# Software Product Lines, Group 5, Project final report

## Context and Background:

“Battleships” is a strategy-type guessing tabletop game for two players. In the standard variant of the game, each player has two 10x10 game grids: the first one is used to place the ships from the fleet of the first player, and the second one is used to mark the hits to the opponent’s fleet. At the beginning of the game, each player locates the ships of their fleet on the first game grid (usually, ten ships are used with different lengths). After that, players start “hitting” the opponent's ships by guessing their positions in the game grid. Depending on the game variant, if the player successfully hits the ship, the player is allowed to guess another position in the game grid. If the player fails to hit the opponent's ship, the turn passes to the other player. The game stops when one of the players manages to sink all the opponent’s ships.

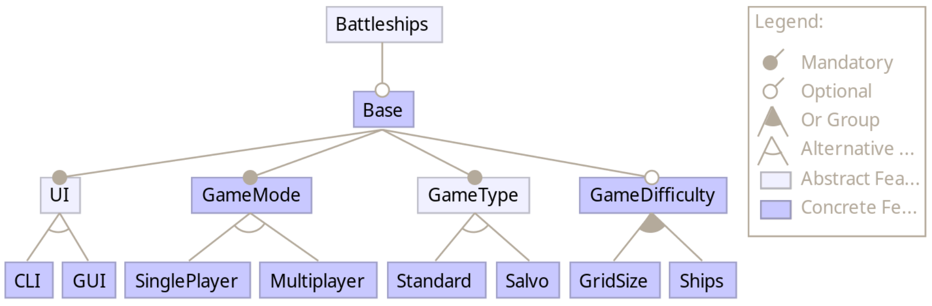
## Motivation:

The ‘Battleships’ project is based on the idea of developing a product line for the popular strategic naval battle game. This includes the creation of variants of the game that exploit different features, such as alternative user interfaces (CLI/GUI), different game modes (singleplayer/multiplayer), and varying difficulty (grid size and ship types). The main motivation lies in the possibility of introducing variability and innovation, such as the ‘1931 Salvo’ variant, which is not implemented in current games.

## Project Idea:

The core objective is to create a product line that allows the generation of various Battleships game versions by combining different features. Key deliverables include:

- UI Variants: CLI with optional color customization, and GUI with images.
- Game Modes: Singleplayer with AI, Multiplayer through socket communication.
- Game Types: Standard gameplay and the "Salvo" variation.
- Difficulty Levels: Configurable grid sizes and variable ship types and quantities.



To achieve this, we will adopt Feature-Oriented Programming (FOP), using tools like FeatureIDE and FeatureHouse to manage feature variability and modularity effectively.

## Methodology:

To develop the project, we will be using the Scrum methodology with 2 sprints:

- In the first sprint, we focused on developing the minimum viable product.
- In the second sprint, we focused on enriching the project with additional functionality.

At the end of each sprint, we evaluated progress and identified any problems. We also organized several face-to-face meetings to discuss the work done and plan the next steps, and for some complex tasks, such as the feature base, we used pair programming.

*Application architecture:*

The application follows a two-threaded architecture to separate game logic from the user interface, ensuring responsiveness and modularity. This design employs the Listener pattern to manage communication between game and game-view threads via a shared event queue, enabling seamless interaction between the game's logic and its visual representation.

The **Game Thread** is responsible for executing game logic, such as handling turns, checking hits or misses, and determining game outcomes. It generates events, represented as objects extending the `GameAction` class, and adds them to a thread-safe ConcurrentLinkedQueue. Examples of such events include `RequestCoordinates`, `Hit`, `Miss`, and `GameWin`. These events encapsulate specific game actions, enabling the logic to remain decoupled from the UI.

The **Game View Thread** monitors the event queue and processes each action. Depending on the type of event, it updates the interface (CLI or GUI) or requests user input. For instance, a `RequestCoordinates` event prompts the player to enter target coordinates, while a `Hit` or `Miss` event displays the result of a turn. The interaction between the game view thread and the players of the game is also implemented using the thread-safe ConcurrentLinkedQueue with messages. For example, this way we can “send” the attack coordinates from the game view thread to the game thread.

This architecture facilitates flexibility and scalability, effectively supporting both local and online players. Local players maintain game data on their client, enabling functionalities such as visualizing the board state directly within their system. This allows for a richer user experience, as local players can interact with the game in a more dynamic and responsive manner.

**Quality assurance**

To ensure the maintainability of the code we decided to perform unit-testing while developing different features of the application. We are using the feature-oriented programming approach to create JUnit tests for the application: each feature also implements related unit tests, and FeatureHouse combines these tests to the complete set suite from the project.

## Results:

The project has been completed successfully, delivering a fully modular implementation where every feature seamlessly integrates with the rest of the codebase.

Because of the difficulty of the Multiplayer feature, we decided to truncate the feature model a bit. The ColoredInterface feature was already implemented in the CLI feature, so we removed this functionality as a separate feature. Talking about the Pictures feature in the GUI, we decided to remove it due to its irrelevance to the product functionality.

We also updated the application launching approach. We pass parameters to the game in the command-line interface.
Here is the list of parameters required by different features:

| **Feature** | **Argument** |
|---|---|
| Base | 1) Nickname<br>2) Player type (“human”/”ai”) |
| Game Mode: Multiplayer | 3) Player port<br>4) Opponent host<br>5) Opponent port |
| Game Difficulty: Grid Size | 3) Game grid width<br>4) Game grid height<br>(or 6,7 if Multiplayer is activated) |
| Game Difficulty: Ships | 3) Game difficulty (1-3)<br>(or 6 if only Multiplayer is activated)<br>(or 5 if only Game Difficulty: Grid Size is activated)<br>(or 8 if Game Difficulty: Grid Size is activated) |

**Bonus feature:** We added the possibility to run the game from the perspective of the game AI, this way for instance we can see how well the AI algorithm is performing.

***GameMode Multiplayer:***
The Multiplayer Mode, initially planned for the first sprint, was completed during the second sprint due to its complexity. This feature allows players to connect and compete via sockets, enabling real-time interaction. The mode leverages a robust protocol system to handle communication, ensuring a smooth and reliable gameplay experience.

The multiplayer is organized the following way: We implement the “distributed” system for the Multiplayer game mode. In this approach, both players are servers and clients simultaneously. Both players run the same game loop in parallel and synchronize and interact within the game setting using the special communication protocol. In the Multiplayer mode, we create a special class OnlinePlayer that implements the Player interface. This is a sort of “virtual” player who neither holds any information about the ships nor implements any specific game logic. It serves as a “connector” with the other player on another host. So, whenever the local game needs anything from the OnlinePlayer, it sends the request to another host and waits until the response.

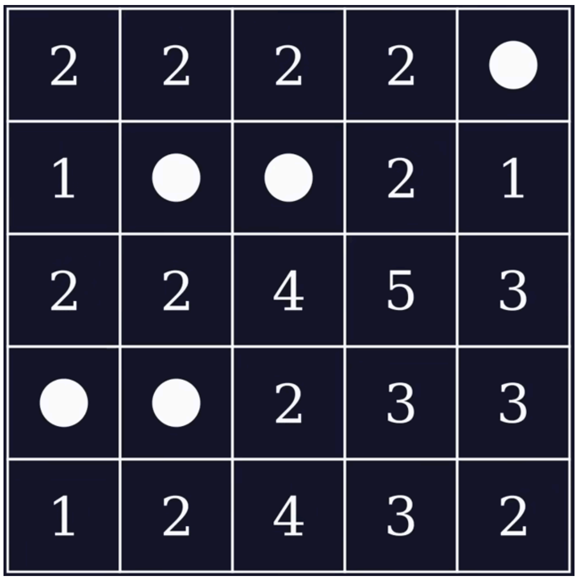
Each local player has queues for the results of different actions, so whenever a player performs a hit or requests the coordinates it saves the result of the operation to this queue.

Finally, every local player contains a special “Online player” handler working in the background. This handler reacts to the requests from the online player and fetches the operation results from the action result queues. To make this communication possible, every request contains a special identifier which depends on the game round. This way we avoid the deadlocks and inconsistencies between online players. Additionally, we implemented the “acknowledgment” feature, so whenever the player receives information from another online player, it sends the special “acknowledgment” message meaning that the player received the necessary information. Using this information players can confirm that the data transfer was successful and delete the result from the appropriate action queue.

***Singleplayer Mode (AI Improvements):***
The AIPlayer now uses a probability-based targeting system, significantly improving its efficiency and accuracy compared to the initial random strategy.

Key Improvements:

- Probability Mapping: The AI dynamically calculates and updates a grid-based probability map to prioritize likely ship locations. It considers all possible placements of remaining ships, aggregating scores to identify the best target.
- Focused Targeting: After a hit, the AI targets adjacent squares to sink the ship. If it encounters a miss, it reverses direction until the ship is sunk.
- Adaptive Behavior: Probability maps are recalculated after each move, incorporating the latest game state.



| | | | | |
|---|---|---|---|---|
| 2 | 2 | 2 | 2 | ● |
| 1 | ● | ● | 2 | 1 |
| 2 | 2 | 4 | 5 | 3 |
| ● | ● | 2 | 3 | 3 |
| 1 | 2 | 4 | 3 | 2 |

*Example of probability map for a Destroyer (3x3)*

Results concern game efficiency, in fact, the AI can sink all opposing ships in about 46 moves on average. Furthermore, the strategy is perfectly adapted to various grid sizes and ship configurations.

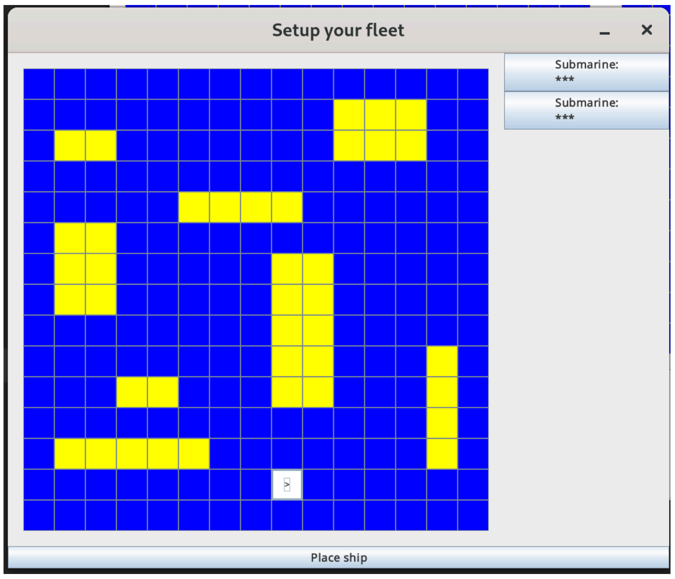
***Salvo Game Type:***
The Salvo Variation of Battleships has been successfully implemented, adding a dynamic and strategic layer to the gameplay. Unlike the standard game, this variation allows players to fire multiple shots per turn, making the experience faster-paced and more engaging for advanced players. The number of shots a player can take corresponds to the number of ships remaining in their fleet, introducing a unique balance between aggression and survival. As ships are sunk, the available shots decrease, forcing players to adapt their strategies as the game progresses.

Players receive feedback after selecting all hits during their turn. The system processes the coordinates, updating the game status to reflect hits, misses, and sunken ships.

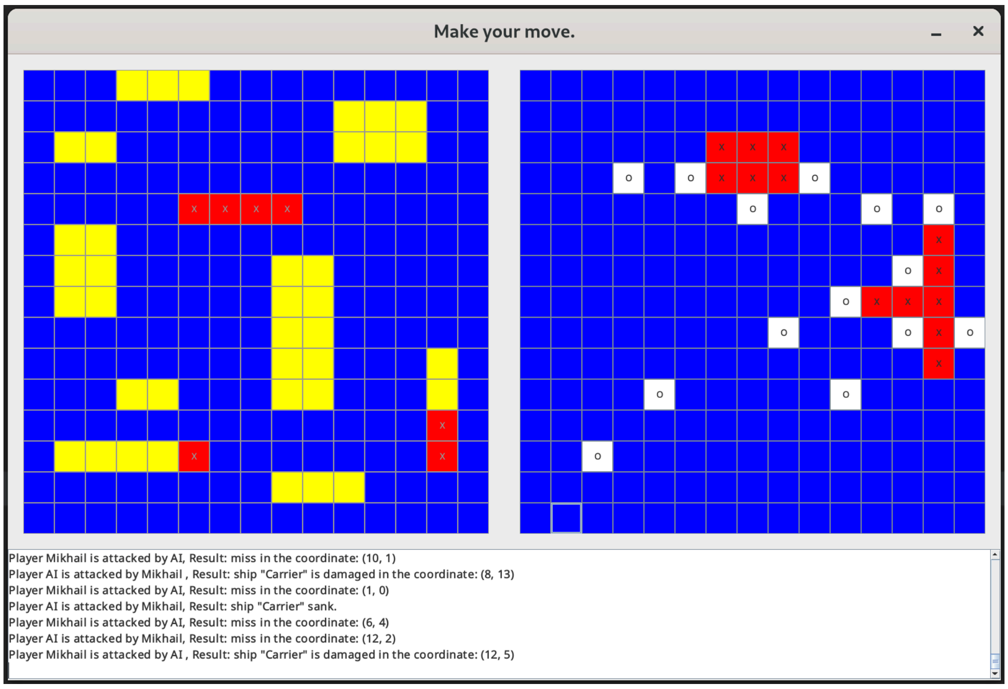
The Salvo Variation encourages players to adopt a more calculated approach, weighing the benefits of spreading shots across the grid to locate ships against focusing fire on already damaged targets to secure a sink. This variation has been fully integrated into the product line and complements the standard game mode, offering an enriched experience tailored to more strategic and experienced players.

***Graphical User Interface (GUI):***
A graphical user interface was fully implemented, offering players an interactive and visually appealing experience. This modern interface corresponds to the previously developed command line interface (CLI) but offers users more options for involvement in the game. The GUI dynamically updates according to the game, ensuring that players receive immediate feedback on successes, errors, and other in-game events.

GUI supports the interactive procedure for setting up the fleet. Players can select the ship from the menu on the right side of the frame, after that select the coordinate on the game field and click on the “Place ship” button. The ships are represented using the yellow color on a game grid.



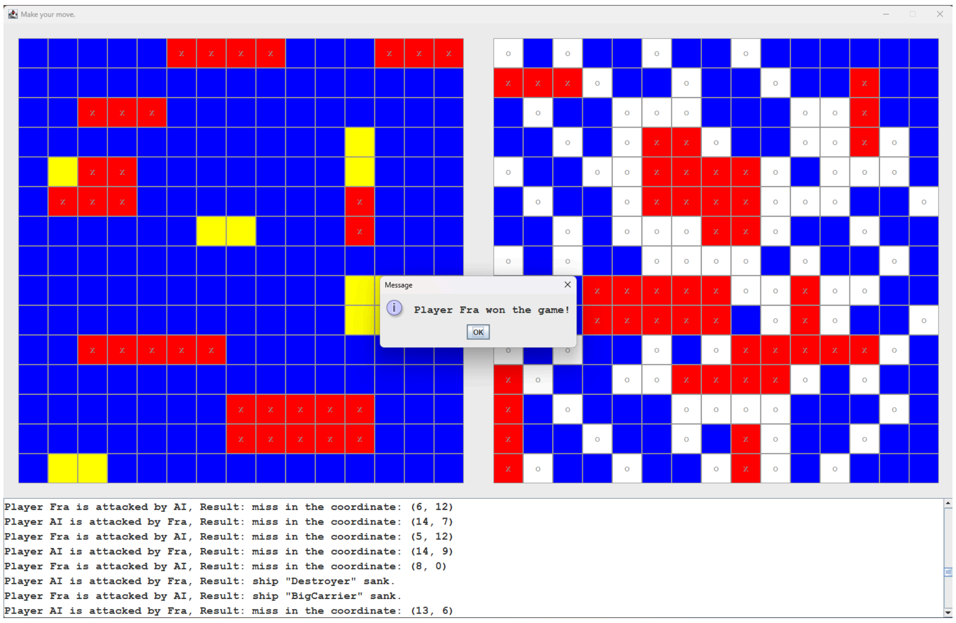
After all ships are placed on the grid the game starts. The left grid in the main game window represents the Player's ships, and the right grid represents the opponent’s field. We denote the ship hits with the red color and misses with the white color. At each turn, the payer can select the coordinate on the opponent's gird performing a “hit” on the enemy fleet.



```
Player Mikhail is attacked by AI, Result: miss in the coordinate: (10, 1)
Player AI is attacked by Mikhail , Result: ship "Carrier" is damaged in the coordinate: (8, 13)
Player Mikhail is attacked by AI, Result: miss in the coordinate: (1, 0)
Player AI is attacked by Mikhail, Result: ship "Carrier" sank.
Player Mikhail is attacked by AI, Result: miss in the coordinate: (6, 4)
Player AI is attacked by Mikhail, Result: miss in the coordinate: (12, 2)
Player Mikhail is attacked by AI , Result: ship "Carrier" is damaged in the coordinate: (12, 5)
```

The panel at the bottom of the game window contains the game event messages.

***GameDifficulty GridSize:***
A game mode has been introduced in which the difficulty is increased by the possibility of customizing the grid size. Unlike the classic 10x10 grid, users can select the desired size for the playing field. This makes it possible to create more complex game scenarios (e.g. with larger grids) or simpler ones (with smaller grids), offering more flexibility and adaptability to the player's preferences.



```
Player Fra is attacked by AI, Result: miss in the coordinate: (6, 12)
Player AI is attacked by Fra, Result: miss in the coordinate: (14, 7)
Player Fra is attacked by AI, Result: miss in the coordinate: (5, 12)
Player AI is attacked by Fra, Result: miss in the coordinate: (14, 9)
Player Fra is attacked by AI, Result: miss in the coordinate: (8, 0)
Player AI is attacked by Fra, Result: ship "Destroyer" sank.
Player Fra is attacked by AI, Result: ship "BigCarrier" sank.
Player AI is attacked by Fra, Result: miss in the coordinate: (13, 6)
```

*Example of a game with increased Game Difficulty: grid size (15x15) and more complex ship templates (such as the 5x2 big carrier)*

|  | **Sprint 1** | **Sprint 2** |
|---|---|---|
| Mikhail: | • GameMode: Multiplayer (in progress)<br>• GameType: Standard<br>• GameDifficulty: Ships | • GameMode: Multiplayer<br>• GameMode: Singleplayer (Improved AI) |
| Francesco: | • GameMode: Singleplayer<br>• UI: CLI<br>• CLI: ColoredInterface | • GameType: Salvo<br>• GameDifficulty: GridSize |
| Together: | • Base (took more time than expected) | • UI: GUI |