**Com Sci 32**

**Project 3 Report**

# **Design Documentation**

**Class Board**

The decision of choosing an appropriate data structure for implementing the Board class was a critical one, and considering all possible alternatives known at the time, dynamic arrays seemed to best fit my intention. The parameter nHoles provided in the constructor was also stored as a private member variable m\_Holes for any later usage. Considering the following private data members, each public member function of the class was implemented as follows –

1. **Board::Board(int nHoles, int nInitialBeansPerHole)** – the constructor stored the value of the parameter nHoles into the private data member m\_Holes and initialized the two dynamic arrays, m\_NorthSide and m\_SouthSide to be of size m\_Holes+1 (as a pot can also be considered a hole at index 0).
2. **Board::Board(const Board& otherBoard)** – as we frequently need to create copies of Boards for SmartPlayer class’s chooseMove() function, and as the default copy constructor would result in a shallow copy of the private data member dynamic arrays, the copy constructor had to implemented to ensure a deep copy.
3. **Board& Board::operator= (const Board& otherBoard)** – although my current implementation doesn’t rely on assigning one board to another anywhere, as a good practice, the assignment operator has been overloaded to ensure deep copy in any cases when a board might be assigned to another.
4. **Board::~Board()** – the default destructor would overlook dynamic memory allocation resulting in a memory leak, hence we must clean up the environment manually by implementing the destructor.
5. **int Board::holes() const** – returns m\_Holes.
6. **int Board::beans(Side s, int hole) const** – references the appropriate private data member dynamic array to return the number of beans at that position (works for pots too).
7. **int Board::beansInPlay(Side s) const** – loops through the appropriate private data member array and sums up the beans in each hole excluding the pot, returning the sum.
8. **int Board::totalBeans() const** – sums up each element of each data member array and returns this value representing the total number of beans on board.
9. **bool Board::sow(Side s, int hole, Side& endSide, int& endhole)** – sets array[hole] to zero, where array is an appropriate data member array for side s, and thereafter moves counterclockwise (according to the number system provided by the spec), incrementing the beans in each hole by 1 (doing so for its own pot but not the opponent’s, should the need arise) until all beans extracted from the initial hole have been exhausted – in essence, the function runs a while loop until beans extracted from the initial holes remain, undertaking appropriate actions for the current hole in each iteration. Due to the higher complexity of this function, the pseudocode for this function provided below will be a better way to understand how this function works.
10. **bool Board::moveToPot(Side s, int hole, Side potOwner)** – simply adds the output of beans(s, hole) to the array[0], where array is the appropriate array for the side potOwner.
11. **bool Board::setBeans(Side s, int hole, int beans)** – simply sets array[hole] to beans, where array is the appropriate data member array for the side s.

**Class Player**

The class Player by itself is an abstract class which provides a foundation for implementing other classes deriving from it. It has exactly one private data member, m\_name, which is representative of the Player’s name. Other class member functions are as follows –

1. **Player::Player(std::string name)** – simply initializes m\_name to name.
2. **std::string Player::name() const** – returns m\_name.
3. **virtual bool Player::isInteractive() const** – returns false as most kinds of players are not interactive, but is declared virtual so that if any derived classes may indeed be interactive, they could override this function to return true.
4. **virtual int Player::chooseMove(const Board& b, Side s) const = 0** – pure virtual function as each class deriving from Player will require to implement chooseMove() differently.
5. **virtual ~Player()** – as good practice, we include the virtual destructor here since this is a base class and any classes deriving from it might require to override it to deal with any dynamic data they might be using.

**Class HumanPlayer (derives from Player)**

The class HumanPlayer derives from Player and allows interactivity to the player, allowing them to choose whatever move they wish, and notifying them if they attempt to make an illegal move. Its member functions are implemented as follows –

1. **HumanPlayer::HumanPlayer(std::string name)** – simply constructs the base part of HumanPlayer using name.
2. **bool HumanPlayer::isInteractive() const** – overrides the base class’s function to return true, as a human player must be interactive.
3. **virtual int HumanPlayer::chooseMove(const Board& b, Side s)** – this function is implemented in two steps:
4. It loops through b’s side s to see whether or not any of the holes (excluding the pot) are non-empty (this can be done using b’s member function beans(s, i), where i is a legal integer). If all of them are empty, it means that the game is over, and thus -1 must be returned. If any of the holes are non-empty, it means that at least one move can be made.
5. Having assured that the game isn’t over, the function reads input for a hole number from the user. If the hole is non-empty, then the function simply returns this hole number, if not, it asks the user to provide a valid hole which contains at least a bean.

**Class BadPlayer (derives from Player)**

This class can be thought of as a means to play a quick game without considering much possibilities. As one possible rendition of arbitrary gameplay styles, this class’s objects will make a move at the first encountered non-empty hole (considering holes from 1 to b.holes()). The class and its member functions are implemented as follows –

1. **BadPlayer::BadPlayer(std::string name)** – constructs the base part of BadPlayer using name.
2. **virtual int BadPlayer::chooseMove(const Board& b, Side s)** – this function is also implemented in two steps:
3. It traverses through side s in b to check whether or not all holes are empty. If all of them are empty, it means that the game is already over and -1 must be returned.
4. If not, it means that at least one move can be made. To do so, it loops through the side s of b, and as soon as it encounters a non-empty hole, it returns its address.

**Class SmartPlayer (derives from Player)**

This class’s object take well-informed decisions by utilizing the concept of a decision-tree. It is implemented as follows –

1. **SmartPlayer::SmartPlayer(std::string name)** – constructs the base part of SmartPlayer using name.
2. **virtual int SmartPlayer::chooseMove(const Board& b, Side s) const** – returns the best possible hole for the player on Side s to make a move on by utilizing the minimax algorithm. However, instead of doing all the work itself, it instead utilizes the following helper function.
3. **int SmartPlayer::decide(const Board& b, Side s, int& value, int depth, AlarmClock& a)** – this function performs the actual work of returning the best move to make on Side s for a Board b. Note that the decision making capacity of this function is based on both time and depth constraints (whichever is exhausted first). The significance of its various parameters are as follows –
4. value – this is an integer parameter passed by reference. What it will do throughout the function’s working is return the value (assigned according to the minimax algorithm) of a particular hole. One may then decide which hole has the best value (the definition of best varies depending on whether the current player is on the North or the South). This in turn would allow one to return the corresponding hole number, which would be the best hole to make a move on (the decision was made within time/ depth constraints, as will be explained later).
5. depth – this is a simple pass-by-value integer parameter that allows the recursive function to keep track at what depth it is in the decision tree. Additional moves by the same player involve recursive calls to this function with the same side and the same depth. However, if the opponent must play next, the function is called recursively on the opponent at a depth one greater than the present. The function has a terminating condition in terms of depth as well.
6. a – this is an AlarmClock passed by reference to the function to allow it to keep track of whether or not it should start wrapping up its job, depending on how much time a takes to time out. Thus, the AlarmClock timing out is a terminating condition for the recursive function.
7. **int SmartPlayer::evaluate (const Board& b) const** – this includes the heuristics utilized for evaluating the “value” of a particular hole on the board. The heuristic chosen for assigning values to leaves of the decision tree if the game isn’t yet over is the difference in the number of beans in South’s pot and North’s pot. However, if the game is over, we return 999 for South’s victory, -999 for North’s victory, and 0 for a tie. The appropriate values that go up the decision tree is decided by the minimax algorithm which is implemented within the SmartPlayer::decide() function.

A more comprehensive discussion for the implementation of the SmartPlayer class would be undertaken in the section following the rest of the Design Documentation due to the overly complex nature of the SmartPlayer class and its various member functions, especially the SmartPlayer::decide() function.

**Class AlarmClock**

Has been utilized as provided by the spec, and is a means to ensure time efficiency of instances of the SmartPlayer class.

**Class Game**

Before the discussion of its member functions, it is worth discussing the various private data members chosen for this class –

1. Board m\_Board – an actual Board class object to keep track the current state of the Game Board. This allows us to properly implement various public member functions to return appropriate outputs.
2. Player \*m\_South – a pointer to a Player object for the South side of m\_Board. Note that it is wise to have a pointer to player instead of a pre-decided Player type to allow flexibility of the Game class for Players of any kind.
3. Player\* m\_North – a pointer to a Player object for the North side of m\_Board.
4. Side m\_currSide – an instance of the Side enum which allows tracking which side is currently playing. This will in particular assist Game::display() (to print m\_Board with an appropriate message every time as long as the game is going on) as well as the Game::move() function (to allow making moves from the appropriate side).

Next, we undertake a discussion of various member functions of this class –

1. **Game::Game(const Board& b, Player\* south, Player\* north)** – We construct the various private data members of the Game class by assigning b to m\_Board, south to m\_South, north to m\_North, and m\_currSide to Side::SOUTH (as the player on the South must always make the first move).
2. **void Game::display() const** – this function prints m\_Board in its current state. One of the following three appropriate actions might be made –
3. If the game isn’t yet over, the current state of m\_Board is printed along with a message notifying whose turn it is next.
4. If the game is over, the final state of m\_Board (with zero beans in each hole but non-zero beans in each pot) is displayed, followed by a message saying that the game is over. The task of deciding the winner is left to the Game::play() function.

The display function is generalized to appropriately display the board for any number of holes (not just 6). A typical board display will contain the North player’s name centered on the top, followed by a centered singly-spaced display of the North side’s holes each with its own respective bean count, followed by a balanced display of North’s pot on the left and South’s pot on the right with their respective bean counts, followed by a centered display of the South player’s name on the bottom of the board. This display of the board is followed by a message saying who is to play next/ the game is over.

1. **void Game::status(bool& over, bool& hasWinner, Side& winner)** – this assesses the private data member m\_Board for its current state as follows –
2. It checks whether the beans in play on both sides of m\_Board are zero (using the Board::beansInPlay() function), and if they aren’t, it sets over to false and leaves the rest of the two data members unaffected. However, if the Board::beansInPlay() function called for both sides returns 0, then it does the following.
3. It compares the bean count in both pots to see if they are equal, and if they are, it sets hasWinner to false (since a tie means no winner) and leaves winner unaffected. However if this isn’t the case, it does the following.
4. Since the game is over and it isn’t a tie, there must be a decisive winner, which it sets winner to by comparing the bean count in each pot using the Board::beans() function.
5. **bool Game::move()** – this function appropriately updates m\_Board depending on m\_currSide (that is whose move it is). We first check if the game is over using Game::status() function. If it is, we return false to break out of the function, but if it isn’t, we go ahead and make a move/ multiple moves, returning true. All the moves are made inside a while loop which only terminates when the game is over/ the current Player’s turn has ended. To do so, we call Board::sow() on the opponent’s side at an appropriate hole which will be provided by the Player’s chooseMove() function. Since Board::sow()doesn’t deal with what must be done if there is a capture or the game ends or if the Player ends up in their own pot, we deal with appropriately with these issues after calling Board::sow() (the pseudo-code provided for this function at a later section will give a better idea about how this is done). In essence, if the game is already over, false will be returned since a move cannot possibly moved to alter the board in any way, but if the game isn’t over, m\_Board will be updated accordingly according to one/ multiple moves that were made by the current player. The responsibility of informing the user that the current player received an additional move (if they did) is taken care of by the move() function itself by calling display() whenever the player ended in their own pot.
6. **void Game::play()** – after printing a welcome message, while the game isn’t yet over (which can be check by calling Game::status()), this function displays the board (by calling Game::display()), and then calls Game::move(), which appropriately modifies m\_Board according to one/ multiple moves which the Player displayed at the end of the interface printed by the Game::display() function may have printed (note – this player is to make the next move to affect the current state of m\_Board which was printed). Display for multiple moves are handled within Game::move() itself. The loop is broken out of when the game is over, and using Game::status() a message for who the winner is may be printed.
7. **int Game::beans(Side s, int hole) const** – returns the number of beans in Side s’s hole by using public member functions for the private data member m\_Board (which couldn’t be otherwise accessed by a public point of view since m\_Board is a private data member). This function will be of greatest utility for writing test cases for the Game class.

# **Further description of the SmartPlayer Class**

The function SmartPlayer::chooseMove() returns the best possible hole to make a move on the SmartPlayer’s side. The SmartPlayer::chooseMove() function utilizes the function SmartPlayer::decide() a private helper function to actually obtain the best hole. The heuristics used to evaluate board positions (specifically the leaf positions) is provided by the SmartPlayer::evaluate() private member function, while the values for any nodes in the decision tree above the leaves is determined by the minimax algorithm, which is implemented by the SmartPlayer::decide() function. Thus, we will first describe the chosen heuristics for implementing the SmartPlayer::evaluate(), followed by a discussion for SmartPlayer::decide(), and conclude by saying how SmartPlayer::chooseMove() combines these functionalities to return the best hole on the SmartPlayer’s side.

1. **int SmartPlayer::evaluate(const Board& b) const** – this function houses the heuristics chosen for determining value of leaves in the decision tree, and these heuristics are as follows –
2. If the game isn’t yet over, return the difference in the number of beans in South’s pot and North’s pot.
3. If the game is over, and South won, return 999 (a highly positive number indicating that South won, since the minimax algorithm advocates that the more positive the value, the better it is for South).
4. If the game is over, and North won, return -999 (a highly negative number indicating that North won, since the minimax algorithm advocates that the more negative the value, the better it is for North).
5. If the game is over and it was a tie, return 0.

All these conditions can be checked by utilizing Board::beans() function on m\_Board.

1. **int SmartPlayer::decide(const Board& b, Side s, int& value, int depth, AlarmClock& a)** – this is a recursive function which returns the best hole to make a move on for a Board b and Side s. Its decisions are constraint to whether or not the AlarmClock a has timed out as well as whether or not the recursive function has hit the max depth, whichever of these two conditions is exhausted first. Its working can be separated into two major sections –
2. Base cases – if the game is over for the Board b or the AlarmClock a has timed out or the max depth has been reached, simply set value to evaluate(b) (as the present layer of search must be the leaf layer since we go no further inside the tree), and returns -1 as the best hole value (as no more moves can be made further).
3. General cases – one starts traversing the Side s of the Board b looking for all available non-empty holes (on which Board::sow() can be called). However, to analyze the effects of what move made along a certain hole might result in, we must not make any changes to the Board b itself. So for every non-empty hole that we discover, we create a copy of the Board to analyze what the effects of making that particular move would be. Thus, we call Board::sow() on the copy of b, say tempboard that we created. After having called Board::sow() however, we must cover any possibilities of further movement of beans due to a capture/ the game ending. After having done so, if you know that the game is over, and so in the current Player’s favor, set value to the appropriate victory value and return this particular hole, since the Player cannot have a better choice. However, if the game isn’t over after sowing and doing whatever else is required, there are two possibilities –
4. It is the current Player’s turn again due to landing in their own pot, for which we call this function recursively at the same depth for the current Player itself.
5. The current Player’s turn ended, so that we can now call the function recursively for the opponent at a depth one greater than the present (since the minimax algorithm assumes that the opponent will also play as optimally as the SmartPlayer).

By setting up temporary variables which store the best value and the best hole encountered so far, we can check whether the newly affected state of value is better than its previous ones. The definition of best can vary depending on whether the side currently playing is South or North. In any case, after having looped through the side s of Board b, we should have one of the candidate non-empty holes being stored in the temporary variable which was created to keep track of the best hole, as well as the best value corresponding to this best hole stored in the temporary variable created to keep track of the best values. We can ultimately set value to bestvalue (the temporary variable which now holds the best case of value obtained from recursive searches) and return besthole (which is the best hole encountered in b on Side s). Note that while recursively calling the function we aren’t actually concerned with the return value of the recursive call (which is the hole number), but rather the value corresponding to a particular node (which gets stored in the value parameter for the function call, since the former is passed by reference). Thus while working recursively, we care only about value, and when we are about to finally exit the functions top level routine, we care about the best hole that is being returned (since we must inform the caller of this function, SmartPlayer::chooseMove() about this hole).

1. **virtual int SmartPlayer::chooseMove(const Board& b, Side s) const** – chooseMove() for SmartPlayer simply sets up an AlarmClock object ac, initializing it to a value of 4900 ms, to give it 100 ms to wrap up, and then returns whatever value SmartPlayer::decide() has to return for the parameter set (b, s, value, 0, ac), where value is a random uninitialized integer and 0 is the beginning depth of the recursive search. The result returned by this function will be the best hole that could possibly be chosen given the chosen constraints for the algorithm to be made on SmartPlayer’s side. Note that the efficiency of this function is decided by both a terminal time and depth condition, since that is how its helper function’s efficiency is decided.

# **Pseudo-code for non-trivial algorithms**

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

*if the move is invalid*

*return false*

*else*

*if beginning side is South*

*keep sowing until South’s last hole, and then pot is reached or beans are exhausted*

*if beans are left over*

*until beans remain*

*sow in North from last hole to first hole*

*if beans are left, until they remain*

*sow from South’s first hole to last hole to pot*

*if beans are still left*

*begin again from North’s last hole*

*after beans are exhausted, store end side and end hole and return true*

*if beginning side is North*

*similar approach for beginning from North*

SmartPlayer::decide(const Board& b, Side s, int& value, int depth, AlarmClock& a)

*if the game is over or the AlarmClock has timed out*

*set value to evaluate called on the current board*

*return -1*

*if the game is not over*

*for every non-empty hole found on the current side*

*create a copy of b and call sow over it*

*further shift beans due to a capture/ the game ending*

*if you end on your own side’s pot*

*call decide for yourself at same depth as before*

*if not so*

*call decide for the opponent at a depth one greater than the previous*

*check if value has attained its best so far*

*if yes, set best hole found so far to the current hole*

*set value to be the best value detected in the current side’s traversal*

*return corresponding best hole*

bool Game::move()

*if the game is over*

*return false*

*until it’s the current player’s turn*

*sow using Player’s chooseMove()*

*if a capture is possible, do so*

*if the game got over shift beans on board to their respective pots*

*if you didn’t end up in your pot*

*inform the user that it’s now the opponent’s move and return true*

*if you ended up in your own pot*

*take another chance by reiterating*

# **Test Cases**

Test cases for Board Class

*// test cases already provided by the spec*

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);

*// testing for major illegal behavior*

Board testnull(0, -1); *// bad inputs for constructor parameters are handled as specified in spec.*

*// 0*

*// 0 0*

*// 0*

assert(testnull.holes() == 1); *// non-positive values for holes argument are treated as a 1.*

assert(testnull.totalBeans() == 0); *// as negative values for initial beans per hole are treated as 0.*

assert(testnull.beans(Side::SOUTH, 20) == -1); *// cannot access beans from a non-existent hole. An illegal call to Board::beans() returns -1.*

assert(!testnull.setBeans(Side::SOUTH, 20, 2)); *// illegal calls to Board::setBeans() will return false.*

*// testing for legal behavior (and some illegal behavior that might occur on the way)*

Board testb1(6, 2); *// constructing a usual board.*

*// 2 2 2 2 2 2*

*// 0 0*

*// 2 2 2 2 2 2*

assert(testb1.holes() == 6); *// the Board::holes() function works appropriately. As specified in the constructor, this board has 6 holes on each side.*

assert(testb1.totalBeans() == 24); *// the Board::totalBeans() function works appropriately. The total number of beans on the board should remain constant throughout the game and should be equal to twice of number of holes times the number of beans supplied to each hole initially. Note however that the total number of beans can change if the user uses Board::setBeans() anytime.*

*// the Board::beansInPlay() function works appropriately. To begin with, each side should have the number of total beans in play (note that this quantity doesn't count the number of beans in the respective pots of each side, which is also zero to begin with).*

assert(testb1.beansInPlay(Side::SOUTH) == testb1.totalBeans()/2);

assert(testb1.beansInPlay(Side::NORTH) == testb1.totalBeans()/2);

**for**(**int** i=1; i<=6; i++)

{

*// the Board::beans() funciton works appropriately. Each hole along each side should have 2 beans.*

assert(testb1.beans(Side::SOUTH, i) == 2);

assert(testb1.beans(Side::NORTH, i) == 2);

}

*// each pot should have 0 beans to begin with*

assert(testb1.beans(Side::SOUTH, 0) == 0);

assert(testb1.beans(Side::NORTH, 0) == 0);

*// sowing hole number 5 along South, which is a legal move since the number of beans in hole 5 for South is non-zero. The end side for this move should be South and the end hole should be 0 (South's pot). The movement of beans should always be in the counter-clockwise direction.*

Side endSide;

**int** endHole;

assert(testb1.sow(Side::SOUTH, 5, endSide, endHole));

*// 2 2 2 2 2 2*

*// 0 1*

*// 2 2 2 2 0 3*

assert(endSide == Side::SOUTH);

assert(endHole == 0);

assert(testb1.beans(Side::SOUTH, 5) == 0); *// as all beans from this hole were sowed.*

assert(testb1.beansInPlay(Side::SOUTH) == 11); *// the number of beans in South's holes is now one less than before since one of the beans on South's side ended up in its pot.*

assert(testb1.beans(Side::SOUTH, 6) == 3); *// as the hole 6 in South will now have an additional bean.*

assert(testb1.beans(Side::SOUTH, 0) == 1); *// the South side's pot now has one bean.*

assert(testb1.beans(Side::SOUTH, 4) == 2); *// anything to the left of hole 5 on South wasn't affected, since movement of beans is always counter-clockwise.*

assert(!testb1.sow(Side::SOUTH, 5, endSide, endHole)); *// hole number 5 for South is empty, so cannot be used to make a move.*

assert(testb1.sow(Side::SOUTH, 6, endSide, endHole)); *// a move should be legal from South's 6th hole, and the endSide and endHole should respectively be North and 5. The rotation of beans will be counter-clockwise once again.*

*// 2 2 2 2 3 3*

*// 0 2*

*// 2 2 2 2 0 0*

assert(endSide == Side::NORTH);

assert(endHole == 5);

assert(testb1.beans(Side::SOUTH, 6) == 0); *// as all beans from this hole were sowed.*

assert(testb1.beans(Side::SOUTH, 0) == 2); *// South's pot gained another bean.*

assert(testb1.beans(Side::NORTH, 6) == 3); *// North's 6th hole gained another bean.*

assert(testb1.beans(Side::NORTH, 5) == 3); *// North's 5th hole gained another bean as well.*

assert(!testb1.moveToPot(Side::SOUTH, 0, Side::NORTH)); *// the Board::moveToPot() function is working appropriately. Cannot move from one pot to another.*

*// 2 2 2 2 3 3*

*// 2 2*

*// 0 2 2 2 0 0*

assert(testb1.moveToPot(Side::SOUTH, 1, Side::NORTH)); *// Moving over all beans in South's 1st hole to North's pot.*

assert(testb1.beans(Side::SOUTH, 1) == 0); *// the first hole of South side now has 0 beans.*

assert(testb1.beans(Side::NORTH, 0) == 2); *// as North's pot received these beans.*

*// South can sow its beans in its own pot but not in North's.*

assert(testb1.setBeans(Side::SOUTH, 6, 8)); *// setting the number of beans in South's 6th hole to be 8 to illustrate our point here.*

*// 2 2 2 2 3 3*

*// 2 2*

*// 0 2 2 2 0 8*

assert(testb1.beans(Side::SOUTH, 6) == 8); *// Board::setBeans() function is working appropriately.*

assert(testb1.sow(Side::SOUTH, 6, endSide, endHole)); *// sowing all of the beans in South's 6th hole. The endSide and endHole should respectively be Side::SOUTH and 1, as North's pot was skipped along the way.*

*// 3 3 3 3 4 4*

*// 2 3*

*// 1 2 2 2 0 0*

assert(endSide == Side::SOUTH);

assert(endHole == 1);

assert(testb1.beans(Side::SOUTH, 6) == 0); *// as all the beans in this hole were sowed.*

assert(testb1.beans(Side::SOUTH, 0) == 3); *// as South's pot gained another bean (it previously had 2).*

assert(testb1.beans(Side::NORTH, 6) == 4); *// North's 6th hole gains another bean (it had 3 before).*

assert(testb1.beans(Side::NORTH, 5) == 4); *// North's 5th hole gains another bean (it ahd 3 before).*

**for**(**int** i=4; i>=1; i--)

assert(testb1.beans(Side::NORTH, i) == 3); *// all of North's rest of the holes gain a bean each (they had 2 before).*

assert(testb1.beans(Side::NORTH, 0) == 2); *// the number of beans in North's pot isn't affected as it was skipped (it previously had 2, and has the same even now).*

assert(testb1.beans(Side::SOUTH, 1) == 1); *// South's 1st hole, where we end, gains another bean (it had 0 before as its beans were moved over to North's pot).*

assert(testb1.setBeans(Side::NORTH, 1, 8)); *// a move from North can sow in its own pot but not in South's pot. This example will illustrate the point.*

*// 8 3 3 3 4 4*

*// 2 3*

*// 1 2 2 2 0 0*

assert(testb1.sow(Side::NORTH, 1, endSide, endHole)); *// for this case, the endSide and endHole should respectively be North and 6, as South's pot was skipped over.*

*// 0 3 3 3 4 5*

*// 3 3*

*// 2 3 3 3 1 1*

assert(testb1.beans(Side::NORTH, 1) == 0); *// as all the beans in this hole were sowed over.*

assert(endSide == Side::NORTH);

assert(endHole == 6);

assert(testb1.beans(Side::NORTH, 0) == 3); *// North's pot gains a bean (it previously had 2).*

assert(testb1.beans(Side::SOUTH, 1) == 2); *// South's first hole gains a bean (it previously had 1).*

**for**(**int** i=2; i<=4; i++)

assert(testb1.beans(Side::SOUTH, i) == 3); *// South's second to fourth hole gaina a bean (they previously had 2).*

assert(testb1.beans(Side::SOUTH, 5) == 1); *// South's 5th hole has 1 bean now (it previously had 0 because we sowed from it).*

assert(testb1.beans(Side::SOUTH, 6) == 1); *// South's 6th hole has 1 bean now (it previously had 0 beacuse we sowed from it).*

assert(testb1.beans(Side::SOUTH, 0) == 3); *// South's pot is unaffected, and continues to have 3 beans.*

assert(testb1.beans(Side::NORTH, 6) == 5); *// North's 6th hole gains another bean (it previously had 4).*

*// This is how the board should currently look*

*// 0 3 3 3 4 5*

*// 3 3*

*// 2 3 3 3 1 1*

*// let's test to see if this is true, moving counter-clockwise from South's first hole.*

assert(testb1.beans(Side::SOUTH, 1) == 2);

assert(testb1.beans(Side::SOUTH, 2) == 3);

assert(testb1.beans(Side::SOUTH, 3) == 3);

assert(testb1.beans(Side::SOUTH, 4) == 3);

assert(testb1.beans(Side::SOUTH, 5) == 1);

assert(testb1.beans(Side::SOUTH, 6) == 1);

assert(testb1.beans(Side::SOUTH, 0) == 3);

assert(testb1.beans(Side::NORTH, 6) == 5);

assert(testb1.beans(Side::NORTH, 5) == 4);

assert(testb1.beans(Side::NORTH, 4) == 3);

assert(testb1.beans(Side::NORTH, 3) == 3);

assert(testb1.beans(Side::NORTH, 2) == 3);

assert(testb1.beans(Side::NORTH, 1) == 0);

assert(testb1.beans(Side::NORTH, 0) == 3);

Test cases for Player Class (and its derivatives)

*// test cases already provided by the spec.*

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);

*// 2 2 2*

*// 0 0*

*// 2 0 2*

std::cout << "Please enter a move for a HumanPlayer playing on the South side represented by {2, 0, 2}" << std::endl;

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

assert(n == 1 || n == 3); *// must be a legal move. In fact, HumanPlayer::chooseMove() will keep demanding an input from the user until they provide a legal one.*

n = bp.chooseMove(b, SOUTH);

assert(n == 1 || n == 3); *// BadPlayer::chooseMove() will return a move which, although arbitrary and not well-thought of, will still be legal (i.e. not from an empty hole or a non-existent hole).*

n = sp.chooseMove(b, SOUTH); *// SmartPlayer::chooseMove() will return a move which is both well-thought of and also legal (i.e. not from an empty hole or a non-existent hole).*

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

*// further test cases.*

HumanPlayer testhuman("HumanTester");

BadPlayer testbad("BadTester");

SmartPlayer testsmart("SmartTester");

assert(testhuman.name() == "HumanTester" && testhuman.isInteractive()); *// a human player is interactive.*

assert(testbad.name() == "BadTester" && !testbad.isInteractive()); *// a bad player is not interactive.*

assert(testsmart.name() == "SmartTester" && !testsmart.isInteractive()); *// a smart player is not interactive.*

Board testb2(5, 3);

*// 3 3 3 3 3*

*// 0 0*

*// 3 3 3 3 3*

assert(testb2.setBeans(Side::SOUTH, 1, 0));

assert(testb2.setBeans(Side::SOUTH, 2, 1));

assert(testb2.setBeans(Side::SOUTH, 4, 1));

assert(testb2.setBeans(Side::SOUTH, 5, 0));

assert(testb2.setBeans(Side::NORTH, 1, 0));

assert(testb2.setBeans(Side::NORTH, 2, 0));

assert(testb2.setBeans(Side::NORTH, 3, 0));

*// 0 0 0 3 3*

*// 0 0*

*// 0 1 3 1 0*

std::cout << "Please enter a move for a HumanPlayer playing on the South side represented by {0, 3, 3, 3, 0}" << std::endl;

**int** testhole = testhuman.chooseMove(testb2, Side::SOUTH);

assert(testhole == 2 || testhole == 3 || testhole == 4); *// the way HumanPlayer::chooseMove() is designed, the chosen move by the human will ultimately have to be a legal one.*

testhole = testbad.chooseMove(testb2, Side::SOUTH); *// a BadPlayer will always choose the a non-empty hole which is equal to or closest to 1 on its side. In this case, it will, this move will be 2.*

assert(testhole == 2);

testhole = testsmart.chooseMove(testb2, Side::SOUTH); *// a SmartPlayer will choose a move which is the best possible in a given situation. Of course, this move will also be a legal one. In this case, it chooses to move along hole 3, which results in it ending in its own pot, giving it another chance to play.*

assert(testhole == 3);

Test cases for Game Class

*// test cases already provided by 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** over;

**bool** hasWinner;

Side winner;

*// Homer*

*// 0 1 2*

*// 0 0*

*// 2 0 0*

*// Bart*

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();

*// 0 1 0*

*// 0 3*

*// 0 1 0*

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();

*// 1 0 0*

*// 0 3*

*// 0 1 0*

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();

*// 1 0 0*

*// 0 3*

*// 0 0 1*

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();

*// 0 0 0*

*// 1 4*

*// 0 0 0*

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);

*// further test cases*

*// here is a mid-game situation where SmartPlayer is expected to win at once using a series of multiple moves according to the spec. To test this hypothesis, let's play a SmartPlayer against another SmartPlayer.*

*// any player*

*// 1 0 0 0 2 0*

*// 22 20*

*// 0 0 0 1 2 0*

*// Smart*

Board testb3(6, 0);

assert(testb3.setBeans(Side::SOUTH, 4, 1));

assert(testb3.setBeans(Side::SOUTH, 5, 2));

assert(testb3.setBeans(Side::SOUTH, 0, 20));

assert(testb3.setBeans(Side::NORTH, 5, 2));

assert(testb3.setBeans(Side::NORTH, 1, 1));

assert(testb3.setBeans(Side::NORTH, 0, 22));

SmartPlayer testSouth("SouthSmart");

SmartPlayer testNorth("NorthSmart");

Game testgame(testb3, &testSouth, &testNorth);

assert(testgame.beans(Side::SOUTH, 1) == 0);

assert(testgame.beans(Side::SOUTH, 2) == 0);

assert(testgame.beans(Side::SOUTH, 3) == 0);

assert(testgame.beans(Side::SOUTH, 4) == 1);

assert(testgame.beans(Side::SOUTH, 5) == 2);

assert(testgame.beans(Side::SOUTH, 6) == 0);

assert(testgame.beans(Side::SOUTH, 0) == 20);

assert(testgame.beans(Side::NORTH, 6) == 0);

assert(testgame.beans(Side::NORTH, 5) == 2);

assert(testgame.beans(Side::NORTH, 4) == 0);

assert(testgame.beans(Side::NORTH, 3) == 0);

assert(testgame.beans(Side::NORTH, 2) == 0);

assert(testgame.beans(Side::NORTH, 1) == 1);

assert(testgame.beans(Side::NORTH, 0) == 22);

testgame.move();

*// any player*

*// 0 0 0 0 0 0*

*// 23 25*

*// 0 0 0 0 0 0*

*// Smart*

**for**(**int** i=1; i<=6; i++)

{

assert(testgame.beans(Side::SOUTH, i) == 0);

assert(testgame.beans(Side::NORTH, i) == 0);

}

assert(testgame.beans(Side::SOUTH, 0) == 25);

assert(testgame.beans(Side::NORTH, 0) == 23);

testgame.status(over, hasWinner, winner);

assert(over && hasWinner && winner == Side::SOUTH);

*// thus, the SmartPlayer on the South side is winning this game, as was required by the spec. We can further test out a few games.*

*// Game::display() function is working appropriately since in between multiple moves, the displaying for another move is a job of Game::move(), and it happened and should be visible in the console.*

*// some games to play*

std::cout << "BadPlayer vs. BadPlayer!" << std::endl << std::endl;

Board gameboard(6, 4);

BadPlayer bad1("Bad1");

BadPlayer bad2("Bad2");

Game gbb(gameboard, &bad1, &bad2);

gbb.play();

std::cout << "SmartPlayer (South) vs. BadPlayer (North)!" << std::endl << std::endl;

SmartPlayer smart1("Smart1");

Game gsb(gameboard, &smart1, &bad1);

gsb.play();

std::cout << "SmartPlayer (North) vs. BadPlayer(South)!" << std::endl << std::endl;

Game gbs(gameboard, &bad1,&smart1);

gbs.play();

std::cout << "SmartPlayer vs. SmartPlayer!" << std::endl << std::endl;

SmartPlayer smart2("Smart2");

Game gss(gameboard, &smart1, &smart2);

gss.play();

std::cout << "SmartPlayer (South) vs. HumanPlayer (North)!" << std::endl << std::endl;

HumanPlayer human("Human");

Game gsh(gameboard, &smart1, &human);

gsh.play();

std::cout << "HumanPlayer (North) vs. SmartPlayer (South)!" << std::endl << std::endl;

Game ghs(gameboard, &human, &smart1);

ghs.play();

Here is the output for one of the above-mentioned games which proves that Smart1 can play quite intelligently –

**SmartPlayer (North) vs. BadPlayer(South)!**

**Welcome to Kalah!**

**Smart1**

**4 4 4 4 4 4**

**0 0**

**4 4 4 4 4 4**

**Bad1**

**It is Bad1's turn!**

**Press enter to continue.**

**Smart1**

**4 4 4 4 4 4**

**0 0**

**0 5 5 5 5 4**

**Bad1**

**It is Smart1's turn!**

**Press enter to continue.**

**Smart1**

**5 5 5 0 4 4**

**1 0**

**0 5 5 5 5 4**

**Bad1**

**It is Smart1's turn!**

**Smart1**

**0 5 5 0 4 4**

**2 0**

**1 6 6 6 5 4**

**Bad1**

**It is Bad1's turn!**

**Press enter to continue.**

**Smart1**

**0 5 5 0 4 4**

**2 0**

**0 7 6 6 5 4**

**Bad1**

**It is Smart1's turn!**

**Press enter to continue.**

**Smart1**

**1 0 5 0 4 4**

**3 0**

**1 8 7 6 5 4**

**Bad1**

**It is Bad1's turn!**

**Press enter to continue.**

**Smart1**

**1 0 5 0 4 4**

**3 0**

**0 9 7 6 5 4**

**Bad1**

**It is Smart1's turn!**

**Press enter to continue.**

**Smart1**

**0 0 5 0 4 4**

**4 0**

**0 9 7 6 5 4**

**Bad1**

**It is Smart1's turn!**

**Smart1**

**1 1 0 0 4 4**

**5 0**

**1 10 7 6 5 4**

**Bad1**

**It is Bad1's turn!**

**Press enter to continue.**

**Smart1**

**1 1 0 0 4 4**

**5 0**

**0 11 7 6 5 4**

**Bad1**

**It is Smart1's turn!**

**Press enter to continue.**

**Smart1**

**0 1 0 0 4 4**

**6 0**

**0 11 7 6 5 4**

**Bad1**

**It is Smart1's turn!**

**Smart1**

**1 0 0 0 4 4**

**6 0**

**0 11 7 6 5 4**

**Bad1**

**It is Bad1's turn!**

**Press enter to continue.**

**Smart1**

**2 1 1 1 5 5**

**6 1**

**0 0 8 7 6 5**

**Bad1**

**It is Smart1's turn!**

**Press enter to continue.**

**Smart1**

**3 2 2 2 0 5**

**7 1**

**0 0 8 7 6 5**

**Bad1**

**It is Smart1's turn!**

**Smart1**

**4 0 2 2 0 5**

**8 1**

**0 0 8 7 6 5**

**Bad1**

**It is Smart1's turn!**

**Smart1**

**5 1 0 2 0 5**

**8 1**

**0 0 8 7 6 5**

**Bad1**

**It is Bad1's turn!**

**Press enter to continue.**

**Smart1**

**5 1 1 3 1 6**

**8 2**

**0 0 0 8 7 6**

**Bad1**

**It is Smart1's turn!**

**Press enter to continue.**

**Smart1**

**6 2 2 4 2 0**

**9 2**

**0 0 0 8 7 6**

**Bad1**

**It is Smart1's turn!**

**Smart1**

**7 0 2 4 2 0**

**10 2**

**0 0 0 8 7 6**

**Bad1**

**It is Smart1's turn!**

**Smart1**

**8 1 3 0 2 0**

**11 2**

**0 0 0 8 7 6**

**Bad1**

**It is Smart1's turn!**

**Smart1**

**9 2 0 0 2 0**

**12 2**

**0 0 0 8 7 6**

**Bad1**

**It is Smart1's turn!**

**Smart1**

**10 0 0 0 2 0**

**13 2**

**0 0 0 8 7 6**

**Bad1**

**It is Smart1's turn!**

**Smart1**

**0 0 0 0 3 1**

**24 2**

**1 1 1 0 8 7**

**Bad1**

**It is Bad1's turn!**

**Press enter to continue.**

**Smart1**

**0 0 0 0 3 1**

**24 2**

**0 2 1 0 8 7**

**Bad1**

**It is Smart1's turn!**

**Press enter to continue.**

**Smart1**

**0 0 1 1 0 1**

**27 2**

**0 0 1 0 8 7**

**Bad1**

**It is Bad1's turn!**

**Press enter to continue.**

**Smart1**

**0 0 1 0 0 1**

**27 4**

**0 0 0 0 8 7**

**Bad1**

**It is Smart1's turn!**

**Press enter to continue.**

**Smart1**

**0 0 1 0 0 0**

**36 4**

**0 0 0 0 0 7**

**Bad1**

**It is Bad1's turn!**

**Press enter to continue.**

**Smart1**

**0 0 0 0 0 0**

**43 5**

**0 0 0 0 0 0**

**Bad1**

**Game Over!**

**Press enter to continue.**

**Smart1 won!**