CS 319 TERM PROJECT

Design Report

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 main Menu frames. First one is the Start Menu which can alert the management subsystem to start a new or a saved game through New Game and Saved Games frames or open the Credits frame. 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 frames. Access to these frames can also be done through the Start Menu. The user interface subsystem includes the Game End frame which can only pop up when the management system declares that the game is over. It also includes a Game Screen frame which will be updated by the model subsystem.

2.1.2 Management Subsystem

Management system is the controller of RISK101. The most significant component is the Game State Management 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 Management. 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 and reflect these updates to the Game Screen frame in the user interface subsystem. It also uses Game Manager Menu to access the Pause Menu frame. Sound Management 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.

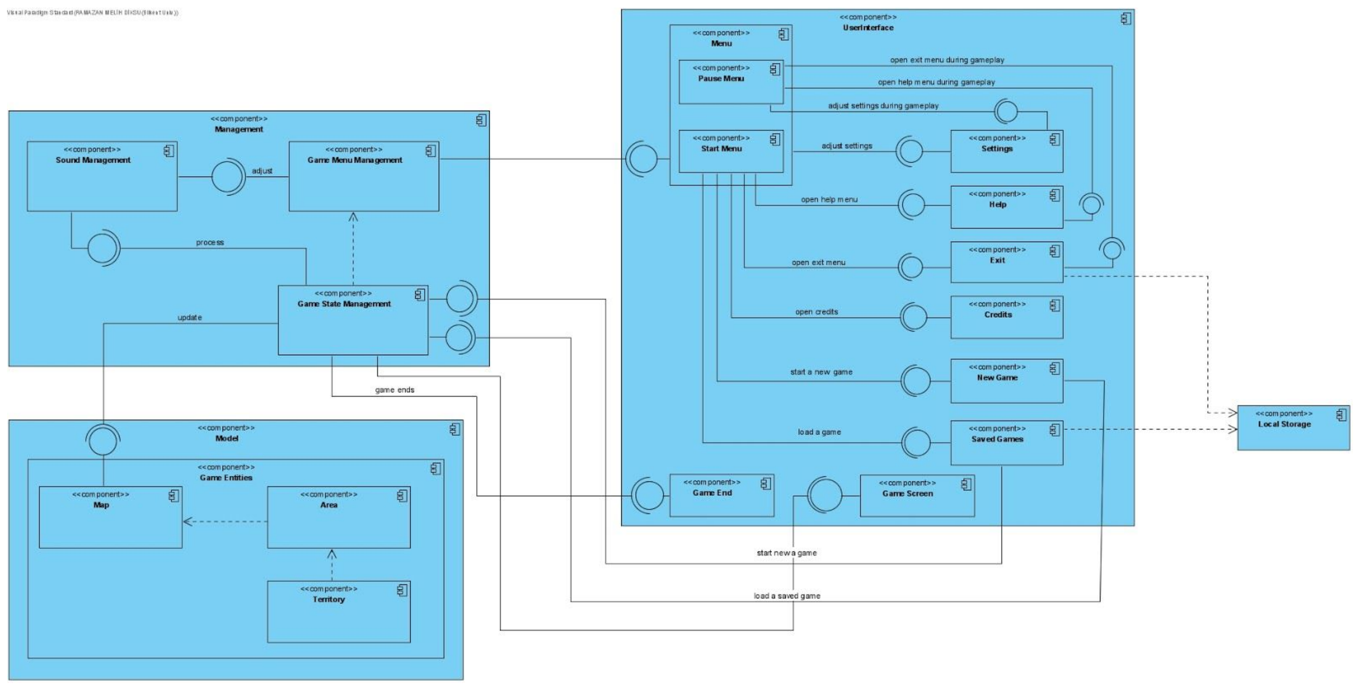


Figure.1

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 state of the game, the state of the map and the progress of the players 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 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.

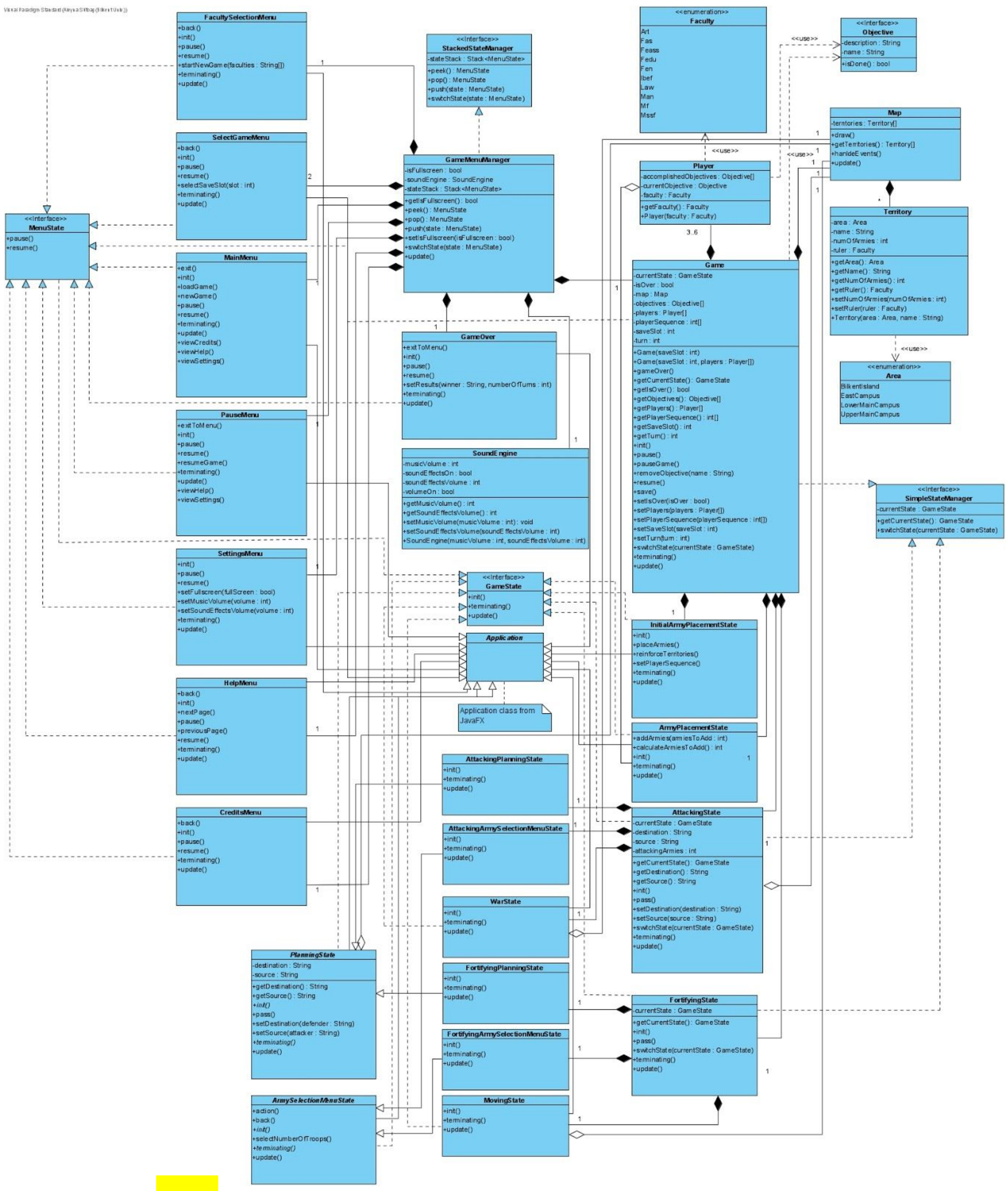
If a performance related crash occurs during gameplay, the save-file will not be corrupted because it is only changed at the end of a session.

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



URL: https://imgur.com/a/fXvGkEt

3.3 Packages

3.3.1 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.

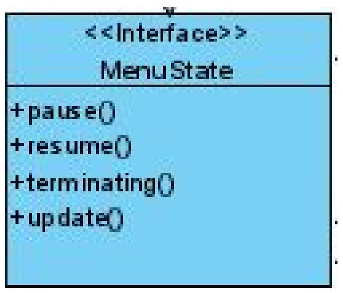
3.3.2 External Packages

1. **java.util**
   Data structures that will be included in our implementation such as arrays will be imported through util. Also, by taking advantage of Serializable we will be able to use hard drivers to store our game data.
2. **javafx.scene**
   We will use several classes from the scene package(paint, image, layout...) in order to create a graphical user interface which is organized and polished. Also, we will use events and input classes in order to handle the interaction between users and the game.

3. **javafx.animations**
   We will use this package to make our game more interesting in terms of the visuals by incorporating animations.

3.4 Class Interfaces

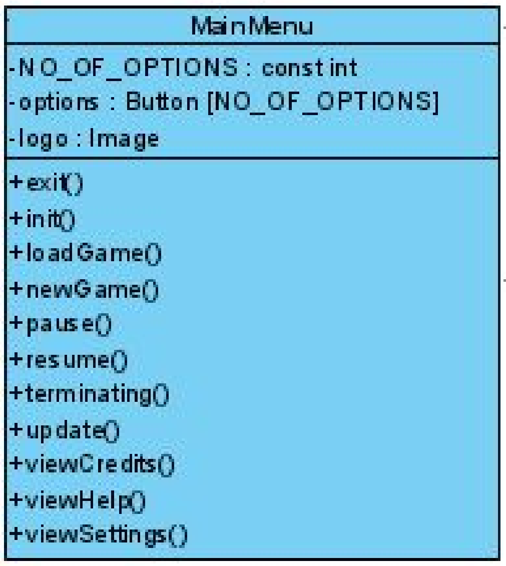
Menu State



Operations:

+pause(): This will pause the execution of a MenuState when it is not active, e.g. not on top of the StateStack.
+resume(): This will resume the execution of a MenuState when it comes to the top of the StateStack.
+terminating(): Called when the program is terminating.
+update(): Used to update the MenuState.

MainMenu



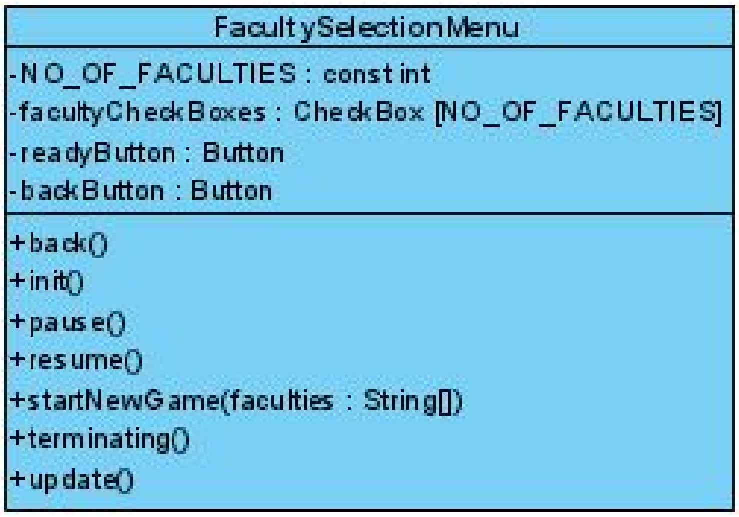
Attributes:

-NO_OF_OPTIONS: This constant int indicates how many options there are to select from in the options menu.
-options: This array of JavaFX Button objects holds all the buttons on the main menu.
-logo: This Image object is used to display the logo for the game.

Operations:

-exit(): Exits the program.
-init(): Initializes the MainMenu.
-loadGame(): Opens the SelectGameMenu.
-newGame(): Opens the FacultySelectionMenu.
-viewCredits(): Opens the CreditsMenu.
-viewHelp(): Opens the HelpMenu.
-viewSettings(): Opens the SettingsMenu.

FacultySelectionMenu



Attributes:

-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.

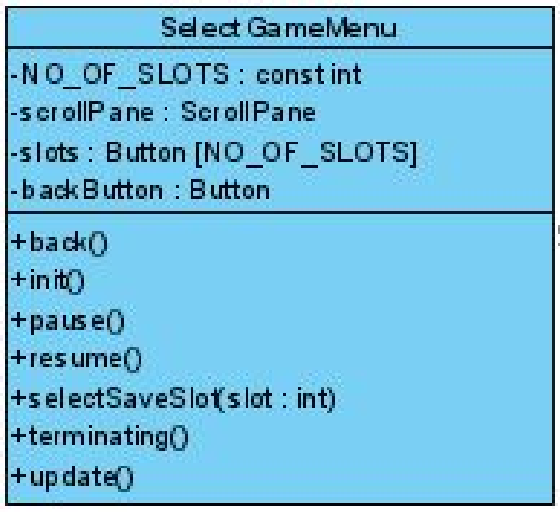
Operations:

+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.

SelectGameMenu



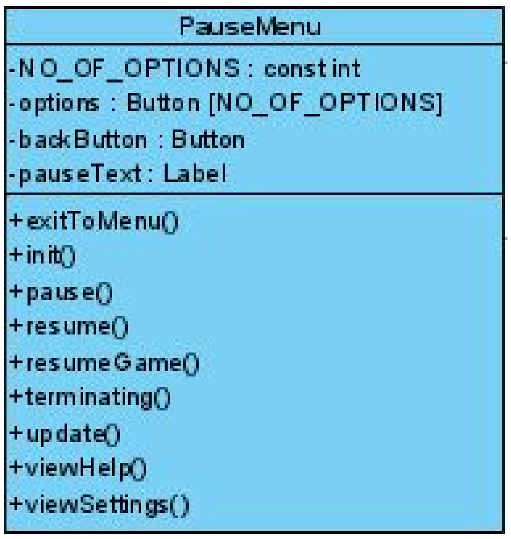
Attributes:

-NO_OF_SLOTS: This constant int indicates the number of total save slots.
-scrollPane: This JavaFX ScrollPane object is used to hold the Buttons for the 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.

Operations:

+back(): This operation goes back to the Main Menu.
+init(): Initializes the SelectGameMenu object.
+selectSaveSlot(slot: int): This operation is called when the user selects a save slot to load a game from.

PauseMenu



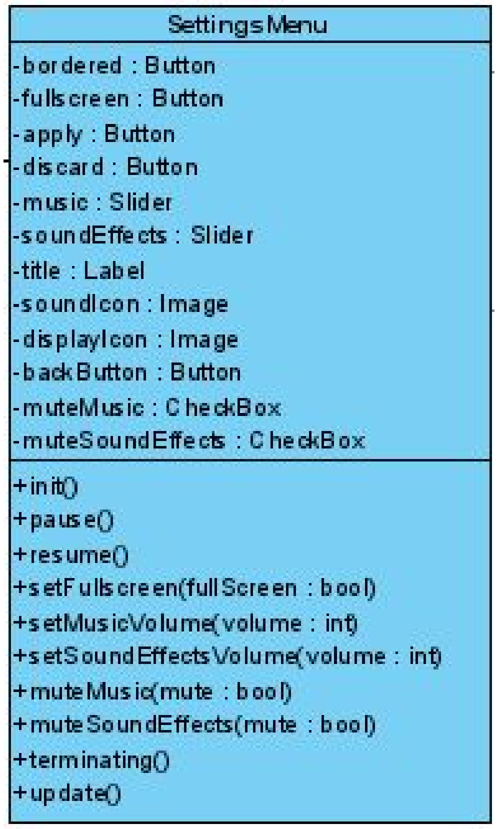
Attributes:

-NO_OF_OPTIONS: This constant int indicates the number of options available in the menu.
-options: This array of JavaFX Button objects holds a button for each option.
-backButton: This JavaFX Button object is used to go back to game.
-pauseText: This JavaFX Label object is used to represent the title of the pause menu.

Operations:

+exitToMenu(): This operation exits to the main menu.
+init(): Initializes the PauseMenu.
+resumeGame(): Resumes the game.
+viewHelp(): Opens the HelpMenu.
+viewSettings(): Opens the SettingsMenu.

SettingsMenu



Attributes:

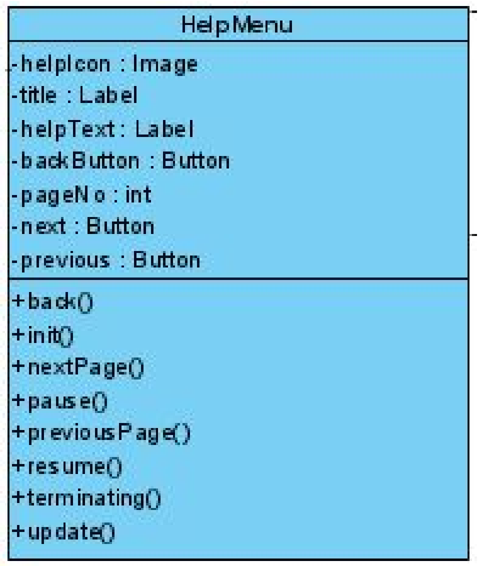
-bordered: This JavaFX Button is used to select the bordered display option.
-fullscreen: This JavaFX Button is used to select the full screen display option.
-apply: This JavaFX Button is used to apply the changes.
-discard: This JavaFX Button is used to discard the changes and go back to the previous settings.
-music: This JavaFX Slider is used to change the music volume.
-soundEffects: This JavaFX Slider is used to change the sound effect volume.
-title: This JavaFX label is used to represent the title for the settings Menu.
-soundIcon: This JavaFX Image is used for aesthetic purposes.
-displayIcon: This JavaFX Image is used for aesthetic purposes.
-backButton: This JavaFX Button is used to go back to the previous menu.
-muteMusic: This JavaFX CheckBox is used to mute the music.
-muteSoundEffects: This JavaFX CheckBox is used to mute the sound effects.

Operations:

+init(): Initializes the SettingsMenu.
+setFullScreen(fullScreen: bool): This operation sets the fullscreen option.
+setMusicVolume(volume: int): This operation sets the music volume.

+setSoundEffectsVolume(volume: int): This operation sets the sound effect volume.
+muteMusic(mute: bool): This operation is used to mute or unmute the music.
+muteSoundEffects(mute: bool): This operation is used to mute or unmute the sound effects.

HelpMenu



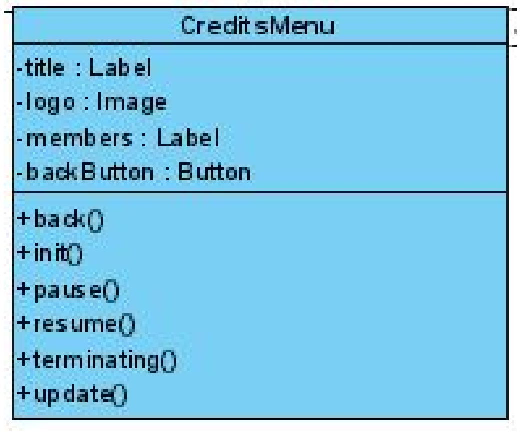
Attributes:

-helpIcon: This Image is used for aesthetic purposes.
-title: This JavaFX Label is used to represent the title of the menu.
-helpText: This JavaFX Label shows the help text.
-backButton: This Button is used to go back to the previous menu.
-pageNo: Holds the number of the current page the player is on in the help manual.
-next: This Button is used to go to the next page in the help manual.
-previous: This Button is used to go to the previous page in the help manual.

Operations:

+back(): Used to go back to the previous menu.
+init(): Initializes the HelpMenu.
+nextPage(): Displays the next page in the help manual.
+previousPage(): Displays the previous page in the help manual.

CreditsMenu



Attributes:

-title: This Label is used to represent the title of the menu.

-logo: This Image is used for aesthetic purposes.

-members: This Label lists the members of the development team.

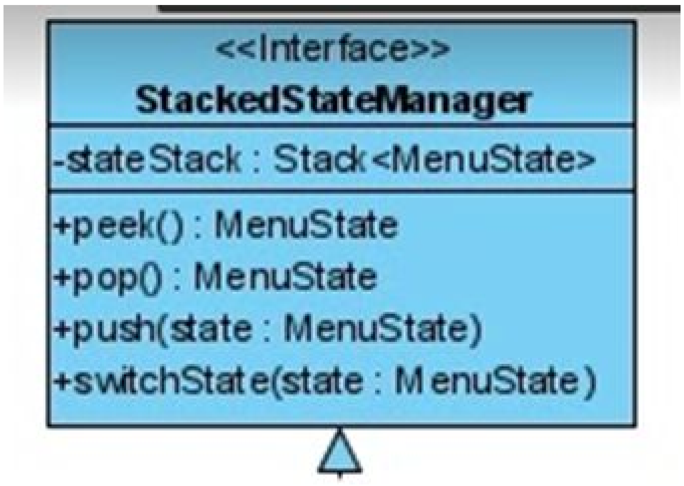
-backButton: This Button is used to go back to the previous menu.

Operations:

+back(): Goes back to the previous menu.

+init(): Initializes the CreditsMenu.

StackedStateManager



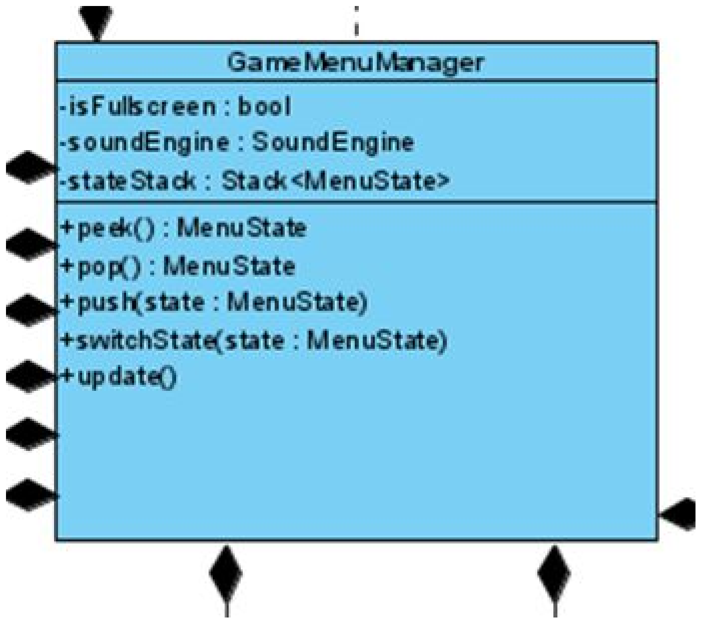
Attributes:

-stateStack: A stack that will hold all of the states of a game.

Operations:

+peek(): This operation will enable us to get the state at the top of the stateStack.
+pop(): This operation will enable us to get and remove the state at the top of the stateStack.
+push(state: MenuState): This operation will enable us to add a new state to stateStack.
+switchState(state: MenuState): This operation will enable us to change the state at the top of the stack.

GameMenuManager



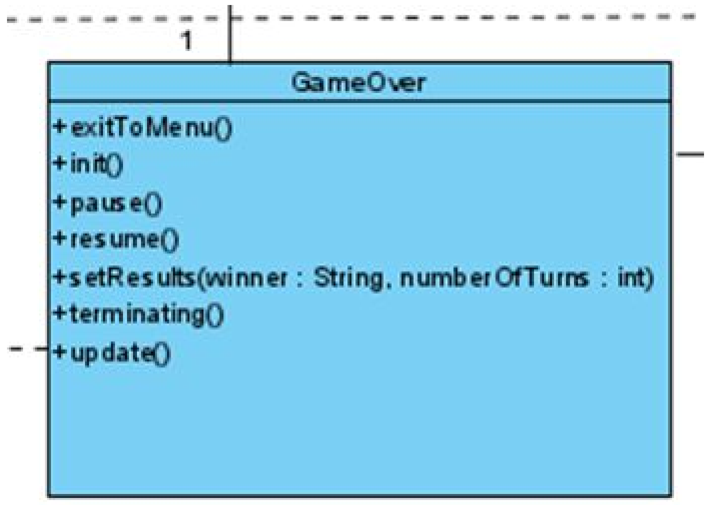
Attributes:

-isFullscreen: This will hold the screen size preference of the user. Will be true if the game is in fullscreen mode, false otherwise.
-soundEngine: This attribute is an instance of the SoundEngine class which will hold and handle operations about the sound effects and music.
-stateStack: A stack that will hold all of the states of a game.

Operations:

+peek(): This will return the active state of the game, which is the top state in stateStack.
+pop(): This will return and remove the active state of the game, which is the top state in stateStack.
+pushState(state: MenuState): This will enable us to add a new state to stateStack.
+switchState(state: MenuState): This will enable us to change the state at the top of the stack which is the active state of the game.
+update(): This operation will update the GameMenuManager.

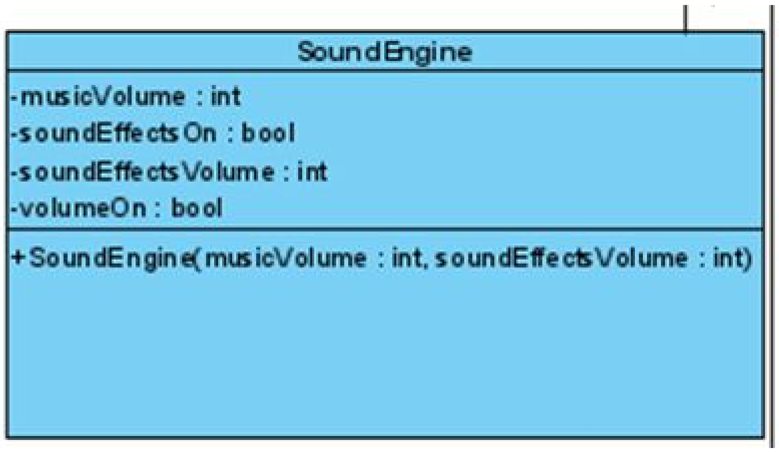
GameOver



Operations:

+exitToMenu(): This will enable users to return to the main menu.
+init(): Initializes the GameOver menu state.
+pause(): Pauses the GameOver menu state
+resume(): Resumes the GameOver menu state.
+setResults(): Sets the winner of the game and number of turns that the game lasted.
+terminating(): Terminates the GameOver screen.
+update(): Updates the GameOver screen.

SoundEngine


SoundEngine
-musicVolume : int
-soundEffectsOn : bool
-soundEffectsVolume : int
-volumeOn : bool
+SoundEngine(musicVolume : int, soundEffectsVolume : int)

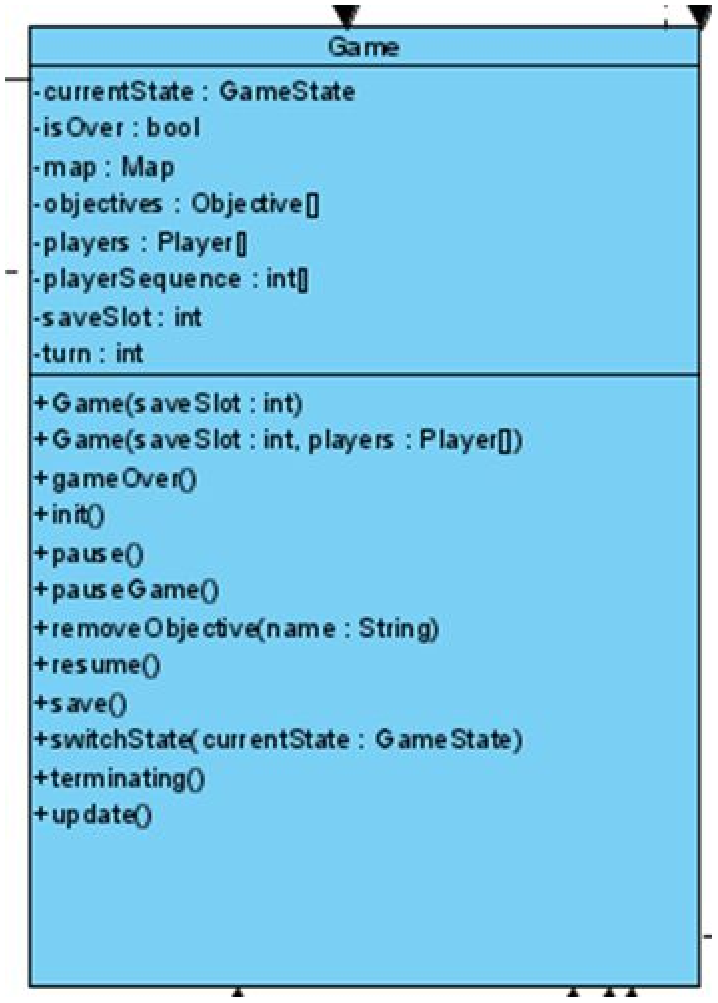
Attributes:

-musicVolume: This will hold an integer which represents the volume of the music of the game.
-soundEffectsOn: This will hold a boolean which will be true if the sound effects are on, false otherwise.
-soundEffectsVolume: This will hold an integer which represents the volume of sound effects of the game.
-volumeOn: This will hold a boolean which will be true if the volume is on, false otherwise.

Operations:

+SoundEngine(musicVolume: int, soundEffectsVolume: int): Constructor for the SoundEngine

Game



Attributes:

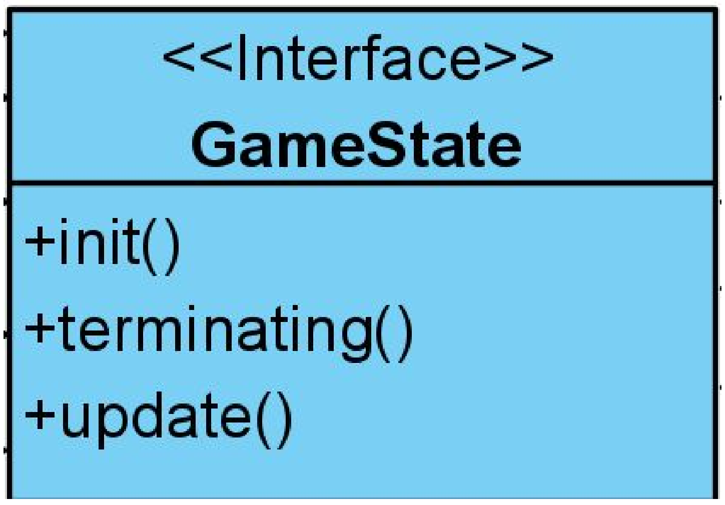
-currentsState: Holds the active state of the game.
-isOver: A boolean that determines whether the game is over or not.
-map: An instance of the game map.
-objectives: An array which includes the objectives in a game.
-players: An array which includes the players.
-playerSequence: An array which includes the sequence of the players.
-saveSlot: Holds the number of the current save slot.
-turn: Holds the number of turns in the game.

Operations:

+Game(saveSlot: int): A constructor for the Game class with a saveSlot parameter which indicates the slot that the game will be saved into.
+Game(saveSlot: int,players: Player[]): A constructor for the Game class with a saveSlot parameter which indicates the slot that the game will be saved into and players parameter which will hold the players in the game.

+gameOver(): A method that will be called when the game is over.
+handleEvents(): An event handler for the game.
+init(): Initializes the Game state.
+pause(): Pauses the Game state.
+pauseGame(): Pauses the game itself.
+removeObjective(name: String): Removes a completed objective from the game.
+resume(): Resumes the Game state.
+save(): Saves the game when users exit from the game.
+switchState(currentState: GameState): Changes the active state.
+terminating(): Terminates the game.
+update(): Updates the game.

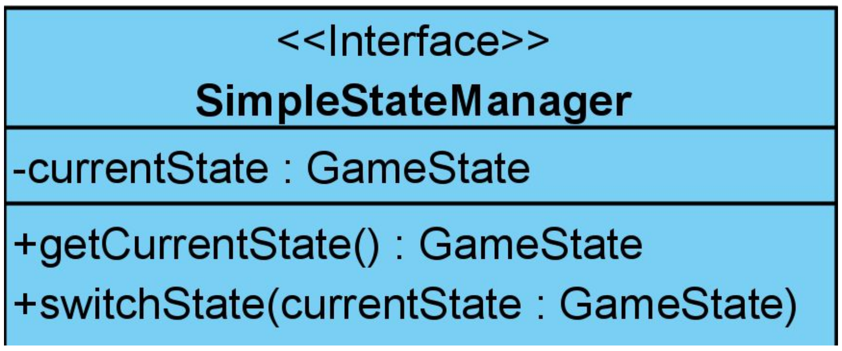
GameState



Operations:

+init(): Initializes the current state.
+terminating(): Terminates the current state.
+update(): Updates the display.

SimpleStateManager


<<Interface>>
SimpleStateManager
-currentState : GameState
+getCurrentState() : GameState
+switchState(currentState : GameState)

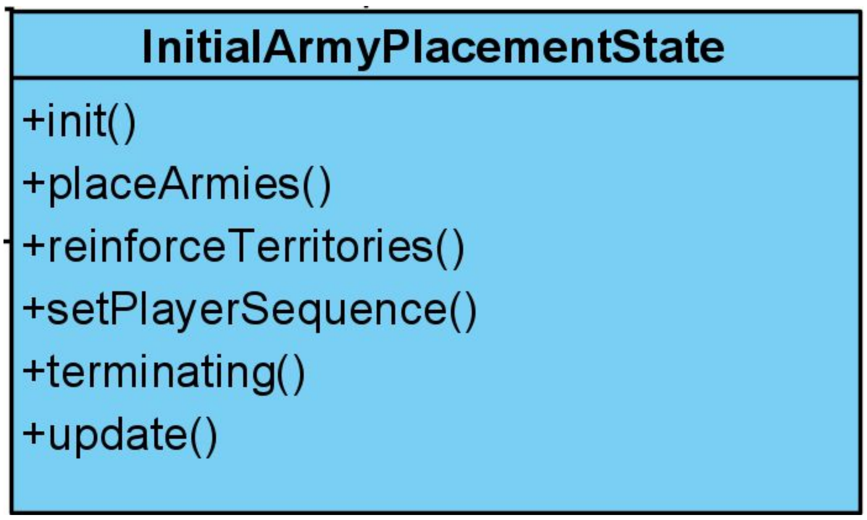
Attributes:

-currentState: Holds the active state of the game.

Operations:

+switchCurrentState(currentState: GameState): Switches to the received state.

InitialArmyPlacementState

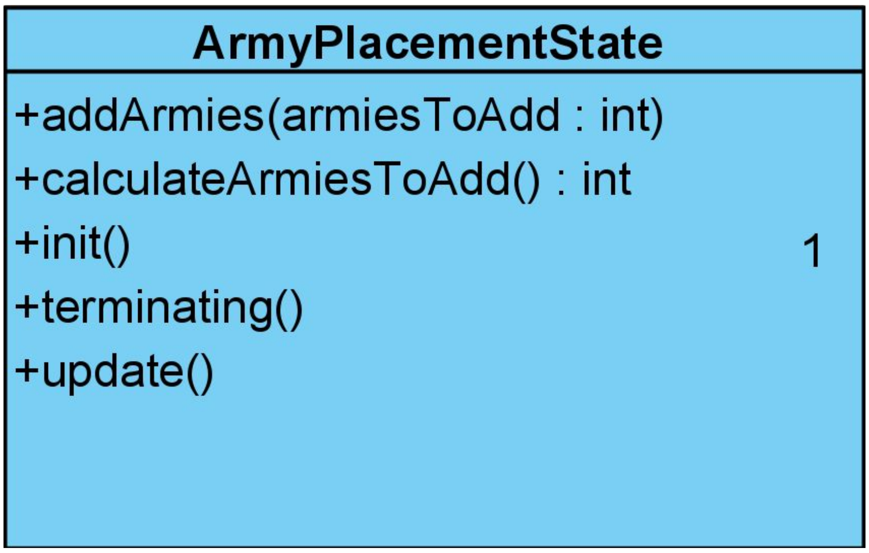


Implements the GameState interface.

Operations:

+placeArmies(): Enables users to place armies on unoccupied territories at the initial state in turns.
+reinforceTerritories(): After all the territories are occupied, this enables users to place one additional army to a territory occupied by them in turns.
+setPlayerSequence(): Determines the sequence of the player turns.

ArmyPlacementState

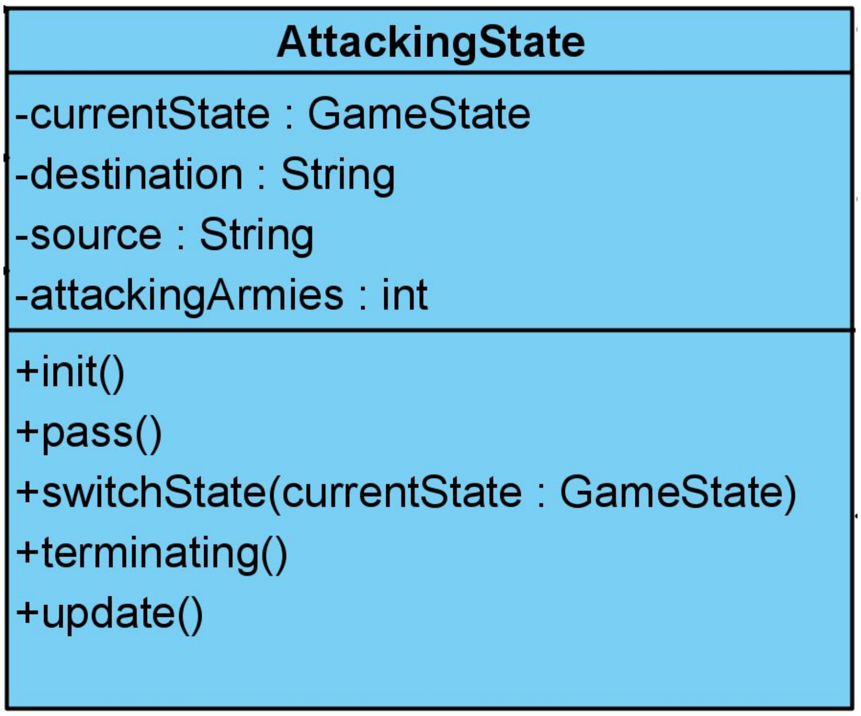


Implements the GameState interface.

Operations:

+addArmies(armiesToAdd: int): Enables users to add armies during the placement process.
+calculateArmiesToAdd(): int: Calculates the amount of armies for the placement process.

AttackingState



Implements the GameState and the SimpleStateManager interfaces.

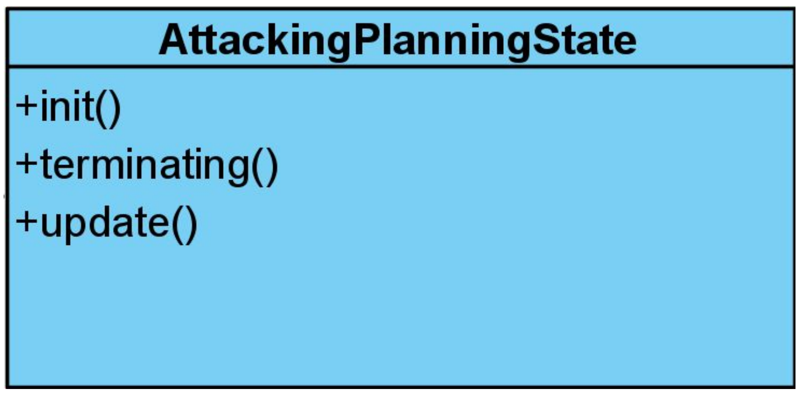
Attributes:

-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.

Operations:

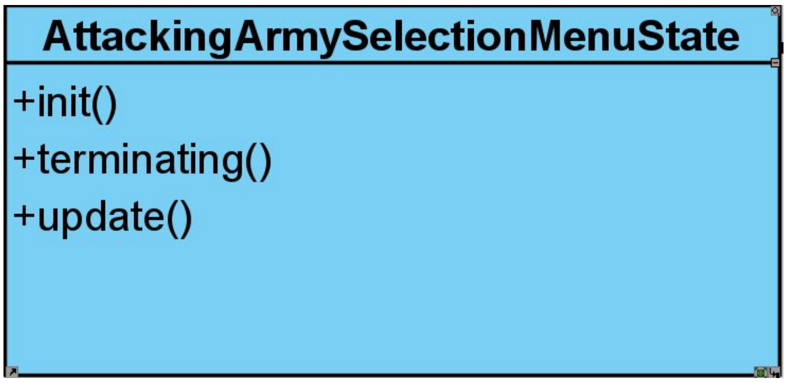
+pass(): Enables users to pass to the next state.

AttackingPlanningState



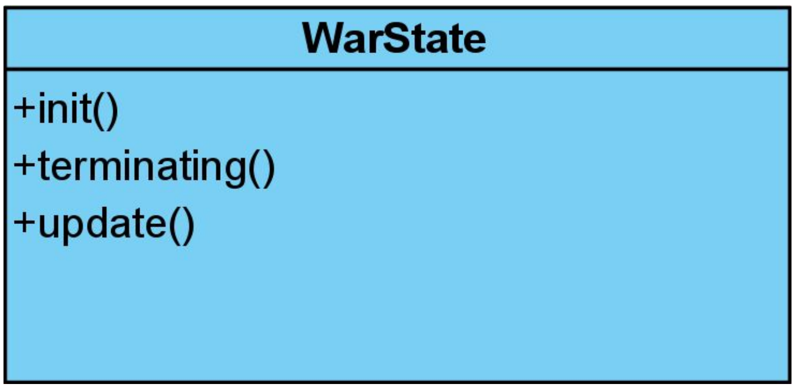
Implements the GameState interface.

AttackingArmySelectionMenuState



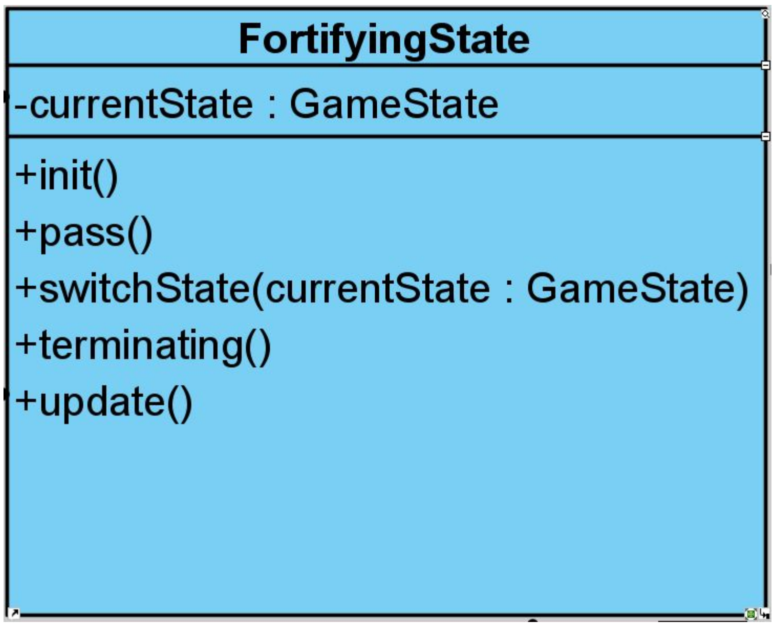
Implements the GameState interface.

WarState



Implements the GameState interface.

FortifyingState

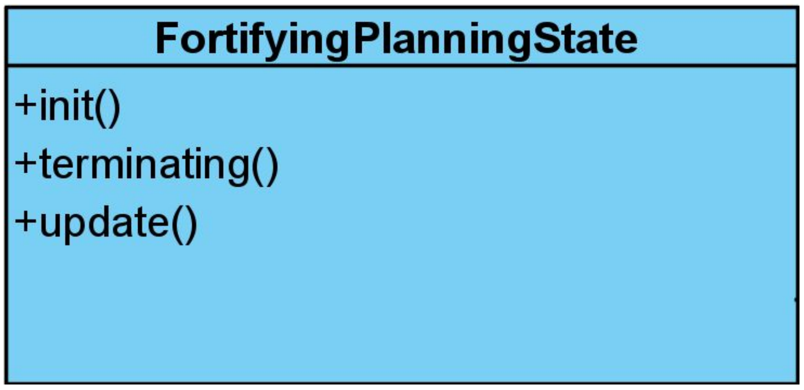

FortifyingState
-currentState : GameState
+init()
+pass()
+switchState(currentState : GameState)
+terminating()
+update()

Implements the GameState and the SimpleStateManager interfaces.

Operations:

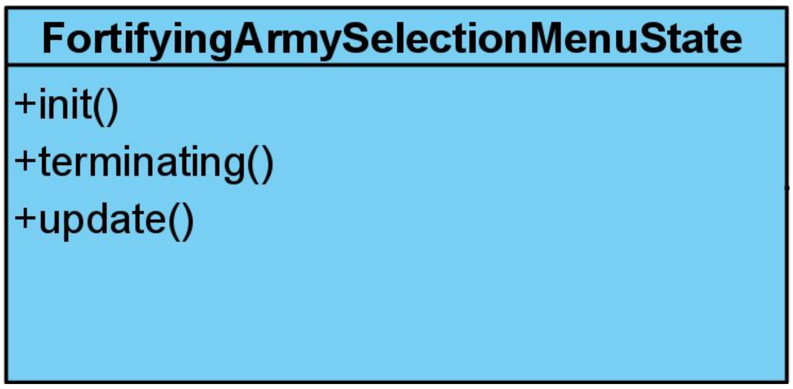
+pass(): Enables users to pass to the next state.

FortifyingPlanningState



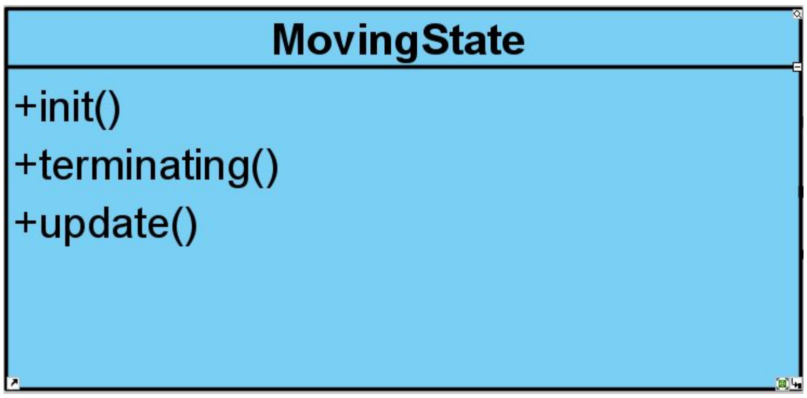
Implements the GameState interface.

FortifyingArmySelectionMenuState



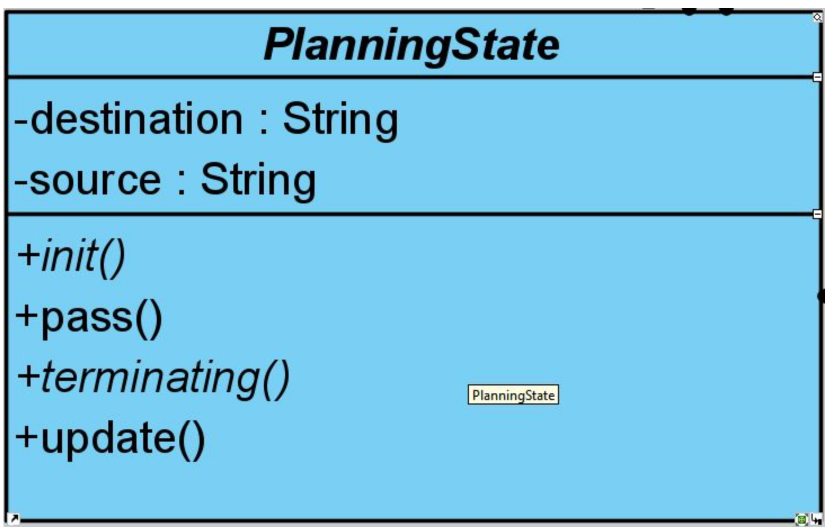
Implements the GameState interface.

MovingState



Implements the GameState interface.

PlanningState



Abstract class that implements the GameState interface.

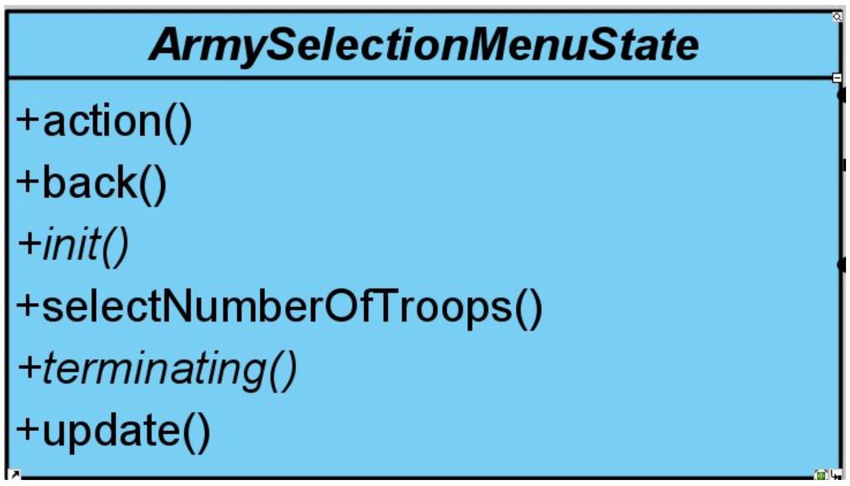
Attributes:

-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:

+pass(): Skips to the next state.

ArmySelectionMenuState

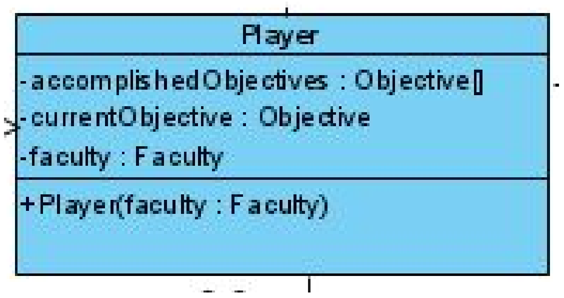


Abstract class that implements the GameState interface.

Operations:

+action(): Enables users to confirm the selected number of troops.
+back(): Enables users to go back in order to reselect the number of troops.
+selectNumberOfTroops(): Enables users to select the number of troops for the related action (e.g. selecting number of troops to attack or number of troops to fortify with)

Player



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

Attributes:

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

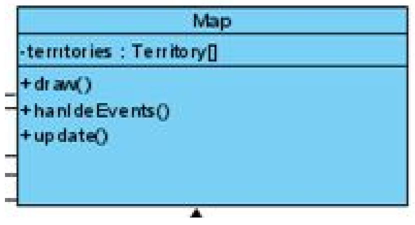
-currentObjective: This attribute holds the Objective object which is the current objective.

-faculty: This attribute holds the Faculty object.

Constructor:

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

Map



Attributes:

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

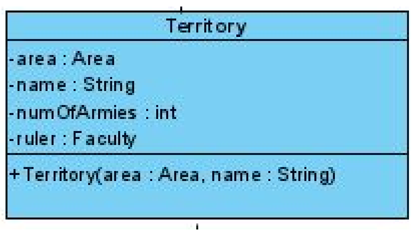
Methods:

+draw(): This method draws the map.

+handleEvents(): This method handles the events on the map.

+update(): This method updates the map according to the player's moves.

Territory



Attributes:

-area: This attribute holds the Area object.

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

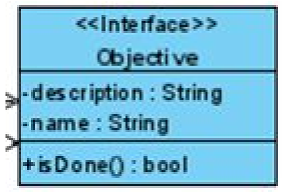
-numOfArmies: This attribute holds the value of the armies in that territory as a int value.

-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



Attributes:

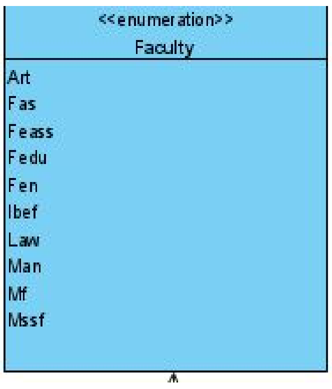
-description: This attribute holds the description of the objective as a String.

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

Methods:

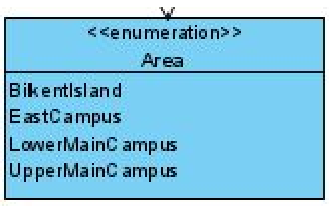
+isDone(): bool: This method checks if the objective is done or not. It returns true if the objective is done and it returns false if the objective is not done.

Faculty



Risk101 contains Art, Fas, Feass, Fedu, Fen, Ibef, Law, Man, Mf, Mssf as a Faculties.

Area



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