Design of Classes:

Scaffold - I used a 2D vector to represent the grid of the Connect N game, with each nested vector being a row of the grid. This 2D vector is utilized in checkerAt() to determine what is in each spot of the grid, in display() to print the correct grid, and in makeMove() to reflect a move in the scaffold. I also used a stack that held a vector (the size of the vector is always two in this case) that contained the column and level of the last made move. This was necessary to implement undoMove(), as the most recently added vector to the stack would contain the coordinates of the last move made.

Player - I did not add any private members to this class, and implemented all but SmartPlayer::chooseMove() trivially and without non-member methods or data members. The sole exception will be described in a section below.

Game - For this class, I passed in two players as private data members using pointers, and created a Scaffold object to be kept track of as the game progresses. The completed() function scoured the Scaffold object to determine whether the game was over and someone had won, the takeTurn() function called on each player depending on the turn and executed a move, and the play() function looped between these two functions until the game was over, displaying the grid after each turn.

Design of SmartPlayer::chooseMove:

I designed this function primarily using two helper functions that were not members of the Player class, evaluate and determineBestMove. The evaluate function works in a similar way to the completed() function in the Game class, though it not only checks a win for the player of the last move, but also does so for the other player, giving it a possible return value of a win, a tie, or a loss for the player passed in. Evaluate does this by running through each of the four directions (horizontal, vertical, diagonal, and other diagonal) N distance away from the last move, thus the last move resides in the middle of this check, looking for connect N for both colors. By using this function to rate the state of a given board, determineBestMove recursively calls itself using the minimax algorithm until it reaches down the tree to a board state that can be evaluated. This evaluation is stored in a vector, and the corresponding move that leads to that score is kept in a map. For each level of the tree, determineBestMove picks either the maximum or minimum score (the former for the player’s moves and the latter for the opponent’s moves) from the vector, and returns a vector with the first element being the score and the second being the corresponding move, ultimately returning the optimal move at the initial node. I did not need to implement two determineBestMove functions, one for the computer and one for the human, as for a game like connect N, the score of your opponent is the direct opposite of your score (Negamax). One other wrinkle that chooseMove() implements is an alarmClock class that limits the recursion of determineBestMove to 10 seconds. If that time elapses, determinebestMove stops at whatever level down the tree it has reached and immediately returns the optimal score and move that it has discovered so far, or a vector containing a tie score and the first column if no board state has been evaluated.

Pseudocode:

* Scaffold constructor
  + If dimensions are non-positive
    - Exit
  + Set dimensions
  + Initialize number of checkers
  + Initialize grid to vacant spaces
* Scaffold display
  + Print out checkers with “|” in between
  + Print out bottom row of “+” and “-”
* Scaffold makeMove
  + If open space in selected column exists
    - Set position to the color of the player
    - Increment number of checkers
    - Push coordinates of the move onto move stack
    - Return true
  + Otherwise return false
* Scaffold undoMove
  + If move has not been made
    - Return 0
  + Pop last move from stack
  + Set status of move coordinates to vacant
  + Decrement number of checkers
  + Return column of last move
* Game constructor
  + If N is not valid (i.e. larger than the board)
    - Exit
  + Initialize scaffold
  + Set turn to red
  + Set N and lastcolumn to respective inputs
  + Set player to respective pointers
* Game completed
  + Find level of last move
  + Loop: If checker on the horizontal containing the last move in the middle is the same color as the last move
    - Increment number of checkers in a row
    - If number of checkers in a row needed to win is achieved
      * Set winner to color and return true
  + Else reset number of checkers in a row
  + Loop: If checker on the vertical containing the last move in the middle is the same color as the last move
    - Increment number of checkers in a row
    - If number of checkers in a row needed to win is achieved
      * Set winner to color and return true
  + Else reset number of checkers in a row
  + Loop: If checker on the diagonal containing the last move in the middle is the same color as the last move
    - Increment number of checkers in a row
    - If number of checkers in a row needed to win is achieved
      * Set winner to color and return true
  + Else reset number of checkers in a row
  + Loop: If checker on the other diagonal containing the last move in the middle is the same color as the last move
    - Increment number of checkers in a row
    - If number of checkers in a row needed to win is achieved
      * Set winner to color and return true
  + Else reset number of checkers in a row
  + If no moves remain
    - Set winner to tie game and return true
  + Return false
* Game takeTurn
  + If it is red’s turn
    - Make a move based on the type of player passed in
    - Set lastmove to the move made
    - Change turn to black
    - Return true
  + Otherwise
    - Make a move based on the type of player passed in
    - Set lastmove to the move made
    - Change turn to red
    - Return true
  + Return false
* Game play
  + Loop until game is completed
    - Take a turn
    - Display new board
  + Print the result of the game
* HumanPlayer chooseMove
  + If inputs aren’t valid
    - Exit
  + Loop until valid move is inputed
  + Return move
* BadPlayer chooseMove
  + If inputs aren’t valid
    - Exit
  + Set seed for random number generator
  + Loop until chosen move in valid
    - Pick a random number between 1 and the number of columns
  + Return move
* Evaluate
  + Find level of last move
  + Loop: If checker on the horizontal containing the last move in the middle is not vacant
    - Increment number of checkers in a row for the checker’s color
    - If number of checkers in a row needed to win is achieved for the player
      * Return 1000+number of empty spaces in the grid
    - If number of checkers in a row needed to win is achieved for the opponent
      * Return -(1000+number of empty spaces in the grid)
  + Else reset number of checkers in a row for both colors
  + Loop: If checker on the vertical containing the last move in the middle is not vacant
    - Increment number of checkers in a row for the checker’s color
    - If number of checkers in a row needed to win is achieved for the player
      * Return 1000+number of empty spaces in the grid
    - If number of checkers in a row needed to win is achieved for the opponent
      * Return -(1000+number of empty spaces in the grid)
  + Else reset number of checkers in a row for both colors
  + Loop: If checker on the diagonal containing the last move in the middle is not vacant
    - Increment number of checkers in a row for the checker’s color
    - If number of checkers in a row needed to win is achieved for the player
      * Return 1000+number of empty spaces in the grid
    - If number of checkers in a row needed to win is achieved for the opponent
      * Return -(1000+number of empty spaces in the grid)
  + Else reset number of checkers in a row for both colors
  + Loop: If checker on the other diagonal containing the last move in the middle is not vacant
    - Increment number of checkers in a row for the checker’s color
    - If number of checkers in a row needed to win is achieved for the player
      * Return 1000+number of empty spaces in the grid
    - If number of checkers in a row needed to win is achieved for the opponent
      * Return -(1000+number of empty spaces in the grid)
  + Else reset number of checkers in a row for both colors
  + If no moves remain
    - Return 0
  + Return -10000
* determineBestMove
  + If recursion depth is even set turn variable to player’s color
  + Otherwise set turn variable to opponent’s color
  + If no moves have been made
    - Return the middle column (rounded down)
  + Loop through each possible move
    - Make the move
    - If 10 second limit has not been reached
      * Evaluate the board
      * If board evaluation is not conclusive (returns -10000)
        + Call determineBestMove with opposite color and incremented depth
    - Otherwise
      * Set evaluation to vector containing 0 and i (iteration of the loop)
      * Break out of loop
    - Push evaluation into board rating vector
    - Store corresponding move to the evaluation in a map
    - Undo the move
  + If recursion level is at player’s move
    - Find the maximum evaluation in rating vector
  + If recursion level is at opponent’s move
    - Find the minimum evaluation in rating vector
  + Return a vector with the optimal rating and the corresponding move
* SmartPlayer chooseMove
  + If inputs aren’t valid
    - Exit
  + Initialize alarm clock to 10 seconds
  + Make a copy of the passed in scaffold
  + Determine the best move vector
  + Return only the move (item 2 in the vector)

Other Notes:

* I had difficulty deciding where to declare evaluate and determineBestMove, as I didn’t know if I should put them in a certain class
* After deciding to make the above two functions non-class members, I had to figure out how to implement them in SmartPlayer::chooseMove, as it passed in a const Scaffold
* Determining where to break out of the recursion in determineBestMove when the 10 seconds elapsed was also a difficulty, as I needed to make sure that it still returned a valid vector