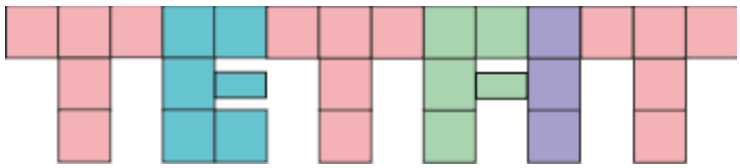
CS 319 - Object-Oriented Software Engineering

System Design Report



Group 1F

Irmak Akkuzuluoğlu

Irmak Türköz

Mustafa Culban

Yasin Erdoğdu

1. Introduction 6
1.1 Purpose of the system 6
1.2 Design Goals 8
1.3 Definitions, acronyms, and abbreviations 8
1.4. References 8
1.5. Overview 8
2.1. Overview 9
2.2. Subsystem Decomposition 9
2.3. Architectural Styles 12
2.3.1 Layers 12
2.4. Persistent Data Management 13
2.5. Access Control and Security 14
2.6. Boundary Conditions 14
2.6.1 Execution 14
2.6.2 First Run 14
2.6.3 Termination 14
3. Subsystem Services 15
3.1. Design Patterns 15
3.1.1 Factory pattern 15
3.1.2 Decorator Pattern 15
3.1.3 Memento Pattern 15
3.2. User Interface Subsystem Interface 16
1.GameMenu 17
Attributes: 17
Constructors: 17
Methods: 18
Attributes: 18
Constructors: 18
Methods: 19
Attributes: 19
Constructors: 19
Methods: 19
Attributes: 20
Constructors: 20
Methods: 20
3.3 Game Management Subsystem Interface 20
1. GameEngine Class 21
Attributes: 22
Constructors: 22
Methods: 22

2. GameMap Class **22**
Attributes: **23**
Constructors: **23**
Methods: **23**
3. StroageManager **23**
Methods: **23**
4. MiniMenuManager **24**
Attributes: **24**
Constructors: **24**
Methods: **24**
5. InputManager Class **24**
6. SoundManager **25**
Attributes: **25**
Methods: **25**
7. OutputManager **25**
Attributes: **26**
Methods: **26**
8. ScreenManager **26**
Methods: **26**
9. CollisionManager **26**
Methods: **27**
3.4 Game Entities Subsystem Interface **27**
1. MapObject **27**
Attributes: **28**
Methods: **28**
2. Sprite **28**
Attributes: **29**
Methods: **29**
3.Location **29**
Attributes: **29**
Constructors: **29**
Methods: **29**
4. Brick **30**
Attributes: **30**
Constructors: **30**
Methods: **30**
3.5 Detailed System Design **30**

1. Introduction

1.1 Purpose of the system

Tetfit is a 2-D tile matching puzzle game, which is intended to entertain users greatly. It has minimalistic graphics, consisting of simple shapes and a user-friendly interface. The gameplay is easy to understand, but the game gets challenging as the player carries on with the game. Tetfit also has different levels of difficulty which gives the players the freedom to choose where to start from, making gameplay more individualistic and enjoyable. It is a skill-based game, it has two different modes that both triggers strategical thinking. Overall, it is a great pastime activity with a pleasing design.

1.2 Design Goals

Below are some important design goals of the system which are proposed to meet non functional requirements of our project, which were mentioned previously in the analysis report.

Usability: The ease of use is one of the most important aspects of the system for it is a game aiming to entertain people. In order to provide this, player should be able to get a grip of the system easily. For this purpose, we will provide the user friendly simplistic interfaces, so the user can achieve each and every operation he desires to perform. Moreover, it is decided that the menu operations will be done by mouse clicks and the game will be controlled with the arrow keys for they are the most common and easy ways to control a game.

Ease of Learning: In order to let the user have fun without dealing with confusion we intended to give the game the ease of learning. Tetfit is overall a simplistic game, with an easy main goal. The information about the main goal and how to play the game will be

provided by the help document which will can be accessed through the main menu. This documentation will be detailed enough to avoid any kind of confusion. Since the game is already an easy to understand game, we expect the users to learn and play the game easily.

Extendibility: Letting the game to extend and change is a crucial point to maintain the interest of the users in general, by adding new components, features to the game. In order to achieve this our system will be open to any kind of change, making it convenient for us to add new functionalities and entities easily to the present system.

Modifiability: Extendibility also requires modifiability. Modifiability provides the ease to modify the system to correct faults, improve performance or other attributes, or adapt to a changed environment. Throughout the implementation this specification will be kept in mind so that our system will allow us to improve it.

Portability: This is also an important issue for a software system, for it can allow a wide range of users to access it. With this consideration, we decided to implement our system in such manner. Our game will be implemented in Java, because its JVM maintains platform independency, allowing the system to achieve the portability.

Response Time: For our project is a game, another crucial goal is to keep a short response time. It is vital to provide quick responses to the users' requests, in order to maintain a pleasing experience for the users. Our goal is for the system to respond to users' actions as quick as possible, while also smoothly displaying animations and screening the gameplay.

Trade Offs:

a) **Efficiency – Reusability:**

Reusability is not a concern for this particular project because we do not have any intention to reuse any of our classes for a different system. Thus, the classes are designed just for our system, specifically for the tasks needed, so the code is not any complex than necessary. This design approach helps us to maintain our efficiency, for there will not be any extra complexity for making it reusable.

b) **Ease Of Use and Ease of Learning vs. Functionality:**

Our system aims to provide an ease of understanding and usability. Hence, we need to trade off the functionality to keep the game simple. This is planned to be achieved by

avoiding complex functions and by not exceeding a certain number of different functional abilities that will be provided to the user. We will keep our minimalistic view, and implement the best possible game which easy to understand and play by sacrificing a bit of the functionality.

c) **Performance vs. Memory:**

In our game, it is vital to provide the player smooth animations, effects, transitions, for an enjoyable gameplay. thus, performance is the primary focus of the system. However, in order to achieve this we had to sacrifice the memory.

1.3 Definitions, acronyms, and abbreviations

Abbreviations:

1.4. References

[1] http://en.wikipedia.org/wiki/Java_(programming_language)

[2] http://www.nada.kth.se/~karlm/prutt05/lectures/prutt05_lec7.pdf

[3] https://drive.google.com/file/d/0B9ApNnKlfgcHeUxKbFAyQTNQbE0/view

[4] https://drive.google.com/file/d/0B9ApNnKlfgcHYllRbEZEYjFRRzg/view

[5] http://www.ieee.org.ar/downloads/Barbacci-05-notas1.pdf

1.5. Overview

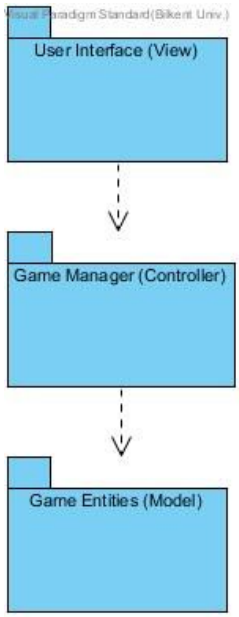
This section, generally summed up the main purpose of the system, that is to provide entertainment to users. In order to meet every intended goal, we defined them clearly, in detail. Our main focus on design goals are usability, ease of learning, extendibility, portability, modifiability and high performance. We are aware that every system requires some trade offs to be made, hence we planned these trade offs to be as favourable as possible for our project. We made some trade offs between development time and complexity, ease of use and functionality, lastly performance and memory.

2. Software Architecture

2.1. Overview

The purpose of section to decompose our system into maintainable subsystems. Our software architecture aims to have reduced coupling which measures the dependencies between two subsystems and increased cohesion which measures dependencies among classes within a subsystem. To apply this method most efficiently to our system we will use MVC (Model View Controller) architectural style.

2.2. Subsystem Decomposition



This section will divide system into subsystems with package diagram to organize the design of the project. The subsystems should separate project into independent parts whereas their dependency to each other must be prevented.

Since we use MVC (Model View Controller) Architecture; we designed our subsystems as User Interface which represents View, Game Manager which represents Controller and Game Entities which represents the Model part. Their dependency between each other is lowered which means they are loosely coupled. Moreover, they do not depend on each other with their functionalities which provides sustainable project management as well as projects error management since when a subsystem is corrupted or giving error other systems could still run. For example, when there is an error in the User interface subsystem, the error will occur in that subsystem which would not affect the ”Game Manager”s or the “Game Entities”s functionalities. Also, this way of programming would separate subsystems in such a way that they can be parallely implemented. Figure -1 shows a very rough package diagram of the subsystems.

In Figure 2 and Figure 3 there are more extended versions of our package models. The main purpose of such design was to better understand the game logic, interactions between model, controller and view and the subsystems. It is important to observe that subsystems in the same package has relation between each other whereas they do not directly affect to each other’s functionalities. We designed our system in such a way that it has high cohesion and low coupling.

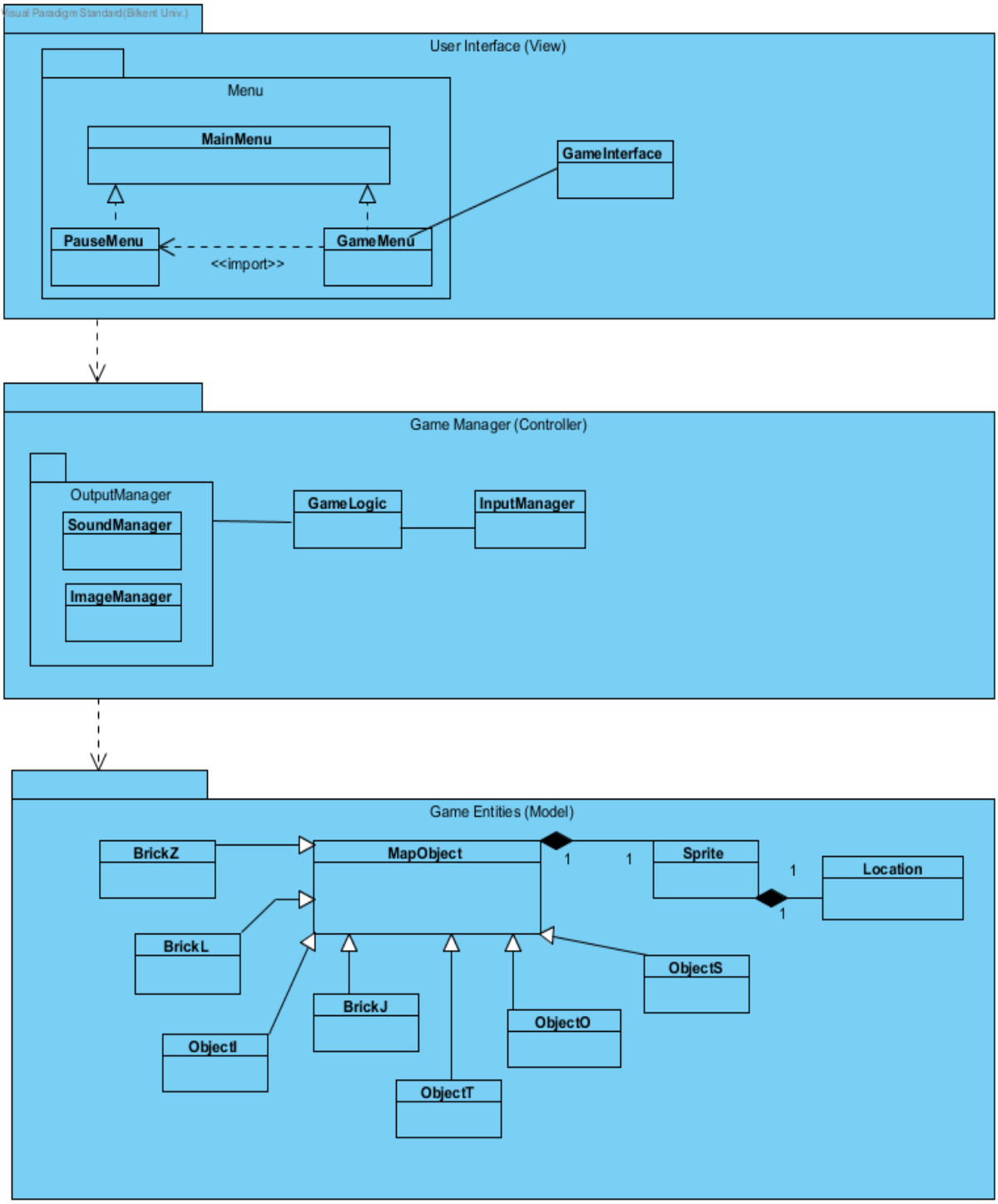


Figure-1(Subsystem Decomposition)

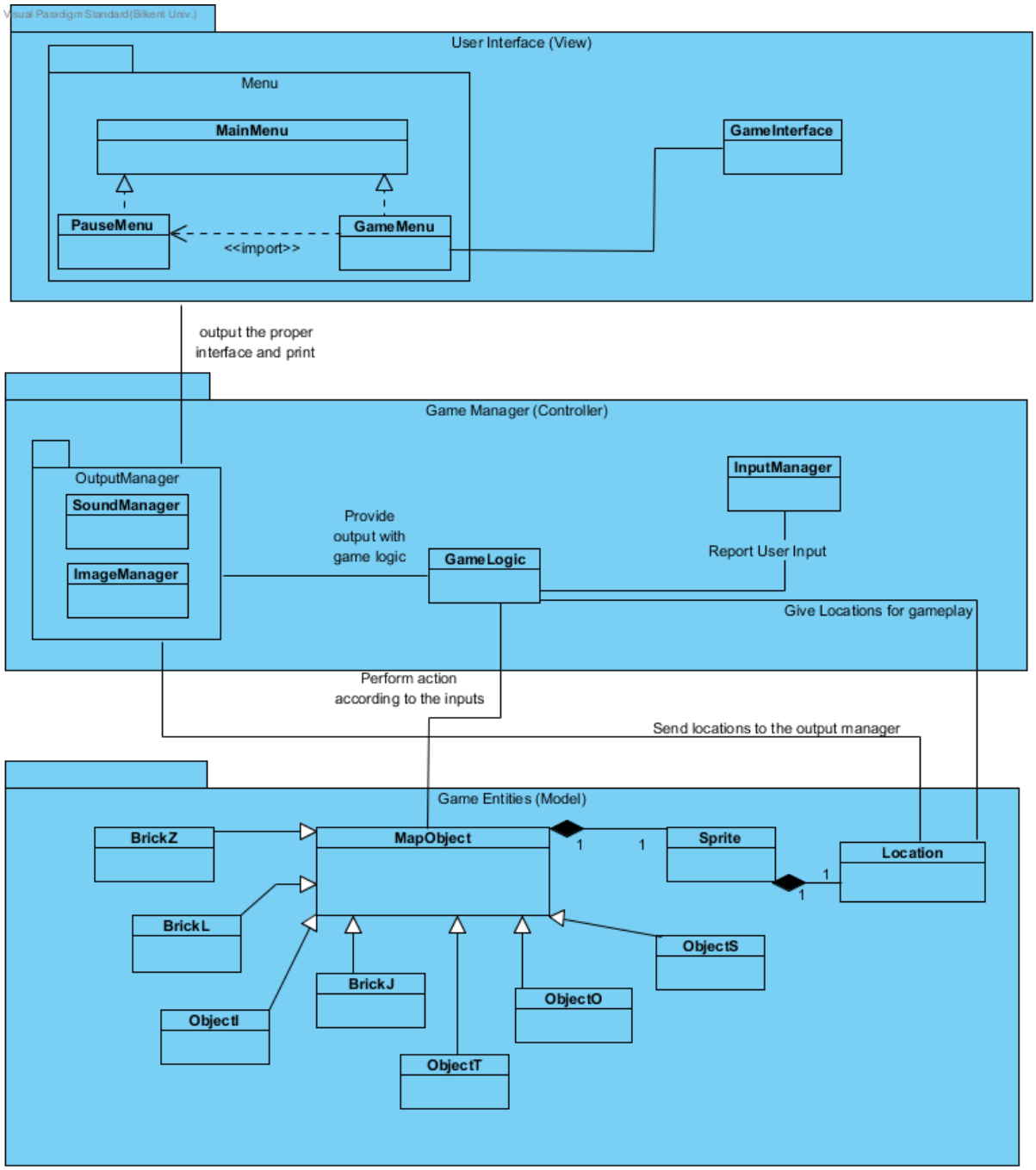


Figure-1(Detailed Subsystem Decomposition)

2.3. Architectural Styles

2.3.1 Layers

We decomposes our system into three hierarchical layers during the grouping subsystem which prıvides related services. These layers hierarchically are User Interface, Game Manager and Game Entities. Since our layer architecture is a closed architecture, each layer in our system only calls operations from one layer below. User interface is top at the layer hierarchy, therefore, It depends on below layer results and it ınteracts with user. The middle layer is Gma Manager layer which is responsible for controlling game logic. Game Entities layer exists at the bottom of the hierarchy and ıt holds information about entity objects. Decomposition of our system is as below;

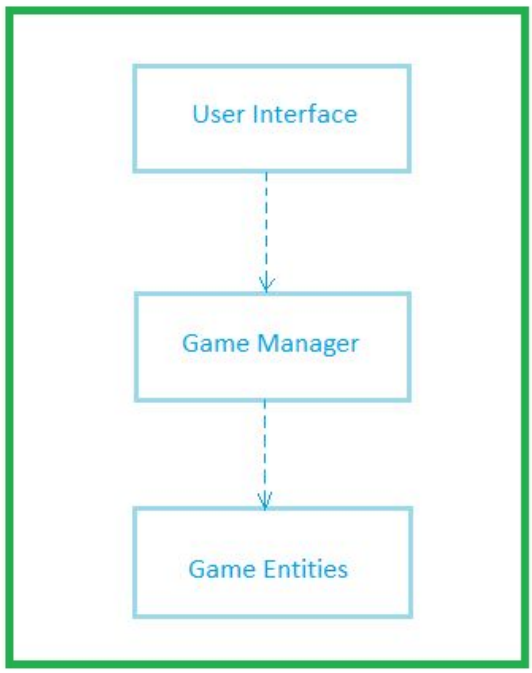


Figure-3 (Layers of System)

2.3.2 Model View Controller

The main point of using Model-View-Controller architectural style is to divide system into three different parts and manage interaction policies these three parts. We grouped classes which are under Game Manager layer into Controller part which is basically the managing part of the system and provide communication between Model and View part.

Game entities are grouped into Model part and the classes that are responsible user interaction are grouped into View part. With controller part, View part cannot access directly Model part. By MVC architectural style, it is enabled that any changes comes from user interface do not cause any changes also in domain part of the system, thanks to this, this architectural style is good for the games.

2.3. Hardware / Software Mapping

Tetfit is going to be implemented in Java, hence, the latest JDK (1.7) is going to be used for implementation. For hardware configuration, Tetfit needs a basic keyboard inputting for playing the game and also the typing the name for highscores, and mouse for users to go through the menu. Because of the fact that the game is going to be implemented in Java and system requirements are going to be minimal, all it needs is a basic computer with basic softwares installed such as an operating system and a java compiler to compile and run the files. Moreover, Java's platform independency will be beneficial. Our game will not require internet connection to operate.

2.4. Persistent Data Management

Our project will not require any complicated data storage, therefore, it does not need a complex database system. It will store high score list as a text file in disk. If text file gets corrupted, it will not be effecting in-game issues such as game objects, but of course the scoreboard will be faulty. Furthermore, our plan is to store object images and sound effects in hard disk drive with simple sound and image formats.

2.5. Access Control and Security

As mentioned before, Tetfit does not need any kind of internet connection. Anyone who initializes the game will be able to play the game, if the necessary basic softwares are available. Thus, there will be no restrictions or control for access. Moreover, it will not include any user related data or profile, so, there will not be any kind of security issues.

2.6. Boundary Conditions

2.6.1 Execution

TETFIT will be only requiring Java Runtime Environment on the gamer's computers. That will be only necessity.

2.6.2 First Run

On first run, TETFIT will start gradually and nothing will be happened on the saving purposes but if user wants to save a game on the startup of the TETFIT , s/he will choose from the menu and after selection, TETFIT will check for the saved games on the directory *svdGames.* If TETFIT Memento pattern handler finds a game in the folder, it will lists all the saved games but if not, TETFIT will ask to user to create new game.

2.6.3 Termination

There are only one chance to exit a game. On every men you should go the Main Menu in order to exit. This was planned because keeping player on the game whether or not he or she wants to close the game. We are trying to take her/him in the game by not allowing him/her to close game in any menu. Also there is X button on every operating system (for Windows, it is on the top and the rightmost button, for MacOS it is on the top left and red button). If user clicks on this button we are again asking whether wants to leave game seriously or not to hold him/her on the game.

3. Subsystem Services

3.1. Design Patterns

In our game we decided to use more than one design patterns to do our complicated jobs in a separated and in an easy way.

3.1.1 Factory pattern

In order to accomplish this pattern, We firstly need to implement a Factory class to create all the pieces in the game that are used. So we create *PieceFactory* class for that purpose. On this case, This class is concerned with creation of the pieces which are derived classes of *Grid* base classes will be created so that *TetrisGame class will not concern with the job that PieceFactory class does.*

3.1.2 Decorator Pattern

In this pattern we are aiming to create popup window messages for example “You win the game” or “ Game is over”. We have *Shape* class for drawing items to the UI. In other words, this class is used for drawing every pieces , every text, every messages etc. to the screen. So decorator Pattern will be using because we are creating messages to the screen. *TextDecorator* class will be using instance of *Shape* class and *TextDecorator* will be able to write a message every class that is derived from *Shape* class temporarily. *TextDecorator* class will be able to draw a message on the center of the game screen. This can be used with examples that were given below and for example to disable buttons in the games.

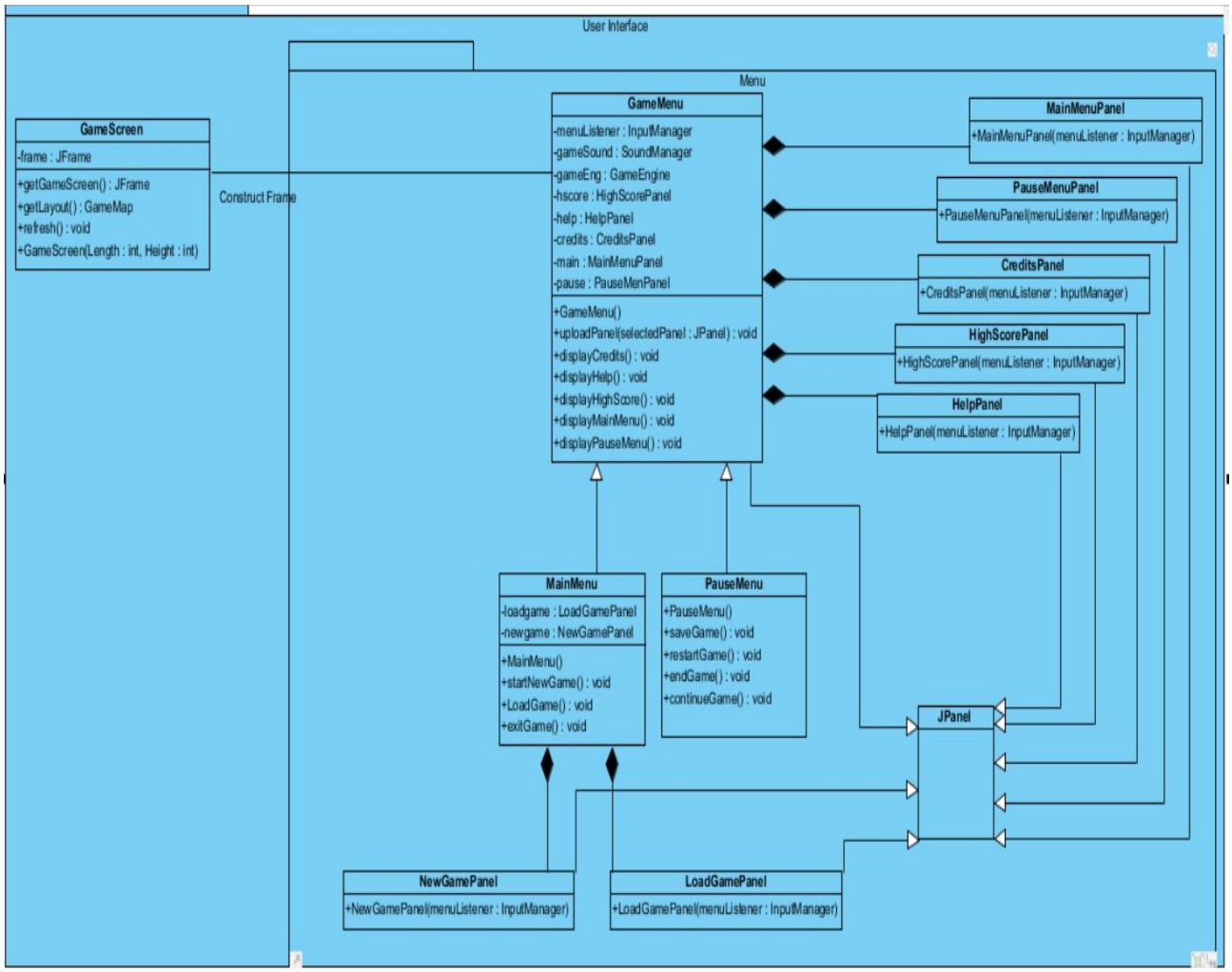
3.1.3 Memento Pattern

Inside the game play , at any instance of a time (except the game over state) game will be able to save and restore the game. Memento Pattern will be sing to satisfy this

purpose. Every instance of the game are stored in the *TetrisGame* class , blocks type, their positions, score, fallen blocks etc. So Saving *TetrisGame* class will be sufficient to save the game. So the *Memento* class is taking *TetrisGame* as a target and save it with all its instances. Memento pattern is implemented with 3 objects[1] : originator, caretaker and memento. Originator is *TetrisGame* in our game , Memento is with a same named class Memento and caretaker is the main game runner class *Tetris.* By encapsulating in this pattern *Tetris* class won't consider with saving game in its own and game will be saved and be restored by this Pattern.

3.2. User Interface Subsystem Interface

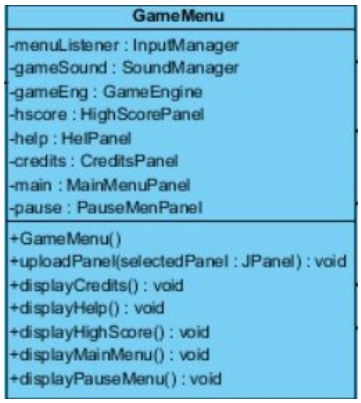
In User Interface subsystem, the graphical components of our system is provided. The classes which are responsible for displaying our game menu and game playing screens are grouped together.



[1] https://en.wikipedia.org/wiki/Memento_pattern

Figure-4(User Interface)

1.GameMenu



Attributes:

- **menuListener : InputManager**→It is InputManager instance to gets the inputs that is given by the mouse
- **gameSound : SoundManager** →It is SoundManager instance to manages the sounds played during the game
- **gameEng : GameEngine** →It is GameEngine instance and when game playing is selected, ıt provides reference to GameEngine subsystem.
- **hscore : HighScorePanel** →It is HighScorePanel class instance property which is in JPanel type and it is responsible for displaying Highscores on screen.
- **credits : CreditsPanel** → It is CreditPanel class instance property which is in JPanel type and it is responsible for displaying Credits on screen.
- **help : HelpPanel** → It is HelpPanel class instance property which is in JPanel type and it is responsible for displaying Help/Tutorial on screen.
- **main : MainMenuPanel** → It is MianMenuPanel class instance property which is in JPanel type and it is responsible for displaying Main Menu on screen.
- **pause : PauseMenuPanel** → It is PauseMenuPanel class instance property which is in JPanel type and it is responsible for displaying Pause Menu on screen.

Constructors:

- **GameMenu()** →It is constructor to initailzes MainMenuPanel, PauseMenuPanel, CreditsPanel,HelpPanel, HighScorePanel, menuListener, soundListener and gameEng attributes.

Methods:

- **uploadPanel(selectedPanel: JPanel)** → It is public function which is responsible for uploading panel on frame based on selected panel from GameMenu.

- **displayCredits()** → It is public function which includes uploadPanel() method to display CreditsPanel to frame.

- **displayHelp()** → It is public function which includes uploadPanel() method to dsiplay HelpPanel to frame.

- **displayHighScore()** → It is public function which includes uploadPanel() method to dsiplay HighScorePanel to frame.

- **displayMainMenu()** → It is public function which includes uploadPanel() method to dsiplay MainMenuPanel to frame.

- **displayPauseMenu()** → It is public function which includes uploadPanel() method to dsiplay PauseMenuPanel to frame.

2.MainMenu

Attributes:

- **loadgame: LoadGamePanel** → It is LoadGamePanel class instance property which is in JPanel type and it is used to show Saved Games on screen.
- **Newgame : NewGamePanel** → It is NewGamePanel class instance property which is in JPanel type and it is used to show game modes and levels to start on screen..
-

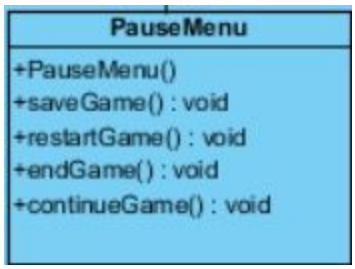
Constructors:

- **MainMenu()** → it is constructor to construct MainMenu object.

Methods:

- **startNewGame()** → It is method to call GameEngine subsytem to start new game by the reference gameEng property of GameMenu class.
- **LoadGame()** → It is method to call GameEngine subsytem to load selected saved game by the reference gameEng property of GameMenu class.
- **exitGame()** → This terminates the game running.

3.Pause Menu



Attributes:

Constructors:

- **PauseMenu()** → it is constructor to construct PauseMenu object.

Methods:

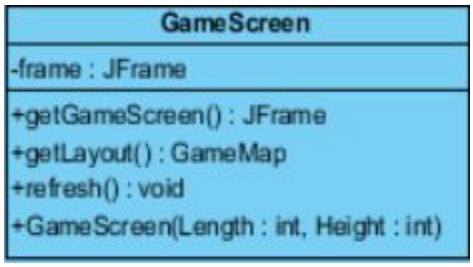
- **saveGame()** → It is method to call GameEngine subsytem to save the paused game by the reference gameEng property of GameMenu class.

- **restartGame()** → It is method to call GameEngine subsytem to restart the paused game by the reference gameEng property of GameMenu class.

- **endGame()** → It is method to call GameEngine subsytem to end game and return main menu by the reference gameEng property of GameMenu class.

- **continueGame()** → It is method to call GameEngine subsytem to resume the game by the reference gameEng property of GameMenu class.

4.GameScreen



Attributes:

- **frame : JFrame** → It is private attribute which is the main frame of game.

Constructors:

- **GameScreen(int Length, int Height)** → It takes the screen's height and length as parameter to construct.

Methods:

- **getGameScreen()** → It is public method to display constructed game screen.
- **getLayout()** → It is public method to draw context of off-screen image
- **refresh()** → It is public method to refresh game screen

3.3 Game Management Subsystem Interface

In this subsystem our controller objects are grouped together to manage the actual game dynamics and game logic. We have bla bla. These classes will be explained in detail, in this section.

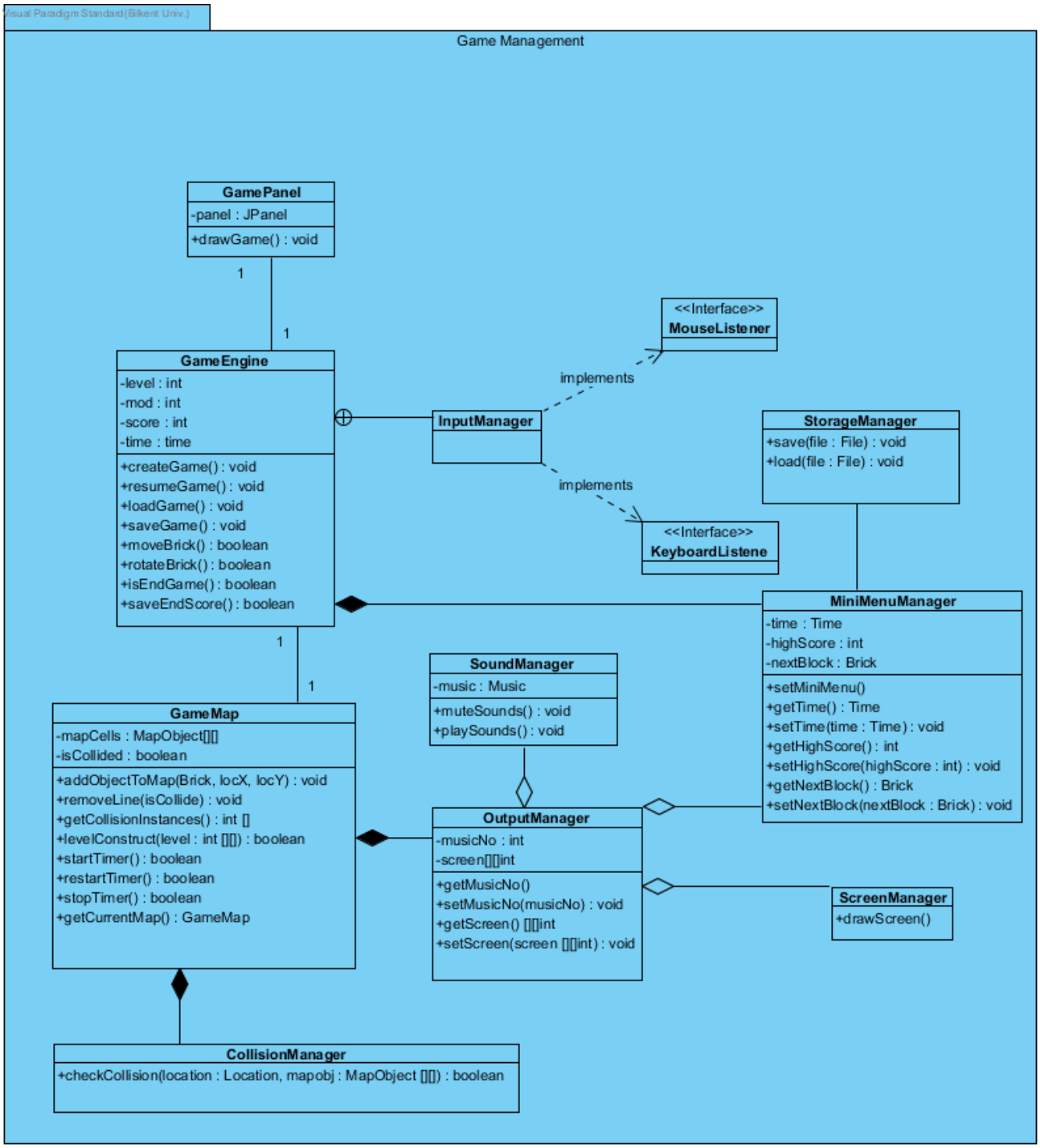
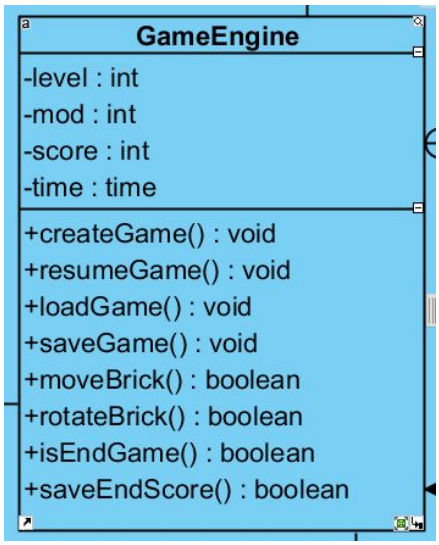


Figure-5(Game Management Subsystem)

1. GameEngine Class



Attributes:

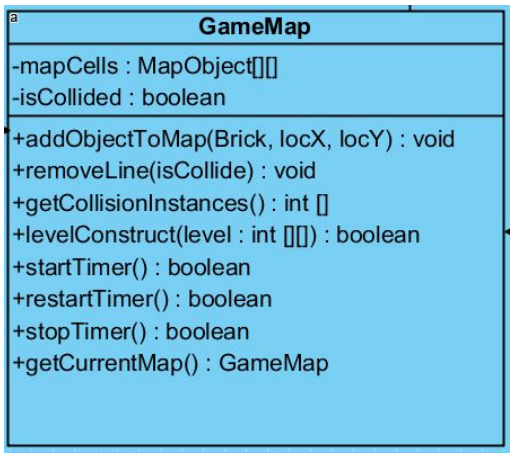
- -level : int → indicates the level of the current game which determines the speed of bricks
- -mod : int → the type of the game which is chosen by the player at the start of the game
- -score : int → the total points collected by the player during the current game
- -time : time → the total time spent starting from the initiation of the current game

Constructors:

Methods:

- +createGame() : void → initiation of a new game
- +resumeGame() : void → continuation of a paused game
- +loadGame() : void → continuation of a previously started and saved game
- +saveGame() : void → saves the current game to be loaded later
- +moveBrick() : boolean → moves brick left and right or makes it fall faster
- +rotateBrick() : boolean → rotates the brick in the desired direction
- +isEndGame() : boolean → indicates that the current game has ended
- +saveEndScore() : boolean → saves the score of currently ended game

2. GameMap Class



Attributes:

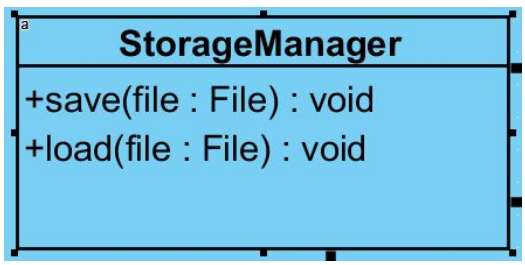
- -mapCells :MapObject[][] → a 2D array that keeps track of the map consisting of cells
- -isCollided : boolean → indicates that two bricks have touched each other

Constructors:

Methods:

- +addObjectToMap(Brick,int locX,int locY) → adds a brick to the current game map
- +removeLine(isCollided) : void → if a line is complete this method removes it and make sthe other lines above it fall 1 unit down
- +levelConstruct(level : int [][]) : boolean → constructs a level
- +startTimer() : boolean → starts the timer for the current game
- +restartTimer() : boolean → restarts the timer of the game
- +stopTimer() : boolean → stops the timer of the game
- +getCurrentMap() :GameMap → returns the current game map

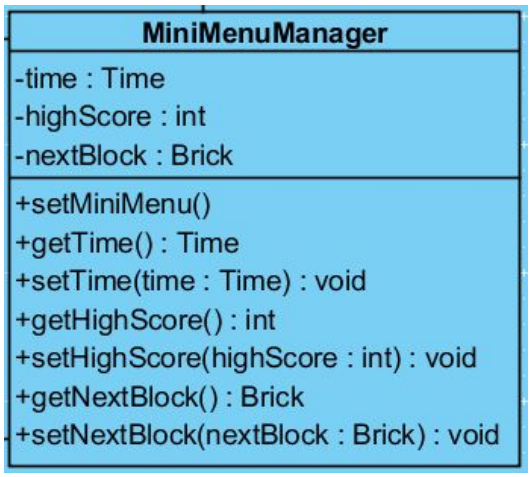
3. StroageManager



Methods:

- +save(file : File) : void → saves given parameter file to the database.
- +load(file : File) : void→ loads given parameter file from the database.

4. MiniMenuManager



Attributes:

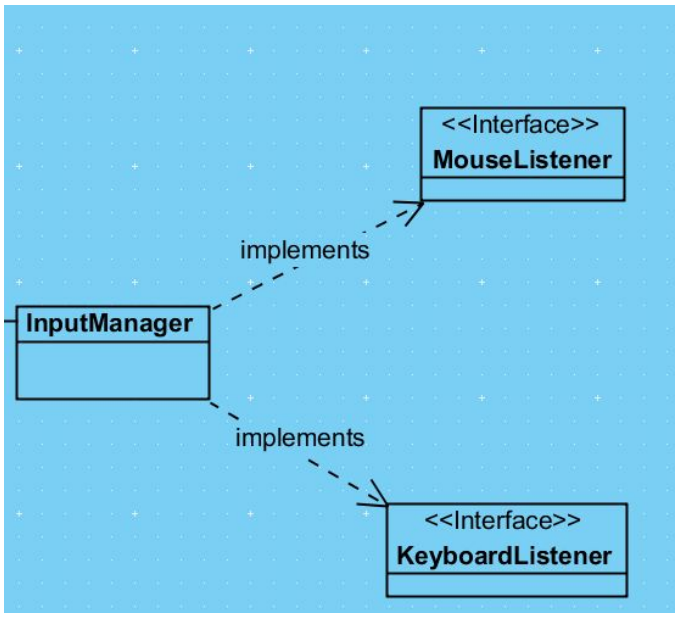
- -time : Time
- -highScore : int
- -nextBlock : Brick

Constructors:

Methods:

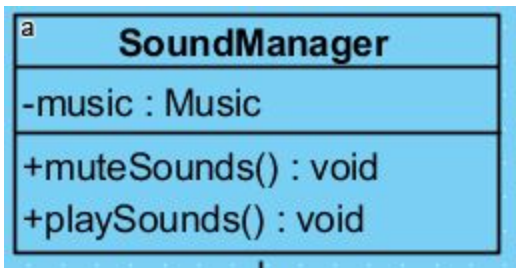
- +setMiniMenu()
- +getTime() : Time
- +setTime(time : Time) : void
- +getHighScore() : int
- +setHighScore(highScore : int) : void
- +getNextBlock() : Brick
- +setNextBlock(nextBlock : Brick) : void

5. InputManager Class



This class is designed to detect the user actions performed by mouse to move/ turn/change the speed of the bricks, . In this context, this class implements proper interfaces of Java.

6. SoundManager



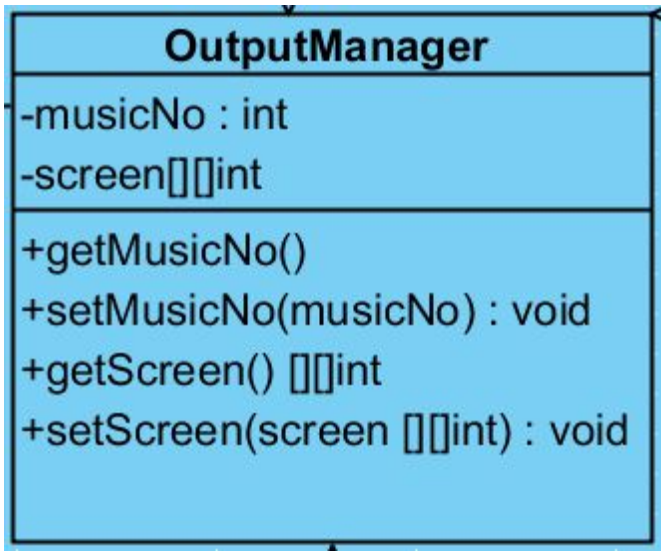
Attributes:

- -music : Music

Methods:

- +muteSounds() : void
- +playSounds() : void

7. OutputManager



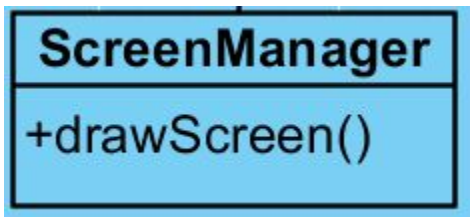
Attributes:

- -musicNo : int
- -screen[][]int

Methods:

- +getMusicNo()
- +setMusicNo(musicNo) : void
- +getScreen() [][]int
- +setScreen(screen [][]int) : void

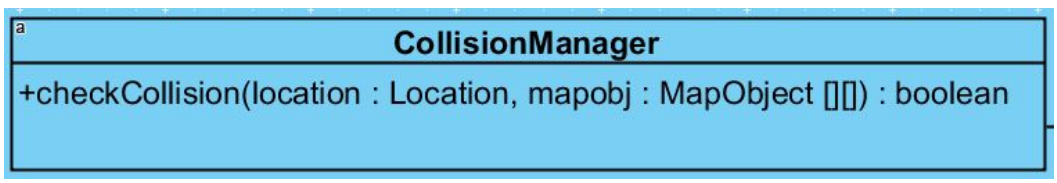
8. ScreenManager



Methods:

- +drawScreen()

9. CollisionManager



Methods:

- +checkCollision(location :Location, mapobj :MapObject [][]) : boolean

3.4 Game Entities Subsystem Interface

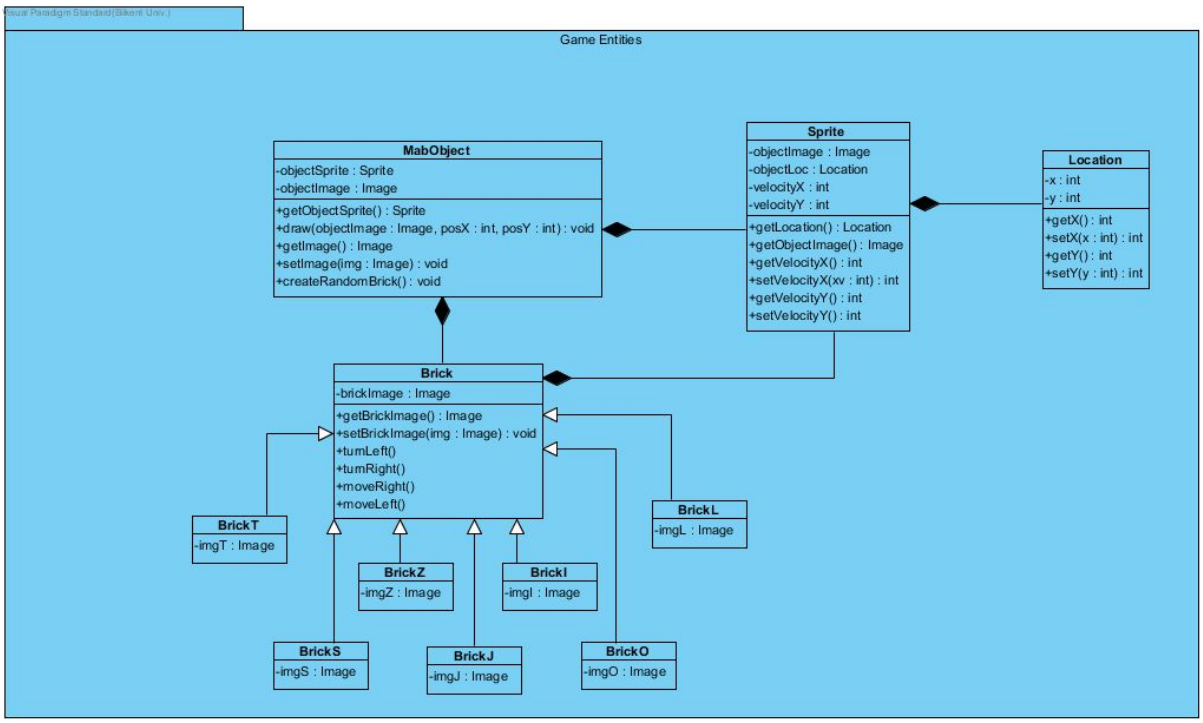
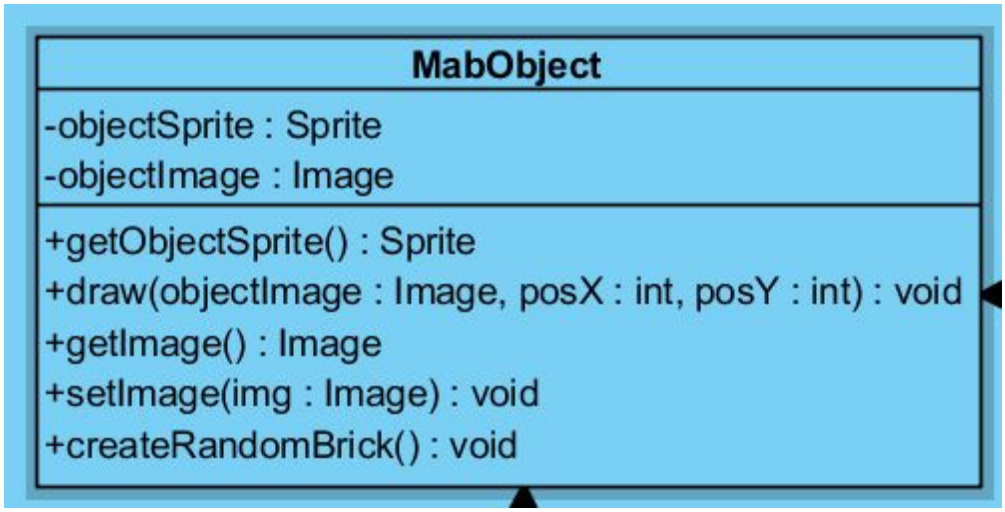


Figure-6(Game Entities Subsystem)

1. MapObject



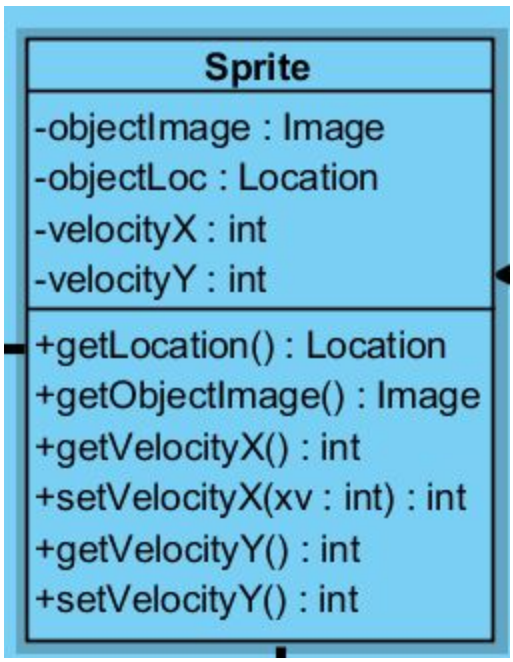
Attributes:

- -objectSprite :Sprite
- -objectImage : Image

Methods:

- +getObjectSprite() :Sprite
- +draw(objectImage : Image, posX : int, posY : int) : void
- +getImage() : Image
- +setImage(img : Image) : void
- +createRandomBrick() : void

2. Sprite



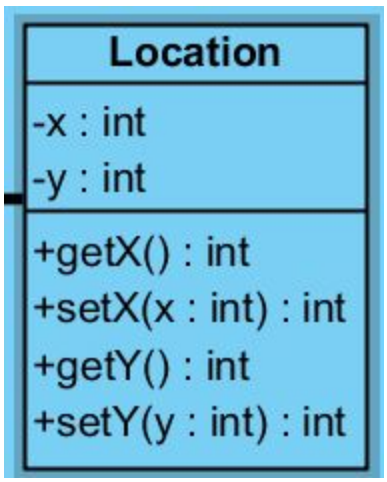
Attributes:

- -objectImage : Image
- -objectLoc : Location
- -velocityX : int
- -velocityY : int

Methods:

- +getLocation() : Location
- +getObjectImage() : Image
- +getVelocityX() : int
- +setVelocityX(xv : int) : int
- +getVelocityY() : int
- +setVelocityY() : int

3.Location



Attributes:

- -x : int
- -y : int

Constructors:

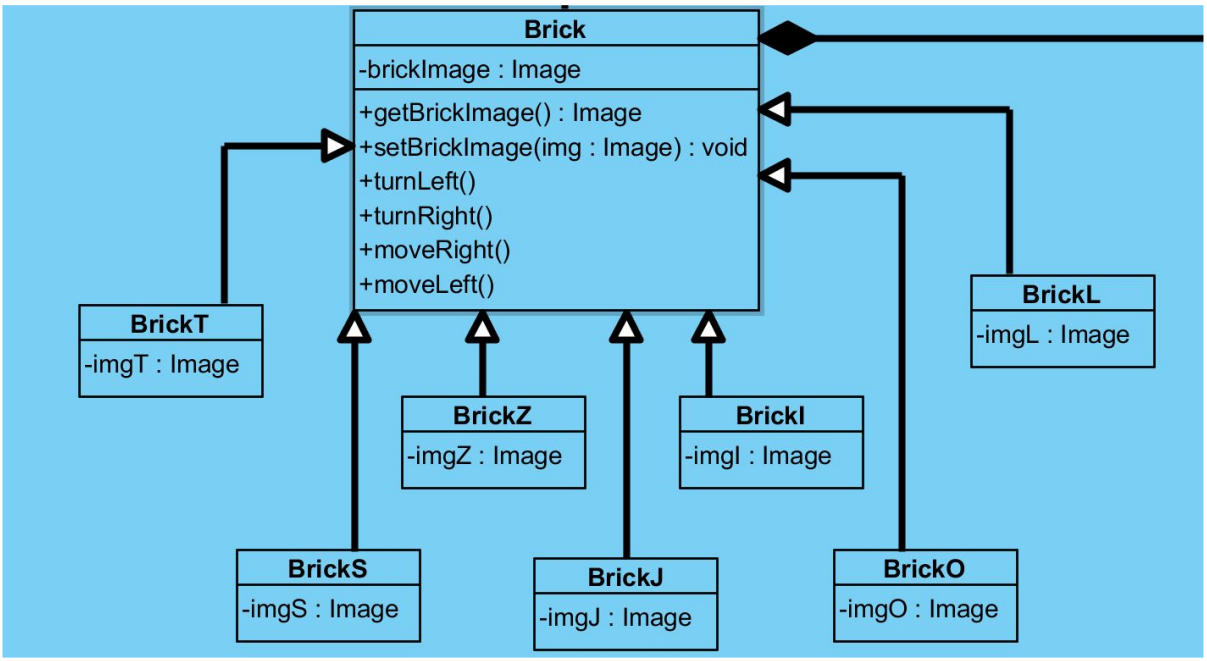
- Location(x,y)

Methods:

- +getX() : int
- +setX(x : int) : int
- +getY() : int

- +setY(y : int) : int

4. Brick



Since all of the brick objects logically works the same however their shapes are different, we will only explain a shape's attributes, constructors and methods which is similar.

Attributes:

- -brickImage : Image

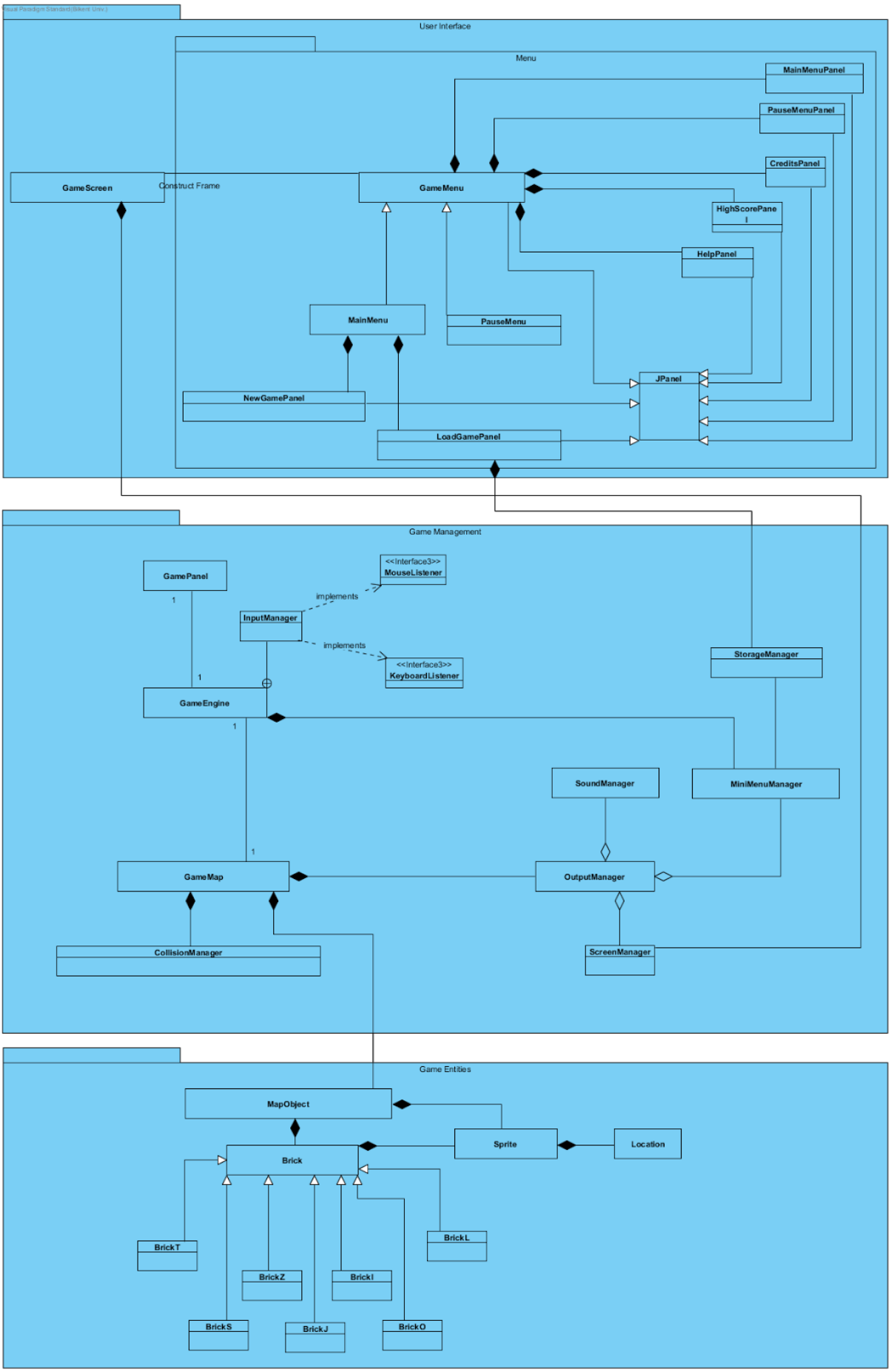
Constructors:

- Brick(Image)

Methods:

- +getBrickImage() : Image
- +setBrickImage(img : Image) : void
- +turnLeft()
- +turnRight()
- +moveRight()
- +moveLeft()

3.5 Detailed System Design


User Interface
Menu
MainMenuPanel
PauseMenuPanel
CreditsPanel
GameScreen
Construct Frame
GameMenu
HighScorePane
l
HelpPanel
MainMenu
PauseMenu
JPanel
NewGamePanel
LoadGamePanel
Game Management
GamePanel
<<Interface3>>
MouseListener
implements
InputManager
implements
<<Interface3>>
KeyboardListener
StorageManager
GameEngine
SoundManager
MiniMenuManager
GameMap
OutputManager
CollisionManager
ScreenManager
Game Entities
MapObject
Sprite
Location
Brick
BrickT
BrickL
BrickZ
BrickI
BrickS
BrickJ
BrickO