Department of Computer Engineering

Bilkent University

CS 319 Project

Chess Battle

Design Report

Abdullah Al Wali, Shahriyar Mammadli, Melih Sancak, Mert Ege Can

1. Introduction

1.1. Purpose of the System

Chess battle extends regular chess game and enriches it by adding strategic elements. It is a system which its objective is to entertain people by forcing the users to generate different strategies with well designed gameplay. Chess battle is a user-friendly interface which provides the user a gameplay in different methods according to their aims. Even though chess battle is an extended game from regular chess, users can learn and get involved in the game quickly. There are different game modes that will increase the option of the user and gaining different strategies as regard to different gameplay. Therefore, the fundamental purpose is to design a strategic game which provides the user to increase the decision making ability with spending good time in different gameplays.

1.2. Design Goals

Before attempting to dive into the system and object design, the design goals, criteria, and trade offs will be clarified. These goals are inherited from the functional and nonfunctional requirements of the design elicited in the analysis stage.

End User Criteria:

- **Ease of Use:**

Due to the fact that the the desired system is a game, it should be simple and entertaining. Users should experience no difficulty while using the system. In that aspect, users should have a user-friendly interface in menu option. In menu options, the player should find accurate options in order to perform the desirable actions. Since our game is very simple and smooth, the default game settings will be kept simple. The settings contain game sound options, which is on or off according to desire of the user.

- **Ease of Learning:**

The system will assume that the user has no background knowledge about the game, game modes, game rules, specific win-loss conditions for specific game modes, piece roles, piece skills and usage. For this objective, the system will provide an instructive, well-documented and simple tutorial in the options menu before entering game. So that, players become familiar with the basics of the game.

Maintenance Criteria:

- **Extensibility**

The object oriented software of systems provide system specification without causing bugs during making new modification . It is crucial to add new component(new skills), features of the game(such as new game method) to sustain excitement of the player in the long run. To cite an example, if a new skill is added to knight special ability, there should not be any mistake in the existing system.

- **Portability**

  Portability is an important feature in order to address a wide range of users. The system is implemented in Java. Java Virtual Machine enables the system to be independable. Hence, the system meets the portability feature demands.

- **Reusability**

  Classes and the structure of the code will be designed in a manner and structure that encourages and facilitates adding new features to the game or reusing existing ones. It will be ensured that most of the classes are loosely coupled and supports new additions through inheritance and polymorphism.

- **Reliability**

  System will have no bugs and will be consistent in itself. The specific boundary conditions for specific inputs will be checked in the gameplay to prevent unexpected crushes. In order to make the system reliable, every stage of the system will be checked part by part when during the testing procedures. To cite an example, if the desired move is out of the board, the system should not give a warning message to user :"it is not valid move" and game should stay same until a valid move happens.

- **Effectiveness**

This system will be able to run with desired performance requirements stated in this section. Chess battle will run with at least 30 fps, in order to provide smooth movement on each piece during the move actions. Therefore, the game will be more playable with desired effectiveness.

Extensive memory storage is not expected in high frequency, since the future of the game is not complicated and allocating much space in the memory is not needed.

- **Good Documentation**

The documentation of the project plays a significant role in the base of the project. Documentation of the project should be well-organized and it should be understandable for the developers. The design report and analysis report should be consistent. Therefore, developers can understand the current situation of the system and make new modifications easier.

Performance Criteria

- **Response Time**

Response time is one of the fundamental aspects of the gameplay. The response time will be minimized between the user input and user’s desired action.

Trade-offs

- **Portability vs Performance**

For the implementation, Java was preferred over C++ in order to maximize portability as the game will be available on any system that has an implementation of the JVM. Implementing in C++ would have offered a better performance at the cost of having to distribute executables for different systems, however, we have agreed that our game does not benefit from the increased performance and thus we have decided to use Java implementation.

1.3. Definitions, Acronyms and Abbreviations

Fps[1]:It is a unit that measure display device performance. It consist of the number of complete number of complete scans of the display screen that occur in each second.

Java Virtual Machine[2]:It is an implementation of the Java virtual machine specifications, interprets compiled Java Binary code(called bytecode) for computer’s processor so that it can perform a Java program instructions.

1.4. References

1)https://www.techopedia.com/definition/7297/frames-per-second-fp
s

2)http://www.theserverside.com/definition/Java-virtual-machine-JVM

3) https://en.wikipedia.org/wiki/Java_Development_Kit

2. Software Architecture

2.1 Overview

This part is the section which our system is splitted into subsystems. By this, we simplify our complex subsystems relations, and demonstrate correlation of components of subsystems.

2.2. Architectural Style

We use the MVC three-layer architectural style for our system as it would be the most convenient way for us to split our interface, game logic, and game entities. On the View layer, we will include the user interface and the frame that contains the representation of the game. The View will delegate actions to the Controller which will control the flow of the game and check progress, validate moves, etc. The Controller updates and changes the Model which contains the information about each piece, and their abilities, and stats. The Three layers will correspond as one to one with our three subsystems: User Interface, Game Logic, and Game Entities.

2.3. Subsystem Decomposition

The top subsystem will be the User Interface Subsystem, it will contain the visual aspects of the game and will facilitate user input.

The middle subsystem is where all the logical operations take place, it will practically be the brain of our game and will be called the Game Logic Subsystem.

The bottom subsystem will contain the game model and objects which will be managed and controlled by the middle layer. We have followed an opaque layers pattern and we will not allow the first layer to access the bottom one, control on the bottom layer will have to go through the middle layer first.

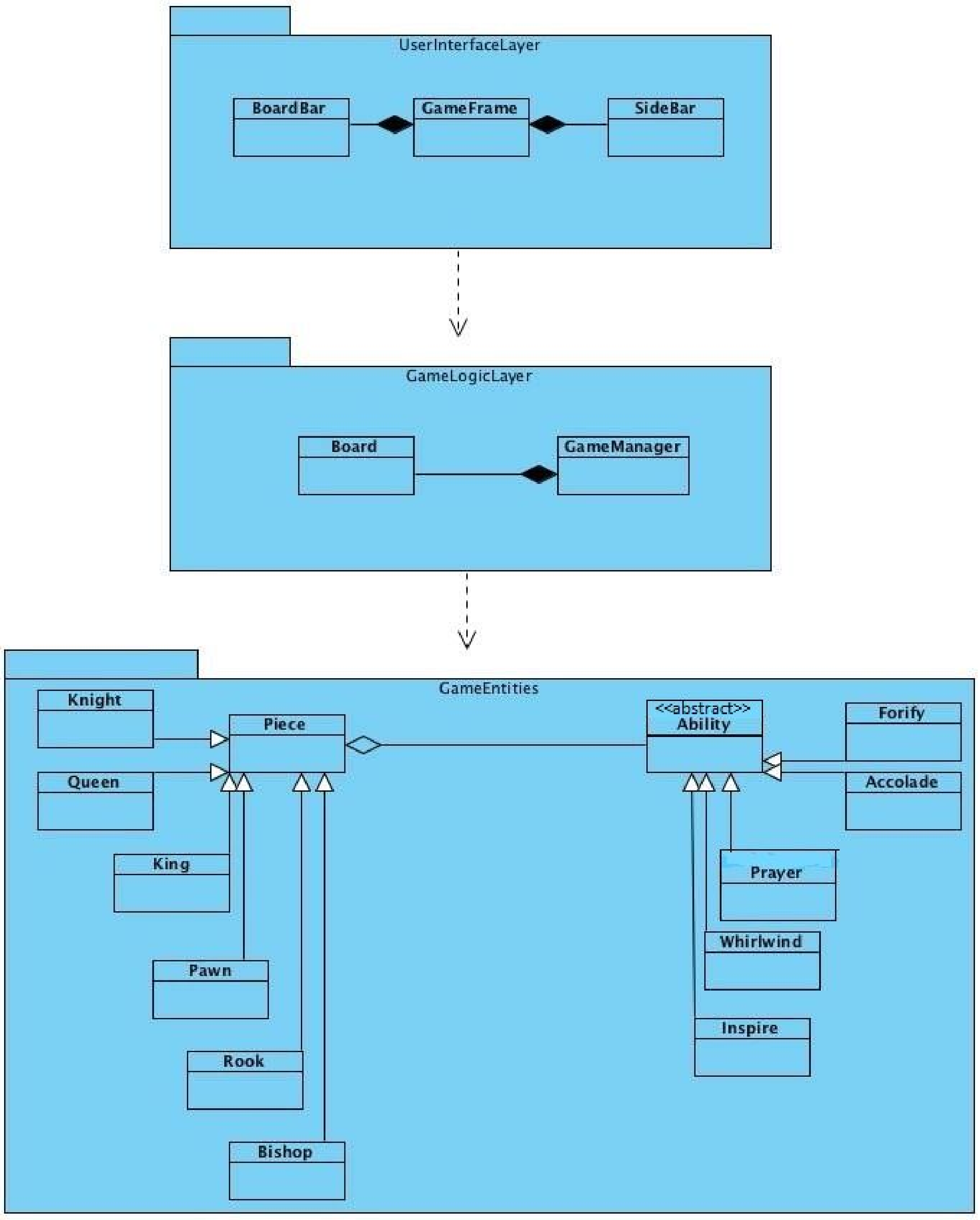

Figure 01: System Architecture

2.3. Hardware/ Software Mapping

Our game will be implemented in java language and we will need JDK for this. Because our game is not very complex, it can be player on any average computer. A Mouse is necessary to be able to play game and a keyboard will help for shortcuts and quicker moves but it is not required.

2.4. Persistent Data Management

Game data will be local and written by a single operation and therefore, there is no need for database to save them. Since the game is logically played through one sitting, there is no need to save the game state, and all the files that are necessary for the game will be stored in memory and will be reachable any time. Images, background music, and sounds will be stored unencrypted in the hard disk.

2.5. Access Control and Security

Our game will not need any kind of security prevention and is accessible for anyone. The game does not require database and network, only through its .jar file program can be executed. The User does not need to register for the game or have an account to be able to play.

2.6 Boundary Conditions

The game is in format of a .jar file and does not save game data. If user exit game by clicking exit game or “x” button through frame, game data is lost.

By running the game, it starts from the beginning and displays new game option, help and settings. In case of error occur while playing game because of performance issue, program need to be started again and game data cannot be loaded again, thus, players need to start new game. If problem shows up while loading game(cannot load piece icons, background music, etc.), it will continue without these features.

3. Subsystem Services

3.1. Design Patterns

Façade Pattern:

In order to simplify our design and reduce complexity we have decided to follow the facade design pattern, in which a class hides the complexity of the underlying system by acting as a wrapper to other classes. This proves most useful in the Game Entities Subsystem in which many classes exist (8 types of Pieces and 8 Types of Abilities), yet they are all wrapped in an abstract class that is Piece that will be a parent of the different Piece type classes and will aggregate the Ability abstract class, which will in turn be the parent of different ability types.

The Facade pattern is also used in the Game Logic subsystem where GameManager acts as a link between the User Interface layer and the Game Entities layer.

Model View Controller:

Since the project aims to provide a simple usage in gameplay we plan to use Model-View-Controller pattern because it will provide us with a simple, safe, and fast cycle between the actions which the user have taken and their effects on the game model. GameManager class will have the duty of the Controller and will update the game values according to the incoming user inputs. The changed game values by GameManager will be reflected in GameFrame, which behaves as the view module and refreshes the user interface for the upcoming user inputs.

The Model part of the system will be the Game Entities Subsystem, which contains the classes for all the game pieces and abilities and their attributes & details.

Though following the MVC pattern might lead to larger classes and cost more design and planning time, we have decided to use it since it reduces the complexity of the system on the long run and allows for fixing any problems much more easily than if we were to have one subsystem which includes all the classes. In our design we are also trying to optimize the game for any future developments or additions to the game since it will be an open source project.

3.2. User Interface Subsystem

The User Interface Subsystem will consist of the collection of classes that are responsible for displaying the game state and user interface and piece information for the user. It will also act as the View in our MVC design model. The User Interface Subsystem will have one facade class, GameFrame, which will aggregate two other classes, SidePanel and BoardPanel.

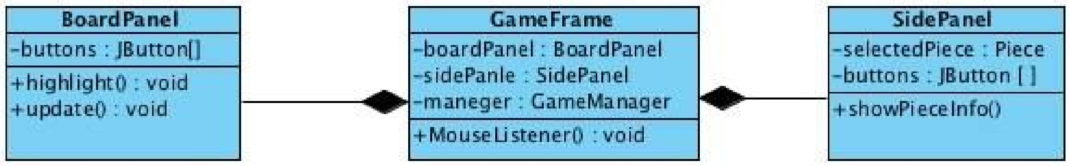

Figure 02: User interface subsystem class diagram

GameFrame Class

GameFrame Class will be the main control unit for the User Interface Subsystem. The interactions between user are controlled and transmitted to the GameManager from GameFrame. User Interface Subsystem will have two more classes which are controlled by GameFrame: BoardPanel and SidePanel.

GameFrame will have instances of both BoardPanel and SidePanel for updating them with the changes made in the end of user inputs. It also have an instance of GameManager to deliver the command which the user gave via the user interface.

BoardPanel Class

The BoardPanel Class is the largest panel that the user will encounter. It is consisted of 8x8 interactive button array (board), which the players will play the game on. The BoardPanel Class has the entities of highlighting and updating. The highlight() method will change the color of the tiles, according to the selected pieces' desired move. After the user clicks the move button in the SidePanel, the available and movable squares will be highlighted to green. A similar operation will exist for the attack button which highlights the available and attackable enemy pieces to red.

In the end of each round, the update() method will update the current state of the pieces on the board and show the user the most recent position of the pieces.

BoardPanel carries the purpose of providing a visual presentation of the game.

SidePanel Class

The SidePanel Class is designed with the purpose of containing the majority of the options that can be obtained from the user. It will show the selected pieces' information, attack, move and special ability commands.

Selected piece information will be updated by showPieceInfo() method whenever the user clicks a piece in the BoardPanel class, it will show the user, the selected pieces, the piece's color, remaining hp, ap and the remaining cooldown of its special ability.

The attack move and special ability buttons will be placed under the information box. When clicked to the attack button, the attackable pieces will be highlighted via BoardPanel class. If the user selects one of the attackable enemy pieces, it will be directed to the GameManager, and the action will take place.

Similarly, the move button behaves like the attack button, such that it will highlight the empty available squares in green, so that the user can click one of the available tiles and the GameManager will execute the move command.

The special ability button will use the pieces' skill if the skill is ready and has no cooldown left.

Any of the incorrect inputs - like clicking a friendly or empty tile in attack move, clicking to an unreachable tile in move or trying to use a special ability while cooldown is not completed- results in a cancellation of the operation and returns to the player's initial turn state. If the sequence is correct, the turn will advance and the game continues as normal.

3.3. Game Logic Subsystem

This Subsystem will contain the logic part of the game, it will control the flow of the program and perform tests, moves, attacks, etc. Its main class

GameManager, will initiate a Board instance which will contain the Pieces and methods to manage those pieces.

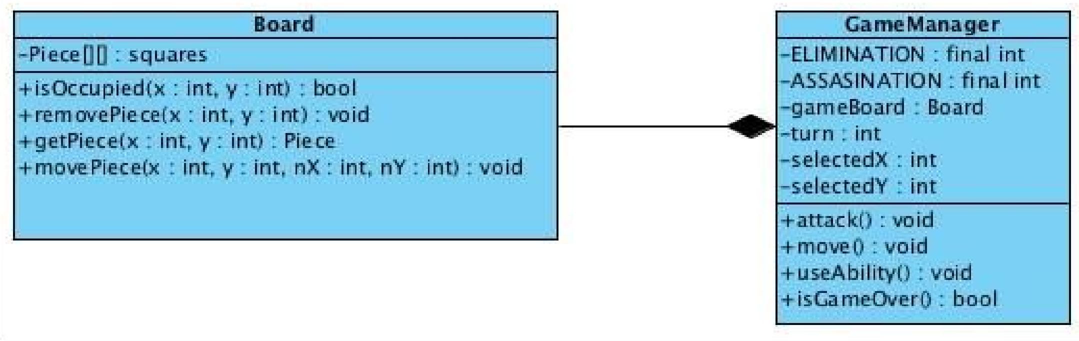

Figure 03: Game Logic subsystem class diagram

GameManager

The GameManager class will be the facade of Game Logic subsystem and will provide a link between the model and the view. It will control the flow of the game and act according to inputs from the view models. It also keeps track of the turns in a turn int attribute. The gameBoard will be an instance of Board on which the GameManager will perform some actions. attack( ), move( ), and useAbility( ) will go through checks to validate the move and then perform the updates on the gameBoard when used. isGameOver( ) will check whether the game has finished at the end of every turn.

Board

The board will contain a 2-dimensional array of Piece objects. This array represents the gameboard and will be of size 8x8. A null value inside the array will represent an empty square on the board. This check will be provided to

GameManager using isOccupied(x,y). movePiece(x, y , nX, nY) will change the position of one piece to another empty square on the board. removePiece(x ,y) will delete a piece from the board, and getPiece(x,y) will return a piece at certain coordinates if that piece exists.

3.4. Game Entities Subsystem

This subsystem contains the classes that hold information about the game and each piece and ability. It is the Model in the MVC pattern. Its facade class is the abstract Piece class, which will contain another abstract class, Ability, and will be extended by 6 children corresponding to each Piece type.

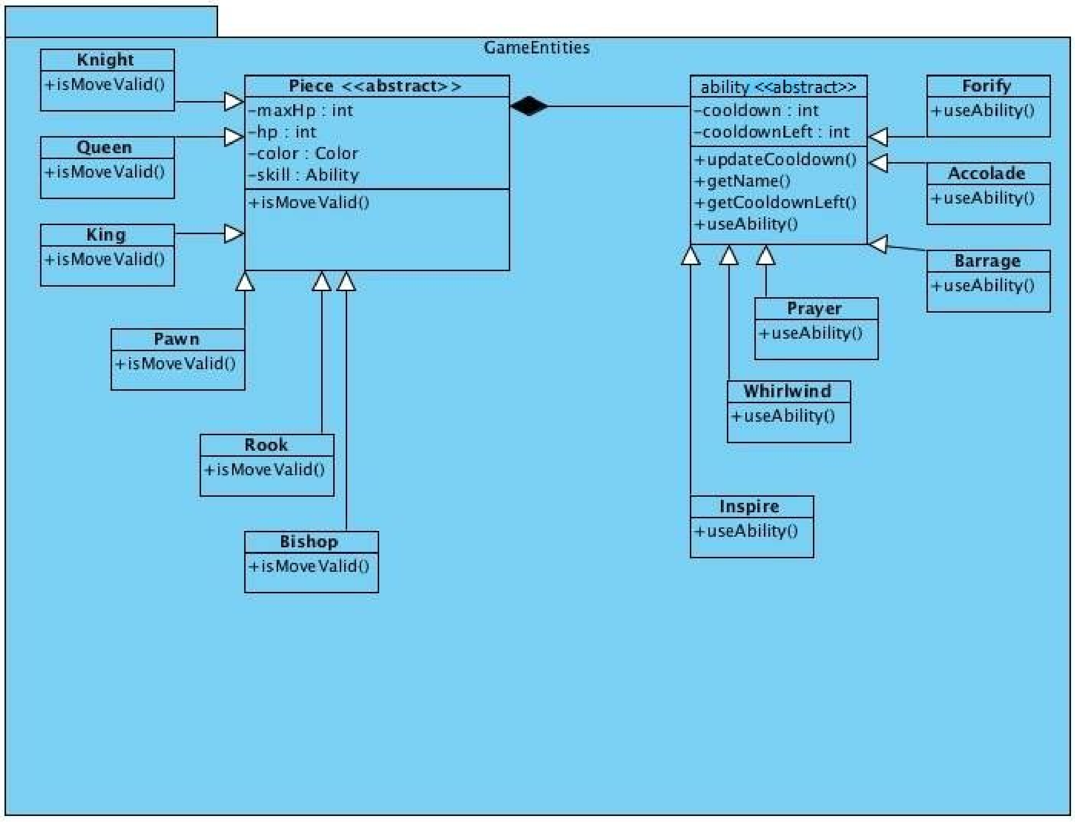

Figure 04: Game Logic subsystem class diagram

Piece Class

The Piece class contains attributes shared to every piece, such as color, maxHP, current HP (Health Points), and AP (Attack Points). It also includes an abstract method isMoveValid( ) which will be implemented by each child to find the valid moves for each Piece Type. There will be 6 Children, namely: Pawn, Rook, Knight, Bishop, Queen, and King.

Ability Class

The Ability class will be used to model special abilities of every piece. It is an abstract class and will be extended by 6 children to model the ability of every piece type. All Abilities have a shared coolDownsLeft attribute that keeps track of how many turns are left until the next use. Every child of Ability will implement its own version of useAbility according to the corresponding pieceType.