CS 319 - Object-Oriented Software Engineering

Design Report

Chess Game

Prepared by Group 04
Kaya YILDIRIM - 21002071
Burak MANDIRA - 21301474
Berk ABBASOGLU - 21400812
Rabia Ecem AFACAN - 21101259

Proposed to
Bora GÜNGÖREN

Table of Contents

1. Introduction

1.1 Purpose of the System

Chess is a well-known strategy game and it is known almost by everyone. Our chess game, called Chess, is virtual version of the usual chess game. Chess supports all kind of fancy rules of chess game such as promotion, castling etc. Although Chess looks so plain in terms of user interface which is two dimension, it tries to compensate this drawback by adding some extra visual features like showing the possible moves when a piece is clicked. Chess is designed for both entertainment and intelligence-boosting purposes so that players can boost their brains and have enjoyable time while playing it.

1.2 Design Goals

As every system has design issues, there must be design goals to handle with those issues. For that reason, Chess has some design goals as well. Chess mainly put emphasis on non-functional requirements to improve its quality, user friendliness and sustainability. The design goals of Chess are described in detail below.

End User Criteria:

Ease of Use: Because our product is a game, it should provide user having a good time while s/he is playing. To do this, the game should not raise difficulties to users. For example, the system should ensure user-friendly interface to be helpful to user to easily find desired operations on menus. And also, the game is played with mouse, this makes easy to play the game, players who can see computer screen and use mouse can play the game by just clicking left button of mouse. If they cannot such as disabled people can get help from others.

Ease of Learning: The game is not required any advanced knowledge. If player just know how to play chess, s/he can easily play the game. If s/he have difficulty to understand to open a game, s/he can read our help file in “Help” section. Also if s/he does not know playing chess, s/he can go to online chess wikipedia page via our help page through the link on that page.

Maintenance Criteria:

Extensibility*:* Extensibility is a systems design principle where the implementation takes future growth into consideration. It is a systemic measure of the ability to extend a system and the level of effort required to implement the extension. Extensions can be through the addition of new functionality or through modification of existing functionality. The central theme is to provide for change – typically enhancements – while minimizing impact to existing system functions. In this respect our design will be suitable to add new functionalities easily to the existing system.

Portability*:* Portability in high-level computer programming is the usability of the same software in different environments. The pre-requirement for portability is the generalized abstraction between the application logic and system interfaces. When software with the same functionality is produced for several computing platforms, portability is the key issue for development cost reduction. In this respect we are determined that the system will be implemented in Java, since its JVM provides platform independency, our system will satisfy the portability.

Modifiability*:* In our software system the existing functionalities would be easy to modify. We are planning to minimize the coupling of subsystems to avoid undesired impacts on system components in order to achieve this.

Performance Criteria:

Response Time: Since this is a turn based game, no back to back action with high memory consumption is expected to take place. Even if the players try to make many moves

one after the other though, since we keep the valid moves for each piece and all the pieces on the board in memory, we can quickly check whether they are valid moves and act accordingly. The player experience will be smooth.

Trade Offs:

Ease of Use and Ease of Learning vs. Functionality: Chess is a game which is easy to learn but difficult to master. We will try to make it simple for a beginner to learn and average player to enjoy fully. We will not add any further functionality to the game, like boosts or effects. This is to make the learning and using process of the program easier.

Performance vs. Memory: We will keep the locations of all the remaining pieces on board and the valid moves for each piece in memory. With this, we aim to swiftly and smoothly deliver player's commands on the board. Instead of checking whether a move is valid when the player calls for it, we will know it beforehand, so we can determine the result fast. We will opt for performance in this matter instead of memory.

1.3 Overview

In part 1, we represented purposes of the system, which are basically entertaining the players and providing having a good time. In order to achieve these purposes, we defined our design goals which are providing the portability, ease of use, ease of learning, high performance and high maintainability.

2. Software Architecture

2.1 Overview

In this part, we will decompose our system into maintainable subsystems. Our main aim is not only reducing the coupling between the different subsystems of the system, but

also increasing the cohesion of subsystem components. In addition, we tried to decompose our system in order to apply MVC architectural style on our system.

2.2 Subsystem Decomposition

In this section, the system is divided into relatively independent parts to clarify how it is organized. Decomposition of independent parts is very important in terms of meeting non-functional requirements and creating a high quality software. Therefore, the decisions we made in identifying subsystems will affect fundamental features of our software system such as extendibility, performance and modifiability.

As we can see, in Figure 1 system is separated into three subsystems which are focusing on different parts of the software system. These subsystems are UI, Game Management and Game Entities. They are working on separate parts and they are connected to each other for possibility any change in the future. In Figure 2 two subsystems which are Game Management and Game Entities are loosely coupled. In addition, Connection between User Interface subsystem and Game Management subsystem is provided via Game Manager class. This is because, if any changes or errors are occurred on User Interface subsystem, only Game Manager class will be affected.

In Figure 2, it is seen that classes which are working on common purpose are grouped and they are performing similar tasks. For example, classes whose name ends with “Manager” are associated to perform to control tasks by sending requests and responds to each others.

When we design our subsystems, our aim is to meet system goals. For this reason, we tried to design productive system with high cohesion and loosely coupling. This makes our software flexible to make any changes.

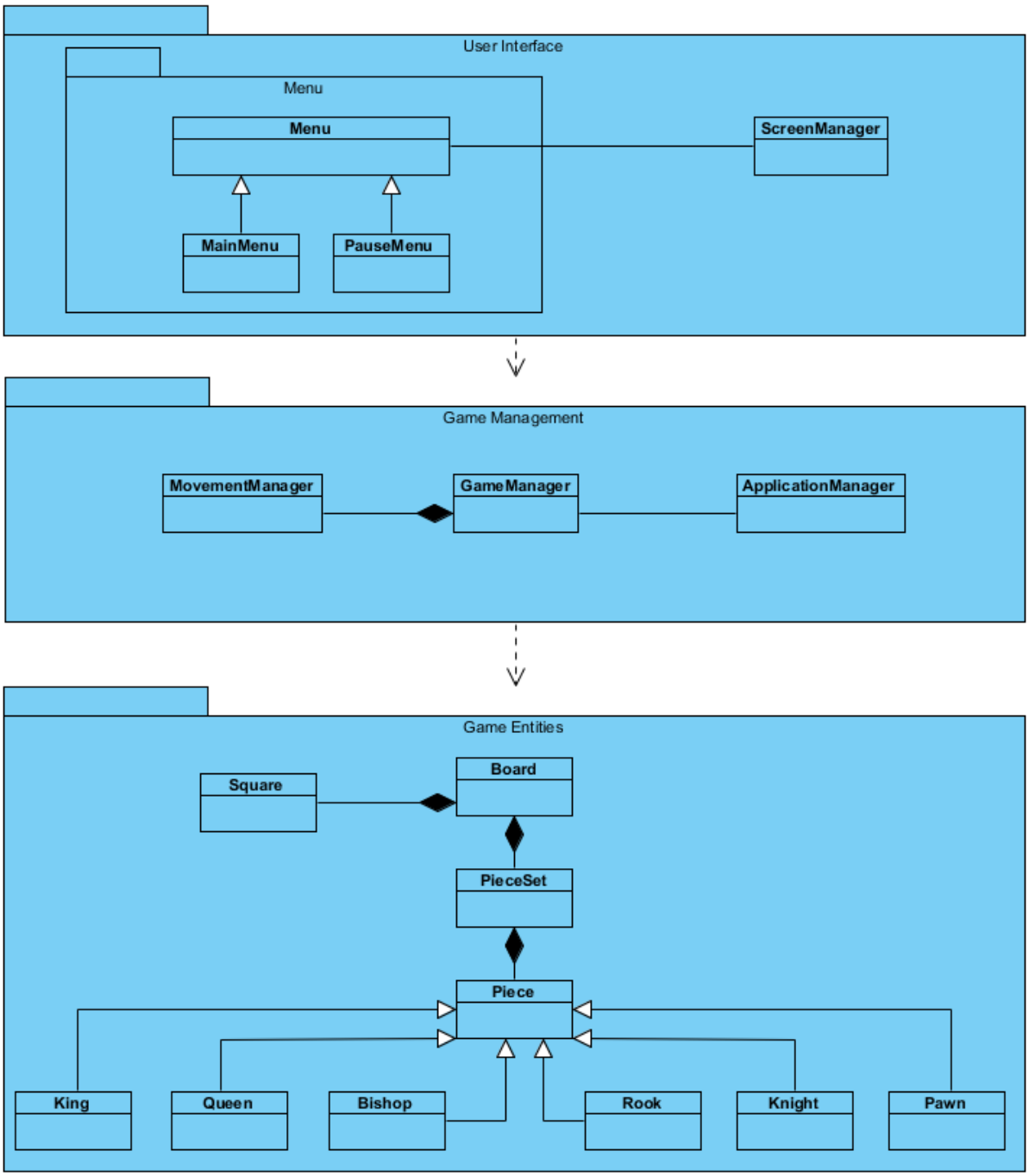


Figure – 1 (Basic Subsystem Decomposition)

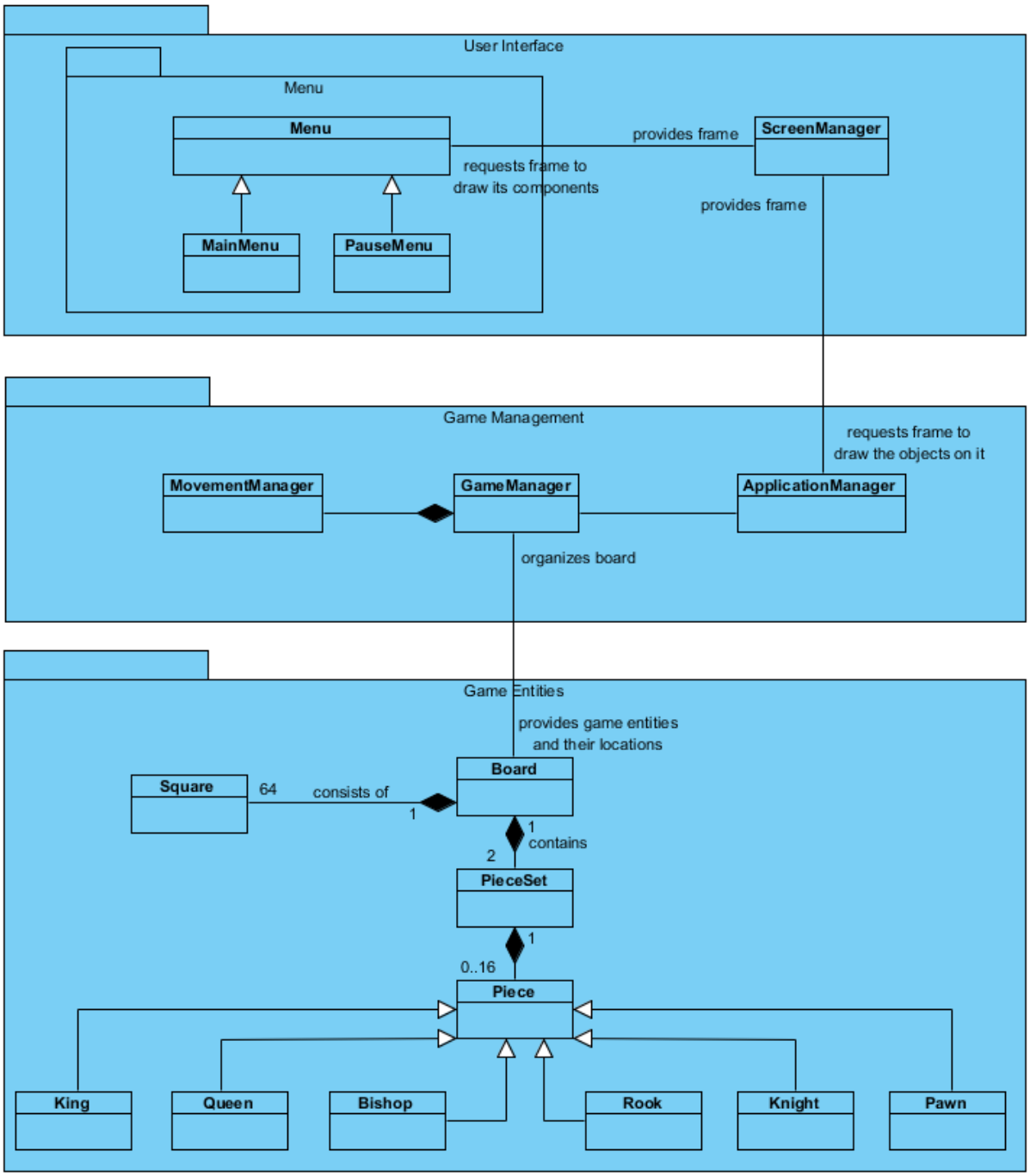


Figure-2 (Detailed Subsystem Decomposition)

2.3 Architectural Layers

2.3.1 Layers

We decomposed the system into three layers which are User Interface, Game Management and Game Entities. These three layers are decomposed as hierarchical. The first layer which is the highest layer also is User Interface layer because it has only interaction with the user, other layers do not use User Interface layer. The second layer is called Game Management Layer which also the layer where game logic is controlled. The last layer is Game Entities Layer. As is evident from its name it includes the game entities. This layer decomposition also proposed the closed architectural style, it means that a layer can only access to the layer that below it. Our layer decomposition is demonstrated in Figure-3.

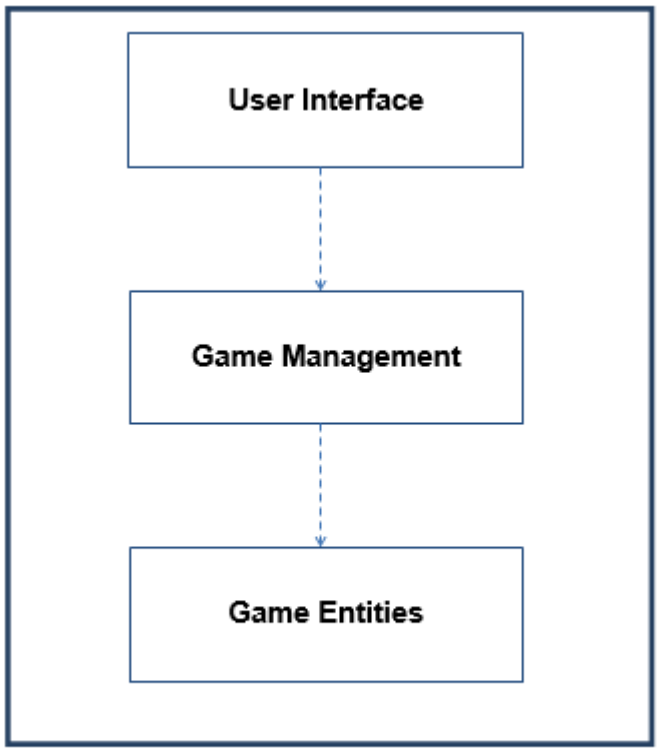


Figure-3

2.3.2 Model View Controller

Main purpose of this architectural style is classifying the subsystems into three parts as is evident from its name model, view and controller. Purpose of the dividing the subsystems into three parts is isolating the domain knowledge from the user interface by adding a controller part between them. In this system, as it can be seen that we grouped our domain objects in the game entity layer which generates the model part of the system. The domain object of this system is only accessible and controllable by manager classes which are grouped under Game Management layer which generates controller part. Because User Interface layer just communicates with the model part and controller part, it generates our view part. The benefit of the MVC is that changes on the interface do not affect the model of the system. Therefore, we think that using MVC is good choice for games.

2.4 Hardware / Software Mapping

Our Chess game will be implemented in Java programming language. To implement the game, we are going to use latest java JDK 1.8.0_112. To play game, only keyboard and mouse are enough. To type username, player needs keyboard and to go around the menu and make a move, player needs a mouse. Because the game will be implemented in Java, system requirements are minimal; actually there is no system requirement. All computers which have operating system which run java files can execute our game. To store the user's name and last winner list we will use .txt files. Since most of the operating systems supports .txt files there will be no problem on that issue. Lastly, the game does not require any network or internet connection.

2.5 Persistent Data Management

The chess game does not require database system. It's all files work on client's computer. The game stores board and past winner list as text files on computer's disk. The

game board as a text file is created before the game is executed therefore this data is persistent. If the text file is corrupted after game execution, this situation is not affecting in-game issues for example piece sets and pieces.

2.6 Access Control and Security

As mentioned in earlier parts of the report, the chess game is not require network or internet connection. If player has game's .jar file, s/he is able to play the game. Therefore, there is no restriction or control for access the game. In addition, in game there is no user player therefore security issues is not necessary for the game. Also since it is free to play game, there is no protection for privacy.

2.7 Boundary Conditions

Initialization

To initialize the game, just opening game's .jar file is enough. Installation is not required to play.

Termination

There several ways to terminate the chess game. One of them is just clicking [X] button at the right upper corner for windows; left upper corner for Mac OS X. Other one is clicking "Quit Game" button at the main menu. While playing game, after pressing "ESC" from keyboard, either click "Exit Game" or click "Return to Main Menu" then click "Quit Game" at main menu.

Error

If game resources are not loaded such as images, the game starts without these files.

If program is not respond, player can lose game progress, unless program respond again.

3. Subsystem Services

3.1 Design Patterns

Façade Design Pattern:

The facade pattern (or façade pattern) is a commonly used design pattern in object-oriented programming. A facade is useful because large body of code can have simplified interface by using this pattern. Since any change in the components of this subsystem can be reflected by making changes in the façade class; advantages of facade design pattern can be listed by readability, reusability, extendibility, easier library use and reducing dependencies.

We will use facade design pattern in our two subsystems which are Game Management and Game Entities. In Game Management subsystem our façade class is ApplicationManager which communicates with the components of the Game Management subsystem according to the requests of User Interface subsystem. For our Game Entities subsystem our façade class is GameManager class which handles the operations on entity objects of our system, according to the needs of Game Management subsystem.

3.2 User Interface Subsystem Interface

The graphical system components of our software system and transition between panels which are constructed for different selections in menu screen are provided by User Interface Subsystem. The reference of User Interface Subsystem to other subsystems is provided by an interface which is menu class.

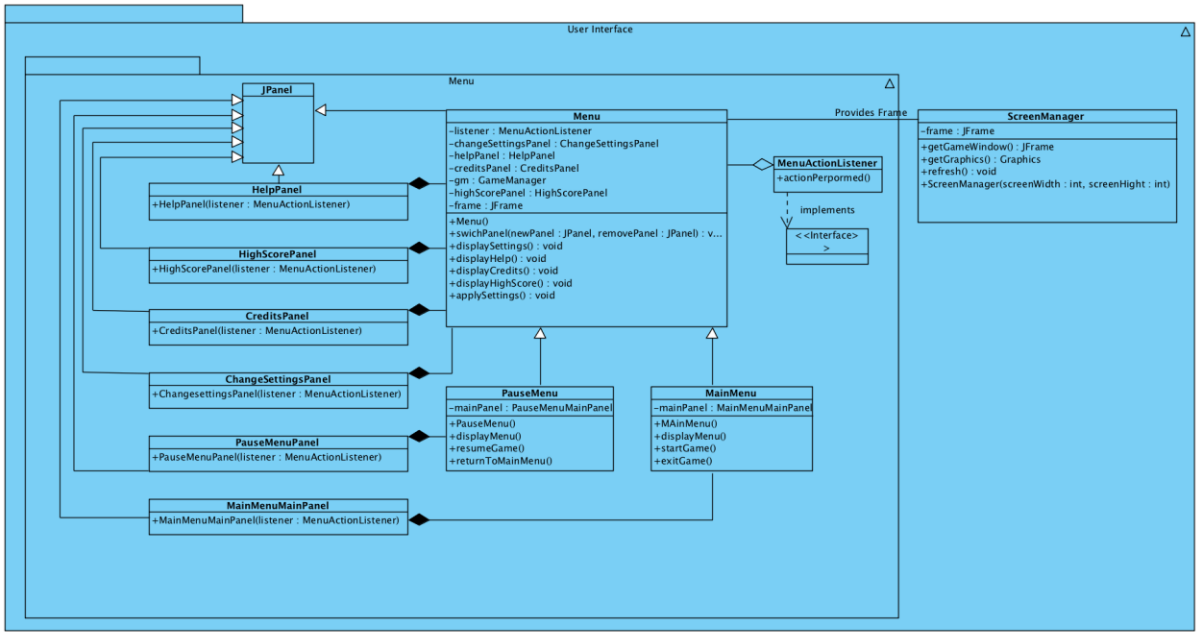
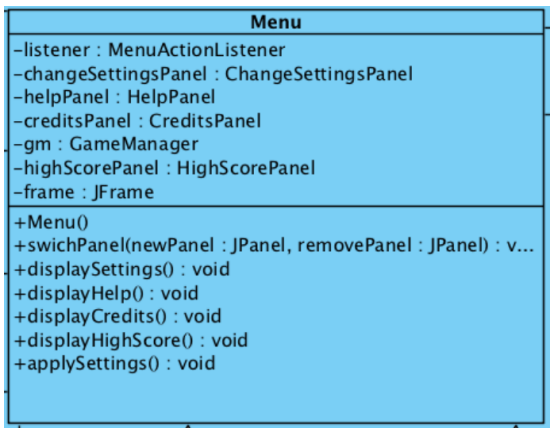


Figure 3.2.1 : User Interface Subsystem

3.2.1 Menu Class



Attributes:

- **private JFrame frame:** This is the main frame in which all visual context will be displayed.

- **private MenuActionListener listener:** This is for getting input from user via graphical user interface.
- **private ChangeSettingsPanel changeSettingsPanel:** This is for showing change settings menu. ChangeSettingsPanel is a JPanel type.
- This property is constructed by ChangeSettingsPanel class with its components like buttons, labels etc…
- **private HelpPanel helpPanel:** This is for showing help menu. HelpPanel is a JPanel type.
- This property is constructed by HelpPanel class with its components like buttons, labels etc…
- **private CreditsPanel creditsPanel:** This is for showing credits menu. CreditsPanel is a JPanel type.
- This property is constructed by CreditsPanel class with its components like buttons, labels etc…
- **private GameManager gm:** When Play Game selection is received from graphical user interface, Menu class provides reference to GameManagement subsystem. This is a GameManagement type.
- **private Setting settings:** This Setting type property of Menu class keeps the system settings adjusted by user from change settings menu. *settings* object has instance variables as *background colour and showing or hiding possible movements.*

Constructors:

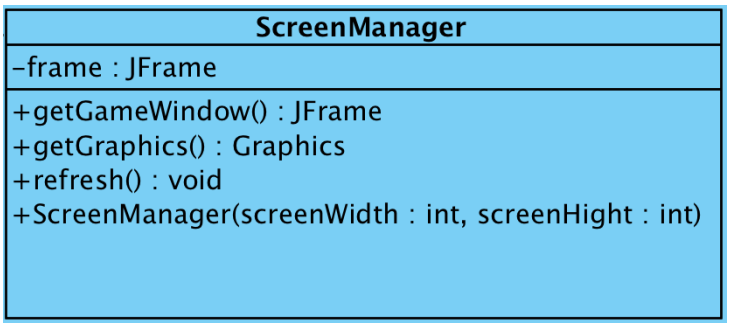
- **public Menu:** Initializes *gm, settings, listener* properties, *changeSettingsPanel, creditsPanel and helppanel.*

Methods:

- **public void switchPanel(newPanel,removedPanel):** It changes the panel on frame according to received selection from game menu.

- **public void displayHelp():** This method includes switchPanel() method and adds *helpPanel* to frame by replacing the current panel on frame.
- **public void displayCredits():** This method includes switchPanel() method and adds *creditsPanel* to frame by replacing the current panel on frame.
- **public void displaySettings():** This method includes switchPanel() method and adds *changeSettingsPanel* to frame by replacing the current panel on frame.
- **public void applySettings():** This method applies the requested settings by setting values of *setting* object which are *background colour and showing or hiding possible movements.*

3.2.2 ScreenManager Class



Attributes:

- **private JFrame frame:** This is main frame of our program. In this frame all visual context will be displayed.

Constructor:

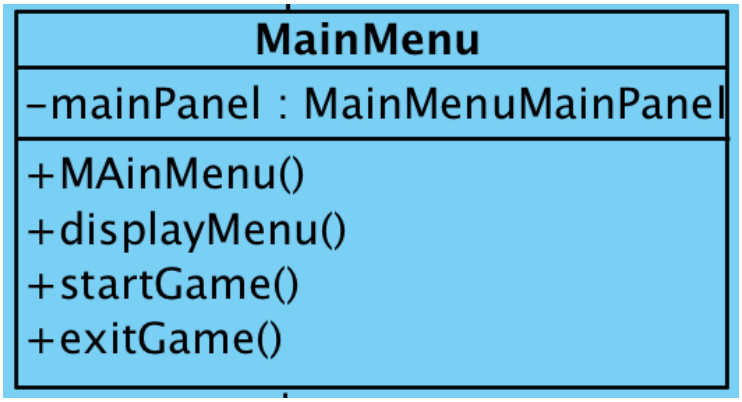
- **public ScreenManager(int screenWidth, int screenHeight):** for constructing an object it takes measurements of screen; hight and width.

Methods:

- **public JFrame getGameWindow():** returns a JFrame object to display game screen.
- **public Graphics getGraphics():** returns a graphics context for drawing to an off-screen image.

- **public void refresh():** It repaints the game components in the frame.

3.2.3 MainMenu Class



Attributes:

- **private MainMenuMainPanel mainPanel:** MainMenu class is used in graphical user interface to show Main Menu on screen. This is a MainMenuMainPanel type.

Constructor:

- **public MainMenu():** It initialises instances of MainMenu object.

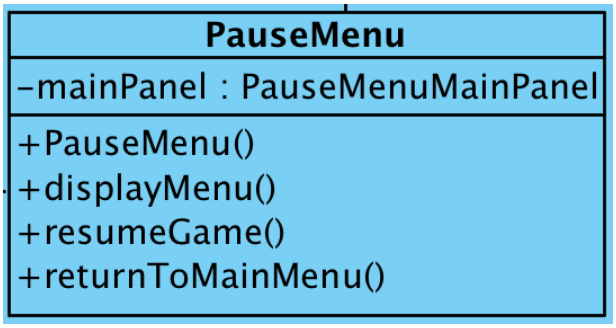
Methods:

- **public void displayMainMenu():** By using switchPanel() method this method adds mainPanel to frame via replacing the current panel on frame.
- **public void startGame():** To control gameplay routine this method invokes gameManagement subsystem by the reference of gm attribute of MainMenu class (gm attribute is inherited from Menu class).
- **public void exitGame():** The application ends by this method.

- The following methods are inherited from parent Menu class.
- switchPanel(newPanel : JPanel, removedPanel : JPanel),
- displaySettings()
- displayCredits()

- displayHighScore()
- displayHelp()

3.2.4 PauseMenu Class



Attributes:

- **private PauseMenuMainPanel mainPanel:** This is used in graphical user interface to show Pause Menu on screen. This is PauseMenuMainPanel type attribute of PauseMenu class.

Constructor:

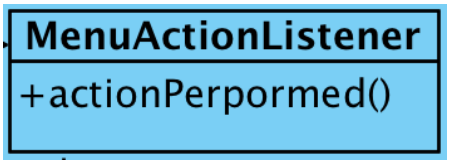
- **public pauseMenu():** Instances of PauseMenu object are initialised by this.

Methods:

- **public void resumeGame():** This method continuous game routine by removing pauseMenu,
- **public void returnToMainMenu():** This method creates a new mainMenu object and displays it on screen.

- The following methods are inherited from parent Menu class.
- switchPanel(newPanel : JPanel, removedPanel : JPanel),
- displaySettings()
- displayCredits()
- displayHighScore()
- displayHelp()

3.2.5 MenuAction Listener



Methods:

- **public void actionPerformed(ActionEvent e):** This method overrides the ActionListener interface's actionPerformed method.
- HighScorePanel, HelpPanel, CreditsPanel, ChangeSettingsPanel, MainMenuMainPanel, PauseMenuMainPanel classes are not described in detail. These classes instantiates requested panels and these panels are placed on frame by MenuActionListener class.

3.3 Game Management Subsystem Interface

This is the subsystem with the purpose of managing an ongoing game. The most important class here is the GameManager class as it holds all the critical information about the game and is responsible for making the moves, switching turns, and deciding if a game winning condition is reached after each move. The other classes are there to help the GameManager class. MovementManager is responsible for making sure the moves are valid and determining the consequences of each move (whether an opponent's piece will be removed etc.). InputManager is responsible for reading the inputs from the user and the Player class holds the information for the players.

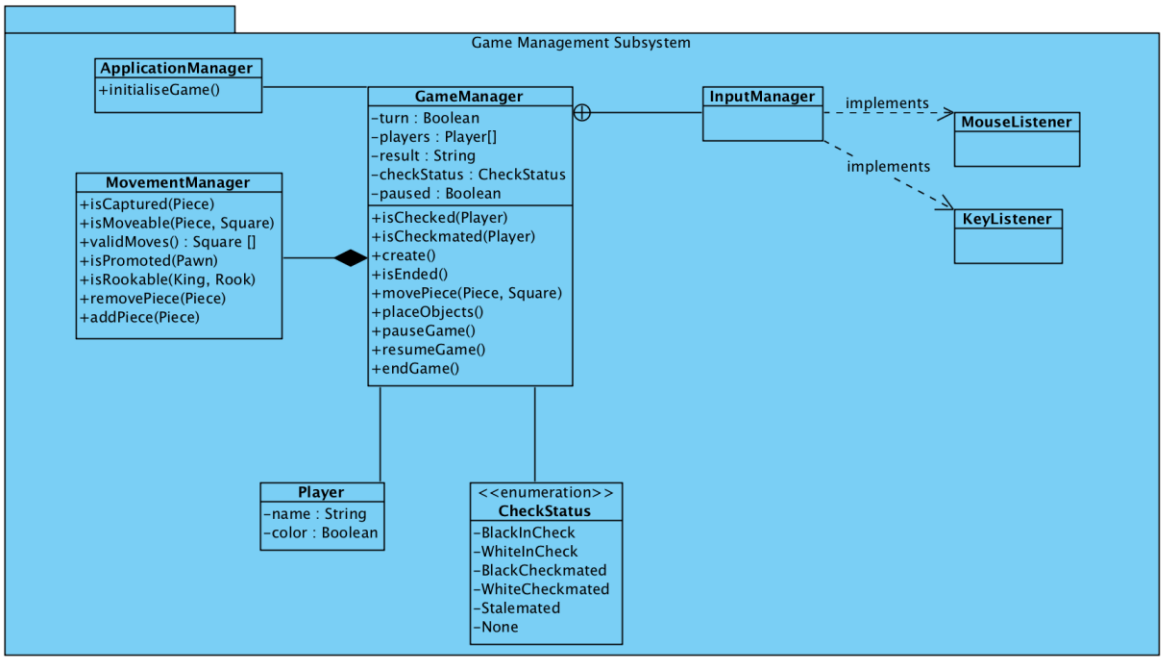


Figure 3.3: Game Management Subsystem

3.3.1 GameManager Class

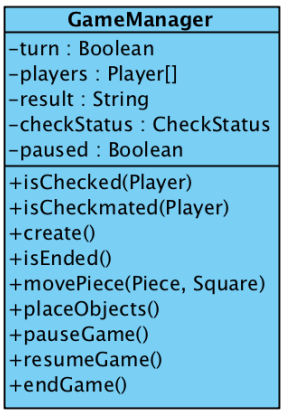


Figure 3.3.1: GameManager Class Diagram

This is the main class to manager our game. Holds the conditions of checked or checkmated for both players, moves the pieces on player's demands (given they are valid moves), is responsible for ending the game upon checkmate or stalemate and recording the result.

Attributes:

private boolean turn: tells whether the turn belongs to White or Black

private Player[] players: holds the players that are playing the current game

private String result: holds the name of the winner, empty until the game ends

private CheckStatus checkStatus: this is an enumeration to tell us some critical conditions of the game, check the description of the enumeration for further information

private boolean paused: tells whether the game is paused or not

Constructors:

public GameManager(): initialises the game, the turn belongs to White as the starter and checkStatus indicates “None”

Methods:

public boolean isChecked(Player): returns if the player is checked or not

public boolean isCheckmated(Player): returns if the player is checkmated or not, it this returns true, this means the game has ended

public boolean isStalemated(Player): returns if the game has ended with a stalemate, if this returns true, this means the game has ended

public boolean isEnded(): returns if the game has ended, or in other words, if a stalemate or checkmate has occurred

public boolean movePiece(Piece, Square): takes the piece and the target square, if the move is possible, does it, returns true if successful

public void placeObjects(): places all the pieces at the start of the game into their starting positions

public void pauseGame(): pauses the game for as long as the player wishes

public void resumeGame(): resumes a paused game**public void endGame():** once the isEnded() method returns true, records the winner and ends the game

3.3.2 MovementManager Class

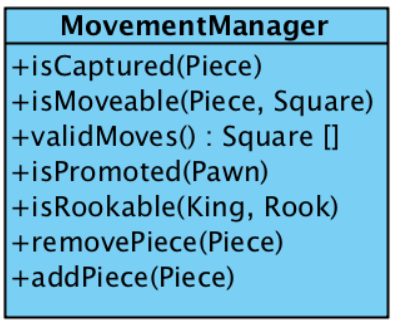


Figure 3.3.2: MovementManager Class Diagram

For perhaps the most critical method of GameManager, movePiece(Piece, Square), we are going to depend on this class. MovementManager is responsible for making sure a move issued by the player is valid. Then, it proceeds to fulfil the consequences of such a move, like removing a piece, recognising a check, or promoting a pawn.

Methods:

public boolean isCaptured(): upon the movement of a piece, if its destination conflicts with that of an opponent's piece, then the opponent's piece is captured. This method returns true if so.

public boolean isMoveable(Piece, Square): checks if a move issued by the player is valid, returns true if so

public Square[] validMoves(Piece): stores the possible moves for a given piece

public boolean isPromoted(Pawn): checks if a pawn is promoted, meaning if it has reached the end of the board. If so, calls the removePiece(Piece) method for the pawn and addPiece(Piece) method for the to-be-added-piece.

public boolean isRookable(King, Rook): checks if the given king and rook and perform the special move called rook

public void removePiece(Piece): if isCaptured() method returns true for a piece, this method removes it from the game

public void addPiece(Piece): when a pawn is promoted, given by the isPromoted(Pawn) method, adds the given piece to its former place.

3.3.3 InputManager Class

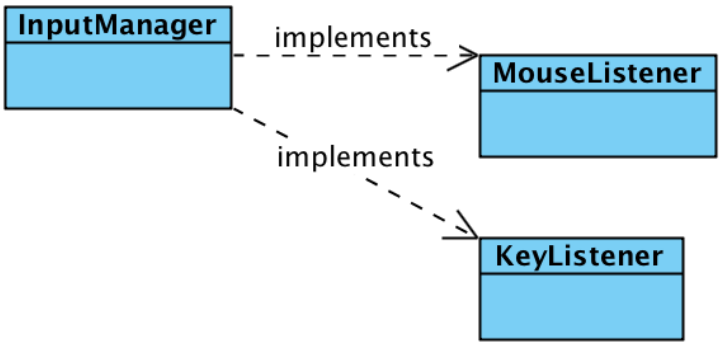


Figure 3.3.3: InputManager Class Diagram

The user will be able to select the pieces they want to move and their destinations using the mouse. Our MouseListener will be responsible for catching those requests. The KeyListener is so that the player can pause the game using the Esc button.

3.3.4 Player Class

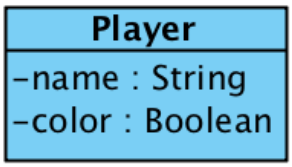


Figure 3.3.4: Player Class Diagram

It holds information of the players. Their names are kept for record once the game ends and their colors are used to define whose turn it is.

3.3.5 CheckStatus Enumeration

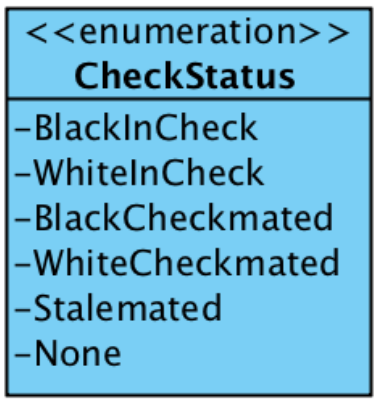


Figure 3.3.5: CheckStatus Enumeration Diagram

This is an enumeration we decided to add so we can have a clear look on the critical situations related to our game. Being checked or checkmated is very important to be aware of, as well as the stalemate case. However, note that this is expected to hold the "none" value most of the time.

3.3.6 ApplicationManager Class

Figure 3.3.6: ApplicationManager Class Diagram

This class is responsible for connecting our GameManager class (and therefore the entirety of the game management subsystem) to the other subsystems. It also initialises the game.

3.4 Game Entities Subsystem

This section clarifies the model part of Chess. It provides details about the objects that exist in the game. These objects are constituted by 10 classes. The relations between these classes are can be seen below in the Figure 3.4. Also, each class of this subsystem is explained in detail after Figure 3.4.

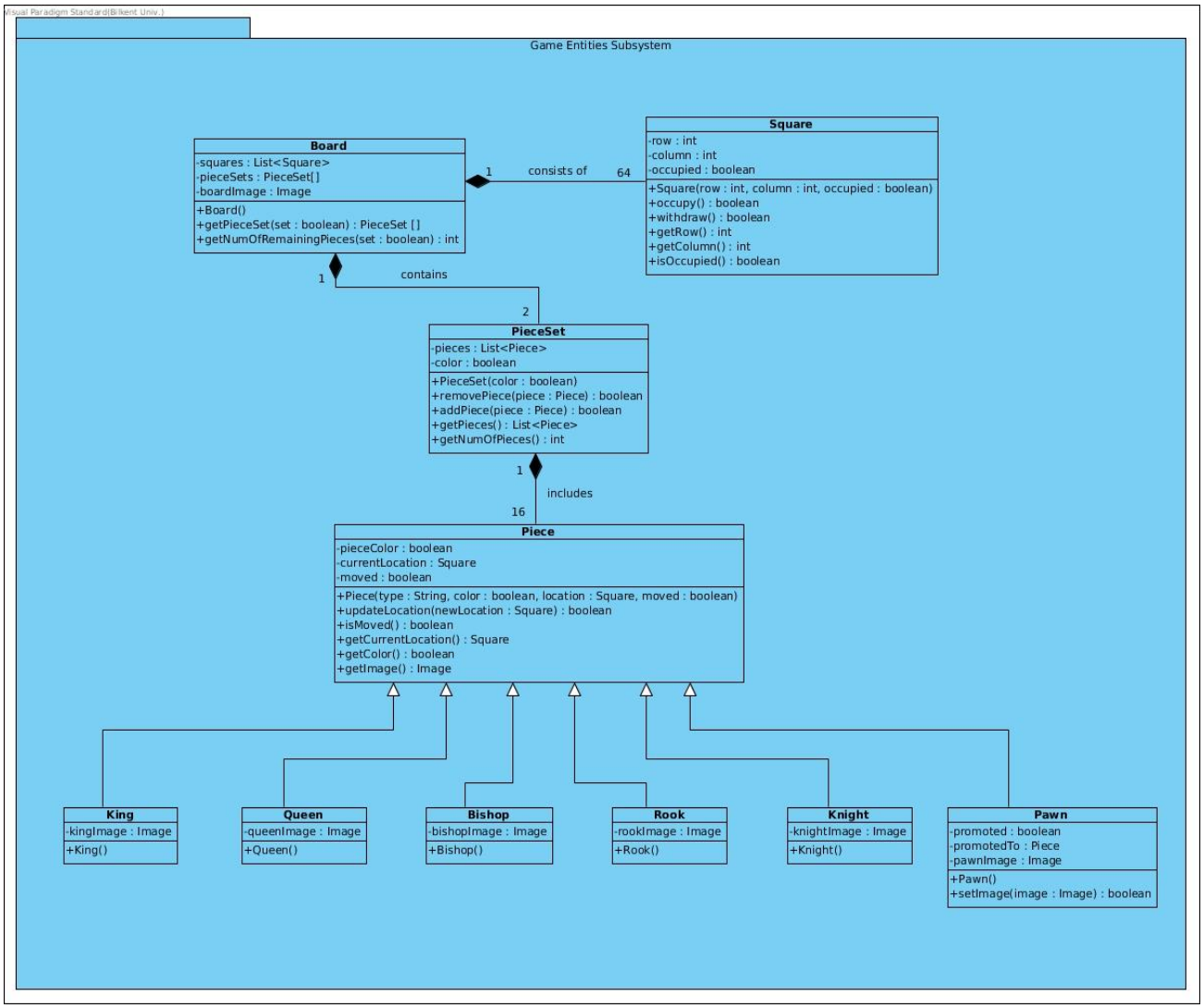


Figure 3.4: Game Entities Subsystem

3.4.1 Board Class

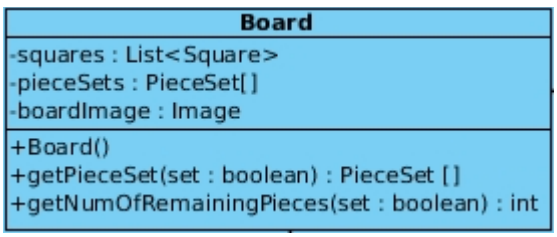


Figure 3.4.1: Board Class Diagram

This class represents the board upon which the game is played.

Attributes:

private List<Square> squares: list of squares that the board contains.

private PieceSet[] pieceSets: array of PieceSet that stores each player's piece set.

private Image boardImage: image object of the board image.

Constructors:

public Board(): creates a Board object.

Methods:

public PieceSet[] getPieceSet(boolean set): returns the specified PieceSet (i.e. it returns one of the player's piece set).

public int getNumOfRemainingPieces(boolean set): returns the number of remaining pieces in the specified PieceSet.

3.4.2 Square Class

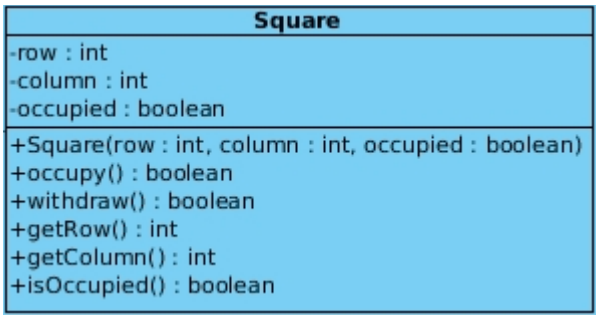


Figure 3.4.1: Square Class Diagram

This class represents each square of which the board consists.

Attributes:

private int row: row number of the square.

private int column: column number of the square.

private boolean occupied: indicates whether the square contains a piece.

Constructors:

public Square(int row, int column, boolean occupied): creates a Square object with the specified attributes.

Methods:

public boolean occupy(): sets the square as occupied.

public boolean withdraw(): sets the square as unoccupied.

public int getRow(): returns the row number of the square.

public int getColumn(): returns the column number of the square.

public boolean isOccupied(): returns whether the square is occupied.

3.4.3 PieceSet Class

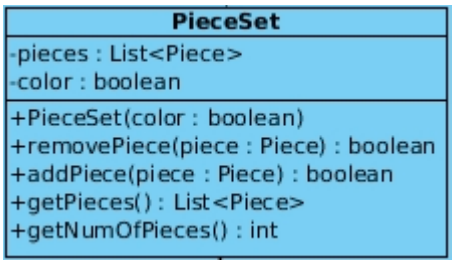


Figure 3.4.1: PieceSet Class Diagram

This class represents each player's piece set (i.e. an instance of this class contains one of the player's pieces).

Attributes:

private List<Piece> pieces: list of pieces that a player has.

private boolean color: color of the pieces (e.g. white or black).

Constructors:

public PieceSet(boolean color): creates a PieceSet object which contains all pieces with the specified color.

Methods:

public boolean removePiece(Piece piece): removes the specified piece from the PieceSet.

public boolean addPiece(Piece piece): adds the specified piece to the PieceSet.

public List<Piece> getPieces(): returns the list of pieces that the PieceSet contains.

public int getNumOfPieces(): returns the number of pieces that the PieceSet contains (i.e. returns the length of List<Piece>).

3.4.4 Piece Class

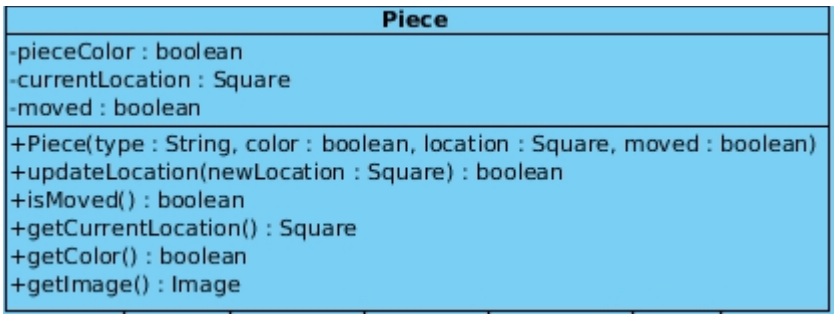


Figure 3.4.1: Piece Class Diagram

This class represents general form of each piece. It contains common attributes that each piece has. It is the parent class of each individual pieces (e.g. King, Queen, Bishop etc.).

Attributes:

private boolean pieceColor: indicates the color of the piece (e.g. white or black).

private Square currentLocation: current location of the piece on the board.

private PieceSet[] pieceSets: array of PieceSet that stores each player's piece set.

private Image boardImage: image object of the board image.

Constructors:

public Piece(String type, boolean color, Square location, boolean moved): creates an instance of the associated descendant object with the given parameters. Here, type indicates the kind of the piece such as “king”, “queen” etc.

Methods:

public boolean updateLocation(Square newLocation): updates the piece’s location to the given new location.

public boolean isMoved(boolean set): returns whether the piece has been moved or not from the beginning of the game. This is required for castling.

public Square getCurrentLocation(): returns the current location of the piece on the board.

public boolean getColor(): returns the color of the piece.

public Image getImage(): returns the piece image.

3.4.5 King Class

Figure 3.4.1: King Class Diagram

This class represents the king piece.

Attributes:

private Image kingImage: image of the king piece.

Constructors:

public King(): creates a King object.

3.4.6 Queen Class

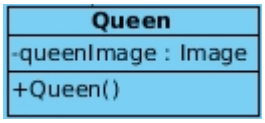


Figure 3.4.1: Queen Class Diagram

This class represents the queen piece.

Attributes:

private Image queenImage: image of the queen piece.

Constructors:

public Queen(): creates a Queen object.

3.4.7 Bishop Class

Figure 3.4.1: Bishop Class Diagram

This class represents the bishop piece.

Attributes:

private Image bishopImage: image of a bishop piece.

Constructors:

public Bishop(): creates a Bishop object.

3.4.8 Rook Class

Figure 3.4.1: Rook Class Diagram

This class represents the rook piece.

Attributes:

private Image rookImage: image of a rook piece.

Constructors:

public Rook(): creates a Rook object.

3.4.9 Knight Class

Figure 3.4.1: Knight Class Diagram

This class represents the knight piece.

Attributes:

private Image knightImage: image of a knight piece.

Constructors:

public Knight(): creates a Knight object.

3.4.10 Pawn Class

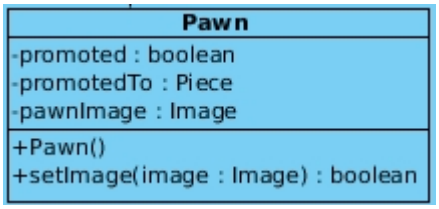


Figure 3.4.1: Pawn Class Diagram

This class represents the pawn piece.

Attributes:

private boolean promoted: indicates whether the pawn is promoted.

private Piece promotedTo: indicates the piece that the pawn was promoted to.

private Image pawnImage: image of a pawn piece.

Constructors:

public Pawn(): creates a Pawn object.

Methods:

public boolean setImage(Image image): updates the pawn's image in case of promotion.

3.5 Detailed System Design