CS 319 TERM PROJECT

Design Report Iteration 2

Section 3
Group 3B
Project Name: RISK 101

Group Members

Ramazan Melih DİKSU - 21802361
Aleyna SÜTBAŞ - 21803174
Yiğit ERKAL - 21601521
Baykam SAY - 21802030
Berk TAKIT - 21803147

Contents

1 Introduction

1.1 Purpose of the System

Risk 101 is a digital implementation of the popular board game Risk. Its purpose is to entertain the players. It implements all the major features of the classic Risk and extends the game by changing where the game takes place. Unlike classic Risk, Risk 101 is set in Bilkent. Players can select between different faculties and each faculty adds a new mechanic to the game.

1.2 Design Goals

- **Maintainability & Extensibility:** The game is designed using both the object-oriented programming paradigm and model view controller design pattern. This allows us to maintain the project easily as different parts of the project can be updated without harming the other parts. We can also add new features without completely changing the code using these methods.
- **Rapid Development:** As we have a limited timeframe for the project, rapid development is an important factor in our design. The game will be designed in a way to allow development in a few weeks.
- **Usability:** Our project will aim to have a user-friendly interface. It will feature understandable buttons, a clear UI, and an easy to access help screen.
- **High-Performance:** Our game should be responsive in order to not annoy the players. Therefore, we will design the game to have at most 0.5s response time for the actions users take. The game will also use low system resources which allows it to run on low-end machines.
- **Portability:** While our game is a desktop game, we decided to implement the game using Java. This allows the game to run on any modern desktop operating system.

2 High-level software architecture

2.1 Subsystem Decomposition

RISK101's system decomposition will follow the MVC (Model-View-Controller) architecture. During the decomposition, the goal is to ensure that the further enhancements will be implemented in an easy manner. Furthermore, in order to maintain a sufficient design process, the amount of coherence in between the components was maximized and the amount of the coupling in between the components was minimized.

As mentioned, the system of RISK101 is divided into components under the directives of MVC architecture. MVC architecture is the proper design choice for several reasons. In the following these reasons will be discussed regarding each layer of the MVC architecture, precisely in the order of View, Model and Controller layers.

- Initially, RISK101 is composed of a user interface, a game controller and a model consisting of a map and other map related objects. So it is the most efficient choice to divide the system into 3 subsystems which matches the properties of MVC architecture.
- View Layer:
    - RISK101 includes a user interface which fits into the view layer.
    - The user interface subsystem interacts with the user and acts as a bridge between the other components and the user, which fulfils the properties of the view layer.
- Controller Layer:
    - Gameplay of RISK101 is conducted with the assistance of certain management components. These management components work as a controller when grouped. Therefore, the management subsystem fits the controller layer.
- Model Layer:
    - The game related objects of RISK101 form the model subsystem which is handled by the management subsystem. This relation fulfils the property of the controller-model link in the MVC architecture. Therefore, the model subsystem fits the model layer.
- Other reasons:
    - MVC architecture is easy to understand and read.
    - MVC architecture clearly shows the relations between modules, therefore enabling a smooth communication process between teammates.

- MVC architecture saves time and enables using the resources effectively.
- A newly introduced modification will not affect the entirety of the model.

In the following, the relations and the dependencies of the modules implemented into the subsystem decomposition of RISK101 will be examined in detail while making references to *Figure.1*.

2.1.1 User Interface Subsystem

Due to the existence of the user interface subsystem, no direct changes can be made to the model subsystem by the user since the user interface subsystem is only linked to the management subsystem with certain limited actions. This increases the maintainability of RISK101. The user interface system consists of the Menu screens. First one is the Main Menu which can alert the management subsystem to start a new or a saved game through New Game and Load Game screens or open the Credits screen. Second one is the Pause Menu which is invoked during the gameplay according to the actions of the user. Through the pause menu, users may reach the Exit, Help and Setting screens. Access to these screens can also be done through the Main Menu. The user interface subsystem includes the Game End Screen which can only pop up when the management system declares that the game is over. There are also two more pop up screens. First for enabling the user to select the number of dice, second for enabling the user to select the number of armies to move. These two pop-ups will be enabled by the management system according to the state of the game.

2.1.2 Management Subsystem

Management system is the controller of RISK101. The most significant component is the Game Engine which can take the input from the user and traverse through the game states according to it. According to the state of the game, it will invoke the Sound Engine for sound adjustments. Furthermore, regarding the state changes in the game, it will handle the required updates in the map and other game related entities included in the model subsystem. It also uses Game Manager Menu to access the Pause Menu screen. Sound Engine adjusts the sound of the game according to the state of the game or the actions made by the users through the Settings frame which can be accessed from the Game Menu Management.

2.1.3 Model Subsystem

Model subsystem consists of the game entities which are map, areas and territories. It can only be accessed and altered by the management system.

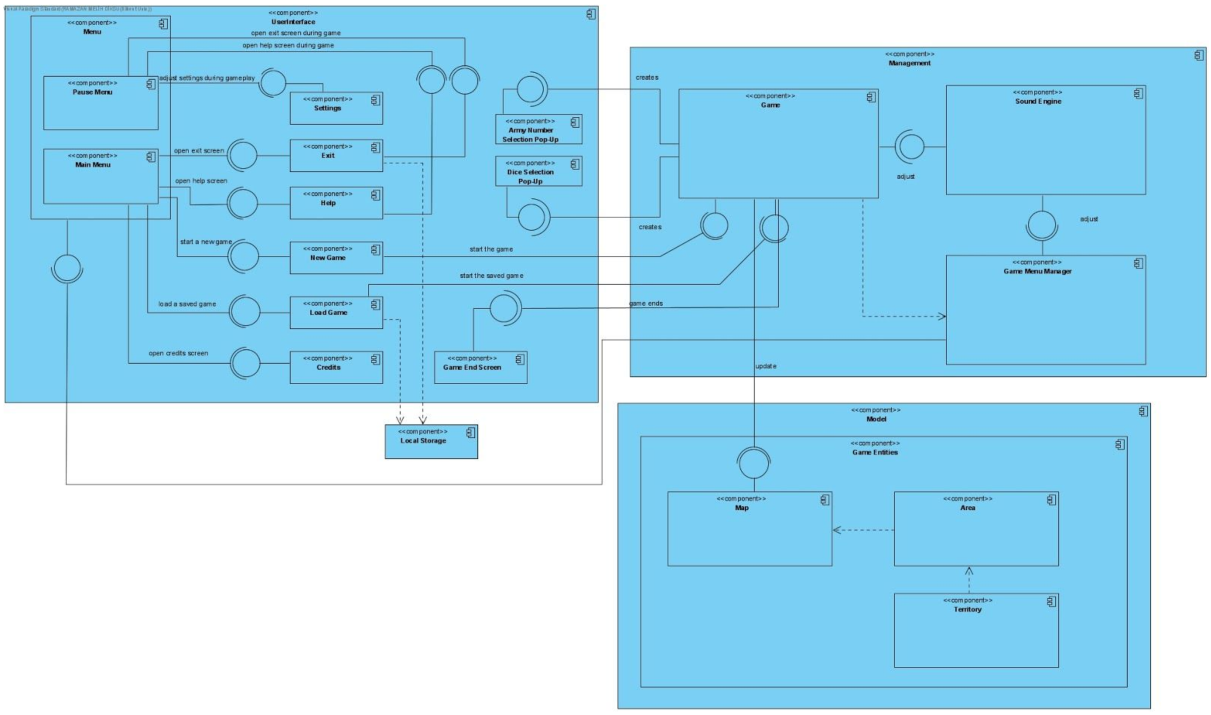


Figure 1. Model Subsystem

Higher Resolution: https://drive.google.com/file/d/15K5iCBnIWEsbr_m-vIBTRa2hu39kV0kz/view?usp=sharing

2.2 Access Control and Security

In Risk101, there is no need for any database or internet connection because our game will be played offline. In other words, players will use the same computer to play and if someone wants to play one of the old, saved games, he or she just loads the game and they can start to play Risk101. Hence, Risk101 will not have any access control and security control system. Every saved data such as play time or fastest win will be recorded in their own computer.

2.3 Persistent Data Management

Risk101 will use local data management. Since the game is designed to be played in a single computer, there will not be multiple users that will try to access the game data remotely. Thus, a database implementation is not required for Risk101. Players of

Risk101 will be given the choice of starting a new game or loading a game that was previously saved. These save files will be stored in the hard drives of the users and will be updated when another save of the game is created. This will be achieved by writing key components like the current situation of the map, players' objectives, territories that they hold, the turn count of the game and the current player turn into text files. Also, the users' preferences about the sound and display mode will be stored in local drivers and will be modifiable.

2.4 Hardware/Software Mapping

Risk101 will be implemented in Java. Therefore, Risk101 will require a Java Runtime Environment to run the game. Furthermore, Risk101 will require Java 7 or more recent versions because JavaFX is compatible for Java 7 and later versions.

Mouse, keyboard and a basic computer with essential software installed can compile and run the game. Mouse will be used to pass the turn, to choose a territory and also it will be used to roll the dice. Consequently, Risk101 will run most of the computers because of the basic system requirements.

To storage allocation, we will store all the data in a text file so our game will not require internet connection. Hence, Risk101 will not need any database and network to store the data.

2.5 Boundary Conditions

2.5.1 Initialization

For the main menu to load, no prerequisites are required. From the main menu, there are two ways to initialize a game.

First option is New Game, which creates a new save file and initializes the game to the initial state defined in the code.

The second option, Load Game becomes available only if there is at least one save-file on the local system. It initializes the game by reading that save-file and setting the game state to the saved state.

2.5.2 Termination

When the user wants to exit the game either by returning to the main menu or exiting the game through the pause menu, the save function will be called which will update the game's corresponding save-file to reflect the current game state.

2.5.3 Failure

If the save-file for a game is deleted while the game is running, the program will make a new save-file with the current state.

If a save-file is corrupted during saving due to an unforeseen event, and then the user tries to load the corrupted save-file, the program will inform the user that the save-file is corrupted and remove the corrupted save-file from the system.

If a performance related crash occurs during gameplay, the save-file will not be corrupted because it is only changed at the end of a session.

3 Low Level Design

3.1 Object design trade-offs

- **Rapid Development v. Functionality:** Because of our limited time, we are only planning to implement a single-device multiplayer game. If we had more time, we could add online matchmaking or single-player functionalities too. However, in this project, we will be prioritizing rapid development.
- **Security v. Usability:** As our game is meant to be played on a single device, there is no password protection or user login. Anyone that has access to the same device can continue the game which another user started. We chose usability over security because the game does not contain any private information. It also does not have access to the internet.

3.2 Final object design

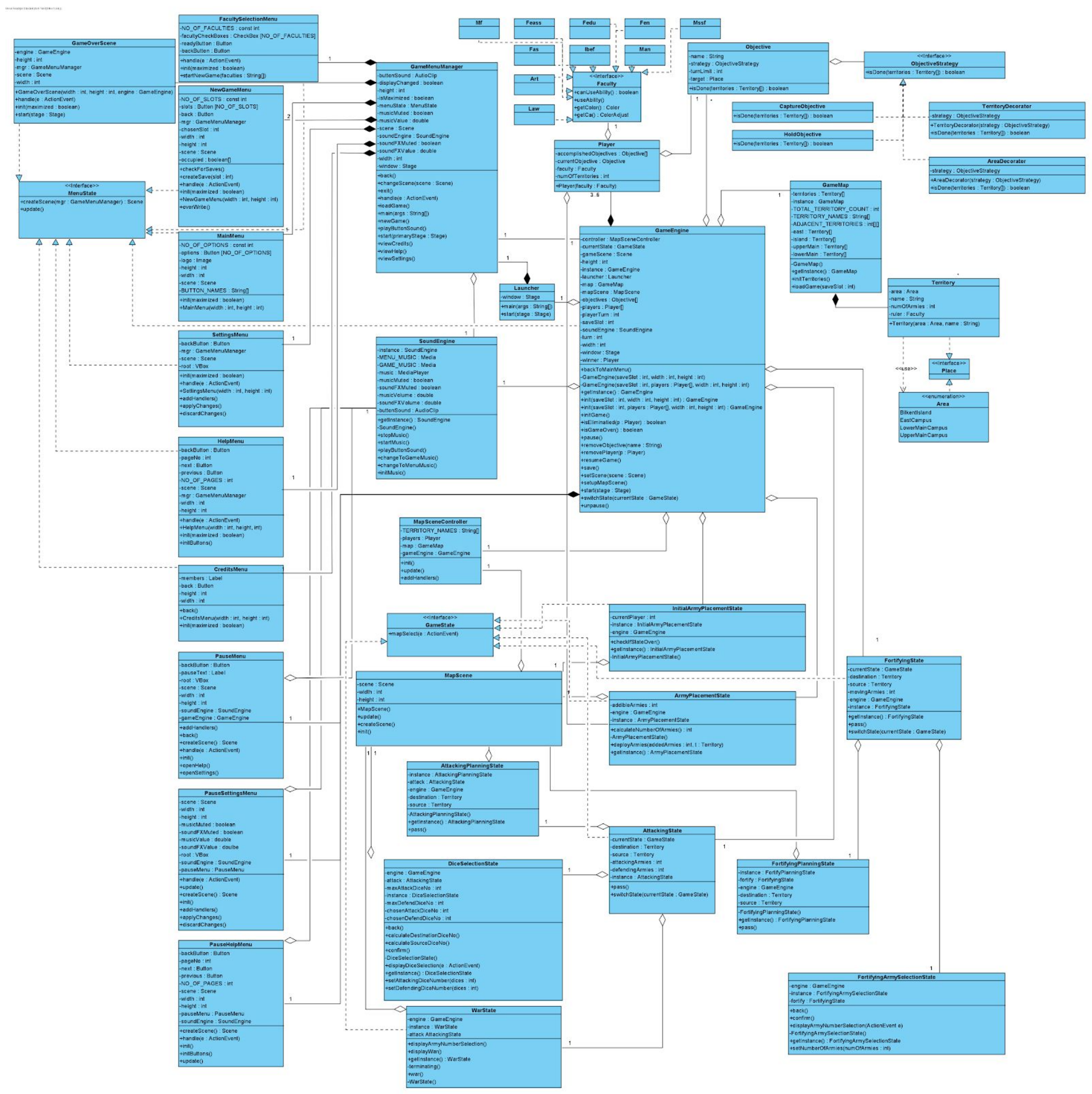


Figure 2. Class Diagram

Higher Resolution: [Group3B_ClassDiagram.png]()

3.3 Design Patterns

3.3.1 Singleton Design Pattern

In our design, we used Singleton design pattern to differentiate certain classes that needed to have single instances throughout the game. These classes were the game states, gameEngine, gameMap, soundEngine.

3.3.2 Strategy Design Pattern

3.3.3 Decorator Design Pattern

In our design, we used Strategy and Decorator design patterns together to create different Objective types with a more concise hierarchy and less effort. We have CaptureObjective and HoldObjective strategies to separate objectives that require capturing territories and holding territories for a certain amount of round. Moreover, we have Territory and Area Decorator classes that take an ObjectiveStrategy to create different combinations of objectives regarding territories or areas.

3.4 Packages

3.4.1 Internal Packages

1. **Menu Package**
   Consists of menu classes.
2. **Game Management Package**
   Consists of classes that manage the game and the menus.
3. **Game States Package**
   Consists of gameplay classes and game screens.
4. **Entity Package**
   Consists of game entities like the map and the players.

3.4.2 External Packages

1. **java.util**
   Data structures that will be included in our implementation such as arrays, arrayLists will be imported through util. Also, we will use the scanner from this package.
2. **javafx.scene**
   We will use several classes from this package to control the game windows and their internal interaction.
3. **javafx.application**
   We will use this package to create a desktop application from our implementation.
4. **javafx.geometry**
   We will use this package to organize the layout in our scenes.
5. **javafx.event**
   We will use this package to handle the interactions between the users and our game.
6. **javafx.util**
   We will use the Duration class from this package to add delays to descriptions on some buttons in the game menu
7. **javafx.stage**
   We will use this package for our desktop application.
8. **java.io**
   We will use this class to be able to write and read from text files.

3.5 Class Interfaces

Menu State

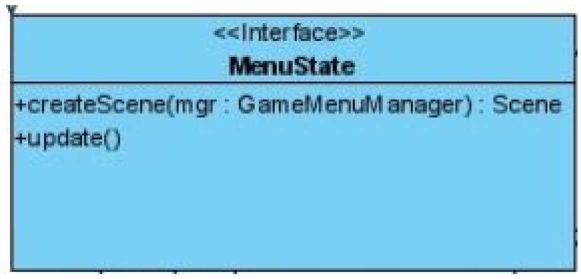
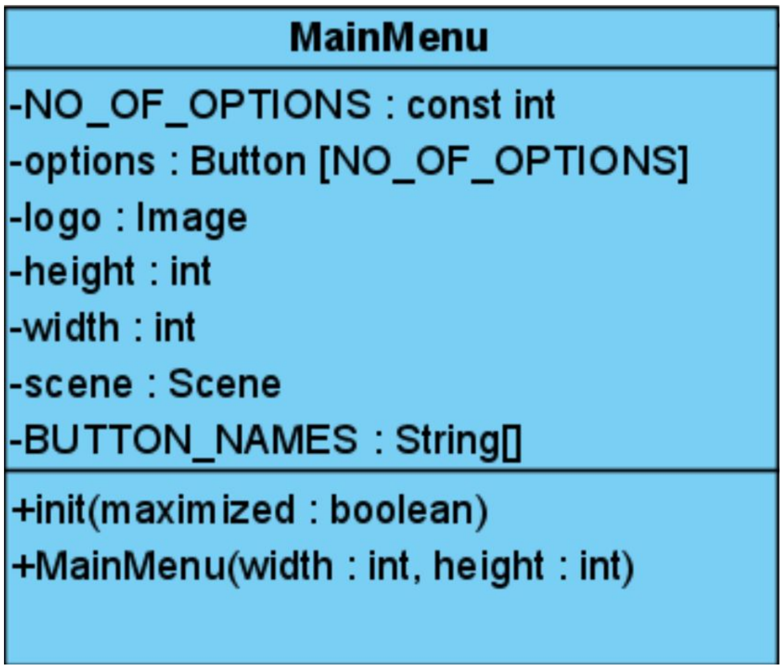


Figure 3. MenuState Interface

Operations:

+createScene(mgr: GameMenuManager): This operation creates and returns a scene for the GameMenuManager to display.
+update(): Used to update the MenuState.

MainMenu



MainMenu
-NO_OF_OPTIONS : const int
-options : Button [NO_OF_OPTIONS]
-logo : Image
-height : int
-width : int
-scene : Scene
-BUTTON_NAMES : String[]
+init(maximized : boolean)
+MainMenu(width : int, height : int)

Figure 4. MainMenu Class

Attributes:

(Some JavaFX attributes eg. Buttons are not included for brevity.)
-NO_OF_OPTIONS: Holds how many buttons there are on the main menu screen.

-logo: Holds the logo for the main screen.
-height,width: Holds the height and width of the scene.
-scene: The scene for the menu.
-BUTTON_NAMES: Holds the names for the buttons in the main menu.

Operations:

+init(maximized: boolean): Initializes the JavaFX attributes for scene creation.

FacultySelectionMenu

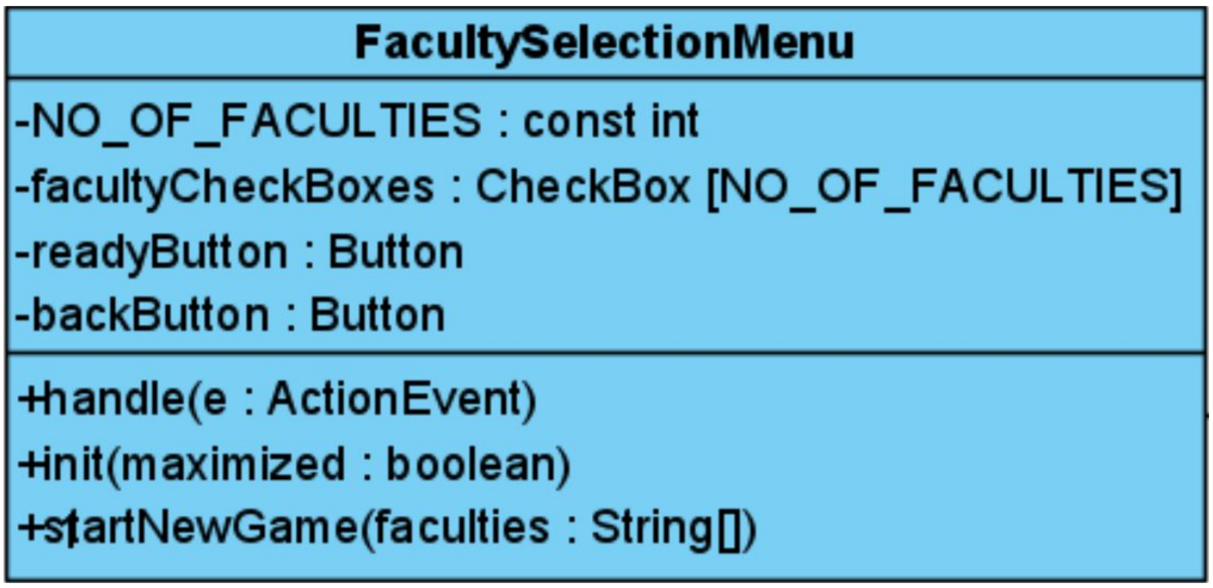

FacultySelectionMenu
-NO_OF_FACULTIES : const int
-facultyCheckBoxes : CheckBox [NO_OF_FACULTIES]
-readyButton : Button
-backButton : Button
+handle(e : ActionEvent)
+init(maximized : boolean)
+startNewGame(faculties : String[])

Figure 5. FacultySelectionMenu Class

Attributes:

(Some JavaFX attributes eg. Buttons are not included for brevity.)
-NO_OF_FACULTIES: This constant int holds the number of faculties implemented.
-facultyCheckBoxes: This array of JavaFX CheckBox objects holds a CheckBox for each faculty.
-readyButton: This JavaFX Button is used to proceed to the game.
-backButton: This JavaFX Button is used to go back to the Main Menu.

Operations:

+back(): This operation goes back to the Main Menu.
+init(): Initializes the FacultySelectionMenu.
+startNewGame(faculties: String[]): Used to start a new game with the faculties given in the String array.

NewGameMenu

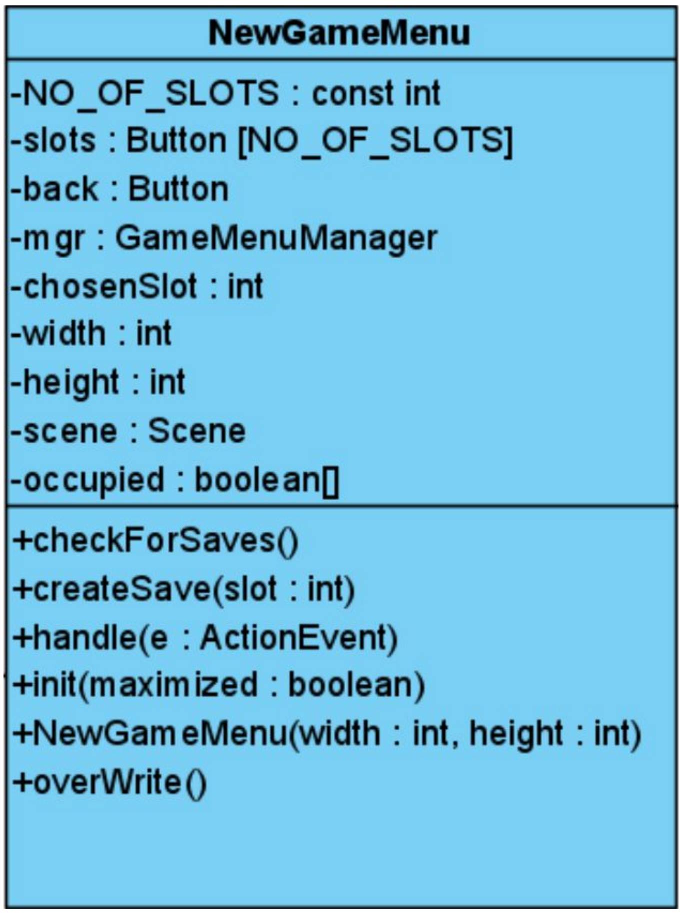


Figure 6. NewGameMenu Class

Attributes:

(Some JavaFX attributes eg. Buttons are not included for brevity.)
-NO_OF_SLOTS: This constant int indicates the number of total save slots.
-slots: This array of JavaFX Button objects holds a button for each save slot.
-backButton: This JavaFX Button object is used to go back to the Main Menu.
-mgr: Holds the GameMenuManager instance.
-chosenSlot: holds which save slot has been chosen.
-height,width: Holds the height and width of the scene.
-scene: The scene for the menu.
-occupied: This boolean array indicates whether a specified save slot is occupied.

Operations:

+checkForSaves(): This method checks for any available saves.
+createSave(int slot): This method creates a new save file in the specified slot.
+handle(ActionEvent e): Used to handle button events.

+init(boolean maximized): Initializes the scene.
+overWrite(): Deletes a save and writes a new save file in its place.

LoadGameMenu

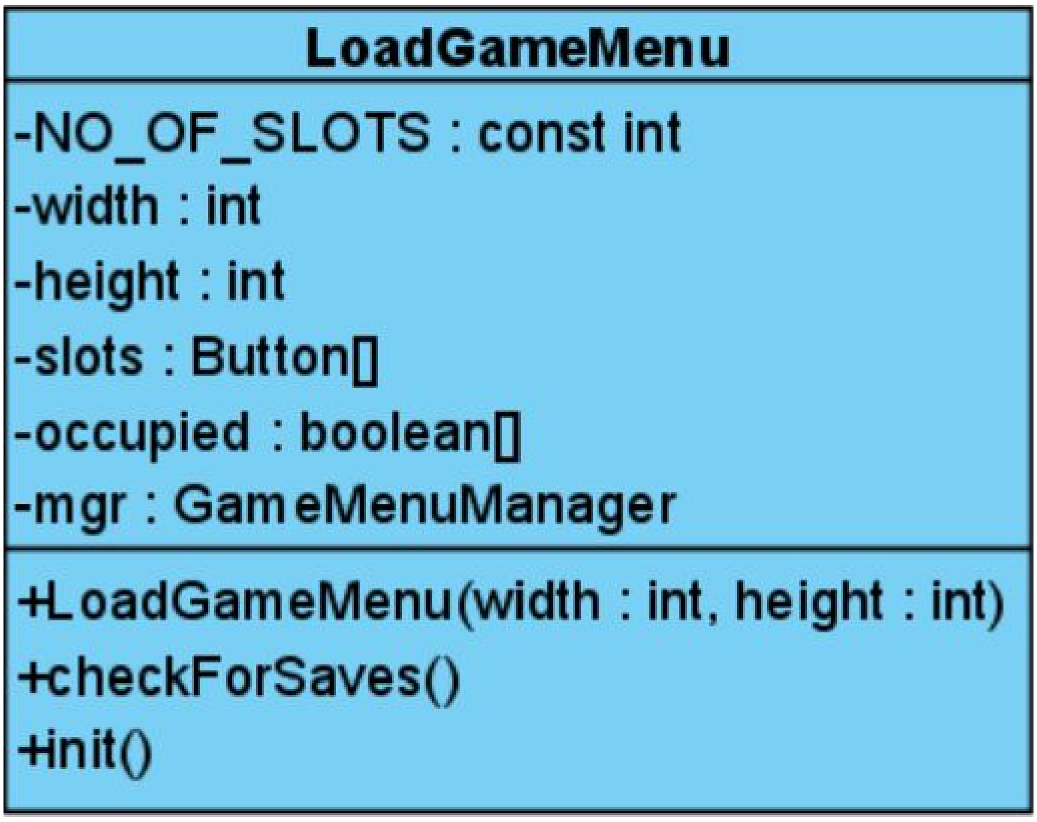


Figure 7. LoadGameMenu Class

Attributes:

(Some JavaFX attributes eg. Buttons are not included for brevity.)
-NO_OF_SLOTS: This constant int indicates the number of total save slots.
-slots: This array of JavaFX Button objects holds a button for each save slot.
-backButton: This JavaFX Button object is used to go back to the Main Menu.
-mgr: Holds the GameMenuManager instance.
-height,width: Holds the height and width of the scene.
-occupied: This boolean array indicates whether a specified save slot is occupied.

Operations:

+checkForSaves(): This method checks for any available saves.
+init(boolean maximized): Initializes the scene.

SettingsMenu

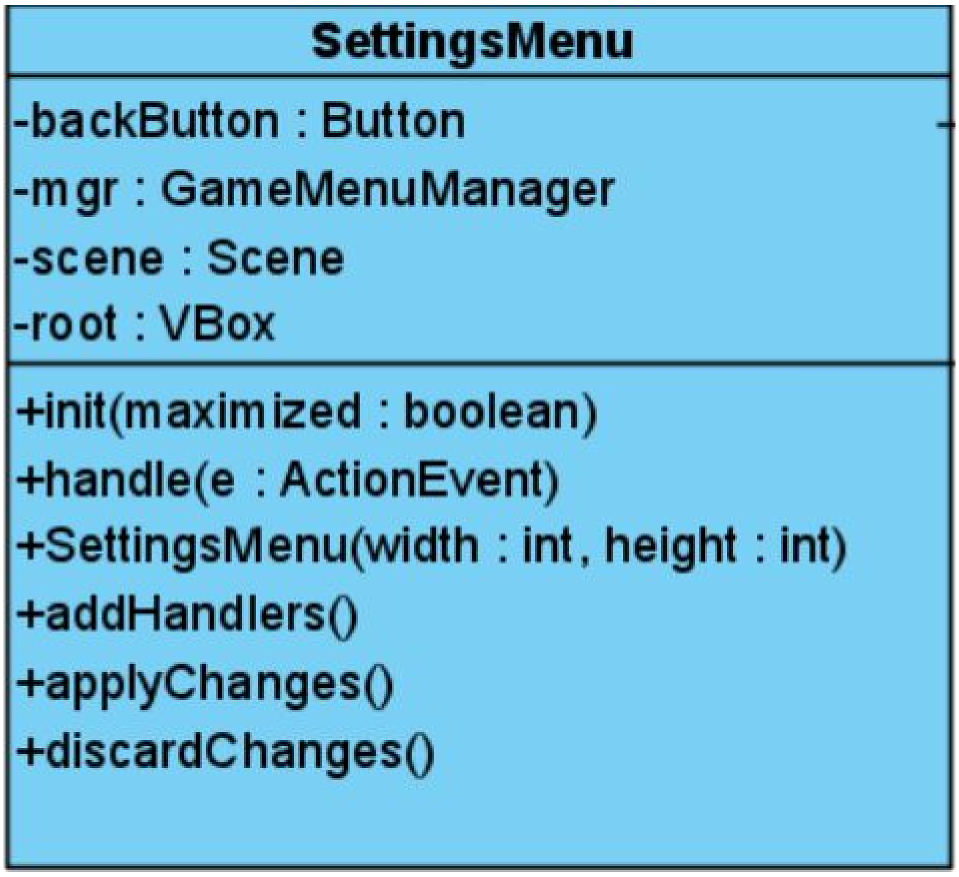


Figure 8. SettingsMenu Class

Attributes:

(Some JavaFX attributes eg. Buttons are not included for brevity.)
-backButton: This JavaFX Button is used to go back to the previous menu.
-scene: Holds the scene for the menu.
-root: The root container for all the components in the menu.
-mgr: The GameMenuManager instance.

Operations:

+init(): Initializes the SettingsMenu.
+handle(e: ActionEvent): Handles all the action events in the settings menu.
+addHandlers(): Adds the handlers for the interactive components of the menu.
+applyChanges(): Applies the setting changes.
+discardChanges(): Discards the setting changes and loads the current settings.

HelpMenu

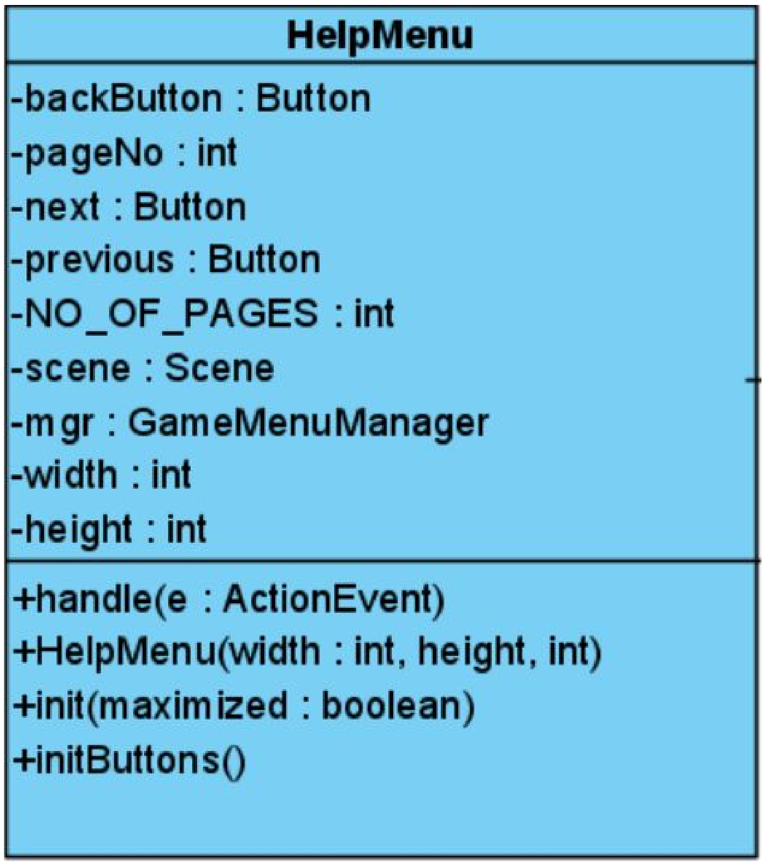


Figure 9. HelpMenu Class

Attributes:

(Some JavaFX attributes eg. Buttons are not included for brevity.)
-backButton: The button to go back to the main menu.
-pageNo: The current page number.
-next: The button to go to the next help page.
-previous: The button to go back to the previous help page.
-NO_OF_PAGES: Holds how many help pages in total there are.
-scene: The scene for the menu.
-mgr: Holds the GameMenuManager instance.
-width,height: Width and height for the scene.

Operations:

+handle(e: ActionEvent): Handles all the button events for the help menu.
+init(maximized: boolean): Initializes the help menu.
+initButtons(): Initializes the buttons for the help menu.

CreditsMenu

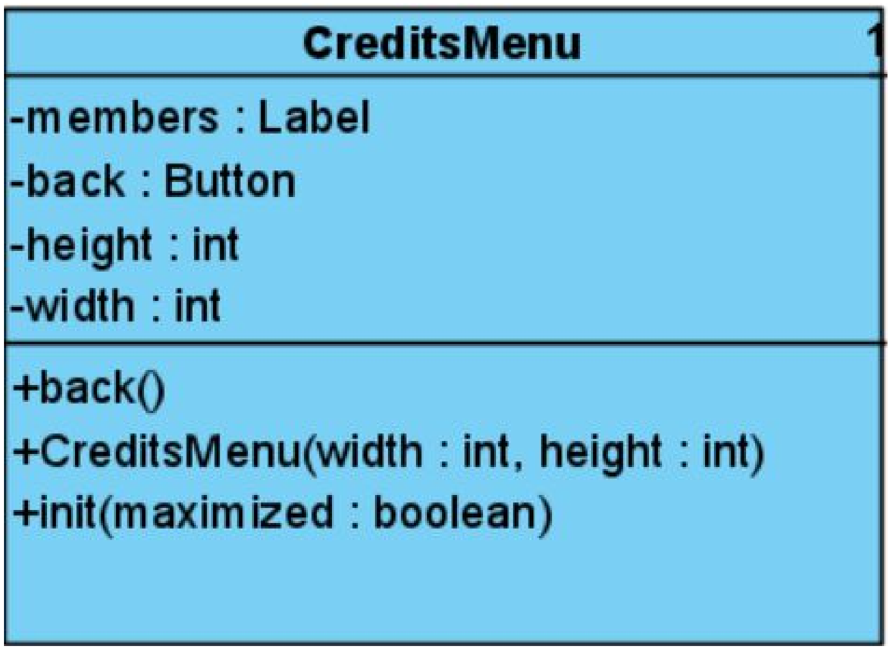


Figure 10. CreditsMenu Class

Attributes:

(Some JavaFX attributes eg. Buttons are not included for brevity.)
-members: This label is used to display the names of the team members.
-width,height: Holds the width and height for the scene.
-back: This button is used to go back to the main menu.

Operations:

+back(): Goes back to the previous menu.
+init(maximized: boolean): Initializes the CreditsMenu.

PauseMenu

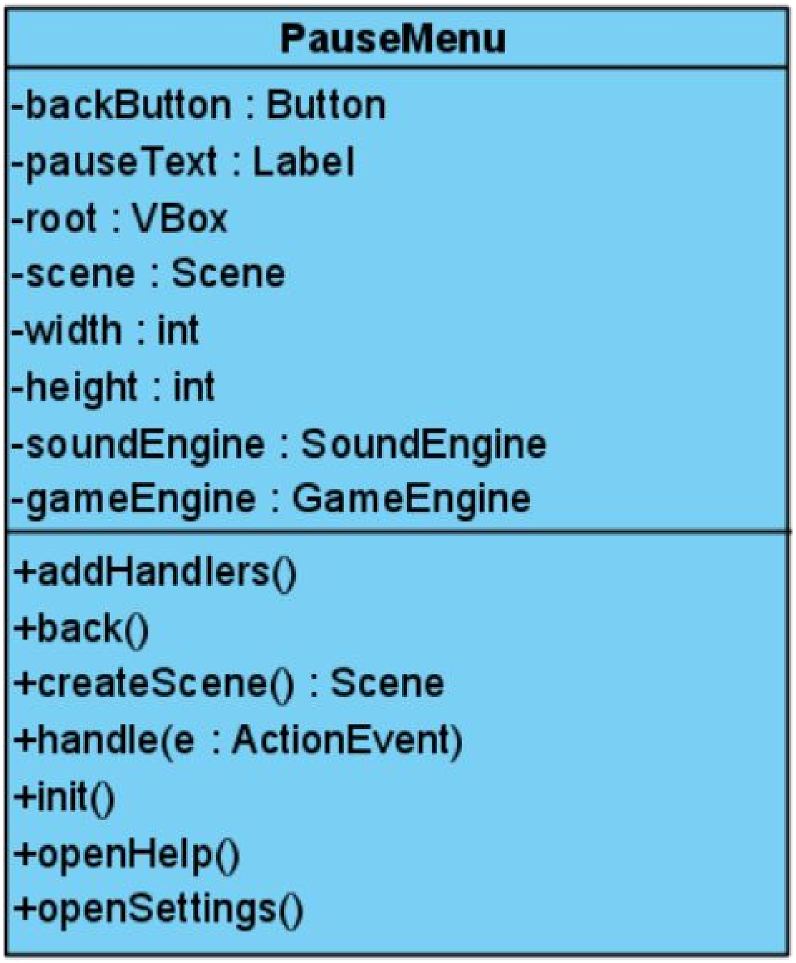


Figure 11. PauseMenu Class

Attributes:

(Some JavaFX attributes eg. Buttons are not included for brevity.)
-backButton: Used to go back to the game.
-pauseText: Used to display "Paused".
-width,height: Holds the width and height of the scene.
-root: The root container for all the JavaFX components.
-soundEngine: Holds the SoundEngine instance.
-gameEngine: Holds the GameEngine instance.

Operations:

+back(): Goes back to the game.
+openHelp(): Opens the PauseHelpMenu.
+openSetting(): Opens the PauseSettingsMenu.
+createScene(): Creates and returns the scene for the menu.
+init(): Initializes the menu.

+addHandler(): Adds action handlers to the interactive components in the menu.

PauseSettingsMenu

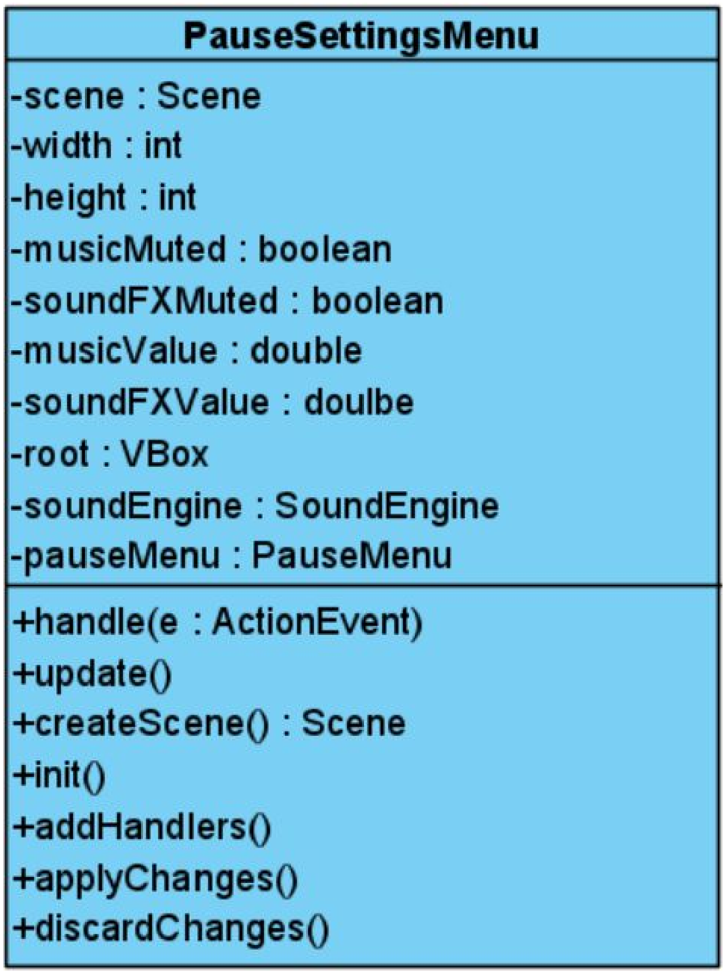


Figure 12. PauseSettingsMenu Class

Attributes:

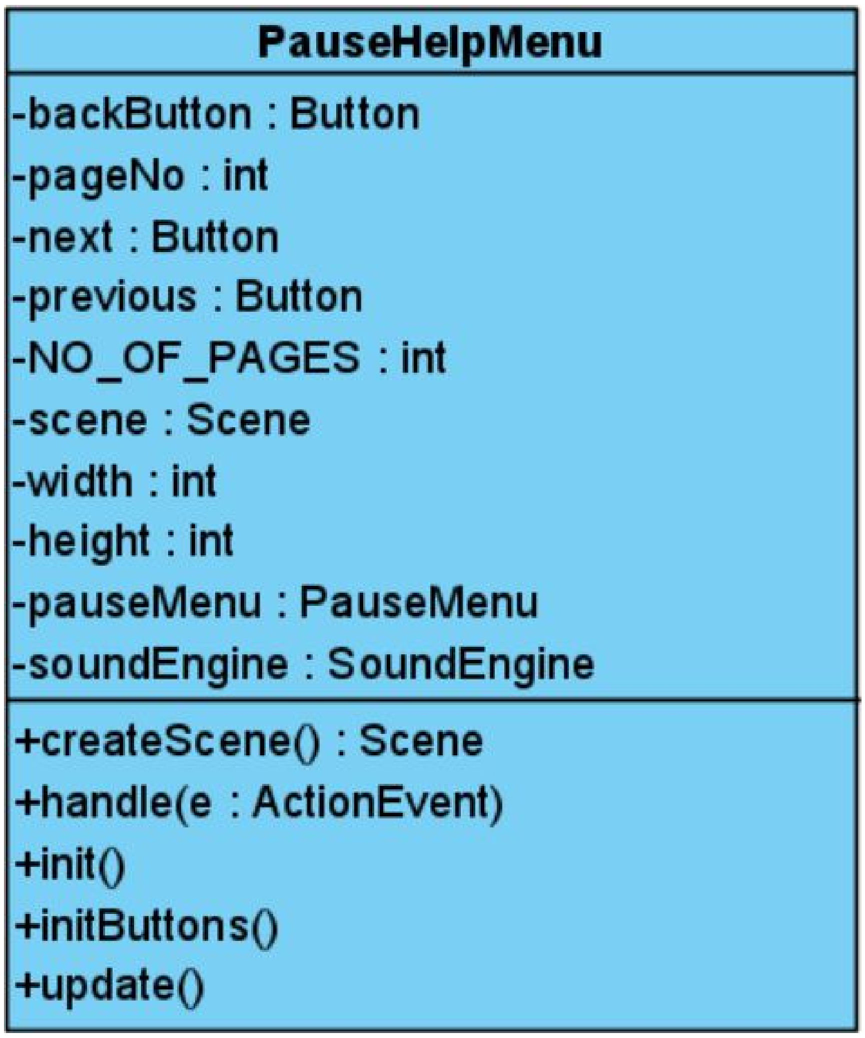
(Some JavaFX attributes eg. Buttons are not included for brevity.)
-scene: Holds the scene for the menu.
-width,height: Holds the width and height of the scene.
-musicMuted, soundFXMuted: Indicates whether the music/soundFX is muted.
-musicValue, soundFXValue: Indicates the volume for the music/soundFX.
-root: The root container for all the JavaFX components.
-soundEngine: Holds the SoundEngine instance.
-pauseMenu: Holds the PauseMenu instance.

Operations:

+handle(e: ActionEvent): Handles action events.
+update(): Updates the scene.

+createScene(): Creates and returns the scene for the menu.
+init(): Initializes the menu.
+addHandler(): Adds action handlers to the interactive components in the menu.
+applyChanges(): Applies the setting changes.
+discardChanges(): Discards the setting changes and loads the current settings.

PauseHelpMenu


PauseHelpMenu

-backButton : Button
-pageNo : int
-next : Button
-previous : Button
-NO_OF_PAGES : int
-scene : Scene
-width : int
-height : int
-pauseMenu : PauseMenu
-soundEngine : SoundEngine

+createScene() : Scene
+handle(e : ActionEvent)
+init()
+initButtons()
+update()

Figure 13. PauseHelpMenu Class

Attributes:

(Some JavaFX attributes eg. Buttons are not included for brevity.)
-backButton: The button to go back to the main menu.
-pageNo: The current page number.
-next: The button to go to the next help page.
-previous: The button to go back to the previous help page.
-NO_OF_PAGES: Holds how many help pages in total there are.

-scene: The scene for the menu.
-mgr: Holds the GameMenuManager instance.
-width,height: Width and height for the scene.
-soundEngine: Holds the SoundEngine instance.
-pauseMenu: Holds the PauseMenu instance.

Operations:

+handle(e: ActionEvent): Handles all the button events for the help menu.
+init(maximized: boolean): Initializes the help menu.
+initButtons(): Initializes the buttons for the help menu.
+update(): Updates the scene.
+createScene(): Creates and returns the scene.

GameOverScene

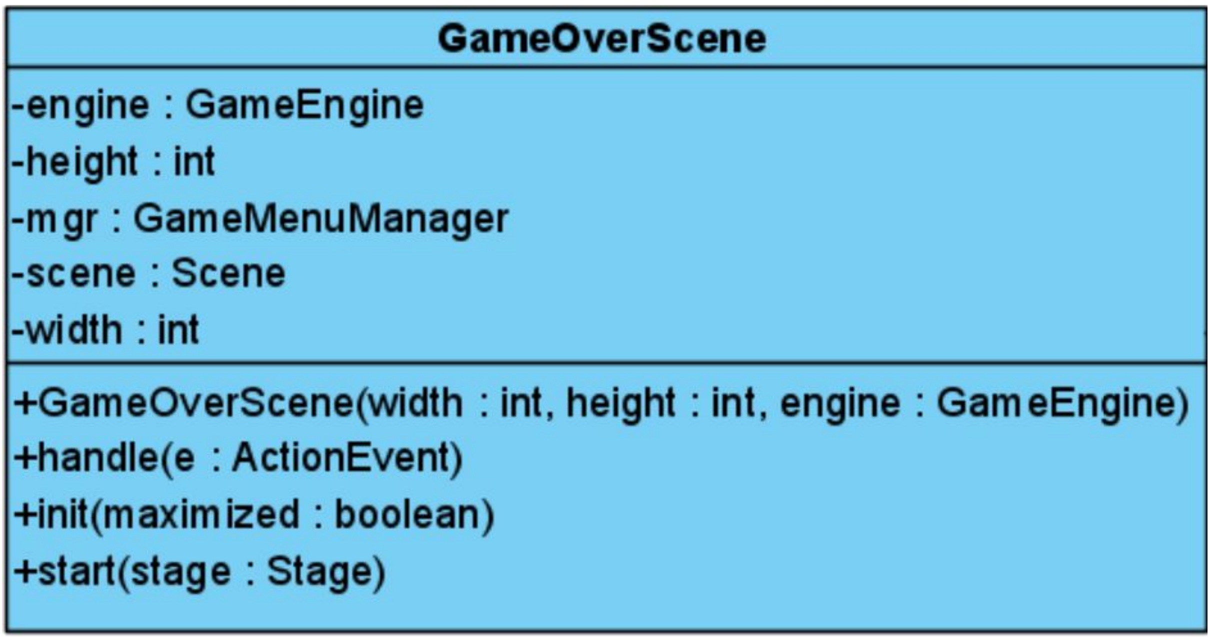


Figure 14. GameOverScene Class

Attributes:

(Some JavaFX attributes eg. Buttons are not included for brevity.)
-engine: Holds the GameEngine instance.
-width,height: Holds the width and height for the scene.
-scene: Holds the scene.
-mgr: Holds the GameMenuManager instance.

Operations:

+handle(e: ActionEvent): Handles all the button events.
+init(maximized: boolean): Initializes the Game Over screen.
+start(stage: Stage): Starts the game over scene stage.

GameMenuManager

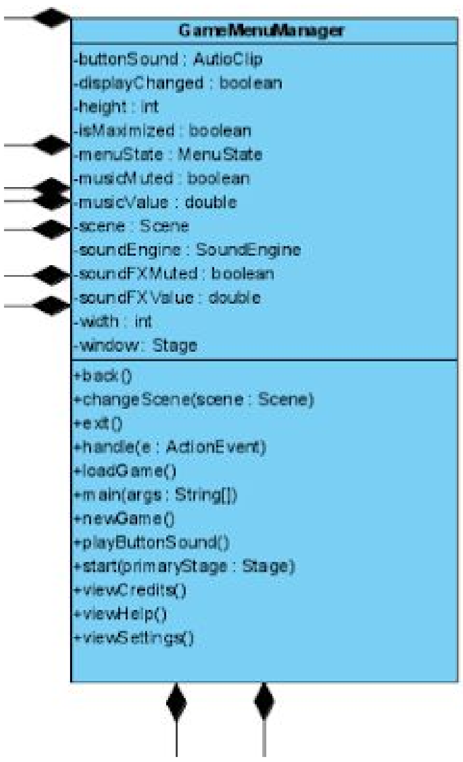


Figure 15. GameMenuManager Class

Attributes:

-buttonSound: Holds an audio clip for the button sounds.
-displayChanged: This attribute is a boolean to check whether the display mode is changed or not.
-height: This attribute is an integer that holds the height of the screen.
-isMaximized: This attribute is a boolean to check whether the screen is maximized or not.
-menuState: This attribute holds a menuState to keep track of the current state.
-musicMuted: This attribute is a boolean that checks whether the music is muted or not.
-musicValue: This attribute is a double that holds the value of the music.
-scene: This attribute holds the scene.
-soundEngine: This attribute is an instance of the SoundEngine class which will hold and handle operations about the sound effects and music.
-soundFXMuted: This attribute checks whether the sound effects are muted or not.
-soundFXValue: This attribute holds the value of the sound effects.

-width: This attribute is an integer that holds the width of the screen.
-window: This attribute is a stage to show the window.

Operations:

+back(): Returns to the previous screen.
+changeScreen(scene: Scene): Changes the current scene.
+exit(): Exits from the game.
+handle(e: ActionEvent): An event handler for the game menu.
+loadGame(): Loads a game.
+newGame(): Starts a new game.
+playButton(): Plays the audio clip of a button.
+start(primaryStage: Stage): Starts the stage.
+viewCredits(): Shows the credits screen.
+viewHelp(): Shows the help screen.
+viewSettings(): Shows the settings menu screen.

SoundEngine

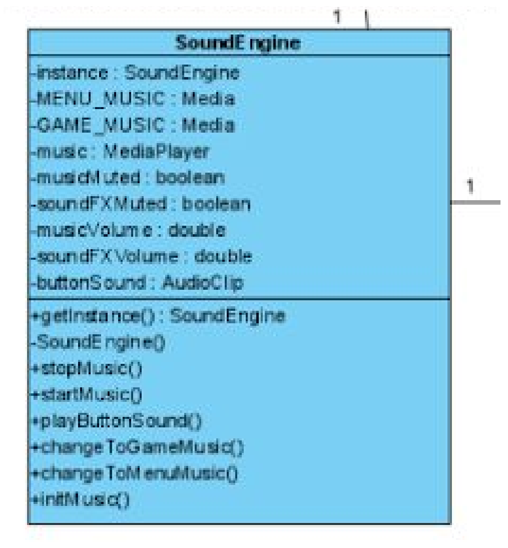


Figure 16. SoundEngine Class

Attributes:

-instance: Holds the instance of the sound engine.
-MENU_MUSIC: Holds the musşc of the menu.
-GAME_MENU: Holds the music of the game.
-music: Holds a MediaPlayer to play the music.
-soundFXMuted: A boolean that checks if the sound effects are muted or not.
-soundFXVolume: Holds a double which represents the volume of the sound effects.
-musicMuted: This attribute is a boolean that checks whether the music is muted or not.
-musicVolume: Holds a double which represents the volume of the music of the game.

-buttonSound: Holds an AudioClip for the button sounds.

Operations:

+SoundEngine(): Calls the SoundEngine
+stopMusic(): Stops the music.
+startMusic(): Starts the music.
+playButtonSound(): Plays the button sound.
+changeToGameMusic(): Changes the current music to the game music.
+changeToMenuMusic(): Changes the current music to the menu music.
+initMusic(): Initializes the music.

GameEngine

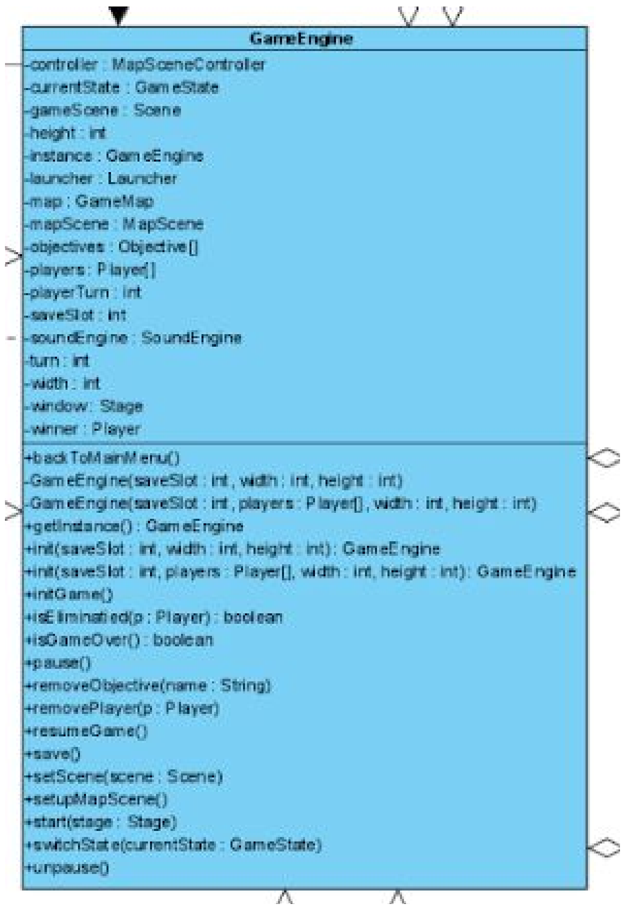

GameEngine
-controller : MapSceneController
-currentState : GameState
-gameScene : Scene
-height : int
-instance : GameEngine
-launcher : Launcher
-map : GameMap
-mapScene : MapScene
-objectives : Objective[]
-players : Player[]
-playerTurn : int
-saveSlot : int
-soundEngine : SoundEngine
-turn : int
-width : int
-window : Stage
-winner : Player
+backToMainMenu()
-GameEngine(saveSlot : int, width : int, height : int)
-GameEngine(saveSlot : int, players : Player[], width : int, height : int)
+getInstance() : GameEngine
+init(saveSlot : int, width : int, height : int) : GameEngine
+init(saveSlot : int, players : Player[], width : int, height : int) : GameEngine
+initGame()
+isEliminated(p : Player) : boolean
+isGameOver() : boolean
+pause()
+removeObjective(name : String)
+removePlayer(p : Player)
+resumeGame()
+save()
+setScene(scene : Scene)
+setupMapScene()
+start(stage : Stage)
+switchState(currentState : GameState)
+unpause()

Figure 17. GameEngine Class

Attributes:

-controller: Holds the MapSceneController
-currentsState: Holds the active state of the game.
-height: Holds the height of the scene

-instance: The instance of the GameEngine Class
-launcher: An instance of the launcher.
-map: An instance of the game map.
-mapScene: An instance of the map scene.
-objectives: An array which includes the objectives in a game.
-players: An array which includes the players.
-playerTurn: An integer that holds the index of the current player.
-saveSlot: Holds the number of the current save slot.
-soundEngine: An instance of the soundEngine
-width: Holds the width of the scene.
-turn: Holds the number of turns in the game.
-window: An instance of the game stage.
-winner: Holds the winner of the game.

Operations:

+backToMainMenu(): Returns to main menu.
+Game(saveSlot: int, width: int, height: int): A constructor for the Game class with a saveSlot parameter which indicates the slot that the game will be saved into and height,width parameters to indicate the size of the window.
+Game(saveSlot: int,players: Player[], width: int, height: int): A constructor for the Game class with a saveSlot parameter which indicates the slot that the game will be saved into, players parameter which will hold the players in the game and and height,width parameters to indicate the size of the window.
+init(saveSlot: int, width: int, height: int): Initializes the game engine and returns it.
+init(saveSlot: int, players: Player[], width: int, height: int): Initializes the game engine and returns it.
+isEliminated(p: Player): Checks if a player is eliminated or not.
+isGameOver(): Checks if the game is over or not.
+pause(): Pauses the Game.
+removeObjective(name: String): Removes a completed objective from the game.
+resumeGame(): Resumes the Game.
+save(): Saves the game when users exit from the game.
+setScene(scene: Scene): Sets the preferred scene to the window.
+setupMapScene(): Prepares the map scene.
+start(stage: Stage): Starts the stage.
+switchState(currentState: GameState): Changes the active state.
+unpause(): Unpauses the game.

MapSceneController

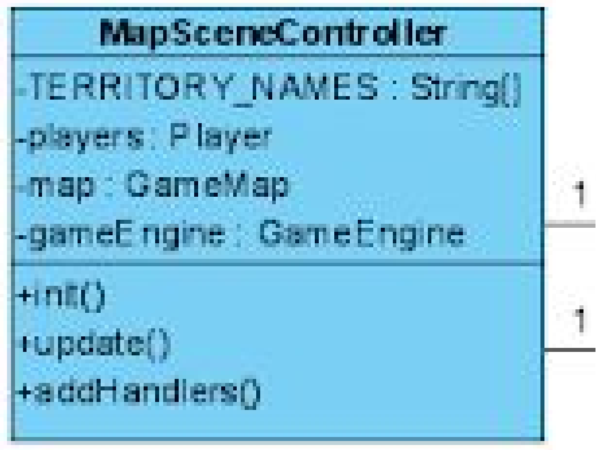


Figure 18. MapSceneController Class

Attributes:

-TERRITORY_NAMES: An array that holds the list of the territories.
-players: An ArrayList that holds the Players.
-map: An instance of the game map.
-gameEngine: An instance of the game engine.

Operations:

+init(): Initialize the controller.
+update(): Updates the controller.
+addHandlers(): Adds handlers to the territories.

GameState

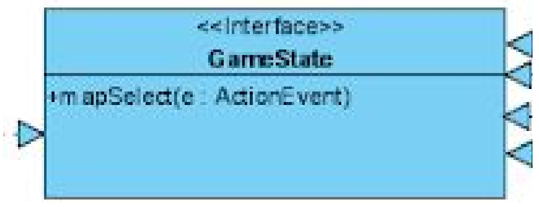


Figure 19. GameState Interface

Operations:

+mapSelect(e: ActionEvent): A method that will handle the ActionEvent according to the current state of the game.

Launcher

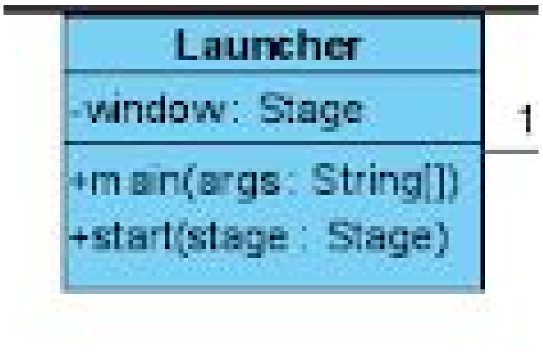


Figure 20. Launcher Class

Attributes:

-window: The stage of the game.

Operations:

+main(args: String[]): Main method to launch the game.
+start(start: Stage): Starts the game.

MapScene

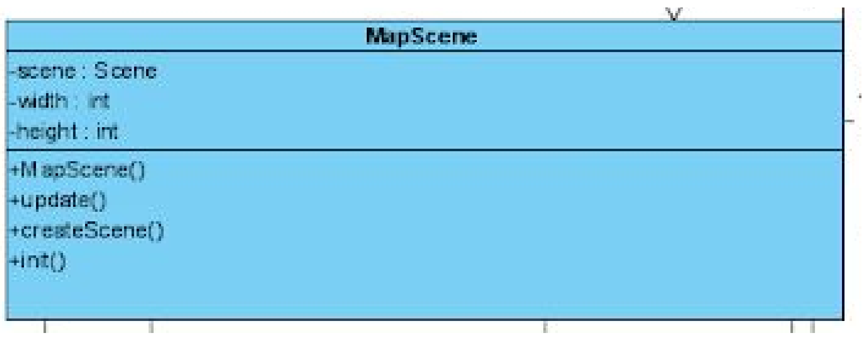


Figure 21. MapScene Class

Attributes:

-scene: Holds the scene.
-width: An integer that holds the width of the scene.
-height: An integer that holds the height of the scene.

Operations:

+MapScene(): Constructor for the map scene.
+update(): Updates the map scene
+createScene(): Creates the map scene.

+init(): Initialize the map scene.

InitialArmyPlacementState

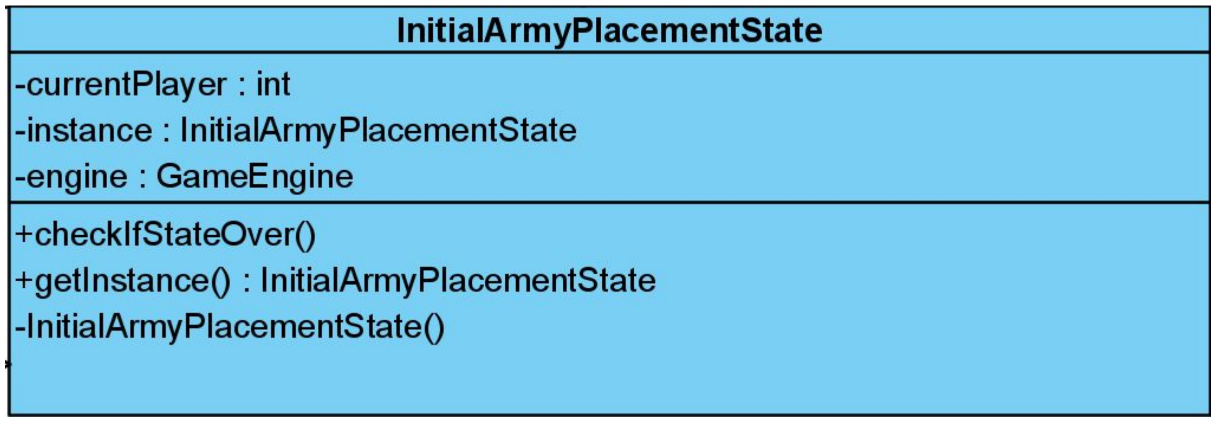

InitialArmyPlacementState
-currentPlayer : int
-instance : InitialArmyPlacementState
-engine : GameEngine
+checkIfStateOver()
+getInstance() : InitialArmyPlacementState
-InitialArmyPlacementState()

Figure 22. InitialArmyPlacementState Class

Implements the GameState interface.

Attributes:

-currentPlayer: An integer tag for the current player.

-instance: The instance of the InitialArmyPlacementState class.

-engine: The instance of GameEngine class.

Operations:

+checkIfStateOver(): Checks if the state is over.

+getInstance(): Returns the instance of InitialArmyPlacementState.

-InitialArmyPlacementState(): Constructor for InitialArmyPlacementState.

ArmyPlacementState

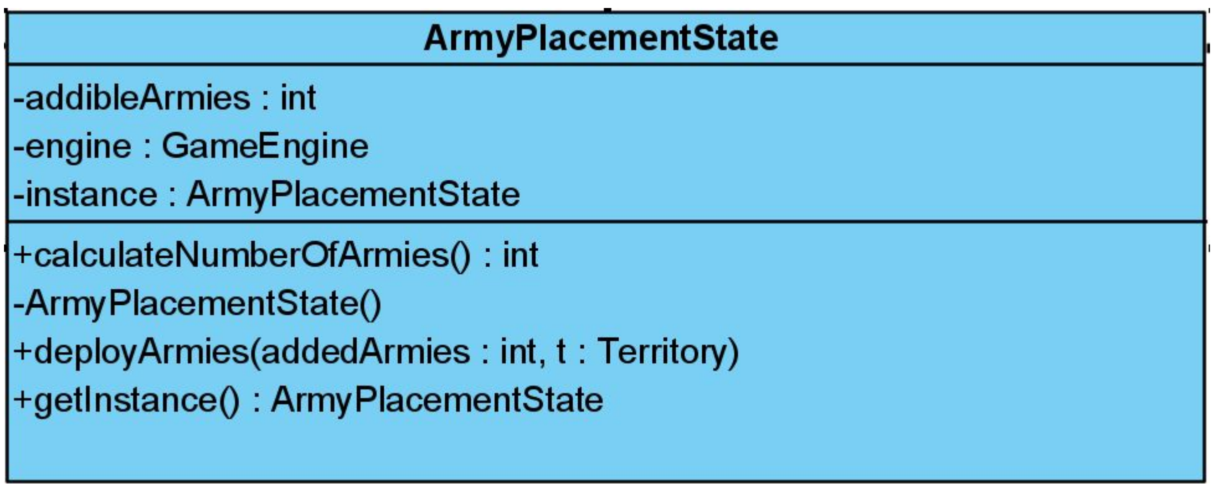

ArmyPlacementState
-addibleArmies : int
-engine : GameEngine
-instance : ArmyPlacementState
+calculateNumberOfArmies() : int
-ArmyPlacementState()
+deployArmies(addedArmies : int, t : Territory)
+getInstance() : ArmyPlacementState

Figure 23. ArmyPlacementState Class

Implements the GameState interface.

Attributes:

-addibleArmies: An integer that holds the maximum possible number of armies that can be added to a territory.
-engine: The instance of GameEngine class.
-instance: The instance of the ArmyPlacementState.

Operations:

+calculateNumberOfArmies(): Calculates the maximum amount of armies that can be deployed.
-ArmyPlacementState(): Constructor for the ArmyPlacementState.
+deployArmies(armiesToAdd: int, t : Territory): Enables users to deploy a given number of armies to a given territory.
+getInstance(): Returns the instance of ArmyPlacementState.

AttackingState

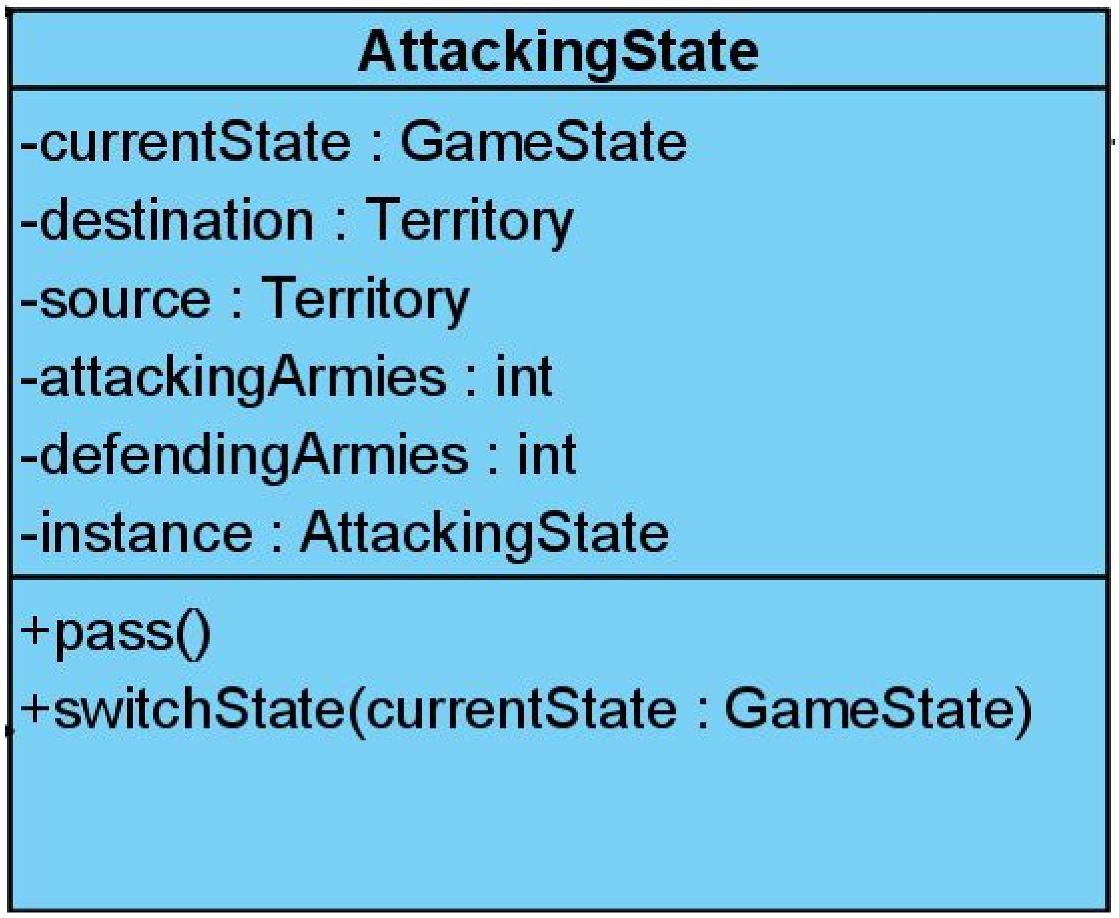


Figure 24. AttackingState Class

Implements the GameState interface.

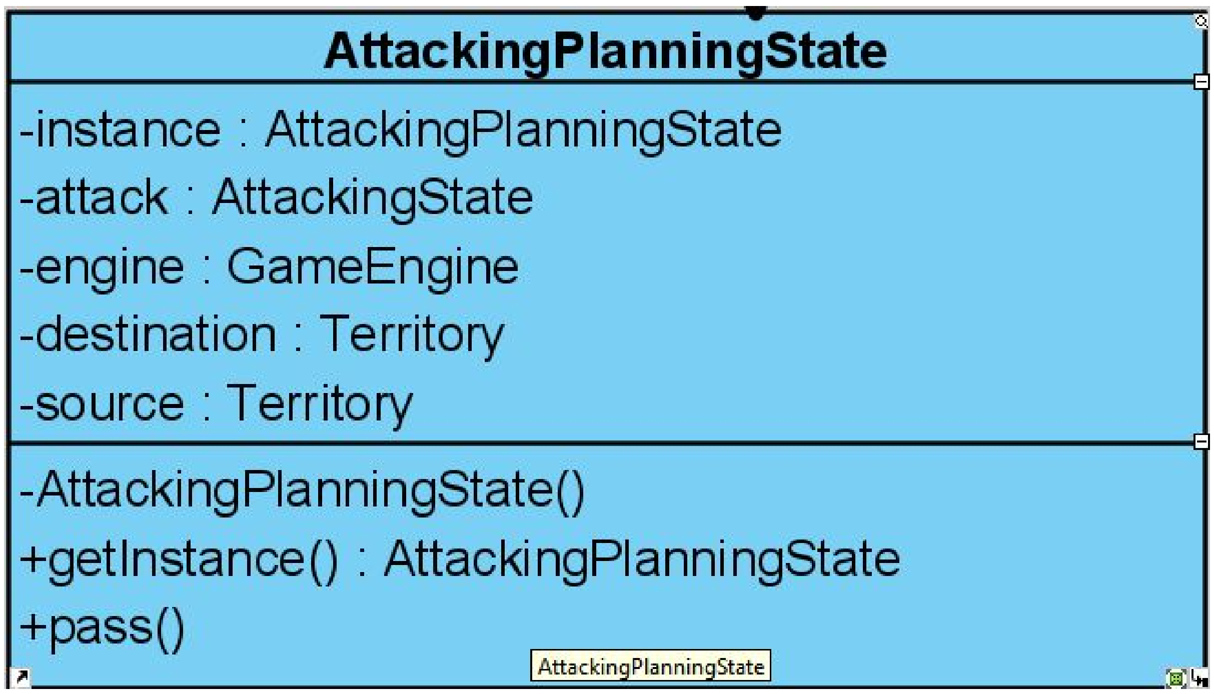
Attributes:

-currentState: Holds the current state.
-destination: Holds the tag of the territory selected to be attacked.
-source: Holds the tag of the territory where the attack will be performed from.
-attackingArmies: Holds the number of the armies that will be used to attack.
-defendingArmies: Holds the number of the armies that will be used to defend.
-instance: The instance of AttackingState.

Operations:

+pass(): Passes to the next state.
+switchState(currentState : GameState): Switches state to a different one.

AttackingPlanningState


AttackingPlanningState

-instance : AttackingPlanningState
-attack : AttackingState
-engine : GameEngine
-destination : Territory
-source : Territory

-AttackingPlanningState()
+getInstance() : AttackingPlanningState
+pass()

Figure 25. AttackingPlanningStateClass

Implements the GameState interface.

Attributes:

-instance: The instance of the AttackingPlanningState.
-attack: The instance of the AttackingState.
-engine: The instance of the GameEngine.
-destination: Holds the tag of the territory selected to be attacked.
-source: Holds the tag of the territory where the attack will be performed from.

Operations:

-AttackingPlanningState(): Constructor for the AttackingPlanningState.
+getInstance(): Returns the instance of AttackingPlanningState.
+pass(): Passes to the next state.

DiceSelectionState

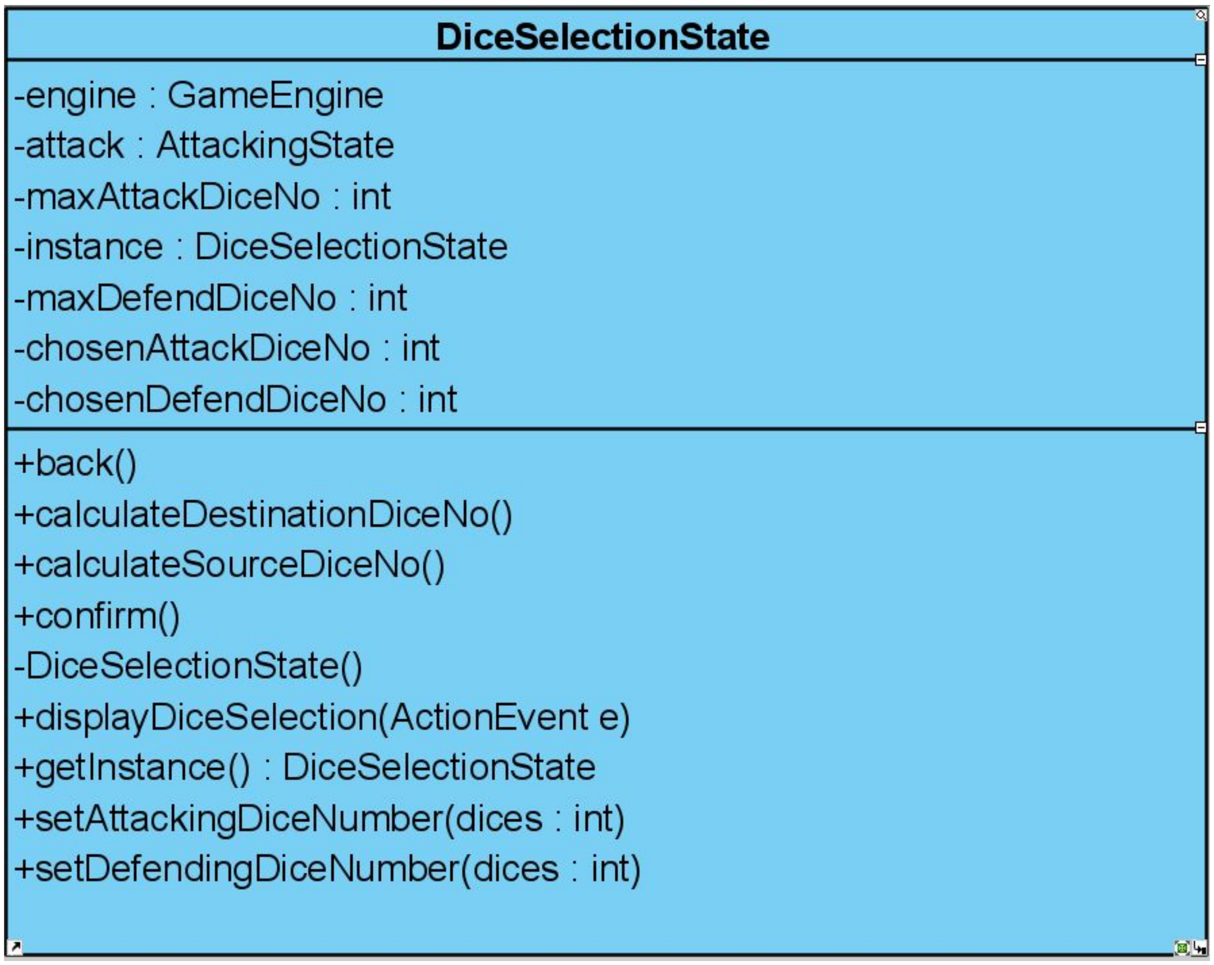

DiceSelectionState
-engine : GameEngine
-attack : AttackingState
-maxAttackDiceNo : int
-instance : DiceSelectionState
-maxDefendDiceNo : int
-chosenAttackDiceNo : int
-chosenDefendDiceNo : int
+back()
+calculateDestinationDiceNo()
+calculateSourceDiceNo()
+confirm()
-DiceSelectionState()
+displayDiceSelection(ActionEvent e)
+getInstance() : DiceSelectionState
+setAttackingDiceNumber(dices : int)
+setDefendingDiceNumber(dices : int)

Figure 26. DiceSelectionState Class

Implements the GameState interface.

Attributes:

-engine: The instance of GameEngine.
-attack: The instance of AttackingState.
-maxAttackDiceNo: An integer that holds the maximum amount of dice that can be rolled to attack.
-maxDefendDiceNo: An integer that holds the maximum amount of dice that can be rolled to defend.
-instance: The instance of DiceSelectionState.
-chosenAttackDiceNo: An integer that holds the amount of dice chosen to roll for attacking.
-chosenDefendDiceNo: An integer that holds the amount of dice chosen to roll for defending.

Operations:

+back(): Closes the pop-up dice selection screen.
+calculateDestinationDiceNo(): Calculates the maximum amount of dice that can be rolled to defend.
+calculateSourceDiceNo(): Calculates the maximum amount of dice that can be rolled to attack.
+confirm(): Confirms the selected amount of dice.
-DiceSelectionState(): Constructor for DiceSelectionState.
+displayDiceSelection(e : ActionEvent): Pops up a dice selection menu.
+getInstance(): Returns the instance of DiceSelectionState.
+setAttackingDiceNumber(dices : int): Sets the amount of dice chosen to attack.
+setDefendingDiceNumber(dices : int): Sets the amount of dice chosen to defend.

WarState

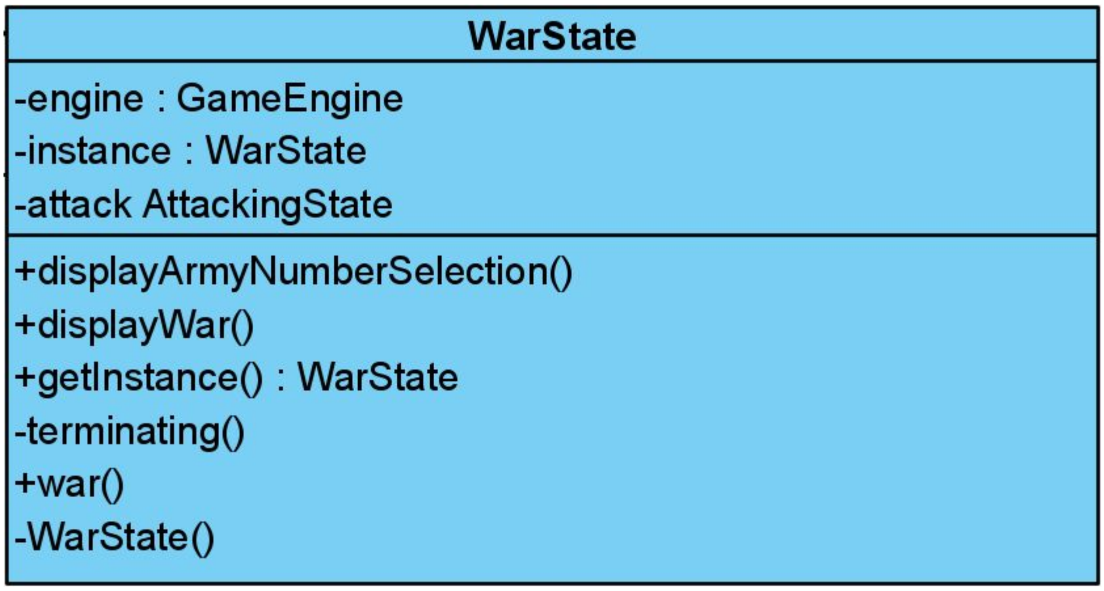


Figure 27. WarState Class

Implements the GameState interface.

Attributes:

-engine: The instance of GameEngine.
-instance: The instance of WarState.
-attack: The instance of AttackingState

Operations:

+displayArmyNumberSelection(): Displays a pop-up that lets the player select the number of armies they want to move when they take over a territory

+displayWar(): Displays the war sequence.
+getInstance(): Returns the instance of DiceSelectionState.
-terminating(): Checks if the war is over. If the war is over, it switches the state to AttackingPlanningState. If not, it switches the state to DiceSelection state and initiates the dice selections sequence again.
+war(): Calculates the result of the war.
-WarState(): Constructor for WarState.

FortifyingState

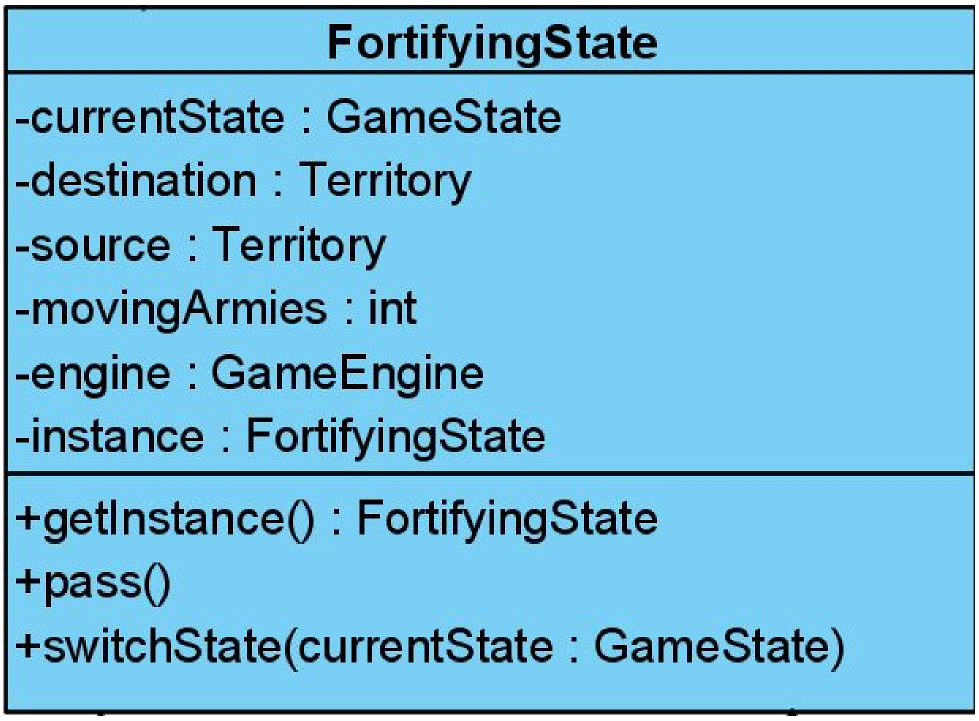

FortifyingState
-currentState : GameState
-destination : Territory
-source : Territory
-movingArmies : int
-engine : GameEngine
-instance : FortifyingState
+getInstance() : FortifyingState
+pass()
+switchState(currentState : GameState)

Figure 28. FortifyingState Class

Implements the GameState interface.

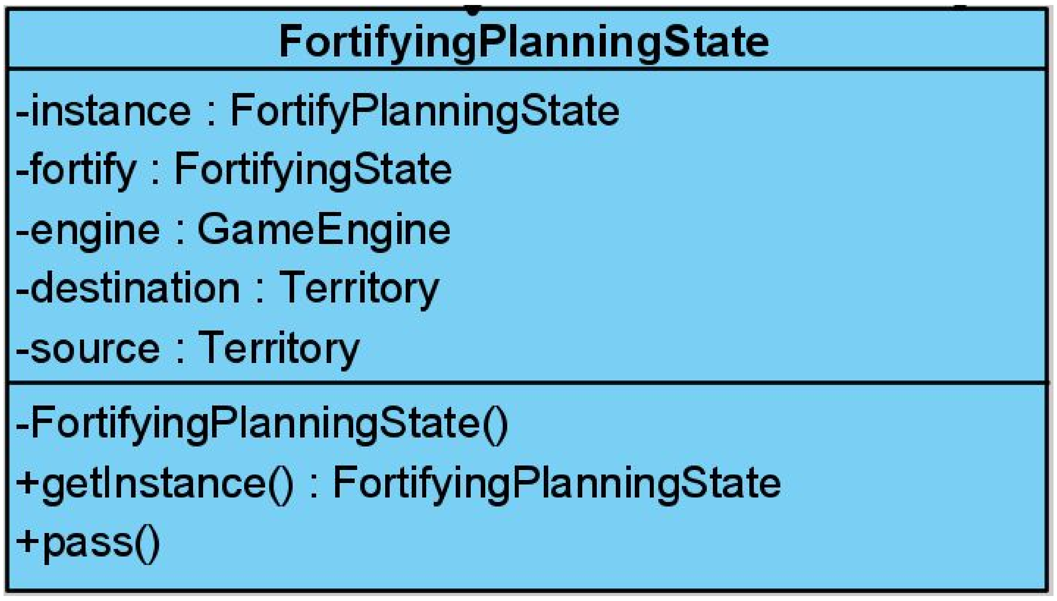
Attributes:

-currentState: Holds the current state of the game
-destination: Holds the territory where the armies will be moved to.
-source: Holds the territories where the armies will be moved from.
-movingArmies: Number of armies that will be moved.
-engine: The instance of GameEngine.
-instance: The instance of FortifyingState.

Operations:

+getInstance(): Returns the instance of FortifyingState.
+pass(): Passes to the next state.
+switchState(currentState: GameState): Switches the game state to the given state.

FortifyingPlanningState


FortifyingPlanningState

-instance : FortifyPlanningState
-fortify : FortifyingState
-engine : GameEngine
-destination : Territory
-source : Territory

-FortifyingPlanningState()
+getInstance() : FortifyingPlanningState
+pass()

Figure 29. FortifyingPlanningState Class

Implements the GameState interface.

Attributes:

-instance: The instance of FortifyPlanningState.
-fortify: The instance of FortifyingState.
-engine: The instance of GameEngine.
-destination: Holds the territory where the armies will be moved to.
-source: Holds the territories where the armies will be moved from.

Operations:

-FortifyPlanningState(): Constructor for FortifyPlanningState.
+getInstance(): Returns the instance of FortifyPlanningState.
+pass(): Passes to the next state.

FortifyingArmySelectionState

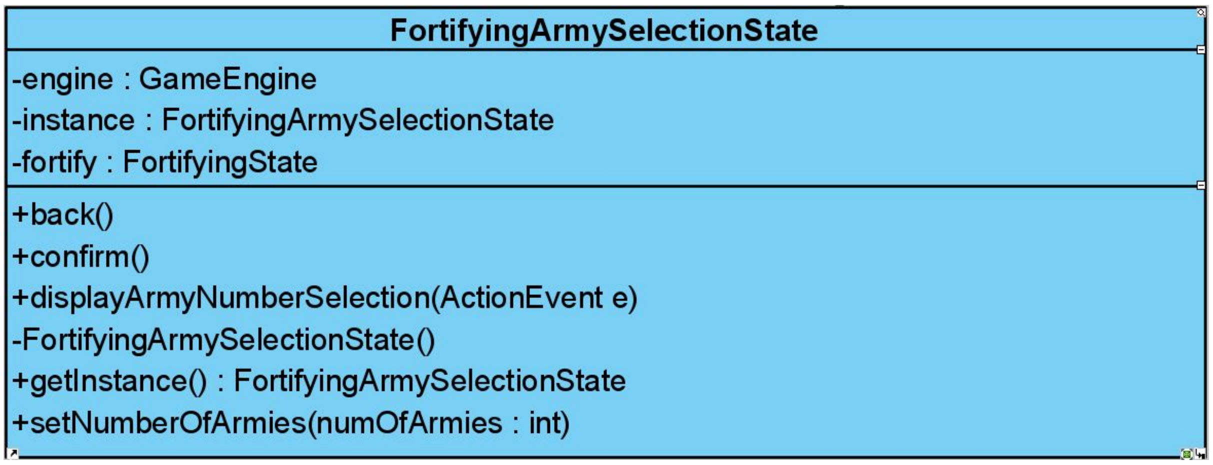

FortifyingArmySelectionState
-engine : GameEngine
-instance : FortifyingArmySelectionState
-fortify : FortifyingState
+back()
+confirm()
+displayArmyNumberSelection(ActionEvent e)
-FortifyingArmySelectionState()
+getInstance() : FortifyingArmySelectionState
+setNumberOfArmies(numOfArmies : int)

Figure 30. FortifyingArmySelectionState Class

Implements the GameState interface.

Attributes:

-engine: The instance of GameEngine.

-instance: The instance of FortifyPlanningState.

-fortify: The instance of FortifyingState.

Operations:

+back(): Closes the pop-up army number selection screen.

+confirm(): Confirms the selected amount of armies.

+displayArmyNumberSelection(e : ActionEvent): Pops up an army selection selection menu.

-FortifyingArmySelectionState(): Constructor for FortifyingArmySelectionState.

+getInstance(): Returns the instance of FortifyPlanningState.

+setNumberOfArmies(numOfArmies : int): Sets the amount of armies that will be moved.

Player

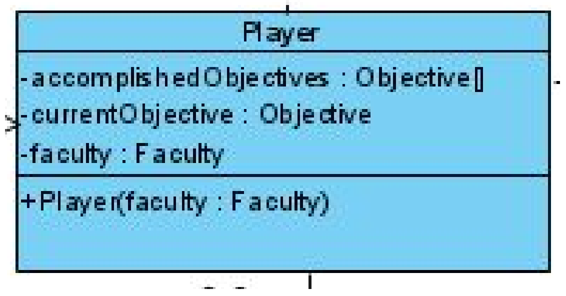

Player
-accomplishedObjectives : Objective[]
-currentObjective : Objective
-faculty : Faculty
+Player(faculty : Faculty)

Figure 31. Player Class

Attributes:

-accomplishedObjectives: This attribute holds the Objective array. It contains the finished objectives.

-currentObjective: This attribute holds the Objective object which is the current objective.

-faculty: This attribute holds the Faculty object.

Constructor:

+Player(faculty: Faculty): This constructor takes faculty and it initializes the player.

GameMap

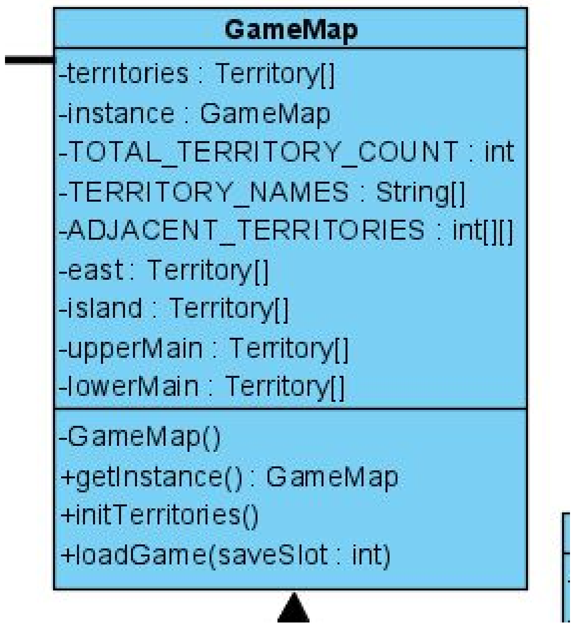


Figure 32. GameMap Class

Attributes:

-territories: This attribute holds the Territory array which will be placed on the map.

-instance: This attribute holds the GameMap.

-TOTAL_TERRITORY_COUNT: This attribute holds the total count of territories in the map as an integer value.

-TERRITORY_NAMES: This attribute holds the String array that contains the name of the territories.

-ADJACENT_TERRITORIES: This attribute holds the integer array that contains the count of adjacent territories.

-east: This attribute holds the Territory array for the east area.

-island: This attribute holds the Territory array for the island area.

-upperMain: This attribute holds the Territory array for the upperMain area.

-lowerMain: This attribute holds the Territory array for the lowerMain area.

Operations:

+getInstance(): GameMap: This method returns the GameMap.

+initTerritories(): This method initializes the territories on the map.

+loadGame(saveSlot: int): This method takes int value to determine which save slot will be loaded and it loads the game.

Territory

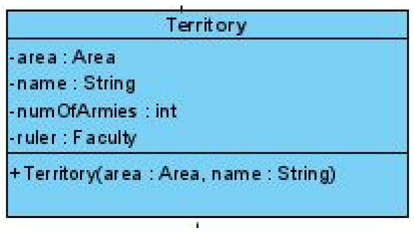


Figure 33. Territory Class

Attributes:

-area: This attribute holds the Area object.

-name: This attribute holds the name of the territory as a String.

-numOfArmies: This attribute holds the value of the armies in that territory as a int value.

-ruler: This attribute holds the Faculty object to determine the ruler.

Constructor:

+Territory(area: Area, name: String): This constructor takes the area object and name variable to specify the name of the territory and initialize.

Objective

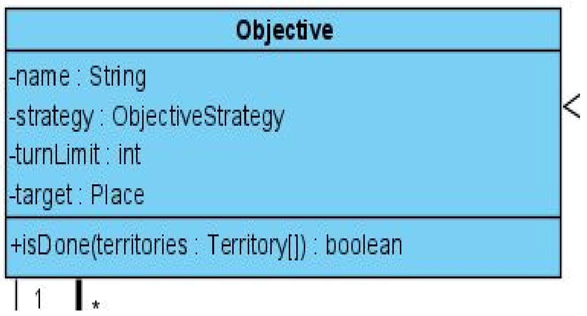

Objective
-name : String
-strategy : ObjectiveStrategy
-turnLimit : int
-target : Place
+isDone(territories : Territory[]) : boolean
1
*

Figure 34. Objective Class

Attributes:

-strategy: This attribute holds the ObjectiveStrategy.

-name: This attribute holds the name of the objective as a String.

-turnLimit: This attribute holds the integer value for calculating turn limit.

-target: This attribute holds the Place to determine the target.

Operations:

+isDone(): boolean: This method checks if the objective is done or not. It returns true if the objective is done and it returns false if the objective is not done.

Faculty

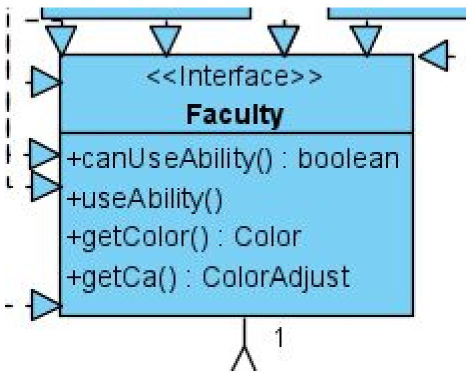


Figure 35. Faculty Interface

Operations:

+canUseAbility(): boolean: This method checks if the Faculty can use ability or not.

+useAbility(): This method uses the special ability.

Area

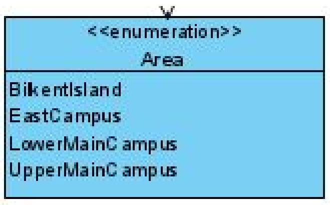


Figure 36. Area Enumeration

Risk101 contains BilkentIsland, EastCampus, LowerMainCampus, UpperMainCampus as an Area.

ObjectiveStrategy

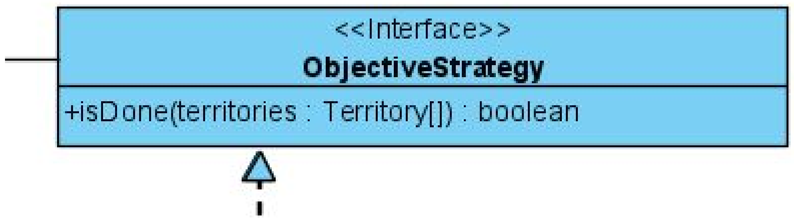


Figure 37. ObjectiveStrategy Interface

Operations:

+isDone(territories: Territory[]): boolean: This method is to determine whether the objective is done or not.

CaptureObjective

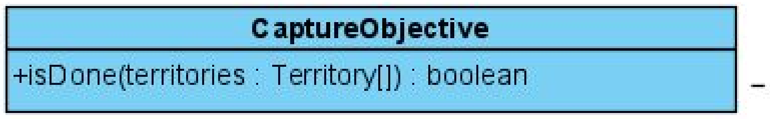


Figure 38. CaptureObjective Class

Operations:

+isDone(territories: Territory[]): boolean: This method is to determine whether the capture objective is done or not.

CaptureObjective

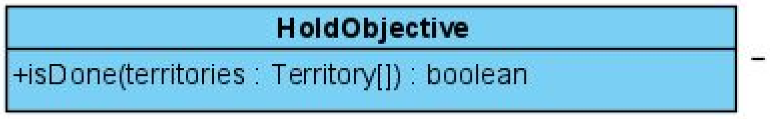


Figure 39. HoldObjective Class

Operations:

+isDone(territories: Territory[]): boolean: This method is to determine whether the hold objective is done or not.

TerritoryDecorator

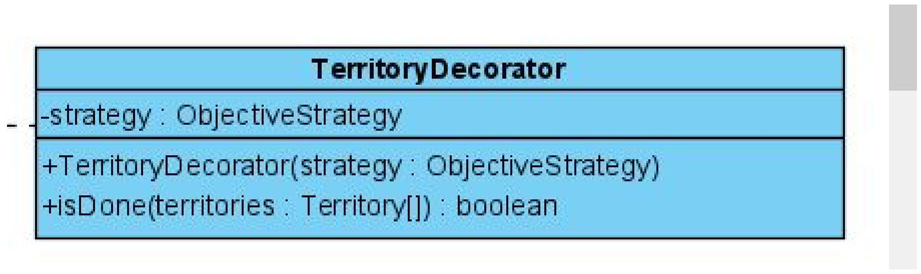

TerritoryDecorator
-strategy : ObjectiveStrategy
+TerritoryDecorator(strategy : ObjectiveStrategy)
+isDone(territories : Territory[]) : boolean

Figure 40. TerritoryDecorator Class

Attributes:

-strategy: This attribute holds the ObjectiveStrategy.

Constructor:

+Territory(strategy: ObjectiveStrategy): This constructor takes the ObjectiveStrategy and initializes.

Operations:

+isDone(territories: Territory[]): boolean: This method is to determine the hold objective is done or not.

AreaDecorator

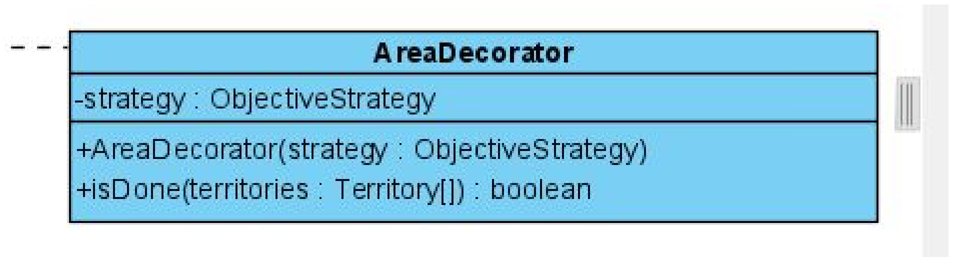

AreaDecorator
-strategy : ObjectiveStrategy
+AreaDecorator(strategy : ObjectiveStrategy)
+isDone(territories : Territory[]) : boolean

Figure 41. AreaDecorator Class

Attributes:

-strategy: This attribute holds the ObjectiveStrategy.

Constructor:

+Territory(strategy: ObjectiveStrategy): This constructor takes the ObjectiveStrategy and initializes.

Operations:

+isDone(territories: Territory[]): boolean: This method is to determine whether the hold objective is done or not.