CPSC 490: Project Report

Dalya Dickstein

Advisor: Ruzica Piskac

Building a Versatile, Abstract API for Creating Digital Board Games

1 Abstract

Board games have been a popular form of entertainment for generations. However, as the world becomes increasingly technology-oriented, so too become games. From game systems to computers, entertainment is moving increasingly on screens and online. This does not mean that board games are becoming obsolete; rather, there has been a surge of digital versions of board games that are available online or as applications. Most popular card games, and even some of the favorite board games such as Settlers of Catan, have been digitized and made into websites and apps. But there are thousands of beloved board games that are still waiting to be programmed and playable on the computer. However, programming board games from the ground up can be a daunting task. There are so many game components that must be written and given functionality before any of the actual game logic can be written at all. Programmers have to first build the logic behind boards, pieces, players, and everything else before they can even begin to think about the specifics of the game.

My project aims to help streamline this endeavor by providing an API for developers of board games to use. Many board games have similar or the same components: a board, players, and game pieces, for example. If these components can be abstracted away, while still being flexible enough for various types of usage, then developers can focus on writing game logic instead of writing the same code over and over again for the same core components. This means that in the end, it will be easier and faster for developers to bring more board games to our fit our new technology.

2 Challenges

There were two main challenges in my execution of this project that I contended with.

The Design Challenge

The tricky thing about writing an API for board games was balancing my code's generic abstraction with useful specifics. My goal for the project was to have an API that would be useful for not just one game or one category of board game, but for many different types of games. But every game has different gameplay, components, board structure, pieces, and capabilities. For example, one game's board, like with Tic Tac Toe, might only allow one player to occupy a spot, and players cannot move their marks. Other games might allow multiple players to be in the same spot, or even allow multiple pieces from multiple players to occupy it. I needed to decide how to represent a board and board slots, and provide functionality for all scenarios, such that it is still easy to use for both of these games. To this end, I often decided to provide various functions that might or might not be useful for a given game, to maximize my components' flexibility. I discuss the specifics of the code in the *Project Details* section (section 3).

At the end of the day, I decided to work out my major design questions by considering them from a user's point of view. I starting thinking of writing a game, and then thought about what functions I wanted to call. Then, I starting with coding a few components at a time, with small amounts of functionality. I would continue cycling through these two steps as I built up my API and provided more functions and went through multiple iterations of the code structure. Finally, I wrote some sample games as my own user, which helped me see how to best optimize my API for the convenience of a real user.

The Coding Challenge

The main challenge for me from a programming standpoint was implementing a networking layer. I did not have very much experience with cross-device connection and communication, but an integral part of playing board games is being able to play against a friend. I also had to figure out how to keep the networking abstract, so that the API user could write and receive messages between the server and the client as they wished throughout each player's turn. I wanted the actual network connection and

socket steps to be completely abstracted away from the turn logic of sending messages back and forth.

3 Project Details

The API includes the following components:

- Game State class to keep track of things like the board, bank, and players
- Core game components such as a bank, dice, game pieces, players, and board
- Networking layer with server and client classes to allow for 2-player connection by hostname across devices using sockets and ports
- Sample games to demonstrate how various API components can be used by board game developers

Below I outline some of the more complex or challenging pieces that were central to my design and coding process.

GameState.java

One of my major questions was how to keep track of the state of the game and the major components. When normally playing a board game, there are players, the board, the bank, dice, and other physical objects; there isn't really a "game state" component to see. The state is just there – it just exists. I debated at first whether I should have a class to monitor the game state at all, or whether it would be better to just let the users keep track of the components they create. In the end, I made a small GameState class with static variables and methods, so that they could be accessed at all times and the user wouldn't need to pass around tons of different components. The values of the GameState static variables (i.e. board, players list, dice, etc.) are set and augmented in object constructors whenever the user creates a new instance of those variables.

When I was coding my sample games and networking layer, I saw that having this type of GameState class was extremely helpful. I could access the list of players and key components from anywhere with the public getter methods in the GameState class, instead of passing them around everywhere. It ended up much like in a real board game: if the user wants to see the players, just look up.

GameBoard.java

Because so many board games allow for different types of boards, I struggled at first to decide how to make my board class detailed enough to be useful, but still generic enough to fit all sorts of games. A major part of this was handling how the board keeps track of the pieces. I chose to use a matrix (implemented as ArrayList<ArrayList<>> because of some type erasure issues during board initialization with a regular array matrix), with each cell containing an ArrayList of GamePieces. This way, any number of any player's pieces could fit onto any board. Games that don't want to allow for this could simply check whether the list in a given cell is empty or full. To compensate for this potential added overhead for the user, I added functions such as isEmpty to check if a cell is empty, in addition to a general getContents to get the entire list of pieces. While writing my Tic Tac Toe game, I also realized that it would be useful to just check if a player has a piece in a given cell, and so I added a playerOccupies function as well.

Networking Layer

Server.java: My API allows two players to connect easily using one player's hostname and a port to listen on. I wanted this part of the API to be especially easy to use, so that users would not have to struggle with all of the details of setting up the connection. To make this happen, I created a simple connect() function that handles all of the connection logic. The user writes a class that extends my ServerHandler interface, and provides implementations for the methods there. These methods include the server-side turn logic for both players' turns, as well as the win condition. Then, all the user has to do is instantiate a Server object in the user's main function, pass in a port number and their handler class instance, and call server.connect(). The connect function will create the socket and input/output channels to the client. Then, it will keep looping through players' turns until the endgame condition is met. Finally, it will call the handler's handleEndgame function (implementation provided by the user).

ServerHandler.java: This is an interface that ensures that the user provides implementations for the callbacks used by Server.java. createPlayers() is where the user instantiates the Player objects with constructor values for the Player's instance fields

(playerNum for an integer ID, a list of the player's game pieces, and the player's money). handleEndgame() is where the user can send final printouts and messages to the client to announce the winner of the game. takeServerTurn() provides the logic for when it is the player running the server's turn (so prints for example can be done with System.out instead of sending messages using the PrintWriter), and takeClientTurn() has the logic for sending and receiving messages to the player on the client side.

Client.java: While the server-side player needs to run `java myGame` and have the game code, the client-side player would only need the API and the Client class. To connect, the client player needs to run `java Client <Server's hostname> <port>`. The client reads the server's messages from the socket's input stream. I had some trouble at first because I wanted to allow users to send messages that expected user input, as well as those that did not. But with my initial simple loop of reading line and then awaiting client input, this didn't allow the server to just send non-response messages and have them print out on the client's end. My solution was to have messages that expect a response to begin with "QQ:" (which would be removed before being printed to the client). This way, the loop could check each incoming line and wait for the client's user input or not accordingly.

Turn handling

One of my initial struggles when trying to allow for multiplayer games was figuring out how I should handle turns, and what class the overall game should be run from. I wanted the user to be able to set up the game, write turn logic, and then essentially just run some "playGame" function. I started working on a couple small classes to this end. TurnController.java and BoardGame.java were both initial attempts to handle turns and provide this functionality. At the end of the day, I instead decided to handle turns within the Server.java class for games with networking, and the PlayGame.java class without. I still included TurnController.java and BoardGame.java in my submission because an important part of this project for me was figuring out how to design the most user-friendly API, and handling the game flow and turns was one of my biggest challenges design-wise.