Bilkent University

Department of Computer Engineering

CS 319 Term Project

Group 1H

War of Domination

Design Report

Project Group Members:

Kazım Ayberk Tecimer

Akant Atılgan

Unas Sikandar Butt

Supervisor: Eray Tüzün

Contents

1. Introduction

1.1 Purpose of The System

War of Domination is a 2D turn-based strategy game which aims to provide users with an enjoyable gaming experience. The game consists of a user-friendly interface with How-to-Play tutorials and basic settings options. Users can choose to play in a single player mode or compete with each other using the multiplayer mode. The main aim of the player in the game is to control his character to defeat the enemy character. The game will come with multiple maps which the user can choose from to play from a selection of mods for extended user entertainment. The tools given to the player are a rifle, shotgun and a knife. Other weapons or power-ups may be found on the map.

1.2 Design Goals

Performance

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). Our system can justify this by showing the simultaneous FPS on the screen.We can achieve this by reducing the number of un-necessary calculations and object creation. Secondly we are using a Graphics library called slick2d to perform actions like draw(), render() (which are very common in our game) more efficiently. Finally we can optimize most commonly used methods from the library if we cannot reach our goal.

Usability

The game will provide a friendly user interface which can be navigated with ease by the use of both the mouse and the keyboard. For example the user should be able to see the manipulations happening on the interface upon interaction. Also the meanings of the texts in main-menu should be crystal clear to avoid any confusion.(Ex: How-To-Play:Opens a tutorial. etc.) To increase usability in our game we allowed user to manipulate any kind of input buttons according to their specifications. Users can easily bind any valid action to any key on their device. Therefore we implemented our design as in a way that our game works with both mouse and keyboard in case user cannot access one specific device like “mouse” as an example. This way the gaming experience of the user is not gated behind a single input device.

Efficiency

Efficiency is one of the systems main goals.Our system should be efficient enough to ensure user can interact with the game in less than 0.2 seconds. The system will run moderate graphics so that the game performance is not gated behind the quality of images. To ensure, to reach this goal we implemented our system in a way that Image files will be compressed to the limit as to not hold any extra data. This will also ensure efficiency in the management of data. Also the maps and images will be loaded before game initializes , so loading time of the game will take less time. Additionally the system won’t render the objects that are not currently visible to the user to avoid excessive object rendering.

Scalability

System should be able to expand its multiplayer capabilities upwards, e.g. (The game should be able to support at least up to 2 players playing the game

simultaneously.) To achieve this we have implemented our design in an hierarchical way that which allows system to support more than 1 players. Our system also used observable design pattern which allows us to add more characters into the game if we need to.

2 High Level Software Architecture

2.1 Subsystems Decomposition

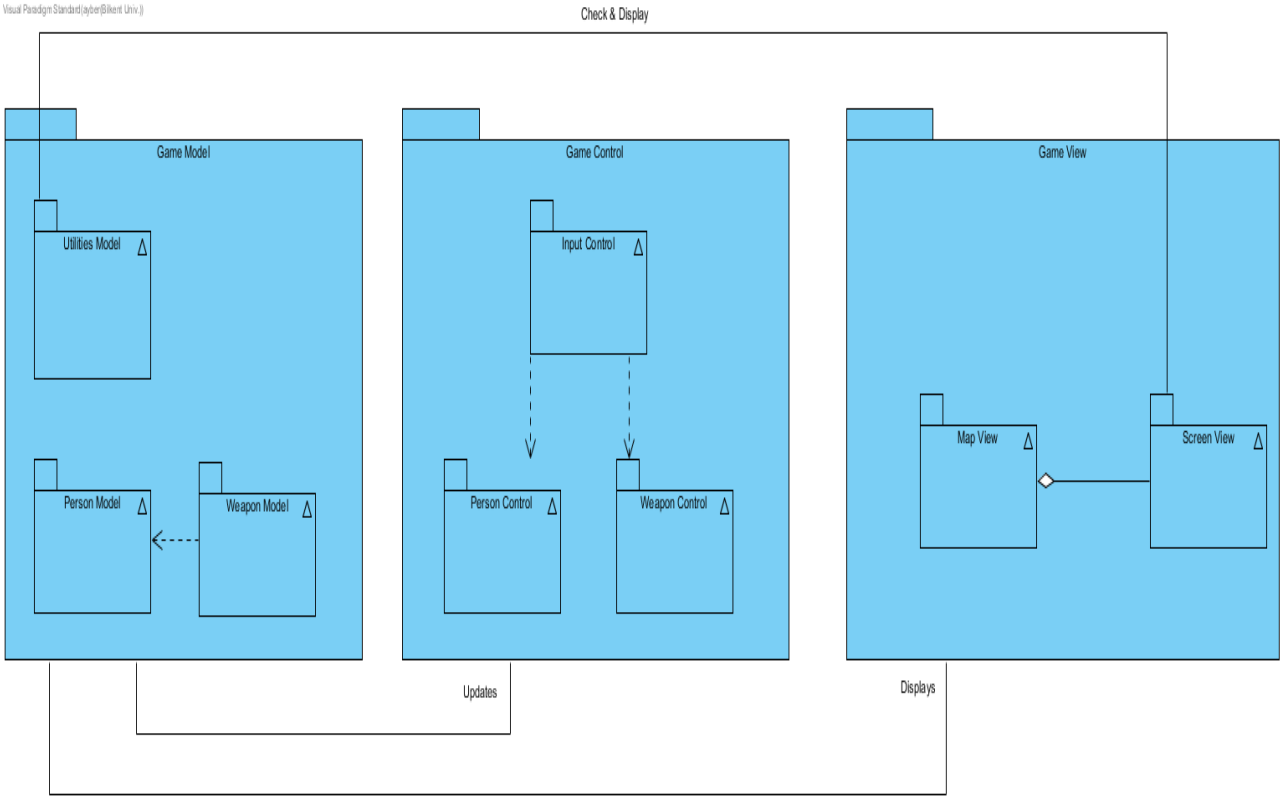


Figure 1: Subsystem Decomposition

Our system is decomposed into three parts. These are, **Game Model, Game Controller and Game View**. The system adopts a MVC(Model-View-Controller) architecture.

Game Model subsystem contains 3 subsystems **(Utilities, Weapon, and Person)**.

- **Game Model** subsystem responsible for storing necessary data of the **Weapon, Person and Utilities**.

Game Control subsystem contains 3 subsystems **(Input Control, Person Control, and Weapon Control**).

- **Input Control** subsystem is for controlling the inputs.
- **Person Control** subsystem manipulates the **Person** in **Game Model.**
- **Weapon Control** subsytem manipulates the **Weapon** in **Game Model.**

Game View subsystem contains 2 subystems (**Screen View, Map View)**

- **Screen View** is for displaying screens of the game.
- **Map View** renders **Game Model** elements.

Screen View subsystem checks **Utilities Subsystem** and displays according to the data stored in **Utilities subsystem.**

When **Game Control** subsystem manipulates any of the subsystem in **Game Model** subsystem, **Game View** Subsystem gets notified and **Map View** displays the updated models.

When user interacts with the **Screen View** Subsystem, and according to this interaction **Game Control** Subsystem updates **Game Models**.

Weapon Control and **Person Control** directly communicates with **Game Model** subsystem to change the data of Game Models.

Input Control communicates with **Weapon** and **Person Control** subsystems when user interacts with Screen View subsystem.

2.2 Hardware/Software mapping

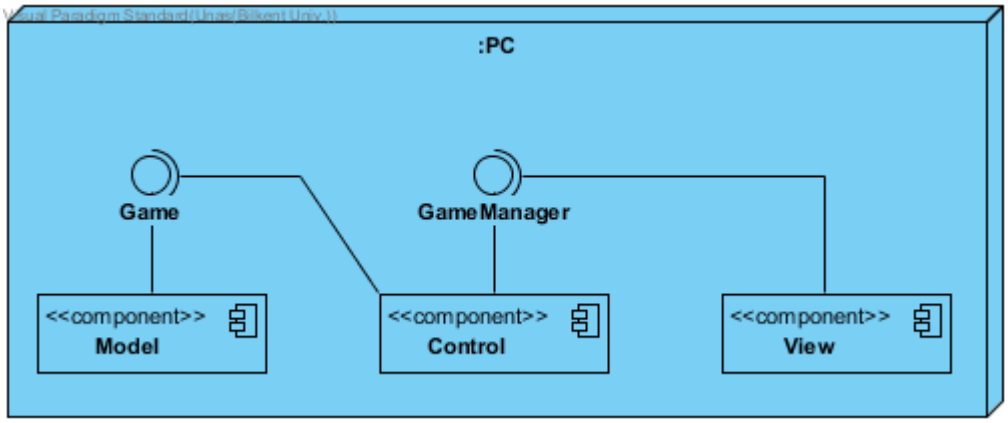


Figure 2: Deployment Diagram for Hardware& Software Mapping

The system does not need any additional hardware to run, beside users' computer. It also does not require any internet connection or a server. As our system runs on one computer (local host) there are no needs for servers therefore we only have 1 hardware in deployment diagram.

For the software, as the game is being implemented using Java language the system will require JRE version 1.8 or higher. The game will also require Slick2D and lwjgl (check glossary) libraries to installed on the system.

2.3 Persistent data management

To manage the data on our system we decided to use the file system as our game does not require usage of any complex database. The only persistent data that the game will generate are the **user highscores**, **and data related with map in txt file**. Therefore, all the necessary files and data will load onto file system which can be stored as a .txt file.

2.4 Access control and security

Our system is not concerned with access control as our system has only 1 user. User is allowed to access any type of data on the system. This can be confirmed by the use case diagram produced in the analysis phase of the development. For this reason no authentication functionality is provided in the

system. Furthermore, as the game does not require any internet connectivity, the system is secure against unauthorized access from online entities such malicious software. To secure the system against user modifications, the game will be provided as a .jar file which will ensure that the source code of the game is not easily modified.

2.5 Boundary Conditions

Since there are sensible data that need to be managed in the game, how this management will be handled in different cases must be stated in boundary conditions. There are three boundary conditions for the game.

2.5.1 Initialization

The game will be initialized by executing the .jar file in PC. The Game and GameController will be initialized in the main function and then the Game Controller will initialize Game's state which is the state of MainMenu, then the Game updates itself to notify the ScreenContainer to display the current state. The images of the first state are loaded and the music that will be played in the first state will be loaded and played. Then the game loop for the current state begins and the flow of events are determined according to user input afterwards.

2.5.2 Normal Termination

The game will terminate normally when user clicks the button "Quit" or click X from the top right corner whether he is in the mid game or in the main menu. The game will save the persistent data (highscore) to the file that will be determined later (probably a flat file which will be encrypted.) Terminating the application will be handled by Java Runtime Environment.

2.5.3 Termination with Failure

To avoid data corruption due to failure that may result from various reasons, the game have mechanisms for backing up the data and in case of failure, it will

restore them after the next initialization of the game. However, the backup data may not be up to date.

3.Subsystem Services (please zoom x3)

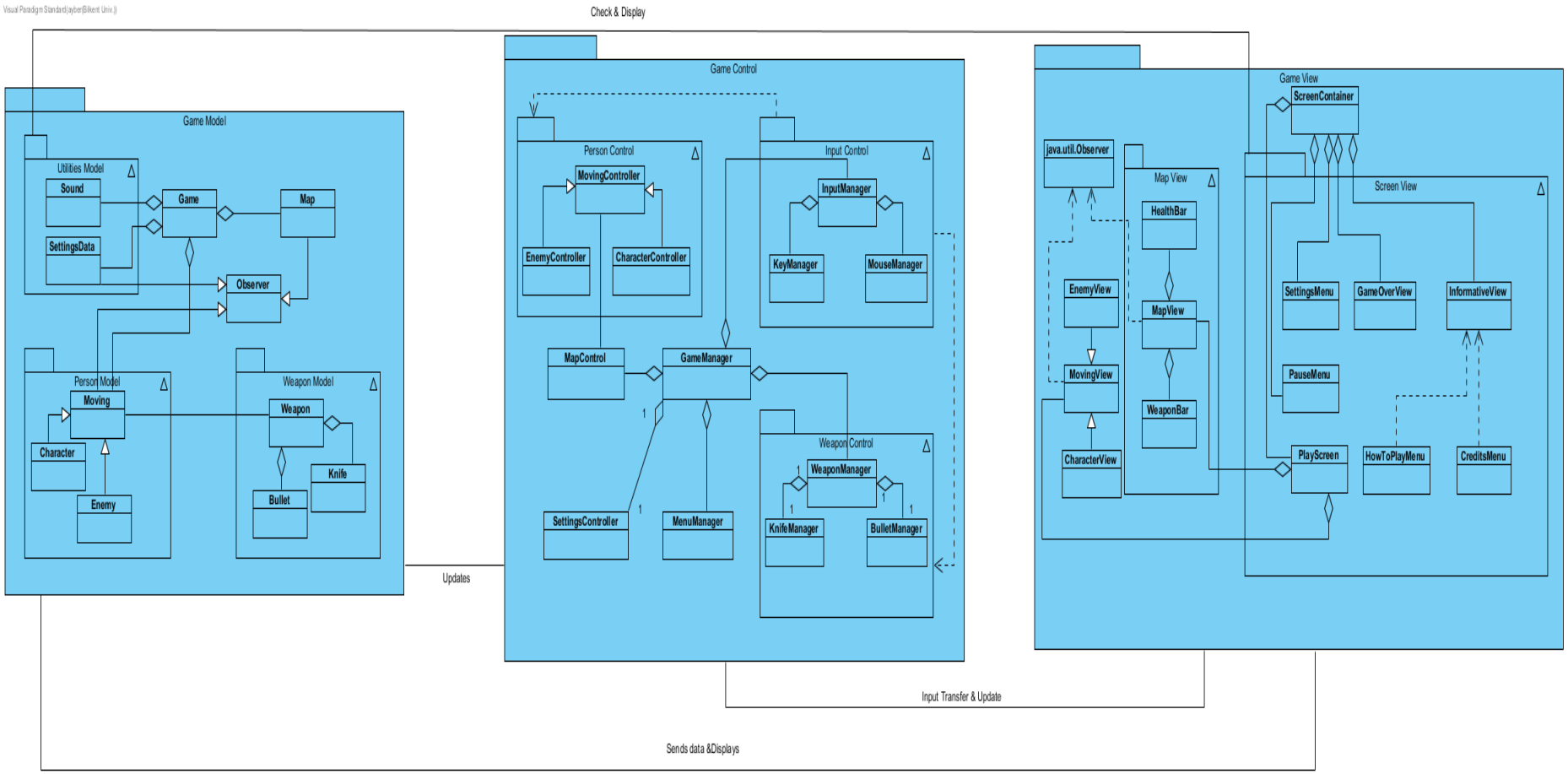


Figure 3: Subsystem Decomposition

3.1 Game Model Subsystem

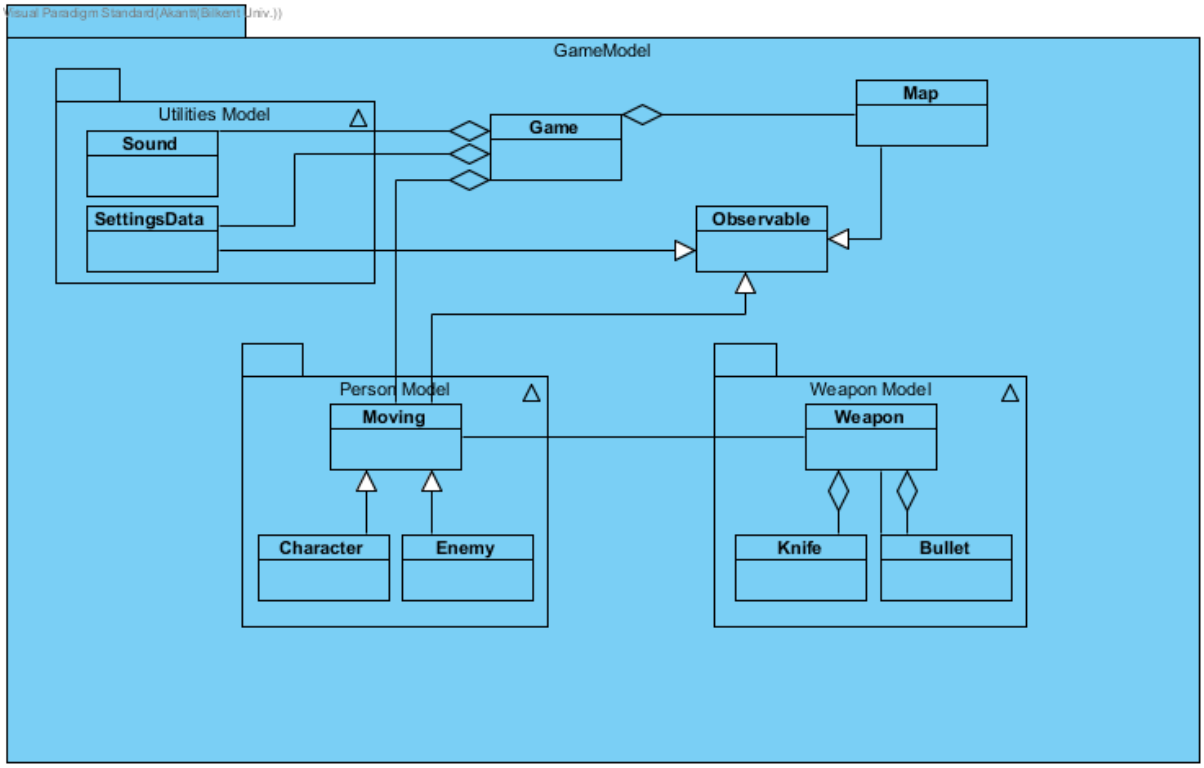


Figure 4: Game Model Subsystem

Game Model subsystem represents our models. This subsystem consists of 3 subsystem(Utilities, Person, and Weapon) and also 3 classes(Map, Game and Observable). **Game Model Subsystem** is responsible for storing all of our game models data.

Subsystems inside **Game Model Subsystem (Person, Utilities, Weapon)** can only be manipulated by **Game Control Subsystem.**

Weapon Model Subsystem can access **Person Model Subsystem's** data but cant change it.

When **Person Model Subsystem** or **Utilities Model Subsystem** gets manipulated by **Game Control Subsystem (please check 3.3); Game Model Subsystem**

sends updated **Models(Person, Weapon, Utilities)** to **Game View Subsystem(please check 3.2)** to be displayed.

3.2 Game View Subsystem

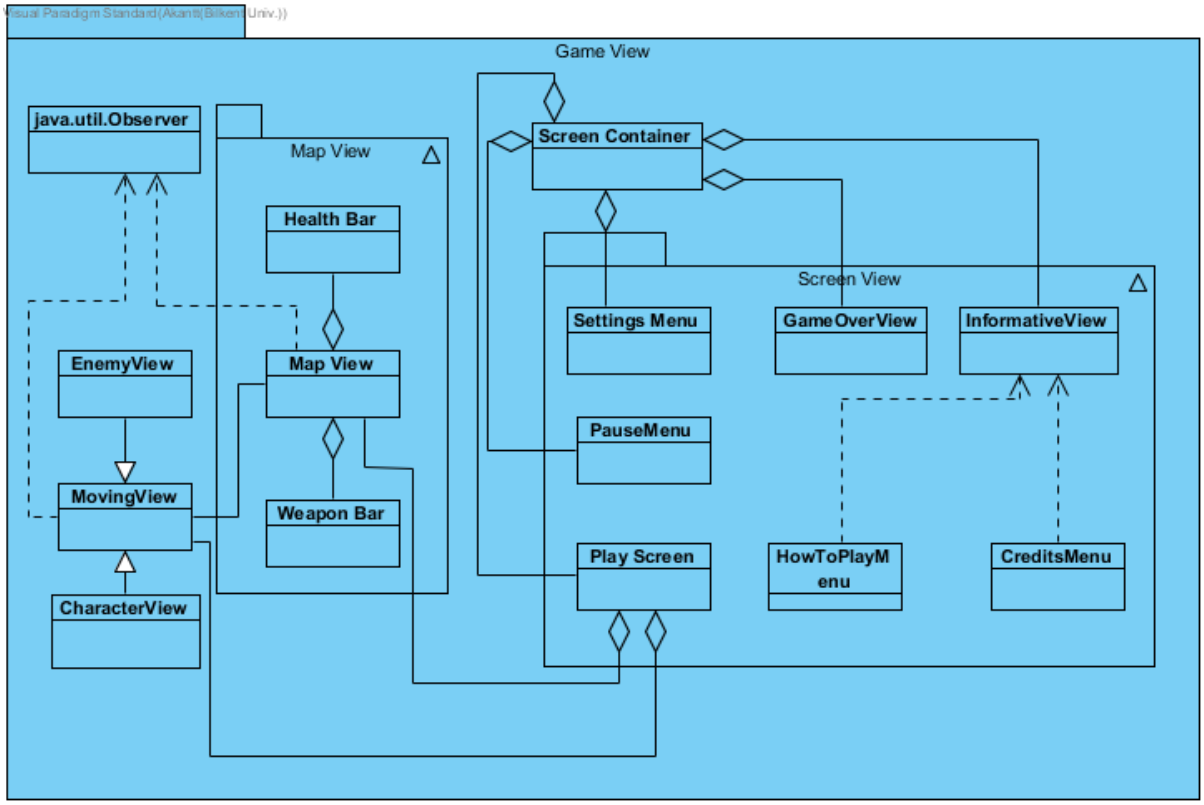


Figure 5: Game View Subsystem

Game View subsystem is responsible of displaying **Game Model Subsystem** elements and **ScreenView**. **Game View** subsystem is also responsible for receiving user interaction and sends this interaction to **Game Control subsystem** (please check 3.3).

ScreenView consists of the following view classes: (these classes are **briefly** explained as they are crucial for **ScreenView).**

-Settings Menu: Displays a settings menu; has its own listener to create real-time user interaction.

-Pause Menu:Holds the view for a pause menu when game is paused.

-Play Screen: Main Game screen.

-HowToPlayMenu:Holds the view for a how to play menu.

-CreditsMenu:Holds the view for credits.

-GameOverView:Holds the view for game over menu.

Game View Subsystem communicates with following **Game Model** Subsystems (**Utilities Model Subsystem, Person Model Subsystem**)**. ScreenView Subsystem** gets **Game Models data** from **Game Model Subsystem** and display it.

MapView Subsystem re-display when **Utilities Model** and **Person Model Subsystems** are updated by **Game Control**.

3.3 Game Controller Subsystem

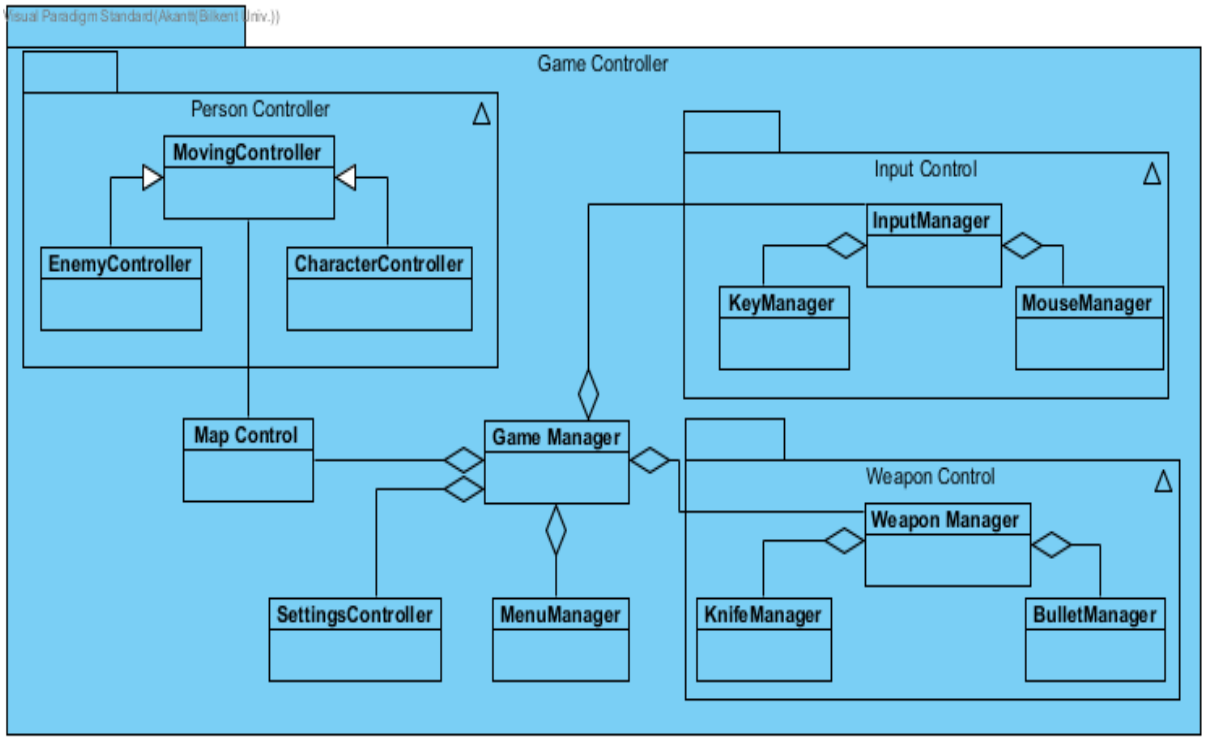


Figure 6: Game Controller Subsystem

Game Controller subsystem is responsible for controlling and handling the game.

Game Controller Subsystem consists of 3 subsystems and 4 classes.

-Input Control Subsystem

-Weapon Control Subsystem

-Person Controller Subsystem

-Map Control class:Helps with collision check for **Person Control**.

-Settings Controller class: Manipulates **Utilities Model Subsystem. (**in whole class diagram it is shown by association named **Updates,** check figure 3**)**

-MenuManager class: This class receives inputs and updates **Game View Subsystem.**

-Game Manager: It is responsible for updating **Game Model Subsystem** elements and responsible for receiving input from **Game View subsystem. (**in whole class diagram it is shown by association named **Updates** check figure 3**)**

Input Control Subsystem:

When user interacts with the **ScreenView Subsystem, Input Control Subsystem** is invoked.**Input Control** subsystem is responsible for processing inputs. To process input, **Input Control Subsystem** invokes both **Person Control** and **Weapon Control Subsystems. (**in whole class diagram it is shown by association named **Input Transfer& Update** check figure 3**)**

Weapon Control:

Weapon Control Subsystem is responsible for controlling and updating **Weapon Model**. It is invoked by **Input Control Subsystem.**

Person Control:

Person Control Subsystem is responsible for controlling and updating **Person Model** according to user inputs**. Input Control Subsystem** invokes the **Person Control Subsystem.**

4. Low Level Design

4.1 Object Design Trade-Offs

Memory Consumption vs Visual Quality

Our game is based on tiles(32x32 pixels = 1 tile). We can adjust the quality of tiles or get HD tilesets to generate our maps from tiled map. But this will be a problem when we have multiple maps with full .png files to render them. Images are so far the most memory consuming parts of our game so we choose to use moderate quality tilesets in our maps and moderate quality picture as map background to save memory. While this choice sacrifices from visual entertainment the difference is not worth a few more MB of hard drive space consumed. Also it decreases the loading time of the game since each map picture is pre-loaded into the game when the game is launched.

Game Efficiency vs Scalability

Our game should be efficient enough for users to enjoy without spending time by waiting computer to response. 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.) Therefore we designed our system in a way that response time is less than 0.2. However, we want our system should be able to expand its processing capabilities upward (e.g. Game should support the 2 players to play the game simultaneously). On the other hand, Increasing the number of the player decreases the game efficiency of the system. Hence, we only can support 2 players in system to users to be able to interact with the game less than 0.2 seconds.

4.2 Final object design (please zoom x3)

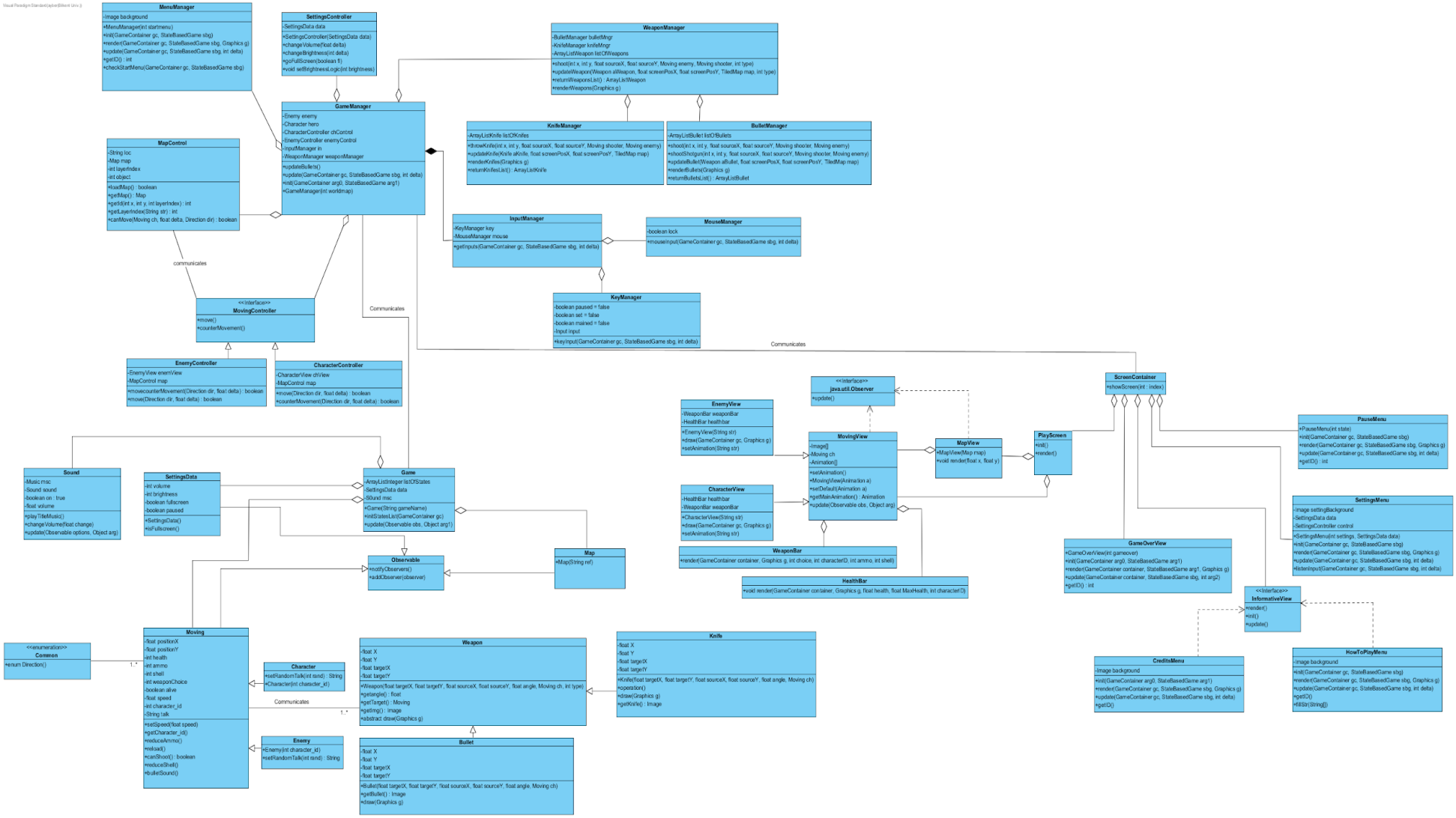


Figure 7: Final Object Diagram

4.3 Packages

4.3.1 org.newdawn.slick.command

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

4.3.2 org.newdawn.slick.gui

This package includes some extremely simple GUI elements which should be used where a game does not require a full GUI

4.3.3 org.newdawn.slick.tiled

This package contains utilities for working with the tiled utility for creating tiled maps.

4.3.4 org.newdawn.slick.state

This package is for creating state based games which allow the game to be broken down into the different activities the player may take part in, for instance menu, highscores, play and credits.

4.3.5 org.newdawn.slick.loading

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

4.4 Design Patterns

The system uses a **facade design pattern** for subsystem communication. The Model subsystem provides a Game class which is the facade class for the model subsystem. This class delegates requests to the appropriate components within the Model subsystem. The Control subsystem uses this Game class as an interface to the Model subsystem. Another advantage of using this facade class is that it does not need to change if any of the models are changed.

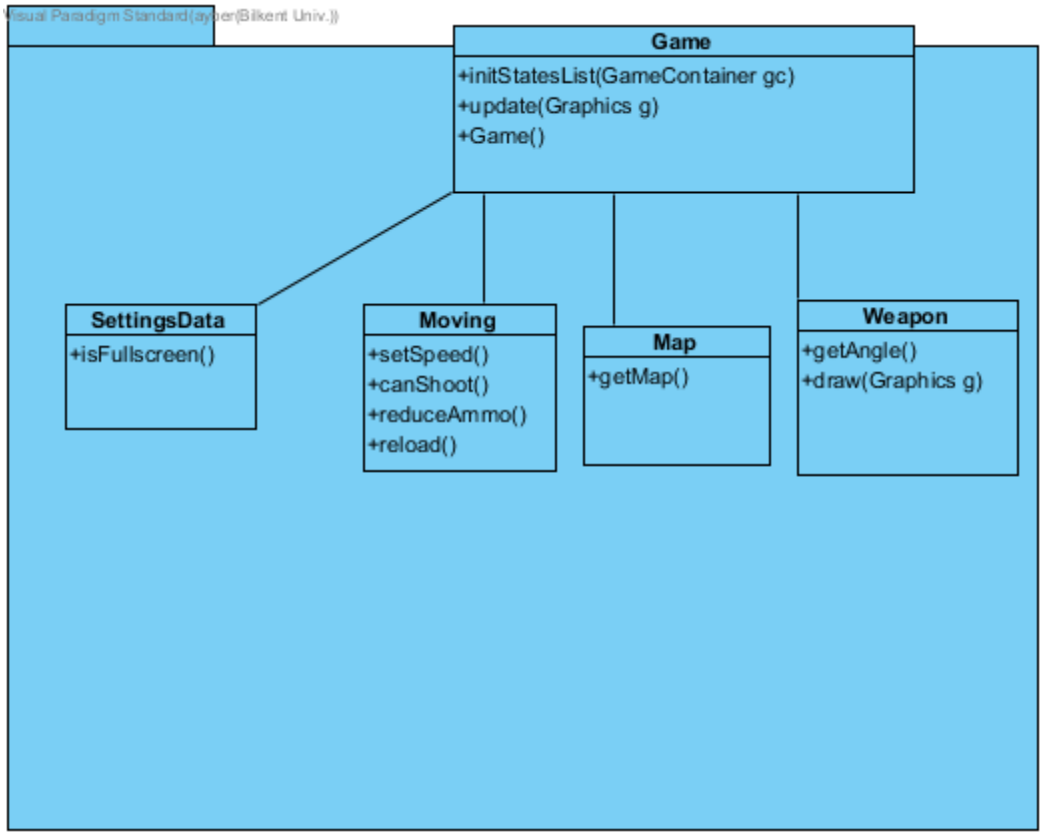


Figure 8: Facade Design Pattern for Game Model Subsystem

Similarly the GameManager acts as a facade class for the Control subsystem. This class is used by the View subsystem to receive control decisions from the Control subsystems.

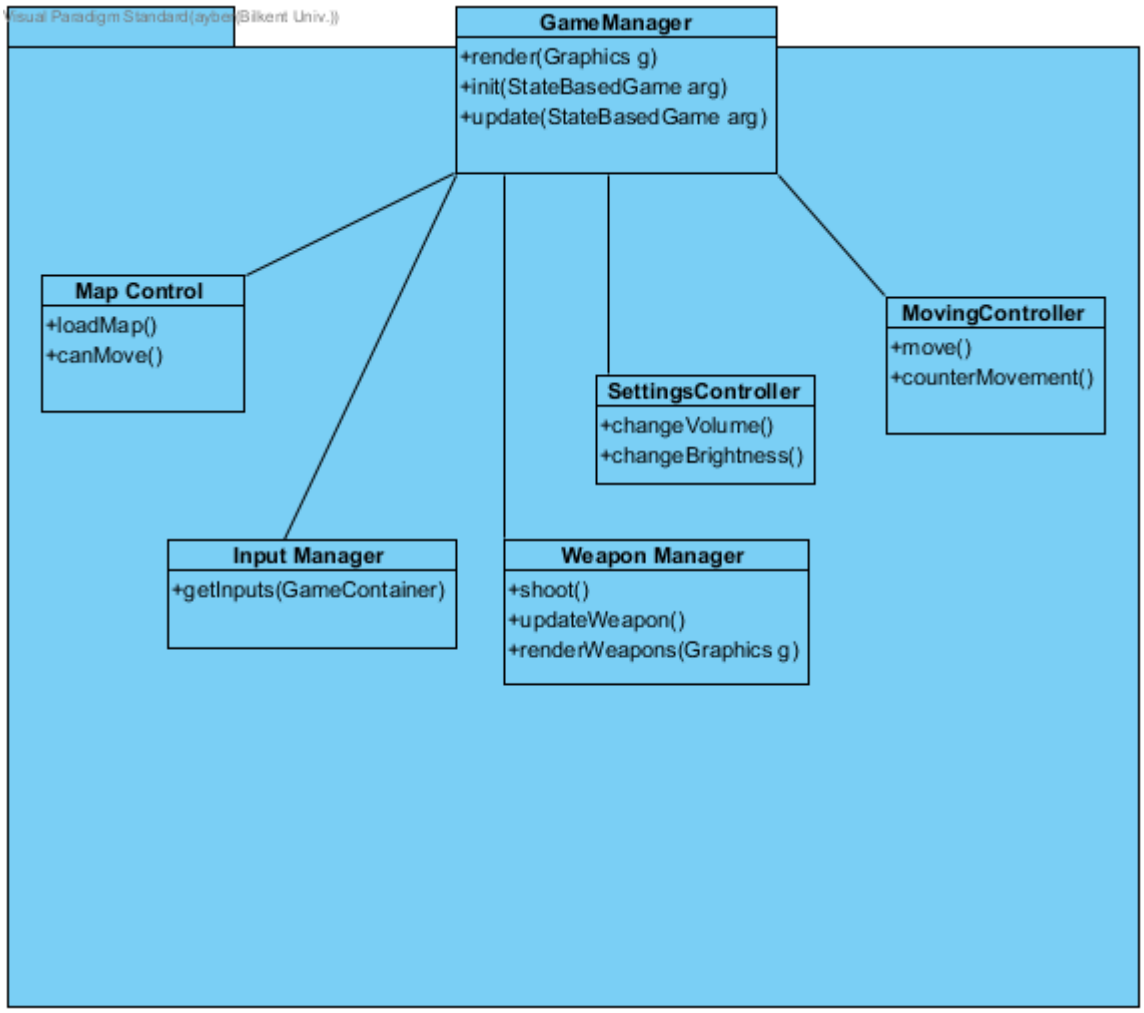


Figure 9: Facade Design Pattern for Game Control Subsystem

Observer Design Pattern: We have used the Observer Design Pattern to create a link between entity objects that are frequently updated with their view counterparts. To do this we used the Observer interface provided in java.util library. Using this design pattern the view`s of these entity objects can be notified by calling notifyObservers() method of java.util library on the model objects. This way when we change the model objects we can notify the views

of these models to make them re-display these changed models. This communication is directly in between model and view classes so there is no delay because of in-direct communication. The view then updates itself according to the changes made to the entity object.

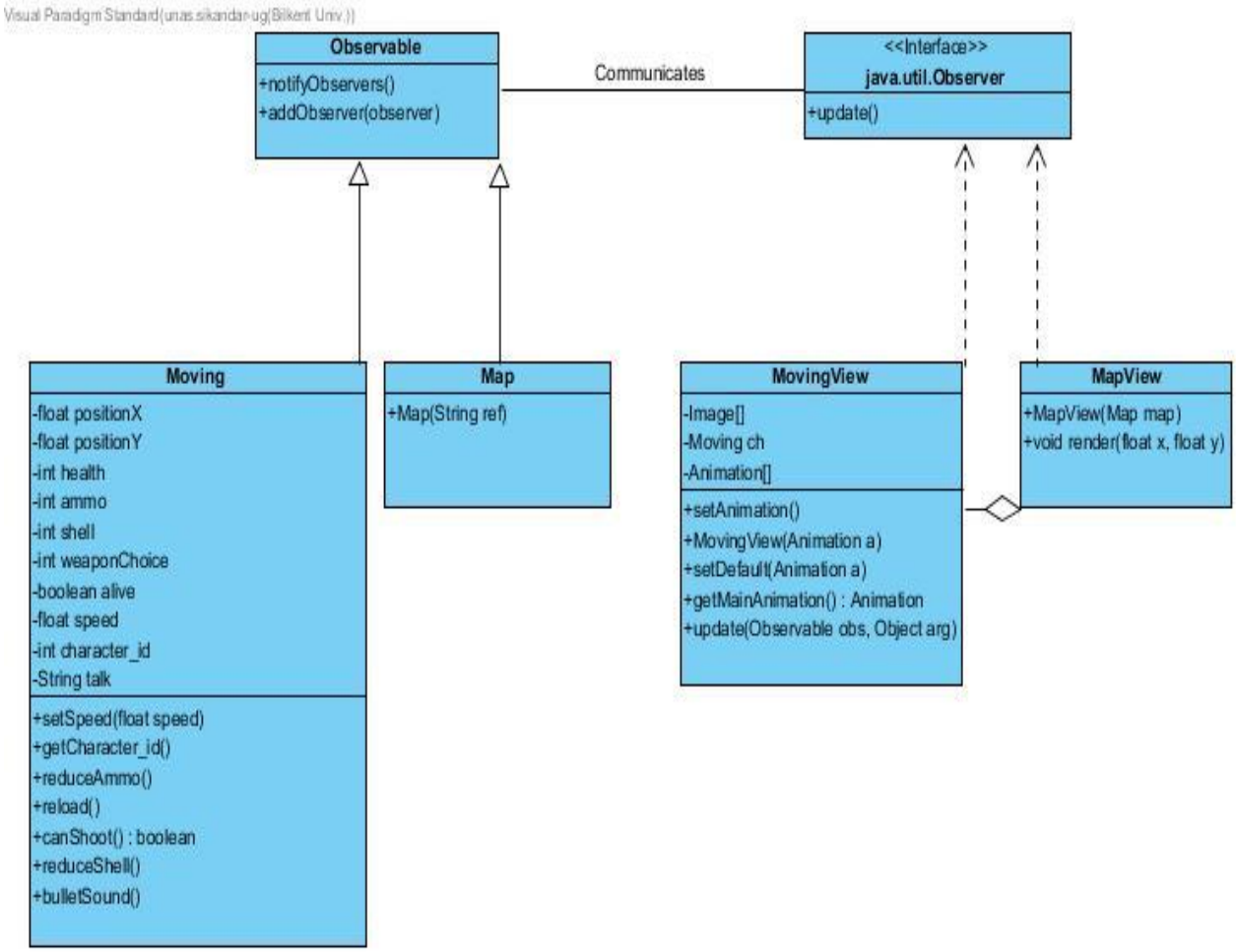
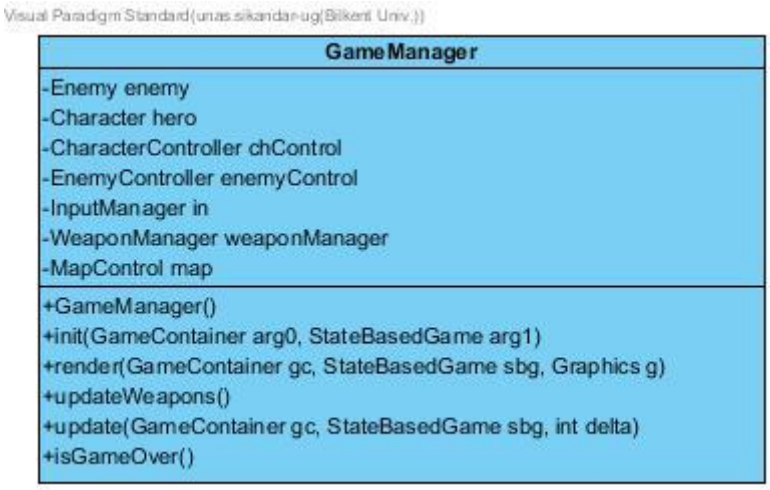


Figure 10: Observer Design Pattern

4.5 Class Interface

4.5.1 GameManager Class



Attributes:

- **Character : hero**

This is the main instance of the Character object that the User will control.

- **Enemy : enemy**

This is the instance of the enemy object, which the User will have to defeat.

- **CharacterController : chControl**

This attribute is used to relay input commands to the CharacterController which can interpret these commands into Character actions.

- **EnemyController : enemyControl**

This attribute is used to relay input commands to the EnemyController which can interpret these commands into Enemy actions.

- **InputManager : inMngr**

This control object attribute will be used to check the validity of user inputs.

- **WeaponManager : WeaponManager**

This attribute will keep track of the weapons in the game. Our weapons in the game doesn't belongs to a specific character, they are given a starting position based on the character or enemy shooting them.

- **MapControl: Map**

This control object will be used to calculate Character's and Enemy's collision checks for movement.

Operations:

+ void : GameManager()
This is the class constructor. It will create other control objects like Weapon Manager , MapControl, InputManager.

+ void: init(GameContainer arg0, StateBasedGame arg1)
This is the method used to initialize the game when the game is first started. It creates all necessary model-view-control objects.

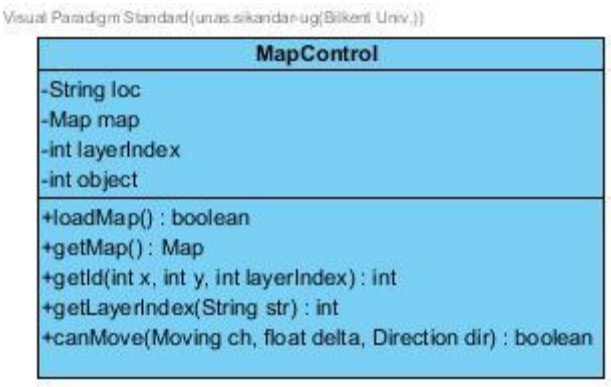
+ void: render(GameContainer gc, StateBasedGame sbg, Graphics g)
Render method draws all necessary display components on the screen by using Graphics Component “g” passed as a parameter. So other view classes can send their display to this method for Game Manager to render on them. This method also calls other render functions of other control objects if necessary.

+ void : updateWeapons()
This method is used to calculate weapon trajectories and collisions while updating their position. This method is provided in this class to reduce communication time and enhance performance by directly connecting to Weapon Manager control object.

+ void : update(GameContainer gc, StateBasedGame sbg, int delta)
This method is called periodically to update Model objects data. It also checks user inputs by calling Input Manager’s getInputs() method while isGameOver boolean is false. This allows instantaneous results displayed when user presses a valid input.

+ boolean : isGameOver()
This method checks if the game over state is reached by checking character lifes. This method is called on each update() of Game Manager so if the game ends Game Manager can instantly do the necessary actions.

4.5.2 MapControl Class



Attributes:

- String : loc
This will store the location string of the current location of the map in the game window/container.

- **Map : map**

This will hold the map object of the game which will be controlled by the MapControl.

- **int : layerIndex**

This will hold the current layer index of the map. Since our map is formed from layers this value is very important for us while calculating collision checks.

Operations:

+ **Boolean : loadMap()**

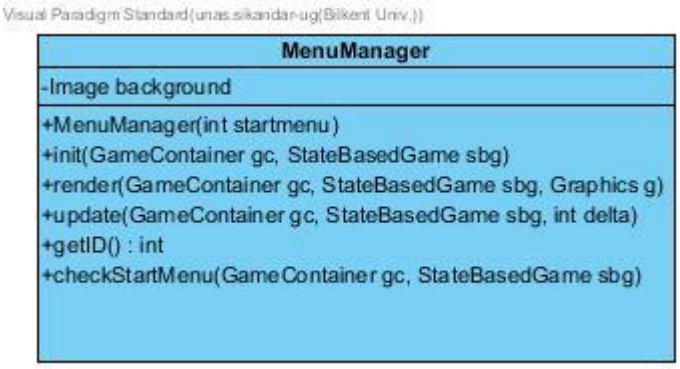
This method will be called to load the desired map into the map object. Method returns true if successful and false if it fails to load the map.

+ **boolean : canMove(Moving ch, float delta, Direction dir)**

This method determines whether the Moving object (ch) can move in the Direction specified.

4.5.3 MenuManager Class

Extends BasicGameState class provided in the slick library



Attributes:

- **Image : background**

This image object will be displayed as the background which other menu elements will be drawn on.

Operations:

+ **init(GameContainer gc, StateBasedGame sbg)**

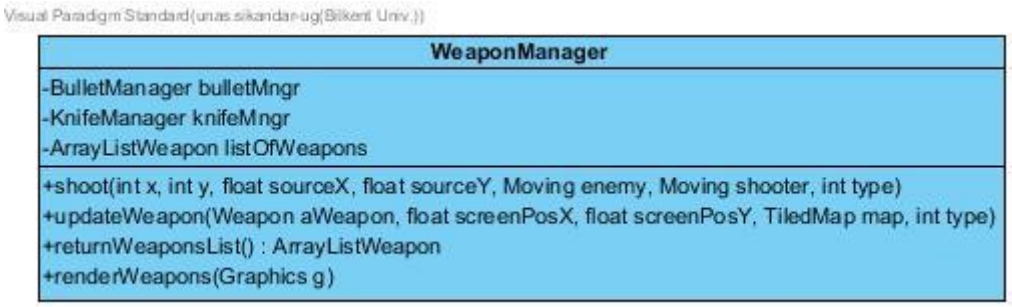
This method is provided in the parent class and is overridden here. This method will initialize the view of the menu. For example the button positions and background image will be initialized here.

+ **void render (gc : GameContainer, sbg : StateBasedGame, g : graphics)**

This method will be in charge of drawing all of the screen elements. Overridden from the parent class.

+ void update (gc : GameContainer, sbg : StateBasedGame, delta : int)
This method will be called each delta time we have specified. It calls a helper method called checkStartMenu() to determine if menu has changed or not.
+ void checkStartMenu(GameContainer gc, StateBasedGame sbg)
This method checks if any of the virtual buttons are clicked on main-menu. If mouse has been pressed in between valid coordinates we have specified this method will change the state of the game by manipulating “StateBasedGame sbg”. When the state of the game changed a new screen will be displayed with the help of “Screen Container” class.

4.5.4 WeaponManager Class



Attributes:

- BulletManager : bulletMngr
Instance of BulletManager used to update bullets separately.

- KnifeManager : knifeMngr
Instance of KnifeManager used to update knives separately.

- ArrayList<Weapon> : listOfWeapons
A list containing all the weapon objects that are currently on the game.

Operations:

+ void shoot(int x, int y, float sourceX, float sourceY, Moving shooter, Moving enemy,int type)
This method is used to launch a projectile(Weapon object) towards the target (x,y). The projectile can either be a bullet or a knife. This method creates a new Weapon object and adds it to “listOfWeapons”

+ void updateWeapon(Weapon aWeapon, float screenPosX, float screenPosY, TiledMap map, int type)
This method is used in combination with the shoot method to update the weapons on the map. This method will check the weapon collisions while updating and delete the weapon from “listOfWeapons” if they collide with an object.

+ void renderWeapons(Graphics g)

This method is used to render all weapons “listOfWeapons” arraylist has on the screen by taking Graphics component as a parameter. This way all of the weapons will be drawn on the component when this method is invoked.

4.5.5 KnifeManager Class

Attributes:

- **ArrayList<Knife> : listOfKnifes**

Operations:

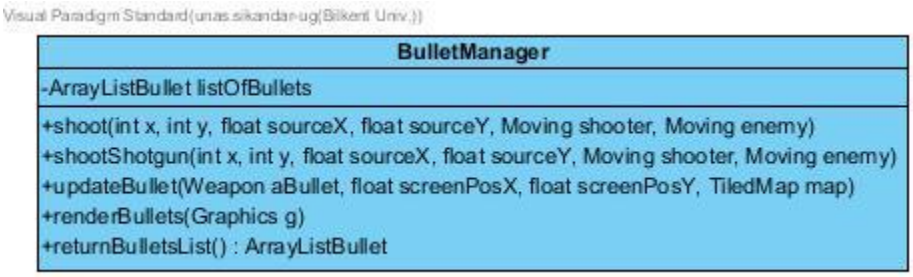
+ throwKnife(int X, int Y, float sourceX, sourceY, Moving shooter, Moving enemy)
This method is invoked when the input manager determines that the Moving character (shooter) wants to throw knife at the target Moving object (enemy). The function is given the source and the target coordinates of the projectile.

+ updateKnife(Knife aKnife, float screenPosX, float screenPosY, TiledMap map)
This method is used to update the position of the single Knife object on the map. Collision check is overwritten in this method from parent method because the type of the weapon is different.(It is calculated with different parameters.)

+ renderKnifes(Graphics g)
This method is used to render all the knives currently in the listOfKnifes on the screen.

+ returnKnifesList()
This method returns the current knifes list.

4.5.6 BulletManager Class



Attributes:

- **ArrayList<Bullet> : listOfBullets**

Operations:

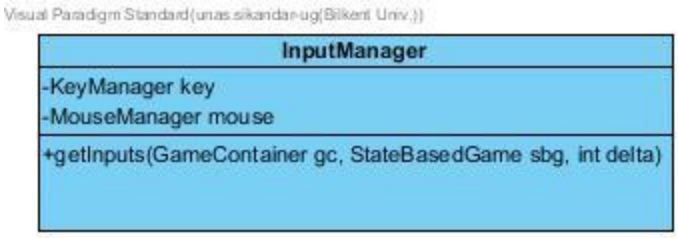
+ shoot(int X, int Y, float sourceX, sourceY, Moving shooter, Moving enemy)
This method is invoked when the input manager determines that the Moving object (shooter) wants to shoot bullets at the target Moving object (enemy). The function is given the source and the target coordinates of the projectile.

+ updateBullet(Bullet aBullet, float screenPosX, float screenPosY, TiledMap map)
This method is used to update the position of the single Bullet object on the map.Collision check is overwritten in this method from parent method because the type of the weapon is different.(It is calculated with different parameters.)

+ renderBullets(Graphics g)
This method is used to render all the bullets currently in the listOfBullets on the screen.

+ returnBulletList()
This method returns the current Bullets list.

4.5.7 InputManager Class



Attributes:

- **KeyManager : key**

Instance of KeyManager used in input manager to extract inputs from both mouse and keyboard at once.

- **MouseManager : mouse**

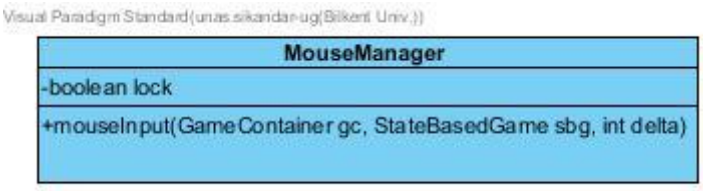
Instance of MouseManager used in input manager to extract inputs from both mouse and keyboard at once.

Operations:

+ **getInputs(GameContainer gc, StateBasedGame sbg, int delta)**

This method is used to extract inputs from the game container. The input can be either mouse input or keyboard input.

4.5.8 MouseManager Class



Attributes:

Operations:

+ **mouseInput(GameContainer gc, StateBasedGame sbg, int delta)**

This method creates an action if the Mouse Input is a valid request. Basically if mouse input is done while the game is being played it shoots the weapon of the character at mouse direction.

4.5.9 KeyManager Class

Attributes:

- **Input : input**

This variable is used to store the current key input of the user

Operations:

+ keyInput(GameContainer gc, StateBasedGame sbg, int delta)
This method invokes the actions for all valid key inputs while the game is being played. Such as moving left, moving right, shooting with enemy...

4.5.10 SettingsController Class

Attributes:

- SettingsData : data
A SettingsData structure that holds the data of different fields of the settings menu.

Operations:

+ changeVolume(float delta)
Used in the settings menu to change volume level

+ changeBrightness(int delta)
Used in the settings menu to adjust screen brightness

+ goFullScreen(boolean b)
Used to toggle fullscreen

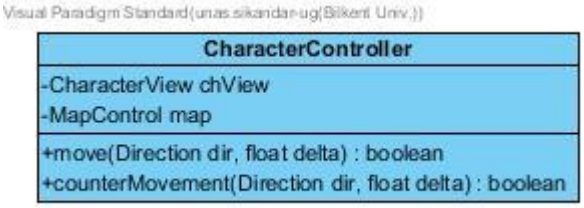
4.5.11 Interface MovingController

Operations:
+ move()
Inherited by the EnemyController and CharacterController to move the character according to the inputs given.

+ counterMovement()

Inherited by the EnemyController and CharacterController to counter move according to the movement of the opposite character i.e. the enemy should also move according to the main character's manipulation of the map

4.5.12 CharacterController Class



Attributes:

- **CharacterView : characterView**

Holds the View of the character to display the movement decisions.When a movement is done; the properties of this object will be manipulated by this class.

- **MapControl : map**

Instance of the MapControl object used to determine whether a particular movement command is possible or not.

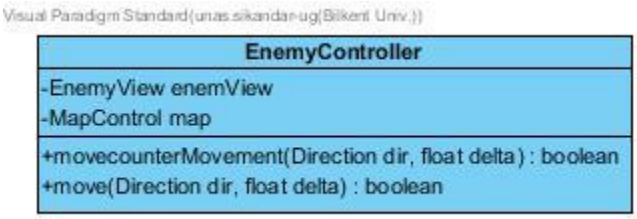
Operations:

+ **boolean move(Common.Direction dir, float delta)**

Overwritten from Movement Controller to perform a specific move command for character. Uses Map Control to determine if the move is possible or not. If it is possible this method sets the properties of character view according to the direction and movement. When this returns true; enemy controllers counterMovement() method is called to create the effect of screen focused to character.

4.5.13 EnemyController Class

Implements the MovingController interface.



Attributes:

- **EnemyView : enemyView**

Holds the View of the Enemy to make movement decisions. EnemyView will provide data such as the locations and the directions of the character

- **MapControl : map**

Instance of the MapControl object used to determine whether a particular movement command is legal or not.

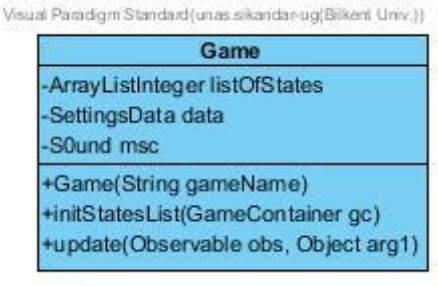
Operations:

+ boolean move(Common.Direction dir, float delta)
Overwritten from Movement Controller to perform a specific move command for enemy. Uses Map Control to determine if the move is possible or not. If it is possible this method sets the properties of enemy view according to the direction and movement.

+ boolean counterMovement(Common.Direction dir, float delta)
This method is used to make the effect of locking to character by moving the enemy in the opposite direction. What we actually do is push every object in opposite direction when character is moving on a specified direction.

4.5.14 Game Class

This class extends StateBasedGame from slick2d library.



Attributes:

- ArrayList<int> : listOfStates
Stores the list of all the available states of the game e.g. Pause Menu, Main Menu etc.

- Sound msc
Stores the sound being played currently

Operations:

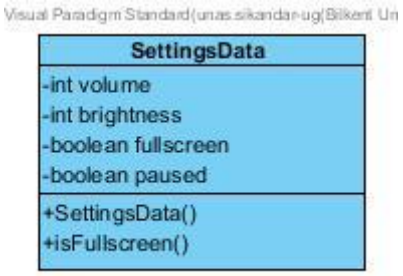
+ initStatesList(GameContainer gc)
Initializes all the game states by calling their init() methods. Game states are accessed from this class by calling getState() method from parent class “StateBasedGame” from slick2d.

+ update(Observable obs, Object arg1)
This method is called whenever the observed object is changed

+ Game(String gameName)

This constructor creates all the states of the game and adds them to StateBasedGame sbg so other classes can access them or check which state the game is currently on.

4.5.15 SettingsData Class



Attributes:

- **int : volume**

Stores the current level of the volume

- **int : brightness**

Stores the current level of the screen brightness

- **boolean : fullsceen**

Stores the boolean value of the full screen (true if fullscreen, false if windowed).

- **boolean : paused**

Stores the boolean value of the pause state (true if game is paused, false otherwise).

Operations:

+ **isFullscreen()**

Returns the value of the paused variable.

4.5.16 Sound Class

Attributes:

- **Music : msc**

Contains the music theme of the game

- **Boolean : on**

Used to set sound on/off

- **float : volume**

Stores the current volume level

Operations:

+ playTitleMusic()

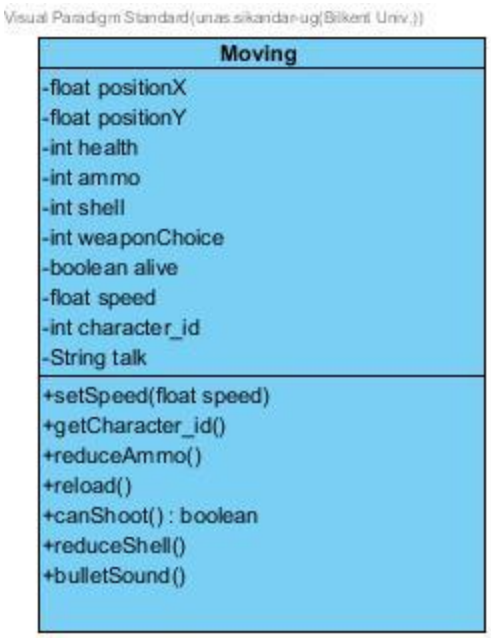
This method is used to start the title music track in the game

+ changeVolume(float change)

This method is used to change the volume level by adding the float argument specified to it.

4.5.17 Moving Class

This class is inherited by the classes Character and Enemy. These are the main players in the game. The Moving class inherits Java's Observable class which allows us to notify view objects that are concerned with the Moving objects about their changes.



Attributes:

- **float : positionX**

Stores the X coordinate of Character/Enemy on the map

- **float : positionY**

Stores the Y coordinate of Character/Enemy on the map

- **int health**

Stores the current health points of the Character/Enemy

- **int ammo**

Stores the amount of ammo that Character/Enemy has currently

- **int weaponChoice**

Used to determine which weapon is currently active

- **boolean alive**

It is used to determine whether the Character/Enemy is alive/dead.

- **float speed**

Stores the current speed of the Character/Enemy

- **int character_id**

Stores the unique id that every Moving object is given at creation

- **String talk**

Stores the String value of some of the dialogues that the character/enemy can speak. These are stored in the resources.

Operations:

+ **setSpeed(float speed)**

This method is used to set the speed of the moving object (Character/Enemy)

+ **getCharacter_id()**

Returns the ID of the moving object

+ **reduceAmmo()**

Used to reduce the amount of ammo when the Character/Enemy is shooting bullets.

+ **reduceShell()**

Used to reduce the amount of shells when the Character/Enemy is shooting shotgun shells.

+ **reload()**

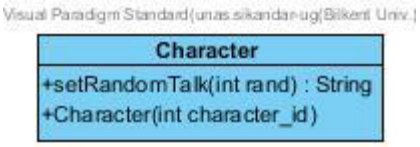
Called when the character/enemy reloads the currently active weapon. Reload is only performed if the currently chosen weapon has ammunition.

+ **bulletSound()**

This method is called while shooting so that the shooting sound is triggered.

4.5.18 Character Class

Extends the moving class



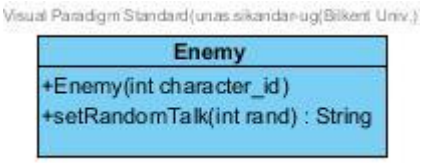
Operations:

+ setRandomTalk(int rand)

Takes a random integer as a parameter and uses it to select a random dialogue for the character.

4.5.19 Enemy Class

Extends the moving class

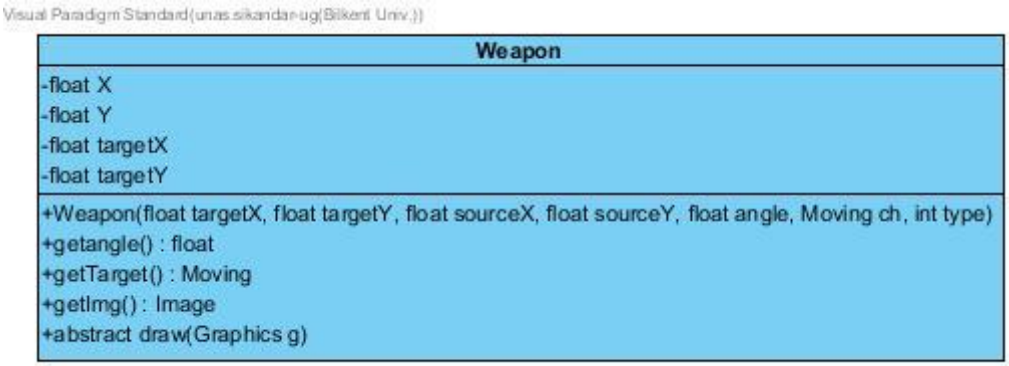


Operations:

+ setRandomTalk(int rand)

Takes a random integer as a parameter and uses it to select a random dialogue for the enemy.

4.5.20 Weapon Class



Attributes:

- **float : X**

Holds the current X coordinate of the projectile.

- **float : Y**

Holds the current Y coordinate of the projectile.

- **float : targetX**

Holds the X coordinate of the projectile's destination.

- **float : targetY**

Holds the Y coordinate of the projectile's destination.

Operations:

+ **getAngle()**

This method is used to get the angle between the projectile and the horizontal axis.

+ **getTarget()**

This method returns the target (Moving object) that is in the projectile path.

+ **getImage()**

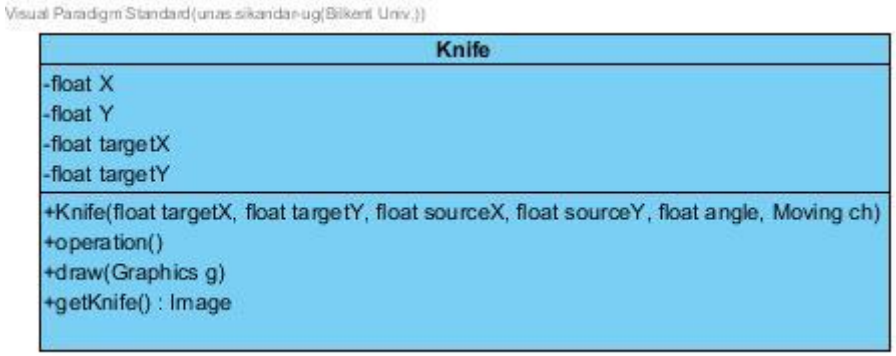
This method gets the appropriate image of the projectile from the resources and returns it.

+ **abstract draw(graphics g)**

This method is provided here so that it can be overwritten by the classes that inherit Weapon, namely Bullet and Knife.

4.5.21 Knife Class

Extends the Weapon class



Attributes:

- **float : X**

Holds the current X coordinate of the Knife.

- **float : Y**

Holds the current Y coordinate of the Knife.

- **float : targetX**

Holds the X coordinate of the Knife's destination.

- **float : targetY**

Holds the Y coordinate of the Knife's destination.

Operations:

+ **draw(Graphics g)**

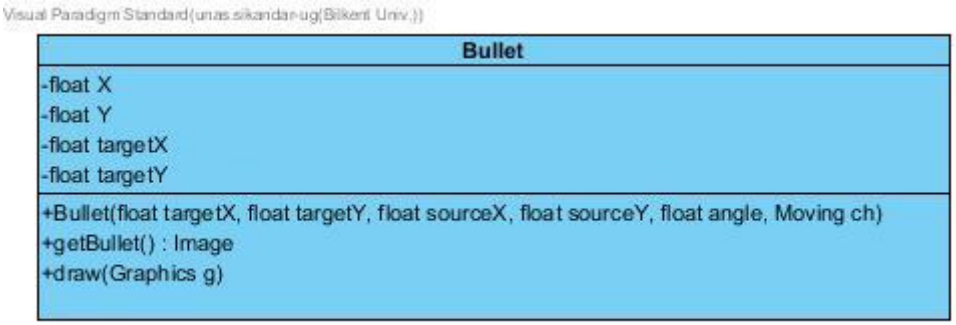
This method is used to render the Knife

+ **getKnife()**

This method returns the Image object holding the image of the Knife projectile.

4.5.22 Class Bullet

Extends the Weapon class



Attributes:

- **float : X**

Holds the current X coordinate of the Bullet.

- **float : Y**

Holds the current Y coordinate of the Bullet.

- **float : targetX**

Holds the X coordinate of the Bullet's destination.

- **float : targetY**

Holds the Y coordinate of the Bullet's destination.

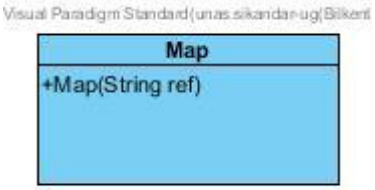
Operations:

+ **draw(Graphics g)**

This method is used to render the Bullet

+ getBullet()
This method returns the Image object holding the image of the Bullet projectile.

4.5.23 Map Class
This class extends TiledMap to access it's methods and functionalities. TiledMap is a exterior library we used to create our maps. This class allows us to access TiledMap without having to hold a TiledMap model object.



Attributes:

+ Map(String ref)
This constructor takes the location of the map from it's parent.

4.5.24 ScreenContainer Class

Attributes:

- ArrayList<BasicGameState> : screens
This attribute holds the states of the game to determine which screen is going to be displayed when the game is on a specific "BasicGameState".

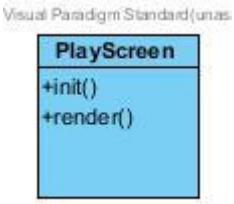
- Graphics : g
This attribute holds the main display graphics component of our game. It is passed to other view objects or controller objects if necessary. This way a single display component can be used to represent our visual content in the game.

Operations:

+ BasicGameState : getCurrent()
This method tells the class which state is the game on at the time of this function call. This is used while rendering the game's visual content.

+ **render(GameContainer container, Graphics g, StateBasedGame sbg)**
This render method acts like a general render. It calls the render function of a BasicGameState view by checking the current state. This way any unnecessary rendering are avoided.

4.5.25 PlayScreen Class

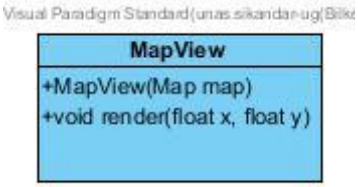


Operations:

+ **init()**
This method is used to initialize the MapControl and the MapView.

+ **render()**
This method is used to render the play screen which consists of the MapView and the MovingView drawn on top of MapView.

4.5.26 MapView Class

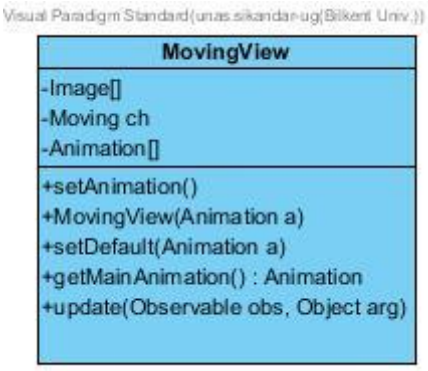


Attributes:
- **Map : map**
This attribute holds the map of the current Game. This map object is used as a connection between “TiledMap” and our game.

Operations:

+ **render(float X, float Y)**
This method is used to render the map image into the PlayScreen on the specified X,Y coordinates given in the parameters.

4.5.27 MovingView Class



Attributes

- **Image[ ] : images**

Holds an array of images that are required to create the animations of character/enemy objects.

- **Animation[ ] : animations**

Holds the animations related to the character/enemy. These are used when the character/enemy is moving on a specific direction.

- **Moving : ch**

The character or enemy moving object.

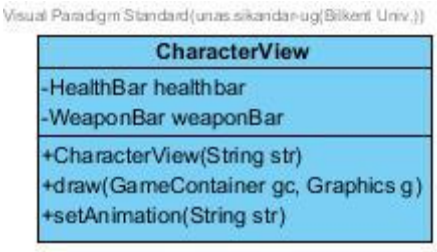
Operations:

+ **setAnimation()**

Chooses the appropriate animation object for the character from the list of animations and assigns it for display purposes.

4.5.28 CharacterView Class

Extends the MovingView class.



Attributes:

- **WeaponBar : weaponBar**

Contains the WeaponBar view object of the character

- **HealthBar : healthBar**

Contains the HealthBar view object of the character

Operations:

+ **draw(GameContainer gc, Graphics g)**

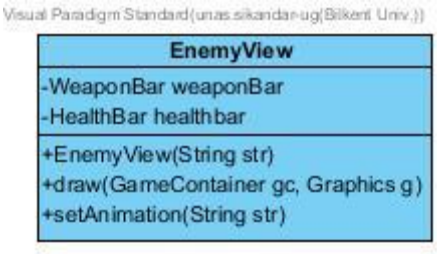
Used to render the Character in the play screen. This method also calls the render methods of WeaponBar and HealthBar view objects on this class.

+ **setAnimation(String str)**

Used to change the animation of the character according to the name given as a String parameter.

4.5.29 EnemyView Class

Extends the MovingView class.



Attributes:

- **WeaponBar : weaponBar**

Contains the WeaponBar view object of the enemy

- **HealthBar : healthBar**

Contains the HealthBar view object of the enemy

Operations:

+ **draw(GameContainer gc, Graphics g)**

Used to render the Enemy on the play screen. This method also calls the render methods of WeaponBar and HealthBar view objects on this class.

+ **setAnimation(String str)**

Used to change the animation of the Enemy according to the name given as a String parameter.

4.5.30 PauseMenu Class

Attributes:

- **Image : background**

Stores the background image of the menu which is drawn first before every other elements.

Operations:

+ **init (GameContainer : gc, StateBasedGame : sbg)**

Initializes the view, loads the view into the screen.

+ **render(GameContainer : gc, StateBasedGame : sbg, Graphics g)**

This method is used to draw all the elements on the screen like buttons and text.

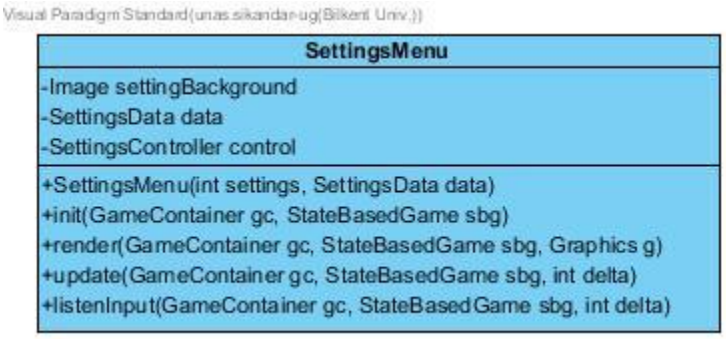
+ **update (GameContainer : gc, StateBasedGame : sbg, int delta)**

This method is called periodically to update the data that is being rendered.

+ **getID()**

Returns the ID of this state.

4.5.31 SettingsMenu Class



Attributes:

- **Image : background**

Stores the background image of the menu which is drawn first before every other elements.

- **SettingsData : data**

Instance of the SettingsData class is used to determine what values of the settings should be rendered.

- **SettingsController : control**

Instance of SettingsController used to determine how a settings data can be updated.

Operations:

+ **init (GameContainer : gc, StateBasedGame : sbg)**

Initializes the view, loads the view into the screen.

+ **render(GameContainer : gc, StateBasedGame : sbg, Graphics g)**

This method is used to draw all the elements on the screen like buttons and text.

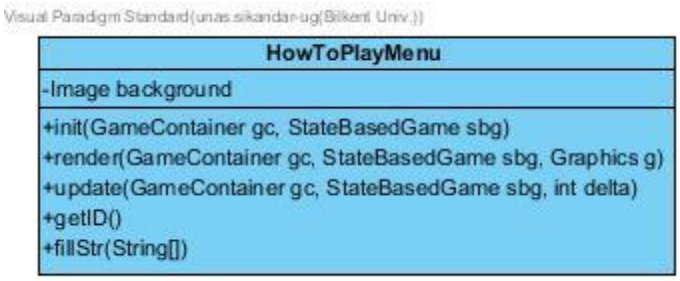
+ **update (GameContainer : gc, StateBasedGame : sbg, int delta)**

This method is called periodically to update the data that is being rendered.

+ **listenInput(GameContainer gc, StateBasedGame sbg, int delta)**

This method is used to convert user inputs to valid actions that manipulates “SettingsData” model object.

4.5.32 Class HowToPlayMenu



Attributes:

- **Image : background**

Stores the background image of the menu which is drawn first before every other elements.

Operations:

+ **init (GameContainer : gc, StateBasedGame : sbg)**

Initializes the view, loads the view into the screen.

+ **render(GameContainer : gc, StateBasedGame : sbg, Graphics g)**

This method is used to draw all the elements on the screen like buttons and text.

+ update (GameContainer : gc, StateBasedGame : sbg, int delta)
This method is called periodically to update the data that is being rendered.

+ getID()
Returns the ID of this state.

4.5.33 CreditsMenu Class

Attributes:

- Image : background
Stores the background image of the menu which is drawn first before every other elements.

Operations:

+ init (GameContainer : gc, StateBasedGame : sbg)
Initializes the view, loads the view into the screen.

+ render(GameContainer : gc, StateBasedGame : sbg, Graphics g)
This method is used to draw all the elements on the screen like buttons and text.

+ update (GameContainer : gc, StateBasedGame : sbg, int delta)
This method is called periodically to update the data that is being rendered.

+ getID()
Returns the ID of this state.

4.5.34 GameOverView Class

Attributes:

- **Image : background**

Stores the background image of the menu which is drawn first before every other elements.

Operations:

+ **init (GameContainer : gc, StateBasedGame : sbg)**

Initializes the view, loads the view into the screen.

+ **render(GameContainer : gc, StateBasedGame : sbg, Graphics g)**

This method is used to draw all the elements on the screen like buttons and text.

+ **update (GameContainer : gc, StateBasedGame : sbg, int delta)**

This method is called periodically to update the data that is being rendered.

+ **getID()**

Returns the ID of this state.

5. Improvement Summary

We have made changes in design since first iteration. First of all, starting the implementation made a lot of classes clearer in our minds so we have added and removed so classes or some methods from our design. It was much easier to understand what was really necessary and what was not.
Changes are as follows:

- The **GameModel subsystem** was changed significantly. In the first iteration the **GameModel subsystem** consisted of a GameObject class which acted as a parent to all Model classes. The subsystem contained too many dependencies as each class had parent and grandparent classes. The subsystem was changed and these dependencies were diminished to reduce coupling within the subsystem. (Non-Stationary, Non-Passable, Stationary removed to reduce number of child classes which were dependent on them. Moving, Enemy classes were added instead. Bush, Grenade, Water classes are removed (they were unnecessary for our design).

- In the second iteration we decided to apply the **Observer Design Pattern.** For this we added the **Observable class** to the **GameModel subsystem** and the **Observer Interface** to the **GameView Subsystem.** This improvement allows the View classes to update their view using the **Observer interface** without containing or accessing the instance of the entity object.

- **GameControl subsystem** now contains 3 further subsystems namely **InputControl, PersonControl** and **WeaponControl.** Similarly **GameView** and **GameModel subsystems** were further divided into respective subsystems to increase cohesion.

- The option to change keyboard configurations was added. So now the user can define their own controller settings which they are comfortable with. To do so **SettingsData** class was added to the **GameModel subsystem**. So that the system can keep track of the keys assigned to the controls.

- The game was scaled to 2 players from a single player game. To achieve this an **Enemy class** was created to represent the second player.

- Implementation of the map design was changed. Before the map was stored in the system as an image file. Now the map is stored as a text document. This change allows us to handle collisions, character and map updates better because of the String representation of data which can be compared easily.

6. References & Glossary

MVC: Model View Controller

System: our game “ War of Domination “

Slick2d: http://slick.ninjacave.com/

Façade: design pattern https://www.tutorialspoint.com/design_pattern/facade_pattern.htm

Bruegge, B., & Dutoit, A. H. (2014). Object-oriented software engineering: using UML, patterns, and Java. Harlow, Essex: Pearson.

https://www.visual-paradigm.com/