game underscores and reinforces the value of achievement.

Our game caters to the innate human desire for achievement, embedding this value into its very design. This approach not only enhances the player's gaming experience but also imparts a sense of accomplishment when they fulfil their objectives and overcome challenges in the game.

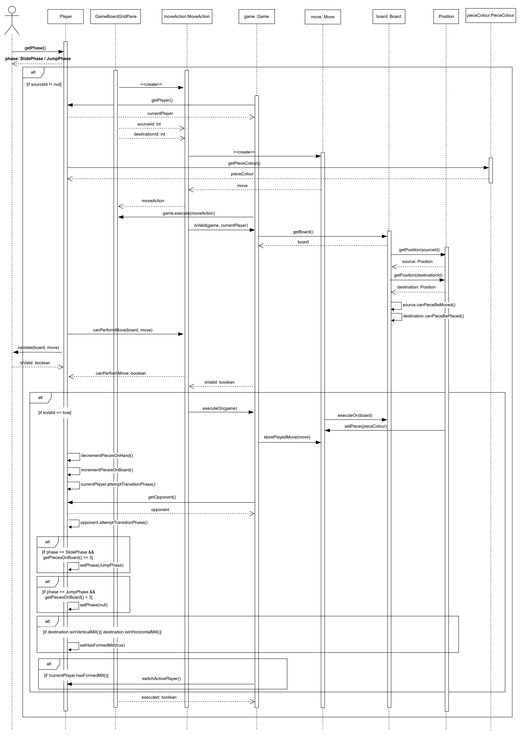

edu.monash
game
actions
<<Interface>>
Action
+ isValid(game : Game, player : Player): boolean
+ executeOn(Game game): void
LoadAction
- filepath: String
- deserializer: Deserializer
+ LoadAction(player: Player) <<constructor>>
+ isValid(game : Game, player : Player): boolean
+ executeOn(Game game): boolean
- load() : void
MoveAction
- move: Move {read only}
+ MoveAction(player: Player, from: Integer, to: Integer) <<constructor>>
+ isValid(game : Game, player : Player): boolean
+ executeOn(Game game): void
SaveAction
- filePath : String
- serializer : Serializer
+ SaveAction(player: Player) <<constructor>>
+ isValid(game : Game, player : Player): boolean
+ executeOn(Game game): boolean
- save() : void
UndoAction
+ UndoAction(player: Player) <<constructor>>
+ isValid(game : Game, player : Player): boolean
+ executeOn(Game game): void
PlaceAction
- move: Move {read only}
+ PlaceAction(player: Player, to: Integer) <<constructor>>
+ isValid(game : Game, player : Player): boolean
+ executeOn(Game game): void
RemoveAction
- move: Move
+ RemoveAction(player: Player, to: Integer) <<constructor>>
+ isValid(game : Game, player : Player): boolean
+ executeOn(Game game): void
creates
Deserializer
- filePath: String
- load(): String
Serializer
- filePath: String
- save(): String
creates
creates
creates
<<creates>>
<<executes>>
applies to
«enumeration»
PieceColour
WHITE
BLACK
Board
- NUM_RINGS : int {read only} <<static>>
- NUM_POSITIONS_PER_RING : int {read only} <<static>>
- postions : List<Position> {read only}
+ Board() <<constructor>>
- createBoardStructure() : List<Position>
+ getPosition(id : Integer) : Position
- previousOfY(y : int) : int
- nextOfY(y : int) : int
- previousOfX(x : int) : int
- nextOfX(x : int) : int
- toPositiveIndex(index : int, length : int) : int
+ toString() : String
+ hasNoValidMove() :boolean
occupied by
has neighbours
Position
- id : int {read only}
- neighbourUp : Position
- neighbourDown : Position
- neighbourLeft : Position
- neighbourRight : Position
- occupiedBy : PieceColour
+ Position(id: int) <<constructor>>
+ getId() : int
+ withUpNeighbour(neighbourUp : Position): Position
+ withDownNeighbour(neighbourDown : Position): Position
+ withLeftNeighbour(neighbourLeft : Position): Position
+ withRightNeighbour(neighbourRight : Position): Position
+ isNeighbourTo(other: Position)
+ canPieceBePlaced(player: Player) : boolean
+ canPieceBeMoved(player : Player) : boolean
+ canPieceBeRemoved(player : Player) : boolean
+ getPiece() : PieceColour
+ setPiece(pieceColour : PieceColour) : void
+ isInHorizontalMill() : boolean
+ isInVerticalMill() : boolean
+ isHorizointalAnchor() : boolean
+ isVerticalAnchor() : boolean
+ toString(): String
has
is played on
<<executes on>>
Move
- piece : PieceColour {read only}
- from : Integer {read only}
- to : Integer {read only}
+ Move(pieceColour : PieceColour,from : Integer,to : Integer) <<constructor>>
+ getFrom() : Integer
+ getTo(): Integer
+ executeOn(board : Board) : void
Game
- board : Board
- player1 : Player
- player2 : Player
- currentPlayer : Player
- turnCount : int
- movesPlayed : Stack<Move>
+ Game() <<constructor>>
+ getBoard() : Board
+ execute(action : Action) : boolean
+ storePlayedMove(move : Move) : void
+ initializeNewGame() : void
+ switchActivePlayer() : void
+ getPlayer() : Player
+ getOpponent(): Player
initialize
initialize
ViewController
- game : Game
- boardGrid : GameBoardPane
- blackGrid : PlayerHandGridPane
- whiteGrid : PlayerHandGridPane
- initialize() : void
resetView(): void
+ showGameWonDialog(): void
+ showDrawDialog(): void
- promptDialog(title: String, header: String, contentText: String, id: int): void
Main
+ start(stage : Stage) : void
+ main(args : String[]) : void <<static>>
initialise
GameBoardGridPane
- boardMapping : Integer[][] {read only}<<static>>
- game: Game
initialize(game: Game, viewController: ViewController) : void
getPositionId(x: Integer, y: Integer): Integer <<static>>
initState(): void
initialise
initialise
initialise
MouseEventHandler
- MouseEventHandler(game: Game, imageView: ImageView, viewController: ViewController)
+ handle(event: MouseEvent) : void
- onDragDetectedHandler(event: MouseEvent): void
- onMouseClickedHandler(event: MouseEvent): void
DragEventHandler
- DragEventHandler(game: Game, imageView: ImageView)
+ handle(event: DragEvent) : void
- onDragOverHandler(event: DragEvent): void
- onDragDroppedHandler(event: DragEvent): void
- onDragDoneHandler(event: DragEvent): void
player
<<Interface>>
PlayerPhase
+ validate(board : Board, move : Move): boolean
+ transition(): void
has
Player
- pieceColour : PieceColour {read only}
- piecesOnHand: int
- piecesOnBoard: int
- hasFormedMill: boolean
- phase : PlayerPhase
+ Player(pieceColour : PieceColour, initialPhaseConstructor : Function<Player, PlayerPhase>) <<constructor>>
+ getPieceColour() : PieceColour
+ getPiecesOnHand(): int
+ getPiecesOnBoard(): int
+ incrementPiecesOnHand(): void
+ decrementPiecesOnHand(): void
+ incrementPiecesOnBoard(): void
+ decrementPiecesOnBoard(): void
+ hasFormedMill(): boolean
+ setHasFormedMill(hasFormedMill: boolean): void
+ hasLost(): boolean
+ attemptTransitionPhase(): void
- getPhase() : PlayerPhase
+ setPhase(phaseConstructor : Function<Player, PlayerPhase>) : void
+ canPerformMove(board: Board, move: Move): boolean
has
JumpPhase
- player : Player {read only}
+ JumpPhase(player : Player) <<constructor>>
+ validate(board : Board, move : Move): boolean
+ transition(): void
SlidePhase
- player : Player {read only}
+ SlidePhase(player : Player) <<constructor>>
+ validate(board : Board, move : Move): boolean
+ transition(): void
PlacePhase
- player : Player {read only}
+ PlacePhase(player : Player) <<constructor>>
+ validate(board : Board, move : Move): boolean
+ transition(): void
<<creates>>
<<creates>>
PlayerHandGridPane
- game: Game
- pieceColour: PieceColour
- blackImage: Image
- whiteImage: Image
initialize(game: Game, pieceColour: PieceColour) : void
initState(): void
initialise
initialise
MouseEventHandler
- MouseEventHandler(imageView: ImageView)
+ handle(event: MouseEvent) : void
- onDragDetectedHandler(event: MouseEvent): void
DragEventHandler
- DragEventHandler(imageView: ImageView)
+ handle(event: DragEvent) : void
- onDragDoneHandler(event: DragEvent): void


controller: ViewController
:GameBoardGridPane
:PlayerHandGridPane
:Game
:Board
:Position
:Player
:Move
run()
<<create>>
boardGrid
<<create>>
blackGrid
<<create>>
whiteGrid
initialize()
initializeNewGame()
createBoardStructure()
loop
[for y < NUM_RINGS]
loop
[for x < NUM_POSITIONS_PER_RING]
withLeftNeighbour(rings[y][previousOfX(x)])
toPositiveIndex(
(x - 1) % NUM_POSITIONS_PER_RING,
NUM_POSITIONS_PER_RING
)
rings[y][x].neighbourLeft
withRightNeighbour(rings[y][nextOfX(x)])
rings[y][x].neighbourRight
withUpNeighbour(rings[previousOfY(y)][x])
toPositiveIndex(
(y - 1) % NUM_RINGS, NUM_RINGS
)
rings[y][x].neighbourUp
withDownNeighbour(rings[nextOfY(y)][x])
rings[y][x].neighbourDown
rings: List<Position>
board
<<create>>
player1: PieceColour.BLACK
<<create>>
player2: PieceColour.WHITE
<<create>>
movesPlayed: Stack<Move>
game
initialise(game, controller)
initialise(game, PieceColour.BLACK)
initialise(game, PieceColour.WHITE)

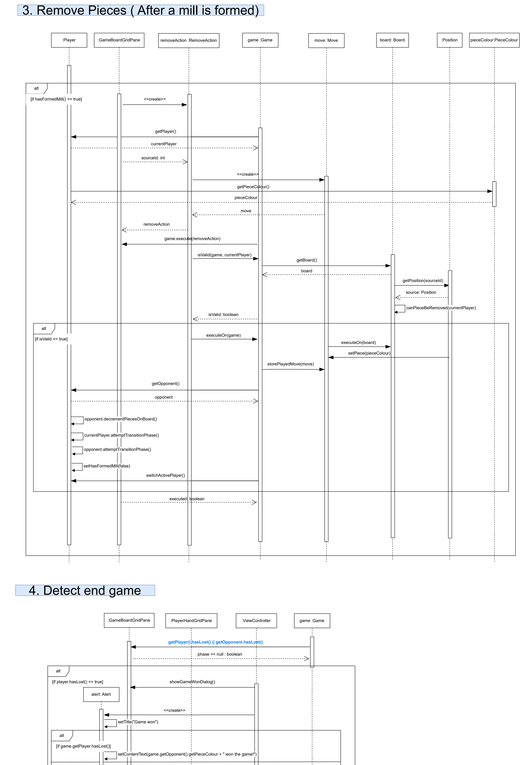
1. Place Pieces (Place Phase)



2. Move Pieces (Slide and Jump Phase)



3. Remove Pieces (After a mill is formed)



4. Detect end game