Battleship

Team 1 Mt of the Holy Cross
(Caroline Hernandez, Alex Kim, Gabe Madera, Drew McFaul)

Table of Contents

Project Description

Preface:

Create the game ‘Battleship’. A two player game, where each player has a 10 by 10 grid representing an ocean and a fleet of ships. Each fleet is made up of 3 ships with the length; 2, 3, and 4. The players' fleet is hidden to the opponent, where each players’ goal is to sink each ship. Each player has two grids; one based off the opponents grid keeping to see where they fired, hit, and missed, and the other consisting of there fleet and where the opponent has fired. A ship is marked as ‘Sunk’ if each space on the grid has been hit by the opponent. The game ends when a player sinks all of the opponent's ships.

Descrition:

This is a battleship game project created by Gabriel Madera, Drew McFaul, Alex Kim, and Caroline Hernandez. This project allows you to play a full game of battleship, where each player has a fleet of 5 ships and exciting abilities at their disposal. There is a high score system based on the number of turns taken to win. This project uses test driven development and incorporates various object oriented practices and software design patterns.

Development Process

Our program was developed iteratively. We began by creating CRC cards to understand the classes we needed, and their responsibilities and communications with other classes. Then we wrote test cases which the implementation was based on.

Iterative Development:

The development of our software was broken down into small chunks, and each part was completed throughout the semester. Our code was repeatedly designed, developed and tested as we completed new milestones. We ran into some issues with iterative development: missing relationships between classes, having too many responsibilities for certain classes and needing new classes, needing better ways to represent column and row combinations, identifying head from tail on ship, adding the ability to move one's ship, creating attributes for hits rather than lists, moving ship coordinates to grid class. The most prominent issue our team experienced due to iterative development was moving the ship coordinates out of the ship class and into the grid class. It seemed like a no brainer for the ship class to hold each player's ship coordinates, however we needed to abstract the coordinates out of the ship class and into the grid class for loosely coupled code. Additionally, our group sometimes implemented new features in certain classes due to the relationships that class had, however to ensure a highly cohesive program we found ourselves needing to make new classes like the Weapon class.

Refactoring:

Code refactoring was needed, as the project grew and more features were implemented. All of the refactoring we did worked in our favor, as some previously implemented code became simpler and declarations became clearer. We renamed our direction variables by creating a direction map for N, S, E, W which allowed for better readability and clarity of understanding.

Another notable code refactor was creating a parameter object to the updateBoards method during which we populated the Weapon class with the different kinds of hits. These changes did not change the functionality or external behavior of the game, but made our code more efficient and maintainable.

Testing:

https://www.tutorialspoint.com/junit/junit_suite_test.htm

We used test-driven development (TDD). TDD is a process that relies on software requirements being converted to test cases before software is developed. Then, implementation is based on tests that were written. All software development is then tracked by repeatedly testing the software against all test cases. We used a Java test suite to ensure full coverage of the code. Test suite is used to bundle a few unit test cases and run them together. We wrote the tests sending in mock values to the functions and checked for the correct response. The tests help identify what the code specifically needs to do as they returned or printed different values. Test-driven development helped the implementation of our project, because it allowed us to have better program design, higher code quality, and more reusable code.

Collaborative Development:

This project had 4 people working to create the product. We had a set meeting time each week but frequently found more time throughout each week to meet when all members were free to work together. The whole project was developed in team meetings where each member was present and we'd code together on a shared version of the project through. We developed collaboratively with Intellij's code with me plugin, because each person could see other's changes and make their own changes to the code in real time. Additionally, we would zoom so we could talk through what was being developed at the moment. Thanks to our team contract we didn’t run into many issues with collaborative development, as each person adhered to our Java coding standards and prioritized attending group meetings for development.

Coding Style:

We follow the Google Java coding standard which is described in:

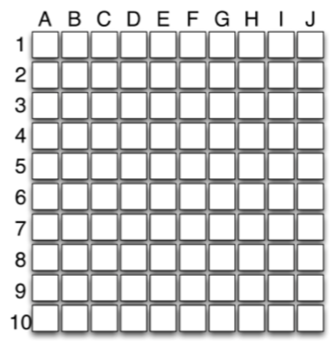
https://google.github.io/styleguide/javaguide.html

Google’s Java coding standard ensures clarity and readability in the code by requiring: braces even when their optional (ie. braces are used with if, else, for, do and while statements, even if the body is empty or only contains a single statement), no line breaks before the opening brace, clear indentation when a new block of code is started, and more.

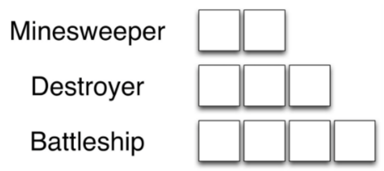
Requirements and Specifications

Requirements

- Battleships is a classic 2 player game.
- Each player has a 10 by 10 grid that represents the ocean their fleet is in.
    - The figure below shows the coordinate system used by the game.

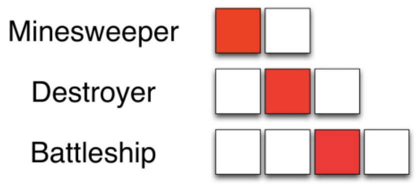


- Each player has the following 3 ships in their fleet, which take up the specified number of adjacent spaces on the grid:

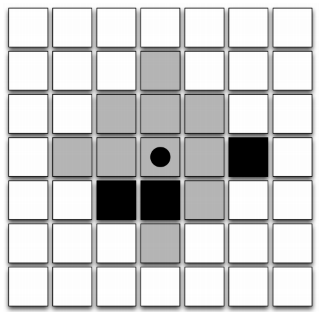


- Ships can be placed horizontally or vertically on the grid, but not diagonally.
- Each player should see two grids, one with their own fleet and where the opponent has fired.
    - Each player positions their fleet on their own grid. These positions are kept secret.
    - The other grid shows what is known so far about the opponent's fleet. That is, where hits and misses have been made so far.

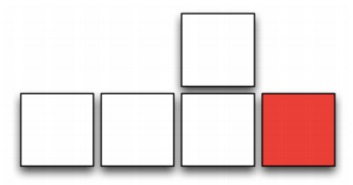
- The players take turns to shoot at their opponent's fleet by telling them the coordinates they are attacking. An attack result is either a “miss”, “hit”, or “sunk X” (where X is the type of ship).
- A ship is sunk once every space it occupies on the grid has been hit.
- The game continues until one player's entire fleet has been sunk.
- Developers calculate the result of an enemy attack on the fleet: given a set of coordinates of the attacked position, deduce the result as one of the options:
    - MISS / HIT
    - SUNK (and determine the type of ship sunk)
    - SURRENDER (all ships have been sunk).
- Captain’s Quarters - If the captain’s quarters are “hit”, the entire ship sinks, regardless of the status of its other elements.
- the captains quarters for battleships and destroyers (but not for minesweepers!) are armored, meaning that it takes two attacks on the same square in order to “hit” it (i.e. the result of the first attack always counts as “missed”).
    - The captain’s quarters are located as follows:



- “Sonar Pulse” Weapon - The sonar pulse allows a player to reveal a portion of the map, as suggested in the figure below. The sonar pulse merely reveals the status of the cell as being free (grey) or occupied(black), but it does not reveal the type of ship or the location of the captain’s quarters. A player can use a total of two (2) sonar pulses in a game, and only AFTER successfully sinking the first enemy ship.



- A new type of ship will be introduced: the submarine. A submarine can be placed on the surface or it can be submerged, and thus be placed on the grid under any other type of surface ship. It takes up five blocks on the grid, as follows:



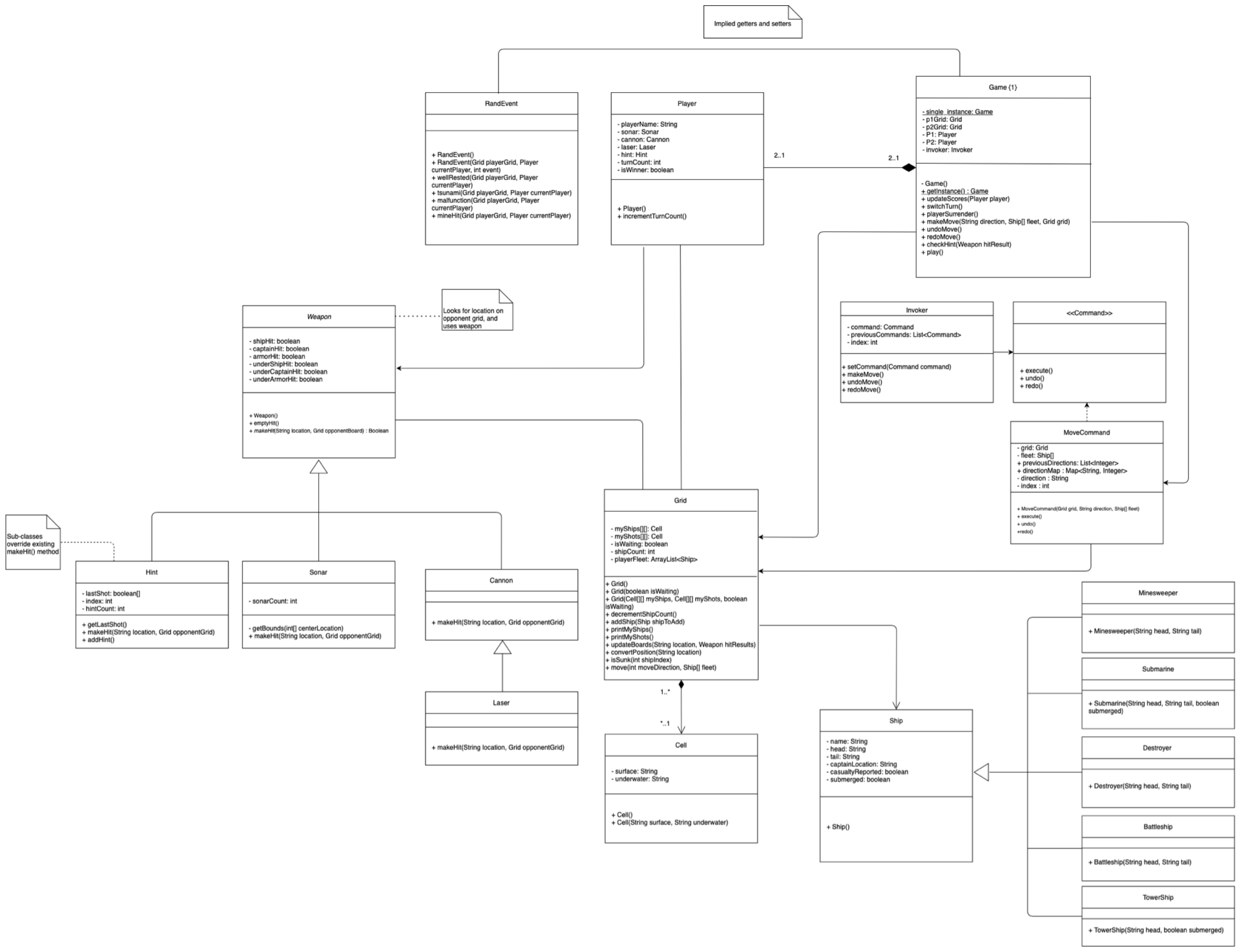
- “Space Laser” Weapon - It is fired from a network of geostationary satellites, and unlike the conventional bombs that we have until now, they penetrate the water, being able to hit both a surface ship, and a sub that is placed below it at the same time (i.e. in contrast the bomb can only hit the surface). The player receives the activation codes for the space laser only after sinking the first enemy ship (i.e. this weapon is an upgrade, and replaces the conventional bomb in the player’s arsenal).
- A player can now move his fleet. When the command to move is given, each ship will move one position. The player can specify the direction of the move (N, S, E, W). North is towards the top of the board. If a ship is already at the edge, it will not move. We also want the ability to undo and redo the moves. The game must support multi level undo.

User Stories

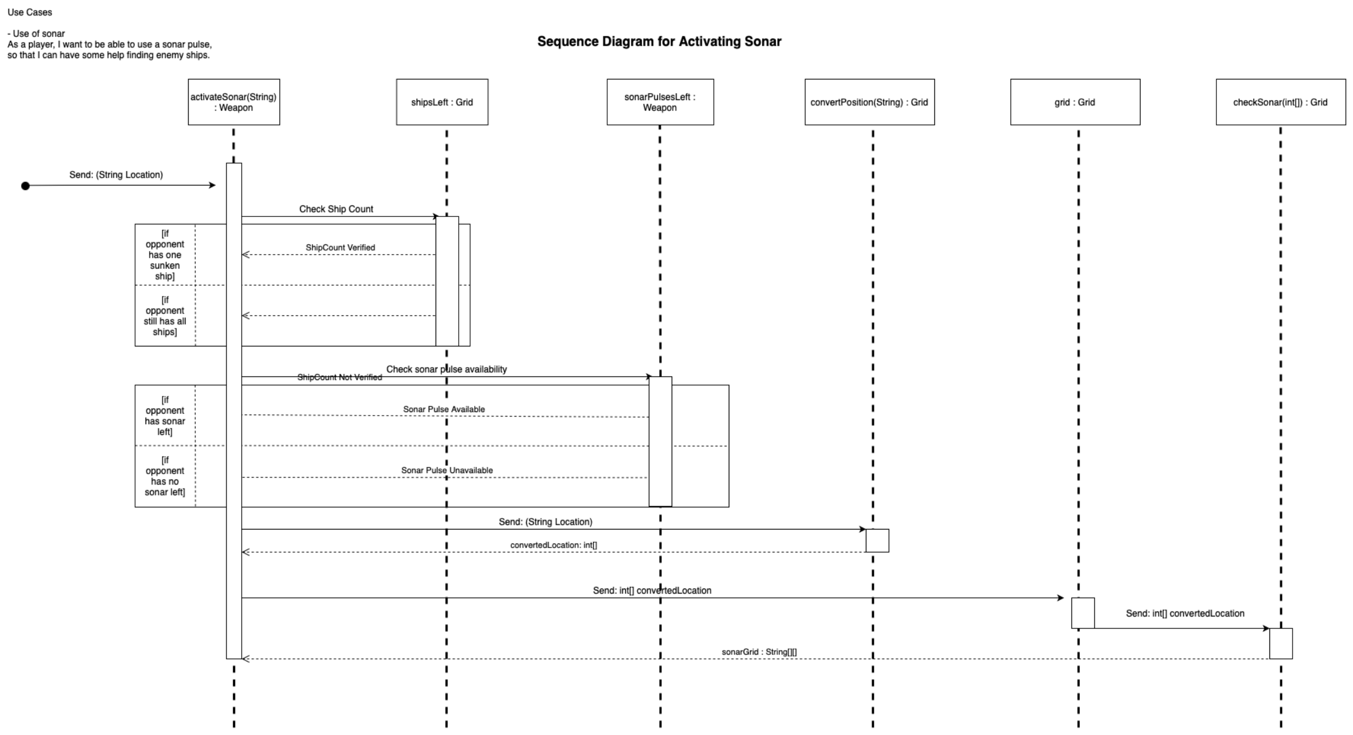
- As a player, I want to choose where to place ships, so the opponent has trouble finding them.
- As a player, I can see two grids, one with my own fleet and where the opponent has fired. The other grid shows what is known so far about the opponent's fleet so I can keep track of game state.
- As a player, I can shoot at their opponent's fleet by telling them the coordinates resulting in a "miss", "hit", or "sunk X" if every space of the opponents ship has been hit so the game can progress.
- As a player, I want to know if my opponent has surrendered, so that I can win the game.
- As a player, I want to be able to hit the captain's quarters, so that I can have some advantage over my enemy.
- As a player, I want to be able to use a sonar pulse, so that I can have some help finding enemy ships.
- As a player, I want to have a submarine ship, so that I can place ships below one another.
- As a player, I want to have a space laser weapon, so that I can sink a submarine.
- As a player, I want to be able to move my ships, so that I have a better chance of not getting hit.
- As a player, I want to be able to have a TowerShip, so that I can place size two ship on one cell on the board.
- As a player, I want to be able to get Hints, so that I can get some help winning the game.
- As a player, I want to have my scores saved, so that I can see how my scores are compared to others.
- As a player, I want random events to take place, so that the game can stay interesting.
- As a player, I want to have a GUI, so that there is a different way to play the game.

Architecture and Design

UML diagrams


Implied getters and setters
RandEvent
+ RandEvent()
+ RandEvent(Grid playerGrid, Player currentPlayer, int event)
+ wellRested(Grid playerGrid, Player currentPlayer)
+ tsunami(Grid playerGrid, Player currentPlayer)
+ malfunction(Grid playerGrid, Player currentPlayer)
+ mineHit(Grid playerGrid, Player currentPlayer)
Player
- playerName: String
- sonar: Sonar
- cannon: Cannon
- laser: Laser
- hint: Hint
- turnCount: int
- isWinner: boolean
+ Player()
+ incrementTurnCount()
2..1
2..1
Game {1}
- single_instance: Game
- p1Grid: Grid
- p2Grid: Grid
- P1: Player
- P2: Player
- invoker: Invoker
- Game()
+ getInstance() : Game
+ updateScores(Player player)
+ switchTurn()
+ playerSurrender()
+ makeMove(String direction, Ship[] fleet, Grid grid)
+ undoMove()
+ redoMove()
+ checkHint(Weapon hitResult)
+ play()
Weapon
- shipHit: boolean
- captainHit: boolean
- armorHit: boolean
- underShipHit: boolean
- underCaptainHit: boolean
- underArmorHit: boolean
+ Weapon()
+ emptyHit()
+ makeHit(String location, Grid opponentBoard) : Boolean
Looks for location on opponent grid, and uses weapon
Sub-classes override existing makeHit() method
Hint
- lastShot: boolean[]
- index: int
- hintCount: int
+ getLastShot()
+ makeHit(String location, Grid opponentGrid)
+ addHint()
Sonar
- sonarCount: int
- getBounds(int[] centerLocation)
+ makeHit(String location, Grid opponentGrid)
Cannon
+ makeHit(String location, Grid opponentGrid)
Laser
+ makeHit(String location, Grid opponentGrid)
Invoker
- command: Command
- previousCommands: List<Command>
- index: int
+ setCommand(Command command)
+ makeMove()
+ undoMove()
+ redoMove()
<<Command>>
+ execute()
+ undo()
+ redo()
MoveCommand
- grid: Grid
- fleet: Ship[]
+ previousDirections: List<Integer>
+ directionMap : Map<String, Integer>
- direction : String
- index : int
+ MoveCommand(Grid grid, String direction, Ship[] fleet)
+ execute()
+ undo()
+redo()
Grid
- myShips[][]: Cell
- myShots[][]: Cell
- isWaiting: boolean
- shipCount: int
- playerFleet: ArrayList<Ship>
+ Grid()
+ Grid(boolean isWaiting)
+ Grid(Cell[][] myShips, Cell[][] myShots, boolean isWaiting)
+ decrementShipCount()
+ addShip(Ship shipToAdd)
+ printMyShips()
+ printMyShots()
+ updateBoards(String location, Weapon hitResults)
+ convertPosition(String location)
+ isSunk(int shipIndex)
+ move(int moveDirection, Ship[] fleet)
1..*
*..1
Cell
- surface: String
- underwater: String
+ Cell()
+ Cell(String surface, String underwater)
Ship
- name: String
- head: String
- tail: String
- captainLocation: String
- casualtyReported: boolean
- submerged: boolean
+ Ship()
Minesweeper
+ Minesweeper(String head, String tail)
Submarine
+ Submarine(String head, String tail, boolean submerged)
Destroyer
+ Destroyer(String head, String tail)
Battleship
+ Battleship(String head, String tail)
TowerShip
+ TowerShip(String head, boolean submerged)

Sequence Diagram - Activate Sonar


Use Cases

- Use of sonar
As a player, I want to be able to use a sonar pulse, so that I can have some help finding enemy ships.

Sequence Diagram for Activating Sonar
activateSonar(String) : Weapon
shipsLeft : Grid
sonarPulsesLeft : Weapon
convertPosition(String) : Grid
grid : Grid
checkSonar(int[]) : Grid
Send: (String Location)
Check Ship Count
[if opponent has one sunken ship]
ShipCount Verified
[if opponent still has all ships]
Check sonar pulse availability
ShipCount Not Verified
[if opponent has sonar left]
Sonar Pulse Available
[if opponent has no sonar left]
Sonar Pulse Unavailable
Send: (String Location)
convertedLocation: int[]
Send: int[] convertedLocation
Send: int[] convertedLocation
sonarGrid : String[][]

Top-Down Description of System

This program is fully coded in Java. Our game is fully interactable and uses command line I/O and a GUI to display each player's grids. The project holds all the Java code to run the logic of the game and also uses Java code (GUI) to display the boards.

Personal reflections

Caroline: This project has been a really good experience for me, as I got to learn a lot of new skills including Java, test suites (&TDD), GUI, OOP analysis and design, and more. I had never been a part of such a big project that was coded with TDD, and I'm so glad I have this skill now. My team and I were very devoted to producing the best code we could, and always prioritized meeting together as a group to get work done. A huge thank you to my amazing team members, the TA's, and the professor!

Alex: I definitely underestimated this project in the beginning of the semester. This project has given me more experience in collaborating with my peers and a huge part of completing the project was because of the team's efforts. I wanted this class to be challenging and heavy in Object Oriented Programming since I haven't done so in an extended period of time. I was able to learn and develop new skills in unit testing, TDD, and GUI. I'm a advocate on project based learning and this class did just that.

Drew: This was a tough project, but I learned a lot. I disliked TDD at the start of the semester but I grew to like it and now I think it's an effective way to program. Working on this project has given me good practice in building something from the ground up and seeing it through to the end. Like Gabe, I think that the milestone write-ups could've been clearer, but I also appreciate the freedom that a project of this scale provides and I understand that it's hard to balance the two. I feel really well versed in Java and was able to dip my toes into unit testing, so that's good.

Gabe: Overall, I was happy with the class project. When signing up for the course, I was looking for something that would help me get a better understanding of Java, and Object-Oriented Programming. This class project definitely did that. Though, I would say I was frequently frustrated with the lack of instructions in Milestone write-ups. Nevertheless, the project taught us to work using TDD, helped us figure out unit testing, and many other things. Overall, the project was good.

Appendix

Requirements to run this game and its tests include:

- Java 1.8.0 or higher

  (https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html)
- junit 1.4 or higher (https://junit.org/junit4/)
- Open terminal and run:
  - git clone https://github.com/cahe6160/OOADProject.git
- Running main should start a game.