Bilkent University

Department of Computer Engineering

**CS 319 Term Project**

*Section 1*

*Group 1A*

*Walls and Warriors*

Design Report

**Project Group Members**

1. Dawood Muzammil Malik
2. Deniz Yüksel
3. Faruk Oruç
4. Mert Soydinç
5. Muhammed Salih Altun

**Supervisor: Eray Tüzün**

**Table of Contents**

[**1 Introduction** 3](#_Toc529390385)

[1.1. Purpose of the System 3](#_Toc529390386)

[1.2. Design Goals 3](#_Toc529390387)

[**2 High-Level Software Architecture** 5](#_Toc529390388)

[2.1 System Decomposition 5](#_Toc529390389)

[2.2 Hardware/Software Mapping 6](#_Toc529390390)

[2.3 Data Persistent Management 6](#_Toc529390391)

[2.4 Access Control and Security 7](#_Toc529390392)

[2.5 Boundary Conditions 7](#_Toc529390393)

[**3 Subsystem Services** 9](#_Toc529390394)

[3.1 Interface Subsystem 10](#_Toc529390395)

[3.2 Game Management Subsystem 10](#_Toc529390396)

[3.3 Data Management Subsystem 11](#_Toc529390397)

[**4 Low- Level Design** 12](#_Toc529390398)

[4.1 Class Interface 12](#_Toc529390399)

[4.2 Packages 25](#_Toc529390400)

[**5 Glossary & References** 26](#_Toc529390401)

# Introduction

## **Purpose of the System**

The purpose of Walls and Warriors Desktop Edition is to interpret the classical board game, Walls and Warriors. Walls and Warriors is a mind game that stimulates the players to think and solve puzzles with time. Our game also adds the feature of sandbox mode, which will allow the users to edit and save their levels locally to challenge others.

## **Design Goals**

Our design goals specify the qualities our design will focus. We have our priorities for our design for the optimal quality. The criteria and how we will satisfy them are described below:

* **Maintainability:**
* ***Expandability:*** Expandability is essential for most video games because users have expectancy for new features in games. Our game design will be in 2D and with Java. It is crucial to develop a game with simple, clean and readable code so that our implementation is understood and can be further developed by other software developers. After reading and understanding our implementation, a software developer should be able to develop new features.
* ***Reliability:*** Reliability is significant for the sustainability of the game. If our game design has the minimum bugs to lead crashes and failures, sustainability can be achieved and therefore our software will be reliable thus expandable.
* **Usability:**
* ***Simplicity:*** Simplicity can be related with user- friendliness in a way that simple designs are usually more user friendly in terms of the limitations of the capabilities of the user. As the complexity of the system increases, it is a key factor to keep the simplicity. This is why our design has simple operations in the menu containing settings, play game, high scores and how to play. If the user selects play game, he/she will be shown two choices that are classical game and editor mode (sandbox).
* ***User- friendliness:*** One of the most important objective of our design is that it is user friendly. The game menu will have an option of how to play, where the game will be demonstrated. Even if a person who did not play the original game will be able to cover the mechanics of the game. Also, we will provide the players with an online documentation, if in any case the game experiences failures or crashes for the tutorial section. Or the players will want to learn the game before playing so they will be ready to play. Also, our user friendly design is intuitive e.g. the settings button will be at the top right corner symbolized with a gear. We will use conventional design methodologies for user friendly design for size of buttons and game objects. Starting from the launching of the game, the player will only use the mouse to play the entire game because the mouse is a common tool to navigate inside a computer. Also it is easy and applicable to our game.
* **Performance:**
* ***Smoothness:*** Smoothness or the fewness of the response time of games is essential for all gamers. Although our game does not have a relationship with the user like such games that performance is extremely important, (e.g. first person shooter, real time strategy, competitive) we will still ensure that our users will enjoy the performance of our game. This includes pressing buttons, selecting wall pieces, rotating them for the classical game. For the sandbox mode we will provide the user with a smooth experience of picking and placing walls, tower and knights in an instant. After all, the user would not want to lose time waiting for the selection of a piece of wall after clicking.

# High-Level Software Architecture

## **System Decomposition**

In order to improve the performance of *Walls and Warriors*, we have decided to follow the MVC (Model-View-Controller) design approach to implement the game. Using this architecture, we intent to make the game more maintainable, meaning that it would be easier to extend or make changes to the game in the future.

In order to make sure that our system architecture complies with the rules of MVC, we have divided our system intro three subsystems. The three subsystems are as follows:

* Interface Subsystem
* Game Management Subsystem
* Data Management Subsystem

A detailed description of the above subsystems can be found in *Section 3* of this report.

## **Hardware/Software Mapping**

Walls & Warriors will be developed in Java programming language. Further details about the libraries to be used will be later discussed in the report.

The primary target audience of Walls & Warriors are children aged between 8-15 years old. Therefore, as developers, we believe that it is essential that the hardware requirements to play the game are as simple as possible. Given that the user has a working computing unit and a monitor, the user will just need to have a mouse and a keyboard in order to play the game.

In order to avoid any compatibility issues, Walls and Warriors will just be implemented on the Windows operating system in this moment of time. The user must have Windows operating system in order to run and play the game.

## **Data Persistent Management**

Walls & Warriors will have a very simple and efficient data management system. Famous for its better speed as compared to MySQL, we will be using MongoDB as our database. All the levels, their states (locked or unlocked), game settings and players’ names along with their high scores will be saved in an instance of a MongoDB database running on user’s system.

Data elements such as levels, their states, players’ name and their high scores will be protected and therefore will not be modifiable. The only option will be to reset the game which will clear all the high scores and start the game from level one will all subsequent levels locked.

User will have the option to update game settings at any time. All the changes applied on the game screen will also be reflected in the database, indicating that the new settings have been updated.

Walls & Warriors will have autosave feature. When the user completes a level, the state of the next level will be automatically updated and saved in the database. The user will never be asked to save the game manually.

## **Access Control and Security**

Being an offline game and having no features related to user accounts, Walls & Warriors will not be requiring any security as the game would not be requiring the user to enter any sensitive information. The game would not require parental consent or any other type of access permissions as the game is targeted for people of all ages, ranging from children to adults.

## **Boundary Conditions**

* + 1. **Initialization**

In order to make the game portable and easy to play, Walls & Warriors will not need to be installed on a computer. In other words, the game will not have an .exe extension that needs to be executed. Instead, the game will have an executable JAR file (.jar) which will start the game right away. Once started, the user will just need to click on the *Start Game* button in order to begin playing.

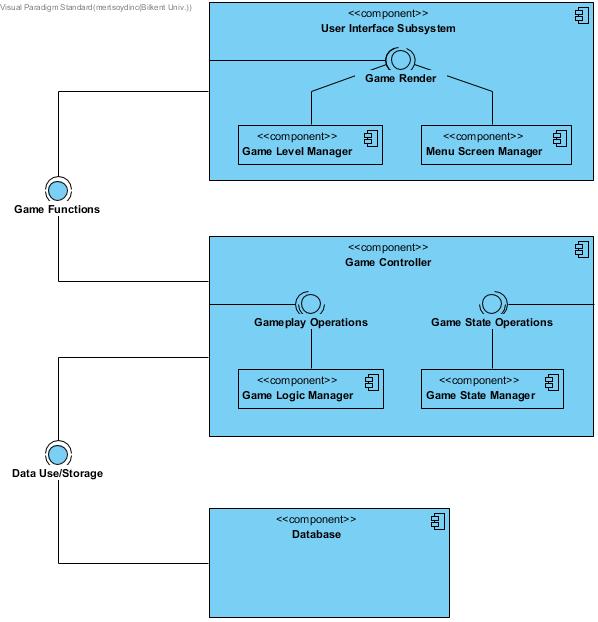
* + 1. **Termination**

The game can easily be terminated by clicking the *Quit Game* button from the main menu and the pause menu.

* + 1. **Failure**

If, at any time, the game fails to fetch the data regarding levels from the database, the game will crash. Having the autosave functionality will be beneficial in this case as user’s progress will be automatically saved and will not be dependent on user remembering to save his progress.

# Subsystem Services

As mentioned in the section 2.1, we will be strictly following the *Model-View-Controller* design approach to implement *Walls and Warriors*. Below is a diagram the three subsystems, following a detailed description of their individual components.

* 1. **Interface Subsystem**
     1. **User Interface Subsystem**

This component handles the graphical user interfaces, which includes the game menus, game levels and the transitions between them. This component can be split into two smaller subcomponents called Game Level Manager and Menu Screen Manager. This subsystem can only communicate with the Game Management System.

* + - 1. **Game Level Manager**

This subcomponent controls the display of the game screen. The game screen will be composed of game pieces but their positions on the board will be different for each level. This component will display the right positions with the data it acquires from the Game Management Subsystem.

* + - 1. **Menu Screen Manager**

This subcomponent will control the display of the menu screens, which include main menu, level selection menu, options menu, etc.

* 1. **Game Management Subsystem**
     1. **Game Controller**

This component controls the entire flow of the game. It is the only subsystem in the project that can communicate with all the other systems. For this purpose, it is composed of two subcomponents, which are Game Logic Manager and Game State Manager.

* + - 1. **Game Logic Manager**

This subsystem imposes the game rules and is responsible for the gameplay of Walls and Warriors. Using the information it receives from the Game State Manager, it utilizes the game logic and controls the Game pieces accordingly.

* + - 1. **Game State Manager**

This subsystem is the main control of the game. Its responsibilities include sending and getting relevant information from the Game Logic Manager, handling the requests and sending commands to the User Interface Subsystem and reading from or writing to the games Database Subsystem. Using the combined data from User İnterface and the Database Subsystems, Game State Manager controls the other subsystems of the game and enables controlled interactions between them.

* 1. **Data Management Subsystem**
     1. **Database**

This subsystem is tasked with storing data and sending data to the Game State Manager. It stores the information of all the predefined levels (where the pieces are), player’s data (Name, scores, unlocked levels) and information regarding user-made levels. This subsystem is controlled entirely by the GSM.

# Low- Level Design

* 1. **Class Interface**
* **Class GamePiece**:

There are 2 types of gamepieces, walls and knights. They have inheritance relationship with GamePiece class.

* + **Attributes:**
* **boolean movable**: indicates whether the piece would be movable or not.(knights vs walls)
* **boolean visible**: indicates whether the piece would be visible or not (ex: corners of the board are not visible)
* **Point position**: the position of the gamepiece
  + **Operations:**
* **public getMovable():**  gets if the piece is movable or not as boolean value
* **public setMovable(boolean movable)**: sets the piece to be movable or not
* **public getVisible()**:gets if the piece is visible or not as boolean value
* **public setVisible(boolean visible)**:sets the piece to be visible or not
* **public getPosition()**:gets the position of the piece as Point
* **public setPosition(Point position)**: sets the position of the piece
* **Class Wall:**

This class inherits gamePiece.

* + **Attributes:**
* **String shape:** the shape of the wall stored as a string (like UUL for 2 upwards 1 left wall)
  + **Operations:**
* **public getShape():** gets the shape of the wall as string
* **public setShape(String shape):** sets the shape of the wall
* **Class Knight:**

This class inherits GamePiece class.

* + **Attributes:**
* **boolean color:** the color of the knight is stored as boolean(0 if red 1 if blue)
  + **Operations:**
* **public getColor ():** gets the color of the knight as boolean
* **public setColor (boolean color):** sets the color of the wall
* **Class GameEngine:**

There are 2 types of game engines, LevelEngine and SandboxEngine. They have inheritance relationship with GameEngine class.

* + **Attributes:**
* **GameBoard** **levelBoard:** the current state of the level is kept here
* **List<Wall> walls:** the walls that are on the current board
* **List<Knights> knights:** the knights that are on the current board
  + **Operations:**
* **getLevelBoard():** gets the current levels board as GameBoard object
* **setLevelBoard(GameBoard gameBoard):** sets the level’s board
* **getWalls():** gets the walls on the board as List<Wall>
* **setWalls(List<Wall> walls):** sets the walls on the board
* **getKnights ():** gets the walls on the board as List<Knight>
* **setKnights(List<Knight> knights):** sets the walls on the board
* **levelComplete():** gets whether the board is complete or not as boolean value
* **Class LevelEngine:**

This class inherits GameEngine. It is used on normal and user made levels.

* + **Attributes:**
* **boolean gameState:** the state of the game(active or not)
* **int currentLevel:** the current level of the game
* **GameTimer timer:** the timer of the game to keep the scores
* **List<GamePiece> gamePieces:** the avaible game pieces for the current level
* **String initialBoardState:** the initial state of the level
  + **Operations:**
* **getGameState():** gets the state of the game as boolean
* **setGameState(boolean gameState):** sets the state of the game
* **getCurrentLevel():** gets the current level as int
* **setCurrentLevel(int currentLevel):** sets the current level of the game
* **checkVictory():** checks the victory of the level. If won returns true
* **Class GameTimer:**

Timer for a normal level

* + **Attributes:**
* **Timer timer:** a timer to keep the time
  + **Operations:**
* **getTime():** gets the current time as string
* **reset():** resets the timer
* **start():** starts the timer
* **stop():** stops the timer
* **Class SandboxEngine:**

This class inherits GameEngine. It is used on level creation(sandbox mode)

* + **Attributes:**
* **int boardDimensionX:** The horizontal size of the board
* **int boardDimensionY:** The vertical size of the board
  + **Operations:**
* **checkValidLevel():** checks whether the level could be completed and returns boolean value
* **addWall(Point[] wallPoints):** add a wall to the level by a point array and return boolean value for its success
* **addKnight(Point knightPoint):** add a knight to the level by a point and return boolean value for its success
* **addBlankSpace(Point[] blankSpaces):** add some blank spaces by a point array and return a boolean value for its success.
* **addTower(Point[] wallPoints):** add a tower to the level by a point array and return boolean value for its success
* **getBoardDimensionX():** gets the board size in horizontal as int value
* **setBoardDimensionX(int boardDimensionX):** sets the horizontal size of the board
* **getBoardDimensionY():** gets the board size in vertical as int value
* **setBoardDimensionY(int boardDimensionX):** sets the vertical size of the board
* **saveLevel(GameBoard gameBoard):** saves the board and returns boolean value for its success
* **Class DatabaseManager**
  + **Attributes:**
* **private List<String> levelInfo:** This list structure holds the info for default and user-created levels. Every level corresponds to a **String** in a certain format.
* **private List<String> highScoreList:** This list structure consists of high scores.
* **private int currentLevel:** Current level that the player is at. Also corresponds to how many levels they have unlocked.
* **Operations:**
* **public boolean isConnected():** This method checks if the connection to database is established.
* **public String getLevelInfo(int ID):** This method is a lookup method for information about a game level using its ID. Since we use **String** data type to store levels, it returns **String**.
* **public void setLevelInfo(List<String> levelInfo):** This method fills the database with level information.
* **public void addHighScore(int ID, String score):** This method will be used to upload a high score to the database.
* **public int getCurrentLevel():** Get method for the **currentLevel** attribute.
* **public void setCurrentLevel(int currentLevel):** Set method for the **currentLevel** attribute.
* **public List<String> getHighScoreList():** Get method for the **highScoreList** attribute.
* **Class MenuPage**
* **Attributes:**
  + - **private BorderPane layout:** This attribute is from javafx library, and is used to set borders and position items. This is the layout that all the menu pages will have.
    - **private Button button:** A button will be used in all menu pages. Button is from javafx from library.
    - **private TextField pageName:** TextField is from java.awt and will be used in all menus that have text entries.
* **Class MainMenu**

MainMenu is a MenuPage.

* **Attributes:** 
  + - **private Button newGameButton:** This button is for the user to start a new game.
    - **private Button optionsButton:** This button is used for accessing the options menu which is described in the “Class OptionsPage” section.
    - **private Button quitButton:** This button is used to quit the game.
    - **private Button settingsButton:** This button is used for accessing the settings menu which is described in the “Class SettingsPage” section.
* **Class NewGamePage**

NewGamePage is a MenuPage.

* **Attributes:** 
  + - **private Button levelsButton:** This button is used to view the levels.
    - **private Button createLevelButton:** This button is used to access the sandbox menu.
    - **private Button backButton:** This button is used to go to the previous screen.
* **Class LevelsPage**

LevelsPage is a MenuPage.

* **Attributes:**
* **private TabPane levelTypeTab:** This attribute is in order to manage between a group of tabs, which is from javafx library.
* **private Button[] levelButtons:** This is a button array that allows the user to choose a playable level. A playable level is either passed, or the next level before the last passed level.
* **private Button backButton:** This button is used to go to the previous page.
* **private Button startButton:** This button is used to start a level.
* **private Button deleteLevelButton:** This button is used to delete a user- created level.
* **private TextField levelHighScore:**levelHighScore is displayed after a level finishes.
* **Operations:**
* **public String getLevelHighScoreInfo():** This method is to display the highscores of a played level in the levels page.
* **public boolean setLevelHighScore( String levelHighScore):** This method is to set an highscore to the level after being played. It return true if an highscore is set.
* **public String getLevels():** This method returns the levels as a String.
* **Class OptionsPage**

OptionsPage is a MenuPage.

* **Attributes**:
* **private Button highScoresButton:** This button is used to display the highscores.
* **private Button gameInstructionsButton:** This button is used to display the game instructions.
* **private Button aboutUs:** This button is used to display the information about the group 404: Name Not Found.
* **private Button creditsButton:** This button is used to display the credits.
* **private Button backButton:** This button is used to go to the previous page.
* **Operations:**
* **public void displayHighScores:** This method is used to display the highscores.
* **public void displayGameInstructions:** This method is used to display the game instructions.
* **public void displayAboutUs:** This method is used to display the information about the group 404: Name Not Found.
* **public void displayCredits:** This method is used to display the credits.
* **Class SettingsPage**
* **Attributes**:
* **private slider soundLevel:** This button is used to adjust the sound level in the game.
* **private choiceBox resolution:** This button is used to change the resolution from low, medium or high.
* **private checkBox fullScreen:** This button is used to play the game in full screen.
* **private Button backButton:** This button is used to go to the previous page.
* **Class GUIManager**
* **Attributes:**
* **private Stage mainStage:** Stage attribute is from javaFX library. mainStage is constructed by the platform.
* **Operations:**
* **public void start():** This method is from javaFX library which is the entry point of all javaFX applications.
* **public void initializeScreen():** This method sets up the initial properties of the screen.
* **public void updateScreen():**This method updates the screen when there is going to be an alteration.
* **public void updateScene():** This method is called when there needs to be a visual effect on the screen e.g. placing a wall, rotating a wall, moving a selected wall.
* **Class InputListener**

This class implements MouseListener and KeyListener interfaces.

* **Operations:**
* **public void mouseClicked( e: MouseEvent):** This method catches a mouse click.
* **public void keyPressed( e: KeyEvent):** This method catches when a corresponding key on the keyboard is pressed.
* **public void mousePressed( e: MouseEvent):** This method is activated when the mouse click is pressed.
* **public void mouseDragged(e: MouseEvent):** This method is activated when the mouse is dragged. This will be used while placing the walls to desired locations in the game.
* **public void mousePressed( e: MouseEvent):** This method is activated when the mouse click is pressed.
* **Class GameController**
* **Attributes:**
* **private GameEngine gameEngine:** This attribute is the game’s engine. Since **GameController** controls both the UI and the game, it has an instance of the game engine inside it. Using this attribute ensures there is only one instance of **GameEngine** in the game.
* **private DatabaseManager databaseManager:** This is an instance of the **DatabaseManager** of the game.
* **private GUIManager GUIManager:** This attribute is an instance of **GUIManager**.
* **Methods:**
* **public void createGameEngine():** This methods creates an instance of **GameEngine** and ensures there is only one **GameEngine** created.
* **public void createDatabaseManager():** This methods creates an instance of **DatabaseManager** and ensures there is only one **DatabaseManager** created.
* **public void create GUIManager():** This methods creates an instance of **GUIManager** and ensures there is only one **GUIManager** created.
* **public Point getMove():** This method is invoked when the player makes a move in the game. We get the move via a mouse event and a **Point** type object is returned.
* **public boolean isMoveValid():** This is a method to confirm whether a move that user made is valid in the current game state or not.
* **public void applyMove():** This method is used to apply the user’s move to the board after it is confirmed that the move was valid.
* **public void updateGameData():** This method updates the state of the board when the user makes a move.
* **public void updateGUI():** This method updates the game’s user interface when game data is updated.
* **public void levelFinished():** This method is invoked when a level is finished.
* **Class GameEngine**
* **Attributes:**
* **private GameBoard levelBoard:** This attribute is a reference to the current state of the game board.
* **private List<Knight> knights:** This **List** structure holds the knights that are available in the current game.
* **private List<Wall> walls:** This **List** structure holds the walls that are available in the current game.
* **Operations:**
* **public GameBoard getLevelBoard():** This is a get method for the **levelBoard** attribute.
* **public void setLevelBoard(levelBoard board):** This is a set method for the **levelBoard** attribute.
* **public List<Wall> getWalls():** This is a get method for the walls attribute.
* **public void setWalls(List<Wall> walls):** This is a set method for the walls attribute.
* **public List<Knight> getKnights():** This is a get method for the knights attribute.
* **public void setKnights(List<Knight> knights):** This is a set method for the knights attribute.
* **public boolean levelComplete():** Returns whether the level is complete or not.
  1. **Packages**
* **java.util:**

From this package, we will use List in order to store the game pieces(knights and walls) since the number of avaible pieces will constantly change during the gameplay. We will also use this package for IO and localization of the game. We will also use timer from this package in order to implement the game timer.

* **java.fx**

We will use this package to control the input from the user(keyboard and mouse inputs) and display the output to the user. Basically, this package will be used to implement the GUI.

# Glossary & References

1. <https://docs.oracle.com/javase/8/javafx/api/toc.htm>

2. <https://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html>