**Executive Summary**

The assignment involved designing a Java software system to control a drone for a rescue mission. The drone was operated in a challenging marine environment, with the goal of finding survivors on islands and identifying safe places for rescuing them from boats. The system had to manage the drone's limitations, including energy, radio range, and command processing capabilities, to avoid losing it. The control software was required to command the drone efficiently to survey the area, identify points of interest such as creeks and the emergency site, and return safely while using as little energy as possible as that is an important factor too.

Key functions included setting up the drone with its battery level and initial heading, making decisions based on the drone's feedback, and producing a final report. The Java code interacted with JSON commands to control the drone and manage its responses. Commands allowed the drone to fly, turn, scan with radar, and take photographs to recognize terrain features and points of interest.

To implement this system in Java, an iterative and incremental development approach was followed, starting with a basic framework that established communication with the drone. Features were added gradually, such as movement, scanning, and efficient navigation, to achieve the objectives. The "Runner" and "Explorer" classes were used as the foundation, with a focus on creating a modular, maintainable, and testable design. Object-oriented design principles and patterns were considered to structure the code effectively.

Given the comprehensive project requirements and objectives, implementing the system required an in-depth understanding of Java programming, JSON data handling, and software design principles. This was necessary to develop a flexible and efficient control system for the drone. But no system is made perfect, so I have identified some pros and cons below to give an exact idea of the scope and limitations of this system that I have designed.

**Pros**

The following are the pros of the system.

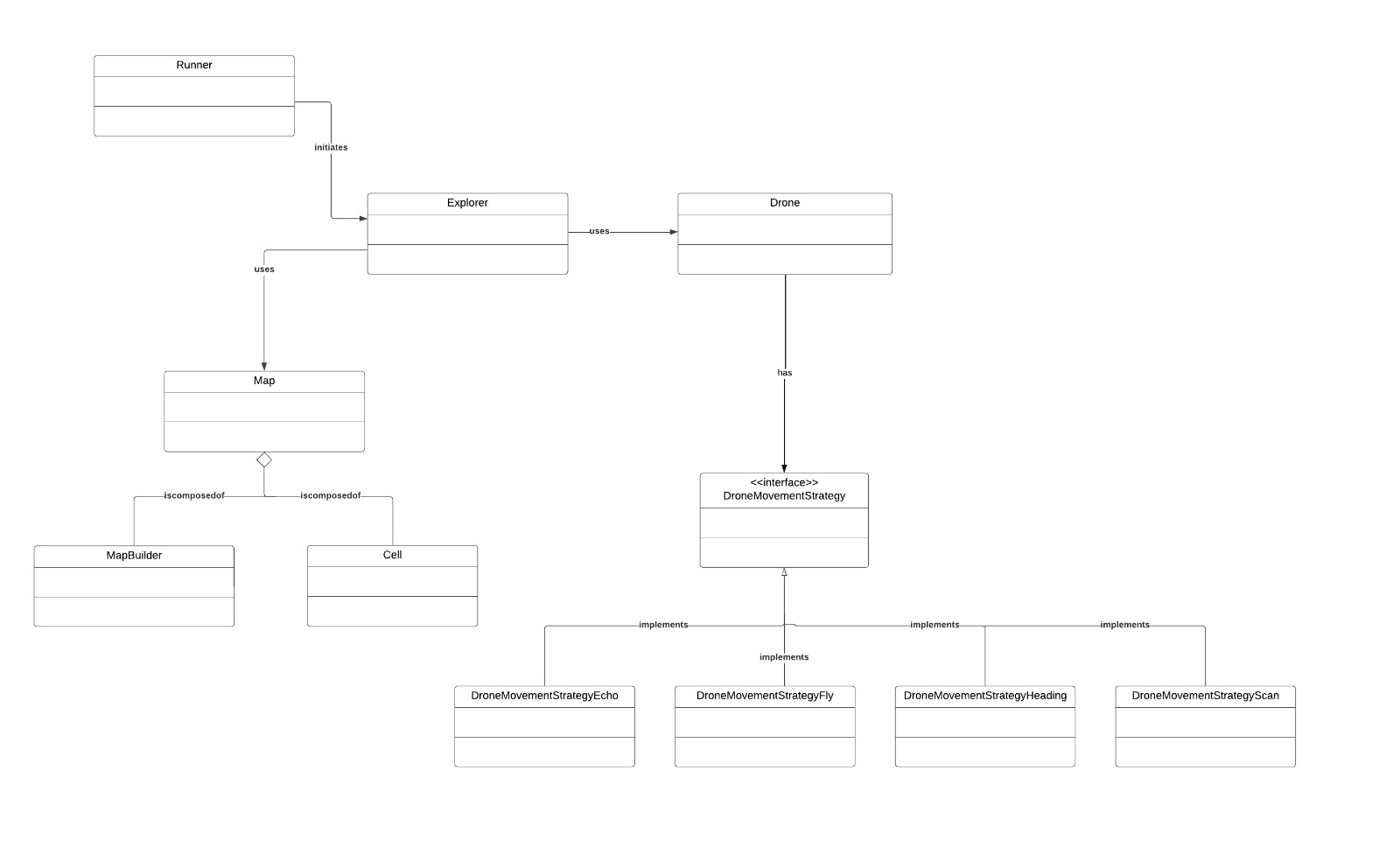
* Modular Design: The code structure suggests a high degree of modularity, with clear separation between different functionalities (e.g., drone movements, map handling). This facilitates easy maintenance and scalability.
* Efficient search: The use of a custom-built algorithm for the search of maps enables the complete as well as efficient search for the point of interest for different maps.
* Automated Testing: The presence of a dedicated test directory indicates a strong emphasis on automated testing, which is crucial for ensuring the reliability and robustness of the software over time.

**Cons**

* Complex Directory Structure: The deep nesting of directories and package names could complicate navigation and understanding of the project structure for new developers.
* Potential Overhead from Multiple Formats: Handling both JSON and SVG for maps might introduce unnecessary complexity and processing overhead, especially if the formats serve similar purposes.
* Less Robustness: Due to lot of variables in the system, there might be a discrepancy while handling different types of terrains, maps, edge cases, etc.

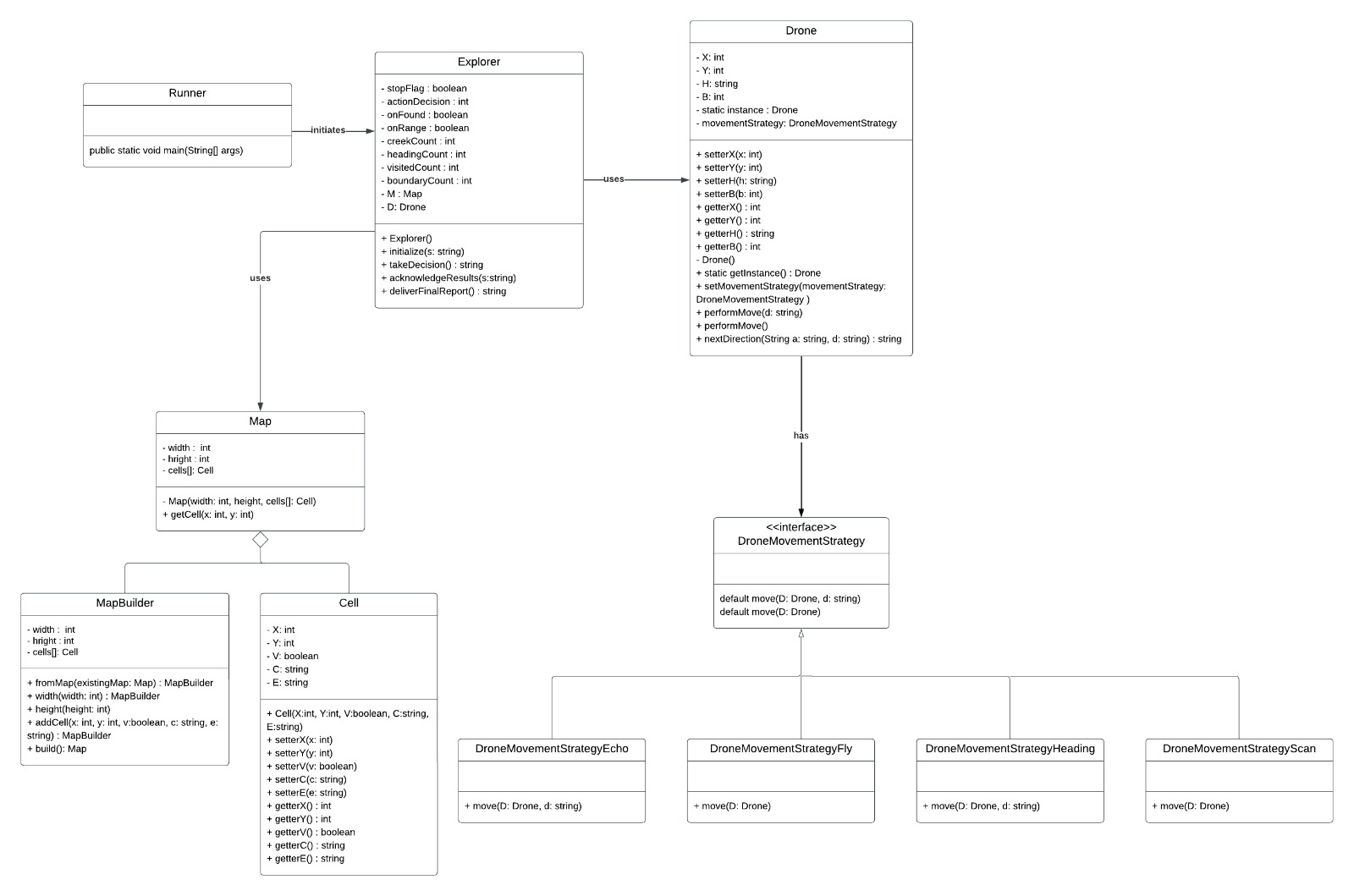
**High-level UML Class Diagram**

A high-level UML class diagram would show the main classes such as Drone, Drone Movement Strategy, and its subclasses Drone Movement Strategy Echo, Drone Movement Strategy Fly, etc, highlighting the pattern implementation. Additionally, relationships with map handling classes and the main Runner class orchestrating the exploration would be depicted. In the same way other classes and their relationships will be shown with the help of UML notation.



**Detailed Class Diagram**

The detailed class diagram includes the classes, their relationships, their attributes, their functionalities and most importantly their division and structure with respect to each other. This has been designed after passing through a very meticulous process.



**SOLID and GRASP Principles Justification**

SOLID and GRASP are two sets of design principles that can help you create software that is easy to understand, maintain, and extend. They have multiple strategies to do this which are defined below in terms of the system I have designed.

* Single Responsibility: Each class focuses on a single responsibility, thus only having encapsulated related behaviors and data.
* Open/Closed Principle: The system's design, particularly the Strategy pattern for drone movements, allows for adding new behaviors without altering existing code. This means that interface classes have been made which just define new functionality and the functionality is added into the system smoothly.
* Controller Class: The explorer class is made the controller of the whole system as majority of the class and function calls are made from here. It also contains the main program logic that decides by analyzing the current state, when to take which action.
* Low Coupling: The classes have been segregated into multiple classes so a high dependency on each other can be removed. Just like the properties of Map and Drone classes could have been made as variables in the Explorer class but then their dependency on each other would have been so high.
* High Cohesion: The classes have been made keeping in mind their clear and focused purpose of use. Just like the Drone Movement Strategy class has been specifically made for having an interface class for the smooth addition of more functionality whenever necessary.

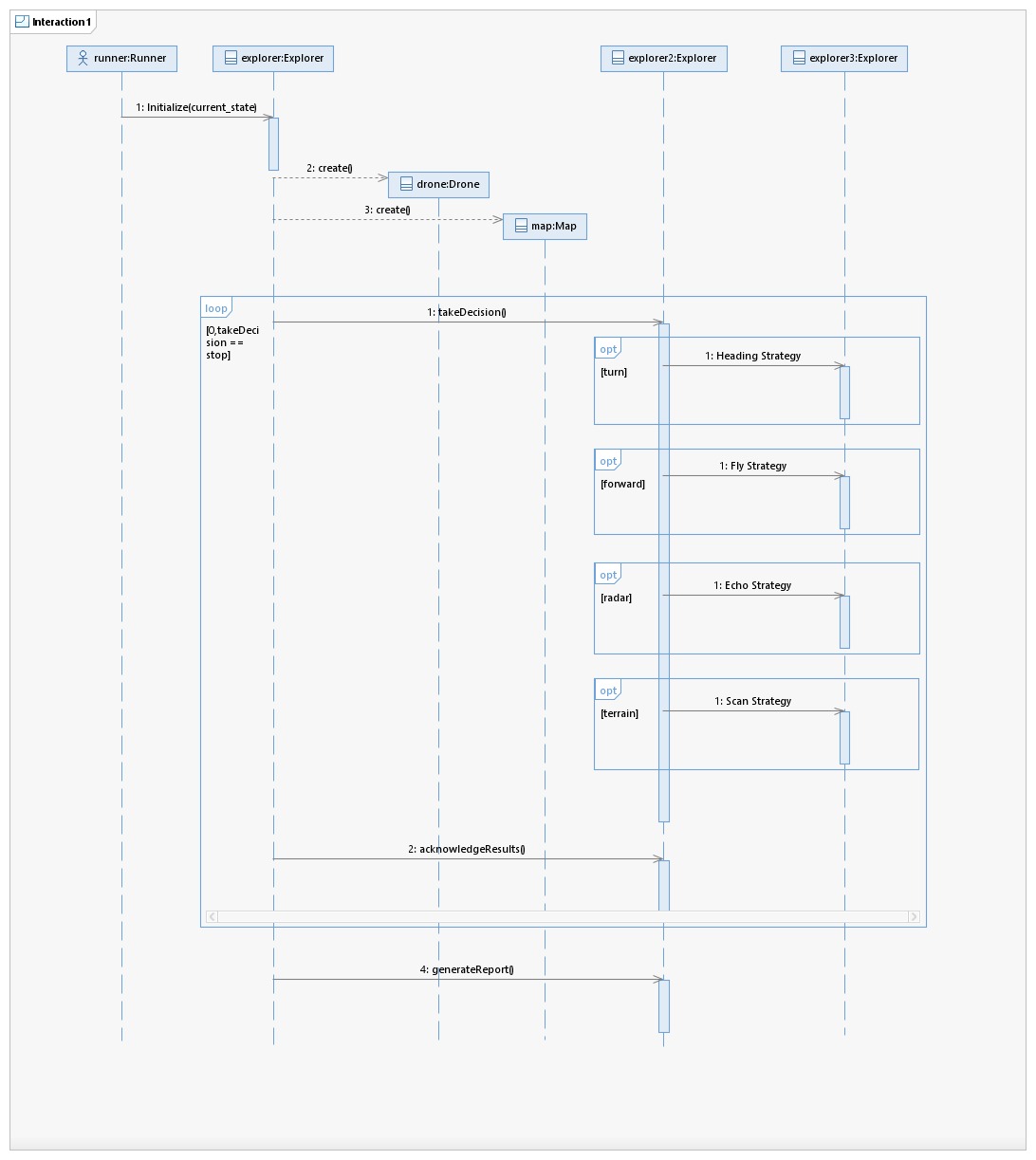
**GoF Patterns Justification**

The "Gang of Four" (GoF) patterns refer to a collection of design patterns that provide solutions to common design problems encountered in software development and are categorized into three main types. The subtypes that I have implemented in my system are given below.

* Strategy Pattern: Clearly used for drone movement strategies, allowing dynamic changes in drone behavior. This had made the use of different strategies and adding more actions of the drone in the system very easy and manageable.
* Builder Pattern: The builder pattern is used for the Map class as it was a complex object to make. So, it was divided into two static subclasses within the Map class. The Map Builder class is responsible for creation of map array and the Cell class is one instance of the Map class which has all its properties.
* Singelton Pattern: This pattern was used for the Drone class to instantiate it only one time throughout the system as there is only one drone to control.
* Factory Pattern: I also tried the factory pattern in the Drone class but then removed it as I was going to apply singleton pattern on it.

**High-level Sequence Diagram for Exploration**

The sequence diagram would illustrate the interaction between the main components during an exploration cycle, starting from the Runner, initiating drone movements through different strategies, and interacting with the map to determine valid movements and discoveries.



**Future Code Modifications for More Rescue Actions**

As the scope of the system is very diverse, there can be a lot of improvement made in this by adding new features, improving the existing features, increasing the efficiency of algorithms used, using more OOP patterns, and simplifying the logic for better run time execution. Some modifications that can be specifically made to cater for more rescue missions are given below.

* Action Interface: the system could introduce an Action interface with different implementations for each rescue type. This would allow drones to execute diverse actions dynamically, based on encountered situations, without requiring significant codebase changes.
* Land Actions: Use the other available actions (land, move, scout and explore) to land a boat and lead a rescue team to the emergency site, consuming as little budget as possible.
* Robust Algorithm: There can be a better algorithm implemented for map search that can efficiently find interest points on the map as well as can be robust for all type of map shapes, sizes and depth.

**Technical Debt Analysis**

I have identified potential areas of technical debt that could be included in this analysis. It refers to the future costs of rework or maintenance that arise from prioritizing speed and short cuts over code quality in software development, with the debt accumulating over time and requiring resources to be paid off, making it crucial to address and minimize from the start of a project.

* Code Duplication: If similar logic is found across multiple strategy implementations, it may need refactoring to reduce redundancy.
* Complex JSON and SVG Handling: The dual format might necessitate consolidation or optimization to simplify map handling and reduce processing time.
* Extensive Directory Structure: Simplification may help improve project navigability and reduce the learning curve for new developers.
* Use of Singletons: The Drone class is implemented as a singleton. While singletons are useful for ensuring only one instance of a class is created, they can also lead to tight coupling and difficulties in testing, especially in complex systems where dependency injection might be a preferable approach.
* Variable Naming: The variable names (X, Y, H, B) are not descriptive. This can lead to confusion about what these variables represent, reducing code readability and maintainability. Using more descriptive names would improve understanding and maintainability.
* Hardcoded Initial Values: The drone's initial position and heading are hardcoded in the constructor. This design choice reduces the flexibility of the Drone class and can lead to issues if the initial state needs to be different or configurable.
* Use of JSONObject: Returning a JSONObject from movement strategies suggests a high degree of coupling between the movement logic and JSON data structures. This might limit the flexibility of the movement strategies and complicate unit testing. Consider using a more domain-specific return type that can be easily converted to JSON if needed for communication or serialization.
* Hardcoded Logic in nextDirection: The method contains hardcoded logic for direction changes based on action strings like "echo", "heading", and "fly". This design is not easily extendable for new actions or directions without modifying the method, leading to potential risks in maintainability.