CS 319 - Object-Oriented Software Engineering

Design Report

IQPuzzler Pro

Group 2J

Burak Alaydin
Kaan Altinay
Muhammad Umair Ahmed
Aliyu Saifullah Vandana
Skerd Xhafa
Supervisor: Eray Tuzun
TA: Mohammed Cavusoglu

Contents

1. Introduction

1.1 Purpose of the System

The objective of this term project is to recreate on a digital platform a version of IQPuzzler Pro, the popular children's learning game. The physical game involves a set of ball-and-stick pieces of different colors and arrangements that have to be arranged in a set pattern identical to one of the predefined "challenge" patterns on a board full of spherical sockets. If the player is able to recreate the pattern on the board from a challenge, then they win that challenge. There are 3 different board socket patterns in the game: standard rectangular and dual-diamond-shaped, both of which are used for the style of challenges mentioned above. The third pattern is a square of sockets which involves a very different challenge: constructing a pyramid with the given "pieces". Taking these core principles, we have set out to create a digital version of the game and adding our own twist to it.

1.2 Design Goals

As with all engineering solutions, the system to be developed should also have prioritized goals and should be able to optimize its resources to achieve them. As the game to be developed is a computer game, it should be able to entertain casual computer users who are not computer-savvy. As in all games, this game should focus on providing good gaming experience.

The best and appropriate software engineering methods should be followed in the development of this game.

Usability: As in all engineering solutions, our will system will have prioritized goals and optimized its resources in order to achieve such goals. Our game will be developed in such a way that it will entertain casual computer users who are not tech-savvy. The user will be given an option among the options menu which can be used to learn about how to play the game. The logic of the game will be kept grounded as it is without fancy inclusions to overwhelm the user. Most of the works on the game can be done with mouse click and sometimes if the user prefers, a keyboard shortcut.

Extendibility: In general, in the lifetime of game software, it is always important to add new components, features to the game in order to sustain the excitement and interest of the player. In this respect our design will be suitable to add new functionalities, entities (i.e. new pieces, new levels) easily to the existing system.

Modifiability: In order to provide different gaming experience to players, modifiability is a key factor. With high degree of modifiability developers can add new content easily. Games sub system will have hierarchical structure so that developers can add new features easily

without much changes in the upper system. Code will be written in a clear and understandable manner.

High Performance: A high performance is especially important for computer games since they usually demand more resources to be used than other programs and users quickly recognize the lack of performance in a game and get bothered. Since the main purpose of the game is to make the users enjoy and have fun, a high performance is a must have quality. As a result, our system will be designed in such a way that it will respond within an average of 2 seconds.

Portability: Portability is an integral part of a software , as it provides that the software can be accessed by a wide range of users on different platforms. With Java as our language of implementation this can be achieved in a much easier fashion. This is because translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only the JVM (Java Virtual Machine) needs to be implemented for each platform.

1.2.1 Trade-Offs

As expected, all optimization problems come with drawbacks. Although we want to minimize such drawbacks and provide great gaming experience, there however, are expected sacrifices. Some of them are below:

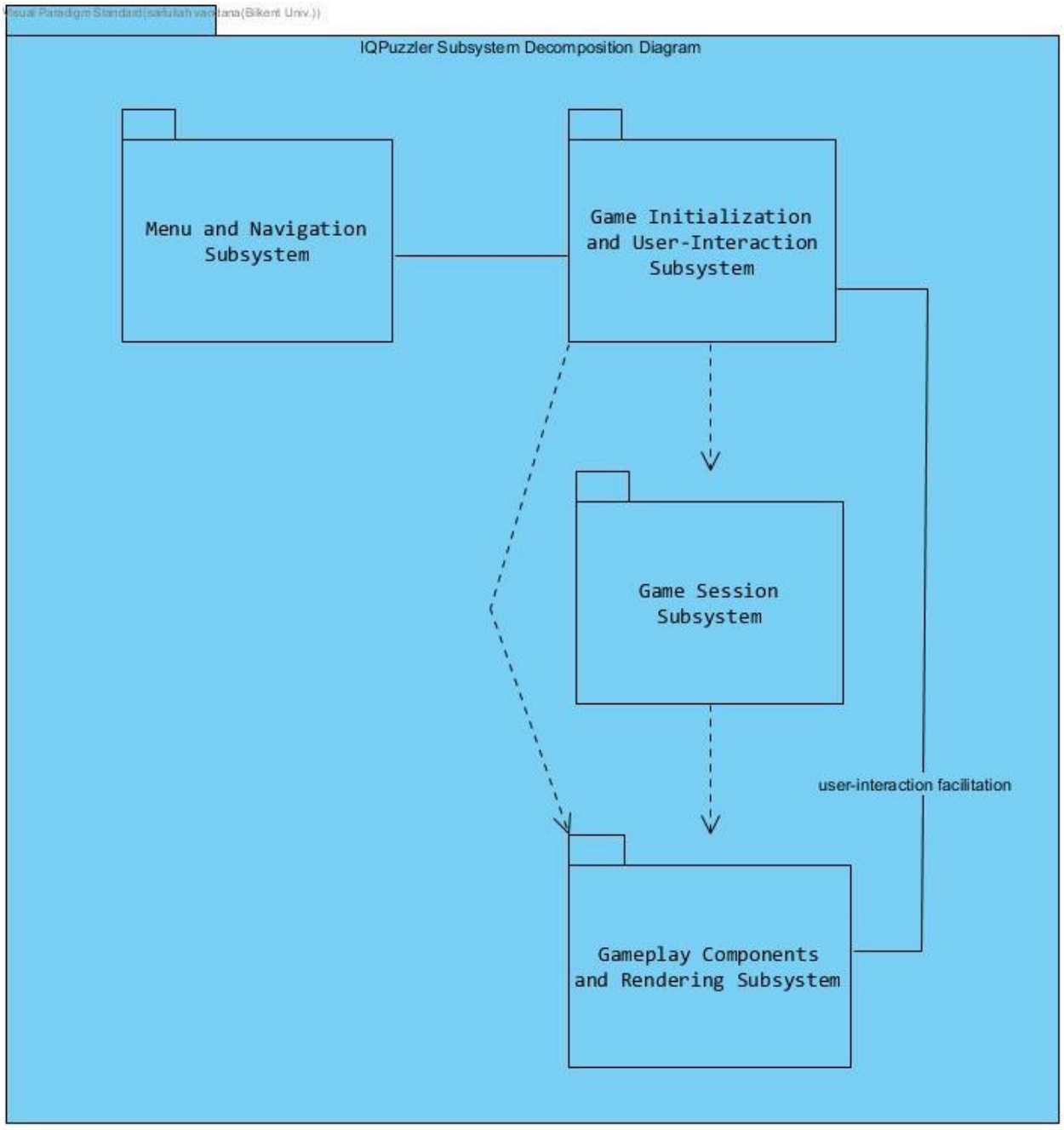
Functionality vs Ease of Learning and Use: The system as said earlier will be designed to address all user groups ranging from novice to tech-savvy individuals. For this reason, it is inevitable that some rather a bit complicated features will be left out of the design to achieve the desirable simplicity. We can either have a system with many functions but also complex, or simple but relatively basic and grounded. To fit our design goals, it is only natural to go with the latter.

Rapid Development vs Minimum errors: The development of a program starts from its initial design steps to the last testing. Thus, any effort that may directly or indirectly extend this time interval will be conflicting with rapid development of the program. The system follows a top-down hierarchical design approach. Rapid development on the other hand focuses more on the implementation rather than organization. Our system will be designed such that much attention will be given to error minimization from design to testing.

Memory vs Performance: Since our game is designed with object-oriented principles, much concern will not be given to the memory usage. Performance of our system is our primary focus. For this purpose, we sacrificed the memory in order to gain the performance. This allows us to gain run time efficiency in the design, by creating, accessing and changing every object on the go.

2. Software Architecture

2.1 Subsystem Decomposition



We have a partially layered sub-system model where 3 out of 5 subsystems are in a layered structure. The Menu and Navigation subsystem is relatively independent as it hands control over to the other subsystems whenever a game of any kind is loaded. The Data Manager subsystem will be called every time a game or game score is loaded or saved and every time the Story Mode of the game is played. The remaining 3 subsystem are in constant communication with each other once they are up and running.

Menu and Navigation Subsystem:

This subsystem will jump into action as soon as the user runs the game. It will handle the entire main menu and all the possible navigational paths from there. It can be used to set the parameters of a new game or to load a game, at which point the next subsystem will take over.

Game Initialization and User-Interaction Subsystem:

This subsystem will handle the initialization of a game session. Additionally, it will serve as the gateway for user interaction with game components including all kinds of mouse movements such as clicks, presses and drags as well as keyboard input such as key presses and key holding states. This subsystem depends on the Gameplay and Components and Rendering Subsystem for facilitating interaction that is meaningful and on the Game Session Subsystem to initialize the former.

Game Session Subsystem:

This subsystem will initialize the game components that are secondary to the gameplay such as the countdown timer and the scoring manager. Additionally, its components depend on the next subsystem that this subsystem will call into action in order to function meaningfully. The next subsystem is responsible for handling core gameplay components.

Gameplay Components and Rendering Subsystem:

This subsystem will initialize and deal with all the core game components such as the Board, and the puzzle Pieces including their state changes and their rendering on-screen. This subsystem is in constant communication with the User-Interaction Subsystem during gameplay.

Data Management Subsystem:

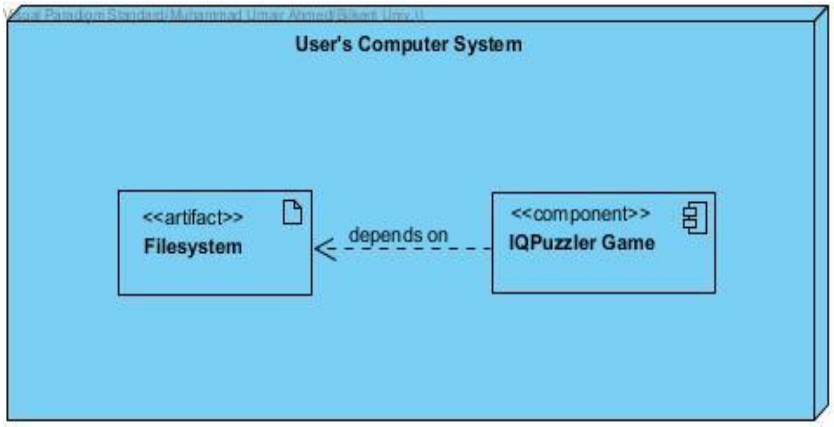
This subsystem handles all game-to-filesystem and filesystem-to-game communication including saving, and loading scores, saving and loading games and loading Story Mode contents, both challenges and story plot reveals.

2.2 Hardware/Software Mappin

With as simple logic as found in our game, it will not require any specific purpose-built hardware in order to run. It can be carried out by the general-purpose hardware. To play our game, the user will need a pc, a keyboard, a mouse, and a monitor. The game will run in various Operating Systems as Linux, Mac, and Windows environments which has Java Runtime Environment(JRA) since we will develop the game using java programming language. For storage such as high scores and user settings, we will use file system. As a result, there would not be any need for a database or an internet connection.

Deployment Diagram

This diagram is fairly simple as our game as a system does not involve multiple nodes or components and artifacts for that matter. Our game will, however, be dependent on the filesystem of whatever computer system it will be run on in order to successfully save and load games, save and load scores and to load Story Mode game elements such as challenges and story plot developments. This dependency is illustrated by the dotted arrow connecting IQPuzzler Game with Filesystem in the diagram**.**



2.3 Persistent Data Management

In our we do not require any complex data storage system, as such for managing persistent data, text files will be used due to their simplicity. Some of the text files like pattern designs will be instantiated during the implementation stage and these files cannot be modified by the user. On the other hand, some data members like user setting preferences, sound settings, and highest score will be kept in a modifiable text files, therefore user can change or update the values of these files during game time. With text files the operation of reading and writing is done during the game and does not require requesting queries.

2.4 Access Control and Security

In our game there is only one actor involved which is the single player so there is no need for access control except for the high scores which

cannot be modified by the user. For that an actor such as the administrator will be in charge of it. Any other function that is offered by the system should be accessible by the user. There may profile set-up which requires only the name of the user which is no crucial data that can be corrupted by the user. Since this system is for games and the convention for games is that all profiles are open to all gamers, there is no call for access control. The one actor which happens to be the user has all privileges and can access all methods some of which are implicitly invoked by other objects but then again, the user starts the chain of invocation.

2.5 Boundary Conditions

2.5.1 Initialization

IQPuzzler Pro will not require any procedural installation. The game will come with an executable .jar file. At the beginning of the initialization, gamer profiles, sounds, level information and stories are read from files and objects are created. The first screen presented to gamer is the opening animation logo. After this short opening animation; main menu screen is presented to the gamer. Main menu screen has a set of buttons associated with different functionalities of the system: Starting a game, Loading a game, settings etc. In story mode the theme of interface is coherent with the story and the background of the game.

2.5.2 Termination

The user can terminate the game by pressing the exit button from the main screen. Upon clicking the exit button, user will be asked to save the game data. In another instance, the user can exit by pressing Alt+F4 or the exit windows button. In all cases the user will be asked to save the game data before leaving.

2.5.3 Failure

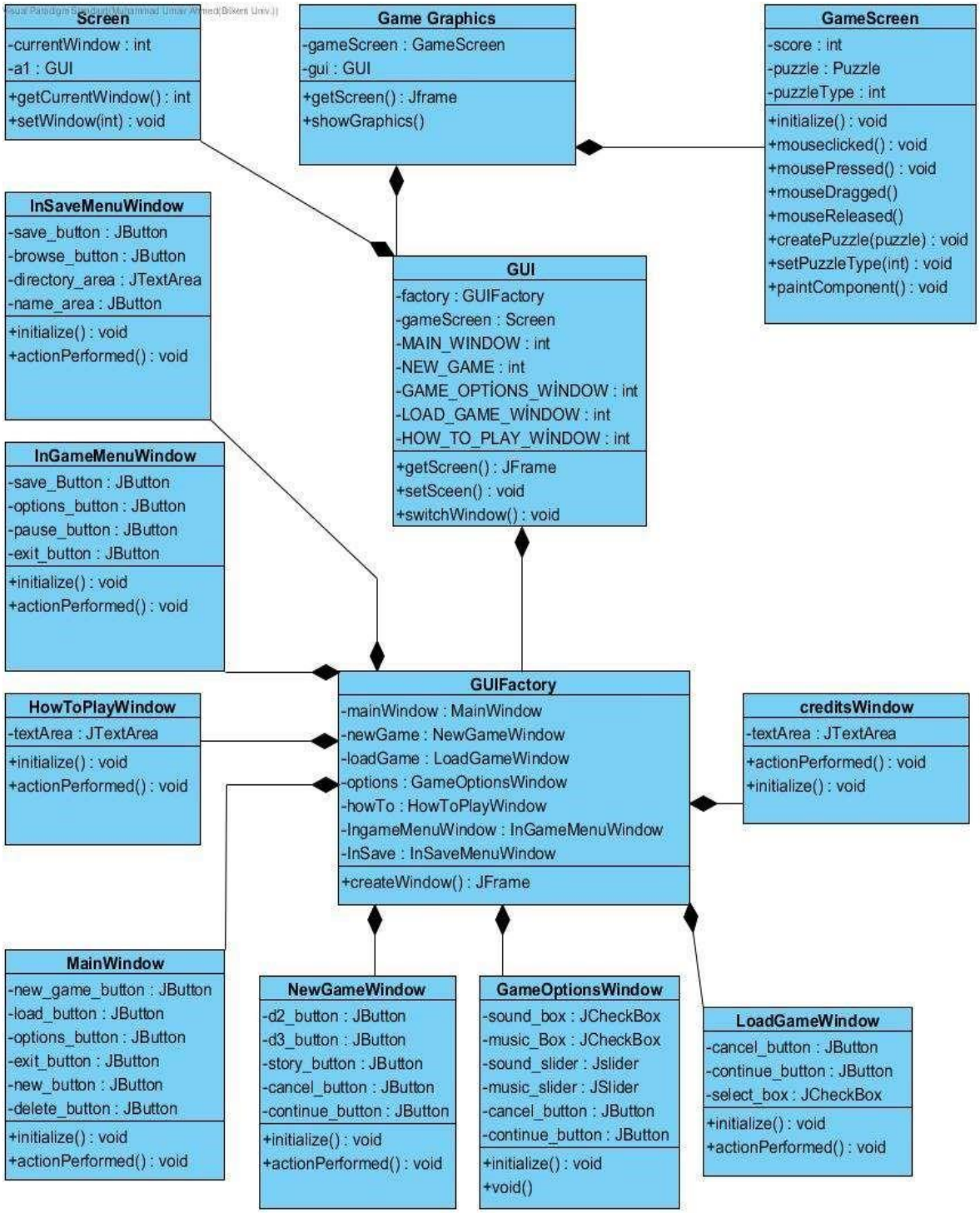
The data of the current session is lost in case of a sudden failure of the system, because data update to the file system occurs at termination of overall system. However, older files are not changed so they may be assumed to be a kind of back up. It is quite frustrating and upsetting to have an unexpected termination or loss of data but since a drawback in performance is far more upsetting than the loss of points gained in a session; there will be no regular write back operations or autosaving which will be sacrificed for better performance. Therefore, the next execution after a failure is similar to any execution of the program.

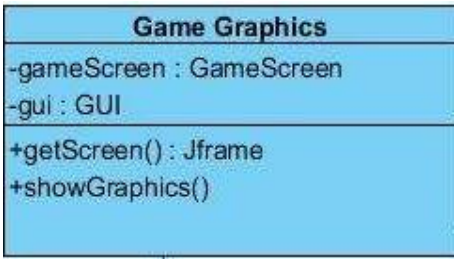
3. Subsystem Services

3.1 User-Interface Subsystem

The user interface provides the graphical system components of our software. The diagram below shows how the various components of the

user interface relate to each other in our game design. The individual classes will be delineated in the subsequent pages.

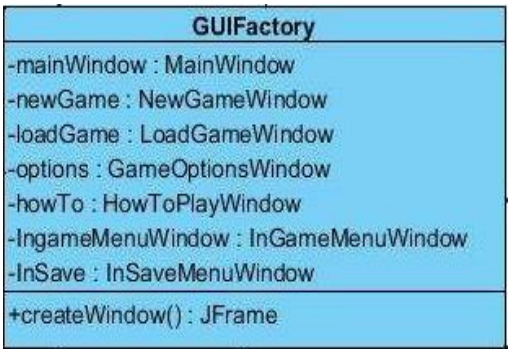




Game Graphics: This component is responsible for all graphics that will be seen in the game. It creates the windows that are shown at the initialization of the game and the windows that are guide the player till the end of the game as well as the battle screen. This component is divided into the subcomponent GUI, ChatBox and BattleScreen; each of which provides services to GameGraphics. This component manifests the graphics that are sending by Server in a package. Any graphics, requested in the package, is created thanks to the subcomponents and shown in the screen.

GUI: This component Component provides services for creating and handling the windows that are guides the initialize game procedure. It gets a window id and then sends the requested window to gameScreen, which is a Screen Object. To do this, it simply uses the services of a Factory Design

Pattern, which is named as GUIFactory. Its methods switchWindow(int windowID) :void simply call the createWindow(int windowID): JFrame service of GUIFactory component. Thus, it gets the window by its id and does not stress about how to create the requested window. Class Screen is a JFrame that holds the currently visible window. Its setWindow() method gets the frame and shows it.



GUIFactory: This component is a Factory design pattern that creates the Frame that requested by its id. It creates the desired Frame by services of its subcomponents then it returns the desired Frame. Its subcomponents are MainWindow, NewGameWindow, LoadGameWindow, CreditsWindow, GameOptionsWindow, InGameMenuWindow, InSaveMenuWindow.

Class MainWindow is a JFrame that holds the graphics shown at the beginning of the game. It simply let the user to select a service among the following options: new game, load game, options, help or exit. This class also handles the required action caused by the selected option.

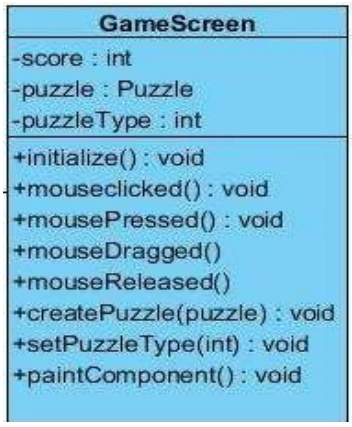
Class NewGameWindow is a JFrame that holds the graphics shown to let user to create a new game. It lets user to join a game or to host the game. In case of host the game, it gets map type and gold amount from the user; in the join game case, it gets IP and port numbers from the user.

Class LoadGameWindow is a JFrame that holds the graphics shown to let user to load a previously saved game. It, too, lets user to join the loading game or to host the loading game. In case of host the game, it gets file name from the user; in the join game case, it gets server computers IP and port numbers from the user.

Class GameOptionsWindow is a JFrame that holds the graphics shown to let user to enable/disable music and sound. If user enables any of music or sound, it is allowed to adjust the amount of them.

Class InGameMenuWindow is a JFrame that holds the graphics shown to let user to exit from the game, to pause, to save or to change the options of the game. Class InSaveMenuWindow is a JFrame that holds the graphics shown to let user to save the current position of the game. It gets the directory, where the saved game will be placed, and the name for saved game from the user.

Class CreditsWindow is a JFrame that holds the texts shown to inform user about the designers of the game.



GameScreen: This component provides services to illustrate the game. It is responsible to illustrate the puzzle seen that has been sent to it in a group of shapes. According to puzzle states, it illustrates the game by using three layers to show the graphics. For drawing of each layer, there is a dedicated paint method as:

paintMap(x : int, y : int) : void paints layer 1 (map)

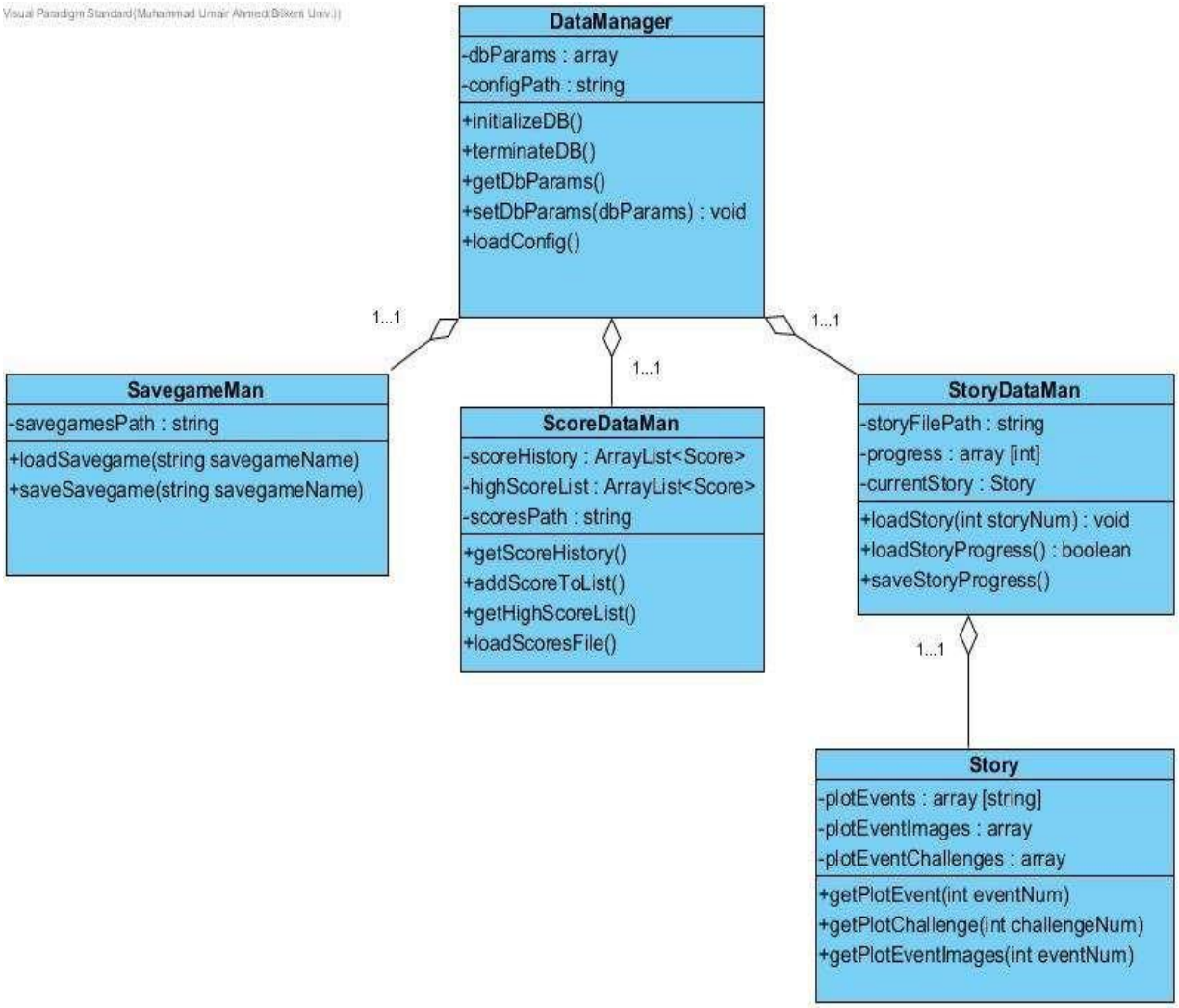
paintPieceSelectedColors(shape):void paints layer 2 (piece and Selections Color)

paintShapes(Puzzle):void paints layer 3 (draws each shape icon)

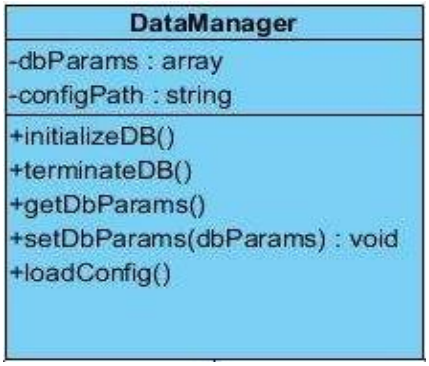
3.2 Game Management Subsystem

The management system is responsible for controlling and managing the game. Currently it consists of 5 major components namely: DataManager, SavegameManager, ScoreDataManager, StoryDataManager

and Story. The diagram below shows the different management systems we have and how they are related.



DataManager:



The DataManager class is responsible for storing all the data relating to the game. Data such as puzzle pieces, highscores, story events etc. will be kept in text files. Upon the start of the game, information in this file will be used to initialize these objects. Therefore, to handle all data interactions, the DataManager is needed. It traverses all game data that is stored in the files at the beginning of the game and loads them.

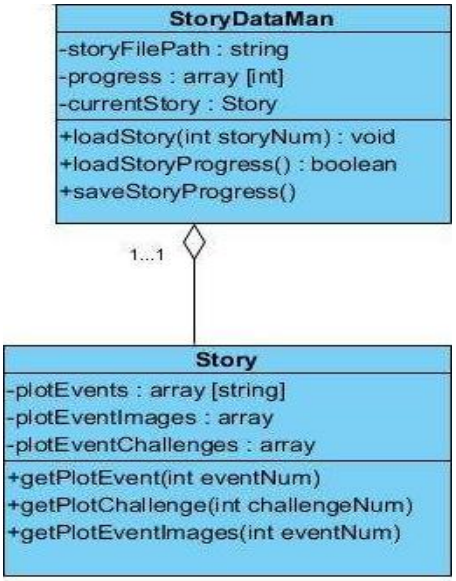
SavegameManager:

This class is responsible for saving the and loading the game. Upon the start of the game, if the user wants to continue playing a game in progress, it can be loaded with the help of this class. Upon closing a game session, the user can be able to save the progress made.

ScoreDataManager:

The score data manager handles the scoring system of the game. It has the history of scores for each player and the highscore attained at a particular level. When the user surpasses his own highscore it is loaded and updated by this class.

StoryDataManager:



The story data manager class oversees the story mode of the game. Plot events and challenges are updated within this class. For each level

progress in a story mode, a new story event is revealed by this class which follows up till the end of the story when the user finished all the related challenges.

4. Low-level Design

4.1 Design Patterns

We have used Façade design pattern which is a structural design pattern which proposes that developers can easily manage a subsystem from a façade class since the communication between outside of this subsystem is performed only by this class. This pattern, provides maintainability, reusability and extendibility since any change in the components of this subsystem can be reflected by making changes in the façade class.

In our design we used façade pattern in two subsystems: User interface subsystem and Data management subsystem. In Data Management subsystem our façade class is DataManager which communicates with the components of the Data Management subsystem according to the requests of User Interface subsystem.

4.2 Final Object Design

Below is the complete class diagram for our game. The sub classes will be delineated in the subsequent pages.

Figure 1

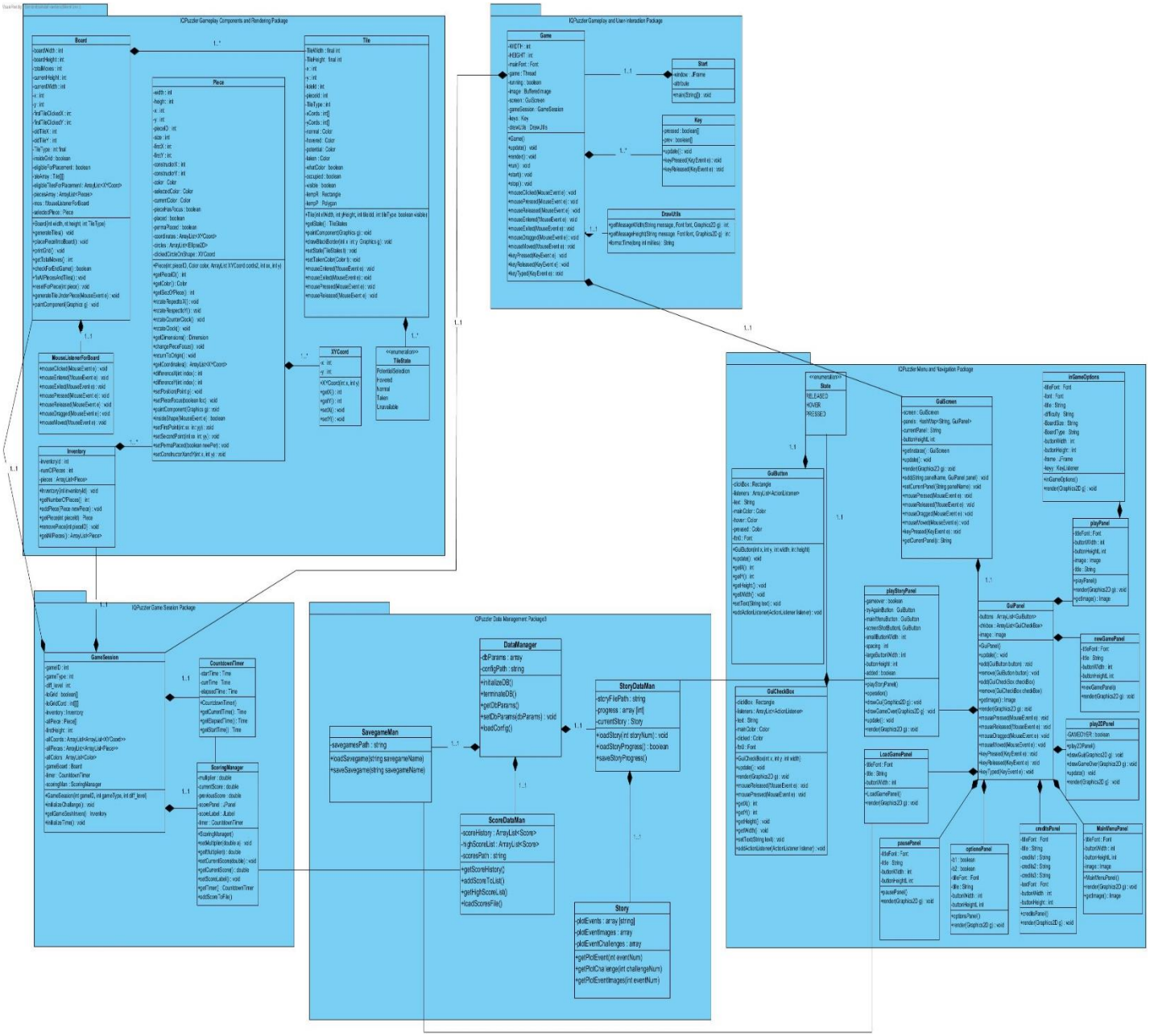


Figure 2

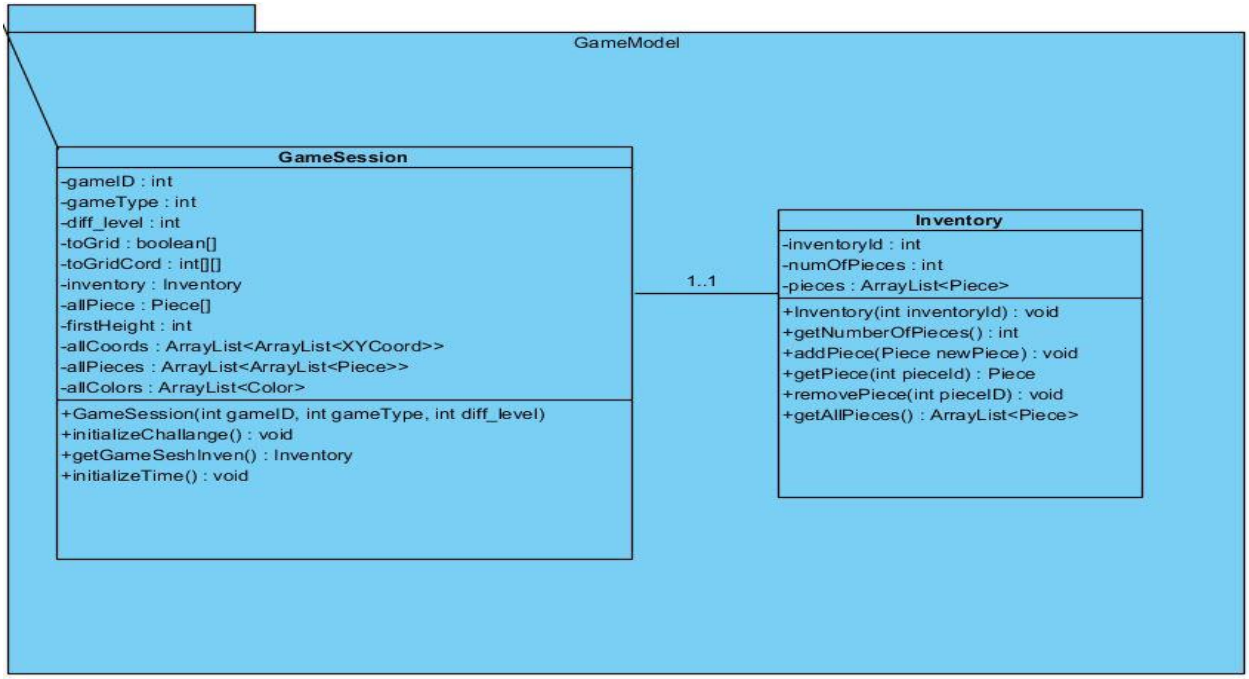


Figure 3

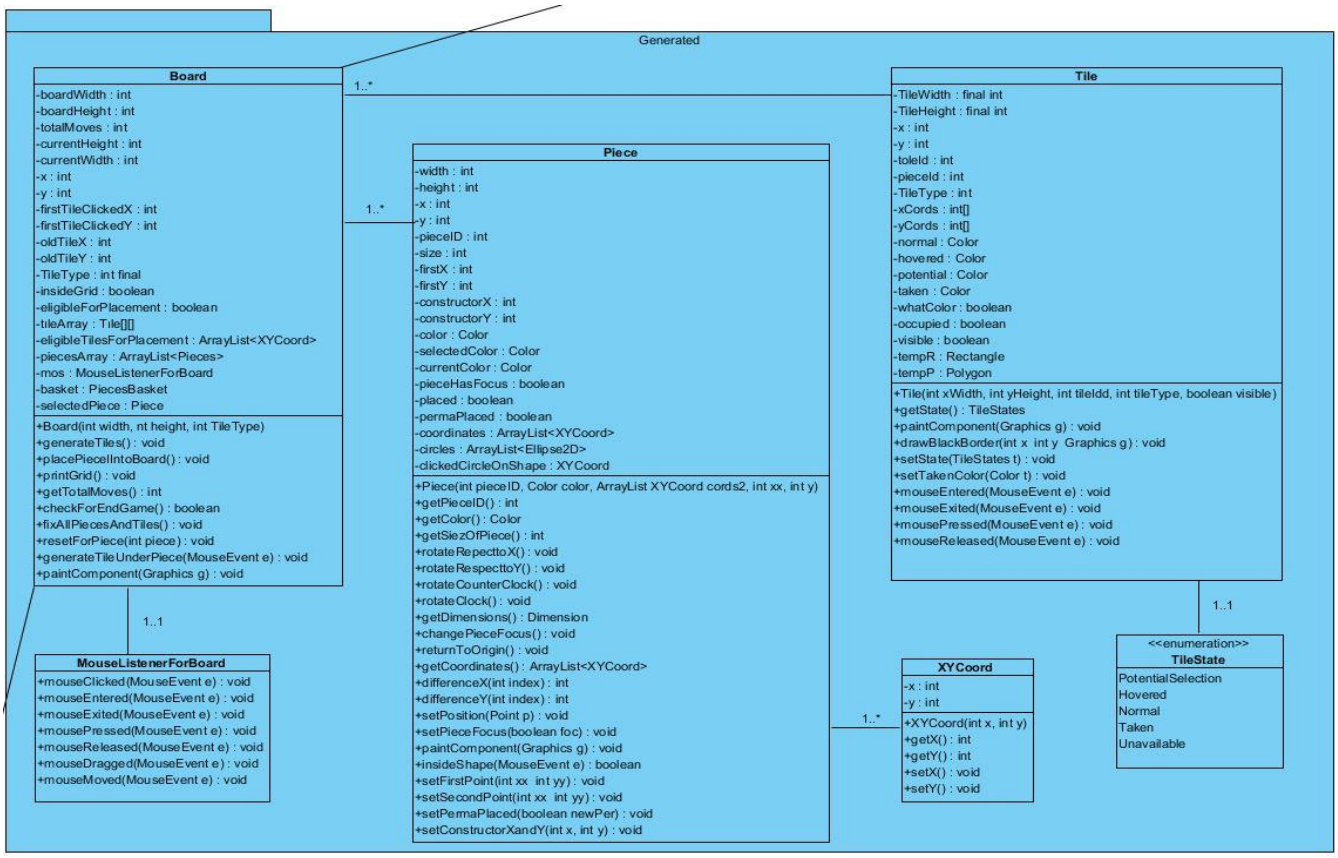
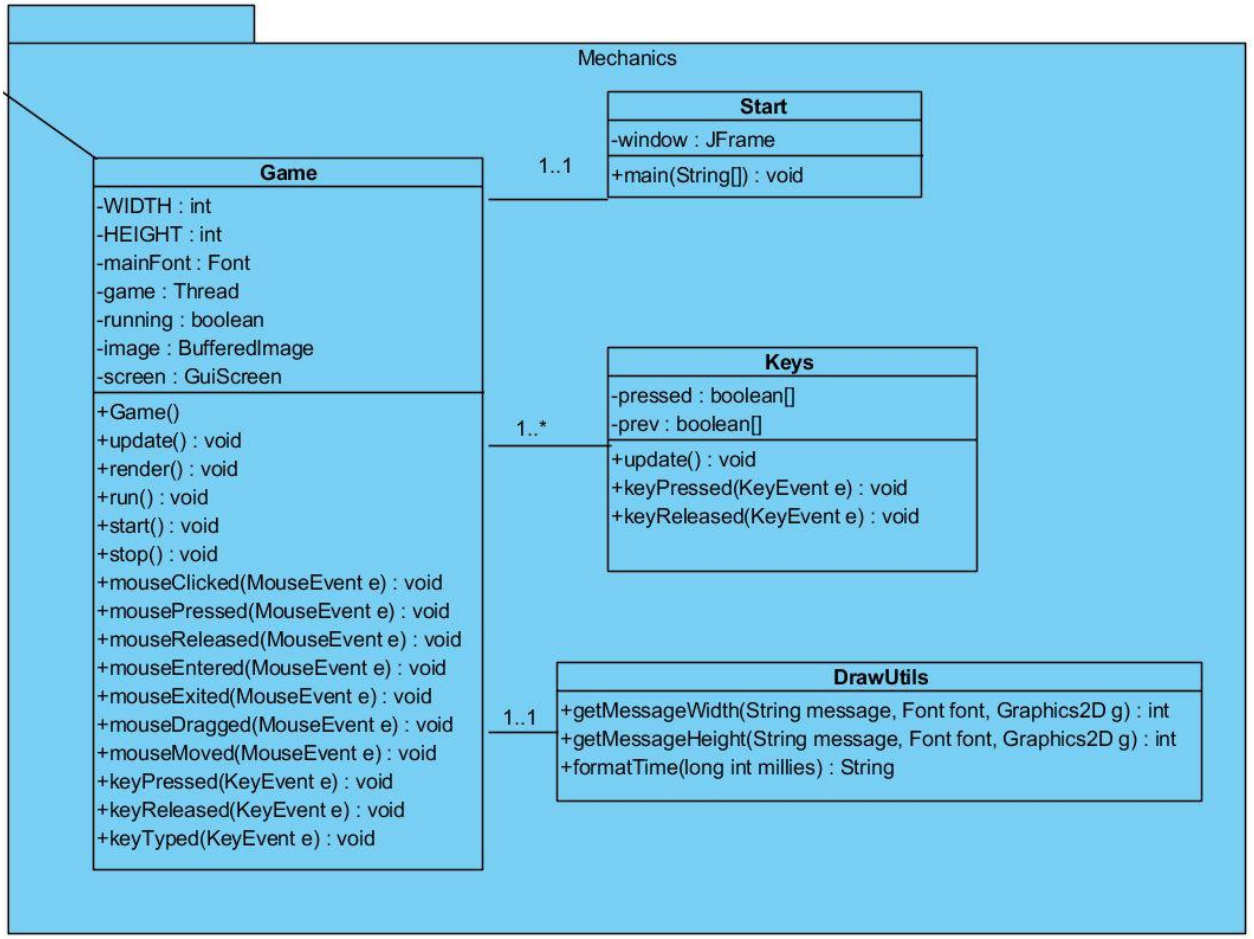
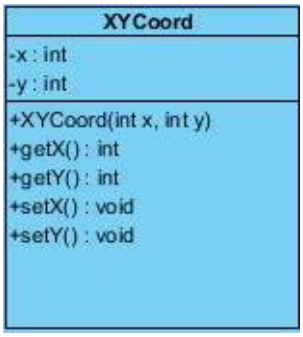


Figure 4



4.3 Class Interfaces

4.3.1 XYCoord



Attributes:

- **private int x:** this represents the x coordinate of a piece.
- **private int y:** this represents the y coordinate of a piece.

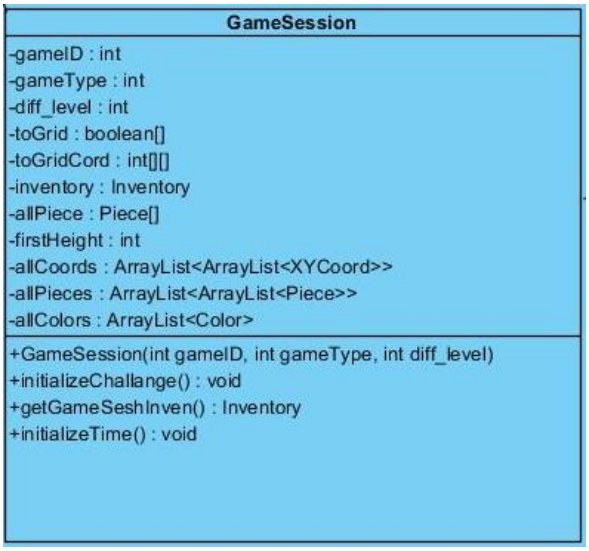
Constructors:

- **XYCoord(int x, int y):** this is the default constructor of the XY coordinate class.

Methods:

- **public int getX():** this method returns the value of x coordinate.
- **public int getY():** this method returns the value of y coordinate.
- **public void setX():** this method updates the value of x coordinate.
- **public void setY():** this method updates the value of y coordinate.

4.3.2 GameSession



Attributes:

- **private ArrayList<XYCoord> allCoords:** this holds the list of all coordinates of the pieces in a game session
- **private ArrayList<Pieces> allPieces:** It holds the entire pieces that will be played in a game session.
- **private int gameID:** It holds the game Id.
- **private int gameType:** this specifies the game type based on the user's preference
- **private int diff_level:** this holds the key for a different level
- **private boolean toGrid:** this holds the value which determines whether we can place the piece on the grid or not.
- **private Grid2D grid:** this initializes the grid for a game session
- **private Board gameboard:** this initializes the board for a game session.
- **private Inventory inventory:** this initializes the inventory or a game session.
- **private Piece selecetedPiece:** this represents the piece selected by the user.
- **private int selectedPiece index:** this returns the index of the selected piece.
- **private int firstHeight:** this returns the value for the first position when clicking the mouse.

- **private ArrayList<Colors> allColors:** this is the colors of the pieces.

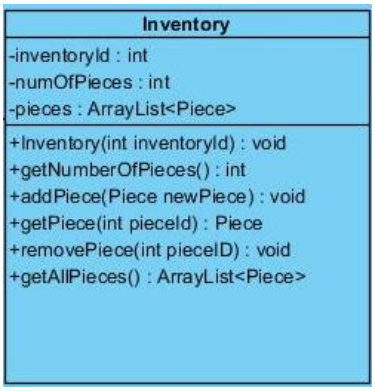
Constructors:

- **GameSession(int gameID, int gameType, int diff_level):** this is the default constructor of the game session

Methods:

- **public void initializeChallenge():** this initialize a challenge for the game session.
- **public Inventory getGameSeshInven():** this method initializes the inventory of the game session.
- **public void initializeTime():** this method initializes the time for

4.3.3 Inventory



Attributes:

- **private int inventoryID:** this value holds the inventory id.
- **private int numOfPieces:** this value holds the number of pieces in the inventory.
- **private ArrayList<Piece> pieces:** this is holds list of all the pieces in the inventory.

Constructors:

- **inventory(int inventoryId):** this is the default constructor of the inventory class.

Methods:

- **public int getNumberOfPieces():** this method returns the number of pieces in the inventory.
- **public void addPiece(Piece newPiece):** this method allows for addition of piece into the inventory
- **public Piece getPiece(int pieceID):** this method returns the the piece seleced.
- **public void removePiece(int pieceID):** this method allows for the removal of pieces from inventory.
- **public ArrayList<Piece> getAllPieces():** this method returns

4.3.4 Board

Attributes:

- **private int boardWidth:** this holds the value for the width of the board.

- **private int boardHeight:** this holds the value for the height of the board.
- **private int totalMoves:** this is used to store the total moves in the game.
- **private int currentHeight:** this holds the value for the current height.
- **private int currentWidth:** this holds the value for the current width.
- **private int x:** this holds the value for the x coordinate.
- **private int y:** this holds the value for the y coordinate.
- **private int firstTileClickedX:** this holds the x coordinate for the first tile clicked.
- **private int firstTileClickedY:** this holds the y coordinate for the first tile clicked.
- **private int oldTileX:** this holds the x coordinate for the old tile.
- **private int oldTileY:** this holds the y coordinate for the old tile.
- **private final int TileType:** this identifies the type of tile.
- **private boolean insideGrid:** this returns whether the piece is inside the grid.
- **private boolean eligibleForPlacement:** this returns whether a piece is eligible for placement on the board.
- **private Tile[][] tileArray:** this holds the collection of tiles.

- **private ArrayList<XYCoord> eligibleTilesForPlacement:** this holds the coordinate of places the user can place a piece.
- **private ArrayList<Pieces> piecesArray:** this holds the collection of pieces.
- **private MouseListenerForBoard mos:** this is the mouse listener for the board.
- **private GameSession gameSesh:** this initializes the game session.
- **private Piece selectedPiece:** this holds the selected piece.

Constructors:

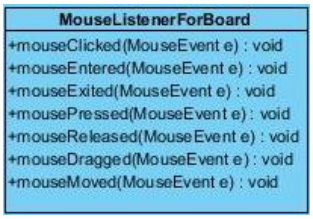
- **Board(int width, int height, int TileType):** this is the default constructor for the board class.

Methods:

- **public void generateTiles():** this method generates tiles.
- **public void placePieceOntoBoard():** this method allows for the placement of piece on the board.
- **public printGrid():** this method prints the grid.
- **public int getTotalMoves():** this method returns the total moves made.
- **public boolean checkForEndGame():** this method checks whether the game has ended.

- **public void resetForPiece(int piece):** this method allows for the piece to be reset.
- **public void generateTileUnderPiece(MouseEvent e):** this method generates the tile under a piece.
- **public void paintComponent(Graphics g):** this method paints the board.

4.3.5 MouseListenerForBoard



Attributes:

- **private void mouseClicked(MouseEvent e):**
- **private void mouseEntered(MouseEvent e):**
- **private void mouseExited(MouseEvent e):**
- **private void mousePressed(MouseEvent e):**
- **private void mouseReleased(MouseEvent e):**
- **private void mouseDragged(MouseEvent e):**
- **private void mouseMoved(MouseEvent e):**

With these functions we capture mouse movement and actions in the board panel which consists of selecting and hovering over a tile, clicking, selecting and dragging a piece, and placing it if possible.

4.3.6 Piece

Attributes:

- **private int width:** this holds the value of the width of a piece.
- **private int height:** this holds the value of the width of a piece.
- **private int x:** this holds the value of the x coordinate of a piece.
- **private int y:** this holds the value of the y coordinate of a piece.
- **private int pieceID:** this holds the piece Id.
- **private int size:** this holds the size of the piece.
- **private int firstX:** this holds the x coordinate of a piece when we are dragging along the board.
- **private int firstY:** this holds the y coordinate of a piece when we are dragging along the board.
- **private int constructorX:** this holds the x coordinate of the piece when first initialized.
- **private int constructorY:** this holds the y coordinate of the piece when first initialized.
- **private Color color:** this holds the default color of the piece.
- **private Color selectedColor:** this holds the color selected by the user.
- **private Color currentColor:** this holds the current color of the piece.
- **private boolean pieceHasFocus:** this identifies whether a piece has focus.
- **private boolean placed:** this determines whether a piece is placed or not.

- **private ArrayList<XYCoord> coordinates:** this holds the x and y coordinate of all piece.
- **private ArrayList<Ellipse2D> circles:** this is a collection of circles for forming a piece.
- **private XYCoord clickedCircleOnShape:** this represents the coordinates we click if we select a piece.

Constructors:

- **Piece(int pieceID, Color color, ArrayList<XYCoord> coord, int xx, int y):** this is the default constructor of the piece class.

Methods:

- **public int getPieceID():** this method returns the piece Id.
- **public Color getColor():** this method returns the color of the piece.
- **public int getSizeOfPiece():** this method returns the size of the piece.
- **public void rotateRespecttoX():** this method allows for rotation with respect to the x axis.
- **public void rotateRespecttoY():** this method allows for rotation with respect to the y axis.
- **public void rotateCounterClock():** this method allows for rotation in the counter clockwise direction.

- **public void rotateClock():** this method allows for rotation in the clockwise direction.
- **public Dimension getDimensions():** this method returns the dimensions of a piece.
- **public void changePieceFocus():** this method changes the focus of a piece.
- **public void returnToOrigin():** this method returns the piece to the origin.
- **public ArrayList<XYCoord> getCoordinates():** this method returns the coordinates of pieces.
- **public int differenceX(int index):** this method returns the difference in placed x position of a piece with actual x position.
- **public int differenceY(int index):** this method returns the difference in placed x position of a piece with actual y position.
- **public void setPosition(Point p):** this method sets the position of a piece.
- **public void setPiecesFocus(boolean foc):** this method sets the focus of a piece.
- **public void paintComponent(Graphics g):** this method paints the piece.
- **public boolean insideShape(MouseEvent e):** this method returns whether mouse position is inside the piece shape.

- **public setFirstPoint(int xx, int yy):** this method sets the first position of piece placement.
- **public setSecondPoint(int xx, int yy):** this method returns the second position of piece placement.

4.3.7 Tile

Attributes:

- **private final int TitleWidth:** the width of one tile
- **private final int TitleHeight:** the height of one tile
- **private int x:** the x position on screen
- **private int y:** the y position on screen
- **private int tileId:** the id of the tile
- **private int pieceId**: the piece's id that rests on the tile(if any)
- **private int TileType:** this indicates what tiles are used for the board
- **private Color normal:** color when a tile is in its normal state
- **private Color hovered:** color when a tile is in its hovered state
- **private Color potential**: color when a tile is a potential candidate to place a piece
- **private Color taken:** color of the tile when a piece is placed on it
- **private boolean whatColor:** Boolean to chose what color to paint the tile with
- **private boolean occupied:** Boolean that indicates if the tile is occupied or not by a piece
- **private boolean visible:** Boolean to show if a tile is visible or not
- **private Rectangle tempR:** used to generate the tile so it can be drawn
- **private Polygon tempP:** used to get the coordinates to draw the tile

Constructors:

- **Tile (int xWidth, int yHeight, int tileId, int tileType, boolean visible):** this generates a tile in the given x and y coordinates, with a specific id and type and a visibility parameter

Methods:

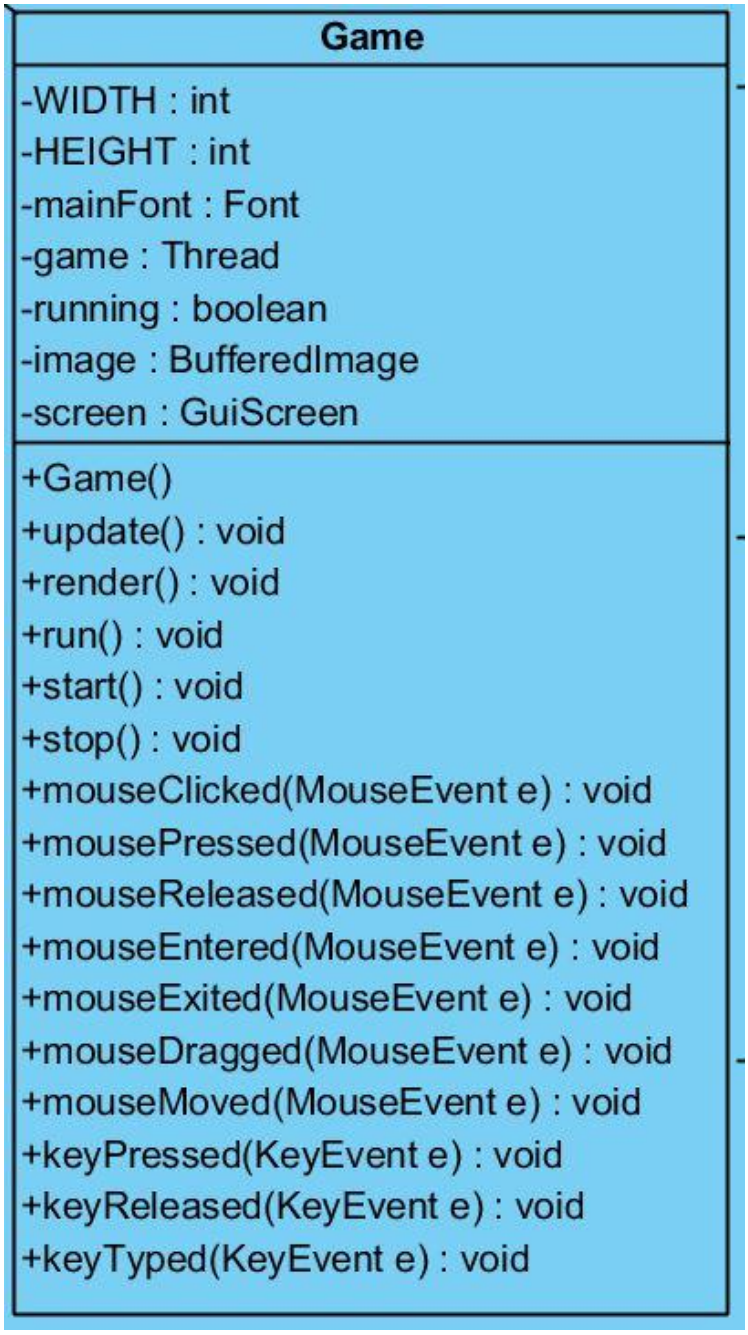
- **public TileStates getState():** returns the state of the tile
- **public void paintComponent(Graphics g):** paints the tile
- **public void drawBlackBorder(int x, int y, Graphics g):** paints border around the tile
- **public void setState(TileStates t):** sets tile state
- **public void setTakenColor(Color t):** sets the color after piece placement
- **public void mouseEntered(MouseEvent e): (1)**
- **public void mouseExited(MouseEvent e): (2)**
- **public void mousePressed(MouseEvent e): (3)**
- **public void mouseReleased(MouseEvent e): (4)**
    - **From 1 to 4:** these functions monitor mouse movement and actions inside a tile, and do the appropriate actions to select, hover, etc**.**

4.3.8 TileState

Attributes:

- **PotentialSelection:** this will be used when a piece is over a tile, and the tiles will change color to indicate that a certain piece may be played on top of them
- **Hovered:** this will be used when mouse is hovered over a tile, without having any pieces selected or dragged
- **Normal:** the state in which the tile is supposed to be at all times, unless the user acts on it
- **Taken:** this will be used as a state when a piece is placed on top of a tile, thus the tile will be marked as taken
- **Unavailable:** this is used when the users drags a piece on the board, but the place the user want to put the piece is already occupied by another one, the tile will be marked as unavailable

4.3.9 Game



Attributes:

- **private int WIDTH:** this is the width of the game panel

- **private int HEIGHT:** this is the height of the game panel
- **private Font main:** this is the main font for the game
- **private Thread game:** a thread that will keep on running to assure all subsystems work and communicate
- **private boolean running:** this indicates that a game is running
- **private BufferedImage image:** the background image for the panel
- **private GuiScreen screen:** the screen gui will be shown

Constructors:

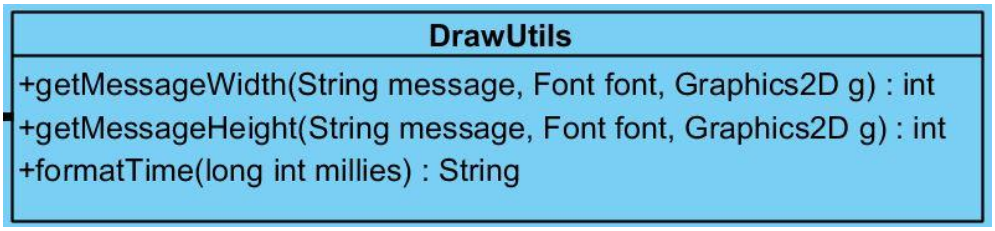
- **Game():** this is the constructor for the game, where the attributes will be initialized

Methods:

- **public void update():** this will update the attributes and the panel
- **public void render():** this method will paint all components on screen
- **public void run(): this will make the game run**
- **public void start():** this method is used to start a game
- **public void stop():** this method is used to stop a game
- **public void mouseEntered(MouseEvent e): (1)**
- **public void mouseExited(MouseEvent e): (2)**
- **public void mousePressed(MouseEvent e): (3)**

- **public void mouseReleased(MouseEvent e): (4)**
- **public void mouseClicked(MouseEvent e): (5)**
- **public void mouseDragged(MouseEvent e): (6)**
- **public void mouseMoved(MouseEvent e): (7)**
    - **From 1 to 7:** all these functions are used to monitor mouse movement and actions within the panel
- **public void keyPressed(KeyEvent e): (8)**
- **public void keyReleased(KeyEvent e): (9)**
- **public void keyTyped(KeyEvent e): (10)**
    - **From 8 to 10:** these functions are used to monitor key presses

4.3.10 DrawUtils

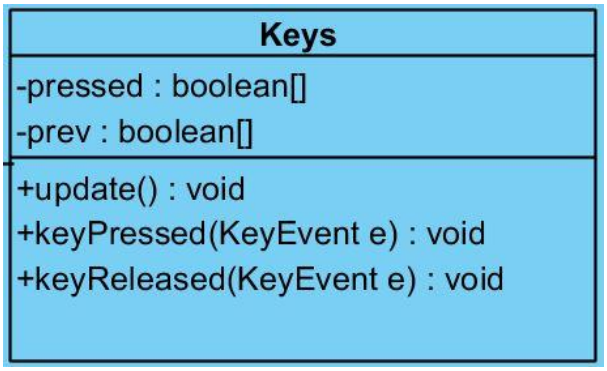


Methods:

- **public int getMessageWidth(String message, Font font, Grapics2D g):** this method will receive the width of the message that should be displayed

- **public int getMessageHeight(String message, Font font, Grapics2D g):** this method will receive the height of the message that should be displayed
- **public String formatTime(long millies):** this will take a long integer as parameter and will calculate the time passed and return it as a formatted string

4.3.11 Keys



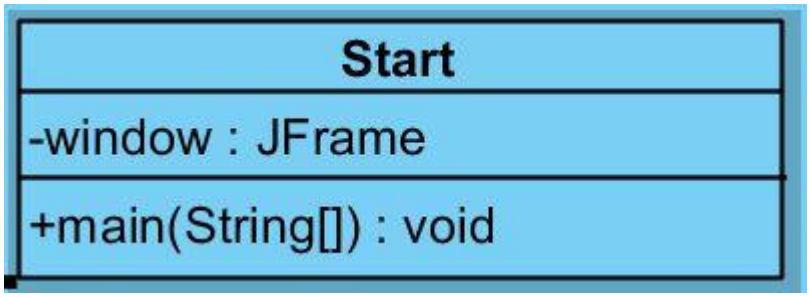
Attributes:

- **private boolean[] pressed :** this will keep an array of booleans for the pressed keys
- **private boolean[] prev:** this will keep an array of booleans for the previous keys

Methods:

- **public void update():** this will update the screen while keys are pressed
- **public void KeyPressed(KeyEvent e):** this will do the operations on what will happen when a key is pressed
- **public void keyReleased(KeyEvent e):** this will do the operations on what will happen when a key is released

4.3.12 Start



Attributes:

- **private JFrame window:** this window will be used to deploy the game on

Methods:

- **public void main(String[] args):** this is the main driver that will start executing when we start the game