Bilkent University
Department of Computer Engineering

CS319 Design Project

IQPuzzler Pro:

Design Report

Burak Alaydin, Kaan Altinay, Muhammad Umair Ahmed, Saifullah Vandana and Skerd Xhafa

Supervisor: Prof. Eray Tuzun

Progress Report

November 8, 2018

This report is submitted to the Department of Computer Engineering of Bilkent University in partial fullfilment of the requirements of the Object-Oriented Software Engineering project CS319.

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

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. It will a simple and comprehensible user interface that the user can understand without much difficulty. This can be achieved using the Object-Oriented Design concepts which will make it easy for an extendible design to be made.

Reliability: The system should run consistently without intermittent stops which could also result in data loss. The game progress of a player will be stored in an online database which can be accessed from home screen using the

load game. It safe to say that the records, rewards and game progress of a player will not be lost due to a system failure or power outage.

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 important 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.

Minimum number of errors: With reliability as one of our design goals, our system must be with minimum errors or at best error-free in order to make it reliable. Errors may result in loss of game data and unreliable gameplay; therefore, it is vital to eliminate errors in our system. Most games are often unplayable due to errors and often users would prefer other alternatives

available. To prevent this, project must be free of errors and must be tested thoroughly during developmental stage.

Ease of Learning and Use: 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.

Trade-Offs

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 decided to split our software into 3 main modules. This division is discussed in sections 4.2 and 4.3

2.2. Hardware/Software Mapping

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).

2.3. Persistent Data Management

For managing persistent data, text files will be used due to their simplicity. Text files also avoid some of the problems encountered with other file formats. When data mismanagement occurs in a text file it is often easier to recover and continue processing the remaining contents during the game. Another advantage of text file usage is its cheap and permanent storage. 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 can 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 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

At initialization, program is loaded into the memory and code is started to be executed. 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. Software Structure

3.1 Overall Structure

The approach we adopted in our object design process was not a standard

design pattern but a mixture of the Model-View-Controller approach with a strong focus on modularity, which meant a strong focus on object independence.

3.2 Division of Modules

Our system is divided into 3 main parts: the Data Handling module, the Gameplay module(including Scoring) and the User Interface module. From the MVC perspective, the Gameplay module consists mainly of models with a single Controller class and the User Interface module is the View.

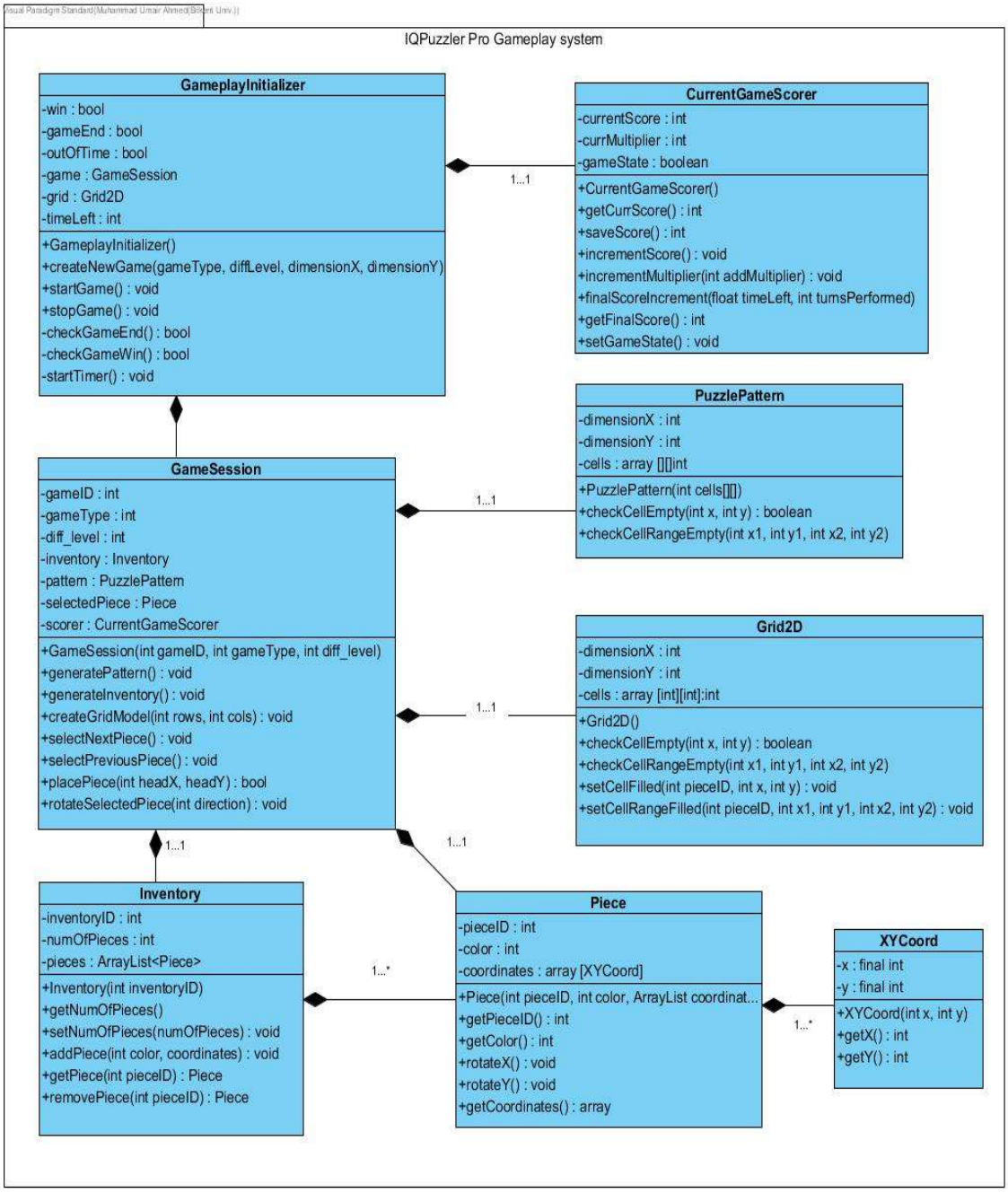
4. Low Level Design

4.1. Design Trade-offs

Efficiency versus Ease-of-Understanding: We decided collectively to keep our objects as close to the real-life model of the game as we could and went for a verbose, modular class structures which resulted in a higher number of more independent object classes, that are easier to understand, as a consequence, than if we had gone for the more efficient route. This additional overhead is affordable, we believe, because our game will be a fairly simple one in terms of complexity and will from no perspective be resource-intensive on today's computer systems.

4.2. Object Design

4.2.1. Objects in GamePlay Module



GameplayInitializer:

This class is responsible for creating all the elements needed for a complete game (i.e. challenge solving session). It creates a new GameSession object with the parameters of the player's choice (**diffLevel = Difficulty Level, dimensionX = number of rows in the Grid, dimensionY = number of columns in the Grid**), initiates the timer when the player indicates that they are ready. It has attributes that indicate whether or not a game is in progress once the game has ended, whether or not the challenge resulted in a win. Once the game ends, the stopGame() method executes a predefined set of operations in order to finish up and dispose of the GameSession.

Attributes:

private bool win

private bool gameEnd

private bool outOfTime

private GameSession game

private Grid2D grid

private int timeLeft

Operations:

public createNewGame(gameType, diffLevel, dimensionX, dimensionY): This method is used to create a new GameSession object with the given parameters.

public void startGame(): This method is used to indicate that the player is ready and to start the game, including the countdownTimer

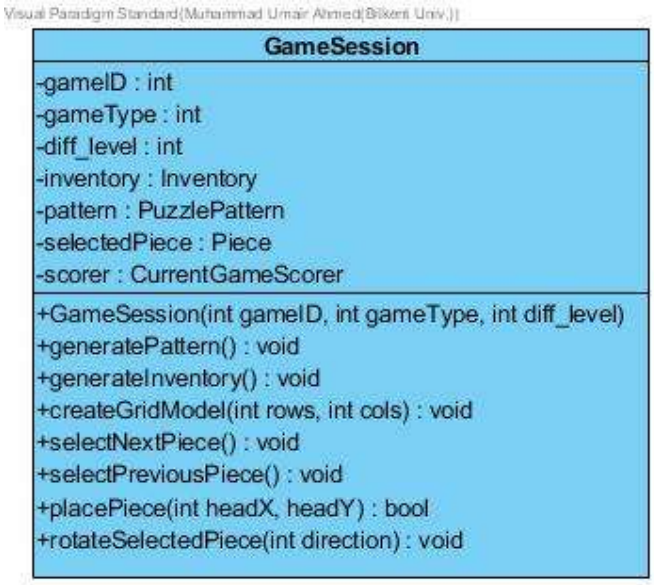
public void stopGame(): This method is called after the game ends to perform some post-game operations such as saving the score to persistent storage.

private void checkGameEnd(): This method simply checks whether or not the game is still in progress.

private void checkGameWin(): This method checks whether, at the end of a game, it was a win or a loss.

private void startTimer(): This method is called in the startGame() method to start the countdownTimer.

GameSession:



GameSession holds all the MVC model category objects for a game to be played. This includes a randomly generated PuzzlePattern which the player has to recreate, a matching Inventory of Pieces that the player can try placing on the Grid2D in order to achieve their objective. The organization is such that from the Inventory, the currently selected Piece by the player in the user interface is actually stored in this GameSession object.

Attributes:

int gameID

int gameType

int diff_level

Inventory inventory

PuzzlePattern pattern

Piece selectedPiece

CurrGameScorer scorer

Operations:

public generatePattern() : This method randomly generates a PuzzlePattern for the player as a challenge

public generateInventory(): This method generates an inventory of Pieces from which he player will have to recreate the PuzzlePattern.

public createGridModel(int rows, int cols): This method creates a Grid2D object on which the player will place their puzzle pieces

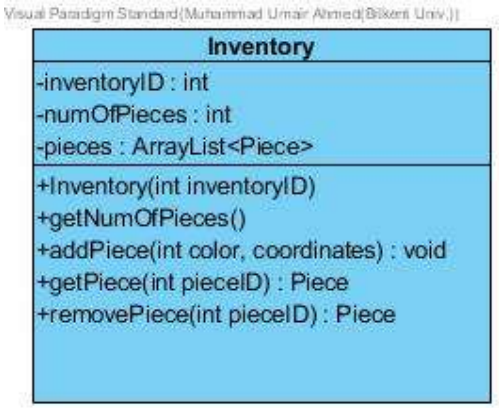
public selectNextPiece(): This changes the selectedPiece attribute to the next Piece in the inventory.

public selectPreviousPiece(): This changes the selectedPiece attribute to the previous Piece in the inventory.

public placePiece(int headX, headY): This method matches the selectedPiece with where the player tried to place on the Grid via the user interface.

public rotateSelectedPiece(int direction): This method transforms the selectedPiece's coordinates in order to reflect the rotation(direction = 0 for x, direction = 1 for y).

Inventory



The Inventory object holds a collection of Pieces for the current GameSession in the form of an ArrayList ,allowing easy removal and element-by-element traversal for when the pieces are being cycled through by the player via the user interface.

Attributes:

private int inventoryID

private int numOfPieces

ArrayList<Piece> pieces

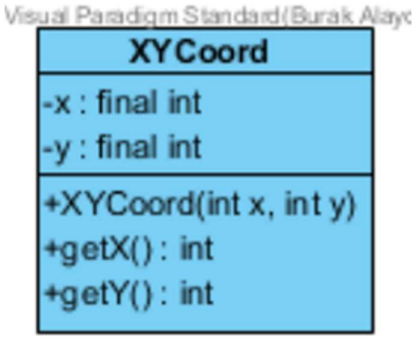
Operations:

public getNumOfPieces(): This method returns the number of pieces left in the Inventory.

public addPiece(int color, int coordinates): This method adds a Piece to the Inventory and increments the numOfPieces attribute.

public getPiece(int pieceID): This method returns a Piece from the Inventory without removing it from there.

XYCoord



Attributes:

private final int x: Row coordinate of a piece.

private final int y: Column coordinate of a piece.

Operations:

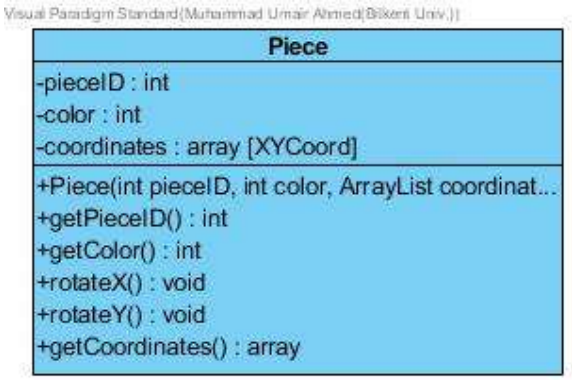
public void XYCoord(int x, int y): Initialize the coordinates to keep track of pieces easily.

public int getX(): Gets the row coordinate.

public int getY(): Gets the column coordinate.

public removePiece(int pieceID): This method returns a Piece from the Inventory and also removes it from the Inventory.

Piece



Attributes:

private int pieceID: Keeps an ID for a particular piece to identify it and make its use simple.

private int color: Stores the color of the piece as int. Each color has its unique number to be distinguished.

private XYCoord[] coordinates: Array to store the coordinates of the piece.

Operations:

public void Piece(int pieceID, int color, ArrayList<XYCoord>): Initialize the attributes of the piece.

public int getPieceID(): Gets the piece's ID.

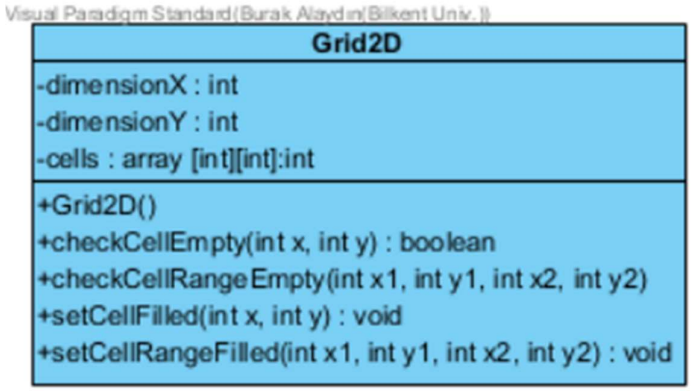
public int getColor(): Gets the piece's color.

public void rotateX(): Rotates the piece's X coordinate.

public void rotateY(): Rotates the piece's Y coordinate.

public XYCoord[] getCoordimates(): Returns the piece's coordinates.

Grid2D:



Attributes:

private int dimensionX: Number of rows.

private int dimensionY: Number of columns.

private int[][] cells: This is the 2D array which the game will be played on.

Operations:

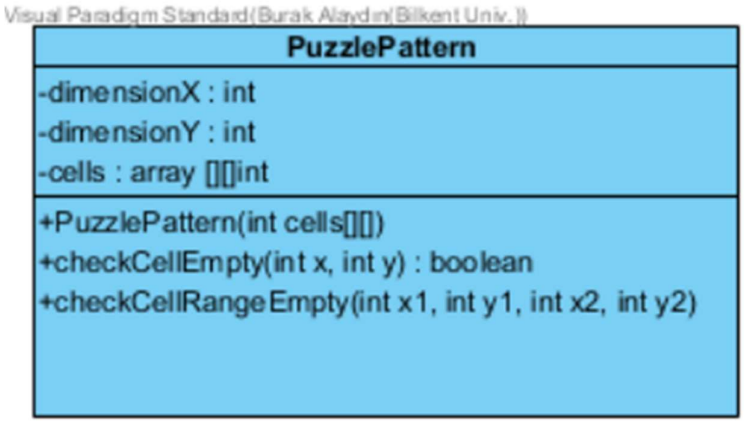
Public void Grid2D(): It is the constructor of Grid2D class. It will create the two dimensional array.

Public boolean checkCellEmpty(int x, int y): Checks whether the cells those user trying to fill is empty or not.

public void setCellFilled(int x, int y): Fills a particular cell if it is empty.

public void setCellRangeFilled(int x1, int y1, int x2, int y2): Depending on the piece that user use and where it is used, fills the given set of cells.

PuzzlePattern



Attributes:

private int dimensionX: Number of rows.

private int dimensionY: Number of columns.

private int[][] cells: The two dimensional array that keeps the pattern.

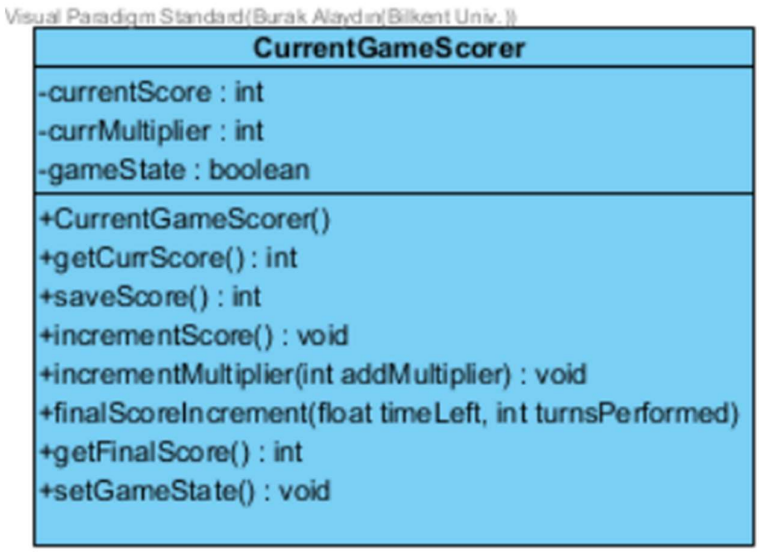
Operations:

public void PuzzlePattern(int cells[][]): Creates a pattern in "cells" array.

public boolean checkCellEmpty(int x, int y): Checks whether a particular cell is empty.

public boolean checkCellRangeEmpty(int x1, int y1, int x2, int y2): Checks whether a set of cells is empty or not.

CurrentGameScorer



Attributes:

private int currentscore: Keeps the current score.

private int currMultiplier: keeps the value of what the current score will be multiplied by.

private boolean gameState: shows the current state of the game (game is active or game is ended).

Operations:

public void CurrentGameScorer(): Initialize the attributes.

public int getCurrScore(): Gets the current score.

public int saveScore(): Saves the current score to database.

public void incrementScore(): After each valid turn, increases the current score by multipliying it with currentMultiplier.

public void incrementMultiplier(int addMultiplier): Calculates correct turn reward and bonuses, then updates currMultiplier.

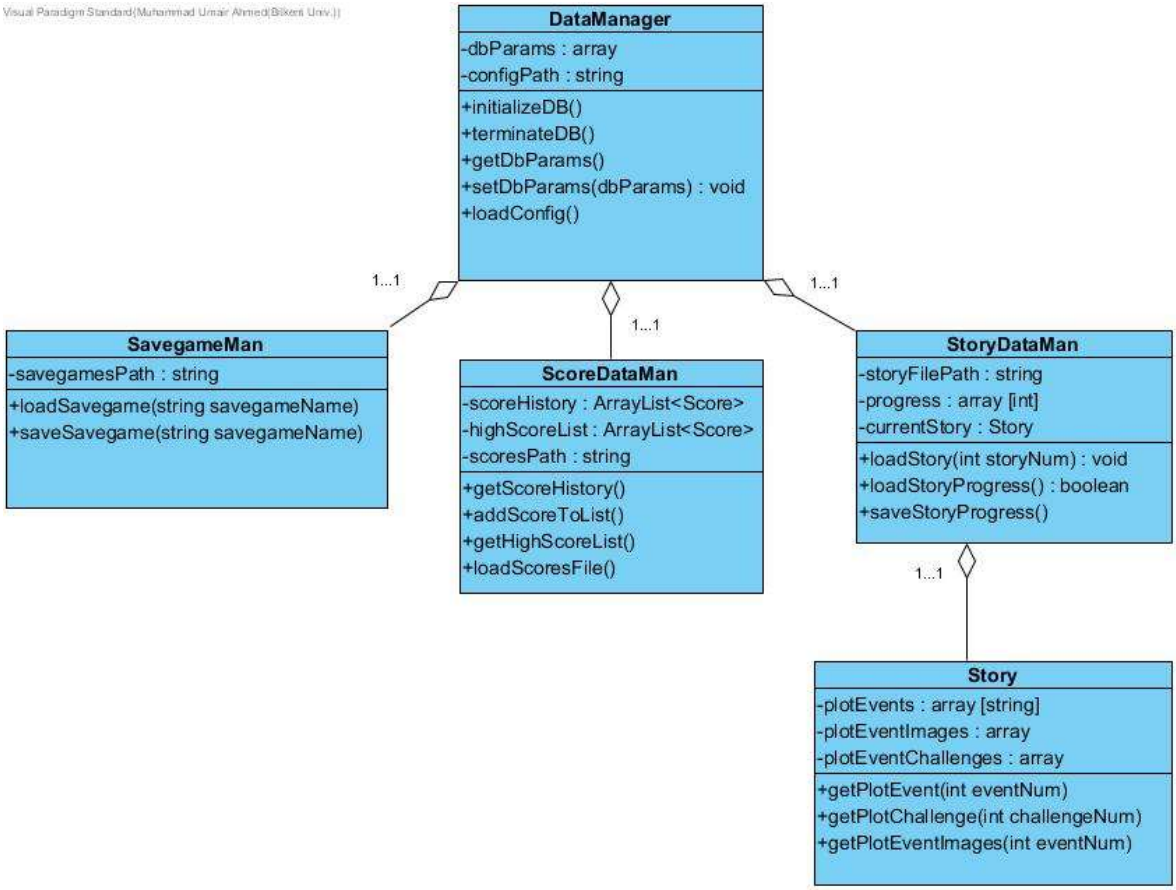
public void finalScoreIncrement(float timeLeft, int turnsPerformed):

According to succes of the user at the end of the game, increases the current score one last time.

public int getFinalScore(): Returns the final score of the user.

public void setGameState(): Changes game state at the end of the game.

4.3.2. Objects in DataManager Module

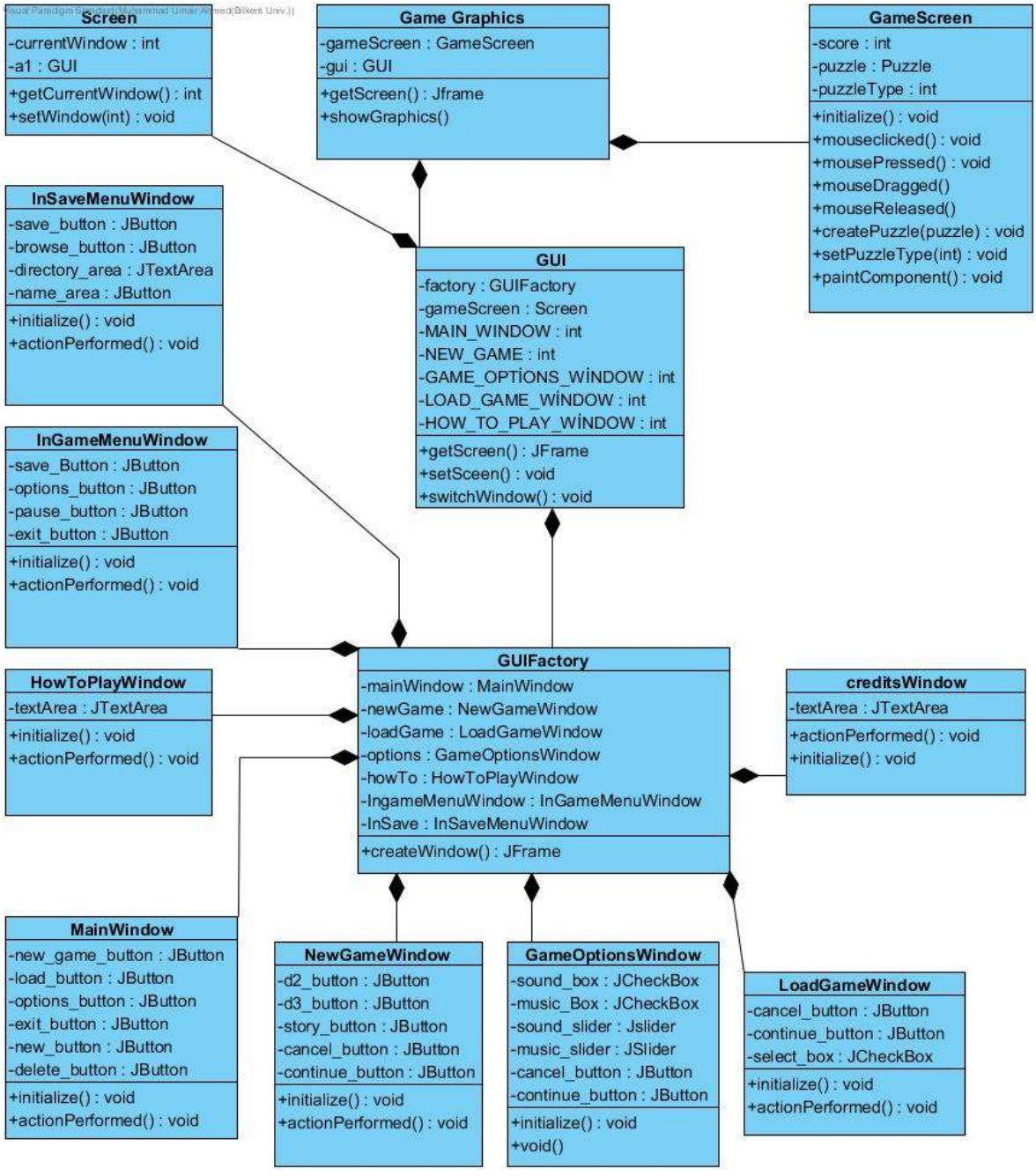


SavegameMan: This class handles saving games to text files using a given file-name and also loading a saved game from a given file-name.

ScoreDataMan: This class handles retrieving score history from a text file and handing it over to the UI to be displayed and also adding the score of every game played to this record file once the game ends.

StoryDataMan: Since the elements involved in the Story Mode involves only predefined elements, both in terms of the Stories available to play, the plot events for each story and the related challenges, this class handles loading these elements for gameplay.

4.3.3. Objects in UI Module



Client.GameGraphics

GameGraphics 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 subcomponents 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

GUI Component provides services for creating and handling the windows that are guides the initialize game procedure. It gets a window id and than 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 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

GUIFactory 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.

GUI. GameScreen

GameScreen 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 as follows:

It makes use of 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)