1 Code Designs and Discussions

Our antibattleship code designs laid out in the deliverable 2A have not changed substantially. In the process of implementation, we have made significant observations and adjusted our initial framework accordingly.

MasterServer provides the main method that runs our application. Within the main thread, a GUI is launched to allow the user to connect to other users by specifying an address and a port number. The ClientManager thread begins operation when the host initiates a connection with an opponent, or vice-versa.

ClientManager is the overarching class for each individual game. it provides the main thread for gameplay, spawning threads to read input (OpponentReader) and initializing the frame through which the host interacts with the game. We added a message queue (ArrayBlockingQueue) to ClientManager that can be accessed by other threads, including the OpponentReader. As only one message can be read in or sent out at a time, this implementation ensures that no problems arise due to multiple messages accessing components of the game representation at the same time.

When a message (e.g., init-game, target, results) is taken off the queue, it is read by ClientManager and checked for compatibility with the current state of the game (allowable messages for sending/receiving are included in a separate protocol class); for example, it would be nonsensical to send an init-game message in the middle of the game. If the message can be parsed (using methods in ConstructMessage) to a valid action, it is then interpreted by the 'handler' class MessageProcessor. MessageProcessor can update the state of the game, including components of the playing board model and view, according to the content of the message.

As the state of the game progresses, from setup of game parameters (or acceptance/rejection of an opponent's game parameters) to ship layout to gameplay, our GUI changes the featured JPanel object accordingly. We have implemented intuitive mechanisms for laying out ships, firing into opposing waters, and viewing the results of changes in game state.

After beginning to program the implementation of both boardModel and boardGUI, we realized that there had to be further differentiation between a host and an opponent. For instance, the opponentGUI would need to instantiate with all squares being blue, since nothing is known, and the model would reflect the same behavior. The update method will be based on a response from the opponent who confirms a hit or miss, and both components update accordingly. On the other hand, hostModel and hostGUI would check the current fields to update its appearance and location.

Initially these classes were children of two abstract classes. We then realized that it was not necessary to have abstract classes, in fact it was probably easier to just have the boardModel and boardGUI be regular classes that are extended. The behavior varies significantly between host and opponent and therefore warrants different classes that are held to abstract method definitions.

Also, another change was the controller reference from the original plans. The controller now is represented by multiple action/mouse listeners that can update the GUI, add messages to the queue in ClientManager (to initiate games, send board hashes, etc.). While this is not a drastic change, its worth noting that there will no longer be a single controller class.

1.1 Design Amendment

Due to the amendment, we allow our game to be played in salvo mode. In salvo mode, the player should be able to select multiple squares and target them all. We are adding a Boolean field, salvo-mode, to Client Manager, which is flagged if the game is started in salvo. Salvo-mode is turned to true if an init game message where salvo is marked is accepted, or if we accept such message. Both the GUI and messaging suite need to reflect this new capability. We are adding makeSalvoTargetMessage(), makeSalvoInitGameMessage(), and makeSalvoResultsMessage(), with corresponding parsing and receiving said message capabilities. If salvo-mode is flagged, we will call these corresponding message constructors and handlers automatically. We will make similar changes in the GUI to save the state of multiple messages the user clicks if the Boolean is flagged.

2 Usability

1. Interface
    - Will focus on providing step by step text instructions in each stage of the game
    - Also will provide a way to restore/go back to previous state
    - Maintain very intuitive commands to carry out the game
2. Setting up game
    - Network applications will be well documented step by step
    - *Chat window will hopefully be implemented for constant communication
    - Will only allow options to initiate the next stage if inputs are legal

We plan to focus on the 5 categories described in lecture to evaluate and improve our GUIs usability. We feel that the best way to evaluate these subjective points are to allow plain users without experience with our GUI (novices), as well as us (supposed experts) to use it and reflect on our experience. We will have a basic testing session where we, and the novices (primarily our willing friends) will play a game. We will then ask them questions related to Robert Miller's five principle focus areas: learnability, visibility, efficiency, errors, satisfaction.[1]

Pursuant to their feedback and what can reasonably be achieved in the remaining timeframe, we'll make the appropriate adjustments to the GUI.

3 Testing Strategies

Our group feels that testing is essential to show the correctness of our code, as well as to prevent serious errors. This testing includes the modular components of functions, as well as the integrated system.

Some areas we want to focus on are:

- **Message syntax:** Make sure that our code will always send message with correct syntax. It should parse messages correctly, and handle errors in receiving messages. We have already completely test cases for most of the messages we can receive and send, to make sure we flag errors appropriately, and that our syntax is consistent with the protocol. We plan to include more to make sure we are OK in cases like: receiving the same message twice, and correct performance if the user tries to send multiple messages.
- **Handling victory:** We want to do extensive cases to make sure our system handles victory messages in particular. Especially, we want to handle cheating. We realize that there are multiple ways to cheat, including
- **Incorrect hashing**
    - Inconsistent with their reported internal state
    - Not having a valid board (ship) placement
- **Network connections:** We will do a few cases to make sure our system handles network edge cases appropriately, such as playing multiple games at once with the same player, different players, etc. We will also test this with our to-be-done AI bot.
- **GUI:** As stated in our usability section, we plan to evaluate our GUI primarily through user feedback. Of course, we ourselves will need to handle edge cases to show correctness. We will also be the primary testers of effieciency.

[1] See Lecture 16: *Usability*, from the Spring 2011 6.005 website

- **Overall System**: We feel the best way to test the overall system is to have hard coded drivers, or AI bots, which will play the game in a variety of ways. We will have our drivers play the game in edge case game conditions, as well as randomly generated ones. Examples of edge cases include:
  - Small board, dense/large ships
  - Large board, dense ships
  - Large board, few ships
  - Small board, few ships
  - The same as above, repeated for salvo.