**DESIGN DOCUMENT**

**Specification:** Our application allows two users to connect onto a game 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**

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.

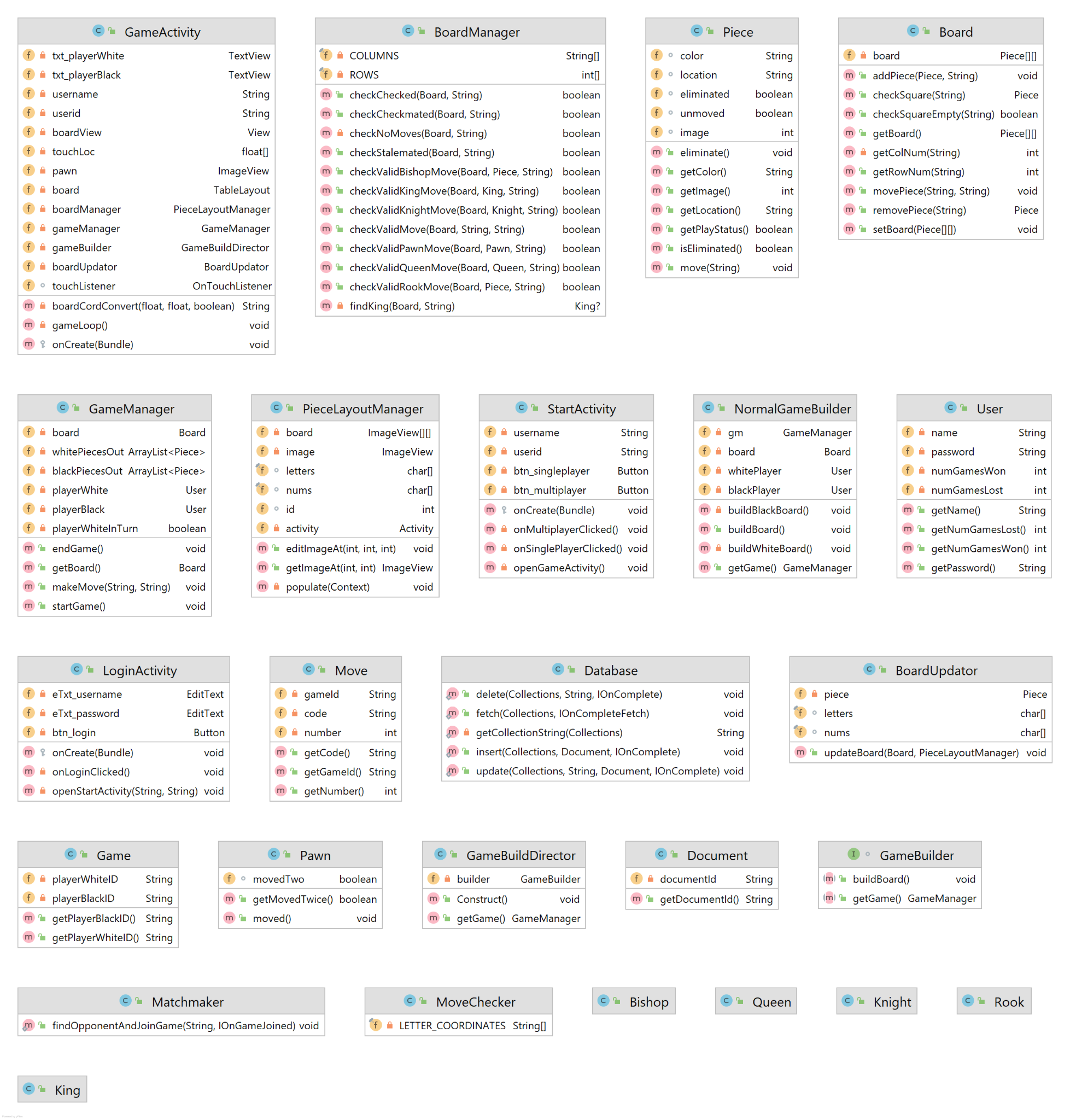
The dependency inversion principle is followed well and can be seen in many aspects of our program. If we consider the piece classes again, we can see that the six piece classes depend on a higher level class, Piece, which is an abstraction. We have made a conscious effort to reduce as much coupling 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.

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.

**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. Brief scenario walkthrough:



Class Diagram of Our Code

**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.

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 and wanted to update 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.

**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. There are very few warnings in our code as a whole and none of them impacts how the code runs. Nevertheless, we aim to reduce this to zero warnings by the end of phase 2. 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. 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 trying out interactions.

**Refactoring**

There are many methods for checking moves, and most of them are considerably long. They remain bloated because the algorithms were often complex. We may implement a strategy design pattern to separate the algorithms for each piece.

**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.

**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.