CS 319 - Object-Oriented Software Engineering

Design Report

Virus Attack

Section 2 - Group F

Cholpon Mambetova

Doğan Can Eren

Melisa Onaran

Nergiz Ünal

Course Instructor: Uğur Doğrusöz

Table of Contents

1 - Introduction

1.1 Purpose of The System

Virus Attack is a system which aims to amuse users with its carefully designed, with different difficulty levels, gameplay. The system is based on user-friendly interface which basically guides the user how to play properly. The levels are well-designed in different difficulty levels, each aiming different purposes. For instance, the game begins with an easy level which aims to guide the user to become involved in game and experience how to play accordingly. Then, here comes a different level with an updated difficulty and aims to maintain the attention of the user. Subsequently, levels becomes different and much more difficult to accomplish.

1.2 Design Goals

It is quite hard to identify the design goals of the system with respect to further upcoming design qualities that the system will be based on and mainly focused. With this aspect it is appropriate to mention that numerous of our design goals follow non- functional requirements of our system that is mentioned in "Analysis Report". Critical design goals of our system are described as follows.

1.2.1 End User Criteria

Ease of Use: The system that we design is a game, therefore it is designed on a user-friendly interface aiming to make sure the player finds it easy to play and enjoyable while using the system provided by creators. From that point of view, system provides user-friendly menus, easily accomplished operations and well-designed navigation through menus and well-designed performance.

Ease of Learning: The player is not obligated to have a background knowledge about the system of the game. Therefore, it is one of the most important things to guide the player and make sure that the instructions are easy to follow. For this purpose, a document exists, created by creators of the game, which is available in the "View Help" section and aims to have player become updated about the qualities of the game.

1.2.2 Maintenance Criteria

Extendibility: To make the game alive it is mainly important to keep the game updated and to present the players new components and features about the game in order to have interest of players. That is why, the design is available to be updated with new versions and easy to modify to the existing game.

Portability: In this maintenance, we have decided that the system will be implemented in Java, it involves platform independency therefore the system will provide portability.

Modifiability: The system(game) is designed to be easily modified. Therefore, to be able to achieve the goal this quality would make the minimization of the couples of the subsystems and therefore perfectly avoids the impacts of components of the system when there is a change.

1.2.3 Performance Criteria

Response Time: As in most of the games, in this game it is quite important to have pressure on the players to be effective immediately in order not to get killed, therefore the player would be focused and interested during the game and also enjoys the challenge. When they are focused to the game, visual animation is also provided almost immediate through the player's action and the process of the game.

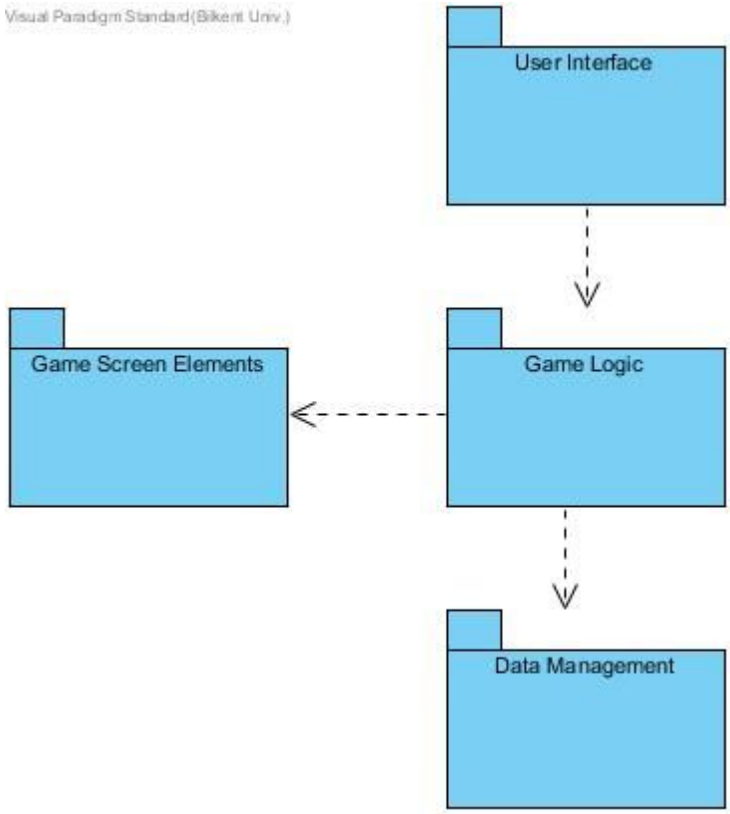
1.2.4 Trade - Offs

Ease of Use and Ease of Learning vs. Functionality: We propose our priority as usability being higher than functionality. Meaning the user should not be having trouble while learning how to play and how to process the game. Therefore the system does not make conflictions about the functionality to the player and the player does not get lost around the system process. When the game is easily understandable player would enjoy more.

Performance vs. Memory: The main goal of the system is to make the animations, transitions and effects very intense. Therefore, the performance is the system is one of the main things that is focused and thought. For instance the game involves a bonus item that makes the user gain the invincibility, freezing and getting a life as a bonus. Because system needs the best performance as possible it has a game map in the memory which is to access fast when it is needed rather than having iteration for all the map.

2 - Software Architecture

2.1 Subsystem Decomposition



The system uses a 3-layer architectural style. Also, as it has a closed architecture style, a component can only communicate with its neighbors or the lower-layered ones. There are three layers which are in a hierarchical order. First layer is called as “UI Layer”, which includes the User Interface component. This layer provides the ability to interact with the user. The second layer is called “Application Logic Layer”, which includes the Game Logic and Game Screen Elements components. This layer handles the logic operations of the system. The third layer is called “Data Persistency Layer”, which includes the Data Management component. This layer provides the ability to work with the data such as saving the highscores and retrieving them.

2.2 Hardware/Software Mapping

Virus Attack will be implemented in Java programming language and for this purpose the latest version of JDK(8) will be used. In hardware configuration, Virus Attack needs keyboard, for to type the names of the players which is necessary for the high score list and

also to make move the virus. Since it is going to be implemented in Java, a computer with basic softwares installed as an operating system and a java compiler to compile the code and also to run the .java.file. And, also the main reason to choose Java ,which was its platform independency, will be used. And to be able to store the data we already will have a .txt based structures to formalize the game maps and to store the high score list, chosen operating system will need to support .txt file formats. On the other hand the system will not require any internet based connection to operate.

2.3 Persistent Data Management

The game basically does not involve a complex database system, it just stores the map structure and the high score list as text files. The game maps will necessarily be created before the Virus Attack is executed, therefore its data will be permanent. If text file is corrupted, through the working process of the system that we have designed it would not affect in-game, such as items but the system would no longer be able to load the corrupted map. Also it is planned to store the sound of the effects and the images of the game items in hard disk with proper sound and in image formats such as .gif.

2.4 Access Control and Security

As mentioned earlier(Hardware/Software Mapping section), Virus Attack does not require any internet connection. Therefore, the players will not be in trouble with internet based security issues or else will be freely playing the game after installation.

2.5 Boundary Conditions

Initialization

Virus Attack does not have a regular .exe or such extension, therefore it does not require an install. Though the game is going to come with an executable .jar file.

Termination

Virus Attack is designed to be able to be terminated, in other words closed, by clicking “Quit Game” button in the main menu. If player wants to quit during game play, the system

provides a pause menu by which player can return to main menu and perform quit.

Error

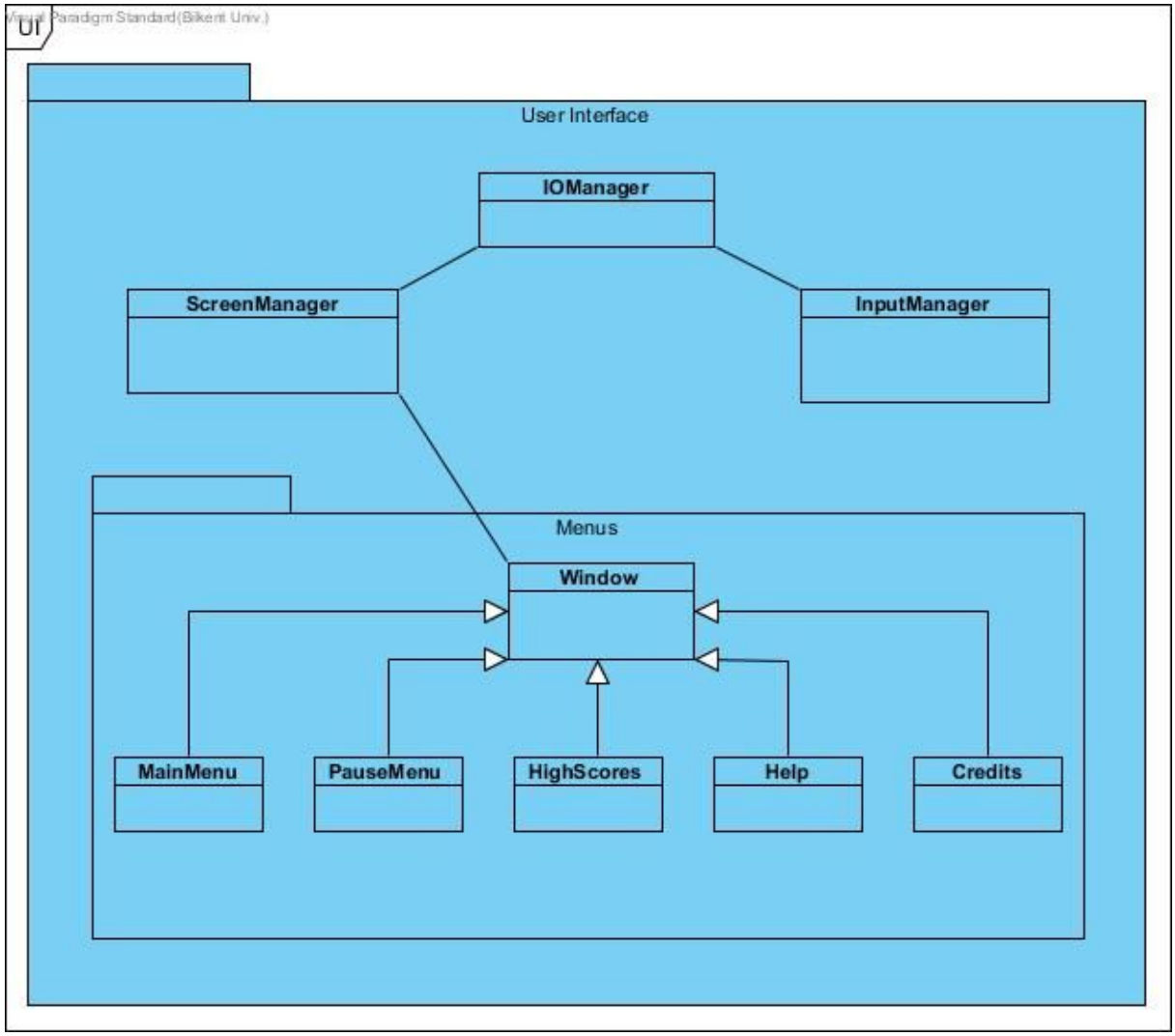
If an error occurs that game resources could not be loaded such as sound and images, the game will become starting without having any images or sound. Also, if program does not respond because of a performance issue, even we keep these kind of problems minimum, Player will lose all of current data.

3 - Subsystem Services

3.1 Design Patterns

Façade design is a design pattern which provides developers a unified interface to a set of interfaces by defining a higher-level of interface in a subsystem which is basically easier to use. This pattern is known for its opportunities such as providing maintainability, flexibility when developing the system, extendibility and reusability. Because through the mechanism of this pattern, any changes made in components of the subsystem can be reflected when they are made in the "Façade Class". In our design we have chosen to use façade pattern in the following subsystem, Game Entities. For this subsystem our façade class is MapManager class which basically executes the operations to deal with the entity objects in this system.

3.2 User Interface Subsystem Interface



HighScores Class:

HighScores class uses the HighScoresManager that holds the highest scores that was ever achieved in this game, to print them to screen using ScreenManager. It also extends the Window class.

Help Class:

Help class is used to display some useful information for the user to easily learn how to play the game. It also extends the Window class and uses ScreenManager to print itself.

Credits Class:

This class is responsible to display the information about the creators of the game. It also extends the Window class and uses ScreenManager to print itself.

Window Class:

This class is used to bring together the common features of its child classes such as menu classes and other displayed windows used in the game.

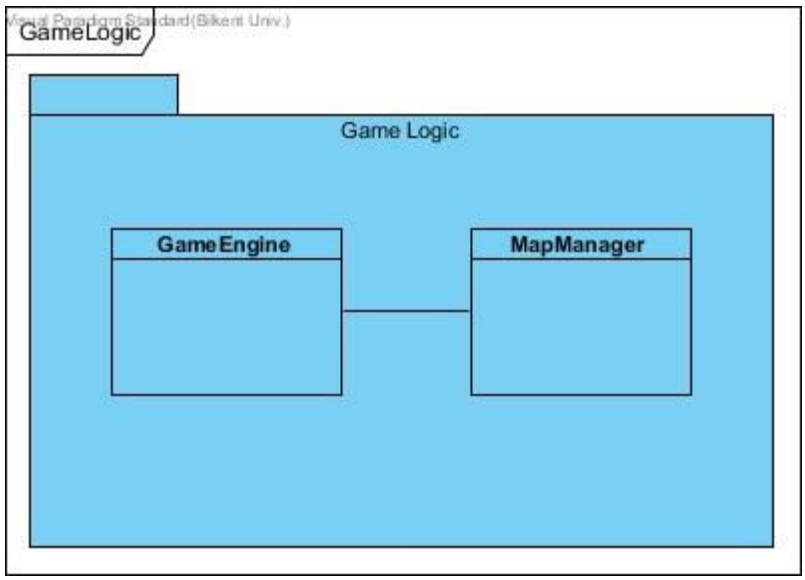
InputManager Class:

This class is responsible for taking the user input and report it to the IO Manager.

IOManager Class:

This class outputs the ScreenManager to the display and gets the input and sends the input to the InputManager.

3.3 Game Management Subsystem Interface



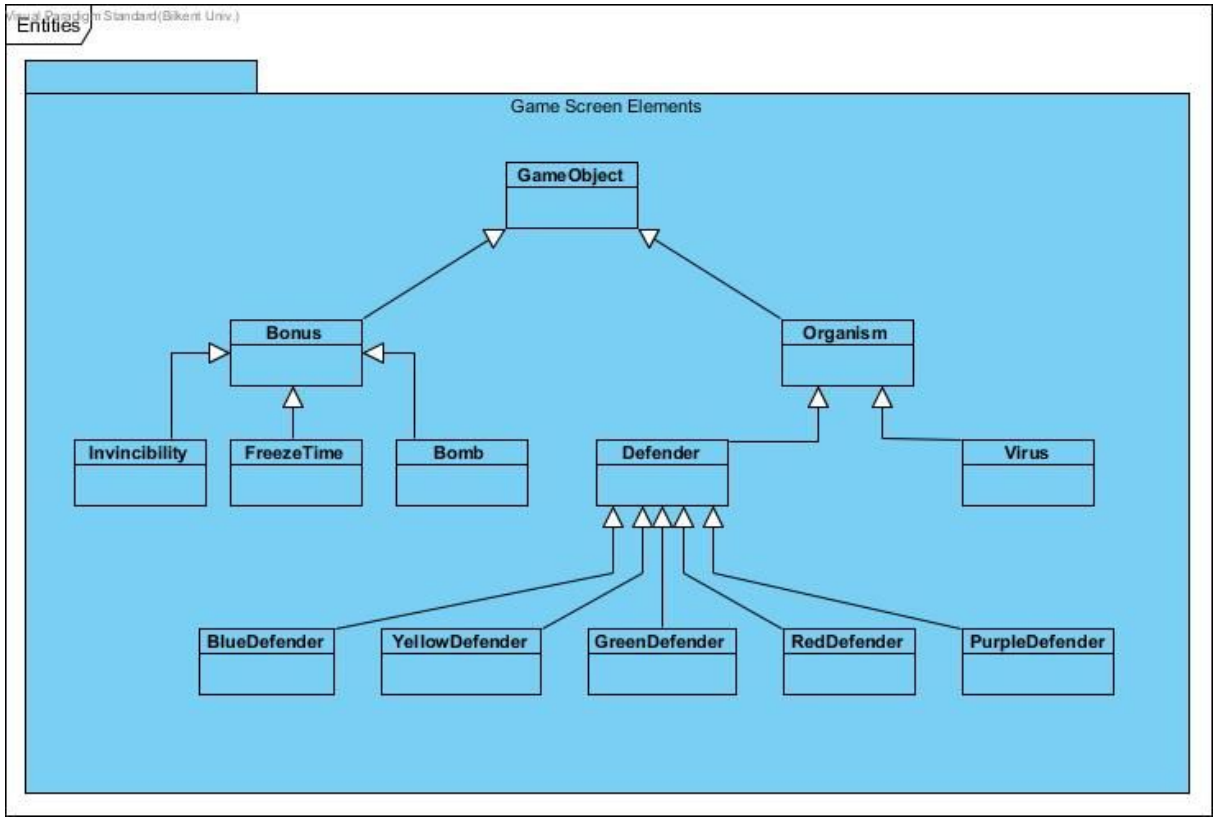
GameEngine Class:

This class is responsible for handling the main logic operations of the system. It communicates with all the other subsystems to process.

MapManager Class:

This class is responsible for tracking and reporting the objects on the map. It interacts with the GameEngine class to provide information.

3.4 Game Entities Subsystem Interface



c

GameObject Class:

This class is responsible for every type of game objects, like the virus or bonuses, on the game. It is a parent class that holds positions, colors, sizes are inherited from this class to every child classes. Drawing of objects and updating their positions are also provided by this parental class to the child classes.

Bonus Class:

This class brings together different type of bonus classes' common features. These features are:

Activation: When virus and the bonus collide, the bonus is being activated.

Lifetime: Determines how much time the bonus can be available on the screen.

Duration: Determines how much time the bonus power will be efficient.

Invincibility Class:

Child class of Bonus class, this bonus makes the virus invincible for a given time duration.

Freeze Class:

Child class of Bonus class, this class is responsible for freezing the defenders for a time duration of the bonus.

Bomb Class:

Child class of Bonus class, this class responsible for destroying the defenders within a radius of damage. When the virus collides with it, it gets activated and kills the nearby defenders.

Organism Class:

This class is parent class for defenders and the virus. This class is responsible for organism's vertical and horizontal velocities as a common features of Virus and Defender classes.

Virus Class:

This class extends the Organism class. It is responsible for all virus data such as HP, Health Point, max HP and invincibility. If the virus collides with the defender and the invincibility is active, the virus doesn't lose any HP.

Defender Class:

This class gets together common features of five different defender classes and extends the Organism class. These five child classes of Defender class have damage, color, and size as common properties. Also responsible for velocities of virus due to the parental features. Child classes are for specification of these common features.

BlueDefender Class:

Defender that creates low damage, blue colored, medium sized and slow.

YellowDefender Class:

Defender that creates low damage, yellow colored, small sized and fast.

GreenDefender Class:

Defender that creates medium damage, green colored, big sized and slow.

RedDefender Class:

Defender that creates high damage, red colored, small sized and fast.

PurpleDefender Class:

Defender that creates high damage, purple colored, big sized and fast.

3.5 Data Management Subsystem Interface

HighScoresManager Class:

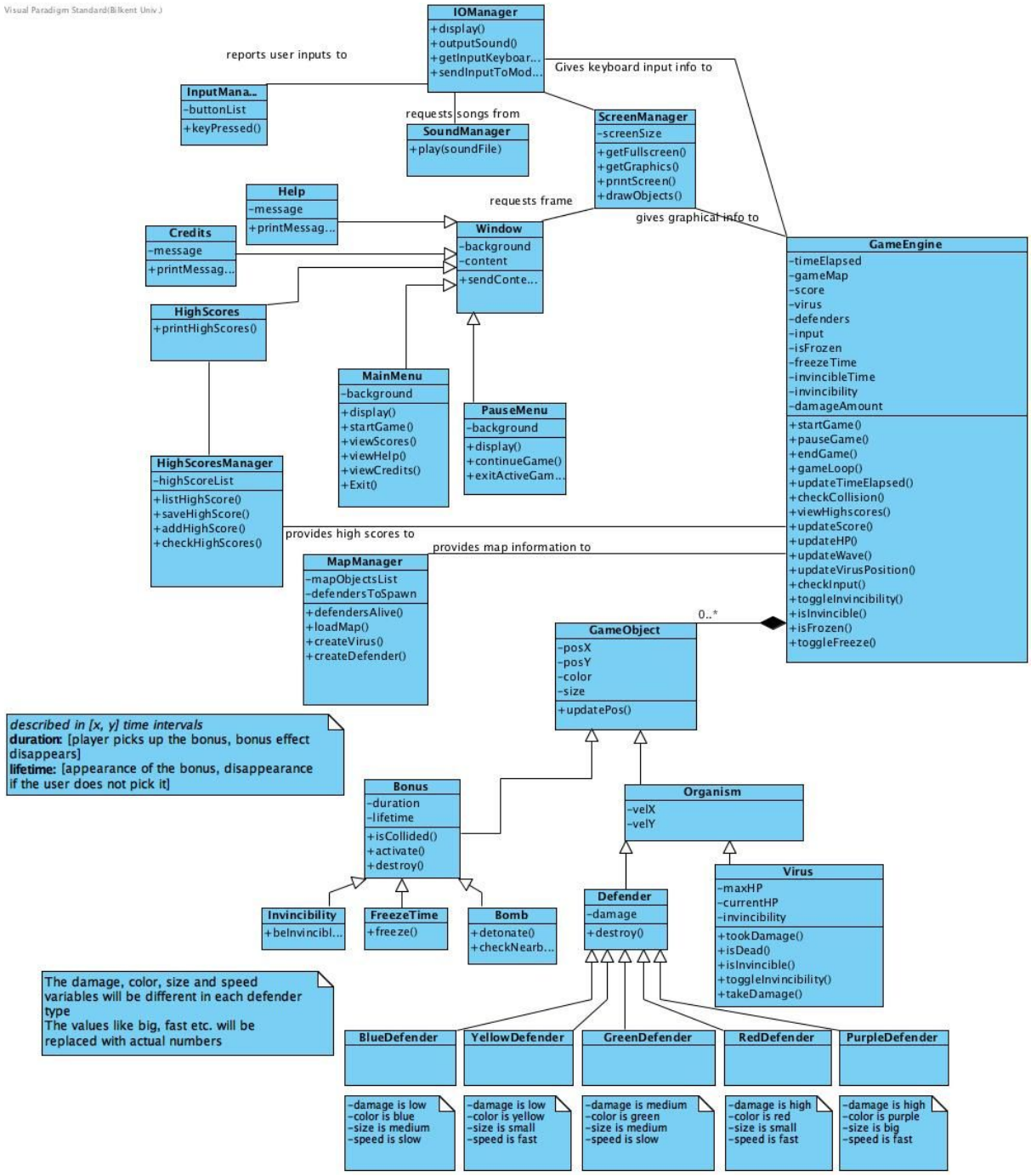
This class is responsible for keeping, saving and retrieving the highscores. It interacts with the GameEngine class to process data.

4.0 Low Level Design

4.1 Object Design Trade-Offs

Sometimes it is mainly important to choose in between defined design goals. Therefore when we had the same situation that we needed to choose one instead of the other we had necessary trade-offs. The first trade-off that we deal was in between run-time efficiency and ease of writing. Rather than dealing and checking collisions in between virus and defender, virus and bonus, etc. it is decided to be better to have one single collision checker method and pass two game objects to it. The main idea that is thought to be better is that the system of computers in these days are quite fast and it can handle a checker method like this very easily. Another trade-off that we needed to deal was in between usage of memory efficiency and performance flexibility. It is thought to be a good idea to use static variables mostly because it was needed in many classes and it was going to be used constantly. Therefore it is decided to use memory much more efficient by using static variables but when done that it caused and it will cause more than usual in the upcoming days therefore it will be a bit up more problematic in side of performance flexibility.

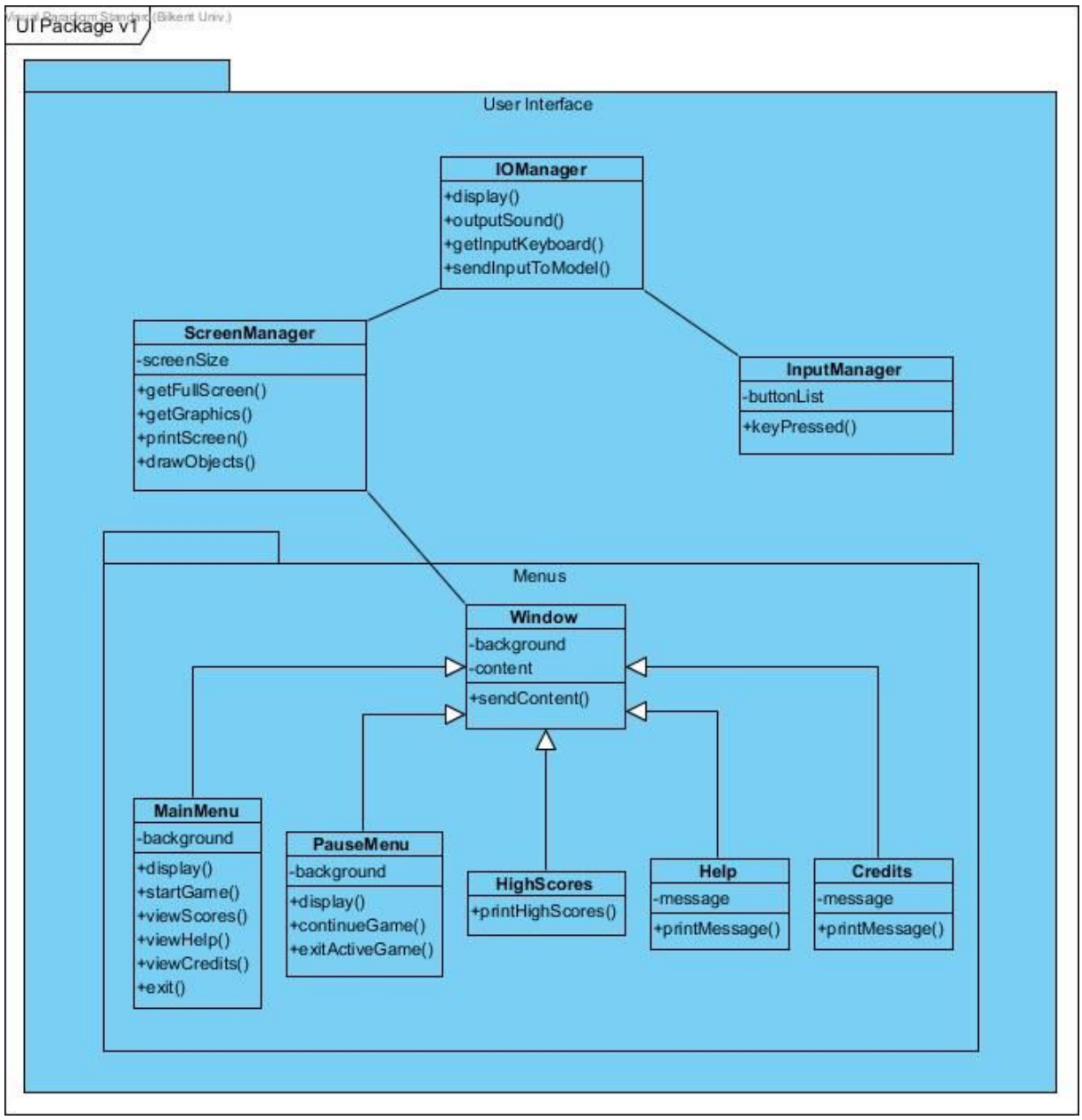
4.2 Final Object Design



4.3 Packages

4.3.1 User Interface

This subsystem provides connection between the user and program. It takes user's input and sends it to the Game Engine and prints the screen generated by Game Engine to the user.



IOManager Class

This is the main class that maintains the goal of the subsystem. It has no attributes, since it serves as a communicative class among other classes, and has only functions/methods.

Methods:

- display (): void: this method prints the screen provided as an argument by

other classes.

-getInputKeyboard(): int: this method reads the input from keyboard by using the Input Manager class and returns to a caller as int which represents inputs ASCII code.

-sendInput(): void: it sends the input read from keyboard by calling getInputKeyboard() method to Game Engine.

InputManager Class

This is the class that helps to read input from keyboard.

Attributes:

- buttonList: int array: it holds the ASCII codes of all acceptable input keys from keyboard.

Methods:

- keyPressed(): int: this method gets the input from keyboards when it is pressed and holds it only if it qualifies with the list of acceptable keys.

ScreenManager Class

This is the class that takes the information from Game Engine and Window class and sends it to IOManager class to print.

Attributes:

- screenSize: int array: it has two elements holding width and height of the screen.

Methods:

- getFullScreen(): void: it gets the measurement of the full screen in pixels and stores them in screenSize variables.

-getGraphics(): image: it gets the graphics that needs to be printed as argument.

-printScreen(): void: uses the IOManager to print the screen.

-drawObjects() void: uses to draw objects taken from Game Engine. It also used IO Manager to print it.

Window Class

This is the parent class for classes that to be printed other than the actual game objects during the active game.

Attributes:

- background: image: the background of the screen.

- contents: array of string:

Methods:

- sendContent(): void: sends the content of the screen to be printed to ScreenManager class object.

MainMenu Class

This is the child class of the Window class. It represents the main menu that user sees when they just open the game.

Attributes:

- background: image: the background of the screen.

Methods:

- display(): void: requests the menu to be displayed through Screen Manager and IO Manager.

-startGame(): void: requests the game to start by Game Engine.

-viewScores(): void: calls the HighScore class object to print the highest scores.

-viewHelp(): void: calls the Help class object to print the helpful information.

-viewCredits(): void: calls the Credits class object to print the info about the creator of the game.

-exit(); void: exits the game.

PauseMenu Class

This is the child class of the Window class. It represents the pause menu that user sees when they pause the active game.

Attributes:

- background: image: the background of the screen.

Methods:

- display(): void: requests the menu to be displayed through Screen Manager and IO Manager.

-continueGame(): void: closes itself and resumes the game.

-exitActiveGame(); void: exits active game and goes to main menu.

HighScores Class

This is the child class of the Window class. It is used to print the High Scores achieved and stored in the game.

Methods:

- printHighScores(): void: it calls the HighScoresManager to print the highest scores and prints the provided info by the ScreenManager, which calls the IOManager. The HighScoresManager class will be described in following sections.

Help Class

This is the child class of the Window class. It is used to print the useful information that might help the user to understand the game and how it is played. It can also contain some tips for better performance.

Attributes:

- message: string: contains the actual text to be printed.

Methods:

- printMessage(): void: it prints the string message (calls the ScreenManager, which class the IOManager).

Credits Class

This is the child class of the Window class. It is used to print the information on the authors of the game and their contact information.

Attributes:

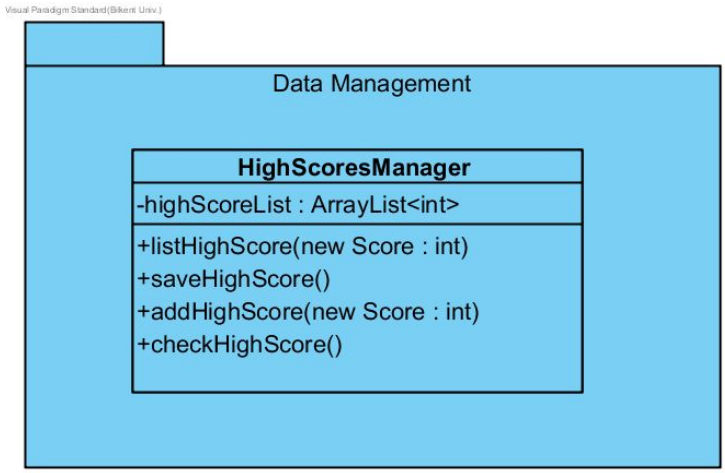
- message: string: contains the actual text to be printed.

Methods:

- printMessage(): void: it prints the string message (calls the ScreenManager, which class the IOManager).

4.3.2 Data Management

This subsystem is basically responsible for holding the high scores in data and updating the upcoming high scores in Virus Attack.



HighScoresManager Class

This class a mere class of Data Management and is responsible for all the high scores and it checks, adds if it is higher than previous, saves the new one, lists and adds to the array list.

Attributes:

- highScoreList(): int score: It holds the high scores as a list.

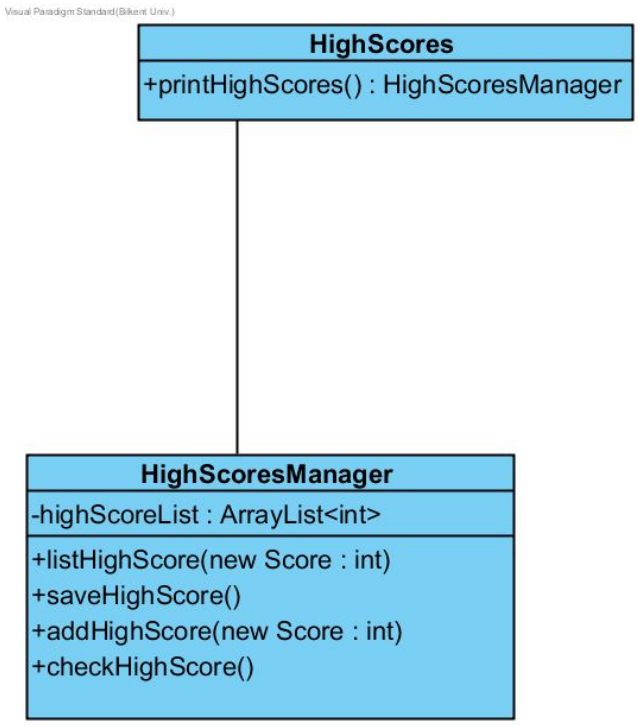
Methods:

-listHighScore() int: it gets the high score and lists.

-saveHighScore(): int: it gets and saves the latest high score that is achieved.

-addHighScore(): int: it gets the newest high score and it adds if the newest high score is higher than the previous one.

-checkHighScore(): int: it checks the high score whether achieved a new one.



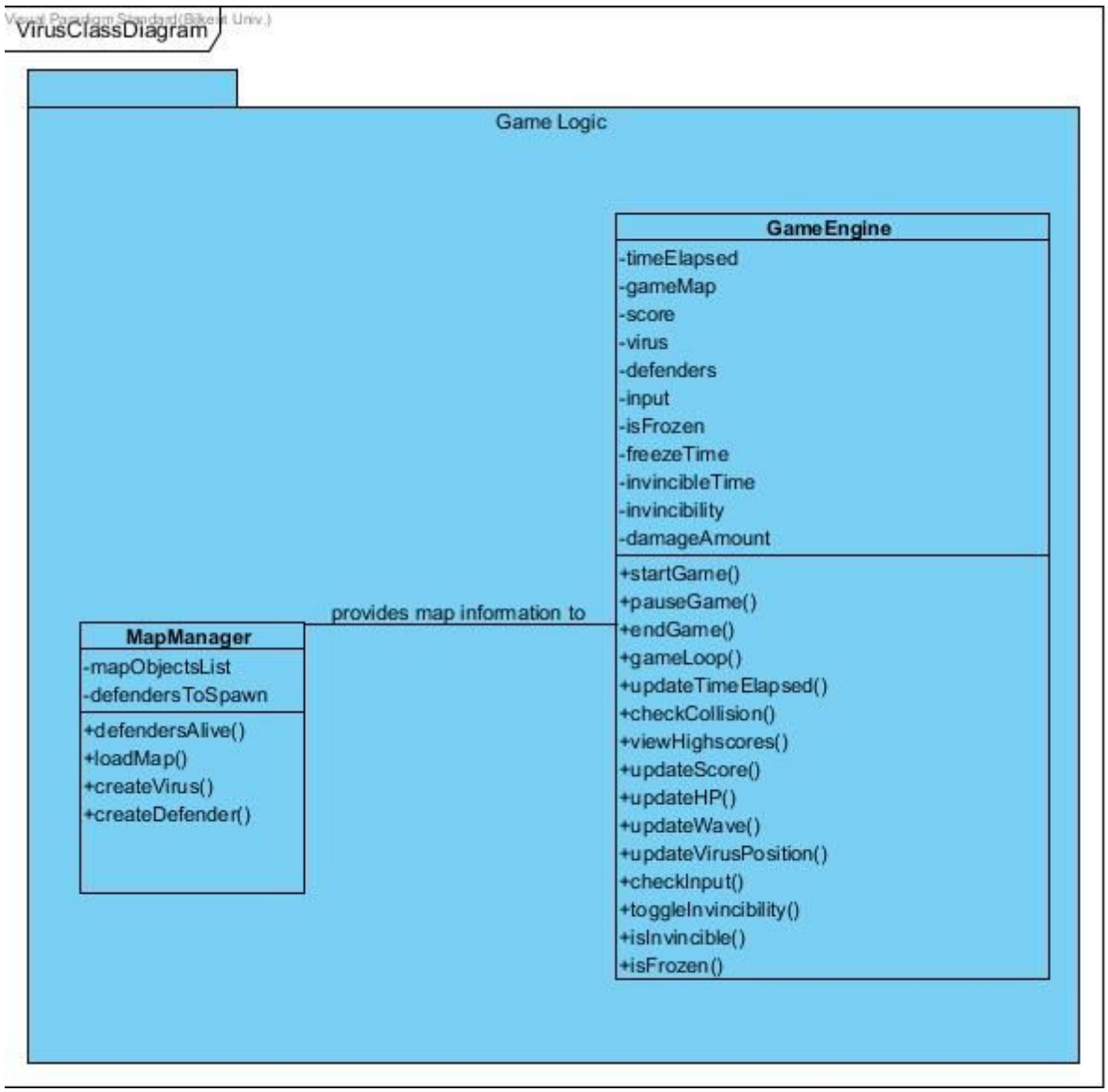
HighScores Class

After all the changes checked, added, saved and listed with an update in highScoreList this class would print all the data and send it to the Window class to display.

Methods:

-printHighScores(): int: It prints all the high scores after all the updates are made.

4.3.3 Game Logic



Game Engine Class

Attributes:

timeElapsed: long: Starts with zero when the game starts. Type is long becaues it's unit is milisecond so it can be huge.

GameMap: Map: In every level there is only one Map object. It is constructed with beginning of every new level. When every new level starts it is modified according to management subsystems.

Score: int: It represents player's score.

Defenders: Defender: Every defender objects in the game are needed as an attributes.

Input: IOManager: User inputs to the game, such as pressing up button to move the virus

through the upward.

isFrozen: boolean: It indicates that whether defenders are freezy situation or not.

freezeTime: long: It indicates how much time defenders will be frozen.

invincinbilitiyTime : long: It indicates how much time virus will be invincible.

invincibility: double: It indicates whether the virus invincible or not.

damageAmounts: float: It indicates how much the virus took damage from defenders, and how much HP(Health Points) it will lose.

Methods:

startGame(): When the user press the button *"start game"* on the Main Menu Screen, Game will be started. All defenders and the virus are created end elapsed time starts with and start to increase by the time.

pauseGame(): User can stop the game without exiting. Elapsed time will not increase and all game objects cannot move at that time. Pause menu will be displayed.

gameLoop():

updateTimeElapsed(): It increments timeElapsed() regularly.

checkCollision(): Check whether the virus has any collision with any defenders or bonusses.

viewHighScores(): When the user have high score the game store user's information and their score in the list. Also these values displayed on the screens.

updateScore(): As the user continiue the game score changes. This method increments or decrements the *"score"*.

updateHP(): The virus stars the game with given HP and as the game continue when the virus has collision with defenders*(checkCollision())* the HP will be decreased by damage amount *(damageAmount)*.

updateVirusPosition(): As the user directs the virus by up – down and left- right bottom the virus's position changes. This method chage the posX an posY values of the virus.

checkInput():

toggleInvincibility() : Check whether the virus has collision with invincibility bonus. If they have collided this method make the virus invincible for a given time.

isInvincible(): Check whether the virus invincible and the given time for this bonus does not expire yet.

toggleFreeze(): Check whether the virus has collision with freeze bonus. If they have collided this method make the game frozen for a given time.

isFrozen(): Check whether the game frozen and the given time for this bonus does not expire yet.

MapManager Class.

Attributes:

mapObjectsList: GameObject: ArrayList of every GameObjects in the game will be disappeared.

defendersToSpawn: Defender: Every defenders that will be generated during the game time.

Methods:

defenderAlive(): Check how many defender alives and which one's alive.

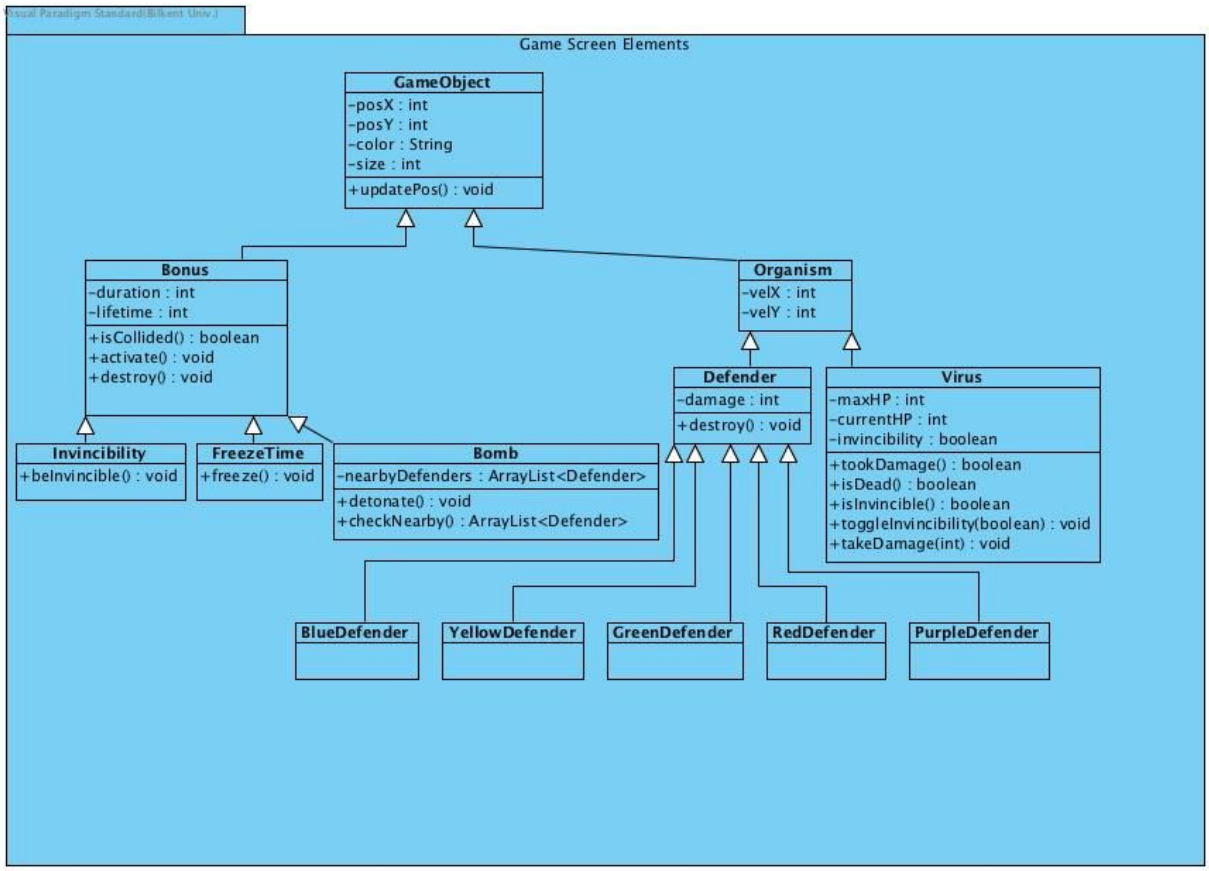
loadMap():

createVirus(): When the game has been started the virus created by using "Virus" class.

createDefender(): When the game has been started the defenders of the game created by using "Defender" class. These defenders are listed in "defendersTo Spawn".

Game Entities

This subsystem is responsible for holding the game objects and their attributes used in the Virus Attack. In this subsystem, there are a total of 13 classes. All of the objects in the game inherits from an abstract class called GameObject. It has such a design that enables the programmers to do modifications such as changing, deleting or inserting objects quite easily.



GameObject Class

Attributes:

- **float** posX: X position of the object
- **float** posY: Y position of the object
- **String** color: defines the color of an object
- **int** size: defines the size of an object

Methods:

- updatePos(): updates the position of an object considering its velocity values and/or its existence

Bonus Class

Attributes:

- **int** duration: defines the time interval starting from the pick-up moment until the effect goes off
- **int** lifetime: defines the time interval between appearance and disappearance if the bonus is not picked

Methods:

- **boolean** isCollided(): returns true if the virus collides with such bonus
- **void** activate(): applies the bonus effect to the virus
- **void** destroy(): destroys such bonus

Invincibility Class

Attributes: -

Methods:

- **void** beInvincible(): makes the virus invulnerable

FreezeTime Class

Attributes: -

Methods:

- **void** freeze(): slows down the defenders, so that the virus can evade easily

Bomb Class

Attributes:

- **ArrayList(Defender)** nearbyDefenders: keeps the objects nearby in an array (this attribute will be further considered)

Methods:

- **void** detonate(): detonates the bomb
- **Defender[]** checkNearby(): checks the nearby defenders

Organism Class

Attributes:

- **int** velX: defines the velocity of the organism in the x axis
- **int** velY: defines the velocity of the organism in the y axis

Methods: -

Defender Class

This class is an abstract class of all defenders. The empty-looking subclasses won't be described here, all of the defenders have different colors, velocities, damages and sizes which will be overwritten as the implementation goes on.

Attributes:

- **int** damage: defines how much of the HP would lost from the virus if a collision happens

Methods:

- **void** destroy(): removes such defender from the map

Virus Class

Attributes:

- **int** maxHP: defines the maximum value of health points of a virus
- **int** currentHP: defines the current value of health points of such virus
- **boolean** invincibility: holds true when the virus picks up an invincibility bonus

Methods:

- **boolean** tookDamage(): checks if the virus took damage
- **boolean** isDead(): returns true when the currentHP value reaches 0
- **boolean** isInvincible(): checks if such virus is invincible by checking the invincibility value
- **void** toggleInvincibility(boolean): changes the value of invincibility
- **void** takeDamage(int damage): decreases the currentHP value of such virus

5. Glossary and References

5.1. Glossary

1. Defense mechanism : This mechanism represents the immune system cells which attack to the player(virus).
2. Health Points(HP) : At all levels the player (the virus) has a limited life, as the virus gets hit from the cells, HP will be decrease. If health points are drained away the virus will die.
3. Host Body : This term represents the game space of virus, the virus try to reach it's target in the human body. Every level occur in the different part of the body. The goal is to stay alive as long as possible.

4. Immune System Cells : When the virus enters into the host body, the immune system cells try to hit the virus and kill it. These cells defend the host body.
5. Invincible Virus : The immune system cells have no effect on the virus during invincibility time.
6. Power - Ups : Power - ups give some benefits to the virus during the game. Power - ups are sent through the game as an item, if the virus collides with them it will gain their benefits.
7. Virus : The user play the game as a virus, in other words the virus represents the player. The player should stay alive as the virus throughout the game.

5.2. References

1. The game we have inspired: https://www.youtube.com/watch?v=1gir2R7G9ws&t=150s

4. For the models : https://www.visual-paradigm.com