Cs319 Term Project

Section 3

Group 3D

Nomanleft

Iteration 2
Design Report

Group Members:
Ahmet Akif Uğurtan
Berkay Karlık
Can Kaplan
Teymur Bakhishli
Eren Aytüre

Supervisor: Eray Tüzün

1.Introduction

1.1. Purpose of the system

In our project, we will implement Walls & Warriors board game to a PC environment. The original game setting includes a game board, 4 walls of different shape, 1 high tower, 3 blue knights and 4 red knights. Players then place the knights and high towers to the game board in the way game's challenge booklet suggest. Once the setup is completed, players may start the game. The purpose of the game is placing the walls on the selected challenge in such a way that all the red knights are left outside the walls while all the blue knights and the high tower is inside. Once this achieved the given challenge is considered solved. The booklet has 80 of such challenges. Our implementation has many expansions and changes over the original one. A major difference is beside the original challenge solving mode, there will be time trial mode where the player will solve as many puzzles as he/she can against time and a sandbox mode where the player can design he/she's own challenge and then play it. Alongside the new mods, original gameplay's challenge design will be expanded as well. Different landforms such as hills or lava pits; the variety of characters with distinct attributes such as giant which covers two unit ground, battering-rams that requires more unit of walls to complete will save gameplay from monotony by enforcing player to think new ways of solving the puzzle. A set of boosters will be available player's aid to complete the challenge rather easily. Each level completion will include some gold reward so that player can refill boosters from the shop ones they finish. Besides the gold, depending on the number of steps taken towards the completion, a level can be completed with up to three stars, which shows player's puzzle solving skills.[1]

1.2. Design goals

1.2.1. **Trade-offs**

1.2.1.1 Server Response Time vs. Modifiability

The images as icons will be kept in the game directory. The icons are constant unless a new version comes to force.

1.2.1.2 Definitions, acronyms and abbreviations

1.2.1.2.1 MVC

Model View Controller (MVC) is the design pattern that we are going to use in the game.

1.2.1.2.2 JRE

Java RunTime Environment (JRE) is a part of the Java Development Kit (JDK) that we are going to use in the game.

1.2.1.2.3 FPS

Frames Per Second (FPS) of the game is going to be as high as it is not decreasing our game performance since FPS is affecting the general appearance of the game to the user.

1.2.1.2.4 GUI

Graphical User İnterface (GUI) is a form of User that we are going to use in our game. GUI allows the user to interact with electronic devices by the help of graphical icons, images, buttons and etc.

1.2.2. Criteria

1.2.2.1 Performance Criteria

To satisfy response time, while moving objects like boosters, walls or warriors, in to a map, and deployed, desired response time cannot pass 1 second. The display of animations, movements and effects should have proper response time to satisfy users` request.

1.2.2.1.1 Memory

The game has just a few icon images, text files and sounds to store. Therefore, the application does not require more than 1GB memory.

1.2.2.2 Dependability Criteria

1.2.2.2.1 Reliability

Application will be developed in a manner that bug occurrence will be as little as possible. The process will be running on Java 8 and upper models of the Java. However, continuity of service is until the end of the semester.

1.2.2.2.2 Availability And Dependability

Although, this software app is applicable for people who are mentally, physically healthy and literal, this application is only deliverable to Bilkent University academic studies. However, according to demand, it is ready for usage.

1.2.2.3 Cost Criteria

1.2.2.3.1 Development Cost

This application is not designed to be tradable. Moreover, its resources like images and sounds are going to be free. This app is a free application.

1.2.2.4 Maintenance Criteria

1.2.2.4.1 Modifiability

Features can be modifiable. Since there is a weak connection between subsystems, modification in a subsystem will not affect the other subsystems. Such as a new type of wall, new costume, new boosters can be added according to

Open, Close or Single Responsibility Principle.

1.2.2.4.2 Portability

Portability is a crucial factor regarding the usability of the game by enabling the users to be played in different devices. Since this application is developed on Java, Java has Java Virtual Machine, known as JVM, which lets the system to be portable.

1.2.2.4.3 Readability

The code segment of the application has to be understandable. Developer has to easily able to add features or apply changes (modification) in the code.

1.2.2.4.5 Security

This application does not require external services such as database or web component. Therefore, regarding this aspect, this application does not need security precautions such as SQL injection or xss. However, since this application is a jar file, it is vulnerable to the external decompiler tools (unauthorized modification).

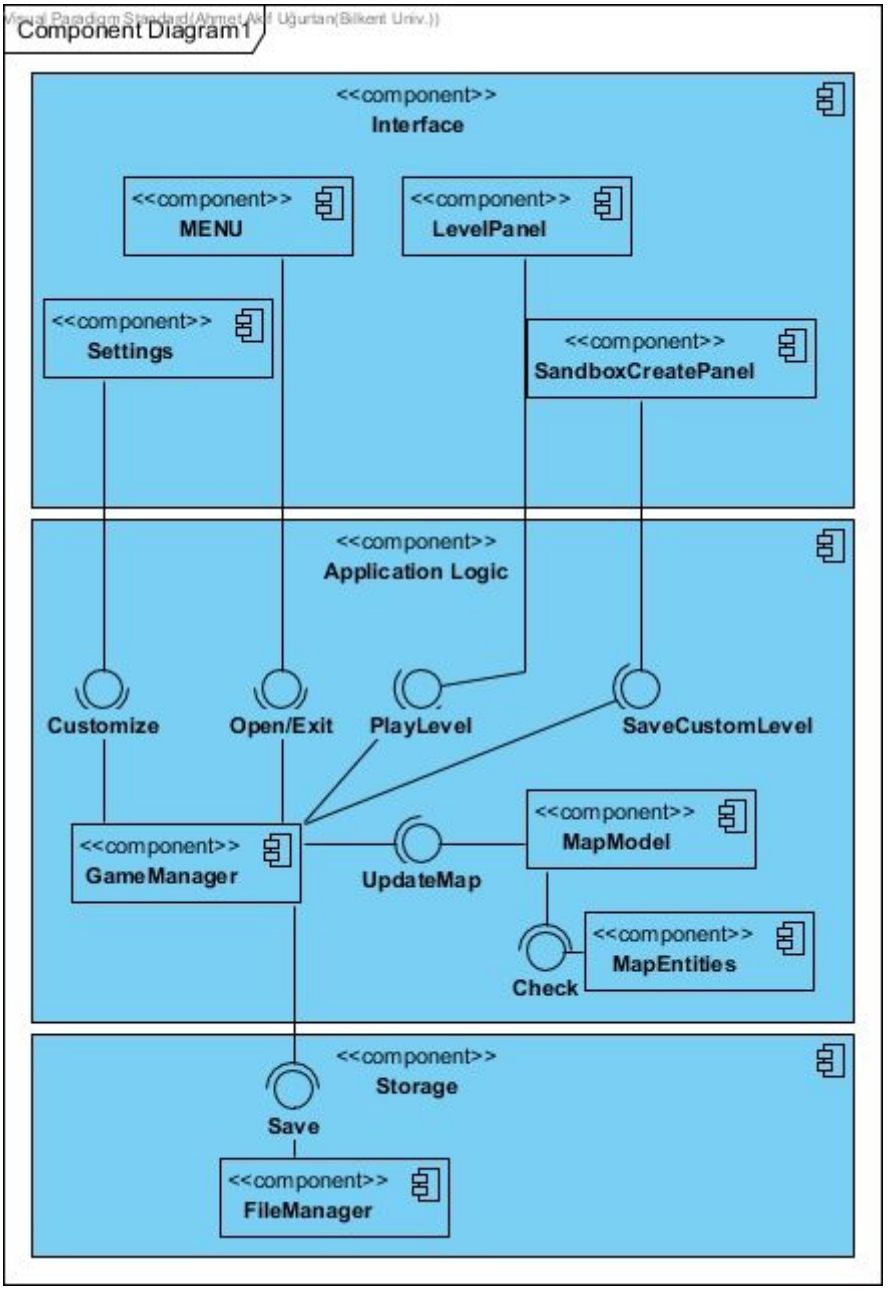
1.2.2.5 End User Criteria

1.2.2.5.1 Usability

To gain user attraction, simplicity is significant enough to enforce. For the sake of user friendly application, the user interface supports intelligibility of moving which objects like boosters, warriors, walls maped easily for player. Users are supposed to be productive due to its efficiency. Also, this application does not require, so much thing to learn or remember. Its features are simple to play even for children who become literal.

2. High-level software architecture

2.1. Subsystem decomposition



We will decompose our system into three subsystems named Application Logic, Storage, and Interface. the aim of our decomposition into three subsystems is managing the project easily and updating or extending the game partly so we do not have to update every component.

We choose this design since we want to manage user interfaces and model objects independently. In our project, Interface subsystem manages the actions of all view components on interfaces such as buttons, labels etc.. Application logic subsystem manages the change on the map and updates it if necessary so it controls game objects and boosters

and become bridge between model and view by deciding which function of Model component should be called according to events from view component.

Storage subsystem stores the user specific and game specific images in FileManager. By this way, the loading images will be easier.

2.2. Hardware/software mapping

We will implement Nomanleft game by using Java SE libraries and Java version 8. Thus, in order to run Nomanleft, Java Virtual Machine and at least Java version 8 is necessary as software but also thanks to JVM our game can run on any computer that has installed JVM.

In order to play Nomanleft, users will need a mouse and any computer as hardware. The mouse will be used to click, press, drag events in order to use boosters or put walls etc. as an input tool.

Our application will store maps on the local storage as files and there is no multiplayer game or high score comparison. Thus, our game does not need an internet connection or it has neither a database nor a server.

2.3. Persistent data management

Our project will have images for better user experience by changing button images and also in order to describe objects on the game map such as soldiers, walls etc.. Additionally, it will use .txt files in order to save and load maps and settings choices of the user by representing a map using ASCII characters and keeping configuration choices as key-value parameters. These image and text files will be stored on the local store so any data storage rather than the user's computer data disk will not be needed.

2.4. Access control and security

In order to play Nomanleft, there is no need for any internet connection and we will not use any database system for our game. After loading the game and starting the Nomanleft, anyone can open and play the game. Therefore, we won't have any need for any security requirements.

2.5. Boundary conditions

Nomanleft will not require any installation and it will not have any .exe extension. The game will have an executable .jar file and it will be executed from that .jar file. With this game will be portable and easy to move and it will be ready to play always.

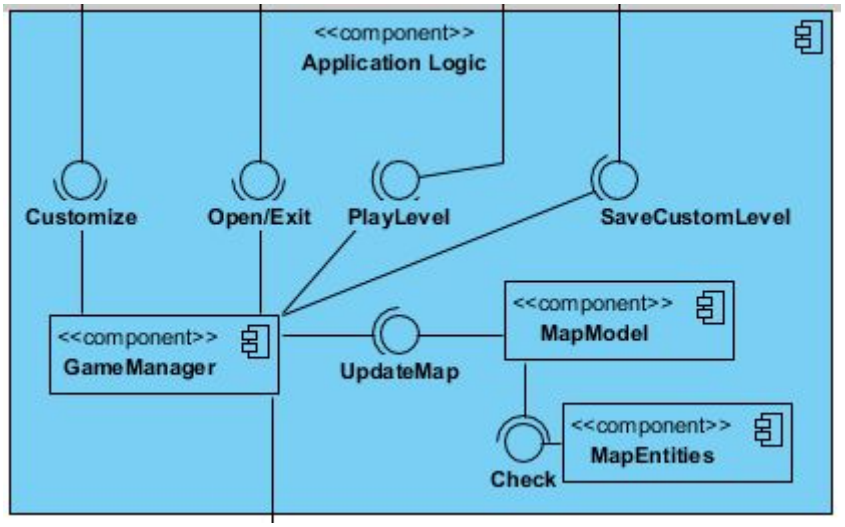
Nomanleft game can be terminated via by clicking the "Exit" button in the main menu. There will be the option to go back from level to the menu.

There can be an error during the start of the game which can cause by errors while reading the file. This problem can be solved by fixing or replacing text files.

If the game crashes during the level because of performance or design issue, the level will not be saved and boosters will be replaced.

3. Subsystem services

3.1. Application Logic Subsystem



Model subsystem is responsible for providing a map for Nomanleft. It has 3 main components which are:

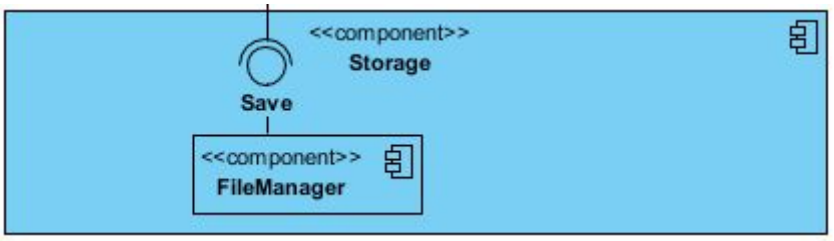
1. MapModel Component.
2. MapEntities Component.
3. GameManager Component

This system will be responsible for the creation and update of the map. First **MapModel** will create a map for the according to the level and provide a mapObject array according to that level.

After that **MapEntities** will be placed in that mapObject array. This MapEntities will consist of objects like various tiles(Ground, lava, mountain), walls or human objects(soldier, battle ram, peasant). There are other MapEntities which are boosters. These boosters let the player change the MapModel by chancing array. In order to change MapModel user need to use required booster or walls. Because each booster works on the different type of object. And walls can only be placed on ground type. This action needs to be checked and MapModel will update according to that.

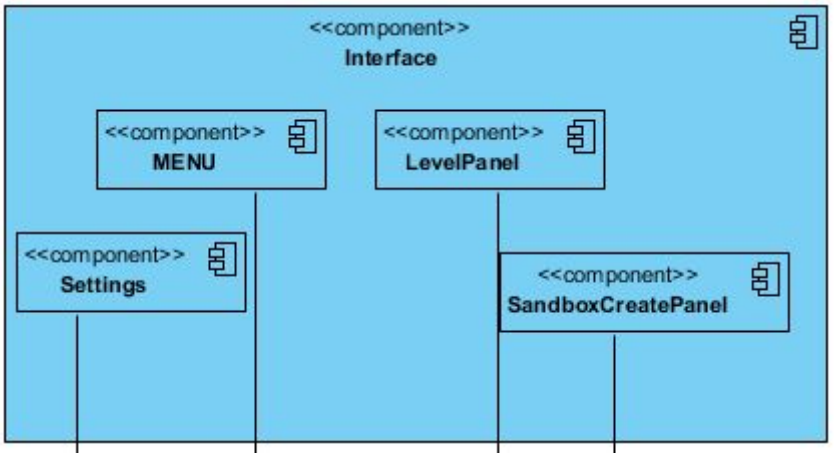
The **GameManager** component of this subsystem have two important function. First it's responsible for managing all the interactions between classes, thus this component is reachable by all other components. By doing that we aim to eliminate any chance of complex interactions between components which would make design harder, more prone to errors. That also responsible of the game mechanics and logic. Main game mechanics such as, game start, wall placing, booster use, endgame check and game finish are handled on this component.

3.2 Storage Subsystem



The **FileManager** component is responsible for writing and reading from the files. The images and the sounds of the game, player preferences (customization) and information (high score), the custom level designs has to be read from and written to files. This component is reached by other components via GameManager.

3.3 Interface Subsystem



This subsystem is responsible for all the visuals that user will see on the screen throughout the game. There are four components in that subsystem:

1.MENU
2.LevelPanel
3.Settings
4.SandboxCreatePanel

Here the components do not interact with each other directly but through the GameManager.

The **MENU** component manages the Menu that greets the player at the start. This component can be considered as a button handler. All the menu actions such as pressing play game (or other modes), options, shop buttons results with this component invoking the GameManager.

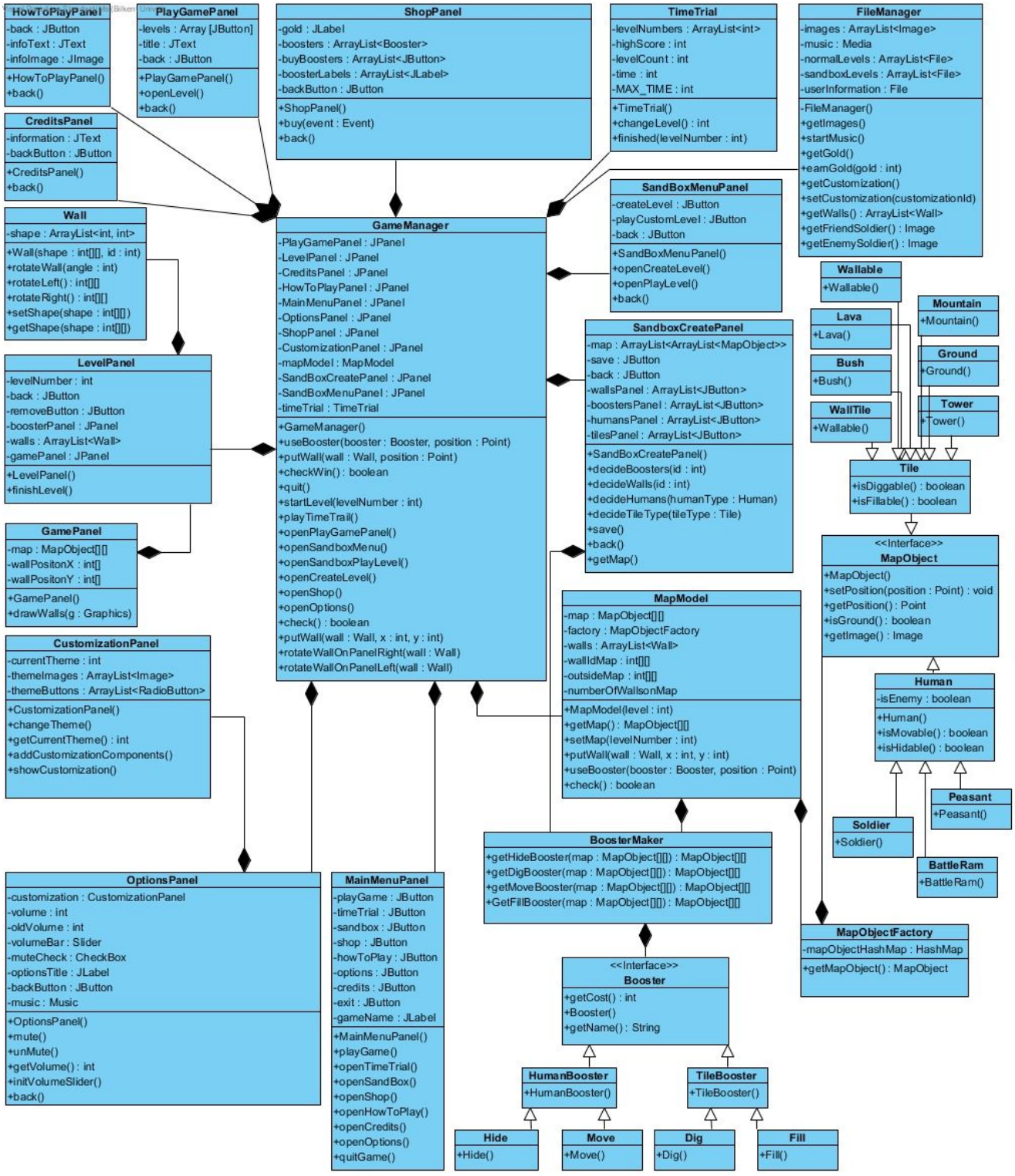
The **LevelPanel** component is responsible for the in-game view. The panels that contains walls, boosters, the view of the map, the visual interactions between them such as placing a wall, dragging a booster to screen etc. are handled here.

The **Settings** component handles players preferences. Player can adjust sound level, mute/unmute sound, customize the visuals of characters through the options menu. All of these, will be handled through this component and will be applied to others by the GameManager.

Finally the **SandboxCreatePanel** component manages everything related to SandBox game mode. Sandbox Game mode have its own component due to its inherent complexity. Custom level making, saving and playing handled in this component. To save and reload the made level, This components interacts with FileManager via GameManager.

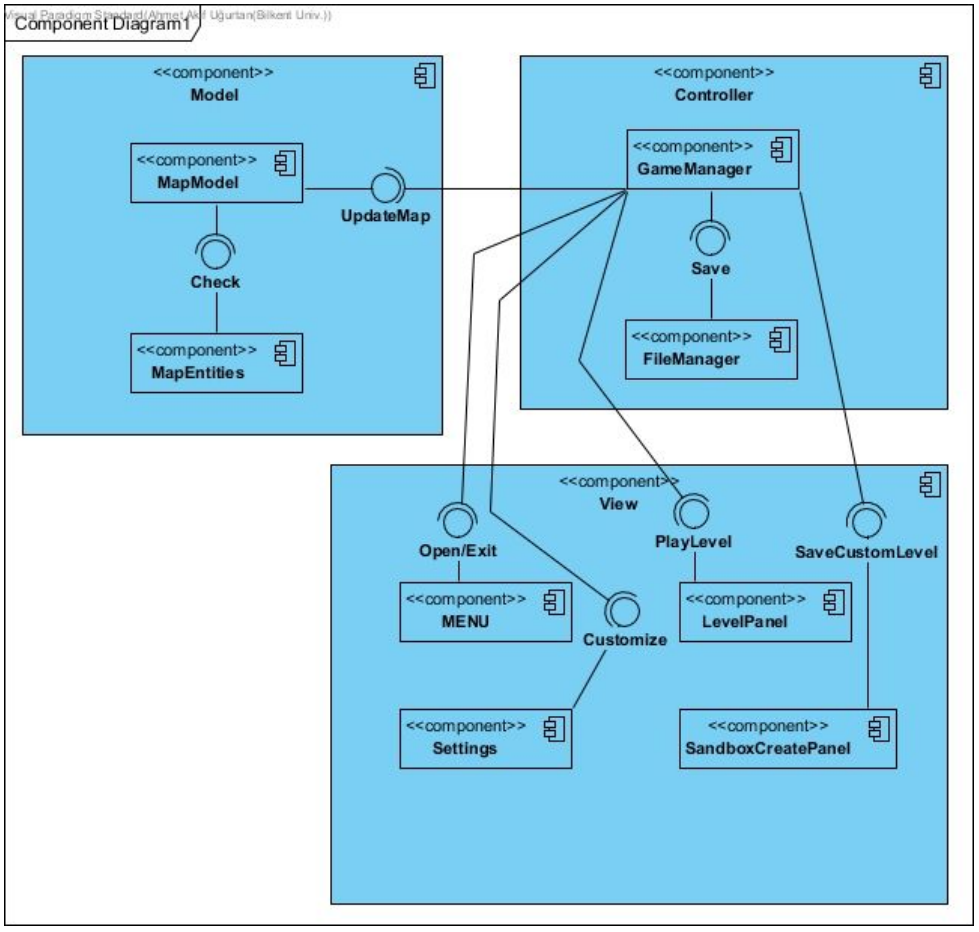
4. Low-level design

4.1. Object design



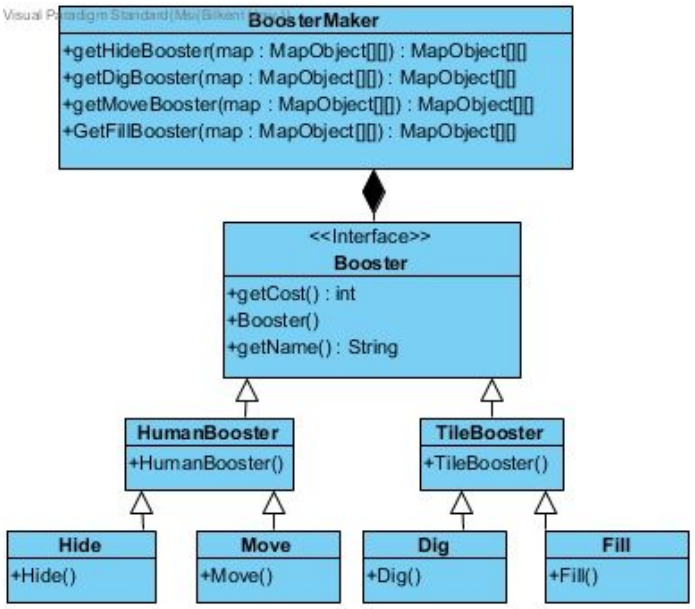
4.2. Design Patterns

4.3.1. Model View Controller



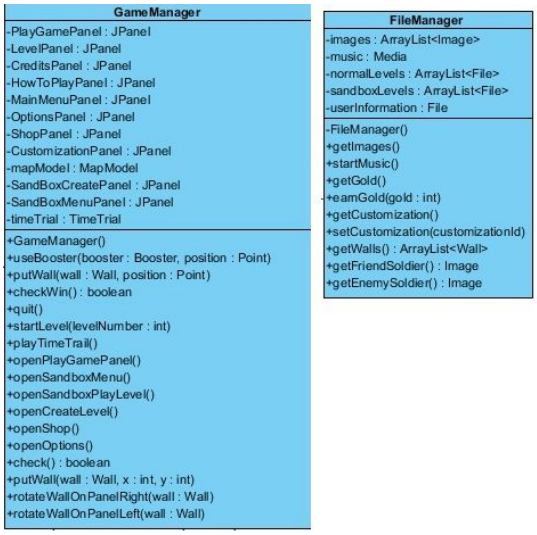
In order to separate the implementation into different parts, we used MVC and decomposed the project as Model, View and Controller components where all components have different classes. Thanks to this design pattern, we can easily update Model or View component without changing the other component so our classes gain independence of each other. Also it enables different people to work on different parts without confronting each other.

4.3.2. Facade



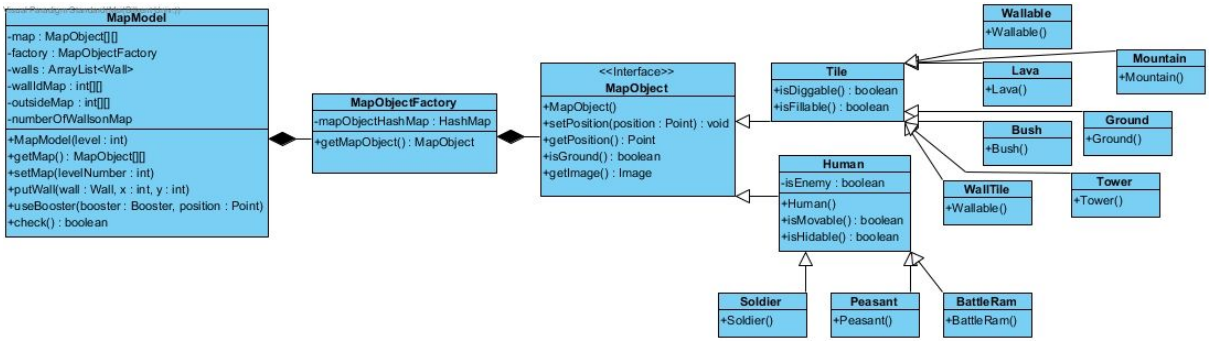
In order to hide complexities of the project implementation, we used Facade design which is a structural design pattern. We grouped booster classes on Booster interface and hide the implementation thanks to the interface. Both MapModel and SandBoxCreatePanel uses boosters. Facade design pattern makes it easier to reach to the functions related to these classes without involving the actual objects.

4.3.3. Singleton



In order to increase the operation of our game system more efficiently, we only need one instance to reach FileManager and GameManager. Since we will create FileManager only once, every file will be static. Addition to that, GameManager needs to be created just once to maintain the game's reliability because more than one manager can affect the game's integrity. This two classes also perform many functions and share data in between them so reaching them directly makes design easier.

4.3.4. Flyweight



In our game, there will be mapObjects which will be formed by tiles and human objects. We have a mapObjectFactory class which lets us the copy and reuse a mapObject which was created before. We will create less mapObject objects via this design pattern and this will let us use less memory. In our game we have multiples of each map object (multiple soldiers, wallTiles, ground etc.) type that can be on the game map so creating only one object and referring it multiple times prevent extra memory usage.

4.3.1 External Library Packages

4.3.1.1 Java.Util

This package mainly used for it's ArrayList class which is used in many of our classes. Other utilities it provides might be used if implementation requires it such as Random, Scanner etc.

4.3.1.2 Javax.imageio

This package will be used to import the images to the game.

4.3.1.3 Java.io

Writing and reading of the game related files such as predefined and custom levels, player preferences, sounds will be handled with this package.

4.3.1.4 Javax.Sound.Sampled

This packages provides classes and methods for processing and playing audio data which will be used to play the music and sounds effects in the game.

4.3.1.5 Javax.swing

In our project swing package will be used as the main package for the GUI implementation. This package provides lightweight GUI components.

4.3.1.6 Javax.swing.event & Java.awt.event

Swing.event package doesn't include all the listeners we need such as key listeners etc. Thus we will use these packages to include all the listeners we need.

4.3.1.7 Java.awt.image

This packages provide classes for creating and modifying images.

4.3.2 Internal Packages

4.3.2.1 Menu Package

This package includes all the menu related classes such as MainMenuPanel, ShopPanel, OptionsPanel etc.

4.3.2.2 Map Package

This package includes classes and interfaces related to in-game view and mechanics.

4.3.2.3 GameMode Package

Different game modes are initialized by different classes. Each of these classes included in this package.

4.4. Class Interfaces

4.4.1. File Manager Class

private ArrayList<Image> images: This attribute keeps Image object instances of image files of shapes in the game such as boosters, walls, soldiers, etc.

private Media music: This Attribute keeps Media object instance of the music which will be played in the background.

private ArrayList<File> normalLevels: This attribute keeps the list of default level maps which are stored as files in local storage and map objects are described by some ASCII characters.

private ArrayList<File> sandboxLevels: This attribute keeps the list of level maps which are created customly by users.

private File userInformation: This attribute keeps user settings and customization choices and gold etc.

private FileManager(): Constructor but it is private since it will be implemented by singleton pattern.

public ArrayList<File> getImages(): the function to get images array.

public void startMusic(): the function in order to open and play the music file.

public int getGold(): the function to get the gold amount of user from user information file.

public void earnGold(): the function to save the gold amount into the user information file.

public int getCustomization(): the function to get current customization choice of the user.

public void setCustomization(int customizationId): the function to save new customization choice of the user into user information file.

4.4.2. TimeTrial Class

private ArrayList<int> levelNumbers: id numbers of levels that are passed successfully during time trial challenge.

private int highScore: the maximum number of levels user passed among all trials.

private int levelCount: number of passed levels during this particular time trial challenge.

private int time: the time that increases during the trial and indicates how long time user spent.

private final int MAX_TIME: the time limit for time trial challenge.

public TimeTrial(): Constructor for initialization.

public int changeLevel(): Chooses new level for a challenge and sends it to the manager to be played.

public void finished(int levelNumber): Gets the passed level id and stores it in the list and calls changeLevel() function to create a new level.

4.4.3. SandboxCreatePanel Class

private ArrayList<ArrayList<MapObject>> map: Current map object instance which is created and modified by user.

private JButton save: The button to save the current map into a file in order to play later.

private JButton back: The button for navigation by going back to the menu.

private ArrayList<JButton> wallsPanel: Different buttons for differently shaped walls so that user can click and choose whether this wall will be available in game or not.

private ArrayList<JButton> boostersPanel: Different buttons for different boosters so that user can click and choose whether this booster will be available in game or not.

private ArrayList<JButton> humansPanel: Different buttons for different human objects so that user can click and drag the human onto the map.

private ArrayList<JButton> tilesPanel: Different buttons for different tiles so that user can click and choose the type of a tile on the map.

public SandboxCreatePanel(): Constructor for initialization.

public void decideBoosters(int boosterId): Click listener for booster buttons.

public void decideWalls(int wallId): Click listener for wall buttons.

public void decideHumans(int humanId): Click listener for human buttons.

public void decideTiles(int tileId): Click listener for tile buttons.

public void save(): Click listener for the save button.

public void back(): Click listener for the back button.

public ArrayList<ArrayList<mapObject>> getMap(): Click listener for tile buttons.

4.4.4. Tile Class

public boolean isDiggable(): Checks the tile if it is usable for dig booster.
public boolean isFillable(): Checks the tile if it is usable for fill booster.

4.4.5. Lava Class

public Lava(): Constructor for creating instance of lava for limiting places for walls.

4.4.6. Bush Class

public Bush(): Constructor for creating instance of bush for decreasing pressure on user.

4.4.7. WallTile Class

public WallTile(): Constructor for creating instance of wall tile for limiting user.

4.4.8. Ground Class

public Ground(): Constructor for creating instance of ground on tiles.

4.4.9. Mountain Class

public Mountain(): Constructor for creating instance of mountain for limiting places for walls.

4.4.10.Tower Class

public Tower(): Constructor for creating instance of tower for limiting user.

4.4.11. Wallable Class

public Wallable(): Constructor for creating place for walls to put in map.

4.4.12. Human Class

private boolean isEnemy: Attribute for checking the if human-friendly or not.
public BattleRam(): Constructor for initializing battle ram.
public Soldier(): Constructor for initializing soldier.
public Peasant(): Constructor for initializing peasant.
public Human(): Constructor for initializing human.
public isMoveable(): Checks the human if it is usable for move booster.
public isHideable(): Checks the human if it is usable for hide booster.

4.4.13. MapObject Interface

public mabObject(): Constructor for initializing map object.
public void setPosition(Point position): Setter function of map object's position.
public Point getPosition(): Getter function of map object's position.
public boolean isGround(): Checks the tile and returns true if ti is ground.

4.4.14. MapObjectFactory Class

private HashMap mapObjectHashMap: Stores the mapObjects in hash according to their existence in hash before.

public MapObject getTile(): Getter function of map object's tile.

public MapObject getHuman(): Getter function of map object's human.

4.4.15. Booster Interface

public HumanBooster(): Constructor for initializing human booster.

public TileBooster(): Constructor for initializing tile booster.

public Hide(): Constructor for initializing hide.

public Move(): Constructor for initializing move.

public Dig(): Constructor for initializing dig.

public Fill(): Constructor for initializing fill.

public Booster(): Constructor for initializing Booster.

public int getCost(): Getter function of the price of the booster.

public String getName(): Getter function of the name of the booster.

4.4.16. Wall Class

private ArrayList<ArrayList<int>> shape : Stores the shape of the walls in a 2D array.

public Wall(): Constructor for initializing a wall.

public rotateWall(int angle): Rotates the wall in object by the angle provided.

4.4.17. LevelPanel Class

private int levelNumber: Stores the number of the level that will be loaded.

private JButton back: The GUI button that closes the level and loads the menu.

private JPanel boosterPanel: The boosters are displayed on this panel.

private ArrayList<Wall>: Keeps the walls on the level that put by the player.

private JPanel wallPanel: Keeps the walls that player can place to the map.

public back(): when JButton back clicked this method is awaken. It closes the level and loads the main menu.

public levelPanel(): constructor that initializes the LevelPanel.

public boosterClick(): when player uses a booster via dragging or clicking this method is invoked. Then it invokes the useBooster method in GameManager which checks the conditions and applies booster if its valid.

public wallClick(): when player places a wall to the map this method is invoked. Then it invokes putWall method from the GameManager which checks the conditions and places the wall if its valid.

4.4.18. GamePanel Class

private MapObject[][] map: Stores the map shape and ids.
private int[] wallPositionX: Stores the x position of walls for rotate**.**
private int[] wallPositionY: Stores the y position of walls for rotate.
public GamePanel(): Constructor of GamePanel for initializing game map, walls and boosters.
public drawWalls(Graphics g): Draws the wall to the panel and change according to rotates.

4.4.19. CustomizationPanel Class

private int currentTheme: Stores the number that represents the selected theme
private ArrayList<Image> themeImages: Stores the theme images.
private ArrayList<JButton> themeButtons: The reference to the buttons that will be used to choose themes will be stored here.
public CustomizationPanel(): Constructor that will initialize the customization panel.
public changeTheme(): Whenever player click to a themeButton this method will be invoked.
public int getCurrentTheme(): returns the number of the current theme.
public addCustomizationComponents(): adds the GUI componenets to the panel.
public showCustomization(): opens the customization panel.

4.4.20. GameManager Class

private Jpanel PlayGamePanel: keeps a reference to the playGamePanel.
private Jpanel LevelPanel: keeps a reference to the LevelPanel.
private Jpanel CreditsPanel: keeps a reference to the CreditsPanel.
private Jpanel HowToPlayPanel: keeps a reference to the HowToPlayPanel.
private Jpanel Main Menu Panel: keeps a reference to the Menu .
private Jpanel OptionPanel: keeps a reference to the OptionPanel.
private Jpanel ShopPanel: keeps a reference to the ShopPanel.
private Jpanel CustomizationPanel: keeps a reference to the CustomizationPanel.
private Jpanel SandBoxCreatePanel: keeps a reference to the SandBoxCreatePanel.
private Jpanel SandBoxMenuPanel: keeps a reference to the SandBoxMenuPanel.
private MapModel mapModel: keeps a reference to the MapModel class.
private TimeTrial timeTrial: keeps a reference to the TimeTrial class.
public GameManager(): Constructor for initializing the gameManager.
public UseBooster(Booster booster,Point position): checks if the given booster can be applied to given point if is applicable it applies the booster.
public putWall(Wall wall, Point position): checks if the given wall can be applied to given point if is applicable it places the wall.

public boolean checkWin(): if the current wall placement is correct returns true else false.

public quit(): the method to close the game.

public startLevel(int LevelNumber): loads the level with the given number.

public playTimeTriall(): starts a game with time trial mode.

public openPlayGamePanel(): activates the play game panel.

public sandBoxMenu(): activates the sandbox menu.

public openSandBoxLevel(): loads a sandbox level.

public createLevel(): activates sandboxCreatePanel.

public openShop(): activates the shop panel.

public openOptions(): opactivates ens the options panel.

4.4.21. ShopPanel Class

private JLabel gold: displays the gold on the shop panel.

private ArrayList<Booster> boosters: keeps references to the boosters on sale.

private ArrayList<JButton> buyBoosters: keeps references to the buttons that will be used to buy boosters.

private ArrayList<JLabel> boosterLabels: keeps references to the labels of the boosters on sale.

private JButton backButton: keeps a refference to the back button.

public ShopPanel(): constructor for initializing the shop panel.

public buy(Event event): invoked when player attempts to buy a booster.

public back(): closes shop panel.

4.4.22. PlayGamePanel Class

private ArrayList<JButton> levels: keeps references to the level selection buttons.

private JText title: keeps references to the JText which keeps the title.

private JButton back: keeps a refference to the back button.

public PlayGamePanel(): constructor for initializing the playGamePanel class.

public openLevel(): when a level selection button clicked this method is invoked.

public back(): closes playGamePanel.

5. References

Original game:

https://www.smartgames.eu/uk/one-player-games/walls-warriors[1]