Bilkent University

**Object Oriented Design**

IMPRISONMENT

Design Report

İdil Ağabeyoğlu 21300761 Evren Ergen 21300781 Ismail Kerimov 21300355 Sinan Öndül 21200962

Progress Report

March 27,2016

This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfilment of the requirements of the Object Oriented Design course CS319/1.

[1. Introduction](#h.e297dzk0ti5n)

[1.1 Purpose of the system](#h.30j0zll)

[1.2 Design Goals](#h.3znysh7)

[1.3. Overview](#h.3dy6vkm)

[2. Software Architecture](#h.752h0o9qemql)

[2.1. Overview](#h.xove84n5p4bp)

[2.2. Subsystem Decomposition](#h.7glsfpr7jl41)

[2.2. Architectural Style](#h.1ujrivjgkutj)

[2.2.1. Layers](#h.tceh3l64m7dq)

[2.2.2. Model View Controller](#h.iilynb7fu77a)

[2.3. Hardware / Software Mapping](#h.995bt8sk2ndq)

[2.4. Persistent Data Management](#h.bgv3rs74qwhk)

[2.5. Access Control and Security](#h.7wzzhswvtm55)

[2.6. Boundary Conditions](#h.qme6esaj7n0z)

[3. Subsystem Services](#h.ueopkdsn2lkg)

[3.1 Design Patterns](#h.57zxbn5oc632)

[3.2 User Interface Subsystem Interface](#h.4z2flidaycz1)

[3.3 Game Control Subsystem Interface](#h.mdakokabo9lv)

[Constructors:](#h.cnulmlk9brc2)

[Methods:](#h.ge2tqxbdnhaf)

[3.4 Game Objects Subsystem Interface](#h.kqq6eodwyzun)

[4.1 Object Design trade-offs](#h.g40jsxjupdh2)

[4.2 Final object design](#h.5rusgwi5ncx8)

[4.2.1 User Interface Management Interface](#h.dl4nf94vdy0z)

[4.2.2 Game Logic Subsystem Interface](#h.f3dx7jb9o0vp)

[4.2.3 Input Manager Subsystem Interface](#h.i7c3wqw05k5w)

[4.2.4 Game Elements Subsystem Interface](#h.wbmkl4amfq0b)

[4.3 Packages](#h.3eulktv8w8z2)

[4.4 Interfaces](#h.r25qh0kiik5y)

[5 Conclusion](#h.a6lsxec9zbzd)

# 1. Introduction

## 1.1 Purpose of the system

The game of “Imprisonment” is an action game that aims to provide the users a well designed platform and a friendly user interface for an easy comprehension. In order to make the game riveting as much as possible, the game designed as a very challenging way with different level options. New features such as bonuses, maps and sound effects, distinguished the Imprisonment from other games and also from its inspiration Volfied. The last but not least, although the fundamental reason behind the game is to enable the users to have good time, it also improves reflexes and hand-coordination.

## 1.2 Design Goals

1. Adaptability: Java® with its cross-platform(can run in the same way in

different platforms such as Windows, Apple, Linux) portability makes our system work in all JRE(Java Runtime Environment) installed platform and with that feature it eliminates the worries of the user related with operating system requirements one of the few programming languages which provide cross-platform portability.

1. Efficiency: In order to make game more efficient, we cut down the response

time. Although the high performance is not obligatory in order to make a difference in the platform and make the game distinguishable, we implement our codes with high coherence, low coupling and make running in for at least 40 fps(frames per second).

1. Reliability: Since reliability is one of the most essential designed goals, it

should be also accomplished by Imprisonment. It requires strong implementation, testing and consistency among the boundary conditions(can cause runtime errors). The system should not the user to have the fear of crush with unexpected inputs by evaluating the boundary conditions in very detail aspects having the various testing procedures.

1. Usability: By making the game in a way that requires no prior knowledge for

the players, the interface of the game becomes easy to understand and user friendly and appeals to all kinds of users from a newbie to experts at the game. One of the main targets of the game is to make the user feel comfortable for the continuity of playing by not making the game easier but easy to understand.

1. Extensibility: Game's object oriented structure makes system customizations

easier by not causing bugs and not making mandatory to modify the unrelated classes. To keep the excitement alive for the users and to be the persistent, our game will be extendable for the new features such as new bonuses and new levels.

**Tradeoffs**

Efficiency – Reusability: Since our fundamental goal is not to insert our classes or our implementation to other similar systems or games, the classes are designed according for the tasks regarding their specifications. Realization of this distinction enables our implementation to stay simple and increases our attention and focus on efficiency.

Functionality – Usability: In order to make the game more user friendly, easier to learn and to have a wide range of customers, it is not unusual to give up some advanced features by proving a plain and not too complicated usage to users to have wide range of customers. Since our ultimate goal is to entertain the users and fill their spare times with a fruitness way, functionality does not have more than it needed unlike usability. In order not to torture or make the users struggle, we keep our interface and instructions in a basic level.

Space – Speed: In order to make our game more efficient and fast, we create objects separately, sacrifice the used memory spaces which increases the speed of some operations such as the detection of collisions.

## 1.3. Overview

In this section, we illustrated the designing goals of our system, with having the most essential one as entertaining the player as much as possible. They are determined according to their adaptability, efficiency, reliability, usability, extensibility and through them we made some trade-offs in order to accomplish our goals. By sacrifying from functionality we intended to make our game simpler and easily comprehensible, also sacrifying from memory we targeted to gain performance by providing a game with smooth animations and effects. 

# 2. Software Architecture

## 2.1. Overview

The main part of this section is to Show the process of decomposing our system into easily-maintainable subsystems. Within this process, the main aim is getting rid of coupling between subsystems while make the subsystems interact with each other properly. To achieve this goal, we decided that Model View Controller design architectural style would be most beneficial style to decompose the system into subsystems.

## 2.2. Subsystem Decomposition

This section describes that the overall system is divided into rational independent parts to make organization easier. Since the software system’s performance, maintainability and customizability is affected significantly by identifying the subsystems properly, decomposition of these subsystems is crucial in terms of meeting the non-functional requirements correctly and producing a high quality software. As it is seen, in Figure 1 the overall system is separated into three subsystems to meet the MVC style’s requirements. The names of the subsystems are User Interface, Game Control and Game Objects, relatively. Within the Figure 2, Game Control and Game Objects subsystems are slightly coupled. Furthermore, the interaction between User Interface subsystem and Game Object subsystem is provided through Game Control subsystem. This implies that any modification or error made in User Interface subsystem does not affect Game Object subsystem. Only the Game Control subsystem might be affected throughout the modifications or errors.

The classes are put into the subsystems so that the classes work on similar properties and tasks are in the same subsystems to achieve the common purposes. The Manager classes is put into the Game Control subsystem since the control of the whole game is maintained by those manager classes.

The flexibility and coherency is the main goal to decompose our system into three subsystems by following MVC design pattern. Thus, this will provide further changes and customizations on our software easily.

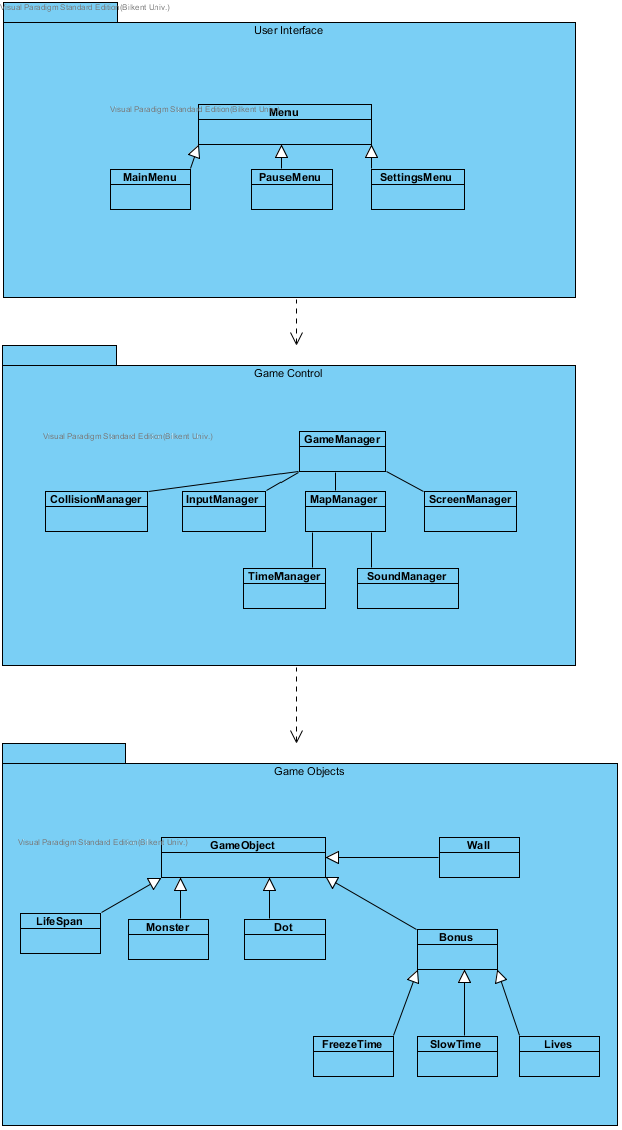
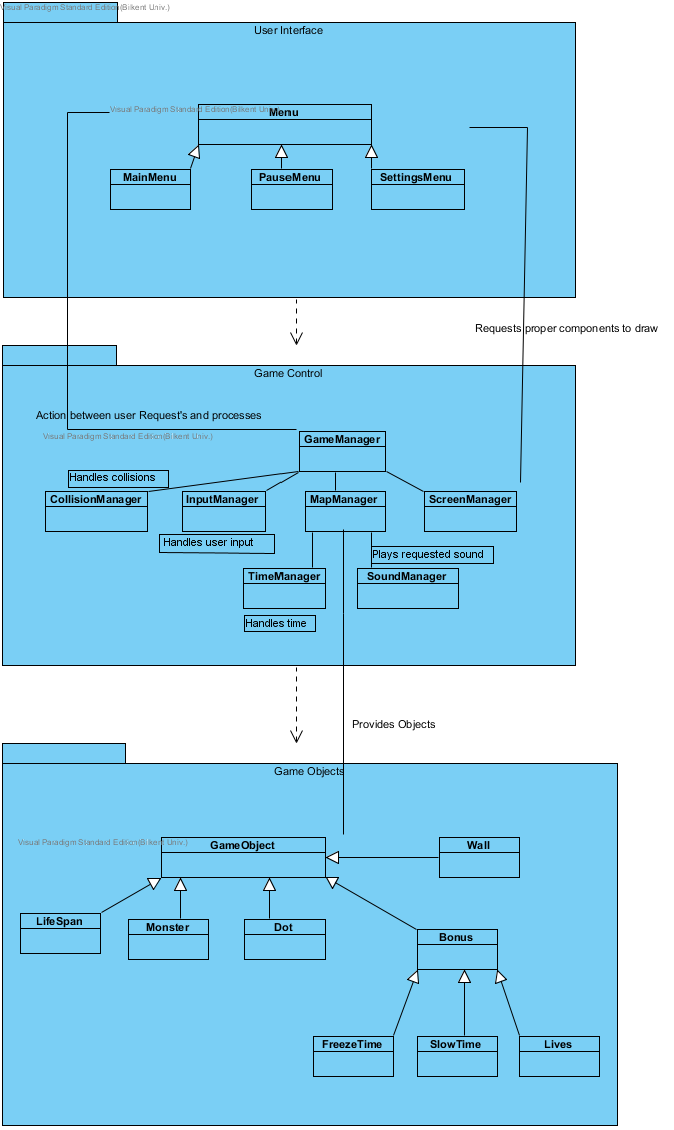


Figure 1(Basic Subsystem Decomposition)

 Figure 2 (Detailed Subsystem Composition)

## 2.2. Architectural Style

### 2.2.1. Layers

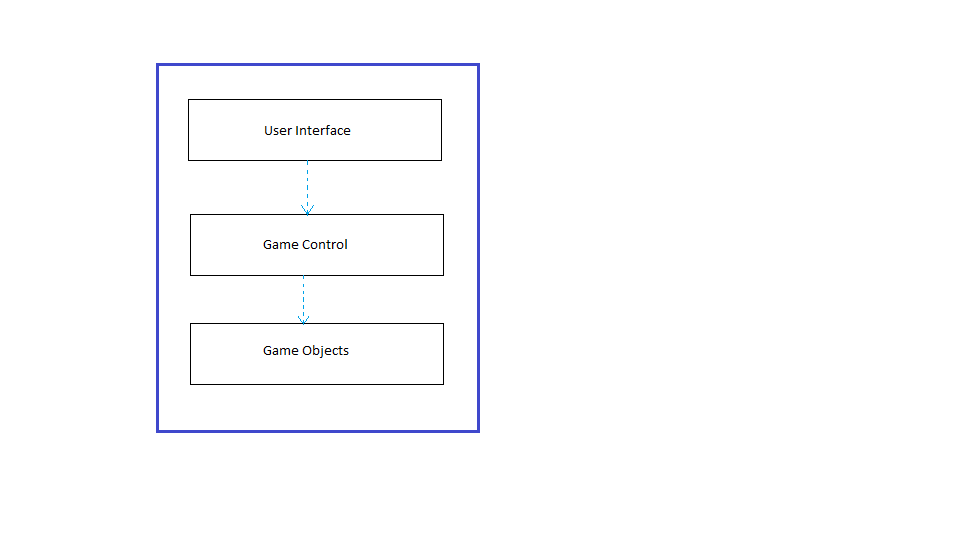
The system is decomposed into three layers, namely User Interface, Game Control and Game Objects. These layers is divided as the subsystems divided so that they provide related tasks. The top layer which has the highest hierarchy is User Interface layer, since it is not used by any subsystems above and it is only responsible to interact with the player. The second and middle layer is Game Control layer. In this layer, the game’s whole control is managed. The last layer is Game Objects layer, in which all gameobject needed are put. The layer decomposition provides the closed architectural style, means a layer can only interact with the layer below it. The layer decomposition can be seen in Figure 3.

Figure 3(Layers of the System)

### 2.2.2. Model View Controller

This MVC design pattern’s main goal is classifying the system into the three

subsystems, called model, view and controller, respectively. Separating the domain from the user interface and controller part is the essential part of the pattern. In our system, domain objects grouped into game objects, which is the model of the system. The domain objects can only be managed by manager classes that are grouped into Game Object subsystem that stands for the controller part. The interaction between the user and the software is provided by menus that are grouped into the User Interface subsystem, which corresponds to the view part. The model part communicates with the controller part and software works through the interaction between model, controller and view parts. By this architectural pattern, any changes on interface of the software does not affect the model of the system. Thus, it provides maintainability and customizability of the software.

## 2.3. Hardware / Software Mapping

Imprisonment will be implemented in the Java programming language.

Therefore, the latest Java Development Kit, JDK 1.8, will be used throughout the implementation. The hardware requirements for Imprisonment are a standart keyboard to play the game and a mouse for to push the buttons on various menus. Since the system requirements for the Imprisonment demands the minimal hardware, a basic computer with needed softwares to run the computer and JDK1.8 would be enough. Further, since the Java provides platform independency, there is no specific operating system to run the software. For storage, the system needs a memory for sounds and the software. Furthermore, Imprisonment does not need internet connection to play the game.

## 2.4. Persistent Data Management

In Imprisonment, there is not any usage of a database or complicated data structures. Imprisonment has the map structure and sounds in the game folder. Thus, any corruption on the game folder might affect the gameplay such as unloaded maps and objects.

## 2.5. Access Control and Security

Since there is not internet connection required for the software, Imprisonment, players are able to play the game after the executing the software. Thus, there will not be access preventions for the game access. Since the game is out of internet and single player, there is not security issues about the game Imprisonment.

## 2.6. Boundary Conditions

The game will be executed through an executable file, so the player does not have to do any installation. The game is over if there are no lives left, and finite number of levels that a player can play. The game can be closed via Exit Game, however the player can terminate the game through the “x” button at the upper right corner of the game screen. If the player terminates the game with “x” button, all of the current data will be lost.

If any folder or file is corrupted in the game folder, game is probably will not work properly, or even not open through the executable.

# 3. Subsystem Services

This section will illustrate the interface of the subsystems that was required in the project.

## 3.1 Design Patterns

**Façade Design Pattern:**

Façade design pattern will be used in the construction of the program due to the fact that this class will give us access to communicate with the outside of the system. We will use Facade pattern in order to reduce complexities in subsystems such as Game Management and Game Object. Moreover, this pattern will give us flexibilities such as reusability, maintainability and extendibility. In the Game Management subsystem GameManager will be the façade class since it is the communication access for the other subsystems and GameObject Class as shown above is the façade class for the communication with the other systems.

## 3.2 User Interface Subsystem Interface

User Interface Subsystem is the graphical part of the whole system. Beside its own properties this part will illustrate all the inner part of the overall system.

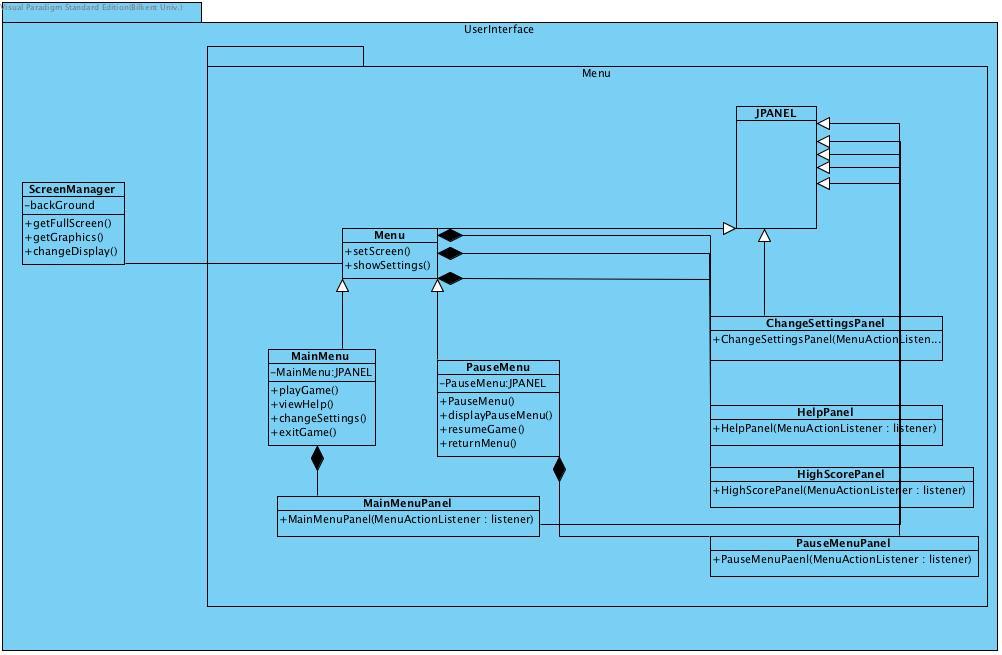


Figure 4

## 3.3 Game Control Subsystem Interface

This Subsystem holds our controller objects and manage the game dynamics and the game’s logic. We have 4 controller and 2 property classes which is a total 6 components of the subsystem. As illustrated in the figure we have SoundManager, GameManager, MapManager and GameManager’s subclass InputManager which handles the user input by mouse and keyboard. The 2 property classes are GameInformation and Settings. These classes will be explained in detail in this section.

**GameManager Class**

This class is Façade class of GameManagement subsystem. This class runs the game loop according to the actions of the player and performs assigned operations received by User Interface Subsystem. This class also implements an interface since the game loop will continue.

**Attributes:**

**private Boolean statuspause:** this is the status attribute for the pause option in the game loop.

**private Settings settingsmenu:** this attribute saves the settings of the game

**private gameMapManager mManager:** this attribute associates proper methods of GameManager with GameMapManager.

**Constructors:**

**public GameManager():** initializes the attributes of the GameManager on the first run.

**Methods:**

**public void gameLoop():** this is the main loop that continues to run as long as the game continues.

**public void applySettings( Settings settings ):** applies the given settings to system.

**public boolean success():** communicates with GameMapManager class by mapManager attribute checks the conditions if the player collided with a monster or %80 of the map is occupied.

**public boolean hasLive():** checks whether the number of lives bigger than zero, to check this communicates with GameMapManager class by mapManager attribute.

**public void startGame():** starts a new game, by resetting game information stored on GameMapManager class, obviously it is made by mapManager attribute

**public void pauseGame():** when called causes to prevent the game loop to iterate by setting paused attribute to false.

**public void resumeGame():** when called causes the game loop to continue to iterate, by setting the paused attribute to true.

**public boolean isHighScore( score: int):** returns true if the given score is eligible for the high score list else it returns false.

**public void endGame():** if the success method and the hasLive method returns true, this method invokes the isHighScore method.

**SoundManager Class**

* This class is referenced by GameMapManager when the system needs to play a sound.

**Attributes**

**private boolean soundOn:** determines sound enabled or disabled.

**Constructors:**

**public SoundManager():**  initializes the object of this class, soundOn is set true as default.

**Methods:**

**public void playSound(int soundID) :**  this method is called by GameMapManager class when needed. This method plays a sound sample according to given value.

**Settings Class**

This class is used by GameManager class. It holds default values.

**GameInformation Class**

This class holds the default values of the game.

**GameMapManager Class**

**Attributes:**

**private GameMap currentMap:** this class performs the map specific operations.

**private int dotState:** this class performs operations on Dot’s states.

**private GameInformation info:** this attribute holds the current game information, current difficulty, number of lives. These informations will be updated according to the situation of the game.

**private SoundManager soundManager:**  reference to SoundManager class to play proper sounds when needed.

#### Constructors:

**public GameMapManager():** initializes a GameMapManager object with default values.

#### Methods:

**public GameMap getCurrentMap():** returns the current map which is processed by GameMapManager class.

**public drawCurrentMap(Graphics g, int screenWidth,int screenHeigh,GameInformation info):** draws the current map according to given attributes. The attributes will be default for all three difficulties.

**public void applyPowerUp(int powerUpID):**  applies the powerUp to the game by referencing the currentMap attribute.

**public void handleWallCollisions() :** by referencing the currentMap object, this method finds and handles the collisions between dot and wall objects of currentMap object.

**public void handleMonsterCollisions() :** by referencing the currentMap object, this method finds and handles the collisions between dot and monster objects of currentMap object.

**public void handlePowerUpCollisions() :** by referencing the currentMap object, this method finds and handles the collisions between dot and power up objects of currentMap object.

**public void checkAndHandleCollisions():** this method invokes the methods of this class, which are designed to handle the collision issues, and finds and handles the collisions on the currentMap by one invocation.

**public void updateDotLocation(int posX, int posY):**  updates the location of Dot on gameMap, since dot is controlled by keyboard actions. Also this method is invoked by GameManager when the keyboard inputs.

## **3.4 Game Objects Subsystem Interface**

This subsystem holds the domain objects of our system. Our game object inherits from abstract class called GameObject. In this subsystem, façade class GameMap will create and modify game entities. Each class of this subsystem will be explained in detail in this section.

**Attributes:**

**private cell monster[ ][ ]:** this attribute represents the monsters as two dimensional cells, for the fast detection of collisions and the fast creation of game map.

**private ArrayList<GameObject>[] gameObjects:** this array of gameobject arraylists**,** references to the all lists which store the game entities of the system.

**private ArrayList<PowerUp> powerUps:** holds the reference to the all PowerUp objects exists on this map.

**private ArrayList<Monsters> monster:** holds the reference to the all Monster objects exists on this map.

**private Dot dot:** since there is one paddle for each map, this attribute represents is paddle of game map.

**private int mapWidth:** number of map cell columns.

**private int mapHeight:** number of map cell rows.

**Constructors:**

**public GameMap():** constructs a game map with default attribute values.

**Methods:**

**public void loadMap( int difficulty ):**  constructs a GameMap object by reading a map from disk which is represented as given ID. Maps are stored in the disk and can be mapped easily by and ID by this method. This method converts the GameMap object into new loaded map.

**public void addMonsterToCell(int x, int y) :** adds a Monster object to the cell, which is specfied by the x and y.

**public void deleteMonsterFromCell(x : int, y : int) :** frees the map cell in case of occupying the area with monsters in it

**public PowerUp createPowerUp( int powerUpID ):**  creates a PowerUp object with given ID.

**public void updateBrick(brickObj : Brick):**  calls the update method of the given brick object.

**public void applyPowerUp(int powerUpID):** applies power up according to the given power up ID.

## 4.1 Object Design trade-offs

We sacrificed efficiency to reusability, since our design goals are concerned more on simplicity for further developments rather than

to come up with most efficient implementation. That is why we used Facade pattern. We sacrificed functionality to usability since our implementation is a game and simplicity of this game would attract more customers rather than complex tutorials and game rules. We sacrificed memory to speed in order to come up with better and less complex runtime and less run time bugs we have created objects separately although it would cost us using more space in computer**.**

## 4.2 Final object design

### 4.2.1 User Interface Management Interface

**Menu Class**



Figure 5

**Methods:**

**public void setScreen(Screen s) :** This method is used in order to determine which screen will be used.

**public static void showSettings():** This method shows the settings that the user may or may not to change . Additionally determines which menu will be called(MainMenu or PauseMenu).

**MainMenu Class**

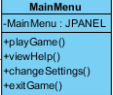


Figure 6

**Attributes:**

private JPanel MainMenu: mainMenu used in graphical user interface in order to display Main Menu on screen.

**Methods:**

**public void playGame():**  This method has a JButton which provides the name “Play Game” to be displayed in Main Menu. Additionally this method invokes game logic subsystem to start the game.

**public void viewHelp():** This method has a JButton which provides the name “View Help ”to be displayed in Main Menu. Additionally this method creates a new JPanel and displays the help that user needs.

**public void changeSettings():** This method has a JButton which provides the name “Change Settings” to be displayed in Main Menu. Additionally this method invokes the class SettingsMenu.

**public void exitGame():** This method has a JButton which provides the name “Exit Gamel” to be displayed in Main Menu and is responsible from the closure of the game.

**Pause Menu Class**



Figure 7

**Attributes:**

**private JPanel PauseMenu:**  PauseMenu used in graphical interface to show Pause Menu on screen.

**Constructor:**

**Public PauseMenu()**: It initializes the instance of PauseMenu object

**Methods:**

**public void displayPauseMenu():** This method is used in order to add pauseMenu panel on frame.

**public void resumeGame():** This method has the ability to invoke game logic subsystem to return game.

**public void returnMenu():** This method add Menu panel on frame by ending the current game.

**Screen Manager**

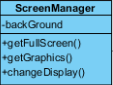


Figure 8

**Attributes:**

**privateJPanelbackGround:** Background is used in ScreenManeger to print the background image.

**Methods:**

**public JFrame getFullScreen():** This method returns a JFrame object to display game screen with the appropriate background..

**public Graphics getGraphics():** returns a graphics context for drawing to an off-screen image.

**public static void changeDisplay():** This method is used for the changes of Background.

**Main Menu Panel**



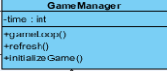
Figure 9

**Methods:**

**public MainMenuJPanelMenuActionListerner:** This method overrides MainMenu method of ActionListener interface.

### 4.2.2 Game Logic Subsystem Interface

**Game Manager Class**



**Figure 10**

**Attributes:**

**private int time:**

**Methods:**

**public void gameLoop():** this methods runs a loop in which the system is updated continuously.

**public void refresh():** It repaints the components in game frame.

**public void startGame():** starts a new game, by resetting game information stored on mapManager class.

**Sound Manager Class**



Figure 11

**Methods:**

**public void increaseSound():** this method increases the sound.

**public void decreaseSound():** this method decreases the sound until it is off.

**Settings Menu Class**



Figure 12

**Methods:**

**public void setDefaultSound():** It initializes a default sound volume.

**public void setDefaultBackground():** It initializes a default background for the game.

**Collision Manager Class**



Figure 13

**Methods:**

**public void collides(Object1,Object2):** This method collides the two objects and calculates the angles after the collision.

**public boolean wallCollided():** This method checks the collision with wall.

**public boolean bonusCollided():** This method checks the collision with bonus.

**public boolean checkCollided():** This method checks the collision.

**Map Manager Class**

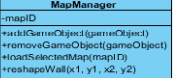


Figure 14

**Attributes:**

**private int mapID:** This attribute holds what map has chosen by the user

**Methods:**

**addGameObject(gameObject):** adds the gameobject that taken as a parameter, to the map

**public void removeGameObject(gameObject):** removes the gameobject that taken as a parameter, from the map.

**public void loadSelectedMap(mapID ):** sets the background image according to given ID, by loading a proper image by this id.

**public reshapeWall(int x1,int y1,int x2,int y2):** draws the wall according to given attributes from 0 and also updates the wall.

### 4.2.3 Input Manager Subsystem Interface

**Input Manager Class**

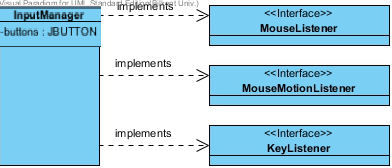


Figure 15

**Attributes:**

**private JButton buttons:** This attribute creates a button that is used in order to get the inputs from the user and also transfers to the listeners.

### 4.2.4 Game Elements Subsystem Interface

**Game Object Class**

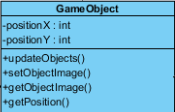


Figure 16

**Attributes:**

**private int positionX:** It holds the position of the object in X axis.

**private int positionY:** It holds the position of the object in Y axis.

**Methods:**

**public void updateObjects():** updates the X axis and Y axis attributes of the game object.

**public gameObjects getObjectImage():** This method returns the game object with its image.

**public void setObjectImage(gameObject object):** This method sets the appropriate image to the object.

**public int getPosition():** This method return the integer value about the position of the objects.

**Life Span Class**

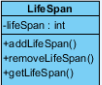


Figure 17

**Attributes:**

**private int lifeSpan:** holds how many life span does the user have.

**Methods:**

**public void addLifeSpan():** This method adds life spans. The reason can be the new game and default life spans or it is gained during the game with bonuses.

**public void removeLifeSpan():** removes one of the life spans. This method is used, during the game if the user loses one of the lives he/she has. Life span cannot have a negative number so while the lifespan is zero and again the user loses one life than this time method finishes the game .

**public int getLifeSpan():** This method returns the number of life spans.

**Wall Class**



Figure 18

**Methods:**

**public void removeWall():** This method removes the wall from the map.

**public void addWall()**: This method adds the wall to the map.

**Monster Class**

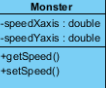


Figure 19

**Attributes:**

**private double speedXaxis**: The monster’s speed on X axis.

**private double speedYaxis:** The monster’s speed on Y axis.

**Methods:**

**public double getSpeed()**: returns the speed.

**public void setSpeed(double value):** sets the speed of the monster in the axis it moves.

**Dot Class**

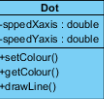


Figure 20

**Attributes:**

**private double speedXaxis**: The dot’s speed on X axis.

**private double speedYaxis:** The dot’s speed on Y axis.

**Methods:**

**public Colour getColour()**: returns the colour of the dot since it will be different in each level.

**public void setColour(Colour colour):** sets the colour of the dot.

**public void drawLine (int positionX , int positionY ):** This method draws a line to the positions dot passed.

**Bonus Class**

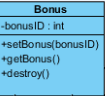


Figure 21

**Attributes:**

**private int bonusID:** This attribute holds the Id of the bonus to specify the kind of the bonus.

**Methods:**

**public void setBonus(bonusID):** According to the bonusID taken as a parameter it sets the bonus.

**public int getBonus():** It returns to the bonusID of the specified bonus.

**public void destroy():** removes the bonus from map after the collision

\*\*\*Lives Class, Monster Destroyer Class, Map Shrinker Class, Freeze Time Class, and Slow Time class inherits the features of Bonus class.

**Time Manager Class**

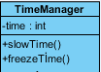


Figure 22

**Attributes:**

**private int time**: this attribute is used in order to modify the bonus types regarding as Slow Time Class and Freeze Time Class

**Methods:**

**public void slowTime():** By decreasing the rate of increment in time this method changes the motions of the related objects in the map.

**public void freezeTime():** This method stops the increment in time and the motions of the related objects stays same.

## 4.3 Packages

With the Java Development Kit’s default packages, the implementation of the Imprisonment will be a lot easier. Therefore, in the software, there are twelve different types of package used properly in the specific classes.

1. java.awt.\*

The java.awt package is provided by Java Development Kit. The main purpose of using the java.awt package is creating user interface and visual components of Imprisonment. The graphics and buttons are main part of the user interface and the components’ visualization. Thus, the java.awt package is used in all boundary classes.

2. java.awt.event.\*

Since the user inputs generate different types of events, the Game Control subsystem’s control classes need to use java.awt.event package. With the usage of keyboard and mouse, the controller can take what the user wants to do with the help of this packages and its ‘Action Listener’s.

3. java.applet.\*

In Imprisonment, the software is designed as an applet. With the support of java to implement an applet, the software can be used on web based type of the program in the future development of the game.

4. java.util.\*

The utilization of our game is done by this java.util package. The framework, time, random-number generator and many utilization components of the software is implemented via java.util package’s specifications.

5. java.awt.Color

Since the monsters, background, bonuses and the Dot has different colours, the software needs to make those components of the game colourful. With java.awt.color, the visual part of the game and the experience of the user in terms of learning curve of the game and distinction between objects and background is increased on the Imprisonment.

6. java.awt.Graphics

The current colour, shape, font and many attributes of a specific object such as monsters and the Dot is maintained by the help of Graphics class of the Java Development Kit. The package is used on boundary objects to handle the graphics of the software.

7. java.awt.Graphics2D

Since our game is two dimensional, some specifications of the java.awt.Graphics2D can be useful of our implementation. The changeable shape of the wall, especially, is handled by graphics2D. Since the wall will be a polygon, the shape and the colour of the wall and geometric calculations to determine whether the 80% of the game area is filled by the user’s movements.

8. java.awt.geom.Ellipse2D

Since some of the monsters can shoot arbitrarily, the bombs would be ellipses and handled by Ellipse2D package. The implementation of the elliptic object’s geometry, position and colour is easier with the help of this package.

9. java.awt.Polygon

In implementation, the main geometric shapes of the objects will be polygons. All the objects shape and area occupied by the object must be determined. Without the help of Polygon package, this would be tremendously hard job for the implementation of objects, especially walls. Everything about Game Object subsystem classes will be handled by Polygon package.

10. java.awt.event.KeyEvent

Among all event listeners, key event is the key part of the Imprisonment. The movement of the Dot is main event of this game. Therefore, the KeyEvent provides a considerable help to handle the movements of the user and make the Dot act properly.

11. java.util.ArrayList

The ArrayList package is helpful due to hold the different points located on the game map of different object’s coordinates. Within the ArrayList of coordinates, it is easy to calculate map area filled and handle collisions.

12. java.util.Collections

Within the time when the player tries to cover an area of the map, there are some broken lines of polygons which are not closed geometric shapes. The control classes need to collect this coordinates of ‘broken polygons’ so that if a monster hits those lines, the lives of the dot are decreased properly. The Collections package provides multiple coordinates and handle those points so that we can calculate the positions of those ‘broken lines’.

## 4.4 Interfaces

The user interface of our system will be as simple as possible, regarding performance. It will not be hard and time consuming for the users since we aim to build a game that has reusability and reliability, considering the future improvements to the software. Our main goal in user interface is eliminating the less-used functionalities and making the system’s functionalities used.

# 5 Conclusion

In conclusion, in this report we tried to come up with more concrete solutions

and to become one step closer to the creation of Imprisonment before the implementation part. The essentiality of design which is one step ahead of the analysis part cannot be ignored. Thus we tried to show up with an effective design report believing it will lift a big burden from our shoulders during implementation phase.

Moreover, in the Subsystem Decomposition part, in order to provide a better understanding related with our interactions and basic essentialities of our software, we illustrated a detailed class diagram as dividing our class diagram into subsystems  by following one of the most effective methods which is divide and conquer. Thus it is possible to say the better understanding that is provided comes from this tactic.

Last but not least, Model View Controller allows to changes in the interfaces of the system

without not changing the algorithm, which is an huge advantage to not to notice and it made

us realize that it is one of the best paths for better software engineers.