Bilkent University

Department of Computer Engineering

**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

## 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 run time 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 east 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 similiar 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 seperately, sacrifice the used memory spaces which increases the speed of some operations such as the detection of collisions.

## 1.3. References

1) https://en.wikipedia.org/wiki/Java\_virtual\_machine

2) Object-Oriented Software Engineering, Using UML, Patterns, and Java, 3rd Edition*, by Bernd Bruegge and Allen H. Dutoit, Prentice-Hall, 2010, ISBN-10: 0136066836.*

## 1.4. 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 subsytems 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 subsytem 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.

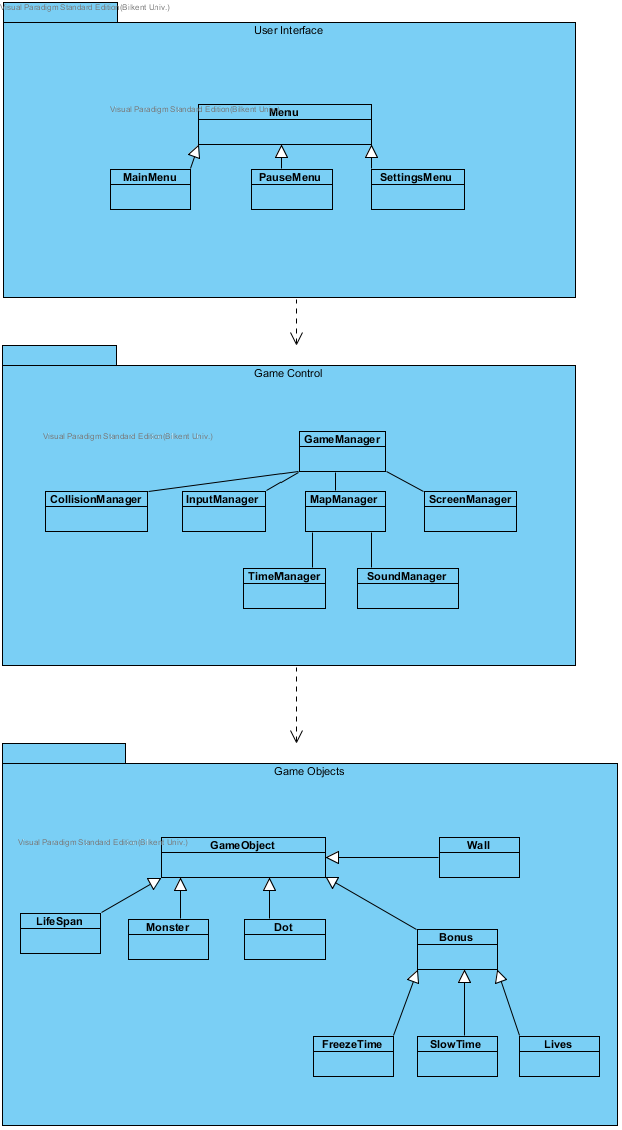
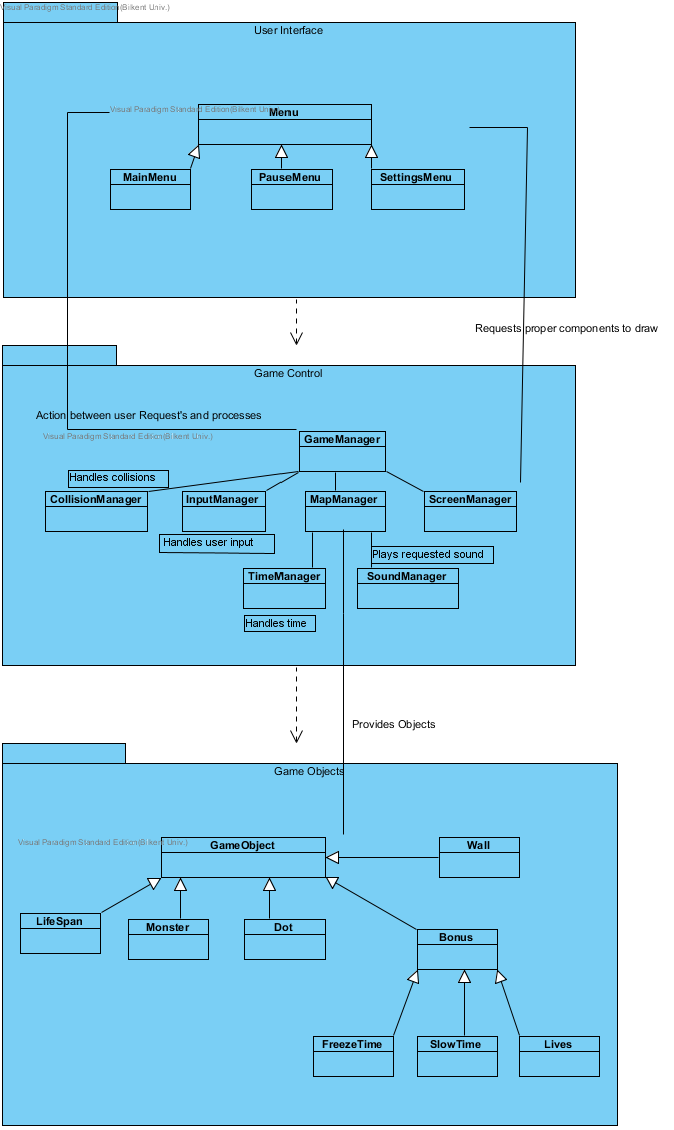


Figure 1(Basic Subsystem Decomposition)

 Figure 2 (Detailed Subsystem Composition)

## 2.2. Architectural Style

### 2.2.1. Layers

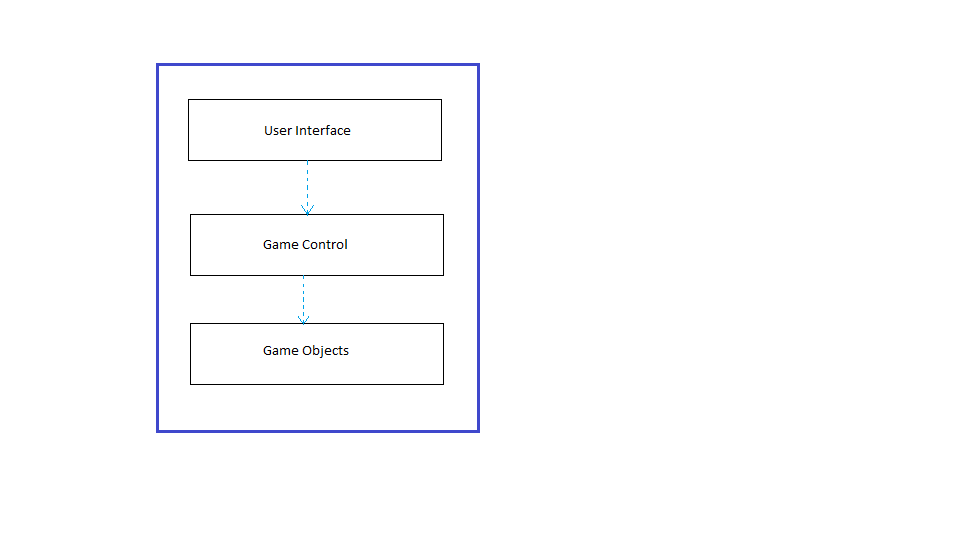
The system is decomposed into three layers, namely User Interface, Game Control and Game Objects. These layers is divided as the subsystems divided so that they provide related tasks. The top layer which has the highest hierarchy is User Interface layer, since it is not used by any subsystems above and 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 game object 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.

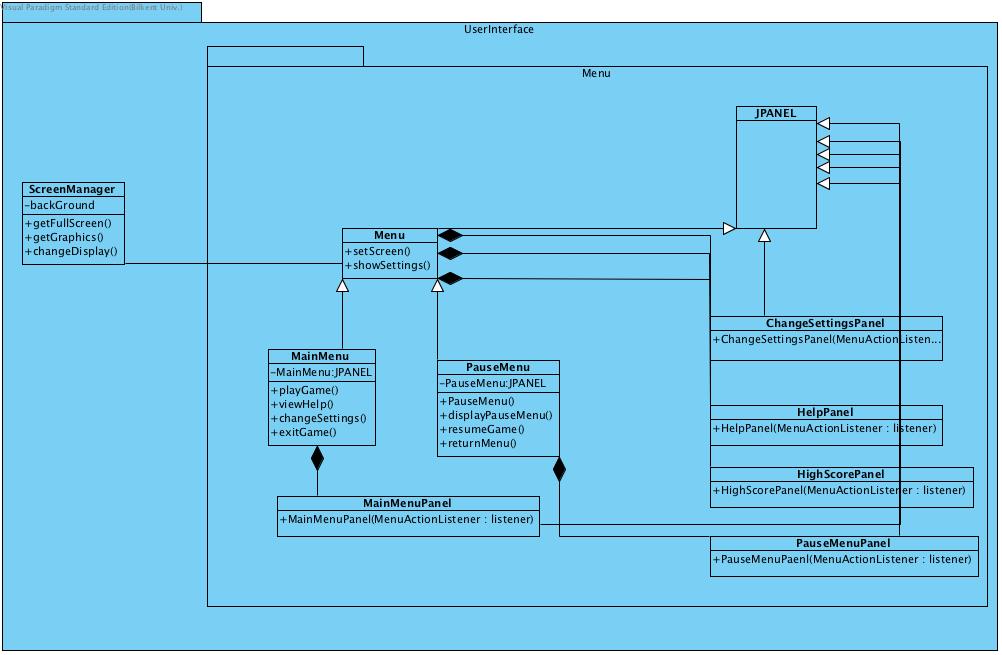
**3.1 Design Patterns**

**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.



**3.3 Game Management 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, GameMapManager 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. This class also implements the runnable interface since SoundManager runs in a different thread.

**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 by referencing to this object. Like getting the Ball objects on the map etc.

**private int paddleKind:** this value is for determining the paddle kind, since there can be different kinds of paddles in our system, this value is needed to create various types of paddles like green, blue,gray paddle.

**private int commonBallState:** this attribute is used to determine the common ball type, since it will be needed to determine the actions performed in collision according to ball type. This value is obtained by currentMap attribute by getting the type of one ball object in ball list. This attribute is hold for fast determinations in collisions.

**private GameInformation info:** this attribute holds the current game information, like current level , number of lives. This information is updated according to game dynamics which are also handled in this class.

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

* CELL\_WIDTH and CELL\_HEIGHT are static constant variables, they are used to divide the map into equally sized pieces according to brick sizes.

#### Constructors:

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

#### Methods:

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

**public drawCurrentMap(Graphics g, int screenWidth,int screenHeigh,GameInformation info):** draws the current map according to given attributes.

**public void applyPowerUp(int powerUpID):**  applies the powerUp to the game by referencing the currentMap attribute, also it references to PowerUpTable class for interpreting which id refers to which powerUp ( which is a class of Game Entities subsystem ).

**public void update(long elapsedTime) :**  updates the game objects of currentMap by given elapsed time.

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

**3.4 Game Entities Subsystem Interface**

This subsystem holds the domain

**Attributes:**

**Constructors:**

**Methods:**

**3.5 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.