CS246 Chess Documentation and Guide

By: Mike Vu, Daniel McIntosh, Allen Fan

Program Architecture

The program overall is comprised of a View, Controller, Game, Move, Player and Piece classes with minimizing coupling and cohesion as a first priority and the use of design patterns as a second priority.

Here is a quick rundown of the classes, their functionality, and their coupling and cohesion with the other classes within the program:

View: Controls all IO with the player, has a general input method that takes commands from users and multiple output methods to output to the user.

- Coupling: *Minimal* does not affect the flow of other classes, only uses method parameters to pass data around, and has no access to any other class.
- Cohesion: *High* only responsibility of this class is to handle I/O

Controller: Responsible for the game's runtime. This class contains the programs main function that contains the logic which dictates the flow of the game. This class is the interface that calls the game's setup, the start/end of the game and also controls the games turns and match making.

- Coupling: Low this class controls the flow of game and players but has no access to any other classes.
- Cohesion: High this class only controls the flow of the game and makes calls to View

Game: Holds the games data, such as the board and piece locations and has the majority of the logic for the games rules. This class interprets move commands, checks the validity of the command (checks if it is a legal move in chess, this includes things like checks and piece moves) and then performs the move which alters the board.

- Coupling: Low this class controls the certain aspects of chesses logic, but has no access to other classes
- Cohesion: Low as this class has a large amount of functions that serves the same role but can be broken down further into subclasses

Move: An object which serves as data block that is passed between the player and the game. Has the data for a player's move in chess.

- Note: This is an object passed between classes in the program and is completely independent

Player: Comprised of a Human and AI subclass. The Human class allows users to play the game and the AI subclass holds logic for the computer to play the game. In addition the AI subclass has 4 unique levels of behavior.

- Coupling: High has read access to the entire game class
- Cohesion: High methods in this class only send moves to the board, AI subclass has set of pieces for decision making

Piece: Contains the Pawn, Bishop, Queen, King, Rook and Knight subclasses that define the behavior of each type of piece. Each Piece contains a method to check if a move is within its move set and capture set and data regarding the type of piece it is, the colour of the piece and whether or not the piece has moved or not.

- Coupling: Low has no access to other classes and does not alter any data
- Cohesion: High each piece only serves as providers for the logic around that particular piece.

Overall Coupling & Cohesion

Coupling: The program has relatively low coupling given how each class is independent of each other with the exception of the Player class which needs read access to the Game class (this is required for AI to work). No two classes have access to each others internals.

Throughout the program all data is passed to each other through functions calls with data being passed in as parameters and results passed out of the function. Methods that dealt with moving pieces required more information being passed in/out used the Move class (passed the move class back and forth).

Cohesion: Each class has one specific responsibility, and no two classes have similar responsibilities thus this program has very high cohesion.

Design Patterns

The primarily implements two design patterns; the MVC and Template pattern.

MVC Design Pattern: The program uses an MVC design pattern, which is evident in the class designs.

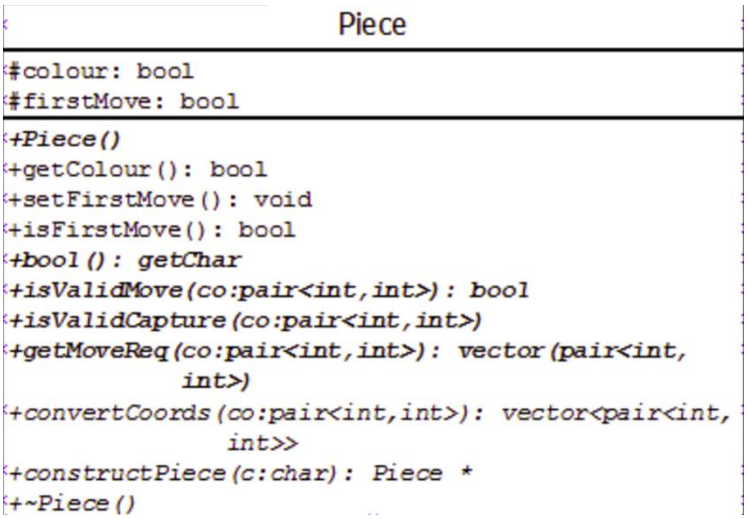
In an MVC design pattern there are three distinct layers: The Model, the View and the Controller. In this particular program each layer of the MVC design pattern is comprised of the following classes.

Model: the Player, Piece classes (and also the board field in Game) serve as the model in the program. These are classes that are manipulated by game and controller as have very little access and control over the rest of the program.

View: the View class is distinctly responsible for this layer, with all views (Input/Output) being handled by this class.

Controller: Game and Controller classes manipulate the game and directly control the legality of game moves, turn taking and match making. These two classes also do all of the calls to view in order to get Input/Output.

Template Pattern: the Piece class is implemented with a Template Design Pattern. The Piece class by itself is an abstract superclass and the Pawn, Bishop, Rook, Queen, King, Knight subclasses override some aspects of this superclass.



All pieces share the same methods, with the exception of getChar, isValidMove, getMoveReq and the constructor, which need to be different for each pieces behavior.

Flexibility of Program (Accommodation of Changes)

By nature of our design that prioritizes low coupling and high cohesion our program is very receptive to changes. It is very easy to see that each class within the program has a specific role within the program and governs its own subset of rule for chess (each class is mostly independent of each other as well), thus any specific changes can be made to the specific class to accommodate it.

To list out each role each class accomplishes:

Controller: Turn and Match Flow (back and forth between players and setting up games)

Piece: Rules for each pieces move set

Game: Controls the game state (with things like King Checks, Checkmate, Stalemate, etc...)

View: Handles all user interaction

Player: Handles all player decisions

Say for example you want to swap the rook and bishops behaviour (so Rook moves like a Bishop and vice versa), then you can simply change the subclasses in piece (specifically the isValidMove() and isValidCapture() methods) without it affecting other classes and still have it functioning.

Furthermore some classes within the program have the addition of extra features in mind. For example the Move class can be used in a variety of ways to add features to the game. A move log can be added by storing all instances of Move created and this can be used to undo moves.

Implementation Differences

Differences in Code: As we wrote the program, we noticed many gaps in our original design where we were missing methods, so most of our changes involved adding new methods. The largest additions were isThreatened, doesBoardPermit, and isValidCastle in Game, and the static constructPiece in Piece. Additionally, we had some gaps in our model which required the addition of new fields. Most of these new fields were a Boolean to represent colour. A colour field was added to Move so that we could verify that the move was made by the player controlling the piece, and to Piece to store the piece's colour.

Differences in Schedule: The program took longer than expected and we missed deadlines 1 and 2. For deadline 1 we finished on the 22nd instead of the 19th and we never completed all the requirements for deadline 2.

Known Bugs & Issues

Here is a list of bugs, undefined behaviors, and unimplemented features within this version of Chess:

- King Checking: has some undefined behavior with multiple pieces that can be blocked or captured when placing the king in check.
- Computer level 4 has yet to be completed
- En Passent has yet to be completed
- Graphical Display has yet to be completed
- A View class has been completed but a large portion of our program has yet to make use of it

Questions

1) Developing software in a team is has taught us to overcome issues with merging errors when multiple people are working on the same file, finding time to work with each other considering individual schedules and understanding each other's code. First, it is important to check if someone else is working on a given file or else you have problems merging files since multiple people made changes to the same file. Merging files more often than not causes errors in a program so merging files should always be a last resort. Another issue we had to work around is the differences in each person's schedule, not everyone can work on the project at a given time so we had to split up work then push our changes to a repository for the others to use. Lastly we learned that we need to write code that is formatted well so others can actually understand it. Documenting what each function does, and having consistent styling makes code easier for another group member to understand.

2) There are plenty of things we would have done differently programmatically and outside of the program. Within the program (the code) we definitely should have had more utility functions (perhaps contained in a class or library of some sort) for some common tasks such as case conversion of characters. Furthermore some of our condition checks within game should be implemented differently to make things more efficient (such as king checking with the use of size comparisons vs unit comparisons used in our program now). Outside of the program we definitely should have tried to finish earlier in order to have more time to fix bugs in the program, and perhaps should have approached this in a test development driven manner in order to discover and fix more bugs.