

CS 319

Object-Oriented Software Engineering  
Fall 2014

**Conquerors**

Design Report

Group H  
Çağla Arıkan

Metin Enes Uçan

Can Akgün

İsmail Can Dereli

Table of Contents

[1. INTRODUCTION 4](#_Toc405144535)

[1.1. Purpose of the System 4](#_Toc405144536)

[1.2. Design Goals 4](#_Toc405144537)

[1.2.1. Goals 4](#_Toc405144538)

[1.2.2. Trade-Offs 5](#_Toc405144539)

[1.3. Definitions, acronyms, and abbreviations 6](#_Toc405144540)

[1.3.1. Definitions 6](#_Toc405144541)

[1.3.2. Abbreviations 6](#_Toc405144542)

[2. SOFTWARE ARCHITECTURE 7](#_Toc405144543)

[2.1. Subsystem Decomposition 7](#_Toc405144544)

[2.2. Hardware / Software Mapping 8](#_Toc405144545)

[2.3. Persistent Data Management 8](#_Toc405144546)

[2.4. Access Control and Security 8](#_Toc405144547)

[2.5. Boundary Conditions 8](#_Toc405144548)

[2.5.1. Initialization 8](#_Toc405144549)

[2.5.2. Termination 9](#_Toc405144550)

[2.5.3. Error 9](#_Toc405144551)

[2.5.4. Game Updating 9](#_Toc405144552)

[3. SUBSYSTEM SERVICES 9](#_Toc405144553)

[4. Low-Level Design 12](#_Toc405144554)

[4.1. Object Design Trade-offs 12](#_Toc405144555)

[4.2. Final Object Design 13](#_Toc405144556)

[4.3. Packages 15](#_Toc405144557)

[4.4. Class Interfaces 18](#_Toc405144558)

[4.4.1. Main Menu 18](#_Toc405144559)

[4.4.2. Input Manager 19](#_Toc405144560)

[4.4.3. Point 21](#_Toc405144561)

[4.4.4. Edge 22](#_Toc405144562)

[4.4.5. Edge Manager 23](#_Toc405144563)

[4.4.6. Game Objects 24](#_Toc405144564)

[4.4.7. Hero 25](#_Toc405144565)

[4.4.8. Castle 27](#_Toc405144566)

[4.4.9. Cannonball 27](#_Toc405144567)

[4.4.10. Wall 28](#_Toc405144568)

[4.4.11. Soldiers 29](#_Toc405144569)

[4.4.12. Mine 30](#_Toc405144570)

[4.4.13. Bonus 30](#_Toc405144571)

[4.4.14. Enemies Gone 31](#_Toc405144572)

[4.4.15. Bonus Manager 32](#_Toc405144573)

[4.4.16. Entity Manager 33](#_Toc405144574)

[4.4.17. Conquered Area Detector 35](#_Toc405144575)

[4.4.18. Collision Detector 36](#_Toc405144576)

[4.4.19. Graphic Engine 37](#_Toc405144577)

[4.4.20. Game Manager 39](#_Toc405144578)

[5. References 42](#_Toc405144579)

# INTRODUCTION

In this section of the report we will talk about the purpose of the system of our game, our goals of design and some definitions, acronyms and abbreviations we will use throughout the report.

## Purpose of the System

The game “Conquerors” is an arcade game which is inspired from the Atari game “*Volfied*”. The aim of this game is to conquer the castle by drawing lines before time runs out. During the game player will face with different enemies that will obstruct our hero to reach the goal. Unlike the original game, our game has historical and military elements in it. We aim to make the game more enjoyable than its ancestor. That’s why we added different bonuses to make the player’s job easier or harder. Our game has 5 different levels and as levels passed, completing the levels will be harder and the player will face with more enemies. With its user-friendly interface, the “Conquerors” will be easy to learn and play but hard to win.

## Design Goals

In order to make our game a high-quality one, we should set some design goals. We set those goals based on the non-functional requirements which are determined in the analysis phase.

### Goals

* ***Reliability:*** Making a reliable game is very important for us. We aim to create a game with no bugs. Moreover game will not crush unexpectedly. When an unknown input is given, it won’t stop working or close itself.
* ***Modifiability [1]:*** It is vital to create a system which can be modified easily. We may want to add new features to our game or remove some other. In order to reach that goal we will minimize the coupling of our subsystems. Furthermore using encapsulation and polymorphism in our design will make our system easier to modify.
* ***Adaptability:*** It is necessary for us to create a game which can be used in other platforms. We choose Java for writing the code of the game. Java provides us cross-platform portability. In every platform that JRE is installed, our game can be executed. That feature of Java made us choose it.
* ***Ease of Use:*** We aim to create a game which has a wide range of audience so *Conquerors* should be very easy to use. The interface of the game will be very friendly. Menu part will be not complex but informative. Moreover, game is only controlled by the arrow keys on the keyboard. Thus there will be no confusion for the player and it will be easier to play. However, those features will not make the game itself easy since we aim to create hard and fun game.
* ***Ease of Learning:*** The game will be very easy to learn. There will be a “View Help” part for the player. In there, the player can see what will create the movement of the Hero, and shows all the objects. The game logic will be very easy too.
* ***Response Time:*** This is a very important aspect for our system. Since we are planning to create an arcade game, the player would like to see the object move when s/he asks for it. If it does not happen, it will be very annoying. That problem will make people not to play our game. Thus we aim to make our game respond the actions of the player (nearly) immediately.

### Trade-Offs

* ***Ease of Use and Learning vs. Functionality:*** As we mentioned before, we want our game to have a wide range of audience. That’s why making the game easy to use and understand is a high priority for us. This fact means that we will does not put functionality of the system prior than ease of use and learning. Thus we won’t create complex functionalities. For instance the player can only use arrow keys to move the Hero. However if we wanted to make our game have more functionalities, we may had “A, W, S, D” too but this would made the use of the game harder.
* ***Performance vs. Memory:*** We believe that performance of a game is really important. We want our game to have smooth graphics and high response time. To reach our goal we need to make sacrifices of the memory. For instance we have a bonus which is Enemies Gone. When user got that bonus all of the enemies on the screen should be destroyed quickly. To make that happen fast we need to use a large memory to hold all of the enemies inside the game and destroying all together, other than removing them one by one.

## Definitions, acronyms, and abbreviations

### Definitions

* ***Coupling [2]:*** It is the degree to which software components are dependent upon each other.
* ***Cross-Platform [3]:*** It is an attribute conferred to computer software or computing methods and concepts that are implemented and inter-operate on multiple computer platforms.

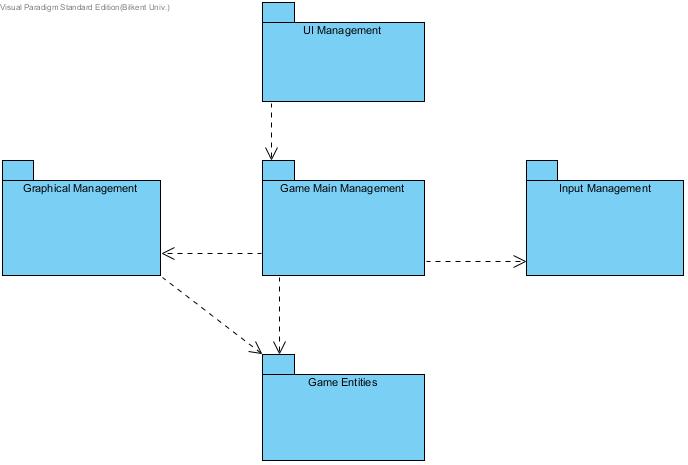
### Abbreviations

* ***JRE [4]:*** Java Runtime Environment

# SOFTWARE ARCHITECTURE

## Subsystem Decomposition

In this section, it is going to be clarified that how our system will be divided into subsystems as well as how they will be reacted each other according to the main structure of the game. We have decided to use a most common use of multi-tier architecture for our system decomposition which is three-tier architecture. This architecture is a well-established software design pattern and it provides a flexible and reusable model since presentation, application processing, and data management functions are separated. The below diagram illustrates the three-tier implementation of our system from an overview perspective.



**Image 1:** High-Level Demonstration of the Subsystem Decomposition

## Hardware / Software Mapping

Conquerors game will be implemented in Java programming language; therefore we will use latest JDK (8u25). As hardware configuration, user needs a basic keyboard to give input to the system. Because of Java implementation, system requirements will be minimal so any kinds of computer, which can compile java and run .java file, easily run conquerors.

Storage will base on .txt structures to high score list because of that, the operating system should support .txt file formats. On the other hand, this game will not have multiplayer mode so system does not need any network connection.

## Persistent Data Management

Conquerors game does not need complex database system. System will only store high score list as text files in disk. System will also store some pictures for game objects in hard disk drive.

## Access Control and Security

There is no requiring of any kind of network connection such as database system, Conquerors game can be played by anyone. Therefore there is no limited access or restriction for, controlling. The only access to the system is through Game Manager which is the main game logic class, for the security. Controlling images and objects, managing input and the game map is delegating by game logic.

## Boundary Conditions

### Initialization

Conquerors game does not require installing since it does not have regular .exe extension; instead game will be constituted as an executable .jar file.

### Termination

In any moment of the game user can terminate the game by simply pressing the “Exit” button on the JFrame.

### Error

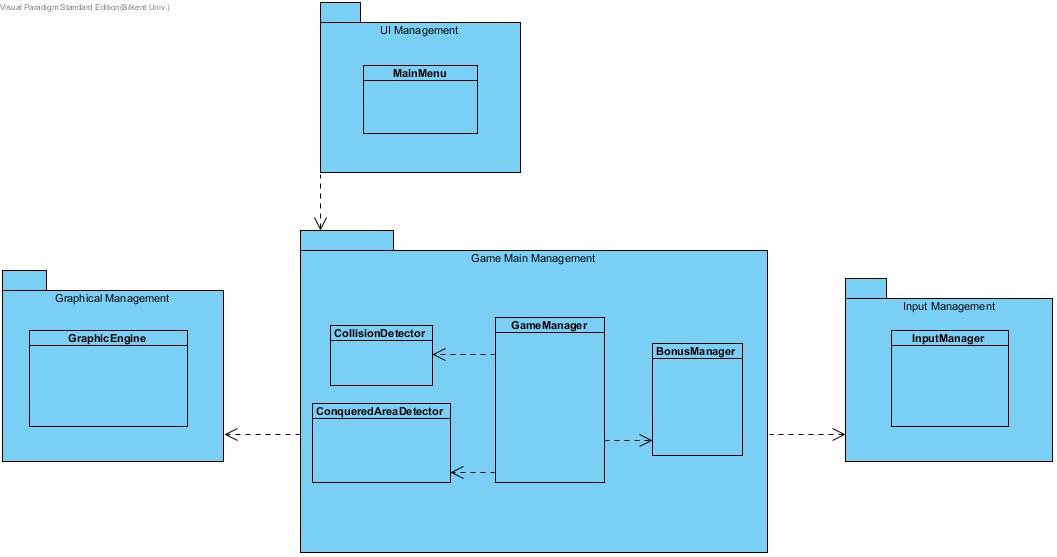
If the program does not respond because of a performance issue, player loses his current data because game data such as high scores or player’s current level are updated the end of the game.

### Game Updating

When the life of hero is finished, system automatically returns the main menu and in case of that, the high scores will be updated if a new record is broken and also level is updated.

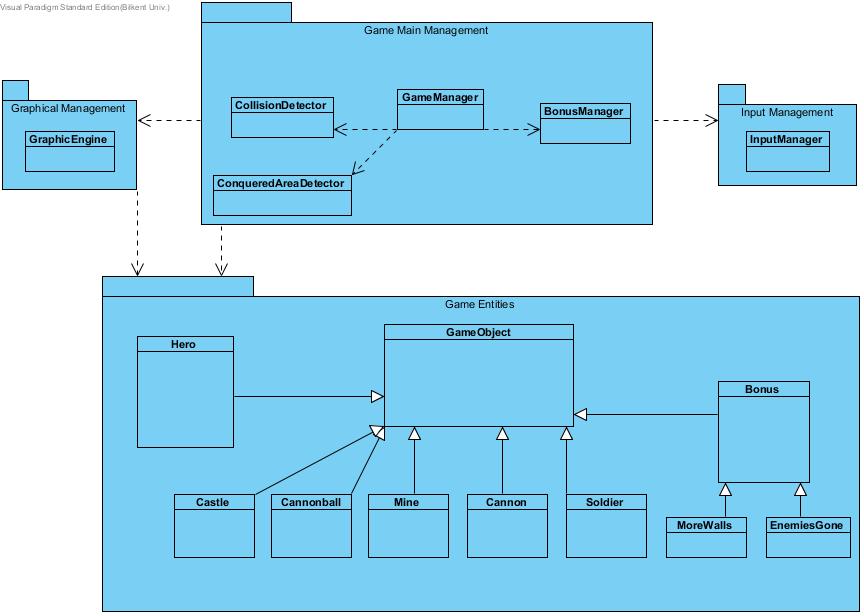
# SUBSYSTEM SERVICES

The first layer of the three-tier architecture is defined as presentation of the system so that interaction between the end users of the program and the system itself can be established. Therefore, in our game, players start the first interaction between the game layers with the help of the interface that is provided by the presentation layer. For that reason the “MainMenu” class is located in the UI Management package and with the methods of this class player choices can be transferred to the Game Main Management package which is stated in the application layer such as sound and hero type options or level selection. The diagram that states in the below demonstrates the connection between presentation and the application layers graphically.

**Image 2:** Illustration of the connection between presentation and application layers

The second layer is the application processing layer where the main logic of the program is managed. After an end user starts the interaction from presentation layer to the application layer, in other words when player passes information from the interface of UI Management package to the Game Main Management package, systems starts to run the logic of the game. Three packages of Graphical Management, Input Management and Game Main Management work together for the processing of game logic. Input Management package keeps track of the user inputs with the help of classes in it while game is running; Graphical Management package responsible from the rendering game objects and the game map, and Game Main Management package serves as a controller package that manages internal game events as well as has a connection with other two Graphical and Input Management packages in that same layer.

Last layer in three-tier architecture is the data management layer, thus in our design we have Game Entities package in that layer in order to control the manipulations on game objects. That’s why this data management layer interacts with the application layer. Game Main Management package passes information about changes in the properties of objects; Graphical Management package pursues the locations of game entities to re-draw them. The below package diagram depicts the interaction between layers of data management and the application.



**Image 3:** Illustration of the connection between data management and application layers

# Low-Level Design

## Object Design Trade-offs

At beginning of the low level design of the game “Conquerors”, we had to make important design decisions.

First of all it is decided that we did not need to create a map for our game. We only need to keep positions of objects and the background is stable. That’s why we decided only create a frame and inside it our objects will be located. Thus we do not create a map in our game.

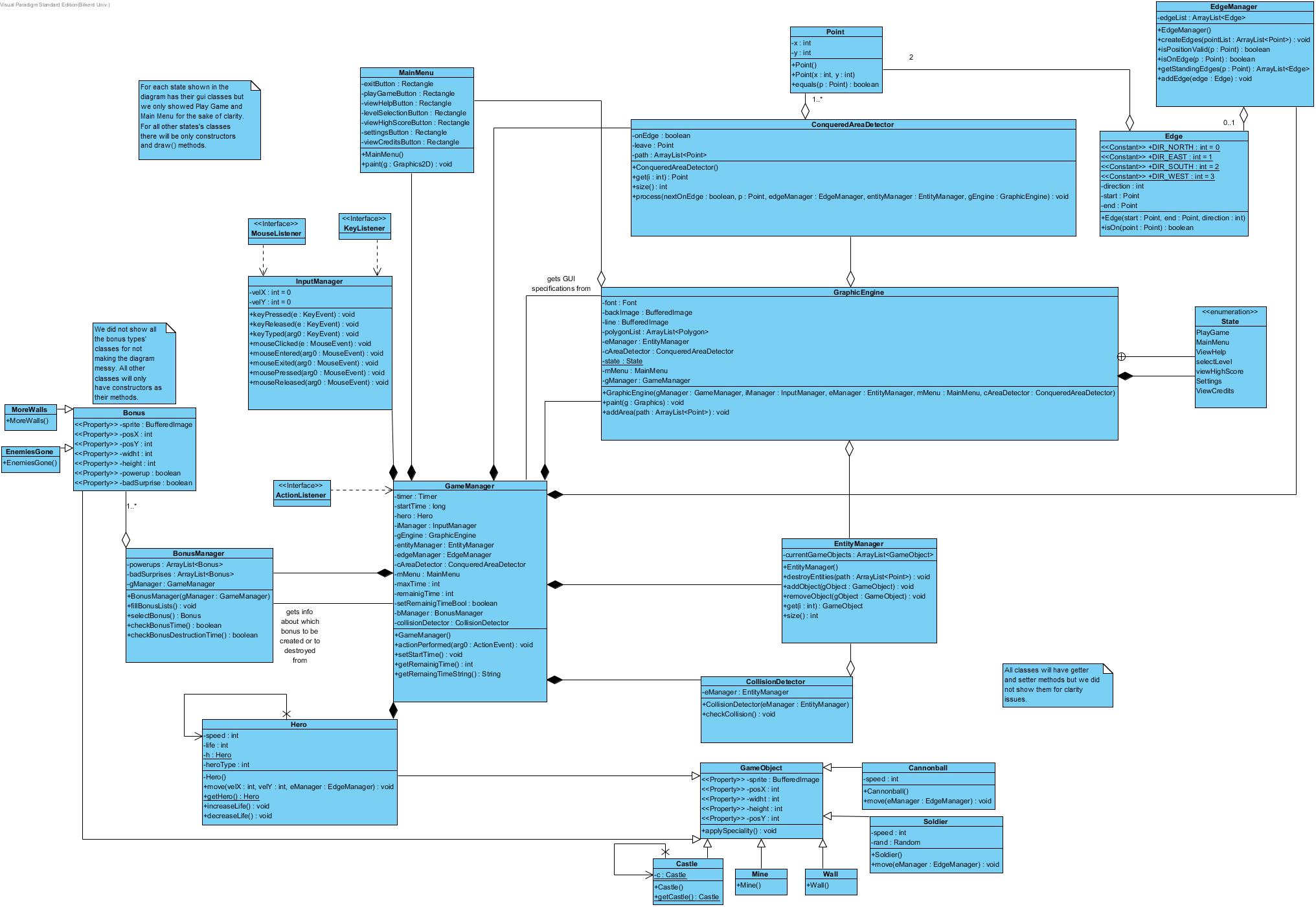
Secondly we decided not to update our game in a game loop. What we thought is we will use timer and at every 5 milliseconds the action performer will be called. Inside it will check all the necessities to keep playing the game. If there is anything wrong with those then it will call repaint() method which will call the paint() method which is overridden in the GraphicEngine class. This method will create all the objects which should be on the screen at the desired positions. Thus game will be updated. This was a major decision for us but we thought that it will be better in our game and this creates smoother graphics which was a requirement for the game.

Moreover we argued on which design patterns should be used in our game. First of all we decided to use *Façade pattern*. *Façade pattern* provides a unified interface to a set of interfaces in a subsystem [5]. It allows developers to manage a subsystem from a Façade class easily. Façade defines a higher level interface. This pattern is chosen since it provides maintainability and reusability since if a change is needed on the subsystem can be done by changing the necessary things in the Façade class. In our design the Façade class is GameManager class in the Game Main Management subsystem. This class manages all information flow between the classes. It manages whole game logic.

Other than Façade design pattern *singleton design pattern* is also chosen. This pattern is used when only one instance of an object should be created. The *hero* object our game has to be created only once, this is important for our game. Thus while creating this object singleton design pattern will be suitable. This pattern involves a class that is responsible for instantiate itself to make sure it creates not more than one instance; at the same time it provides a global point of access to that instance [6]. We need to create a private static attribute of Hero then make a private constructor and another method which is public. This method should be static too. In that method, the constructor will be called if a *Hero* object is not created before. We will use this method for the class Castle too since castle has to be only created once for each level.

## Final Object Design

At the next page the final object design is shown. We aim to make our design of the game more understandable with the help of this diagram. It will show the hierarchy of our classes and illustrate how classes interacts one another. It will also show the design patterns that we used.

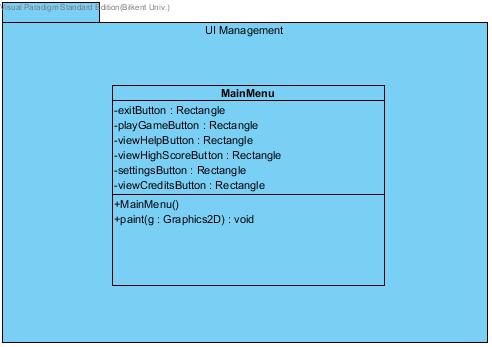


**Image 4:** Detailed Object Design

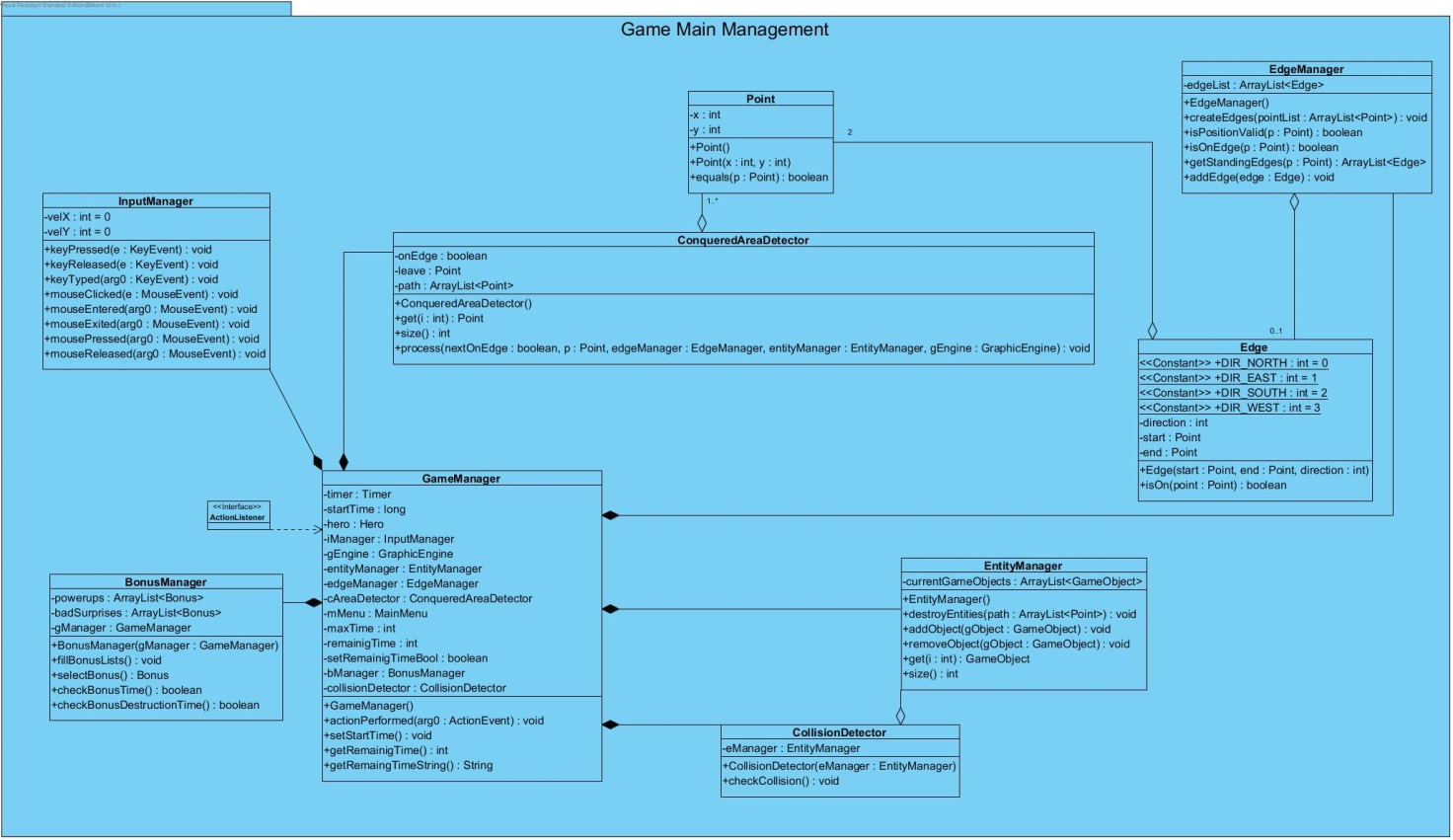
## Packages

In this section we will talk about the low level design of packages that are told before.

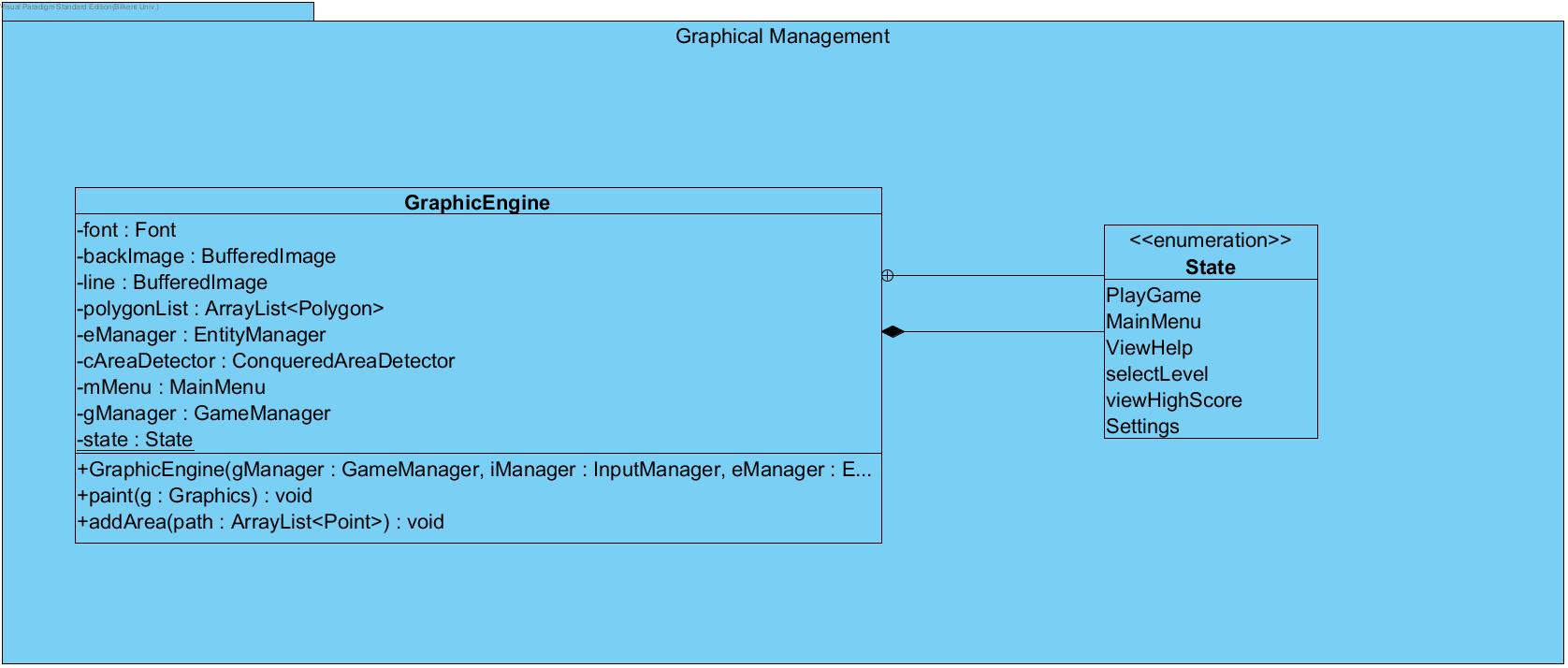
### UI Management Package



### Game Main Management Package



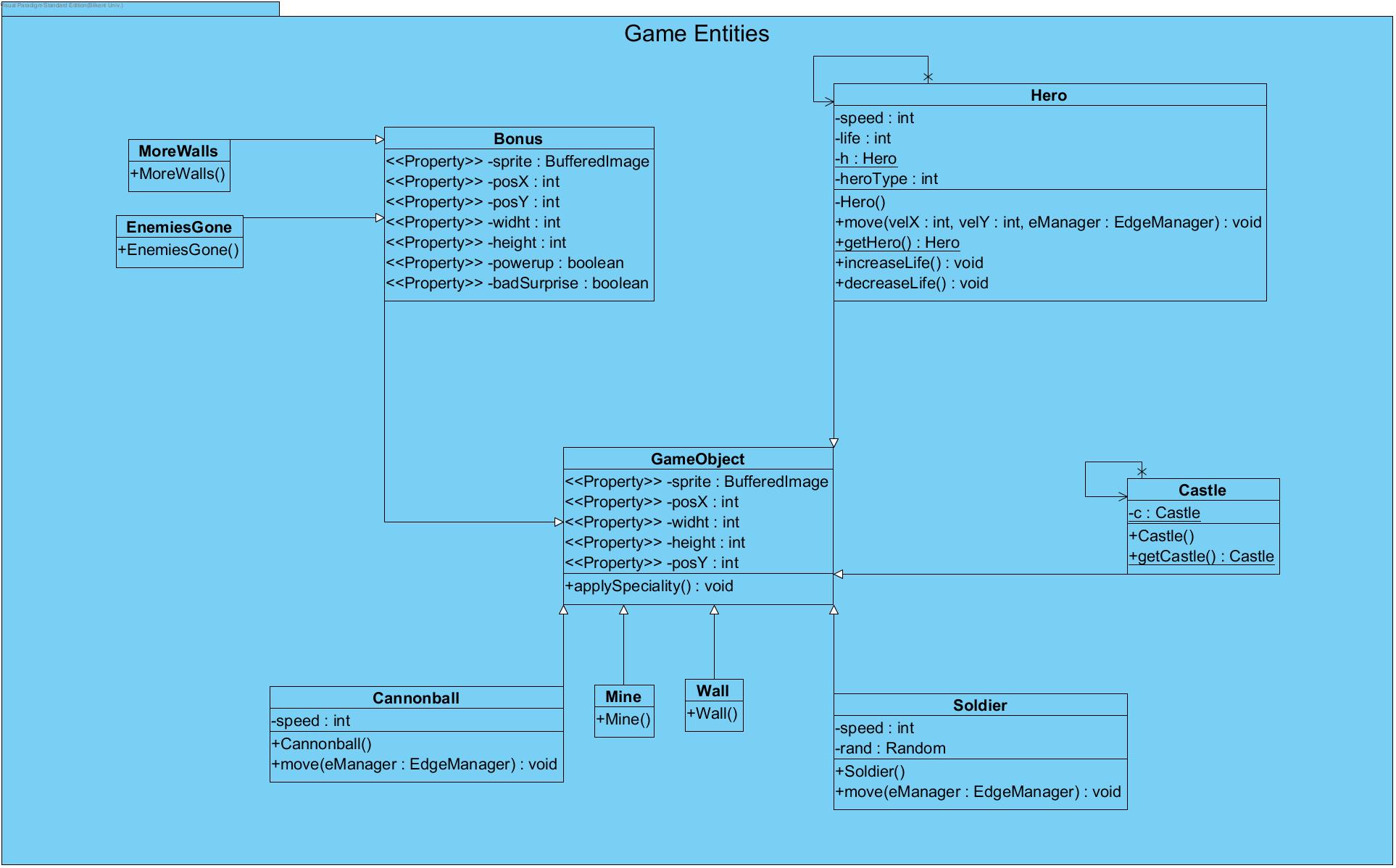
### Graphical Management Package



### 

### C:\Users\User\Desktop\package_diagram_final\Package Diagram7.jpgInput Management Package

### Game Entities Package



## Class Interfaces

In this section the information of class interfaces will be given. Class interfaces’ methods and attributes will be introduced and their tasks will be explained.

### Main Menu



**Image 5:** MainMenu Class

This class is there to create the main menu of our game. We decided not to use JButton. We create our buttons by rectangles and clicking somewhere in that area will be enough to go that section. The following class creates the interface of main menu of the game.

public class MainMenu {

//atributes

private Rectangle playButton;

private Rectangle viewHelp;

private Rectangle levelSelection;

private Rectangle highScore;

private Rectangle settings;

private Rectangle viewCredits;

private Rectangle exitButton;

//methods

/\* The method below is the constructor of our MainMenu class

\* In this constructor all rectangles for attributes above will

\* be created.

\*/

public MainMenu();

---------------------------------

/\* This method is for painting the buttons we want to create

\* The font and color will be chosen and rectangles are filled

\* with desired strings. Then draw() method of Java’s graphics

\* class is called for each attribute

\*/

public void paint(Graphics2D g);

}

### Input Manager



**Image 6:** InputManager Class

In this game player uses both mouse and keyboard. InputManager class will manage all the inputs from the player and informs the GameManager class which is the main controller of the game.

public class InputManager implements KeyListener, MouseListener{

//properties

private int velX = 0;

private int velY = 0;

//methods

/\* This method will look which key is pressed. If one of the

\*arrows are chosen then it will update the velX or velY

\* according to the movement’s direction.

\*/

public void keyPressed(KeyEvent e);

----------------------------------------

/\* It will override the keyReleased method of Java’s KeyEvent

\* It will make velX and velY’s values 0.

\*/

public void keyReleased(KeyEvent e);

----------------------------------------

public void keyTyped(KeyEvent arg0);

----------------------------------------

/\* It will check which button is chosen. Then according to the

\* chosen button it will set the state of the game.

\*/

public void mouseClicked(MouseEvent e);

----------------------------------------

public void mouseEntered(MouseEvent arg0);

----------------------------------------

public void mouseExited(MouseEvent arg0);

----------------------------------------

public void mousePressed(MouseEvent arg0);

----------------------------------------

public void mouseReleased(MouseEvent arg0);

}

### Point



**Image 7:** Point Class

We decided to create a Point class to keep positions of any point.

public class Point {

//properties

private int x;

private int y;

//methods

/\* The constructor of the class which takes no parameters. It

\* will set point x’s and y’s values as zero.

\*/

Point();

------------------------------------------------------

// The constructor of the class which takes int x and int y

Point(int x, int y);

------------------------------------------------------

// Checks if two points are at the same positions.

public boolean equals(Point p);

}

### Edge



**Image 8:** Edge Class

For our game *edges* are really important so we decided to make a class for them. This will check if the hero is on an edge or not and finds the starting and end points of the edge and shows which direction cannot be moved to.

public class Edge {

//properties

public static final int DIR\_NORTH = 0;

public static final int DIR\_EAST = 1;

public static final int DIR\_SOUTH = 2;

public static final int DIR\_WEST = 3;

private Point start, end;

private int direction;

//methods

// The constructer of the Edge class is given below.

public Edge(Point start, Point end, int direction);

---------------------------------------------

/\* This method will check whether the point is on an edge or

\* not

\*/

public boolean isOn(Point point);

}

### Edge Manager



**Image 9:** EdgeManager Class

This method will keep a list of edges in an array list. If a new area is conquered then the points of that area will be on the edge list because the points need to act like edges.

public class EdgeManager {

//properties

private ArrayList<Edge> edgeList; //the array list which

// is mentioned above

//methods

/\* This is the constructor of the EdgeManager. It will

\* create a list which has the 4 edge points of the frame

\*/

public EdgeManager();

-------------------------------------------------

/\* This method will turn the pointList instances into

\* edges and add them into edge list.

\*/

public void createEdges(ArrayList<Point> pointList);

---------------------------------------------------

/\*It will check if the movement of the point is to \*

\* somewhere in the conquered area or a free space. With

\* this method we will prevent the point to move inside

\* the conquered area.

\*/

public boolean isPositionValid(Point p);

----------------------------------------------------

/\* It checks if the point is on edge by looking at the

\* edge list.

\*/

public boolean isOnEdge(Point p);

-----------------------------------------------------

// It checks on which edges the point p stands

public ArrayList<Edge> getStandingEdges(Point p);

------------------------------------------------

// It adds an edge to the list

public void addEdge(Edge edge);

}

### Game Objects



**Image 10:** GameObject Class

We created a parent class for all game objects which is called GameObjects. It keeps common information for all child game objects.

public class GameObject {

//properties

private BufferedImage sprite;

private int posX;

private int posY;

private int widht;

private int height;

}

### Hero



**Image 11:** Hero Class

This is a game object so this class will *extends* GameObject class. This will control the hero. It will control its speed, life and its look (since we have three types of hero).

public class Hero extends GameObject{

//properties

private int speed;

private int life;

private static Hero h;//for the sake of singleton pattern

//methods

/\* The constructor of the class. It will set the sprite,

\* speed, its width, height and position and its life(life \* will be one initially). It is private

\* for the sake of singleton pattern.

\*/

private Hero();

------------------------------------------------------

/\* In this method if a hero is not created before new

\* *hero* is created else the one which exists is return. It

\* is the last step of the singleton pattern.

\*/

public static Hero getHero();

-----------------------------------------------------

/\* It will manage the movement of the hero. While doing

\* that, it will use an instance of edge manager to check

\* if it is on the edge or not or to check if it can move

\* or not

\*/

public void move(int velX, int velY, EdgeManager

eManager);

-------------------------------------------------------

// The life attribute will be increased by one.

public void increaseLife();

--------------------------------------------------------

/\*If life is more than zero the life will be decreased by \* one

\*/

public void decreaseLife();

}

### Castle



**Image 12:** Castle Class

This is a game object so this class will *extends* GameObject class. This will create a castle, it is a non-motile object.

public class Castle extends GameObject{

/\* This is the constructor of the castle. It will set its

\* position.

\*/

public Castle();

}

### Cannonball



**Image 13:** Cannonball Class

This is a game object so this class will *extends* GameObject class. This will create a cannonball which has speed and moves.

public class Cannonball extends GameObject {

//properties

private int speed;

//methods

/\*This is the constructer of the class. This will set the speed

\* and its position.

\*/

public Cannonball();

--------------------------------------------------

// This will manage then movement of the cannonball.

public void move(EdgeManager eManager);

-------------------------------------------------

}

### Wall



**Image 14:** Wall Class

This is a game object so this class will *extends* GameObject class. This will create a wall instance which is non-motile.

public class Wall extends GameObject {

//methods

/\* This is the constructor of the wall class and sets its

\* position.

\*/

public Wall();

}

### Soldiers



**Image 15:** Soldier Class

This is a game object so this class will *extends* GameObject class. This will move the soldiers randomly and sets its speed.

public class Soldier extends GameObject{

//properties

private int speed;

private Random rand;

//methods

/\*This is the constructor of the soldier which sets its speed

\* and new Random()

\*/

public Soldier()

-----------------------------------------------------------

/\* This will control the movement of the soldier and it will

\* setits position randomly

\*/

public void move(EdgeManager eManager);

}

### Mine



**Image 16:** Mine Class

This is a game object so this class will *extends* GameObject class. This will be non-motile object.

public class Mine extends GameObject {

//methods

// The constructor of the mine class

public Mine()

}

### Bonus



**Image 17:** Bonus Class

This is a game object so this class will *extends* GameObject class. This is also a parent of the special bonus classes. It will keep the common properties of each bonus.

public class Bonus {

//properties

private BufferedImage sprite;

private int posX;

private int posY;

private int widht;

private int height;

private boolean powerup;

private boolean badSurprise;

//methods

// This is the constructor of the Bonus

public Bonus();

----------------------------------------

// Returns if bonus is a power-up or not

public boolean isPowerup();

----------------------------------------

// Returns if bonus is a bad-surprise or not

public boolean isBadSurprise();

}

### Enemies Gone



**Image 18:** EnemiesGone Class

This is a bonus object so it will *extends* the Bonus class. We did not write all of the bonus classes, we choose this to show the design since they all only have constructor as their methods.

public class EnemiesGone extends Bonus{

/\* This is the constructor of the class. If this bonus is

\* chosen all enemies will be deleted on the screen.

\*/

public EnemiesGone();

}

### Bonus Manager



**Image 19:** BonusManager Class

This class will manage the bonuses in the game. This will keep two array lists; one for power-ups and one for bad surprises. According to levels it will fill those lists. It will also handle the bonus selection for the game.

public class BonusManager {

//properties

private ArrayList<Bonus> powerups;

private ArrayList<Bonus> badSurprises;

private GameManager gManager;

//methods

/\* This is the constructor of the BonusManager class.

\* This will create instances of powerups and bad-surprises and

\* instantiate gManager and finally call the method

\* fillBonusList().

\*/

public BonusManager(GameManager gManager);

----------------------------------------------

/\* This method will randomly create and add bonus objects to

\* the power-ups and bad-Surprises ArrayLists.

\*/

public void fillBonusLists();

-----------------------------------------------

/\* This method randomly selects and returns a bonus object from

\* the powerups and badSurprises ArrayLists.

\*/

public Bonus selectBonus();

-----------------------------------------------

/\* This method will check the time of the creation of the

\* bonus on the screen. It informs the GameManager class

\*/

public boolean checkBonusTime();

------------------------------------------------

/\* This method will check the time of the destruction of the

\* bonus on the screen. It informs the GameManager class.

\*/

public boolean checkBonusDestructionTime();

}

### Entity Manager

****

**Image 20:** EntityManager Class

We decided that we need to have a class that will keep the list of current game objects.

public class EntityManager {

//properties

private ArrayList<GameObject> currentGameObjects;

//methods

/\* This is the constructor of the EntityManager class.

\* it will create a new list of game objects.

\*/

public EntityManager();

-------------------------------------------------------

/\* This will destroy all enemy entities that are in the

\* area under the points in the path ArrayList

\*/

public void destroyEntities(ArrayList<Point> path);

-------------------------------------------------------

/\*This will add the game object (gObject) into the list

\*/

public void addObject(GameObject gObject);

--------------------------------------------------------

/\* This will remove the game object (gObject) from the

\* list

\*/

public void removeObject(GameObject gObject)

-------------------------------------------------------

/\* This will get the object on the given index from the

\* currentGameObjects list.

\*/

public GameObject get(int i)

--------------------------------------------------------

/\* This will return the size of the currentGameObjects

\* list.

\*/

public int size()

}

### Conquered Area Detector



**Image 21:** ConqueredAreaDetector Class

This class is responsible for checking if an area is conquered or not.

public class ConqueredAreaDetector{

//properties

private Point leave;

private boolean onEdge;

private ArrayList<Point> path;

//methods

/\* This is the constructor of the class. The leave will be set

\* as Point(-1,-1), it will create a new path and onEdge will be \* set as true.

\*/

public ConqueredAreaDetector();

--------------------------------------------------

//This will get the point on given index from the path list.

public Point get(int i)

---------------------------------------------------

// This will return the size of the path array list.

public int size()

---------------------------------------------------

/\* This is the most important method of this class. Since this \* method checks whether an area is conquered or not before the \* every call of the repaint method of GraphicEngine in the \*GameManager. All of the parameters are given above will be \*needed to write this method. It’s main logic is not very \*complex, it firstly checks whether hero’s current position is \*on the one of created edges or not. Then it does specific \*operations according to four possible situations that; \*currently standing on an edge with not standing in the \*previous position or standing in the previous position, and \*currently not standing on an edge with not standing in the \*previous position or standing in the previous position. \*Simply it holds the hero’s positions as a path of Point \*instances and if a conquering occurs, this method invokes \*EdgeManager to create necessary Edge instances according to \*the path, calls EntityManager to destroy game objects that are \*standing on this conquered area and lastly informs \*GraphicEngine with this path in order to add this conquered \*area into its list.

\*/

public void process(boolean nextOnEdge, Point p, EdgeManager edgeManager, EntityManager entityManager, GraphicEngine gEngine);

}

### Collision Detector

****

**Image 22:** CollisionDetector Class

This will check if a collision between the hero and another object occurred or not. We need this since collision of hero and another object will decrease hero’s life.

public class CollisionDetector {

// properties

private EntityManager eManager;

// methods

// The constructor of the class

public CollisionDetector(EntityManager eManager)

-------------------------------------------------

/\* Checks the hero position and detects whether any other game

\* object collides with hero according to their current

\* positions and height/width values. If there is a collision

\* it will decrease life

\*/

public void checkCollision();

}

### Graphic Engine



**Image 23:** GraphicEngine Class

This engine will provide a GUI specification to GameManager class. This is a very important class for our game. It will handle GUI specifications. This class needs to *extend* JPanel since it creates the game panel.

public class GraphicEngine extends JPanel{

//properties

private Font font;

private BufferedImage backImage;

private BufferedImage line;

private EntityManager eManager;

private ConqueredAreaDetector cAreaDetector;

public static enum State {PlayGame, MainMenu, ViewHelp,

selectLevel, viewHighScore, Settings, ViewCredits};/\*this

\* is needed since there are different

\* states of the game

\*/

private static State state;

private MainMenu mMenu;

private ArrayList<Polygon> polygonList; /\*the conquered are

\* will be drawn as

\* polygons

\*/

private GameManager gManager;

//methods

/\* This is the constructor of the GraphicEngine class.

\* This will create a background image and instantiate

\* all the parameters given. Moreover it will create a

\* frame of the game.

\*/

public GraphicEngine(GameManager gManager, InputManager

iManager, EntityManager eManager, MainMenu mMenu,

ConqueredAreaDetector cAreaDetector);

----------------------------------------------------------

/\* This will override the paint class of the Java’s Graphics

\* class. It will draw the graphics according to given state.

\*/

public void paint(Graphics g);

----------------------------------------------------------

/\* This method is created for adding areas as polygons when

\* an area is conquered. This will traverse the path and

\* two paths will be found after that smallest path will be

\* chosen since this is the drawn path. Then that path create a

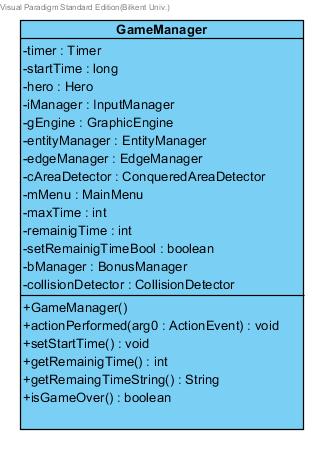
\* polygon and added to the polygon ArrayList

\*/

public void addArea(ArrayList<Point> path);

}

### Game Manager



**Image 24:** GameManager Class

This is the Façade class of our game. It will control the main logic of the game. It is responsible for updating the game, setting the timer and managing communication of other classes. This will implements ActionListener which is for updating the game.

public class GameManager implements ActionListener{

//properties

private int maxTime;

private Hero hero;

private InputManager iManager;

private GraphicEngine gEngine;

private EntityManager entityManager;

private EdgeManager edgeManager;

private ConqueredAreaDetector cAreaDetector;

private BonusManager bManager;

private CollisionDetector collisionDetector;

private MainMenu mMenu;

private Timer timer;

private long startTime;

private int remainigTime;

private boolean setRemainigTimeBool;

//methods

/\* This is the constructor class. It will instantiate the

\* properties given above and starts the timer.

\*/

public GameManager();

----------------------------------------------------------

/\* If startTime is -1 then setStartTime() will be called. Else

\* it will check the game is running correctly and all objects

\* are in the states that they should be at. If everything is

\* correct and life of the hero is bigger than zero then it will

\* call gEngine.repaint()and update the time. It will do it

\* every 5 milliseconds.

\* This will provide a smooth graphic.

\*/

public void actionPerformed(ActionEvent arg0);

------------------------------------------------------------

// This method will set startTime as the currentTimeMillis()

public void setStartTime();

------------------------------------------------------------

//This will return the remaining time of the game.

public int getRemainigTime();

-----------------------------------------------------------

//This method will convert the time into string.

public String getRemaingTimeString()

----------------------------------------------------------

//Checks whether game is over or not

public boolean isGameOver();

}

# References

[1] <http://www.sei.cmu.edu/reports/07tr002.pdf>

[2] <http://whatis.techtarget.com/definition/coupling>

[3] <http://en.wikipedia.org/wiki/Cross-platform>

[4] <http://www.oracle.com/technetwork/java/javase/downloads/jre7-downloads-1880261.html>

[5] <http://www.dofactory.com/net/facade-design-pattern>

[6] <http://www.oodesign.com/singleton-pattern.html>