Group 5

Milestone 2 (A-D)

Group Members:

Andrew Chow

Elman Islam

Akaash Patel

Zach Cymbaluk

Dimitri Papadedes

TABLE OF CONTENTS

Fanorona Game

Subsystem Game View

Subsystem Board Updater

UC06: Win Game (View)

UC09: Undo Move (View)

UC01: Start Game (View)

UC04: Capture Pieces (View)

UC02: Move (View) *

Subsystem Current Player View

Move *

UC02: Capture Move (View)

UC03: Non Capture Move (View)

UC01: Start Game (View)

UC06: Win Game (View)

UC09: Undo Move (View)

UC04: Capture Pieces (View)

UC08: Pause Game (View)

UC07: Forfeit Game (View)

UC05: End Turn (View)

Subsystem Opponent View

UC04: Capture Pieces (View)

UC07: Forfeit Game (View)

Current Active Player

Player

Opponent

User

Subsystem Game Model

UC01: Start Game (Model)

UC02: Capturing Move (Model)

UC03: Non Capturing Move (Model)

UC04: Capture Pieces (Model)

UC05: End Turn (Model)

UC06: Win Game (Model)

UC08: Pause Game (Model)

UC09: Undo Move (Model)

UC07: Forfeit Game (Model)

Subsystem Board

UC01: Start Game (Model)

UC02: Capturing Move (Model)

UC03: Non Capturing Move (Model)

UC04: Capture Pieces (Model)

UC05: End Turn (Model)

UC06: Win Game (Model)

UC08: Pause Game (Model)

UC09: Undo Move (Model)

Subsystem Referee

UC01: Start Game (Model)

UC02: Capturing Move (Model)

UC03: Non Capturing Move (Model)

UC05: End Turn (Model)

UC08: Pause Game (Model)

UC09: Undo Move (Model)

Subsystem Turn Operator

UC01: Start Game (Model)

UC05: End Turn (Model)

UC01: Start a New Game (View)	
Primary Actor	Player
Goal	Begin a new game of Fanorona
Stakeholders	Player, Opponent, View, BoardUpdater
Init Event	The View asks each player to start a new game of Fanorona
Pre-Conditions	Players have decided to play a game of Fanorona
Main Success Scenario	1. The **View** asks the players which color to choose. 2. The **player** and **opponent** choose their color. 3. The **View** tell's the player which player selected which color. 4. The **View** requests a new board from the Model. 5. The **BoardUpdater** displays the new board. 6. The **View** asks the Model whose turn it is. 7. The **View** informs the Player whose turn it is to make the first move.
Alternate Flow	1a. One or both of the players ask what the rules of the game are 1a1. The View asks the Model for the rules 1a2. The View explains the rules of Fanarona
Post-Conditions	The View and Board Updater has successfully set up a game of Fanorona for both players Player who chose white begins the first move.

UC01: Start a New Game (Model)	
Primary Actor	View
Goal	Begin a new game of Fanorona
Stakeholders	Referee, BoardMaster, BoardUpdater
Init Event	The View asked the Model to start a new game of Fanorona
Pre-Conditions	Players have decided to play a game of Fanorona
Main Success Scenario	1. The **Model** receives information about the Player colors. 2. The Model updates the Player color's. 3. The **BoardMaster** sets up the board with the Fanorona game pieces. 4. The **Model** notifies the **View** that the board is ready. 5. The **Model** gives the new board to the **View.**
Alternate Flow	1a. One or both of the players ask what the rules of the game are 1a1. View shows rules
Post-Conditions	The BoardMaster and Model has successfully set up a game of Fanorona for both players Player who chose white begins the first move.

UC02: Capturing Move (View)	
Primary Actor	Player performing the move
Goal	Allow the Player to interact with the game interface to perform a capturing move.

Stakeholders	Player, Opponent, View, BoardUpdater
Init Event	The Player's turn begins, and the View prompts them to select and move a piece on the board.
Pre-Conditions	It is the Player's turn. The Player has at least one valid capture move available.
Main Success Scenario	1. **View** prompts the Player to select a piece to move and the position to move to. 2. **Player** selects a piece and the position they want to move the piece to via the View interface. 3. **View** sends the selected piece and move position change information to the **Model** for validation. 4. **Model** makes the move and captures the pieces 5. The **Model** notifies the **BoardUpdater** to update the board display. 6. **View** requests the updated board state from the Model. 7. **View** displays the updated board using the **BoardUpdater**. 8. **View** asks the Model if additional captures are available. 9. **View** informs the Player if they can make another move. 10. **Player** declines making another move.
Alternate Flow	3a. Player tries to make invalid move: 3a1. View informs the Player that the move is invalid. 3a2. Return to Main Success Scenario at Step 1. 9a. No additional captures are available: 9a1. View informs the Player that their move is complete.

	10a. Player accepts making another move: 10a1. Return to Main Success Scenario at Step 1.
Post-Conditions	The player's capturing move is successfully processed through the View. The board display is updated to reflect the new game state.

UC02: Capturing Move (Model)	
Primary Actor	View
Goal	Process the Player’s capturing move, validate it according to the game rules, update the game state, and provide necessary information back to the view.
Stakeholders	Referee, Boardmaster, BoardUpdater
Init Event	The View sends the Model the Player's selected piece and move direction.
Pre-Conditions	It is the Player's turn. The Player has at least one valid capture move available. The Referee has determined which pieces can be moved.
Main Success Scenario	1. **Model** receives the selected piece and position change from the View. 2. **Referee** validates the move. 3. **Model** makes the move and captures the pieces 4. **Board** updates the game state by moving the piece and capturing the opponent's pieces. 5. **View** is notified to update the board display. 6. **Model** checks for additional captures available to the Player. 7. **Model** informs the View whether the Player can make another move.

Alternate Flow	2a. Referee determines the selected piece is invalid: 2a1. Model informs the View that the piece selection is invalid. 2a2. Use Case ends until a new valid selection is made. 7a. No additional captures are available: 7a1. Model informs the View that the Player's move is complete.
Post-Conditions	The Model has updated the game state to reflect the Player's capturing move. The game state is ready for the next action (either another move by the Player or the Opponent's turn).

UC03: Non-Capturing Move (View)	
Primary Actor	Player performing the move
Goal	Allow the Player to perform a non-capturing move through the game interface.
Stakeholders	Player, Opponent, View, BoardUpdater
Init Event	The Player's turn begins, and the View prompts them to select and move a piece on the board.
Pre-Conditions	It is the Player's turn. No capturing moves are available.
Main Success Scenario	1. **View** prompts the Player to select a piece to move and the place they want to move it to. 2. **Player** selects a piece and the position they want to move the piece to via the View interface. 3. **View** sends the move information to the **Model** for validation. 4. The **Model** makes the move. 5. **View** requests the updated board state from the Model.

	6. **View** displays the updated board using the **BoardUpdater**.
Alternate Flow	3a. Player tries to make invalid move: 3a1. View informs the Player that a capturing move is available and they must select a capturing move. 3a2. Return to Main Success Scenario at Step 1.
Post-Conditions	The Player's non-capturing move is successfully processed through the View. The board display is updated to reflect the new game state.

UC03: Non-Capturing Move (Model)	
Primary Actor	View
Goal	Process the Player's non-capturing move, validate it according to game rules, update the game state, and provide necessary information back to the View.
Stakeholders	Referee, Board, BoardUpdater
Init Event	The View sends the Model the Player's selected piece and move direction.
Pre-Conditions	It is the Player's turn. No capturing moves are available. The Referee has determined which pieces can be moved.
Main Success Scenario	1. **Model** receives the selected piece and position change from **View**. 2. **Referee** checks if any capturing moves are available. 2.1. If capturing moves are available, proceed to UC02. 3. **Referee** validates the move. 4. **Model** makes the move by using **Player** class

	5. **Board** updates the game state by moving the piece. 6. **View** is notified to update the board display.
Alternate Flow	3a. Referee determines the selected piece is invalid: 3a1. Model informs the View that the piece selection is invalid. 3a2. Use Case ends until a valid piece is selected.
Post-Conditions	The Model has updated the game state to reflect the Player's non-capturing move. The game state is ready for the next player's turn.

UC04: Capture Pieces (View)	
Primary Actor	Player capturing
Goal	Allow the Player to perform a capturing move (by Approach or Withdraw) through the game interface.
Stakeholders	Player, Opponent, View, BoardUpdater
Init Event	The Player's turn begins, and the View prompts them to perform a capturing move.
Pre-Conditions	A capture move is possible for the Player. It is the Player's turn.
Main Success Scenario	1. **View** prompts the Player to select a piece to move and the position to move it to. 2. **View** sends the selected piece information to the **Model** for validation. 3. **View** receives confirmation from the Model that the piece selection is valid and captures are possible. 4. **View** receives from the **Model** whether both Capture by Approach and Capture by Withdraw are possible. 5. Only one capture type is possible.

	6. **Player** selects the type of capture (Approach or Withdraw) via the View interface. 7. **View** sends the capture type and move information to the **Model**. 8. **View** receives confirmation from the Model that the move is valid. 9. **View** requests the updated board state from the Model. 10. **View** displays the updated board using the **BoardUpdater**. 11. **View** informs the Player of the captured opponent's pieces.
Alternate Flow	2a. Player selects a piece that cannot capture: 2a1. View informs the Player that the selection is invalid. 2a2. Return to Main Success Scenario at Step 1. 5a. Two capture types are of capture is possible: 6a1. View informs the Player of the available capture type. 6a2. Proceed to Main Success Scenario at Step 7
Post-Conditions	The Player's capturing move is successfully processed through the View. The board display is updated to reflect the new game state. The captured opponent's pieces are removed from the display.

UC04: Capture Pieces (Model)	
Primary Actor	View
Goal	Process the Player's capturing move, validate it according to game rules, update the game state, and provide necessary information back to the View.
Stakeholders	Referee, Boardmaster, BoardUpdater

Init Event	The **View** sends the Model the Player's selected piece, move direction, and capture type (Approach or Withdraw).
Pre-Conditions	A capture move is possible for the Player. It is the Player's turn. The Referee has determined which pieces can be moved.
Main Success Scenario	1. **Model** receives the selected piece and moves direction from the View. 2. **Referee** validates the move 3. **Model** informs the View that the piece selection is valid. 4. **Referee** determines possible capture types (Approach, Withdraw, or both), and the **Model** informs the **View** of available capture types. 5. **Model** receives the Player's selected capture type and move direction from the View. 6. **BoardMaster** updates the game state by moving the piece and removing the captured opponent's pieces. 7. **BoardUpdater** is notified to update the board display. 8. **Referee** records the captured pieces. 9. **Model** provides the View with information about the captured pieces.
Alternate Flow	2a. Referee determines the selected piece is invalid: 2a1. Model informs the View that the piece selection is invalid. 2a2. Use Case ends until a valid piece is selected. 4a. Only one capture type is available: 4a1. Model informs the View of the available capture type. 4a2. Proceed to Main Success Scenario at Step 6.
Post-Conditions	The Model has updated the game state to reflect the Player's capturing move. The opponent's pieces are captured and removed from the game state.

	The game state is ready for the next action (either another move by the Player if additional captures are possible or the Opponent's turn).

UC05: End a Turn (View)	
Primary Actor	Player, Opponent
Goal	Transfer turn control to the other player.
Stakeholders	Player, Opponent, View, Model
Init Event	Player or Opponent ends their turn
Pre-Conditions	The player has made their move and cannot do anything else.
Main Success Scenario	1. **Player** or **Opponent** has finished their turn. 2. Player or Opponent selects **End Turn** **3.** **View** tells the **Model** player has decided to end their turn 4. The **Model** validates that the player has made one move this turn 5. The **Model** switches control to **Opponent.** 6. **View** informs the **Opponent** that it is now their turn.
Alternate Flow	4a. Player has not made a move this turn: 4a1. Model informs the View a move hasn't been made 4a2. View asks the player to make a move first
Post-Conditions	The Player's turn is ended in the View.

	The Opponent is informed that it is their turn. The game display is updated accordingly.

UC05: End a Turn (Model)	
Primary Actor	View
Goal	Process the end of a Player's turn, validate that no additional actions are required, update the game state, and prepare for the next player's turn.
Stakeholders	Referee, Turn Operator, BoardMaster, BoardUpdater
Init Event	The View sends a request to the Model to end the Player's turn.
Pre-Conditions	The Player has indicated they wish to end their turn. The Player has completed at least one move or has passed their turn.
Main Success Scenario	1. **Model** receives the end-turn request from the View. 2. **Referee** (within the Model) validates that the Player has made at least one move. 3. **Model** updates the game state to reflect the change in turn. 4. **Model** notifies the View that the Player's turn has ended. 5. **Model** sets the next player (Opponent) as the active player. 6. **View** is notified to update any necessary display elements.
Alternate Flow	2a. Referee detects Player has not made a move: 2a1. Model informs the View that the Player still needs to make a move.
Post-Conditions	The Model has updated the game state to reflect that the Player's turn has ended. The next player is set as the active player. The game state is ready for the next player's turn.

UC06: Win Game (View)	
Primary Actor	Player, Opponent
Goal	Declare a winner when the **Player** or **Opponent** has no pieces left or valid moves.
Stakeholders	Player, Opponent, Board Updater
Init Event	The **Model** informs the **View** that a winning condition has been met.
Pre-Conditions	The Opponent has one piece remaining A win condition can be met with a valid move
Main Success Scenario	1. The **Player** performs the winning move against the **Opponent.** 2. **View** receives notification from the **Model** about the winner. 3. The **View** informs about the winner. 4. The **View** asks for the board. 5. The **Board Updater** displays the final game state. 6. The **Board Updater** resets the game board to the original state.
Alternate Flow	1a: The Model detects a tie condition. 1a1: The Model informs the View about a draw. 1a2: The View informs about a draw 1a3: The Board Updater displays the final game state.
Post-Conditions	The Fanarona game ends. The Winner is declared, or the game is declared a draw. **View** asks if the Players want to start a new game

UC06: Win Game (Model)	
Primary Actor	View
Goal	Declare a winner when the **Player** or **Opponent** has no pieces left or valid moves.
Stakeholders	Player, Opponent, View, Boardmaster
Init Event	A winning move has been made by the **Player** or **Opponent**
Pre-Conditions	The Opponent has one piece remaining A win condition can be met with a valid move
Main Success Scenario	1. The **Player** performs the winning move against the **Opponent.** 2. The **Model** determines the win condition and identifies the winner. 3. The **Model** passes the information to the **View.** 4. The **Boardmaster** resets the game board to the original state. 5. The **Boardmaster** gives the new board to the **View.**
Alternate Flow	1a: The Referee detects a tie condition. 1a1: The Referee declares a draw. 1a2: The BoardUpdater displays the final game state.
Post-Conditions	The Fanarona game ends. The Winner is declared, or the game is declared a draw.

UC07: Forfeit Game (View)	
Primary Actor	View
Goal	End the game of Fanorona by forfeiting
Stakeholders	Player, Opponent, View
Init Event	The Player or Opponent chooses to forfeit the game of Fanorona
Pre-Conditions	The game is in progress.
Main Success Scenario	1. Player or Opponent tells the View that they would like to forfeit. 2. The **View** confirms the player's decision. 3. **Player** confirms their option forfeit. 4. The Model determines the winner by forfeit and informs the View. 5. The **View** asks the Players if they would like to play again.
Alternate Flow	3a: Player cancels the forfeit decision. 3a1: Player says no to the forfeit decision 3a2: The View resumes the game from the current state 5a: BoardUpdater sets up the game board to start a new game of Fanorona 5a1: View asks the Players to choose their colour 5a2: View tells the Player who chose white to make their first move

Post-Conditions	The game ends. The View declares the winner by forfeit. Players may choose to start a new game or exit.

UC07: Forfeit Game (Model)	
Primary Actor	Player, Opponent, Referee
Goal	End the game of Fanorona by forfeiting
Stakeholders	Player, Opponent, Referee
Init Event	The Player or Opponent chooses to forfeit the game of Fanorona
Pre-Conditions	The game is in progress.
Main Success Scenario	1. Player or Opponent tells the **View** that they would like to forfeit. 2. **Player** confirms their forfeit. 3. **View** passes information to the Model about a forfeit. 4. The **Model** determines the winner by forfeit and informs the **View**. 5. The **Model** passes this info to the View. 6. The **View** asks the Players if they would like to play again.
Alternate Flow	2a: Player cancels the forfeit decision. 3a1: Player says no to the forfeit decision 3a2: The View resumes the game from the current state 6a: Boardmaster sets up the game board to start a new game of Fanorona

	6a1: The View asks for the new board. 6a2: View asks the Players to choose their color 6a3: View tells the Player who chose white to make their first move
Post-Conditions	The game ends. The Referee declares the winner by forfeit. Players may choose to start a new game or exit.

UC08: Pause Game (View)	
Primary Actor	Player requesting the pause
Goal	Allow a player to temporarily pause the game, subject to the opponent's agreement.
Stakeholders	Player, Opponent, Referee, View, Boardmaster
Init Event	Player needs to pause the game due to personal reasons.
Pre-Conditions	The game is in progress.
Main Success Scenario	1. **Player** requests the **View** for a game pause. 2. **View** communicates the pause request to the Opponent. 3. **Opponent** considers the request. 4. **Opponent** agrees to the pause. 5. **View** requests the **Model** to pause the game. 6. **View** announces the game is paused. 7. After the agreed pause duration, the **View** resumes the game. 8. Fanorona game continues from where it was paused.

Alternate Flow	3a: Opponent declines the pause request. 3a1: The View informs the Player of the decision. 3a2: Player chooses to continue playing or forfeit the game.
Post-Conditions	If paused, the game resumes after the pause. If not paused, the game continues or the Player may forfeit.

UC08: Pause Game (Model)	
Primary Actor	Player requesting the pause
Goal	Allow a player to temporarily pause the game, subject to the opponent's agreement.
Stakeholders	Player, Opponent, Referee, Boardmaster
Init Event	Player needs to pause the game due to personal reasons.
Pre-Conditions	The game is in progress.
Main Success Scenario	1. **View** requests the **Model** for a game pause. 2. **Boardmaster** secures the current game state. 3. **Turn Operator** secures the current turn state. 4. After the agreed pause duration, the **View** requests the **Model** to resume the game, and the **Model** resumes the game. 5. Fanorona game continues from where it was paused.

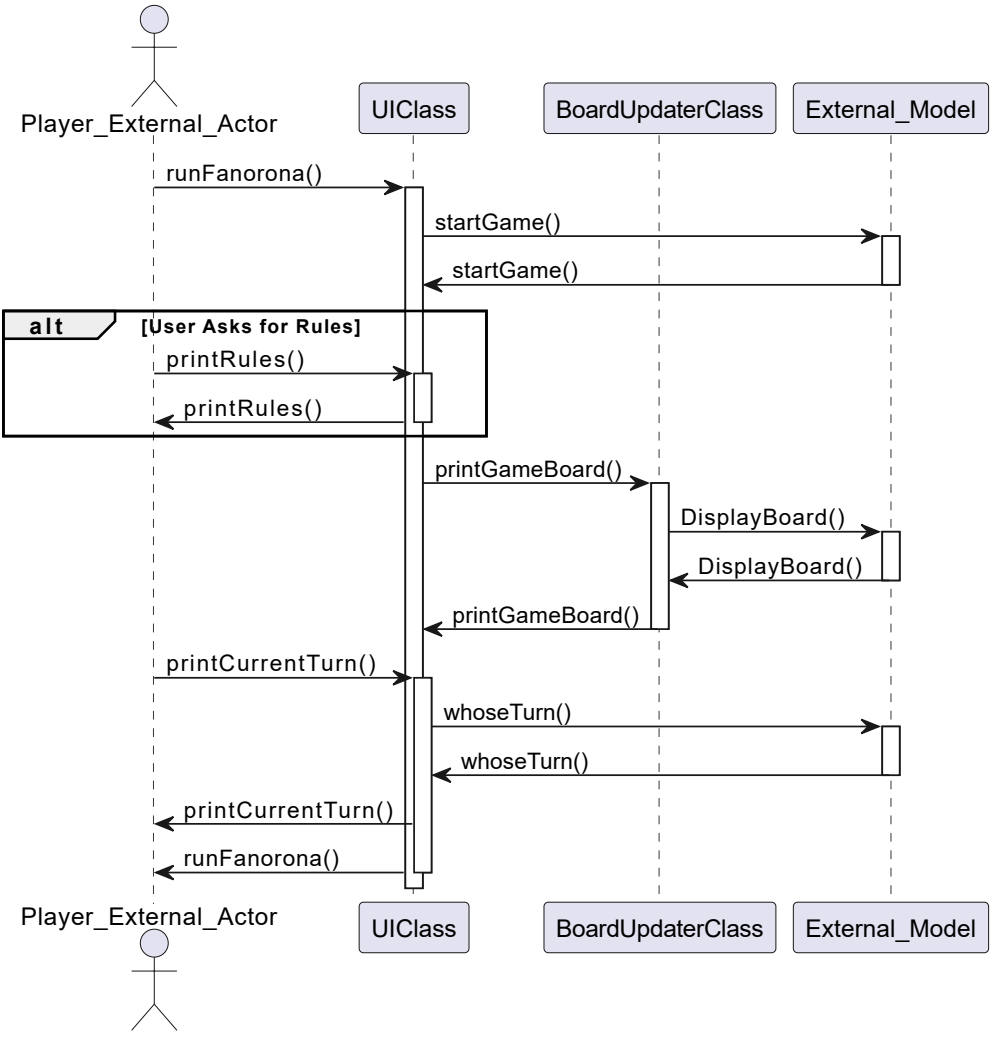
Alternate Flow	3a: Opponent declines the pause request. 3a1: The View informs the Player of the decision. 3a2: Player chooses to continue playing or forfeit the game.
Post-Conditions	If paused, the game resumes after the pause. If not paused, the game continues or the Player may forfeit.

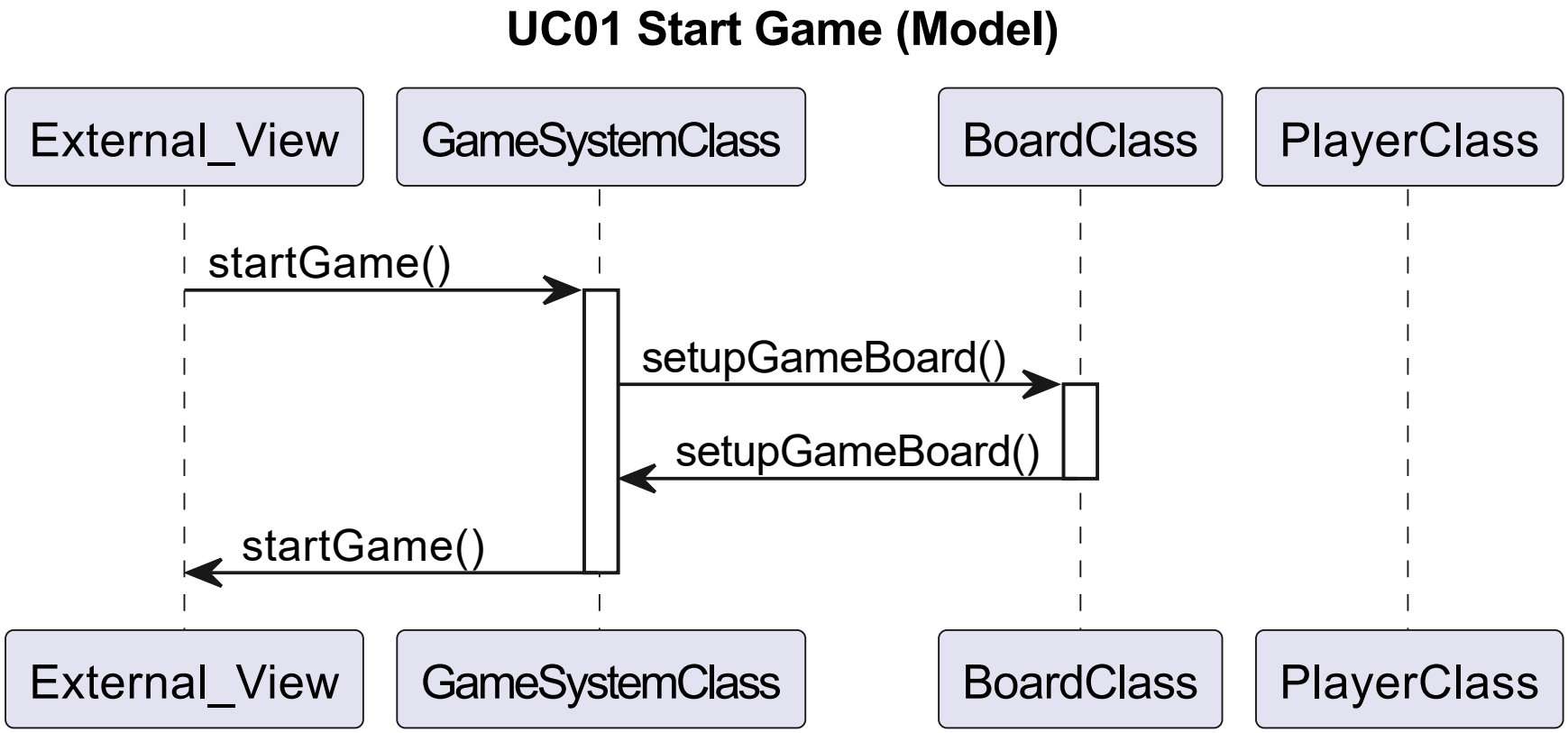
UC09: Undo Last Move (View)	
Primary Actor	Player requesting the undo
Goal	Allow a player to request to undo their last move, subject to the opponent's approval.
Stakeholders	Player, Opponent, Referee, Boardmaster, Board Updater
Init Event	Player realizes a mistake after making a move.
Pre-Conditions	Player requesting the undo has just made the last move. The game is in progress.
Main Success Scenario	1. **Player** requests the **View** to undo their last move. 2. **View** communicates the request to the Opponent. 3. **Opponent** considers the request. 4. **Opponent** agrees to allow the undo. 5. **View** requests the **Model** to undo previous move. 6. The **BoardUpdater** gets a new board.

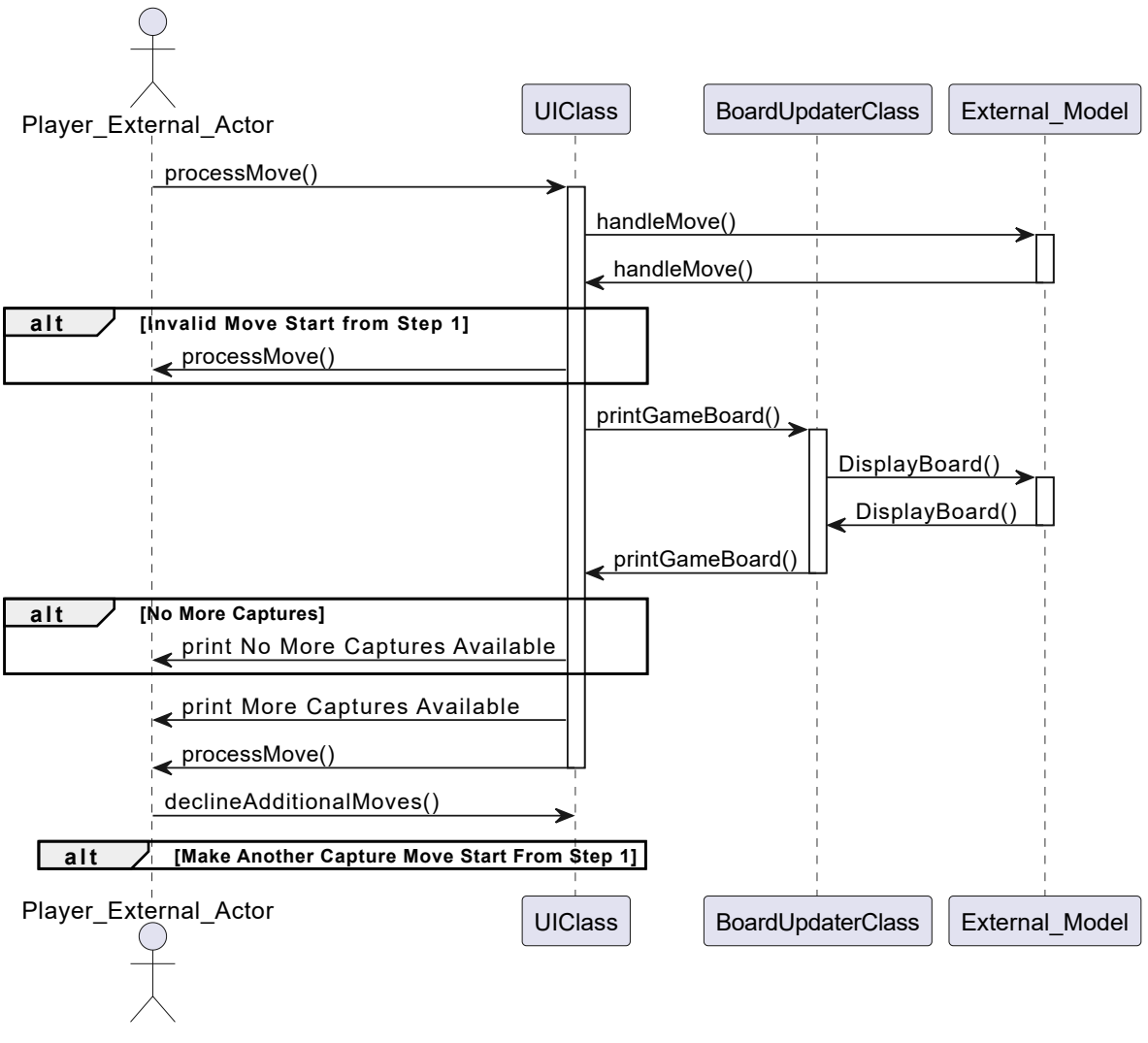
	7. **View** displays the reverted board using the **BoardUpdater**. 8. **Player** makes a new move. 9. The game continues as normal.
Alternate Flow	3a: Opponent declines the undo request. 3a1: View informs the Player of the Opponent's decision. 3a2: The game proceeds from the current state. 5a: Model cannot revert the move. 5a1: Model informs the View of the issue. 5a2: View notifies both Players. 5a3: The game continues from the current state.
Post-Conditions	If accepted, the game state is reverted, and the Player makes a new move. If declined, the game continues without changes.

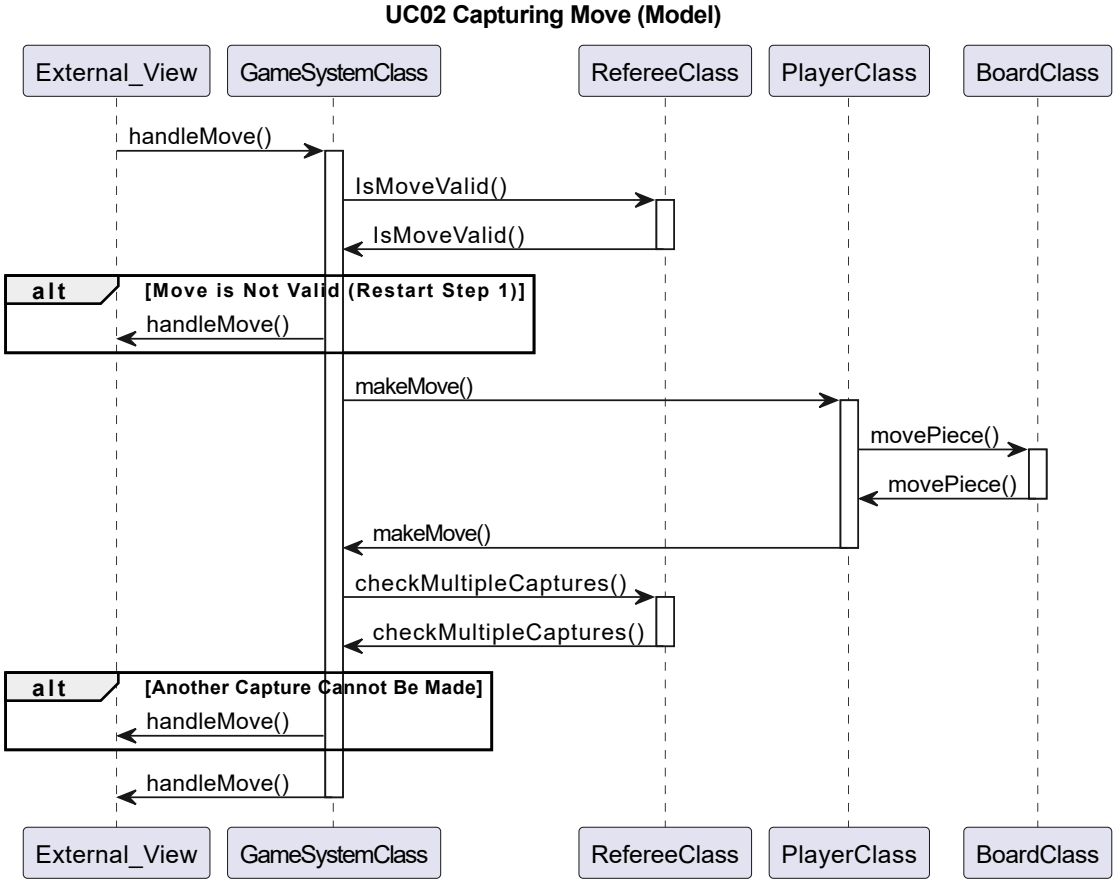
UC09: Undo Last Move (Model)	
Primary Actor	View requesting the undo
Goal	Allow a player to request to undo their last move, subject to the opponent’s approval.
Stakeholders	View, Referee, Boardmaster, Board Updater
Init Event	Player realizes a mistake after making a move.

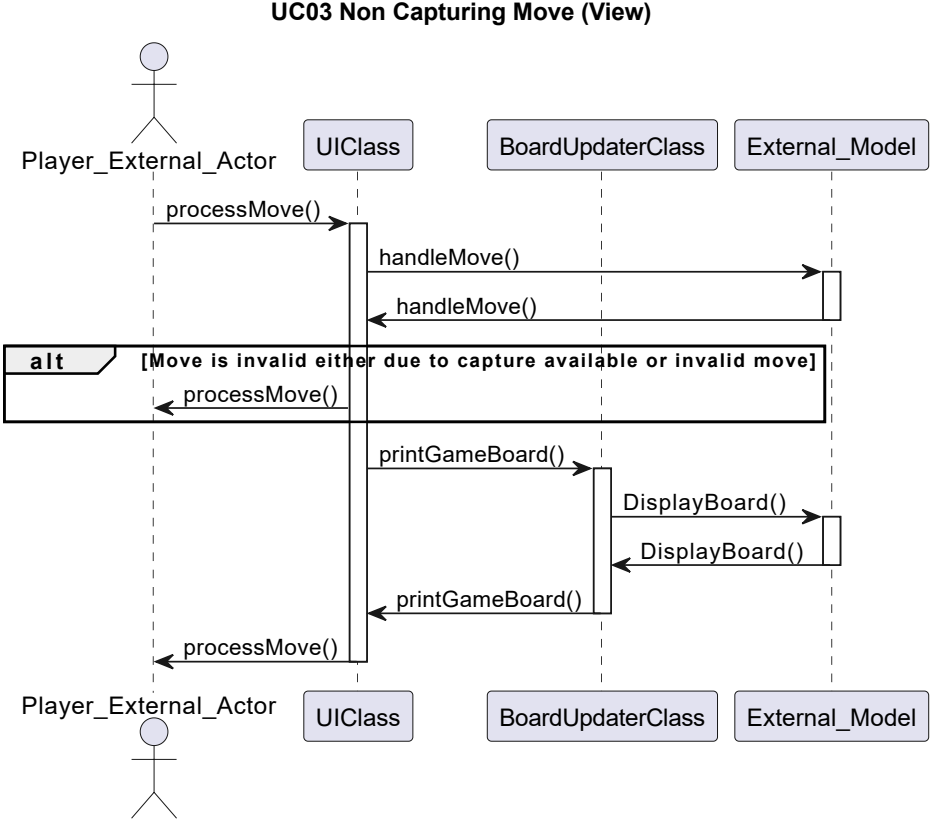
Pre-Conditions	Player requesting the undo has just made the last move. The game is in progress.
Main Success Scenario	1. **Model** gets request from View to revert board by one move. 2. **Referee** validates if the move can be undone. 3. **Board** reverts the game state to before the last move. 4. **Model** notifies the **View** that the board has been updated. 5. The game continues as normal.
Alternate Flow	2a: Undo move is not valid 2a1: **Model** informs the **View** that it cannot be done. 2a2: **View** notifies both Players. 2a3: The game continues from the current state. 3a: **Boardmaster** cannot revert the move. 5a1: **Boardmaster** informs the **View** that it cannot be done. 5a2: **View** notifies both Players. 5a3: The game continues from the current state.
Post-Conditions	If accepted, the game state is reverted, and the Player makes a new move. If declined, the game continues without changes.

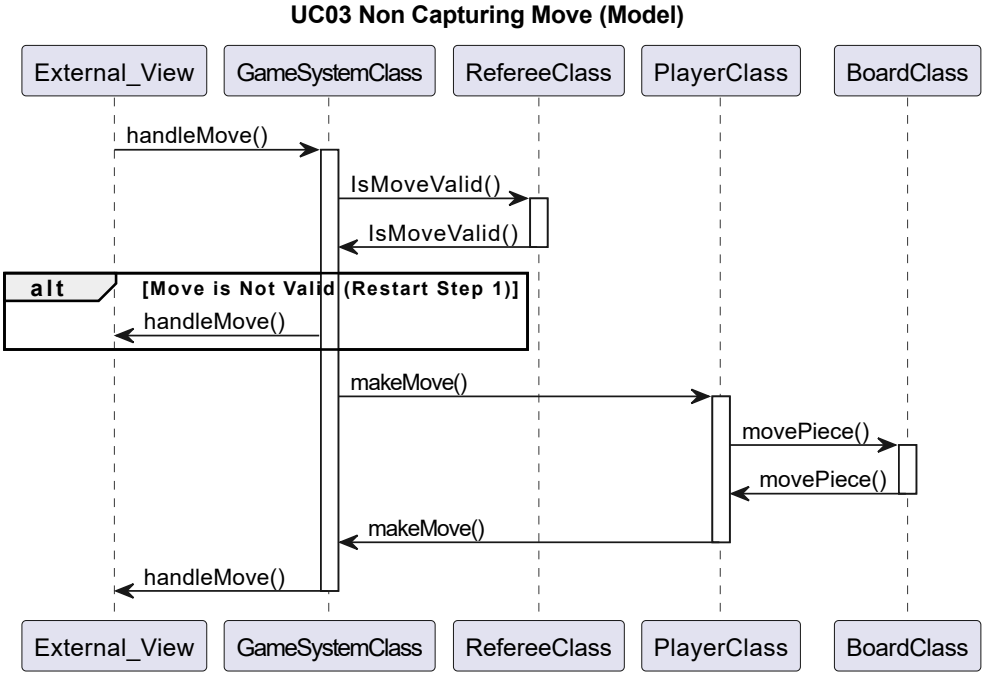

UC01 Start a New Game (View)
Player_External_Actor
UIClass
BoardUpdaterClass
External_Model
runFanorona()
startGame()
startGame()
alt
[User Asks for Rules]
printRules()
printRules()
printGameBoard()
DisplayBoard()
DisplayBoard()
printGameBoard()
printCurrentTurn()
whoseTurn()
whoseTurn()
printCurrentTurn()
runFanorona()
Player_External_Actor
UIClass
BoardUpdaterClass
External_Model

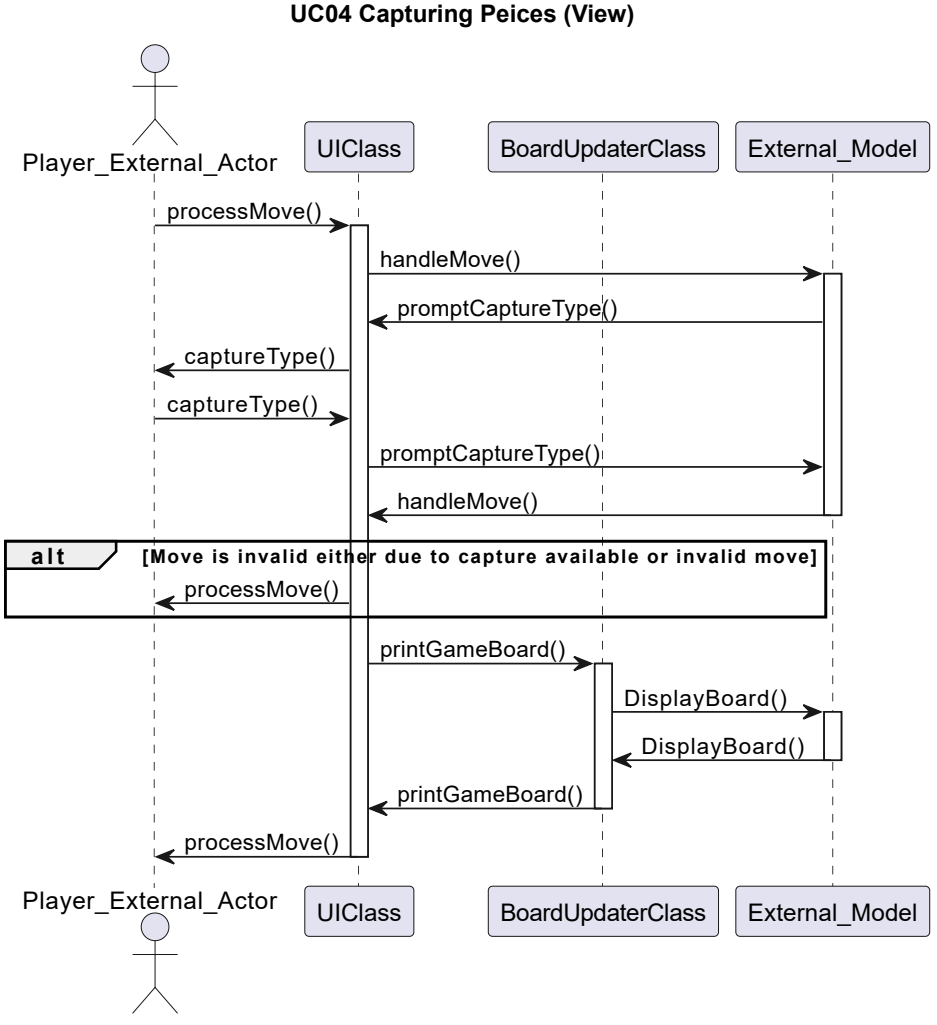

UC01 Start Game (Model)
External_View
GameSystemClass
BoardClass
PlayerClass
startGame()
setupGameBoard()
setupGameBoard()
startGame()
External_View
GameSystemClass
BoardClass
PlayerClass

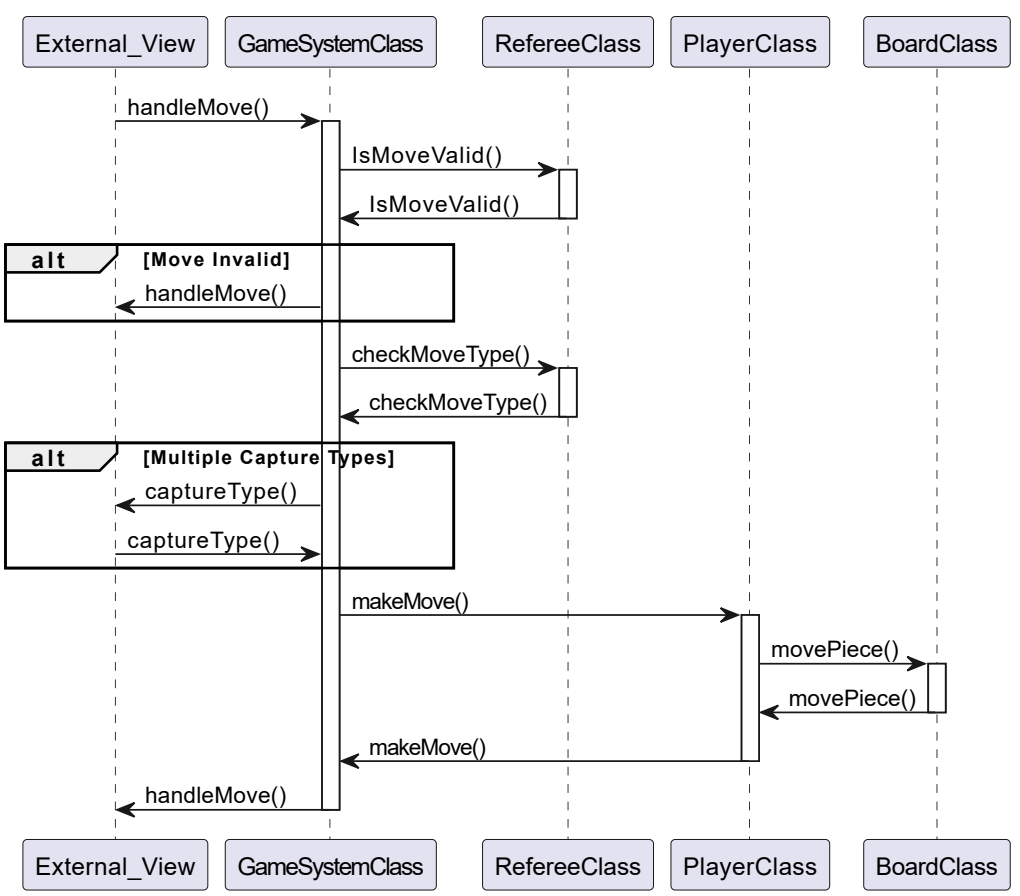

UC02 Capturing Move (View)
Player_External_Actor
UIClass
BoardUpdaterClass
External_Model
processMove()
handleMove()
handleMove()
alt
[Invalid Move Start from Step 1]
processMove()
printGameBoard()
DisplayBoard()
DisplayBoard()
printGameBoard()
alt
[No More Captures]
print No More Captures Available
print More Captures Available
processMove()
declineAdditionalMoves()
alt
[Make Another Capture Move Start From Step 1]
Player_External_Actor
UIClass
BoardUpdaterClass
External_Model

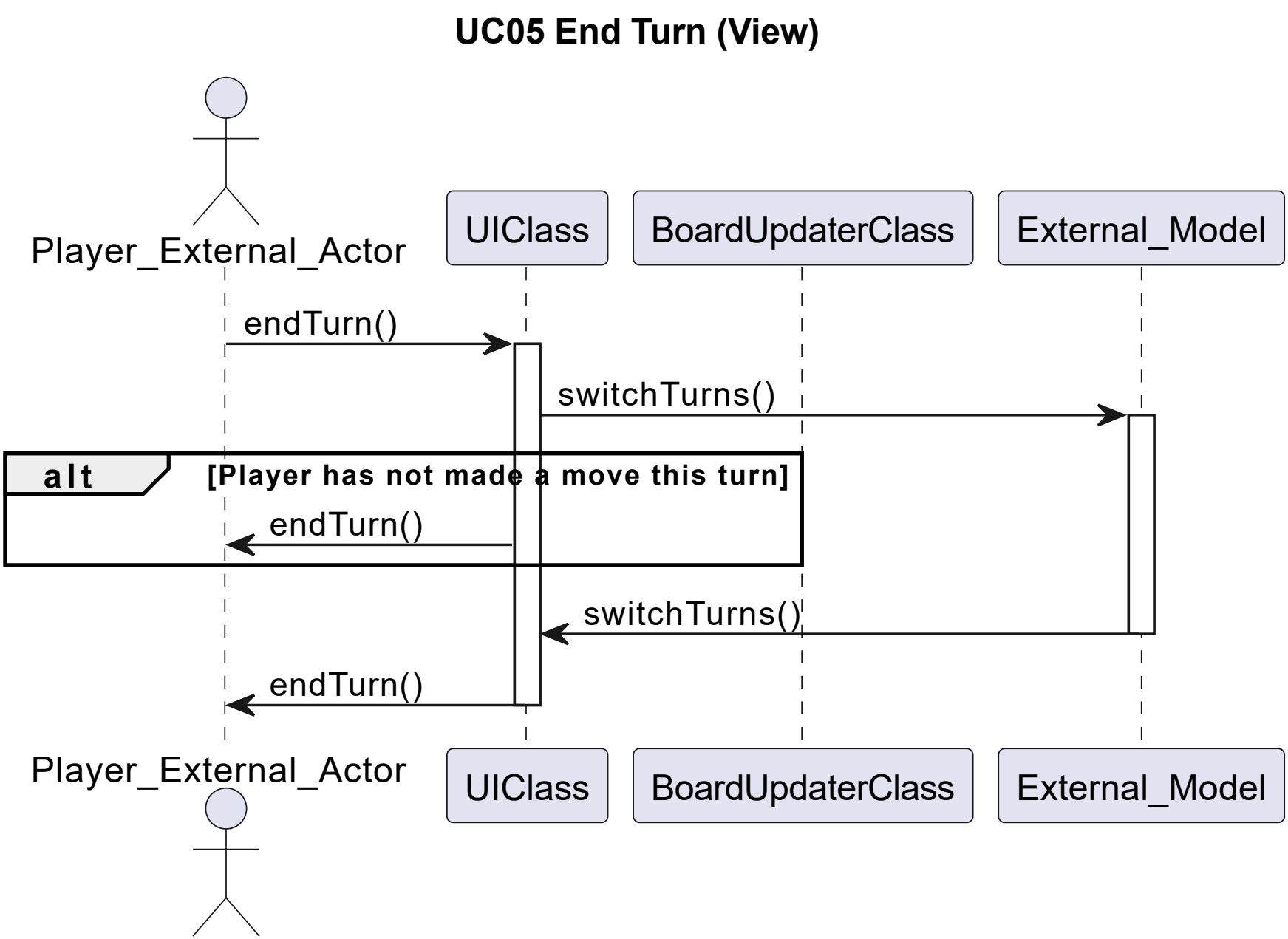

UC02 Capturing Move (Model)
External_View
GameSystemClass
RefereeClass
PlayerClass
BoardClass
handleMove()
IsMoveValid()
IsMoveValid()
alt
[Move is Not Valid (Restart Step 1)]
handleMove()
makeMove()
movePiece()
movePiece()
makeMove()
checkMultipleCaptures()
checkMultipleCaptures()
alt
[Another Capture Cannot Be Made]
handleMove()
handleMove()
External_View
GameSystemClass
RefereeClass
PlayerClass
BoardClass

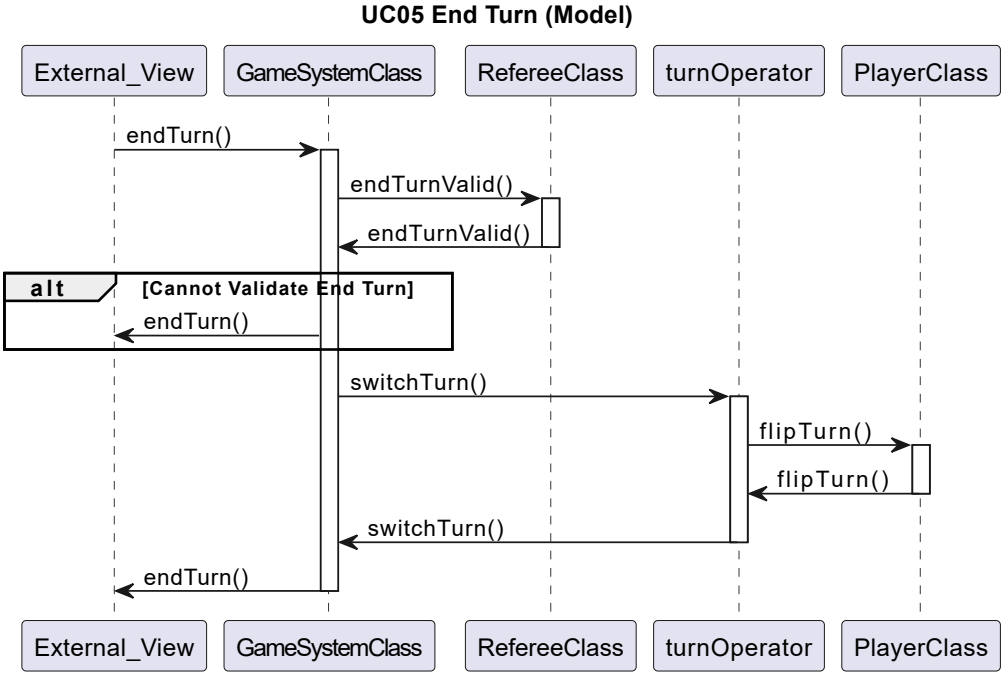

UC03 Non Capturing Move (View)
Player_External_Actor
UIClass
BoardUpdaterClass
External_Model
processMove()
handleMove()
handleMove()
alt
[Move is invalid either due to capture available or invalid move]
processMove()
printGameBoard()
DisplayBoard()
DisplayBoard()
printGameBoard()
processMove()
Player_External_Actor
UIClass
BoardUpdaterClass
External_Model

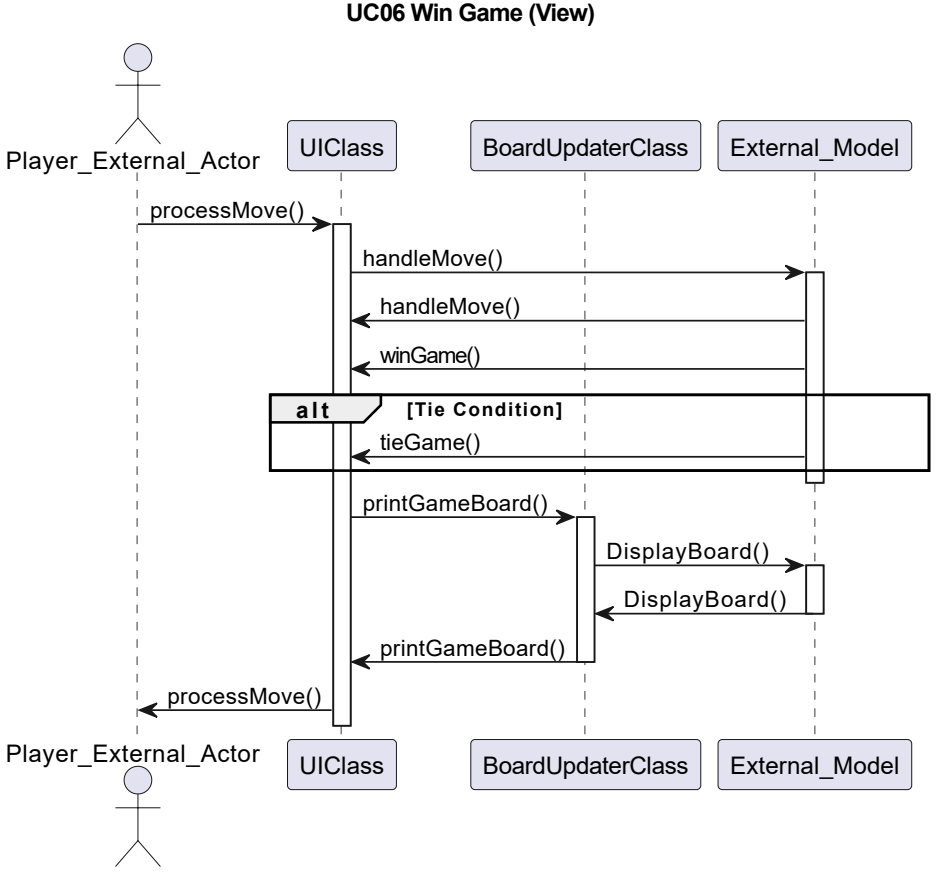

UC03 Non Capturing Move (Model)
External_View
GameSystemClass
RefereeClass
PlayerClass
BoardClass
handleMove()
IsMoveValid()
IsMoveValid()
alt
[Move is Not Valid (Restart Step 1)]
handleMove()
makeMove()
movePiece()
movePiece()
makeMove()
handleMove()
External_View
GameSystemClass
RefereeClass
PlayerClass
BoardClass

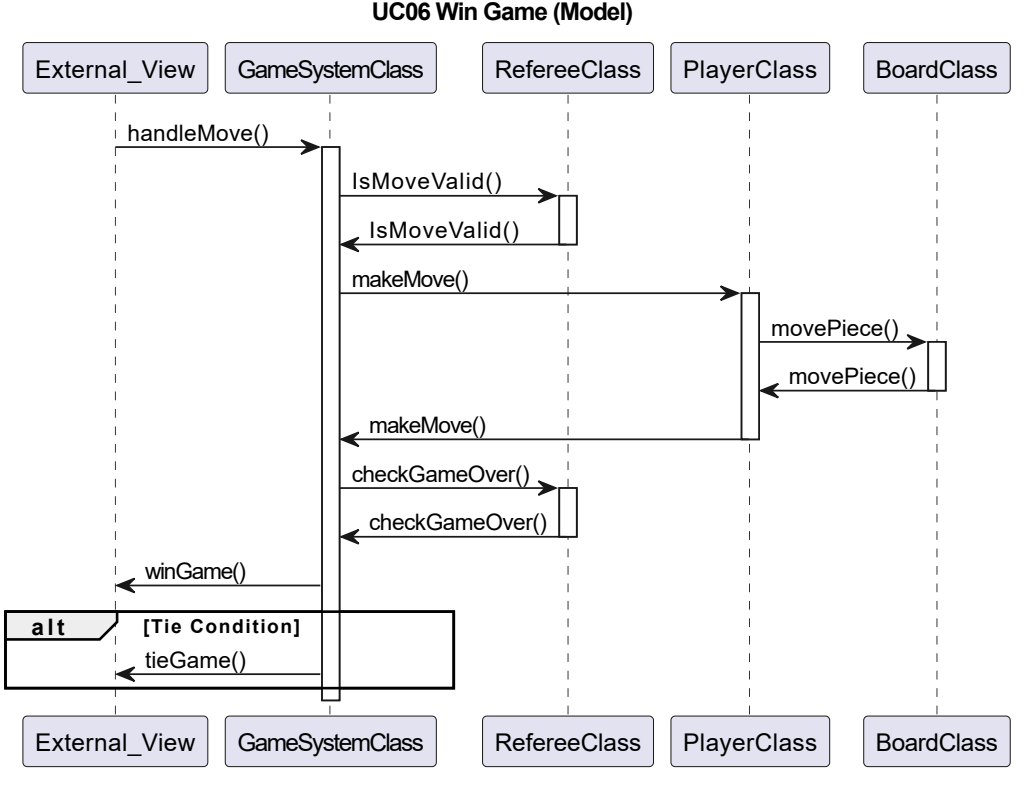

UC04 Capturing Peices (View)
Player_External_Actor
UIClass
BoardUpdaterClass
External_Model
processMove()
handleMove()
promptCaptureType()
captureType()
captureType()
promptCaptureType()
handleMove()
alt
[Move is invalid either due to capture available or invalid move]
processMove()
printGameBoard()
DisplayBoard()
DisplayBoard()
printGameBoard()
processMove()
Player_External_Actor
UIClass
BoardUpdaterClass
External_Model


UC04 Capturing Peices (Model)
External_View
GameSystemClass
RefereeClass
PlayerClass
BoardClass
handleMove()
IsMoveValid()
IsMoveValid()
alt
[Move Invalid]
handleMove()
checkMoveType()
checkMoveType()
alt
[Multiple Capture Types]
captureType()
captureType()
makeMove()
movePiece()
movePiece()
makeMove()
handleMove()
External_View
GameSystemClass
RefereeClass
PlayerClass
BoardClass

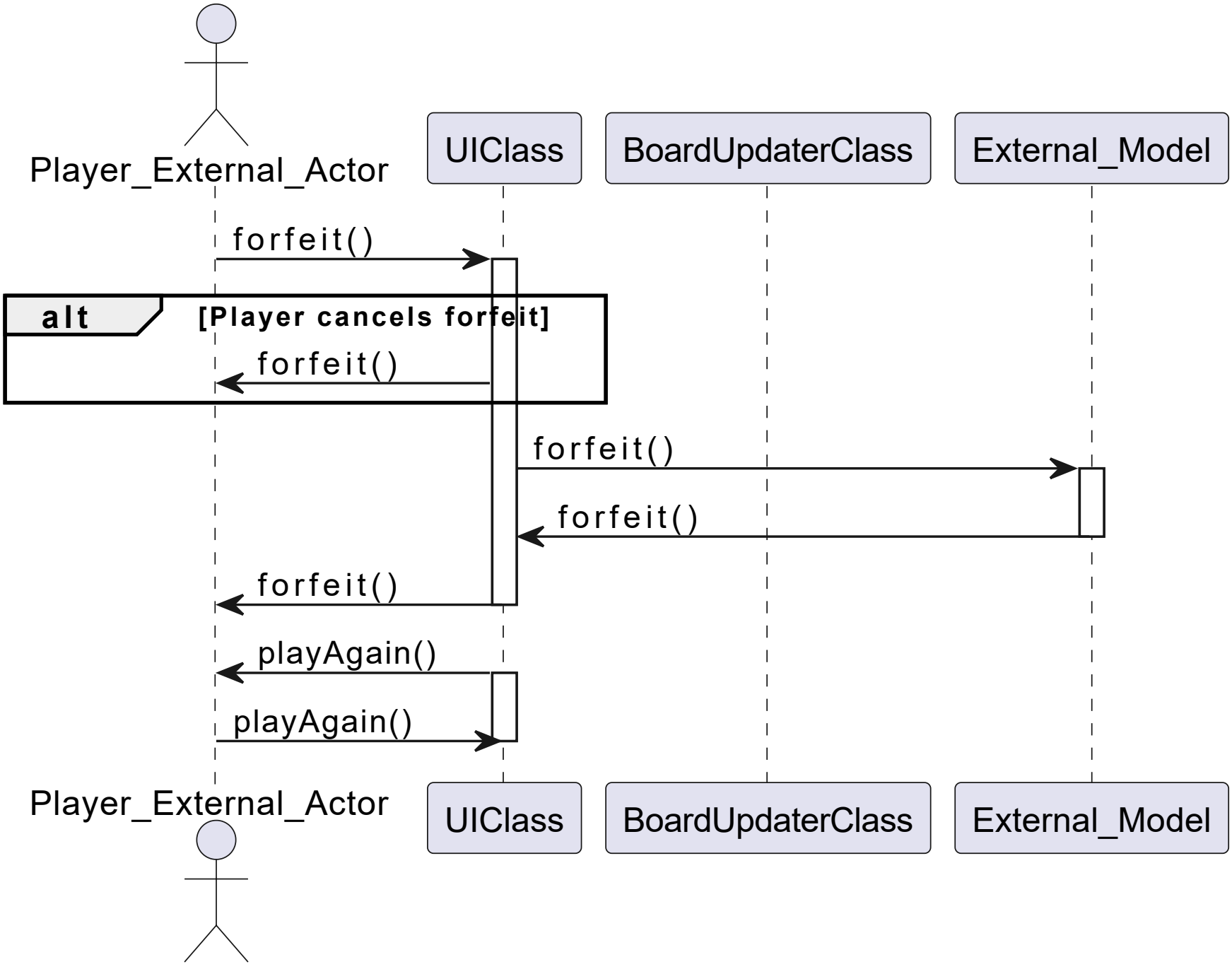

UC05 End Turn (View)
Player_External_Actor
UIClass
BoardUpdaterClass
External_Model
endTurn()
switchTurns()
alt
[Player has not made a move this turn]
endTurn()
switchTurns()
endTurn()
Player_External_Actor
UIClass
BoardUpdaterClass
External_Model


UC05 End Turn (Model)
External_View
GameSystemClass
RefereeClass
turnOperator
PlayerClass
endTurn()
endTurnValid()
endTurnValid()
alt
[Cannot Validate End Turn]
endTurn()
switchTurn()
flipTurn()
flipTurn()
switchTurn()
endTurn()
External_View
GameSystemClass
RefereeClass
turnOperator
PlayerClass

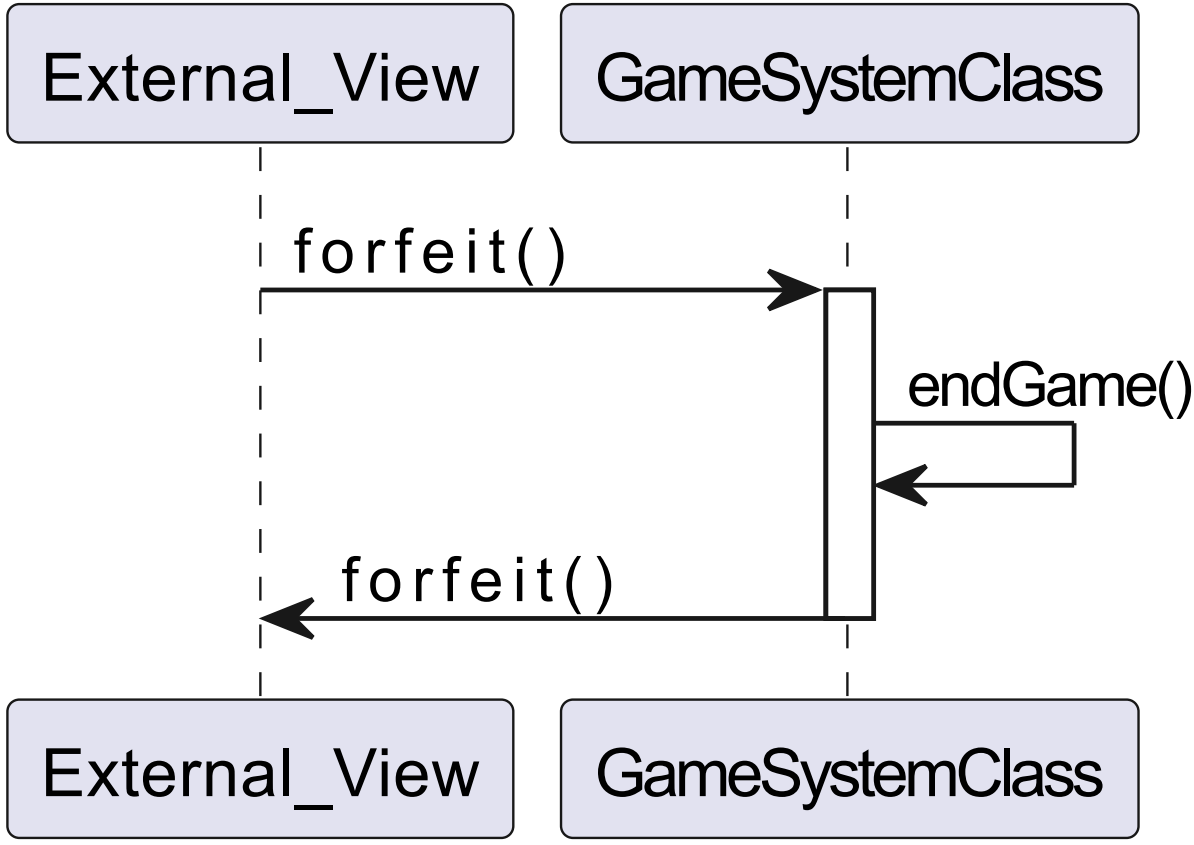

UC06 Win Game (View)
Player_External_Actor
UIClass
BoardUpdaterClass
External_Model
processMove()
handleMove()
handleMove()
winGame()
alt
[Tie Condition]
tieGame()
printGameBoard()
DisplayBoard()
DisplayBoard()
printGameBoard()
processMove()
Player_External_Actor
UIClass
BoardUpdaterClass
External_Model

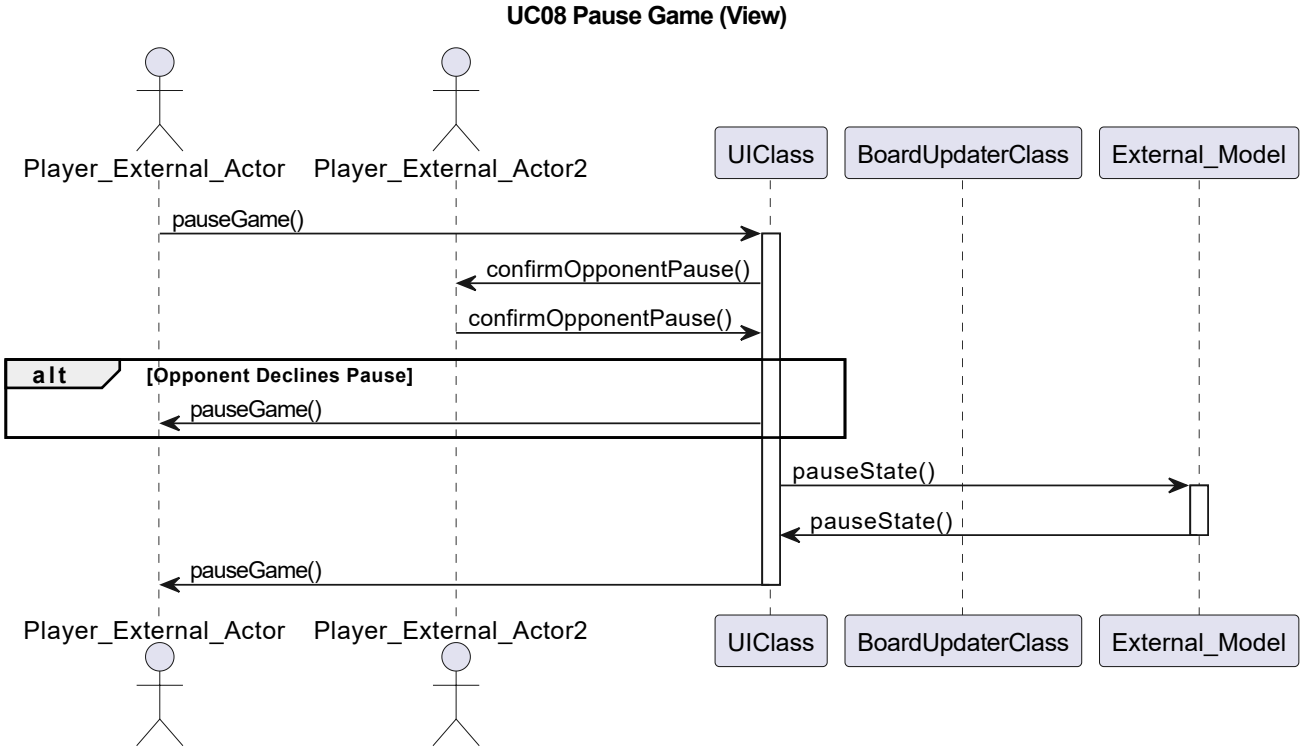

UC06 Win Game (Model)
External_View
GameSystemClass
RefereeClass
PlayerClass
BoardClass
handleMove()
IsMoveValid()
IsMoveValid()
makeMove()
movePiece()
movePiece()
makeMove()
checkGameOver()
checkGameOver()
winGame()
alt
[Tie Condition]
tieGame()
External_View
GameSystemClass
RefereeClass
PlayerClass
BoardClass

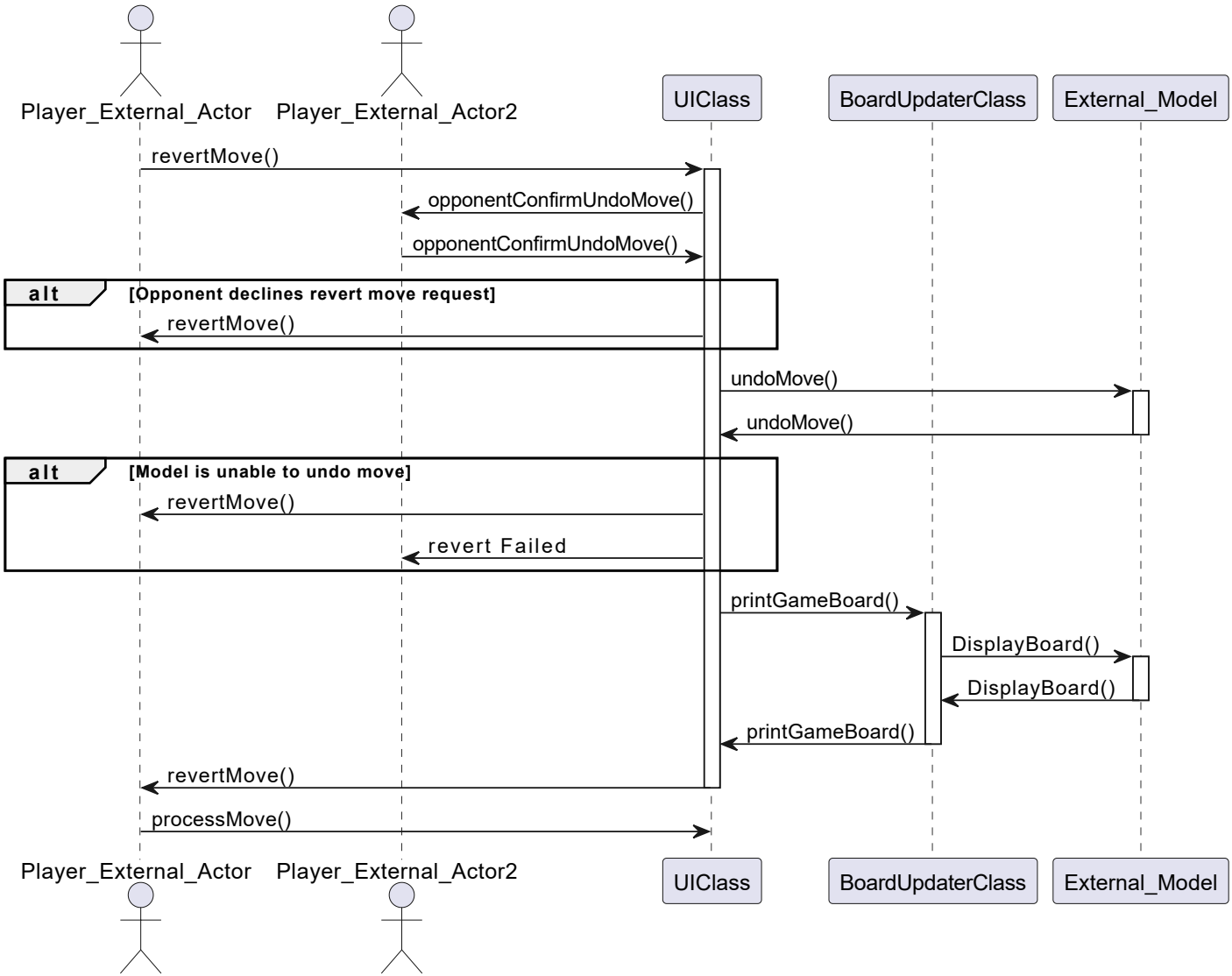
UC07 Forfeit Game (View)

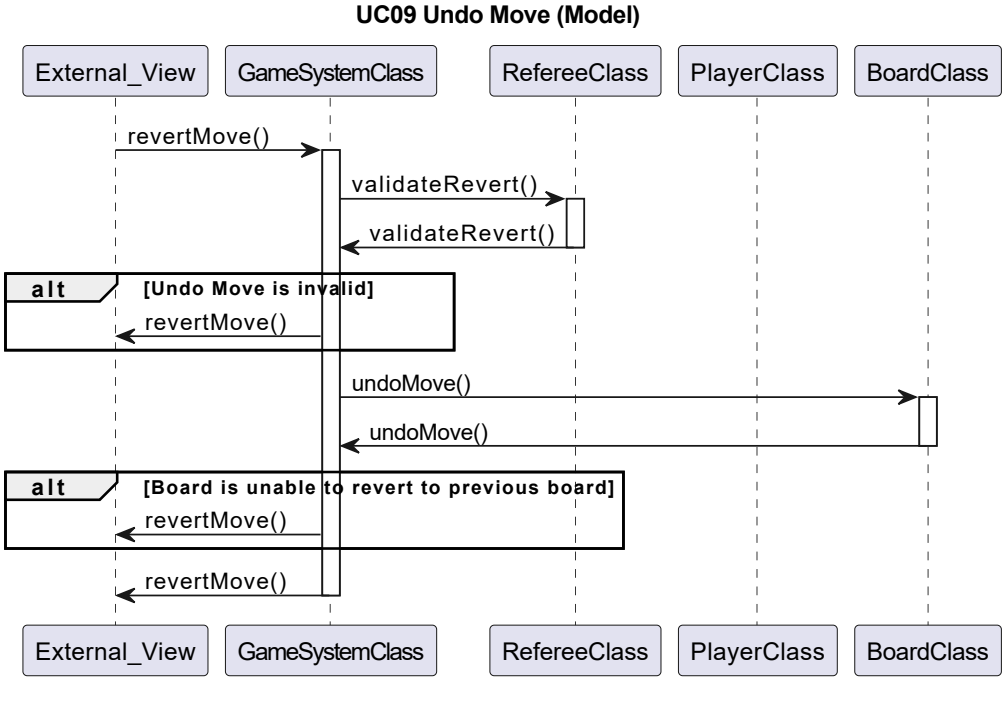


UC07 Forfeit Game (Model)




UC08 Pause Game (View)
Player_External_Actor
Player_External_Actor2
UIClass
BoardUpdaterClass
External_Model
pauseGame()
confirmOpponentPause()
confirmOpponentPause()
alt
[Opponent Declines Pause]
pauseGame()
pauseState()
pauseState()
pauseGame()
Player_External_Actor
Player_External_Actor2
UIClass
BoardUpdaterClass
External_Model


UC09 Undo Move (View)
Player_External_Actor
Player_External_Actor2
UIClass
BoardUpdaterClass
External_Model
revertMove()
opponentConfirmUndoMove()
opponentConfirmUndoMove()
alt
[Opponent declines revert move request]
revertMove()
undoMove()
undoMove()
alt
[Model is unable to undo move]
revertMove()
revert Failed
printGameBoard()
DisplayBoard()
DisplayBoard()
printGameBoard()
revertMove()
processMove()
Player_External_Actor
Player_External_Actor2
UIClass
BoardUpdaterClass
External_Model


UC09 Undo Move (Model)
External_View
GameSystemClass
RefereeClass
PlayerClass
BoardClass
revertMove()
validateRevert()
validateRevert()
alt
[Undo Move is invalid]
revertMove()
undoMove()
undoMove()
alt
[Board is unable to revert to previous board]
revertMove()
revertMove()
External_View
GameSystemClass
RefereeClass
PlayerClass
BoardClass

Class Diagram for Fanorona

Fanorona

```
Welcome to Fanorona

What would you like to do today?

1. Play a game of Fanorona
2. How to Play
3. Exit Game

> 1

Choose a colour for Player 1 (Enter W for white and B for Black)

> W

Player 1 has chosen White

Player 2 is Black



Here is the initial game board

  1 2 3 4 5 6 7 8 9
1 B B B B B B B B B
2 B B B B B B B B B
3 B W B W . B W B W
4 W W W W W W W W W
5 W W W W W W W W W


It is the Player 1's turn

Select a piece to move (Row, Column):

> 3, 4

Where do you want to move the piece:

> 3, 5

> There are 2 available pieces to capture, which one would you like to capture

To capture the piece at 3,4 enter 1
To capture the piece at 3,6 enter 2

> 2

> Move performed! Piece captured

Here is the updated board:

  1 2 3 4 5 6 7 8 9
1 B B B B B B B B B
2 B B B B B B B B B
3 B W B . W . W B W
4 W W W W W W W W W
5 W W W W W W W W W
```

Fanorona

```
Welcome to Fanorona

What would you like to do today?

1. Play a game of Fanorona
2. How to Play
3. Exit Game

> 2

Here are the rules of Fanorona


1. Players take turns to move one of their pieces. Each turn a player is able to perform a move or capture.
   A piece can be moved along a line adjacent
2. There are two types of capture moves
   a. Capture by Approach:
      i. A piece is captured by moving to an adjacent empty point so that the enemy piece is on the next
         point within the movement of direction
   b. Capture by Withdrawal:
      i. The piece moves away to from an adjacent empty point in the opposite direction, any opponent pieces
         that are adjacent behind that piece are captured
3. Multiple captures are allowed meaning the player must make multiple captures after the first one if
   there is one available
```

Fanorona

```
Welcome to Fanorona

What would you like to do today?

1. Play a game of Fanorona
2. How to Play
3. Exit Game

> 3

Thanks for playing Fanorona! Have a great day!
```

Class Name: UI

Single Responsibility: Responsible for displaying information to the user while playing Fanorona, including the game rules and moves

Instance Variables:

- Private GameSystem model;
- Private BoardUpdater boardUpdater;

Constructor(s):

- Public UI();
    - Initializes variables and sets up any required configurations for the class

Public Methods:

- Public runFanorona()
    - This function starts the game of Fanorona, which will start by printing a menu for the user to select the three options
        - View Rules of the game
        - Start Game
        - Exit Game
    - Rules of the game will call printRules() and start game will run startGame in the gameSystem after asking the users for their respective colours. When a move is made this will call processMove().
- Public printRules()
    - This function prints out the game rules for Fanorona
        - This is called when the user selects rules
- Public processMove(fromRow: int, fromCol: int, toRow: int, toCol: int)
    - This function will take the move from the UI and prompt the gamesystems handleMove() function.
- Public revertMove()
    - This function will call revertMove() within the gameSystem
- Public forfeit()
    - This function will call forfeit() within the game system class
- Public printCurrentTurn
    - This function will call whoseTurn() within turnOperator and print which player's turn it currently is.
- Public pauseGame()
    - This function will ask the opponent if they want to pause the game. The opponent can enact a timer of 90 seconds at any time, and if the player does not make it back in time, the opponent wins by default. (This is done by calling its own forfeit

function.) In the event the opponent does not agree to a pause, the player is given two options: forfeit or continue the game.

- Public :moveType pickCaptureType()
    - This function will ask the user to pick a capture type for the move they just made which is sent back to the gamesystem.
- Public endTurn()
    - This function will run once a user selects end turn or there is no more moves to make. It will end the turn of the user by moving on with the ui input to the next move.

Class Name: Game System

Single Responsibility: Responsible for controlling the flow of Fanorona by managing the setup, player moves and game states

Instance Variables:

- Private Board board;
- Private Player player1;
- Private Player player2;
- Private Referee ref;
- Private UI ui;

Constructor(s):

- Public GameSystem();
    - Initializes variables and sets up any required configurations for the class

Public Methods:

- Public startGame(:colour colour1, :colour colour2)
    - This prompts setupGameBoard() in board and assigns colours to the players
    - **Parameters:**
        - Colour1: the colour of player 1
        - Colour2: the colour of player 2
- Public endGame(:winner Player)
    - If the game is forfeited then it would default say who one and if not it will end by telling the view that the game is over by calling endGame() in UI
- Public HandleMove(fromRow, FromCol, toRow, toCol)
    - This function will call isMoveValid() within the referee, if this returns true then it will call pickCaptureType() in the UI for user input. Once it has gotten the user input it can then run processMove() within the referee. After the move is processed we call checkMultipleCaptures() to see if the player can still make moves, if they can we loop, if they cant we endTurn().

    - **Parameters:**
        - FromRow: the row the move is coming from
        - FromCol: the column the move is coming from
        - toRow: the row the move is going to
        - toCol: the column the move is going to
- Public Forfeit()
    - This function will call whoseTurn() in turnoperator and then pass that when calling endGame() within itself.
- Public revertMove()
    - This function will call revertMove() within the board and also within the player
- Public endTurn():
    - This function will first call the endTurnValidate() function within the referee class and depending on the return value the players turn is then ended by switchTurn() within the turnoperator and then we will call checkGameOver() in referee and if it is True it will end the game

Class Name: Player

Single Responsibility: Manages the Player's state, by keeping track of their turns and pieces captured

Instance Variables:

- Private String colour;
    - Keeps track of the player's piece colour
- Private Boolean turn;
    - Keeps track of the player's turn
- Private Integer piecesCaptured;
    - Keeps track of how many pieces the player has captured from the opponent
- Private Integer movesMadeOnTurn
    - Keeps track of the amount of moves a player has made for a turn

Constructor(s):

- Public Player(colour, turn)
    - Initializes variables and sets up any required configurations for the class
    - **Parameters:**
        - Colour: pass the colour of the player in
        - Turn: pass the state of the player's turn

Public Methods:

- Public String checkColour()
    - Returns the colour of the player

- Public Boolean checkTurn()
    - Returns the status of the player's turn
- Public Boolean makeMove(fromRow, FromCol, toRow, toCol, board)
    - This function prompts the board that is passed to it to movePiece() and if it is a capture move it will call its function capturePiece()
    - **Parameters:**
        - FromRow: the row the move is coming from
        - FromCol: the column the move is coming from
        - toRow: the row the move is going to
        - toCol: the column the move is going to
        - Board: the board this is happening on

- Public CapturePiece()
    - Increments its captured pieces
- Public revertMove(Board board)
    - This function will use the board passed to it to ask for the last board using getLastBoard(). From there it will take this and check for any captures the player might have made and decrease their captures. This function also decrements movesMadeOnTurn.
    - **Parameters:**
        - Board: this is the current board
- Public endTurn()
    - This function will reset the movesMadeOnTurn variable to 0
- Public flipTurn()
    - This function will flip the turn boolean
- checkMovesMade()
    - This function will return the amount of moves made by the player on this turn

Class Name: Board

Single Responsibility: Manages the state and the movement of pieces on the game board

Instance Variables:

- :colour board[9][5];
- :colour lastBoard[9][5]

Constructor(s):

- Public Board()
    - Initializes variables and sets up any required configurations for the class

Public Methods:

- Public displayBoard()
    - Returns the current board for use in the view
- Public getLastBoard()
    - Returns the last board for undo
- Public setupGameBoard()
    - A function that sets up the initial 9x5 gameboard filled in with colour symbols which is used when starting a game
- howManyColour(:colour)
    - This function returns how many of whatever colour is passed to it remaining on the board
    - **Parameters:**
        - colour: the colour that the board will check for
- checkDiagonals(int row, int col)
    - This function will check if the given piece is on a diagonal space, then if it is will return the diagonals to it (an array of the spaces and colour of each spot).
    - **Parameters:**
        - Row: the row of the piece to check
        - Col: the column of the piece to check
- checkHorizontals(int row, int col)
    - This function will return the horizontal spaces of the passed-through move (an array of the spaces and colour of each spot).
    - **Parameters:**
        - Row: the row of the piece to check
        - Col: the column of the piece to check
- checkVerticals(int row, int col)
    - This function will return the vertical spaces of the passed-through move (an array of the spaces and colour of each spot).
    - **Parameters:**
        - Row: the row of the piece to check
        - Col: the column of the piece to check
- checkEmptySpaces(int row, int col)
    - This function will check if the coordinates passed-though are empty and return true or false
    - **Parameters:**
        - Row: the row of the piece to check
        - Col: the column of the piece to check
- Public movePiece(fromRow, FromCol, toRow, toCol)
    - This function moves a piece from one place to another
    - **Parameters:**
        - FromRow: the row the move is coming from
        - FromCol: the column the move is coming from
        - toRow: the row the move is going to
        - toCol: the column the move is going to
- Public revertMove()

    - This function will replace the current board with the previous board from getLastBoard()
- Public :moveType checkMoveType(fromRow, fromCol, toRow, toColumn):
    - This function returns a :moveType symbol. It will use its knowledge of the board and the incoming move to determine if the move will be a withdrawal or approach move
    - **Parameters:**
        - FromRow: the row the move is coming from
        - FromCol: the column the move is coming from
        - toRow: the row the move is going to
        - toCol: the column the move is going to

Class Name: Referee

Single Responsibility: In charge of validating game actions, including game rules and win conditions

Instance Variables:

- Private Board board

Constructor(s):

- Public Referee(board)
    - Initializes variables and sets up any required configurations for the class
    - **Parameters:**
        - Board: the board that this referee is managing

Public Methods:

- Public Boolean isMoveValid(fromRow, fromCol, toRow, toColumn)
    - This function checks to see if a move is valid (returns true or false). This should be done by asking the board if the space the player is trying to move is empty. If it is, the referee will ask the board for the horizontal, diagonal and vertical spaces. The referee will use this info to check if there are any force captures (b/c if there is a capture you need to make it).
    - **Parameters:**
        - FromRow: the row the move is coming from
        - FromCol: the column the move is coming from
        - toRow: the row the move is going to
        - toCol: the column the move is going to
- Public Boolean checkMultipleCaptures()
    - This function checks if the current Player can make more captures

- Public Boolean checkGameOver()
    - This function checks if the game is over (returns the winner if there is one). It does this by asking the board how many of each colour there are and then deciding who won based on that info.
- Public Boolean endTurnValidate()
    - This function will make sure the current player has made a move by calling checkMovesMade() in the player. Then returning true if its >= 1.
- Public Boolean revertMoveValidate()
    - This function will ensure the current player has made a move by calling checkMovesMade() in the player. Then returning true if its >= 1.
- Public :moveType checkMoveType(fromRow, fromCol, toRow, toColumn):
    - This function returns a :moveType symbol. It will pass the board this move and ask what kind of move it is by calling checkMoveType() in the board class
    - **Parameters:**
        - FromRow: the row the move is coming from
        - FromCol: the column the move is coming from
        - toRow: the row the move is going to
        - toCol: the column the move is going to

Class Name: BoardUpdater

Single Responsibility: Responsible for keeping track and displaying the board to the players during the game

Instance Variables:

- Private Board board();

Constructor(s):

- public BoardUpdater(board)
    - Initializes the BoardUpdater that will be used to display the board throughout the game

Public Methods:

- public printGameBoard(board)
    - A function that prints out the latest update for the 9x5 game board
    - **Parameters:**
        - Board - the game board

Class Name: Turn Operator

Single Responsibility: Responsible for keeping track of the player's current turn during the game

1. **Instance Variables:**
    - Private :whoseTurn turn

2. **Constructor(s):**
    - public TurnOperator(Player player1, Player player2)
        - Initializes the turn operator class that will be used to keep track of turns in the game
        - **Parameters:**
            - Player1 - the player
            - Player2 - the opponent
3. **Public Methods:**
    - Public switchTurns(Player player1, Player player2)
        - A function that switches the turn variable by flipping the whoseTurn variable and calling flipTurn() in Player. This will also call endTurn() in the player class.
    - Public whoseTurn()
        - This function will return which player has the current turn