PA 04 - BattleSalvo - Part 2

Title	BattleSalvo Part 2
Issues Due Date	@June 6, 2023 10:00 PM
EC Due Date	@June 11, 2023 10:00 PM
Due Date	@June 12, 2023 10:00 PM
GitHub Classroom Link	[redacted]

💟 Please read the ***entire*** assignment description before jumping into the UML or implementation parts of the PA.

Table of Contents

🤖 Background
👾 Recap
- ✅ **Key Takeaways**
- 🚧 Important Note

🔀 Remote Interactions Protocol
- 🔌 Connecting to Server Protocol

- ♻️ Game Loop Protocol
- 🖨️ JSON: Server-Client Communications
  - ✉️ JSON Message Formats
  - 🗃️ Special JSON Object Data Definitions
- ♟️ Supporting Interactions: How to?
- 🏗️ Setting up
  - 📦 Project Structure
  - 🌐 Local Server
- 🚚 Deliverables
  - 🐶 Partners (@June 5, 2023 10:00 PM)
  - 🎨 Design (@June 6, 2023 10:00 PM)
  - 🛠️ Implementation (@June 12, 2023 10:00 PM)
- ✨ Hints
  - 🌎 **General**
  - 👉 **PA Specific**

🤖 Background

Now that your BattleSalvo game works locally, it's time to add multiplayer support using Sockets.

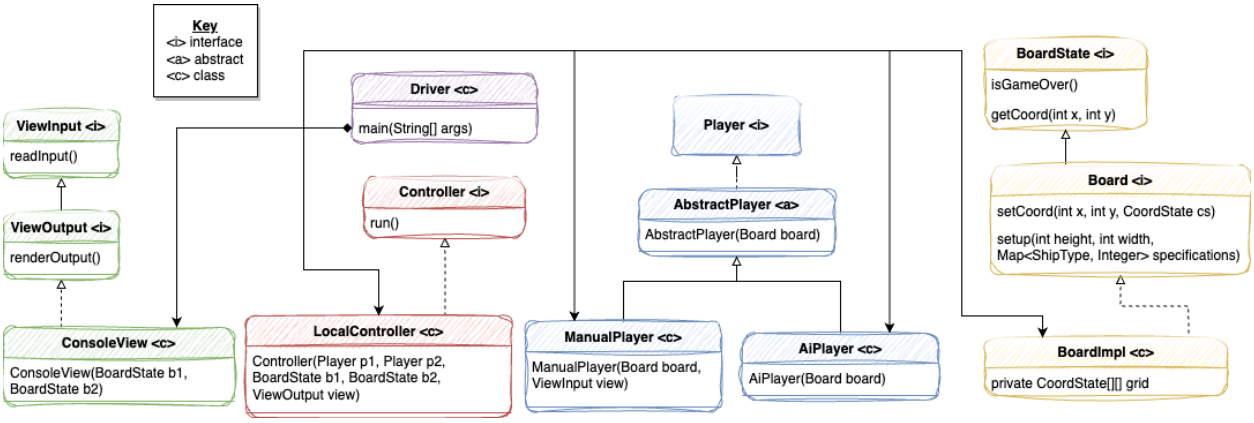
It'd be far too time consuming to play multiplayer BattleSalvo yourself, so you'll be turning to an automated solution: bolstering your BattleSalvo game to embrace bot-fighting to reliably destroy TA- and student-created BattleSalvo fleets.

What you'll be doing, but BattleSalvo.

👾 Recap

In PA03 you created a BattleSalvo game where a human player could compete against a CPU. Because you followed SOLID principles and the MVC design pattern, it should be straightforward to substitute modules of your program, such as the players.

To make sure you have a strong foundation to build from (and in lieu of being able to instantly grade PA03), here is a rough structure for ***one of many*** possible PA03 implementations:



This is an abbreviated UML class diagram. Focus on how the classes and interfaces relate to each other, not on each individual method necessary.

~~~

`renderOutput()` is acting as shorthand for the many different print methods you would want in the `ViewOutput` interface.

✅ Key Takeaways

- **S:** Separating into Model, View, and Controller responsibilities ensures modules can be easily substituted
- **O:** Interfaces uses generalized naming so as to not limit the scope of extension. See `renderOutput()` in `ViewOutput`
- **L:** `ManualPlayer` and `AiPlayer` do not contain methods not in their abstract class and interface. Instead, take advantage of the constructor to introduce getters through a new class (the `BoardImpl`).
- **I**: Take advantage of Interface Segregation to give `Board` and View to other classes without breaking MVC responsibilities. MVC is a tool which helps us ensure we're following Single-responsibility, not a hard-and-fast rule.
  This technique was not required for PA03; do not worry if you did not make use of it.
- **D:** Instantiate classes in the `Driver` and then inject them to many other classes so that multiple classes can reference the same data models ("sources of truth").

  ▼ Example `main()` method
~~~

```
public static void main(string[] args) {
  Board userBoard = new Board();
  Board aiBoard = new Board();

  Player user = new Player(userBoard);
  Player ai = new Player(aiBoard);

  ViewOutput view = new ConsoleView(userBoard, aiBoard);

  Controller controller = new LocalController(
    user, ai, userBoard, aiBoard, view)
}
```

🚧 Important Note

This is a diagram of one of many ideal solutions. If your implementation does not follow this structure, you should still be able to implement PA04. If the route towards implementing PA04 is incredibly unclear, it may be worth touching up your PA03 foundation before moving forward and extending it to implement PA04.

Please talk to a TA if you feel you need to make major changes and are unsure how to proceed.

🔀 Remote Interactions Protocol

To support playing multiplayer BattleSalvo, your implementation will need to match the process format the server expects of all BattleSalvo clients. These formats are called *protocols*.

Below are some diagrams which show the protocols your program must support to interact with our server. Note how these are similar to the logic in your PA03 controller.

▼ 🔌 Connecting to Server Protocol

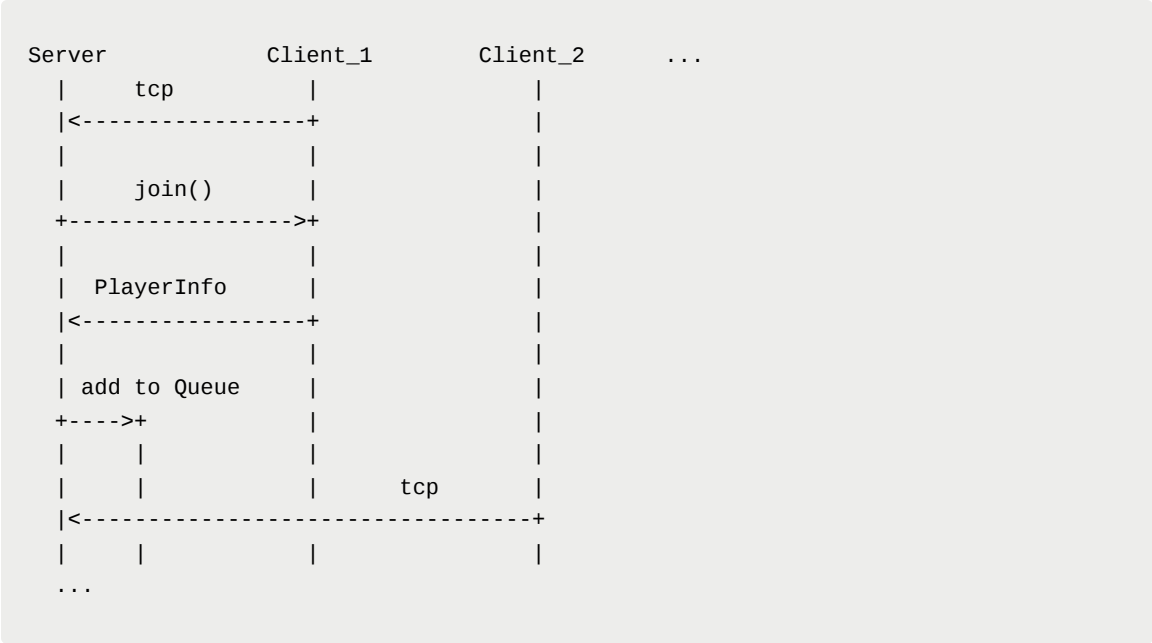
This is the connection protocol:

1. Each client will connect to the server using a `Socket`. (Review Lab 06 for a Socket refresher)
    a. `Socket` will take care of following Transmission Control Protocol (TCP) — this is how the connection is actually made.

b. If you want to learn more, here's a Khan Academy article which provides some explanation. Networks (CS 3700) is also a class your should look into taking if you find this interesting.

2. The server will request the client to join a game
3. The client will respond with the player name and the preferred game type (more on this later)
4. The server will add the player to the join queue and start a game when available

Below is a UML Sequence Diagram on how the connection protocol works for each client:



▼ ♻️ Game Loop Protocol

The game loop is comprised of three stages (we also provide UML Sequence Diagrams for each):

1. **The setup phase**

   Each player is given the dimensions of the grid and will return a Fleet of Ships with each Ship's coordinates on the grid.

```
  +-------------------->|===========\\\\==========>|--------------->|
  |                     |           \\\\           |     Fleet      |
  |<--------------------|<==========\\\\===========|<---------------+
  |                     |           \\\\           |                |
  ...
```

2. **The play phase**

   Each client will receive the following three messages at roughly the same time for each round. Note how similar this is to your own controller implementation.

```
Server               ProxyPlayer     \\\\      ProxyController    Player
  |                      |           \\\\           |                |
  | takeShots()          |           \\\\           |                |
  +--------------------->|===========\\\\==========>|--------------->|
  |                      |           \\\\           |   List Coord   |
  |<---------------------|<==========\\\\===========|<---------------+
  |                      |           \\\\           |                |
  | reportDamage(coords) |           \\\\           |                |
  +--------------------->|===========\\\\==========>|--------------->|
  |                      |           \\\\           |                |
  |                      |           \\\\           |   List Coord   |
  |<---------------------|<==========\\\\===========|<---------------+
  |                      |           \\\\           |                |
  |successfulHits(coords)|           \\\\           |                |
  +--------------------->|===========\\\\==========>|--------------->|
  |                      |           \\\\           |                |
  |                      |           \\\\           |   void         |
  |<---------------------|<==========\\\\===========|<---------------+
  ...
```

3. **The scoring phase**

   When the game is over, each player is notified if they won, lost, or tied the game, along with a rationale.

```
Server              ProxyPlayer     \\\\      ProxyController     Player
  |                     |           \\\\           |                |
  |endGame(Result, Reas)|           \\\\           |                |
  +-------------------->|===========\\\\==========>|--------------->|
  |                     |           \\\\           |                |
  |                     |           \\\\           |   void         |
  |<--------------------|<==========\\\\===========|<---------------+
  ...
```

🖨️ JSON: Server-Client Communications

Messages between the client and server which follow the above protocols are sent using JSON. We introduced JSON during Lab 06; as a reminder JSON is the format we are using to encode objects. This ensures a standardized and interoperable encoding format between the server and the client so they can interpret each other's messages.

▼ The messages between the server (TA code) and client (your code) will all be based on the structure shown below:

```
{
  "method-name": "SOME METHOD NAME",
  "arguments": {
    ...
  }
}
```

💡 If your implementation produces a response that the server does not expect (malformed JSON or invalid arguments), then the server controller will handle the error by stopping the game.

▼ ✉️ JSON Message Formats

There are **six (6)** different message types based on MessageJson (given here) that your `ProxyController` needs to be able to handle and delegate to your `AiPlayer`:

▼ 1. `join`

- server request:

```
{
  "method-name": "join",
  "arguments": {}
}
```

- client response:

```
{
  "method-name": "join",
  "arguments": {
    "name": "github_username",
    "game-type": "SINGLE"
  }
}
```

📌 Notes:

- `name` must be your **GitHub PA04 repo/team name** to receive credit for your AI Player absolutely demolishing the competition.
  - *Do **NOT** change your repo name from when you initially created it.*
- `game-type` will be either `“SINGLE”` or `“MULTI”` but not both.

▼ 2. `setup`

- sample server request:

```
{
  "method-name": "setup",
  "arguments": {
    "width": 10,
    "height": 10,
    "fleet-spec": {
      "CARRIER": 2,
      "BATTLESHIP": 4,
      "DESTROYER": 1,
      "SUBMARINE": 3
    }
  }
}
```

- sample client response:

```
{
  "method-name": "setup",
```

```
  "arguments": {
    "fleet": [
      {
        "coord": {"x": 0, "y": 0},
        "length": 6,
        "direction": "VERTICAL"
      },
      {
        "coord": {"x": 1, "y": 0},
        "length": 5,
        "direction": "HORIZONTAL"
      }
    ]
  }
}
```

▼ 3. `take-shots`

- sample server request:

```
{
  "method-name": "take-shots",
  "arguments": {}
}
```

- sample client response:

```
{
  "method-name": "take-shots",
  "arguments": {
    "coordinates": [
      {"x": 4, "y": 2},
      {"x": 7, "y": 1}
    ]
  }
}
```

▼ 4. `report-damage`

- sample server request:

```
{
  "method-name": "report-damage",
  "arguments": {
    "coordinates": [
```

```
      {"x": 0, "y": 1},
      {"x": 3, "y": 2}
    ]
  }
}
```

- sample client response:

```
{
  "method-name": "report-damage",
  "arguments": {
    "coordinates": [
      {"x": 3, "y": 2}
    ]
  }
}
```

▼ 5. `successful-hits`

- sample server request:

```
{
  "method-name": "successful-hits",
  "arguments": {
    "coordinates": [
      {"x": 0, "y": 1},
      {"x": 3, "y": 2}
    ]
  }
}
```

- sample client response:

```
{
  "method-name": "successful-hits",
  "arguments": {}
}
```

▼ 6. `end-game`

- sample server request:

```
{
  "method-name": "end-game",
  "arguments": {
    "result": "WIN",
    "reason": "Player 1 sank all of Player 2's ships"
  }
}
```

- `result` can be `“WIN”` or `“LOSE”` or `“DRAW”`
- `reason` can be any reasonable description.

- sample client response:

```
{
  "method-name": "end-game",
  "arguments": {}
}
```

▼ 🗃️ Special JSON Object Data Definitions

At first glance, the arguments of each message can be quite confusing. However, each expected client response only uses the following **four (4)** JSON data definitions for the message arguments:

▼ 1. `coord`

A Coord (coordinate) is a JSON object with two fields (`x` and `y`).

- `x` represents the **column index** this Coord represents on a board
- `y` represents the **row index** this Coord represents on a board

> 💡 This coordinate system starts with (0,0) at the top left. A board coordinate may not be negative.

The JSON structure looks like:

```
{
  "x": 0,
  "y": 4
}
```

2. `coordinates`

Coordinates (previously referred to as a “volley”) is an array of Coords.

The JSON structure looks like:

```
[
  {"x": 4, "y": 2},
  {"x": 7, "y": 1}
]
```

3. `ship`

A Ship is a JSON object with three fields (`coord`, `length`, and `direction`).

- `coord` represents the starting Coord of the ship
- `length` represents the length of the ship
- `direction` is one of `HORIZONTAL`. or `VERTICAL`. It represents the direction the ship is facing

The JSON structure looks like:

```
{
  "coord": {"x": 0, "y": 0},
  "length": 4,
  "direction": "VERTICAL"
}
```

> 💡 **Note**: The example JSON ship representation above shows a ship which occupies the following coordinates: `(0, 0), (0, 1), (0, 2), (0, 3)`.
>
> The same ship with a `HORIZONTAL` direction would occupy: `(0, 0), (1, 0), (2, 0), (3, 0)`

▼ 4. `fleet`

A Fleet is an array of Ships.

The JSON structure looks like:

```
[
  {
    "coord": {"x": 0, "y": 0},
    "length": 6,
    "direction": "VERTICAL"
  },
  {
    "coord": {"x": 1, "y": 0},
    "length": 5,
    "direction": "VERTICAL"
  }
]
```

♟️ Supporting Interactions: How to?

You will develop the `ProxyController` class which parses incoming messages from the server, delegates the incoming message to the correct `Player` method, parses the response, and sends the response back to the server.

Your `ProxyController` will follow the protocols described in the ***Remote Interactions Protocol*** section using JSON objects which follow the formats described in the ***JSON: Server-Client Communications*** section.

The `ProxyController` should take in a `Socket` connection to the server and a `Player` implementation you want the `ProxyController` to use to play against the server. The `host` and `port` needed to create the Socket will be given as command-line arguments.

▼ As a very bare-bones guide for setting up a `ProxyController` in `main(…)`:

```
// In Driver.java

public static void main(String[] args) {
  // ... part of your implementation ...

  Controller controller;
  Socket socket;
  Player player;

  // ... part of your implementation ...

  controller = new ProxyController(socket, player);
  controller.run();

  // ... part of your implementation ...
}
```

💡 Your `ProxyController` will follow an implementation similar to the `ProxyDealer` in Lab 06. It's recommended you review this code before starting your own implementation.

🏗️ Setting up

▼ 📦 Project Structure

‼️ Follow these instructions to set up your project **before** getting started with your PA04 implementation.

1. Create your PA04 repository using the GitHub Classroom flow [redacted] and open it in IntelliJ
    a. When you click on the link, you'll be presented with a list of existing teams as well as the opportunity to create a new team.
        i. **If your partner has already created a team**, you must join that team. (Teams are limited to two students).

ii. **If your partner has not created a team**, go ahead and create one.

1. You can choose your own team name… but keep them civil and under 15 characters.

iii. **If you plan to work alone**, create a team just for yourself and join that.

iv. You will need to coordinate with your partner to make sure that you don't make two different teams. We advise you to do this step with your partner at the same time.

b. Once you've created your team and accepted the assignment invite, clone the created repository as you did for prior assignments.

For Teams of 2:

1. **First:**
   - Discuss both of your approaches to PA03 and your completion status. Decide whose source code you want to build from for PA04.
   - Use that partner's source code for Steps 2 & 3 below.
2. **Then:**
   - Before following steps 2 & 3, both partners should make pushes (if necessary) and resolve any merge conflicts.
   - Next, **one** partner follows steps 2 & 3 before the other partner makes any other changes.
   - Finally, Once Steps 2 & 3 are complete, make a `Commit & Push`, then both `Fetch & Pull`. Finally, both partners make a branch to work from.

2. Copy the contents of your `PA03 > src > main > java` directory to the `PA04 > src > main > java > cs3500.pa03` package. Move your `Driver` to the `cs3500.pa04` package.
3. Copy the contents of your `PA03 > src > test > java` folder to the `PA04 > src > test > java` folder.

Create your PA04 implementation in the `PA04 > src > main > java > cs3500.pa04 > …` package, and put your tests in the `PA04 > src > test > java > cs3500.pa04 > …`.

> 💡 For tips on working collaboratively on this assignment, check out:
>
> Partner Work Strategies

▼ 🌐 Local Server

When it's time to test your implementation with a real server, you'll be able to test on a server running on your own computer (like in Lab 06). The server is provided as a JAR in the root folder of your PA04 project.

Download the updated Server.jar [redacted]

To run the server and client:

1. Right-click the `Server.jar` in your project root directory
2. Select `Run 'Server.jar'`
3. Wait for the message `"Accepting clients ..."` to appear
4. Navigate to the main method you added to the `cs3500.pa04.Driver` class
5. Run the `main()` method with your specified configuration
6. Both the Server and your Driver will run at the same time in two different tabs.

💡 1. Unless otherwise specified, the `host` and `port` of the server are `0.0.0.0` and `35001`, respectively.

2. Type `quit` in the `Server` console to stop the server.

3. Add `--help` as a `program argument` to the `Server` run configuration for info on optional program arguments you can add to configure the server.

Below is a video walkthrough of how you can run the `Server` and `Driver` at the same time:

🪖 Local Testing

- To test your client against the **TA agent**, set your `game-type` parameter in `join` to `"SINGLE"`.
- If you run into issues with your code where additional information from the server would be helpful, add the program argument `--debug all`. This makes the server print useful information on the state of the game and all sent and received json messages.

🚚 Deliverables

▼ 🐶 Partners (@June 5, 2023 10:00 PM)

Partners for PA04 - BattleSalvo - Part 2

▼ 🎨 Design (@June 6, 2023 10:00 PM)

- [ ] Backlog of Tasks entered as Issues (similar stipulations as with PA01-PA03)
- [ ] UML Class Diagram describing your PA04 design

▼ 🛠️ Implementation (@June 12, 2023 10:00 PM)

- [ ] A `CHANGELOG.md` in your assignment's root directory.
    - Lists the changes to PA03 files made for PA04. Short bullets are more than sufficient.
    - Provides a short (one sentence) justification for each change.

> 💡 The goal is to minimize changes and instead use abstraction/extension/adaptations where possible.

- [ ] A `ProxyController` implementation which supports the **Game Loop Protocol** described in the **Remote Interactions Protocols** section.
    - Use the `Lab 06 Proxy Dealer` as a reference to help you get started.
    - When the game is finished being played (`end-game` message received), the `ProxyController` should close the `Socket` connection and the program should end.
- [ ] An updated `Driver` implementation.
    - Runs your PA03 implementation (human vs. CPU) when no command-line arguments are provided.
    - Runs your PA04 version (local CPU vs. server CPU) when a `host` and `port` are provided.
    - Throws an exception (with a descriptive message) otherwise.
- [ ] An up-to-date UML diagram describing your implementation.
- [ ] You are not necessarily required to update your CPU player, but you will probably want to make a better CPU implementation if you'd like to win the BattleSalvo tournament. 🤓

✨ Hints

▼ 🌎 General

- Before starting to design your solution, remind yourself of the **SOLID principles**. These articles (DigitalOcean and FreeCodeCamp) an overview of them if you need a very brief reminder. **Consider not only how this can help your implementation but also your testing.**
- If you need help creating a **UML diagram**, this is a reference sheet you may find useful. We are not expecting you to label packages, but we are expecting classes, interfaces, fields, and methods to be described, as well as their relationships.
- **Modeling interactions and details on paper before trying to formalize them in code** will help with your understanding of the assignment as well. If you feel overwhelmed, break down the problem into smaller parts and tackle them one at a time.
- If you're feeling stuck, **follow the *"I'm Stuck" Workflow*** as described in lab. 1) Understand your knowledge gap, 2) patch your knowledge gap (through review, additional reading, or office hours), then 3) re-approach the problem and make use of your newly gained understanding.
- Finally, don't forget your essential problem solving concepts. This JavaDoc (this reference page could be helpful) and **following the design recipe to break down problems** will help you greatly.

▼ 👉 PA Specific

- If you're not sure where to start, we suggest you model your data first before coding the rest of your implementation. For this PA that means to **set up your JSON records first,** before building the `ProxyController`.
- This is an example of a design pattern called the **Proxy Pattern**. If you'd like to explore this pattern some more, this is a helpful read with several illustrations. Lab 06 has diagrams of the proxy pattern as well if you find them useful.
- The server will kick your player after more than 2 seconds of no response, so keep this in mind as you design your CPU strategies for the tournament.