CS 319 - Object-Oriented Software Engineering

Final Report

Last Man

<u>Group 2-8</u>

Abdullah Enes Akdoğan

Burcu Çanakcı

Yasemin Doğancı

Özgür Taşoluk

1 CONTENTS

2 INTRODUCTION

Last Man is a basic strategy video game we decided to develop. There are lots of strategy type video games in market and each of them has its different features. Last Man is a game which is very similar to the basic Bomberman game, and the main purpose in the game is to place bombs and use different weapons in order to kill enemies and destroy the walls. The game we were influenced by, Bomberman, can be seen in the following link:
http://tr.y8.com/games/bomber_man

We plan to develop our game with different features. In Last Man, there will be different heroes to select, special weapons, and different packs that appear during gameplay. There will be different game maps to choose from, and although Last Man does not have a level-based system, different maps will give the user the experience of encountering different levels.

The game will be a desktop application and will be controlled by both mouse and keyboard.

Analysis

3 REQUIREMENTS ANALYSIS

3.1 OVERVIEW

Last Man is a strategy game like the Bomberman. Like the game it's based on, it is fun and very easy to play at the same time. The game provides the user a map to play in and this map consists of different walls (obstacles) and paths, and the game also has different enemies to beat. Player tries to destroy the obstacles and kill his/her enemies by placing bombs and using weapons in the desired locations. Every explosion (meaning the bomb/weapon is used in some location on the map), can either damage enemies, or damage the walls. Enemies are destroyed depending on their health points and the walls crack or are destroyed depending on their type. If the player destroys all the enemies in the map, the player wins the game. The goal is to kill enemies before the player dies. In our project, we named these computer controlled enemies **Bots**, and the user-controlled player **the Player**. Both Bots and the Player are **characters** in the game.

3.1.1 Gameplay

The player can use the mouse and keyboard to play. Actions like using weapons are fulfilled by using specific buttons at the keyboard, and the movement of the player is provided by the arrow keys. Mouse will be useful while choosing the game specifications and while the player is in the main menu of the game.

3.1.2 Maps and Walls

Last Man will have different maps and these maps differ in the type and number of the walls they have. Walls are designed to make the game more fun and challenging. There are 3 type of walls and they differ in their resistance.

Normal walls: Can be destroyed by one hit by Weapon 1 (default weapon of all heroes – the bomb)
Strong walls: Can be destroyed by one hit by Weapon 2 (special weapon of heroes if specified) or by two hits with the default weapon.
Unbreakable walls: The default and unbreakable walls of the map.

3.1.3 Heroes

There are various types of heroes to choose from. The heroes differ from each other as their health points, speed and special weapons are diverse. Different hero selections will help player to enjoy the game even more and make the game more or less difficult. Below are some of heroes to choose from to play Last Man:

Table 1 List of Heroes

Name	Speed(1-10)	HP(1-1000)	Special Weapon
Kil	10	800	Lightning
Jas	8	900	Arrows
Ustah	8	700	Building Walls
Ayibogan	5	1000	Unbearable Attack
Drogon	7	760	Fireball
Nemo	8	650	Armageddon

3.1.4 Default Weapon & Special Weapons

Weapon types are the essential part of the game and they make the game very different from the basic Bomberman. The game style changes according to the hero selection of the player. **Damage** represents the damage of the weapon. **Delay** represents how often they can be used.

Table 2 List of Weapons

Name	Description	Damage	Delay
Default weapon – The bomb	All heroes have this default weapon.	100	Depends on assigned hero
Lightning	Hits specific location in the map	250	10s
Building Wall	Builds wall in a specific location. Wall type is normal.	-	2s
Arrow	Hero sends 3 arrow at intervals of 1 seconds to the row and column which hero standing. Arrows can pass through walls. Arrow goes until it reaches the boundaries of the map or until it hits another player.	60	15s
Armageddon	Hits different places in the map without any delay.	75	20s
Fireball	Destroys the walls in its path and damages all enemies in the path. Fireball's direction is specified as the hero's direction. (If the hero is facing North, fireball is sent in North direction.)	250	12s
Unbearable attack	Hero gains 10 second strength. Hero is able to damage any enemies in his path.	400	35s

3.1.5 Packs

Packs are the basic power-ups and power-downs which occur during gameplay. They make the game even more fun and enjoyable. The player will not be able to know what the package has, so it will be a surprise for him/her and eventually the game will become easier or harder. The packs will appear in specific locations on map and the number of packs and their types will be random during one game. The packs are as follows:

Shield Pack: For 15 seconds the hero cannot take any damage.

Increase Speed Pack: Hero's speed is increased for 15 seconds.

Decrease Speed Pack: Hero's speed is decreased for 15 seconds.

First Aid Pack: Hero's Health Points increase by 250.

Range Bonus Pack: Default weapon's range is increased.

Extra Weapon Pack: The player can use default weapon more than once. The number of weapons used before explosion is increasing in a cumulative manner.

3.2 Functional Requirements

3.2.1 Create New Game

From the main menu of the game, the user will be able to choose from one of several options. One of them is the New Game option, which will proceed the user to a sequence of selections in order to start the game, when it's chosen.

3.2.2 Select Number of Characters & Bot Level

This is the first selection of the sequence. Here the user can select to play against from 1 to 3 Bots, so they can create a game that has 2 to 4 characters.

After that, the player can decide the skill level of the Bots. There will be 3 basic levels based on Bots' skill, defined as: Easy, Medium and Hard. These levels are based on Bots' ability to consume and/or use the packs. In Easy difficulty, Bots won't consume the packs; in Medium difficulty, they will consume the packs but will not be able to use them. This will cause a change in the game because in Medium difficulty, the Player will have a hard time when it comes to gaining packs. In Hard difficulty, Bots will be able to consume and use the packs, and therefore the game will be harder than usual.

3.2.3 Select Hero

Hero selection is another requirement of the game. The Player will be able to choose from various hero types. Heroes differ from each other as their initial health points, speed, damage potential and weapons change. Hero selection gives the Player the chance to gain experience in different playstyles.

3.2.4 Select Map

There are different types of maps the game can be played with. These maps differ in the amount of walls they have, in the properties of the walls they have and in the placement of these walls. The Player should be able to select one from these maps. The variety of maps makes the game more dynamic.

3.2.5 Select Maximum Game Time (Timer)

Just before starting the game, the user will set a timer value for the maximum length of the game. The input the user provides for this will be checked for validity before the game is started. When the specified time is reached, the player with the highest health points immediately wins.

3.2.6 Start Game

Start Game function will appear after every selection is made and when the Player chooses this option, their specifically created game will start. The Player begins to play and their hero will act according to their specifications.

3.2.7 End Game

Except from the game's ending conditions (death of Player, death of all bots or timer runs out), the Player should be able to end the game manually by clicking a button from the game view.

3.2.8 See Results Screen

Once the game ends, the user should be able to see a results screen that consists of the characters in the game and their rankings according to their health points and death times.

3.2.9 See Help

From the main menu, the user should be able to click on the help button in order to see the picture-text tutorials that teach how to play the game. This help section will provide detailed information about the rules and controls of the game and specifications about maps, heroes and packs.

3.2.10 See Credits

Again from the main menu, the user will be able to click on the credits button for information about the developers.

3.2.11 Change Settings

The user should also be able to adjust sound and music settings by clicking on the settings button from the main menu. The music option can be turned off in which case just the sound effects will be active. The volume of these effects can also be adjusted from this settings option.

3.3 Nonfunctional Requirements

3.3.1 Usability

- The game should be accessible, that is it should be easy to use by as many people as possible. For this, the user interface will be made simple and consistent.
- The amount of user documentation provided in the help section should also be sufficient for anyone with minimal computer experience to enjoy the game.

3.3.2 Reliability

- The inputs given to the game are almost always selected from a pre-determined menu, therefore unexpected cases are hard to occur.
- The game should always work smoothly independent of users' choices, however, if an event of failure occurs, the game can be manually shut down and restarted.
- This is not a level-based game, therefore no data is stored once the current game is over. Hence, data loss will not be a problem.
- It is also a simple game with basic graphics so it should not create stressful conditions for the computer.

3.3.3 Performance

- The game should not take more than one minute to load. The users should be able to iterate through game creation smoothly and once the game starts it should never stall.
- While creating the game, the user can take as much time as they want at each step, so there will be no time constraints for the user during game creation.
- The game supports only one player. Only one game can be played at a time.
- The game does not store permanent data, so there shouldn't be storage problems. Since this a simple game, the maximum latency shouldn't be more than one minute.

3.3.4 Supportability

- The game can be played on any desktop or laptop computer with any operating system.
- The game is very basic and therefore should require minimal maintenance.
- An extension to the game can be making it a multiplayer game that people can play from different computers. This game builds on simple fundamental game ideas, so it is open to a lot of creative extensions.

3.4 Constraints

3.4.1 Implementation

- The game will be programmed using the Java programming language and the Swing or OpenGL framework.
- Software like Photoshop, Paint.NET or AutoDesk Sketchbook will be used while creating the user interface.
- Software like Audacity will be used to manage sound effects and music.

3.4.2 Packaging

The game requires no installation. It can be downloaded as a .jar file and run smoothly.

3.4.3 Legal

This will be a non-commercial open-source project. The code and the game will be accessible to everyone by GitHub. All software and techniques used to develop the game are open source or already licensed so there will be no royalties or licensing fees.

3.5 Main Scenario

Scenario name mainScenario

Participating actor instances alice: Player

Flow of events

1. Alice clicks on the game icon from the desktop and opens the game. After the game loads, she activates the "Create New Game" system by choosing the "New Game" option from the main menu.

2. Alice selects to play with 4 characters in total and sets bots' skill level to hard.

3. She selects the hero named "Kil" and the map named "Mountain".

4. She sets the timer to 5 minutes and starts the game.

5. Alice uses Kil's speed skills to avoid getting hit by weapons and successfully manages to place weapons near the Bots. She manages to make all of the Bots' health points hit 0 by the third minute and so she wins the game.

6. The game ends and the program directs Alice to the results screen. This screen shows Alice in the first place and then the rest of the Bots in order from the one that was killed last to the one that was killed first.

7. Alice chooses to go back to the main menu from the results screen.

8. Alice chooses the "Change Settings" option from the main menu. The program shows her the settings menu. She chooses to turn the music off and she lowers down the sound level. She chooses the "Save Settings" option and a confirmation message is shown. Alice goes back to the main menu.

9. Alice chooses the "See Help" option from the main menu. The program creates a pop-up window with the user documentation. Alice observes the documentation and closes the window. She is back at the main window of the program.

10. Alice closes the main window of the program, a confirmation pop-up box appears. She confirms and exits the game.

This is the main usage scenario of our program. Several smaller and more specific scenarios are provided with the sequence diagrams.

3.6 USE CASE MODELS

3.6.1 Use Case Model

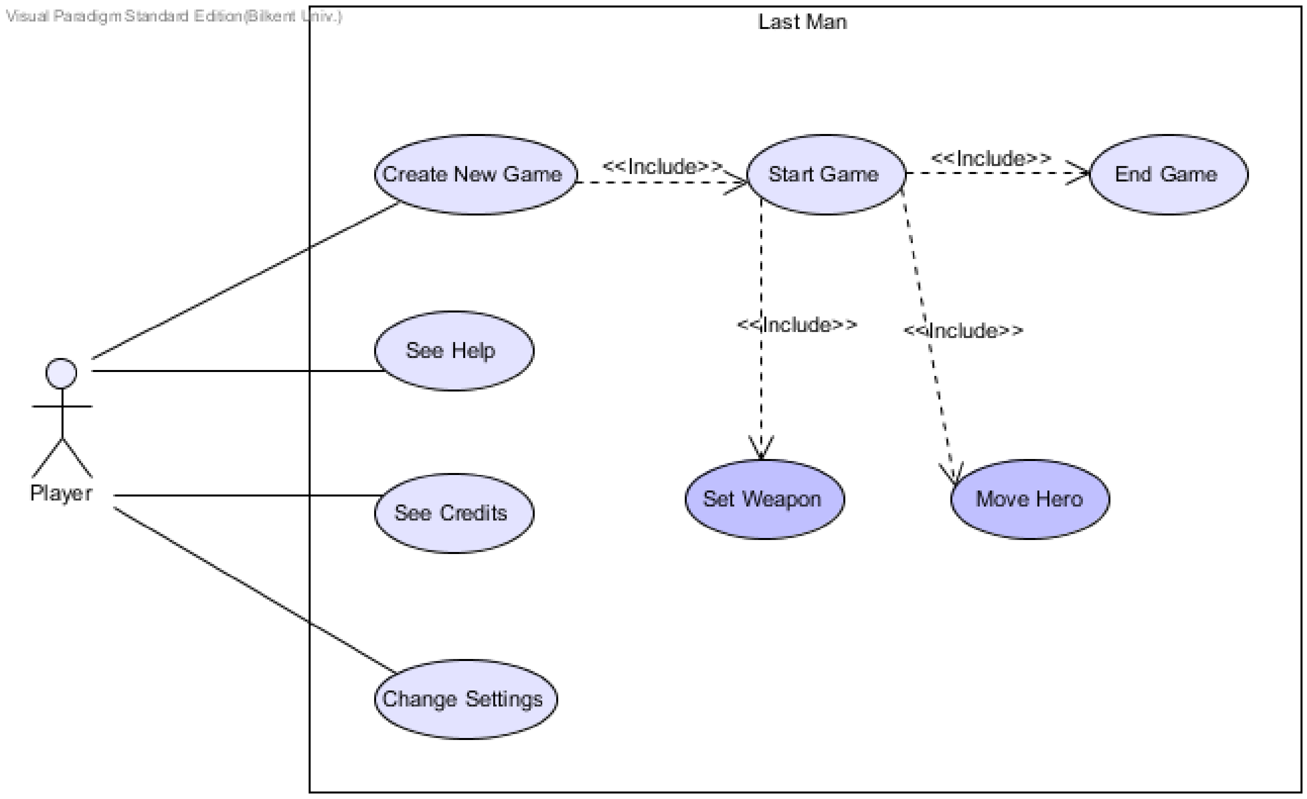


Figure 1 Use Case Diagram

Above is the main use case diagram for "Last Man". From the main menu, the player can create new game, see user documentation/help, see credits and change settings. During creating new game, they can choose to start the game and start playing. While playing they can choose to end the game. The use cases are investigated in detail below.

3.6.2 Use Case Definitions

Use case name **CreateNewGame**

Participating actors Initiated by Player

Flow of events

1. Player selects the "New Game" option from the main menu of the game
2. The program creates a default Game and starts to provide the Player with a series of selections to set up the game. The program dynamically updates the game after each selection according to the Player's choices.
3. The first selection is to determine the number of characters and general Bot skill level. Player chooses to play a 2-4 character-game and with Bot skill from 3 difficulty levels: Easy, Medium, Hard.
4. Player chooses a Hero from the Hero list provided based on their preference.
5. Player chooses a Map from the Map list provided.
6. Player sets up the maximum time for the game by entering time in seconds.
7. The program checks the value entered for validity, asks until a valid response.
8. The program finishes updating the game and provides Player with the option "Start Game".

Entry condition Player has opened the game and chosen the "New Game" option.

Exit conditions Player is provided with a set-up game of their preference to play or not.

Use case name **SeeHelp**

Participating actors Initiated by Player

Flow of events

1. Player chooses the "SeeHelp" option from the main menu of the game.
2. The program creates another window with the user documentation provided in picture & text form in a scroll tab.
3. Player finishes observing the Help information and closes the window.

Entry condition Player has opened the game and chosen the "See Help" option from the main menu.

Exit conditions Player is provided with the user documentation.

Use case name	**SeeCredits**
Participating actors	Initiated by Player
Flow of events	1. Player chooses the “See Credits” option from the main menu of the game. 2. The program creates another window with the credits. 3. Player finishes looking at credits and closes the window.
Entry condition	Player has opened the game and chosen the “See Credits” option from the main menu.
Exit conditions	Player is provided with the credits.

Use case name	**ChangeSettings**
Participating actors	Initiated by Player
Flow of events	1. Player chooses the “Change Settings” option from the main menu of the game. 2. The program shows the available settings to the Player: volume & music 3. Player adjusts the settings according to their preference and chooses to save the settings OR chooses to go back to the main menu. 4. The program saves Player’s preferences if instructed. The program goes back to the main menu.
Entry condition	Player has opened the game and chosen the “Change Settings” option from the main menu.
Exit conditions	The changes have been made according to Player’s choice. Player has chosen to go back to the main menu.

Use case name	**StartGame**
Participating actors	Initiated by Player
Flow of events	1. Player chooses to "Start Game" after Game creation is complete. 2. The program sets all characters' health points and weapons to default values, starts the timer and starts the game. 3. Player starts to play the game with their Hero. 4. At any point if the Player's health points count reaches 0 OR if the timer finishes OR if all of the bots' health points counts have reached 0 OR if Player chooses to manually end the game, the program ends the game. 5. The player is directed to the results screen to see their ranking.
Entry condition	Player has already created a game of their preference using the "Create New Game" system and has chosen to start the game.
Exit conditions	Game is over according to the games' rules and the Player is provided with a results screen from which they can go back to the main menu.
Quality Requirements	• At any point while the Player is playing the game with their hero, this use case can include the **MoveHero** use case. The MoveHero use case is initiated when the Player presses a direction key. When invoked, the program moves the Player's hero according to their specification. • At any point while the Player is playing the game with their hero, this use case can include the **SetWeapon** use case. The SetWeapon use case is initiated when the Player presses one of the predetermined two weapon keys. When invoked, the program sets a weapon in the Game according to the user's specification. • At any point while the Player is playing the game with their hero, this use case can include the **EndGame** use case. The EndGame use case is initiated when the Player chooses the "End Game" option from the Game view. When invoked, the program manually ends the game and directs the Player to the results screen.

3.7 USER INTERFACE

3.7.1 Main Menu Screen

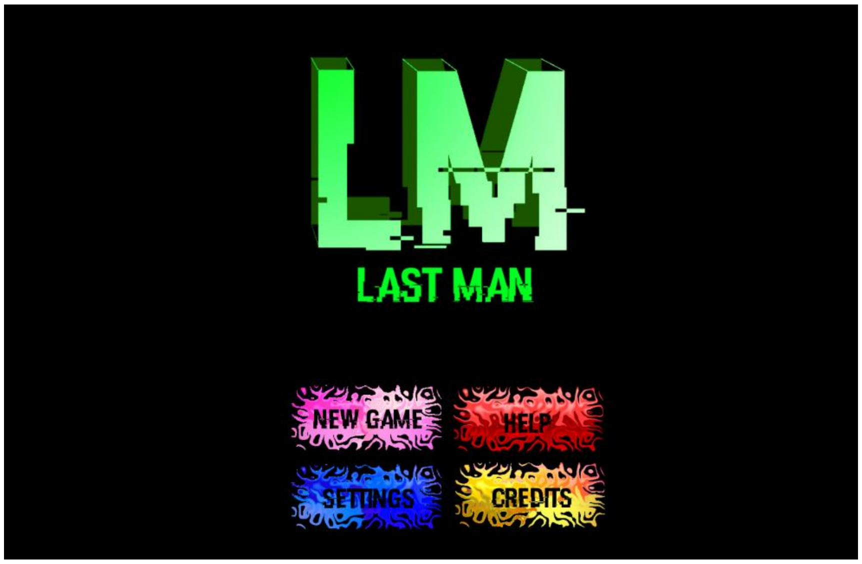
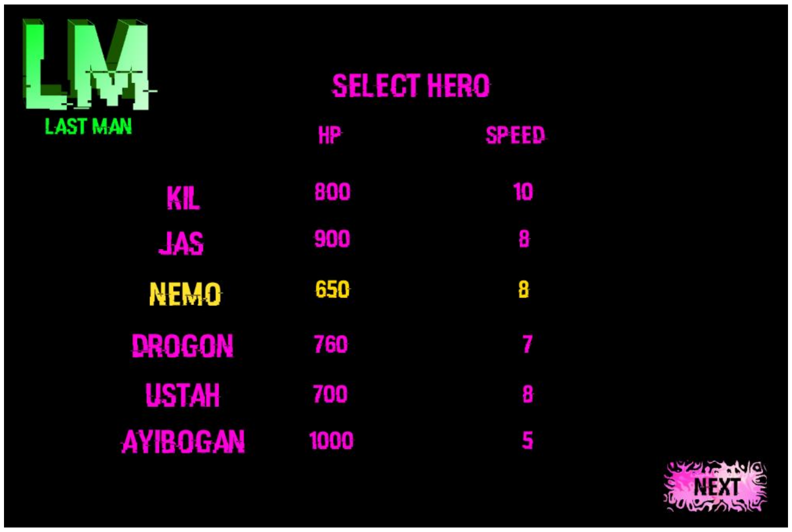


Figure 2 Main Menu Screen of Last Man

This is the main menu screen for our project. The user can click on buttons to navigate through the program.

3.7.2 Game Creation Screen



	HP	SPEED
KIL	800	10
JAS	900	8
NEMO	650	8
DROGON	760	7
USTAH	700	8
AYIBOGAN	1000	5

Figure 3 Hero Selection Screen

This is one example screen from the game creation process. The user can click on a hero and click next to select the hero.

3.7.3 Gameplay Screen

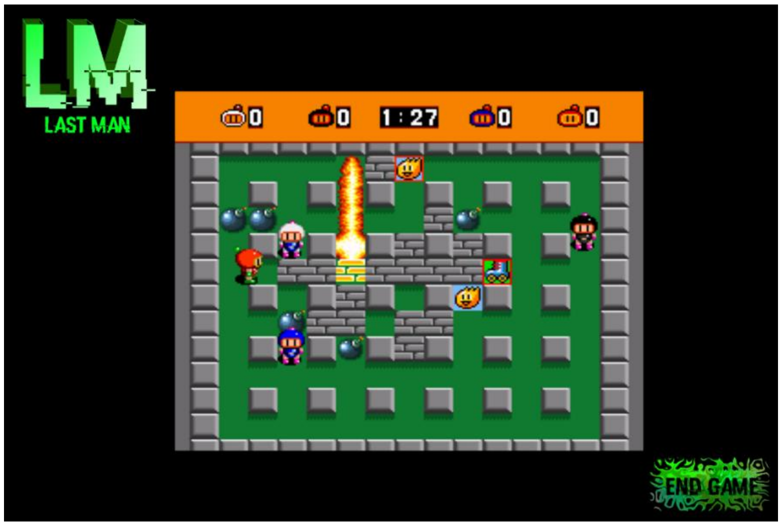
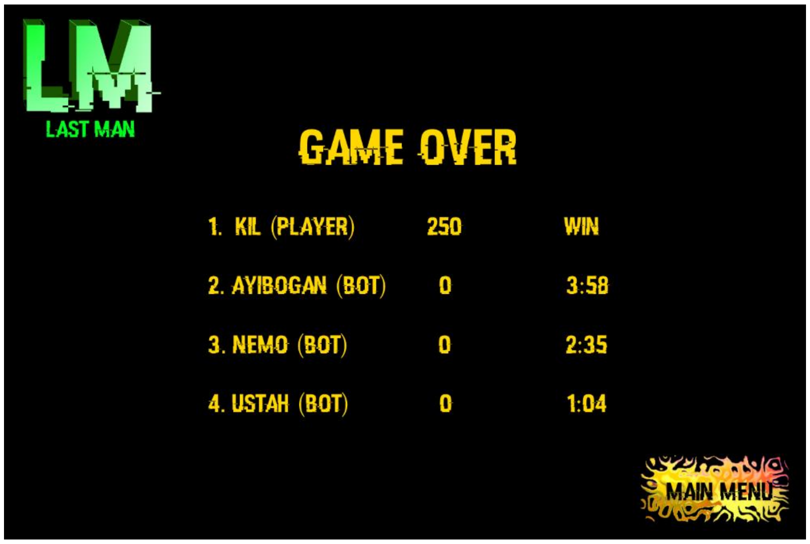


Figure 4 Gameplay Screen

The user navigates in the game play box via keyboard controls. The information on the bar in our program will be similar, except instead of zeros we will display HP counts[1].

3.7.4 Results Screen



GAME OVER

1. KIL (PLAYER)	250	WIN
2. AYIBOGAN (BOT)	0	3:58
3. NEMO (BOT)	0	2:35
4. USTAH (BOT)	0	1:04

Figure 5 Results Screen

This is the basic results screen. It appears after game ends. It displays the characters' remaining HP by the end of the game, and their death times.

[1] The inner picture is taken from http://gamefabrique.com/games/super-bomberman/

3.7.5 Settings Screen

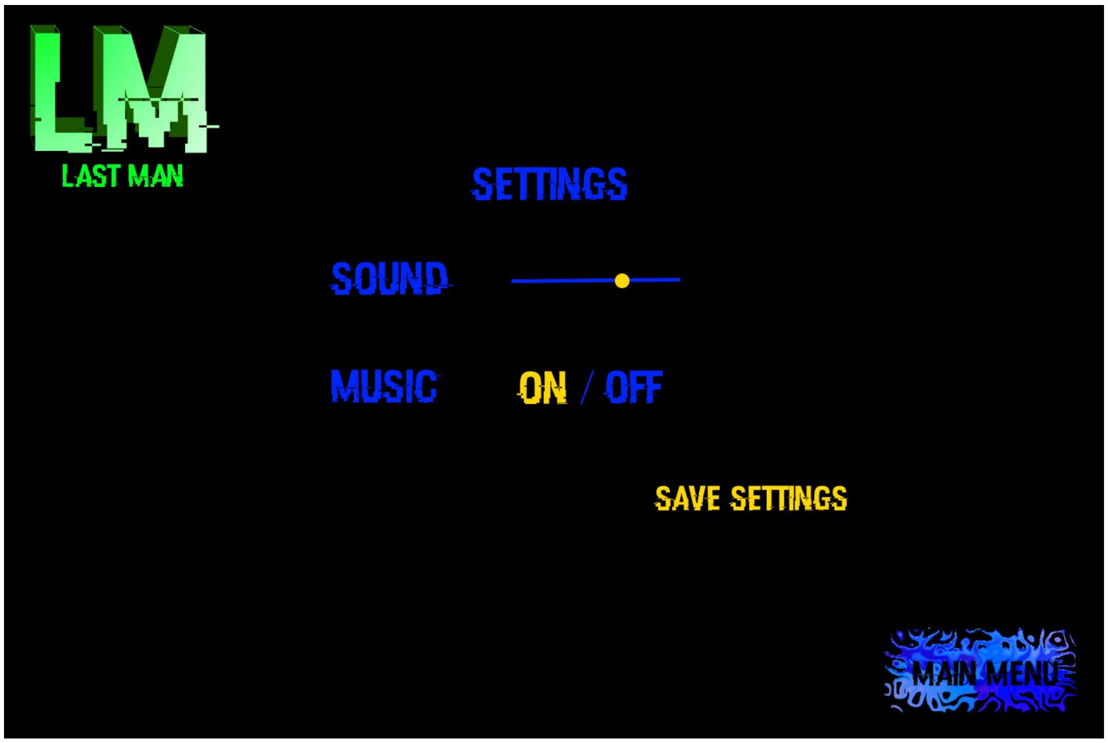


Figure 6 Setting Screen

This screen is for adjusting the game's settings. The user may choose to save the settings before going back to the main menu or not.

4 Analysis

4.1 Object Model

4.1.1 Class Diagram

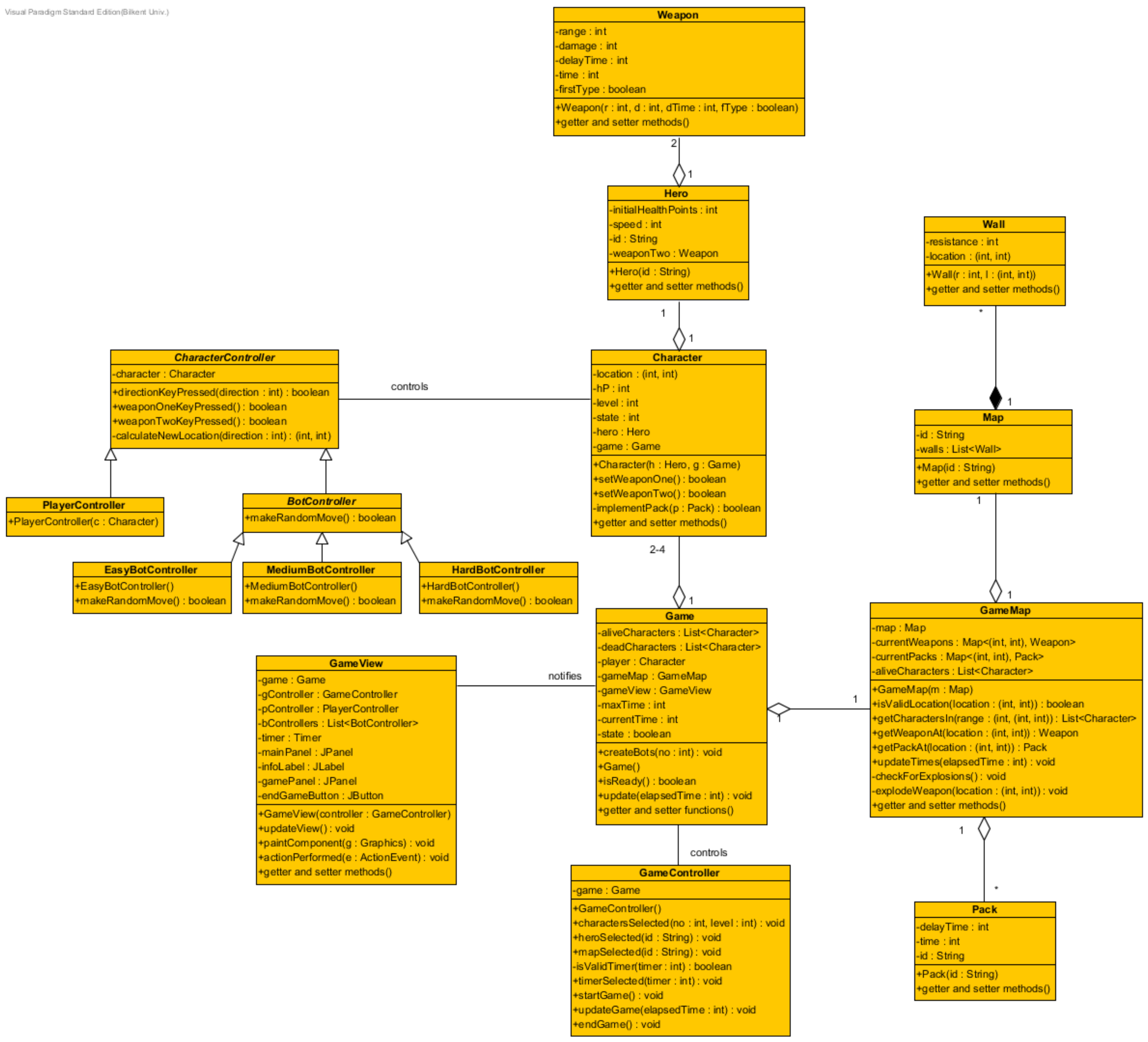

Visual Paradigm Standard Edition(Bilkent Univ.)
Weapon
-range : int
-damage : int
-delayTime : int
-time : int
-firstType : boolean
+Weapon(r : int, d : int, dTime : int, fType : boolean)
+getter and setter methods()
2
1
Hero
-initialHealthPoints : int
-speed : int
-id : String
-weaponTwo : Weapon
+Hero(id : String)
+getter and setter methods()
1
1
Wall
-resistance : int
-location : (int, int)
+Wall(r : int, l : (int, int))
+getter and setter methods()
*
1
Map
-id : String
-walls : List<Wall>
+Map(id : String)
+getter and setter methods()
1
1
CharacterController
-character : Character
+directionKeyPressed(direction : int) : boolean
+weaponOneKeyPressed() : boolean
+weaponTwoKeyPressed() : boolean
-calculateNewLocation(direction : int) : (int, int)
controls
Character
-location : (int, int)
-hP : int
-level : int
-state : int
-hero : Hero
-game : Game
+Character(h : Hero, g : Game)
+setWeaponOne() : boolean
+setWeaponTwo() : boolean
-implementPack(p : Pack) : boolean
+getter and setter methods()
PlayerController
+PlayerController(c : Character)
BotController
+makeRandomMove() : boolean
EasyBotController
+EasyBotController()
+makeRandomMove() : boolean
MediumBotController
+MediumBotController()
+makeRandomMove() : boolean
HardBotController
+HardBotController()
+makeRandomMove() : boolean
2-4
1
Game
-aliveCharacters : List<Character>
-deadCharacters : List<Character>
-player : Character
-gameMap : GameMap
-gameView : GameView
-maxTime : int
-currentTime : int
-state : boolean
+createBots(no : int) : void
+Game()
+isReady() : boolean
+update(elapsedTime : int) : void
+getter and setter functions()
GameMap
-map : Map
-currentWeapons : Map<(int, int), Weapon>
-currentPacks : Map<(int, int), Pack>
-aliveCharacters : List<Character>
+GameMap(m : Map)
+isValidLocation(location : (int, int)) : boolean
+getCharactersIn(range : (int, (int, int)) : List<Character>
+getWeaponAt(location : (int, int)) : Weapon
+getPackAt(location : (int, int)) : Pack
+updateTimes(elapsedTime : int) : void
-checkForExplosions() : void
-explodeWeapon(location : (int, int)) : void
+getter and setter methods()
1
1
GameView
-game : Game
-gController : GameController
-pController : PlayerController
-bControllers : List<BotController>
-timer : Timer
-mainPanel : JPanel
-infoLabel : JLabel
-gamePanel : JPanel
-endGameButton : JButton
+GameView(controller : GameController)
+updateView() : void
+paintComponent(g : Graphics) : void
+actionPerformed(e : ActionEvent) : void
+getter and setter methods()
notifies
controls
GameController
-game : Game
+GameController()
+charactersSelected(no : int, level : int) : void
+heroSelected(id : String) : void
+mapSelected(id : String) : void
-isValidTimer(timer : int) : boolean
+timerSelected(timer : int) : void
+startGame() : void
+updateGame(elapsedTime : int) : void
+endGame() : void
1
*
Pack
-delayTime : int
-time : int
-id : String
+Pack(id : String)
+getter and setter methods()

Figure 7 Class Diagram for "Last Man"

Above are the core Game-related functional classes for our project “Last Man”.

4.1.2 Domain Lexicon

GameView: This class is responsible for the display of the Game class and for interacting with it. It is responsible for providing GUI and sound elements for both the Game class and classes related to it.

GameController: This class is one of the most important classes of the project. It is used to take user input to control the game. This class is responsible for creating and setting up the game, to organize the game dynamics during the game, to finish a game and to direct the Player to the Results Screen at the end of the game.

Game: This is the core model Game class of our project. A game has a GameMap and 2 to 4 Characters. Basic functionalities of this class include creating and managing the characters and the GameMap, updating itself as time progresses and preparing the game to start playing.

GameMap: This represents the active map the characters are playing on during the game. It has a list of active planted weapons. It has an underlying Map structure that determines the initial default state of the GameMap. It also has multiple Packs randomly planted or to be planted. Some functionalities of this class are updating the times of planted Game objects, checking for explosions, and managing the locations of the Game objects.

Map: This is the underlying map structure. The map has a type that serves as an ID, so that players will be able to choose them during the "Create New Game" stage. The map consists of walls.

Wall: This class represents a one unit-area sized block. It has a resistance value and a location on the map.

Pack: This class represents a pack, that is a power-up or power-down. It has a location on the map, a type that serves as an ID to determine the effect and a delay time that represents the maximum amount of time it can spend on the GameMap without being picked.

Character: This class represents all active players in the game, including both the Player (the user) and the bots. A character has a location on the GameMap, a health points count, and a level that determines its' skill with picking up and using packs, and a state that determines whether it is allowed to use weapons or not. (In the case of the Player, this is automatically set to hard since the Player can use the packs freely). A character has one hero and one controller.

CharacterController: This is an abstract controller class for the Character class.

- **PlayerController:** This is used to control the user's character.
- **Bot Controller:** This is an abstract class used to control the bots' characters. It separates into 3 level-specific bot controllers.

Hero: This represents the playing individual of a Character. A hero has an initial health points count, a speed level and a type that serves as an ID to distinguish from other heroes. A hero has two weapons. One weapon is common to all heroes, whereas the other weapon is special to each individual hero.

Weapon: A weapon has a range that represents how much area it will affect. It has a damage value that represents how much harm the weapon will cause and a type that serves as an ID. Its delayTime attribute represents how often the weapon can be used if it's a special type weapon. If the weapons type is the default weapon (the bomb), the delayTime attribute represents after how many seconds the bomb will set-off.

4.2 Dynamic Models

4.2.1 State Chart Diagrams

4.2.1.1 Game State Diagram

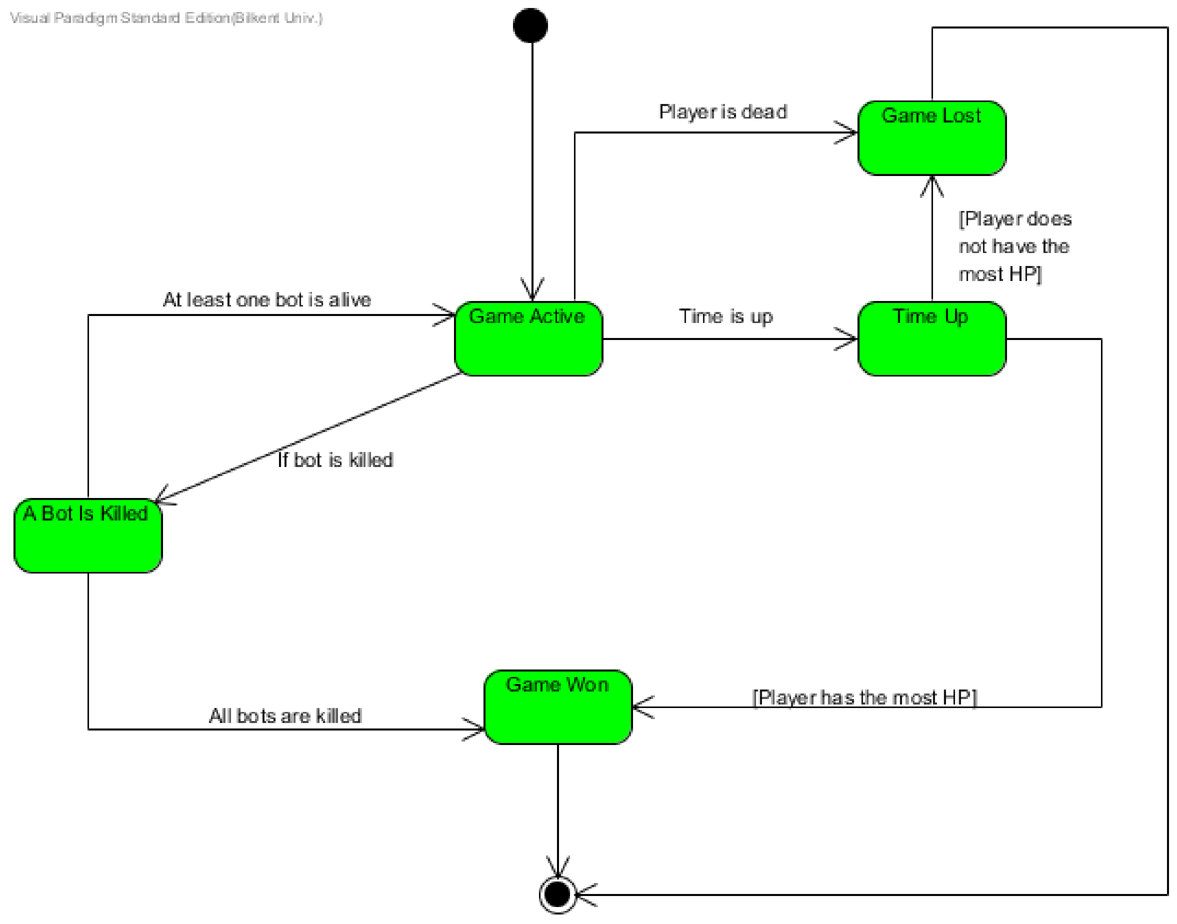


Figure 8 Game State Diagram

Last Man Survived starts with Game Active state and in this state characters can control their heroes. If player kills one of the bots, the game state is changed to A Bot Is Killed. If all bots are killed, the state is changed to Game Won state and then game ends. When player kills one bot and there is at least one bot alive, the state would change back to the Game Active state again. On the other hand if player is killed, the state is immediately changed to Game Lost without waiting for the death of other bots and game is ended. In addition to these, there is also another ending condition which is time. When time is up, state changes to the Time Up state. In this state, if player has the highest health points the state is changed to Game Won, whereas if the Player doesn't have the highest health points, the state is changed to Game Lost. Then the game is ended.

4.2.1.2 Character State Diagram

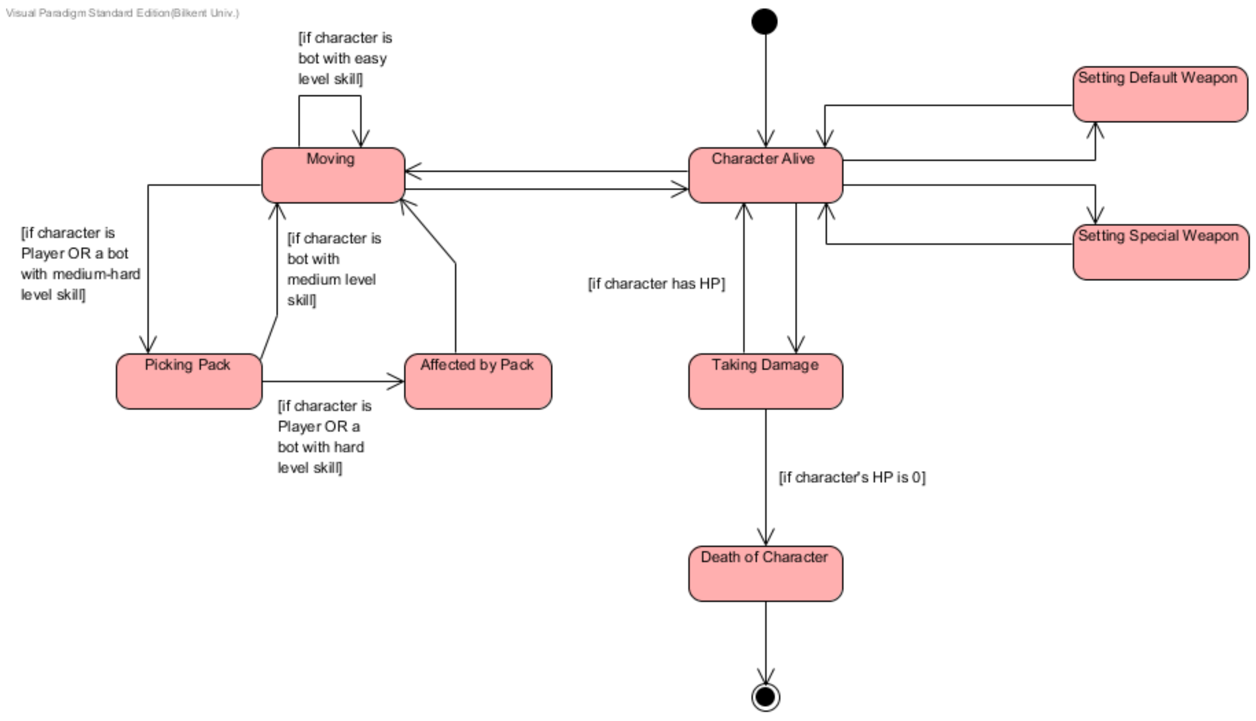

Visual Paradigm Standard Edition(Bilkent Univ.)
[if character is bot with easy level skill]
Moving
Character Alive
Setting Default Weapon
Setting Special Weapon
[if character is Player OR a bot with medium-hard level skill]
[if character is bot with medium level skill]
[if character has HP]
Picking Pack
Affected by Pack
Taking Damage
[if character is Player OR a bot with hard level skill]
[if character's HP is 0]
Death of Character

Figure 9 Character State Diagram

The character's default state is the Character Alive state. There are 4 states that the character can pass from this state. The character can set the default weapon and the special weapon. Then, state is changed to Character Alive again. In addition to using weapons, the character can move in the game map. While moving, the character comes across packs. If the character is a bot and the skill level of it is easy, the character stays in the moving state. Otherwise if the Bot's level is medium or hard or the character is the Player, the pack is picked. If the character is the Player or a Bot with the hard level, the pack affects the character. From the pack related states, the character goes back to the Moving state. If the character stops moving, it is returned to the Character Alive state. If character is affected by an explosion, the character is taken to the Taking Damage state. From this, if its' health points count has reached 0, the character is taken to the Death of Character state, otherwise Character returns to the Character Alive state.

4.2.2 Sequence Diagrams

4.2.2.1 *Create Game*

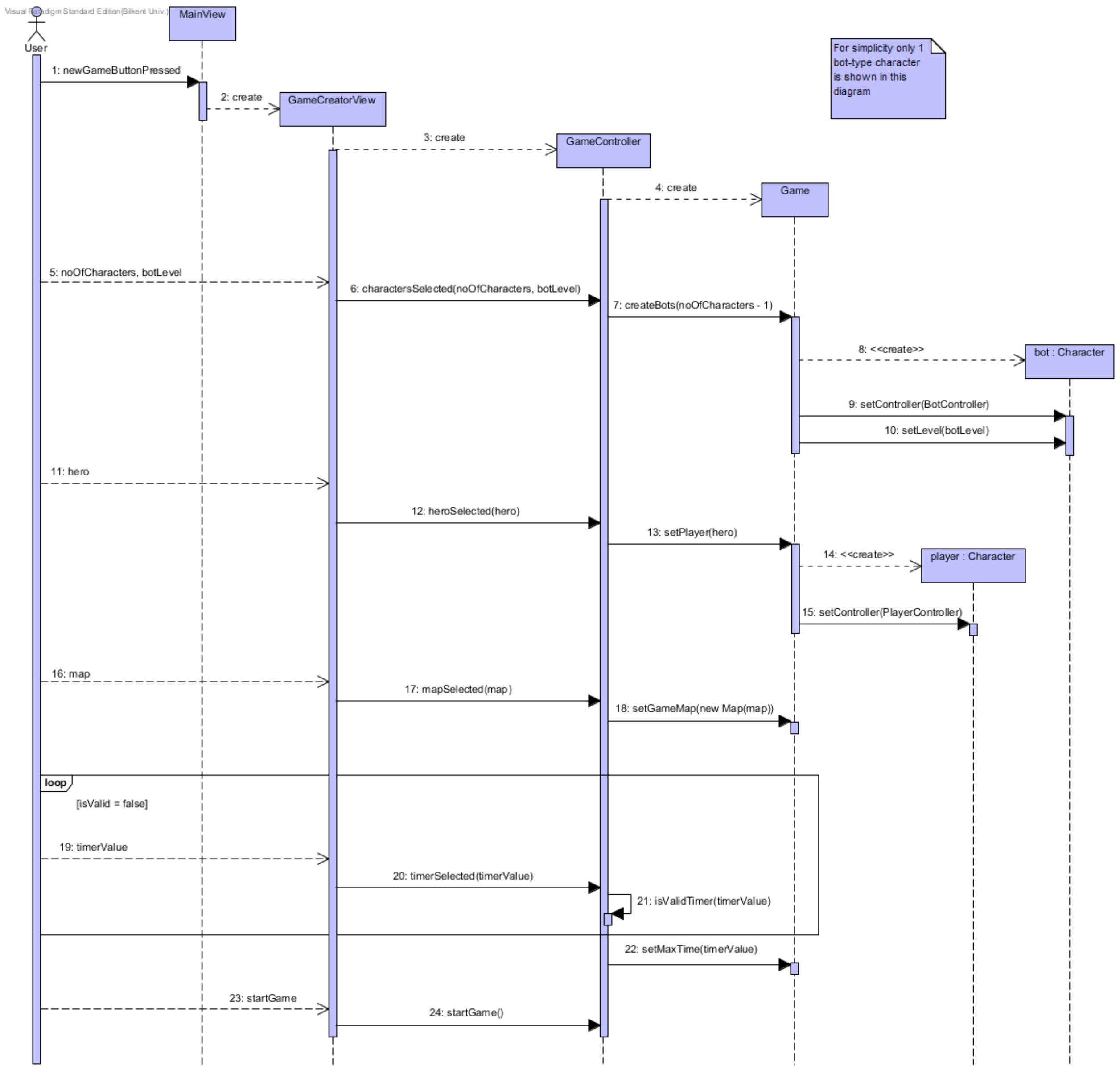

Visual Paradigm Standard Edition(Bilkent Univ.)
User
MainView
GameCreatorView
GameController
Game
bot : Character
player : Character
For simplicity only 1 bot-type character is shown in this diagram
1: newGameButtonPressed
2: create
3: create
4: create
5: noOfCharacters, botLevel
6: charactersSelected(noOfCharacters, botLevel)
7: createBots(noOfCharacters - 1)
8: <<create>>
9: setController(BotController)
10: setLevel(botLevel)
11: hero
12: heroSelected(hero)
13: setPlayer(hero)
14: <<create>>
15: setController(PlayerController)
16: map
17: mapSelected(map)
18: setGameMap(new Map(map))
loop
[isValid = false]
19: timerValue
20: timerSelected(timerValue)
21: isValidTimer(timerValue)
22: setMaxTime(timerValue)
23: startGame
24: startGame()

Figure 10 Create Game Sequence Diagram

In this sequence diagram, user interacts with GUI and presses the new game button from the MainView which will create a request for GameCreatorView to create a new game. GameCreatorView creates the GameController object which creates the Game object. The user enters the number of characters and bot levels as integer values through the GameCreatorView. With given data as a parameter, GameController tells Game to create bots, then Game creates the desired amount of Bots. After that, Game sets Bots by setting controllers. After assigning level of the Bots, GameCreatorView asks the user for hero selection and GameController takes the selection data to send request to Game for setting the hero type with given value as parameter and creates a player with the given hero. Having hero selection done, map type should be selected. The user returns map data to GameController through GameCreatorView and GameController asks Game to set its map type using given data. GameCreatorView waits for the user to enter time value for the timer and GameController checks if returned timerValue data is valid or not, this operation is repeated until timerValue is a valid value. Then GameController sets the maxTime of the Game to the timerValue. If the user chooses to start the game through GameCreatorView, GameController starts the game.

4.2.2.2 Character Movement & Pack Management

Scenario name	moveAndPickPack
Participating actor instances	lina: Player
Flow of events	1. Lina has opened the program, created a game and is currently playing the game. 2. Lina presses the up key from her keyboard. 3. The program calculates Lina's hero's new location considering her hero's speed and current location. The new location is in bounds and so the program updates the position of the hero. A pack exists in the new location and affects her hero.

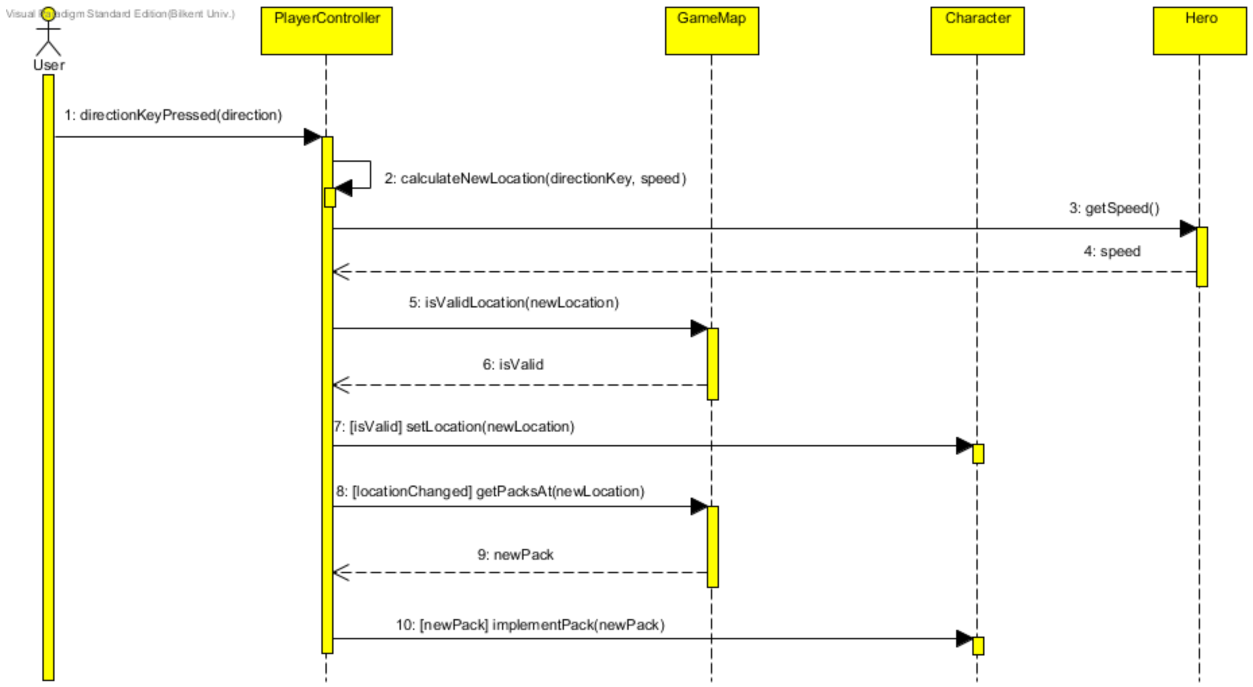

Visual Paradigm Standard Edition(Bilkent Univ.)
User
PlayerController
GameMap
Character
Hero
1: directionKeyPressed(direction)
2: calculateNewLocation(directionKey, speed)
3: getSpeed()
4: speed
5: isValidLocation(newLocation)
6: isValid
7: [isValid] setLocation(newLocation)
8: [locationChanged] getPacksAt(newLocation)
9: newPack
10: [newPack] implementPack(newPack)

Figure 11 Move and Pack Sequence Diagram

This sequence diagram analyzes the movement and pack management of a character. It starts with the user pressing a direction key. When User presses a direction key, they warn the PlayerController by telling a direction key is pressed. After that, PlayerController internally calculates the new location of the character. Since speed abilities of heroes vary, PlayerController asks Hero for the speed value and Hero returns the value. PlayerController calculates the new location with the given direction and speed and asks GameMap if given newLocation is a valid location or not and takes the answer in isValid stored as a Boolean. If isValid is true, PlayerController asks Character to move to the new location. If the location is changed, PlayerController tells GameMap to check for packs in the newLocation. If there is a new pack found, PlayerController tells Character to implement the pack.

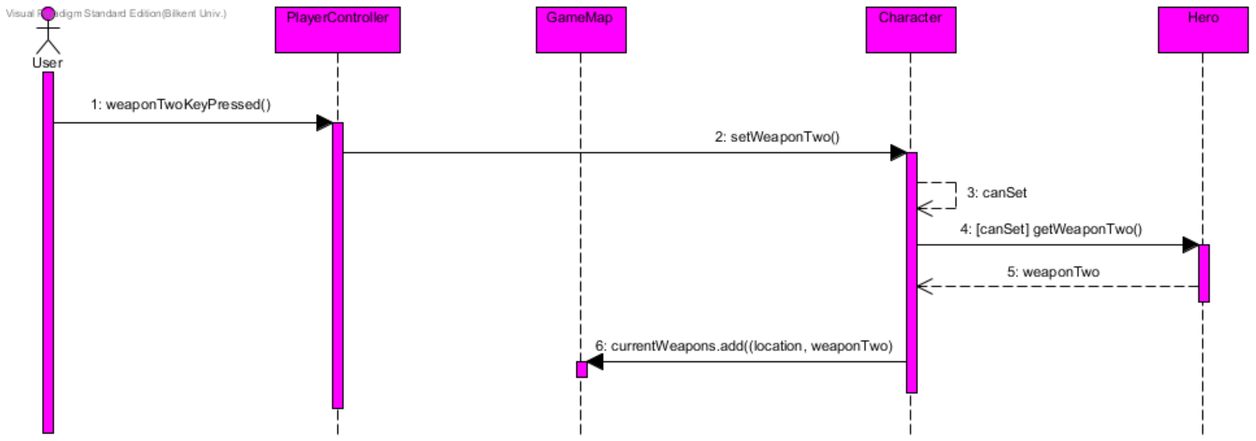
4.2.2.3 Using Special Weapon

Scenario name useWeaponTwo

Participating actor instances oz: Player

Flow of events

1. Oz has opened the program, created a game and is currently playing the game.
2. Oz presses the 'A' key from his keyboard.
3. The program calculates if Oz is allowed to use another weapon. Oz is allowed to do so, so his hero's special power is activated.



Visual Paradigm Standard Edition(Bilkent Univ.)
User
PlayerController
GameMap
Character
Hero
1: weaponTwoKeyPressed()
2: setWeaponTwo()
3: canSet
4: [canSet] getWeaponTwo()
5: weaponTwo
6: currentWeapons.add((location, weaponTwo)

Figure 12 Use Special Weapon Sequence Diagram

Using Special Weapon sequence diagram simply explains how special weapon usage works. First, the user presses the WeaponTwoKey and sends information to PlayerController that weapon two key is pressed. PlayerController asks Character to set the desired weapon. For this, Character checks if it is allowed to use weapon two (weapons have delay times). Character checks its state and provides itself with a Boolean canSet value. If canSet is true, Character gets its weaponTwo from its Hero and requests GameMap to set the weapon in its current location.

4.2.2.4 Updating Time & Explosions & Death

Scenario name oneSecondPasses

Participating actor instances gon: Player

Flow of events

1. Gon has opened the program, created a game and is currently playing the game.
2. One second passes in the game.
3. The game updates itself, a bomb near Gon explodes and Gon's character dies. The game ends.

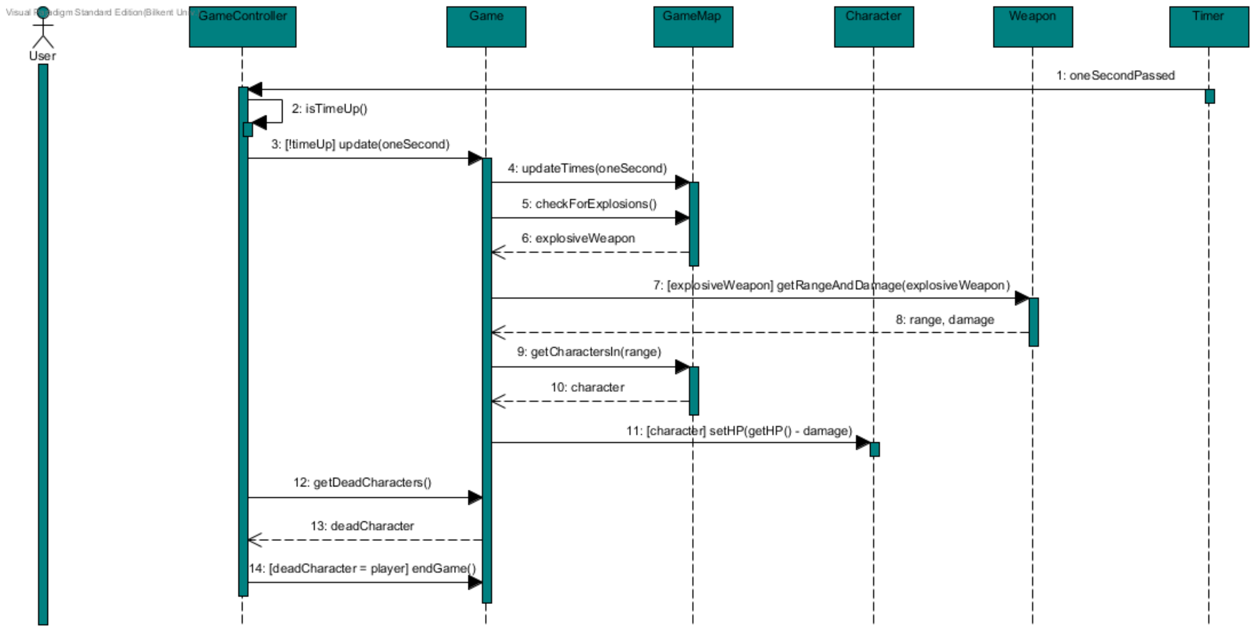

Visual Paradigm Standard Edition(Bilkent Un
User
GameController
Game
GameMap
Character
Weapon
Timer
1: oneSecondPassed
2: isTimeUp()
3: [!timeUp] update(oneSecond)
4: updateTimes(oneSecond)
5: checkForExplosions()
6: explosiveWeapon
7: [explosiveWeapon] getRangeAndDamage(explosiveWeapon)
8: range, damage
9: getCharactersIn(range)
10: character
11: [character] setHP(getHP() - damage)
12: getDeadCharacters()
13: deadCharacter
14: [deadCharacter = player] endGame()

Figure 13 Time and Explosions Sequence Diagram

This sequence diagram starts with Timer warning GameController by telling one second passed. GameController checks if the time is up after that and if not, updates the Game as one second passed. Then Game updates times of game objects in the GameMap with the one second passed info. Then Game checks if there are any explosions. GameMap returns if the ready explosions. If there is one, Game wants the range and damage information of that explosive weapon from Weapon. Range and damage data are transferred to Game. Game asks the GameMap if there are any characters in the explosion range. GameMap returns the characters in the range to Game. If there is a returned character, Game tells Character to inflict damage to the character. After infliction of damage, GameController asks game to return dead characters if there are any. Game returns if any. GameController checks if the deadCharacter is Player, and if the condition is true, GameController ends the game.

4.2.2.5 See Credits

Scenario name seeCredits

Participating actor instances gilbert: User

Flow of events

1. Gilbert has opened the program and chosen to see the credits.
2. He clicks on the “Credits” button from the main menu.
3. He observes the credits in the new window.
4. He closes the window.

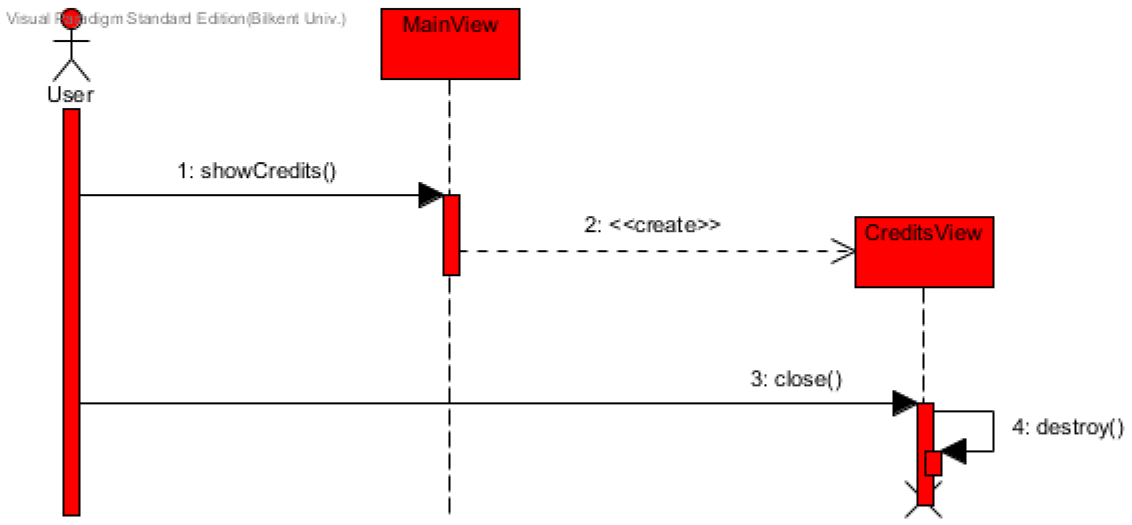


Figure 14 See Credits Sequence Diagram

See Credits sequence diagram basically provides credits info for user. First of all the user uses MainView GUI and requests from it to show credits. MainView creates a CreditsView. Then with User's attempt to close the CreditsView, window destroys itself and closes.

5 Analysis Conclusion

In this analysis report, we focused on the basic functionalities and design issues of our project. This helped us to consider the scope of our project and the methods we will use as well as the underlying programming structure.

In Requirements Specification, we tried to specify all the requirements for our project, both functional and non-functional. We also specified the design constraints for our project. It is our goal to fulfill the requirements we mentioned.

In the Use Case model, we wanted to explain the basic actions the user can take by writing scenarios and use case descriptions. We also made a class diagram for the fundamental objects in our project to describe their attributes and relations. We wanted to map the use cases we described to our objects using our sequence diagrams. We wanted to make the game states more clear so we prepared our state diagrams. We used VisualParadigm to create all of our diagrams.

We also aimed to keep our graphical user interface stylish, simple and easy to use.

We wanted to create a detailed and organized analysis report for both ourselves, so that we can handle the further steps of the project more easily, and for the reader, so they have a more concrete idea of our project.

Design

6 Design Goals

6.1 Performance

- **Response Time:** Last Man has an interactive and responsive system, it handles the input coming from the user. Therefore the response time of the game is very important. We will address this concern specifically in section 10.5.

- **Memory:** Memory usage is an important factor for a game, but Last Man will not require huge amounts of memory since its graphics are very simple and there is no storage of in-game data. This is because Last Man does not consist of different levels and therefore there isn't any save game – continue game functions. So it is safe to say that memory will not be a problem for Last Man. We will use image and sound files that have small sizes to ensure that Last Man won't have memory problems. The persistent data files are .txt files that are also very small in size.

6.2 Dependability

- **Robustness:** Last Man will have specified key configurations. Any pressed key other than those specified keys will not be taken as an input. All game options and program settings are selected from a predetermined set of values. This will prevent the user from entering invalid input, so this will keep the system robust.

- **Reliability:** Unexpected errors cause systems to become less reliable, but in Last Man, as the inputs given to the game are very expectable, these exceptional cases are really hard to occur. As the user chooses different options, the system will work smoothly. However in case some problems occur, we will design our program so that our system can be shut down manually. Also data loss will not be a problem since the game does not have a save/load system. In order to ensure reliability, we will focus on making user choices dependent on a given menu.

- **Fault Tolerance:** The system should handle errors and exceptions properly and this is because the gameplay of Last Man can be affected from simple problems. The game should not crash and therefore fault tolerance is one of the key design goals of our project. In order to ensure this, we will do extensive testing and exception handling.

6.3 Cost

There will be no developing, deploying, upgrading, maintenance or administration costs associated with our project.

6.4 Maintenance

- **Extensibility:** The source code of the Last Man project will be designed in a way such that it permits adding some new features to the game including new heroes, maps or game types etc. We believe the design we have currently provides this opportunity since we tried to keep our classes separated and organized. Our MVC pattern will also help with extensibility sufficiently.

- **Modifiability:** Modifying some components of the Last Man to make the game more functional or more playable for the user will be easy. Again, the object oriented design structure we have should make this fairly easy since our entities are systematized. Adding new classes will be fairly easy considering our MVC design.

- **Readability:** While creating the Last Man project, we will code the game attentively. Source code of the system should be neat for other developers to read and understand easily. This will make it easier for other developers to improve our system. We will pay attention to commenting and following the Java coding conventions to address this problem.

- **Supportability:** We want our game to reach a lot of users, so our game should be compatible with most computers. To achieve this we will implement the system with an appropriate programming language, such as Java, so that our game can be played on any desktop or laptop computer with any operating system.

6.5 END USER

- **Usability:** Last Man can be played by users from every age group, so it should be easy for all users to learn and understand the instructions. Our system will be user-friendly, so the user interface will be made in a simple and consistent way. Users with minimal gaming or computing experience can also enjoy the game because of the documentation provided in the help section of the game.

6.6 TRADE-OFFS

- **Functionality vs Usability:** Last Game is intended to be user-friendly and easy to use for all groups of users. For keeping the Usability goal steady, we will create a simple system. To prevent users from playing Last Man without difficulties, the system will be implemented with simple functions which sometimes may limit the functionality of the program.

- **Rapid Development vs Dependability & Maintenance:** Dependency and maintenance are key concepts. To keep the system robust, reliable and fault tolerant, we will minimize the errors in the system. Again, for extensibility, modifiability and readability goals, we will implement the system cautiously. So, to reach the dependability and maintenance goals, we should develop Last Man carefully and somewhat slower than rapid development.

- **Efficiency vs Portability:** For the portability goal, we will code the system with a programming language that is compatible with most computers (e.g. Java). Although implementing Last Man system with Java will provide us more portability, it will reduce efficiency of the system since Java is not the most efficient programming language.

7 SUBSYSTEM DECOMPOSITION

In our project we have the following subsystems:

- UserInterface subsystem: This subsystem realizes the user interface for the Player.
- ProgramController subsystem: This is the core controller subsystem of our project. It is responsible for managing the sequence of interactions with the user. It is broken down to smaller subsystems for ease of design:
    - GameController subsystem: This subsystem is for managing the interactions that affect game creation and play. It is responsible for the creation and the modification of the Game entity.
    - SettingsController subsystem: This subsystem is responsible for creation and modification of the Settings entity.
- ProgramModels subsystem: Each of the subsystems below are responsible for maintaining domain knowledge about the related entities.
    - GameModel subsystem
    - SettingsModel subsystem
    - HelpAndCreditsModel subsystem

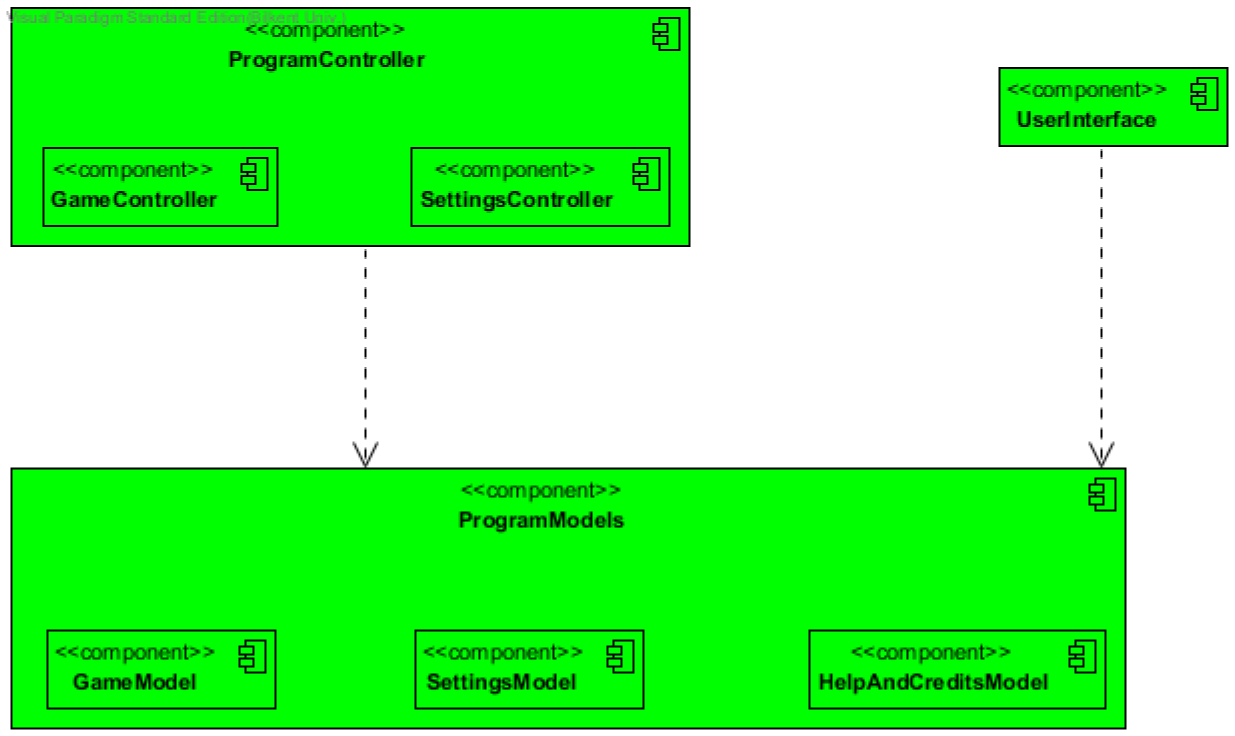


Figure 15 High Level Representation of System Decomposition

7.1 UserInterface Subsystem

This subsystem is responsible for handling user interaction with the Player, namely the boundary objects. It maintains the view classes which are responsible for displaying the entities to the Player and creating and managing some of the controller classes for these entities.

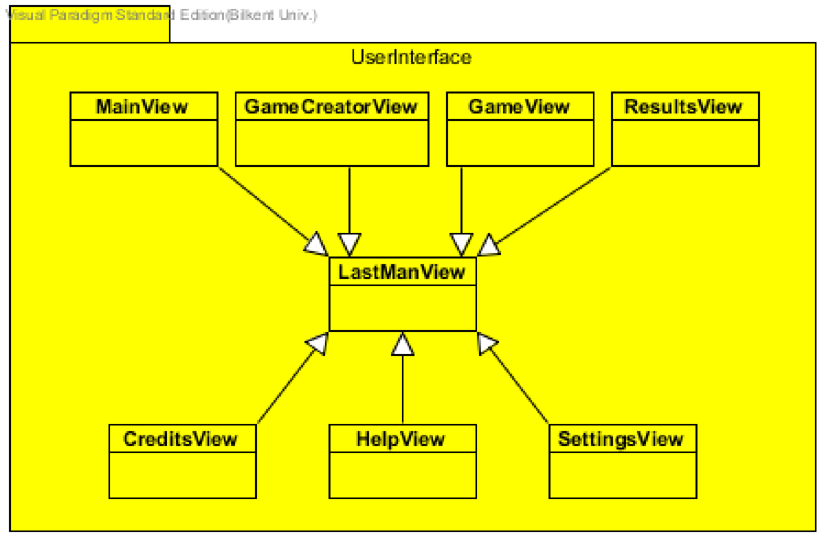


Figure 16 The UserInterface Subsystem

All views implement the LastManView class for similarities. These views also include various type of panels, buttons and other graphical elements.

7.2 GameController Subsystem

This subsystem is responsible for handling the control objects for the Game entity.

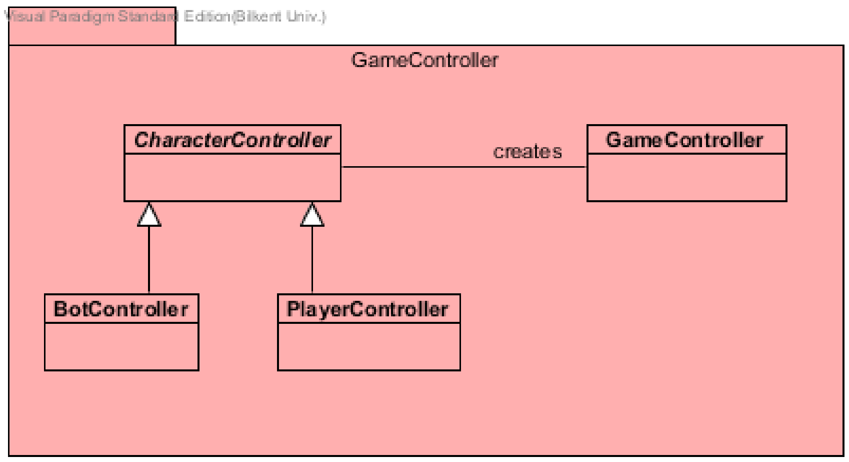


Figure 17 The GameController Subsystem

7.3 SettingsController Subsystem

This simple subsystem includes the control object for the Settings entity. This subsystem and the following three subsystems are separated for organization purposes rather than workload or complexity reasons.

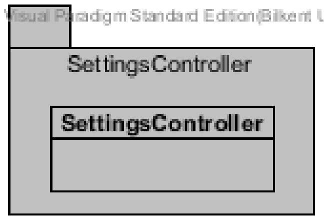


Figure 18 The SettingsController Subsystem

7.4 SettingsModel Subsystem

This simple subsystem handles the Settings entity.

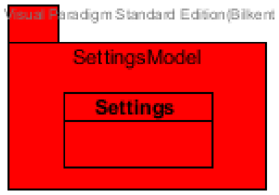


Figure 19 The SettingsModel Subsystem

7.5 HelpAndCreditsModel Subsystem

This simple subsystem maintains the Help and Credits Entities.

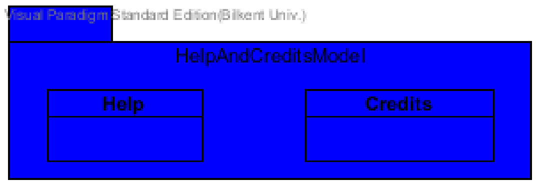


Figure 20 The HelpAndCreditsModel Subsystem

7.6 GameModel Subsystem

This subsystem represents the entity objects related to the main Game logic.

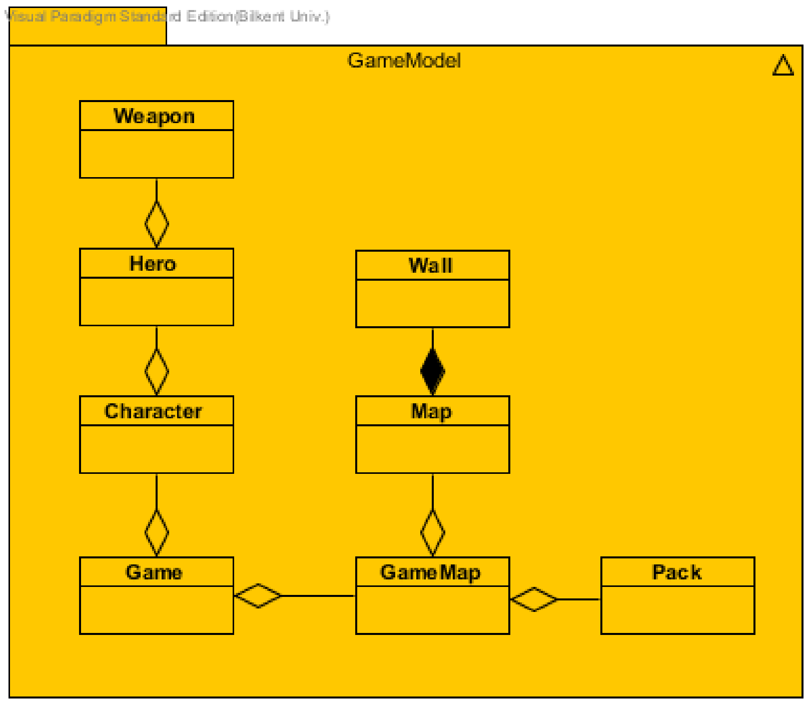


Figure 21 The GameModel Subsystem

The GameModel subsystem consists of the entity classes related to the Game entity.

8 Architectural Patterns

In our system we decided to use the MVC (Model/View/Controller) architectural style for organization, ease of use and compatibility reasons. Because we do not work with networking, with managing large amounts of data, with data that constantly changes or needs to be processed in complex manners, with a system where persistent data needs to be accessed by multiple applications or with a system where there needs to be more than one form of user interface for an application, we decided to not use architectural patterns like Client/Server, Repository or Three-tier. We work with very limited persistent data and our system is an interactive one, so we thought using MVC was consistent and appropriate. Since the game is quite simple and there aren’t a large number of classes, the performance bottleneck caused by the architecture’s nature will not be a problem. Our model subsystems will maintain the domain knowledge, namely qualities of the active game and other entities like settings. The view subsystem will consist of the view classes that are responsible for displaying the program to the user and for maintaining user interaction. The controller subsystems are responsible for altering the model classes using information provided from the user. The model subsystems are independent of both the view and controller subsystems. When a model entity changes as a result of a controller action, the changes are observed by the user with help of a subscribe-notify protocol between the view and model classes.

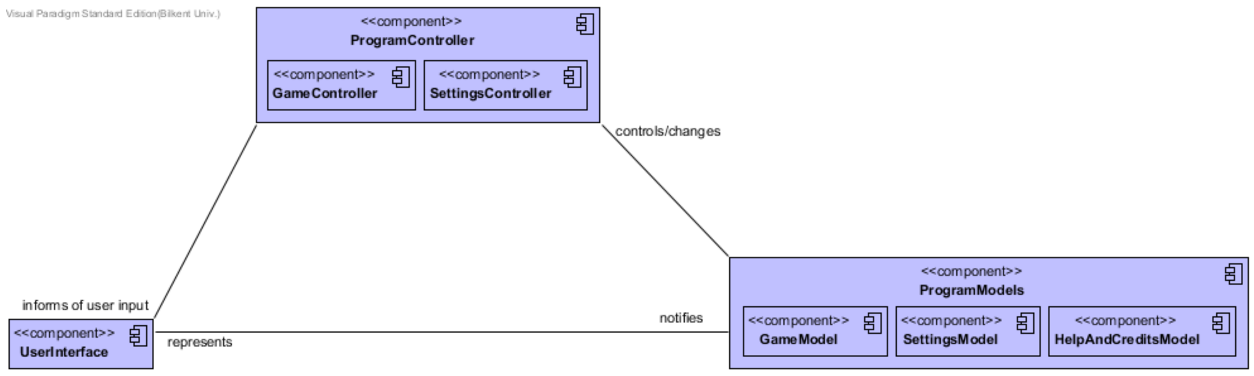


Figure 22 The MVC Architectural Pattern of LastMan

9 HARDWARE-SOFTWARE MAPPING

Last Man is a system that will work on one desktop or laptop computer and it is controlled by one actor. There are no multiplayer options or no network connections. In the below diagram, Last Man represents the main application logic of our program, basically the entity and control objects. We will use the Java Swing library to implement the user interface which is dependent on the JRE package. Therefore, our application logic component Last Man is dependent on the Java Swing component. Last Man also depends on the persistent data files we need for the program to function properly.

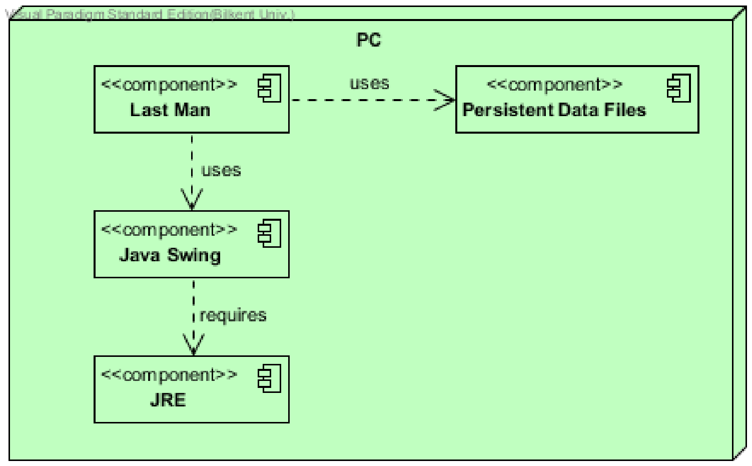


Figure 23 Last Man Deployment Diagram

In terms of mapping objects onto hardware, we can map our controller objects onto the processor. The program is fairly simple so the computations required aren't too demanding for a single processor. There is almost no occurrences of concurrency so using multiple processors is not necessary. We can map our model objects onto the memory. Again, there isn't considerable memory requirements for the program. We can map our view objects onto the I/O devices. We require a screen, a mouse, speakers and a keyboard for the program to function fully.

We will use the Java programming language to implement our project for a number of reasons: familiarity, ease of use, and ability to produce something that will work on multiple operating systems. Our software will be presented as a single executable file.

10 ADDRESSING KEY CONCERNS

10.1 PERSISTENT DATA MANAGEMENT

Last Man is a non-level based game where the player creates a game of their choice and plays it out. The game cannot be paused and once the game finishes and the user decides to go back to the main menu from the results screen, all data about the recently played game is discarded. All individual games are independent from one another. Therefore holding persistent data about user progress is not necessary. However, there is still some nonvolatile data that needs to be kept for the game to function properly. All these data needs to be accessed only when the program is active. We do not need to access the program data concurrently and there will not be multiple platforms using the same data. We also do not need to perform complex queries and we are not working with a large data set. Therefore, we have decided to use flat files to store our persistent data rather than database systems. These files will be stored in the user's hard disk drive and they will be accessed by the program when necessary.

We divide these data into two categories depending on their accessibility. The first group includes data that will be stored with no encryption and that will be user accessible and modifiable by hand, not from the scope of the LastMan program. These include graphical user interface images and sound files. The user can modify these files in order to alter the program's graphical design according to their personal preference. The second group consists of data that is user-inaccessible. These include definition of maps and heroes. Remember that while creating the game, the Player was asked to choose a hero and a game map from determined lists of heroes and games. Information for these lists will be encrypted into files and it will be accessed by the program when necessary. These files are read only, and they are constant.

10.2 Access Control and Security

The Last Man game is not a multi-user system. It is an offline single-player game. Therefore, all users interacting with the game will have access to the same functionality and same data. The only actor in the system is the Player so there will not be not be any user authentication system. The persistent data that is accessible (e.g. sound files) can be modified by everyone who knows the directory of these data files. However, these files will not be accessible while the program is running and they have to be altered competently to not disrupt the program. The persistent data that is inaccessible by the users will only be accessible by the program's controller classes and they will be read-only files. Also, the volatile in-game data is controlled in an organized manner as a result of our MVC architecture. The program will not request access to any personal computer data that is not related with itself. Since this is a simple offline system that requires no user authentication, security issues will not be a problem.

10.3 Global Software Control

The control flow mechanism for Last Game is event-driven control. The system will wait for an external event and whenever an event becomes available (e.g. pressing a keyboard button), the appropriate controller class will update the related model class which will in turn will notify the associated view classes.

In our program, the control is distributed between many controller objects. These distinct objects are all responsible for handling different events and modifying different entities. Therefore our project will have decentralized control design. We didn't prefer the centralized design because having various kinds of entity objects in our program, it is easier and more appropriate to control events in smaller pieces. We also believe decentralized design goes along well with the MVC architecture of our project and object-oriented development.

10.4 Boundary Conditions

In this section we model the boundary conditions using use cases.

Use case name	**InitializeProgram**
Participating actors	Initiated by User
Entry condition	1. The user runs the program executable file and starts the program.
Flow of events	2. The program invokes the CheckData use case mentioned in the following section. 3. The program uses the GUI-related persistent data to create the appropriate views and uses the sound-related persistent data to prepare appropriate audio elements. 4. The program provides the user with the "Main Menu Screen" which consists of the options "Play Game", "Settings", "Help" and "Credits".
Exit conditions	5. The program is prepared and the User can initiate any of the use cases mentioned before.

Use case name	**TerminateProgram**
Participating actors	Initiated by User
Entry condition	1. User clicks on the close button in the main frame of the program. The program asks for confirmation via a pop-up dialog, and the User confirms termination.
Flow of events	2. The program stops any ongoing game and destroys all non-persistent objects. 3. The frame closes.
Exit conditions	4. The program has closed and the user can no longer interact with the program.
Notes	Single program subsystems are not allowed to terminate the program. Since no in-game data is saved in the project, no subsystem notification or file writing occurs

Failure Use Cases

There are two main failure cases that can occur in our program. These are:

- A data failure where a user-accessible persistent data file is corrupted
- A failure where the user tries to run the program executable while there is an active program window

Use case name	**CheckData**
Participating actors	Initiated by LastMan
Entry condition	1. The InitializeProgram use case is invoked.
Flow of events	2. LastMan checks for the integrity of the user-accessible persistent data by checking the contents of the image and sound files. 3. If any file is corrupted LastMan provides an error message in a pop-up window and terminates itself. Otherwise the InitializeProgram use case is continued.
Exit conditions	4. Either LastMan is terminated or InitalizeProgram use case is continued.

Use case name	**TerminateActiveProgram**
Participating actors	Initiated by LastMan
Entry condition	1. The user runs the program executable file while there is an active program running.
Flow of events	2. LastMan terminates the active program.
Exit conditions	4. The active program is terminated and InitializeProgram use case is invoked by the user for the secondary window.

10.5 Response Time

One other important concern about LastMan is the response time of the game. A game should response as quickly as possible because it affects gameplay directly. To make the users pleased with our game, we will attempt to reduce the response time.

To decrease the response time, we used the MVC architectural pattern which should work smoothly. The view part of MVC enables the program to get only the necessary data which enables the system to keep away from redundant data. The controller system of the design ensures only relevant views are active.

In addition to these, we separated our classes to reduce the work load. Also, we will try to optimize our implementation of the source code as much as we can. To do this, we can use additional libraries to make our program faster or we can use additional faster input and error handlers in our system. We can also test for the fastest and most appropriate data structures.

Another factor which affects response time of the system is graphics. We did not use high quality images and sounds. We preferred to use basic images and icons to initialize the game map and views. We also did not use game effects in our game.

With all these efforts, we hope our game will not have any lag or response time issues.

11 Design Conclusion

In this design report, we focused on the structural properties of our project. This helped us to think about the implementation process.

In Design Goals, we specified several quality requirements we wish to fulfill.

In Subsystem Decomposition, we tried to break our system into smaller parts for ease of implementation and design. Looking at our subsystems, we decided that the MVC architectural pattern would fit the Last Man project the best. We also tried to reevaluate our hardware and software requirements. We addressed our concerns about data handling, access control and software control. Upon consideration of many different methods, we decided to store our data using files and have a decentralized control mechanism. Finally, we used boundary use cases to further clarify the initiation and termination of our program and some exceptional points.

We think the preliminary work we have done for this report will help us greatly during the implementation stage.

Object Design

12 OBJECT DESIGN

12.1 PATTERN APPLICATIONS

12.1.1 Composite Design Pattern

In LastMan when we try to create the GUI, we aim to treat individual objects and compositions of these objects uniformly. As our GUI objects become more complex and include other GUI objects, a specific design pattern is needed to manage the entities. This pattern should also support any GUI hierarchy with arbitrary depth and width.

We're going to use the Java Swing Library for the user interface of our game. Java Swing Library provides us JFrames, JLabels, JPanels and JButtons etc. to create the views of our game. Our game has views like MainView, ResultsView, GameView etc. These views are all JPanels and are all composed of Java Swing Library Components and varying composition of those components. To address this situation, Java Swing uses the Composite Design Pattern.

Composite design pattern provides a superclass for aggregate and leaf nodes. By using this pattern, we can add new types of nodes without modifying the existing code.

In our implementation of Composite Design Pattern, the superclass is JPanel because it will contain all the other components, leaves are JButton, JLabels and etc. Also there is 1 to many relationship between JPanel and JComponent.

Below is an illustration for this design pattern.

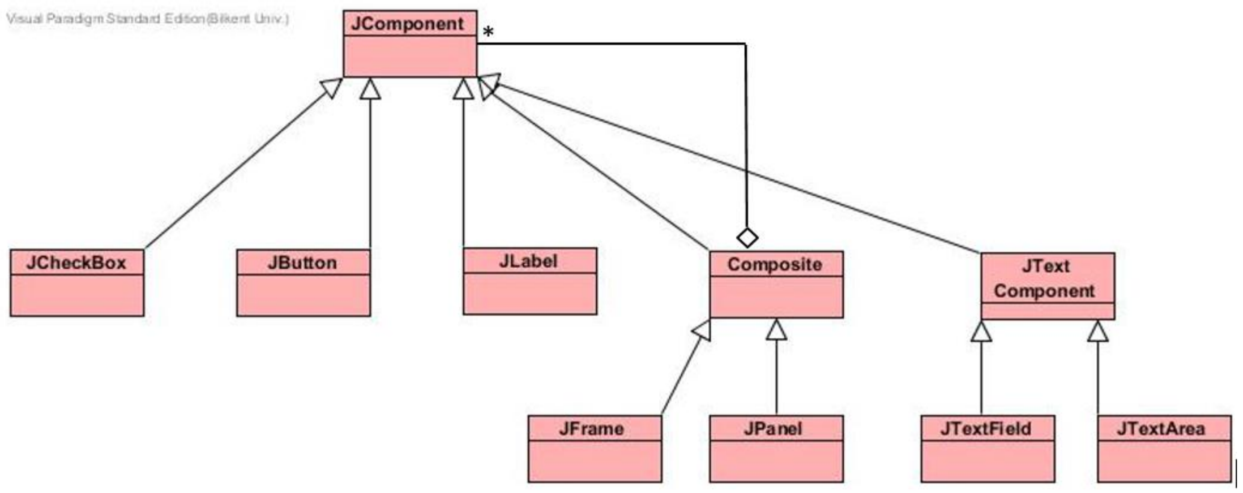


Figure 24 Illustration of the Composite Design Pattern

12.1.2 Strategy Design Pattern

In Last Man, we have 3 different difficulty levels for bots. These are easy, medium and hard. Easy bots, can move and can deploy bombs but they cannot take packages. Medium bots can move, deploy bombs and they can take packs but they cannot use them. Hard bots are the most developed bots. They can move, deploy bombs and take packs, they can also use these packs. The quality of the moves made by different level bots are also different.

All these different level bot methods are implemented using different algorithms, and according to the user choice at run-time, we would like to use these algorithms in an interchanging manner. Therefore we need to decouple the algorithm from its implementation. To achieve this, we can use the Strategy Design Pattern.

The Game class will decide which specific controller the specific bot character will use according to user specifications. Below is a sample illustration for this pattern.

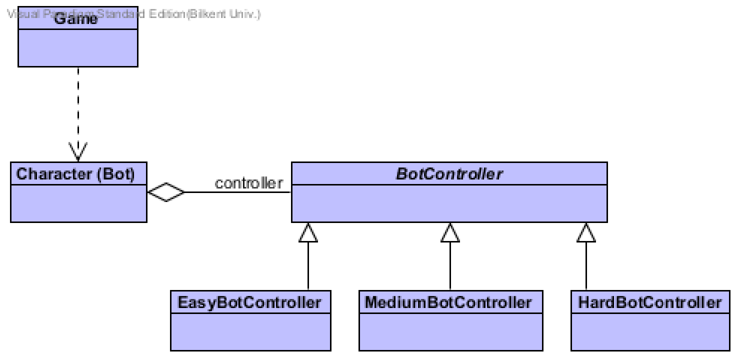


Figure 25 Illustration for the Strategy Pattern

12.1.3 Façade Design Pattern

In Last Man, there are a variety of classes, each having different complicated operations. However, not all classes need to know about the complexities of the sub-end-classes such as Weapon or Hero. These higher-level classes are mainly the View classes. To hide the complexities of the model-subsystem classes from the view-subsystem classes, we can implement Façade classes in between so that the model classes are easier to use.

For example, the GameController class in our design provides several methods like heroSelected and mapSelected for the GameCreatorView to interact with. Internally, these methods use the specific methods of the Game, Character, GameMap, Hero and Map classes to create entities and associate these with the main Game object. However, the GameCreatorView class does not need to know about these tedious details. The GameCreatorView class simply takes input from the user and interacts with the simple methods of the GameController class to handle this input. In a way, the GameController class serves as a Façade class in our design.

Below is a sample illustration for this pattern.

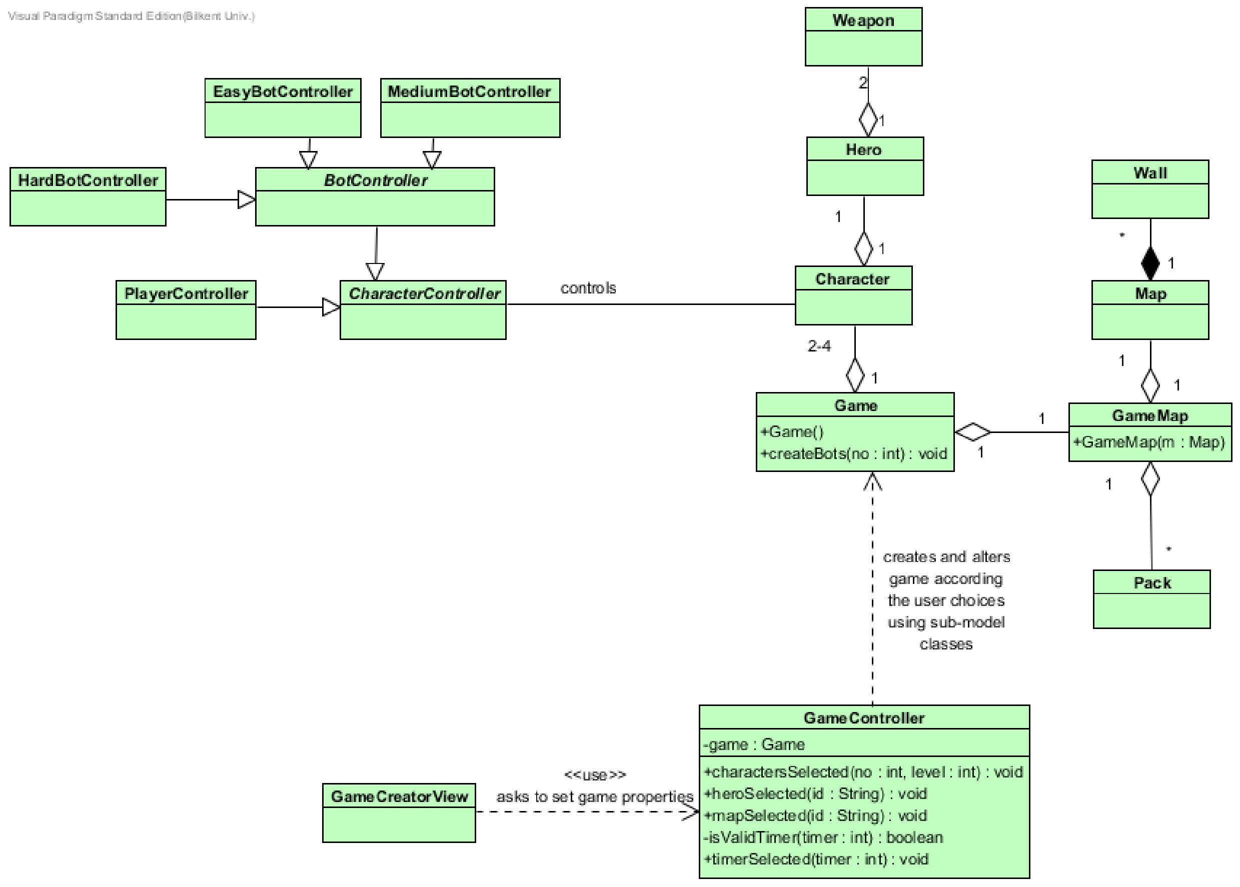

Visual Paradigm Standard Edition(Bilkent Univ.)
Weapon
EasyBotController
MediumBotController
HardBotController
BotController
PlayerController
CharacterController
controls
Hero
Character
Wall
Map
Game
+Game()
+createBots(no : int) : void
GameMap
+GameMap(m : Map)
Pack
creates and alters game according the user choices using sub-model classes
GameController
-game : Game
+charactersSelected(no : int, level : int) : void
+heroSelected(id : String) : void
+mapSelected(id : String) : void
-isValidTimer(timer : int) : boolean
+timerSelected(timer : int) : void
GameCreatorView
<<use>>
asks to set game properties
2
1
2-4
*

Figure 26 Illustration for the Facade Pattern

12.1.4 Observer Design Pattern

The Observer Design Pattern consists of an object, maintaining a list of its dependents (called observers), and notifying these objects at any time when change occurs by calling the observers related methods. It is a key part of the MVC architectural style.

In the current version of our project our subject classes mainly have only one observer, however, this might change in future developments.

For example, our Game model class has a reference to its GameView class. When any of the properties of the Game object is changed, the GameView is updated accordingly. Our design has a push notification system, instead of a push-update or pull system. That is, the view does not return information back about being changed, it is just informed that it needs to change.

Below is a very simple illustration for this design pattern.

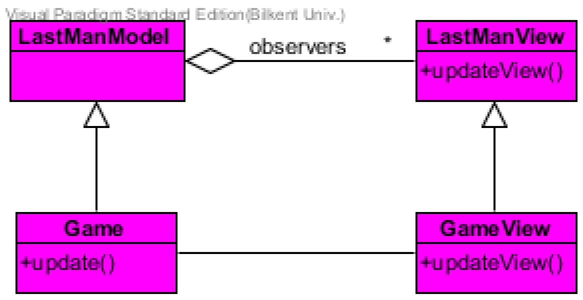


Figure 27 Illustration for the Observer Pattern

12.2 CLASS INTERFACES

12.2.1 Weapon Class

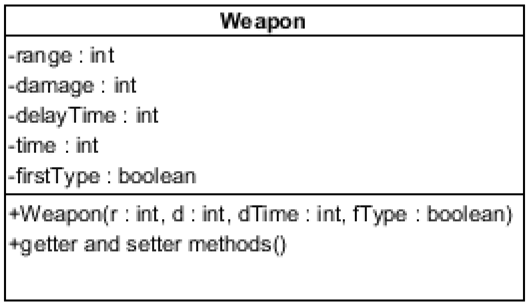

Weapon
-range : int
-damage : int
-delayTime : int
-time : int
-firstType : boolean
+Weapon(r : int, d : int, dTime : int, fType : boolean)
+getter and setter methods()

Figure 28 Weapon Class

Attributes:

- **private int range:** This represents the radius of the area the weapon will effect once it activates.
- **private int damage:** This represents the effectiveness of the weapon.
- **private int delayTime:** If firstType is true, this represents how many seconds needs to pass after the weapon is set before it explodes. If firstType is false, this represents how often the weapon can be used.
- **private int time:** This specifies the amount of time the weapon has spent in the game map.
- **private boolean firstType:** This specifies whether the weapon is type one or type two.

Constructors:

- **public Weapon(int r, int d, int dTime, boolean fType):** Initializes the weapon according to the values entered.

Methods:

- **getter and setter methods:** These provide accessibility to the private attributes of the class.

12.2.2 Hero Class

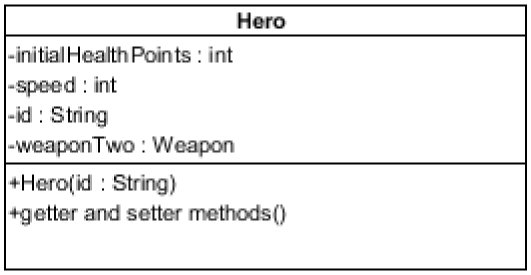


Figure 29 Hero Class

Attributes:

- **private int initialHealthPoints:** This specifies the default health point count of the hero.
- **private int speed:** This represents the speed value of the hero.

- **private int id:** This is the name of the hero.
- **private Weapon weaponTwo:** This specifies the second-type special weapon of the hero.

Constructors:

- **public Hero(String id):** Initializes the object according to the value entered. The other attributes of the hero are constructed using the id and a data file.

Methods:

- **getter and setter methods:** These provide accessibility to the private attributes of the class.

12.2.3 Character Class

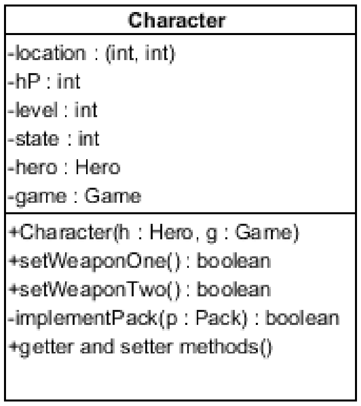

Character
-location : (int, int)
-hP : int
-level : int
-state : int
-hero : Hero
-game : Game
+Character(h : Hero, g : Game)
+setWeaponOne() : boolean
+setWeaponTwo() : boolean
-implementPack(p : Pack) : boolean
+getter and setter methods()

Figure 30 Character Class

Attributes:

- **private (int, int) location:** This specifies the current location of the character.
- **private int hP:** This represents the current health point count of the character.
- **private int level:** This specifies the pack usability of the character.
- **private int state:** This represents the current state of the character i.e. whether if it can use weapons or not.
- **private Hero hero:** The hero of the character.
- **private Game game:** This is a reference to the game object the character belongs to. It is required to access the game map.

Constructors:

- **public Character(Hero h, Game g):** Initializes the object according to the values entered.

Methods:

- **public boolean setWeaponOne():** This method checks if the character can set weapon one, by the character’s state and sets the weapon to the characters current location if true.
- **public boolean setWeaponTwo():** This method checks if the character can set weapon two, by the character’s state and sets the weapon to the characters current location if true.

- **private boolean implementPack(Pack p):** This method checks the pack's attributes and implements it accordingly.
- **getter and setter methods:** These provide accessibility to the private attributes of the class. Note: the setLocation method of this class checks for new packs in the new location and calls implementPack if any suck pack is found.

12.2.4 CharacterController Class

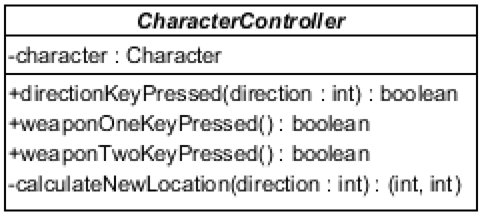

CharacterController
-character : Character
+directionKeyPressed(direction : int) : boolean
+weaponOneKeyPressed() : boolean
+weaponTwoKeyPressed() : boolean
-calculateNewLocation(direction : int) : (int, int)

Figure 31 CharacterController Class

Attributes:

- **private Character character:** This specifies the character to control.

Methods:

- **public boolean directionKeyPressed(int direction):** This method first calculates the character's new location using the direction value and the character's hero's speed and checks in the game map if the new location is valid. If the location is valid, it moves the character to the new location.
- **public boolean weaponOneKeyPressed():** This method activates the setWeaponOne method of the character class.
- **public boolean weaponTwoKeyPressed():** This method activates the setWeaponTwo method of the character class.
- **private (int,int) calculateNewLocation(int direction):** This is the private method used to calculate the character's new location.

12.2.5 PlayerController Class

PlayerController
+PlayerController(c : Character)

Figure 32 PlayerController Class

This class inherits the attributes and methods of the CharacterController class.

Constructors:

- **public PlayerController(Character c):** Initializes the object according to the value entered.

12.2.6 BotController Class

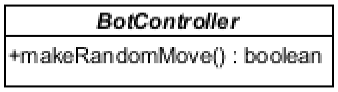


Figure 33 BotController Class

This class inherits the attributes and methods of the CharacterController class. This is an abstract class that has 3 subclasses for each difficulty.

Methods:

- **public boolean makeRandomMove():** This is an abstract method for this class. This will be implemented differently according to bot difficulty. It will mainly call one of the methods of the CharacterController class randomly, with more probability of calling the directionKeyPressed method.

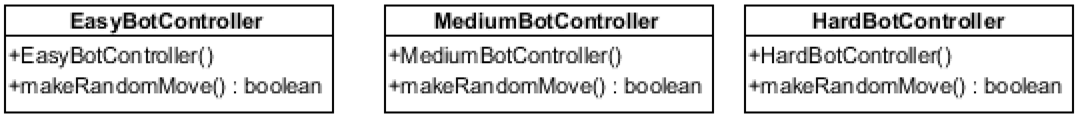


Figure 34 Subclasses of The BotContoller Class

12.2.7 Wall Class

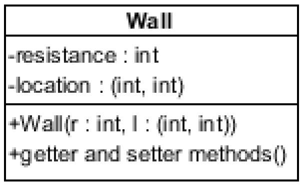


Figure 35 Wall Class

Attributes:

- **private (int, int) location:** This specifies the location of the wall.
- **private int resistance:** The resistance level of the wall is specified by this.

Constructors:

- **public Wall(int r, (int, int) l):** Initializes the object according to the values entered.

Methods:

- **getter and setter methods:** These provide accessibility to the private attributes of the class.

12.2.8 Map Class

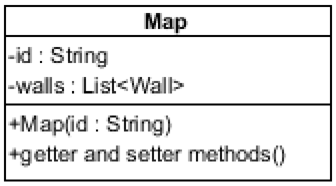


Figure 36 Map Class

Attributes:

- **private String id:** This specifies the id name of the map.
- **private List<Wall> walls:** The walls that make up the map are specified using this.

Constructors:

- **public Map(String id):** Initializes the object according to the values entered. The walls are constructed using the id and a data file.

Methods:

- **getter and setter methods:** These provide accessibility to the private attributes of the class.

12.2.9 Pack Class

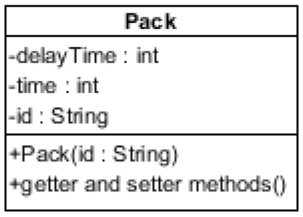


Figure 37 Pack Class

Attributes:

- **private int delayTime:** This represents the amount of time the pack is allowed to be on the map.
- **private int time:** This specifies the amount of time the pack has spent in the game map.
- **private String id:** This is the special id value of the pack.

Constructors:

- **public Pack(String id):** Initializes the pack according to the value entered. The delayTime value is initially default for all packs.

Methods:

- **getter and setter methods:** These provide accessibility to the private attributes of the class.

12.2.10 GameMap Class

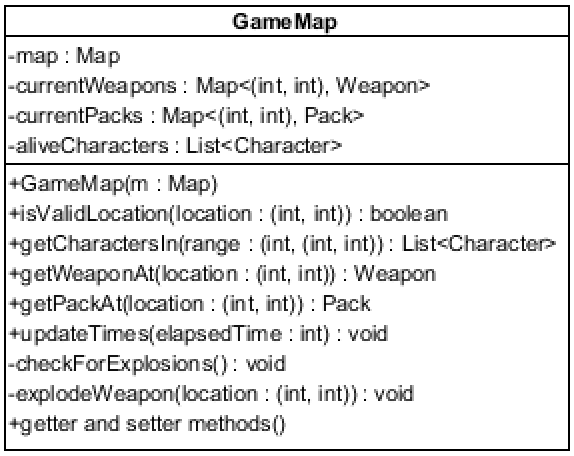

GameMap
-map : Map
-currentWeapons : Map<(int, int), Weapon>
-currentPacks : Map<(int, int), Pack>
-aliveCharacters : List<Character>
+GameMap(m : Map)
+isValidLocation(location : (int, int)) : boolean
+getCharactersIn(range : (int, (int, int)) : List<Character>
+getWeaponAt(location : (int, int)) : Weapon
+getPackAt(location : (int, int)) : Pack
+updateTimes(elapsedTime : int) : void
-checkForExplosions() : void
-explodeWeapon(location : (int, int)) : void
+getter and setter methods()

Figure 38 GameMap Class

Attributes:

- **private Map map:** This is the map object associated with the class.
- **private Map<(int,int), Weapon> currentWeapons:** This is a mapping from a specific location to a first-type weapon object. Notice only one weapon can be present at one location.
- **private Map<(int,int), Pack> currentPacks:** This is a mapping from a specific location to a pack object. Notice only one pack can be present at one location**.**
- **private List<Character> aliveCharacters:** This is a list of references to the characters that are alive in the game.

Constructors:

- **public GameMap(Map m):** Initializes the object according to the values entered.

Methods:

- **public boolean isValidLocation((int, int) location):** This checks if the location value is valid, i.e. it is in bounds and not on a wall.
- **public List<Character> getCharactersIn((int,(int,int)) range):** The range value denotes a circle with the given radius and location of center. This method returns the list of characters that are in the given range.
- **public Weapon getWeaponAt((int,int) location):** This method returns the weapon at the given location. It returns null, if there is none.
- **public Pack getPackAt((int,int) location):** This method returns the pack at the given location. It returns null, if there is none.
- **public void updateTime(int elapsedTime):** This updates the time values of the weapons and packs on the game map. It then calls the internal checkForExplosions method.
- **private void checkForExplosions():** This method first “explodes” the packs whose time value reached their delayTime value by removing them from the map. It then looks for any weapon whose time value reached its limit and calls the internal explodeWeapon method with the weapon’s location.

- **private void explodeWeapon((int,int) location):** This method explodes the weapon given in the location by getting the weapons range and damage attributes and affecting the characters and walls in the range accordingly.
- **getter and setter methods:** These provide accessibility to the private attributes of the class.

12.2.11 Game Class

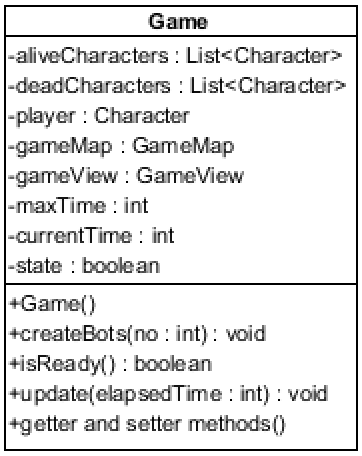

Game
-aliveCharacters : List<Character>
-deadCharacters : List<Character>
-player : Character
-gameMap : GameMap
-gameView : GameView
-maxTime : int
-currentTime : int
-state : boolean
+Game()
+createBots(no : int) : void
+isReady() : boolean
+update(elapsedTime : int) : void
+getter and setter methods()

Figure 39 Game Class

Attributes:

- **private List<Character> aliveCharacters:** This is a list of the characters that are alive in the game.
- **private List<Character> aliveCharacters:** This is a list of the characters that are dead in the game.
- **private Character player:** This is a specific reference to the user’s character.
- **private GameMap gameMap:** This is the GameMap instance associated with the game.
- **private GameView gameView:** This is the GameView instance associated with the game.
- **private int maxTime:** This is the maximum timer value set while creating the game.
- **private int currentTime:** This is the current time value of the game.
- **private boolean state:** This determines whether the game is active or not.

Constructors:

- **public Game():** This is the default constructor of the game class.

Methods:

- **public void createBots(int no):** This method creates bots and adds them to the list of characters according to the specified value.
- **public boolean isReady():** This method returns if the game is ready to start or not, i.e. all attributes are set to reasonable values.
- **public void update (int elapsedTime):** This method first updates the current time of the game. Then it calls the updateTime method of the GameMap class. When the activity of that method is done, this method updates the list of alive and dead characters accordingly.
- **getter and setter methods:** These provide accessibility to the private attributes of the class.

12.2.12 GameController Class

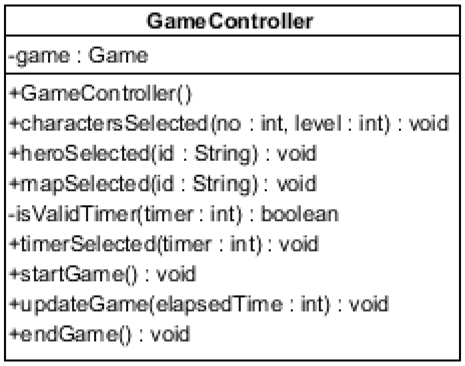

GameController
-game : Game
+GameController()
+charactersSelected(no : int, level : int) : void
+heroSelected(id : String) : void
+mapSelected(id : String) : void
-isValidTimer(timer : int) : boolean
+timerSelected(timer : int) : void
+startGame() : void
+updateGame(elapsedTime : int) : void
+endGame() : void

Figure 40 GameController Class

Attributes:

- **private Game game:** This is the game object to control.

Constructors:

- **public GameController():** This is the default constructor of the class. This also creates a default game object.

Methods:

- **public void charactersSelected (int no, int level):** This is activated after the number of bots and bot level is selected from the GameCreatorView. This method creates bot characters accordingly and adds them to the list of alive characters of the game.
- **public void heroSelected (String id):** This is activated after the hero is selected from the GameCreatorView. This method creates the player's character accordingly and adds it to the list of alive characters of the game.
- **public void mapSelected (String id):** This is activated after the map is selected from the GameCreatorView. This method creates Map and GameMap objects accordingly and associates them with the game object.
- **private boolean isValidTimer (int timer) :** This checks if the timer value specified in the GameCreatorView is in constraints.
- **public void timerSelected (int timer):** This is activated after the timer is successfuly selected from the GameCreatorView. This method sets the maxTime attribute of the game object.
- **public void startGame():** This is activated when the user chooses to start the game.
- **public void updateGame (int elapsedTime):** This method first checks if game is over by any game-over conditions (player is dead, all bots are dead, time is up) and if not calls the update method of the Game class. If game is over it calls the endGame method this class.
- **public void endGame (int elapsedTime):** This method ends the game and directs the user to the results screen.

12.2.13 GameView Class

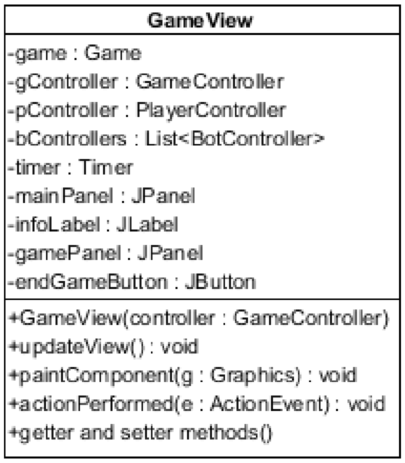

GameView
-game : Game
-gController : GameController
-pController : PlayerController
-bControllers : List<BotController>
-timer : Timer
-mainPanel : JPanel
-infoLabel : JLabel
-gamePanel : JPanel
-endGameButton : JButton
+GameView(controller : GameController)
+updateView() : void
+paintComponent(g : Graphics) : void
+actionPerformed(e : ActionEvent) : void
+getter and setter methods()

Figure 41 GameView Class

Attributes:

- **private Game game:** This is the game object associated.
- **private GameController gController:** This is the GameController object associated with the view. It is taken from the GameCreatorView class when that class is terminated.
- **private PlayerController pController:** This is the PlayerController object associated with the view.
- **private List<BotController> bControllers:** This is the BotController objects associated with the view.
- **private Timer timer:** This is the timer object associated.
- **private JPanel mainPanel:** This is the main panel of this view.
- **private JPanel gamePanel:** This is the game panel of this view i.e. the panel that shows the game map.
- **private JLabel infoLabel:** This label displays information about characters healh points and time left.
- **private JButton endGameButton:** This is the button provided to end the game manually.

Constructors:

- **public GameView(GameController controller):** This is the constructor of the class, it uses a GameController object that was created during GameCreatorView's activity. This class also creates related PlayerController and BotController objects.

Methods:

- **public void updateView ():** This updates the graphical components of the GameView.
- **public void paintComponent (Graphics g):** This is the required method to draw graphical components.
- **public void actionPerformed (ActionEvent e):** This specifies the actions to take when the user interacts with the view by clicking or keyboard action.
- **getter and setter methods:** These provide accessibility to the private attributes of the class.

12.2.14 GameCreatorView Class

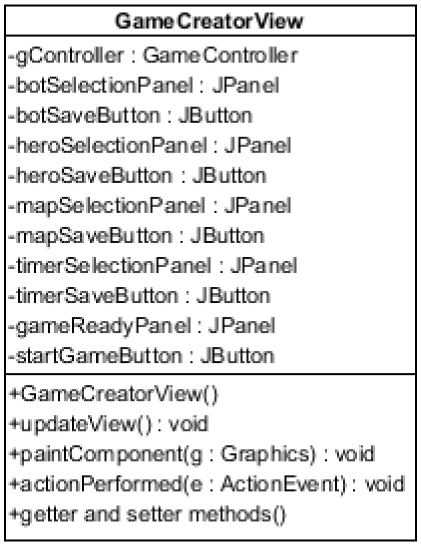

GameCreatorView
-gController : GameController
-botSelectionPanel : JPanel
-botSaveButton : JButton
-heroSelectionPanel : JPanel
-heroSaveButton : JButton
-mapSelectionPanel : JPanel
-mapSaveButton : JButton
-timerSelectionPanel : JPanel
-timerSaveButton : JButton
-gameReadyPanel : JPanel
-startGameButton : JButton
+GameCreatorView()
+updateView() : void
+paintComponent(g : Graphics) : void
+actionPerformed(e : ActionEvent) : void
+getter and setter methods()

Figure 42 GameCreatorView Class

Attributes:

- **private GameController gController:** This is the GameController object associated with the view.
- **private JPanel botSelectionPanel:** This is the panel responsible for selecting the number of bots and bot level.
- **private JButton botSaveButton:** This is button that needs to be pressed to switch from botSelectionPanel to heroSelectionPanel and inform the game controller to update the game object accordingly.
- **private JPanel heroSelectionPanel:** This is the panel responsible for selecting a hero.
- **private JButton heroSaveButton:** This is button that needs to be pressed to switch from heroSelectionPanel to mapSelectionPanel and inform the game controller to update the game object accordingly.
- **private JPanel mapSelectionPanel:** This is the panel responsible for selecting the game map.
- **private JButton mapSaveButton:** This is button that needs to be pressed to switch from mapSelectionPanel to timerSelectionPanel and inform the game controller to update the game object accordingly.
- **private JPanel timerSelectionPanel:** This is the panel responsible for selecting the maximum game time.
- **private JButton timerSaveButton:** This is button that needs to be pressed to switch from timerSelectionPanel to gameReadyPanel and inform the game controller to update the game object accordingly.
- **private JPanel gameReadyPanel:** This is the panel that appears after game option selecting is complete.
- **private JButton startGameButton:** This is button that needs to be pressed to start the game.

Constructors:

- **public GameCreatorView():** This is the default constructor for this class. It also creates the GameCreator object.

Methods:

- **public void updateView ():** This updates the graphical components of the GameCreatorView.
- **public void paintComponent (Graphics g):** This is the required method to draw graphical components.
- **public void actionPerformed (ActionEvent e):** This specifies the actions to take when the user interacts with the view by clicking or keyboard action.
- **getter and setter methods:** These provide accessibility to the private attributes of the class.

12.2.15 ResultsView Class

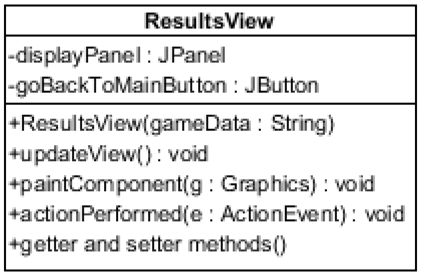

ResultsView
-displayPanel : JPanel
-goBackToMainButton : JButton
+ResultsView(gameData : String)
+updateView() : void
+paintComponent(g : Graphics) : void
+actionPerformed(e : ActionEvent) : void
+getter and setter methods()

Figure 43 ResultsView Class

Attributes:

- **private JPanel displayPanel:** This is the panel that displays information about the game, namely the rankings of the player and the bots.
- **private JButton goBackToMainButton:** This button is used to go back to the MainView.

Constructors:

- **public ResultsView(String gameData):** This is the default constructor for this class. It uses gameData provided by the GameController.

Methods:

- **public void updateView ():** This updates the graphical components of the class.
- **public void paintComponent (Graphics g):** This is the required method to draw graphical components.
- **public void actionPerformed (ActionEvent e):** This specifies the actions to take when the user interacts with the view by clicking or keyboard action.
- **getter and setter methods:** These provide accessibility to the private attributes of the class.

12.2.16 MainView Class

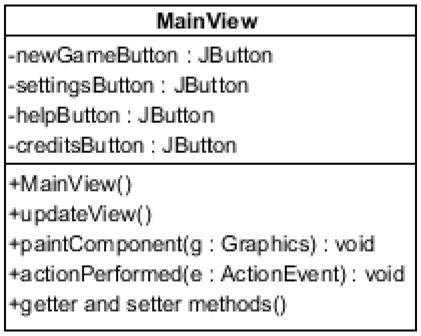

MainView
-newGameButton : JButton
-settingsButton : JButton
-helpButton : JButton
-creditsButton : JButton
+MainView()
+updateView()
+paintComponent(g : Graphics) : void
+actionPerformed(e : ActionEvent) : void
+getter and setter methods()

Figure 44 MainView Class

Attributes:

- **private JButton newGameButton:** This button is used to go to the GameCreationView.
- **private JButton settingsButton:** This button is used to go to the SettingsView.
- **private JButton helpButton:** This button is used to go to the HelpView.
- **private JButton creditsButton:** This button is used to go to the CreditsView.
-

Constructors:

- **public MainView():** This is the default constructor for this class.

Methods:

- **public void updateView ():** This updates the graphical components of the class.
- **public void paintComponent (Graphics g):** This is the required method to draw graphical components.
- **public void actionPerformed (ActionEvent e):** This specifies the actions to take when the user interacts with the view by clicking or keyboard action.
- **getter and setter methods:** These provide accessibility to the private attributes of the class.

12.2.17 HelpView Class

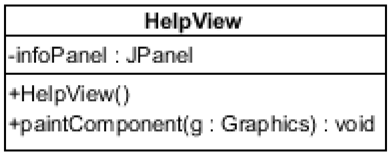


Figure 45 HelpView Class

Attributes:

- **private JPanel infoPanel :** This is the panel with the user documentation.

Constructors:

- **public HelpView():** This is the default constructor for this class.

Methods:

- **public void paintComponent (Graphics g):** This is the required method to draw graphical components.

12.2.18 CreditsView Class

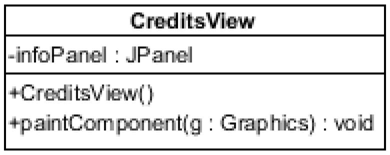


Figure 46 CreditsView Class

Attributes:

- **private JPanel infoPanel :** This is the panel with the program credits.

Constructors:

- **public CreditsView():** This is the default constructor for this class.

Methods:

- **public void paintComponent (Graphics g):** This is the required method to draw graphical components.

12.2.19 Settings Class

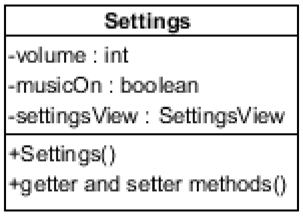


Figure 47 Settings Class

Attributes:

- **private int volume:** This specifies the volume level of the sound effects in the game.
- **private boolean musicOn:** This specifies whether music plays during game play or not.
- **private SettingsView settingsView:** The settings view associated with the class.

Constructors:

- **public Settings():** This initializes the attributes to default values.

Methods:

- **getter and setter methods:** These provide accessibility to the private attributes of the class.

12.2.20 SettingsController Class

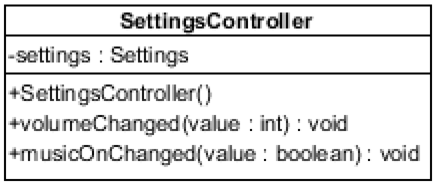

SettingsController
-settings : Settings
+SettingsController()
+volumeChanged(value : int) : void
+musicOnChanged(value : boolean) : void

Figure 48 SettingsController Class

Attributes:

- **private Settings settings:** The settings object that is being controlled by this class.

Constructors:

- **public SettingsController():** This is the default constructor for this class.

Methods:

- **public void volumeChanged (int value):** This is invoked when the user changes the volume from the SettingsView. It adjusts the settings object accordingly.
- **public void musicOnChanged (boolean value):** This is invoked when the user changes the musicOn state from the SettingsView. It adjusts the settings object accordingly.

12.2.21 SettingsView Class

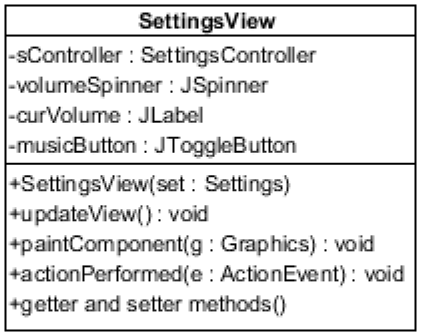

SettingsView
-sController : SettingsController
-volumeSpinner : JSpinner
-curVolume : JLabel
-musicButton : JToggleButton
+SettingsView(set : Settings)
+updateView() : void
+paintComponent(g : Graphics) : void
+actionPerformed(e : ActionEvent) : void
+getter and setter methods()

Figure 49 SettingsView Class

Attributes:

- **private SettingsController sController:** This is the SettingsController object that is created to alter the settings in this view.
- **private JSpinner volumeSpinner:** This is the tool used to alter the volume of settings.
- **private JLabel curVolume:** This displays the current volume set.
- **private JToggleButton musicButton:** This is the tool to edit the availability of the music during games.

Constructors:

- **public SettingsView(Settings set):** This is the constructor for this class. It creates the view using the settings object provided.

Methods:

- **public void updateView ():** This updates the graphical components of the class.
- **public void paintComponent (Graphics g):** This is the required method to draw graphical components.
- **public void actionPerformed (ActionEvent e):** This specifies the actions to take when the user interacts with the view by clicking or keyboard action.
- **getter and setter methods:** These provide accessibility to the private attributes of the class.

12.2.22 GameFrame Class

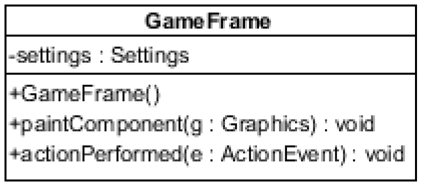

GameFrame
-settings : Settings
+GameFrame()
+paintComponent(g : Graphics) : void
+actionPerformed(e : ActionEvent) : void

Figure 50 GameFrame Class

This is the main GameFrame class of our project. It will be a JFrame that implements a card layout.

Attributes:

- **private Settings settings:** This represents the game settings for the project.

Constructors:

- **public GameFrame():** This is the default constructor of the class. This will also instantiate views like MainView, CreditsView etc.

Methods:

- **public void updateView ():** This updates the graphical components of the class.
- **public void paintComponent (Graphics g):** This is the required method to draw graphical components.
- **public void actionPerformed (ActionEvent e):** This specifies the actions to take when the user interacts with the view by clicking or keyboard action.

12.3 Specifying Contracts

We express our contracts in Object Constraint Language (OCL).

12.3.1 Character Class

OCL Contracts for setWeaponOne() in Character

1 - context Character::setWeaponOne() **pre**:

self.state == self.CAN_SET_BOMB

2 - context Character::setWeaponOne() **post**:

self.state = self.BOMB_SET

OCL Contracts for setWeaponTwo() in Character

3 - context Character::setWeaponTwo() **pre**:

self.state == self.CAN_USE_SPECIAL_WEAPON

4 - context Character::setWeaponOne() **post**:

self.state = self.CANT_USE_SPECIAL_WEAPON

OCL Contracts for implementPack(p) in Character

5 - context Character::implementPack(p) **pre**:

self.level == self.HARD

6 - context Character::setWeaponOne() **post**:

self.hP = self@pre.hP + self.PACK_BOOST

12.3.2 CharacterController Class

OCL Contracts for directionKeyPressed(d) in CharacterController

7 - context CharacterController::directionKeyPressed(d) **pre**:

    d == 1 | d == 2 | d == 3 | d == 4

8 - context CharacterController::directionKeyPressed(d) **pre**:

    self.character.getHP() > 0

9 - context CharacterController::directionKeyPressed(d) **pre**:

    self.character.getGame().getGameMap().isValidLocation(self.calculateNewLocation(d))

10 - context CharacterController::directionKeyPressed(d) **post**:

    self.character.getLocation() = self.calculateNewLocation(d)

The preconditions for this method include the direction being valid, the character being alive and the new location of the character being in game map bounds. The location is updated at the end of this method's activity.

12.3.3 GameMap Class

OCL Contracts for updateTimes(t) in GameMap

11 - context GameMap::updateTimes(t) **pre**:

    self.currentWeapons -> notEmpty | self.currentPacks -> notEmpty

12 - context GameMap::updateTimes(t) **post**:

    self.currentWeapons->at(i).time = self@pre.currentWeapons->at(i).time – t

13 - context GameMap::updateTimes(t) **post**:

    self.currentPacks->at(i).time = self@pre.currentPacks->at(i).time – t

The preconditions for this method include the list of weapons or packs on the game map not being empty. The timer values of these components are decreased as a result of this method.

OCL Contracts for explodeWeapon(l) in GameMap

14 - context GameMap::explodeWeapon(l) **pre**:

self.currentWeapons->at(l) != NULL

15 - context GameMap::explodeWeapon(l) **post**:

```
self.getCharactersIn(self.currentWeapons->at(l).range)->at(i).hP =
                    self@pre.getCharactersIn(self.currentWeapons->at(l).range)->at(i).hP -
                                                  self.currentWeapons->at(l).damage
```

The precondition for this method is that the given location has a weapon. As a result, all characters in the range of this weapon are damaged.

12.3.4 Game Class

OCL Contracts for update(t) in Game

16 - context Game::update (t) **pre**:

t > 0

17 - context Game::update (t) **post**:

self.currentTime = self@pre.currentTime – t

18 - context Game::update (t) **post**:

self.gameMap.updateTimes(t)

The precondition for this method is that the time value t is valid. As a result the game map is asked to update itself.

12.3.5 GameController Class

OCL Contracts for heroSelected(id) in GameController

19 - context GameController::heroSelected (id) **pre**:

id != NULL

20 - context GameController::heroSelected (id) **post**:

self.game.getPlayer() = new Character(new Hero(id), self.game)

21 - context GameController::heroSelected (id) **post**:

self.game.getAliveCharacters() = self@pre.game.getAliveCharacters() + self.game.getPlayer()

OCL Contracts for timerSelected(t) in GameController

22 - context GameController::timerSelected (t) **pre**:

self.isValidTimer(t)

23 - context GameController::timerSelected (t) **post**:

self.game.getMaxTime() = t

OCL Contracts for startGame() in GameController

24 - context GameController::startGame() **pre**:

self.game.isReady()

25 - context GameController::startGame() **post**:

self.game.getState() = !self@pre.getState()

OCL Contracts for updateGame(t) in GameController

26 - context GameController::updateGame(t) **pre**:

self.game.getCurrentTime() < self.game.getMaxTime()

27 - context GameController::updateGame(t) **pre**:

self.game.getAliveCharacters() != self.game.getPlayer()

28 - context GameController::updateGame(t) **pre**:

not self.game.getDeadCharacters() -> includes(self.game.getPlayer())

29 - context GameController::updateGame(t) **post**:

self.game.update(t)

The preconditions for this method include checking if any game-over condition holds. That is this method is only invoked if there is still time left and if there are still bots alive and if the player is still alive. As a result, the game object is asked to update itself.

12.3.6 SettingsController Class

OCL Contracts for volumeChanged(v) in SettingsController

30 - context SettingsController::volumeChanged(v) **pre**:

    0 <= v <= 10

31 - context SettingsController::volumeChanged(v) **post**:

    self.settings.getVolume() = v

OCL Contracts for musicOnChanged(v) in SettingsController

32 - context SettingsController::musicOnChanged(v) **pre**:

    v == true | v == false

33 - context SettingsController::musicOnChanged(v) **post**:

    self.settings.getMusicOn() = v

13 CONCLUSIONS AND LESSONS LEARNED

Last Man, is a strategy video game, and it is very fun and easy to play. Last Man offers the users a gameplay that is simple, easy to master and also entertaining. It also keeps an end-of-game score table to see the winner and different statistics of the game, which also makes it even more enjoyable. The Settings options make the user change the game environment as they wish.

As our game is an object-oriented programming project, the development parts were split up as analysis, design and object design in order to ease the project management.

The first stage was the analysis stage, which was about describing the general features of our game, such as functional requirements, non-functional requirements and constraints. In order to represent our project more easily, we also used different UML diagrams (Use Case, Class, Sequence, Activity). We tried to look from the user’s point of view and improved our project analysis accordingly. In our project, we decided to give the user the options to create a new game, select from different heroes, move their character in a specified map, use different weapons, or see their ranking. We decided to put some other game options such as volume and music control as well.

The second stage, which was more about the system decomposition and design goals, was the design stage. We first considered our non-functional requirements and turned them into design goals. After that, as we divided our system into subsystems, we thought that it would be better for our project to use the Model-View-Controller architecture. In this stage, we also described the project's hardware-software mapping and we addressed some key concerns about our project, such as data handling, access control, global software control and response time.

The final stage included the object design and we applied different design patterns to our project. These are mainly the composite design pattern, strategy design pattern, façade design pattern and observer design pattern. We believe that these really improved our project design. After describing our class interfaces and specifying our contracts, the only thing left to do was the implementation, and completing these three stages would help during the implementation tremendously. Last Man, turned out to be what we have imagined in the beginning: usable, reliable, and fun.

While doing this project we came across many challenging software engineering problems and we learned how to deal with them. As we understood how to use Java in a more efficient way and improved our skills in object oriented programming, we learned the details of the different stages of development of a software project. We learned the difference between functional and non-functional requirements, we learned how to draw different UML diagrams. While writing the design report, we understood that what we have done in the analysis part was actually really useful. We also learned how to use the MVC pattern in a more productive and effective way.

While writing our reports and coding, we used different software such as Visual Paradigm, GitHub, Google Drive and Eclipse. Especially GitHub and Google Drive helped us work as a team in a better way.

We believe that Last Man can be improved in different ways, and more can be added to its enjoyable content. One way of doing this could be implementing a multiplayer or online version of our game. So, our project still has room for more development.