CS246 Final Project Plan of Attack

Siddharth Gupta

Decemeber 2, 2020

1 Introduction

I am working on the 1-person project "Straights," therefore I will be taking the responsibility of completion for every aspect of this project. As you will see by reading this document, I hope to complete the project well in advance of the deadline with enough time to hopefully complete some extra features.

2 Project Breakdown and Timeline

Item	**Start Date**	**End Date**
Organize project file structure & make a private Git repo for version control	Dec 1	Dec 2
Write `main.cc` as a controller and write all header files	Dec 1	Dec 2
Complete implementation of the project as per the specifications	Dec 3	Dec 6
Complete thorough testing	Dec 7	Dec 7
Make fixes based on any errors found during testing	Dec 8	Dec 8
Finish working on extra features	Dec 9	Dec 10
Thoroughly test all code: basic & extra features	Dec 11	Dec 11
Implement fixes or roll back to previous stable version	Dec 12	Dec 12
Complete writing of demo plan, design document, and UML	Dec 13	Dec 14
Submit final code, demo plan, design document, and UML	Dec 15	Dec 15

Here, start dates represent the morning of the specified date, while the end date represents midnight. For example, "Complete thorough testing" is scheduled to take all of December 7^{th}. I expect and will try to match the above mentioned completion dates (or better).

3 Responses to Project Specification

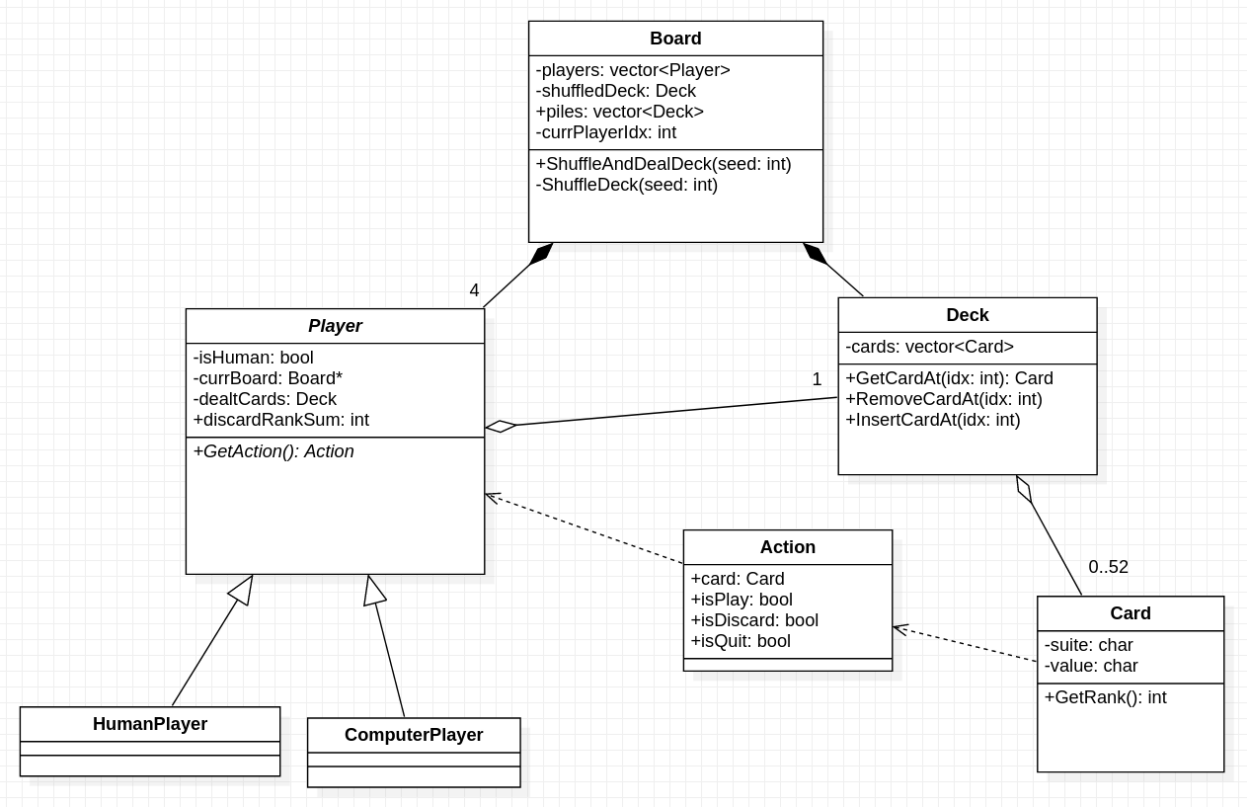

Figure 1: The anticipated UML diagram representing the project design

Question: What sort of class design or design pattern should you use to structure your game classes so that changing the interface or changing the game rules would have as little impact on the code as possible? Explain how your classes fit this framework.
Answer: Since this card game is relatively simple, the majority of game rule handling is being done in the `Board` class, `Player::GetAction` method, and somewhat in `main.cc`. Such a structure allows the ruleset to be broken into parts, handled by the class which it is most related to. For example, if we were to allow cards with rank 6 to be played without a neighbouring ranked card of the same suite (instead of 7), this can become a one-digit change in the implementation of `Player::GetAction`. Since this is expected to be a abstract class, such a change would have to be propagated down to the subclasses, something for which I expect to design a more elegant solution later in this process.

Question: If you want to allow computer players, in addition to human players, how might you structure your classes? Consider that different types of computer players might also have differing play strategies, and that strategies might change as the game progresses i.e. dynamically during the play of the game. How would that affect your structure?
Answer: Again, referring to the UML diagram, computer players are represented as child classes of a generic `Player` class with a virtual `GetAction` method. This method will be overridden in the implementation of the `ComputerPlayer` class. Since players (both computer and human) have access to the "board" through the `currBoard` attribute, they can adapt their logic (i.e. strategy) in the `Player::GetAction` method by simply reading the state of the board, allowing for dynamic playstyles during the play of the game.

Question: If a human player wanted to stop playing, but the other players wished to continue, it would be reasonable to replace them with a computer player. How might you structure your classes to allow an easy transfer of the information associated with the human player to the computer player?
Answer: In the case where a human player wanted to stop playing while keeping the game running (i.e. ragequit), this design allows for a relatively easy switch from a code design perspective. Since any (human) player's state is uniquely identified (relative to other players) by `dealtCards` and `discardRankSum`, that person's `HumanPlayer` object can be deleted and replaced with a `ComputerPlayer` object where the `dealtCards` and `discardRankSum` attributes of the `ComputerPlayer` set to original values (from the human who "ragequit"). Note, for this player to be "replaced," the change will have to be reflected in `Board::players`, which will eventually be accessible through `Player::currBoard`.

Note the UML diagram is subject to (and is likely to) change over the course of this project, however the basic design structure and organization is likely to be retained throughout.