Bilkent University

Department of Computer Engineering

CS319 – Object Oriented Software Engineering Project

SuperKatamino

Design Report – First Iteration

Group 2K – Section 02

Sine Mete - 21302158
Mert Özerdem - 21300835
Hasan Doğan - 21402109
Sencer Umut Balkan - 21401911
Mustafa Azyoksul - 21501426

Instructor: Eray Tüzün

Design Report Final
December 12, 2018

This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfillment of the requirements of the Object Oriented Software Engineering CS319/3

Contents

1. Introduction

1.1 Purpose of the system

SuperKatamino is a 2-D puzzle game inspired by original board game Katamino. SuperKatamino as a puzzle game, aims to improve decision making ability, solving problems efficiently, while having pleasure of game.

Our focus group is same with original Katamino; children and teenagers, but of course, SuperKatamino is designed for everyone. Therefore, the game is portable and very user friendly in purpose of reach everyone. The design of the GUI must be entertaining and not distractive because the game is already difficult for our focus group.

The game starts with an easy level which aims to make the user familiar with game; then the levels get harder but not that difficult for our focus group as in the original Katamino. Our main purpose is to design a game which has user-friendly interface and make entertaining a puzzle game.

1.2 Design Goals

1.2.1 Usability:

Usability is a big factor in systems to keep user interested. So the game will be designed as user-friendly as possible. Therefore, the system will be designed to make users be able to play the game without any background. System will provide user-friendly menu interface, the user will be able to reach game instructions and settings easily. We are not going to use complicated controls; user will be able to perform actions with mouse events, like clicking buttons and dragging pieces.

1.2.2 Ease of Learning:

The user may not have knowledge about how the game is played, scores and logic of the game; it is fundamental for the user to obtain information about the game concepts to play the game. Therefore, we will provide a how to play screen which is easily accessible from main menu. The logic of the game is also very simple.

1.2.3 Portability:

Portability is an important issue for any software since it makes software be accessible for wide range of users. Therefore, we determined to implement the system in Java for providing cross-platform portability in any operating system in which JVM exists.

1.2.4 Reliability:

Our game will be bug-free. The system should not crash with unexpected inputs. The system will be tested after implementation.

1.2.5 Extendibility:

In general, it is important to be able to add new components and facilities to systems is important because of adapt the changes and time. Object oriented architecture of the game provides system cope with changes without causing any bugs. For example, in the future, we are planning to add more levels. We will be able to add those levels easily and without modification of few classes.

1.3 Trade-offs:

1.3.1 Functionality – Usability:

It is very important to have wide range of customers. Therefore, the system should not be too complex to play like described in usability. So, the functionality of the system will be simple, not comprehensive and various. Our priority is usability rather than functionality.

1.3.2 Reusability – Space/Performance:

Since we have already used extensive space for the pieces and levels, we did not consider reusability to not increase space and decrease performance. Also we are not planning to integrate any of our classes with another system. Therefore, we are not considering reusability in the design process of our game.

1.4 Definitions, Acronyms, and Abbreviations

JVM: [1] Java Virtual Machine
Cross-Platforms: Being runnable in the different Operating Systems like Windows, Linux, and MAC OS X.

2. High-Level Software Architecture

2.1 Subsystem Decomposition

We choose the MVC (Model-View—Controller) architecture to design our system because it is the most suitable architectural style which fits with our system; because we have 4 subsystems but two of them works parallel, we analyze our object model, we have seen that we can use MVC while we decomposing our system. We had MVC experience before and Engine subsystem behaves like controller, we have UI subsystem as a view which can't change things in other subsystems. And Object subsystem behaves like a model part. Basically, our class relationships can most fit with MVC. MVC has three interconnected parts; view, model and controller. We applied the same structure.

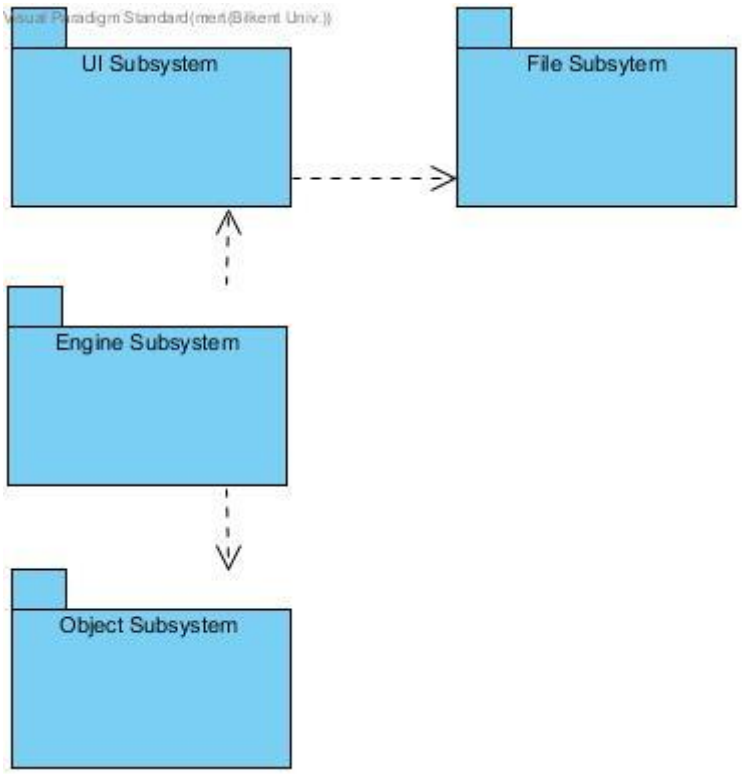


Figure 1: High-level Representation of Subsystem Decomposition

We have decomposed our system in 4 subsystems. Our UI subsystems runs as View part, Engine Subsystem runs as controller part, Object subsystem and File subsystem runs as Model part.

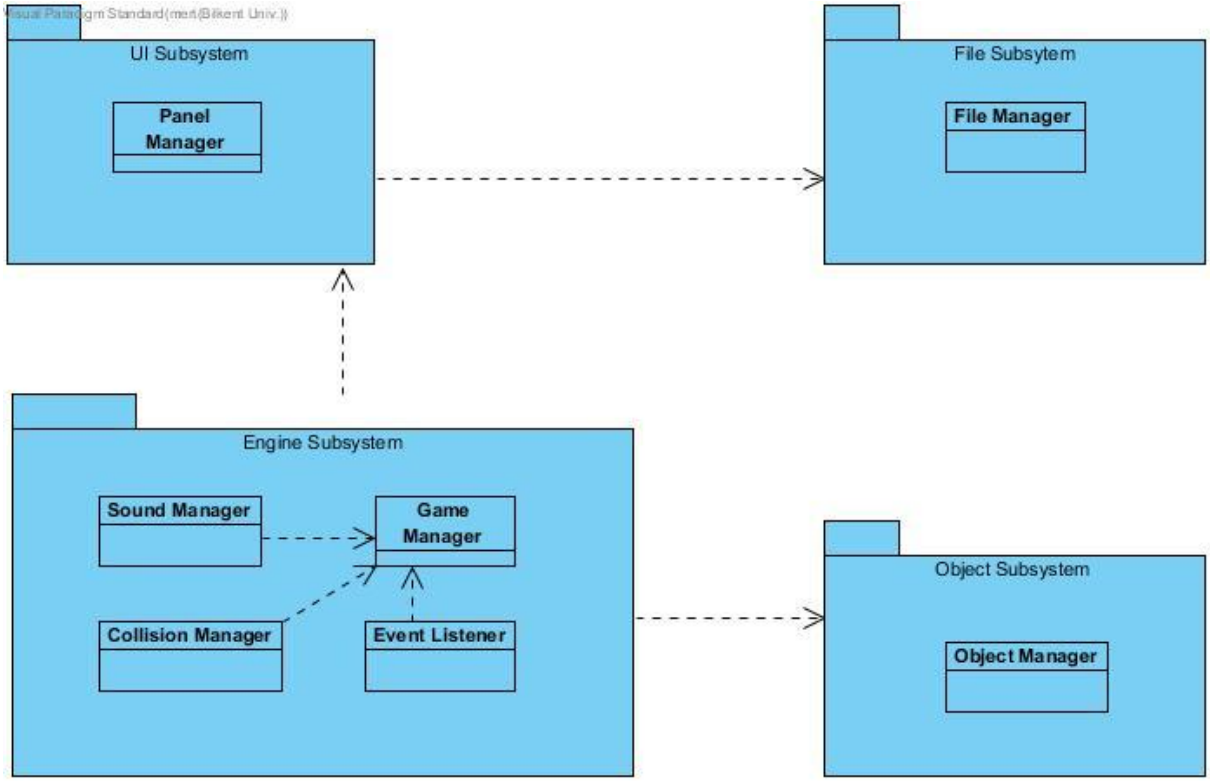


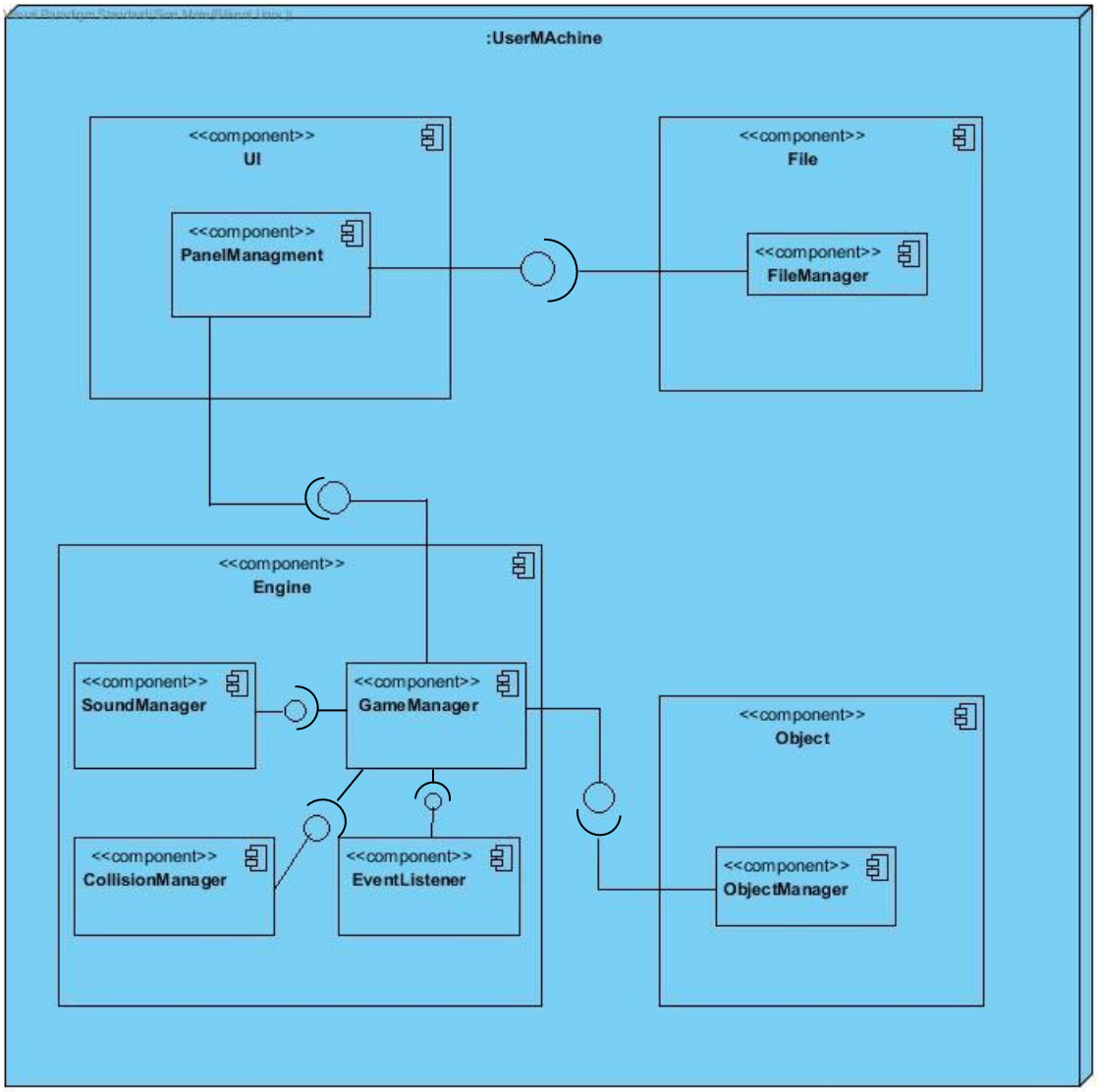
Figure 2: Detailed Subsystem Decomposition Diagram Showing Data Relations

UI Subsystem has Panel Manager managing the UI and has no dependency to the rest of the system. It includes the panels, UI objects and menu.

Engine Subsystem is the controller; main entry point of the system. The game core starts with it and it controls Object Subsystem as model by user inputs and renders. It contains GameManager, Sound Manager, CollisionManager and EventListener. Which are taking input from users like sound preferences, mouse clicks, game preferences and game moves then updates Object Subsystem.

Object Subsystem is model, contains data and takes data changes from controller Engine Subsystem then updates UI Subsystem. It has ObjectManager storing the qualities of objects like pieces and table.

2.1.1 Deployment Diagram



2.2 Hardware / Software Mapping

SuperKatamino will be implemented in Java so it will require Java Runtime Environment and latest JDK 11.0. As hardware components, it only requires mouse. As system requirements, a basic computer with basic softwares will be enough to run our game. We will use txt files to store high scores; txt files must be supported. Therefore, our system has basic requirements in terms of hardware software mapping. It does not require significant amount of memory allocation or CPU allocation.

2.3 Persistent Data Management

We are not planning to use database for SuperKatamino, since we do not plan to deal with data management with a complex database. We will need .txt files for high scores, sound effects, and image files for our game objects in user's hard disk drive, they will be used when rendering the game.

2.4. Access Control and Security

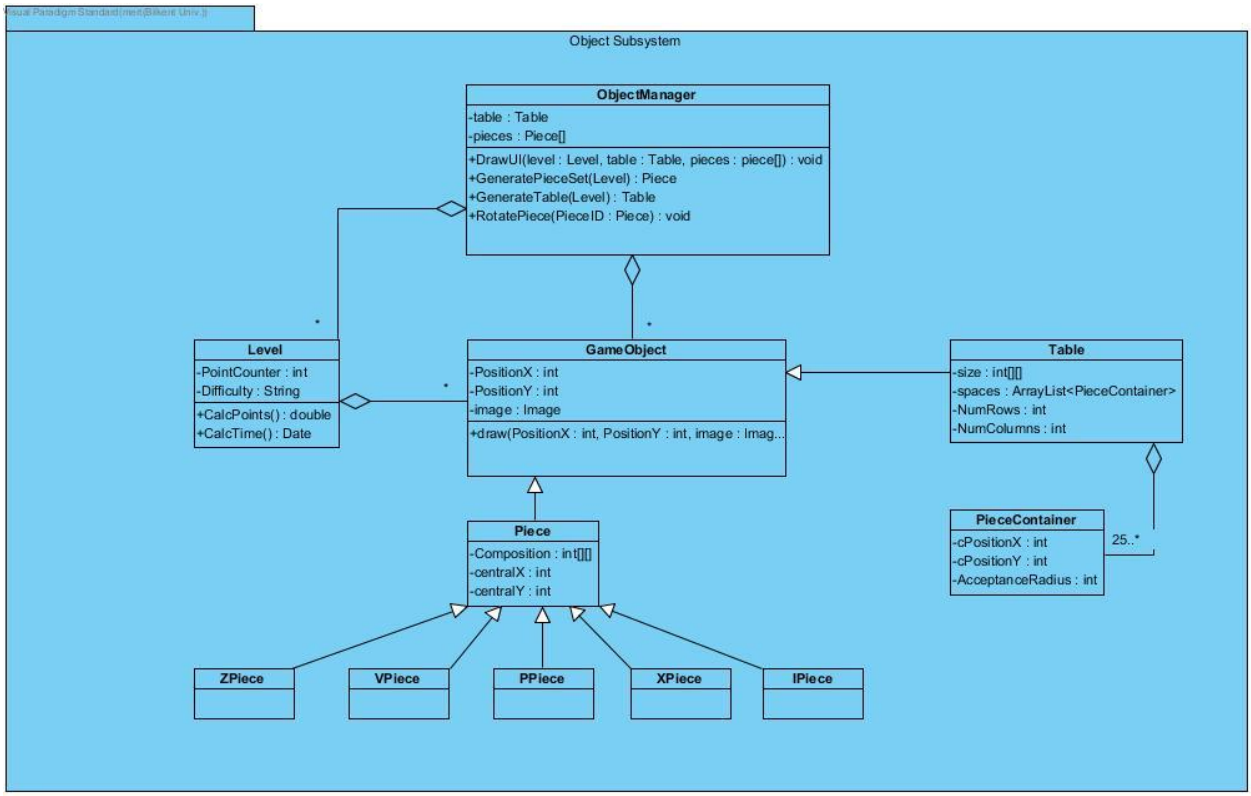
SuperKatamino does not need any internet connection or include of user profiles. Therefore, there will not be security issues in SuperKatamino. About reaching and changing files, HighScores class is the only one can reach the files on user's hard disk, this provides security because they will be private.

2.5. Boundary Conditions

SuperKatamino will be an executable .jar file. It can be terminated by clicking exit game in man menu or there will be x button on upper right of the screen. The program will be able to being terminated during the game process. If program does not respond, it will be terminated but the user data will be lost. When a game/level is finished, automatically the next level shows up. When game is finished, program returns main menu and updates high scores.

3 Subsystem Services

3.1. Object Subsystem



Level

Fields

Private **pointCounter** **int**
Each level has its own pointCounter to calculate points while player progressing through the game.

Private **difficulty** **string**
Each level has its own difficulty level to calculate points while player progressing through the game.

Methods

Public **calcPoints()** **double**
Gets “pointCounter” and “difficulty” and multiplies them in order to get level score.

Public **calcTime()** **date**
Calculates time that has passed to finish the current level.

GameObject

Fields

Private **positionX** **int**

x position vector of the object on the UI.

Private **positionY** **int**

y position vector of the object on the UI.

Private **image** **image**

Image of the object.

Methods

Public **draw(positionX: int, positionY: int, image: image)** **void**

Draws the object on the UI with the given positionX, positionY and image properties.

Table

Fields

Private **size** **int[][]**

This attribute keeps object's specific shape by utilizing 2D array structure just like matrixes: "1" for full and "0" for empty spaces.

Private **spaces** **ArrayList<PieceCounter>**

Keeps pieceCounters on the arraylist so we can add new spaces on the table while increasing its size.

Private **NumRows** **int**

Number of the rows that table has.

Private **NumColumns** **int**

Number of the columns that table has.

PieceContainer

Fields

Private **cPositionX** **int**

x position vector of the pieceContainer on the UI.

Private **cPositionY** **int**

y position vector of the pieceContainer on the UI.

Private **AcceptanceRadius** **int**

The radius of the circle that is used by container to be attached to closer piece.

Piece

Fields

Private **Composition** **int[][]**

This attribute keeps object's specific shape by utilizing 2D array structure just like matrixes: "1" for full and "0" for empty spaces.

Private **centralX** **int**

Indicates the central point of the composition on the x axis.

Private **centralY** **int**

Indicates the central point of the composition on the y axis.

ObjectManager

Fields

Private **table** **Table**

Table of generated for the level is contained here.

Private **pieces** **Piece[]**

Table of generated for the level is contained here.

Methods

Public	**DrawUI(level: Level, table: Table, peices: piece[])**	**void**

Draws the level on the UI with the given level, table and piece[] properties.

Public	**GeneratePieceSet(Level)**	**Piece**

Generates pieces for the given level property.

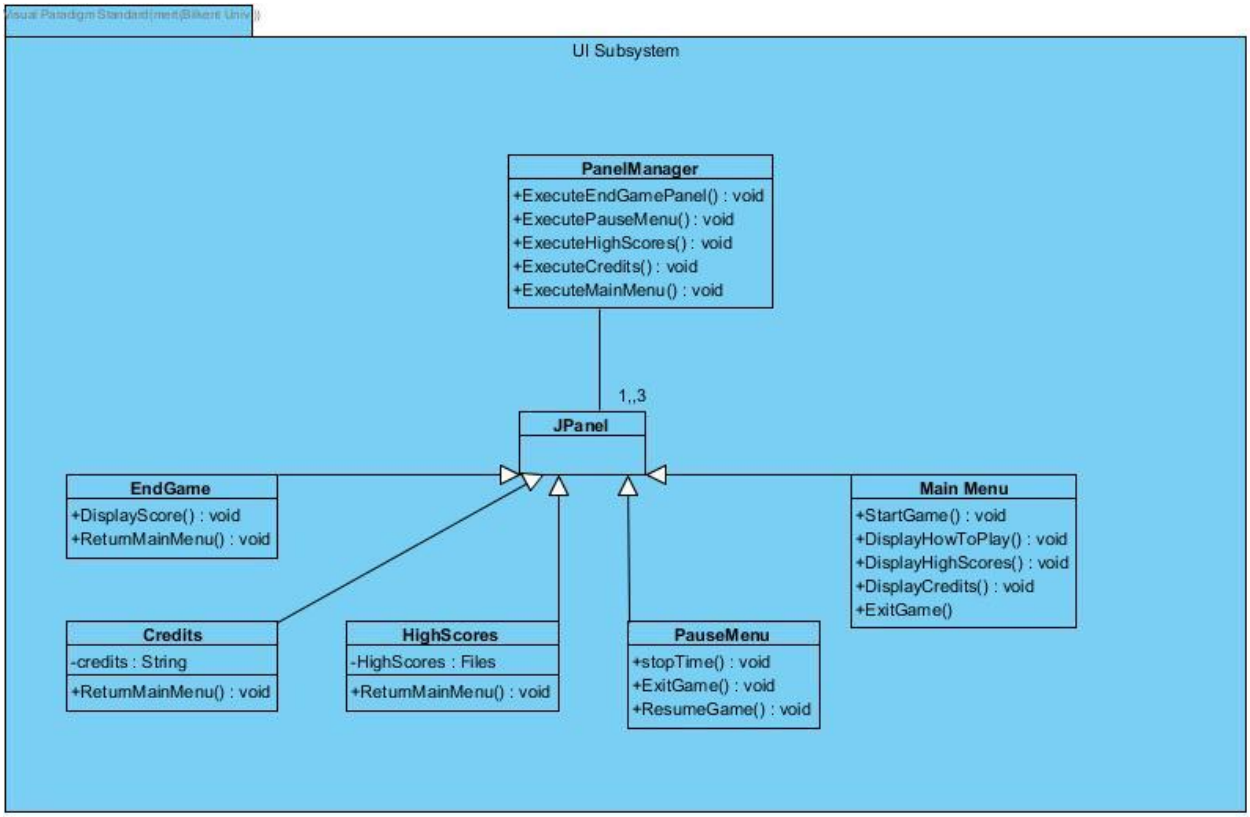
Public	**GenerateTable(Level)**	**Table**

Generates a table for the given level property.

Public	**RotatePiece(PieceID: Piece)**	**void**

Rotates desired piece 90 degrees clockwise.

3.2. UI Subsystem



EndGame

Methods

Public **DisplayScore()** **void**

Displays last total score of the player in cases s/he prematurely exits from the game or all levels are accomplished.

Public **ReturnMainMenu()** **void**

Returns to main menu.

Credits

Fields

Private **credits** **string**

Contains names of the people who worked on the project.

Methods

Public **ReturnMainMenu()** **void**

Returns to main menu.

HighScores

Fields

Private **HighScores** **Files**

Contains the file that scores are kept.

Methods

Public **ReturnMainMenu()** **void**

Returns to main menu.

PauseMenu

Methods

Public **stopTime()** **void**

This method counts the time passed while game is paused so it can deduce it from final time.

Public **ExitGame()** **void**

Exits from the game and returns to main menu.

Public **ResumeGame()** **void**

Resumes the game.

MainMenu

Methods

P**ublic** **StartGame()** **void**

Initializes the game.

Public **DisplayHowToPlay** **void**

Opens “how to play” page.

Public **DisplayHighScores** **void**

Opens “HighScores” page.

Public **DisplayCredits()** **void**

Opens “Display Credits” page.

Public **ExitGame()** **void**

Exits from the program.

PanelManager

Methods

Public	**ExecuteEndGamePanel()**	**void**

draws EndGamePanel on UI.

Public	**ExecutePauseMenu()**	**void**

draws pauseMenu panel on UI.

Public	**ExecuteHighScores()**	**void**

draws HighScores panel on UI.

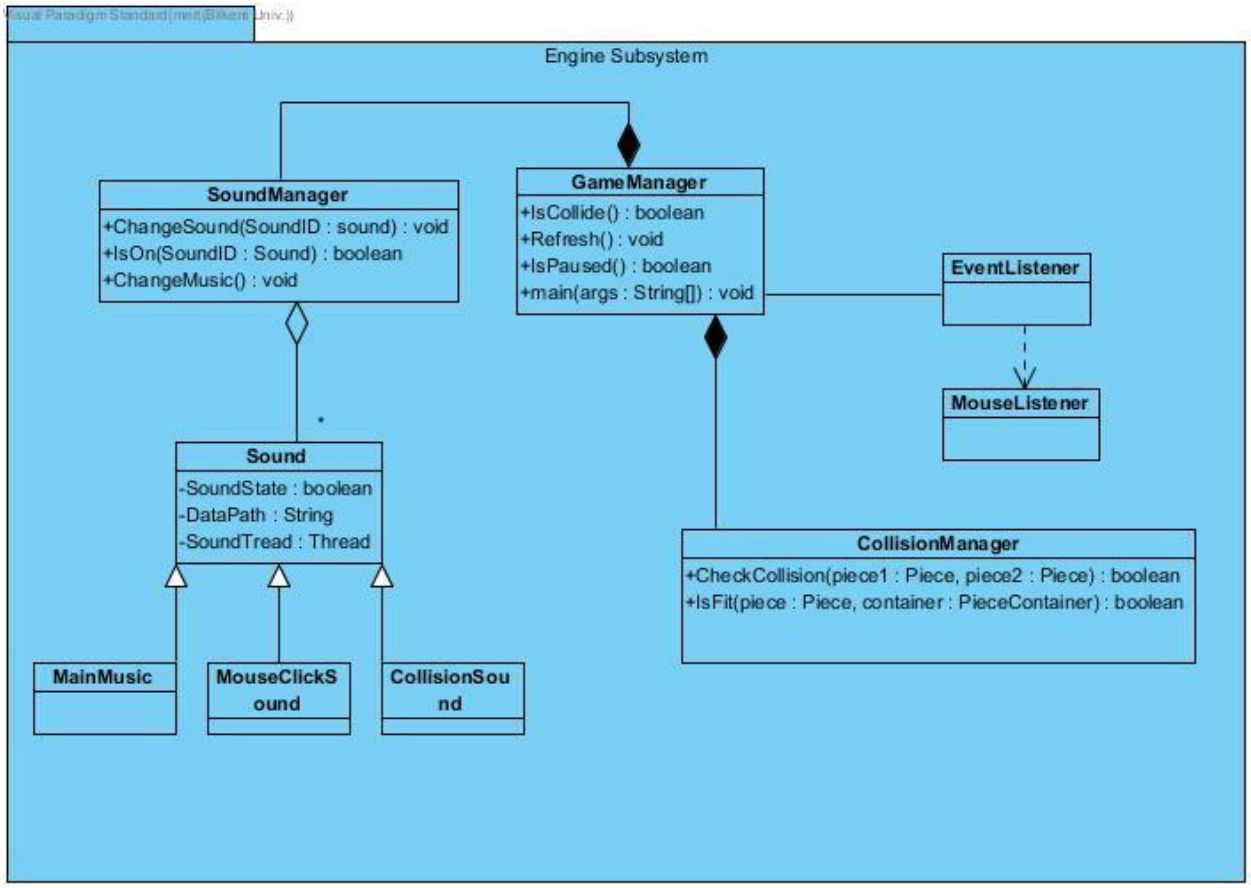
Public	**ExecuteCredits()**	**void**

draws credits panel on UI.

Public	**ExecuteMainMenu()**	**void**

draws MainMenu panel on UI.

3.3. Engine Subsystem



GameManager

Methods

Public	**IsCollide()**	**boolean**

Calls CollisionManager object so it can check collision status and fitting status of the piece object.

Public	**Refresh()**	**void**

Refreshes the game UI.

Public	**IsPaused()**	**boolean**

checks the running state of the game.

Public	**main(args: String[])**	**void**

main function of the game. Simply functions by calling IsCollide, Refresh and IsPaused functions of the game in a loop that exits until player quits the game.

SoundManager

Methods

Public	**ChangeSound(SoundID: sound)**	**void**

changes the soundState of the sound object. Changes true to false and false to true.

Public	**IsOn(SoundID: Sound)**	**boolean**

Runs a check on sound object to get its soundState.

Public	**ChangeMusic()**	**void**

Changes the main music of the game. Music circulates between 3 pre-determined musics.

Sound

Fields

Private	**SoundState**	**boolean**

If sound is playing its status is 1 else its 0.

Private	**Datapath**	**String**

Datapath of the music or sounds.

Private	**SoundTread**	**Thread**

Thread object will be used in order to play sound on different thread to get some speed up on the multicored systems.

CollisionManager

Methods

Public	**CheckCollision(piece1: Piece , piece2: Piece)**	**boolean**

checks whether or not two pieces (piece1 and piece2) are colliding. returns 1 if colliding else returns 0.

Public	**IsFit(piece: Piece, container: PieceContainer)**	**boolean**

returns 1 if the piece is on the acceptance radius of the pieceContainer object else returns 0.

3.4. File Subsystem

FileManager

Fields

Private	**scoreFile**	**File**

Keeps the file to compare and update with other users' scores.

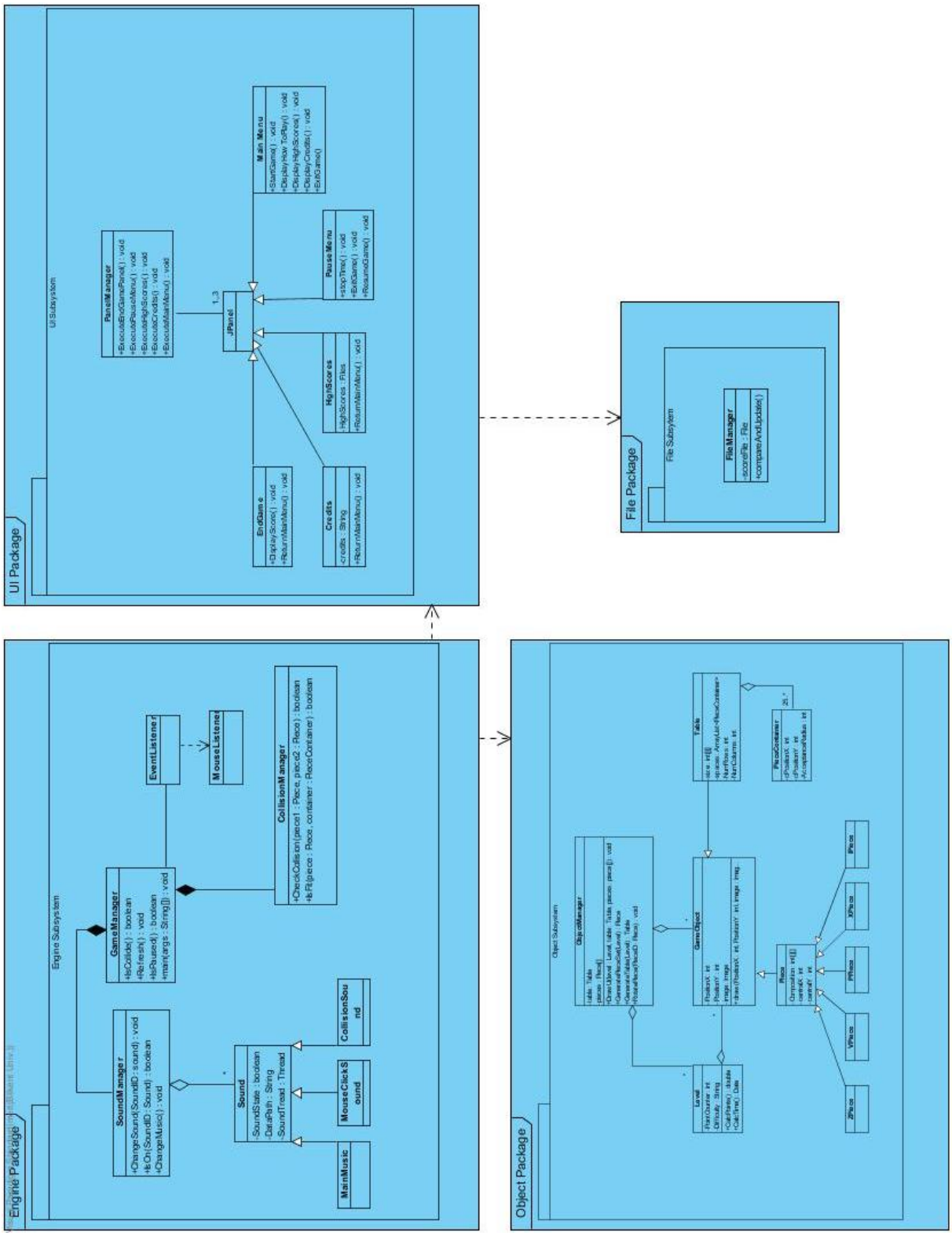
Methods

Public	**compareAndUpdate()**	**void**

Runs a check on the file and updates high scores file if current user score is higher than existing scores.

4 Low-Level Design

4.1 Final Object Design



5 References

[1] https://en.wikipedia.org/wiki/Low-level_design

[2] https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

[3] https://www.java.com/what-is-java/