Wildcats - CSC207
Ultimate Chess

System Design Document

08/12/2021

Group Members (GitHub usernames)
JuanDeAguero
VJalal
mibrah615
guobryan
jarenworme

Repository link:
https://github.com/CSC207-UofT/course-project-wildcats-1

Table of Contents

Specification

Our application allows two users to connect on an app from separate android devices to play a chess game. Users will be able to login to their account with their saved details, or make a new account, simply by entering their username to load their data. Upon conclusion of a game, players can choose to either rematch, or exit, and their record will be saved.

SOLID Principles

The design of our program adheres very well to the SOLID design principles.

Firstly, every class follows the single responsibility principle. This is especially true of the controller classes for example, where each class handles its own activity to control the flow of data between their respective components of the game. Additionally, if we consider the entity classes, the same follows. There are six separate classes, one for each type of piece, which inherit from an abstract piece class, keeping every class only responsible for that kind of piece. Several changes were discussed and made during our work on phase one, the most noteworthy being splitting the controller classes from one controller over the entire game to now a controller for each aspect, namely login, gameplay, startup, matchmaker and piece layout.

The open/closed principle is followed as well and is most easily seen in our piece classes. As we have mentioned, there are six classes for each piece that inherit from an abstract piece class. This keeps this aspect closed for modification, but open for extension, where a new piece type can be added as its own class without affecting any of the written code. A specific example of how following this principle could prove useful would be if we wanted to add a class for a promoted pawn piece, which could track what the former piece was and be an easier way to implement promoting said piece. If PromotedPiece was added as a new entity class, it would inherit its methods from the Piece class and would not require changing any code in any other places.This example can also be used to show how our program follows the Liskov Substitution principle, where inheritance is clear and does not require changing any code to accommodate a new child class of Piece.

While the interface segregation principle can be considered to be followed, it is not adhered to as strongly as the other principles. We have no need for many interfaces, and the one we do have, GameBuilder, is still specific in its functionality. While this is not deviant enough to be considered bad design, we acknowledge that it is an area that can be improved in relation to the SOLID principles.

The dependency inversion principle is followed well and can be seen in many aspects of our program. If we consider the piece classes again, we can see that the six piece

classes depend on a higher level class, Piece, which is an abstraction. We have made a conscious effort to reduce as much coupling as possible. Each class that depends on another has been programmed in such a way so the dependent class only calls a method from such class. This allows us to change methods freely without affecting the code in the dependent class.

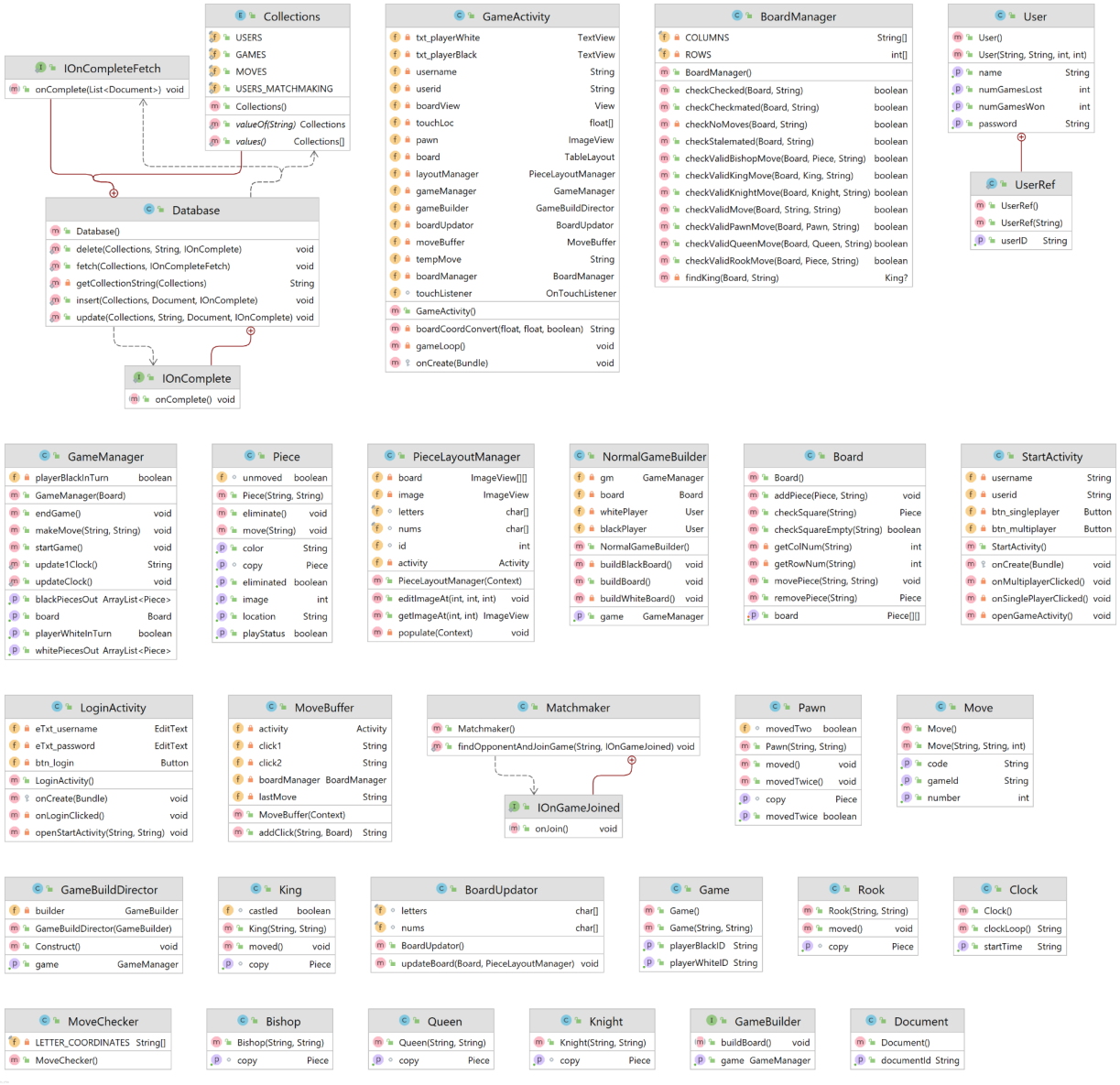
Clean Architecture

Each class in our program is designed specifically to adhere to the principle of clean architecture. Furthermore, we implement the packaging strategy to reflect this. All of our classes are in their respective package based on which level of clean architecture they relate to. For example, all classes related to pieces are in the entity package, ‘manager’ classes are in the use cases, ‘activity’ classes in the presenter/gateway/controller package and our GUI and database classes in the package for the outmost layer.

A scenario walkthrough that can demonstrate the different layers of clean architecture quite well is the event of moving a piece. First, the piece needs to be displayed to the screen (GUI) for the user to click on it. The GUI is a layout called activity_game.xml. What gets displayed onto the screen is managed by PieceLayoutManager, which is initialized in the GameActivity class, both of which are controller classes that deal with the Interfaces as well as the use cases. When a click on the screen is detected by the GameActivity class, the coordinates are passed onto the MoveBuffer class, which sits on the use case layer. This class will then store the click in an instance of itself. Upon the user clicking on where they would like to move the piece, MoveBuffer will receive the second coordinate and use the BoardManager class, along with an instance of the board to check if the move is valid. BoardManger is another use case class which can, depending on the pieces and the locations on the board, decide if a move is valid. If the move is valid, the MoveBuffer class will return the move made to the GameActivity class, where it was initialized. GameActivity will then update the PieceLayoutManager class using the BoardUpdator class, which uses the board given by GameManager and the layout in PieceLayoutManager to update what will be read from the latter to the GUI.

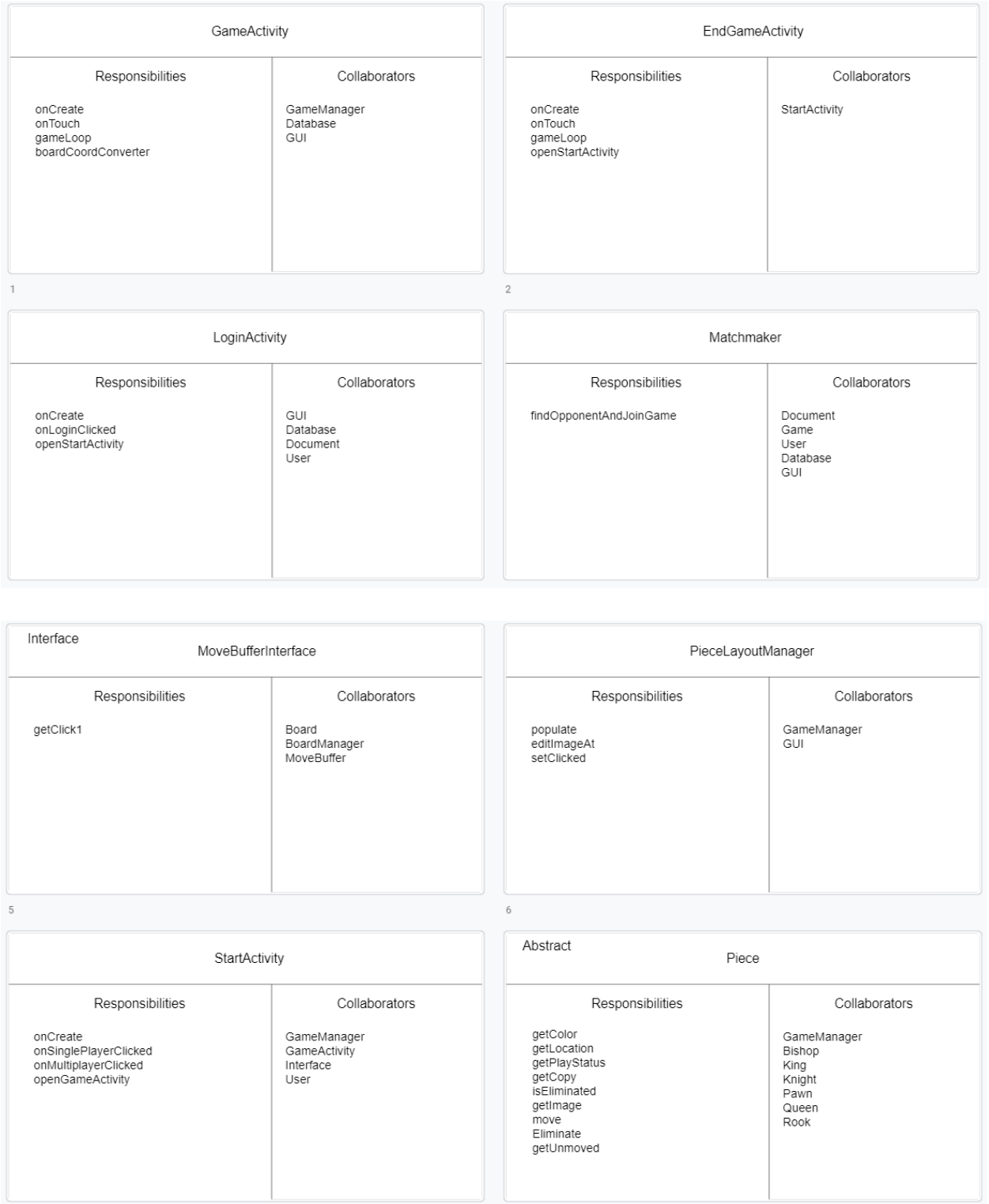
This walkthrough demonstrates the lack of interference between non-adjacent layers, thus showing Clean architecture.

User is a class stored by the database and is not an entity class. We decided to implement it this way because we determined the entity class would not perform any additional function besides storing information that would be kept in the database anyway. Instead, information about the user (wins and losses) in the database will be updated at the end of each game.



Class Diagram of Our Code

CRC Cards



GameActivity

Responsibilities	Collaborators
onCreate onTouch gameLoop boardCoordConverter	GameManager Database GUI

1

EndGameActivity

Responsibilities	Collaborators
onCreate onTouch gameLoop openStartActivity	StartActivity

2

LoginActivity

Responsibilities	Collaborators
onCreate onLoginClicked openStartActivity	GUI Database Document User

Matchmaker

Responsibilities	Collaborators
findOpponentAndJoinGame	Document Game User Database GUI

Interface

MoveBufferInterface

Responsibilities	Collaborators
getClick1	Board BoardManager MoveBuffer

5

PieceLayoutManager

Responsibilities	Collaborators
populate editImageAt setClicked	GameManager GUI

6

StartActivity

Responsibilities	Collaborators
onCreate onSinglePlayerClicked onMultiplayerClicked openGameActivity	GameManager GameActivity Interface User

Abstract

Piece

Responsibilities	Collaborators
getColor getLocation getPlayStatus getCopy isEliminated getImage move Eliminate getUnmoved	GameManager Bishop King Knight Pawn Queen Rook

Bishop	Piece
Responsibilities	Collaborators
getCopy	Piece

9

Knight	Piece
Responsibilities	Collaborators
getCopy	Piece

10

Rook	Piece
Responsibilities	Collaborators
getCopy	Piece

Queen	Piece
Responsibilities	Collaborators
getCopy	Piece

King	Piece
Responsibilities	Collaborators
getCopy	Piece

13

Pawn	Piece
Responsibilities	Collaborators
getMovedTwice movedTwice clearMovedTwice getCopy isPromoted	Piece PawnState

14

Interface PawnState	
Responsibilities	Collaborators
getMovedTwice movedTwice clearMovedTwice isPromoted	Pawn NormalPawnState PromotedPawnState

NormalPawnState	PawnState
Responsibilities	Collaborators
	Pawn PawnState

PawnState

PromotedPawnState

Responsibilities	Collaborators
	Pawn PawnState

17

Board

Responsibilities	Collaborators
getBoard setBoard addPiece removePiece movePiece checkSquareEmpty checkSquare getRowNum getColNum	Piece (and subclasses) GameManager BoardManager BoardUpdator

18

Clock

Responsibilities	Collaborators
getStartTimeWhite getStartTimeBlack clockStringConverterWhite clockStringConverterBlack	GameManager

BoardManager

Responsibilities	Collaborators
checkMoveChecker checkCheckmated checkStalemated	Board MoveChecker

MoveChecker

Responsibilities	Collaborators
checkValidMove checkValidPawnMove checkValidKnightMove checkValidBishopMove checkValidRookMove checkValidQueenMove checkValidKingMove checkChecked findKing checkNoMoves	Piece (and subclasses) Board BoardManager

21

BoardUpdator

Responsibilities	Collaborators
updateBoard	Piece Board PieceLayoutManager

22

GameBuilder

GameBuildDirector

Responsibilities	Collaborators
construct getGame	GameManager GameBuilder

Interface

GameBuilder

Responsibilities	Collaborators
buildBoard getGame	GameBuilderDirector

GameBuilder

NormalGameBuilder

Responsibilities	Collaborators
buildBoard buildWhiteBoard builderBlackBoard getGame	GameBuilderDirector GameBuilder

25

GameManager

Responsibilities	Collaborators
startGame endGame getBoard makeMove isPlayerWhiteTurn initializeClockWhite initializeClockBlack clockUpdatorWhite clockUpdatorBlack getWhitePiecesOut getBlackPiecesOut	GameActivity Piece (and subclasses) Board Clock

26

MoveBufferInterface

MoveBuffer

Responsibilities	Collaborators
addClick	Board BoardManager

GameBuilder

NormalGameBuilder

Responsibilities	Collaborators
buildBoard buildWhiteBoard buildBoardBlack getGame	GameActivity Piece (and subclasses) Board User GameManager

Database

Responsibilities	Collaborators
Collections IOnComplete IOnCompleteFetch insert update delete fetch getCollectionString	User Document

29

LocalDatabase

Responsibilities	Collaborators
loginInfoSaved saveLoginInfo getLoginInfo	Database

30

Document

Responsibilities	Collaborators
documentId getDocumentId	Game LoginActivity MatchMaker

Document

Game

Responsibilities	Collaborators
getPlayerWhiteID getPlaterBlackID	Document

Design Patterns

We decided to incorporate a builder design pattern to set up the Board and its GameManager instance. This was done with the intent of allowing easier integration of more game features later, such as customizing game modes.

For the Pawn class, a state design pattern was employed to facilitate its promotion and subsequent change in behavior.

To address very long algorithms for checking moves, we may apply a strategy design pattern. This way each algorithm can be implemented in separate classes, and as such would be easier to manage if changes needed to be made for each case.

Use of GitHub Features

We utilised Github features quite well during the course of working on our project. Firstly, we each created our own branches to work on once we had a substantial enough main branch. We would each commit and push our changes to our own branches once we had finished a certain aspect, and used a couple pull requests when we thought we had done enough to warrant updating the main branch. Since we worked separately for the most part on our respective aspects of the program, there was no need for any pull requests until later on. We had some merge conflict issues that were eventually overcome through running Git in a console and manually downloading/uploading files.

In phase two, we followed this same pattern, implementing pull requests after substantial work was done on a team member's branch. We only accepted pull requests after we had each member talk about their work, and we broke down the work into pros/cons until we collectively agreed.

We also used the issue feature to raise an issue one of our members had, and we documented and closed it once the issue was rectified in our code.

Code Style and Documentation

Our code style and documentation is one of the stronger aspects of our work. We have used Javadoc and other snippets of documentation for almost every method in every class. This has proven to be helpful while coding as it makes it easier to understand what other members of our group did as we reviewed their work. Our aim of reducing this to zero warnings by the end of phase 2 was a tough goal to reach, but currently there are very few warnings in our code as a whole and none of them impact how the code runs. We try to implement good style as well, applying uses of good spacing, reducing instances of repetitive code, utilizing consistent indents and keeping methods short. This makes our code more easily readable, comprehensive and understandable for our other group members, as well as for any user who opens our repository to examine our code.

Testing

Many use cases and entities were tested, such as the GameManager and Board classes. Phase 2 incorporates more extensive unit tests of these classes. Methods for these classes are relatively straightforward to test. However, the Interface and Controller classes remain untested because the functionality and interactions between their associated classes are complicated to test. These outer layer components were better tested by running emulators and manually trying out different interactions, to cover a breadth of different scenarios.

Refactoring

There are many methods for checking moves, and most of them are considerably long. They remain bloated because the algorithms were often complex. Much refactoring was done in Phase 2 to incorporate more design patterns in areas that could utilize them. For example, the state design pattern required refactoring the Pawn class.

Code Organization

As touched on earlier, we structured our code into packages based on what level of clean architecture it referred to, i.e., we have four packages with one for entities, one for use cases, one for controllers and one for interfaces. Each class goes into a package based on which layer of clean architecture the class belongs to. This helps us find classes easily and has the added benefit of grouping similar classes as similar classes occupy the same layer of clean architecture. For example, all of the piece classes are in the entity package.

Additionally, subpackages within each package were made to further organize the classes, with specific classes with subclasses contained together.

Functionality

Our code is functional in most aspects. The program runs and the android AVD is instantiated correctly. The gui responds to user interaction and successfully moves pieces, as outlined in our specification, with small bugs for certain pieces and special moves. The login feature is also functional and stores user data in a database that is successfully retrieved when a user logs into the app. The matchmaking aspect of the program is also functional.

Running the program

Currently, the program only runs on android, and you would need an android device to do so. Here is a link for help running an android emulator on a computer.

https://developer.android.com/studio/run/emulator

- Open the app.
- Log in with username and password.
- Select multiplayer. The game will start matchmaking.
- Once another player is found the game page will open.
- Now you can play chess!
- Move your pieces by clicking a piece and the location you want to move it to.

Technologies Used

Android Studio: This program offers the better capabilities and technologies required to build an Android application. We decided to use Android as our target platform because it is compatible with Java and IDE is very similar to IntelliJ. It is also easy to debug the program using Android emulators on our computers.

Firebase: There are several ways of implementing multiplayer on an app but we found that using Firebase was relatively straightforward and had a good integration within Android Studio. Firebase is a database system so instead of having a server handling the moves in the game, we store the moves in the database and the players can update their board by fetching it.

SQLite: To store information locally such as the user's info we needed a light and easy to use database system. This is why we chose SQLite, it allows basic database operations and it is stored in a single file inside the Android device.

Expansion

These are a couple ways outlining how our app can be modified and extended in the future, which we could not implement due to time constraints:

- Expanding the app to platforms other than android is the most significant area where this program can be extended in the future
- Extra formatting options to allow the user to choose their own themes for customisation of its visual aspect
- Additional single player functionality such as a pause screen to stop you timer when playing against an AI

Roles and Responsibilities

See Progress Report

Accessibility Report

1. Principles of Universal design

Equitable Use

Firstly, our entire design is identical to all users who may access it, with no suggestions of stigmatization in any aspects. The login feature provides privacy and security for all users. The use of an android app in the formatting we have implemented makes the design appealing to all users.

Flexibility in Use

There is a large feature of our gui where a user has a choice for methods of use, either to connect to another device for multiplayer or by playing a single player version by yourself. The nature of chess accommodates both right- and left-handed access and use, and we designed the gui such that all of our icons are centered in the screen, further accommodating both. We have also made the piece icons large enough to easily facilitate the user's accuracy and precision.

Simple and Intuitive Use

Our GUI provides effective prompting from the feature we implemented where a selected square is highlighted before a player moves. Chess by nature is very abstract and has no language barriers. The only text we provide is for the login and choice between single player and multiplayer. While minor, this is an area that can be implemented to further adhere to this principle by letting the user have a choice of language at the beginning.

Perceptible Information

We present adequate contrast between both the chessboard and its icons, and the clocks and forfeit button and the background. We have also made sure to implement a large font size in the clock feature so it appears large enough on the screen for any user.One feature that can be added in the future to further adhere to this principle is providing a sound aspect to the game so that the user gets further feedback to let them know they have moved a piece, or when the clock hits certain increments in time.

Tolerance for Error

Firstly, we have arranged the gui in such a way where it is not easy to press a button by accident that would be a hazard. The forfeit button is at the top of the screen away from the board. Other than that, this principle does not apply to our program as there are very little hazards surrounding a chess game.

Low Physical Effort

The gui allows the user to play a game of chess with just taps or chicks on the screen of their device. There are no unnecessary actions that the user could possibly perform and a low physical effort required.

Size and Space for Approach and Use

This aspect is not fully applicable to our program, we do not outline any constraints on approach and use. A user can access the app on the full size of their device and they choose how to approach it.

2. Marketable Audience

The program would be most successfully marketed to individuals who already know how to play chess as a physical board game. We would advertise the ease of playing on your phone without having to lug around a chess board and chess pieces, and also emphasize the feature of being able to play against an AI or anyone that is not physically with you that has the app. It may also help to market to a younger to middle aged audience who have a higher chance of having a smartphone than more elderly people.

3. Demographic Use

Unfortunately, due to the nature of a chess game, there is not much accessibility for the blind community. There is no feasible way we can think of to adapt the chess game, in its digital format especially, that the blind would have any meaningful or reasonable

interactions with it. On the other hand, the deaf community will have no issues using the program as it includes only visual data.