Bilkent University

Group: 1G

Design Report Draft

Emoji Stkire

Author:

Ali Salemwala

Bora Bardük

Eren Çalik

Kıvanç Gümüş

Supervisor:

Bora Güngören

March 25, 2017

Contents

1 Introduction

1.1 Purpose of the System

Emoji Strike is a 2-D arcade game. The game is designed so that a group of friends can have good a time playing the game on the same device competitively. The game firstly introduces the user with a brief tutorial to teach the user how the game is played. As the user(s) decide that they have learned this gameplay and controls, they move on to the game building stage which includes user's picking their desired character and map of choice. When this game building stage is done users can enjoy the game they built for themselves until one of the them becomes the victor. Therefore, our main purpose is to design a dynamic and a competitive game.

1.2 Design Goals

1.2.1 End User Criteria

Usability: The system is a game, so the main priority should be making sure that the user enjoys the game. For this purpose, the game is designed in a way that a newbie player and an experienced player can both enjoy the game same way. This means that a first time user can just run the game, learn the game and play the game accordingly and a more experienced player can run the game, play the game with better knowledge so that he/she can play it more efficiently.

Learnability: All the times that the game is launched, the user can check the controls and the gameplay features via tutorial screen. Since this is an arcade game, the features and controls are kept simple with just keyboard inputs. The in-game features like power-ups and weapons will be learned through experience but since the logic of the game is simple, the user can easily learn these features.

Portability: Since the game will run on the Java Virtual Machine, the system is highly portable. The system has no other portability considerations rather than these.

1.2.2 Maintenance Criteria

1.2.3 Performance Criteria

1.2.4 Trade Offs

1.3 Definitions, Acronyms, Abbreviations

1.4 References

1.5 Overview

2 Software Architecture

2.1 Overview

This section will decompose our program into smaller, more manageable parts, so that it may be built more efficiently and cleanly. Additionally, partitioning the sections will mean that the different program parts will not interfere with each other, and one misbehaving section may not disturb other sections. We will try to use a Model-View-Controller style architecture on our program.

2.2 Subsystem Decomposition

To ensure that our system contains maintanable, discrete, independant sections, we will break it up into 3 section: mainly the User Interface Section, which will serves as the view, the Game Management Section, which serves as the controller, and the Game system itself, which is the model. This way, we can not only independantly develop the sections, but also prevent bugs from one section from leaking into others. Additionally, our code will remain maintainable in the future, since related functionalities will be grouped together and the future maintainers will have a well-documented and partitioned system to maintain. Below are diagrams indicating how the system and actions inside it are divided up. They can serve not only as guidelines while the system is being developed, but also as references for the future when the system is already running.

2.3 Architectural Styles

2.3.1 Layers

This system has been divided into three layers, which are closely mirrored by the MVC-style architectre the entire program follows. The top-most layer, the User-Interface Layer, is the one that is in direct contact with the user. It is the only layer which the user is allowed to directly influence. Beneath this comes the Game Management layer, which is synonomous to the Game Management subsystem. This can only be influenced by the User Interface layer, and directly influences the Game layer. It serves as a pipeline between what the user sees and the actual system, and transfers any changes from one end to the other. Finally, comes the Game layer, which can be considered to be the engine of the entire system. It is the actual state of the system, and is altered directly by the Game Management layer. Changes are passed back through the same layer. This layered system has been created to prevent any undue influencing of the system by the user, so that only appropriate layers can be changed, and these changes are done in a controlled, pre-decided manner.

2.3.2 Model View Controller

The Model-View-Controller style serves as a complement to the layer sytle described above. It divides the system into the Model, which cannot be altered directly by anybody except the Controller, which is merged with the View to display the whole program to the user. Since our program is a gme, it is especially useful for our system to be divided as an MVC-style architecture, since the game state must be influenced only by the user, but must also not be freely changed, and should instead have a controlled pathway for changes to take the proper effect.

2.4 Hardware/Software Mapping

EmojiStrike will be implemented in Java, and will use JDK 1.8. The hardware used are a keyboard, speakers, and the monitor display. The software needed is only an operating systme which can run the .jar file which will contain our game. Storage will be done in .txt format, so the operating system should be able to store the saved files.

2.5 Persistent Data Management

The game does not have any complex or detailed information that needs to be stored. The default maps will be stored, and the edited versions with other game details will be stored as individual files. The game will be able to function if the saved files are corrupted, although the saved game will obviously be lost. Sound files and images for the emojis, weapons, and animations will also be stored, and if these are corrupted, the game will experience significant problems.

2.6 Access Control and Security

There are no network or security requirements for this game.

2.7 Boundary Conditions

Below is a list of boundary conditions in our application.

Initialization: There is no need for any installation. Simply running the main .jar file is enough to run the game, as long as the system has Java installed.

Termination: The system can exit with a simple keypress, both with or without saving the game. Saving the game will also require the user to enter a name for the current game.

Error: The game may not load saved games if the files are corrupted.

The game may not load certain images or sounds if the files are corrupted.

An error while playing the game might cause the entire system to shut down, but we will try to include ways to handle all errors and maintain user experience.

3 Subsystem Design

In this section, we will bridge the gap between a problem and an existing system in a manageable way by using divide and conquer approach.

3.1 Architectural Style

3.1.1 Model View Controller Design

We chose to implement our subsystem decomposition in MVC style. That is, our subsystems are decomposed into 3 different types.

Model: Responsible for application domain knowledge
View: Responsible for displaying application domain objects to the user
Controller: Responsible for sequence of interactions with the user and notifying views of changes in the model

This design pattern enables us to differentiate between three main components of our system, providing the required abstraction during the software architecture tasks.

3.2 Detailed Object Design

Figure 1 illustrates the User Interface Management Subsystem Diagram.

User Interface Management Subsystem consists of 11 classes which contains the views and information necessary to establish communication between users and Emoji Strike. There are two interfaces implemented and used during the design of out interface. These classes and their properties will be discussed in-depth in the following sections.

3.2.1 Menu

Figure 2 illustrates the Menu Class.

3.2.1.1 Attributes

List of attributes used in Menu class.

private JFrame frame: All visuals of the window will be displayed here.

private KeyListener listenerKey: Our users will provide input to Emoji Strike using keyboard buttons.

private ActionListener listener: When a certain menu button is activated, this listener will retrieve the activated object to perform related operations.

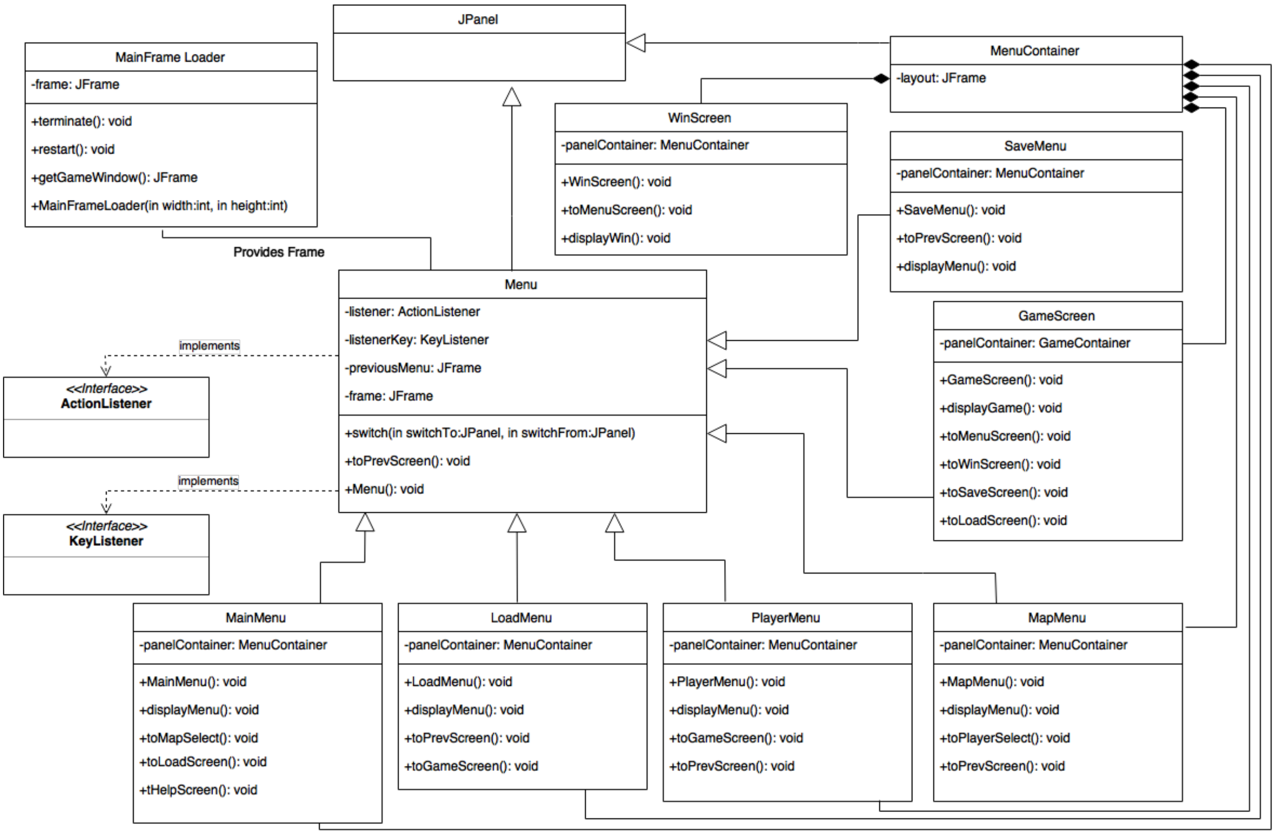


Figure 1: User Interface Management Subsystem

private JFrame previousMenu: This attribute will contain the last accessed JFrame before the current one. This is used to provide ”go back” function to any screen in the game.

3.2.1.2 Constructors

List of constructors used in Menu class.

public Menu: Initializes frame, listener, listenerKey, previousMenu which are the attributes of Menu class.

3.2.1.3 Operations

List of operations used in Menu class.

public void switch: Used to switch between different panel instances.

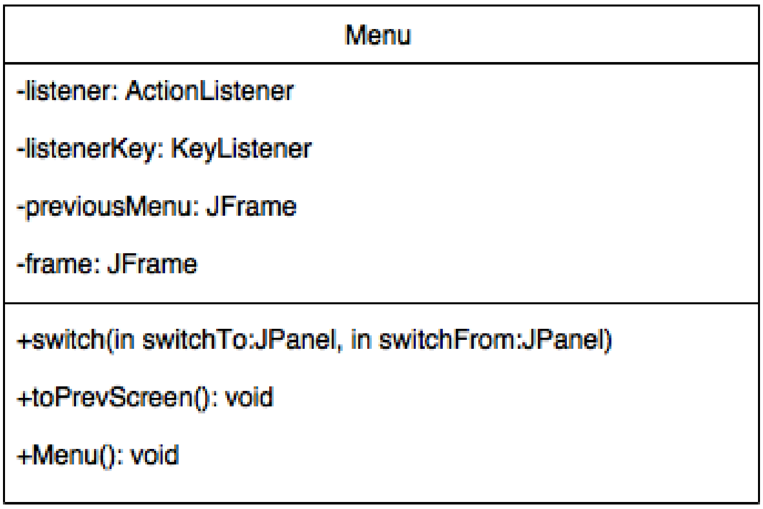

Menu
-listener: ActionListener
-listenerKey: KeyListener
-previousMenu: JFrame
-frame: JFrame
+switch(in switchTo:JPanel, in switchFrom:JPanel)
+toPrevScreen(): void
+Menu(): void

Figure 2: Menu Class

public void toPrevScreen: Used to switch to the previous screen by accessing the private attribute *previousMenu*.

3.2.2 MainFrame Loader

Figure **??** illustrates the MainFrameLoader Class.

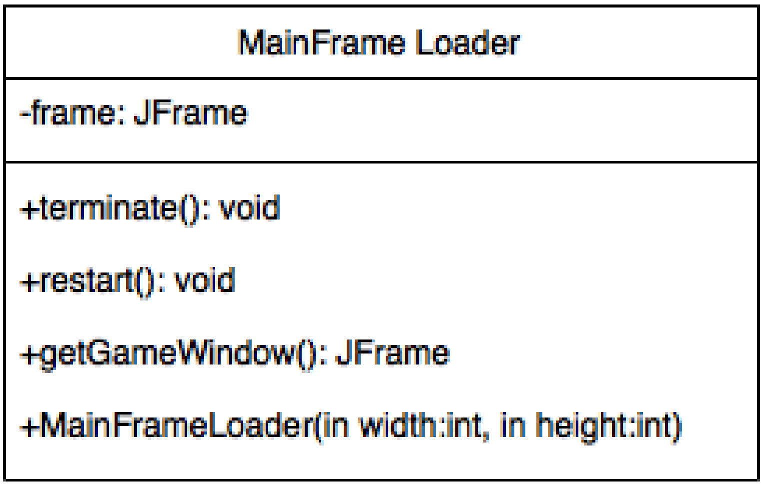


Figure 3: MainFrame Loader Class

3.2.2.1 Attributes

List of attributes used in MainFrameLoader class.

private JFrame frame: Opens up the main screen which will then be used by the Menu class and its panels. All visuals of the window will be displayed here.

3.2.2.2 Constructors

List of constructors used in MainFrameLoader class.

public MainFrameLoader: Initializes frame which is the attribute of MainMenuLoader class. It consists of parameters width and height, which will be the screen resolution determined for a display.

3.2.2.3 Operations

List of operations used in MainFrameLoader class.

public JFrame getGameWindow: Returns the JFrame object which will be displayed on the screen. **public void terminate:** Used to terminate the main screen instance when operations like *quit* are performed.

public void restart: Used to restart the main frame of the game when user prompts.

3.2.3 MainMenu

Figure 4 illustrates the MainMenu Class.

3.2.3.1 Attributes

List of attributes used in MainMenu class.

private MenuContainer panelContainer: The container which holds the graphical elements of the MainMenu class.

3.2.3.2 Constructors

List of constructors used in MainMenu class.

public MainMenu: Initializes panel which is the attribute of MainMenu class.

3.2.3.3 Operations

List of operations used in MainMenu class.

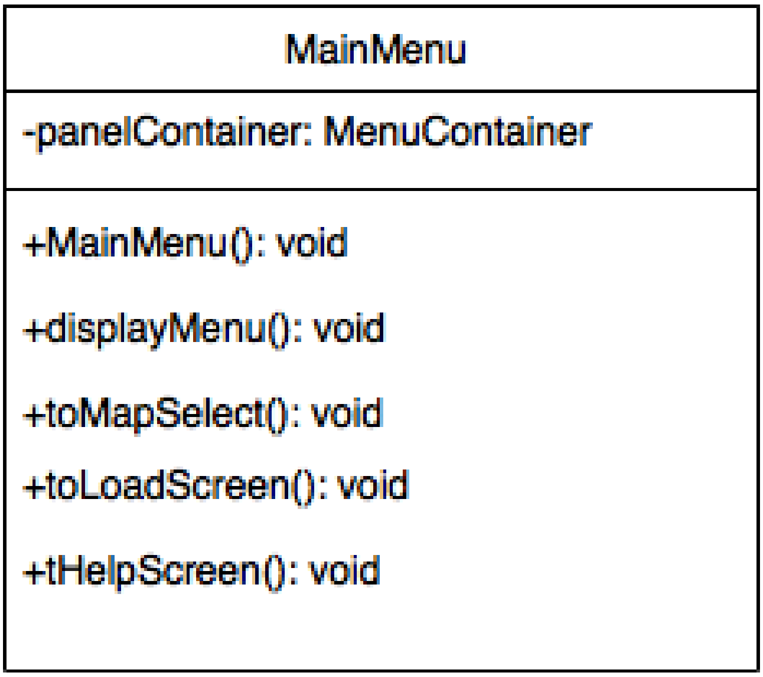


Figure 4: MainMenu Class

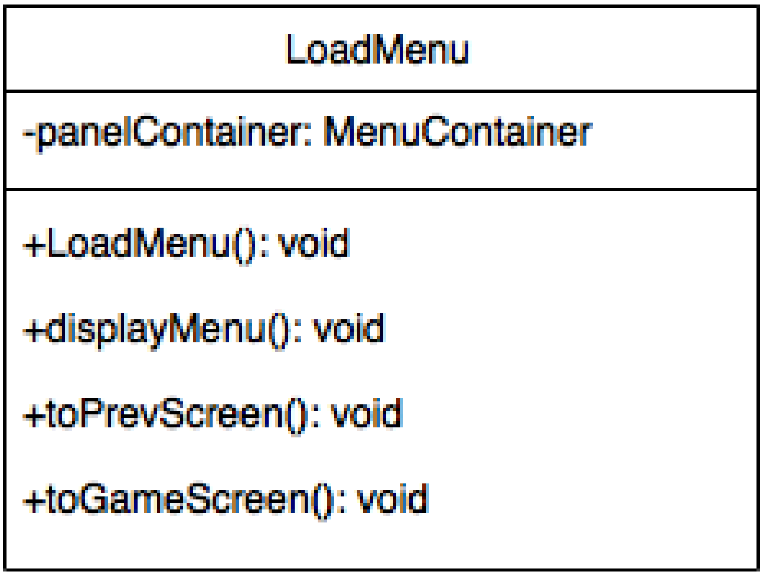
public JFrame getGameWindow: Returns the JFrame object which will be displayed on the screen.
public void displayMenu: Used to load the graphical elements after the instance of the class is created.
public void toMapSelect: Switches the panel view to *MapSelectMenu* which is where user selects the map to play the game on.
public void toLoadScreen: Switches the panel view to *LoadMenu* which is where user selects from the previous game files saved.
public void tHelpScreen: Toggles the graphical elements which will load the *How to play* instructions.

3.2.4 LoadMenu

Figure 5 illustrates the LoadMenu Class.

3.2.4.1 Attributes

List of attributes used in LoadMenu class.



LoadMenu

-panelContainer: MenuContainer

+LoadMenu(): void

+displayMenu(): void

+toPrevScreen(): void

+toGameScreen(): void

Figure 5: LoadMenu Class

private MenuContainer panelContainer: The container which holds the graphical elements of the LoadMenu class.

3.2.4.2 Constructors

List of constructors used in LoadMenu class.

public LoadMenu: Initializes panel which is the attribute of LoadMenu class.

3.2.4.3 Operations

List of operations used in LoadMenu class.

public void displayMenu: Used to load the graphical elements after the instance of the class is created.

public void toPrevScreen: Used to switch to the previous screen by accessing the private attribute *previousMenu*.

public void toGameScreen: After the the load file is prompted by the user, the information is initialized to appear in *GameScreen*.

3.2.5 PlayerMenu

Figure 6 illustrates the PlayerMenu Class.

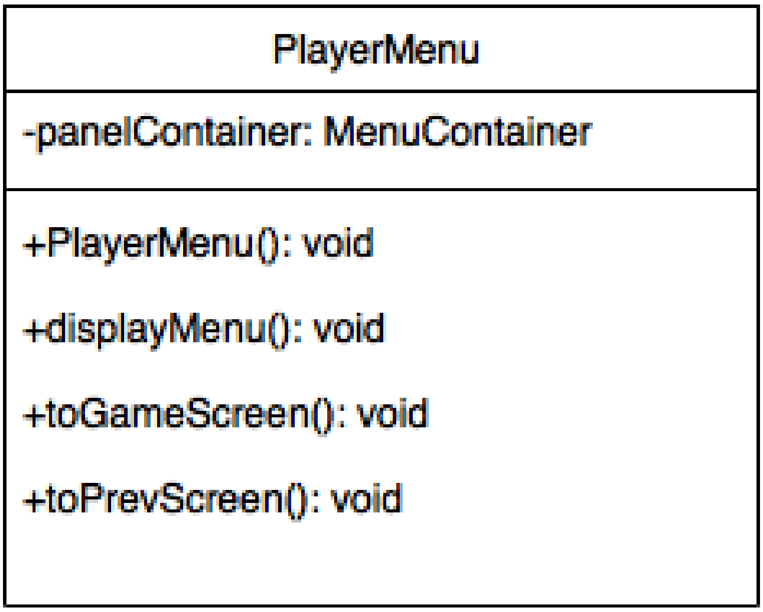


Figure 6: PlayerMenu Class

3.2.5.1 Attributes

List of attributes used in PlayerMenu class.

private MenuContainer panelContainer: The container which holds the graphical elements of the PlayerMenu class.

3.2.5.2 Constructors

List of constructors used in PlayerMenu class.

public PlayerMenu: Initializes panel which is the attribute of PlayerMenu class.

3.2.5.3 Operations

List of operations used in PlayerMenu class.

public void displayMenu: Used to load the graphical elements after the instance of the class is created.

public void toPrevScreen: Used to switch to the previous screen by accessing the private attribute *previousMenu*.

public void toGameScreen: After the players are initialized, the information is directed to appear in *GameScreen*.

3.2.6 MapMenu

Figure 7 illustrates the MapMenu Class.

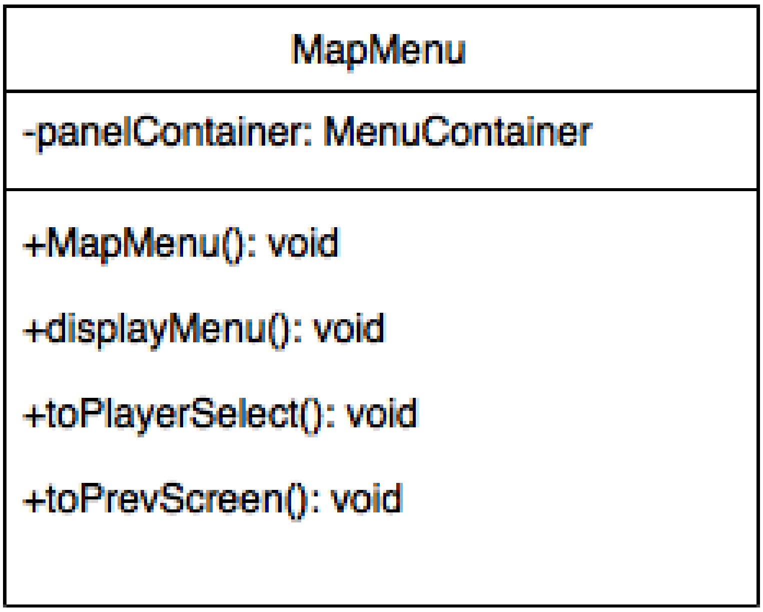


Figure 7: MapMenu Class

3.2.6.1 Attributes

List of attributes used in MapMenu class.

private MenuContainer panelContainer: The container which holds the graphical elements of the MapMenu class.

3.2.6.2 Constructors

List of constructors used in MapMenu class.

public MapMenu: Initializes panel which is the attribute of MapMenu class.

3.2.6.3 Operations

List of operations used in MapMenu class.

public void displayMenu: Used to load the graphical elements after the instance of the class is created.

public void toPrevScreen: Used to switch to the previous screen by accessing the private attribute *previousMenu*.

public void toPlayerSelect: After the map is selected, user is directed to enter player information for them to appear in the *GameScreen*.

3.2.7 SaveMenu

Figure 8 illustrates the SaveMenu Class.

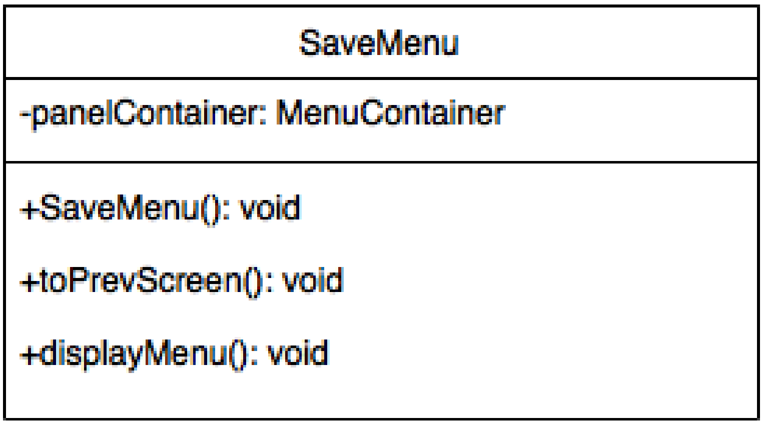


Figure 8: SaveMenu Class

3.2.7.1 Attributes

List of attributes used in SaveMenu class.

private MenuContainer panelContainer: The container which holds the graphical elements of the SaveMenu class.

3.2.7.2 Constructors

List of constructors used in SaveMenu class.

public SaveMenu: Initializes panel which is the attribute of SaveMenu class.

3.2.7.3 Operations

List of operations used in SaveMenu class.

public void displayMenu: Used to load the graphical elements after the instance of the class is created.

public void toPrevScreen: Used to switch to the previous screen by accessing the private attribute *previousMenu*.

3.2.8 GameScreen

Figure 9 illustrates the GameScreen Class.

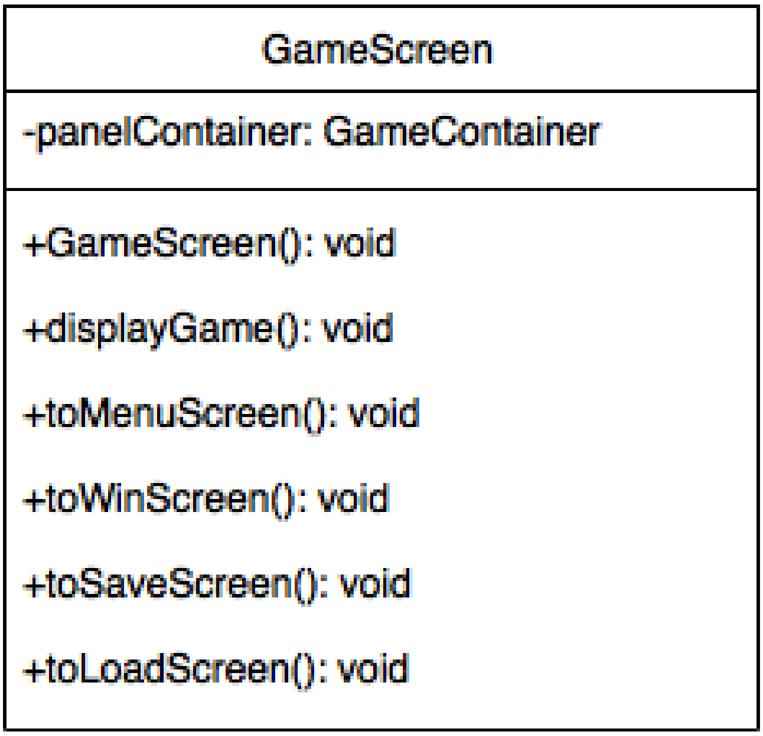


Figure 9: GameScreen Class

3.2.8.1 Attributes

List of attributes used in GameScreen class.

private MenuContainer panelContainer: The container which holds the graphical elements of the GameScreen class.

3.2.8.2 Constructors

List of constructors used in GameScreen class.

public GameScreen: Initializes panel which is the attribute of GameScreen class.

3.2.8.3 Operations

List of operations used in GameScreen class.

public void displayGame: Used to load the graphical elements after the instance of the class is created.

public void toMenuScreen: Used to immediately return to the view panel store in *MainMenu*.

public void toWinScreen: After the winner is determined, this operation is used to switch to view *WinScreen*.

public void toLoadScreen: Switches the panel view to *LoadMenu* which is where user selects from the previous game files saved.

public void toSaveScreen: Switches the panel view to *LoadMenu* which is where user selects from the previous game files saved.

3.2.9 WinScreen

Figure 10 illustrates the WinScreen Class.

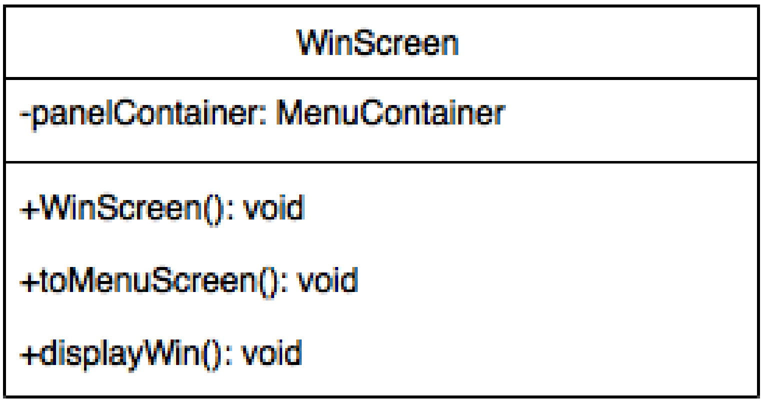


Figure 10: WinScreen Class

3.2.9.1 Attributes

List of attributes used in WinScreen class.

private MenuContainer panelContainer: The container which holds the graphical elements of the GameScreen class.

3.2.9.2 Constructors

List of constructors used in WinScreen class.

public WinScreen: Initializes panel which is the attribute of WinScreen class.

3.2.9.3 Operations

List of operations used in GameScreen class.

public void displayWin: Used to load the graphical elements after the instance of the class is created.

public void toMenuScreen: Used to immediately return to the view panel store in *MainMenu*.

3.2.10 MenuContainer

Figure 11 illustrates the MenuContainer Class.

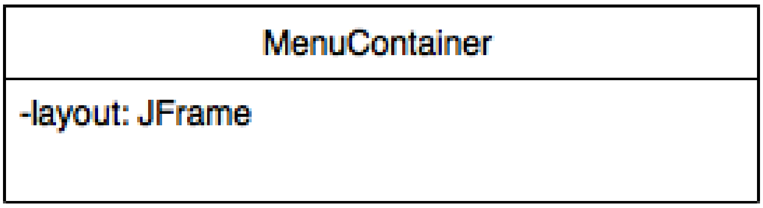


Figure 11: MenuContainer Class

3.2.10.1 Attributes

List of attributes used in MenuContainer class.

private JFrame layout: All visuals of the window will be displayed here.

References