Lost Cities

Object-Oriented Programming (HY-252)
Project 2024-2025

Presented By:

Dimitrios Dimitropoulos

Computer Science

Date Last Edited:

On: February 28, 2025

This project was completed as part of the HY-252 course requirements.

Abstract

In lost cities is a turn-based strategy game designed to combine the excitement of exploration with educational insights into the Minoan civilization. This project models a dynamic board game environment in which two players, represented by pawns, compete to discover lost palaces and archaeological treasures such as rare findings, murals, and Snake Goddess statues. The game incorporates elements of card-based movement, strategic decision-making, and resource management.

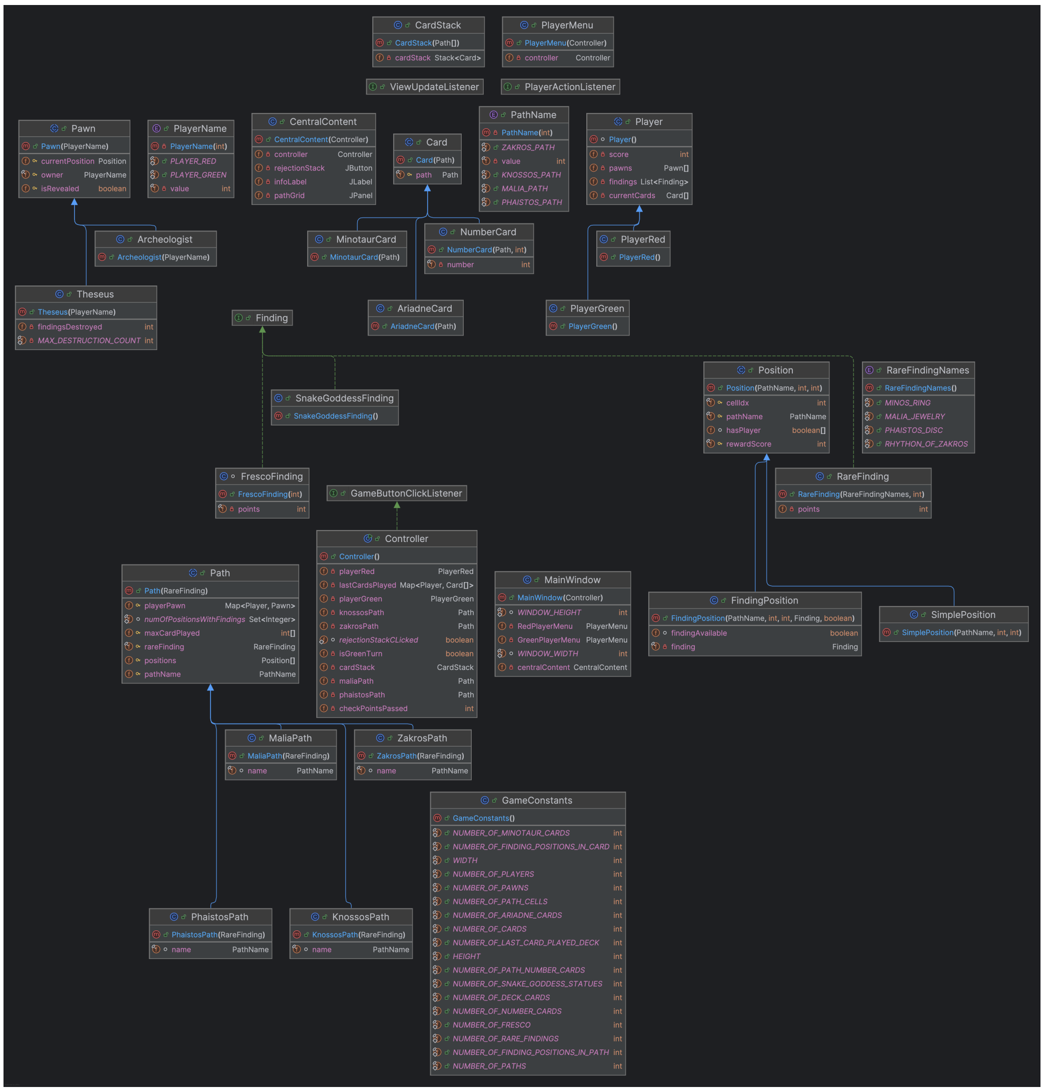
The game board features four paths, each representing a lost Minoan palace, and players use a deck of 100 cards to navigate the board. Special cards, including Ariadne and Minotaur cards, introduce unique mechanics, such as advanced movement and hindering opponent progress. Each pawn's journey involves overcoming penalties in early stages while striving for high-value rewards in later positions.

This project follows the Model-View-Controller (MVC) architecture, ensuring a clear separation between game logic, user interface, and interaction handling. The model includes entities like players, pawns, cards, and findings, while the view offers an interactive graphical interface with player menus, a central game board, and dynamic updates for real-time feedback. The controller orchestrates gameplay, enforcing rules and mediating between the model and view.

The game emphasizes modularity and scalability, making it easy to extend features, such as introducing new card types or paths. With a balance of strategy, historical intrigue, and modern programming practices, "In Search of the Minoans" serves as an engaging and educational experience for players while showcasing effective object-oriented design and software engineering principles

Application UML

This is the full diagram of our application. More detailed diagram and documentation is provided below.



Model Package Structure

- **card package:**
  - `AriadneCard`
  - `Card`
  - `MinotaurCard`
  - `NumberCard`
- **findings:**
  - `Findings`
  - `FrescoFinding`
  - `RareFinding`
  - `RareFindingNames`
  - `SnakeGoddnessFinding`
- **paths:**
  - `KnossoPath`
  - `MaliaPath`
  - `Path`
  - `PhaistosPath`
  - `ZakrosPath`
- **pawns:**
  - `Archeologist`
  - `Pawn`
  - `Theseus`
- **players:**
  - `Player`
  - `PlayerGreen`
  - `PlayerRed`
- **positions:**
  - `FindingPosition`

    - `Position`
    - `SimplePosition`

- **util:**

    - `PathName`
    - `PlayerName`

View Package Structure

- **components package:**

    - **menus package:**
        - `PlayerMenu`
    - **centralContent package:**
        - `centralContent`

- **window package:**

    - `MainWindow`

Assets Package Structure

- **csvFile:**

    - `csv_greek.csv`
    - `csv_greeklish.csv`

- **images package:**

    - `cards package`
    - `findings package`
    - `paths package`
    - `pionia package`

- **music package:**

    - `Player1.wav`
    - `Player2.wav`

Controller Package Structure

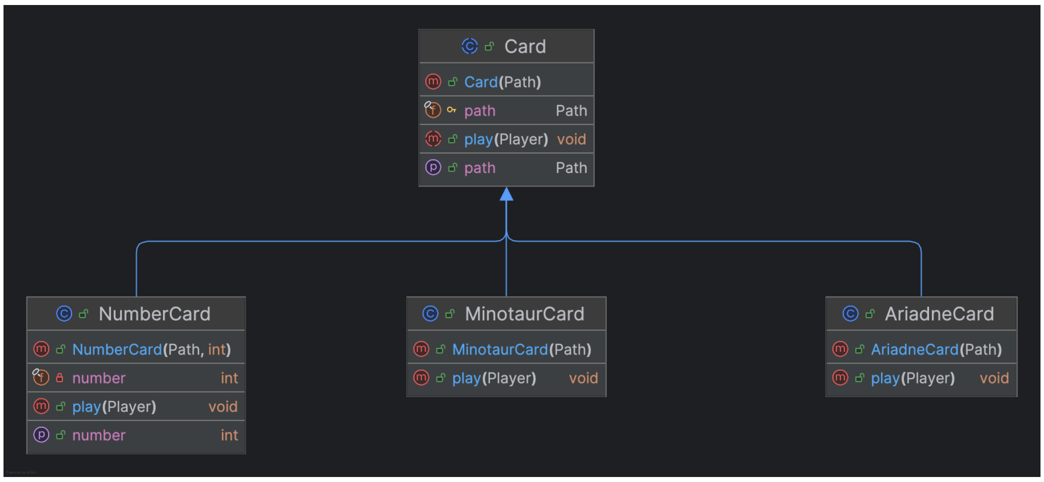
- **controller package:**
  - `GameButtonClickListener`
  - `Controller`

Util Package Structure

- **util package:**
  - `GameConstants`

Model Package structure

Card UML:



Card

The `Card` class is an abstract representation of all cards used in the game. These cards can be one of three types: Ariadne, Minotaur, or Number cards.

Key Details:

- Each card is associated with a specific path (`Path`).
- Cards allow players to move their pawns along the paths.
- The game consists of 100 cards in total:
  - 80 Number cards.
  - 12 Ariadne cards.
  - 8 Minotaur cards.

Public Methods:

- `Card(Path path)`: Constructor that initializes a card with an associated path.
- `void play(Player player)`: Abstract method to implement the specific behavior of the card for the given player.
- `Path getPath()`: Returns the path associated with this card.

—

AriadneCard

The `AriadneCard` class extends `Card` and represents an Ariadne card in the game.

Key Details:

- Ariadne cards are special cards that allow players to perform specific moves under certain conditions.
- Players cannot use an Ariadne card during the first round.

Public Methods:

- `AriadneCard(Path path)`: Constructor that initializes the Ariadne card with an associated path.
- `void play(Player player)`: Implementation of the card's behavior. Throws an exception if used during the first round.

—

MinotaurCard

The `MinotaurCard` class extends `Card` and represents a Minotaur card in the game.

Key Details:

- Minotaur cards have unique effects, although their exact behavior is to be determined.

Public Methods:

- `MinotaurCard(Path path)`: Constructor that initializes the Minotaur card with an associated path.
- `void play(Player player)`: Implementation of the card's behavior (currently undefined in the code).

—

NumberCard

The `NumberCard` class extends `Card` and represents a number card in the game.

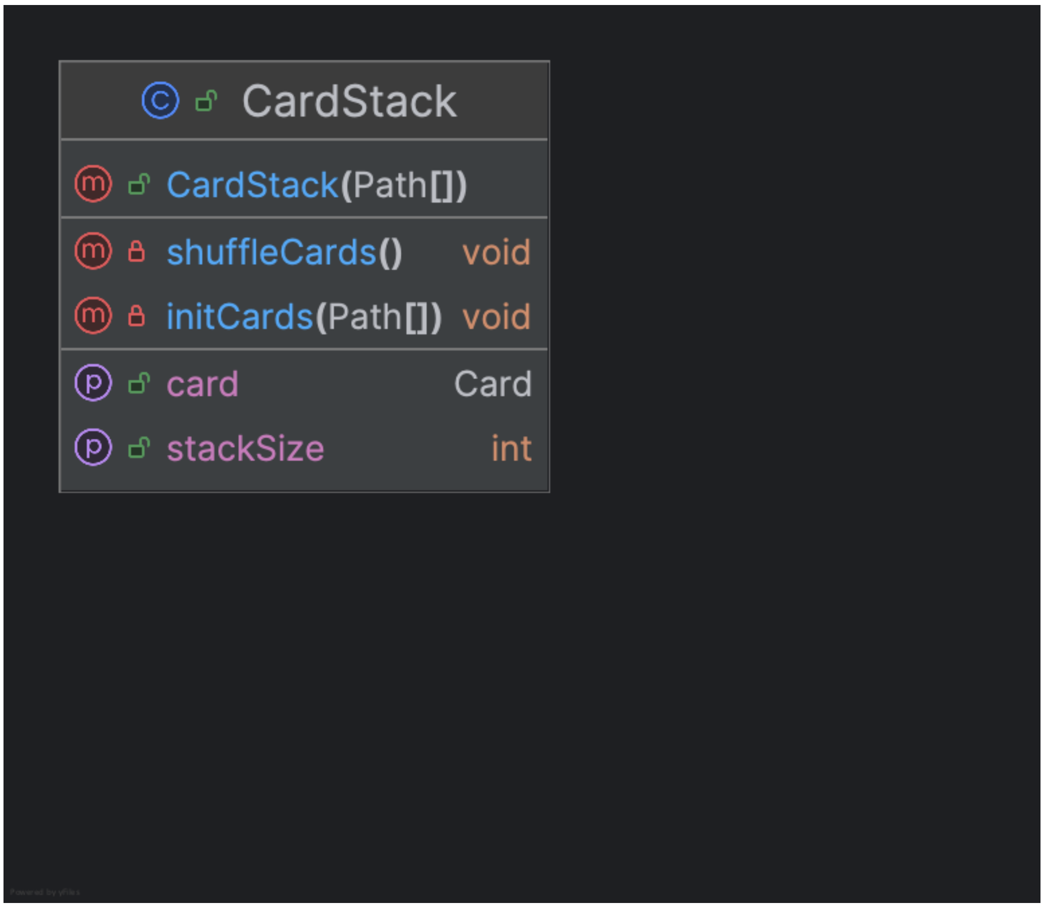
Key Details:

- Number cards allow players to move their pawns forward.
- The number on the card must be greater than any previously played number card for the given path.
- If a pawn reaches the last position on the path, it is removed from the map.

Public Methods:

- `NumberCard(Path path, int number)`: Constructor that initializes a number card with an associated path and a number.
  - Throws an exception if the number is not between 0 and 10.
- `void play(Player player)`: Moves the player's pawn based on the number on the card. Updates the maximum number played for the path.
- `int getNumber()`: Returns the number associated with the card.

cardStack UML:



CardStack

The `CardStack` class represents the draw pile of free cards in the game.
Key Details:

- Initially, the `CardStack` contains 100 cards:
  - 80 Number cards (20 for each ancient city or path).
  - 12 Ariadne cards (3 for each path).
  - 8 Minotaur cards (2 for each path).
- Cards are shuffled before the game starts.
- Cards are drawn from the top of the stack.

Public Methods:

- `CardStack(Path[] paths)`:
  - Constructor initializes the `CardStack` with 100 cards distributed among the paths.
  - Calls `initCards` to populate the stack and `shuffleCards` to shuffle it.
- `int getStackSize()`:
  - Returns the current size of the card stack.
- `Card getCard()`:
  - Returns the top card from the stack.
  - Throws an `IllegalArgumentException` if the stack is empty.

Private Methods:

- `void shuffleCards()`:
  - Shuffles the stack using `Collections.shuffle`.
- `void initCards(Path[] paths)`:
  - Populates the card stack with:
    - `NumberCard`: 20 cards for each path.
    - `AriadneCard`: 3 cards for each path.
    - `MinotaurCard`: 2 cards for each path.
  - Uses constants from `GameConstants` to determine card distribution.

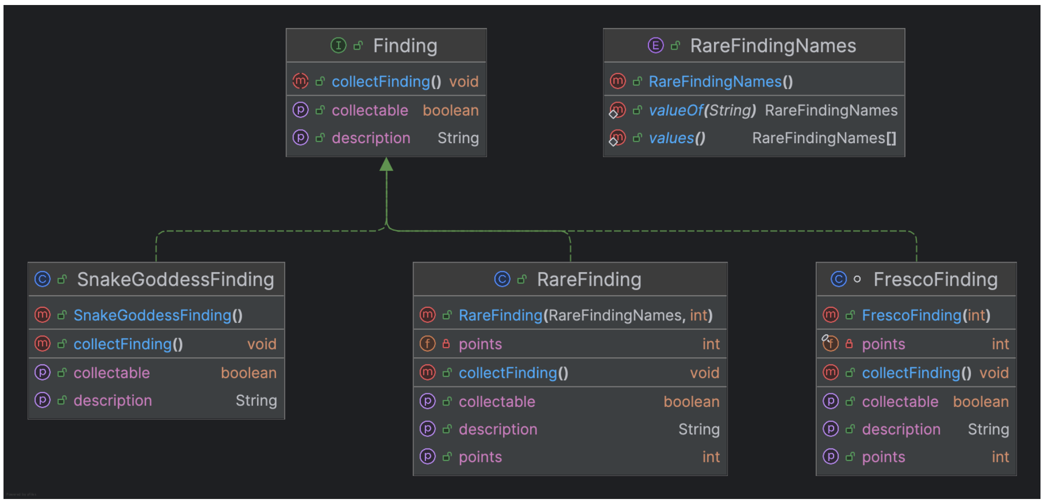
Preconditions:

- `getCard()` requires that the stack is not empty.

Finding

The `Finding` interface represents a generic finding in the game. A finding can be one of the following types:

Finding UML:



- Fresco.
- Rare finding.
- Snake goddess statue.

Key Details:

- Each finding has a name and rewards points (either directly or indirectly) upon discovery.

Public Methods:

- `void collectFinding()`: Adds the finding to the player's inventory.
- `String getDescription()`: Returns a description of the finding, including its name, type, or other details.
- `boolean isCollectable()`: Indicates whether the finding is collectable by a player.

—

FrescoFinding

The `FrescoFinding` class implements the `Finding` interface and represents a fresco finding in the game.

Key Details:

- Fresco findings directly reward the player with points upon discovery.

Public Methods:

- `FrescoFinding(int points)`:
  - Constructor that initializes the fresco finding with a specific point value.
- `int getPoints()`: Returns the points rewarded for discovering the fresco finding.

Overrides:

- `void collectFinding()`.
- `String getDescription()`.
- `boolean isCollectable()`.

—

RareFinding

The `RareFinding` class implements the `Finding` interface and represents a rare finding in the game.

Key Details:

- Rare findings directly reward the player with points upon discovery.
- Rare findings are validated using the `RareFindingNames` enum.

Public Methods:

- `RareFinding(RareFindingNames name, int points)`:
  - Constructor that initializes the rare finding with a name and point value.
  - Validates the name against the `RareFindingNames` enum.
  - Throws an `IllegalArgumentException` if the name is invalid.

- `int getPoints()`: Returns the points rewarded for discovering the rare finding.

Overrides:

- `void collectFinding()`.
- `String getDescription()`.
- `boolean isCollectable()`.

—

RareFindingNames

The `RareFindingNames` enum defines the valid names for rare findings. The available names are:

- `PHAISTOS_DISC`.
- `MINOS_RING`.
- `MALIA_JEWELRY`.
- `RHYTHON_OF_ZAKROS`.

—

SnakeGoddessFinding

The `SnakeGoddessFinding` class implements the `Finding` interface and represents a snake goddess statue in the game.

Key Details:

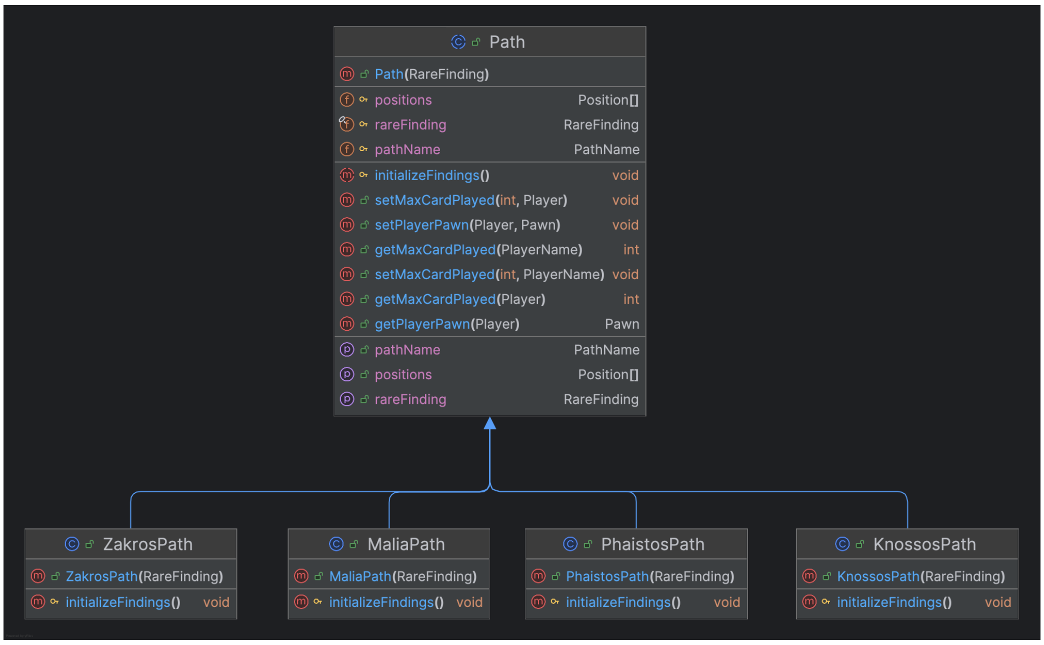
- The discovery of a snake goddess statue does not directly reward points.
- Instead, points are calculated at the end of the game based on the number of statues discovered, as per the following table:

Overrides:

- `void collectFinding()`.
- `String getDescription()`.
- `boolean isCollectable()`.

Number of Statues	Scores
0	0
1	-20
2	-15
3	10
4	15
5	30
6	50

Path UML:



Path
Path(RareFinding)
positions Position[]
rareFinding RareFinding
pathName PathName
initializeFindings() void
setMaxCardPlayed(int, Player) void
setPlayerPawn(Player, Pawn) void
getMaxCardPlayed(PlayerName) int
setMaxCardPlayed(int, PlayerName) void
getMaxCardPlayed(Player) int
getPlayerPawn(Player) Pawn
pathName PathName
positions Position[]
rareFinding RareFinding
ZakrosPath
ZakrosPath(RareFinding)
initializeFindings() void
MaliaPath
MaliaPath(RareFinding)
initializeFindings() void
PhaistosPath
PhaistosPath(RareFinding)
initializeFindings() void
KnossosPath
KnossosPath(RareFinding)
initializeFindings() void

Path

The `Path` class represents a generic path in the game. A path is an array of 9 positions, and completing it marks the discovery of a lost city.

Key Details:

- Paths are associated with unique rare findings (`RareFinding`).
- Certain positions on the path contain findings (positions 2, 4, 6, 8, and 9, 1-indexed).

Public Methods:

- `RareFinding getRareFinding()`: Returns the unique rare finding of the path.
- `Position[] getPositions()`: Returns the array of 9 positions on the path.
- `int getMaxCardPlayed(Player player)`: Returns the maximum card played for the given player.
- `void setPlayerPawn(Player player, Pawn pawn)`: Sets the player's pawn on the path.
- `Pawn getPlayerPawn(Player player)`: Retrieves the player's pawn on the path.
- `PathName getPathName()`: Returns the name of the path.
- `void setMaxCardPlayed(int maxCardPlayedValue, Player player)`: Updates the maximum card value played for a player.

Abstract Methods:

- `protected abstract void initializeFindings()`: Forces subclasses to define which findings will appear on the path.

—

KnossosPath

The `KnossosPath` class extends `Path` and represents the Knossos path in the game.

Key Details:

- The path's name is `PathName.KNOSSOS_PATH`.
- Implements the `initializeFindings` method to populate the path with findings or empty positions.

Overrides:

- `protected void initializeFindings()`: Populates the path with findings at specified positions (2, 4, 6, 8, and 9) and empty positions otherwise.

—

MaliaPath

The `MaliaPath` class extends `Path` and represents the Malia path in the game.

Key Details:

- The path's name is `PathName.MALIA_PATH`.

Overrides:

- `protected void initializeFindings()`: Defines the findings for the path (currently unimplemented).

—

PhaistosPath

The `PhaistosPath` class extends `Path` and represents the Phaistos path in the game.

Key Details:

- The path's name is `PathName.PHAISTOS_PATH`.

Overrides:

- `protected void initializeFindings()`: Defines the findings for the path (currently unimplemented).

—

ZakrosPath

The `ZakrosPath` class extends `Path` and represents the Zakros path in the game.

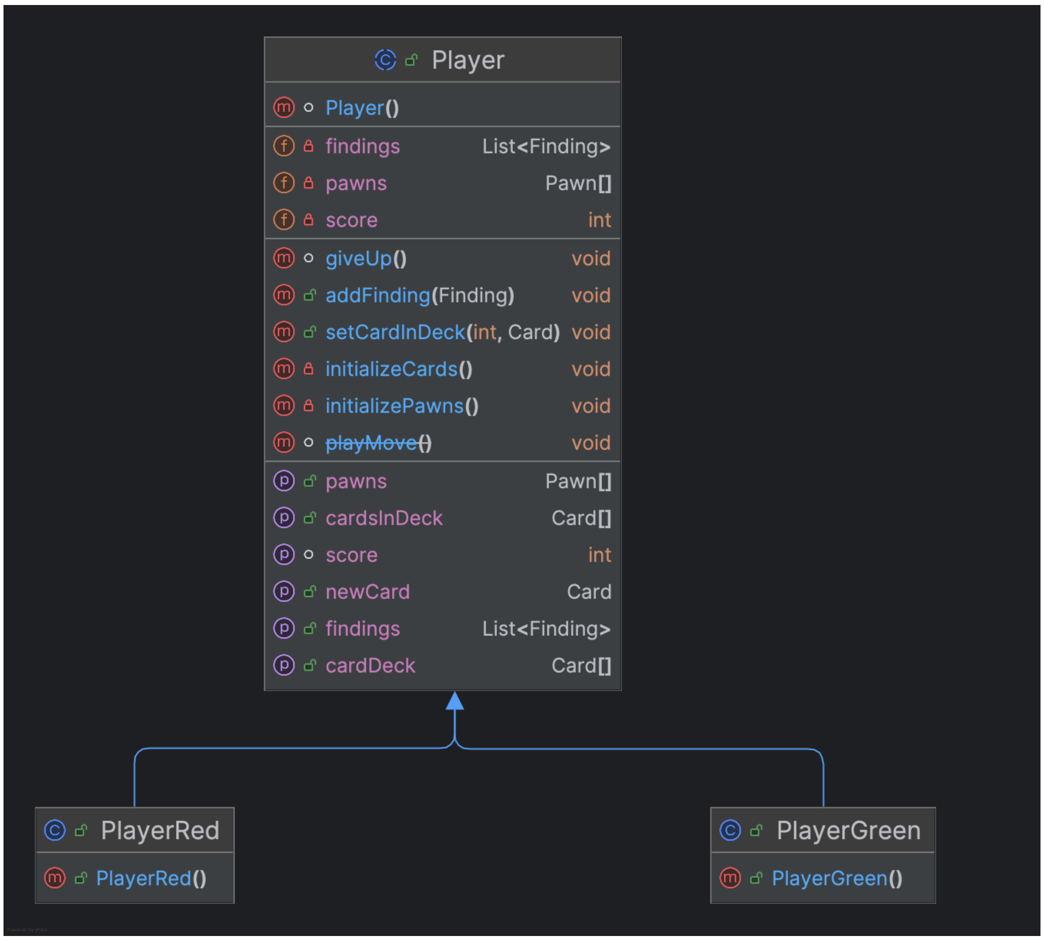
Key Details:

- The path's name is `PathName.ZAKROS_PATH`.

Overrides:

- `protected void initializeFindings()`: Defines the findings for the path (currently unimplemented).

Player UML:



Player

The `Player` class is an abstract representation of a player in the game. A player has:

- A score.
- A deck of cards.
- A collection of pawns.
- A collection of findings.

Key Details:

- The `PlayerRed` and `PlayerGreen` subclasses allow for future customization of behavior or traits for each player.

- The design ensures that only two players exist, simplifying independent management.

Public Methods:

- `Player()`:
  - Constructs a new player with:
    * An initial score of 0.
    * A hand of 8 cards.
    * 4 initialized pawns.
- `void setCardInDeck(int cardIdx, Card card)`:
  - Sets a specific card in the player's deck at the given index.
- `void setCardsInDeck(Card[] cards)`:
  - Replaces the player's entire deck of cards.
- `void addFinding(Finding finding)`:
  - Adds a finding to the player's collection of findings.
- `int getScore()`:
  - Returns the player's current score, calculated based on findings and pawn positions.
- `Card getNewCard()`:
  - Gets a new card from the available card stack. (Precondition: stack is not empty.)
- `Card[] getCardDeck()`:
  - Returns the deck of cards the player currently possesses.
- `Pawn[] getPawns()`:
  - Returns the player's pawns (4 total: 3 archeologists and 1 Theseus).
- `List<Finding> getFindings()`:
  - Returns the player's findings, which can include fresco, rare, or snake goddess findings.

Protected/Private Methods:

- `void initializeCards()`:
  - Initializes the player's hand of cards.
- `void initializePawns()`:
  - Initializes the player's pawns.
- `void giveUp()`:
  - Ends the game and exits the application. The player calling this method is considered to have lost.
- `void playMove()` (`@Deprecated`):
  - Represents the player's move in the game. This method is unlikely to be used.

—

PlayerGreen

The `PlayerGreen` class extends `Player` and represents the green player in the game.

Key Details:

- Created to allow future customization of unique abilities or traits for the green player.

Public Methods:

- `PlayerGreen()`:
  - Constructs a green player by calling the `Player` constructor.

—

PlayerRed

The `PlayerRed` class extends `Player` and represents the red player in the game.

Key Details:

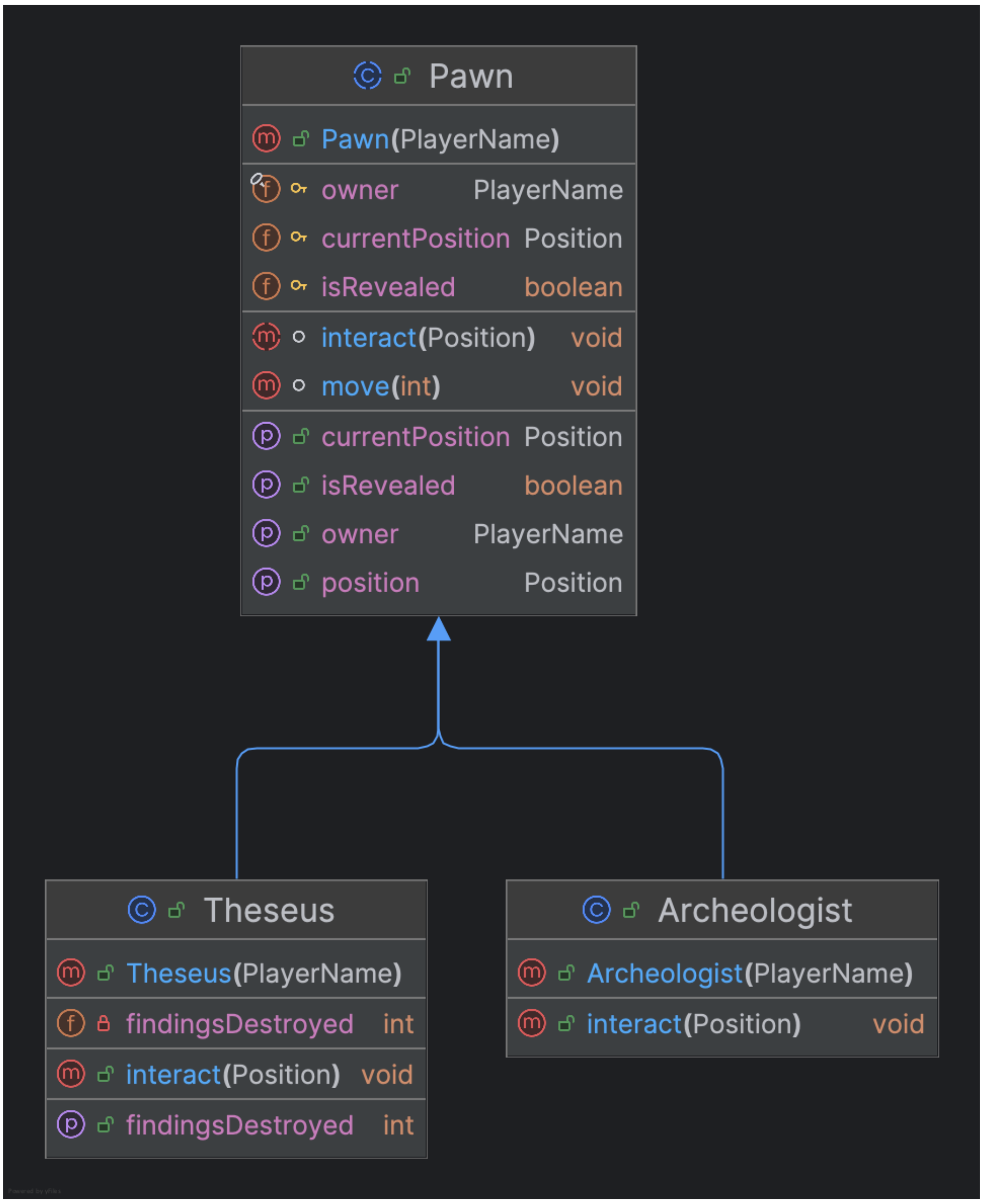
- Created to allow future customization of unique abilities or traits for the red player.

Public Methods:

- `PlayerRed()`:
  - Constructs a red player by calling the `Player` constructor.

Pawn

Pawn UML:



The Pawn class represents a general pawn in the game. All pawns have:

- A position on the game board.
- An owner (either PlayerGreen or PlayerRed).

- A revealable identity.

Public Methods:

- `Pawn(PlayerName owner)`:
  - Constructor to initialize a pawn with an owner.
  - Sets the initial position to `null` and the pawn as unrevealed.
- `void interact(Position position)`: Abstract method to define the pawn's special interaction with a position.
- `void move(int steps)`: Moves the pawn by the specified number of steps.
- `Position getPosition()`: Returns the current position of the pawn.
- `PlayerName getOwner()`: Returns the owner of the pawn.
- `boolean isRevealed()`: Returns whether the pawn is revealed or not.
- `void setRevealed(boolean revealed)`: Sets the revealed state of the pawn.
- `void setCurrentPosition(Position currentPosition)`: Sets the current position of the pawn.

—

Archeologist

The `Archeologist` class extends `Pawn` and represents a pawn with the ability to excavate or ignore a finding.

Key Details:

- If the player chooses to exploit a finding, their archeologist will be revealed to their opponent.
- Can excavate a finding, including photographing frescoes.
- Can also choose to ignore a finding, leaving it available for other players (except frescoes).

Public Methods:

- `Archeologist(PlayerName owner)`:

  - Constructor to initialize the archeologist pawn with an owner.

- `void interact(Position position)`:
  - Allows the archeologist to interact with a position containing a finding.
  - Throws an exception if the position does not contain a finding (`SimplePosition` is invalid).

Preconditions:

- The position must contain a finding.

—

Theseus

The `Theseus` class extends `Pawn` and represents a pawn with the ability to destroy findings but not collect them.

Key Details:

- Theseus can destroy findings, but this reveals his position to opponents.
- He cannot collect findings.
- Can destroy up to 3 findings during the game.

Public Methods:

- `Theseus(PlayerName owner)`:
  - Constructor to initialize the Theseus pawn with an owner.
- `void interact(Position position)`:
  - Allows Theseus to interact with a position containing a finding.
- `int getFindingsDestroyed()`:
  - Returns the number of findings destroyed by Theseus.
- `void setFindingsDestroyed(int findingsDestroyed)`:
  - Sets the number of findings destroyed by Theseus.

Preconditions:

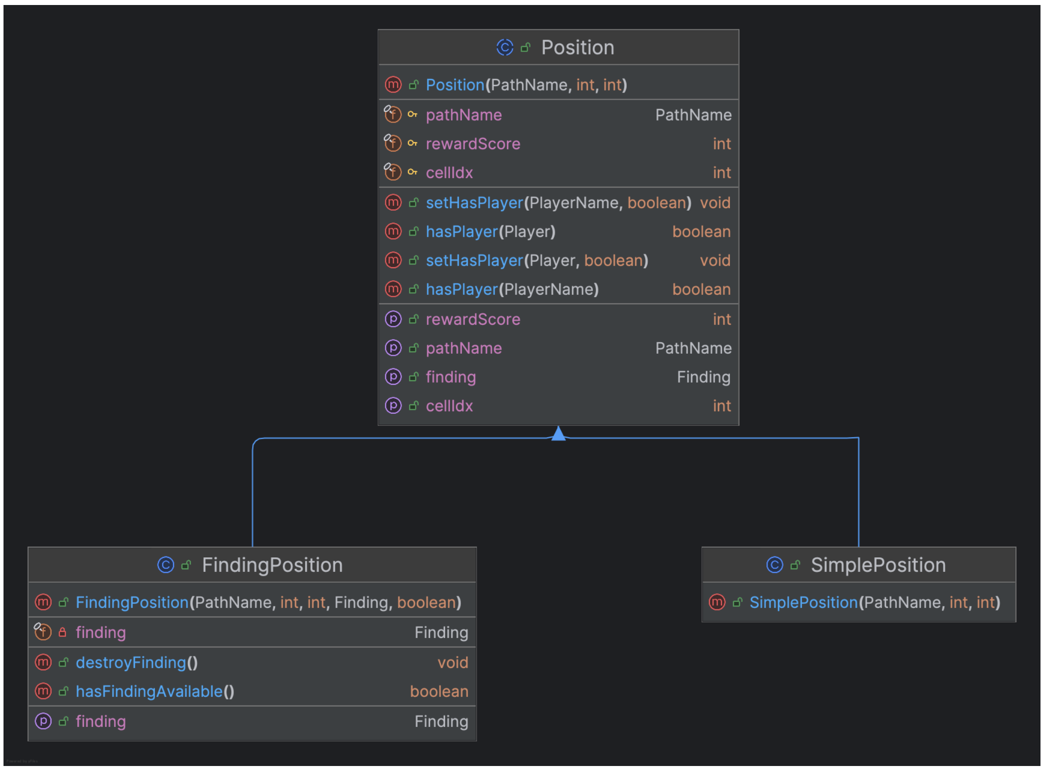
- The position must contain a finding.

Constants:

- `MAX_DESTRUCTION_COUNT = 3`: Maximum number of findings Theseus can destroy.

Position

Position UML:


Position
Position(PathName, int, int)
pathName PathName
rewardScore int
cellIdx int
setHasPlayer(PlayerName, boolean) void
hasPlayer(Player) boolean
setHasPlayer(Player, boolean) void
hasPlayer(PlayerName) boolean
rewardScore int
pathName PathName
finding Finding
cellIdx int
FindingPosition
FindingPosition(PathName, int, int, Finding, boolean)
finding Finding
destroyFinding() void
hasFindingAvailable() boolean
finding Finding
SimplePosition
SimplePosition(PathName, int, int)

The `Position` class represents a cell on a path. Each position may have:

- An index on the path.
- A reward score gained when the player passes by it.
- An optional finding (for subclasses that support it).

Key Details:

- Positions track which players are currently present.
- Subclasses define whether a position has a finding (`FindingPosition`) or does not (`SimplePosition`).

Public Methods:

- `Position(PathName pathName, int cellIdx, int rewardScore)`:
  - Constructor to initialize a position with a path name, cell index, and reward score.

- `PathName getPathName()`: Returns the path name this position belongs to.
- `int getCellIdx()`: Returns the index of this cell on the path (0–8).
- `int getRewardScore()`: Returns the reward score associated with this position.
- `void setHasPlayer(Player player, boolean hasPlayerValue)`: Sets whether a specific player is on this position.
- `boolean hasPlayer(Player player)`: Checks if a specific player is on this position.
- `Finding getFinding()`: Returns the finding in this position (default is `null`).

—

FindingPosition

The `FindingPosition` class extends `Position` and represents a position that contains a finding.

Key Details:

- A finding is available if it has not been collected or destroyed.
- Collecting the finding marks it as unavailable (unless it is a fresco).

Public Methods:

- `FindingPosition(PathName pathName, int cellIdx, int rewardScore, Finding finding, boolean findingAvailable)`:
  - Constructor to initialize a position with a finding and its availability status.
- `Finding getFinding()`:
  - Returns the finding if it is available.
  - Marks the finding as unavailable if it is collectable.
- `void destroyFinding()`: Destroys the finding, making it unavailable.
- `boolean hasFindingAvailable()`: Checks if the finding is available.

—

SimplePosition

The `SimplePosition` class extends `Position` and represents a position that does not contain a finding.

Key Details:

- Simple positions provide only a reward score and do not have findings.

Public Methods:

- `SimplePosition(PathName pathName, int cellIdx, int rewardScore)`:
  - Constructor to initialize a simple position with a path name, cell index, and reward score.

PathName

The `PathName` enum represents the names of paths in the game. Each path is associated with a numeric value.

Enum Values:

- `KNOSSOS_PATH(0)`: Represents the Knossos path.
- `MALIA_PATH(1)`: Represents the Malia path.
- `PHAISTOS_PATH(2)`: Represents the Phaistos path.
- `ZAKROS_PATH(3)`: Represents the Zakros path.

Public Methods:

- `int getValue()`:
  - Returns the numeric value of the path.
- `static PathName fromValue(int value)`:
  - Returns the `PathName` corresponding to the given numeric value.
  - Throws an `IllegalArgumentException` if the value is invalid.
- `static int playerObjectEncode(Player player)`:
  - Encodes a player object into a numeric value.
  - (Method implementation is incomplete in the current code.)

—

PlayerName

The `PlayerName` enum represents the names of players in the game. Each player is associated with a numeric value.

Enum Values:

- `PLAYER_RED(0)`: Represents the red player.
- `PLAYER_GREEN(1)`: Represents the green player.

Public Methods:

- `int getValue()`:

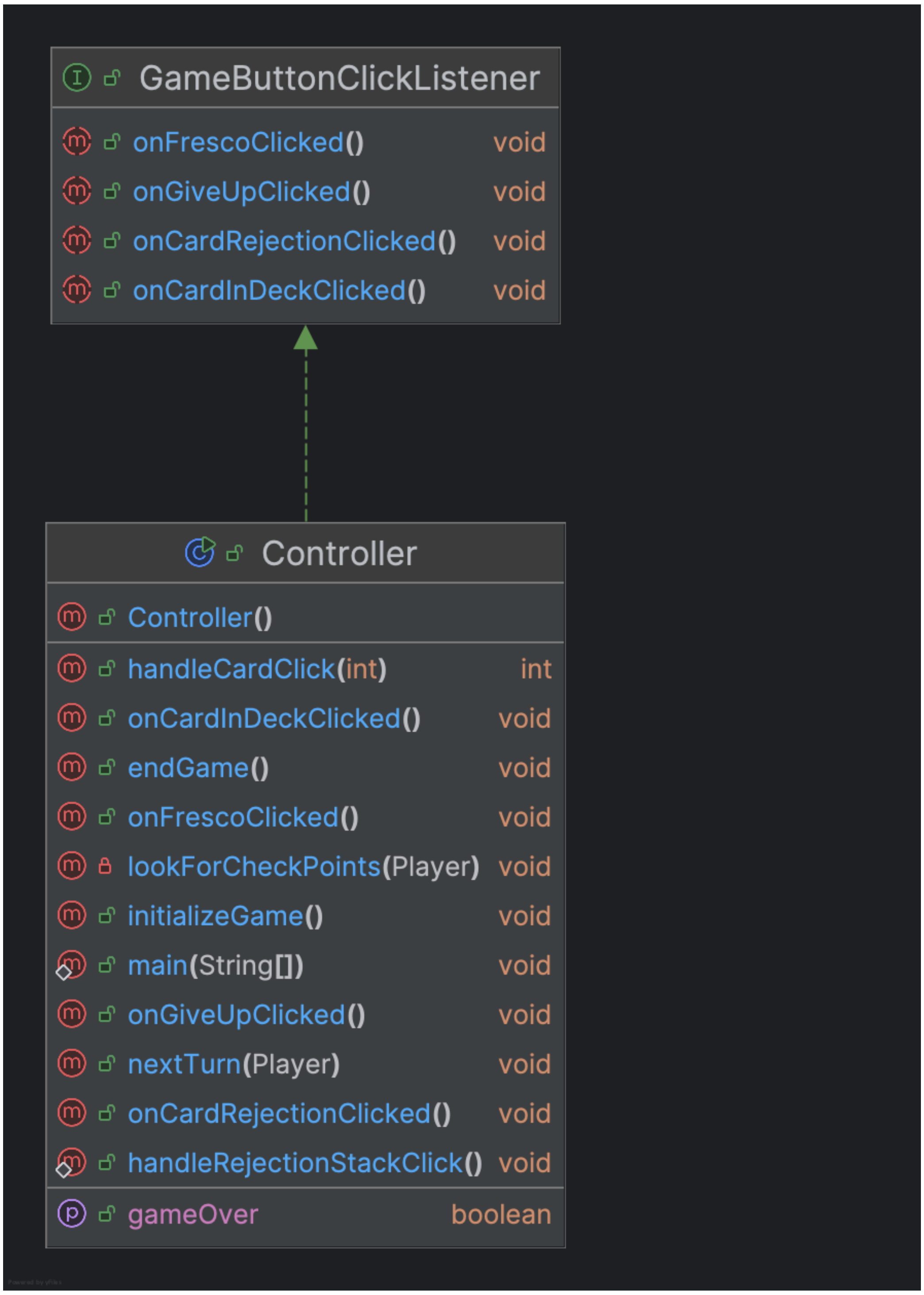
  - Returns the numeric value of the player.

- `static PlayerName fromValue(int value)`:
  - Returns the `PlayerName` corresponding to the given numeric value.
  - Throws an `IllegalArgumentException` if the value is invalid.

- `static PlayerName playerObjectEncode(Player player)`:
  - Encodes a `Player` object into a `PlayerName`.
  - Returns `PLAYER_RED` for `PlayerRed` objects and `PLAYER_GREEN` for `PlayerGreen` objects.
  - Throws an `IllegalArgumentException` for unknown player types.

GameButtonClickListener

Controller UML:



The `GameButtonClickListener` interface defines listeners for various button click events in the game. Implementations of this interface handle specific actions triggered by user interactions.

Key Details:

- Provides methods to handle clicks on game-related buttons or elements.

Public Methods:

- `void onCardRejectionClicked()`:
  - Triggered when the "Reject Card" button is clicked.
- `void onCardInDeckClicked()`:
  - Triggered when a card in the deck is clicked.
- `void onFrescoClicked()`:
  - Triggered when a fresco is clicked.
- `void onGiveUpClicked()`:
  - Triggered when the "Give Up" button is clicked.

—

Controller

The `Controller` class is the main controller of the application, integrating the model and view components of the game.

Key Responsibilities:

- Manages the game state and logic.
- Handles user interactions through the UI.
- Coordinates between the model (game state) and the view (UI).

Implements:

- `GameButtonClickListener` to handle user actions such as card interactions.

Public Methods:

- `void initializeGame()`:

  - Initializes the game state, including players, paths, and the card stack.
  - Configures the initial UI and sets the turn order (green plays first).

- `void nextTurn(Player player)`:
  - Advances the game to the next player's turn.
  - Handles card interactions, pawn movement, findings, and game state updates.
- `static void handleRejectionStackClick()`:
  - Handles the event where the rejection stack is clicked.
- `int handleCardClick(int cardId)`:
  - Handles the event where a card is clicked.
  - Returns an integer indicating the result of the click.
- `void endGame()`:
  - Closes the game window and ends the game.
- `boolean isGameOver()`:
  - Returns `true` if the game is over, otherwise `false`.

Overrides:

- `void onCardRejectionClicked()`:
  - Handles actions when the "Reject Card" button is clicked.
- `void onCardInDeckClicked()`:
  - Handles actions when a card in the deck is clicked.
- `void onFrescoClicked()`:
  - Handles actions when a fresco is clicked.
- `void onGiveUpClicked()`:
  - Handles actions when the "Give Up" button is clicked.

Game Logic:

- The game alternates turns between `PlayerGreen` and `PlayerRed`.
- Game logic includes:
  - Card interactions.
  - Pawn movements.
  - Findings collection.
  - Game state updates.
- The game ends when a player wins, gives up, or another end-game condition is met.

Internal State:

- `boolean isGreenTurn`: Tracks whether it's green's turn.
- `Map<Player, Card[]> lastCardsPlayed`: Stores the last cards played by each player.
- `CardStack cardStack`: Represents the stack of cards in the game.
- `Path maliaPath, knossosPath, phaistosPath, zakrosPath`: Paths in the game.
- `int checkPointsPassed`: Tracks the number of checkpoints passed.
- `static boolean rejectionStackClicked`: Indicates whether the rejection stack was clicked.

GameConstants

The `GameConstants` class defines various constants used throughout the game. These constants represent fixed values such as dimensions, numbers of game elements, and quantities of different cards and findings.

Key Details:

- Constants are grouped into categories such as screen dimensions, players and pawns, paths and cells, cards, and findings.
- These constants are publicly accessible and provide consistency across the game.

Constants:
Screen Dimensions:

- `WIDTH = 1920`: The width of the game screen in pixels.
- `HEIGHT = 1080`: The height of the game screen in pixels.

Players and Pawns:

- `NUMBER_OF_PLAYERS = 2`: The total number of players in the game.
- `NUMBER_OF_PAWNS = 4`: The total number of pawns per player.

Paths and Cells:

- `NUMBER_OF_PATHS = 4`: The total number of paths in the game.
- `NUMBER_OF_PATH_CELLS = 9`: The total number of cells in each path.

Cards:

- `NUMBER_OF_PATH_NUMBER_CARDS = 20`: The total number of number cards per path.
- `NUMBER_OF_CARDS = 100`: The total number of cards in the game.
- `NUMBER_OF_NUMBER_CARDS = 80`: The total number of number cards in the game.
- `NUMBER_OF_MINOTAUR_CARDS = 8`: The total number of Minotaur cards in the game.
- `NUMBER_OF_ARIADNE_CARDS = 12`: The total number of Ariadne cards in the game.

- `NUMBER_OF_DECK_CARDS = 8`: The number of cards in a player's deck.

Findings:

- `NUMBER_OF_RARE_FINDINGS = 4`: The total number of rare findings in the game.
- `NUMBER_OF_SNAKE_GODDESS_STATUES = 10`: The total number of Snake Goddess statues in the game.
- `NUMBER_OF_FRESCO = 6`: The total number of Fresco findings in the game.

Finding Positions:

- `NUMBER_OF_FINDING_POSITIONS_IN_PATH = 6`: The number of finding positions on each path.
- `NUMBER_OF_FINDING_POSITIONS_IN_CARD = 4`: The number of finding positions on cards.
- `NUMBER_OF_LAST_CARD_PLAYED_DECK = 4`: The number of last cards played section displayed on the player's menu.

GameConstants

The `GameConstants` class defines various constants used throughout the game. These constants represent fixed values such as dimensions, numbers of game elements, and quantities of different cards and findings.

Key Details:

- Constants are grouped into categories such as screen dimensions, players and pawns, paths and cells, cards, and findings.
- These constants are publicly accessible and provide consistency across the game.

Constants:

Screen Dimensions:

- `WIDTH = 1920`: The width of the game screen in pixels.
- `HEIGHT = 1080`: The height of the game screen in pixels.

Players and Pawns:

- `NUMBER_OF_PLAYERS = 2`: The total number of players in the game.
- `NUMBER_OF_PAWNS = 4`: The total number of pawns per player.

Paths and Cells:

- `NUMBER_OF_PATHS = 4`: The total number of paths in the game.
- `NUMBER_OF_PATH_CELLS = 9`: The total number of cells in each path.

Cards:

- `NUMBER_OF_PATH_NUMBER_CARDS = 20`: The total number of number cards per path.
- `NUMBER_OF_CARDS = 100`: The total number of cards in the game.
- `NUMBER_OF_NUMBER_CARDS = 80`: The total number of number cards in the game.
- `NUMBER_OF_MINOTAUR_CARDS = 8`: The total number of Minotaur cards in the game.
- `NUMBER_OF_ARIADNE_CARDS = 12`: The total number of Ariadne cards in the game.

- `NUMBER_OF_DECK_CARDS = 8`: The number of cards in a player's deck.

Findings:

- `NUMBER_OF_RARE_FINDINGS = 4`: The total number of rare findings in the game.
- `NUMBER_OF_SNAKE_GODDESS_STATUES = 10`: The total number of Snake Goddess statues in the game.
- `NUMBER_OF_FRESCO = 6`: The total number of Fresco findings in the game.

Finding Positions:

- `NUMBER_OF_FINDING_POSITIONS_IN_PATH = 6`: The number of finding positions on each path.
- `NUMBER_OF_FINDING_POSITIONS_IN_CARD = 4`: The number of finding positions on cards.
- `NUMBER_OF_LAST_CARD_PLAYED_DECK = 4`: The number of last cards played section displayed on the player's menu.