Javellship A modern rendition of the classic game Battleship in Java and Haskell

Cameron Wheeler	cdwheele@ucsc.edu
Jose Valencia	jolvalen@ucsc.edu
Erick Rodriguez	ergerodr@ucsc.edu
Cesar Kyle Casil	ccasil@ucsc.edu

The game *Battleship* first started as a pencil paper game which dates back to World War I. Various companies published the game in the 1930's until finally being released as a plastic board game in 1967. The game *Battleship* no longer needs to be played face to face; it can be played online against other human players, or against a simple AI. Even though it has been done over and over again, we are here to produce another rendition of a well-known game.

How the official *Battleship* game is played?

The object of the game is to be the first player of two to sink all five of your opponent's ships.

1. To prepare the game for gameplay, each player must place each of their five ships on the playing field (horizontally or vertically) without showing their opponent. The position of these ships will remain the same during the entire round. (The five ships take up five, four, three, three, and two spaces on the ten by ten ocean grid)
2. Once the ships are placed, each player takes a turn by calling out a shot. Each

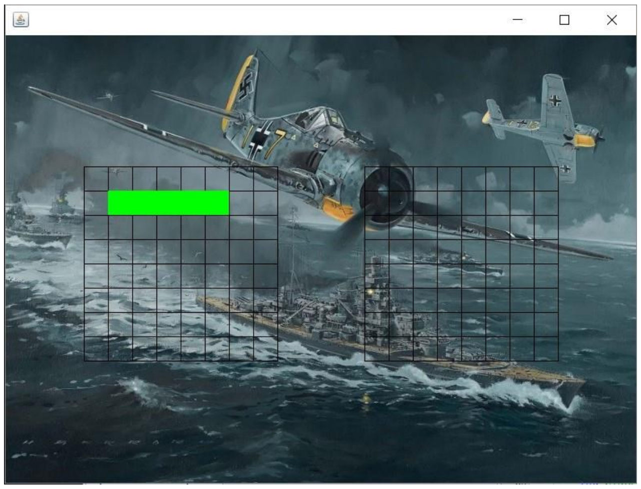
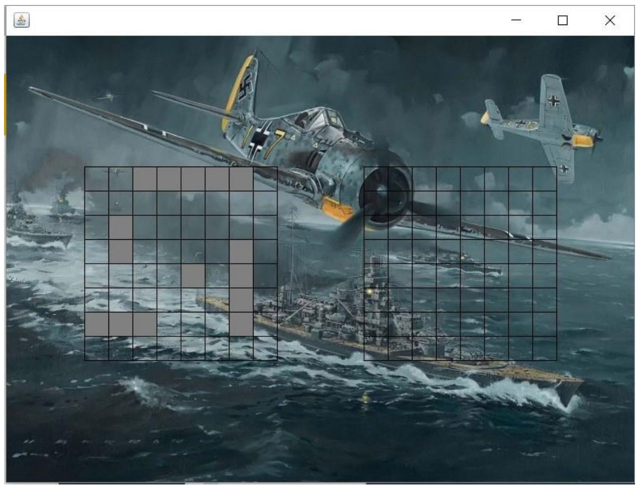
shot is marked by a white peg unless it hits a ship which then it is marked by a red peg.

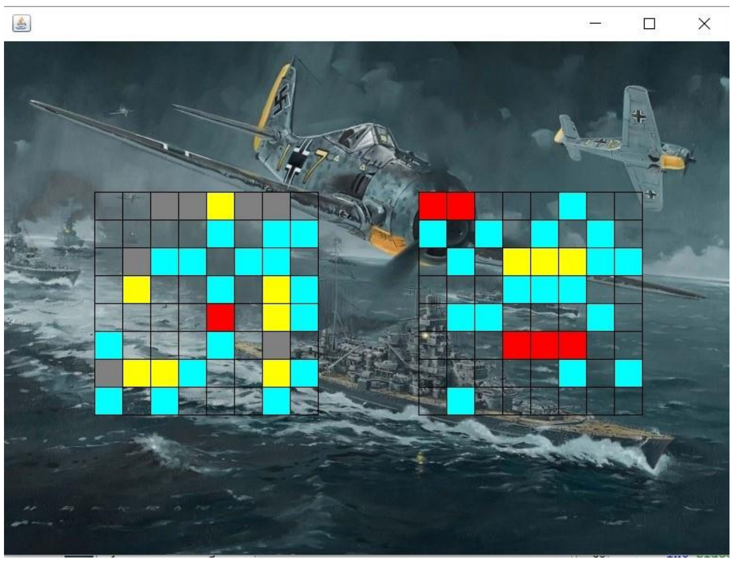
3. By keeping track of hits and misses each player can distinguish the location of the ship and its whereabouts. A ship has sunk once all of the spaces it occupies have been hit.
4. The game continues until one of the two players loses all of their ships. The winner is the player with ships remaining.

Our rendition in Java:

Our remake of the game in java shows some slight differences from the original; however, the gameplay and mechanics stay true. Let's call this replica Javaship. In Javaship, we have an eight by eight grid instead of the regular ten by ten. You may think this hinders the game, but in our implementation, we have one ship that takes up five spaces, one that takes up four spaces, one that takes up three, and another that takes up two spaces. In short, our board and design of the game is different from the original game, but the idea remain the same.

In Javaship we implement a simple AI that shoots at random. For every turn the computer makes, a random shot is made at the player. This AI does not take account of the hits he makes and does not make the choice to shoot in the areas around the hit. This AI could be greatly improved by just adding that one feature; a method to target shots after a shot hits. In other words, when the AI hits a ship, it's next shot would either be above, below, left, or right of the previous shot; in an attempt to keep hitting the ship until it is sunk. In a sense a human player does the same thing, shoots at random until a hit, then shoot at the area around the hits until a ship has sunk. Our issues with this are brought up in depth in the changes category.

	Human player placing first 'ship'.
	Human player places all ships on left grid. Human player uses right grid to shoot.



Left grid:

- Gray (Human ships)
- Blue (AI shots)
- Yellow (AI hits)
- Red (Human sunk ships)

Right grid:

- Blue (Human shots)
- Yellow (Human hits)
- Red (AI sunk ships)

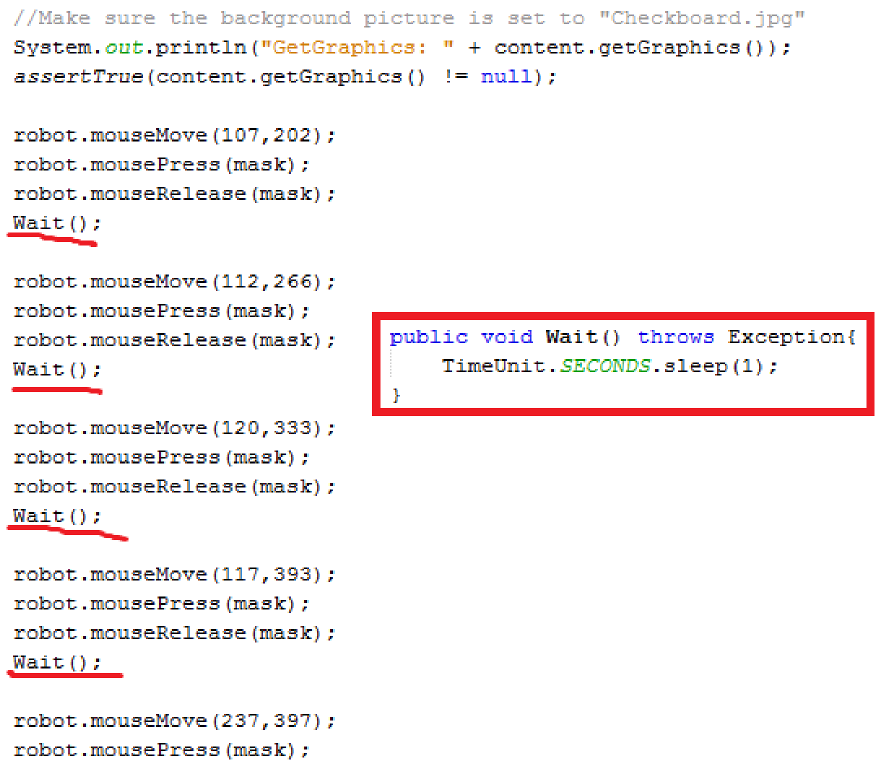
UNIT TESTS IN JAVA

Having no prior background doing any sort of unit testing in either language. We saw how valuable it was to do so. Especially in the haskell portion of code as everything that is typed into the terminal can be lengthy as slow when compared to placing ships with a GUI.

However, one of the large downsides of unit testing with a GUI application was the fact that we had to instantiate both mouse button presses and mouse button releases to test the overall runtime of the application. The *java.awt.robot* object that we used to simulate button clicks was so primitive, it led to verbose testing code and to some unwanted behavior as the button clicks and releases were much too fast. These button clicks were so fast that the ships were not being placed properly when testing, So i used five calls to a Wait() function as seen below.

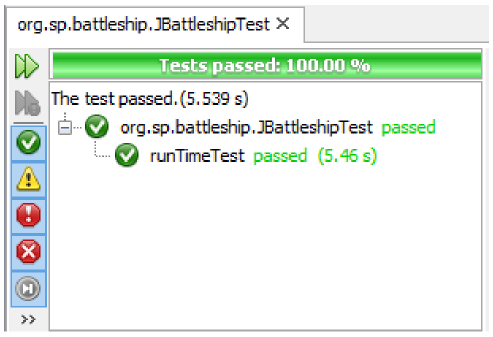
Each of which adds one second to the total runtime, which puts it between 0.378s - 0.46s.

Below is the overall runtime of the application after four separate instances. While there were some libraries that allowed for much more direct manipulation of JFrame/swing object click events, they were much more difficult to use.



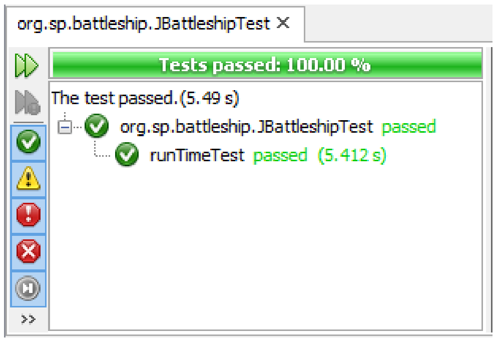
```
//Make sure the background picture is set to "Checkboard.jpg"
System.out.println("GetGraphics: " + content.getGraphics());
assertTrue(content.getGraphics() != null);

robot.mouseMove(107,202);
robot.mousePress(mask);
robot.mouseRelease(mask);
Wait();

robot.mouseMove(112,266);
robot.mousePress(mask);
robot.mouseRelease(mask);
Wait();

robot.mouseMove(120,333);
robot.mousePress(mask);
robot.mouseRelease(mask);
Wait();

robot.mouseMove(117,393);
robot.mousePress(mask);
robot.mouseRelease(mask);
Wait();

robot.mouseMove(237,397);
robot.mousePress(mask);
```

```
public void Wait() throws Exception{
    TimeUnit.SECONDS.sleep(1);
}
```


org.sp.battleship.JBattleshipTest ✕

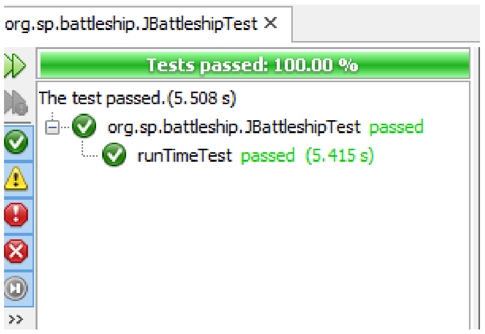
Tests passed: 100.00 %

The test passed.(5.539 s)
- org.sp.battleship.JBattleshipTest passed
  - runTimeTest passed (5.46 s)


org.sp.battleship.JBattleshipTest ✕

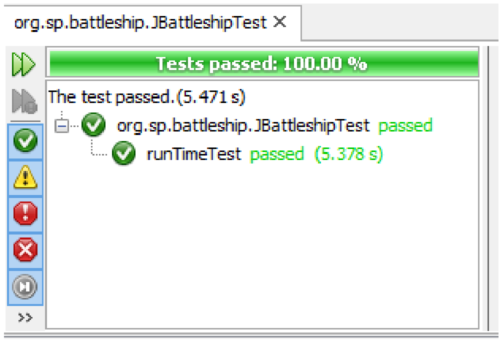
Tests passed: 100.00 %

The test passed.(5.49 s)
- org.sp.battleship.JBattleshipTest passed
  - runTimeTest passed (5.412 s)


org.sp.battleship.JBattleshipTest ✕

Tests passed: 100.00 %

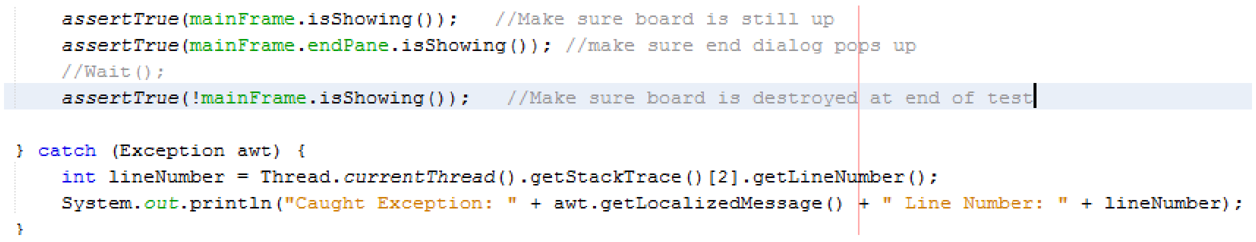
The test passed.(5.508 s)
- org.sp.battleship.JBattleshipTest passed
  - runTimeTest passed (5.415 s)


org.sp.battleship.JBattleshipTest ✕

Tests passed: 100.00 %

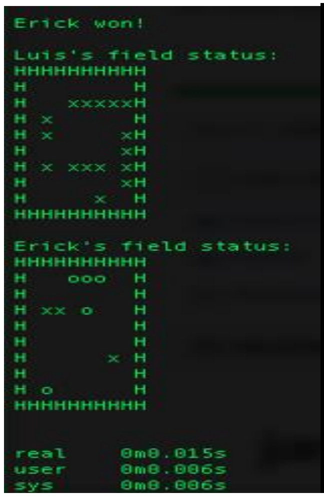
The test passed.(5.471 s)
- org.sp.battleship.JBattleshipTest passed
  - runTimeTest passed (5.378 s)

Unlike the haskell version of our unit test, the java portion is incredibly deterministic as it starts clicking the first square of the enemy row and clicks every single one until either the computer or the end user wins the game. Unit tests in GUI application generally tend to be more involved as we also have certain assertions to ensure that all the application windows are closed at the end of the test as shown below.

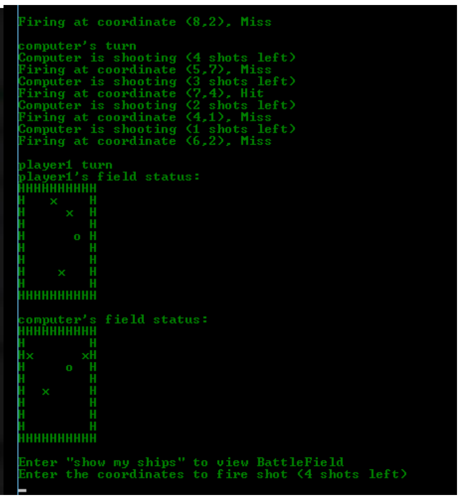

```
    assertTrue(mainFrame.isShowing());    //Make sure board is still up
    assertTrue(mainFrame.endPane.isShowing()); //make sure end dialog pops up
    //Wait();
    assertTrue(!mainFrame.isShowing());    //Make sure board is destroyed at end of test

} catch (Exception awt) {
    int lineNumber = Thread.currentThread().getStackTrace()[2].getLineNumber();
    System.out.println("Caught Exception: " + awt.getLocalizedMessage() + " Line Number: " + lineNumber);
}
```

Our rendition in Haskell:

The version we programmed in Haskell ended up becoming strictly text based. Using a coordinate system, we are able to place ships and make shots. Like in Javaship, we have an eight by eight grid bordered by ‘H’ marks. A slight difference from both *Battleship* and Javaship, Haskellship only uses four ships. The ships used take up five, four, three, and two spaces. Haskellship is based on the popular variation of *Battleship* called SALVO. Since in the beginning of the round we have four ships, we get to shoot four shots; one for each ship. When we sink an opponent's ship, or when one of our ships have sunk, we only get three shots; one for each of the remaining. For example, let's say two of our ships have sunk, we then only have two ships left; which in turn means only have two shots and we type our two coordinates hoping they would hit. In the case we have one ship left, we only get one shot and can only type in one set of coordinates.



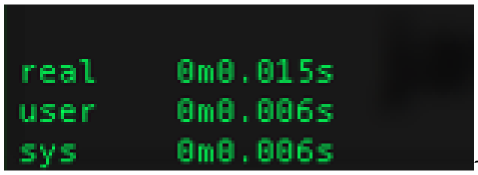
```
Erick won!

Luis's field status:
HHHHHHHHHH
H        H
H   xxxxxH
H x      H
H x     xH
H       xH
H x xxx xH
H       xH
H     x  H
HHHHHHHHHH

Erick's field status:
HHHHHHHHHH
H   ooo  H
H        H
H xx o   H
H        H
H        H
H      x H
H        H
H o      H
HHHHHHHHHH


real    0m0.015s
user    0m0.006s
sys     0m0.006s
```



```
Firing at coordinate (8,2), Miss

computer's turn
Computer is shooting (4 shots left)
Firing at coordinate (5,7), Miss
Computer is shooting (3 shots left)
Firing at coordinate (7,4), Hit
Computer is shooting (2 shots left)
Firing at coordinate (4,1), Miss
Computer is shooting (1 shots left)
Firing at coordinate (6,2), Miss

player1 turn
player1's field status:
HHHHHHHHHH
H   x    H
H     x  H
H        H
H      o H
H        H
H        H
H    x   H
H        H
HHHHHHHHHH

computer's field status:
HHHHHHHHHH
H        H
Hx      xH
H     o  H
H        H
H  x     H
H        H
H        H
H        H
HHHHHHHHHH

Enter "show my ships" to view BattleField
Enter the coordinates to fire shot (4 shots left)
_
```

UNIT TESTS IN HASKELL

As mentioned before, unit testing in GUI applications was incredibly deterministic because the java portion of our project would have to click on every single square on the opponent’s board in the worst case. In haskell however, we simply fed the program the same coordinates and it would finish testing much faster than the java program. There was also no need for any mouse manipulation with “*java.awt.robot*”, but we did have to type out all of the input that our program needed beforehand. We used the **time(1)** call in the bash shell to measure the running time, which was much faster than the java counterpart and much easier to implement as well.



```
real    0m0.015s
user    0m0.006s
sys     0m0.006s
```

--> after running **“time ./Battleship < Coordinates.txt”**

Motivation

The idea behind our project is to create a working implementation of the popular game *Battleship* and see either how much shorter or easier each implementation of the game is. By creating this in multiple languages we will learn how to translate logic and see how the different languages look while doing the same thing.

Java:

Why java?

We all had a good degree of familiarity with Java, and were relatively confident that we could create a working version of Battleship without too many issues. Using Java allowed us to have a baseline comparison for what writing this program should be like, giving us a good knowledge of implementing the logic and creating the data structures in a language that was familiar to us. By writing this in Java first, we hoped to gain intuition that would help us create our Haskell program in a more efficient manner.

Haskell:

Why Haskell?

Compared to Java, we were all substantially less confident in Haskell. We decided to use this language as a test to see how the logic and Graphical User Interface (GUI) translated between the two languages and to compare whether Haskell would be more efficient. As told in class, Haskell could perform the same task in considerably less lines of code; however, figuring out how to perform certain functions made up for the more condensed program.

Project Outline

For the java rendition, we ended up getting a version of Battleship working with an interactive GUI. This version only has the option of playing against an AI; where each player gets one shot per turn. The AI randomly selects where to shoot on the map and the game continues until one player's ships have all been sunk.

The Haskell rendition ended up being run through the command line by inputting coordinates. This version has a player versus AI setting and a player versus player setting. In this rendition, a player gets as many shots per turn as they have ships left unsunk. Again, the AI randomly shoots on the map, and the game ends once all of one of the player's ships have been sunk.

Features

Javaship features the common game *Battleship* interfaced with a GUI so all actions can be done through clicking parts of the interface. This version of the game is extremely user friendly. The left grid in Javaship mimics the bottom grid in the board game while the right grid in Javaship mimics the top grid in the original. The Haskell version implements a similar rendition of *Battleship.* However, this time, through the command line interface. The board is displayed as a series of characters and shots are inputted by typing in coordinates as opposed to clicking squares. In this version, the amount of shots a player gets each turn is determined by the number of ships the player has remaining (not sunk). The Haskell version also has the option to either play against an AI or to play against another human player.

Project Architecture

The data structures we decided to use in our Haskell program were an array of Booleans to represent board tiles, a structure called Coordinate which was simply a pair of ints to give locations to items, and a Ship data type which was an array of coordinates giving the positions of the placed ships. Using these data structures, we were able to create a firing mechanism that first checked whether the entered coordinates where in the scope of the board, then parsed through the Boolean data structure to determine if a shot had already been fired there, and finally comparing with the Ship array to see if the shot was a hit or a miss. This simple data structures were the easiest ones we found to keep track of all the information we needed without creating an unnecessary web of information.

As an overview, the Haskell program takes in user input as a set of coordinates. After taking in a list of coordinates and placing ships on these given places on the board by storing them into an array, it begins to allow shots to be fired. Basically, it switches between two sets of functions, one taking in shots from the human and one creating shots for the AI (in player versus player mode, only the human shot functions would be used). Basically, when shooting, the code takes in a coordinate to represent the shot, and then checks the coordinate against the "field" we have created. If it finds a ship, it branches to say the shot is a hit and saves accordingly, if there is nothing it branches to say it was a miss and again saves the shot, and if the shot is invalid, it loops back and calls for another coordinate (for the human shots only). It continues back and forth until one player has all of their ships sunk and then the game is over.

Changes from Original Plan

Our first change from the original plan came when dealing with the interface for Haskell. After hours of trying to get a GUI to work in Haskell using various platforms such as gtk2hs, grapefruit, and threepenny we decided to scratch the entire thing; after finally getting these to install, the process of implementing them became too tedious.

Our second change came in the implementation of an intelligent AI. Initially, we were planning on having an AI that could keep track of previous shots by referencing them every time it fired to see if they were a hit or miss to determine the next shot. This proved to be less straightforward than we initially thought, as it would end up having to keep a long list of previous shots to accurately keep firing at a ship until it sunk. This is due to the possibility of having misses on a ship after initially hitting it while searching for the direction the ship is in. It was also a challenge to convey to the AI when it had entirely sunk a ship. After our previous setbacks with the Haskell GUIs, we decided to abandon the AI until we had completed the more complicated Haskell version of the code.

Initially in Haskell, we tried saving the fired shots in a 2D array as we did in the Java version. However, this proved unwieldy for a lot of the later functions needed to complete the logic, thus this was quickly abandoned. Then, we tried creating a board data structure, in the form of an array of a data type we called squares. These squares held all of the information; a boolean for a missed shot, for a hit shot, for an un-hit ship, and the coordinates of the square stored as an int. After spending a good amount of time writing the logic, it became clear that this method was not nearly the best way to store this information. So again, we started from the top, this time creating our field to be an array of booleans, as talked about in program architecture.

Conclusion

The biggest fact that we concluded from this project was that Java, in every way, is easier to use that Haskell. Java is more widely used, there is a lot more documentation and there are more tutorials, whereas in Haskell it is hard to find even simple examples for most things. Java also has more support for GUIs, whereas Haskell hardly has any. Contrary to what we originally thought, the Haskell code was not shorter than the code in Java, and the

Haskell code took exponentially longer to create due to multiple setbacks and an overall lesser understanding of the language. Similarly, when reading through the Haskell code, it is harder to determine what is happening, and often times many operations are calling each other or working simultaneously on a line. This made editing the code we had already made in Haskell (such as when changing the data structures) significantly harder than it was in Java. However, if we had as much experience with Haskell as we did in java, and there was more widespread support for the language in documentation and content, Haskell could possibly be a more efficient language to create battleship in, but given the current state of our knowledge and the language, it was not.