**Project 3 Report**

DESCRIPTION OF THE DESIGN OF THE CLASSES:

*class Board*

* Private data members:
  + two dynamically-allocated arrays of integers that represent the north and south sides of the board
  + an integer variable that keeps track of the number of holes on each side, including the pot (for example, if the north side had 3 holes and a pot, this variable would have a value of 4)
* destructor: deletes the dynamically allocated arrays
* the only other addition to the public interface is a copy constructor that creates new dynamically allocated arrays

class Player

* Private data members:
  + a string that records the name of the player

*class HumanPlayer*

* no additional private data members or private member functions

class BadPlayer

* no additional private data members or private member functions

class SmartPlayer

* no additional private data members
* Private member functions:
  + findBestMove is a helper function for chooseMove that picks the best hole, given a certain board configuration
  + evaluatePosition is a helper function for findBestMove that applies certain heuristics to numerically rate a particular board configuration

class Game

* private data members:
  + a player pointer that points to the north player
  + a player pointer that points to the south player
  + a board that is a copy of the board passed into the game constructor
  + a bool that keeps track of whose turn it is
* private member functions:
  + takeTurn is a helper function for move calls the chooseMove function of the player whose turn it is and sows the board based on the hole the player chooses. It also checks for and makes any captures and repeated turns.

DESCRIPTION OF THE DESIGN FOR SMARTPLAYER::CHOOSEMOVE

My SmartPlayer::ChooseMove method is composed of the ChooseMove function itself and two additional helper functions. The first helper function, called “findBestMove” takes as parameters an alarm clock, the player’s side, a board, a depth, a bestHole, and a value. The findBestMove function calls the second helper function, called “evaluate”, which takes a board as its only parameter. Therefore, the general structure is that evaluate is a helper function for findBestMove, which is a helper function for ChooseMove.

The purpose of the evaluate function is to apply the heuristics, or to return an integer that represents the current state of the board, where an extremely positive number indicates a win for north and an extremely negative number indicates a win for south. If the game is not over, the evaluate function returns the difference between the number of beans in north’s holes (including north’s pot) and the number of beans in south’s holes (including south’s pot). Lastly, if the north side has over half the total beans, it returns an extremely positive number, and if south side has over half the beans, it returns an extremely negative number.

The purpose of the findBestMove function is return the hole that SmartPlayer should pick, or -1 if there are no possible valid moves. The findBestMove function creates a game tree for each hole that it considers, and it decides on the most advantageous hole to choose by applying the heuristics in the aforementioned evaluate function. If the SmartPlayer is playing the south side, he always picks the hole that leads to the lowest value, and if SmartPlayer is playing the north side, he always picks the hole that leads to the highest possible value. Therefore, the SmartPlayer and its opponent always choose the holes and values that are most advantageous to themselves, no matter whether SmartPlayer is playing the south or north side of the board.

Lastly, the ChooseMove method is the most simple of the three functions because all it does is create an alarm clock and call the findBestMove function with an appropriate depth. It then returns the hole chosen by the findBestMove function.

Now that I have described the general outline of the ChooseMove function, the pseudocode for this method and its helper functions is shown below:

int SmartPlayer::chooseMove(const Board& b, Side s) const

create an alarm clock with 4900 milliseconds

call the findBestMove function with the alarm clock and a depth of 5

return the bestHole chosen by the findBestMove function

void SmartPlayer::findBestMove(AlarmClock& ac, Side playerSide, const Board& b, int depth, int& bestHole, int& value) const

if no move for the player exists (i.e. the game is over)

bestHole = -1

value = return value of the evaluate function for board b

return

if the alarm clock timed out or our depth is exausted

bestHole = -1

value = return value of the evaluate function for board b

return

for every hole h the player can choose

go to the next hole if the current one is empty

sow a copy of board b using hole h

for as many times as the player needs to take another turn

use findBestMove to choose a hole

sow the board using that hole

check for a capture

have the opponent choose his move using findBestMove

if playerSide is north

if the value chosen by the opponent is higher than any seen so far

bestHole = h

value = value chosen by opponent

if playerSide is south

if value chosen by the opponent is lower than any seen so far

bestHole = h

value = value chosen by opponent

return

int SmartPlayer::evaluatePosition(const Board& b) const

calculate difference between number of beans in north’s holes (including pot) minus those in south’s

if the game is over

if difference is greater than 0

return a really high positive number

if difference is less than 0

return a really low negative number

if difference equals 0

return 0

otherwise

return the difference

PSUDOCODE FOR NON-TRIVIAL ALGORITHMS

Board::Board(int nHoles, int nInitialBeansPerHole)

if nHoles is less than 1, set it to 1

if nInitialBeansPerHole is negative, set it to 0

record the number of holes plus pots in the Board’s m\_nHolesIncludingPot member variable

dynamically allocate an array of ints of size m\_nHolesIncludingPot for the Board’s m\_north and m\_south pointers

add nInitialBeansPerHole to every hole except the pots (hole 0)

set the number of beans in the pots to 0

int Board::beans(Side s, int hole) const

if hole is not valid

return -1

if s is south

return the number of beans at the element of m\_south at index hole

if s is north

return the number of beans at the element of m\_north at index hole

int Board::beansInPlay(Side s) const

initialize the result to 0

if s is south

for every element of the m\_south array except for the pot

add the beans at the element to result

if s is north

for every element of the m\_north array except for the pot

add the beans at the element to result

return result

int Board::totalBeans() const

initialize the result to 0

add beansInPlay(south) and beansInPlay(north) to result

add the beans in both pots to result

return result

bool Board::sow(Side s, int hole, Side& endSide, int& endHole)

{

if the hole is invalid or a pot

return false

if the hole is empty

return false

initialize a pointer temp to the starting hole

record the number of beans in the starting hole in a variable distributableBeans

set the beans in the starting hole to zero

while the number of distributableBeans is greater than 0

if temp points to the end of m\_south

point it to the start of m\_south

update endHole

if temp points to the start of m\_south

point it to the end of m\_north

update endHole and endSide

if temp points to the beginning of m\_north

point it to the second element of m\_south

update endSide and endHole

in all other cases

point temp to the next hole and update endHole

if temp points to the opponent’s pot

point it to the next hole

add a bean to temp’s hole and decrement the number of distributable beans

return true

bool Board::moveToPot(Side s, int hole, Side potOwner)

if the hole is invalid or a pot

return false

record the number of beans in hole on the side potOwner

set the number of beans in hole on the side potOwner to 0

set the number of beans in potOwner’s pot to the recorded number of beans

return true

bool Board::setBeans(Side s, int hole, int beans)

if the hole or number of beans is invalid

return false

set the number of beans at hole on side s to beans

return true

Board::Board(const Board& other)

set m\_nHolesIncludingPot to other’s equivalent member variable

create dynamically allocated arrays of size m\_nHolesIncludingPot for both m\_south and m\_north

copy the values from other’s m\_south and m\_north to this’s m\_south and m\_north

Board::~Board()

delete m\_north and m\_south

int HumanPlayer::chooseMove(const Board& b, Side s) const

if every hole on side s is empty

return -1

while the chosen hole is invalid or empty

cout “Select a hole, “ followed by the the name of the user

if the hole is invalid

cout “The hole number must be from 1 to " followed by the number of holes

if the hole is empty

cout "There are no beans in that hole."

return the selected hole

int BadPlayer::chooseMove(const Board& b, Side s) const

initialize result to -1

for every hole

if the hole has beans

set result to the hole

break

return result

(All of the SmartPlayer pseudocode is given in the section “Description of the Design for SmartPlayer::chooseMove”)

Game::Game(const Board& b, Player\* south, Player\* north)

initialize m\_board as a copy of b

set m\_southTurn to true

set m\_southPlayer to south and m\_northPlayer to north

void Game::display() const

print the north player’s name, preceded by spaces

print the the number of beans in each hole in the north, separated by spaces

print the pots, separated by a number of spaces equivalent to the number of holes times 4

print the south player’s name, preceded by spaces

void Game::status(bool& over, bool& hasWinner, Side& winner) const

if the game isn’t over

set over to false

return

if the game is over

set over and hasWinner to true

if the number of beans on the south side (including pot) is greater than the number of beans on the north side

set winner to south

if the number of beans on the north side is greater than the number on the south side

set winner to north

otherwise

set hasWinner to false

return

bool Game::move()

set result to the return value of takeTurn()

if the game is over

if there are still beans in play in the south

cout “Sweeping remaining beans into " south player name "'s pot."

if there are still beans in play in the north

cout “Sweeping remaining beans into " north player name "'s pot."

if there are still beans in play

use moveToPot to move all of the beans into the proper pot

display()

if result is false

return false

update m\_southTurn to change whose turn it is

return true

bool Game::takeTurn()

if the game is over

return false

call the chooseMove function of the player whose turn it is

cout name of the player moving, followed by " chooses hole ", followed by the return value of chooseMove

sow m\_board with the hole the player chose

display()

if the player gets another turn

cout name of the player moving, followed by " gets another turn."

recursive call to takeTurn()

if the player achieved a capture

use moveToPot to move the necessary beans to the player’s pot

return true

void Game::play()

display()

while the game is not over

move()

if neither player is interactive

prompt the user to press “enter” key to continue

if neither player is interactive

prompt the user to press “enter” key to continue

call the status function

if the game is over

if there is a winner

if the winner is north

cout "The winner is " followed by north player’s name

if the winner is south

cout "The winner is " followed by south player’s name

otherwise

cout “The game ended in a tie.”

BUGS, INEFFICIENCIES, AND NOTABLE PROBLEMS

To my understanding or recognition, there are no bugs or serious inefficiencies with my program. The most notable problems I faced were associated with the SmartPlayer::ChooseMove function, specifically the part of it where the player must taken a second turn. While debugging, I find that this aspect created an infinite loop, though it has been fixed since then. As of when I turned in this project, I have found no remaining bugs in my code.

TEST CASES TO THOROUGHLY TEST MY FUNCTIONS

**Tests for the Board class:**

// TEST THE BOARD'S CONSTRUCTOR, HOLES METHOD, AND BEANS METHOD

Board b1(-1, 2); // act as if nHoles were 1 if it is negative

assert(b1.holes() == 1);

Board b2(0, 2); // if nHoles is 0, act as if it were 1

assert(b2.holes() == 1);

Board b3(2, -2); // if nInitialBeansPerHole is negative, act as if it were 0

for (int i = 0; i < b3.holes() + 1; i++)

{

assert(b3.beans(SOUTH, i) == 0); // every hole should have 0 beans

assert(b3.beans(NORTH, i) == 0);

}

Board b4(4, 3); // test a typical case

assert(b4.holes() == 4); // b4 should have 4 holes

assert(b4.beans(SOUTH, 0) == 0); // Both pots should start with 0 beans

assert(b4.beans(NORTH, 0) == 0);

for (int i = 1; i < b4.holes() + 1; i++)

{

assert(b4.beans(SOUTH, i) == 3); // every non-pot hole should have 3 beans

assert(b4.beans(NORTH, i) == 3);

}

assert(b4.beans(SOUTH, -5) == -1); // return -1 for an invalid hole number

assert(b4.beans(NORTH, -5) == -1);

assert(b4.beans(SOUTH, 5) == -1);

assert(b4.beans(NORTH, 5) == -1);

// TEST THE BOARD'S BEANSINPLAY METHOD

Board b5(5, 5);

assert(b5.beansInPlay(SOUTH) == 25); // testing a typical case when there's more than 0 beans in play

assert(b5.beansInPlay(NORTH) == 25);

Board b6(5, 0);

assert(b6.beansInPlay(SOUTH) == 0); // both sides should have 0 beans

assert(b6.beansInPlay(NORTH) == 0);

// TEST THE BOARD'S TOTALBEANS METHOD

Board b7(4,2); // a typical case when there are more than 0 beans in the game

assert(b7.totalBeans() == 16);

Board b8(10, 0); // an edge case when there are 0 beans in the game

assert(b8.totalBeans() == 0);

// TEST THE BOARD'S SOW METHOD

Side endSide = SOUTH;

int endHole = -1;

Board b10(4,0);

Board b9(4,4);

assert(b10.sow(SOUTH, 2, endSide, endHole) == false); // returns false for empty holes

assert(b9.sow(SOUTH, -1, endSide, endHole) == false); // invalid holes return false without changing anything

assert(b9.sow(NORTH, 0, endSide, endHole) == false); // either pot returns false

assert(b9.sow(SOUTH, 0, endSide, endHole) == false);

assert(b9.beans(SOUTH, 0) == 0 && b9.beans(NORTH, 0) == 0); // make sure nothing was changed by the previous invalid calls

for (int i = 1; i < b9.holes()+1; i++) {

assert(b9.beans(SOUTH, i) == 4);

assert(b9.beans(NORTH, i) == 4);

}

assert(endHole == -1);

assert(endSide == SOUTH);

Board b11(2,2);

assert(b11.sow(SOUTH, 2, endSide, endHole)); // test a typical non-empty hole

assert(b11.beans(SOUTH, 1) == 2 && b11.beans(SOUTH, 2) == 0 && b11.beans(SOUTH, 0) == 1 && b11.beans(NORTH, 2) == 3 && b11.beans(NORTH, 0) == 0 && b11.beans(NORTH, 1) == 2 && endSide == NORTH && endHole == 2); // holes that are sown are increemented, those not sown are unchanged

assert(b11.sow(SOUTH, 2, endSide, endHole) == false); // returns false for empty holes

Board b12(2,2);

assert(b12.sow(NORTH, 2, endSide, endHole)); // test the case when the last bean is placed in North's pot

assert(b12.beans(SOUTH, 1) == 2 && b12.beans(SOUTH, 2) == 2 && b12.beans(SOUTH, 0) == 0 && b12.beans(NORTH, 2) == 0 && b12.beans(NORTH, 0) == 1 && b12.beans(NORTH, 1) == 3 && endSide == NORTH && endHole == 0);

Board b13(2,2);

assert(b13.sow(SOUTH, 1, endSide, endHole)); // test the case when the last bean is placed in South's pot

assert(b13.beans(SOUTH, 1) == 0 && b13.beans(SOUTH, 2) == 3 && b13.beans(SOUTH, 0) == 1 && b13.beans(NORTH, 2) == 2 && b13.beans(NORTH, 1) == 2 && b13.beans(NORTH, 0) == 0 && endSide == SOUTH && endHole == 0);

Board b14(2,2);

assert(b14.sow(NORTH, 1, endSide, endHole)); // test the case when the board is sown from the north to the south side

assert(b14.beans(SOUTH, 1) == 3 && b14.beans(SOUTH, 2) == 2 && b14.beans(SOUTH, 0) == 0 && b14.beans(NORTH, 2) == 2 && b14.beans(NORTH, 1) == 0 && b14.beans(NORTH, 0) == 1 && endSide == SOUTH && endHole == 1);

Board b15(2,6);

assert(b15.sow(SOUTH, 1, endSide, endHole)); // test when the board is sown from south all the way back to the starting position

assert(b15.beans(SOUTH, 1) == 1 && b15.beans(SOUTH, 2) == 8 && b15.beans(SOUTH, 0) == 1 && b15.beans(NORTH, 2) == 7 && b15.beans(NORTH, 1) == 7 && b15.beans(NORTH, 0) == 0 && endSide == SOUTH && endHole == 2);

Board b16(2,5);

assert(b16.sow(NORTH, 1, endSide, endHole)); // test when the board is sown from north all the way back to the starting position

assert(b16.beans(SOUTH, 1) == 6 && b16.beans(SOUTH, 2) == 6 && b16.beans(SOUTH, 0) == 0 && b16.beans(NORTH, 2) == 6 && b16.beans(NORTH, 1) == 1 && b16.beans(NORTH, 0) == 1 && endSide == NORTH && endHole == 1);

Board b17(2,10);

assert(b17.sow(SOUTH, 1, endSide, endHole)); // test when the board is sown all the way around the board twice

assert(b17.beans(SOUTH, 1) == 2 && b17.beans(SOUTH, 2) == 12 && b17.beans(SOUTH, 0) == 2 && b17.beans(NORTH, 2) == 12 && b17.beans(NORTH, 1) == 12 && b17.beans(NORTH, 0) == 0 && endSide == SOUTH && endHole == 1);

// TESTING THE BOARD'S MOVEPOT METHOD

Board b18(2,2);

assert(!b18.moveToPot(SOUTH, -1, SOUTH)); // return false for invalid hole

assert(b18.beans(SOUTH, 1) == 2 && b18.beans(SOUTH, 2) == 2 && b18.beans(SOUTH, 0) == 0 && b18.beans(NORTH, 2) == 2 && b18.beans(NORTH, 1) == 2 && b18.beans(NORTH, 0) == 0); // board remains unchanged

assert(!b18.moveToPot(SOUTH, 3, NORTH)); // return false for invalid hole

assert(b18.beans(SOUTH, 1) == 2 && b18.beans(SOUTH, 2) == 2 && b18.beans(SOUTH, 0) == 0 && b18.beans(NORTH, 2) == 2 && b18.beans(NORTH, 1) == 2 && b18.beans(NORTH, 0) == 0); // board remains unchanged

assert(!b18.moveToPot(SOUTH, 0, SOUTH)); // return false for a pot

assert(b18.beans(SOUTH, 1) == 2 && b18.beans(SOUTH, 2) == 2 && b18.beans(SOUTH, 0) == 0 && b18.beans(NORTH, 2) == 2 && b18.beans(NORTH, 1) == 2 && b18.beans(NORTH, 0) == 0); // board remains unchanged

assert(b18.moveToPot(SOUTH, 2, SOUTH)); // test the typical case for a non-empty south hole transferred to the south pot

assert(b18.beans(SOUTH, 1) == 2 && b18.beans(SOUTH, 2) == 0 && b18.beans(SOUTH, 0) == 2 && b18.beans(NORTH, 2) == 2 && b18.beans(NORTH, 1) == 2 && b18.beans(NORTH, 0) == 0); // beans are properly transferred

Board b19(2,2);

assert(b19.moveToPot(SOUTH, 2, NORTH)); // test the typical case for a non-empty south hole transffered to the north pot

assert(b19.beans(SOUTH, 1) == 2 && b19.beans(SOUTH, 2) == 0 && b19.beans(SOUTH, 0) == 0 && b19.beans(NORTH, 2) == 2 && b19.beans(NORTH, 1) == 2 && b19.beans(NORTH, 0) == 2); // beans are properly transferred

Board b20(2,2);

assert(b20.moveToPot(NORTH, 2, NORTH)); // test the typical case for a non-empty north hole transffered to the north pot

assert(b20.beans(SOUTH, 1) == 2 && b20.beans(SOUTH, 2) == 2 && b20.beans(SOUTH, 0) == 0 && b20.beans(NORTH, 2) == 0 && b20.beans(NORTH, 1) == 2 && b20.beans(NORTH, 0) == 2); // beans are properly transferred

assert(b20.moveToPot(NORTH, 1, SOUTH)); // test the typical case for a non-empty north hole transffered to the south pot

assert(b20.beans(SOUTH, 1) == 2 && b20.beans(SOUTH, 2) == 2 && b20.beans(SOUTH, 0) == 2 && b20.beans(NORTH, 2) == 0 && b20.beans(NORTH, 1) == 0 && b20.beans(NORTH, 0) == 2); // beans are properly transferred

// TEST THE SETBEANS METHOD

Board b21(2,2);

assert(!b21.setBeans(SOUTH, -1, 5)); // returns false for invalid holes

assert(!b21.setBeans(SOUTH, 3, 5));

assert(!b21.setBeans(SOUTH, 1, -1)); // returns false for invalid numbers of beans

assert(b21.setBeans(NORTH, 2, 0)); // returns true for valid paramters

assert(b21.beans(NORTH, 2) == 0); // properly alters the number of beans in a hole

assert(b21.beansInPlay(NORTH) == 2); // the number of beans in play updates correctly

assert(b21.beansInPlay(SOUTH) == 4); // doesn't affect the other side

// TEST THE BOARD'S COPY CONSTRUCTOR

Board b24(4,3);

Board b25 = b24;

assert(b25.holes() == 4 && b25.beansInPlay(SOUTH) == 12 && b25.beansInPlay(NORTH) == 12 && b25.totalBeans() == 24); // copied boards have the same number of holes and beans

assert(b25.beans(SOUTH, 0) == 0 && b25.beans(SOUTH, 1) == 3 && b25.beans(SOUTH, 2) == 3 && b25.beans(SOUTH, 3) == 3 && b25.beans(SOUTH, 4) == 3 && b25.beans(SOUTH, 5) == -1 && b25.beans(NORTH, 0) == 0 && b25.beans(NORTH, 1) == 3 && b25.beans(NORTH, 2) == 3 && b25.beans(NORTH, 3) == 3 && b25.beans(NORTH, 4) == 3 && b25.beans(NORTH, 5) == -1); // each hole has the same number of beans as the original

assert(b24.setBeans(SOUTH, 0, 11) && b24.setBeans(NORTH, 0, 8));

Board b26 = b24;

assert(b26.beans(SOUTH, 0) == 11 && b26.beans(NORTH, 0) == 8); // copies over beans in both pots

b26.setBeans(SOUTH, 3, 10);

assert(b26.beans(SOUTH, 3) == 10 && b25.beans(SOUTH, 3) == 3); // changes to the original board do not affect the copy

**Tests for the Player class:**

// TEST PLAYER'S METHODS

BadPlayer bp2("ricky");

assert(bp2.name() == "ricky"); // Player's constructor properly records the name, and the name() method correctly returns it

assert(bp2.isInteractive() == false); // BadPlayer's isInteractive method should always return false

// TEST BADPLAYER'S CHOOSEMOVE FUNCTION

Board b22(2,2);

BadPlayer bp1("bp1");

assert(bp1.chooseMove(b22, SOUTH) == 1); // chooses the leftmost non-empty, non-pot hole

b22.setBeans(SOUTH, 1, 0);

assert(bp1.chooseMove(b22, SOUTH) == 2); // chooses the leftmost non-empty, non-pot hole

assert(bp1.chooseMove(b22, NORTH) == 1); // correctly shooses the leftmost hole on the north side

b22.setBeans(NORTH, 1, 0);

assert(bp1.chooseMove(b22, NORTH) == 2);

b22.setBeans(SOUTH, 1, 0);

b22.setBeans(SOUTH, 2, 0);

assert(bp1.chooseMove(b22, NORTH) == 2); // changes to the opposite side don't affect BadPlayer's decisions

b22.setBeans(NORTH, 2, 0);

assert(bp1.chooseMove(b22, NORTH) == -1); // returns -1 if all holes are empty on its designated playing side

// TEST HUMANPLAYER'S CHOOSEMOVE FUNCTION

Board b23(2,0);

HumanPlayer hp1("human");

assert(hp1.chooseMove(b23, SOUTH) == -1); // return -1 if all holes are empty

b23.setBeans(NORTH, 1, 5);

b23.setBeans(NORTH, 2, 5);

assert(hp1.chooseMove(b23, SOUTH) == -1); // return -1 if all holes are empty on SOUTH side

b23.setBeans(SOUTH, 1, 2);

// cout << hp1.chooseMove(b23, SOUTH) << endl; // keeps reprompting user unless a valid, non-empty hole is inputed, and it returns the valid input hole

// TEST THE SMARTPLAYER METHOD

SmartPlayer sm1("first");

Board b43(2,1);

assert(sm1.chooseMove(b43, SOUTH) == 2); // chooses the only possible move to secure a win

Board b44(2,1); b44.setBeans(NORTH, 2, 2);

cout<< sm1.chooseMove(b44, SOUTH) << endl; // chooses a capture over a loss

Board b46(2,1); b46.setBeans(NORTH, 1, 0);

assert(sm1.chooseMove(b46, SOUTH) == 2); // chooses a second turn over a loss

Board b48(3,0); b48.setBeans(SOUTH, 1, 1); b48.setBeans(SOUTH, 2, 1); b48.setBeans(NORTH, 2, 1);

assert(sm1.chooseMove(b48, SOUTH) == 1);

SmartPlayer sp12("smart");

BadPlayer bp3("bad");

bool over, hasWinner;

Side winner;

Board b67(4, 0); b67.setBeans(NORTH, 2, 1); b67.setBeans(SOUTH, 1, 1); b67.setBeans(SOUTH, 3, 1);

// 0 1 0 0

// 0 0

// 1 0 1 0

assert(sp12.chooseMove(b67, SOUTH) == 1); // smartplayer chooses a capture to win the game

Board b68(6,4);

// 4 4 4 4 4 4

//0 0

// 4 4 4 4 4 4

assert(sp12.chooseMove(b68, NORTH) == 4); // smartplayer chooses to get a second turn

Board b71(6,0); b71.setBeans(NORTH, 2, 1); b71.setBeans(SOUTH, 4, 2); b71.setBeans(NORTH, 6, 6);

// 0 1 0 0 0 6

//0 0

// 0 0 0 2 0 0

assert(sp12.chooseMove(b71, NORTH) == 6); // smartplayer predicts and avoids a capture by the opponent

Board b69(6,4);

Game g13(b69, &bp3, &sp12);

while (g13.move()) {}

g13.status(over, hasWinner, winner);

assert(over && hasWinner && winner == NORTH); // when smartplayer is north, it beats the badplayer

Board b70(6,4);

Game g14(b70, &sp12, &bp3);

while (g14.move()) {}

g14.status(over, hasWinner, winner);

assert(over && hasWinner && winner == SOUTH); // when smartplayer is south, it beats the badplayer

**Tests for the Game class**:

// TEST THE GAME'S CONSTRUCTOR AND BEANS METHOD

Board b27(2,3);

BadPlayer\* bp3 = new BadPlayer("Ben");

BadPlayer\* bp4 = new BadPlayer("Pedro");

Game g1(b27, bp3, bp4);

assert(g1.beans(SOUTH, 0) == 0 && g1.beans(SOUTH, 1) == 3 && g1.beans(SOUTH, 2) == 3 && g1.beans(SOUTH, 3) == -1 && g1.beans(NORTH, 0) == 0 && g1.beans(NORTH, 1) == 3 && g1.beans(NORTH, 2) == 3); // game objects have a copy of the board used to construct them, and the beans method returns the number of beans in each hole

assert(g1.beans(NORTH, 3) == -1 && g1.beans(SOUTH, 3) == -1 && g1.beans(NORTH, -1) == -1 && g1.beans(SOUTH, -1) == -1); // beans method returns -1 for invalid holes

g1.display(); // allows you to visually check that the board is properly displayed

delete bp3;

delete bp4;

// TEST THE GAME'S STATUS METHOD

Board b28(3,0);

BadPlayer bp5("bad"), bp6("worse");

Game g2(b28, &bp5, &bp6);

bool over(false), hasWinner(false);

Side winner = SOUTH;

// test the case when there are no beans left (the game is over), but it is a tie

g2.status(over, hasWinner, winner);

assert(over == true && hasWinner == false && winner == SOUTH); // "winner" is unchanged

// test when only one side has beans (the game is over), but it is a tie

b28.setBeans(SOUTH, 2, 4);

b28.setBeans(NORTH, 0, 4);

Game g3(b28, &bp5, &bp6);

g3.status(over, hasWinner, winner);

assert(over == true && hasWinner == false && winner == SOUTH); // "winner" remains unchanged

// test when both sides have no beans (the game is over), and south won

b28.setBeans(SOUTH, 2, 0);

b28.setBeans(SOUTH, 0, 10);

b28.setBeans(NORTH, 0, 5);

Game g4(b28, &bp5, &bp6);

g4.status(over, hasWinner, winner);

assert(over == true && hasWinner == true && winner == SOUTH);

// test when both sides have no beans (the game is over), and north won

b28.setBeans(NORTH, 0, 20);

Game g5(b28, &bp5, &bp6);

g5.status(over, hasWinner, winner);

assert(over == true && hasWinner == true && winner == NORTH);

// test when there are beans remaining only on the south side, with none in the pots

Board b30(2,3);

b30.setBeans(NORTH, 1, 0); b30.setBeans(NORTH, 2, 0);

Game g8(b30, &bp5, &bp6);

g8.status(over, hasWinner, winner);

assert(over == true && hasWinner == true && winner == SOUTH);

// test when there are beans remaining only on the north side, with none in the pots

b30.setBeans(NORTH, 1, 2); b30.setBeans(SOUTH, 1, 0); b30.setBeans(SOUTH, 2, 0);

Game g9(b30, &bp5, &bp6);

g9.status(over, hasWinner, winner);

assert(over == true && hasWinner == true && winner == NORTH);

// test when there are beans remaining only on one side, but it's still a tie

b30.setBeans(SOUTH, 0, 2);

Game g10(b30, &bp5, &bp6);

g10.status(over, hasWinner, winner);

assert(over == true && hasWinner == false && winner == NORTH); // "winner" remains unchanged

// TEST THE GAME'S MOVE FUNCTION

// return false when there are no beans in non-pot holes (the game is over)

Board b31(2,0);

Game g11(b31, &bp5, &bp6);

assert(!g11.move());

// return false when only north side has beans (the game is over)

b31.setBeans(NORTH, 1, 3);

Game g12(b31, &bp5, &bp6);

assert(!g11.move());

// return false when only south side has beans

b31.setBeans(NORTH, 1, 0);

b31.setBeans(SOUTH, 1, 3);

assert(!g11.move());

// test the case when the game is not over

BadPlayer bp10("player 1"), bp20("player 2");

Board b32(2,3);

Game g13(b32, &bp10, &bp20);

assert(g13.move()); // a move is possible, so return true

assert(g13.beans(SOUTH, 1) == 0 && g13.beans(SOUTH, 2) == 4 && g13.beans(SOUTH, 0) == 1 && g13.beans(NORTH, 2) == 4 && g13.beans(NORTH, 1) == 3 && g13.beans(NORTH, 0) == 0); // beans are properly sown

assert(g13.move()); // a move is still possible, so return true

assert(g13.beans(SOUTH, 1) == 1 && g13.beans(SOUTH, 2) == 5 && g13.beans(SOUTH, 0) == 1 && g13.beans(NORTH, 2) == 4 && g13.beans(NORTH, 1) == 0 && g13.beans(NORTH, 0) == 1); // the other player took his turn and the beans were sown properly

assert(g13.move()); // a move is still possible, so return true

assert(g13.beans(SOUTH, 1) == 0 && g13.beans(SOUTH, 2) == 6 && g13.beans(SOUTH, 0) == 1 && g13.beans(NORTH, 2) == 4 && g13.beans(NORTH, 1) == 0 && g13.beans(NORTH, 0) == 1); // the turn switched back to south

// test the case when a player gets a second turn

Board b33(2,2);

Game g14(b33, &bp10, &bp20);

assert(g14.move()); // a move is possible, so return true

assert(g14.beans(SOUTH, 1) == 0 && g14.beans(SOUTH, 2) == 0 && g14.beans(SOUTH, 0) == 2 && g14.beans(NORTH, 2) == 0 && g14.beans(NORTH, 1) == 0 && g14.beans(NORTH, 0) == 6); // a double move was made (based on the new distribution of beans)

// test when a player gets 3 consecutive turns, but can't make the last one because the game ended

Board b34(2,2);

b34.setBeans(SOUTH, 2, 0);

Game g15(b34, &bp10, &bp20);

assert(g15.move()); // a move is possible, so return true

assert(g15.beans(SOUTH, 1) == 0 && g15.beans(SOUTH, 2) == 0 && g15.beans(SOUTH, 0) == 2 && g15.beans(NORTH, 2) == 0 && g15.beans(NORTH, 1) == 0 && g15.beans(NORTH, 0) == 4); // return the result after the first 2 moves; the third can't be completed (because south side is empty), so do nothing

// test when a player gets 3 consecutive turns that can be completed successfully

Board b35(3,0);

b35.setBeans(SOUTH, 1, 3); b35.setBeans(SOUTH, 2, 1); b35.setBeans(NORTH, 2, 1);

Game g16(b35, &bp10, &bp20);

assert(g16.move()); // a move is possible, so return true

assert(g16.beans(SOUTH, 1) == 0 && g16.beans(SOUTH, 2) == 0 && g16.beans(SOUTH, 3) == 0 && g16.beans(SOUTH, 0) == 3 && g16.beans(NORTH, 3) == 0 && g16.beans(NORTH, 2) == 0 && g16.beans(NORTH, 1) == 0 && g16.beans(NORTH, 0) == 2); // return correct result after 3 consecutive moves

// test a capture by the south side

Board b36(3,1);

b36.setBeans(SOUTH, 3, 0); b36.setBeans(SOUTH, 1, 2); b36.setBeans(NORTH, 1, 2);

Game g17(b36, &bp10, &bp20);

assert(g17.move()); // a move is possible, so return true

assert(g17.beans(SOUTH, 1) == 0 && g17.beans(SOUTH, 2) == 2 && g17.beans(SOUTH, 3) == 0 && g17.beans(SOUTH, 0) == 2 && g17.beans(NORTH, 3) == 0 && g17.beans(NORTH, 2) == 1 && g17.beans(NORTH, 1) == 2 && g17.beans(NORTH, 0) == 0); // returns correct bean counts after a capture

assert(g17.move()); // moves are available, so return true

assert(g17.beans(SOUTH, 1) == 1 && g17.beans(SOUTH, 2) == 2 && g17.beans(SOUTH, 3) == 0 && g17.beans(SOUTH, 0) == 2 && g17.beans(NORTH, 3) == 0 && g17.beans(NORTH, 2) == 1 && g17.beans(NORTH, 1) == 0 && g17.beans(NORTH, 0) == 1); // the turn changed correctly

// test a capture by the north side

Board b37(3,0);

b37.setBeans(SOUTH, 1, 1); b37.setBeans(NORTH, 3, 1);

Game g18(b37, &bp10, &bp20);

assert(g18.move()); // moves are available, so return true

assert(g18.move());

assert(g18.beans(SOUTH, 1) == 0 && g18.beans(SOUTH, 2) == 0 && g18.beans(SOUTH, 3) == 0 && g18.beans(SOUTH, 0) == 0 && g18.beans(NORTH, 3) == 0 && g18.beans(NORTH, 2) == 0 && g18.beans(NORTH, 1) == 0 && g18.beans(NORTH, 0) == 2); // north captured the beans

// TEST THE GAME'S PLAY METHOD

// if south starts with no beans, sweep beans into north's pot and treat north as the winner

Board b38(3,0);

b38.setBeans(NORTH, 1, 1); b38.setBeans(NORTH, 3, 1);

Game g19(b38, &bp10, &bp20);

g19.play();

// if both sides have the same number of beans at the end, announce a tie

Board b39(2,0);

b39.setBeans(NORTH, 1, 1); b39.setBeans(SOUTH, 2, 1);

Game g20(b39, &bp10, &bp20);

g20.play();

// test the case when the south side wins

Board b40(2,0);

b40.setBeans(NORTH, 2, 1); b40.setBeans(SOUTH, 1, 1);

Game g21(b40, &bp10, &bp20);

g21.play();

// Test the case when the north side wins

Board b41(2,1);

b41.setBeans(SOUTH, 1, 0);

Game g22(b41, &bp10, &bp20);

g22.play();

// Test gameplay with a human player

Board b42(4,2);

HumanPlayer hp2("me");

Game g23(b42, &hp2, &bp10);

g23.play();

// the rest are tests that were given in the spec:

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 over1;

bool hasWinner1;

Side winner1;

// Homer

// 0 1 2

// 0 0

// 2 0 0

// Bart

g.status(over1, hasWinner1, winner1);

assert(!over1 && 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();

// 0 1 0

// 0 3

// 0 1 0

g.status(over1, hasWinner1, winner1);

assert(!over1 && 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();

// 1 0 0

// 0 3

// 0 1 0

g.status(over1, hasWinner1, winner1);

assert(!over1 && 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();

// 1 0 0

// 0 3

// 0 0 1

g.status(over1, hasWinner1, winner1);

assert(!over1 && 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();

// 0 0 0

// 1 4

// 0 0 0

g.status(over1, hasWinner1, winner1);

assert(over1 && 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(hasWinner1 && winner1 == SOUTH);