Bilkent University

Department of Computer Engineering

CS 319 Term Project

Group 1H

War of Domination

Requirement Analysis Report

Project Group Members:

Akant Atılgan

Unas Sikandar Butt

Kazım Ayberk Tecimer

Supervisor: Eray Tüzün

Contents

1 Introduction

This report will document the revised analysis proceedings of the War of Domination game project. This project is being implemented as a part of the Object Oriented Programming course.

This analysis report will contain the overview of the game which will highlight the main aspects of the game. Afterwards the functional and non-functional requirements will be stated and explained. These requirements will try to summarize the main functionalities of the project and some extra functionality that may or may not be implemented. The functional requirements will also be visualized in the form of the functional model of the system; this model will contain the Use-case diagram.

Other dynamic models will also be presented which will include the sequence diagrams and activity diagram. These diagrams will be used to illustrate the behavior of the system when dealing with some key scenarios.

After the dynamic models, the object model will be presented which will describe different classes of the system and their associations. This model will illustrate all the key classes of the system in detail.

Finally some user interface mockups will be presented which will serve to present a much clearer picture of the game.

2 Overview

This section will summarize the main aspects of the game. Here we explain the Gameplay, as any game will be mainly judged on its gameplay. We also explain the Map of the game because this map will be the main screen which will be displayed when the game is being played. Weapons will also be a huge part of the game as they will provide the player with the capability of defeating his opponent. Menus are also explained, as they are used by the user to navigate through the game and get the desired functionality. The settings menu is explained additionally to explain the game settings that the user can change.

2.1 - Gameplay

War of Domination will be a both single and multiplayer 2D game. A player of this game is going to be represented by a character. Each player will be allowed to control his/her character. The character can attack enemies by either shooting or throwing knives. The aim of the player is to survive by killing the other player, or killing all the enemies. The game will be played with a fixed camera and an overhead view, so players can keep track of everything at all times. The game will have different maps consisting of various elements.

2.2 - Map

The map of this game will be consisting of tiles, bricks, and obstacles. The most basic component of the map will be the tiles. The map will then be further built on those tiles. For different game modes, there will be different types of maps consisting of different kind of tiles. Each turn the player can choose to move a character across the tiles. The map will consist of obstacles which will restrict movement of characters across them.

2.3 - Weapons

All characters will be equipped with a standard gun and throwing knives. These weapons will be used to defeat the opposing team. These weapons will have limited ammo and they will have varying damage.

2.4 - Menu

The Main-Menu in the game will be a button based title screen. This screen will be displayed when the user will first start the game. The options on this screen will include, Play, Settings, How to Play, Credits and Exit.

2.5 Settings

Player will be able to change the settings of the game. He/she will be able to adjust the volume, brightness and the fullscreen status of the game.

3. Functional Requirements

3.1 Main Menu

This screen will let the user to get to other mandatory functions like play game or optional functions like "how to play" "credits" etc. It is the basic screen user will encounter when they first launch the game. Users must be able to interact with a main menu when the game starts where there will be sections.

3.2 Start Game

When players press the start button by default player 1 will get first turn. There will be an indicator on a player's characters to let the user(s) know whose turn is being played at that moment. Players will be able to pick one character from their team and play it; until there are no-longer idle characters in their disposal. When a player has no available characters to play turn will pass to opponent. The player with no-characters left alive loses the game resulting in opponent getting victorious. Players will have a limited space of movement each turn, when they finish the movement phase they will have variety of attacking options listed in hotkeys from 1 to 9. The map itself will contain both obstacles and passable objects as well as pits/water holes (Which would result in the death of the player.) etc. There will be 3 basic weapon types in players' disposal; direct line of sight (los) weapons which results in instant damage and a possibility to headshot for a multiplier to their damage. Users should be able to play game and should be able to use different weapons. Explosive weaponry will create a knockback effect which will cause any characters in the field of effect to be washed off slightly. Third weapon type will be melee weapons which will also have a knockback effect. Users should be able to see the remaining bullets and character's health in the game. The game can be played with mouse only but it will also have keyboard support for hotkeys if players want to

play faster. Mainly the objective of the game is to eliminate other player's all characters by any means.

3.2.1 Local Co-Op Multiplayer Mode:

This mode will be played by default when user selects "Start Game". This mode will enable user to pick a map from the list of available maps; it will also let the user to select the amount of characters will be in the game. This mode will put teams to separate corners or pre-defined spawns in the selected map.

3.3 Save/Load Options

Users should be able to save their game at any time to be able to play them later on.

3.3.1 Load Game

This option will be only available during main-menu. It will load a previously stored game from its latest state and player turn. Load game might not work properly for different versions of the game.

3.3.2 Save Game

This option will only be available if there is a game going on at that moment. Players will be able to save their games to any directory they pick to load them up later.

3.4 How To Play

Users must be able to learn game mechanics via how to play screen.

User will be able to access this screen from the main menu or while playing the game by pressing "esc" to open a pop-up menu window.

This screen will contain:

-Game Mechanics (Knockback, Headshot, Melee, weapon types etc.)

-Game Objective(s)

-Turn management (Movement limits/Shooting/Available options (guard/stand still/aim etc.))

-Map Mechanics (Obstacles, holes, passable objects).

This screen will let users to start playing right away and since they can access this information even while playing they will have a most basic gamepedia in their disposal. This will let players to learn the game much faster.

3.5 Options

Users are able to change the sound, resolution and the brightness of the system.

This screen will contain following options:

-Volume (SFX/Music)

-Resolution (Will re-open the game frame when exiting options)

-Aim Assist (Will enable/disable a line of your aim while using Line Of Sight weaponry.)

-Gamma (Adjust Brightness)

-Change hotkeys for item access, character actions.(Note: These changes will be global and affect all players in the game.)

3.6 Credits

This screen will let players know better about the developers behind the game; and contact them if they want to by giving their e-mail addresses. It will also contain a sliding text which will list the individual parts done to form the game by the developers.

3.7 Additional Requirements

3.7.1 Power-Ups

Player should be able to pick-up power ups from map by walking over them and use them any time they want. These power-ups will re-appear on

the same spot each 30 seconds.Power-ups disappear from map when taken by a character.These power-ups will be

-Piercing Rounds(Allows ammunition to go inside obstacles)
-Extra Damage(Increases base damage of the weapon)
-Health Regen(Character regenerates life over time)
-Infinite Ammo(Weapon requires no ammunition)

3.7.2 Pick-up Items

Player should be able to pick-up ammo ,life or armor from map when their character walks over them. Characters cannot carry more ammo than their inventory allows. These items will disappear when taken and re-appear somewhere else on the map.

3.7.3 Additional Maps

User(s) should be able to pick map from the given list. There will be at least 2 different maps to choose from with varying sizes.

3.7.4 Unique Characters and Character Perks

User(s) characters will have attributes such as:

-More damage on low life.
-Slower but more life.
-Fast but fragile.
-Can’t use rifles but faster throwing speed.
Etc.

These perks will be different for each character; that users can pick from a list of available characters.

4 Non-Functional Requirements:

4.1 Game Performance

Since we are planning to add animations to every action in the game to some extend; we are going to make sure these animations won't cause any performance drops in the gameplay. The gameplay should be always smooth according to user's eye's which means it should have a higher FPS rate than the monitor refresh rate (Above 60 FPS and less than %50 CPU usage). The game also shouldn't strain the computer too much since it should be compatible with all systems available in the market that can run java applications.

4.2 Game Efficiency

The extent to which the software system handles response time (e.g. Users should be able to interact with the game less than 0.2 seconds.)

4.3 Game Reusability

All system that runs on a client device shall be written in a prevalent programming language (e.g. Java) such that the software can be run on a personal computer without having to download a supporting environment.

4.4 Scalability

System should be able to expand its processing capabilities upward (e.g. Game should support the 2 players to play the game simultaneously)**.**

4.5 Additional Non-Functional Requirements

4.5.1 Maintainability

The system should not be shut down for maintenance more than once in a 1-hour working period.

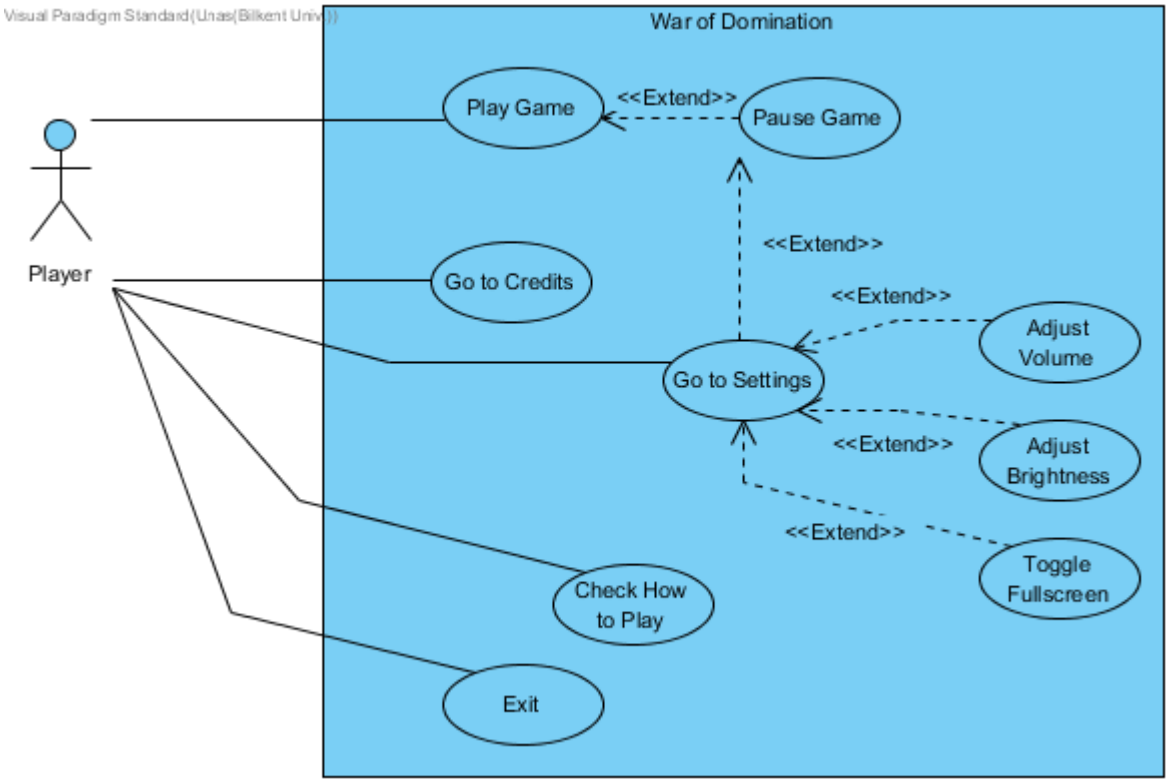
4.5.2 Availability

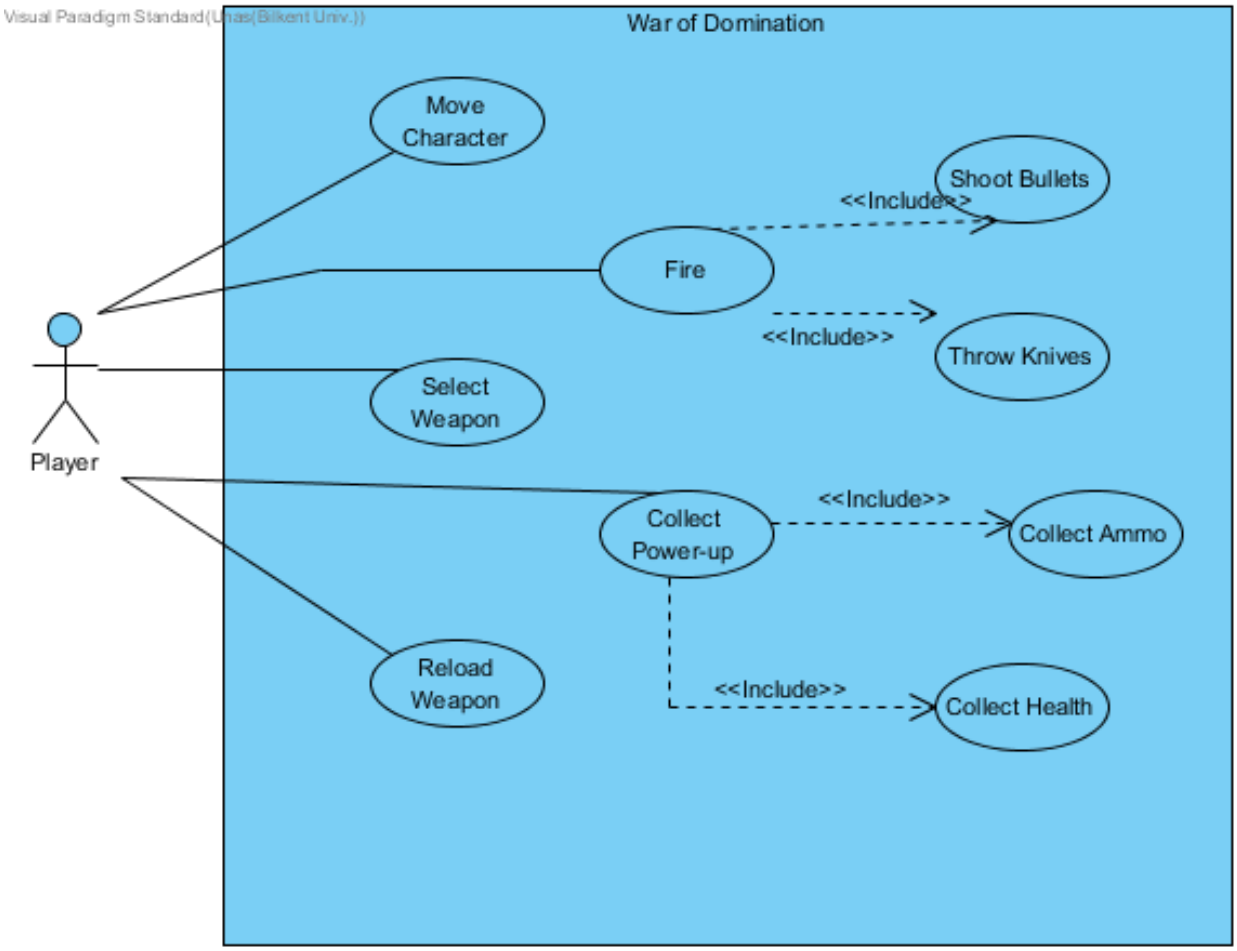
A new installation of the system shall be available for first-time use within 10 minutes of the start of the install.

5 - System Models

5.1 - Use Case Diagram

The use case diagram has been divided into two to ease readability. The first one represents the general use cases of the game and the second one will give further detail to the play game use case.





Use Case: Fire

Primary actor: Player

Interests: 1.Player wants to shoot.

Pre-Conditions:
1. Player must be playing the game.
2. Player's HP must be above 0.
3. Player must have some ammo.
4. Player must have selected a weapon.

Post-Conditions: NULL

Entry conditions: 1. Player controlling character presses "shoot" key.

Exit conditions: 1. Game creates the projectile.

Success scenario event flow:

1. Player chooses rifle.
2. Player presses the shoot key to shoot bullet.
3. Bullet projectile is created.

Alternative scenario event flow:

1. Player chooses rifle.
2. Player presses the shoot key to shoot bullet.
3. The game gives the no-ammo notification via sound.

Use Case: Move Character

Primary Actor: Player

Interests: 1.Player wants to move a character.

Pre-Conditions: 1. Player must be in play game screen.
2. Player's character has to be alive.

Post-Conditions NULL

Entry Conditions: 1.Player presses one of the movement keys of character.

Exit Conditions: 1. Character reaches an obstacle.
2. Character moves successfully.
3. Character dies.

Success scenario event flow:

1. Player presses one of the movement keys of character.
2. Character is moved towards that direction.
3. Game draws the updated view of characters position.

Alternative scenario event flow:

1. Player presses one of the movement keys of character.
2. The direction specified is occupied with an obstacle.
3. Character stays still.

Use Case: Reload Weapon

Primary actor: Player

Interests: 1. Player wants to reload his weapon.

Pre-Conditions: 1. Player must be in play game screen.
2. Player's character must be alive.
3. Player's weapon must not have a full magazine of ammo.

Post-Conditions: NULL

Entry Conditions: 1. Player presses the reload weapon key.

Exit Conditions: 1. Player's weapon is successfully reloaded.

Success scenario event flow:

1. Player chooses a weapon.
2. Player presses reload key.
3. Game replaces updates the magazine ammo weapon to full.

Use Case: Select Weapon

Primary Actors: Player

Interests: 1. Player wants to change his/her character's weapon.

Pre-Conditions: 1. Player must be in play game screen.
2. Player's character must be alive.

Post-Conditions: NULL

Entry Conditions: 1. Player presses one of the weapon selection keys.

Exit Scenarios: 1. Character's weapon is successfully changed.

Success scenario event flow:

1. Player presses one of the weapon selection keys for character.
2. Game updates the selected weapon of character.
3. Game updates the view of weapon selection of character.

Use Case: Play Game

Primary actor: Player

Interests: 1. Player/s want to play the game.
2. System initializes the game.

Pre-conditions: 1. Player must start the game.
2. Player has to be in the main menu.

Post-conditions: NULL

Entry conditions: 1. Player selects “Play” button in the main menu.

Exit conditions: 1. Player selects quit game button from the pause menu.
2. Player finishes the round by either winning or losing.

Success scenario event flow:

1. Player selects “Play” in the main menu.
2. Player moves the character using the direction keys on the keyboard.
3. Player uses the fire key to shoot bullets at enemy.
4. Enemy’s HP decreases
5. Player uses the reload key to reload his weapon.
6. Enemy shoots at the Player’s character.
7. Player moves and dodges the bullets
8. Repeat 3-7
9. Enemy dies.
10. Player finishes the round.

Alternative Event flow:

a. Player presses pause key while in game.
b. Player selects quit game option.
c. System closes the game.

Use Case: Go to Settings

Primary Actor: Player

Interests: 1. Player wishes to view settings.
2. Player wants to change volume level.
3. Player wants to change screen brightness.

Pre-conditions: 1. Player must be in the main menu.
OR
1. Player must be in the pause menu.

Post-conditions: 1. Settings are updated.

Entry conditions: 1. Player selects settings button in the main menu.
2. Player selects settings button from the pause menu.

Exit conditions: 1. Player presses the “esc” key from the keyboard.

Success scenario event flow:

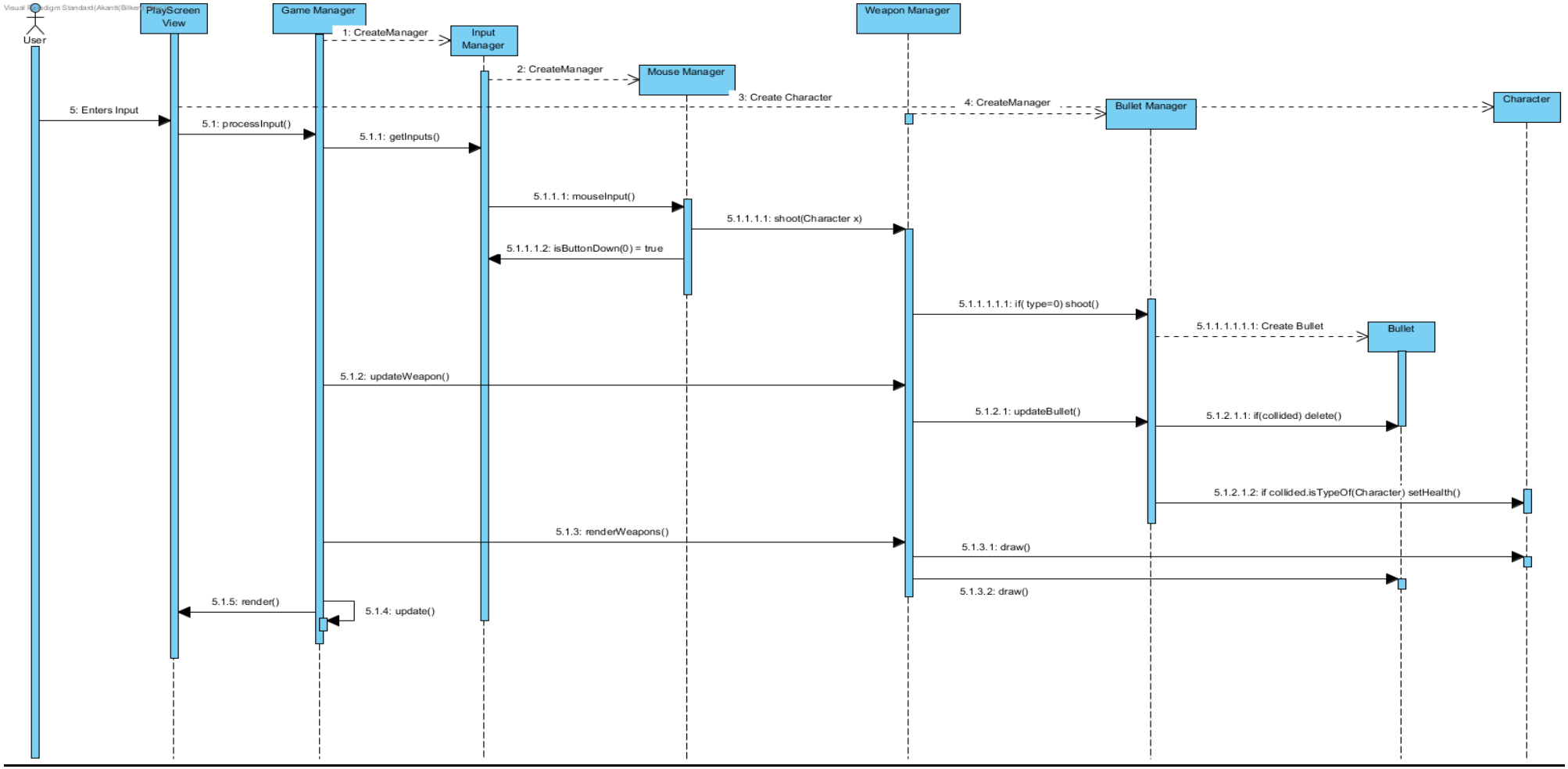
1. Player selects the settings button from the main menu.
2. Player uses the up-down direction keys to highlight volume bar.
3. Player uses the right-left direction keys to adjust volume.
4. The system updates volume.
5. Player presses “esc” key to exit the settings menu.

Alternate flow of events:

1. Player selects the settings button from the pause menu.
2. Player uses the up-down direction keys to highlight brightness bar.
3. Player uses the right-left direction keys to adjust Brightness.
4. System updates the screen brightness.
5. Player presses the “esc” key to go back to continue the game.

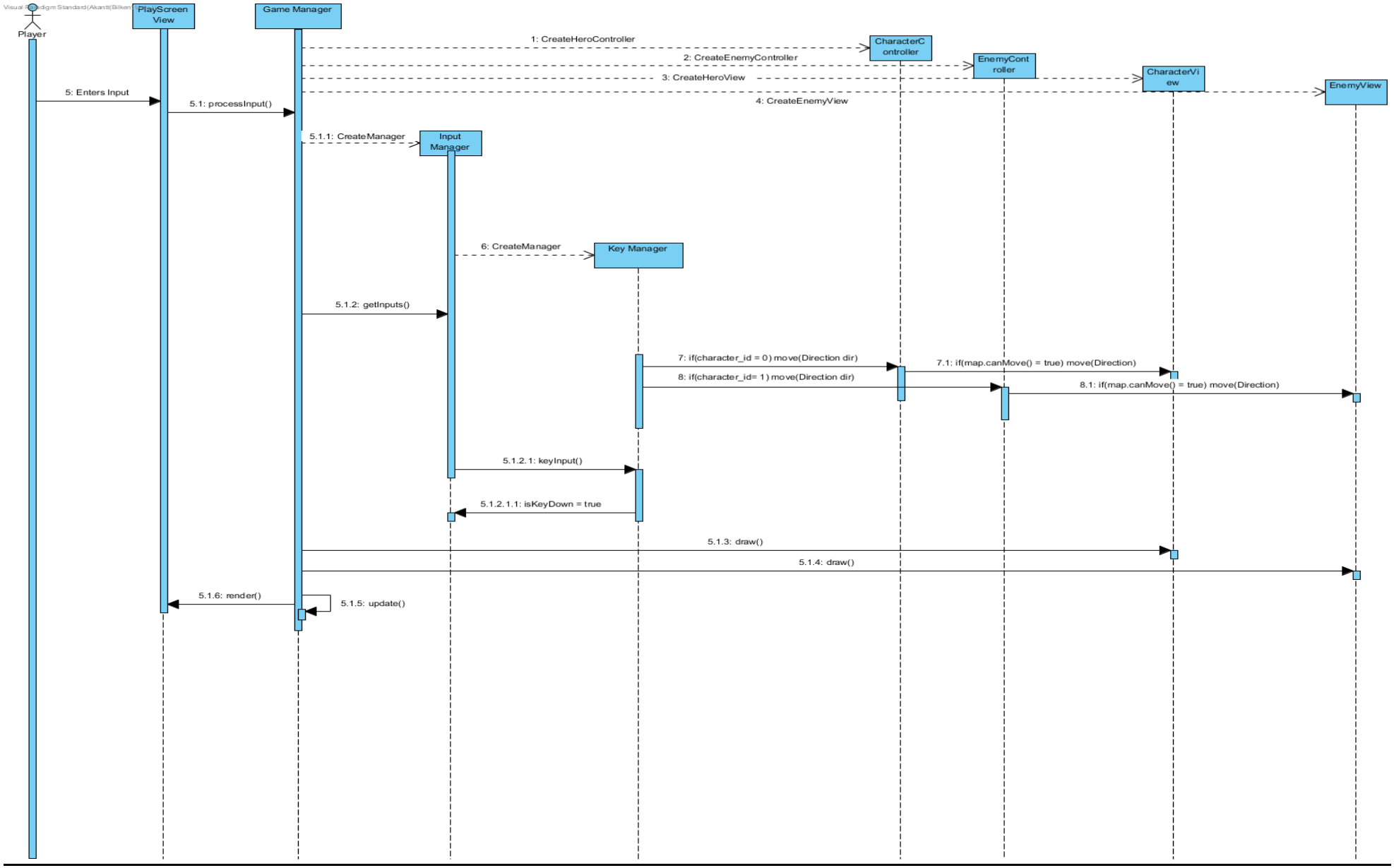
5.2 – Dynamic Models

5.2.1 - Sequence Diagrams



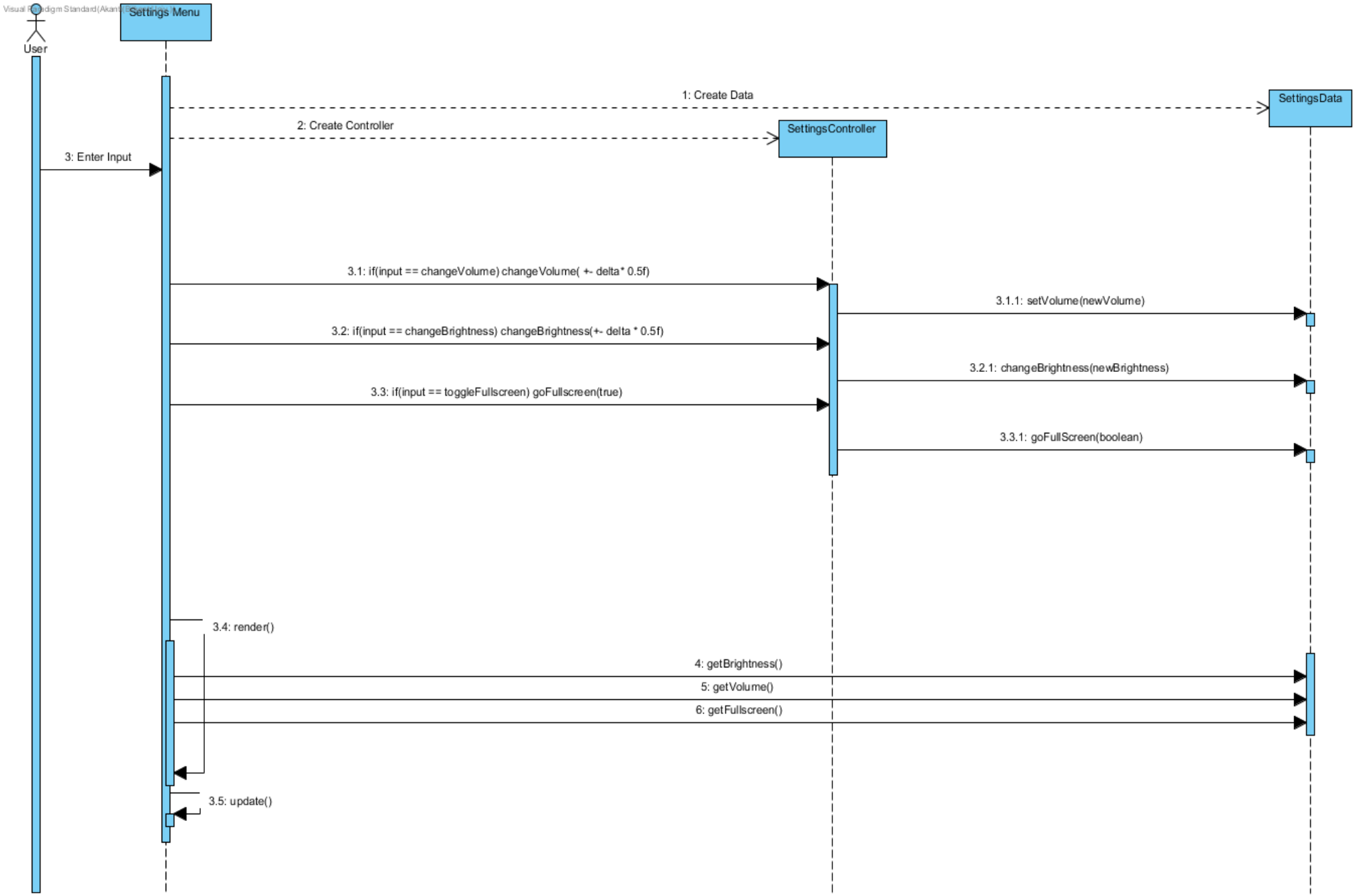
Scenario 1: **User Shooting**

User wants to play the game. User presses the play button in the main menu, and is taken to the **Play Screen view** boundary object. This object communicates and relays user's input to the **Game Manager** which serves as the primary control object of the game. The Game Manager then in turn creates an instance of **Input Manager** which creates **Mouse Manager**. The **Mouse Manager** determines which mouse button is pressed and when it is released. The **Game Manager** collects this information and relays it to the **Weapon Manager**. This creates Bullet Manager and tells it to perform shoot action and to update bullet count. When performing the shoot action the bullet manager creates **Bullet** objects and checks for collision with the **Character** and updates the **Characters** health points accordingly. Here the **Bullet** and **Character** are Entity objects. While all this are happening the Play Screen view calls the render functions of all the entity objects to draw then on screen.


Player
PlayScreen View
Game Manager
1: CreateHeroController
CharacterController
2: CreateEnemyController
EnemyController
3: CreateHeroView
CharacterView
4: CreateEnemyView
EnemyView
5: Enters Input
5.1: processInput()
5.1.1: CreateManager
Input Manager
6: CreateManager
Key Manager
5.1.2: getInputs()
7: if(character_id = 0) move(Direction dir)
7.1: if(map.canMove() = true) move(Direction)
8: if(character_id= 1) move(Direction dir)
8.1: if(map.canMove() = true) move(Direction)
5.1.2.1: keyInput()
5.1.2.1.1: isKeyDown = true
5.1.3: draw()
5.1.4: draw()
5.1.6: render()
5.1.5: update()

Scenario2:User wants to move:

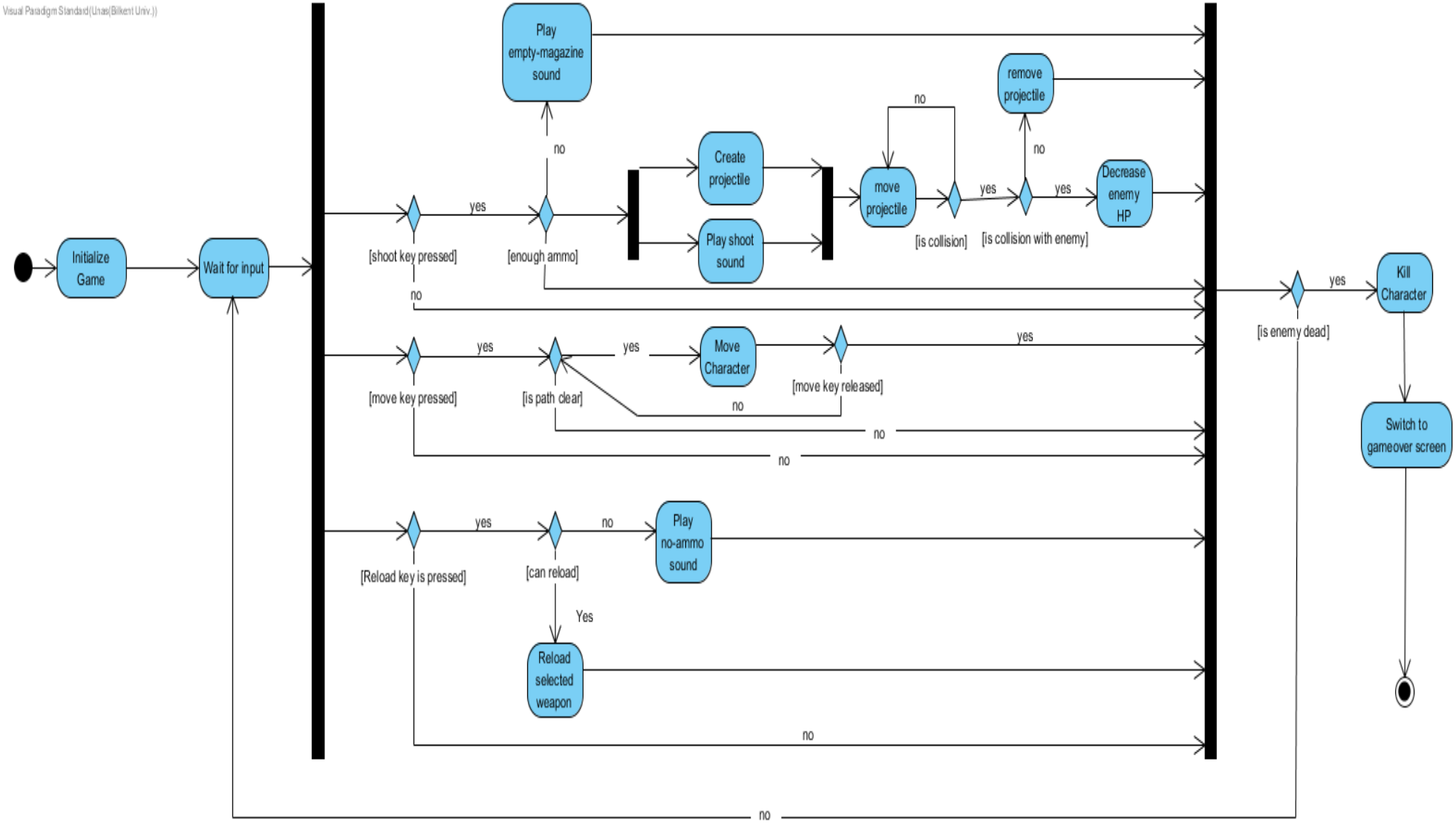
User wants to move while playing the game. User presses one of the defined movement keys. This movement key is taken by the current boundary object player is communicating with which is **Play Screen View**. This object will relay the user input to **Game Manager** control object which is the primary control object of our game. When game manager initialized it already creates instances of **Input Manage**r **, Weapon Manager** , **CharacterController** and **EnemyController** controller objects; also it creates **CharacterView** and **EnemyView** entity objects. Then; with the help of getInputs() method Game Manager relays this user input to Input Manager which already created **KeyManager** as a helper controller object. Input Manager relays this information to **KeyManager** to determine what action should be taken in return. **KeyManager** checks the user input if it is an enemy move or an hero move.(Hero is player1 enemy is player2.) After this check this class calls the move method of corresponding character by calling either **CharacterController** or **EnemyController's** move() methods. When one of these classes move() method is invoked; they communicate with MapControl and use it's canMove() method to determine if the user request is possible or not.If this request returns true these controllers calls move() method of either **EnemyView** or **CharacterView** according to which controller calls it. This method in view classes basically moves the object location to a new one according to the direction. After all these happened; Game Manager calls itself update() method which calls draw() methods of **CharacterView** & **EnemyView**. This update method also calls render() method of **Play Screen View** which allows the user to see the updated view of the game. Since **Game Manager** always calls update() method itself. This allows simultaneous communication between user inputs and display; because update () method also calls getInputs().


User
Settings Menu
1: Create Data
SettingsData
2: Create Controller
SettingsController
3: Enter Input
3.1: if(input == changeVolume) changeVolume(+- delta * 0.5f)
3.1.1: setVolume(newVolume)
3.2: if(input == changeBrightness) changeBrightness(+- delta * 0.5f)
3.2.1: changeBrightness(newBrightness)
3.3: if(input == toggleFullscreen) goFullscreen(true)
3.3.1: goFullScreen(boolean)
3.4: render()
4: getBrightness()
5: getVolume()
6: getFullscreen()
3.5: update()

Scenario3:**User wants to change Settings:**

User wants to change any of the settings while in the settings menu. User enters a form of input by either sliding the bars or toggling the options available. By doing this user communicates directly with the Boundary Object **Settings Menu**. This class has its own listener which allows it to determine the corresponding actions. Settings Menu already creates both **Settings Data**(Entity Object) and **SettingsController**(Control Object) when it is initialized. When user changes a specific settings; Settings Menu calls the specific method like changeVolume() etc. on **SettingsController**. When that specific method on **SettingsController** is invoked; controller calls the respective method of the entity Object(**Settings Data**) that allows controller to manipulate the entity objects data values. After the settings are changed **Settings Menu** calls it's render() and update() function which allows the view to be updated after each user change. render() function in **Settings Menu** gets all data from **Settings Data** entity object before each rendering to display the updated view

5.2.2 - Activity Diagrams



This activity diagram focuses on the play game flowchart to represent to data flow among play game.

Activity Diagram Description: (Play Game)

-The system will first initialize the game by creating the map and player and enemy objects and determine their initial positions.

- Afterwards the system will wait for input and do the following actions **concurrently**.

- The game will perform different checks to determine what the player wishes to do.

- As this is an action game the player can shoot move and reload simultaneously.

- For the move section the system will perform checks to determine whether the character can move to the directed path or not, this check will be repeated until the move key is released and the movement will stop.

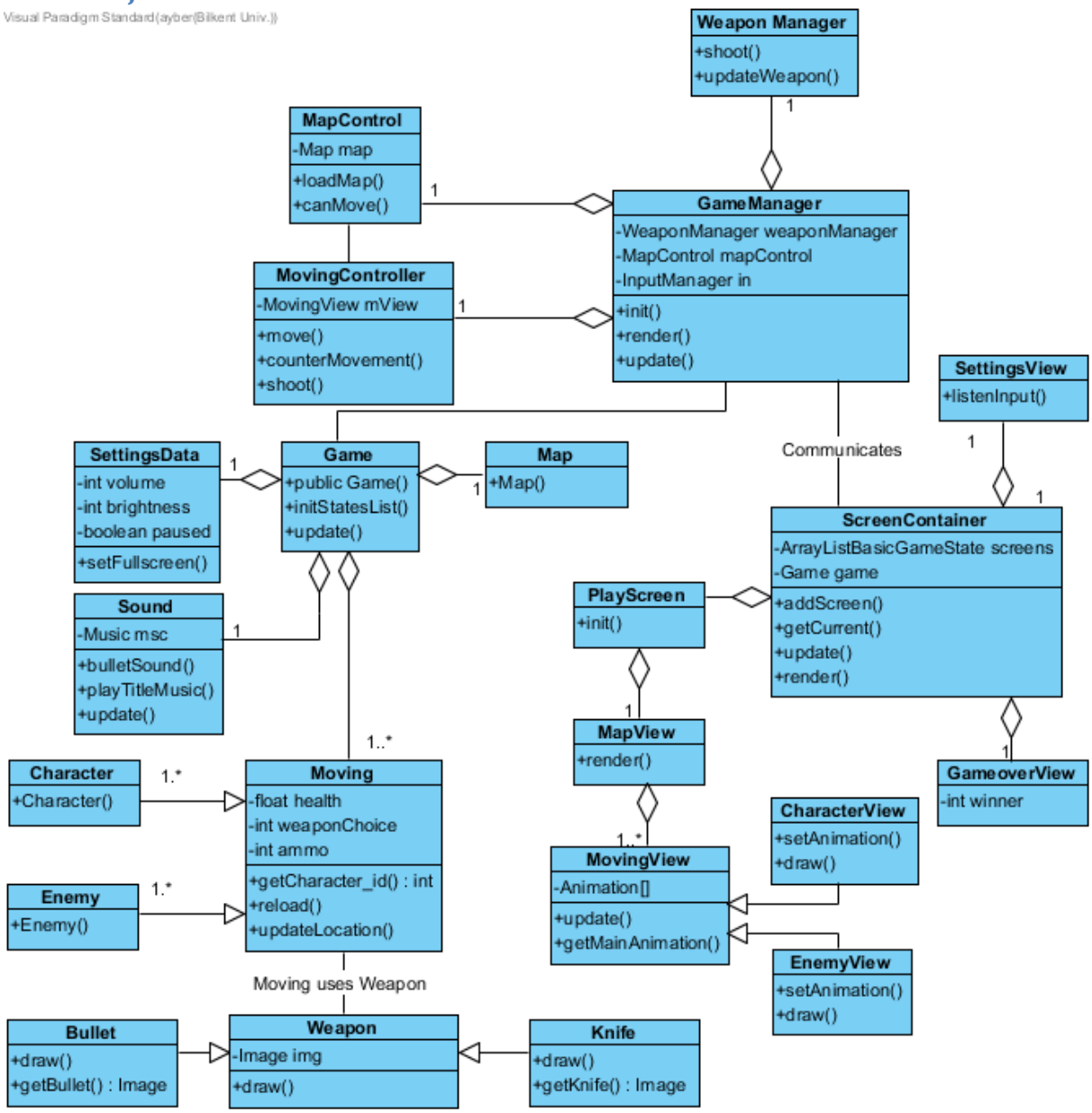
- When the reload key is pressed the system will determine whether the player can reload or this. This will depend on the amount of ammo the character has. If the player can reload than the system will update the weapon magazine accordingly and continue.

- For the shoot input the system will first determine if the player is able to shoot i.e. has enough ammo. If so the system will create the projectile and play the shooting sound clip.

- Afterwards the projectile will be moved until a collision occurs. Then if the collision is with an obstacle then the projectile will be remove and if the collision is with the enemy the system will decrease the enemy's HP.

- After each of these concurrent sections the enemy check will always be performed to determine whether the enemy is dead or not. If the enemy does die then the game will switch to the gameover screen and the playgame activity will come to an end.

5.2.3 - Object Model



Class Descriptions

GameManager:

The GameManager class will be the primary control class of the game. The GameManager will have instances of all the other managers like InputManager, WeaponManager, MenuManager, etc. When the GameManager is initialized, it will in turn initialize all the other control objects. This object will also act as a gateway between boundary and entity objects. This class works in 3 stages; initialization, update, render; these 3 methods are main part of this class; these methods extends to all other control objects if they have the same methods.

WeaponManager:

Weapon Manager is parent class for bullet manager and knife manager. Weapon manager holds instances of managers such as bullet and knife managers. This class determines the type of attack and sends it to the respective manager for collusion checks and updates. This class has mainly 2 parts; shoot: allows to create an attack. Update: allows to update all projectile attacks on the map.

MovingController

This is the parent class of CharacterController and EnemyController. This class will provide basic functions for characters like move and shoot. These functions will be called by the GameManager when it gets the movement input from the user. Countermove is used as a helper method to keep everything else at same spot while focusing on a specific character.This class communicates with MapControl to access its canMove() method to calculate collusions.

MapControl

This class will handle the map movements. The map control object will keep track of the character's movements by communicating with the MovementController and it will tell the movementController either a character can move on a specified direction or not. This way map control does the collusion check for movementController.

ScreenContainer

This class is the primary boundary class of the game. This class will hold instances of all the "screens" from other view classes in a list. This class will display a single screen and act like it is that class because of how observing works; the corresponding class will be notified from any changes when this class's update method is triggered. Also this class will be in charge of relaying all the information from the views to the GameManager. This class will receive decisions taken by the GameManager and will convey those decisions to the view that is held on screenContainer at that moment so that they can update accordingly. This class has basic operations like add screen or display screen.

SettingsView

This class will hold the view of the SettingsMenu. This class will render and update the visuals of the Settings Menu. The User will be brought to this view when the settings button is pressed in the MainMenu. This class will draw all the visual components related to the pause menu on the screen like the background and the

buttons. This class also has it's own listener to determine the changes done on the sliders etc. which makes it unique compared to other screens in screen container.

GameoverView

This view will be displayed when the User ends the game, either by winning or losing. The view will simply contain background, text and sound specific for game ending. It also holds a integer to determine winner which will be displayed on the screen.

PlayScreen

This class will hold the PlayScreen view. This view will be displayed when the user selects the Start Game option from the main menu. This class will contain an instance of MapView class. The methods in this class will be used to draw the game elements on the screen when the game first begins.

MapView

This class will contain the background map view of the game. This class allows direct rendering of the map class by holding its object. It will call it's render and display the map as an background object. By doing this we allow rendering our character over the map.

MovingView

This class serves as a parent class to all the views that are capable of movement on the map. These include: CharacterView, EnemyView, HealthBar and WeaponBar. The MovingView class also implements the Observer Interface, and this class is also instantiated when MapView class is intantiated. This way the elements that are subject to movement can be manipulated as a separate layer than the Map.

CharacterView/EnemyView

These views are responsible for drawing the character or enemy. These views will also take care handling the character/enemy's animations. This is achieved by keeping different frames of the character and enemy as image attributes of the classes. These classes inherit the MovingView as they are capable of changing locations on the map and being animate.

Game

This class will serve as the collection of all the game object models on the map. This class will also contain lists of states of all the game models. This class will also be the main link between the control part of the system and the model part of the system. The GameManager can ask for the state of the models and relay the information to the view side so that the views can be updated. Game basically has 2 very important roles. With it's constructor game creates all the states for our game. Since our game is a state based game because of slick2d library. It also initializes

these states with initStatesList() method which distributes the game container to all these classes to be able to represent every view with just one container.

Map

The Map class will hold the map data i.e. the main background of the map that will be used as the primary layer of the game. This object will be initialized in the Game class. The way map class works is that; it extends something called “TiledMap”. TiledMap is a 3rd party program we used to create our background map. This class basically loads the tiled map which is stored as a string reference and holds it. This way we can access Tiledmap’s properties to check collisions.

SettingsData

This class will hold the information that is needed by the SettingsController to make changes to the game settings. For example the volume data, brightness data and the current screen property. This class is consistent of basic set and gets methods to manipulate the data stored directly. Only difference is; if fullscreen is set this class notifies all observers about the change to make it displayed without delay.

Sound

This class will hold all the sound clips that are used in game. For example the MainMenu music, in-game music and end game screen music. This class gets updated by it’s update() method. If the option entered is a settingdata; the volume of the music will be adjusted accordingly. It also has a bulletSound() method that creates a sound based on the weapon used.

Moving

This class will serve as a parent class to all the object models that can move around on the map. The class will contain the location attributes, health, ammo etc. of all these objects. This class also inherits the Observable class to notify the observers every time there is a change in the model. This class is mostly formed of get and set methods; but it has a very important attribute: character_id. This attribute lets other classes know which character is this one. Since both enemy and character are moving; we need a id to differentiate them. This class also holds the sounds for character actions. getCharacterID() method of this class is used by game manager class occasionally to update health etc. This class also allows other classes to manipulate these location values by updateLocation() method.

Character/Enemy

These classes inherit the Moving class, and hold all the data of the character and the enemy. These classes provide methods to update the states of the Character/Enemy. These classes will also hold information like the Health of the

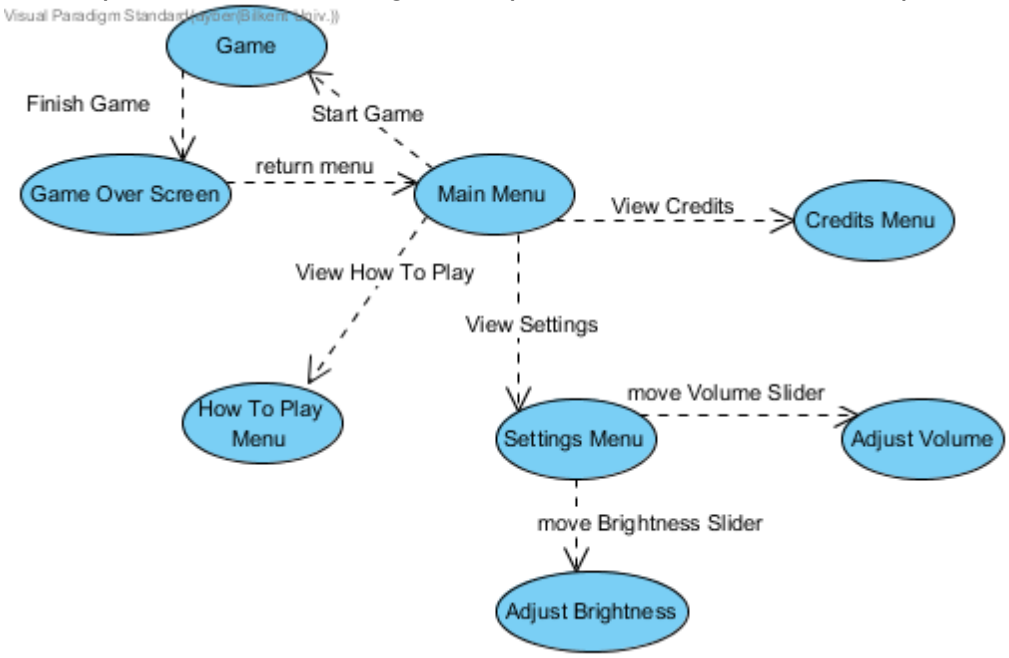
character or the weapons that the player currently has. These classes will also determine whether the Character/Enemy is alive or dead. These classes also has a method called randomTalk() in addition to parent class moving; which allows them to talk randomly during game.

Bullet/Knife

These classes inherit the Weapon class. These are the weapons that can be used by the character/enemy to defeat their opponent. These classes will store information like the bullet/knife dimensions, their move methods, their draw methods, their locations, etc. These classes have basic set and get methods with one exception. They have a draw() method which inherits the image from weapon class. Since we are going to store all weapon models in an array list of weapon's while they are still bullets and knifes actually; it is better to let weapon decide what image the model gets by its type.

5.3 - User Interface: Navigational Path & Screen Mockups

The explanation of the navigational path is embedded in mockup descriptions.



“Main Menu” is the screen which is displayed when the user first starts the game. This menu displays the title of the game and gives the user starting options. These options include “Play”, “Settings”, “How To Play”, “Credits” and “Exit”.

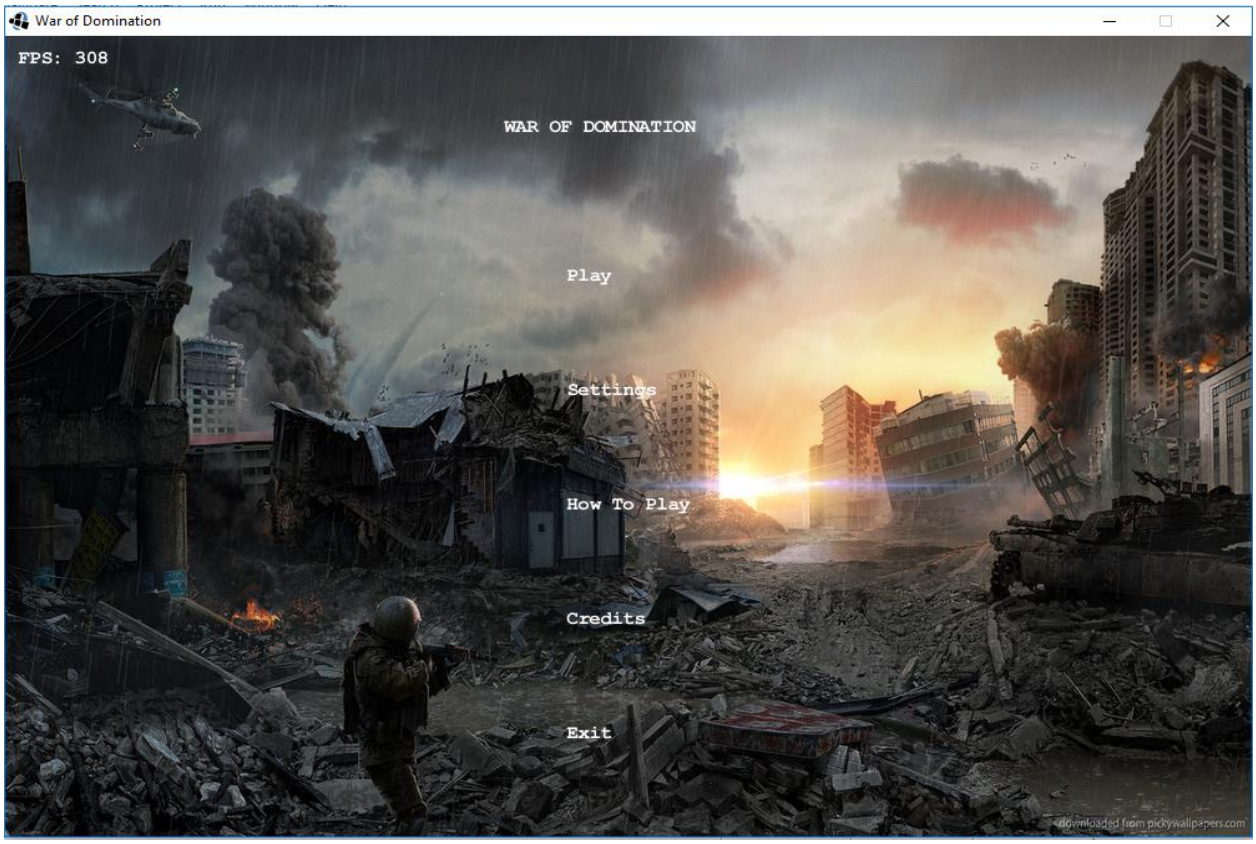


Figure 4 Main Menu Mockup

“Settings” screen can be reached from the Main menu. This screen allows the user to change game settings like volume, fullscreen and brightness.

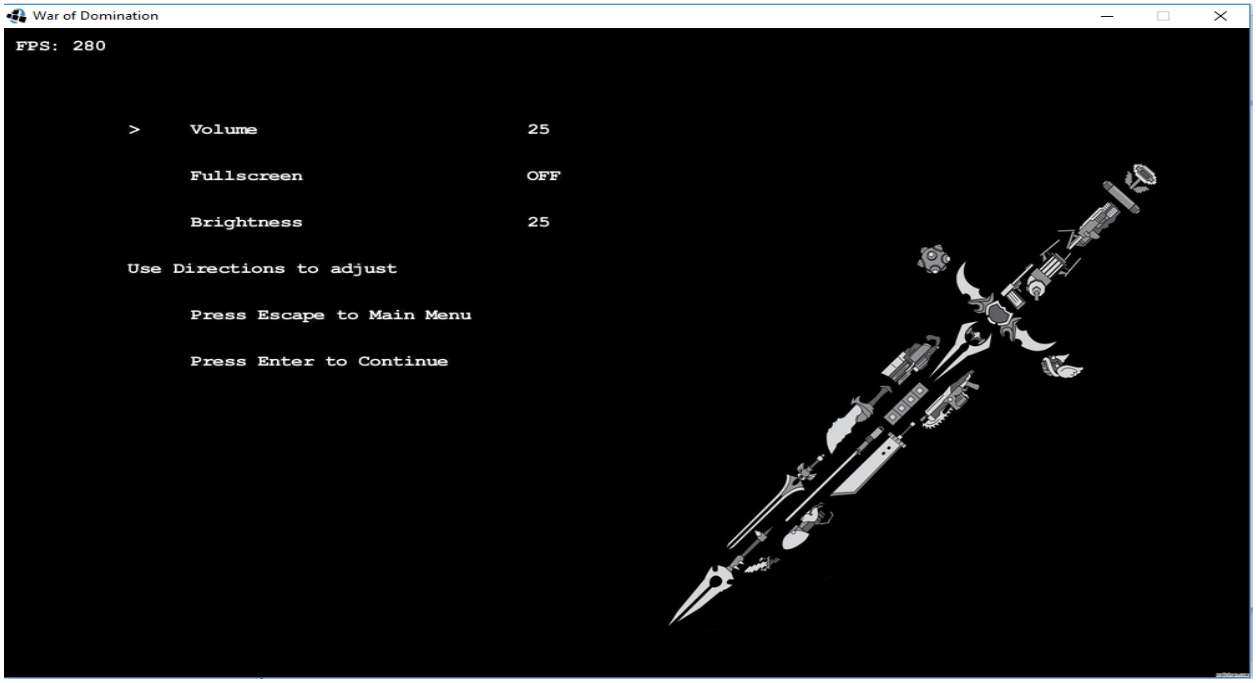


Figure 5 Settings Mockup

The “How to Play” screen can be reached from the main menu. This screen allows the User to learn the keyboard configurations used to play the game.

Figure 6 How to Play Mockup

This is the Main game screen displayed when the user is playing the game. This screen contains the main map where the user can see his character and the enemy. The respective health bars are displayed on the top right and left corners. The weapon bars are displayed on the bottom left and right corners.

Figure 7 Map Screen Mockup

The Game Over screen appears when the User dies or kills all the Enemies. The screen consists of a simple “Game Over” message and the user is given the option to return to the main menu screen.

6 – Improvement Summary

We have improved our game by adding these additional functionalities:

-Pick-up System
-Power-up System
-Additional Maps
-New Characters
-Character perks
-Non-functional requirement additions (Availability, Maintainability

Pickup System:

Pickup system is a new improvement to the game. This improvement allows user to pickup ammo, life , armor from the map by walking over them with their character.

Power-up System:

Powerup system is an addition to our game logic. This addition allows players to pickup power-ups from specific locations on map. These power-ups doesn't affect the player's character immediately. Player must use these power-ups manually by pressing a specified hotkey to get the effect temporarily on their character.

Additional Maps:

We have added additional maps to our game for user(s) to choose from. This will increase their enjoyment from the game.

New Characters:

We have added a list of characters to our game. User(s) can pick their character rather than being given by a default character like in previous iteration.

Character perks:

To make character choice more entertainint and more than just different view; we have added attributes to characters that would enhance a characters certain ability while diminishing other. This will allow different combinations of character combat for user.

Maintainability:

The system should not be shut down for maintenance more than once in a 1-hour working period.

Availability:

A new installation of the system shall be available for first-time use within 10 minutes of the start of the install.

7 – Glossary & References

MVC : Model View Controller

Bruegge, B., & Dutoit, A. H. (2003). Object-oriented software engineering: Using UML, patterns and Java. Upper Saddle River, NJ: Prentice Hall.

www.visual-paradigm.com