6.005 – Antibattleship - Testing and Final Write-up of Results:

Ned Murphy, Julia Boortz, Justin Venezuela
May 10, 2011

Implementation:

Our implementation is in the /src folder of our code. A readme is available in the same folder as this document as well as the root folder. The code design section below describes our how our code works.

Code Design and Reflections Thereon:

In creating the code for our AntiBattleship game we ended up with a design that relied on a limited number of core interfaces and abstract classes with peripheral functionality interacting through the core elements. These core classes and interfaces were ABSGame.java, ABSMessage.java, ABSPlayer.java, and ABSBoard.java.

ABSGame.java serves as the driver for our AntiBattleship engine. ABSGame was responsible for managing communication between the players and reporting the outcome of a completed game. Games begin when ABSGame.Play() is called and end when the method exits returning a game result. While play is proceeding, ABSGame acts as conductor for the two players, orchestrating attacks until the endgame is reached.

ABSMessage is an abstract class that is both as the base class for a number of concrete message types as well as a factory for parsing and creating those concrete classes. When a message String is received from a player, it is parsed by the ABSMessage.CreateMessage() method. If the message is valid, a concrete subclass of ABSMessage is returned and the player to whom the message was sent handles it appropriately. If the message was malformed, one of two things can happen. If the message had an entirely unrecognized format (e.g. “adsllkfja;lsdk”), the ABSMessage factory will throw an InvalidMessageException is thrown which ABSGame handles (in most cases by trying to get another message from the player). If the message has a recognized prefix but an unrecognized syntax after that, NULL is returned and the game raises a syntax error. All game related communications between players is handled through the ABSMessage class and subclasses. Having a robust message parsing strategy ensured that we were able to handle all these communications appropriately.

ABSPlayer is the interface that all players implement in our AntiBattleship game. It contains only a method to get the player’s name, a method to send messages to the player, and a method to get messages from the player. Objects that implement this interface such as NetworkPlayer, GUIPlayer, and Jasper do all the required logic to handle the messages received and send the appropriate messages back. GUIPlayer is especially as this is the logic the players interacting with the GUI will be using. Looking back, there was a fair bit of code duplication between all these player classes and it might have been better to make a base class containing the common logic. Once we changed our message strategy to use subclasses of the abstract ABSMessage, we may also have been able to take advantage of the visitor pattern instead of using the switch blocks that appear in several of the player classes.

Finally, ABSBoard was the abstract superclass for ABSAttackBoard and ABSMarkBoard. These concrete classes stored the board states for the opponent and the player

respectively. As players made attacks at each other, these concrete ABSBoard objects would determine the outcome of a given attack attempt and update the model accordingly.

In addition to these core classes we had a large number of other supporting components. The GUI surfaces and listeners represent a significant chunk of these. The GUI code is closely tied to the ABSPlayer implementing GUIPlayer which it relies on for state updates to display. Another significant but peripheral component was the networking code. Setting up a hosted game or connecting as the client is handled by the code in ABSConnection.

The final components of note are the extensions we chose to implement. We created a chat experience that allows networked players to communicate in a floating GUI as they play. We also implemented code to track win-loss records based on user names stored on a database. These records are displayed when a game completes.

Testing:

Game Engine Testing:

Game engine verification:
As we worked to build our AntiBattleship application we leveraged a mix of automated and manual test strategies at each point in the development cycle to verify first, that basic functionality was implemented correctly, next, that "bad" inputs could be handled appropriately and gracefully, and finally, that the end to end experience of playing our AntiBattleship game worked locally and over the network.

In order to be able to verify that the basic functionality of our game engine was running correctly, we created an automated player (named Jasper) that would play games against the provided reference player and log the results, tracking player victories and errors (if any). As we made changes to the code, we would start an automated test run where Jasper would play some number of games against the reference player and log the results. If any errors in the basic game logic were detected, we would see them in Jasper's interaction with the reference player and have a good idea of what went wrong by looking at the log.

While this automated approach allowed us to find errors that presented when playing against a well behaved opponent (e.g. errors resulting from game logic mistakes on our end), we relied on manual testing strategies to detect errors resulting from unexpected inputs and irregular game actions. The automated player, Jasper, was able to assist in this effort as well by providing an AI for a human tester to play against. As more of our game features came online, we made it a habit to play games against the computer controlled AI before every submission, specifying values for board sizes, number of ships, and ship positions that were known to be invalid and verifying that the game could handle them appropriately. We uncovered a number of bugs using this approach, especially in the position ships phase of the game.

When we connected the networking components, we tested how the game functioned over the network by starting a game as the host and connecting through telnet. This was far from an ideal test strategy as we had to build all the game messages by hand but we were able to verify the basic functions of connecting to a host and initiating a networked game. Towards the end, we

were able to connect two computers running our ABS game code and play over the network using the UI.

While these test strategies worked well for us, we had a different idea originally. We had originally thought that we would be able to create an XML file that represented the game state, read that file in to create the state stored in the file, then perform a game action and verify the result against another XML file that represented the expected game state after the action completed. However, this method turned out to have a poor cost benefit ratio. Our design of the game engine did not operate on states per se so the only way to get the board into state X was to begin a game and proceed through states A through W in order. Implementing the original testing strategy would have required a (perhaps needed but) major redesign of our game engine. Furthermore, since we aren't restricting the size of our game boards (beyond using integers to represent the board dimensions), there are an unmanageably large number of possible board states that are possible. Even though each state transition should be deterministic, and there are only a handful of states that could represent broad categories of other states, we felt that it was not worthwhile to implement this automated testing strategy.

Message component testing:

In addition to our basic verification tests described above, we wrote some component level tests to exercise some of our core components. The ability to create and handle ABSMessage objects is an essential part of our game engine so we created a number of JUnit test cases that verified we could both build and parse good and bad message strings and respond appropriately. We made a design decision to throw an InvalidMessageException when the user typed in a string that was not prefixed with one of the recognized commands ("init-game", "target" etc.) which would be handled in our game engine. In these cases, the game would listen again until a message with a known prefix was detected. If that prefix was followed by something unexpected though, we would return NULL from our message factory which would result in a syntax error being raised and the game terminated. We were able to write JUnit test cases to verify these actions were occurring for the correct families of inputs which aided in our development by alerting us to changes in the message handling that compromised this functionality.

GUI Testing Strategy:

While it is impossible to test every possible input to our GUI, we will aimed for comprehensive coverage of possible scenarios that could arise and will ensure to test the common case as well as all possible edges cases.

Common case: Our most basic testing involved playing many games of Antibattleship using our GUI paying particular attention to what is happening to ensure all is working as expected. Since we have three different modes (play the computer, host a network game, and connect to a network game), we made sure to play games in all three. During this phase of the testing, we ran into several bugs initiating game over the network and with the correct waiting screens/notification screens showing up, but we have resolved all bugs and played many successful games, including games where all possible parties won (GUIPlayer, computer, host, hostee)

We tested the following edge cases, which we believe are representative of all possible problems that could occur.

To ensure that our GUI handles illegal inputs correctly, we tried putting valid inputs in all but one field on each screen and then putting an invalid input into the remaining screen. We tested the following possible inputs for all of the textboxes that take a positive Integer: 1) A letter or other character that is not numerical, 2) A negative number, 3) Zero, 4) One (in some cases this will also be invalid), 5) A standards input (i.e. a number in the range of 2-10), and 6) a really large input that would overflow the Integer field. We did not find any bugs during these tests.

Additionally, we made sure that it is not possible to enter an invalid board. To test this, we tried entering a board that is too short on all sides to fit the longest ship. We also tried entering a board that is large enough to fit the longest ship, but where the other dimension was not two times the number of ships. As part of this test, we made sure that the dimension that is long enough to fit the largest ship is also two times the number of ships to make sure that our code checked that the two conditions were meet by different dimensions (i.e. one ship of length three and a 3x1 board). Our game correctly rejected all of these inputs. We then tried entering a board that was exactly the minimum size allowed (one ship length 3 and 3x2 board) to ensure that it returns true at in the edge case. Our game rejected this valid board so we had to re-examine our code to figure out why. We found that we were checking that each of the two conditions were meet for both directions, which was over constraining our board. After fixing the code, we re-ran the above tests and everything passed. Finally, we will tested a board that is valid and not close to being invalid (two ships lengths 3 and 4 and 7x8 board). This case worked as expected.

To ensure that all of our buttons worked working correctly, we played games that utilized all of them. For instance, in the initiate game phase, the user has the option of adding ships and then clearing them and then re-adding them. We made sure to test this and tested that our game still recognized invalid inputs if a valid input is entered and then cleared or vis versa. Our game passed all of the button tests.

In the place ships screen, in addition to testing all of the buttons and textfields as described above, we also tried to place ships in invalid ways to ensure that our game catches it. For instance, we tried placing ships adjacent and on top of each other and will ensured to test the case where ships are diagonally adjacent. We tried submitting our ships before they were all placed on the board. Finally, we tried placing all ships on the board and then resetting them and make sure that the checks still worked and that the submit button did not work if the ships were no longer on the board. The game did not allow any of these invalid inputs. We checked that ships could be placed vertically and horizontally and tested many random combinations of valid inputs to ensure that all valid placements of ship are allowed. We did not find any valid placements that were rejected during this phase. To make sure the edge case worked, we tried placing ships that were only one square off from being invalid. The game accepted these valid board states as well.

To test the game playing screen, we tested that all buttons were working and ensured that we play games where both the user using the GUI wins and games where the AI bot wins to make

sure that the ending of the game for both possible cases is covered. Our code passed these tests. Finally, we checked that the GUI returns to the main menu after a game is played and that we could then initiate a new game and play it all the way through without problems.

We believe that while this strategy does not cover every possible input (which is impossible) it allows us to be confident that our GUI is fully functional.

Reflection:

Creating our Antibattleship game, we learned to work on a large software project as a team and learned that careful planning beforehand pays off in the long run. Before starting on the project, we meet to design our game and drew a control flow diagram on the board to make sure we were all on the same page as to what the game would do at each step. This meeting turned out to be immensely helpful as we coded our game. We had split up the work into section (GUI, Network, and game architecture/AI) so this diagram enabled us to code separately while ensuring that all of our pieces would fit together in the final game. The control diagram also made writing the game architecture relatively straightforward.

The hardest problems we faced involved networking and combining the GUI with the game. The networking part of the project introduced interesting concurrency problems that we had to address through the use of threads, specifically, by requiring threads to sleep at different times.

Combining the GUI with the game, i.e. our GUIPlayer also proved particularly difficult. Had we planned this part of the project better ahead of time, it would have been much easier. While both parts worked independently and fit with our control flow, we found it especially difficult to pass information between the GUIPlayer and the GUI at the correct time. The GUIPlayer knew when it received responses from its opponent, but then somehow had to get that information to the GUI. We used while loops that remained in the loop until the variable in the GUIPlayer that was supposed to contain the response was no longer null. We then passed the results to the GUI and allowed the GUI to updates its state accordingly. If we were to re-do this project, we would redesign the message passing between the GUI and the GUIPlayer. We would use an event based or message based design similar to the one used in our ABSGame to handle messages between the two players. In the improved design, the game engine would maintain an internal representation of the state of the game that could be passed to the GUI. The GUI would implement queues for messages. This would make our code easier to understand and easier to debug. It would also help eliminate some of the concurrency problems – one of the more unexpected challenges - that we encountered when using while loops to pass control to the GUI.