

Bilkent Universtity

Department of Computer Engineering

CS 319 Term Project

Group 1D  
Wars & Warriors  
Iteration 1

System Design Report

Project Group Members:

Samet DEMIR  
Ahmet MALAL  
Mahammad SHIRINOV  
İbrahim MAMMADOV  
Huseyn ALLAHYAROV  

Supervisor: Eray TUZUN

Table of Contents

[1 Introduction 4](#_Toc529479835)

[1.1 Purpose of the system 4](#_Toc529479836)

[1.2 Design Goals 4](#_Toc529479837)

[1.2.1 Criteria End User Criteria 4](#_Toc529479838)

[2 System Architecture 5](#_Toc529479839)

[2.1 Decomposition 5](#_Toc529479840)

[2.2 Hardware/Software mapping 8](#_Toc529479841)

[2.3. Persistent Data Management 8](#_Toc529479842)

[2.4. Access Control and Security 8](#_Toc529479843)

[2.5. Boundary Conditions 8](#_Toc529479844)

[3 Subsystem Services 9](#_Toc529479845)

[3.1 User Interface Subsystem 9](#_Toc529479846)

[3.1.1 MainFrame Class 9](#_Toc529479847)

[3.1.2 MenuView 11](#_Toc529479848)

[3.1.3 PauseMenu 12](#_Toc529479849)

[3.1.4 SettingsMenu 13](#_Toc529479850)

[3.1.5 GameView 14](#_Toc529479851)

[3.1.6 MapView 15](#_Toc529479852)

[3.2 Game Management Subsystem 16](#_Toc529479853)

[3.2.1 File Manager: 16](#_Toc529479854)

[3.2.2 Sound Manager: 17](#_Toc529479855)

[3.2.3 Game Manager: 17](#_Toc529479856)

[3.2.4 Map Manager: 19](#_Toc529479857)

[3.3 Game Object Subsystem 20](#_Toc529479858)

[3.3.1 Map 21](#_Toc529479859)

[3.3.2 Forest 21](#_Toc529479860)

[3.3.3 Land 22](#_Toc529479861)

[3.3.4 Lake 22](#_Toc529479862)

[3.3.5 GameObject 23](#_Toc529479863)

[3.3.6 Castle 24](#_Toc529479864)

[3.3.7 WallOrChain 24](#_Toc529479865)

[3.3.8 Wall 26](#_Toc529479866)

[3.3.9 Chain 26](#_Toc529479867)

[3.3.10 Warrior 26](#_Toc529479868)

[3.3.11 Ally 27](#_Toc529479869)

[3.3.12 Enemy 27](#_Toc529479870)

[3.3.13 MovableEnemy 28](#_Toc529479871)

[3.3.14 StableEnemy 29](#_Toc529479872)

[3.3.15 EnemyArmada 29](#_Toc529479873)

[3.3.16 EnemyLandForces 29](#_Toc529479874)

[3.3.17 AllyArmada 30](#_Toc529479875)

[3.3.18 AllyLandForces 30](#_Toc529479876)

[3.3.19 Coor 30](#_Toc529479877)

[4 Low-level Design 32](#_Toc529479878)

[4.1 Object Design Trade-Offs 32](#_Toc529479879)

[4.1.1 Initialization Time vs Maintainability: 32](#_Toc529479880)

[4.1.2 Performance vs Graphics: 32](#_Toc529479881)

[4.2 Packages  32](#_Toc529479882)

[4.2.1 org.newdawn.slick.command 32](#_Toc529479883)

[4.2.2 org.newdawn.slick.loading 32](#_Toc529479884)

[4.2.3 org.newdawn.slick.state 32](#_Toc529479885)

[4.3 Class Interfaces 32](#_Toc529479886)

[4.3.1 MouseListener 32](#_Toc529479887)

[4.3.2 KeyListener 32](#_Toc529479888)

# Introduction

## Purpose of the system

Walls and Warriors is a board game which consists of 2 groups of soldiers – red and blue – playing a tactical game to win over the other. The main purpose of the game is to encapsulate all the blue soldiers with the walls, thus building a castle and leaving all the red soldiers outside. The game is being implemented for the desktop environment in 2D graphics and including additional features not originally included in the board game. Game consists of a UI with gameplay guidelines and settings menu and a window giving information about the creators. This implementation of the game comes with new features to increase the player affinity and game complexity which contributes to the competitiveness style of the overall game.

## Design Goals

### Criteria

#### End User Criteria

##### Usability

The game provides a simple and user-friendly user interface for a fluid experience. Game comes with the guidelines about how to play the game, explanation for the game objectives and goals. Game is only using the mouse and keyboard for the input and the display for the output and visual elements.

##### Efficiency

To minimize the space allocated for the game the game graphics should not be demanding. The Image files will have minimal size to not to waste any resource. Bitmap files may be used for this reason. The execution order of the game will be affecting the gameplay and smoothness, thus, map should be loaded before the game and game should be loaded before the full initialization of the game and menus.

##### Reliability

The game should be bug-free before the delivery to avoid any disasters. The game should be tested heavily to ensure the stability and reliability of the game and avoid any unexpected problems about the gameplay.

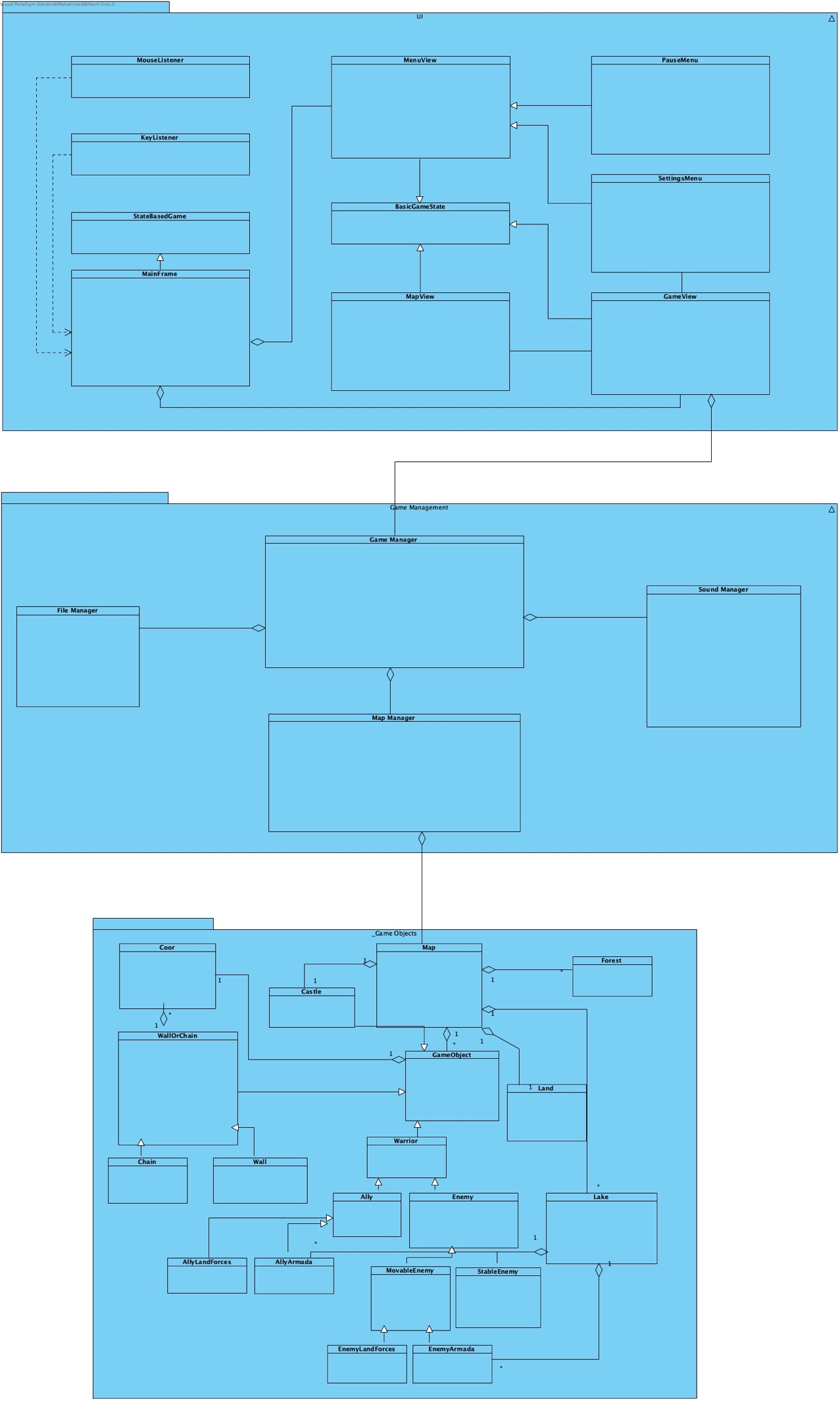
##### Modifiability

The game is a project based on the means of Object-Oriented Design pattern. OOP will ensure that the game will be easy to update and modify. Also, the main approach of the development of the project is MVC (Model – View - Controller) which ensures that a new change won’t affect the system drastically or malfunction it.

# System Architecture

## Decomposition

We decided to use MVC pattern to ease the development and planning process. By dividing the process into Models, Views and Controllers we intend to develop a system that is maintainable, sustainable and modifiable. This approach will ensure the ease of development and modifications in the system which should not affect the whole system. These divisions will also help us for the workload and development organization among the members of the group.  
 First division is the Models division. We will have several models such as map, soldiers, walls, map’s immobile objects etc. The Models will be easy to update, fix or replace which is an example of the visionary approach that will help for the further development of the game. Any new features of the maps, soldiers or walls can be modified at any given time without affecting other participating elements of the overall system. OOP pattern also comes to our help in these situations which led us to have the models as objects to be modifiable and dynamic, but not static.  
 Second division is the Views division. The Views consist of Menus, different states of the game which a user may happen to be in. In-menu state is basically MenuView state which is displaying the menu options to the user. Another view, PauseMenu is a child of the main MenuView class which is the paused state view that will display the pause menu and its associating options to the user. Another View is the settings view which will allow the user to modify the pre-defined default game settings for their own preferences. Having MVC and views as the states led us to use the slick library’s StateBasedGame class to ease the development process. This class provides helpful functionalities to be used. StateBasedGame is the main class type that the main game views will be inheriting and using for the gameplay as it is consisting of the described states.  
 The third and last division is the Controllers division. Controllers are consisting of the Manager classes which will be interacting with the UI instances and Models to update the game, calculate the game state and wall placements and communicate with other divisions of the project to update and act accordingly. Controllers are behaving as the back-end division of the game which is responsible for non-visual functions like calculating, updating variables and key instances in the game and modifying the visual properties accordingly. Listeners will be main input handlers and signals for such updates which will be happening on the front-end of the program, UI. The approaches we used will be the grounds for the reverse engineering and visionary approach and provide modifiability.

Subsystem Decomposition Diagram 

## 2.2 Hardware/Software mapping

The game is implemented by the usage of several Java packages and libraries such as slick2D, awt and respective classes for the I/O signals. The reason behind the selection of the Java language was its wide usage and compatibility with the all widespread OS variations. The game will need basic Java properties such as JRE, JVM to be able to run the executable .jar file to start the game. Keyboard, Mouse and Screen will be the input and output devices for the user and game interaction. Players will be dragging and dropping the wall objects on the map which will change the game state internally (in-code) and externally (visual) with the help of the screen. The keyboard can also be used for the basic functionalities of the game such as going to the menu and pausing the game. External audio output device like a speaker or headphones will be needed for the recommended gameplay and hearing the sounds.

## 2.3. Persistent Data Management

The game won’t need any external files to run with like a database, external text files or storage systems. The game will utilize its storage and use the predefined objects as the player plays the game, so game’s memory allocation will remain static and fixed. Hard-disk will be the main source of the memory that game will be installed and played on.

## 2.4. Access Control and Security

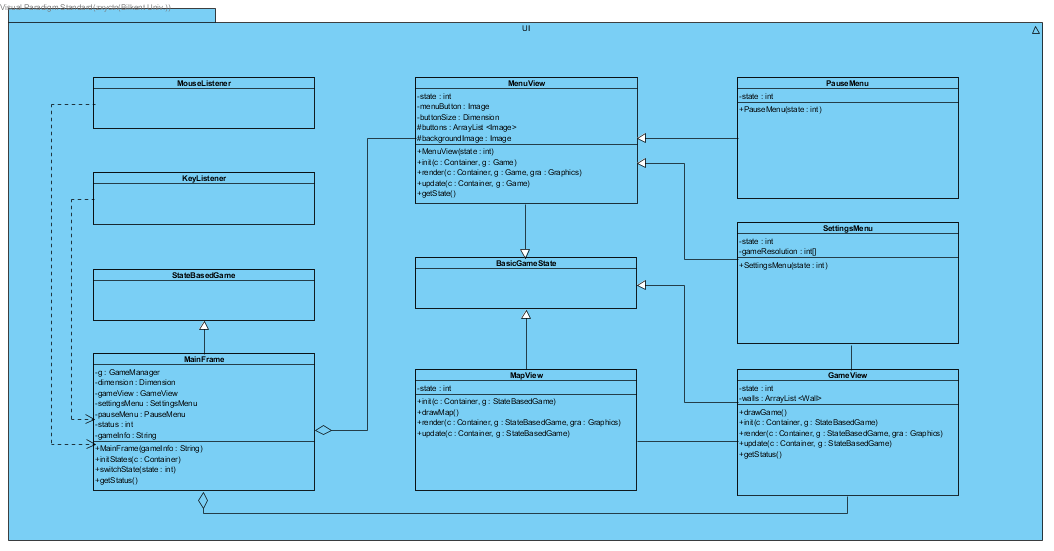
The game does not need any complex security system or firewalls as its basic structure is the point of consideration. The game does not use any authorization or registration system which will avoid the security issues that may happen because of it. The main goal with the decision on these implementations is the stability of the game and secure structure of it. The game classes and their respective attributes and operations will be encapsulated accordingly.

## 2.5. Boundary Conditions

The game will be easily terminated by the help of “Exit” button on the menu or X button of the window for the safe termination. Unsafe termination will not be affecting any fundamental game structure but the all saved temporary memory instances will be wiped in such situations. Delivered game will be a .jar file which needs the user to have the JRE and JVM installed for the execution. Game’s allocated memory size should not change over the lifetime. It should remain static unless an update or a modification takes place.

# Subsystem Services

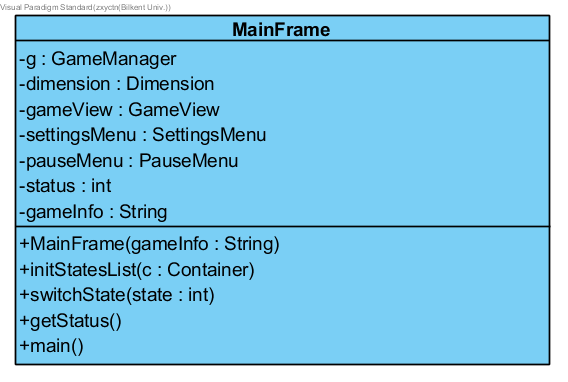
## User Interface Subsystem



The User Interface (UI) will be in charge of the visual game elements. It will be dealing with View part of the MVC design pattern that is used in the project. It will be the bridge between user input and game output. User input will be handled and send to the controller to be used.

Class definitions:

### MainFrame Class



Inherits: StateBasedGame (slick)

Interfaces: KeyListener, MouseListener

Functionality: The main game object to start and initialize the game. This object is in charge of dealing with different states, managing them accordingly and contains main method which initializes the game. Class is also the main link between controllers and views.

#### Attributes:

-g: GameManager

GameManager is one of the controllers as well as a class of the controller package. MainFrame will update the game and view according to the controller’s directions.

-dimension: Dimension

Dimension class is used for the game resolution and consists of width and height. It is a class of Java’s awt library.

-gameView: GameView

The object is one of the states in the game. MainFrame will be displaying this state when player is in the game.

-settingsMenu: SettingsMenu

The object is one of the states in the game. MainFrame will be displaying this state while the player is in the settings menu.

-pauseMenu: PauseMenu

The object is one of the states in the game. MainFrame will be displaying this state whle game is paused and the pause menu is activated.

-status: int

This variable will be used to determine the game’s current state which will be updated whenever the state has changed.

-gameInfo: String

The String variable will be used to display simple game info on the top of the frame as a text label. It will be displaying the game name, level and the state.

#### Methods:

+MainFrame (gameInfo: String)

It is the constructor for the class which will initialize a game with the given name, gameInfo.

+InitStatesList(c: Container)

This method is an inherited and overridden method of StateBasedGame Class which initializes the game states and the game with the initial state. It will call all the states’ init methods.

+switchState(state: int)

This method will be invoked to switch the game state to the indicated one.

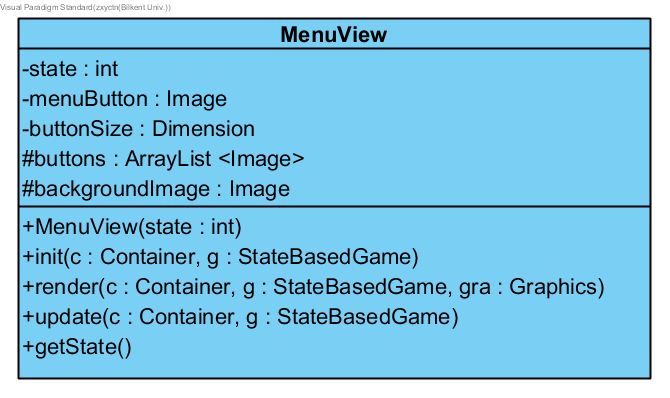
+getStatus()

This is a getter method for the private attribute status which is in charge of the game status.

+main()

Main method will be initializing the game and invoking the appropriate methods.

### MenuView



Inherits: StateBasedGame (slick)

#### Attributes:

-state: int

This variable is used to determine the state of the class.

-menuButton: Image

The button to be pressed to invoke the state switch to the menu.

-buttonSize: Dimension

A variable to be used to determine the button dimensions and resize accordingly.

#buttons: ArrayList <Image>

An array of buttons to be placed on the menu view.

#backgroundImage: Image

A background image for the view to be used.

#### Methods:

+MenuView (state: int)

Constructor method which will initiate the View with the state specified.

+init (c: Container, g: StateBasedGame)

Initializer method for the view and the menu state.

+render (c: Container, g: StateBasedGame, gra: Graphics)

This method will be drawing all the Menu instances to be placed.

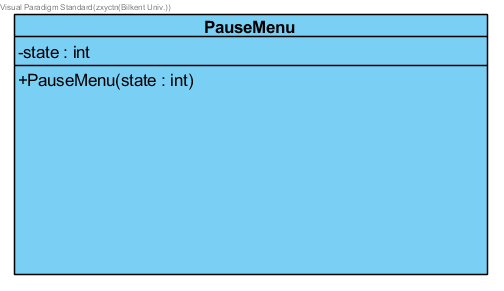
+update (c: Container, g: StateBasedGame)

This method will be updating the game information and states and will render and update the game accordingly.

+getState()

A getter function for the state variable.

### PauseMenu



Inherits: MenuView

#### Attributes:

-state: int

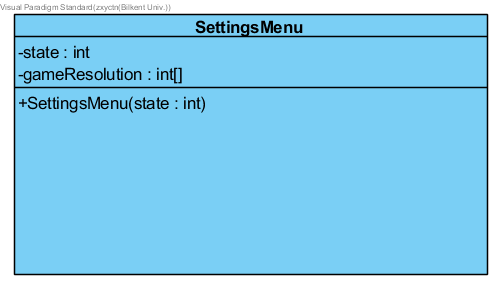
This variable is used to determine the state of the class.

#### Methods:

+PauseMenu (state: int)

A constructor method for the class to be initialized with the specified state.

### SettingsMenu



Inherits: MenuView

#### Attributes:

-state: int

This variable is used to determine the state of the class.

-gameResolution: int[]

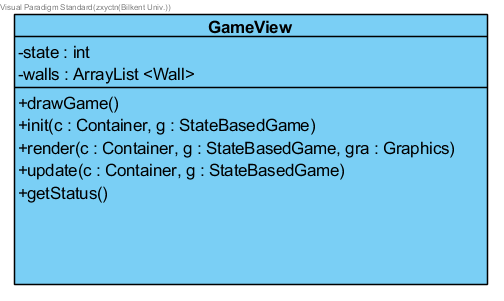
An integer array of variables which will be used to decide game’s screen resolution.

#### Methods:

+SettingsMenu (state: int)

A constructor method for the class to be initialized with the specified state.

### GameView



Inherits: BasicGameState

#### Attributes:

-state: int

This variable is used to determine the state of the class.

-walls: ArrayList <Wall>

An array of the Wall objects indicating available walls.

#### Methods:

+drawGame ()

A constructor method for the class to be initialized with the specified state.

+init (c: Container, g: StateBasedGame)

Initializer method for the view and the menu state.

+render (c: Container, g: StateBasedGame, gra: Graphics)

This method will be drawing all the Menu instances to be placed.

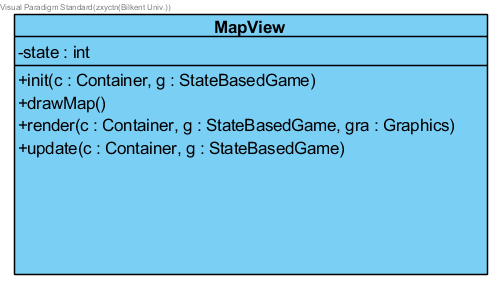
+update (c: Container, g: StateBasedGame)

This method will be updating the game information and states and will render and update the game accordingly.

+getState()

A getter function for the state variable.

### MapView



Inherits: BasicGameState

#### Attributes:

-state: int

This variable is used to determine the state of the class.

#### Methods:

+init (c: Container, g: StateBasedGame)

Initializer method for the view and the menu state.

+drawMap ()

A constructor method for the class to be initialized with the specified state.

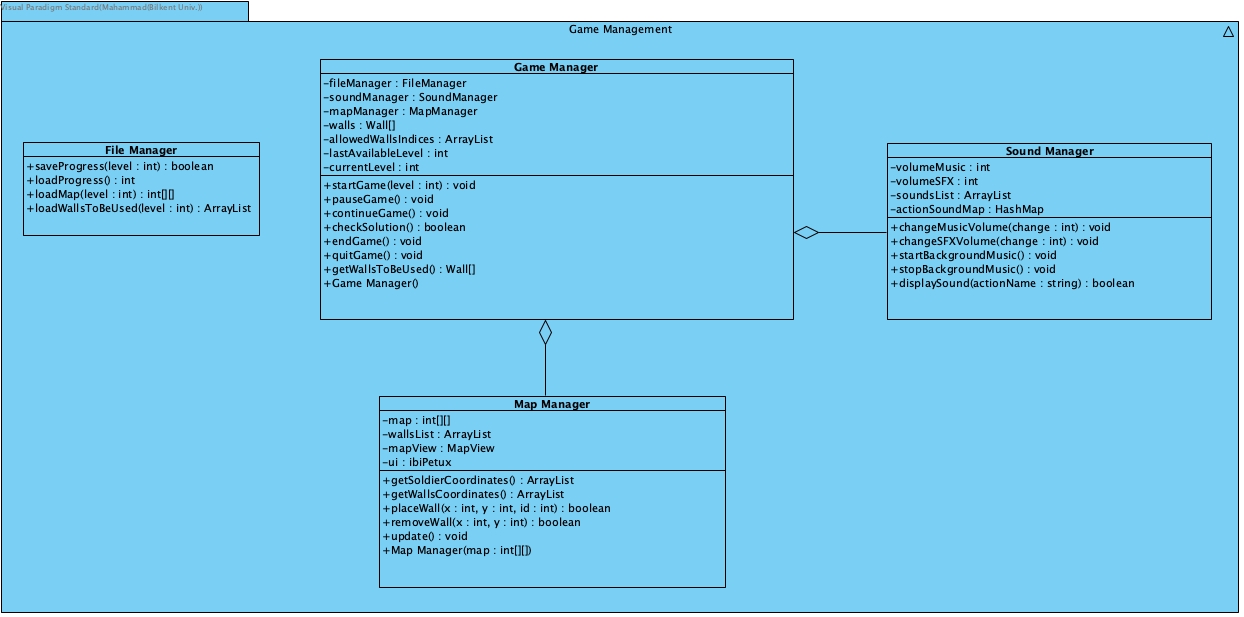
+render (c: Container, g: StateBasedGame, gra: Graphics)

This method will be drawing all the Menu instances to be placed.

+update (c: Container, g: StateBasedGame)

This method will be updating the game information and states and will render and update the game accordingly.

## Game Management Subsystem



### File Manager:

#### Methods:

+ saveProgress(level : int) : boolean

saves the highest available level after each successful level pass.

+ loadProgress() : int

used to load the progress at game start up. Returns number of levels that has been completed.

+ loadMap(level : int) : int [] []

used to load the map of the level specified as a parameter by getting the map cells in a 2D array.

+ loadWallsToBeUsed(level : int) : ArrayList

loads the wall shapes used for the solution of the specified level.

### Sound Manager:

#### Attributes:

- volumeMusic : int

the volume of the background music.

- volumeSFX : int

the volume of game sounds (action sounds, soldiers sounds and etc.

- soundsList : ArrayList

list of all sounds (including background music) in the game.

- actionSoundMap : HashMap

mapping of game actions to their corresponding sounds.

#### Methods:

+ changeMusicVolume(change : int) : void

changes the music volume by the specified (positive or negative) amount.

+ changeSFXVolume(change : int) : void

changes the sounds volume by the specified (positive or negative) amount.

+ strartBackgroundMusic() : void

starts playing background music.

+ stopBackgroundMusic() : void

stops playing background music.

+displaySound(actionName : string) : boolean

plays the appropriate sound(s) for the particular action (found in actionSoundMap and soundsList).

### Game Manager:

#### Attributes:

- fileManager : fileManager

instance of FileManager class.

- soundManager: soundManager

instance of SoundManager class.

- mapManager : mapManager

instance of MapManager class.

- currentLevel : int

the current level being played.

- lastAvailabelLevel : int

the latest level unlocked and available for playing.

- walls : Wall []

list of all the walls in the game created as instances of Wall class.

- allowedWallsIndices : ArrayList

list of walls which are going to be used at current level.

#### Methods:

+GameManager()

calls strartBackgroundMusic(), loadProgress() and specifies lastAvailabelLevel value.

+startGame(level : int) : void

initializes the game in the specified level. Loads the map by loadMap(level : int) method, gets the 2-D array and initializes the mapManager attribute by passing that array to MapManeger’s constructor , initializes allowedWallsIndices by calling loadWallsToBeUsed(level : int) method and specifies the currentLevel.

+ pauseGame() : void

pauses the game.

+ continueGame() : void

continues the game.

+ checkSolution() : boolean

checks whether the the level is solved (the current position of walls on the map is a valid solution) or not. If the level is solved, the method notifies saveProgress(level : int) method which saves the current progress and calls endGame(), method which ends the level.

+ endGame() : void

finishes the level.

+ quitGame() : void

stops all processes and quits the whole game.

+ getsWallsToBeUsed() : Wall []

returns the array of the walls allowed to be used at current level by using allowedWallsIndices ArrayList.

### Map Manager:

#### Attributes:

- map : int [] []

skeleton of the map, with each map cell given specified as some number in the 2D array.

- wallsList : Arraylist

list of walls that are put on the map.

#### Methods:

+MapManager(map : int [] [])

initialized the MapManager by specifying the map value as a passed parameter.

+ getSoldierCoordinates() : ArrayList

return the list of coordinates of all the soldiers on the map.

+ getWallCoordinates() : ArrayList

return the list of coordinates of all the walls on the map.

+ placeWall(x : int, y : int, id : int) : boolean

checks if the placement of the wall (specified by its id) to the specified coordinates is valid or not, and calls updateUI() method and updates wallList method if it is. The function is invoked when user places a wall on the map.

+ removeWall(x : int, y : int) : boolean

removes the wall at the specified coordinates; calls updateUI() method and updates wallList method. The function is invoked when user

+ update() : void

updates the logical part of the map.

## Game Object Subsystem

Game Objets Subsystem comprises 19 classes and is responsible for communicating with the controller class. This class is equivalent to model class in MVC system. Model classes will not modify itself. It will be modified by the managers. Basically, the main classes our model system are composed of Map including the GameObject instances, a land, a forest and many forest objects. Map class can be thought of as a Level class which has all information of the level such as the number of enemy, ally warriors, walls, chains, etc. and the positions of all the objects.

### Map

This is the class that will have all the information of the level.

#### Attribues

-int MapID

id number of map that we use

-GameObject[][] gameObjectList

keeps the whole objects in the map according to their positions

-Warrior[] warriorArr

keeps the warriors

### Forest

It is simply an obstacle in the game that will restrict the user. The user will not be able to put wall on it.

#### Methods

+void draw(Graphics g)

takes a Graphics object and draws the image of itself on a specific place

### Land

It is the Land class that will include the land forces of enemy and ally

#### Attributes

-boolean[][] verLines

In our game board, there are vertical line segments and walls are placed on these line segments. verLines keeps their availability. For instance, if a line is not a suitable line for a wall object, which means it is either inside a lake or occupied by forest, castle or another wall, the index that corresponds to that vertical line segment is true, which means that lines is already occupied.

-boolean[][] horLines

We also have horizontal line segments. We keep their availability in a double boolean array. For instance, if a line is not a suitable line for wall object, the index that corresponds to that horizontal line segment is true, which means it is either inside a lake or occupied by forest, castle or another wall, which means that lines is already occupied.

#### Methods

+void draw(Graphics g)

takes a Graphics object and draws the image of itself on a specific place

### Lake

It is the Lake class that will include the armada of enemy and ally forces

#### Attributes

-boolean[][] verLines

In our game board, there are vertical line segments and chains are placed on these line segments. verLines keeps their availability. For instance, if a line is not a suitable line for a chain object, which means it is either inside a lake or occupied by forest, castle or another chain, the index that corresponds to that vertical line segment is true, which means that lines is already occupied.

-boolean[][] horLines

We also have horizontal line segments. We keep their availability in a double boolean array. For instance, if a line is not a suitable line for chain object, the index that corresponds to that horizontal line segment is true, which means it is either inside a lake or occupied by forest, castle or another chain, which means that lines is already occupied.

#### Methods

+void draw(Graphics g)

takes a Graphics object and draws the image of itself on a specific place

### GameObject

This is the upper class of Warrior, WallOrChain and Castle classes.

#### Attributes

-Image img

image of the object

#### Methods

-int getX()

returns the x component of the coordinate

-int getY()

returns the y component of the coordinate

### Castle

This is the Castle class around which the user should gather all the ally forces.

#### Attribues

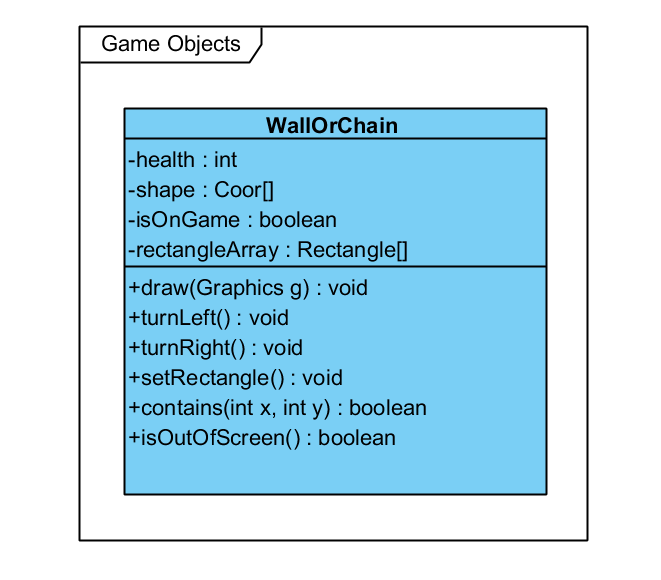
-secondPointDirection int

Castle occupies two units in the map, coordinates of main unit is given by parent class GameObject and second unit coordinates is assigned with this property. If secondPointDirection is 1, it means that caste consists of main unit and north unit of main unit. If it is 2, castle consists of main unit and east unit of main unit. 3 for south, 4 for west.

#### Methods

+void draw(Graphics g)

draws the castle that occupies two units in the map.



### WallOrChain

This is the upper class of Wall and Chain classes.

#### Attributes

-int health

shows the health of wall or chain

-Coor[] shape

includes the coordinates of wall or chain

-boolean inOnGame

indicates whether wall or chain is in the game or not

-Rectangle[] rectangleArray

includes each rectangle which creates the wall or chain

#### Methods

+void draw(Graphics g)

draws wall or chain

+void turnLeft()

turns object to left

+void turnRight()

turns object to right

+void setRectangles()

sets the coordinates of the rectangles that creates the wall or chain

+boolean contains(int x, int y)

checks whether the given coordinate is included by wall/chain or not

+boolean isOutOfScreen()

checks whether the selected object is out of screen or not

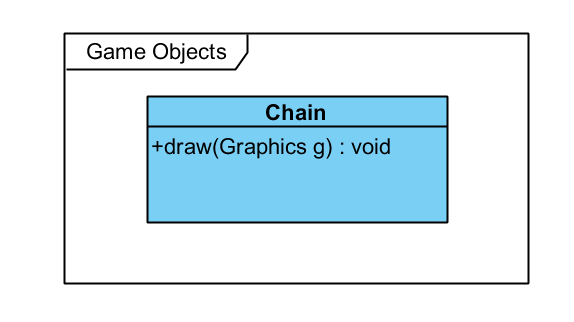
### Wall

This is the Wall class that is used to separate EnemyLandForces and AllyLandForces from each other in land area.

#### Methods

+void draw(Graphics g)

takes a Graphics object and draws the image of itself on a specific place



### Chain

This is the Chain class that is used to separate EnemyArmada and AllyArmada from each other in lake area.

#### Methods

+void draw(Graphics g)

takes a Graphics object and draws the image of itself on a specific place

### Warrior

This is the upper class of all warriors, which are Enemy and Ally

#### Methods

+void draw(Graphics g)

takes a Graphics object and draws the image of itself on a specific place

### Ally

This is the upper class of all allies, such as AllyArmada and AllyLandForces. It does not have any method and any attribute.

### Enemy

This is the upper class of all enemies, such as EnemyArmada and EnemyLandForces

#### Attributes

-int power

the damage does to a wall or a chain in one shot.

#### Methods

+void damageWall()

damages the wall that stands next to itself if there is a such wall.

+Wall findWallToBeDamaged()

returns the the wall that stands next to itself if there is a such wall. Otherwise, it returns null.

+boolean checkWallToBeDamaged()

returns true if there is a such wall. Otherwise, it returns false.

### MovableEnemy

In our game some enemies will be able to move through a predetermined path whereas some of them will be stable. This class is the upper class of all movable enemies.

#### Attributes

-Coor[] path

keeps the coordinates of the path that enemy is able to walk

-int speed

speed value of enemy

#### Methods

+boolean checkWall()

checks whether there is a wall or not

+void move()

moves the enemy

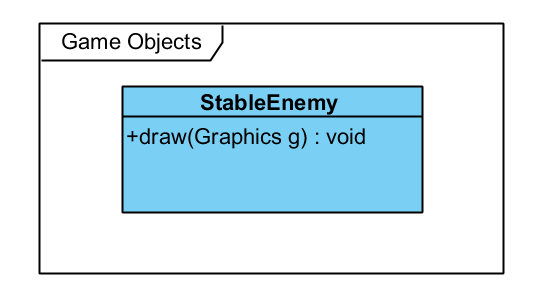
+void draw(Graphics g)

draws the movable enemy

+void setSpeed(int s)

sets the speed of the enemy

### StableEnemy

This class is the upper class of all stable enemies.

#### Method

+void draw(Graphics g)

draws the stable enemy

### EnemyArmada

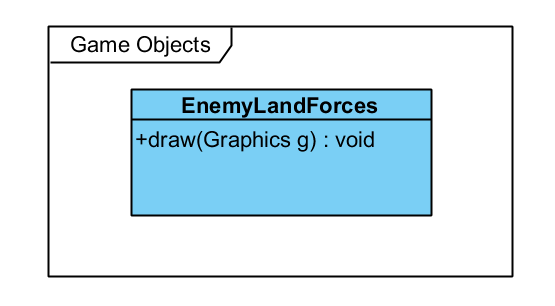
This class is the class of enemy armada such as a ship that patrols in the lake

#### Methods

+void draw(Graphics g)

takes a Graphics object and draws the image of itself on a specific place

### EnemyLandForces



This class is the class of enemy land forces such as a knight that patrols in the land.

#### Methods

+void draw(Graphics g)

takes a Graphics object and draws the image of itself on a specific place

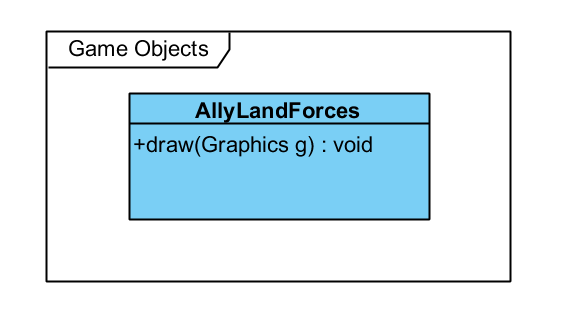
### AllyArmada

This class is the class of ally armada such as a ship that patrols in the lake

#### Methods

+void draw(Graphics g)

takes a Graphics object and draws the image of itself on a specific place



### AllyLandForces

This class is the class of ally land forces such as a knight that patrols in the land.

#### Methods

+void draw(Graphics g)

takes a Graphics object and draws the image of itself on a specific place

### Coor

This is a basic coordinate class that represents a point in 2D.

#### Attributes

-int x

x coordinate

-int y

y coordinate

#### Methods

+int getX()

returns the x coordinate

+int getY()

returns the y coordinate

+void setX(int x)

sets the x coordinate

+void setY(int y)

sets the y coordinate

# Low-level Design

## Object Design Trade-Offs

### Initialization Time vs Maintainability:

We use files to store and load different maps of the game, because it is easier to introduce new maps or modify the existing maps this way. However, the level startup takes slightly more time because of the need to read and interpret the file.

### Performance vs Graphics:

In order to attain greater performance, we chose to use 2D graphics. This also helped make the game look simpler and more playable, which is in lines with the fact that the game is addressing an audience from a wide age range.

## Packages

### org.newdawn.slick.command

Provides abstract input by mapping physical device inputs (mouse, keyboard and controllers) to abstract commands that are relevant to a particular game.

### org.newdawn.slick.loading

Adds support for deferring loading of resources to a set time to allow loading/progress bar style effects.

### org.newdawn.slick.state

State based games allow the game to be broken down into the different activities the player may take part in, for instance menu, highscores, play and credits.

## Class Interfaces

### MouseListener

This interface will be invoked whenever a mouse action is received from the user in order to track the mouse commands. Mouse events are the main means of playing the game.

### KeyListener

This interface will be invoked whenever a key action is received from the user, such as a key press or release. Keyboard can be used to perform various control actions (pause game, mute music) and also to play.

# References

Slick Javadoc page:

<http://slick.ninjacave.com/javadoc/>

Balsamiq – Mockup Software

https://balsamiq.com

Visual Paradigm – Diagram Design Software

https://visual-aparadigm.com