**Project 3 Report**

**Class Descriptions**

Board

For the Board class I chose to represent the Kalah board with two dynamically allocated arrays. Both arrays increased with the increasing hole number (ex. South hole 2 would be m\_south[1] and North hole 2 would be m\_north[1]). In order to fill counterclockwise, I filled the south side array forward and the north side array backwards.

I defined three private member functions:

1. *side* returns a pointer to the first element in either the north or south array, given the Side. I created it b/c I kept checking the side and then selecting the array in multiple functions
2. *sowNorth* adds a bean to each hole on the North side until it reaches the pot. If it reaches the pot, then I determine if beans should be going in that pot or into the next side. I created it so that I could easily add to the remaining holes on the North side, without writing the code multiple times in the sow function.
3. *sowSouth* adds a bean to each hole on the South side until it reaches the pot. If it reaches the pot, then I determine if beans should be going in that pot or into the next side. I created it so that I could easily add to the remaining holes on the South side, without writing the code multiple times in the sow function.

Player

Player is an abstract base class that BadPlayer, HumanPlayer, and SmartPlayer inherit from. Player has a private member variable m\_name, which is sets using nam(). All of the subclasses inherit the name function and in their constructors they initialize the Player superclas with a given name. For chooseMove, BadPlayer chooses the leftmost non-empty hole. HumanPlayer is the only interactive class and it chooses a move when the user is prompted. For SmartPlayer, I created two helper functions helper\_chooseMove and evaluate. helper\_chooseMove does all of the work for chooseMove (which I will go into in my descritpion of chooseMove) and I decided to create this function so that I could evaluate each hole, give a certain depth, and then return the bestHole to the chooseMove function. I created the evaluate function to evaluate each hole (I will also go into this in the next question) and because I wanted to call it recursively.

Game

Game is the class that keeps track of the progress of the game, and tells players to move, etc. display just displays the board. status checks if the game is over and sets values to over, hasWinner, winner. move checks the status and calls chooseMove for the current player. It checks if the last bean is placed in the pot or if there is a capture. It then switches the side of the player and the current player. play calls m\_play with bool false. I decided to create the member function m\_play which takes a bool as an argument. Every time a new player moves it calls itself recursively and changes it's bool. If the bool is true then the user is prompted to press enter. If the game is over it couts the necessary information.

**Description of your design for SmartPlayer::chooseMove**

For the heuristics, I chose to evaluate the difference between the number of beans in my pot vs the number of beans in my opponents pot. If there was a winning possibility I set that value to 1000, if there was a losing possibility I set that value to -1000, if it's a tie the value is zero. If none of those possibilities occur then the value is the difference between the other players pot and the SmartPlayer's pot.

My SmartPlayer::chooseMove calls a helper function with the side, board, bestHole, value, and depth. I chose a value of 5 for depth. After receiving the information from that function it returns the best hole. My helper\_chooseMove checks if the game is over, and if it isn't it loops through the possible hole options. At each hole it calls the evaluate function and resets the bestHole if the evaluate function gives a better value at that hole. The evaluate function takes in a board, side, hole, depth, and value. It sows the beans from the potential option. If the turn ends in a pot, it keeps sowing and evaluating the potential for sowing the next beans, b/c it's technically the same turn. If the game is over, then it sweeps the beans and determines if we have more beans than the other player. It then gives a value to win (1000), loss (-1000), tie (0). If none of those options are met than the value is the difference between the pots.

Now we get to the min/max:

If the side that we are currently on matches the original player. I want to pick the minimum option of the other player's possible moves. I set the current minimum to 1000, b/c everything is smaller than that (if nothing is smaller, then the other player will win) and decrease the depth. I then evaluate the each hole that the other player could choose and set the current minimum to the smallest value. If the side that we are currently on does not match the original player. I want to pick the maximum options of SmartPlayers possible moves. I set the current maximum to -1000, b/c everything is larger than that (if nothing is larger, then we won't win) and decrease the depth. I then evaluate the each hole that the other player could choose and set the current maximum to the largest value.

**Pseudocode (for functions more than 5 lines of code)**

Board

**Board(int nHoles, int nInitialBeansPerHole)**

*if nHoles is negative or 0*

*set it to 1*

*if nInitialBeansPerHole is negative*

*set it to 0*

*set totalBeans to 2\*nHoles\* nInitialBeansPerHole*

*create two dynamically allocated arrays (m\_north and m\_south with size nHoles)*

*repeatedly*

*fill each hole in both arrays with nInitialBeansPerHole*

**Board(const Board& other)**

*set built in member variables equal to other built in member variables*

*create two dynamically allocated arrays (m\_north and m\_south)*

*iterate through m\_nHoles*

*assign the value of m\_north to the value of the other m\_north at position*

*assign the value of m\_south to the value of the other m\_south at position*

**Board& operator(const Board& other)**

*if the current Board is not the same as the other Board (aliasing)*

*set built in member variables equal to other built in member variables*

*delete existing dynamically allocated arrays*

*create two dynamically allocated arrays (m\_north and m\_south)*

*iterate through m\_nHoles*

*assign the value of m\_north to the value of the other m\_north at position*

*assign the value of m\_south to the value of the other m\_south at position*

*return reference to Board*

**int beans(Side s, int hole)**

*set side to pointer to first element of array of correct side*

*if the hole given is negative or greater than nHoles*

*return -1*

*if the hole is a pot*

*return amount in north or south pot, depending on side*

*else*

*return amount in specified hole*

**int beansInPlay(Side s)**

*set side to pointer to first element of array of correct side*

*iterate through elements in array*

*add values to total amount*

*return total amount*

**bool sow(Side s, int hole, Side& endside, int& endHole)**

*set side to pointer to first element of array of correct side*

*if hole is negative, out of bounds, a pot, or empty*

*return false*

*record the number of beans*

*empty hole*

*while the number of beans is greater than 0*

*if side is SOUTH*

*increment hole (so that sowSouth adds too next hole, not current hole)*

*call sowSouth with the numBeans, hole, endSide, endHole, and starting side*

*if no more beans - break*

*call sowNorth with the numBeans, last hole, endSide, endHole, and starting side*

*if no more beans - break*

*set hole to 0, will call sowSouth again starting from first position*

*if side is NORTH*

*decrement hole (so that sowNorth adds to next hole, not current hole)*

*call sowNorth with numBeans, hole, endSide, endHole, and starting side*

*if no more beans - break*

*call sowSouth with numBeans, first hole, endSide, endHole, and starting side*

*if no more beans - break*

*set hole to element after last, will call sowNorth from last hole*

*return true*

**bool moveToPot(Side s, int hole, Side potOwner)**

*set side to pointer to first element of array of correct side*

*if hole is negative or out of bounds or the pot*

*return false*

*if potOwner is NORTH*

*add hole amount to north pot*

*if potOwner is SOUTH*

*add hole amount to south pot*

*remove amount from hole*

*return true*

**bool setBeans(Side s, int hole, int beans)**

*set side to pointer to first element of array of correct side*

*if hole is negative or out of bounds, or there is a negative amount of beans*

*return false*

*if hole is the pot*

*if side is NORTH*

*subtract north pot amount from total*

*add beans to north pot*

*is side is SOUTH*

*subtract south pot amount from total*

*add beans to south pot*

*else*

*subtract hole amount from total*

*add beans to hole*

*add beans to total*

*return true;*

Private Member Functions

**int\* side (Side s)**

*if side is North*

*return pointer to first element in m\_north array*

*if side is South*

*return pointer to first element in m\_south array*

**int sowSouth(int numBeans, int hole, Side& endSide, int& endHole, Side startingSide)**

*while there are still beans and there are still holes in range (not pot) (moving forward)*

*add 1 bean to the hole*

*subtract 1 bean from numBeans*

*move to next hole*

*if there are still beans and the pot corresponds to the startingSide*

*add 1 bean to pot*

*subtract 1 bean from numBeans*

*ending hole is the pot*

*else (if there are no more beans, or not correct pot)*

*ending Side is SOUTH*

*ending hole is the last hole beans went into*

*return the remaining number of beans*

**int sowNorth(int numBeans, int hole, Side& endSide, int& endHole, Side startingSide)**

*while there are still beans and there are still holes in range (not pot) (moving backward)*

*add 1 bean to the hole*

*subtract 1 bean from numBeans*

*move to next hole*

*if there are still beans and the pot corresponds to the startingSide*

*add 1 bean to pot*

*subtract 1 bean from numBeans*

*ending hole is the pot*

*else (if there are no more beans, or not correct pot)*

*ending Side is NORTH*

*ending hole is the last hole beans went into*

*return the remaining number of beans*

Player

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

*loop through holes*

*if hole is not empty*

*return hole (ie. will pick the first non-empty hole)*

*return -1 (ie. no possible moves)*

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

*set bool check to true*

*while check is true and the game is not over*

*prompt human to pick a hole*

*if the the hole is valid*

*set check to false (ie. break out of loop)*

*if the hole is not valid keep prompting*

*return hole value (-1 if game is over)*

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

*call helper\_chooseMove with input side, board, bestHole, and value*

*return best hole*

Private Member Functions

**SmartPlayer::helper\_chooseMove(Side s, const Board& b, int& bestHole, int& value) const**

*set depth to 5*

*make copy of the board*

*if the one side is empty (ie. game is over)*

*set best hole to -1*

*return;*

*repeatedly check for valid holes (not negative, empty, or greater than the amount of hole)*

*create a copy of the board*

*if current value at valid hole is less than the value given by the evaluate function at that hole given the board, side, hole, depth, current value, and player*

*set best hole to current hole*

*set value to the value at that hole*

**SmartPlayer::evaluate(Board& myBoard, Side s, int hole, int& depth, int& value, Side player) const**

*if the depth reaches 0*

*return the value*

*sow the beans at the given hole*

*if the ending side is the same as the starting side, the ending hole has 1 bean (newly added), and the hole directly opposite is not empty, and it's not the pot*

*move all of the beans from the ending hole to the pot of the player*

*move all of the beans from the opposite hole to the pot of the player*

*if the hole chosen is empty*

*while the ending hole is not the pot*

*if the game is over*

*return the value*

*if the game is not over*

*choose a new hole*

*evaluate that hole*

*if one or both sides are empty (game is over)*

*sweep beans*

*if there is a tie*

*set value to 0*

*if our player wins*

*set value to 1000*

*if our player loses*

*set value to -1000*

*return value*

*set value to be the difference between the current side's pot and the other side's pot*

*if the current side is the same as the player's side*

*set current minimum to 1000*

*decrease depth*

*repeatedly check for valid holes (not negative, empty, or greater than the amount of hole)*

*if find a valid choice*

*if current minimum at valid hole is less than the value given by the evaluate function at that hole given the board, the other side, hole, depth, current value, and player*

*set current minimum to the value given by the evaluate function*

*return current minimum*

*if the current side is not the same as the player's side*

*decrease depth*

*set current maximum to 1000*

*repeatedly check for valid holes (not negative, empty, or greater than the amount of hole)*

*if find a valid choice*

*if current maximum at valid hole is greater than the value given by the evaluate function at that hole given the board, the other side, hole, depth, current value, and player*

*set current maximum to the value given by the evaluate function*

*if current maximum is smaller*

*return current maximum*

Game

**void Game::display() const**

*cout players names*

*cout north*

*cout north holes and the amount of beans in hole*

*cout North pot and south pot*

*cout south holes and the amount of beans in hole*

*cout south*

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

*if neither side is empty (game still going)*

*set over to false*

*return*

*if either side is empty (end of game), set over to true*

*set hasWinner to false if it's a tie*

*set hasWinner to true if it's true*

*set Winner to player that has more beans in pot*

**bool Game::move()**

*check status*

*if game is over*

*sweep beans*

*return false*

*player whose turn it is selects a hole*

*if the last bean is placed in a pot*

*repeatedly choose new holes until you don't land on a pot*

*display board*

*select new hole*

*if not over*

*sow beans*

*if ending side is the same as the current player's side and the hole was empty before and there are beans in the other players hole*

*move beans from other players hole to current players pot*

*move beans from ending hole to current players pot*

*switch current side to side of other player*

*switch current player with other player (b/c now it's other player's turn)*

*return true*

**void Game::play()**

*call m\_play with bool false so that*

Private Member Functions

**void Game::m\_play(bool s)**

*if neither of the player are interactive*

*prompt user to press ENTER,*

*check status*

*if the game is over and there's a winner*

*cout the name of the winner*

*if the game is over and there isn't a winner*

*cout that it's a tie*

*if the game is not over*

*cout it's (insert player's name)'s turn*

*call move*

*call display*

*call m\_play with opposite of bool*

**Note about bugs, serious inefficiencies, or notable problems**

While my chooseMove for smart player passes basic tests, I am not sure how it handles more complex cases. I was not able to thoroughly test these complex cases because tracing through recursive functions is extremely difficult, tedious, and not very informative. For smaller games, the game will end within only a few recursive calls so the evaluate function will almost always return 1000 or -1000, whereas with a larger board it may return smaller values. I did play against the SmartPlayer multiple times and I think it does what it should be doing.

**Test Cases**

**void** doBoardTests()

{

Board e(0,-1); *//test empty board (1 hole no beans)*

assert(e.holes() == 1); *//tests constructor default holes to 1*

assert(e.totalBeans() == 0); *//tests negative value constructor totalBeans*

assert(e.beans(SOUTH, POT) == 0); *//test beans with empty pot*

assert(e.beans(SOUTH, 1) == 0); *//test beans with empty hole*

assert(e.beansInPlay(SOUTH) == 0); *//test beans in play w/ no beans*

Board e1(-1,3); *//test 1 hole with beans*

assert(e1.holes() == 1); *//tests constructor default holes to 1*

assert(e1.totalBeans() == 6); *//tests totalBeans*

assert(e1.beans(SOUTH, POT) == 0); *//test beans with empty pot*

assert(e1.beansInPlay(SOUTH) == 3); *//test beans in play w/ beans*

assert(e1.beans(SOUTH, 7) == -1); *//test beans with out of bounds*

assert(e1.beans(SOUTH, -3) == -1); *//test beans with negative hole given*

Board b(3,2); *//test board with multiple beans and holes*

assert(b.holes() == 3); *//test holes with multiple holes*

assert(b.totalBeans() == 12); *//test beans with multiple beans and holes*

assert(b.beans(SOUTH, POT) == 0); *//test beans with empty pot*

assert(b.beansInPlay(SOUTH) == 6 && b.beansInPlay(NORTH) == 6); *//tests beans in play (at starting North and South should be equal*

Board c(3,2); *//test copy constructor*

assert(c.holes() == 3); *//test holes with multiple holes*

assert(c.totalBeans() == 12); *//test beans with multiple beans and holes*

assert(c.beans(SOUTH, POT) == 0); *//test beans with empty pot*

assert(c.beansInPlay(SOUTH) == 6 && c.beansInPlay(NORTH) == 6); *//tests beans in play (at starting North and South should be equal*

e = c; *//test assignment operator*

assert(e.holes() == 3); *//test holes with multiple holes*

assert(e.totalBeans() == 12); *//test beans with multiple beans and holes*

assert(e.beans(SOUTH, POT) == 0); *//test beans with empty pot*

assert(e.beansInPlay(SOUTH) == 6 && e.beansInPlay(NORTH) == 6); *//tests beans in play (at starting North and South should be equal*

assert(!b.setBeans(SOUTH, 5, 2)); *//test setBeans out of range hole*

assert(!b.setBeans(SOUTH, -1, 2)); *//test setBeans with negative hole*

assert(!b.setBeans(SOUTH, 1, -4)); *//test setBeans with negative beans*

assert(b.setBeans(SOUTH, 1, 1)); *//test setBeans with valid hole and bean number*

assert(b.totalBeans() == 11); *//test totalBeans after setBeans*

assert(b.beansInPlay(SOUTH) == 5); *//test beansInPlay after setBeans*

assert(b.beans(SOUTH, 1) == 1); *//test beans after setBeans*

assert(!b.moveToPot(SOUTH, -2, SOUTH)); *//test moveToPot with negative hole*

assert(!b.moveToPot(SOUTH, 7, SOUTH)); *//test moveToPot with out of range hole*

assert(!b.moveToPot(SOUTH, 0, SOUTH)); *//test moveToPot with pot*

assert(b.moveToPot(SOUTH, 2, SOUTH)); *//test moveToPot with valid arguments*

assert(b.totalBeans() == 11); *//test totalBeans after moveToPot (should stay the same)*

assert(b.beans(SOUTH, 2) == 0); *//test beans with hole after moveToPot (shold be empty)*

assert(b.beans(SOUTH, POT) == 2); *//test beans with pot after moveToPot (should have amount of beans from hole)*

assert(b.beansInPlay(SOUTH) == 3); *//test beansInPlay after moveToPot (should be less))*

Side es;

**int** eh;

assert(!b.sow(SOUTH, 7, es, eh)); *//test sow with out of bounds*

assert(!b.sow(SOUTH, -7, es, eh)); *//test sow with negative*

assert(!b.sow(SOUTH, 0, es, eh)); *//test sow with pot*

assert(!b.sow(SOUTH, 2, es, eh)); *//test sow with empty hole*

assert(b.sow(SOUTH, 3, es, eh)); *//test sow with valid inputs*

assert(es == NORTH); *//checks that it made it to other side*

assert(eh == 3); *//checks last hole*

assert(b.beans(SOUTH, 3) == 0); *//empty because removed beans*

assert(b.beans(NORTH, 3) == 3); *//added bean*

assert(b.beans(SOUTH, POT) == 3); *//pot increased*

assert(b.beansInPlay(SOUTH) == 1); *//only 1 bean on south side*

assert(b.beansInPlay(NORTH) == 7); *//7 beans on North side*

b.sow(NORTH, 3, es, eh); *//test sow on North side*

assert(es == NORTH); *//checks that it made it to other side*

assert(eh == 0); *//checks last hole*

assert(b.beans(SOUTH, 3) == 0); *//empty because removed beans*

assert(b.beans(NORTH, 3) == 0); *//added bean*

assert(b.beans(SOUTH, POT) == 3); *//pot stayed the same*

assert(b.beans(NORTH, POT) == 1); *//pot increased*

assert(b.beansInPlay(SOUTH) == 1); *//only 1 bean on south side*

assert(b.beansInPlay(NORTH) == 6); *//7 beans on North side*

b.setBeans(SOUTH,2,10);

b.sow(SOUTH, 2, es, eh); *//test sow going around the board multiple times starting SOUTH*

assert(es == NORTH);

assert(eh == 3);

assert(b.totalBeans() == 21); *//test totalBeans increase*

*//testing each hole*

assert(b.beans(SOUTH, 1) == 2);

assert(b.beans(SOUTH, 2) == 1);

assert(b.beans(SOUTH, 3) == 2);

assert(b.beans(NORTH, 1) == 4);

assert(b.beans(NORTH, 2) == 4);

assert(b.beans(NORTH, 3) == 2);

assert(b.beans(NORTH, POT) == 1); *//stayed the same*

assert(b.beans(SOUTH, POT) == 5); *//pot increased*

b.setBeans(NORTH,2,10);

b.sow(NORTH, 2, es, eh); *//test sow going around the board multiple times starting NORTH*

assert(es == SOUTH);

assert(eh == 1);

assert(b.totalBeans() == 27); *//test totalBeans increase*

*//testing each hole*

assert(b.beans(SOUTH, 1) == 4);

assert(b.beans(SOUTH, 2) == 2);

assert(b.beans(SOUTH, 3) == 3);

assert(b.beans(NORTH, 1) == 6);

assert(b.beans(NORTH, 2) == 1);

assert(b.beans(NORTH, 3) == 3);

assert(b.beans(NORTH, POT) == 3); *//pot increased*

assert(b.beans(SOUTH, POT) == 5); *//stayed the same*

assert(b.moveToPot(SOUTH, 2, NORTH)); *//test moveToPot with valid arguments (opposite sides)*

assert(b.beans(SOUTH, 2) == 0); *//test beans with hole after moveToPot (shold be empty)*

assert(b.beans(SOUTH, POT) == 5); *//test beans with pot after moveToPot (should have amount of beans from hole)*

assert(b.beans(NORTH, POT) == 5); *//test beans with pot after moveToPot (should have amount of beans from hole)*

assert(b.beansInPlay(SOUTH) == 7); *//test beansInPlay after moveToPot (should be less))*

assert(b.beansInPlay(NORTH) == 10); *//test beansInPlay after moveToPot (should be less))*

cout << "Passed all Board Tests" << endl;

}

**void** doPlayerTests() {

Player\* player[3]; *//create array of pointers to class*

player[0] = **new** BadPlayer("penny"); *//test inheritance (superclass pointer can point to subclass)*

player[1] = **new** HumanPlayer("george");

player[2] = **new** SmartPlayer("smartie pants");

assert(player[0]->name() == "penny"); *//test inheritance of name member function*

assert(player[1]->name() == "george");

assert(player[2]->name() == "smartie pants");

assert(!player[0]->isInteractive()); *//test isInteractive virtual function*

assert(player[1]->isInteractive());

assert(!player[2]->isInteractive());

Board b(3,2); *//test chooseMove (BadPlayer will pick leftmost, non-empty hole)*

assert(player[0]->chooseMove(b, SOUTH) == 1);

b.setBeans(SOUTH, 1, 0);

b.setBeans(SOUTH, 2, 0);

b.setBeans(SOUTH, 3, 0);

*//shouldn't be able to pick 2*

assert(player[0]->chooseMove(b,SOUTH) == -1);

**for** (**int** k = 0; k < 3; k++)

**delete** player[k];

cout << "Passed all Player test" << endl;

}

**void** doGameTests() {

BadPlayer p1("bad");

BadPlayer p2("alsoBad");

Board b(3,1);

**bool** hasWinner = **false**;

**bool** isOver = **false**;

Side winner = NORTH;

Game g(b,&p1,&p2); *//test constructor*

assert(g.beans(NORTH, 7) == -1); *//invalid hole #*

assert(g.beans(NORTH, 1) == 1); *//test beans*

assert(g.beans(NORTH, 2) == 1);

assert(g.beans(NORTH, 3) == 1);

assert(g.beans(SOUTH, 1) == 1);

assert(g.beans(SOUTH, 2) == 1);

assert(g.beans(SOUTH, 3) == 1);

g.display(); *//test display*

*/\**

*South player: bad North player: alsoBad*

*North*

*1 1 1*

*North pot 0 0 South Pot*

*1 1 1*

*South*

*\*/*

g.status(isOver,hasWinner,winner);

assert(isOver == **false** && hasWinner == **false**); *//test status when game is not over*

g.move(); *//testing move, SOUTH is first so it shoud move*

assert(g.beans(NORTH, 1) == 1); *//test beans*

assert(g.beans(NORTH, 2) == 1);

assert(g.beans(NORTH, 3) == 1);

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

assert(g.beans(SOUTH, 1) == 0);

assert(g.beans(SOUTH, 2) == 2);

assert(g.beans(SOUTH, 3) == 1);

assert(g.beans(SOUTH, 0) == 0);

g.move(); *//currentPlayer should switch to NORTH, which will move twic, b/c it lands in a hole the first time*

assert(g.beans(NORTH, 1) == 1); *//test beans*

assert(g.beans(NORTH, 2) == 0);

assert(g.beans(NORTH, 3) == 1);

assert(g.beans(NORTH, 0) == 1);

assert(g.beans(SOUTH, 1) == 0);

assert(g.beans(SOUTH, 2) == 2);

assert(g.beans(SOUTH, 3) == 1);

assert(g.beans(SOUTH, 0) == 0);

g.play();

*//should be prompting me to press enter to make sure that I keeo up with the game*

*/\**

*South player: bad North player: alsoBad*

*North*

*1 0 1*

*North pot 1 0 South Pot*

*0 2 1*

*South*

*South player: bad North player: alsoBad*

*North*

*1 0 2*

*North pot 1 2 South Pot*

*0 0 0*

*South*

*South player: bad North player: alsoBad*

*North*

*0 0 0*

*North pot 4 2 South Pot*

*0 0 0*

*South*

*\*/*

assert(g.beans(NORTH, 1) == 0);

assert(g.beans(NORTH, 2) == 0);

assert(g.beans(NORTH, 3) == 0);

assert(g.beans(NORTH, 0) == 4);

assert(g.beans(SOUTH, 1) == 0);

assert(g.beans(SOUTH, 2) == 0);

assert(g.beans(SOUTH, 3) == 0);

assert(g.beans(SOUTH, 0) == 2);

g.display(); *//test display*

g.status(isOver,hasWinner,winner);

assert(g.move() == **false**);

assert(isOver == **true** && hasWinner == **true** && winner == NORTH); *//test status when game is over*

g.play(); *//since nothing to play in SOUTH side, all beans should be swept in the NORTH side and game is over*

assert(g.beans(NORTH, 1) == 0); *//test beans*

assert(g.beans(NORTH, 2) == 0);

assert(g.beans(NORTH, 3) == 0);

assert(g.beans(NORTH, 0) == 4);

assert(g.beans(SOUTH, 1) == 0);

assert(g.beans(SOUTH, 2) == 0);

assert(g.beans(SOUTH, 3) == 0);

assert(g.beans(SOUTH, 0) == 2);

g.display(); *//test display*

g.status(isOver,hasWinner,winner);

assert(isOver == **true** && hasWinner == **true** && winner == NORTH);

cout << "Passed all Game Tests" << endl;

}

**void** doSmartPlayerTests() {

SmartPlayer s("smartie pants");

HumanPlayer e("evelyn");

**bool** hasWinner = **false**;

**bool** isOver = **false**;

Side winner = NORTH;

*//GAME 1*

*//should choose 1, b/c will deposit in pot, then choose 2 deposit in pot, then choose 1 deposit in pot..etc until all beans on that side are in the pot*

Board b1(3, 1);

b1.setBeans(NORTH, 1, 1);

b1.setBeans(NORTH, 2, 0);

b1.setBeans(NORTH, 3, 0);

b1.setBeans(SOUTH, 1, 0);

b1.setBeans(SOUTH, 2, 2);

b1.setBeans(SOUTH, 3, 1);

*/\**

*1 0 0*

*0 0*

*0 2 1*

*\*/*

Game g1(b1, &s, &e);

assert(s.chooseMove(b1, SOUTH) == 3);

g1.play();

assert(g1.beans(SOUTH, 0) == 3);

g1.status(isOver, hasWinner, winner);

assert(isOver == **true** && hasWinner == **true** && winner == SOUTH);

*//GAME 2*

*//should choose 2, b/c will capture the 1 on the other side, and end game since the holes on the north side will be empty*

Board b2(3, 1);

b2.setBeans(NORTH, 1, 0);

b2.setBeans(NORTH, 2, 0);

b2.setBeans(NORTH, 3, 1);

b2.setBeans(SOUTH, 1, 1);

b2.setBeans(SOUTH, 2, 1);

b2.setBeans(SOUTH, 3, 0);

*/\**

*0 0 1*

*0 0*

*1 1 0*

*\*/*

Game g2(b2, &s, &e);

assert(s.chooseMove(b2, SOUTH) == 2);

g2.play();

assert(g2.beans(SOUTH, 0) == 3);

g2.status(isOver, hasWinner, winner);

assert(isOver == **true** && hasWinner == **true** && winner == SOUTH);

*//GAME 3*

*//should pick 1 so that it can capture 2 and win the game*

Board b3(4,1);

b3.setBeans(NORTH, 1, 0);

b3.setBeans(NORTH, 2, 2);

b3.setBeans(NORTH, 3, 0);

b3.setBeans(NORTH, 4, 0);

b3.setBeans(SOUTH, 1, 1);

b3.setBeans(SOUTH, 2, 0);

b3.setBeans(SOUTH, 3, 1);

b3.setBeans(SOUTH, 4, 0);

*/\**

*0 2 0 0*

*0 0*

*1 0 1 0*

*\*/*

Game g3(b3, &s, &e);

assert(s.chooseMove(b3, SOUTH) == 1);

g3.play();

assert(g3.beans(SOUTH, 0) == 4);

g3.status(isOver, hasWinner, winner);

assert(isOver == **true** && hasWinner == **true** && winner == SOUTH);

*//GAME 4*

*//should pick 2 so that it can choose move again*

Board b4(4,1);

b4.setBeans(NORTH, 1, 0);

b4.setBeans(NORTH, 2, 2);

b4.setBeans(NORTH, 3, 0);

b4.setBeans(NORTH, 4, 0);

b4.setBeans(SOUTH, 1, 0);

b4.setBeans(SOUTH, 2, 1);

b4.setBeans(SOUTH, 3, 2);

b4.setBeans(SOUTH, 4, 0);

*/\**

*0 2 0 0*

*0 0*

*0 1 2 0*

*\*/*

Game g4(b4, &s, &e);

assert(s.chooseMove(b4, SOUTH) == 3);

g4.play();

assert(g4.beans(SOUTH, 0) == 3);

g4.status(isOver, hasWinner, winner);

*//after playing my turn (pick 2 and then pick one and tie)*

assert(isOver == **true** && hasWinner == **true** && winner == SOUTH); *//after sweep south wins*

*//GAME 5*

*//empty board*

Board b5(4,1);

b5.setBeans(NORTH, 1, 0);

b5.setBeans(NORTH, 2, 0);

b5.setBeans(NORTH, 3, 0);

b5.setBeans(NORTH, 4, 0);

b5.setBeans(SOUTH, 1, 0);

b5.setBeans(SOUTH, 2, 0);

b5.setBeans(SOUTH, 3, 0);

b5.setBeans(SOUTH, 4, 0);

*/\**

*0 0 0 0*

*0 0*

*0 0 0 0*

*\*/*

Game g5(b5, &s, &e);

assert(s.chooseMove(b5, SOUTH) == -1);

g5.status(isOver, hasWinner, winner);

assert(isOver == **true** && hasWinner == **false**); *//it's a tie*

*//GAME 6*

*//should capture other side's hole*

Board b6(5,1);

b6.setBeans(NORTH, 1, 0);

b6.setBeans(NORTH, 2, 0);

b6.setBeans(NORTH, 3, 1);

b6.setBeans(NORTH, 4, 0);

b6.setBeans(NORTH, 5, 0);

b6.setBeans(SOUTH, 1, 0);

b6.setBeans(SOUTH, 2, 1);

b6.setBeans(SOUTH, 3, 0);

b6.setBeans(SOUTH, 4, 1);

b6.setBeans(SOUTH, 5, 0);

*/\**

*0 0 1 0 0*

*0 0*

*0 1 0 1 0*

*\*/*

Game g6(b6, &s, &e);

assert(s.chooseMove(b6, SOUTH) == 2);

g6.play();

assert(g6.beans(SOUTH, 0) == 3);

g6.status(isOver, hasWinner, winner);

assert(isOver == **true** && hasWinner == **true** && winner == SOUTH); *//wins b/c capture*

*//GAME 7*

*//should pick hole that will let them go again*

Board b7(5,1);

b7.setBeans(NORTH, 1, 0);

b7.setBeans(NORTH, 2, 0);

b7.setBeans(NORTH, 3, 1);

b7.setBeans(NORTH, 4, 0);

b7.setBeans(NORTH, 5, 0);

b7.setBeans(SOUTH, 1, 0);

b7.setBeans(SOUTH, 2, 1);

b7.setBeans(SOUTH, 3, 0);

b7.setBeans(SOUTH, 4, 1);

b7.setBeans(SOUTH, 5, 0);

*/\**

*0 0 1 0 0*

*0 0*

*0 4 0 1 0*

*\*/*

Game g7(b7, &s, &e);

assert(s.chooseMove(b7, SOUTH) == 2);

Board b8(6,4);

Game g8(b8,&s,&e);

g8.play();

cout << "Passed all Smart Player Tests" << endl;

}