System Design Document for Mindchess

Elias Carlson, Elias Hallberg, Arvid Holmqvist, Erik Wessman

October 23rd, 2020

1.0

1 Introduction

This document describes the design of Mindchess. Mindchess is a desktop client of the internationally renowned and ancient game of chess. The purpose of the application is to be a standalone version of chess built with object-oriented design. Its functionality revolves around playing locally against another player or against a computer (AI) player. The rules of chess will be applied automatically and users will only have to worry about strategy, rather than debating over rules. The users the application targets are people who want a local version of chess where they can play against friends or family. This document will describe how mindchess is designed to reach its goals and tries to make the reader understand the reasons behind the design.

1.1 Definitions, acronyms, and abbreviations

GUI - Graphical User Interface: the visual interface of an application, with which users interact.

Object-Oriented Design: rules and patterns used in object-oriented languages such as java to create solid applications.

User Stories: a tool that's commonly used and plays a large role in agile workflows/SCRUM to divide and distribute work as well track progress.

Domain Model: a domain model is an abstract overview of a domain's objects and their relations with each other. Commonly used to communicate and plan between different divisions of a project.

MVC - Model-View-Controller: MVC is a way to divide an application in three components. Its main intention is to separate the application data and logic, the model from the presentation and interaction of the application to the user.

Computer player (AI): A virtual player that can be chosen as your opponent and makes moves as if you were playing against another player online.

2 System architecture

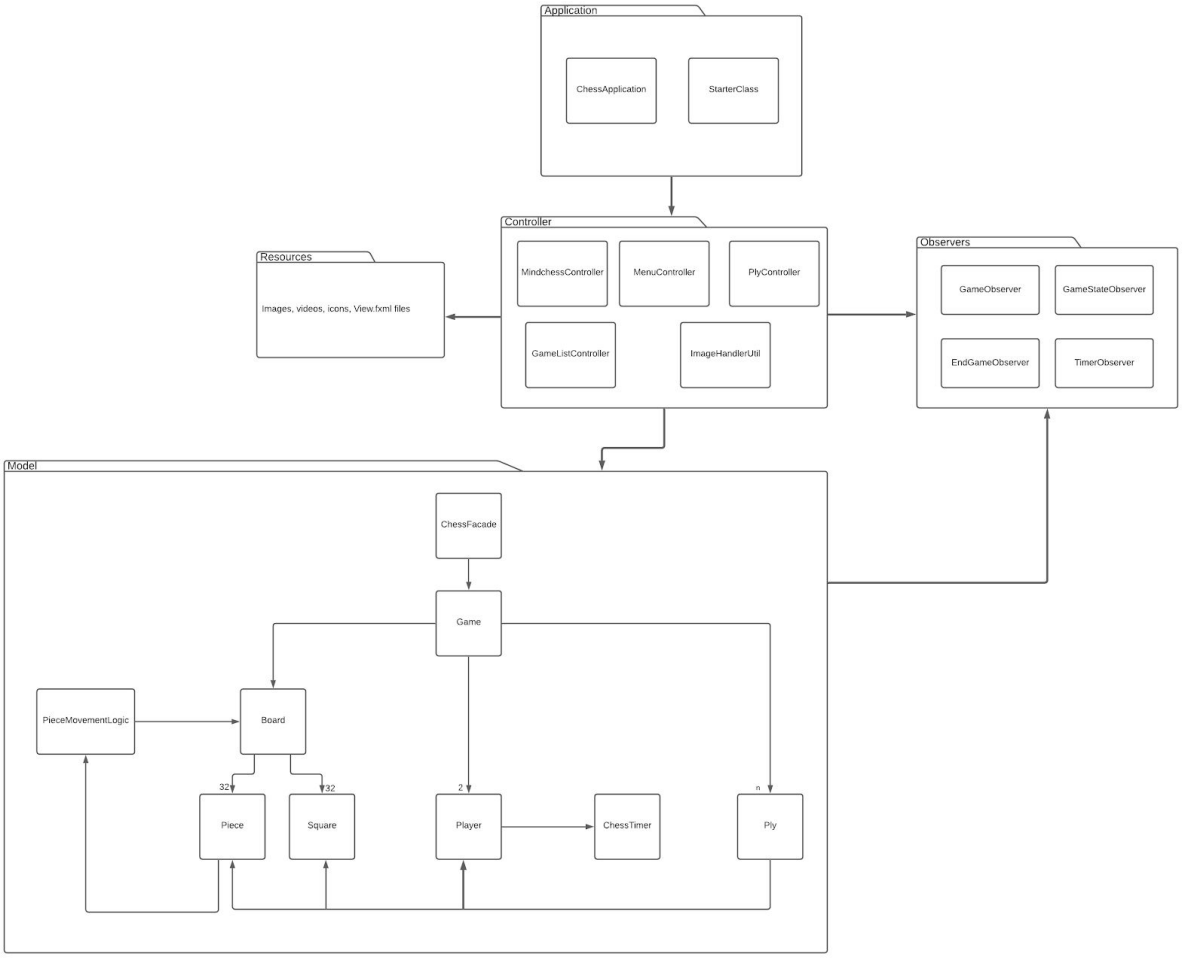


Figure 1. System Architecture [1]

When the program is started, the *StarterClass* starts the *ChessApplication* class which is responsible for initiating the components of the program. It creates the controllers and connects them to their respective .fxml file using the JavaFX package. The Controller package handles updating the views (.fxml files) when there are changes in the model, fetching images from the resource package and relaying input from the user to the model.

The resource package is where the .fxml files, piece & board images and background videos are stored.

The model package is responsible for handling the logic of the application. It keeps track of active games and their players, pieces and boards. When a move is to be made or a piece is to be marked, it’s the model that makes sure it’s done according to the rules of chess.

Two helper packages are used in the application, namely java.Util and java.javaFX. These are used amongst other things used for views, keeping track of time and storing the positions of the game's pieces.

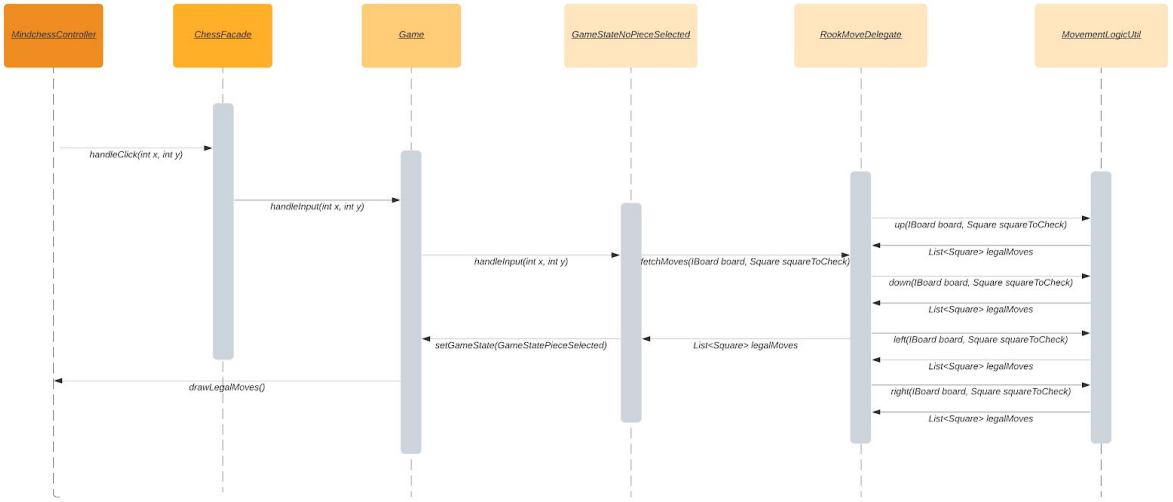


Figure 2. handleClick() sequence diagram [1]

Figure 2 describes the sequence of events that occur when a user selects a rook on the board. The *MindchessController* class registers the input and forwards the request to the *ChessFacade* which in turns sends it to the *Game* class. In this case the game is in a state where no piece has been selected, the default state, and so this request is sent to that state. Logic inside the *GameStateNoPieceSelected* calls upon the appropriate movement delegation class, in this case R*ookMoveDelegation*, which fetches a list of legal moves from *MovementLogicUtil* and returns it to *GameStateNoPieceSelected*. The state of the game is then switched to *GameStatePieceSelected* and finally a command is sent via an observer that tells the *MindchessController* to draw all legal moves for the rook.

MVC(A):

MindChess is split up into four parts: the Model, View, Controller and Application. The Application package starts the program and injects the Controllers with information from the .fxml files. The Model contains the game logic and information. The Controllers convert the information contained in the Model and display it in the View which is dynamically updated when, for example, a piece is moved. The View is represented by .fxml files.

Module pattern:

Our file organization and structure is made in accordance with the Module pattern. This means that related Classes in our program are organized in packages, in our case these are the model, controller and resources packages.

3 System Design

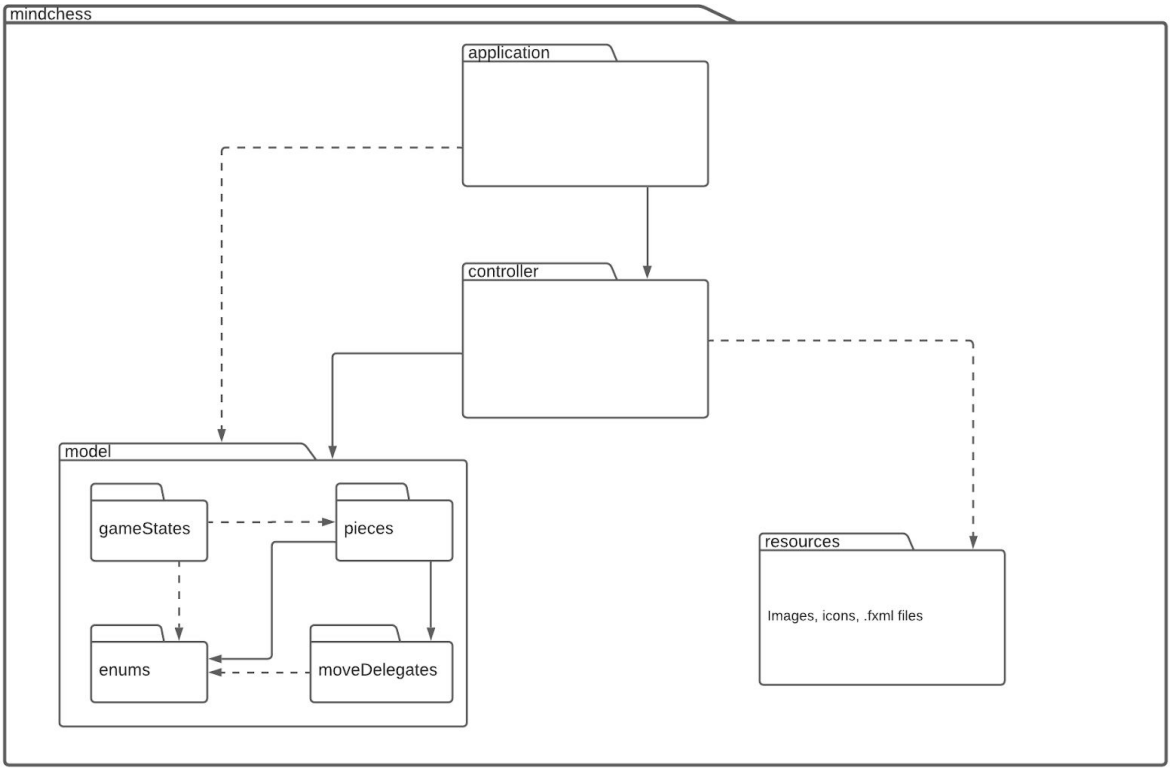


Figure 3. Package diagram [1]

The Controller package is responsible for updating the views, in the form of .fxml files, and does that by getting data from the model package. The Observer pattern is also used for the model to be able to tell the controller when to update the views, for example when a move is made. Input from the user is also handled by the Controller package and relayed to the model when appropriate.

Besides these packages there also exists a Application package that initializes both the model and the controllers, and starts the program. This is in accordance with the MVC(A) design pattern, where the .fxml files represent the Views.

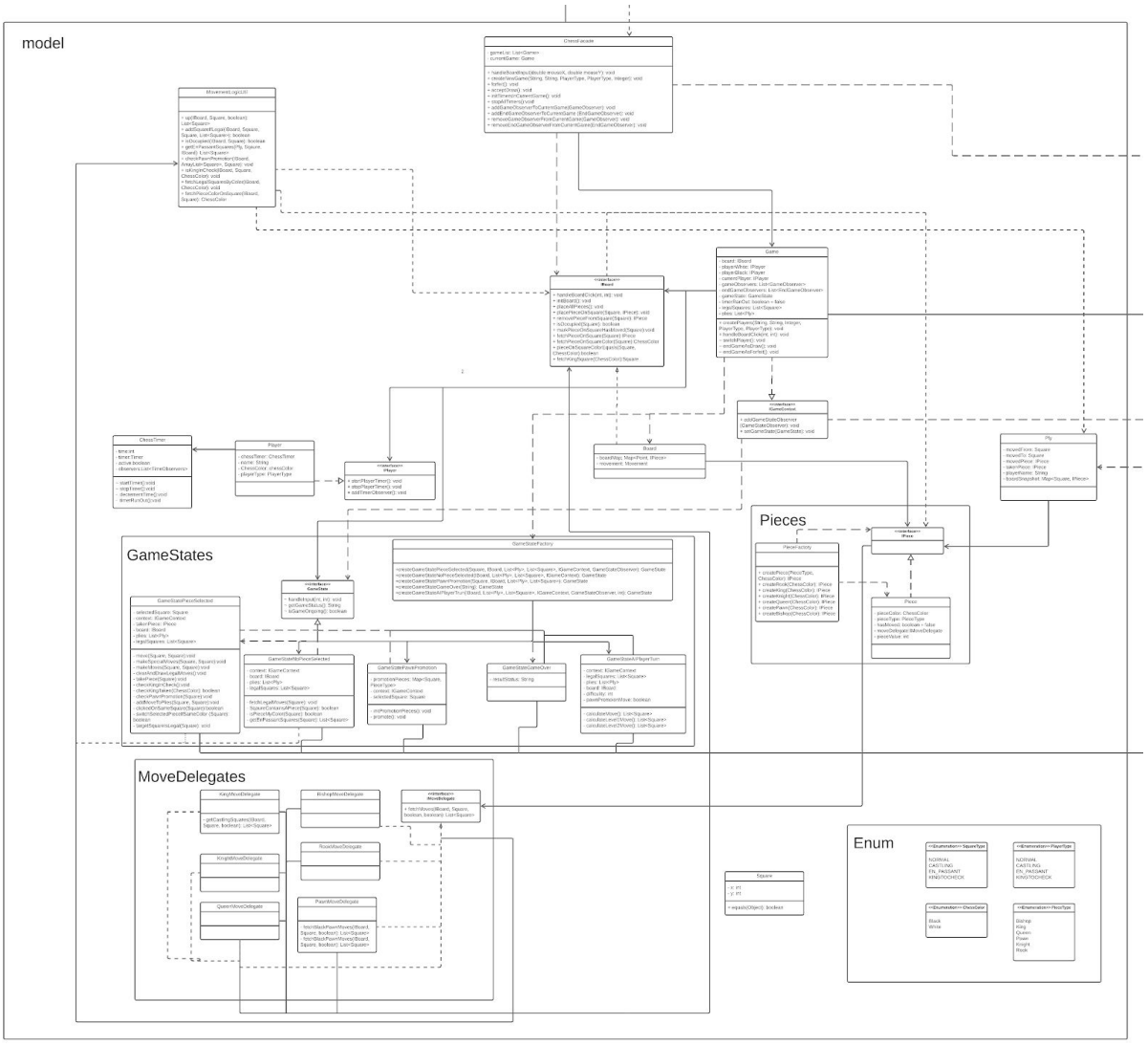


Figure 4. Model package UML diagram [1]

When the program is started, the *ChessFacade* class acts as the interface for the model. It initializes an instance of the *Game* class which in turn initializes the instances of *Board* and *Player* needed for a *Game* of chess. The *ChessFacade* relays input from the application to the *Game* class which then handles delegation of the application logic.

For example when a click is made to either mark or move a piece, the *Game* class makes sure the click is handled correctly. This is done with the help of the *Board* and *Movement delegate* classes, and based on the positions of the pieces on the board. Every move made is saved as an instance of *Ply* and kept in the *Game* class. These moves are used to check special conditions for certain rules as well as observing the match after it has finished. The *Game* class is also responsible for alternating between the two players and making sure that a player can only move his or her pieces. While doing this, it also makes sure the current player's *ChessTimer* is ticking down while it is their turn.

Facade pattern:

The Facade pattern is used in the model package where the *ChessFacade* class represents the interface that other packages can interact with which makes it safer and more simple to interact with the model.

State pattern:

The state pattern is used in the *Game* class to delegate the method which handles input to the chess board. The state changes internally and will handle the input differently depending on the current state it's in. State pattern allowed the *Game* class to be more extensible by being able to add more states to it. It also removed the need for *Game* class to constantly check its own state with booleans and if statements.

Observer pattern:

The Observer pattern is used in two different places in MindChess. The first is with the *ChessFacade* class as an Observable, that is observed by the *MindChessController* in order to update the board and pieces when a piece is marked and/or moved. The second use is with the *ChessTimer* class, which is also observed by *MindChessController*, so that the View can be updated when the timers ticks down.

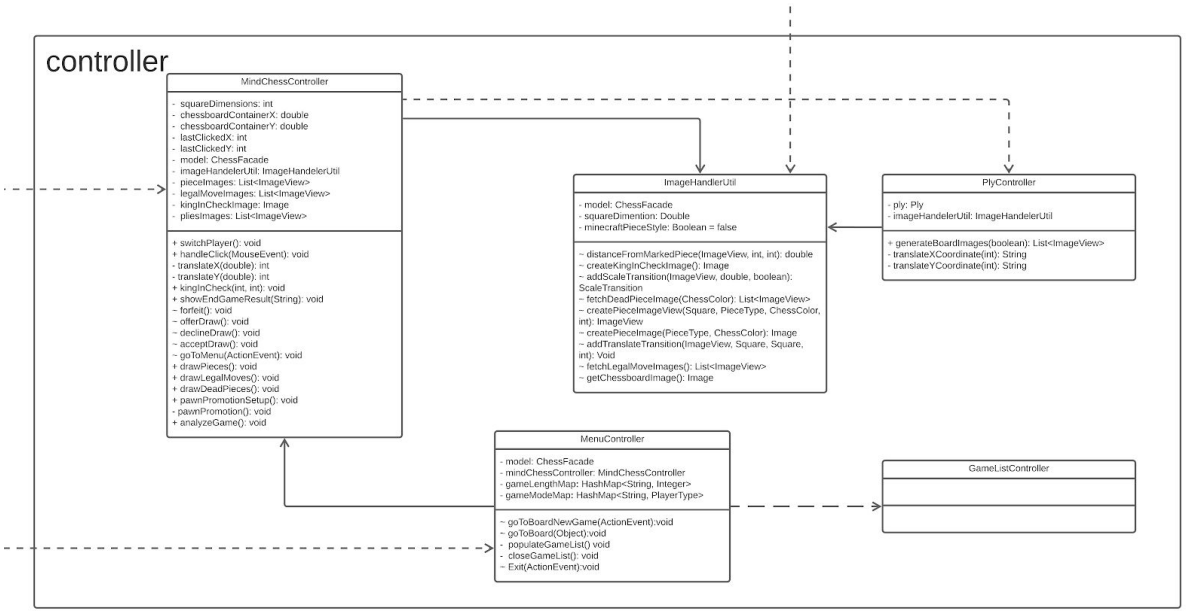


Figure 5. Controller package UML diagram [1]

MindChess' controller package contains five different classes, four of which are controllers that handle a respective View and one that is responsible for fetching the application's images. The *MenuController* class, as the name suggests, is the one that handles the menu View, which allows users to start a game, with a few options, or exit the program. The *MindChessController* class has a few more responsibilities, namely updating the chessboard View based on the current state of the model and when the respective player timers are updated as well as relaying input from the user to the model. *GameListController* however controls the items in the pane that appears when you want to load an old game and sends

the information of the game you choose to the model. *PlyController* is the one that takes care of the items in the pane that appears when you want to analyze a game and shows you the moves that happened on the turn of the item that you pressed. Lastly the *ImageHandlerUtil* class is responsible for matching the pieces of the board with their respective images, and providing these images to the *ChessController*. This is done so that the model does not need to depend on any image to function.

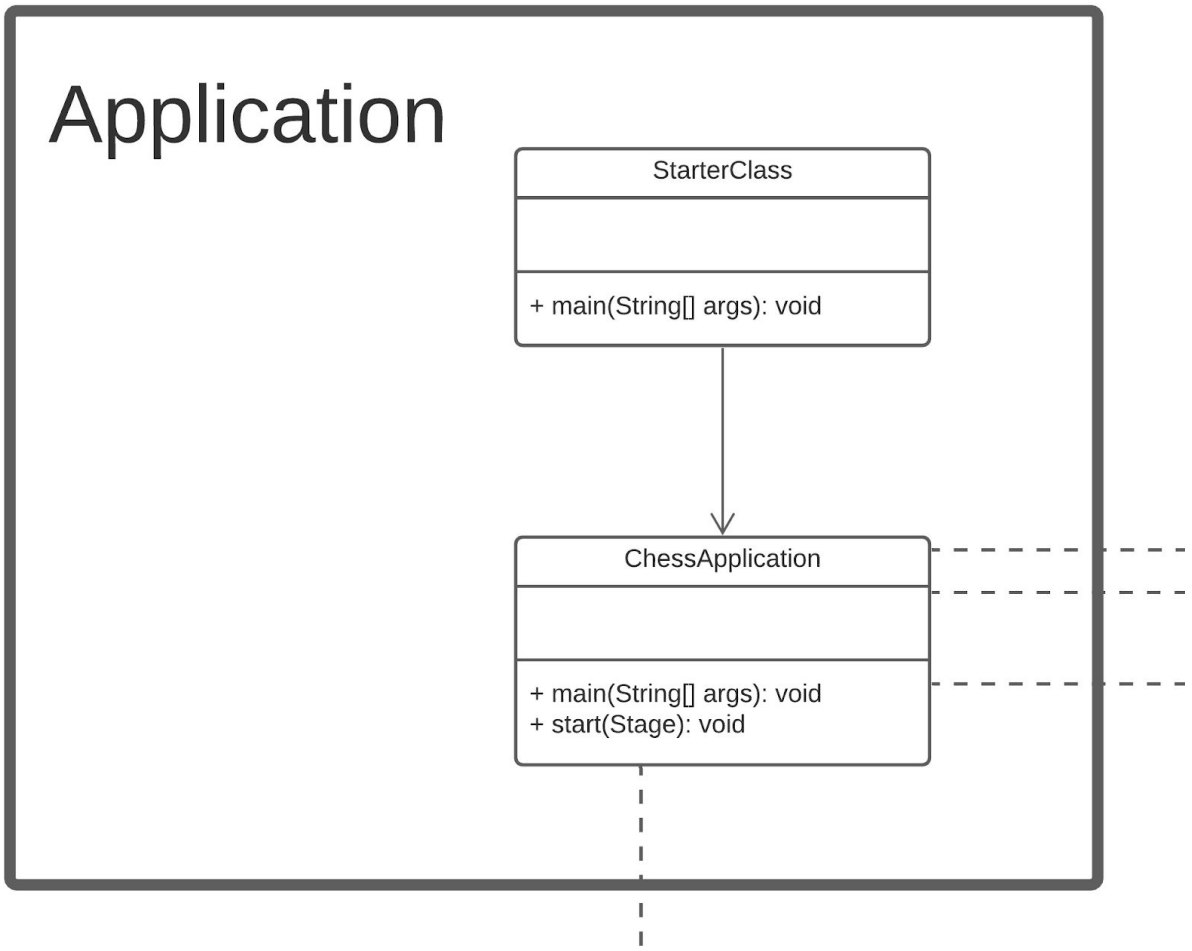


Figure 6. Application package UML diagram [1]

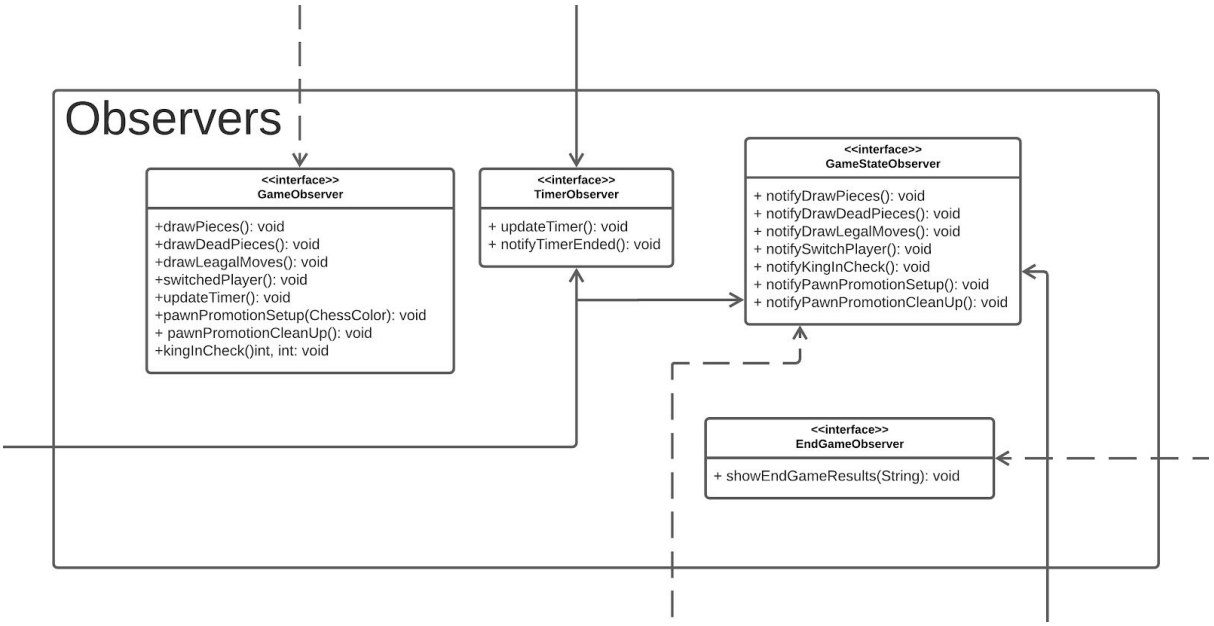


Figure 7. Application package UML diagram [1]

4 Persistent data management

Mindchess does not store any persistent data. It does however access other data in a resources folder which contains images and videos that are used in the application.

5 Quality

To test MindChess the testing framework JUnit [2] is being used making it easy to continuously test the program through its iterations. The tests are located in their own folder called test where either classes or functions have their tests.

The continuous integration tool Travis [3] is used to make integration easier and also to make the testing process automated for each commit to ensure a bug free and stable program.

When developing new features or fixes for MindChess pull requests were used and had to be reviewed by at least two people to ensure the code holds high quality and is understandable by others.

As we were unable to perfect the application on time is there a couple of known issues:

- Edge cases for when we check if the king can move into check, for example:
    - Can move into positions threatened by pawns.
    - When taking a piece can the king move into check.
- Saving and loading game the same game when pawn promotion is in progress causes the pawn promotion pane to disappear and the game to freeze

- If you have too many characters in your name then it does not show the whole name at places.
- When you load a finished game does it not immediately show that it has ended and the user has to click on the board to show the end game screen.
- If you load a game that the AI has won do you have to move a black piece to show that a game has ended.

Analytic tools that were just during development were mainly PMD, JDepend and dependency diagrams generated using IntelliJ. Below we will show the results for all of these on the final version of the program.

PMD Results

The following document contains the results of PMD 6.23.0.

Violations By Priority

Priority 3

mindchess/controller/MenuController.java

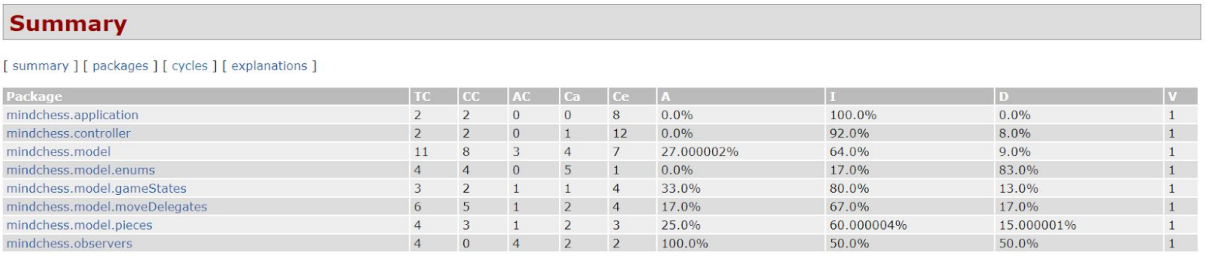
Rule	Violation	Line
UnusedPrivateMethod	Avoid unused private methods such as 'populateGameList()'.	158
UnusedPrivateMethod	Avoid unused private methods such as 'closeGameList()'.	177

mindchess/controller/MindchessController.java

Rule	Violation	Line
UnusedPrivateMethod	Avoid unused private methods such as 'switchPieceStyle()'.	399
UnusedPrivateMethod	Avoid unused private methods such as 'muteUnmute()'.	412

Figure 8. The PMD report for the latest version of the program. [4]

The PMD report in Figure 8 only shows 4 violations of priority 3. It says the violation is that the methods listed aren't used but they are used in runtime when input is given through the GUI. Therefore these violations are not in fact violations.



Summary

[summary] [packages] [cycles] [explanations]

Package	TC	CC	AC	Ca	Ce	A	I	D	V
mindchess.application	2	2	0	0	8	0.0%	100.0%	0.0%	1
mindchess.controller	2	2	0	1	12	0.0%	92.0%	8.0%	1
mindchess.model	11	8	3	4	7	27.000002%	64.0%	9.0%	1
mindchess.model.enums	4	4	0	5	1	0.0%	17.0%	83.0%	1
mindchess.model.gameStates	3	2	1	1	4	33.0%	80.0%	13.0%	1
mindchess.model.moveDelegates	6	5	1	2	4	17.0%	67.0%	17.0%	1
mindchess.model.pieces	4	3	1	2	3	25.0%	60.000004%	15.000001%	1
mindchess.observers	4	0	4	2	2	100.0%	50.0%	50.0%	1

Figure 9. The JDepend report summary for the latest version of the program. [5]

JDepend's summary in Figure 9 shows the different packages in the application and the results for each of its metrics listed beside each package. The most notable one here is instability since it is a measure of how dependent the package is on other packages, where 0% is the ideal score and 100% the worst. For our application this value is fairly high for most packages, which looks worrying, but on closer examination it can be found that the vast majority of the calculated couplings are dependencies of standard java packages (such as java.lang, java.util etc). This means these dependencies are practically inescapable and beyond these the coupling is kept low.

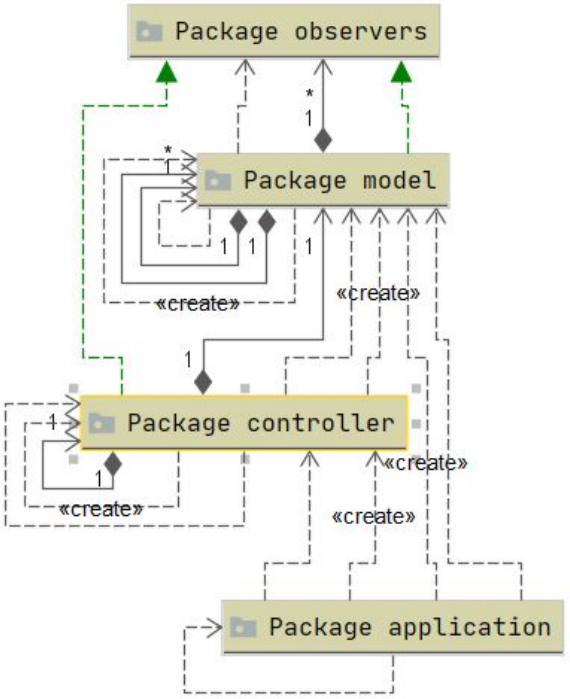


Figure 10. Dependency diagram of Mindchess' packages, generated by IntelliJ [6].

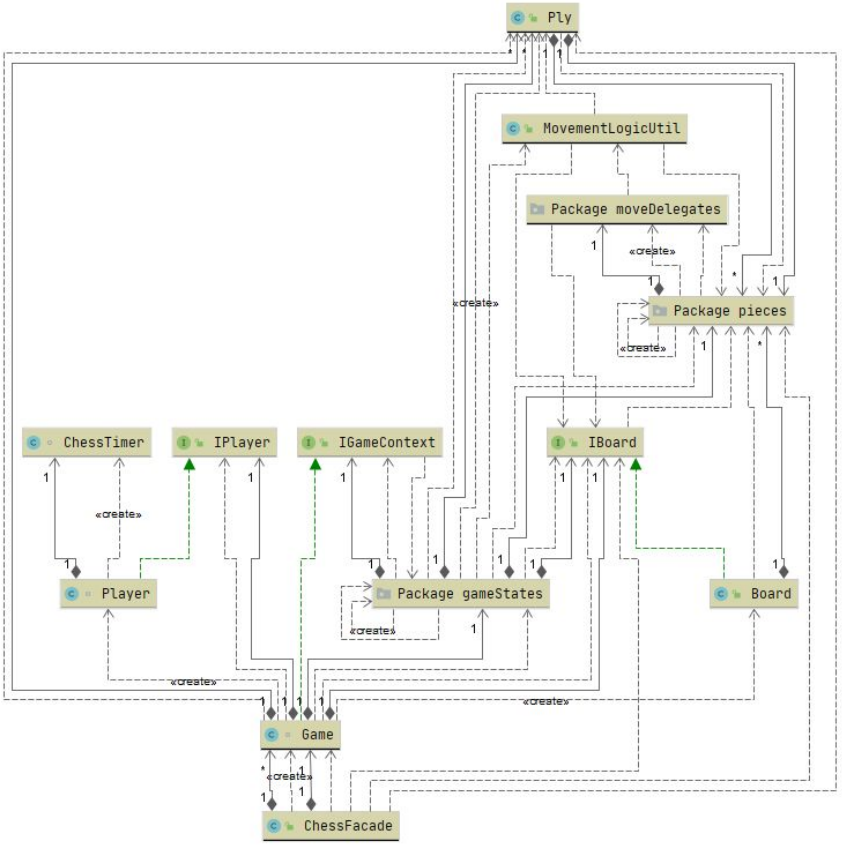


Figure 11. Dependency diagram of Mindchess' model package (excluding enums and Square), generated by IntelliJ [6].

5.1 Access control and security

NA

6 References

[1] LucidChart, "Online Diagram Software & Visual Solution", 2020. [Online]. Available: https://www.lucidchart.com/pages/, accessed on: 2020-10-02
[2] JUnit, "JUnit - About", 2020. [Online]. Available: https://junit.org/junit4/, accessed on: 2020-10-02
[3] Travis CI, "elias-carlson/OOP-Projekt-TDA367", 2020. [Online]. Available: https://travis-ci.org/github/elias-carlson/OOP-Projekt-TDA367 , accessed on 2020-10-23
[4] Maven PMD Plugin, "Apache Maven PMD Plugin - Introduction", 2020. [Online]. Available: https://maven.apache.org/plugins/maven-pmd-plugin/ , accessed on 2020-10-23
[5] Maven JDepend Plugin, "JDepend Maven Plugin - Introduction", 2020. [Online]. Available: https://www.mojohaus.org/jdepend-maven-plugin/ , accessed on 2020-10-23
[6] IntelliJ IDEA Ultimate, Version 2020.2.3, [Software], Prague, Czech Republic: JetBrains 2020