**Design choices:**

**Board:**

Early on in my work for this project I realized that creating an internal class named Hole would be very useful for my Board. Holes can change the number of beans in them and hold many key pieces of information like their position, name, beans and side. My Board class contains two vectors of Hole pointers, one for the north holes and one for the south. The pots are special Hole’s with special names, and pointers that point to these holes are private data members of the board. During construction of a board, we make our dynamically allocated pots (as holes), insert them into element zero of our vectors, and then insert the dynamically allocated holes into both vectors. With this design, most of Board’s public functions call simple hole functions, making the code much simpler.

**Player:**

My inheritance structure for Player is very simple. There are three derived classes of Player, which is an abstract class. The is\_interactive function is implemented in Player and returns false, so HumanPlayer overrides it and returns true (the other two derived classes inherit Player’s implementation). A Player also has a protected member function called canMakeMove, so that each derived class can access it and see if there are possible moves they can make. My SmartPlayer has an evaluate function that gives a rating for the board, and has an overloaded chooseMove function that does most of the work for choosing a move.

**Game:**

My Game class contains a board, two pointers to players, a Boolean, and a capture function. The Boolean is set to false at construction, and only becomes true if the game should be over. My Game::play function stops making moves once the Boolean is true, and says who won the game (or it was a tie). My Game function uses Boards public interface to do a lot of the work, so I don’t need to add many extra private members of functions to Game. Game takes the responsibility of displaying a Kahla game, and does so simply. Any call to display (which will happen a bunch), outputs the north players name, their holes (1 3 4 5 means 1 bean in hole 1, 3 beans in hole 2, etc.), the pots, south’s holes, and south’s name. The game notes if people get another turn, and the capture function I added completes a capture for a certain side.

**Description of SmartPlayer::chooseMove:**

I took a nonconventional approach to SmartPlayer::chooseMove, so first I need to explain things I added to the SmartPlayer class. I added a private class inside of SmartPlayer called possibleMoves, which contains a hole number and its rating. I also have an evaluate function that takes a side and a board and returns a rating of how good the board is for that side. The rating is the number of beans in sides pot minus opponents pot, or a very high number if side is about to win, very low if side is about to lose, and zero for a tie. I have another private function called makeFakeMove, that “makes” a move on a temporary board, following the rules of getting new turns and captures. Now we can explore my implementation of chooseMove. Firstly, I check if a move is possible, if not returning -1. Then, I make a dynamically allocated array of possibleMoves with a size of the number of holes on a side(max number of possible moves). This uses possibleMoves default constructor, but we will never look at these default moves. We then call a function getRootPossibilities, that will change the first n elements of the array to the possiblemoves we can make from our board. The getRootPossibilities function does this by looping over all moves we can make, creating a temporary board and making them, evaluating the new board, and adding this possiblemove to the array (the array is passed into the function, and the size we consider the array to be gets incremented by one). So now we have, given a position, all the possible moves and their corresponding ratings in an array (these moves consider extra moves and captures, making this algorithm reasonably smart and a great offensive player). We know that there is at least one element in this array (otherwise we have returned -1 already) so we set move chosen to Root[0].hole() and our current max rating to Root[0].rating() (functions of the possibleMove class). We then loop over our “size” of the array (size will include only the actual elements, excluding the ones on the end that were default constructed), changing movechosen and max rating if we find a possibleMove with a better rating. We then return our movechosen.

**Psuedocode:**

**Board::sow**

If we cant sow, return false

Get number of beans to sow

Set beans in hole we are sowing from to 0

Set a hole\* at to where the first bean will be dropped

While we have no beans left

Drop a bean in where we are at and decrease how many we still have by 1

If no beans left

break

Increment at to where its supposed to be (bunch of special cases)

Set endSide and endHole to where at is pointing.

Return true

**HumanPlayer::chooseMove**

If we can make a move

Do

Ask and store input for the hole# player wants

While input is invalid

Return input

Return -1

**BadPlayer::chooseMove**

For all holes

If beans in hole > 0

Return hole #

Return -1

//Returns first hole that has at least 1 bean in it, or -1 if no moves possible

**SmartPlayer::chooseMove**

If we cant make a move

Return -1

Create dynamic array with max number of possible moves, containing possibleMove objects.

Initialize “size” of this array to 0

Call getRootPossibilities passing the board, side, dynamic array, “size” and actual size

Set movechosen and maxrating to the hole number and rating of the first element in array

Loop over remaining elements in “size” of the array

If an element has a greater rating then max rating

Set movechosen to this elements hole number

Set maxrating to this elements rating

Return movechosen

**SmartPlayer::makeFakeMove**

if no beans in the hole

return

create variable for end side and end hole

do a sow on the hole

if player deserves another turn

loop over the number of holes + 1 (i as our iterator)

if we are at holes + 1 (we can’t make a move after being rewarded another turn)

set i to garbage value

break;

if beans in hole (side, i) != 0

break;

if i is garbage value

return

recursively call makeFakeMove on the hole we found in the loop (hole i)

else if player deserves a capture

complete the capture

**SmartPlayer::getRootPossibilities**

Set a counter to 0

Loop over all holes in the side (i as iterator)

If beans in current hole is 0

Continue

Make a temporary board and make a fake move on that board

Set rating to the return value of evaluate(s, temporary board after the move)

Make a possibleMove object with hole# i and rating that we found

Add possiblemove to our dynamic array of possible moves

Increase counter and “size” of the array

**SmartPlayer::evaluate**

set mySide to side of player and otherSide to opponent

set rating to number of beans in mySide’s pot – beans in otherSide’s pot

if mySide can’t make a move (it is our turn so the game would be over right after this)

if mySide would win

set rating to 999999999

else if mySide would lose

set rating to -999999999

else

set rating to 0

return rating

**Game::move**

set movechosen to what Side s(argument of function)’s player chose in its chooseMove

if movechosen = -1

set isOver to true

dump all opponents beans into their pot

return false

sow the movechosen

if we get another turn

display the board

call move again on the same side

if we set up a capture

complete a capture

return true

**Game::play**

display the game state

while isOver is false

if both players are bots

cout whos turn it is and ask human to press enter to continue

call move for south

display game state

if isOver got set to true during that move

break out of while loop

if both players are bots

cout whos turn it is and ask human to press enter to continue

call move for north side

display game state

if theres no winner

cout that it’s a tie

cout who won

return

**Bugs/Problems:**

To my knowledge, my program does not have any bugs or incorrect implementations. The first problem I encountered was how to sow, as there are many special cases involved. This took a decent amount of thinking and writing down all the special cases (where my at pointer should jump to next), but I finally figured them out (having two sets of instructions for the two sides was necessary). However, the major problem I ran into was SmartPlayer::chooseMove, and I found this function very difficult. I tried many different methods and wrote down many potential ways of approaching this function, but still took a long time to get a working solution for this function. Testing this function was difficult as well, so I had a hard time progressing. Ultimately, I chose to take a different approach from the recursive pseudocode in the spec, as my implementation is reasonably smart and very high performance. Although we will only be tested on small boards, my SmartPlayer could return it’s moves almost instantly on large boards, which is an advantage over the recursive SmartPlayer::chooseMove implementation.

**Test cases:**

Board b(3, 2)

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

b.beans(SOUTH, POT) == 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, POT) == 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, POT) == 3 && b.beansInPlay(SOUTH) == 1 && b.beansInPlay(NORTH) == 7);

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);

BadPlayer bp1("Bart");

BadPlayer bp2("Homer");

Board b(3, 0);

b.setBeans(SOUTH, 1, 2);

b.setBeans(NORTH, 2, 1);

b.setBeans(NORTH, 3, 2);

Game g(b, &bp1, &bp2);

bool over;

bool hasWinner;

Side winner;

g.status(over, hasWinner, winner);

assert(!over && g.beans(NORTH, POT) == 0 && g.beans(SOUTH, POT) == 0 &&

g.beans(NORTH, 1) == 0 && g.beans(NORTH, 2) == 1 && g.beans(NORTH, 3) == 2 &&

g.beans(SOUTH, 1) == 2 && g.beans(SOUTH, 2) == 0 && g.beans(SOUTH, 3) == 0);

g.move(SOUTH);

g.status(over, hasWinner, winner);

assert(!over && g.beans(NORTH, POT) == 0 && g.beans(SOUTH, POT) == 3 &&

g.beans(NORTH, 1) == 0 && g.beans(NORTH, 2) == 1 && g.beans(NORTH, 3) == 0 &&

g.beans(SOUTH, 1) == 0 && g.beans(SOUTH, 2) == 1 && g.beans(SOUTH, 3) == 0);

g.move(NORTH);

g.status(over, hasWinner, winner);

assert(!over && g.beans(NORTH, POT) == 0 && g.beans(SOUTH, POT) == 3 &&

g.beans(NORTH, 1) == 1 && g.beans(NORTH, 2) == 0 && g.beans(NORTH, 3) == 0 &&

g.beans(SOUTH, 1) == 0 && g.beans(SOUTH, 2) == 1 && g.beans(SOUTH, 3) == 0);

g.move(SOUTH);

g.status(over, hasWinner, winner);

assert(!over && g.beans(NORTH, POT) == 0 && g.beans(SOUTH, POT) == 3 &&

g.beans(NORTH, 1) == 1 && g.beans(NORTH, 2) == 0 && g.beans(NORTH, 3) == 0 &&

g.beans(SOUTH, 1) == 0 && g.beans(SOUTH, 2) == 0 && g.beans(SOUTH, 3) == 1);

g.move(NORTH);

g.status(over, hasWinner, winner);

assert(over && g.beans(NORTH, POT) == 1 && g.beans(SOUTH, POT) == 4 &&

g.beans(NORTH, 1) == 0 && g.beans(NORTH, 2) == 0 && g.beans(NORTH, 3) == 0 &&

g.beans(SOUTH, 1) == 0 && g.beans(SOUTH, 2) == 0 && g.beans(SOUTH, 3) == 0);

assert(hasWinner && winner == SOUTH);

As I progressed through the project, I added additional test cases to test certain edge cases.

Once my game was finished and I had my finished product, I began playing games. I played games of two bots, different types of bots, one human one bot, two humans, switching the sides, etc. I also switched the board up to see if the game worked correctly for any board, and all of these tests worked as expected. These were not coded tests, perse, but very good indicators of if my Kahla game worked.