Antibattleship

6.005 Project 2 Design

Ned Murphy, Julia Boortz, Justin Venezuela

April 26, 2011

1 Overview

Implementing our Antibattleship (ABS) game will consist of coding three main parts: the *client* code, which represents the state of a game of antibattleship outside of representation and user interaction, the *GUI* code, which represents the user's interaction with the client, and the *network* code, which represents the client's interaction with external players.

2 Client

Any game of ABS will create a new instance of the class ABSGame, which will take in two ABSPlayer instances. ABSGame acts as a message handler between the two players, and controls the flow of the game. Each ABSPlayer, whether representing a human player, an AI player, or a network player, must be able to both send and receive ABSMessages from an ABSGame. ABSGame then determines when an ABSMessage should be sent/received to/from either player. However, ABSPlayers need not know the protocol used to communicate with other players: they need only communicate to ABSGame using our ABSMessage class. ABSGame handles stripping down these ABSMessages into strings that follow the protocol, as well as reconstructing ABSMessages from these strings.

It is worth noting that ABSPlayer's ignorance of the protocol allows for, say, easy implementation of AI players, since we needn't concern ourselves with parsing Strings, etc.. However, this is not too restricting in the space of possible ABSPlayers we can make: one ABSPlayer already written simply acts as a wrapper class for human input into a console, in which the human player is expected to simply type messages following the protocol.

3 GUI

We note that one could play a game of Antibattleship using the client code without a graphical user interface. However, this is surely not a good idea if we have user friendliness in mind. Our GUI interacts with ABSGame and displays the relevant game state information to the user. The GUI code also includes listeners which take user inputs and sends the information to ABSGame.

4 Network

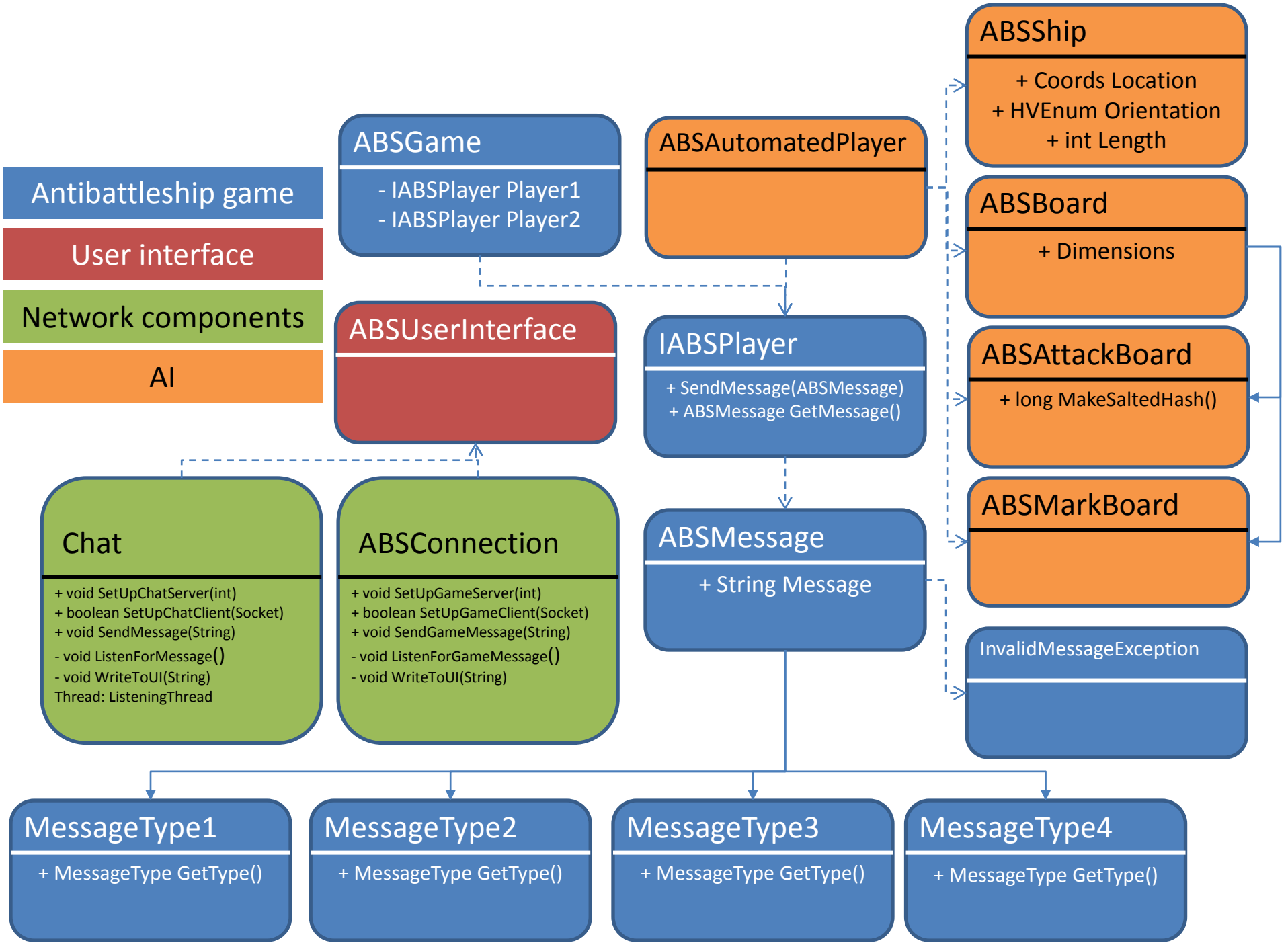
Our network code is responsible for connecting to network players. This code is decoupled from the client. The network code needn't understand our ABSMessage scheme or even the network protocol. It need only be able to connect outside players to ABSGame and handle passing protocol strings between ABSGame and these outside players.

5 Flow Diagrams

We have drawn out flow diagrams that map out the functionality of our ABS game. They are available (as image files) in the SVN in the same folder as this document.

6 Contact

Please direct any questions regarding our design to `nedmurp@mit`, `jboortz@mit`, and `jven@mit`. ■


Antibattleship game
User interface
Network components
AI
ABSGame
- IABSPlayer Player1
- IABSPlayer Player2
ABSAutomatedPlayer
ABSShip
+ Coords Location
+ HVEnum Orientation
+ int Length
ABSBoard
+ Dimensions
ABSUserInterface
IABSPlayer
+ SendMessage(ABSMessage)
+ ABSMessage GetMessage()
ABSAttackBoard
+ long MakeSaltedHash()
ABSMarkBoard
Chat
+ void SetUpChatServer(int)
+ boolean SetUpChatClient(Socket)
+ void SendMessage(String)
- void ListenForMessage()
- void WriteToUI(String)
Thread: ListeningThread
ABSConnection
+ void SetUpGameServer(int)
+ boolean SetUpGameClient(Socket)
+ void SendGameMessage(String)
- void ListenForGameMessage()
- void WriteToUI(String)
ABSMessage
+ String Message
InvalidMessageException
MessageType1
+ MessageType GetType()
MessageType2
+ MessageType GetType()
MessageType3
+ MessageType GetType()
MessageType4
+ MessageType GetType()

Networking architecture and protocol:
Our initial design for the network layer of our Anti-battleship program calls for two TCP/IP socket connections to the opponent. One channel will be for transmitting and receiving game impacting messages. The other channel will be for exchanging chat messages. These two channels will be represented by the classes ABSConnection.java and Chat.java respectively. Each class will be capable of starting as the server or client for the session. Chat messages will be sent over the wire as plain text while game messages will be sent as objects according to an object contract. Both chat messages and game messages that require output will be printed in the chat output window (with different formatting so as to distinguish the two).

Security:
We would like to build in some amount of network security into our game message channel in order to protect against cheaters/hackers. We are considering the cost of implementing SSL through the JSSE API.

Our design should offer some protection against denial of service attacks because we will listen to the opponent only when we want a response and we will take only the first response we receive. The chat channel runs on a separate thread so a denial of service attack against the chat thread should not affect the game thread.

6.005 Spring 2011
Project 2 Anti-Battleship
 # Protocol Definition

A is the initiating player
B is the receiving player

==========
= General Notes
==========
If either player ever receives a message it is not expecting, just ignore it.

msg := syntax-error-msg | game-error-msg | init-game-msg
| game-accept-msg | game-deny-msg | board-hash-msg
| target-msg | results-msg
| victory-msg | accept-victory-msg | reject-victory-msg

==========
= Error Handling
==========
There are two types of errors, syntactic and game.

A syntax error can be issued with the contents of the offending message appended to it. Syntax errors can either be the result of malformed messages or messages sent at the wrong time.

syntax-error-msg := "syntax-error" msg

If the message was well-formed but semantically incorrect (board position that is out of bounds ...),
then a semantic error can be issued. Like the syntax error, the offending message will be appended after the message.

game-error-msg := "game-error" msg

Once an error message is sent the game is now over, and the player that caused the error loses. If you send an error message when the other player did nothing wrong this is a bug. You may lose correctness points, and you will not be able to compete in the tournament.

==========
= Initialization Phase
==========
A sends game request to B

init-game-msg := "init-game" board-size ship-size-list
board-size := number "x" number # rows x columns
ship-size-list := "[" (number ",")* number "]"

The rows and columns in the board are 0-indexed.
(0,0) represents the the top left square of the board.

Example: "init-game 10x9 [1,1,2,3,5]"

This is a game on a board that is 10 squares high, 9 wide. It has 2 ships of length 1, 1 of length 2, 3 and 5.

Example: "init-game 5x5 [2,2,2,2,2]"

This is a game on a board that is 5 squares high and 5 wide. It has 5 ships of length 2.

B sends accept request or deny request to A.

game-accept-msg := "accept-game"
game-deny-msg := "deny-game"

If B has accepted the game, A must then send the SHA hash of the

initial board position to B (see "Hashing").

board-hash-msg := "board-hash" board-hash
board-hash := ...

B will then reply with the hash of its own board in the same format

board-hash-msg, etc

==========
= Game Phase
==========

A always moves first. The move will look like:

target-msg := "target" position
position := "(" number "," number ")" # (row, column)

Again, the rows and columns in the board are 0-indexed.
(0,0) represents the the top left square of the board.

Example: target (4,5)

If the move was valid, then the other player (the one who did not make the move) must reply with the results:

results-msg := "results" outcome
outcome := "water" | "fire" | "sunk"

Example: "results fire"

Note that for each square, you can only have one outcome. So if a square is hit and no ship is sunk: you return 'fire'. If a square is hit and a ship is sunk, you only return 'sunk' for that square. If a square is miss, you return 'water' for that square.

==========
= Victory Phase
==========

Either player can declare victory when all of their ships have been sunk by sending the victory message instead of the target message on that player's turn:

victory-msg := "victory" salt board-state
salt := number

board-state := "" | "0" board-state | "1" board-state

To computer board state, iterate through each row from left to right
starting in the top row. If there is a ship in that square add "1" to
the string, otherwise add a "0".

So suppose you had this 3 by 5 board where a 0 is an empty square and
a x is a ship:

000xx
x0000
x0000

board-state would be: 000111000010000

The other player can either agree with that victory, after checking
the board state for cheating, or reject it

accept-victory-msg := "accept-victory"
reject-victory-msg := "reject-victory"

==========
= Hashing
==========

In this protocol, we wish for you to compute a "SHA" hash of the board
position for security. As before, the board state of an m by n board
can be represented as a string of 0's or 1's. In the example above,
the board-state was "000111000010000".

SHA has different variants (e.g. SHA-1, SHA-2). For this protocol, we
provide code that uses a Message Digest and its default "SHA"
implementation (see code that follows).

To get the hash of a board position, we start with the string
representing the board state, e.g. "000111000010000". We append "m"
and then "n" and a salt that you choose (i.e. salt=10) to the string, to get
"0001110000100003510", and then
convert this String to a byte array (i.e. byte[]) via
String.getBytes(), and then use the byte[] as an input to a "SHA"
Message digest. We convert the returned byte[] into a String that
contains hexadecimal representation of the bytes in the digest.
The salt is your extra protection against dictionary attack against your sent board hash.

So, a board with "000111000010000", m=3, n=5 and salt=10 has the following hash:
"4960876e27116662b8f93a2dbf40b737e50ddf95".

Also, a board with "011000110010110", m=5, n=3 and salt=10 has he following hash: "20fb73777e1523828b897264d212c21407192831".

The code snippet for computing these SHA hashes follow.

```
=======================================
= HashExample code
=======================================

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class HashExample {

	private static String convertToHex(byte[] data) {

		StringBuffer buf = new StringBuffer();
		for (int i = 0; i < data.length; i++) {

			int halfbyte = (data[i] >>> 4) & 0x0F;
			int two_halfs = 0;

			do {

				if ((0 <= halfbyte) && (halfbyte <= 9))

					buf.append((char) ('0' + halfbyte));

				else

					buf.append((char) ('a' + (halfbyte - 10)));

				halfbyte = data[i] & 0x0F;

			} while (two_halfs++ < 1);

		}
		return buf.toString();
	}

	public static String computeBoardHash(String boardRep, int rows, int columns, int salt)
			throws NoSuchAlgorithmException {
		MessageDigest md = MessageDigest.getInstance("SHA");
		String board = boardRep + rows + columns + salt;
```

```
            byte[] bytes = board.getBytes();
            md.update(bytes);
            return convertToHex(md.digest());
        }

        public static void main(String[] args) {

            String board = "000111000010000";
            String board2 = "011000110010110";
            try {
                String hash = computeBoardHash(board, 3, 5, 10);
                System.out.println(hash);
                String hash2 = computeBoardHash(board2, 5, 3, 10);
                System.out.println(hash2);
            } catch (NoSuchAlgorithmException e) {
                e.printStackTrace();
            }

        }
}
```

6.005 - Team Contract

Goals:
1) Have working project that meets all requirements
2) Finish far enough in advance that if something goes wrong we will have time to adapt to it.

Meeting Norms:
1) Times will be determined via email communication. They may either be held over gchat or on campus.
2) Frequency and lengthwill be determined by need and communicated via email
3) Cookies
4) Minutes and Action Lists will be distributed via google documents

Work Norms:
1) Work will be equally distributed to best enable members to work on tasks that most interest them.
2) Deadlines will be mutually agreed on by group members.
3) Team members will review each others work as part of the testing process.
4) Any conflicts will be discussed at a meeting.

Decision Making:
1) Individual group members may make decisions about the part of the code they are working on that does not affect other code. Decisions that span group members work will be mutually agreed on. Major design decisions will be mutually agreed on
2) We will discuss it and accept it or convince the group member that it is a bad idea by giving good reasons.

6.005 - Group Meeting Agenda - 4/18/11

Facilitator: Ned
Note Taker: Julia

Contact Info:
Julia – 614-582-5311, jboortz@mit.edu, juliaann11@gmail.com
Justin – 860-690-5040, jven@mit.edu, popballard11@gmail.com
Ned – 617 – 653 – 2163, nedmurp@mit.edu, ned.e.murphy@gmail.com

Best way to contact:
Julia - email, gchat
Justin - email, gchat
Ned - phone, email

Ideas rejected:
Requiring user to check for cheating (Checking for cheating done automatically).

Completed:
Abstract State Machine Design - Julia will scan and upload

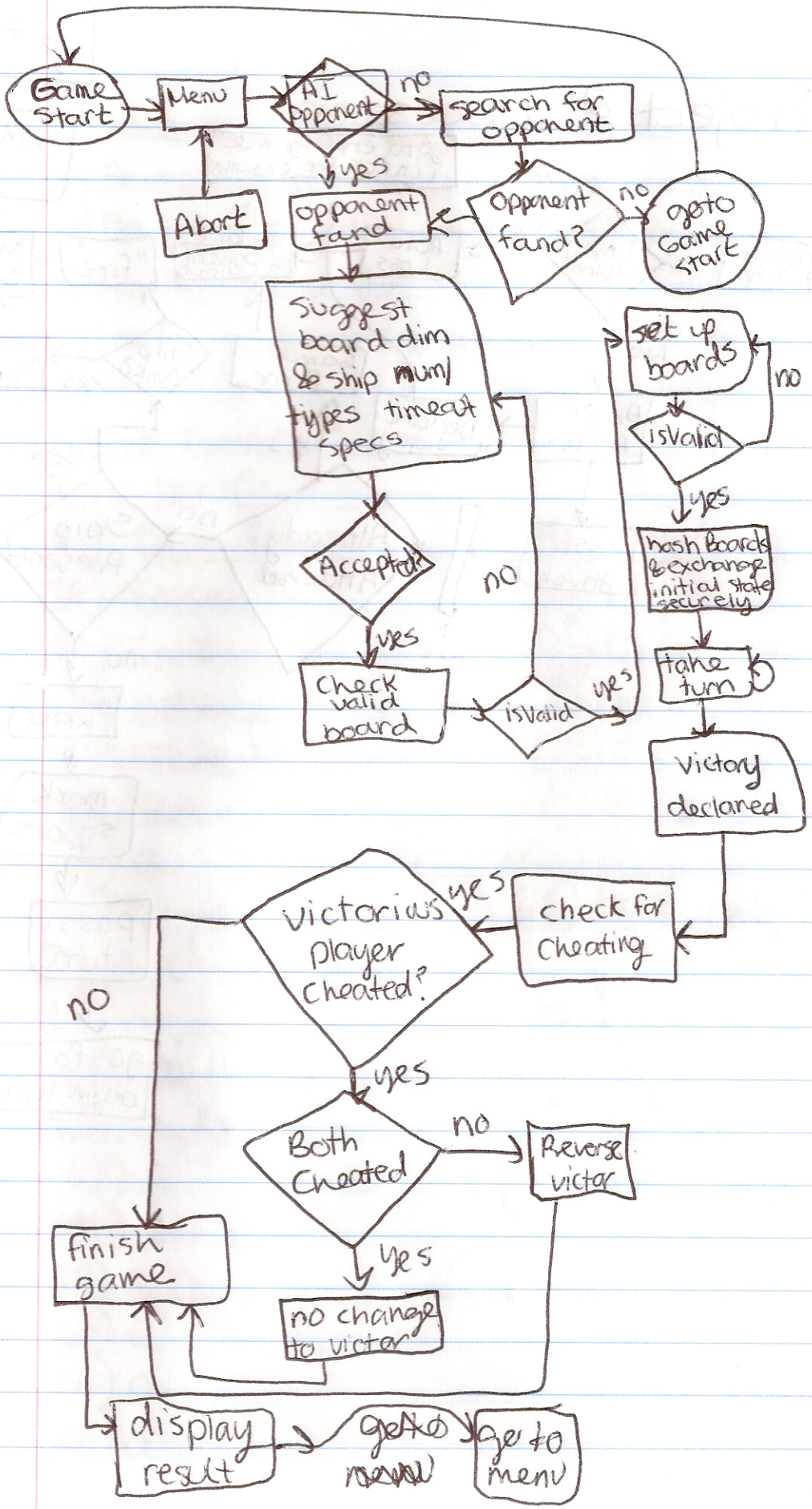
Work Division:
Ned - networking, mock player (eventually for testing)
Justin - inheritance structure and logic
Julia - UI, upload docs to svn

Create Team Contract:
Done

Schedule Second Meeting:
Thursday at 5:30pm over gchat

Create Second Meeting Agenda:
Touch base, see how each person is going
Create plan for work over weekend.

Adjourn - check


Game start
Menu
AI opponent
no
search for opponent
yes
Abort
opponent found
Opponent found?
no
goto Game start
Suggest board dim & ship num/ types timeout specs
set up boards
no
isValid
yes
hash boards & exchange initial state securely
Accepted?
no
yes
take turn
Check valid board
isValid
yes
Victory declared
Victorious player Cheated?
yes
check for Cheating
no
yes
Both Cheated
no
Reverse victor
yes
finish game
no change to victor
display result
goto menu
go to menu

4/18/11 Project 2 Discussion