CS 319

Object-Oriented Software Engineering

Design Report - Iteration 2

World War 3

Group 1-F

Nurefşan Müsevitoğlu

Hüseyin Taşkesen

Halil İbrahim Çavdar

Doğacan Kaynak

1 Introduction

1.1 Purpose of the System

Our purpose in creating World War 3 is offering the player a quality experience of strategy gaming. By creating a simple user interface and making the gameplay very similar to those will(assume to) be in real life of space ages, we give the user an easy grasp of basic concepts so that they can start playing almost immediately. As the gamer keeps playing, they will discover the depths of lying strategies that one can take which is what makes World War 3 special.

The game will be implemented using Java and is designed for PC. In the following sections of this report, we will provide the goals of the design, tradeoffs and different diagrams about the System Decomposition of the game. Moreover, the class diagram and explanation for each class will also be described in the following sections.

1.2 Design Goals

In this section, we have determined our design goals. These design goals are derived from the non-functional requirements, which is stated in our analysis report. In the process of designing our system, we will be focusing on the below mentioned criteria.

1.2.1 Performance Criteria

Efficiency: We think efficiency is extremely crucial when creating an interesting game. That's the reason we made our game as efficient as possible in a Java system because before even starting detailed design of the game, among all tradeoffs efficiency was the number one priority. In several places where we could make our system more memory efficient instead we sacrificed our memory and reduces our number of iterations to maximize performance. We decreased the weight on our game manager as much as possible to squeeze out every possible bit of performance there. For instance, performance of our game will be 60 fps, since our planned methods never use an algorithm which goes exponential.

Response Time: Since our game is so simple, it does not include any database connection or web services which reduces its response time.

Memory: We use the memory only for two things, firstly, for keeping the session name which is the name that player type in before starting the game and secondly, for keeping the high score. It costs very little, approximately 10 kb, space on hard drive and almost no space in RAM is enough for running our game.

1.2.2 Dependability Criteria

Reliability: Our game will be very consistent in its boundary conditions, it will be almost impossible to make our system crash since we will handle almost every exception possible by debugging and testing our game in a lot of different use cases.

Boundary conditions will be properly defined and obeyed with discipline in order to not give the user any chance to crash the game.

Robustness: Java makes it easier for us to deal with this feature since the language can be run in all OS's. Still, while implementing our game, we will also consider the cases when the player performs unwanted actions, so that the game continues and not crash. No kind of input that user can enter will be able to crash our system. Having a robust system is much easier in simple applications.

Fault Tolerance: Our game tolerates erroneous inputs and keeps working correctly under them. We will test our program in as many aspects as we can to increase the fault tolerance of the system.

1.2.3 Maintenance Criteria

Extensibility: Our object oriented software design approach was to divide each and every step that we could divide reasonably. This design approach helped us to build a system similar to a castle of lego bricks. With the detailed documentation given and very small coupling of the system, one could easily take one part off or add another one without having too many difficulties. For example, with creating a new class that will extend our HumanSide class, we can add a new and different kind of human without the modifying the rest of the classes. This can also be done for robots or any other game elements. Therefore, with the help of a few modifications our game will be easily extensible.

Usability: Our game is extremely straightforward to open and start playing. There are some learning curves that the player might have to pass but this does not mean usability of our system is low, this is mainly caused of the numerous ways of strategies that can be pulled off in the game and one has to think and has to weigh their risks sometimes before making their moves in order to be successful in the game. We tried to make this as easy as possible by providing in-game help to user and also a very detailed documentation explaining everything.

Adaptability: The reason we chose Java except that it was preferred by instructors was that it has the ByteCode intermediate step which is used to make the application cross-platform enabled. This makes our program able to run on all JRE installed platforms, in the cost of sacrificing some important amount of efficiency with not creating our game in C/C++.

2 Software architecture

In this section, we describe the subsystem decomposition of our system. The subsystem decomposition has been designed to address our design goals. Furthermore on this section, we outline the hardware/software mapping, persistent data management, access control and security, and finally, boundary conditions.

2.1 Subsystem decomposition

In this section, we will decompose our system into subsystems. We chose to follow the MVC (Model-View-Controller) design pattern, since it best suited our system and therefore, we come up with three packages. Each of these packages contains the classes about its role in an MVC pattern. To name these packages and roles:

- Game Objects package is the Model part of the MVC pattern.
- User Interface package is the View part of the MVC pattern.
- Game Management package is the Controller part of the MVC pattern.

During this decomposition, our aim is to have a complete system that works in harmony and every subsystem has its own clear-cut roles. This will make us work on different parts of the project without the loss of coherence. Besides, we can easily make any updates or required modifications on the system without changing the other parts of it.

To give some detail about the contents of these packages:

- Model package contains GameElement and its subclasses and the User class.

- View package contains the menu view classes such as UIManager, MainMenu and Settings and the MapManager class which is the view part of the actual game.
- Controller package contains GameManager class as their name suggest.

About the connections between these packages as can be seen in Figure 1, User Interface reports the user's actions to Game Management, Game Management package updates both the User Interface and Game Objects after every action and model classes form a basis for the User Interface. The connections between packages are given in Figure-1 in less detail. More detailed version of connections can be seen at Figure-2. The class diagram of our game can be seen at Figure-3. Detailed information about packages are also available. Model package (Game Objects) can be seen at Figure-4. View package (User Interface) can be seen at Figure-5. Controller package (Game Management) can be seen at Figure-6.

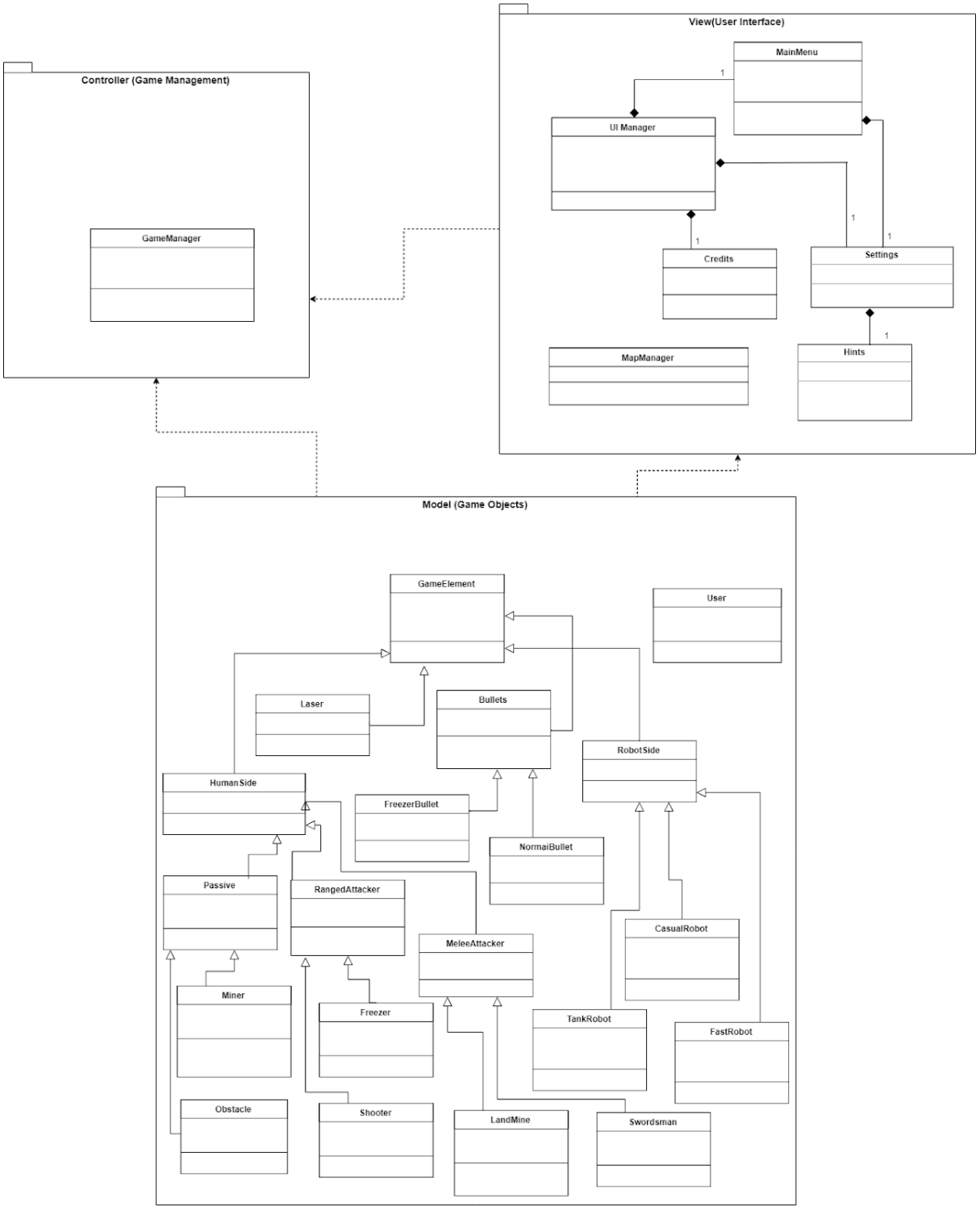


Figure-1 Connections

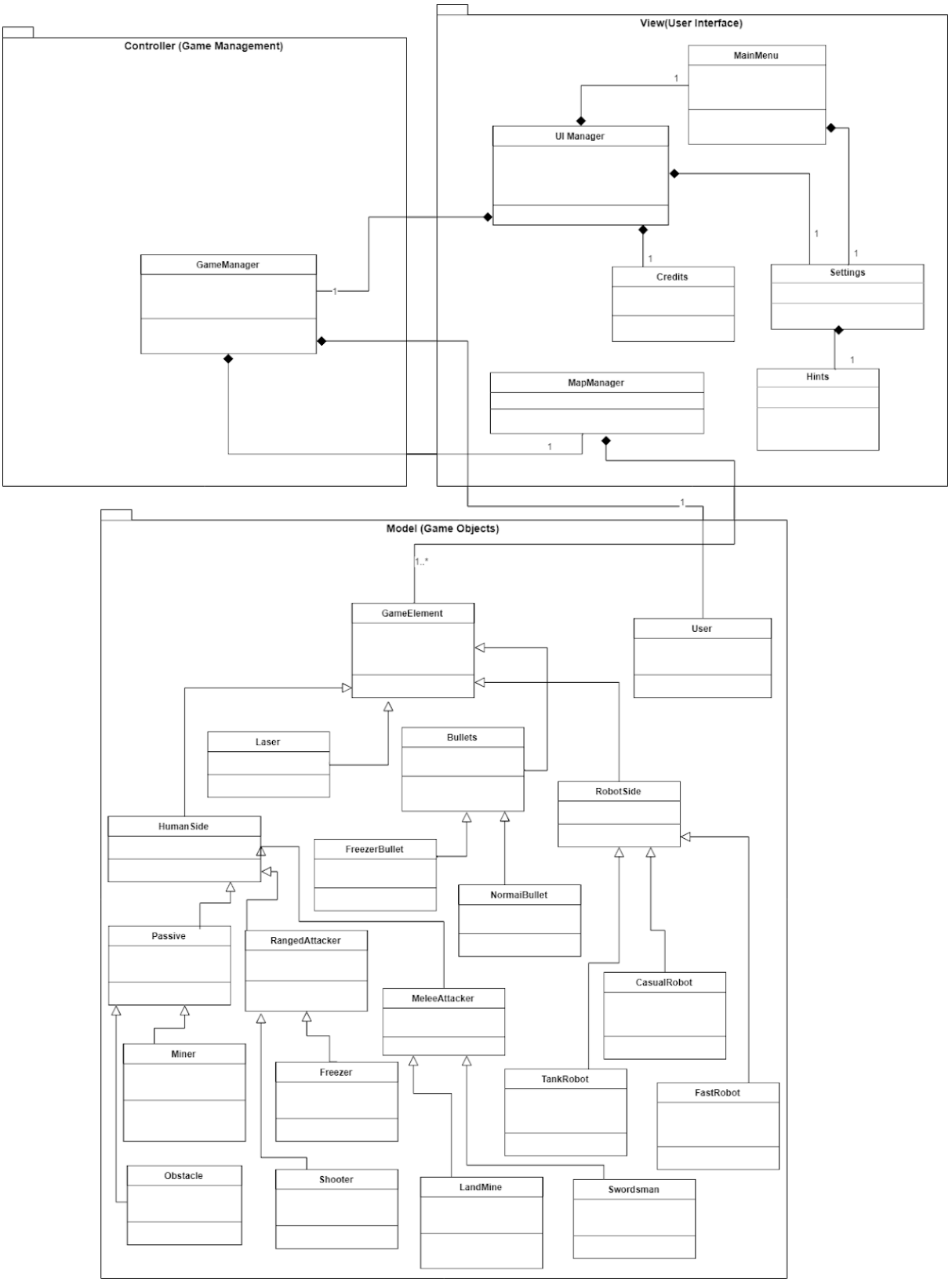


Figure-2 Connections in detail

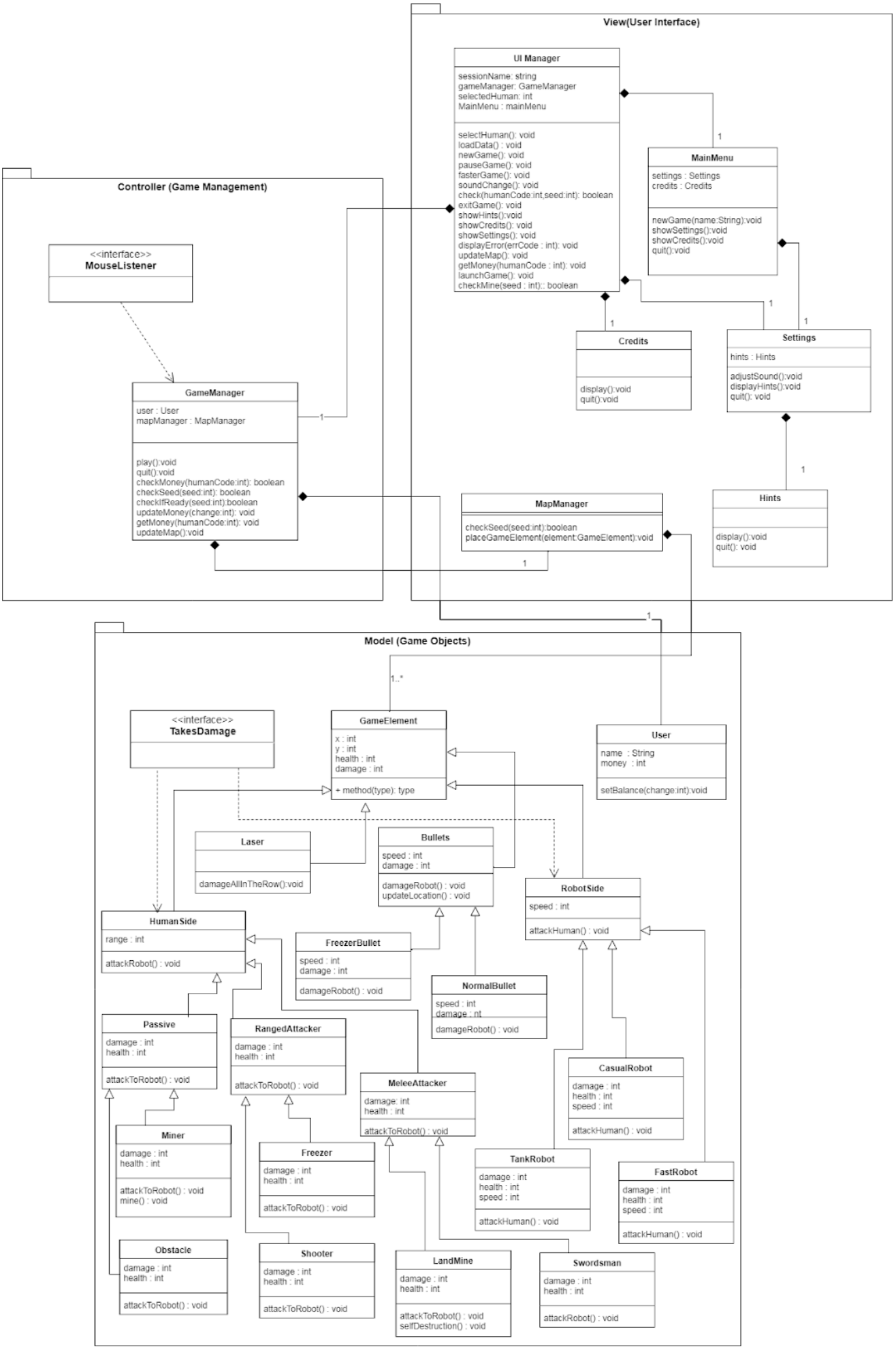

View(User Interface)
UI Manager
sessionName: string
gameManager: GameManager
selectedHuman: int
MainMenu : mainMenu
selectHuman(): void
loadData() : void
newGame(): void
pauseGame(): void
fasterGame(): void
soundChange(): void
check(humanCode:int,seed:int): boolean
exitGame(): void
showHints():void
showCredits(): void
showSettings(): void
displayError(errCode : int): void
updateMap(): void
getMoney(humanCode : int): void
launchGame(): void
checkMine(seed : int):: boolean
MainMenu
settings : Settings
credits : Credits
newGame(name:String):void
showSettings():void
showCredits():void
quit():void
Credits
display():void
quit():void
Settings
hints : Hints
adjustSound():void
displayHints():void
quit(): void
Hints
display():void
quit(): void
Controller (Game Management)
<<interface>>
MouseListener
GameManager
user : User
mapManager : MapManager
play():void
quit():void
checkMoney(humanCode:int): boolean
checkSeed(seed:int): boolean
checkIfReady(seed:int):boolean
updateMoney(change:int): void
getMoney(humanCode:int): void
updateMap():void
MapManager
checkSeed(seed:int):boolean
placeGameElement(element:GameElement):void
Model (Game Objects)
<<interface>>
TakesDamage
GameElement
x : int
y : int
health : int
damage : int
+ method(type): type
User
name : String
money : int
setBalance(change:int):void
Laser
damageAllInTheRow():void
Bullets
speed : int
damage : int
damageRobot() : void
updateLocation() : void
RobotSide
speed : int
attackHuman() : void
HumanSide
range : int
attackRobot() : void
FreezerBullet
speed : int
damage : int
damageRobot() : void
NormalBullet
speed : int
damage : nt
damageRobot() : void
Passive
damage : int
health : int
attackToRobot() : void
RangedAttacker
damage : int
health : int
attackToRobot() : void
MeleeAttacker
damage: int
health : int
attackToRobot() : void
CasualRobot
damage : int
health : int
speed : int
attackHuman() : void
Miner
damage : int
health : int
attackToRobot() : void
mine() : void
Freezer
damage : int
health : int
attackToRobot() : void
TankRobot
damage : int
health : int
speed : int
attackHuman() : void
FastRobot
damage : int
health : int
speed : int
attackHuman() : void
Obstacle
damage : int
health : int
attackToRobot() : void
Shooter
damage : int
health : int
attackToRobot() : void
LandMine
damage : int
health : int
attackToRobot() : void
selfDestruction() : void
Swordsman
damage : int
health : int
attackRobot() : void
1
1..*

Figure 3: The class diagram

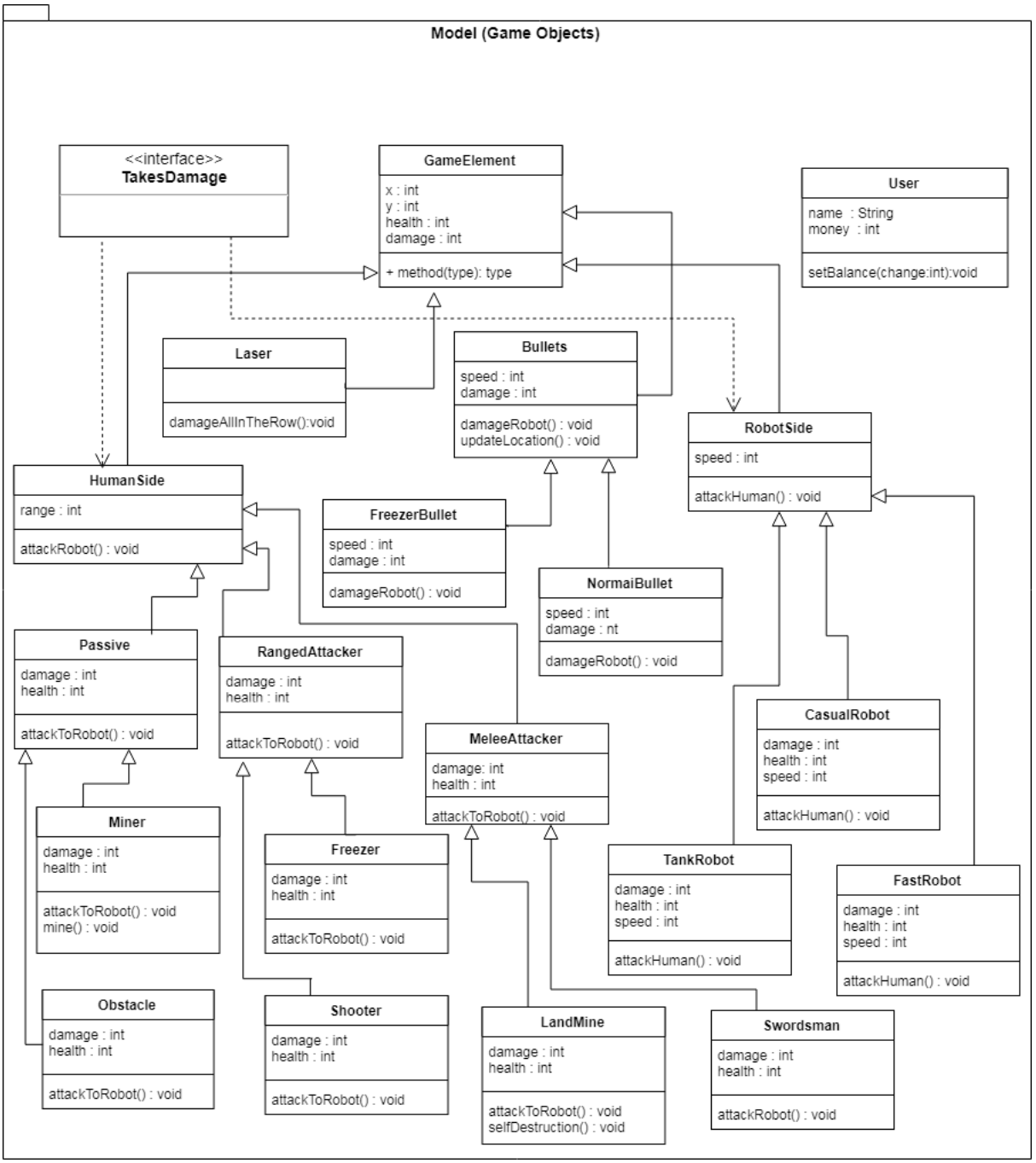


Figure-4 Game Objects

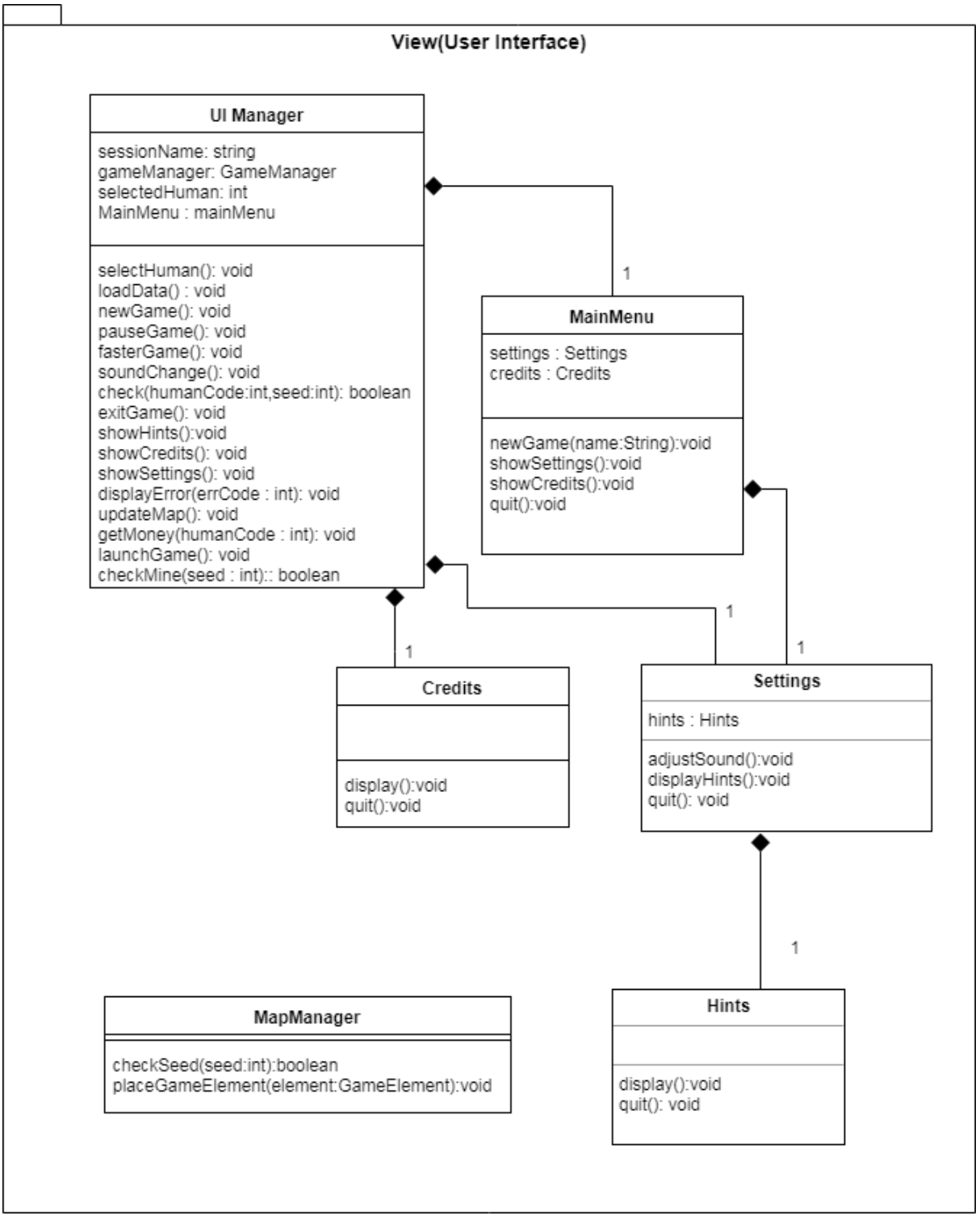


Figure-5 User Interface

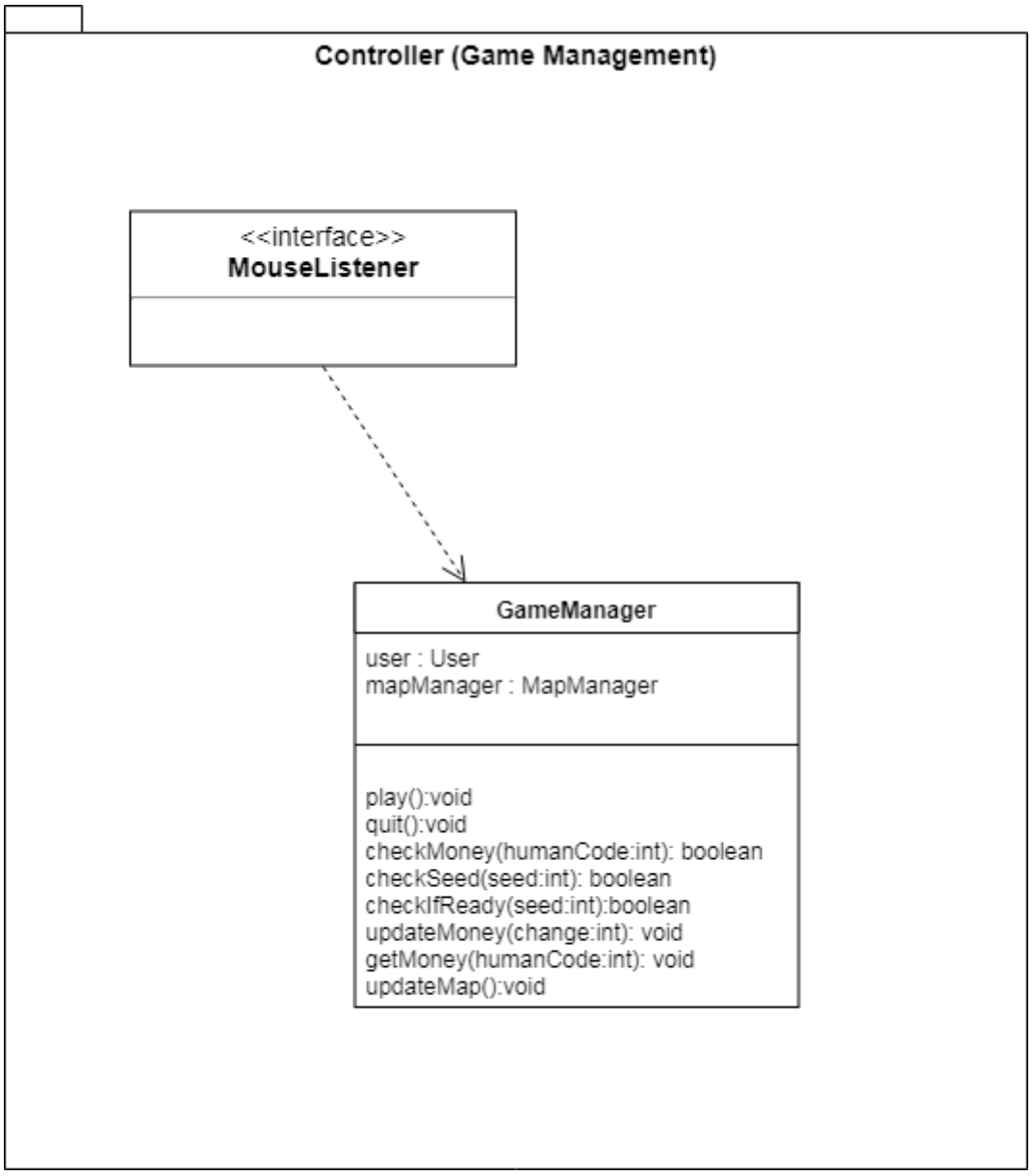


Figure-6 Game Management

2.2 Hardware/Software Mapping

Since our game is written in Java it will require Java Runtime Environment support, which can be found across many operating systems such as Windows, Linux, MacOS or Polaris. Additionally, latest version of JDK and a Java compiler will be needed. Considering hardware requirements one needs to have a keyboard and a mouse, but mainly mouse pointer will be used throughout the game. Game requires minimal amount of today's hardware in order to run smoothly, even if you do not have a considerably reasonable hardware you might enjoy the game. It would be nice to have a good CPU though since we are using Java Slick2D graphics and it would give the user a much more enjoyable experience in gaming. The most important reason that led us to choose Slick2D as a library in our game: ease of implementation. There are couple of pre-written methods to make coder's life easier. We will benefit a lot from Slick2D when creating our GUI and our dynamic motion which should be smooth in our game. There is no requisite for network connection since for stroring the data we will use filing system. Due to portability of Java and basi needs of hardware, standart computers will be enough to run *"The World War 3"*.

2.3 Persistent Data Management

Game data such as player's name, background images, sounds, background music, score of the player, and high score will be stored in the user's local hard drive. Our game does not require any database system since the data that is used in the game need to be accessed in real time. Thus, all the necessary files and data will load onto the memory. Data

that will load onto the memory will take approximately 10 kb memory which is significantly little.

2.4 Access Control and Security

Since the game we are creating is single-player and must be downloaded to individual computers, we did not see any necessity to implement an authentication system in our game. But security of the program is controlled in a different way rather than authentication, we divided our logical elements to many different smaller components in order make it easy to track and debug, this will increase readability and also the robustness of our code which will make it harder to take down. Also only making the necessary parts public will make our code like a black-box which means one will not be able to modify the code as they want and will have to obey the obligations of the system we created.

2.5 Boundary Conditions

In case of letting robots to reach the baseline of the humans, game will go to the game over screen and player will granted the loser title. If user opens the program again while playing, one of the games will exit. Also if an unexpected event occurs, the game will safely exit without harming anything on the system.

3 Subsystem services

We used 2 architectural design patterns in our design, which are Façade design pattern and MVC (Model-View-Controller) design pattern. We need to divide our project into subsystems to reduce the complexity of the system. We do this by using MVC pattern. Each part of the project belongs to controller, model or view part.

We used Façade design pattern for making software libraries easier to use and readable also to reduce dependencies of the system. It provides an interface to a long code sample like a library or a subsystem. This action is done by creating an intermediate class which takes the requests and handles them by using the collection of classes that needs to be simplified. While simplifying the classes and subsystems, it also provides reusability, flexibility, extensibility and easy maintenance. We used Façade design pattern in out interface and gameplay subsystem.

Diagram for Detailed System Design can be seen at Figure-2 and diagram for the Final Object Design of our program is at Figure-3. Now the subsystems will be explained.

3.1 User Interface Subsystem

The User Interface Subsystem is responsible for the creation of the interface for the user. It consists of the MainMenu, Credits, Settings, Hints and MapManager. This subsystem is responsible for displaying the main menu when the user first runs the game, the credits to display whom are the credits for, the settings for navigating through the settings, the hints

for user to learn some example strategies for the game and finally the map manager for the in game adjustment.

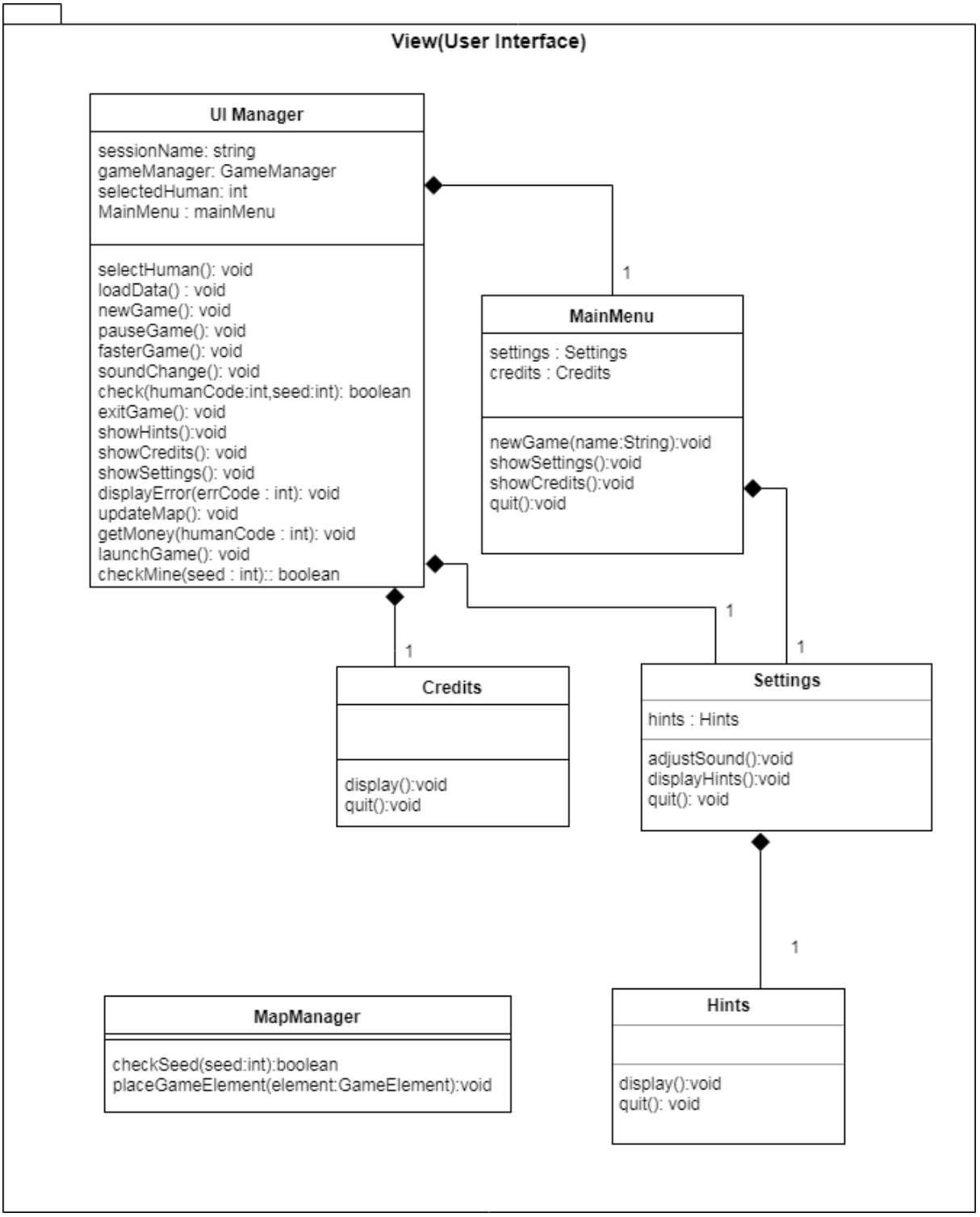


Figure-7 User Interface Subsystem

3.1.1 UIManager Class

Constructor

public UIManager(): This constructor is used for initializing the sessionName, gameManager, selectedHuman and mainMenu.

Attributes

private String sessionName: It is the game's session name which is same with the player's name.

private GameManager gameManager: It is a GameManager property used for the whole operations related with this class.

private int selectedHuman: It is the code of the human selected from the human menu. For example, miner's code is #5.

private MainMenu mainMenu: It is a MainMenu property used for the whole operations related with this class.

Methods

public void selectHuman(): This method is used to select a human from the human menu.

public void loadData(): This method is responsible for loading the data about map, human menu, in short all the data about the game, at the beginning of the game.

public void newGame(): This method is used to create a new game.

public void pauseGame(): This method is used to pause the game.

public void fasterGame(): This method is used to speed up the game's speed.

public void soundChange(): This method is used to change the soundValue.

public boolean check(int humanCode, int seed): This method is used to control whether the player has enough seed to place the selected human.

public void exitGame(): This method is to exit from the game and return to the main menu.

public void showHints(): This method is used to display some information about humans and robots.

public void showCredits(): This method is used to show the Credits screen.

public void showSettings(): This method is used to show the Settings screen.

public void displayError(int errCode): This method will be used to display errors about the game. For example, the seed is not enough for the selected human there will be a warning on the screen.

public void updateMap(): This method is used to update the map after any operation.

public void getMoney(int humanCode): This method helps to get money from user if the seed is empty for the selected human.

public void launchGame(): This method is to launch the game.

public boolean checkMine(int seed): This method is used to control the miner whether there is a mine or not to update the money.

3.1.2 MainMenu Class

This is the user interface class of the Main Menu.

Constructor

public MainMenu(): This constructor is used for initializing the settings and credits.

Attributes

private Settings settings: It is a Settings property used for displaying the “Settings” screen.

private Credits credits: It is a Credits property used for displaying the “Credits” screen.

Methods

public void newGame(String name): This method used to start the game by taking player’s name as a parameter.

public void showSettings(): This method is used to display Settings screen.

public void showCredits(): This method used to display the Credits screen.

public void quit(): This method is used to exit from the game.

3.1.3 Credits Class

This is the user interface class of the Credits screen.

Constructor

public Credits(): This is a default constructor.

Attributes

No attribute is needed.

Methods

public void display(): This method used to display the Credits screen.

public void quit(): This method is used to exit from the Credits screen to MainMenu screen.

3.1.4 Settings Class

This is the user interface class of the Settings screen.

Constructor

public Settings(): This constructor is used for initializing the hints.

Attributes

private Hints hints: It is a Hints property used for displaying the “Hints” screen.

Methods

public void adjustSound(): This method is used to adjust the sound.

public void displayHints(): This method used to display the Hints screen.

public void quit(): This method is used to exit from the Settings screen to MainMenu screen.

3.1.5 Hints Class

This is the user interface class of the Hints screen.

Constructor

public Hints(): This is a default constructor.

Attributes

No attribute is needed.

Methods

public void display(): This method used to display the Hints screen.

public void quit(): This method is used to exit from the Hints screen to MainMenu screen.

3.1.6 MapManager Class

This is the user interface class of the Map Manager.

Constructor

public MapManager(): This is a default constructor.

Attributes

No attribute is needed.

Methods

public boolean checkSeed(int seed): This method takes an integer parameter and is used to check the seed whether the player is able to place a human on the map or not.

public void placeGameElemet(GameElement element): This method takes a GameElement parameter and is used to place a game element on the map.

3.2 Game Management Subsystem

UI Manager and GameManager are the base classes for all the entity and control objects, they are responsible for all the power-ups and enemy objects as well as setting up the background and the enemies for each stage. They take the input from the player through the interface “MouseListener”, and passes that information to the User, MainMenu and the MapManager.

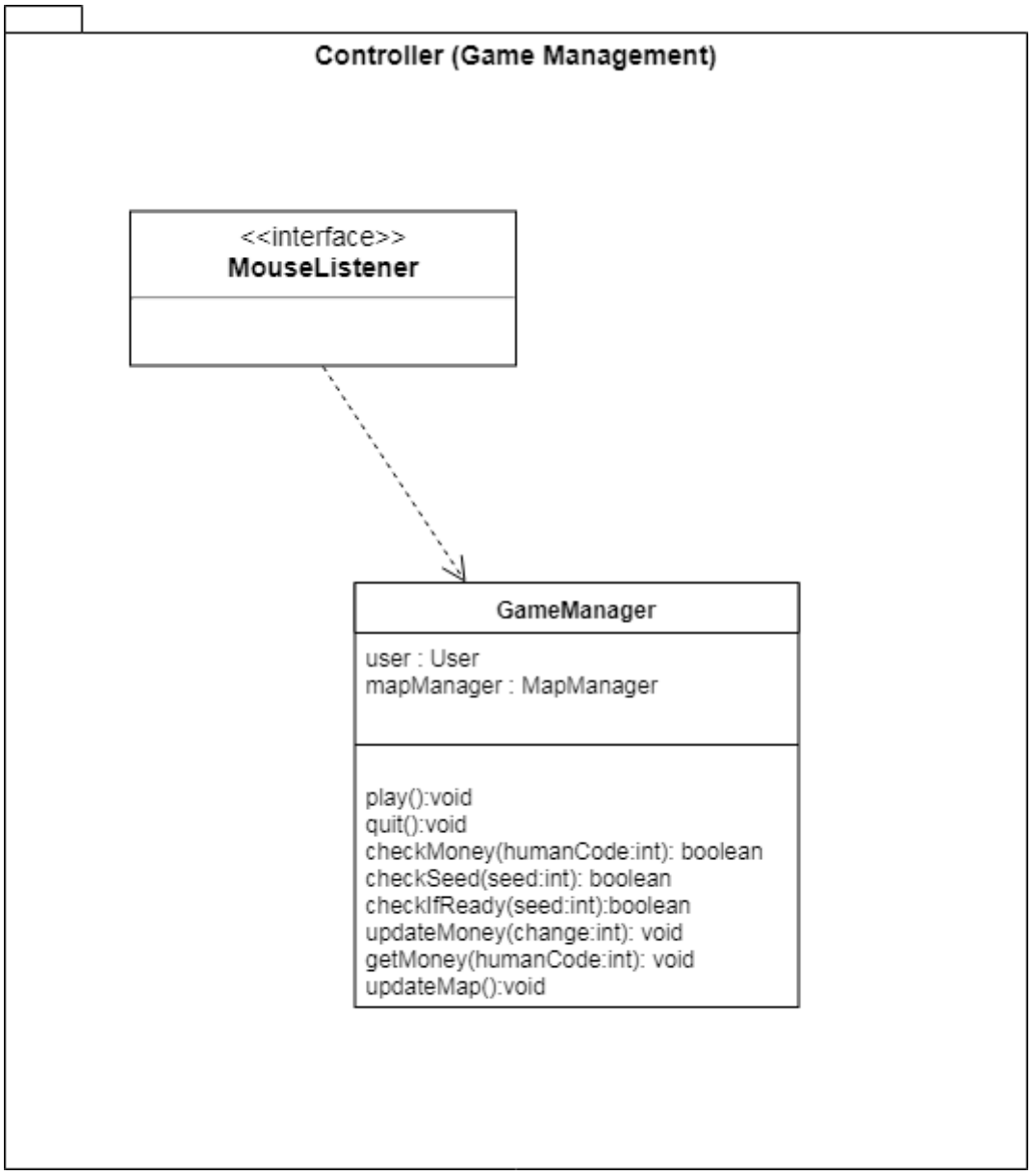


Figure-8 Game Management Subsystem

3.2.1 GameManager Class

Constructor

public GameManager(): This constructor is used for initializing the user and mapManager.

Attributes

private User user: It is an object for the user.

private MapManager mapManager: It is a MapManager property used for the whole operations related with this class.

Methods

public void play(): This method allows user to play the game.

public void quit(): This method is used to exit the game.

public boolean checkMoney(int humanCode): This method calls a parameter called humanCode. According to this parameter this function returns true if the money is sufficient for that human or it returns false if the money is insufficient.

public boolean checkSeed(int seed): This method returns true if the called seed is empty, otherwise it returns false.

public boolean checkIfReady(): There is a reload time for humans to be selected and placed. This function controls the selected human is reloaded or not.

public void updateMoney(int change): This method is responsible for the update the data about money after any operation related with money.

public void getMoney(int humanCode): This method helps to get money from user if the seed is empty for the selected human.

public void updateMap(): This method used to update the map after any map related operation.

3.3 Game Objects Subsystem

This is an abstract class which is the parent of all game related objects(e.g humans, robots). All of these elements have a location according to x and y coordinates. Other variables of this class are health and damage. Health is definitely important thing for object to be able to exist in the game. However, we can’t say the same thing for the damage. There will be objects with 0 damage.

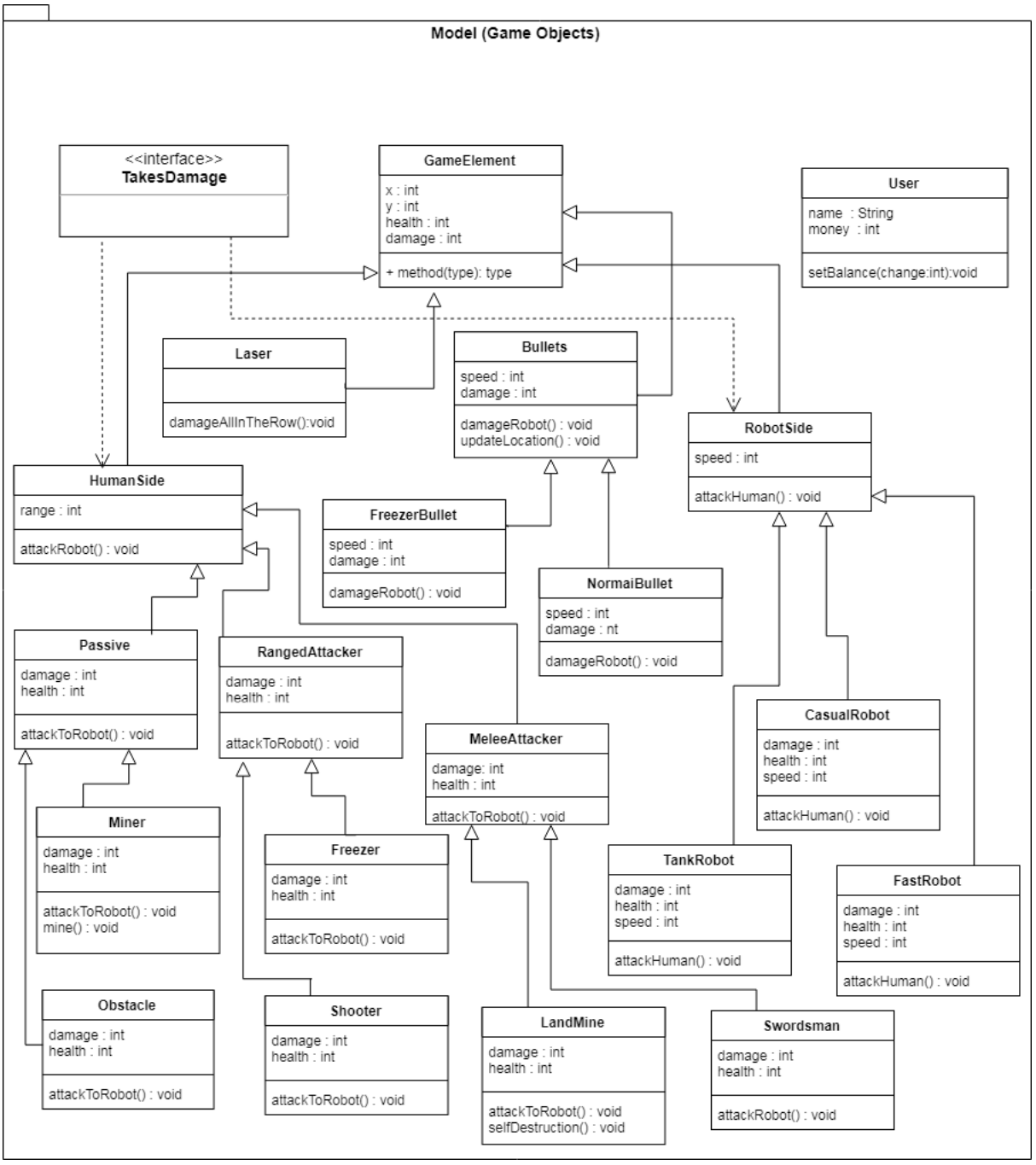


Figure-9 Game Objects Subsystem

3.3.1 Laser

This class is one level below from GameElement class. It is also a normal class (not abstract). It is there to represent the laser at the base that kills all robots in one row when one robot reaches to base. For this destruction, there is a function called damageAllInTheRow().

3.3.2 RobotSide

This class is one level below from GameElement class. It is also an abstract class. It is the parent of all Robot objects in the game. It has the class variable speed which is speed of all those robots.

3.3.2.1 CasualRobot

This class is one level below from Robot class. This represents the normal robots that we face at every stage. These are the most common ones. Its damage, health and speed variable will be regular compared to other robots.

3.3.2.2 TankRobot

This class is one level below from Robot class. This robot will be slower than the casual one but will have much more health. So, we will adjust speed and health variables.

3.3.2.3 FastRobot

This class is one level below from Robot class. This robot will be faster. We will adjust speed variable of the class accordingly.

3.3.3 HumanSide

3.3.3.1 Passive

3.3.3.1.1 Obstacle

This class is one level below from Human class. This object has much more health than other human objects and it has no damage. So, the parameters of the class differentiate more than other Human objects.

3.3.3.1.2 Miner

This class is one level below from Human class. Miners in the game represented by this class. However, class has damage variable which is equivalent to 0 and they have an extra mine() function.

3.3.3.2 RangedAttacker

3.3.3.2.1 Shooter

This class is one level below from Human class. This is another version of shooter which shoots two bullets at the same time which increases its damage.

3.3.3.2.2 Freezer

This class is one level below from Human class. This is another human with a gun but the bullet of this gun will freeze the target. In other words speed variable of robot will decrease class object that it hits.

3.3.3.3 MeleeAttacker

3.3.3.3.1 LandMine

This class is one level below from Human class. This represents the land mines. It can explode when one robot steps onto it. To make it destruct itself, we have the extra function called selfDestruction().

3.3.3.3.2 Swordsman

This class is one level below from Human class. It fights with robots head-to-head because it has lower range than other damaging human classes.

3.3.4 Bullet

This class is one level below from GameElement class. It is also an abstract class. It is the parent of all Bullet objects in the game.

3.3.4.1 NormalBullet

This class is one level below from Bullet class. It represents the normal bullets which are used by shooter and double shooter. It has two functions that are named damageRobot() and updateLocation().

3.3.4.2 FreezerBullet

This class represents bullets that slows the robots. It is quite similar to the NormalBullet. The implementation of damageRobot() method is different to achieve that slowing effect.

4 Low-level design

4.1 Design Pattern

We followed a design pattern in our project to achieve better understanding of our system, making it more readable and understandable by readers. In this section we will be giving examples and explaining why we used it.

4.1.1 Façade Design Pattern

As you know, Façade Design Pattern is a design pattern which helps us to hide complexity of the system and provides a huge but a single interface that user can interact with. It increases simplicity of the program by making that class handle all kinds of inputs and outputs in that subsystem.

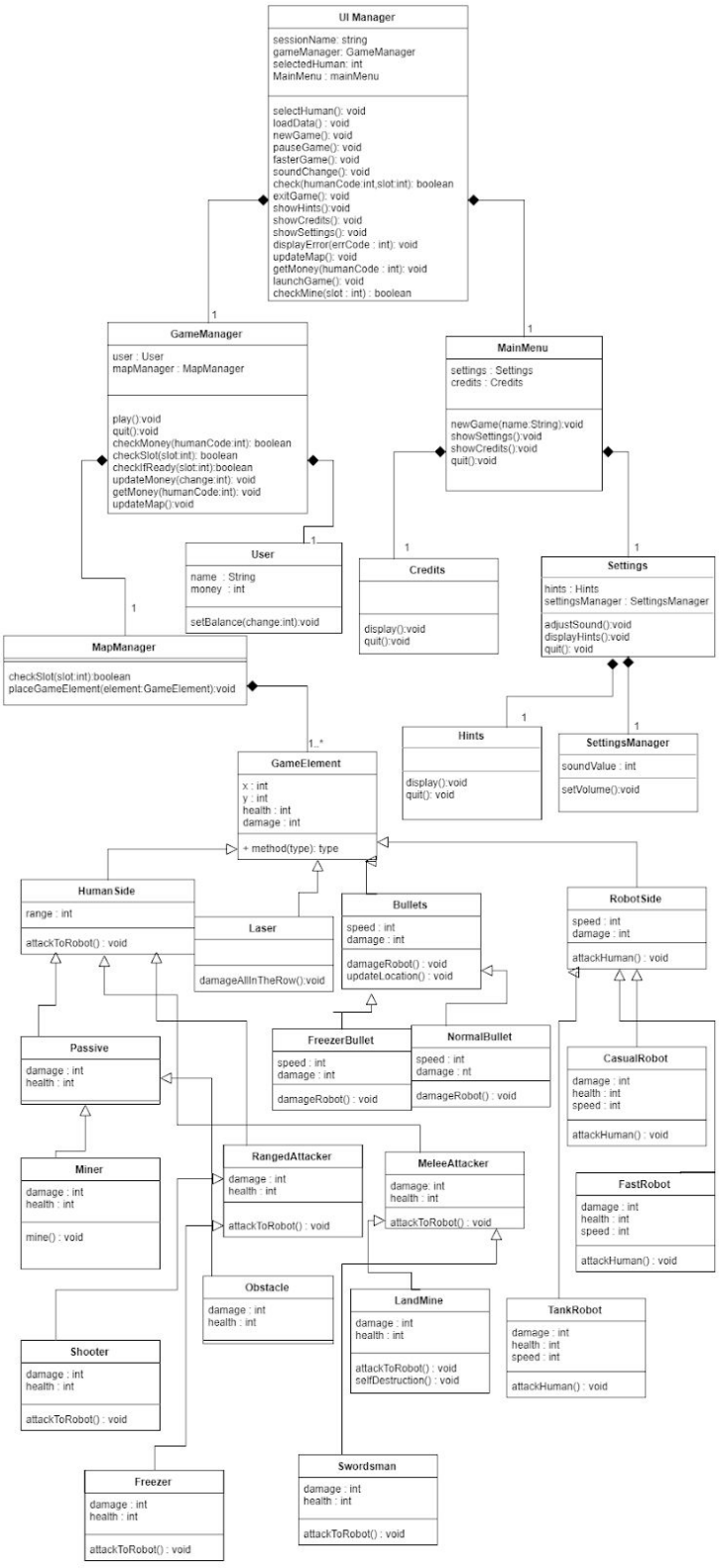
4.2 Object design trade-offs

Memory - Performance: This was by far the hardest tradeoff to consider but from the start we knew that performance would be more important than using memory recklessly. In huge games with high resolution graphics sometimes engineers must consider the fact that memory requirements of the game could be huge, but then again this was not similar to our case. Even if use memory recklessly we would not go any far because we use simple 2d graphics. On the other hand performance was a core key in our game to make it enjoyable that's the main reason we chose performance over memory.

Ease of use - Functionality: In order to reach a wider audience, and provide games that can be played by different age groups with equal ease, we have decided to make the game as easy to play as possible. To achieve this, some functionalities that might take the game much more complicated were left out.

Efficiency - Reusability: We indicated that our game will be reusable. We will try to write the code as much reusable we can. However, if the game's efficiency is affected in a bad way, we will try to write the code more efficiently rather than reusability.

4.3 Final object design



4.4 Packages

4.4.1 org.newdawn.slick.command

This package provides abstract input by mapping physical device inputs like mouse to abstract commands that are relevant to a particular game.

4.4.2 org.newdawn.slick.tiled

This package is useful to generate some maps.

4.4.3 org.newdawn.slick.state

This package allows us to make a state based game to divide the program into states like menu, credits, settings, play.

4.4.4 org.newdawn.slick.particles

This package provides some effects which will be useful when we try to animate the destruction of an object with a special effect.

4.4.5 org.newdawn.slick.gui

This package provides some GUI elements.

4.5 Class Interfaces

4.5.1 MouseListener

This interface is invoked whenever user makes an action with mouse. UIManager class will implement this interface to take necessary actions whenever a mouse action happens.

4.5.2 TakesDamage

This interface is implemented by Human and Robot classes. As its name suggests, objects of the classes that implements this interface will take damage.

5 Glossary

MVC: Model-View-Controller architecture

6 References

Object-Oriented Software Engineering, Using UML, Patterns, and Java, Bernd Bruegge and Allen H. Dutoit, 2010/3rd, Pearson

http://holub.com/uml/