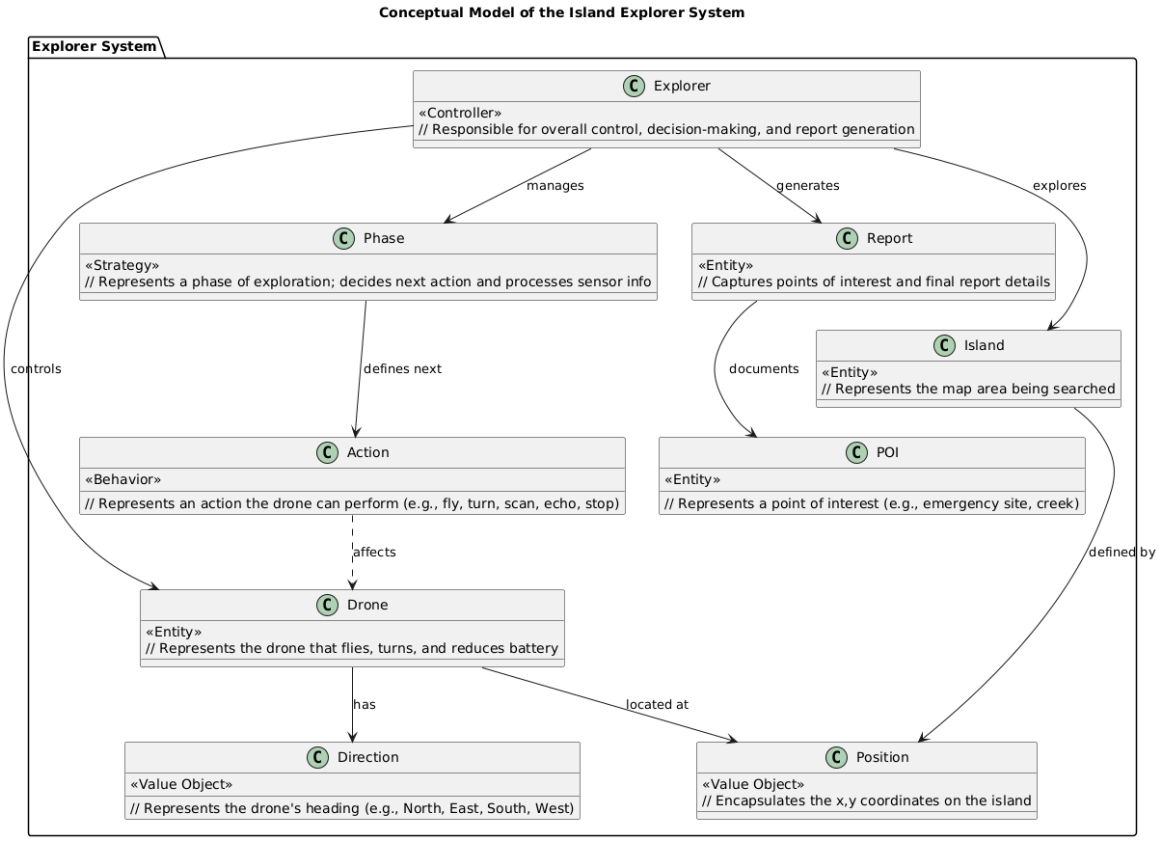
2AA4 A2: Rescue Mission – Design Report

Adrian Winter, Connor McEachern, Angad Chabra

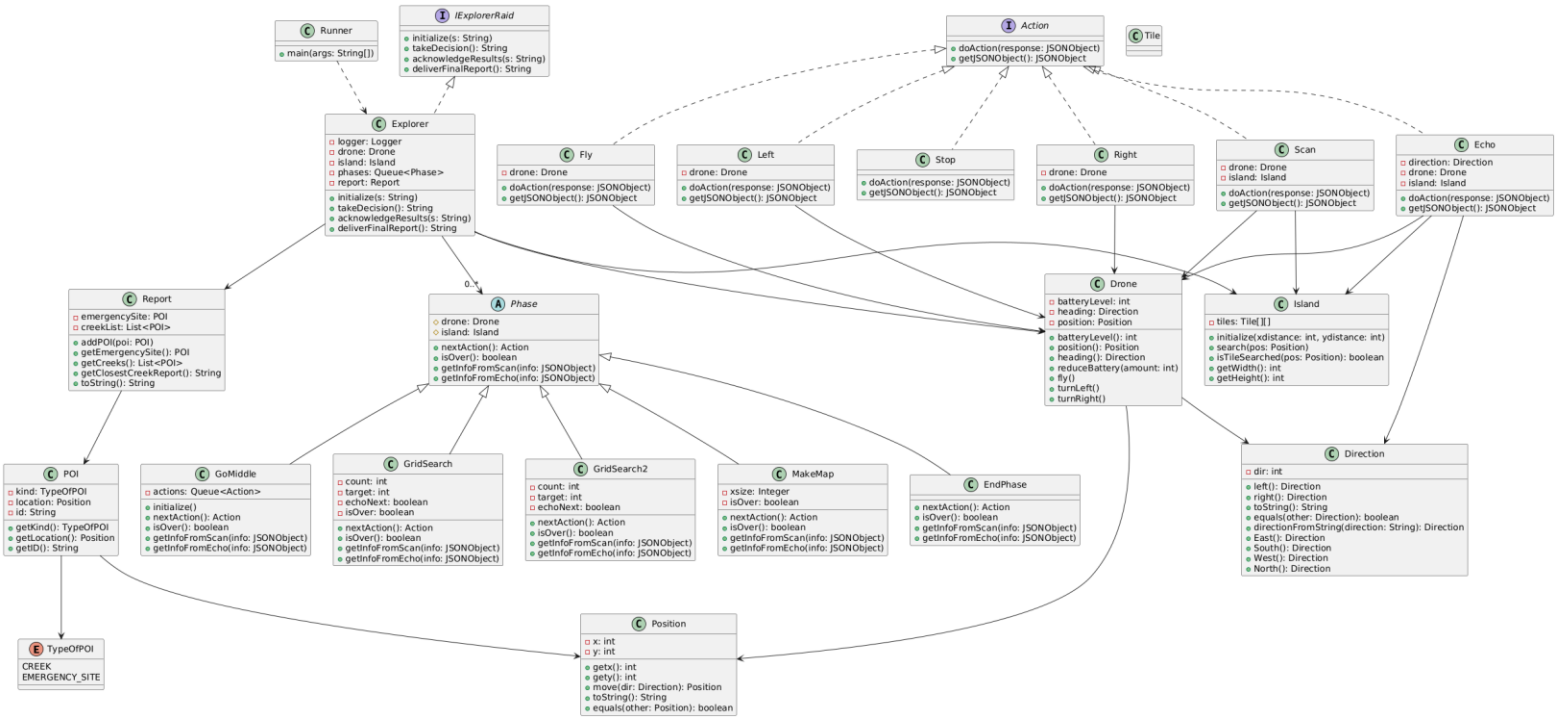
GitHub Repo URL: https://github.com/ConnorMcEachern/A2-template

1. The Rescue Drone System is designed to support outdoor rescue operations in marine environments. In the aftermath of a shipwreck, survivors seek shelter on an island. The system's primary mission consists of locating the people taking shelter at the emergency site and finding a safe creek for a rescue boat to land. The first key operational goal is efficient area coverage. The final design navigates the island using a combination of grid-search and directed exploration phases. The drone begins by mapping the island (MakeMap phase) and then refines the search with directional phases such as GoMiddle, GridSearch, and GridSearch2. The next key goal is to maintain mission safety, done by preventing the drone from going Missing in Action. The final design implements a battery management system to send the done back to base if the battery levels drop below a threshold of 20, ensuring it gets back safely. Additionally, the flight path design commands are designed to ensure that the drone never flies out of the radio range. Finally, turning commands are preceded and proceeded by fly() commands, ensuring that the drone makes a plane like turn, and cannot make an immediate U-turn. The final goal of the operation is intelligent sensing and decision making. Using the drone's sensor actions Scan and Echo, the drone can gather detailed terrain information. In the Report class, the findings of the intelligent decision making are aggregated, including all creeks on the island, and the emergency site. This data allows the system to later calculate the closest creek for rescue operations. The current design meets the core mission requirements of locating an emergency site and the closest creek, however; it has some limitations that suggest areas for future improvements. The limitations include inefficient energy consumption for the grid search, complex command sequencing for flight and sensor operations, and an unoptimized searching algorithm.

2.



To plan and facilitate a practical design, the above conceptual UML diagram was created with an emphasis on functionality and creating the necessary abstractions. This diagram was then refined into a specification style UML diagram with a great emphasis on the use of SOLID and GRASP principles. **Both UML diagrams can be found in the project GitHub repository**



The following design utilizes the SOLID principles in numerous ways. Firstly, the single responsibility principle is used to focus on one aspect of the functionality. For example, the Drone class handles state and movement, the Islan class manages map tiles, and each Action (eg, Fly, Scan, Echo) encapsulates a single behaviour. The use of the open-closed principle is evident throughout the design. For example, by defining the Action interface and the abstract Phase class, new actions or exploration phases can be added without altering the existing code. This allows the system to be more flexible to change and easier to extend, however this may require many additional abstraction layers adding some complexity. The Liskov substitution principle is realized through the GridSearch, MakeMap, and GoMiddle extending Phase classes which can all be used interchangeably without breaking the expected behaviour. This guarantees that you can work with general types like Phase or Action without worrying about the specific subclass, which simplifies the logic in the Explorer class. The interface segregation principle is realized by defining small, focused interfaces such as Action and IExplorerRaid which ensures that clients (like Explorer) only depend on the functionality they actually use. This minimizes the risk of changes in one interface affecting many classes keeping the design more robust. Finally, the dependency inversion principle is realized through the use of high-level modules like explorer which depend on abstractions such as Action and Phase making it easier to substitute new behaviors or implementations.

The design also utilizes GRASP principles in numerous ways. The information expert principle is used as each class is assigned responsibilities based on the information it holds. For example, the Drone knows about its battery level and position while the Island knows the state of each tile. This leads to high cohesion (each class has all the information it needs to fulfill its responsibilities). The creator principle is implemented through the creation of the Phase object in the Explorer class based on the context (i.e MakeMap, GoMiddle). The controller principle is used bu the Explorer class which acts as the central controller that directs the flow of actions via Phase and Action. It receives inputs via JSON responses and delegates the responsibilities to the appropriate objects. This improves clarity and manageability of the overall workflow. Low coupling and high cohesion is used in the design by minimizing dependencies between classes by using interfaces and abstract classes. Finally, Polymorphism is realized through the use of abstract classes and interfaces which simplifies the overall design as components can work with general types without needing to know the specifics of each implementation.

3. The current design supports the primary mission of identifying an emergency site and a viable creek for rescue. Additionally, the system finds the closest creek to the emergency site, ensuring the fastest rescue possible. To further enhance operational capabilities, three adjustments to the system are recommended. Firstly, the grid search classes can be refined to first search the coastline and then patrol the island, further optimized the time and battery consumption of the current search. Next, new functionality can be integrated into the code to allow for additional rescue actions, such as supply drops containing food, water, tools, and medical supplies, communications with the stranded people, or new flight functionality, such as U-turns. This would likely be carried out by extending the action framework with new command types. Since Action is an abstract class, new rescue mission capabilities can be created through an extension without the need to modify old code. The next improvement that can be made is enhanced data integration. The current system only makes use of echo and scan, but additional data sources or sensors could be integrated to provide weather data, tide data, or satellite imagery, improving the decision-making. This would result in an optimization of the flight path and rescue creek selection. This would be seen in the code through the extension of Action and the creation of new interfaces. Classes like Echo and Scan currently extend the abstract Action class. Any new sensor would need its functionality to be declared in a concrete subclass of Action. The current system's Phase subclasses implement the EchoInfoReceiver, ScanInfoReceiver, or both depending on which sensor(s) are used by the class. Each new sensor would also need a new InfoReceiver interface to be implemented and used by any phase class. Finally, the battery management system can be refactored to a dedicated module, making real-time decisions and adapting flight paths or actions based on current battery status.

4. The final design has accumulated a considerable amount of technical debt, which will require future refactoring by architects and developers. The first source of technical debt comes from action command execution complexity. The current design uses multiple action classes such as Fly, Left, Right, Echo, and Scan. As new commands are added to extend rescue operations, managing command sequences and preventing the drone from going Missing in Action will become much more challenging. The action subclasses will also become much more complex as they each need to separately implement decision-making. Implementing a centralized command handler could simply control flow, reduce the risk of errors or bugs, and make the code easier to maintain and modify. This should be an immediate next step, as it will be very beneficial in the long term when the system has a large number of actions.

   Another source of technical debt is the energy management system. Battery consumption is currently handled by individual actions and tracked in the Drone class. The addition of new actions could cause potential risks of battery depletion causing the drone to go Missing in Action. An abstract module to handle energy consumption could centralize energy usage, allowing for better predictions and management of battery levels throughout the mission. This modification is not very urgent to complete as the technical debt accumulated by managing the battery in individual actions exists but is not that significant.

   The next source of technical debt in the current design is the InfoReceiver interfaces used to seperate echo and scan sensor feedback with specific interfaces. In the future, another layer of abstraction should be implemented so a getInfo method can be called and the sensor to use can be decided at runtime. A unified sensor framework would simplify decision making and minimize duplicated logic across phases, minimizing technical debt. This change should take place immediately because it would enhance the precision of POI detection, and it would make it much easier to implement new sensors and external data sources.

   One more source of technical debt arises from code duplication. Multiple classes in the program, such as GridSearch and GridSearch2, share similar logic with only minor differences, for example, turning left versus right. This duplication could cause increasing maintenance overhead costs and inconsistencies over time. The immediate steps taken should be to refactor common logic into a base class to handle grid search pattern, using overloaded methods where parameters determine specific turning behaviour at runtime. Some not so urgent steps that could be taken are to implement design patterns like Template method to abstract searching strategies, which would make the system more modular, easier to maintain, and easier to extend in the long run.

   There is also debt in the testing of the current design. There is a lack of unit tests to cover all critical components, which could make future testing more difficult as the system evolves and there are more features to test. The immediate steps that should be taken are writing unit tests for all core action and phases to ensure expected normal and edge-case behaviour. A less urgent step to resolution in the future is developing an integration test that simulates entire rescue missions, allowing for the validation of interactions between components such as Explorer, Drone, and sensor interfaces.

Finally, some technical debt has accumulated as a result of a lack of detailed comments in some parts of the code. This makes it harder for new team members to understand the design and flow of control between phases and actions. The immediate solution that should be taken is to document key classes and methods, which are very important to improve maintainability and modifiability in the long run. Future steps that are less urgent would be to establish documentation standards to ensure that code remains well-documented as new features are added. This includes UML diagrams as well as inline documentation. Overall, this summary encompasses all the technical debt that the technical experts should be focused on resolving.