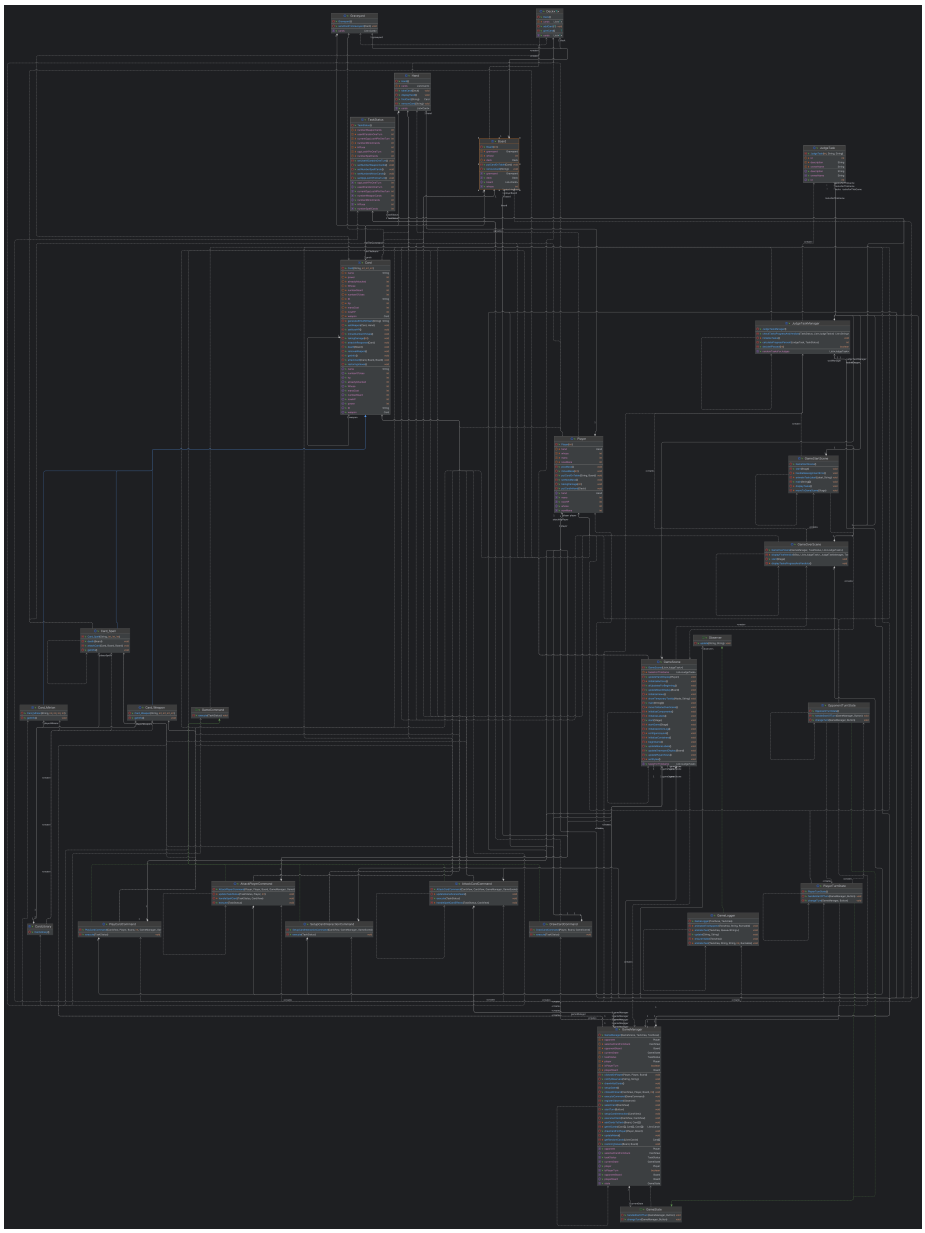
OOP Project Realization Report

Student: Zimnitski Aliaksei

Project Title: Art of Magic

Project Objective: Art of Magic is a unique strategic card game where players are judged after each battle, evaluating their performance in the duel. The assessment is based on various aspects of their play, including remaining health after the battle, the number of cards of a certain class used, and other key moments of strategy and tactics. Judges analyze these factors to deliver a verdict that can influence future games and strategies of the participants.

Class Diagram



Key Classes and Functionalities

The most important classes of the project are:

- **GameStartScene**
- **GameScene**
- **GameOverScene**
- **GameManager**
- **JudgeTaskManager**

GameStartScene

The **GameStartScene** class is designed to manage the initial setup and display of the game start scene in a JavaFX application. Here's a concise overview of its functionality and structure:

Fields

- `JudgeTaskManager taskManager`: Manages judge tasks.
- `List<JudgeTask> tasksForThisGame`: Holds assigned tasks for the game.
- `int reassignmentCount`: Tracks task reassignments.
- `final int MAX_REASSIGNMENTS`: Limits task reassignments to 3.

Key Methods

- `start(Stage primaryStage)`: Sets up the primary stage with UI components.
- `displayTasks()`: Updates tasks in the UI and handles reassignment.
- `animateTaskLabel(Label label, String fullText)`: Animates task descriptions.
- `moveToGameScene(Stage currentStage)`: Transitions to the main game scene.
- `handleReassignmentError()`: Shows an error when reassignment limit is reached.

Utility

Manages UI interactions using JavaFX, ensuring responsive updates through asynchronous operations.

Integration

Interfaces with `JudgeTaskManager` and uses JavaFX's `Stage` and `Scene` for UI management.

This class prepares the game environment by allowing players to view and adjust tasks before starting the game, ensuring they are equipped for gameplay.

GameScene

The **GameScene** class orchestrates the main gameplay interface for a JavaFX application focused on a card game. It sets up and manages the user interface, coordinating with a game manager to handle game logic and state.

Constructor and Key Fields

- Takes a list of `JudgeTask` objects that may influence gameplay, showing tasks that players need to manage or complete.
- `gameManager`: Central component managing game logic, player interactions, and maintaining game state.

Initialization and Layout

- `initializeComponents()`: Sets up UI components, binds them with action handlers, and connects them to the `gameManager`.
- `configureLayout()`: Organizes the game scene layout, arranging player hands, game boards, and action logs.

Gameplay Mechanics

- **Interaction Handling:** Button actions reveal deck counts and end turns, facilitating gameplay flow.
- **Game State Updates:** Methods like `updatePlayerViews` and `updateHandDisplay` keep the UI synchronized with game state changes, showing updates in real-time.
- **GameManager Integration:** Directs game flow, manages card interactions, and updates player states, crucial for enforcing rules and progressing the game.

Utility Functions

- **Tool Tips:** Provide immediate feedback or guidance based on player actions, enhancing the interactive experience.
- **Visual Updates:** Refresh UI components to accurately represent the game's current situation, including board status and graveyard contents.

This class not only manages the game's visual and interactive elements but also closely interacts with `GameManager` to ensure that gameplay mechanics are correctly implemented and the game state is accurately reflected in the UI.

GameOverScene

The **GameOverScene** class in JavaFX handles the display of the game's conclusion, showcasing the outcome, task results, and offering a restart option. Here's a more concise overview:

Purpose

- Provides feedback on the game's outcome and the completion of tasks. It also allows players to restart the game with a dedicated button.

Constructor Parameters

- Accepts `GameManager`, `TaskStatus`, and a list of `JudgeTask` objects to reflect the specific results of the game session.

Main Methods

- `start(Stage stage)`: Sets up and shows the game over scene, indicating whether the player won or lost and displays task verdicts.
- `displayTasksProgressAndVerdicts()`: Animates the completion percentages of tasks and provides a summary of outcomes.

Special Features

- **Animation:** Employs `Timeline` and `KeyFrame` animations for dynamic presentation of task completion percentages.
- **Dynamic Verdict Updates:** Updates the UI based on task results to indicate whether tasks were passed or failed.

This class effectively wraps up the game by giving players a clear understanding of their performance and an option to start anew.

GameManager

The **GameManager** class in JavaFX manages game logic for a card game, including player turns, card interactions, and task management.

Key Components

- **Player and Opponent Boards:** Tracks player states, including health and mana.
- **Card Management:** Manages actions like playing, drawing, and attacking with cards.
- **Task Integration:** Uses a `JudgeTaskManager` to incorporate tasks affecting gameplay.
- **Action Logs:** Maintains logs for player and opponent actions for game transparency.

Core Functionalities

- **Game Setup:** Initializes decks, shuffles cards, and draws initial hands for players.
- **Turn Management:** Controls the flow of the game by managing whose turn it is and what actions are available.
- **Card Interactions:** Handles interactions such as card attacks and plays, updating the game state accordingly.
- **Observer Notifications:** Uses an observer pattern to inform components about game events, enhancing game responsiveness.

Methods Overview

- `startTurn(Button endTurnButton)`: Triggers the beginning of a player's turn.
- `drawCardForPlayer(Player player, Board board)`: Draws a card to the player's hand, respecting hand size limits.
- `executeAttack(CardView attacker, CardView target)`: Manages attacks between cards, updating their states based on game rules.
- `updateMana()`: Updates the display of mana available to each player.

`GameManager` acts as the central hub for executing game logic, ensuring that gameplay follows defined rules and that the game state is correctly updated after each action.

JudgeTaskManager

The **JudgeTaskManager** class in Java manages the tasks set by judges in a card game, handling their initialization, random selection, and evaluation during gameplay.

Key Features

- **Task Initialization:** Sets up a list of predefined tasks with specific game-related goals, such as using a certain number of spell cards or maintaining health levels.
- **Task Selection:** Chooses a subset of tasks from the predefined list to apply in each game, enhancing variability and challenge.
- **Task Completion Evaluation:**
  - `calculateProgressPercent`: Calculates how much of a task has been completed based on the game's current stats, assigning a percentage to each task.
  - `decideIfPassed`: Assesses if a task is passed by comparing its completion percentage against a random threshold.

Methods

- `initializeTasks()`: Populates the task list with various objectives tailored to influence game strategies and outcomes.
- `getRandomTasksForJudges()`: Randomly selects tasks from the list to ensure each game presents unique challenges.
- `calculateProgressPercent(JudgeTask task, TaskStatus status)`: Measures task completion using game statistics, such as card usage or health metrics.
- `decideIfPassed(int progressPercent)`: Determines task success by comparing completion percentages to randomly generated thresholds, adding unpredictability to task outcomes.

This streamlined approach allows the `JudgeTaskManager` to significantly influence game dynamics, challenging players to meet diverse objectives while adapting to varying game conditions.

Criteria

Patterns

GameCommand Pattern

- **Purpose:** I use the GameCommand pattern to encapsulate game actions as objects. This approach allows actions like drawing a card, playing a card, or executing an attack to be treated as discrete, executable commands.

Observer Pattern

- **Real-Time Updates:** I use observers to update the UI and other components in real-time when game events occur, such as changes in player stats or game phase transitions.

GameState Pattern

- **State-Driven Logic:** The game behaves differently based on its current state—such as a player's turn, an opponent's turn.

Overall, these patterns play crucial roles in the architecture of my game. They help me manage complex game interactions more efficiently, ensure that the game interface reflects the current state accurately, and make the game logic easier to adapt and expand as development progresses.

GUI

In my game development process, I adhere to a structured architecture that distinctly separates the graphical user interface (GUI) from the game logic. This separation is crucial for maintaining clean code, simplifying debugging, and enhancing the scalability of the game.

Handling Exceptional States with Custom Exceptions

In my `GameStartScene` class, I address the criterion of handling exceptional states through the method named `handleReassignmentError()`. This method is triggered when the `reassignmentCount` exceeds the defined `MAX_REASSIGNMENTS`. It effectively manages the error by displaying an alert to the user, informing them that no further task reassignments are allowed, thus preventing any disruptions in game setup due to excessive changes. This error handling ensures the game maintains its intended flow and rules integrity.

Multithreading

In the `GameStartScene` class, I've utilized multithreading within the `displayTasks()` method to manage the UI updates efficiently. This approach ensures that the

fetching and displaying of tasks, which might involve time-consuming operations like fetching data or processing it, do not block the main UI thread.

Lambda expression

In the `GameManager` class, I've implemented the `drawInitialCards()` method using a lambda expression to streamline the process of drawing initial cards for both the player and the opponent. This method leverages the `IntStream.range` function to iterate five times, representing the initial number of cards each player starts with.

Handlers

In the `GameOverScene` class, one of the examples where I've implemented event handlers is for the restart button. Upon pressing this button, the current game stage is closed, and a new game is initiated by creating a fresh instance of the `GameStartScene`.

Generic class

In my project, I designed a generic `Deck` class that is tailored to manage a collection of cards in a game environment. This class is parameterized with a generic type `<T>` that extends `Card`, allowing it to handle any specific type of card defined in the game.

RTTI

In my implementation of the `AttackPlayerCommand` class within the game's command structure, I have utilized RTTI to effectively determine the class type of the cards involved in gameplay actions, particularly during attacks. This approach allows me to apply specific game logic based on the type of card being used in the attack.

Overview of Major Version Releases and Key Changes Between Them

08.04.2023: Submitted first version of project.
12.05.2023: Submitted final version of project.

Changes

1. Added Run-Time Type Information (RTTI)
2. Introduced Generic Deck Class
3. Incorporated Lambda Expressions
4. Implemented Multithreading
5. Added `handleReassignmentError` Method
6. Integrated Observer Pattern
7. Enhanced Game Logic