

ACME CORP: Outer Wilds Department Report: Rescue Mission

Mohaimen Ibrahim

Rayan Nasrallah

Moustafa Moustafa

Course: 2AA4

Professor: Dr. Sébastien Mosser

1. An executive summary describing the pros and cons of your software design

Pros:

The software design for this drone exploration project is adaptable and scalable because it is a thoughtfully planned implementation of a variety of fundamental software engineering concepts. The modular structure of this design, which divides the software into various classes each specific to different functions including navigation of the drone, decision-making, calculating which creek is closest to the emergency site using Euclidian distance formula, and battery management, is one of its most notable aspects. This flexibility and division of responsibility makes the code easier to read, understand, maintain, and test. Also, it promotes an extensible and expandable codebase by making it possible to incorporate new patterns or techniques with ease.

Furthermore, the independent separation of tasks (single responsibility) between the drone's decision-making logic and its actions (turning, moving, scanning, etc.) simplifies the system's complexity and makes debugging and extension a lot simpler. With the help of classes like *AcknowledgeResults* and *GridSearch* to interpret and react to the environment, as well as keeping track of its position and its surroundings in the *VirtualCoordinateMap*, the design successfully imitates environmental interaction. This structure offers an accurate simulation framework for the drone exploration. Furthermore, the program fully addresses every feature required to operate a drone like flying and scanning, and is also open for more actions and scalability, guaranteeing a thorough exploration capacity.

The advantages of our software design are primarily due to the design patterns it uses, like the Strategy pattern which is used for managing various decision-making processes via the *DecisionMaker* interface. Our software design also implements the State pattern for managing distinct states of exploration and decision-making. These patterns support loose coupling and high cohesion, which are in line with the concepts of object-oriented design. They also encapsulate certain behaviors and states, which makes the system flexible and manageable. By anticipating the possibility of future expansion and modifications, the implementation of these patterns illustrates effective and adaptable software architecture.

Cons:

While the software design for the drone exploration project has many positive features, it also has a few disadvantages that point out areas in need of development. Having hardcoded values is a notable issue, especially when it comes to features like checking if the battery is low and stopping the drone. Currently, we have it so that if the battery level reaches $(0.0022 * \text{initial battery the rescue mission started with})$, the drone stops. We agreed on 0.002 after running multiple tests with different sized maps and different initial battery levels. However, with this hardcoded value, the system is less flexible and adaptable to different or more complex scenarios, which might require altering the code to make it more in line with requirements that are accurate and closer to in real life examples. This feature highlights a critical area that must be improved to increase the reliability and efficiency of the system.

Furthermore, even though the state management used in decision-making classes such as *FindLand* and *GridSearch* is extremely effective, it might be improved by a more formalized application of the state pattern. This

would improve the overall design of the system by making state transitions easier to understand and less difficult to handle. Performance issues also come up as a possible worry, particularly in the simulation's interactions with the environment and the processes for making decisions. As the complexity of the environment or the number of drones increases (more mission at the same time, more drones, more emergency sites etc.), the efficiency of these operations may impact the system's scalability and overall maintenance.

Thinking back on the design patterns that were not utilized but might have been useful, some of the issues mentioned above might have been avoided or reduced by implementing patterns like the Observer pattern, which would keep track of changes in the drone's environment, or the Factory pattern, which generates different kinds of decision-making strategies dynamically. These patterns could provide more flexibility, less coupling, and better system scalability to new scenarios or requirements, increasing the longevity and stability of the current architecture.

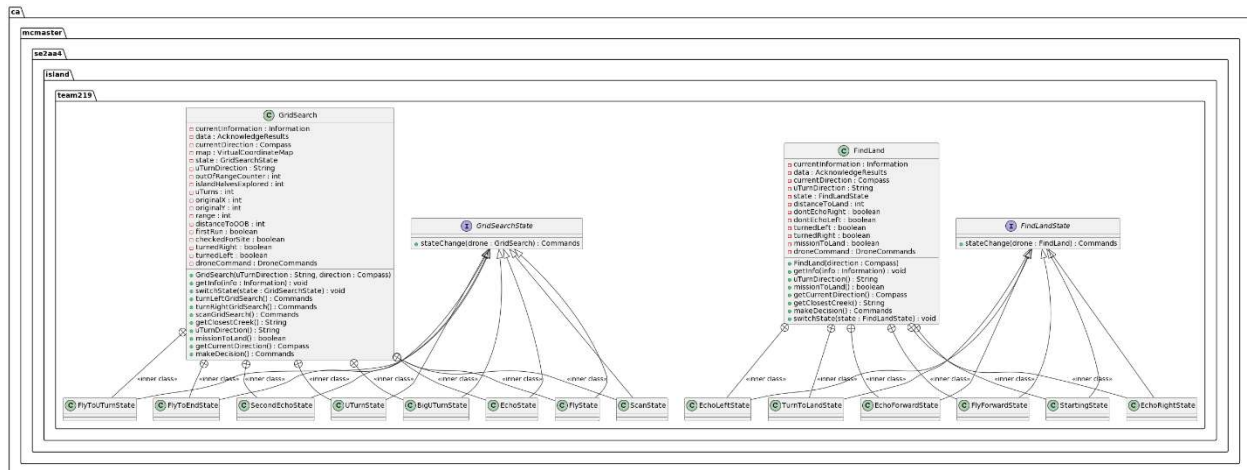
Additionally, an issue we faced when testing the *GridSearch* class for example was keeping track of all the different states we tested. This caused extensive testing of methods and states that may have already been tested but not kept track of. For future projects, running SonarQube on our code throughout the whole project would help in checking the code coverage by the tests. Also, our code contains a few protected access modifiers instead of private due to the need of accessing these variables in tests. Testing may be better facilitated by using methods like using fake objects and test fixtures. Also, package-private visibility can assist in preserving encapsulation and minimizing coupling. Not only would such improvements guarantee the dependability of the system, but they would also make it simpler for new developers or contributors.

```

classDiagram
    class Information {
        +cost : int
        +extras : JSONObject
        +getExtrasJson() : JSONObject
        +getCost() : int
    }
    class Explorer {
        +drone : Drone
        +takeDecision() : String
        +deliverFinalReport() : String
    }
    class Drone {
        +batteryLevel : Battery
        +currentDirection : Compass
        +findLandDecisionMaker : DecisionMaker
        +gridSearchDecisionMaker : DecisionMaker
        +data : AcknowledgeResults
        +makeDecision() : Commands
        +getInfo(info : Information) : void
    }
    class Battery {
        +getBatteryLevel() : int
    }
    class DecisionMaker {
        +makeDecision() : Commands
        +getInfo(info : Information) : void
    }
    class CalculateClosestCreek {
        +creekIds : List<String>
        +calculateClosestCreek() : String
    }
    class GridSearch {
        +map : VirtualCoordinateMap
        +missionToLand() : boolean
    }
    class FindLand {
        +missionToLand() : boolean
    }
    class AcknowledgeResults {
        +emergencySiteFound : boolean
        +creekIds : List<String>
        +initializeExtras(info : Information) : void
        +isFound() : boolean
        +groundsFound() : boolean
        +creekIsFound() : boolean
        +getClosetCreek() : String
    }
    class VirtualCoordinateMap {
        +currentDirection : Compass
        +moveForward() : void
        +turnLeft() : void
        +turnRight() : void
    }
    class Compass {
        +N, S, E, W, NONE
    }
    class Commands {
        +currentDirection : Compass
        +commandTranslator() : JSONObject
    }

    Information --> Drone : uses
    Explorer o-- Drone
    Drone ..> Information : getInfo
    Drone ..> Battery : batteryLevel
    Drone ..> DecisionMaker : findLandDecisionMaker
    Drone ..> DecisionMaker : gridSearchDecisionMaker
    Drone ..> AcknowledgeResults : data
    Drone ..> Commands : makeDecision
    Battery ..> Drone : getBatteryLevel
    DecisionMaker <|-- CalculateClosestCreek
    DecisionMaker <|-- GridSearch
    DecisionMaker <|-- FindLand
    CalculateClosestCreek ..> AcknowledgeResults : calculateClosestCreek
    GridSearch ..> VirtualCoordinateMap : map
    FindLand ..> VirtualCoordinateMap : map
    VirtualCoordinateMap ..> Drone : currentDirection
    VirtualCoordinateMap ..> Commands : moveForward, turnLeft, turnRight
    Compass ..> Drone : currentDirection
    Compass ..> Commands : commandTranslator
    Commands ..> Drone : makeDecision
    
```

3 of 10



Note: Class Diagram for states added separately as there was not enough space to fit it both in one class diagram

4. A justification of how the SOLID and GRASP principles were used in your design

SOLID:

Single Responsibility:

The clear separation of responsibilities among classes is reflective of the Single Responsibility Principle. For example, the *DroneCommands* class focuses purely on generating commands for drone actions, while the *Drone* class manages and controls the drone's states and the decision-making process. We tried to make sure to assign single responsibilities for as many of the classes as we could. Because changes to one part of the system are unlikely to affect other parts of the code, this adherence to the Single Responsibility Principle makes modifications and debugging a lot easier and more efficient.

Open/Closed Principle:

FindLand and *GridSearch* are two examples of how the *DecisionMaker* interface's design reflects the Open/Closed Principle. This framework facilitates extensibility while maintaining the stability of the core system by enabling the incorporation of new decision-making strategies without changing the current code. For example, if you wanted to add a different method of exploring the map other than grid searching, you would not have to adjust any of the current logic in our classes you would just have to add a new class that implements the interface.

Liskov Substitution Principle:

The Liskov Substitution Principle ensures that the validity of the system remains unaffected when instances of base types or interfaces are swapped out for their derived types. By enabling the interchangeability of various *DecisionMaker* implementations and upholding system integrity, our design adheres to this principle.

Interface Segregation Principle:

The specific interfaces and their implementations follow the Interface Segregation Principle, which makes sure that classes don't implement unnecessary methods that would make the code messy and harder to work with. Our classes only implement the methods they need because the interfaces are designed to maintain this principle. This design maintains the system's modularity and encourages more precise functionality reuse.

Dependency Injection Principle:

To improve flexibility and testability, our system decouples class dependencies by utilizing the Dependency Injection Principle. Dependencies, like the *DecisionMaker* implementations, are injected into the *Drone* class to enable dynamic behaviour changes at runtime and increase the system's flexibility to meet changing needs. This idea encourages the development of loosely linked components, which make program maintenance and extension simpler.

GRASP (General Responsibility Assignment Software Patterns)

The principles of GRASP (General Responsibility Assignment Software Patterns) are essential to the structure and design of any software project because they guide how responsibilities are assigned in object-oriented architecture. The Information Specialist, which recommends giving responsibility to the class with the information needed to complete it is clearly incorporated in our design. For example, the *AcknowledgeResults* class acts as an information specialist due to the way it processes and interprets JSON data from exploration findings, and encapsulating the logic for determining whether land or creeks are found. This encapsulation of responsibility enhances modularity and reduces dependencies between components.

Classes in charge of making instances of other classes must contain the knowledge required to generate those instances, according to another principle known as the Creator. This is demonstrated by the way in which our design enables the development of different objects (such as *DroneCommands* and *Commands*) in a way that organises creation logic, minimizes coupling and enhances system integrity.

Low coupling and High Cohesion are achieved by designs that reduce interclass relationships, making software evolution and maintenance simpler. Low coupling is demonstrated by the usage of interfaces like *DecisionMaker* and the assignment of duties to specialised classes like *Drone* and *DroneCommands*. Also, Classes in our design are highly cohesive, with each class focusing on a specific aspect of the system's functionality. This is evident in the separation of concerns between classes like *FindLand*, *GridSearch*, and *DroneCommands*, each handling a distinct part of the system's behavior. Changes to one component of the system can be made due to this design without having any major effects on the others.

Controller patterns can be deduced from the way classes handle and coordinate actions (e.g. *Drone's* coordination between making judgements and carrying out orders based on those decisions). This makes it possible for concerns and control points to be clearly divided within the software.

In conclusion, the design of our project shows a careful application of GRASP principles, which helps to create a codebase that is easier to read, maintain, and adjust to changes in the future. These characteristics could be further improved by further investigation and use of all GRASP patterns, guaranteeing that our program will continue to be resilient in the face of changing needs or modifications in the real world.

5. If any, a justification of the GoF patterns you have used in your design

Facade Pattern:

In our approach, the *Drone* class serves as a facade, making the intricate mechanism that underlies the drone's decision-making process easier to understand for the client. This pattern provides a simple interface to a complex subsystem that includes classes such as *FindLand*, *GridSearch*, and *AcknowledgeResults*. The *Drone* class enables the Explorer to carry out commands without getting entangled down in the details of each decision-making approach by hiding the complexities of these subsystems. This application of the facade design is beneficial because it improves modularity and ease of maintenance by separating the client from the complex processes and improving code usability. It serves as an example of how simplifying a subsystem's complexity through the provision of a single interface to a collection of interfaces can improve usability and maintainability.

Adapter Pattern:

The Adapter pattern is usually used to convert an existing class's interface into one that clients expect. When employed within the Commands class, it has the potential to adjust the many drone action interfaces (such as turn, fly, and scan) into a consistent interface that will be used by other components of the system. With this solution, the drone's actions and decision-making logic would interact more efficiently, guaranteeing compatibility and increasing the ability to add new kinds of orders without changing the decision-making logic or the client code.

Strategy Pattern:

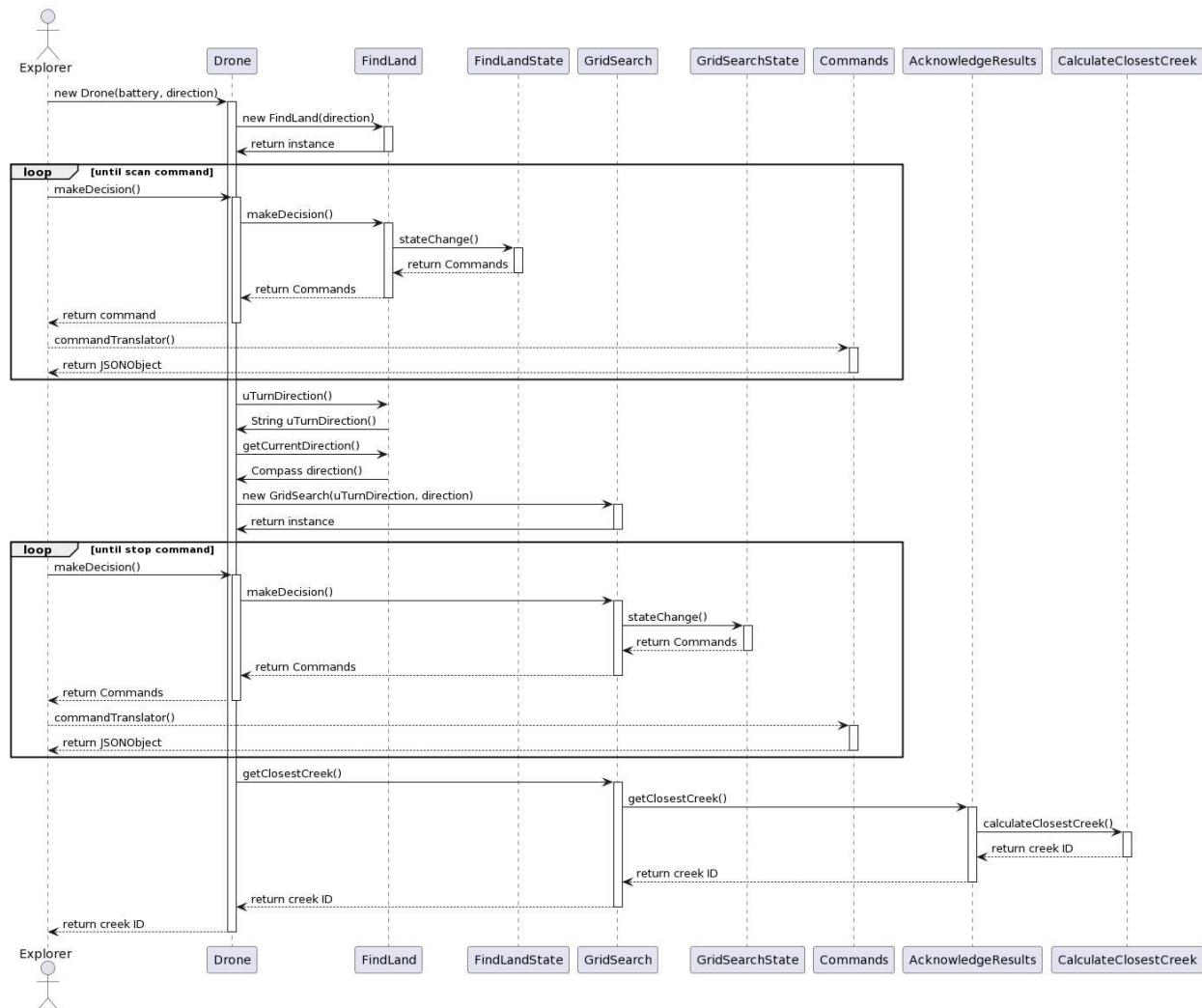
Our approach utilises the Strategy pattern through its practical implementations, *FindLand* and *GridSearch*, and the *DecisionMaker* interface. With this method, the drone can dynamically switch between several decision-making processes, similar to how it can change its behaviour depending on the exploration stage. We adhere to the Open/Closed Principle by encapsulating the algorithms within interchangeable classes, making extension and modification simple. Furthermore, this pattern highlights the flexibility and scalability of our architecture by enabling the possible inclusion of new strategies (for extra rescue missions, for example) without interfering with the functioning of the current code. Our system is compatible with upgrades and modifications because of its modular design, which is largely enhanced by the usage of the Strategy pattern.

State Pattern:

The *FindLand* and *GridSearch* classes demonstrate the State pattern's implementation, with distinct states signifying distinct stages of the drone's journey. This approach eliminates complicated conditionals and numerous boolean variables throughout the code and promotes a dynamic decision shift based on the drone's status and environmental feedback by adapting state-specific actions and enabling smooth transitions between states. It improves the extensibility of the system by making it simple to add new states for additional mission phases, while simultaneously simplifying the context class by offloading state actions to different classes. This pattern serves as an example of how encapsulating different behaviors related to system state changes can result in code that is clearer and easier to maintain. It also offers an organized method for managing extensive state logic and transitions.

Each of these patterns contributes to the overall robustness, extensibility, and maintainability of our design, showcasing the thoughtful application of GoF design principles in addressing the complexities of an automated drone exploration and rescue system.

6. A high-level sequence diagram describing how your exploration works



7. An executive description of how your code will have to be modified to support more rescue actions in the future

In our code, we used the strategy pattern to define a family of algorithms that are used for the rescue mission (i.e *FindLand* and *GridSearch*), and to make these algorithms interchangeable. This allowed our code to have flexibility; as it is easy to switch between different rescue algorithms if needed, and scalability; as adding additional rescue actions/algorithms is simple, as it only requires adding a new class of type *DecisionMaker* to exact a rescue action, making an instance of it in *Drone*, and adding to the already present logic, or modifying the logic in *Drone* to also preform the rescue action without having to modify the structure of the existing code. Furthermore, our code has maintainability, as each rescue action is encapsulated in its own class, making it easier to understand, maintain, extend, and test if necessary. Overall, the use of the strategy pattern allows for seamless extension and modification of the program, without needing to modify the overall structure of the code.

The *DecisionMaker* interface acts as the strategy interface, and its implementations (i.e *FindLand* and *GridSearch*), represent different rescue actions required for finding the creek closest to the emergency site. Adding an additional rescue action would be as easy as:

1. Create new rescue action class (eg *RescueAction*) that implements the *DecisionMaker* interface. This new class would encapsulate the logic for this specific rescue action

2. Add local variables needed in the class *RescueAction* of type *DroneCommands*, *Information*, *Compass*, and *AcknowledgeResults*
3. Override and implement all methods in the *DecisionMaker* interface
4. Initialize and instance of *RescueAction* in the *Drone* class, and add to the already existing logic in the *Drone* class so that the *Drone* also executes this action, for example:

```
private DecisionMaker rescueAction = new RescueAction( , )
```

The 'Drone' class, which acts as the context in the Strategy pattern, can then use this new class to execute the rescue action in question.

Due to the structure of our code, and the use of the strategy pattern, a new rescue action can be added with minimal changes to the existing code. Overall, the Strategy pattern in our code provides a flexible and scalable way to manage different rescue actions, thus making it straight forward and simple to adapt and extend the program to accommodate new requirements or scenarios if necessary.

8. An analysis of the technical debt present in your product

Technical debt in software engineering refers to the cost of additional rework incurred when choosing a short-term, temporary fix today over a more comprehensive, long-term one. The technical debt in a project is the result of several related problems that impact its maintainability and scalability.

The dependence on protected modifiers rather than private ones for testing reasons is a major cause of technical debt in our design. Declaring class members as protected not private can facilitate testing, but it also raises the possibility of inappropriate intimacy—that is, when classes become too familiar with one another's internal operations. Inappropriate intimacy occurs when classes become too closely intertwined with each other, leading to a design where classes are overly dependent on the internal workings of other classes. This situation is a form of technical debt because it makes the system harder to understand, maintain, and extend. Inappropriate intimacy can arise from non-private modifiers being used excessively for testing, as it exposes the internal state and behavior that should ideally be encapsulated.

Because of the system's tight coupling, modifications to one component may have unexpected effects on other components, increasing the system's entropy and instability. Future iterations of the project should aim to reduce the use of protected modifiers and favor private ones to eliminate this technical debt. Testing may be better facilitated by using methods like using fake objects and test fixtures. Also, package-private visibility can assist in preserving encapsulation and minimizing coupling.

Our software design's application of the Interface Segregation Principle (ISP) is noteworthy. However, the *DecisionMaker* interface seems to be resulting in additional technical debt. The interface may have been overused or coupled too closely with specific methods. The goal of the Interface Segregation Principle is to prevent classes from implementing interfaces which contain methods that they do not need, but because the methods in *DecisionMaker* are closely coupled across implementations, there may be situations in which modifications to one interface need changes to all implementations. This could result in a type of technical debt where modifications might become more expensive and there is less flexibility.

Another issue that added to the technical debt of the project is the implementation of checking if the battery level is low or not (if the drone should go stop and go back to base to avoid crashing). Our current implementation of that is checking if the battery level goes below (0.002*initial battery the rescue mission started with) and stopping when that happens. However, there a few issues with that method. Although we ran thorough testing and checking all maps with different initial battery requirements, it is always best to avoid hardcoded values like 0.002 when designing software because this results in the code being less flexible and adaptable to different or more complex scenarios, which might require altering the code to make it more in line with requirements that are accurate and closer to in real life examples.. With the time restrictions of this project, we could not come up with a better implementation on time. For future reference, we could have come up with an equation based on the initial battery level, the size of the map, and the current position of the drone because the further the drone is from the base, the more battery it requires to go back.

These design choices have resulted in a technical debt that not only makes future program maintenance and extensions more difficult, but also goes against fundamental concepts of good object-oriented design, such as modularity and encapsulation. In the future, a concentrated effort to restructure the codebase and lower technical debt could be extremely beneficial to the project. This might entail implementing more design patterns that encourage high cohesion and loose coupling, adding more detailed interfaces, avoiding hardcoding, and implementing procedures that maintain encapsulation even during unit testing.

In summary, the project exhibits strong execution of specific features and adherence to many design principles. However, there are obviously issues arising from the technical debt produced by non-private visibility modifiers and close coupling to interface methods and a few other reasons. Taking care of these problems via refactoring and adherence to principles of encapsulation, SOLID, GRASP and GoF patterns can significantly improve the project's maintainability and extensibility, turning potential cons of the current design into strengths in future iterations.