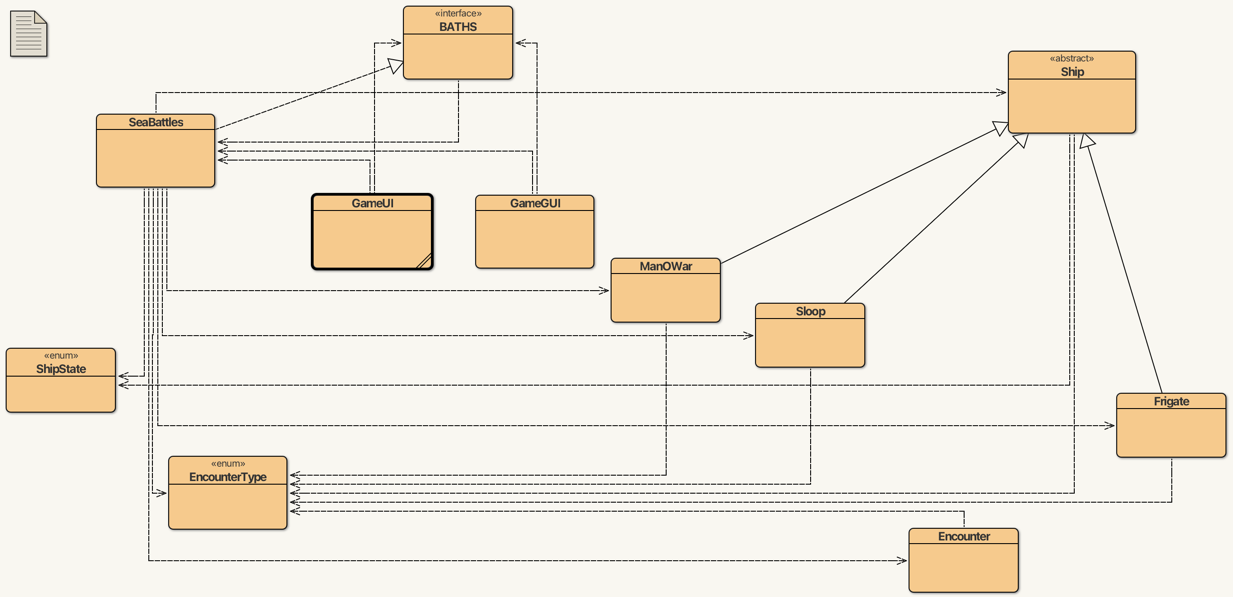
**Technical Report for "Battles and the High Seas (BATHS)" Project**

**Introduction**

This technical report outlines the key design choices made during the development of the *Battles and the High* Seas (BATHS) project. The goal was to create a simple but adaptable Java-based naval strategy game using object-oriented principles, with a focus on maintainability, testability, and BATHS interface conformance. The following sections present the main design decisions, alternatives considered, and justifications for the final implementation.

**UML Class Diagram**



The UML diagram above outlines the structure of our project design showing relationships between classes

Design Decision 1: Use of Supplier Classes for Ships and Encounters

Decision:

We chose to use HashMap collections to store the reserve fleet, squadron, and encounters:

private Map<String, Ship> reserveFleet = new HashMap<>();

private Map<String, Ship> squadron = new HashMap<>();

private Map<Integer, Encounter> encounters = new HashMap<>();

Alternative:  
Using ArrayList<Ship> or ArrayList<Encounter> instead of HashMap.

Benefits of our choice:

* **Quick Lookup:** with a HashMap, we can access a ship or encounter instantly using a key like its name or number. So, we did not have to loop through a list.
* **Easier Uniqueness:** It ensures ships and encounters are uniquely identified by name or ID.
* **Simplifies Game Logic:** Operations like commissioning or checking if a ship is already in the squadron are straightforward using get().

Limitations of alternative:

* Slower: To find a specific ship, we would need a loop. E.g. for (Ship s : reserveFleet), which is slower if we have many ships.
* **More Logic Needed:** We would have to write more code to check if a ship already exists by name.
* **Messy:** Managing state (like removing a ship from one list and adding to another) would be messier compared to moving items in HashMaps by key.

Why did we choose our approach?

We opted for HashMap due to its quicker lookup and uniqueness. Since each ship has a unique name and each encounter has its own ID, it made sense to use a key-value system where we can instantly find what we were looking for without looping through a list every time. It also helped keep the code tidy. For example, moving a ship from the reserve fleet to the squadron was as simple as removing it from one map and adding it to the other using its name. There was no need to search manually or worry about duplicates. Overall, it just made the game logic easier to write, read, and manage, especially when we started adding more features later like saving and restoring the game.

Design Decision 2: WarChest and Game State Tracking in SeaBattles

**Decision:**We chose to keep all key game-related state, like the war chest, squadron, reserve fleet, and encounters, directly inside the SeaBattles class.

**Alternative:**Create separate manager classes like WarChestManager, FleetManager, or EncounterManager, each responsible for a specific part of the game logic.

Pros of our choice:

* It made our code easier to read and manage because everything important was in one place.
* Debugging and following the game’s flow was straightforward since all core functionality lived in SeaBattles.
* It matched nicely with the BATHS interface, making the implementation simpler to follow.

Cons of alternative:

* It would have made our project unnecessarily complex for its size.
* We would have to write extra code to connect the managers together, which could introduce new bugs or delays.

Why did we choose our approach?  
Centralising the state in SeaBattles allowed us to work more quickly and efficiently as this was a project with a tight deadline. We wanted to avoid overthinking the solution. We were able to fulfil deadlines by keeping it brief and straightforward while still thoroughly explaining the game mechanics through comments.

Design Decision 3: Using JUnit for Test-First Development

**Decision:**

We chose to implement an abstract Ship superclass and created three subclasses from it: ManOWar, Frigate, and Sloop. Each of these subclasses has its own properties and behaviour, particularly in how they handle battles and display their details.

**Alternative:**

An option we considered early on was using a single Ship class with a type field (like "Sloop" or "Frigate") and handling the different behaviours using conditional statements (e.g. if type == "Sloop").

Pros of our choice:

* It kept the logic for each ship type neatly separated, so the code was easier to manage and understand.
* We avoided lots of messy if statements by letting each ship subclass define its own behaviour.
* It made it easy to expand the system, if we wanted to add a new type of ship later, we just had to extend Ship.

Disadvantages of the alternative:

* A single class with conditional logic would get very large and harder to maintain.
* Adding or changing ship types would require editing a lot of unrelated code, which could cause bugs.
* It would have been harder to write unit tests for individual ship behaviours if everything was inside one class.

Why do we choose our strategy?

We chose inheritance because it provided a clean and structured way to represent the different ship types. By placing shared attributes and methods in the abstract Ship class and extending it for Sloop, Frigate, and ManOWar, we avoided repetitive code and complex if statements. This also made our code easier to extend in the future if new ship types are introduced, and this enabled us to use polymorphism for functions like canFight().