Design Introduction

Overview:

In this project, our aim is to bridge the gap between entertainment and learning in a seamless and engaging manner. Our specific game, Atlasventure, was born out of the need to make geography more accessible and engaging. The game invites players to interact with diverse geographical locations through a series of images and challenges. By integrating educational content within a gaming framework, Atlasventure aims to enhance players' geographical knowledge and cultural awareness in an enjoyable, interactive manner.

In this section, we focus on the core functionalities that underpin the game's educational goals. For example, we outline the design of a robust game save system that enables players to easily resume their exploration, ensuring a seamless learning journey. The blueprint also details the integration of a hint system, conceived to aid players during more challenging stages without undermining the learning process. Furthermore, we discuss strategies for dynamically adjusting the game's difficulty, tailoring the experience to match the player's learning curve. This technical overview is designed to illuminate the foundational elements of Atlasventure, and showcase our approach to creating a game. It will provide a blueprint for the product we are going to develop.

Objectives:

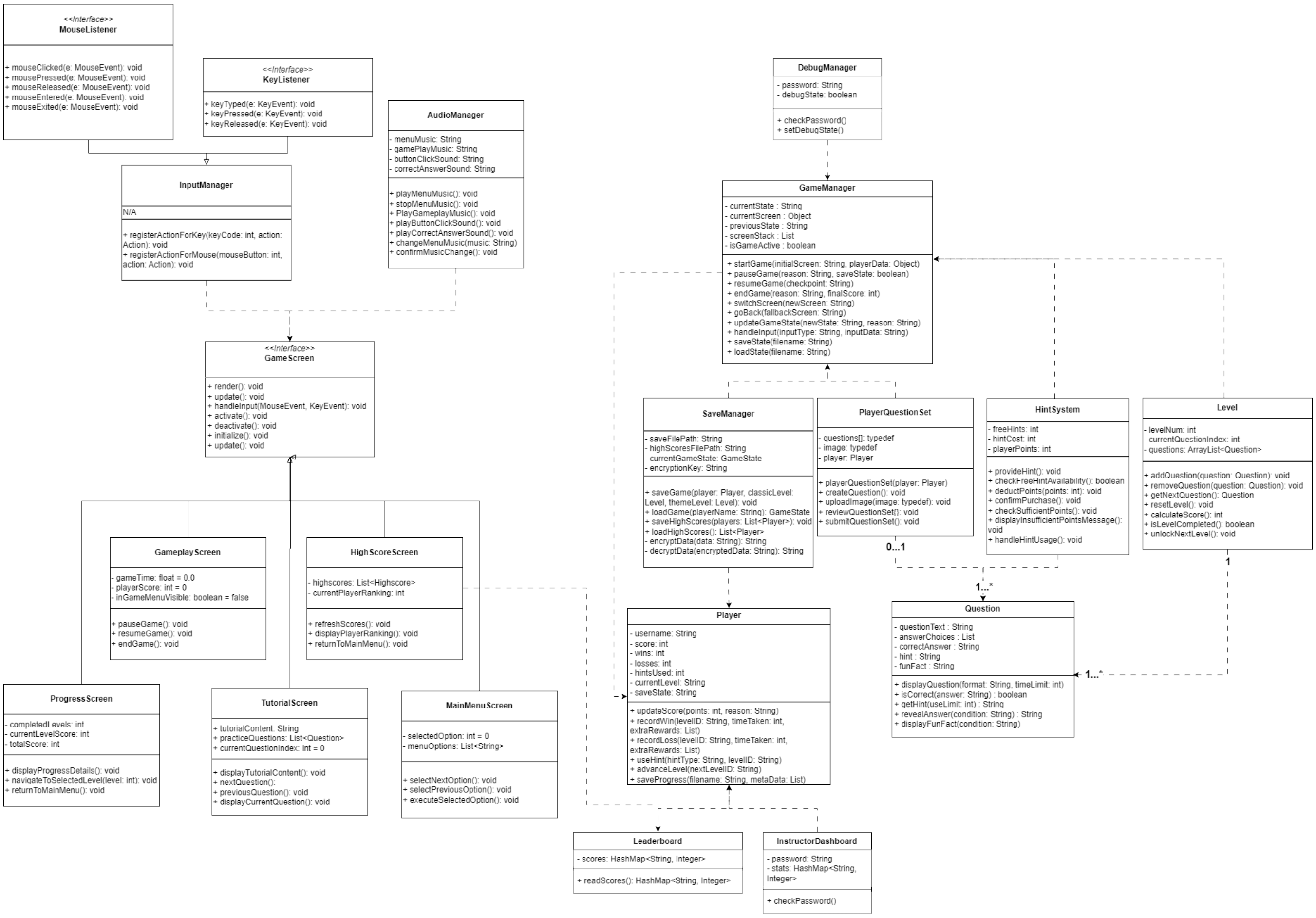
The software will be designed to provide an educational gaming experience focused on geography. It will aim to enhance the ‘players knowledge through interactive, engaging gameplay that includes various levels of difficulty and educational challenges. The system will be structured around a player-friendly graphical interface, and it will support both mouse and keyboard interactions. The software will support navigational keys and keyboard shortcuts to allow users to move seamlessly through all the UI elements and screens we will have. This will help in speeding up the interaction process for experienced users while maintaining simplicity for beginners. The UI will also be designed with intuitiveness in mind, so that users can naturally navigate through the game without needing extensive instructions and going back to the tutorial menu over and over again. Elements like the menus, buttons, and other interface elements will be clearly labeled and logically arranged to reduce the confusion and enhance the user experience. The components will include a main menu which will offer various options such as starting new games, loading saved games, accessing tutorials, viewing leaderboards, and exiting the game. The game will also feature multiple screens, including gameplay, tutorial, high-score list, and progress/results screens, each with specific sub-pages to facilitate navigation and enhance the learning experience. Since the gameplay mechanics are designed to be educational and challenging players will be able to not only learn while playing the game but also apply their critical thinking skills for geography knowledge. The game will include a scoring system, feedback for correct or incorrect actions, and a progression mechanism to encourage continuous learning and engagement. Additionally, a hints system will also be provided to aid players during gameplay, with the option to use their points to purchase additional hints. In our game, there is also a distinctive feature called the “Instructor dashboard,” and this feature will allow the educators or TA’s to monitor each player’s progress. The game also includes a debug mode for testing and development purposes, to ensure that a efficient and a user-friendly experience.

The user interface design will be designed in a clear manner. The design will include clear, readable fonts, with large buttons and icons that are easy to navigate, and it will also feature a organized layout to ensure the key information and the game controls are easily accessible. To ensure consistency and familiarity, the design and the colour scheme will also be maintained to reduce the learning curve for new users. There will be some level of consistency for the buttons and controls uniform colour themes, and similar layouts across different sections. By having these familiar design patterns, it will facilitate intuitive navigation and interaction. In terms of the feedback and responsiveness, visual and auditory feedback will be intreated to the overall design. By doing so, all users will be able to receive immediate and clear responses to their action. This will include, animations, colour changes, and even sounds following a user interaction, providing some affirmation that an action has been successfully completed or guiding the user back on track if an error occurs. For integrating the educational content, the game design will include elements of progression and motivation to keep users engaged and learning. This will include levels of increasing difficulty, rewards for achieving certain milestones, and visual indicators of progress, such as progress bars.

References:

UML Class Diagram

Class Diagram:



<<Interface>>
MouseListener
+ mouseClicked(e: MouseEvent): void
+ mousePressed(e: MouseEvent): void
+ mouseReleased(e: MouseEvent): void
+ mouseEntered(e: MouseEvent): void
+ mouseExited(e: MouseEvent): void
<<Interface>>
KeyListener
+ keyTyped(e: KeyEvent): void
+ keyPressed(e: KeyEvent): void
+ keyReleased(e: KeyEvent): void
AudioManager
- menuMusic: String
- gamePlayMusic: String
- buttonClickSound: String
- correctAnswerSound: String
+ playMenuMusic(): void
+ stopMenuMusic(): void
+ PlayGameplayMusic(): void
+ playButtonClickSound(): void
+ playCorrectAnswerSound(): void
+ changeMenuMusic(music: String)
+ confirmMusicChange(): void
InputManager
N/A
+ registerActionForKey(keyCode: int, action: Action): void
+ registerActionForMouse(mouseButton: int, action: Action): void
<<Interface>>
GameScreen
+ render(): void
+ update(): void
+ handleInput(MouseEvent, KeyEvent): void
+ activate(): void
+ deactivate(): void
+ initialize(): void
+ update(): void
GameplayScreen
- gameTime: float = 0.0
- playerScore: int = 0
- inGameMenuVisible: boolean = false
+ pauseGame(): void
+ resumeGame(): void
+ endGame(): void
HighScoreScreen
- highscores: List<Highscore>
- currentPlayerRanking: int
+ refreshScores(): void
+ displayPlayerRanking(): void
+ returnToMainMenu(): void
ProgressScreen
- completedLevels: int
- currentLevelScore: int
- totalScore: int
+ displayProgressDetails(): void
+ navigateToSelectedLevel(level: int): void
+ returnToMainMenu(): void
TutorialScreen
+ tutorialContent: String
+ practiceQuestions: List<Question>
+ currentQuestionIndex: int = 0
+ displayTutorialContent(): void
+ nextQuestion():
+ previousQuestion(): void
+ displayCurrentQuestion(): void
MainMenuScreen
- selectedOption: int = 0
- menuOptions: List<String>
+ selectNextOption(): void
+ selectPreviousOption(): void
+ executeSelectedOption(): void
DebugManager
- password: String
- debugState: boolean
+ checkPassword()
+ setDebugState()
GameManager
- currentState : String
- currentScreen : Object
- previousState : String
- screenStack : List
- isGameActive : boolean
+ startGame(initialScreen: String, playerData: Object)
+ pauseGame(reason: String, saveState: boolean)
+ resumeGame(checkpoint: String)
+ endGame(reason: String, finalScore: int)
+ switchScreen(newScreen: String)
+ goBack(fallbackScreen: String)
+ updateGameState(newState: String, reason: String)
+ handleInput(inputType: String, inputData: String)
+ saveState(filename: String)
+ loadState(filename: String)
SaveManager
- saveFilePath: String
- highScoresFilePath: String
- currentGameState: GameState
- encryptionKey: String
+ saveGame(player: Player, classicLevel: Level, themeLevel: Level): void
+ loadGame(playerName: String): GameState
+ saveHighScores(players: List<Player>): void
+ loadHighScores(): List<Player>
- encryptData(data: String): String
- decryptData(encryptedData: String): String
PlayerQuestionSet
- questions[]: typedef
- image: typedef
- player: Player
+ playerQuestionSet(player: Player)
+ createQuestion(): void
+ uploadImage(image: typedef): void
+ reviewQuestionSet{}: void
+ submitQuestionSet(): void
HintSystem
- freeHints: int
- hintCost: int
- playerPoints: int
+ provideHint(): void
+ checkFreeHintAvailability(): boolean
+ deductPoints(points: int): void
+ confirmPurchase(): void
+ checkSufficientPoints(): void
+ displayInsufficientPointsMessage(): void
+ handleHintUsage(): void
Level
- levelNum: int
- currentQuestionIndex: int
- questions: ArrayList<Question>
+ addQuestion(question: Question): void
+ removeQuestion(question: Question): void
+ getNextQuestion(): Question
+ resetLevel(): void
+ calculateScore(): int
+ isLevelCompleted(): boolean
+ unlockNextLevel(): void
0...1
1...*
1
Player
- username: String
- score: int
- wins: int
- losses: int
- hintsUsed: int
- currentLevel: String
- saveState: String
+ updateScore(points: int, reason: String)
+ recordWin(levelID: String, timeTaken: int, extraRewards: List)
+ recordLoss(levelID: String, timeTaken: int, extraRewards: List)
+ useHint(hintType: String, levelID: String)
+ advanceLevel(nextLevelID: String)
+ saveProgress(filename: String, metaData: List)
Question
- questionText : String
- answerChoices : List
- correctAnswer : String
- hint : String
- funFact : String
+ displayQuestion(format: String, timeLimit: int)
+ isCorrect(answer: String) : boolean
+ getHint(useLimit: int) : String
+ revealAnswer(condition: String) : String
+ displayFunFact(condition: String)
1...*
Leaderboard
- scores: HashMap<String, Integer>
+ readScores(): HashMap<String, Integer>
InstructorDashboard
- password: String
- stats: HashMap<String, Integer>
+ checkPassword()

Class Descriptions:

Summary:

1. GameScreen: Abstract class or interface for different types of game screens (e.g., main menu, gameplay screen).
2. MainMenuScreen: Inherits from GameScreen; manages the main menu options.
3. GameplayScreen: Inherits from GameScreen; handles the gameplay mechanics, question display, and scoring.
4. TutorialScreen: Inherits from GameScreen; displays tutorial information and interactive practice questions.
5. HighScoreScreen: Inherits from GameScreen; shows the high-score list.
6. ProgressScreen: Inherits from GameScreen; displays the player's current progress and results.
7. GameManager: Manages game states (e.g., starting, pausing, resuming, ending the game) and transitions between game screens.
8. Player: Represents the user playing the game, including attributes like username, score, wins, losses, and number of hints used.
9. Question: Represents a question in the game, including the question text, answer choices, and correct answer.
10. Level: Encapsulates a collection of questions and manages the difficulty and progression through the game.
11. SaveManager: Manages saving and loading game states, including player progress and high scores.
12. InputManager: Handles input from the user, both from the mouse and keyboard shortcuts.
13. Leaderboard: Manages and displays the top player scores.
14. InstructorDashboard: Provides a view for instructors to monitor player progress, secured by a password.
15. DebugManager: Offers debug functionalities, like unlocking levels and editing game states, secured by a password.
16. AudioManager: Manages audio feedback and background music for different game states and actions.
17. HintSystem: Manages the distribution and use of hints during gameplay, including purchasing hints with points.
18. PlayerQuestionSet: Allows players to create and manage their own sets of questions.

Detailed Description:

GameScreen Interface:

The GameScreen interface is the blueprint for all game screens, ensuring they adhere to a uniform set of functionalities. It prescribes methods for rendering visual elements, handling user inputs via mouse and keyboard, and managing screen states—activating, deactivating, and initializing—essential for transitioning between screens. The update method allows each screen to refresh its content and check for interactions or conditions that necessitate changes. This design provides a consistent framework for different screen types, simplifying development and maintenance while enhancing the user experience through predictable and stable screen interactions.

MainMenuScreen Class (Implements GameScreen):

The MainMenuScreen class is central to the user's navigation experience, featuring attributes that control the menu's interactivity. The selectedOption attribute keeps track of the user's current selection, while menuOptions holds the possible choices available. The operations included, such as selectNextOption and selectPreviousOption, allow the user to navigate through the menu, and an additional operation, presumably named something like executeSelectedOption, would activate the selected menu item's function. These design choices in the MainMenuScreen class ensure the user's first interaction with the game is intuitive and accessible, providing a clear path from the main menu to other parts of the game.

GameplayScreen Class (Implements GameScreen):

The GameplayScreen class encapsulates the real-time aspects of the game's interactive environment. It includes a timer (gameTime) to track the duration of gameplay, a scoring mechanism (playerScore) to keep the player's score throughout the session, and a visibility toggle (inGameMenuVisible) for the in-game menu. These attributes are pivotal for creating a dynamic gaming session where time and score are crucial to the gameplay experience. Operations like pauseGame(), resumeGame(), and endGame() provide essential control over the game flow, allowing players to take breaks, resume with focus, or conclude their session. These choices ensure that the GameplayScreen is both functional and responsive to the player's needs, providing a comprehensive and engaging user experience.

TutorialScreen (Implements GameScreen):

The TutorialScreen class, extending from GameScreen, focuses on introducing players to the game through tutorial content and interactive practice questions. It includes attributes for managing tutorial content, a list of practice questions for engagement, and an index to keep track of the current question being displayed. Operations are designed to navigate through tutorial materials and questions, enhancing the learning experience by providing a structured approach to understanding game mechanics and strategies. This setup ensures players are well-prepared and confident as they progress to actual gameplay.

HighScoreScreen Class (Implements GameScreen):

The HighscoreScreen class is equipped with attributes and operations to manage and display high scores. It stores a list of high scores (highscores) and the current player's ranking (currentPlayerRanking). The refreshScores() operation updates the list of high scores, ensuring that the displayed information is current. displayPlayerRanking() allows the player to see their position relative to others, fostering a competitive environment. Finally, returnToMainMenu() provides a seamless way for players to navigate back to the main menu, maintaining a fluid user experience within the game. This class emphasizes competition and achievement, highlighting player success and encouraging replayability to improve rankings.

ProgressScreen Class (Implements GameScreen):

The ProgressScreen class is designed to provide players with a visual representation of their achievements and progress within the game. It showcases the levels completed, the score obtained in the current level, and the total score accumulated across all levels. Through operations like displayProgressDetails, players can see a detailed breakdown of their performance, while navigateToSelectedLevel allows them to revisit and replay completed levels, enhancing the game's replay value. The returnToMainMenu operation ensures that players can easily navigate back to the main interface, maintaining a user-friendly experience.

GameManager:

The GameManager class orchestrates the overall flow and state management of the game. It tracks the game's current state, the active screen, and maintains a history of screens for easy navigation. The methods provided allow for starting, pausing, resuming, and ending the game, as well as switching between screens and handling back navigation. This class also includes functionality for updating the game's state, processing player inputs, and saving or loading the game state. Through these mechanisms, GameManager ensures a seamless and coherent gameplay experience, managing transitions and states effectively to maintain engagement.

Player:

The Player class encapsulates all relevant data and actions pertaining to a game participant. It holds the player's chosen username, their score throughout the game, counts of wins and losses, the number of hints used—which also ties into any additional hints purchased—and the current level or stage they are in within the game. Furthermore, it manages a saveState object to preserve the player's game state for future sessions. Through methods like updateScore, recordWin, recordLoss, useHint, advanceLevel, saveProgress, and loadProgress, the class provides comprehensive management of the player's progress and achievements, enabling a personalized and continuous gaming experience. These functionalities allow the game to dynamically respond to the player's actions, track their progress accurately, and maintain engagement by rewarding successes and providing opportunities for improvement.

Question:

The Question class is designed to manage trivia questions within the game. It stores the question text, a list of answer choices, the correct answer, an optional hint for assistance, and an interesting fun fact related to the question. The methods include displayQuestion, which shows the question and answers to the player; isCorrect, which verifies if a given answer matches the correct one; getHint, providing a hint if available; revealAnswer, which discloses the correct answer post-attempt or in specific game modes; and displayFunFact, which shares an intriguing fact after the question is answered, enriching the learning experience.

Level Class:

The game's educational journey is coordinated by the Level class, which has features designed for a range of skill levels. It stores questions, keeps track of where the player is in a level, and modifies difficulty based on the player's ability. Methods to add or remove questions are used to edit levels, while others like assessing completion and advancing levels encourage continued engagement and a sense of achievement. This class is key in providing a smooth and enriching player experience, designed to motivate and educate within the gameplay.

SaveManager Class:

The SaveManager class is designed to ensure players can easily save and resume their game progress, maintaining the integrity and continuity of the gaming experience. By including attributes like saveFilePath and highScoresFilePath, we ensure that both game states and player achievements are securely stored and readily accessible. The encryptionKey attribute is used to ensure data security, safeguarding player information and progress. The methods of this class, including saveGame, loadGame, saveHighScores, and loadHighScores, provide a seamless interface for managing game data, enhancing user engagement by allowing players to pick up right where they left off, and fostering a competitive environment through high scores. Encryption and decryption methods strengthen the game's security, ensuring that player data remains confidential and tamper-proof. This design reflects a careful balance between user convenience, data security, and the overall integrity of the gaming experience.

KeyListener Interface:

The KeyListener interace is crucial for integrating keyboard interactions, enhancing the gameplay with control over actions. It listens for and responds to keyboard events, enabling specific actions based on player inputs. The keyTyped method is essential for text inputs, such as naming characters or answering in-game prompts, providing immediate feedback. keyPressed facilitates real-time actions like character movement or menu navigation, offering instant response to inputs.. Lastly, keyReleased is pivotal for ceasing actions initiated by keyPressed, ensuring players can accurately control game dynamics. This combination of methods allows for a comprehensive and responsive control scheme, making the game's interface fluid and intuitive.

MouseListener Interface:

The MouseListener interface is designed to facilitate direct interaction with the game's visual and interactive components. Through the mouseClicked method, players can engage with various elements within the game, such as navigating menus, making selections, or interacting with objects, with a simple click. The mousePressed and mouseReleased methods work to support actions like dragging objects or adjusting settings, enhancing gameplay with tactile feedback and precise control. Moreover, mouseEntered and mouseExited provide essential visual cues, highlighting interactive areas as the cursor moves over them and removing highlights when it leaves. This approach combines action initiation and visual feedback and ensures that every mouse interaction is meaningful.

InputManager Class (Implements KeyListener and MouseListener):

The InputManager class is designed to bridge user input with in-game actions, employing both MouseListener and KeyListener interfaces for comprehensive input handling. By allowing the game to map specific keys and mouse buttons to distinct actions, the class offers a high degree of customization and flexibility in gameplay controls. This design ensures that actions like moving, selecting, and interacting within the game environment can be tailored to individual preferences. The methods registerActionForKey and registerActionForMouse are central to this customization, facilitating a dynamic mapping system that can adapt to various gameplay styles and requirements.

Leaderboard:

The Leaderboard class manages the high scores in the game. It uses a HashMap to store scores associated with player usernames. It provides functionality to read and display these scores, fostering a competitive environment where players can compare their performance against others.

InstructorDashboard:

This class offers instructors a specialized view to monitor player progress. It requires password authentication for access and stores player statistics in a HashMap. Instructors can check player progress through the dashboard, which supports educational oversight and tracking learning outcomes.

DebugManager:

The DebugManager class is designed for use during development and testing. It allows developers to set the game in debug mode and requires a password for access, ensuring that debug features are not misused. Methods include checking the password and toggling the debug state, providing developers with tools to test and refine the game.

AudioManager:

The AudioManager class is responsible for handling all audio-related actions in the game. It stores references to the music and sound effects, such as menu music, gameplay music, and sounds for button clicks and correct answers. Methods include playing, stopping, and changing music, which allow for a dynamic audio experience, enhancing the game's atmosphere and providing auditory feedback to player actions.

HintSystem:

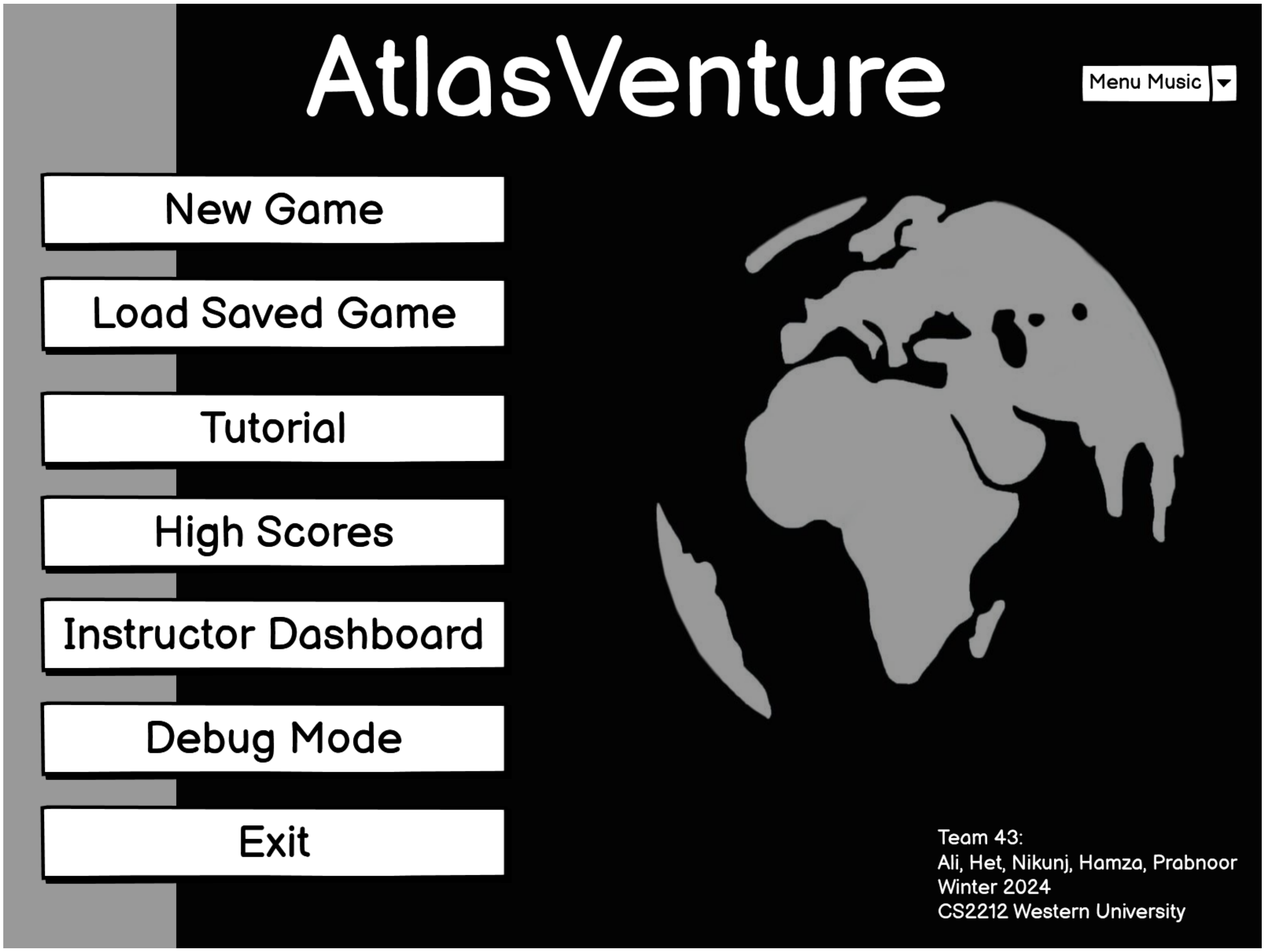
The HintSystem class manages the hints mechanism, offering players help when they're stuck. It keeps track of available free hints, their cost, and the player's points. Methods allow for providing hints, checking availability, deducting points for hints, and ensuring the player has enough points to purchase hints. This system is designed to aid learning and progression through the game while balancing the challenge by regulating hint usage.

PlayerQuestionSet:

This class allows players to create and manage their own sets of questions. It stores a list of questions and associated images, alongside the player's details. Players can create new questions, upload images for them, review their question sets, and submit them, which could be for personal practice or sharing with others.

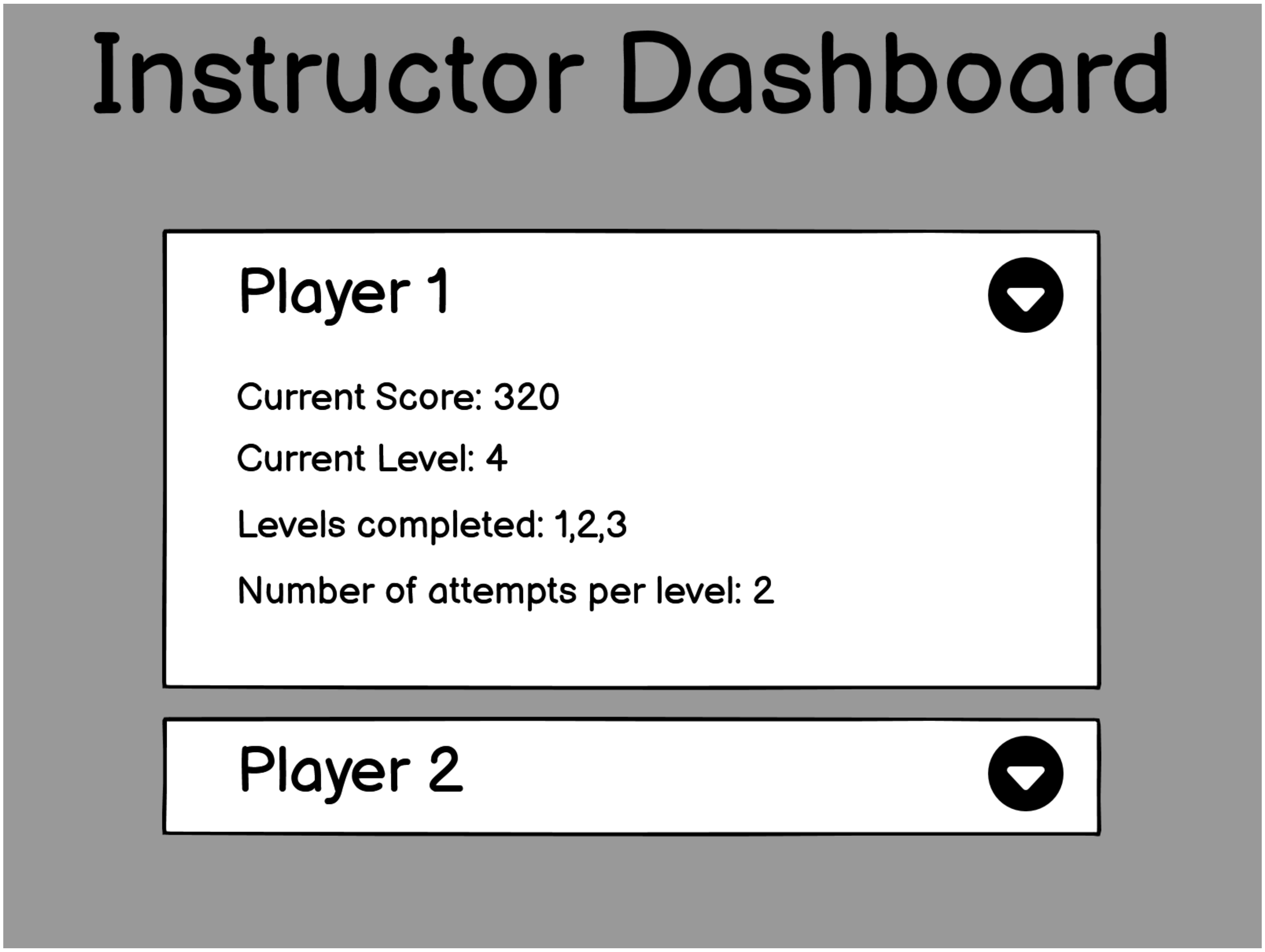
User Interface Mockups

- **Main Menu**: This is the gateway to the game, where players can start a new game, load a saved one, access the tutorial, view high scores, manage instructor settings, enter debug mode, or exit the game. The design prioritizes easy navigation and quick access to all parts of the game.



Team 43:
Ali, Het, Nikunj, Hamza, Prabnoor
Winter 2024
CS2212 Western University

- **High Scores**: This screen displays a leaderboard with player rankings. It lists players by their score, fostering a competitive atmosphere. The design is simple, emphasizing readability and straightforward access to players' standings.



Instructor Dashboard

Player 1

Current Score: 320

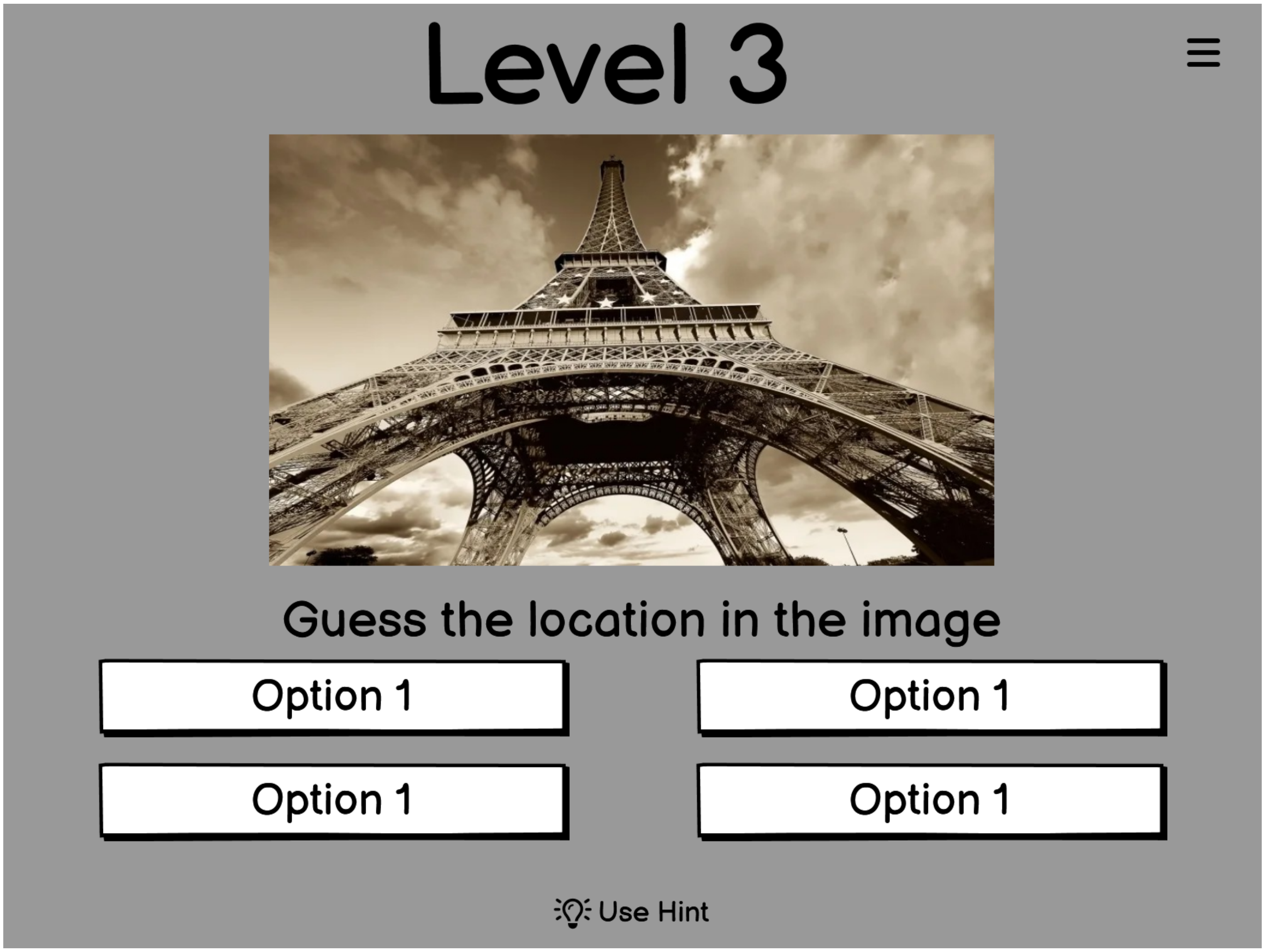
Current Level: 4

Levels completed: 1,2,3

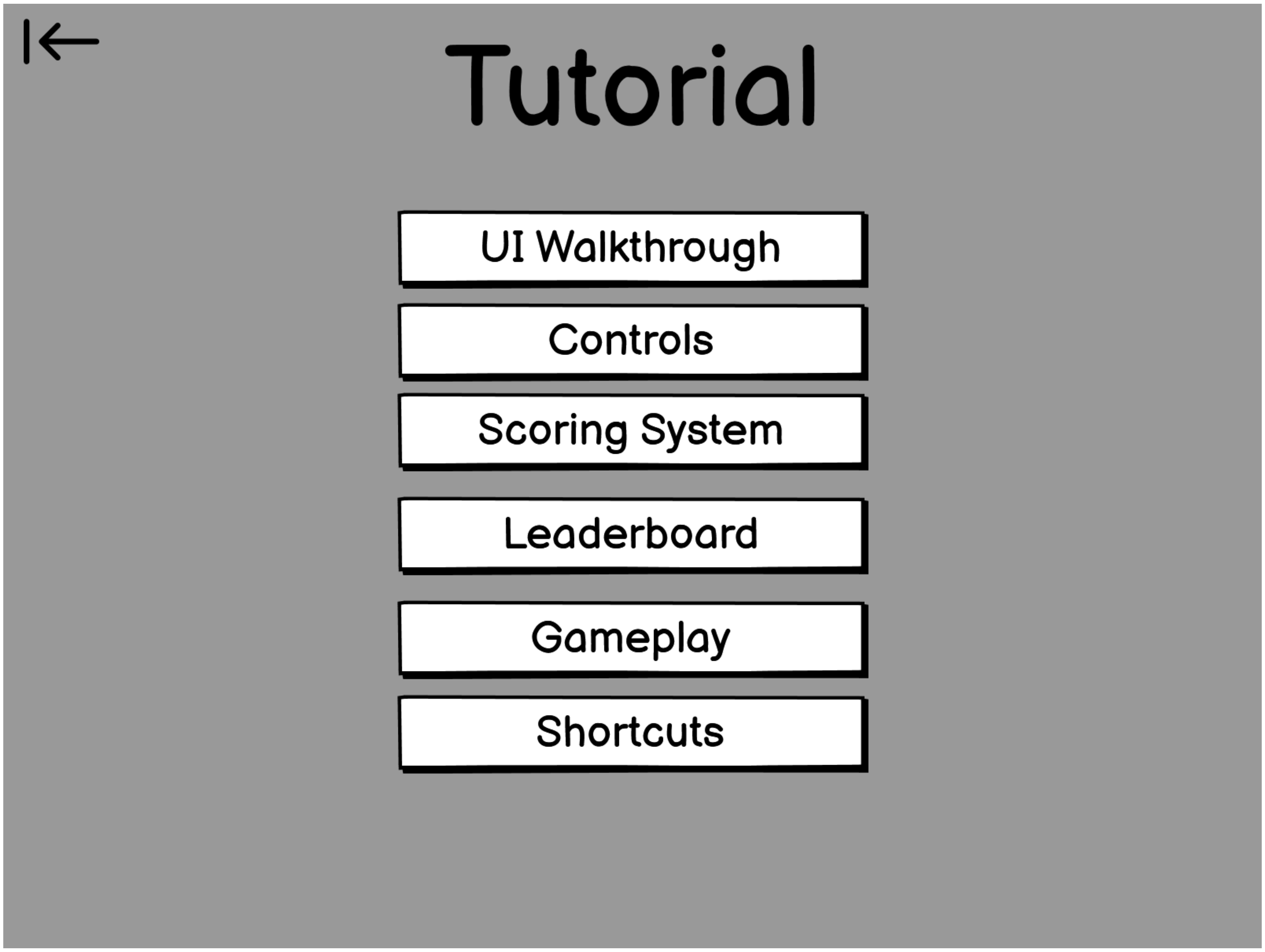
Number of attempts per level: 2

Player 2

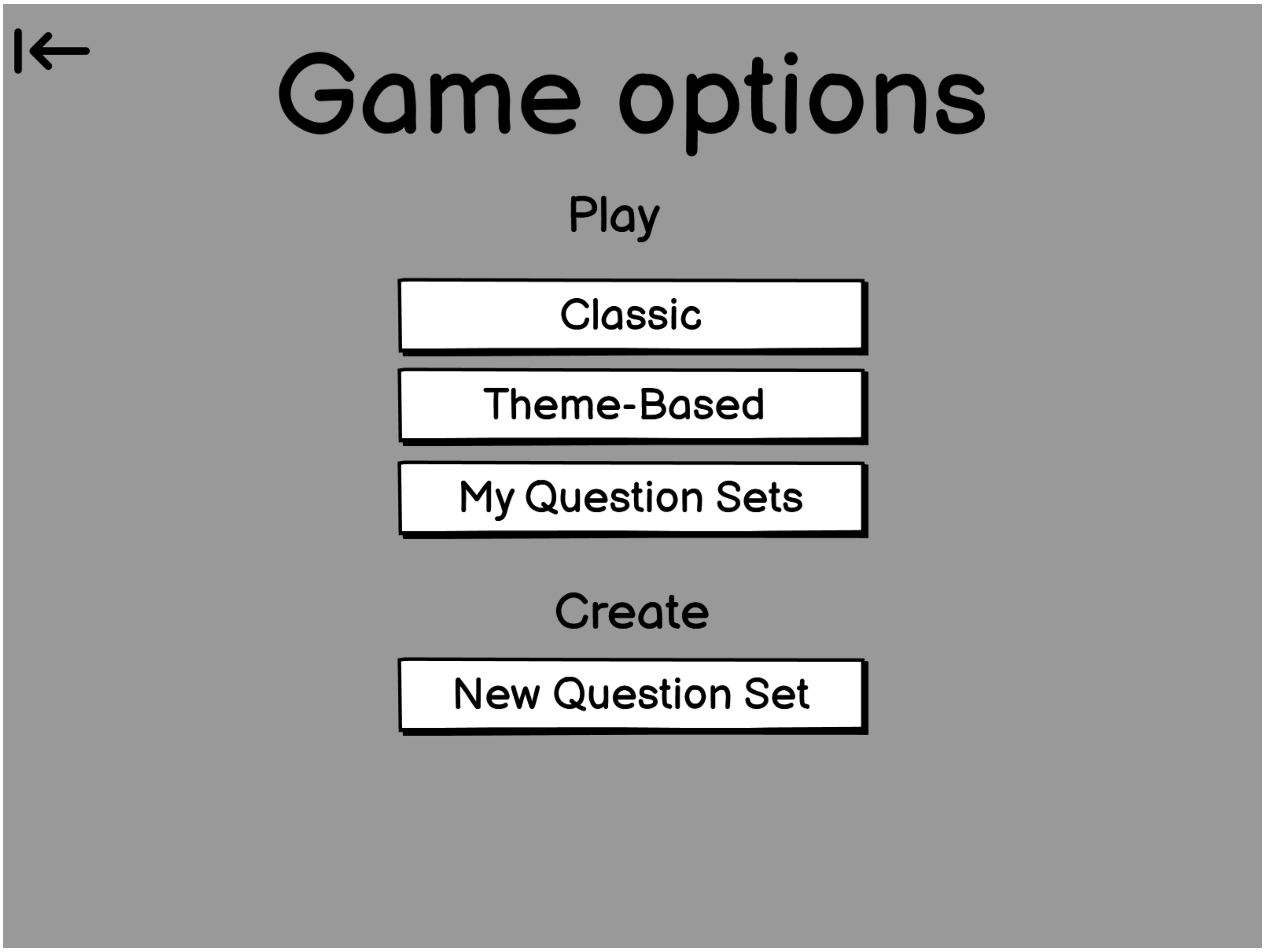
- **Level Gameplay**: Here, players are presented with a challenge, such as identifying a landmark. Options for answers are provided, and there's a hint button for assistance. The UI focuses on clarity and ease of interaction, ensuring players are engaged with the content without distractions.



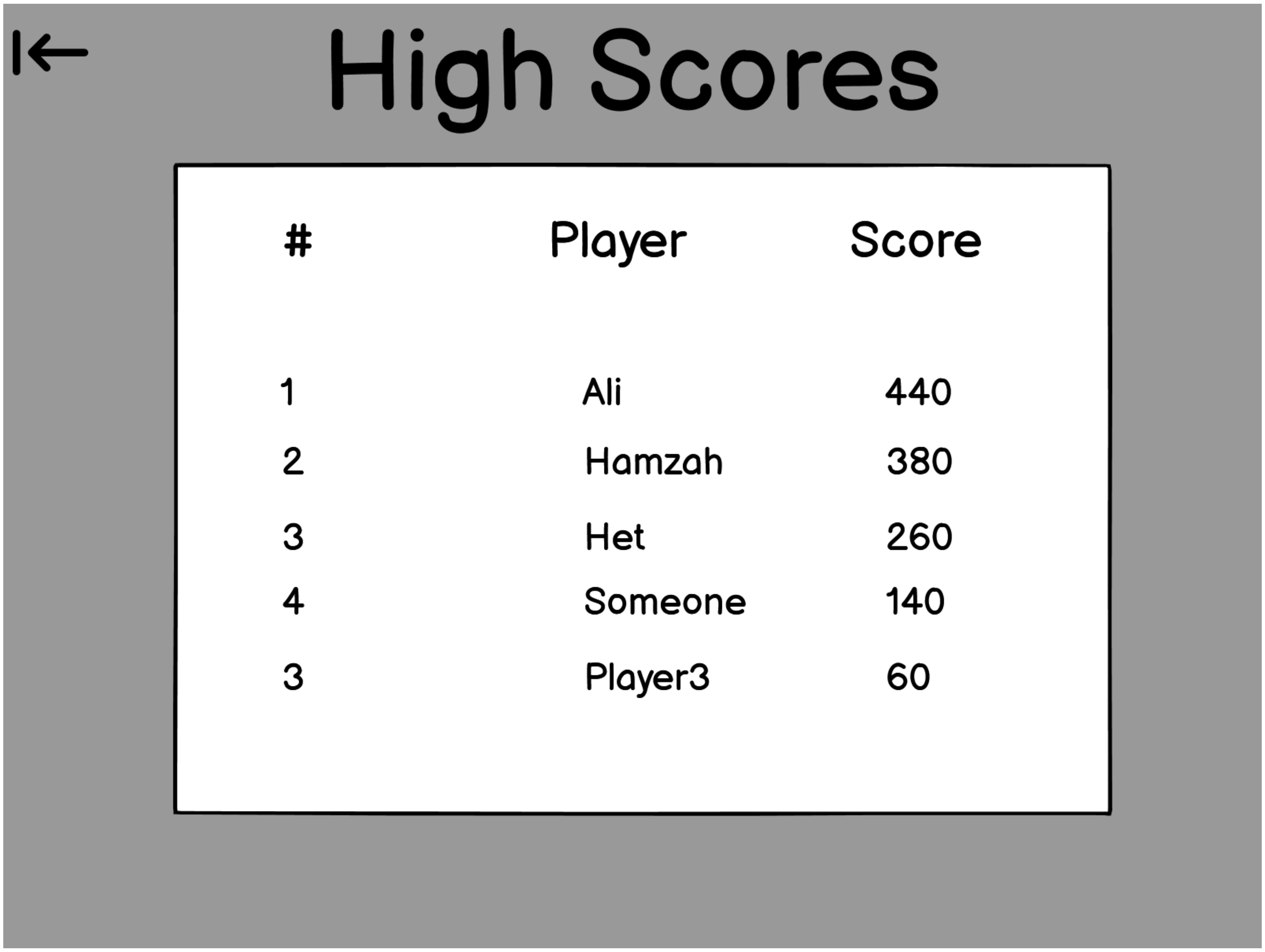
- **Tutorial**: This screen serves as an interactive guide, offering walkthroughs on different aspects like UI, controls, scoring, and gameplay. The design is instructional and user-friendly, aiming to equip players with the knowledge to enjoy the game fully.



- **Instructor Dashboard**: Aimed at educators, this screen tracks and displays student progress. It shows scores, levels completed, and attempts, allowing instructors to monitor and support learning outcomes. The design is functional, providing organized data for educational use.



- **Game Options**: This is where players select how they want to play (classic or theme-based) or create new question sets. The design here is about providing choices and fostering player creativity and customization.



High Scores

#	Player	Score
1	Ali	440
2	Hamzah	380
3	Het	260
4	Someone	140
3	Player3	60

File Formats

For our educational geography game, data persistence is a critical feature that allows us to store player information, game states, and high scores. To manage this data effectively, we will use JSON (JavaScript Object Notation) due to its readability, ease of use, and widespread support in many programming languages, including Java, which is the language of choice for our project.

Player Information:

- **File Name**: `players.json`
- **Description**: This file will store an array of player objects. Each player object encapsulates both performance metrics and the current game state.
- **Data Stored**:
    - `username`: A unique identifier for the player.
    - `metrics`: An object containing metrics such as the number of correct and incorrect responses, and the total hints used by the player.
    - `savedState`: An object that holds the current game state including the current level, the player's score, and the remaining hints available.

High Scores:

- **File Name**: `high_scores.json`
- **Description**: The high scores file will maintain a list of player scores, ranked from highest to lowest.
- **Data Stored**:
    - A list of objects, each containing a `username` and a `score`.

Example Files

Player Information:

- **File**: `players.json`
- **Contents**: Player usernames, scores, levels completed, hints used, and win/loss records.
- **Structure**: An array of player objects, each with the mentioned attributes.
- **Example**:

players.json

```
[
  {
    "username": "Player1",
    "metrics": {
      "correctResponses": 25,
      "incorrectResponses": 5,
      "hintsUsed": 2
    },
    "savedState": {
      "level": 5,
      "score": 1500,
      "remainingHints": 3
    }
  },
  {
    "username": "Player2",
    "metrics": {
      "correctResponses": 40,
      "incorrectResponses": 2,
      "hintsUsed": 0
    },
    "savedState": {
      "level": 8,
      "score": 2700,
      "remainingHints": 5
    }
  }
]
```

High Scores:

- **File**: `high_scores.json`
- **Contents**: Top scores across all players.
- **Structure**: An array of score objects, sorted by score in descending order.
- **Example**:

highScores.json

```
[
  {"username": "Champion", "score": 3200},
  {"username": "RunnerUp", "score": 3100},
  {"username": "ThirdPlace", "score": 3000}
]
```

Development Environments

Our team will develop the software in Java, with IntelliJ IDEA serving as our primary IDE due to its wide range of features for Java development. We intend to use external libraries like Java Swing for the GUI component and Gson for serializing player objects to JSON. Our code will be documented with Javadoc, and for testing, we'll use JUnit to write and run unit tests to ensure our application works properly across updates.

Design Patterns

Model-View-Controller (MVC) Pattern:

The MVC pattern particularly seems to be a good pattern to use for the educational game because of its clear vision of how data is processed, displayed, and interacted with. Given my game's complexity, involving diverse screens, interactive elements, and different data types such as player scores, levels, and statistics, these patterns will give us a structured approach to keeping the components modular. Using this pattern will allow for a clean separation of different functional areas: data management (model), user interface (view), and input/control logic (controller). Hence, the MVC architecture will help to have this more organized and modular approach to facilitate easier updates and maintenance for our game. In terms of how it will be applied, the implementation will be broken down into three parts. Firstly, the model will be designed to include all the game's core data and rules, such as tracking player scores, points, current level, and progress. The model will also be designed to notify the user of any changes so that we can ensure that there is data consistency throughout the gameplay without any direct interaction with the user inputs. So essentially, the model will react to the controller's instructions by updating this data. Secondly, for the view, this component will be responsible for showing all the visual elements of the game (the menu, game screens, leaderboards, and scores). This component will interpret the data from the model and display it in an organized format. By doing so, it will ensure that the user interface remains separate from the game's logic and data management. Hence, the view will update in response to changes in the model, so the player always has the latest game state. Finally, the controller component will handle the user input and any interpretation of player actions (i.e., navigating different menus, answering a question, executing different commands (pausing the game, using hints), and managing the game flows (starting new levels, saving game progress). So the controller will interpret these inputs, update the model accordingly, and finally signal them to the view to update the display based on the new state.

Singleton pattern:

The singleton pattern is another pattern the game will consider. This pattern is particularly relevant to our project since the game requires managing different components, such as game configuration settings, current game sessions, and high score databases. Using the singleton pattern will allow for ensuring that there's only one instance of each of these components throughout the game, preventing any inconsistent states or duplicate instances. For example, for the game configuration settings, we will have a singleton pattern to make sure that there's only one instance throughout the game. Specifically, this singleton instance will act as a centralized repository for all the configuration settings, like sound settings, difficulty levels, etc. By using this approach, any changes that are made to these settings will be globally distributed across all the game screens, ensuring that there is some consistency throughout the gameplay and navigation between screens. Then, for managing the current game session, we will use the Singleton pattern to maintain a unified and consistent state across all game interactions. By implementing this pattern, we will ensure that there exists only one instance that is responsible for gathering information for the current session, which will include vital player information such as progress, scores, and the current level. This Singleton instance will serve as the central hub for updating and tracking the player's journey throughout the game. This singular existence will also guarantee us that session data remains coherent and easily accessible across all game components and screens. Whether the player moves forward to higher levels, earns points, or navigates through different challenges, our Singleton instance will ensure that the session's integrity is maintained. Similarly, for the leaderboards, this design pattern will ensure there's only one instance of this database throughout the game. This means that regardless of where the developer or system accesses or updates the high scores, the same set of data will be displayed. Hence, with the Singleton pattern, when it comes to accessing and updating the leaderboard, it becomes simpler to manage and accurately track the scores.

Factory:

The factory pattern is another pattern that we will consider. This pattern is relevant to our project as it will allow us to create diverse in-game elements such as questions, challenges, and other interactive objects dynamically, according to the player's progression or chosen difficulty level (sequentially). This means that as players advance or change the difficulty settings, the game can adjust the complexity and variety of in-game content accordingly. So this pattern will be useful for us when certain game elements are created dynamically, without needing to go back and specify the exact classes of the objects.

Design Documentation Summary

The design documentation for our group project elaborately defines the software's structural and functional framework while following the pre-established requirements. This document begins with a main page and includes the project's title, an extensive revision history, which allows us to track our updates, and a generated table of contents, allowing for easy navigation through the document. Following that, we have an introductory segment that provides a concise summary of the software's goals and contextual background, with references to educate the reader on the project's scope and objectives. Afterwards, we introduce the UML class diagrams and user interface mockups, which are the main components of the document. The class diagrams provide a detailed visual representation of the system's architecture, describing classes, attributes, methods, and relationships among them. They are accompanied by brief textual explanations that clarify the fundamental design principles. Likewise, user interface mockups created with Balsamiq present an initial concept of the game's user interface and design framework, with descriptive narratives that support each design decision.

The documentation also outlines the data storage plan, which details the use of file formats such as JSON, which results in explicit data handling procedures. Also, the document specifies the development environment, emphasizing the tools, IDEs, and libraries that are compatible with Java development and are optimized for use on the Windows 10 platform. It also emphasizes the use of specific software design patterns to improve design efficiency. This approach ensures a comprehensive understanding of both high-level design requirements and the practical aspects of the development of our game.