**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**

**TA: Gulden Olgun**

**Table of Contents**

[**1 Introduction** 2](#_Toc532414422)

[1.1. Purpose of the System 3](#_Toc532414423)

[1.2. Design Goals 3](#_Toc532414424)

[**2 High-Level Software Architecture** 6](#_Toc532414425)

[2.1 System Decomposition 6](#_Toc532414426)

[2.2 Hardware/Software Mapping 7](#_Toc532414427)

[2.3 Data Persistent Management 8](#_Toc532414428)

[2.4 Access Control and Security 9](#_Toc532414429)

[2.5 Boundary Conditions 9](#_Toc532414430)

[**3 Subsystem Services** 11](#_Toc532414431)

[3.1 Interface Subsystem 12](#_Toc532414432)

[3.2 Game Management Subsystem 12](#_Toc532414433)

[3.3 Data Management Subsystem 13](#_Toc532414434)

[**4 Low- Level Design** 14](#_Toc532414435)

[4.1 Object Design Trade-offs 14](#_Toc532414436)

[4.2 Final Object Design 15](#_Toc532414437)

[4.3 Class Interface 16](#_Toc532414438)

[4.4 Packages 29](#_Toc532414439)

[**5 Improvement Summary** 30](#_Toc532414440)

[**6 Glossary & References** 31](#_Toc532414441)

# Introduction

Walls andWarriors, by the group “404 Name Not Found”, is a software implementation of the classical game with the same name (*Walls and Warriors* [1]). Walls and Warriors is a single player game that challenges the players’ brain power with a wide variety of challenges.

In total, the game has one game board, seven knights (three blue, four red), four walls of different predefined shapes and one blue tower. Out of these, the walls are the only object that the player is allowed to interact with. With the help of those four walls, the player has to make sure that the castle of blue warriors is safe from the red warriors. For that, the player must arrange the walls on the game board such that the blue knights are protected within the castle and the red invaders are kept out of the castle walls. Given that each challenge has only one correct solution, *Walls and Warriors* aims to challenge both children and adults.

In our implementation of the game, we aim to add some additional features to make the game more interesting and different than the classic version. The levels will now be timed and scores will be calculated based on the time taken to complete the challenge and the difficulty level of the challenge. Game experience will be pleasing as the player will be motivated with additional bonuses like coins, diamonds and trophies. The game will include different themes/maps and characters which aim to bring versatility in the user experience (in terms of view).

With the introduction of the Sandbox Mode, we aim to give power to the user and let the player create one’s own challenges. The user will now be able to specify the dimensions of the game board, choose the number of red and blue knights, and pick from a wider range of predefined walls of their own choice. Once valid, the new challenge will be able to be saved in the program and can be played anytime.

## **Purpose of the System**

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. The purpose of the game is to enable players to enhance their puzzle-solving ability by taking them on a journey from easier challenges to much more difficult ones. Players are given the incentives of ‘Coins’, ‘Diamonds’ and ‘Trophies’ to complete challenges in the shortest possible time. Through these incentives, we aim to make the process of puzzle-solving more creative and rewarding.

## **Design Goals**

A set of design goals were laid out to serve as a foundation of our design plan. Our aim was to design the game in a way that would make it easy to run, play and understand without lacking in consistency. Some of the requirements we deemed as most crucial are as follows:

* 1. **Maintainability Criteria:**

1. ***Readability of source code:*** The prospect of extending the game with more features is always expected in video games. Therefore, our aim is to write simple, clean and readable code so that it can be easily understood by us and/or other developers, in case any changes and/or additions are carried out in the future. We will try to achieve this by following coding conventions and writings comments for all pieces of code.
   1. **Reliability Criteria:** 
      1. ***Crash Handling:*** Efforts will be taken to make sure that the game does not crash unexpectedly. In cases where the game crashes, errors will be handled and the user will be notified with the appropriate message screen. All of user’s data and progress within the game will be restored once the game starts again. As the challenged are marked complete only when the challenge is solved correctly, all user progress (within the challenge) will be lost if the game crashes during gameplay. The user must start the challenge again.
      2. ***Invalid user operations:*** All operations that a user is not allowed to do at any moment in the game (e.g. buying a theme without having sufficient diamonds), will be disabled on the user-interface. In an off chance that the user is successful in accessing any of those invalid operations, the situation will be handled and the user will be presented with an appropriate error screen.
   2. **Usability Criteria:**
      1. ***Simple User Interface:*** We aim to present the user with a self-explanatory and aesthetically pleasing user-interface such that one does not face any difficulty while navigating through different menus and features. Feature that would allow the user to view game instructions within the game will be included in the game mode so that one can view the instructions at any time. Minimum number of buttons and panels will be used with appropriate labelling to achieve this goal. Also if the screen size is 1920 x 1080, main menu buttons will be 400x200 and they will glow once the mouse is hovered on them. Thus, the player will be able to browse around the menus easily.
      2. ***Easy-to-learn Gameplay:*** We aim to develop the gameplay in a manner such that 90% of the users playing *Walls and* Warriors for the first time should be able to understand the dynamics of the game within one hour, without reading the user manual. Additionally, the user will still be able to view game instructions if the one wishes.
   3. **Performance Criteria:**

***Smoothness:*** In order to enhance the user experience, it is essential that our game performs up to a specific standard. For a simple game like *Walls & Warriors*, the frame rate should be at least 30 frames per second. However, in order to make sure that user generated movements (e.g. dragging walls, rotating walls and etc.) and animations within the game are smooth and accurate, we will still put in substantial effort to try to achieve a frame rate of 60 frames per second by calling a method which would repaint the screen at least 60 times in a second.

***Response time:*** Again, in order to enhance user experience, it is essential that the user does not encounter any delay during any of one’s actions and its corresponding result on the screen. By writing a renderer, we will try to make sure that the response time of all actions carried out by the user does not exceed 500 miliseconds. Also, we will attempt to make the game as little CPU-intensive as possible so that memory issues does not affect the game’s response time.

***Storage:*** Database will be used to store players’ information, their progress within the game and the arrangement of all challenges. MongoDB[2] database will be used to make data retrieval fast and under 1 second at the worst case.

* 1. **Supportability Criteria:** 
     1. **Setup:** The game will be distributed as a single executable JAR file. The purpose is to save the user from the trouble of going through the entire process of installation by oneself.

# 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.

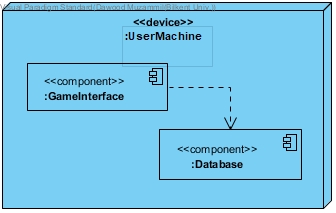
## **Hardware/Software Mapping**

Walls and Warriors is a simple game implemented in Java, and therefore it doesn’t need any extra hardware to run beside a personal computer. Therefore, the subsystems will be realized entirely by software. It can run on any computer that has the Java Runtime Environment.

Keyboard and mouse are sufficient as input devices. Mouse drag inputs are used to move the walls and drop them. Mouse click inputs are used to navigate in the menu and change the options.

The game does not need any excess computational power or memory to run. It does not render any 3D graphics or do lots of calculations. Being an offline game, it also does not need to be connected to any external server at any time.

In terms of memory, the information for the levels will be accessed locally on an instance of a *MongoDB* database. Additionally, the information of user’s progress will also be saved locally.



*Figure 1 Deployment diagram for Walls & Warriors*

## **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, players’ data and their progress within the game 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 progress will be protected and therefore will not be modifiable. The only option will be to reset the game which will clear all the progress 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 auto save 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, Walls & Warriors will not be requiring any security as the game would not require 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.

The player is the only actor. Following is the access matrix for the player.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **GUIManager** | **InputController** | **Engine** | **Wall** | **Handler** | **GameBoard** |
| **Player** | start()  initializeScreen()  setResolution()  setSoundLevel()  setFullScreen()  changeTheme() | tick() | start()  stop()  run()  tick() | rotateRight90Deg()  rotateLeft90Deg() | add(GameObject)  remove(GameObject)  clear() | tick()  addWall()  addKnight()  addTower()  saveLevel()  setDimensionX()  setDimensionY() |

## **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 a single executable JAR file (.jar) which will start the game right away. In order to run the game, the user will just need to double click or press enter on the executable file, just like one would run any other program on one’s personal computer. Once started, the user will just need to click on the *Start Game* button in order to begin playing.

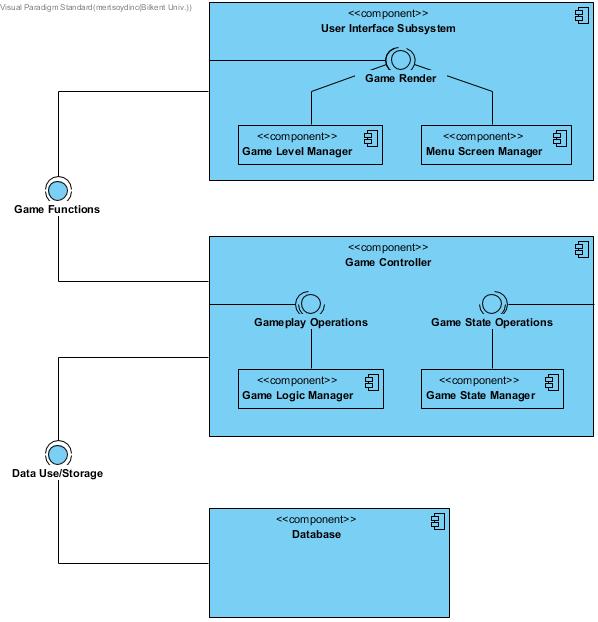
* + 1. **Termination**

The game can easily be terminated by clicking the *Quit Game* button from the main menu and the pause menu. From all other menu screens, we provide the *Back* button through which the user can navigate back to the main menu and quit the game.

* + 1. **Failure**

If, at any time, the game fails to fetch the data regarding levels from the database, the game will crash/display error message to the user. As stated above, the state of the next level is saved in the database *only* when the challenge is completed. Therefore, if the game crashes when the user is on any of the menu screens (i.e. not playing a challenge), having the auto-save functionality will be beneficial as user’s progress will already be saved automatically and will not be dependent on user remembering to save his progress. However, if the game crashes during gameplay, all progress made by the user during gameplay will be lost (for that specific challenge only). The user will have to start the challenge from the beginning when one starts the game for the next time.

# 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*. *Figure 2* below shows the three subsystems, following a detailed description of their individual components.

*Figure 2 Subsystem Decomposition*

* 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, achievements, unlocked levels) and information regarding user-made levels. This subsystem is controlled entirely by the GSM.

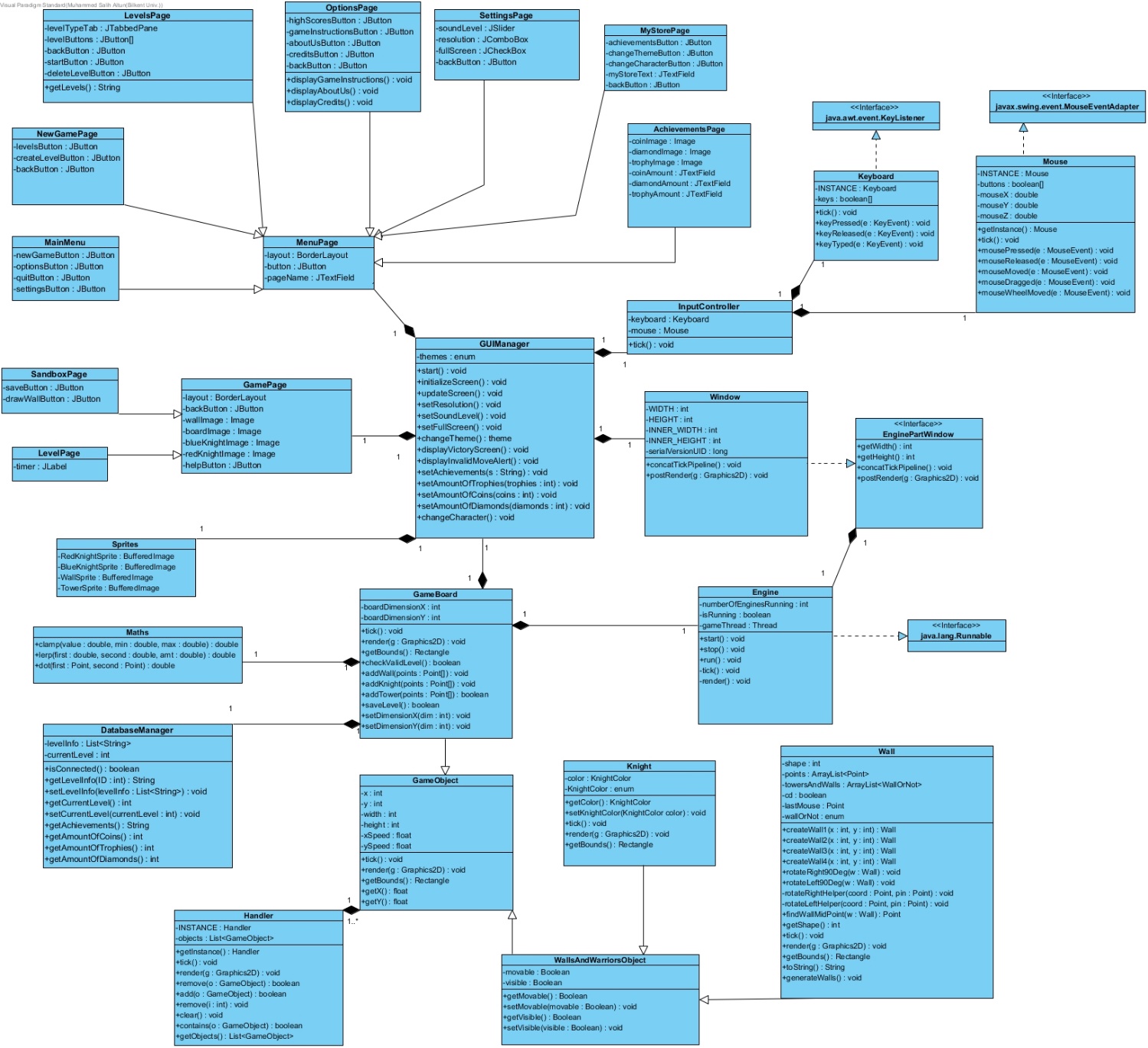
# Low- Level Design

* 1. **Object Design Trade-offs**

1. **Efficiency vs. Portability:** In our design algorithmic and time efficiency is more important than portability because the algorithms that run throughout the game should not slow the game and cause lags. In case of portability, our aim for the product is that it will be able to run with Java Virtual Machine which already runs on many platforms. However, we will not deliver a multiplatform application so the users will not be able to play our game on a phone or a tablet.
2. **Delivery time vs. Functionality:** In case thedevelopment process runs behind schedule, we may decide to leave out some of the features decided earlier and present the software on the date of delivery with less, but working features.
3. **Usability vs. Functionality:** The main purpose of Walls and Warriors Desktop is to encourage children to cope with puzzle games and create an environment to boost their interest to solving puzzles. Therefore, our design prefers usability over functionality so that we will have less functionality but simpler user interfaces in one screenview of the product.
4. **Cost vs. Robustness:** The cost of Walls and Warriors for the developers is only time. However, the robustness is significant for the users. This is why we prefer a better user experience over our cost of time. To make our game more robust, we tested our game with other colleagues to give us feedback.

[ this space is left blank intentionally ]

* 1. **Final Object Design**



*Figure 3 Final object design*

For better readability, view the image from the following link: https://imgur.com/a/LX5Z2Gb

[ this space is left blank intentionally ]

* 1. **Class Interface**
* **Class GameObject**:

There are 2 types of **GameObject**, walls and knights, which are both subclasses of **WallsAndWarriorsObject**.

* + **Attributes:**
* **private int x**: x coordinate of this object
* **private int y**: y coordinate of this object
* **private int width**: Width of this GameObject
* **private int height:** Height of this GameObject
* **private float xSpeed:** Speed of this GameObject when moving, with respect to x axis.
* **private float ySpeed:** Speed of this GameObject when moving, with respect to y axis.
  + **Operations:**
* **public tick():** Updates the state of this GameObject.
* **public render():** Draws this GameObject on the screen.
* **Class Wall:**

This class inherits **WallsAndWarriorsObject**.

* + **Attributes:**
* **private int shape:** The shape of the wall stored as an int value.
* **private ArrayList<Point> points:** The points of this wall object held in an ArrayList structure.
* **Private ArrayList<WallOrNot> towersAndWalls:** The **enum** type WallOrNot stands for towers or knights. This data structure holds towers and knights.
* **private Point lastMouse:** The point mouse clicked last.
* **private enum WallOrNot:** Enumerates walls and knights.
  + **Operations:**
* **public Wall createWall1(int x, int y):** Creates a wall of first type in the specified (x,y) position.
* **public Wall createWall2(int x, int y):** Creates a wall of second type in the specified (x,y) position.
* **public Wall createWall3(int x, int y):** Creates a wall of third type in the specified (x,y) position.
* **public Wall createWall4(int x, int y):** Creates a wall of fourth type in the specified (x,y) position.
* **public void tick():** Updates the state of this **Wall.**
* **public void rotateRight90Deg(Wall w):** Rotates the specified wall 90 degrees to the right.
* **public void rotateLeft90Deg(Wall w):** Rotates the specified wall 90 degrees to the left.
* **public void rotateRightHelper(Point coord, Point pin):** Helper method for **rotateRight90Deg.**
* **public void rotateLeftHelper(Point coord, Point pin):** Helper method for **rotateLeft90Deg.**
* **public Point findWallMidPoint(Wall w):** Finds the midpoint of the wall and returns it.
* **public Rectangle getShape():** Getter method for the **shape** attribute.
* **public void render():** Draw this wall.
* **public Rectangle getBounds():** Returns the **Rectangle** that bounds this wall.
* **public String toString():** Specification of how to convert a wall to String, overridden method from the **Object** class.
* **public void generateWalls():** Generates all 4 walls of the game.
* **Class WallsAndWarriorsObject:**
  + **Attributes:**
* **private Boolean movable:** This attribute specifies if this object is movable or not.
* **private Boolean visible:** This attribute specifies if this object is visible or not.
  + **Operations:**
* **public Boolean getMovable:** Gets the movable attribute of this object.
* **public voidsetMovable(Boolean movable):** Sets the movable attribute of this object.
* **public Boolean getVisible():** Gets the visible attribute of this object.
* **public void setVisible(Boolean visible):** Sets the visible attribute of this object.
* **Class Knight:**

This class inherits **WallsAndWarriorsObject** class.

* + **Attributes:**
* **private KnightColor color:** The color of the knight is stored as an **enum**.
* **private enum KnightColor:** Enumeration of the colors of knights, blue and red.
  + **Operations:**
* **public KnightColor getColor ():** Gets the color of the knight as a **KnightColor**.
* **public void setKnightColor (KnightColor color):** Sets the color of the knight
* **public void tick():** Updates the state of this **Knight** object.
* **public void render():** Draws this **Knight** object.
* **public Rectangle getBounds():** Returns the **Rectangle** that bounds this **Knight** object.
* **Class Handler:**
  + **Attributes:**
* **private Handler lNSTANCE:** Holds an instance of this class.
* **private List<GameObject> objects:** A list of game objects currently in the game.
  + **Operations:**
* **public Handler getInstance():** Returns the current **INSTANCE**.
* **public void tick():** Updates the state of this instance of **Handler**.
* **public void render():** Calls the render method on all the game objects currently in the game.
* **public void remove(GameObject o):** Removes this game object from the list.
* **public void add(GameObject o):** Adds the game object to the list.
* **public void remove(int i):** Removes the i’th game object from the list.
* **public void clear():** Clears the list of game objects.
* **public Boolean contains(GameObject o):** Checks if the given game object is in the list and returns the result as Boolean.

**pPublic List<GameObject> getObjects():** Returns the list of game objects.

* **Class Engine:**

This class implements Runnable interface.

* + **Attributes:**
* **private int numberOfEnginesRunning:** The number of game engines currently running.
* **private Boolean isRunning:** Whether this instance of Engine is running or not.
* **private Thread gameThread:** The main thread of the game.
  + **Operations:**
* **public void start():** Starts this instance of Engine.
* **public void stop():** Stops this instance of Engine.
* **public void run():** Runs this instance of Engine.
* **public void tick():** Updates the state of this instance of Engine.
* **public void render():** Tells all the game objects to draw themselves.
* **Class GameBoard:**

This class represents the game board.

* + **Attributes:**
* **private int boardDimensionX:** The horizontal size of the board
* **private int boardDimensionY:** The vertical size of the board
  + **Operations:**
* **public void tick():** Updates the state of the board.
* **public void render():** Draws the board.
* **public Rectangle getBounds():** Returns the current bounds of the board as a rectangle.
* **public Boolean checkValidLevel():** checks whether the level could be completed and returns boolean value
* **public void addWall(Point[] wallPoints):** add a wall to the level by a point array and return boolean value for its success
* **public void addKnight(Point knightPoint):** add a knight to the level by a point and return boolean value for its success
* **public void addTower(Point[] wallPoints):** add a tower to the level by a point array and return boolean value for its success
* **public void setBoardDimensionX(int boardDimensionX):** sets the horizontal size of the board
* **public void setBoardDimensionY(int boardDimensionX):** sets the vertical size of the board
* **public Boolean 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 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 getAchievements(String username):** This method will be used to get the player’s achievements from the database.
* **public int getCurrentLevel():** Get method for the **currentLevel** attribute.
* **public void setCurrentLevel(int currentLevel):** Set method for the **currentLevel** attribute.
* **public int getAmountOfCoins():** This method receives the amount of coins for the current user.
* **public int getAmountOfTrophies():** This method receives the amount of trophies for the current user.
* **public int getAmountOfDiamonds():** This method receives the amount of diamonds for the current user.
* **Class MenuPage**
* **Attributes:**
  + - **private BorderLayout 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 JButton button:** A button will be used in all menu pages.
    - **private JTextField pageName:** JTextField will be used in all menus that have text entries.
* **Class MainMenu**

MainMenu is a MenuPage.

* **Attributes:** 
  + - **private JButton newGameButton:** This button is for the user to start a new game.
    - **private JButton optionsButton:** This button is used for accessing the options menu which is described in the “Class OptionsPage” section.
    - **private JButton quitButton:** This button is used to quit the game.
    - **private JButton 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 JButton levelsButton:** This button is used to view the levels.
    - **private JButton createLevelButton:** This button is used to access the sandbox menu.
    - **private JButton backButton:** This button is used to go to the previous screen.
* **Class LevelsPage**

LevelsPage is a MenuPage.

* **Attributes:**
* **private JTabbedPane levelTypeTab:** This attribute is in order to manage between a group of tabs.
* **private JButton[] 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 JButton backButton:** This button is used to go to the previous page.
* **private JButton startButton:** This button is used to start a level.
* **private JButton deleteLevelButton:** This button is used to delete a user- created level.
* **Operations:**
* **public String getLevels():** This method returns the levels as a String.
* **Class OptionsPage**

OptionsPage is a MenuPage.

* **Attributes**:
* **private JButton gameInstructionsButton:** This button is used to display the game instructions.
* **private JButton aboutUs:** This button is used to display the information about the group 404: Name Not Found.
* **private JButton creditsButton:** This button is used to display the credits.
* **private JButton backButton:** This button is used to go to the previous page.
* **Operations:**
* **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 JSlider soundLevel:** This button is used to adjust the sound level in the game.
* **private JComboBox resolution:** This button is used to change the resolution from low, medium or high.
* **private JCheckBox fullScreen:** This button is used to play the game in full screen.
* **private JButton backButton:** This button is used to go to the previous page.
* **Class GUIManager**
* **Attributes:**
* **private enum themes:** Enumeration of the themes for the game.
* **Operations:**
* **public void start():** Initializes the first screen of the game which is the main menu.
* **public void initializeScreen():** This method sets up the initial properties of the current screen.
* **public void updateScreen():**This method updates the screen when there is going to be an alteration.
  + - **public void setResolution():** Used to change the resolution when the user selects a different resolution from the settings page.
    - **public void setSoundLevel():** Used to change the sound level when the user changes it from the settings page.
    - **public void changeTheme():** Used to change the theme when the user sets a different theme from the settings page.
    - **public void displayVictoryScreen():** This method tells the game page to display a victory screen when the player completes the level.
    - **public void displayInvalidMoveAlert():** This method tells the game page to display an alert to the user that their move was invalid.
    - **public void setAchievements():** This method changes the achievements for the game.
    - **public void setAmountOfTrophies():** This method changes the amount of trophies for the user when a new trophy is earned.
    - **public void setAmountOfCoins():** This method notifies the game and the database that the coin amount as changed.
    - **public void setAmountOfDiamond():** This method notifies the game and the database that the diamond amount as changed.
    - **public void changeCharacter():** This method changes the character for the user.
* **Class InputController**
* **Attributes:**
* **private Keyboard keyboard**: An instance of keyboard class which can handle key inputs.
* **private Mouse mouse**: An instance of mouse class which can handle mouse button inputs.
* **Operations:**
* **public void tick()**: Update the state of keyboard and mouse, invoking the tick method of those classes.
* **Class Keyboard**

This class deals with keyboard inputs and implements the KeyListener interface from the **java.awt.event** package.

* **Attributes:**
* **private Keyboard INSTANCE:** An instance of this class.
* **private Boolean[] keys:** An array which holds references to keys on the keyboard.
* **Operations:**
* **public void tick():** Updates the state of the array keys.
* **public void keyPressed(KeyEvent e):** Handles a key press from the keyboard.
* **public void keyReleased(KeyEvent e):** Handles a key release from the keyboard.
* **public void keyTyped(KeyEvent e):** Handles a key typed event from the keyboard.
* **Class Mouse**

This class deals with mouse inputs and implements the MouseEventAdapter interface from the **javax.swing.event** package.

* **Attributes:**
* **private Mouse INSTANCE:** An instance of this class.
* **private Boolean[] buttons:** An array which holds references to buttons of the mouse.
* **private double mouseX:** Holds the current x coordinate of the mouse on the screen.
* **private double mouseY:** Holds the current y coordinate of the mouse on the screen.
* **private double mouseZ:** Holds the current z coordinate of the mouse on the screen.
* **Operations:**
* **public void tick():** Updates the state of the mouse and button array.
* **public Mouse getInstance():** Returns the instance of this mouse.
* **public void mousePressed(MouseEvent e):** Handles a mouse button press.
* **public void mouseReleased(MouseEvent e):** Handles a button release from the mouse.
* **public void mouseMoved(MouseEvent e):** Handles a movement of the mouse.
* **public void mouseDrag(MouseEvent e):** Handles a drag of the mouse.
* **public void mouseWheelMoved(MouseEvent e):** Handles a movement of the mouse wheel.
  1. **Packages**
* **java.util**[3]

From this package, we will use List in order to store the game pieces (knights and walls) since the number of available 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.

* **javax.swing**[4]

This package will be used to implement the GUI.

* **java.awt**[5]

This package will be used to help with GUI and event handling.

[ this space is left blank intentionally ]

# Improvement Summary

According to the feedback we received on the first iteration, a number of improvements were made for this iteration of design report. They are as follows:

* + - 1. An introduction was added which explained the game generally.
      2. *Purpose of the System* was refined in accordance to the new features of the game.
      3. *Design goals* were written again so that they were directly driven from the non-functional requirements.
      4. *Design goals trade-offs* were added.
      5. Refined mapping of hardware and software by mentioning all components of hardware and software involved in the system. Demonstrated it using a deployment diagram.
      6. Final object design (class diagram) was updated to represent new functionalities of the game (e.g. *My Store* page).
      7. Packages were updated. We are now using *javax.swing* instead of *JavaFX*.

[ this space is left blank intentionally ]

# Glossary & References

* + - 1. <https://www.smartgames.eu/uk/one-player-games/walls-warriors>
      2. [www.mongodb.org](http://www.mongodb.org)
      3. <https://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html>
      4. <https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html>
      5. <https://docs.oracle.com/javase/7/docs/api/java/awt/package-summary.html>