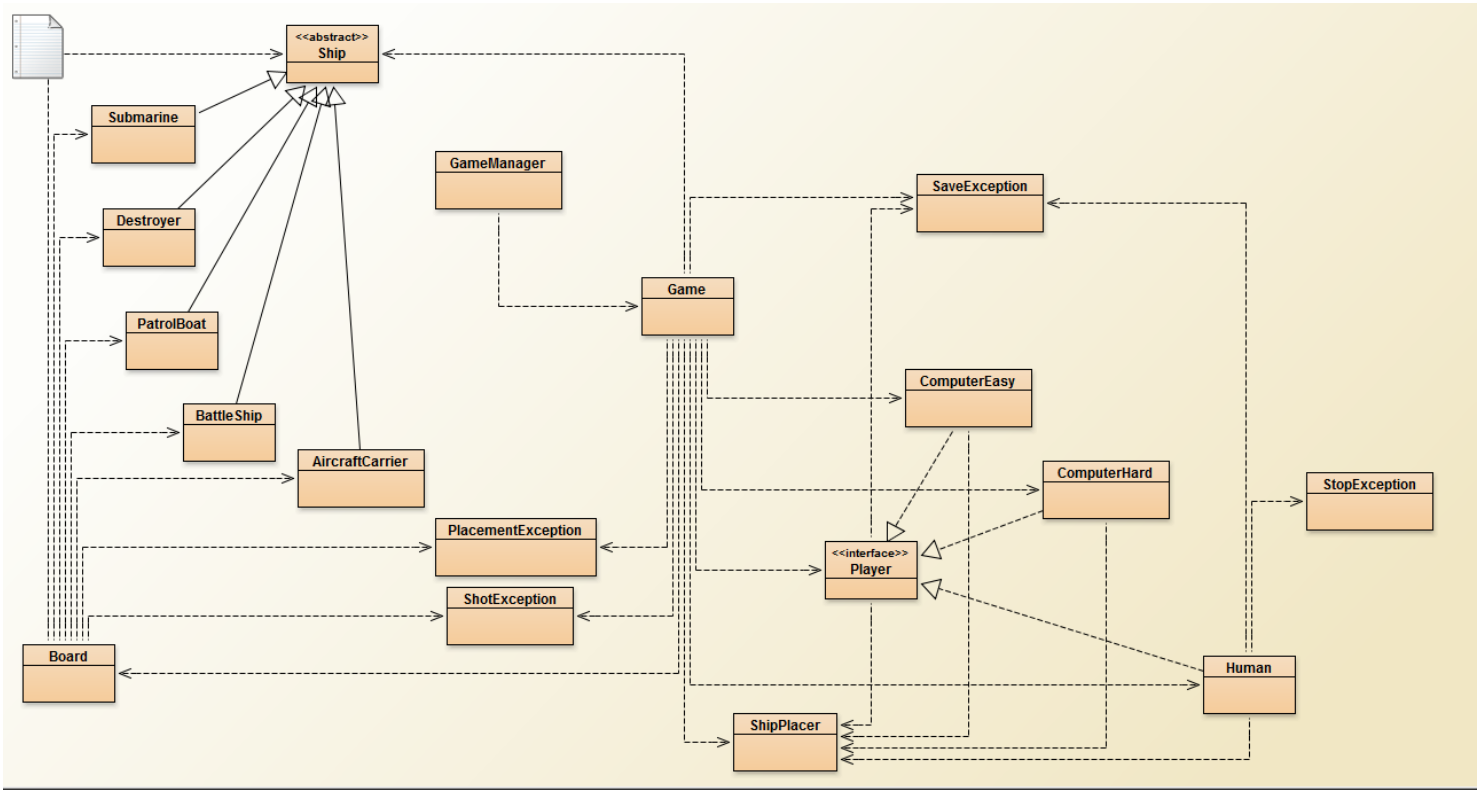
End of Year Coursework: Battleships Game



Board:

The player has a specific board it can access and alter, depending on the game state. Placing ships and firing are just types of Player operations using ships and boards. Therefore a board must contain a set of ships they have in play, a set of ships they do not have in play. Placing and sinking ships is just transferring between these sets. Other game stages could easily be added in later.

This basic functionality is all that is needed from the board, aside from the ability to tell the Game that the player's shot was invalid. Hence, two Exception classes are defined (Shot/Placement). There are two because it is possible that we want to return different things to the player depending on which game stage is throwing exceptions.

The initial capacity inputted could be easily altered for other sized boards, and with this functionality extending through the entire package (e.g. the printboard method in Game could also handle printing of grid references of up to 26).

Ships:

Different types of battleship games often involve different sorts of ship, with different sizes, but have the same sorts of fields and methods. Therefore, subclasses of the ship class were appropriate, not separate classes. Initializing ship objects in the board would not be appropriate because then it would be more difficult for the game to ID ships, and check whether they were on the board. Also some games initialize multiple copies of ships of the same type (Russia), so multiple objects are appropriate.

Game:

I decided to let the Game handle the players entirely, without a 'built-in' computer player. My justification for this is that the operations that the player and the computer make are essentially the same, and their games are the same, so they should not be treated differently, allowing me to easily modularize my Computer classes and human class to involve only the game and the interface, with the capability to choose at the outset who will play who.

I also decided to have the players pass ShipPlacer objects to the game in the placing stage, as this class allows easy maintenance should the ship placement stage be redesigned.

Further to this, the Game completely automates checking the players' moves are valid in each move method: if a player passes it an invalid placement or shot, the custom Exceptions thrown by the board (which do not make use of Array out of bound exceptions, minimizing confusion in maintenance) are contained within the individual method used to make a move. The move method will cycle until a valid shot is made. Then the game progresses (and the board is altered), if and only if a valid shot is made.

I made this decision as the player should not be trusted to validate their own moves. Since an invalid move would cause the method to cycle around again, I did not put a validity check in any of the player classes (Humans or Computer), or return the invalid move exceptions through the Player interface.

Player Interface:

public int[] shoot() throws SaveException;

public void isHit(boolean b);

public ShipPlacer place() throws SaveException;

These correspond to:

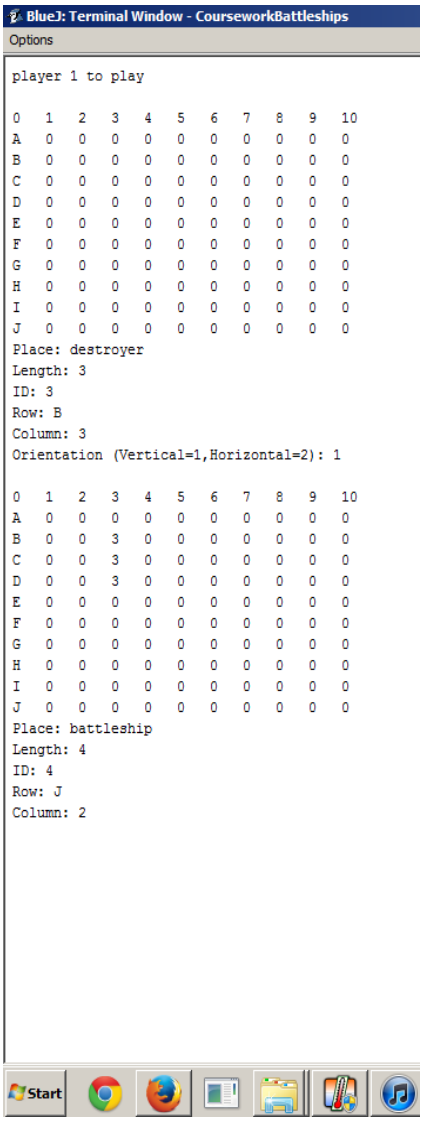
-Player tells the game where to shoot.

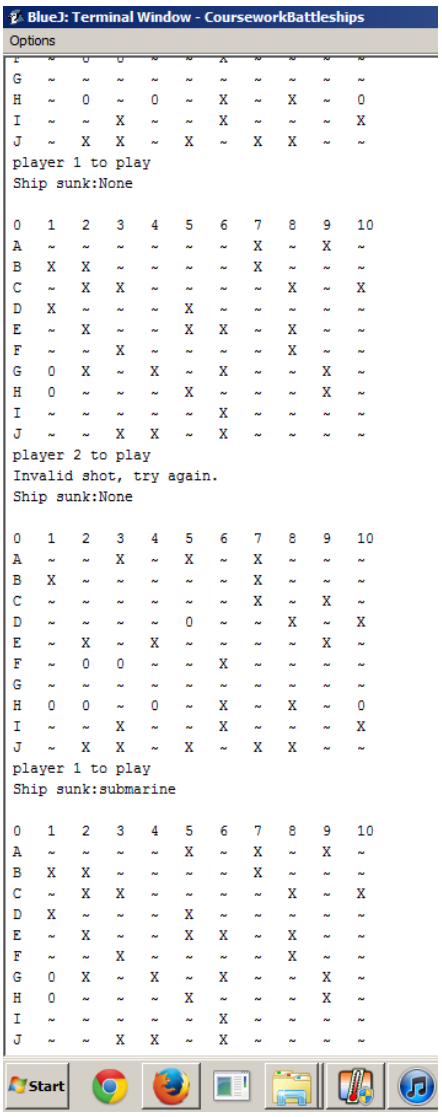
-Game tells player if the shot was a hit.

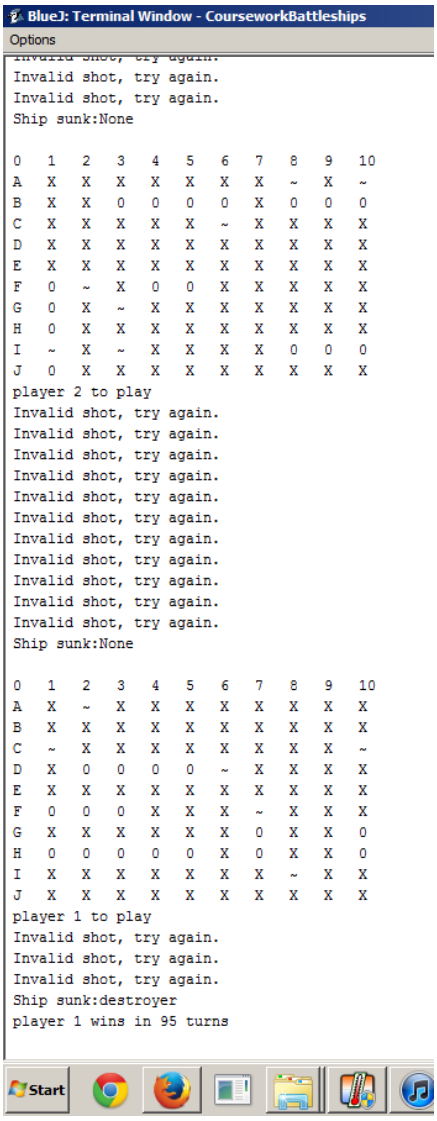
-Player tells the game where and what orientation to place a ship in.

The game class handles all the printing, even for the Computer players. Information is already passed about whether they'd hit or missed and that was all the information they didn't already know. Anymore information would be unnecessary. They can construct their own boards if they so choose (as in ComputerHard), and since they were only given minimal information, it makes it automatically impossible for any player to cheat. It also reduces coupling between classes.

Also passed to the game was save exceptions, which where only thrown by humans typing in the "exit" command, and then asking to quit. This allowed me to stop constructing the game object and initiate the save method all in one, saving on code.







User interface is simple, and presents all the information about the players actions to the player, so the user knows what's happened. For example: invalid shots passed to the game cause a message to be printed explaining to the player why the game is not progressing, and lets the player enter new values. This helps with easy testing and makes it easy for a user to understand what to do if invalid entries are entered into the console.

The user interface will also return to the printscreen if a ship was sunk on the shot, a feature that could have been inluded to extended to passing the player a ship object if it was sunk, but I decided against this as it would have coupled the ship, the game, and the player classes too highly. If players referred to ship objects, it might mean they could cheat by altering the state of a ship variable, for example.

ComputerHard:

The optimized computer player features two settings: seek and destroy. The default seek setting makes use of parity to find ships on the board (firing shots randomly at squares diagonally

orthogonal to each other) since the minimum ship length is two, it must find a ship at some point, more efficiently than randomly firing at every square.

If the game tells it its last shot is a hit, then it will go into destroy mode, trying to fully destroy the ship it hit. It first checks whether the hit is oriented on the vertical or horizontal (if neither, fire randomly in all directions), and then fires randomly at an adjacent square on that axis. Since it must eventually hit a square on the other side (ships must be spaced), once it misses, it will remove that square from the list of potential targets and move in the other direction. If the list of valid targets is empty, it will resume with the seek method.