University of California, Los Angeles

**Kalah Game Code Project Report**

Definition, Implementations, and test cases

William Escobar

CS32

Professor David Smallberg

May 30, 2023

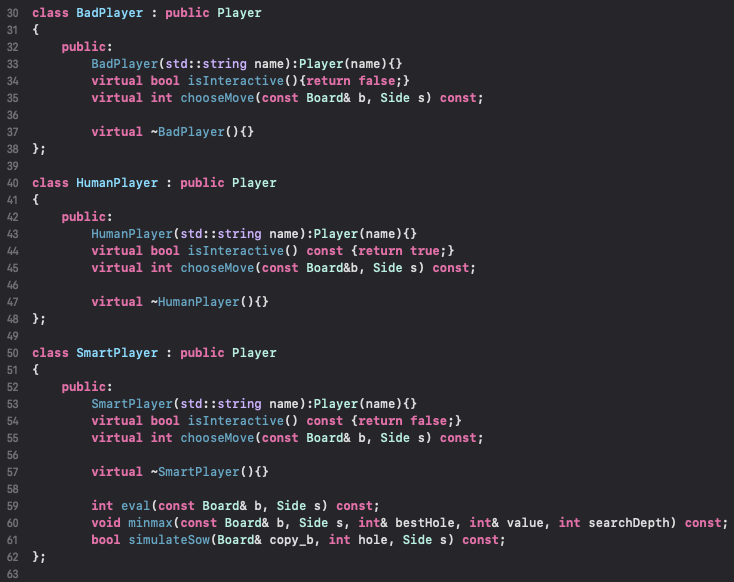
**Description and Design of classes:**

For this project, there are three main objects: **Game, Board, and Player**. The game object houses the board and players, while the board manages the players movements and scores. The players are only concerned with how they want to choose their moves and what their name is.

To better understand how the objects work together, its best to start the description from innermost to outermost object.

**----Player Objects----**

All the player objects are derived from the base player object. There are tree variations of these players: ***DumbPlayer, HumanPlayer, and SmartPlayer****.* Their definitions are shown below:



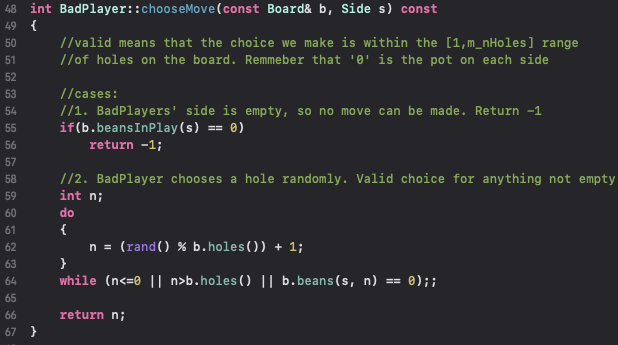
As can be seen, all the functions look very much similar where only SmartPlayer has some extended functionality using member functions eval(), minmax(), and simulateSow(); These functions will be discussed shortly.

These three derived classes maintain a very light load of member functions since their only concern is with making an integer choice of what move they want to make. The constructors and isInteractive() functions are pretty simple to understand, so I will solely focus on discussing the member function that does the brunt of the work.

***virtual int chooseMove(const Board& b, Side s) const{ . . .}***

When implementing the chooseMove() function for the classes, I wanted to keep the overridden virtual functions interface exactly the same. I didn’t want to added complexity. Whenever a function needs a move from the player, regardless of what type it is, they simply need to send the function the board the players are on and whose turn it is. These functions will perform whatever algorithm they employ and return their final desired choice.

**For BadPlayer:**

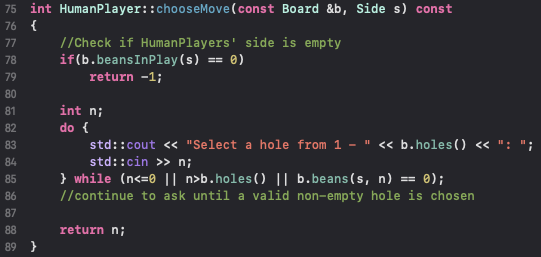


The algorithm employed for this derived class is very simple:

1. Check if the player can make a valid move(i.e. still has beans on its side)
2. Randomly choose a valid number and return it as its choice

Because, the class uses rand(), the choices that it makes are random, but the sequence of choices are the same for every game. Which is why this is a bad player.

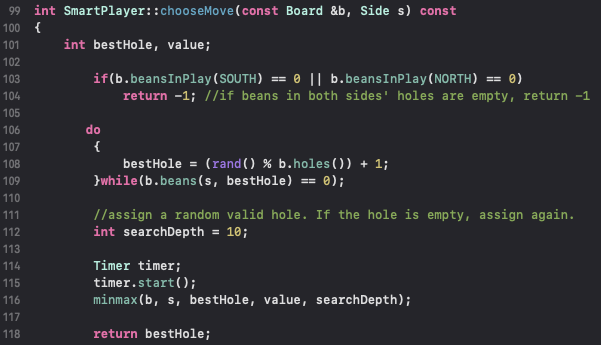
**For HumanPlayer:**



Similarly to the BadPlayer’s algorithm, the HumanPlayer’s code is also very simple.

1. Check if the player can make a valid move(i.e. still has beans on its side)
2. Continually read input choices from the console until the player enters a valid choice and then return it

**For SmartPlayer:**

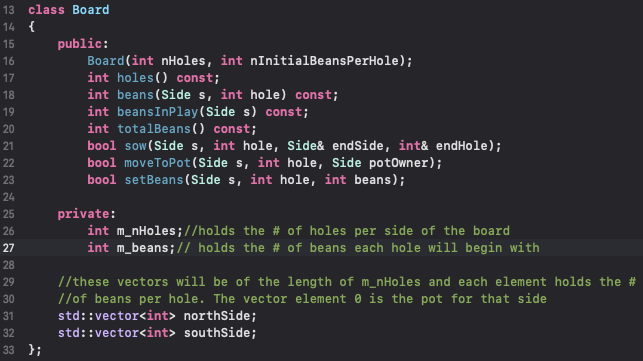
****

The algorithm for this function is much more complicated than that used for the other derived players. The algorithm for this function will be described in more detail later, but the short of it that chooseMove() uses the member functions eval(), minmax(), and simulateSow() to determine the best possible choice based on game move simulation. In the end, the chooseMove() function will have a final “bestHole” value that it will return which is just like the other derived classes.

**----Board Objects----**

The board is strictly responsible for performing a specific set of tasks:

1. Keep track of the score
2. Keep track of how many beans and holes are in the game
3. Perform the moves that were chosen by the players
4. And updating the board

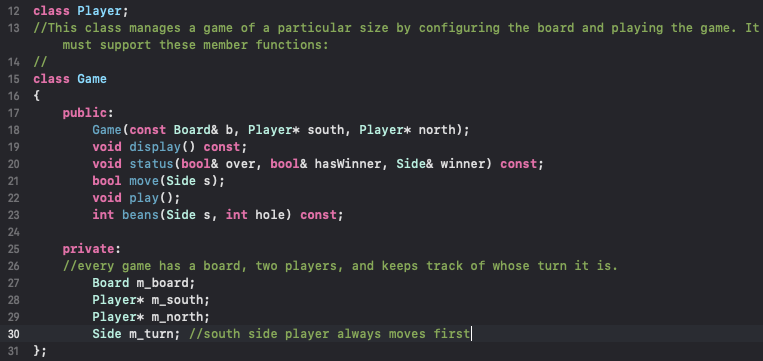


In order to keep track of the score, the board object utilized two vectors of integers (one for each side of the board). In addition, the class used two private variables that keep track of the number of holes the board has and how many beans per hole.

**----Game Objects----**

The game object is strictly responsible for running the code that allows the game to be played. This involves:

1. Displaying the current state of the board to the player
2. Requestion of the player objects to make a move
3. And keep track if there is a winner

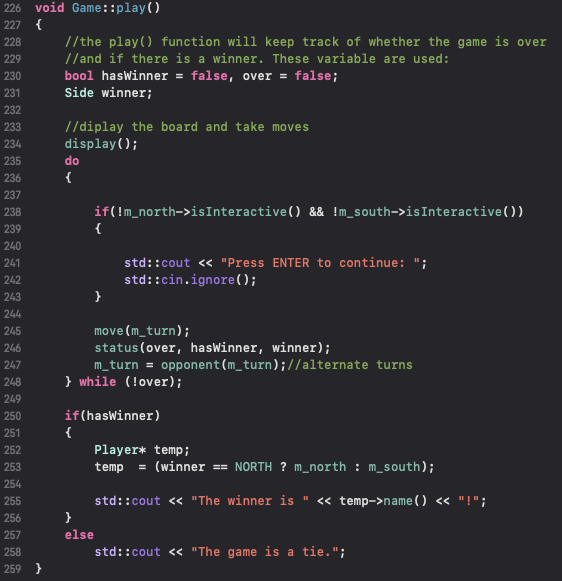


Every game has a board, two players, and turns to play. So, the class has several private member functions that allow it to perform the tasks that it needs to do. The class contains two pointers to the two players of the game, regardless of what type they are, so that it can call on their member functions to make a move. The game also has a board object which the game will call to perform the moves that the players requested. Finally, the class has a variable to keep track of whose turn it is so that the game knows which player to ask for a move. The most important member functions of this class are play(), and move(). So much so, that we discuss their code next . . .

**void Game::play():**

The logic behind this function is rather simple:

1. Display the board
2. Call the move() function to ask whosever’s turn its is to make a choice
3. Continue to do this until we have a winner
4. If we have a winner, announce the winner. Otherwise say it’s a tie



We include a conditional which pauses the screen if there are two AI players, so that the game doesn’t end instantly with very fast output.

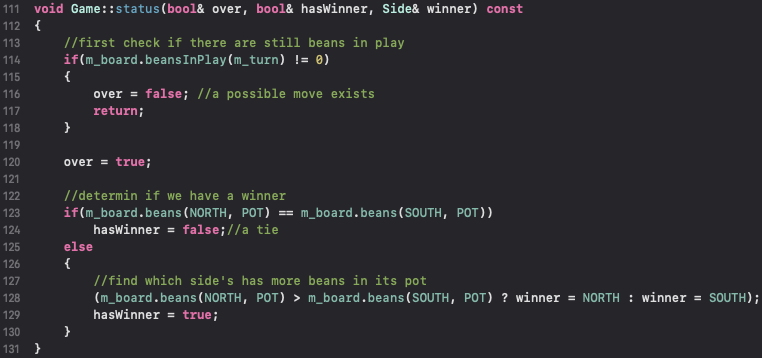
**bool Game::move(Side s):**

This function handles quite a bit of work and keeps track of certain rules for the game. Not matter whose turn it is, the function performs the following tasks:

1. Call the player’s chooseMove() function to return a move for the player
2. If no valid choice can be made, then the game is over. In which case, move all the remaining beans on the other opponents side into their pot.
3. If a move can be made, call on the game’s member function ‘board’ to play those moves
4. If the player gets another turn, repeat the previous process
5. If a capture is possible for the player, perform the capture



Throughout this process, the game keeps track of the games status by continually calling the game’s status() function after every move. This allows the game to determine whether or not it needs to declare a winner and stop asking for turns. That functions implementation follows:



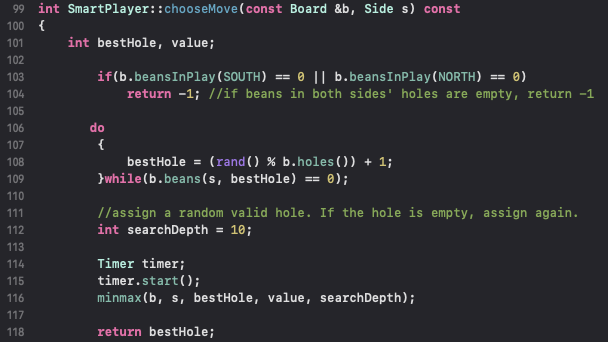
This code checks the amount of beans that are on the player’s (whose turn it is) side. If there are beans, then the player can make a move. Return false to signal this. Otherwise, the game is over (return true) and determine who the winner is by comparing the number of beans in each pot. Whoever has the most wins!

**More on the SmartPlayer’s algorithm:**

**int chooseMove():**

When designing the SmartPlayer’s algorithm, I wanted to have the chooseMove() function do the following:

1. Check to see if a move is possible, if not return -1
2. If a move is possible, call the minmax function which will handle finding the best move
3. Once the best value is found return it

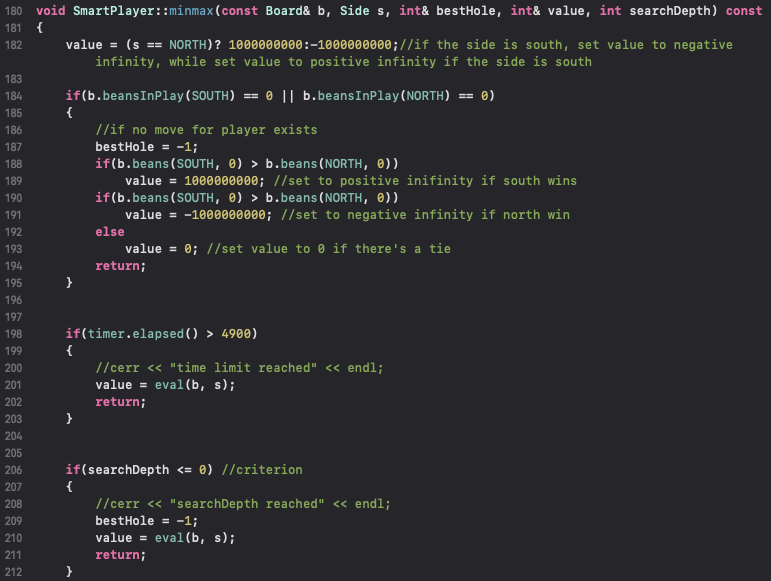


From the code, we can see that this is very much the case. I wanted to keep it in this form because it very much resembles the other chooseMove() functions of player objects. What changes is how the function goes about getting the best value.

To begin, we randomly choose a valid choice and save that in the “bestHole” variable. Then, we send that, along with the board, by reference to minmax. However, before we call minmax(), a predefined search depth and timer are initiated. These are used to keep the algorithm for running on too long. The timer is global, so that chooseove() and minmax() can both keep track of time. This could have been done by passing the timer by reference to avoid any risk of the timer begin reset, which could result in the algorithm running for a very long time. This is something that I would definitely change in the future. minmax() performs its magic and returns the best move, which is then returned by chooseMove() to the game’s move() function.

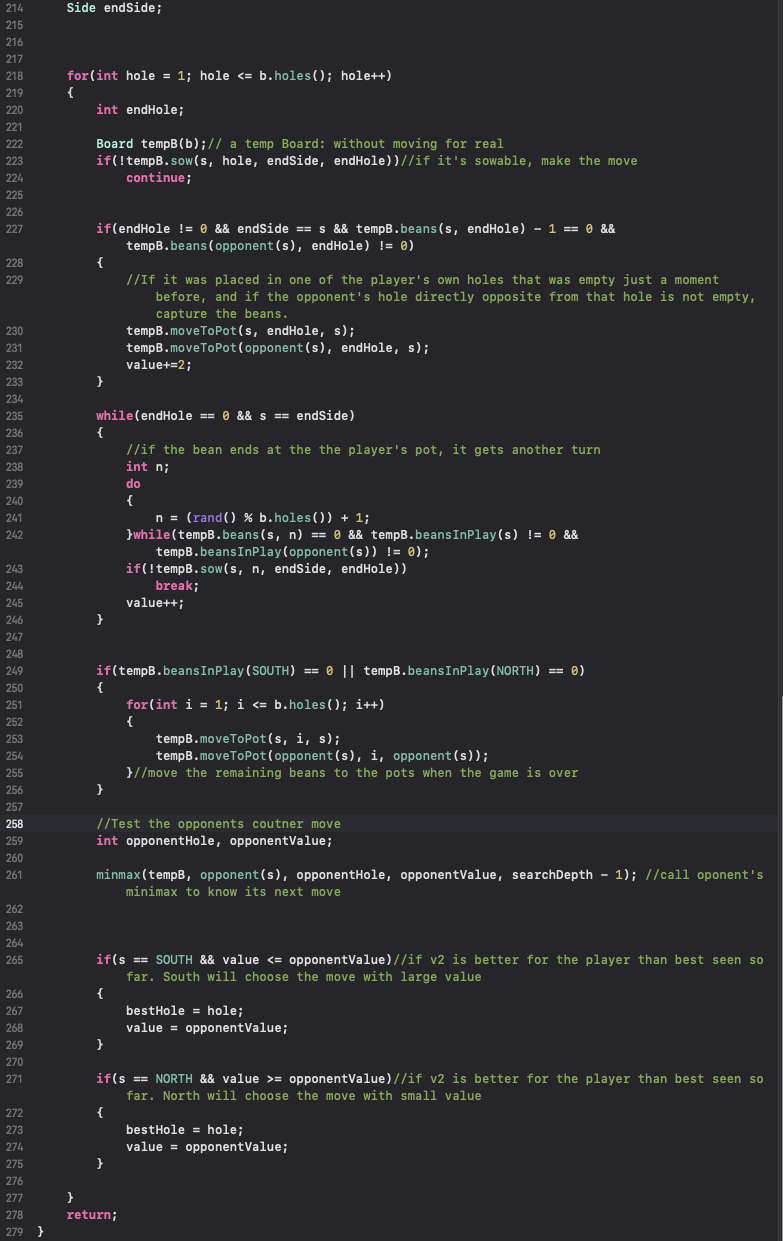
**void minmax():**

This function handles finding the best move by employing iterative testing and recursion. Part1 of the code is show below:



This section of code handles the base cases. If we hit the end of our simulation(i.e. no more moves are available), timer expired, or search depth was reached then terminate the recursion. At this “leaf”, we want to measure the value of the play and return that. This is done by testing the pots. If south player has more return +∞, if north player has more return -∞, otherwise return 0. The same is done for the timer and depth, only we use the eval() function which essentially does the same thing.

Part 2 of minmax() code:



The remainder of this code is aimed at preforming the arduous task of determining the best choice given every possible play on the current board.

* We start with a for-loop which iterates for as many possible plays that are on the current boar. We create a copy of the original board so that we don’t lose our game.
* Determine if the move is able to be made, otherwise continue to next iteration.
* If it was placed in one of the player's own holes which was empty just a moment before, and if the opponent's hole directly opposite from that hole is not empty, capture the beans.
* If player gets another turn, continue to make moves
* If we result in a game over, no more beans left, remove the remaining beans into the pots
* If game not over, then recursively call minmax() but now for the opponent
* Once this recursion comes back, test to see what the overall value of the plays were.
* Whichever play is better, save that to the “bestHole” variable and return(best move found)

Test cases:

#include "Game.h"

#include "Player.h"

#include <iostream>

#include <cassert>

**using** **namespace** std;

**void** doBoardTests()

{

Board b(3,2);

assert(b.holes() == 3 && b.totalBeans() == 12 &&

b.beans(SOUTH, 0) == 0 && b.beansInPlay(SOUTH) == 6);

b.setBeans(SOUTH, 1, 1);

b.moveToPot(SOUTH, 2, SOUTH);

assert(b.totalBeans() == 11 && b.beans(SOUTH, 1) == 1 &&

b.beans(SOUTH, 2) == 0 && b.beans(SOUTH, 0) == 2 &&

b.beansInPlay(SOUTH) == 3);

Side es;

**int** eh;

b.sow(SOUTH, 3, es, eh);

assert(es == NORTH && eh == 3 && b.beans(SOUTH, 3) == 0 &&

b.beans(NORTH, 3) == 3 && b.beans(SOUTH, 0) == 3 &&

b.beansInPlay(SOUTH) == 1 && b.beansInPlay(NORTH) == 7);

b.setBeans(NORTH, 0, 2);

b.setBeans(NORTH, 1, 3);

b.setBeans(NORTH, 2, 3);

b.setBeans(NORTH, 3, 0);

b.setBeans(SOUTH, 0, 1);

b.setBeans(SOUTH, 1, 0);

b.setBeans(SOUTH, 2, 3);

b.setBeans(SOUTH, 3, 0);

assert(!b.sow(SOUTH, 0, es, eh) && !b.sow(NORTH, 0, es, eh));//can't sow hole

assert(!b.sow(SOUTH, 1, es, eh) && !b.sow(SOUTH, 3, es, eh) && !b.sow(NORTH, 3, es, eh));//can't sow empty hole

assert(b.sow(SOUTH, 2, es, eh));

assert(es == NORTH && eh == 3);

}

**void** doPlayerTests()

{

HumanPlayer hp("Marge");

assert(hp.name() == "Marge" && hp.isInteractive());

BadPlayer bp("Homer");

assert(bp.name() == "Homer" && !bp.isInteractive());

SmartPlayer sp("Lisa");

assert(sp.name() == "Lisa" && !sp.isInteractive());

Board b(3,2);

b.setBeans(SOUTH, 2, 0);

cout << "=========" << endl;

**int** n ;//= hp.chooseMove(b, SOUTH);

cout << "=========" << endl;

//assert(n == 1 || n == 3);

//n = bp.chooseMove(b, SOUTH);

//assert(n == 1 || n == 3);

n = sp.chooseMove(b, SOUTH);

assert(n == 1 || n == 3);

b.setBeans(NORTH, 0, 2);

b.setBeans(NORTH, 1, 3);

b.setBeans(NORTH, 2, 3);

b.setBeans(NORTH, 3, 0);

b.setBeans(SOUTH, 0, 1);

b.setBeans(SOUTH, 1, 0);

b.setBeans(SOUTH, 2, 3);

b.setBeans(SOUTH, 3, 0);

n = sp.chooseMove(b, SOUTH);

assert(n == 2); //can only move legally

n = bp.chooseMove(b, SOUTH);

assert(n == 2); //can only move legally

b.setBeans(NORTH, 1, 0);

b.setBeans(NORTH, 2, 0);

b.setBeans(NORTH, 3, 0);

b.setBeans(SOUTH, 1, 0);

b.setBeans(SOUTH, 2, 0);

b.setBeans(SOUTH, 3, 0);

b.setBeans(NORTH, 0, 2);

b.setBeans(SOUTH, 0, 2);

n = sp.chooseMove(b, SOUTH);

assert(n == -1); //while can't move, return -1

n = bp.chooseMove(b, SOUTH);

assert(n == -1); //while can't move, return -1

}

**int** main()

{

doBoardTests();

doPlayerTests();

cout<< "Passed all tests";

}