Chad

Team WareWolves

A Simple and Elegant game of Tactical and Strategic depth

Ben Goodwin, Joshua Munoz, Josiah May, Luis Rodriguez, Miles Wood

Refactoring and Design Patterns

Table of Contents

Refactoring

- Pull up method:
    - In the Model the concrete Pieces were originally all extensions of the abstract class Piece. I realized that that the Queen and Rook shared behaviour for checking along straight lines when trying to find valid moves, so I moved that logic up to an abstract class called LinePiece that both Rook and Queen extend.
    - Multiple JPanels used JTables that needed to be setup and accessed in a very similar way. I created an abstract class for all JPanel that contained a JTable and pulled up the repeated code into the base class.
- Replace magic constants
    - The Game logic had magic constants of the starting game board string in a few places. I refactored this to be a static constant of the Game class.
- Extract method/class
    - In many places throughout the code we found functionality that was being duplicated simply because of writing it and forgetting that we had the first version and extracted it to its own method in either the class that it was in initially or to a super or helper class.
    - For example, in the AiDriver class the logic for making a move was duplicated when receiving a move request and when checking for active games on startup. By extracting this logic to a function I was able to remove the redundancy, and while making the change I found a minor bug in the logic and was able to fix it.
    - As another example, throughout the Game logic there were methods for converting the String representations of out pieces and boards to Integers and back for use in array indexing. When we first changed the representation (from a number,letter pair to a letter,letter pair) instead of going through and changing it everywhere we extracted that logic to a class, Point, that would hold the data and be able to convert it back and forth for all of the classes that needed it. This meant that if we needed to change the logic around again nothing would have to change in the primary game logic.
    - Code climate identified multiple methods that were too long or had a high cognitive complexity. I extracted parts of the methods that a method could easily be defined as an action on the data.
    - The initial implementation of our precenter class and the swing gui controller had most of their logic in a switch statement. I extracted the logic into their own methods to increase readability.

Design Patterns

Model View Presenter

Reasoning

- Newer standard than MVC
- Network manager necessitated separating view and presenter
- We wanted to have the game and UI separate
    - The game logic did not relate to the UI logic which would lead to either classes that were doing more than they needed to or disorganization in the structure of our classes.

Implementation

- Communication
    - The Model communicates with a String representation of game boards and piece locations.
    - The server communication uses the same format.
    - Gameboard string is of the form ([RrKkQq][a-l][A-L])*
    - Move string is of the form ([RrKkQq][a-l][A-L][a-l][A-L])
- Model
    - Storage of active games
    - Logic for calculation of valid moves and moving pieces
- View
    - Command Line Interface
    - Graphical User Interface made with Java Swing
- Presenter
    - The presenter receives view messages from the View and receives network messages from the Network Manager.
    - The presenter does the necessary logic to determine which message was sent and do the logic to handle what each component needs.
        - This can include handling the model to change the representation of the game board when moves are made.
        - This also includes sending the necessary information to the View to handle different menus.

Client-Server Networking

Reasoning

- For the game system to ensure logins, invites, asynchronous game play, and saved game history necessitated the use of a web server to store player and game data.
- We wanted players to be able to play any other player across the world.

Implementation

- Separate packages for the Client and for the Server.
- Limited message passing to network messages that we defined ourselves for our system.
- Server
    - Java NIO package used for networking communication, especially Selector and SocketChannel.
    - Mysql-java-connector used to interact with the mysql database that holds all player, invite, and game data.
    - HashMap used to store active sessions of players.
- Client
    - Java IO Socket package used for networking communications
        - Used DataInputStream and DataOutputStream to transmit data.
    - NetworkManager of the client part of the presenter to pass necessary messages to the View and the Model.

Creational patterns

- Abstract Factory
    - In creating the GUI icons to represent the chess piece, I created a class that loaded and created the icon for each piece from the resource directory. When the I needed a new piece for the game board, I would call this class with the type of piece and its color, and the factory would return the piece with all the attributes I needed for the piece.

Structural patterns

- Observer for the Swing GUI
    - Swing uses extensive use of the Observer pattern. All user actions that control the gui rely on listeners. ActionListener read actions from buttons clicks and selections from lists, and MouseListeners controlled the selecting and movement of the chess pieces.

Behavioural patterns

- Template method
    - In the Game logic the Pieces use the template method for their valid move and move logic. The abstract Piece class defines most of the logic for making a move, but leaves each of its implementations to define what moves are valid, and then the Rook piece adds one more step onto the process when it checks for promotion.