6.005 – Antibattleship – Deliverable #2

Ned Murphy, Julia Boortz, Justin Venezuela

6.005 Spring 2011
Project 2 Anti-Battleship
 # Revisions Caused by Amendment 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
==========
no change in error handling

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

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

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. This is not the salvo variant.

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

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

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 ")"

Regular Variant:
Example: target (4,5)
Salvo Variant:
Example: target (4,5) (1,3) (7,7)

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"

Regular Variant:
Example: "results fire"
Salvo Variant:
Example: "results water fire fire sunk"

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
==========

no change in victory phase

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

no change in hashing

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.

Attached at the end of this documents is a class diagram showing our code design.

Usability:

We designed our GUI with the assumption that the user knows how to play Anti-battleship in general, but has never played our particular version before. The GUI guides the user through each part of the game with four main screens containing a combination of directions and labeled buttons to allow the user to take appropriate action. Popup and wait screens are utilized to notify the user of various other aspects of the game and to allow the user to make yes/no choices. The sketches of the four main screens in the order that the user will encounter them are provided in UIPage1 and UIPage 2 in the deliverables folder.

In addition to designing a self-explanatory GUI, we designed the system so that the user could not easily break it by attempting something illegal. For instance, if a user enters invalid board specs or tries to create a ship of non-integer length, a popup screen notifies him of the problem and allows him to fix it. Under no circumstances should the game crash because of something the user enters.

Additionally, we utilize popup screens to allow the user to accept or reject the game specs of another user. To ensure that a user always knows when she must wait for the other player to take an action, we will implement wait screens that notify the user that the game is waiting for an action from the other player. Therefore, the user will never think that the program has hung up when it is supposed to be waiting.

All of these features are designed to provide the user with a positive gaming experience that enables him to focus on his strategy.

Testing Strategy:

GUI Testing Strategy:

While it is impossible to test every possible input to our GUI, we will aim 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 will involve 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 will make sure to play games in all three.

To test our edge cases we will make sure to cover all possible edge cases.

To ensure that our GUI is handling illegal inputs correctly, we will try putting valid inputs in all but one field on each screen and then putting an invalid input into the remaining screen. To check that it is handling all invalid inputs correctly, we will test the following possible inputs for all textbooks 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.

Additionally, we will make sure that it is not possible to enter an invalid board. To test this, we will try entering a board that is too short on all sides to fit the longest ship. We will also try to enter a board that is large enough to fit the longest ship, but where the other dimension is not two times the number of ships. As part of this test, we will make sure that the dimension that is long enough to fit the largest ship is also two time the number of ships to make sure that our code is checking that the two conditions are meet by different dimensions. We will then try entering a board that is exactly the minimum size allowed to ensure that it returns true at the edge. Finally, we will test a board that is valid and not close to being invalid.

To ensure that all of our buttons are working correctly, we will make sure to play games that utilize 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 will make sure to test this and will test that our game still recognizes invalid inputs if a valid input is entered and then cleared or vis versa.

In the place ships screen, in addition to testing all of the buttons and textfields as described above, we will also try to place ships in invalid ways and see if the game catches it. For instance, we will try places ships adjacent and on top of each other and will make sure to test the case where ships are diagonally adjacent. We will try submitting our ships before they are all placed on the board as well to ensure that the game does not allow this. Finally, we will try placing all ships on the board and then reseting them and make sure that the checks still work and that the submit butting does not work if the ships are no longer on the board. We will also make sure that ships can be placed vertically and horizontally and will test many random combinations of valid inputs to ensure that all valid placements of ship are allowed. To make sure the edge case is working, we will try placing ships that are only one square off from being invalid and make sure that the game accepts them.

To test the game playing screen, we will make sure that all buttons are working and will ensure that we play games where both the user using the GUI wins and the AI bot wins to make sure that the ending of the game for both possible cases is covered. Finally, we will check that the GUI returns to the main menu after a game is played and that we can 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.

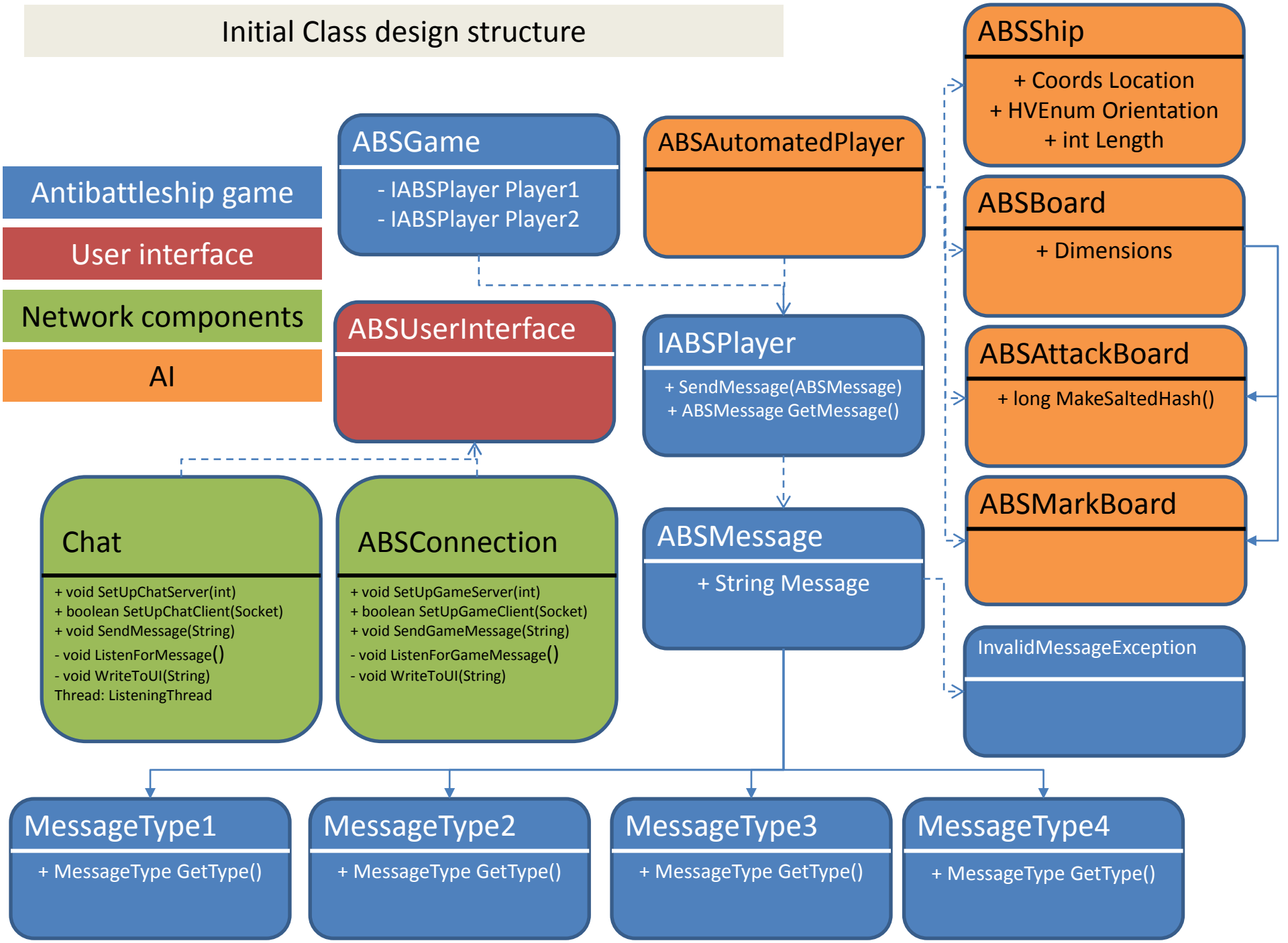
Test Strategy for AntiBattleship Program (minus GUI)

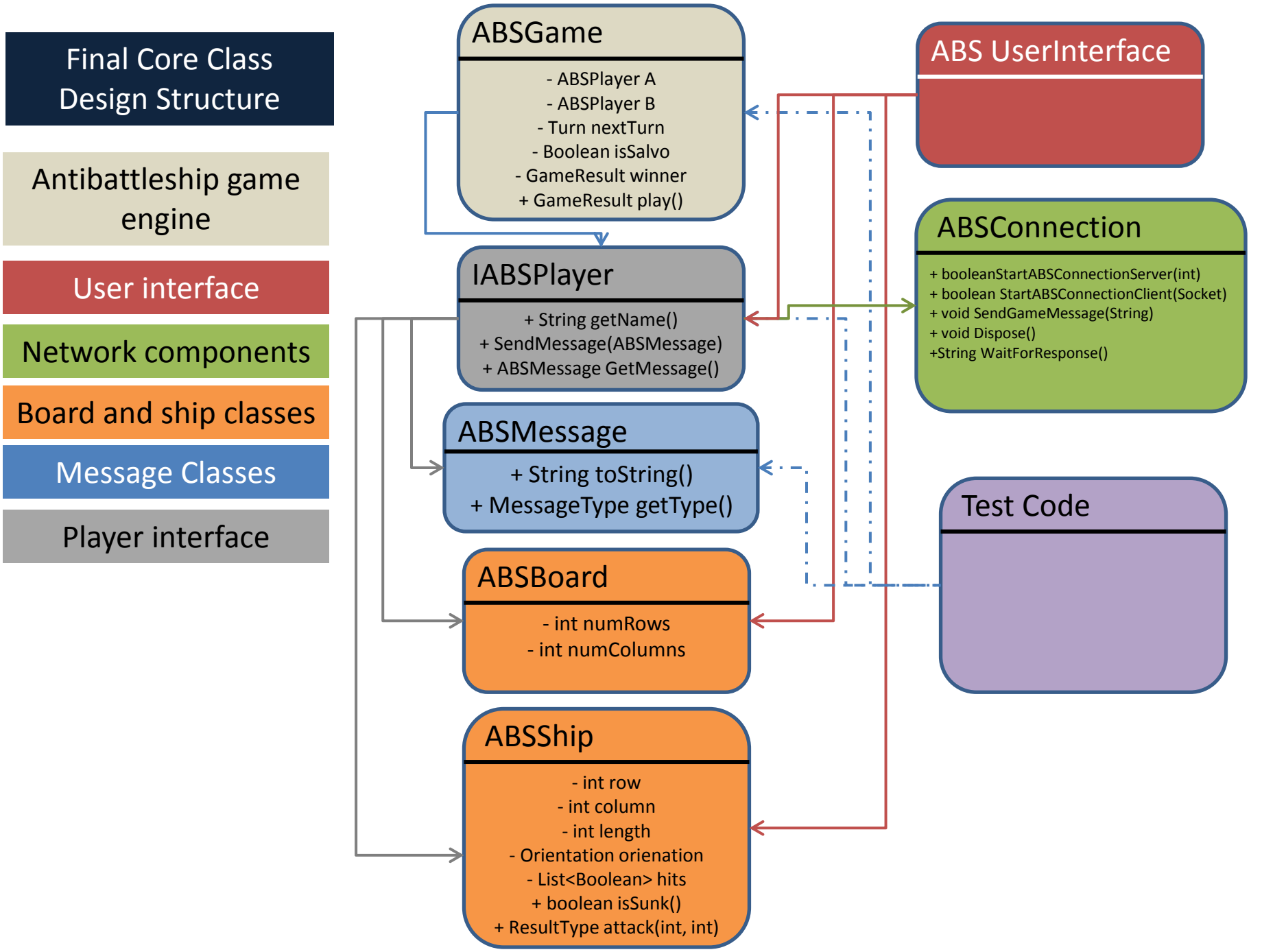
Our plan for testing the AntiBattleship program centers on using pre-generated game states stored in xml files. We plan to create methods that will read in these xml files and set up the internal game state then execute an action, serialize the game state back to xml and compare it against a previously generated file.

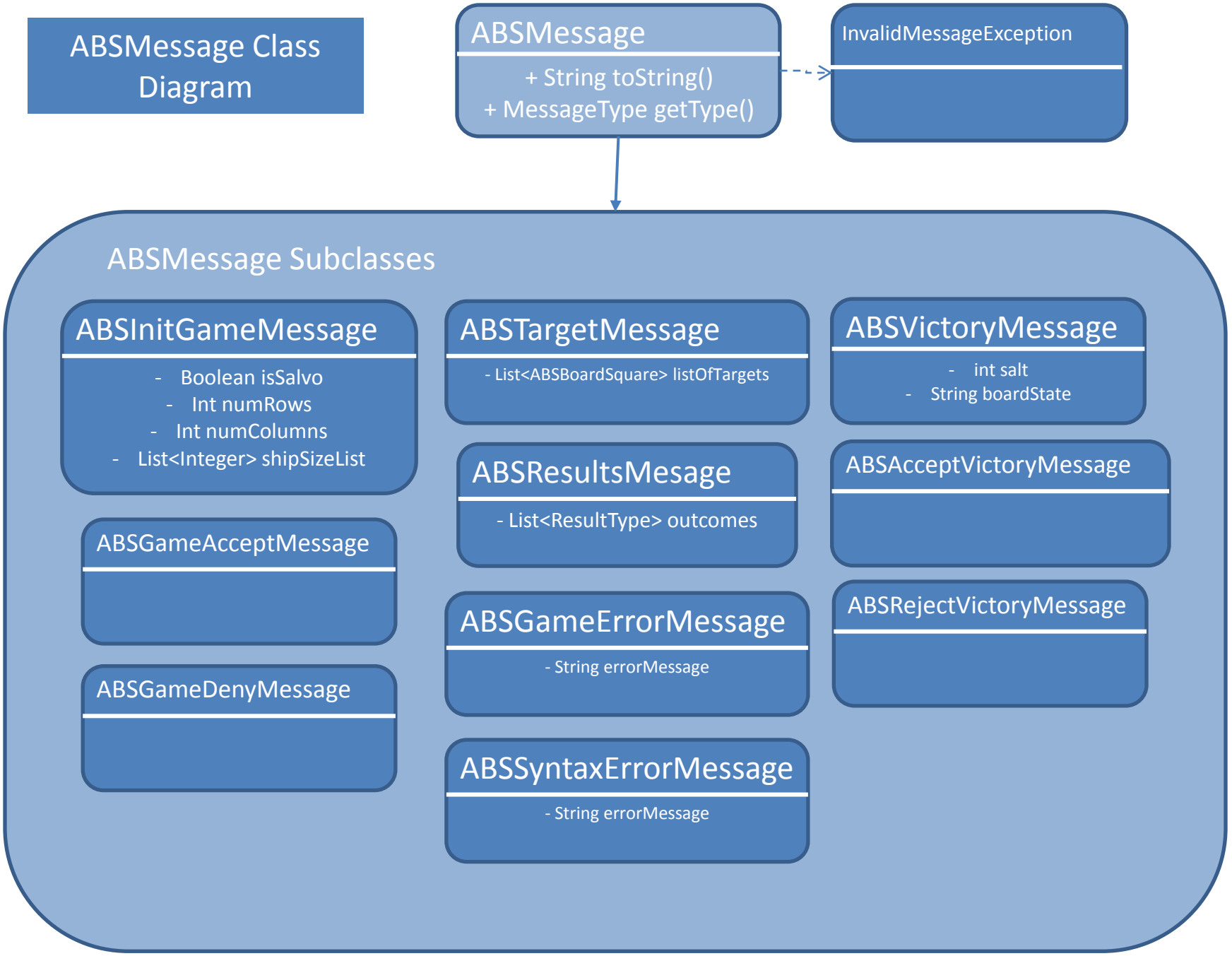
With this plan, we should be able to simulate the following board states: initialize game, board unmarked, board fully marked, waiting for player input, waiting for opponent input, end game. Once we have set up the game state, we should be able to trigger actions within the game and verify the correct state occurred. Most actions should not result in any state transition (negative test cases) but a few will result in a predictable state transition. These new states will be stored in xml files and compared against expected values.

We will also play multiple games with our bot and the reference player to make sure the basic functionality works as expected, and also that the 'common' user input mistakes do not break anything. This will be done with both a combination of just passing raw strings to our AI with telnet, and also using our GUI to play vs our AI and the ref player.

Initial Class design structure

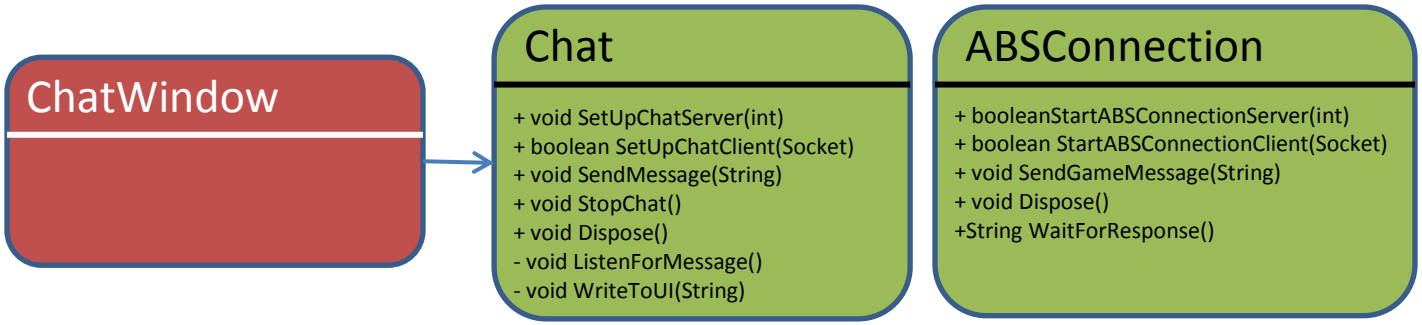



Final Core Class Design Structure
Antibattleship game engine
User interface
Network components
Board and ship classes
Message Classes
Player interface
ABSGame
- ABSPlayer A
- ABSPlayer B
- Turn nextTurn
- Boolean isSalvo
- GameResult winner
+ GameResult play()
ABS UserInterface
ABSConnection
+ booleanStartABSConnectionServer(int)
+ boolean StartABSConnectionClient(Socket)
+ void SendGameMessage(String)
+ void Dispose()
+String WaitForResponse()
IABSPlayer
+ String getName()
+ SendMessage(ABSMessage)
+ ABSMessage GetMessage()
ABSMessage
+ String toString()
+ MessageType getType()
Test Code
ABSBoard
- int numRows
- int numColumns
ABSShip
- int row
- int column
- int length
- Orientation orienation
- List<Boolean> hits
+ boolean isSunk()
+ ResultType attack(int, int)

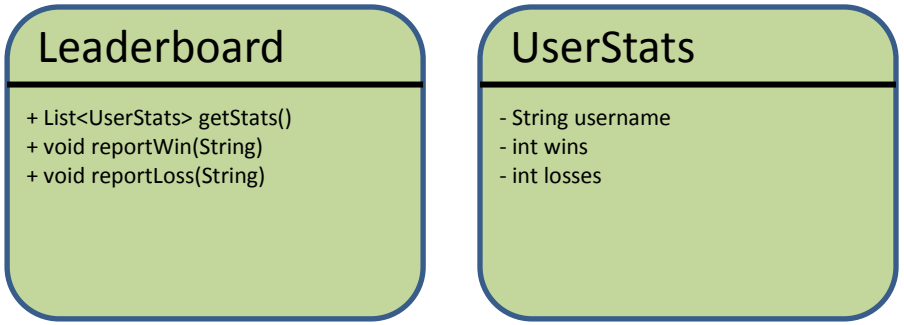

ABSMessage Class Diagram
ABSMessage
+ String toString()
+ MessageType getType()
InvalidMessageException
ABSMessage Subclasses
ABSInitGameMessage
- Boolean isSalvo
- Int numRows
- Int numColumns
- List<Integer> shipSizeList
ABSGameAcceptMessage
ABSGameDenyMessage
ABSTargetMessage
- List<ABSBoardSquare> listOfTargets
ABSResultsMesage
- List<ResultType> outcomes
ABSGameErrorMessage
- String errorMessage
ABSSyntaxErrorMessage
- String errorMessage
ABSVictoryMessage
- int salt
- String boardState
ABSAcceptVictoryMessage
ABSRejectVictoryMessage

ABS Game Add Ons Class Diagram

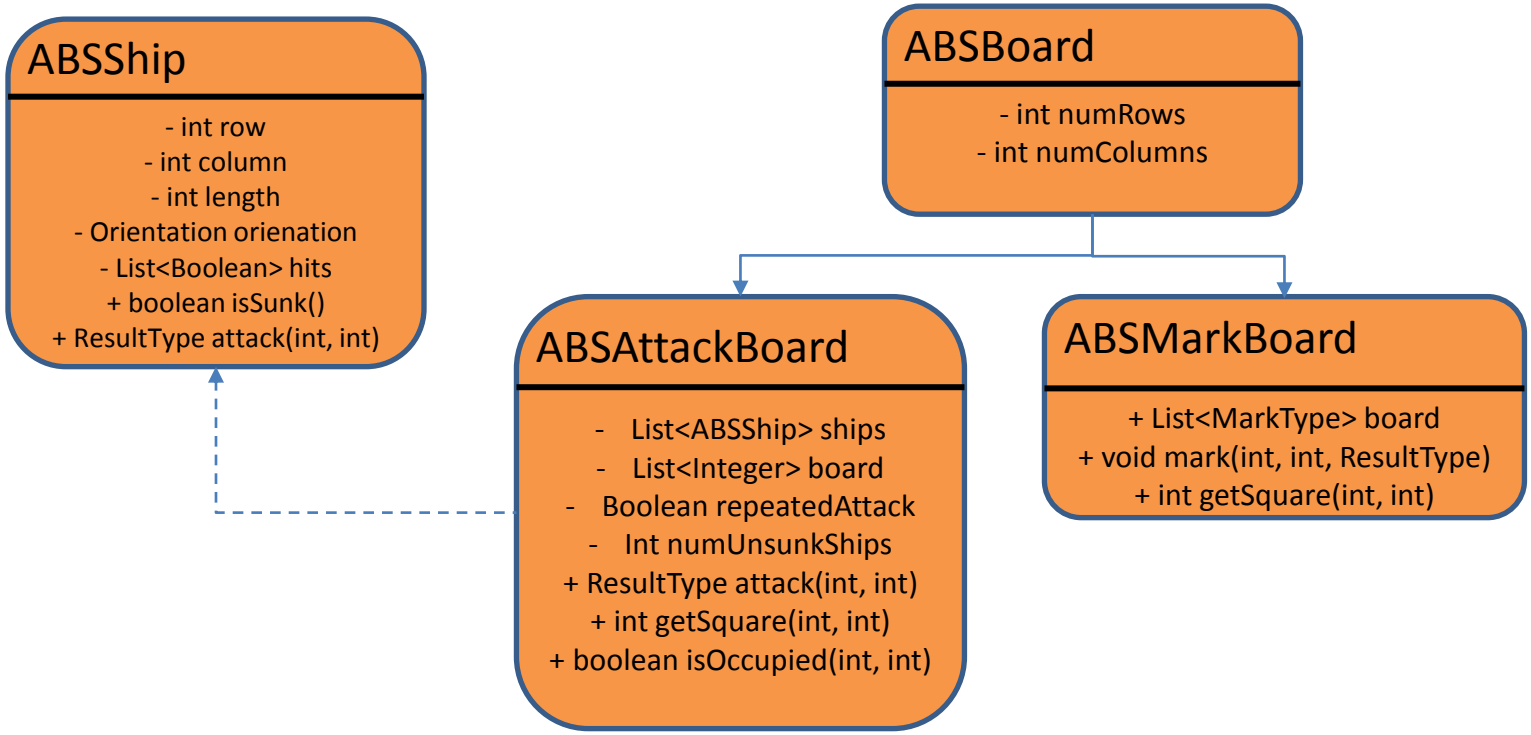
Network components



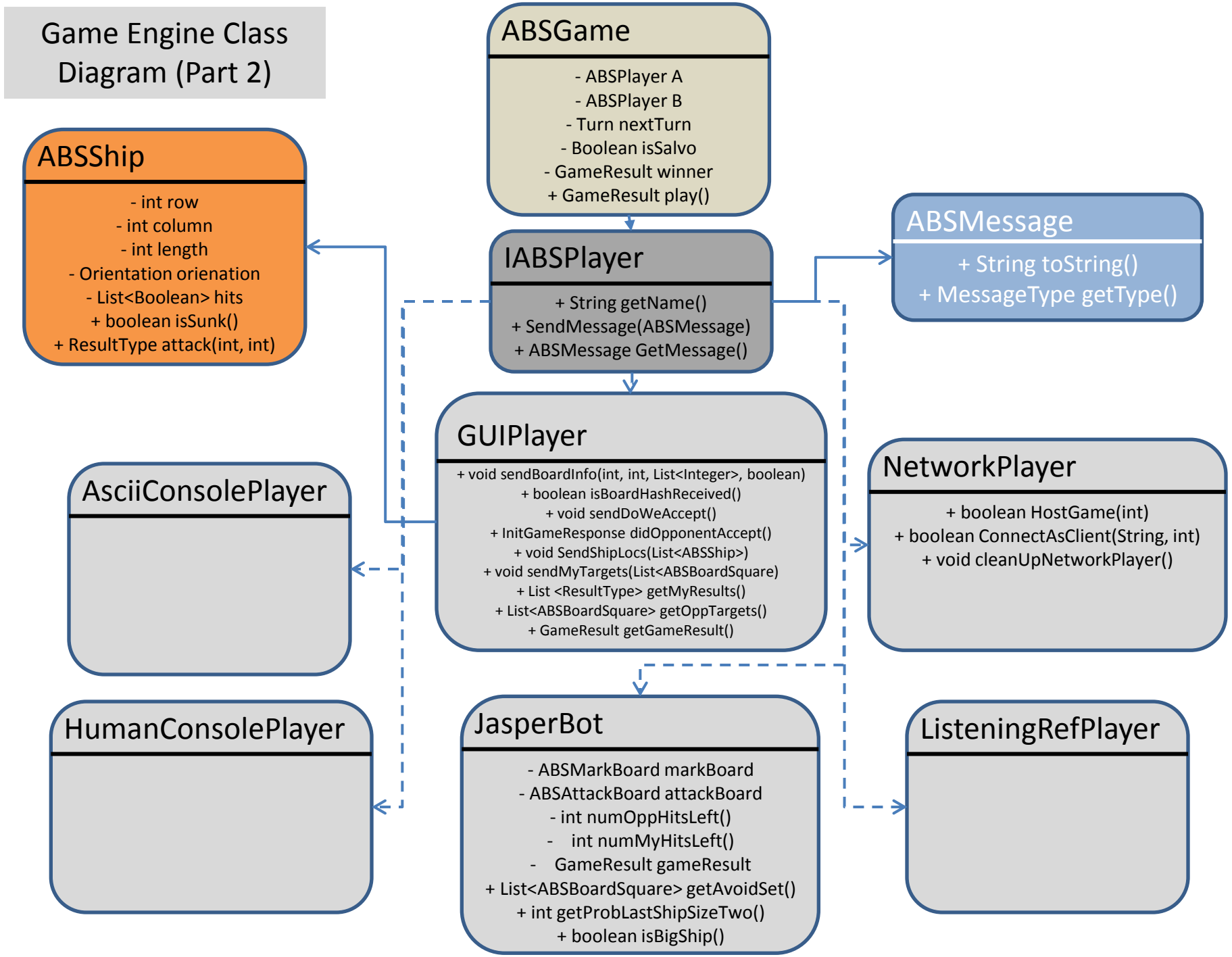
Leader Board components

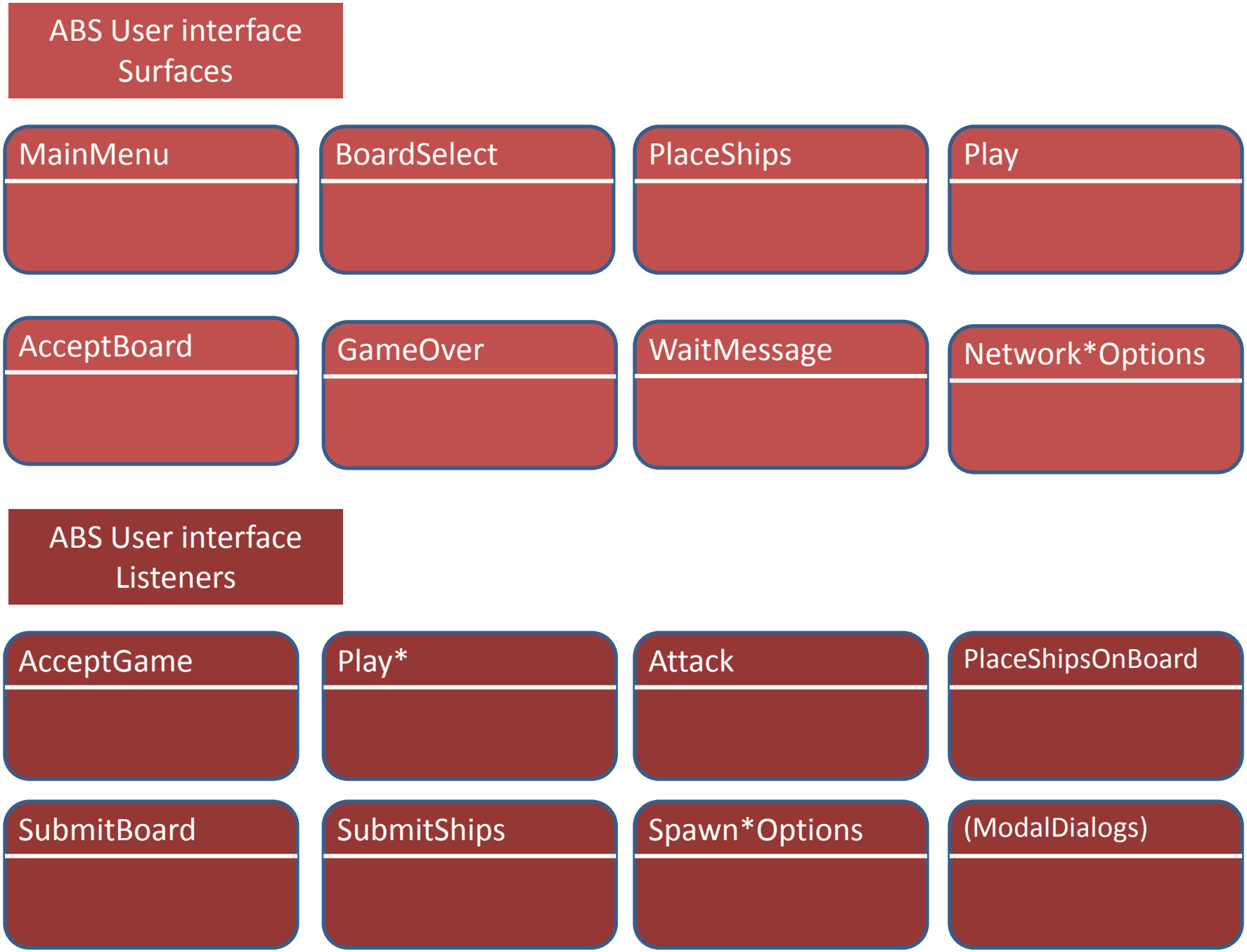


Game Engine Class Diagram (Part 1)



Game Engine Class Diagram (Part 2)




ABS User interface Surfaces
MainMenu
BoardSelect
PlaceShips
Play
AcceptBoard
GameOver
WaitMessage
Network*Options
ABS User interface Listeners
AcceptGame
Play*
Attack
PlaceShipsOnBoard
SubmitBoard
SubmitShips
Spawn*Options
(ModalDialogs)