Bilkent University

Department of Computer Engineering

CS319 Project / Group: 2B

Endless Dungeon: Progressive Dungeon Crawler RPG

Analysis Report

Can Özgürel, Alperen Kaya, Alemdar Salmoor, Batuhan Erarslan

Supervisor: Uğur Doğrusöz
TA: Hasan Balcı

Iteration 2 Analysis Report
April 3, 2018

This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfillment of the requirements of the Object-Oriented Software Engineering course CS319.

Contents

Analysis Report

Endless Dungeon: Progressive Dungeon Crawler RPG

1 Introduction

In Endless Dungeon, you will follow the path of an adventurer and see through his eyes as he decimates his foes. With different classes and several advantages and disadvantages that come with these classes, you will have many ways to (try to) survive the Endless Dungeon. You may find certain items to help you on your journey, but will they be enough to keep you alive? You may feel stronger with each enemy that you kill, but for how long will you be able to keep killing them?

The Endless Dungeon awaits, with endless opportunities as well as endless dangers. Are you up for the challenge?

2 Proposed System

2.1 Overview

- A retro FPS perspective will be used that shows the enemies as seen by the character face on
- Enemies will attack in waves and waves will consist of 3 enemies
- Every fifth wave will be a boss wave and the enemies on the two sides will be led by a boss enemy
- Maps will change as the player progresses further
- After beating every twentieth wave the player will transition to a new map with harder enemies
- Players will be able to improve their characters by slaying enemies and gaining experience
- Enemies will have a chance to drop beneficial items for the player to use, with bosses having a higher drop rate
- Endless Dungeon will consist of 3 classes (Warrior, Mage, Assassin)
- Players will choose a class at the start of the game
- Each class will have certain advantages and disadvantages
- Enemies will have different types that set them apart from one another
- Players will have to employ convenient tactics to thrive against each enemy type

2.2 Functional Requirements

2.2.1 Play Game

The player will be represented with a simple image or animation on the mid-bottom of the screen. Enemies will be presented on the middle of the screen on each level. The player will be able to choose from various attacks and items in order to attack these enemies. Once all the enemies on the screen are destroyed, the player will progress to the next level. With each level the enemies' health and strength will increase. The player will be able to collect items and improve their hero's skills and stats as they progress.

2.2.2 Pause

The player can pause the game to access an in-game menu. They can save the game, resume the game or quit from this menu.

2.2.3 Save

The player can save the game at any point. The system will record the level and hero progress.

2.2.4 Load

If the player has saved a previous game they haven't finished, they can continue playing that session by choosing this option. The system will load the game with the same level and hero progress as they have left.

2.2.5 Change Settings

The player will be able to adjust some aspects of the game. Currently there are two options:

- Change Difficulty: The player will be able to change the difficulty of the game by choosing one of the three options (Easy, Normal, Hard).
- Enable Sounds: The player will be able to enable or disable in-game sounds.

2.2.6 Help

Information about the game such as instructions and game lore will be shown.

2.2.7 Real-Time Progression

The actions of the player will determine whether they can progress through the game or not. The player will be able to gain experience and level up after defeating enemies, which will result in getting a stronger character with an improved stat distribution and stronger skills. The game will be running on real-time, meaning that the turn-based gameplay will not stop the activity of characters and the game animations will show the actions taken during the last turn.

2.3 Non-functional Requirements

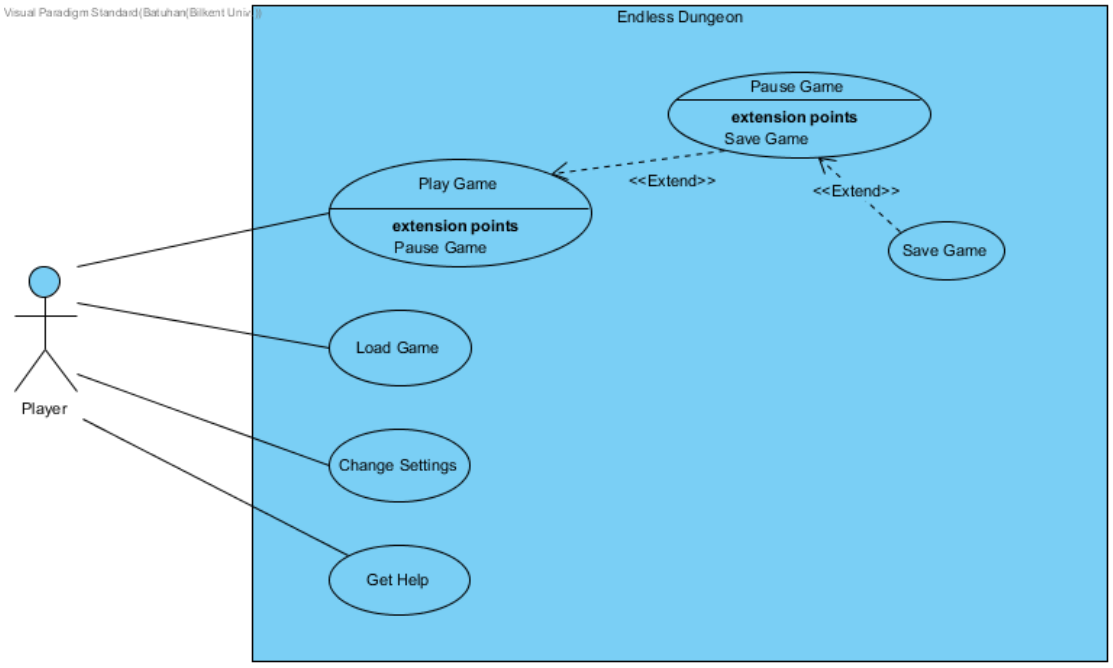
- The game will be in RPG format
- Players will be able to get stronger as they play
- Enemy styles will be in correspondence with the current map

2.4 Pseudo Requirements

- The game will be developed in Java
- The game will be a desktop application for PCs

2.5 System Models

2.5.1 Use-Case Model



Use Case #1

Use Case Name: Play Game

Participating Actors: Player

Entry Conditions:

Player has opened the game and chose the "New Game" or "Load Game" button.

Exit Conditions:

- Player has successfully completed the game.
- Player has died.
- Player has chosen to quit the game.

Main Flow of Events:

1. Player starts the game.
2. The system loads the appropriate level.
3. Player chooses an attack and attacks the enemies.
4. Player destroys all the enemies.
5. Player progresses to the next level.
6. Player encounters a boss.
7. Player destroys the boss by repeating step 3.
8. Player gains item(s) and experience.
9. Player improves their skills and stats.
10. Player completes the game by repeating steps 3-9.

Alternative Flow of Events:

1. Player loses all of their health and dies.
   a. Player chooses to continue from a saved game.
   b. Player chooses to start a new game.
2. Player chooses to quit the game. (return to main menu)

Use Case #2

Use Case Name: Pause Game

Participating Actors: Player

Entry Conditions:

- Player is in-game and chooses to pause the game.

Exit Conditions:

- Player quits the pause menu and returns to the game, OR
- Player quits the pause menu and returns to main menu,

Main Flow of Events:

1. Player chooses to pause the game.
2. Player resumes the game.

Alternative Flow of Events:

1. Player chooses to pause the game.
2. Player quits the game from the pause menu and returns to main menu.

Use Case #3

Use Case Name: Save Game

Participating Actors: Player

Entry Conditions:

- Player is in the Pause menu.

Exit Conditions:

- The game is successfully saved.

Main Flow of Events:

1. Player chooses to save the game in the Pause menu.
2. The system records the current progress.

Use Case #4

Use Case Name: Load Game

Participating Actors: Player

Entry Conditions:

- Player has opened the game and chose the "Load Game" button, AND
- Player has a saved game.

Exit Conditions:

- Player has successfully completed the game, (?) OR
- Player has died, (?) OR
- Player has chosen to quit the game.

Main Flow of Events:

1. Player starts the game on the saved level.

2. Player follows the steps 3-10 from the New Game case.

Alternative Flow of Events:

1. ???

Use Case #5

Use Case Name: Change Settings

Participating Actors: Player

Entry Conditions:

Player has opened the game and chose the "Settings" button.

Exit Conditions:

- Player quits the Settings menu and returns to main menu.

Main Flow of Events:

1. Player selects the "Settings" button.
2. Player adjusts the difficulty of the game.
3. Player goes back to main menu.

Alternative Flow of Events:

1. Player does not want to change any settings.
2. Player goes back to main menu.

Use Case #6

Use Case Name: Get Help

Participating Actors: Player

Entry Conditions:

Player has opened the game and chose the "How to Play" button.

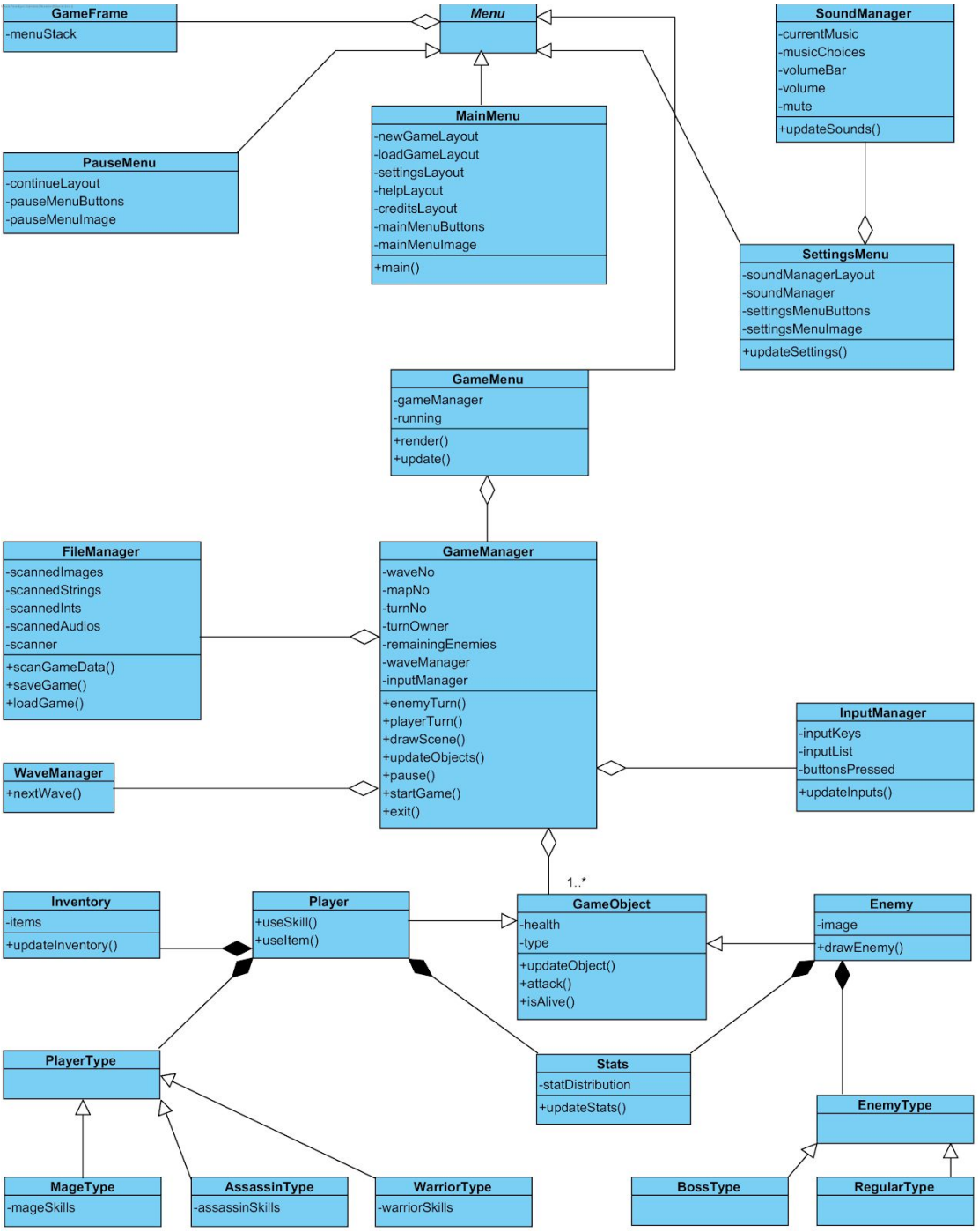
Exit Conditions:

- Player quits the help menu and returns to main menu.

Main Flow of Events:

1. Player chooses the How to Play button from the main menu.
2. Game instructions are displayed on the screen.

2.5.2 Object and Class Model



GameFrame: The main frame for the game panels to be loaded into. This class will work as the frame to show the game menus and the game itself.

- **menuStack:** The menus to be shown will be pushed to a stack when they need to be shown and then popped when the user closes them.

Menu: General Menu class to be inherited by all menu types.

SoundManager: Manages the music played during the game and possibly the sound effects in the game. Its attributes will contain:

- **currentMusic:** The music that is currently playing in the game.
- **musicChoices:** A list of choices for the music to be played during the game.
- **volumeBar:** A bar to modify the current volume level for the game sounds and music.
- **volume:** Current volume level.
- **mute:** Mute button to mute all of the game sounds and music.

and its methods will consist of:

- **updateSounds():** Update the changes made to the game sound and music options.

PauseMenu: This class will be the window that the user sees after pausing the game from the GameManager while playing the game. It will have options to see the help related to the game and to exit the game. Its attributes will contain:

- **continueLayout:** Layout for continuing the gameplay.
- **pauseMenuButtons:** Buttons on the pause menu.
- **pauseMenuImage:** Image to be displayed on the pause menu.

MainMenu: This class will be the first window that the user sees after opening the game up from their desktop. It will have options to start a new game, change settings, load a saved game or quit the game. Its attributes will contain:

- **newGameLayout:** Layout for starting a new game.
- **loadGameLayout:** Layout for loading a saved game.
- **settingsLayout:** Layout for going to the settings menu.
- **helpLayout:** Layout for going to the help screen.
- **creditsLayout:** Layout for going to the credits screen.
- **mainMenuButtons:** Buttons on the main menu.
- **mainMenuImage:** Image to be displayed on the main menu.

and its methods will consist of:

- **main():** The main method.

SettingsMenu: This class will control the settings of the game. It will allow users to personalize their game experience by changing certain aspects of the game to their liking like the music and the difficulty level. Its attributes will contain:

- **soundManagerLayout:** UI layout for the sound manager part of the settings menu.
- **soundManager:** Sound management class will be used inside the SettingsMenu.
- **settingsMenuButtons:** Buttons of the settings menu.
- **updateSettings():** Updates the settings of the game after the user is done changing them.

GameMenu: The menu to be used to run the gameplay on. Its attributes will contain:

- **gameManager:** Game management class will be used to update the states of the game objects according to the user's actions.
- **running:** Value to check whether the game is currently running or not.

and its methods will consist of:

- **render():** Renders the game objects and visual elements visible on the screen.
- **update():** Updates the states of game objects.

FileManager: Manages the file system of the game for the saving and loading operations. Its attributes will contain:

- **scannedImages:** Images that were scanned after the last save operation.
- **scannedStrings:** Strings that were scanned after the last save operation.
- **scannedInts:** Integers that were scanned after the last save operation.
- **scannedAudios:** Audios that were scanned after the last save operation.
- **scanner:** Scanner to be used for scanning the game elements in order to save them to a file.

and its methods will consist of:

- **scanGameData():** Scans the current data of the game.
- **saveGame():** Saves the game whose data was scanned the last.
- **loadGame():** Loads the game from a certain saved game file.

WaveManager: Manages the waves and thus, the progression system of the game. Its only method will be:

- **nextWave():** Advances the game to the next wave after the user completes the current wave.

GameManager: GameManager will be the main class of Endless Dungeon. It will essentially work as the game engine. All main functions of the game will be controlled from here. Its attributes will contain:

- **waveNo:** The current wave number.
- **mapNo:** The current map number.
- **turnNo:** The current turn number.
- **turnOwner:** The player or the enemy that is supposed to play during the current turn.
- **remainingEnemies:** Total number of enemies remaining to pass the current wave.
- **waveManager:** Wave management class controls the progression of the waves and will be used by the GameManager class.
- **inputManager:** Input management class controls the user inputs for the game and will be used by the GameManager class.

and its methods will consist of:

- **enemyTurn():** Makes the enemies play their turns.
- **playerTurn():** Makes the game engine ready to take player input for their next turn.
- **drawScene():** Draws the current scene elements like tabs and enemies.
- **updateObjects():** Updates the states of objects that are on the current scene like the stats of the player and the enemies.
- **pause():** Pauses the game and shows the pause menu.
- **startGame():** Initializes the game objects and starts the game.
- **exit():** Terminates the game objects and exits the game.

InputManager: InputManager will be the class that controls the user-defined shortcuts and input selections (choices to use the keyboard for shortcuts). Its attributes will contain:

- **inputKeys:** The key set that was selected by the user to be used.
- **inputList:** A list of key sets that the user can choose from in order to use while playing the game.
- **buttonsPressed:** Last set of buttons pressed by the user.

and its methods will consist of:

- **updateInputs():** Updates the inputs according to the user's choice.

GameObject: General class to represent the active game objects. Currently there are two, which are Player and Enemy. Its attributes will contain:

- **health:** The current health of the object.
- **type:** The type of the object.

and its methods will consist of:

- **updateObject():** Updates the specific attributes of the object that were changed after the last update.
- **attack():** Attacks another object of choice.
- **isAlive():** Checks if the object is still alive.

Stats: Special attributes used by the Player and Enemy classes to further modify the possible outcomes and play styles of the game. Its attributes will contain:

- **statDistribution:** Distribution of stats will determine the strengths and weaknesses of the player or a certain enemy.

and its methods will consist of:

- **updateStats():** Updates the stats of the object according to recent changes if any were made (level ups or special items may require it).

Player: This class will be the player that the user uses to play the game. The user will experience the world of Endless Dungeon as the player and every player's game experience will be different with the help of our progression system and different classes. Its methods will consist of:

- **useSkill():** Lets the player use the chosen skill.
- **useItem():** Lets the player use the chosen item.

Inventory: This class will symbolize the inventory of the player. Its attributes will contain:

- **items:** The items that the player managed to gather during their adventure.

and its only method will be:

- **updateInventory():** Updates the inventory by adding or removing certain items according to the user's actions.

PlayerType: This class will be the profession (called class in RPGs) of the player character. This choice will determine the most of the experience that the user gets from Endless Dungeon. Its subclasses will contain:

- **MageType**

- **AssassinType**
- **WarriorType**

and they will each determine the stats and the skills of the player to correspond to their specialties in their own respective fields (More Strength for Warriors, more Intelligence for Mages etc.)

Enemy: This class will be the enemies that the player has to fight against in Endless Dungeon. There will be 2 enemy types that determine the significance of that specific enemy in the game and they will contribute in different ways to the experience of the player. Its attributes will contain:

- **image:** Image to be used to visualize the enemy on the screen.

and its methods will consist of:

- **drawEnemy():** Draws the enemy on the screen.

EnemyType: This class will be the type of a specific enemy. It will determine how much hardship the enemy should prove to be against the player. Its subclasses will contain:

- **BossType**
- **RegularType**

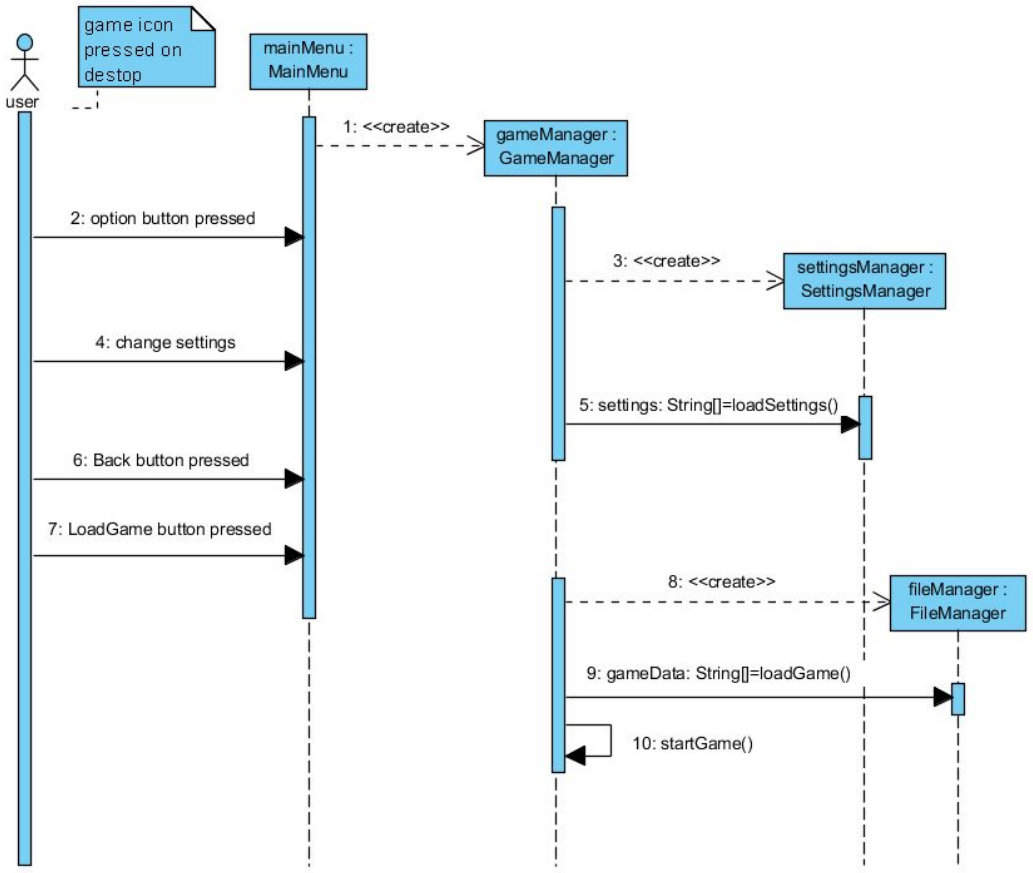
and they will each determine the stats of the enemy to make it harder to beat if it is a boss enemy or make it comparatively easier to beat if it is a regular enemy.

2.5.3 Dynamic Models

Sequence Diagrams

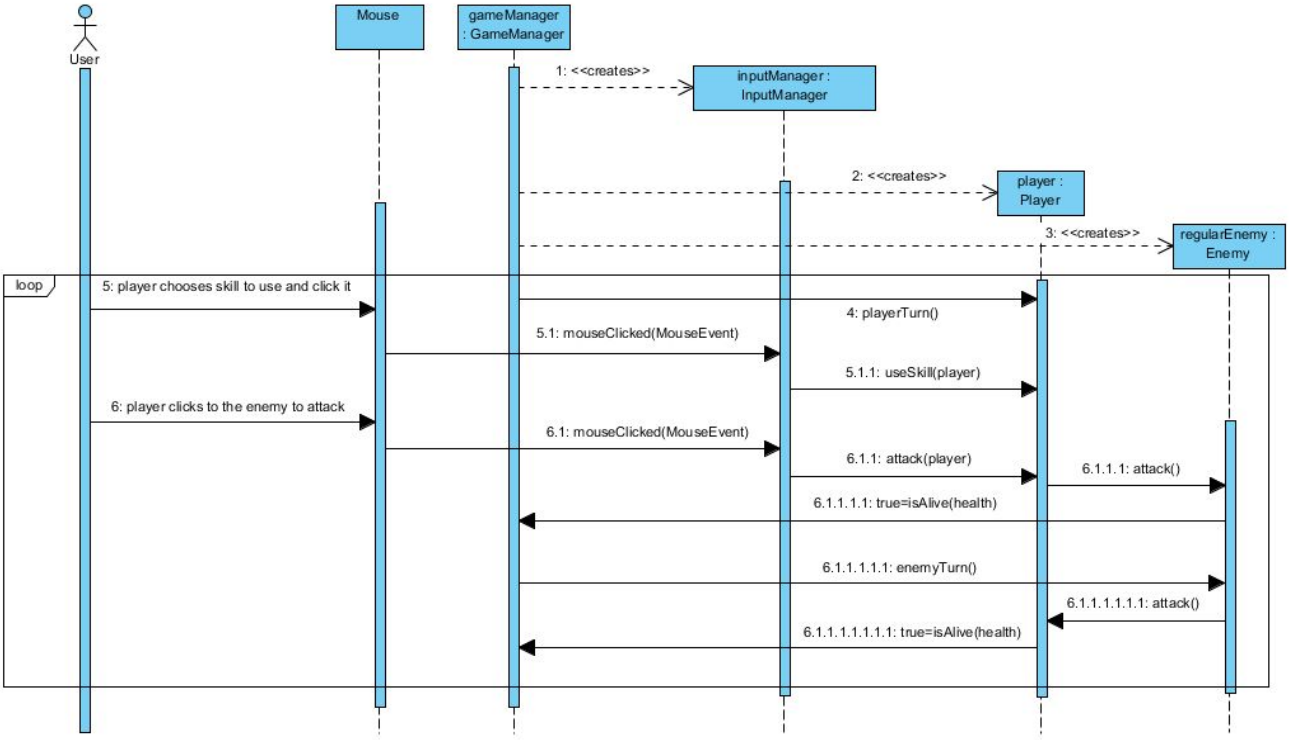
Scenario: Changing Settings and Loading Game

User wants to change the game settings and than wants to load the game that he played before. To do that, user opens the Main Menu screen. If user changes the settings, it is updated by Setting Manager by updateSettings() method. After updating settings according to user preference, user can go back to Main Menu by clicking back button.If user does not change the settings, back button will direct him/her to the Main Menu without updating the Setting Manager. Later user wants to load a game that he/she loaded before by clicking load game button on the Main Menu. By clicking that button Game Manager will load the game by loadGame() method by the help of File Manager and after loading complete Game Manager starts the game by startGame() method.

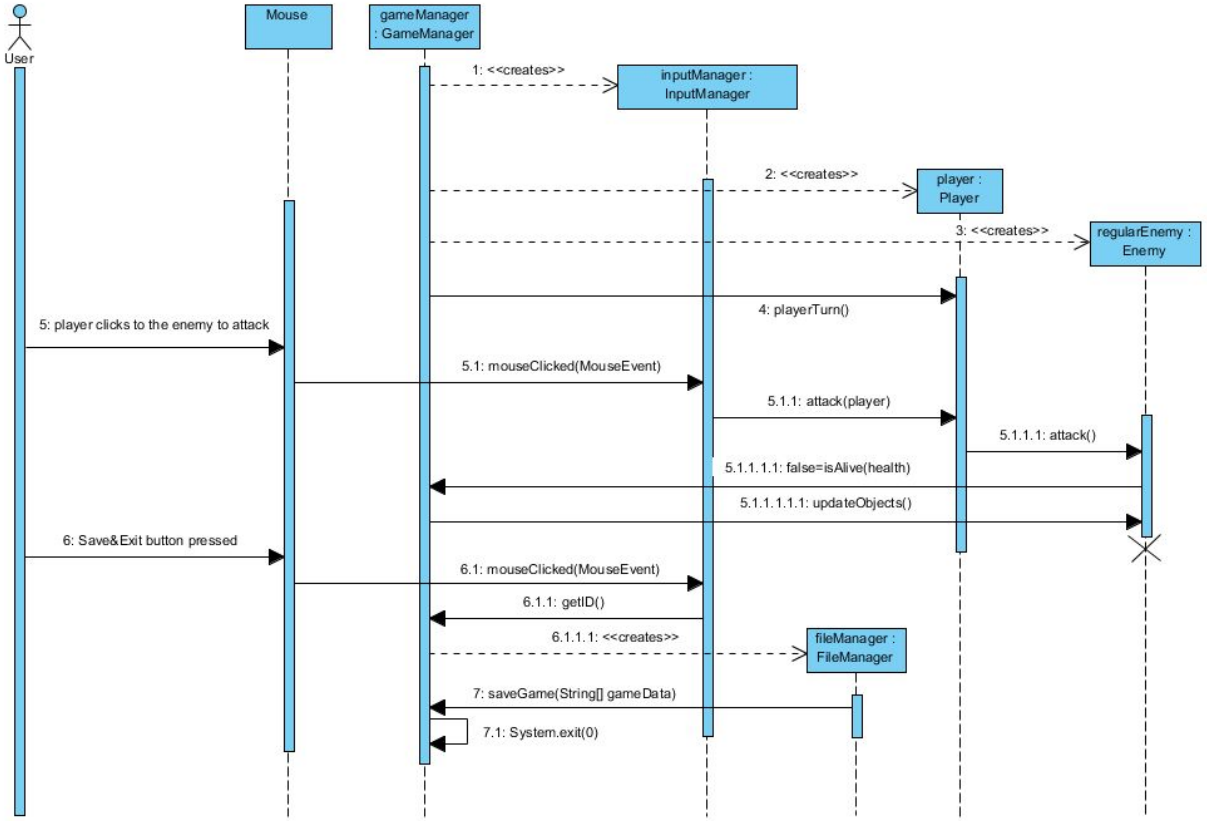


Scenario: Game Dynamic Part1

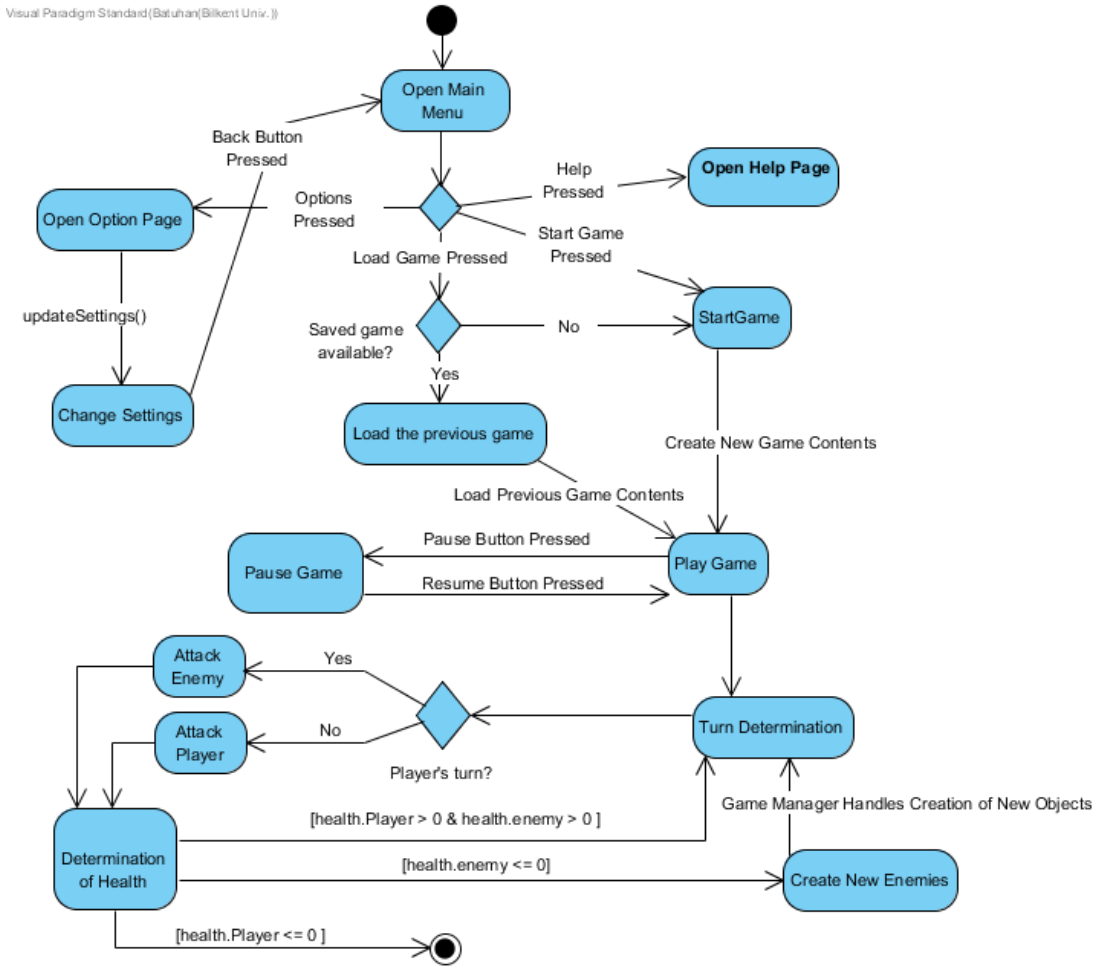
After loading the saved game by the help of the File Manager (provided by loadGame() method) or starting a new game, Game Manager will start the game and Player and Enemy will be created by Game Manager. Game Manager will decide whether is this player's turn or enemies' turn through the playerTurn() and enemyTurn() methods. If it is player's turn user will decide which item to use by useSkill() method by clicking the skill with Mouse and then it will attack the enemy by attack() method. After attacking enemy, Enemy object will send a information to the Game Manager about its health instance.When it is Enemies' turn to attack, enemy will decrease the health of player by attack() method and this time player object will send the necessary information about its health to the Game Manager. Interaction between Player and Enemy will repeat in a loop until the heath of the one of these is equals zero.



Scenario: Game Dynamic Part2
When game loop continues, player attacks to the enemy by attack(). If enemy's health equals to 0, isAlive() function of the corresponding enemy will return false. If this is the case, updateObjects() method of GameManager will destroy the enemy. If the user wants to quit the game, he can click to the Save & Quit button then this leads File Manager to save the current progress by saveGame() method. Then Game Manager will stop the program by exit() method.


User
Mouse
gameManager : GameManager
1: <<creates>>
inputManager : InputManager
2: <<creates>>
player : Player
3: <<creates>>
regularEnemy : Enemy
4: playerTurn()
5: player clicks to the enemy to attack
5.1: mouseClicked(MouseEvent)
5.1.1: attack(player)
5.1.1.1: attack()
5.1.1.1.1: false=isAlive(health)
5.1.1.1.1.1: updateObjects()
6: Save&Exit button pressed
6.1: mouseClicked(MouseEvent)
6.1.1: getID()
6.1.1.1: <<creates>>
fileManager : FileManager
7: saveGame(String[] gameData)
7.1: System.exit(0)

Activity Diagram



When the program starts, the system shows the main menu and waits for user input. The player can choose Start Game, Load Game, Options or Help.

If Start Game is chosen the system creates new game contents and the game starts. After the game starts, the user can keep playing or pause the game.

If the game is paused, it waits until the user resumes the game.

While the user keeps playing, in each turn the system determines whether it is the player's turn or the enemy turn. Depending on whose turn it is, one party attacks the other and their healths are updated accordingly.

If both the player's and the enemies' healths are above zero, the process repeats until the health of one party reaches zero.

If the enemy health equals to zero, new enemies are created and the above process repeats.

If the player's health equals to zero, the player dies and the system goes to final node.

If Load Game is chosen, the system checks whether there is a previous game available to load.

If there is, the system loads previous game contents and the game starts. After that the system works the same as above.

If there is not, the system creates a new game.

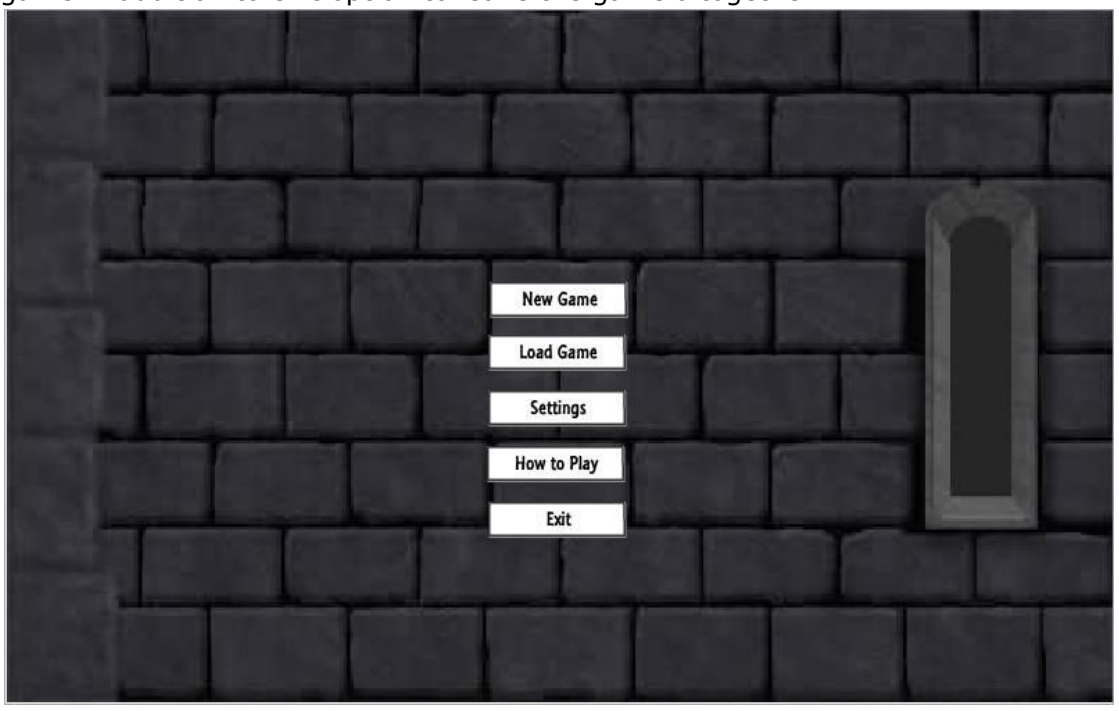
If Options is chosen, the Options page is opened. Then the user can change some game settings and settings are updated accordingly. If Back button is pressed, the system loads the main menu.

If Help is chosen, the system loads the Help page which contains information on how to play the game.

2.5.4 User Interface:

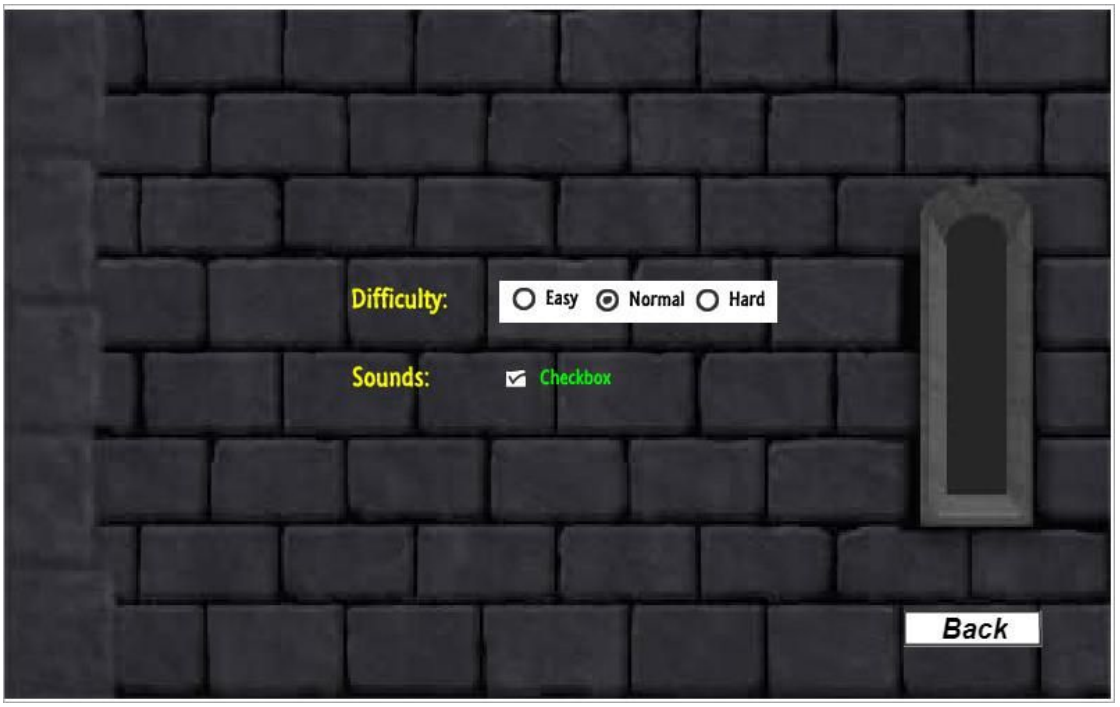
Main Menu:

Main Menu is the first what the user comes across when launching the Endless Dungeon application. From here he/she has multiple options to proceed with the game in addition to exit option to leave the game altogether.



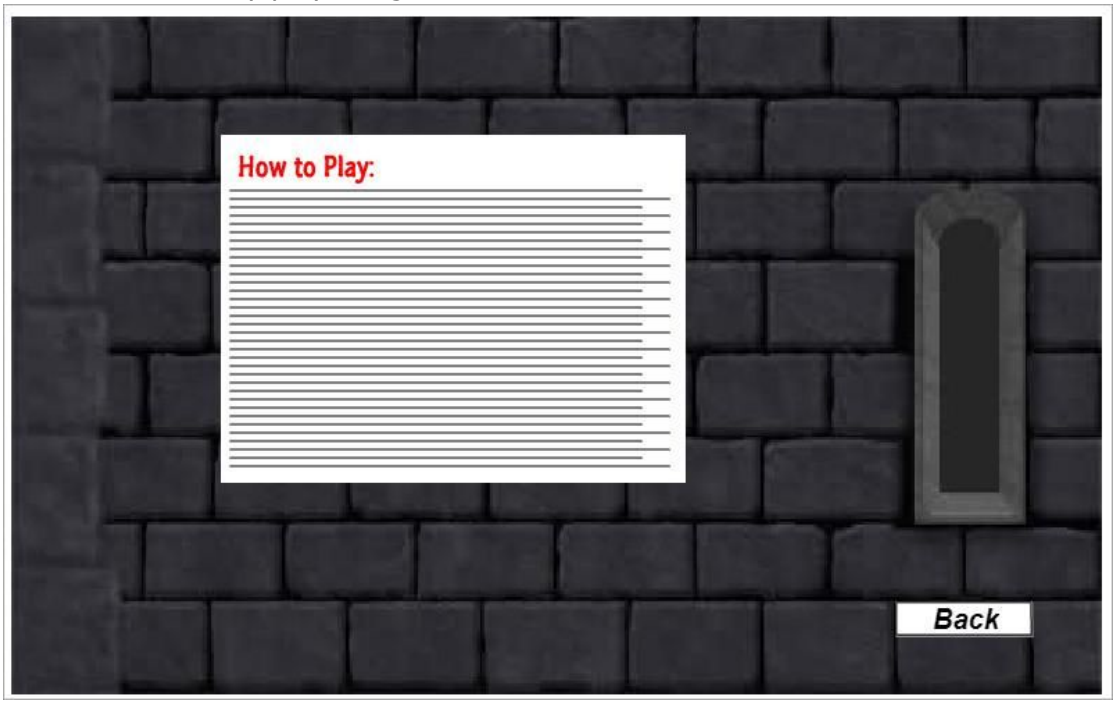
Settings:

"Settings" provides options to adjust game difficulty as well as enabling/disabling in game sounds. Lower right "Back" button brings the user back to the Main Menu



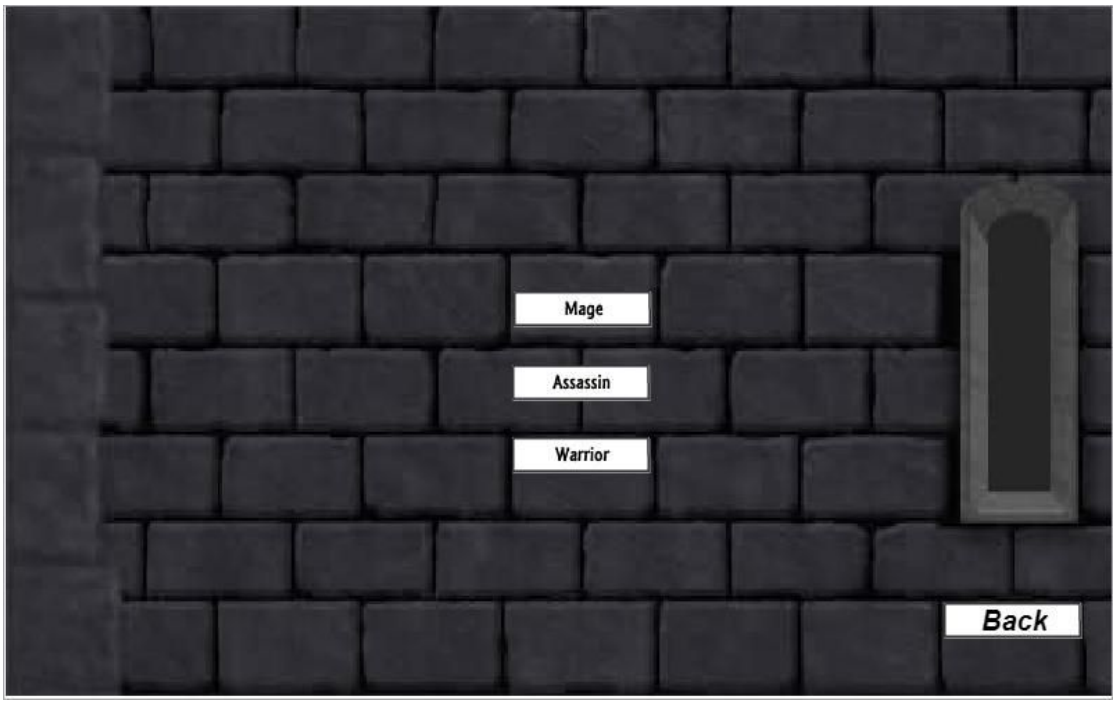
How to Play:

How to Play menu encompasses all the information that would be helpful for the user to successfully play the game



Choose Character:

Choose Character option is displayed when the user chooses "New Game" from the "Main Menu". The user can choose from three types of Characters that will directly affect the gameplay.



Load Game:
If the user wishes to proceed with previously saved game he/she is redirected to the "Load Game" menu, where he/she can choose from multiple(if available) previously saved game progresses.

Play Game:
Play Game is where the player is taken if he/she chose to proceed with the New Game or Load Game options. In the first options the game is started from the initial setup, in the latter one user proceeds with the loaded game progress.

3 References

[1] Object-Oriented Software Engineering, Using UML, Patterns, and Java, 2nd Edition, by Bernd Bruegge and Allen H. Dutoit, Prentice-Hall, 2004, ISBN: 0-13-047110-0.

[2] https://dota2.gamepedia.com/Creeps

[3] http://www.lumzy.com/