Miguel Padilla & Diego Acosta

May 6, 2025

CS 3331-Advanced Object-Oriented Programming-Spring 2025

Dr. Gurijala

Project Part 2

This work was done individually/as a team and completely on our own. we did not share, reproduce, or alter any part of this assignment for any purpose. I/we did not share code, upload this assignment online in any form, or view/received/modified code written from anyone else. All deliverables were produced entirely on my/our own. This assignment is part of an academic course at The University of Texas at El Paso and a grade will be assigned for the work we produced.

Program Explanation

This project had us implement a debris analysis system that simulates the function of space agency control center. It reads a csv file that contains the debris and satellites, and allows different users such as Scientist, Space Agency Rep, Policy maker, and Administrator. This also allows the Scientist to perform the following tasks, perform space-tracking tasks such as filtering objects, assessing orbit status, calculating orbital risk, and generating debris density reports.

To perform this assignment, we first created a rough draft of the UML Diagram, and Level 2 UML use case diagram. For the UML Diagram we decided to use the following classes, Debris, Logger, Filereader, User, Administrator, SpaceAgencyRep, Scientist, RunSimulation, SpaceObject, and UI (Mission Control), UserExt, and Database. This allowed use to break down the problem to smaller more manageable tasks. With the Use case Diagram we created, it allowed us to see a high-level overview of how the program can function. We also used inheritance and encapsulation to ensure that we can properly strengthen our program to run with minimal errors, this allowed us to further breakdown the problem to be smaller and allowed us to check each class and method along the way of creating the project. We also have exception handling and Junit testing in the program as well, to make sure the file runs as intended.

What did I learn?

What we have learned is how to practically use UML use cases, and class diagrams and have it translated into the function of the program, we also learned how UML diagrams have allowed us to break down a problem to be smaller and allow us to transform a high-level overview problem to a low-level overview of the problem. We also have learned how going through a state diagram and check how the user can go through the program. Some classes could have been one space object and not creating a lot of different class files for them, such as Unknown, Satellite, Rocketbody, which we did do, and we then abstracted the User and have the Administrator, Scientist, and SpaceAgencyRep inherit User class. These were one of the ways we have improved the code and reduce the classes and points of failures, as well as, implementing exception handling. We assumed that the problem could have been a bit too daunting given the time frame, but as we slowly broke down the problem into smaller parts and implemented and tested the problem along the way, we found it to be doable. We have also assumed that there is already, and Administrator created.

Solution Design

The system design focuses on modularity, separation of concerns, and efficient data handling. To store and retrieve orbital object data, we used a HashMap<String, SpaceObject> to allow constant-time lookup of individual records using their unique recordId. This approach was chosen over ArrayList to support more efficient filtering and classification based on object properties. Subclasses like Debris, Payload, and Unknown inherit from SpaceObject, and polymorphism allows us to store them uniformly in the map while invoking their overridden methods where appropriate. The system applies object-oriented design principles such as abstraction, inheritance, and encapsulation, reinforced by helper classes like Logger for interaction tracking and Filereader for input management. We changed from ArrayList, since HashMap allowed us to handle the jumbled rso file.

We have also implemented inheritance as mentioned earlier by having Administrator, SpaceAgencyRep, and Scientist inherit from User. We also applied the Interface Segregation Principle, and single-responsibility principle by having Logger, FileReader, and RunSimulation, with each other own distinct role. Additionally, exception handling is encapsulated via custom exceptions like UnauthorizedAccessException, demonstrating clean separation of error logic. This structured use of object-oriented design promotes maintainability, readability, and scalability across the project.

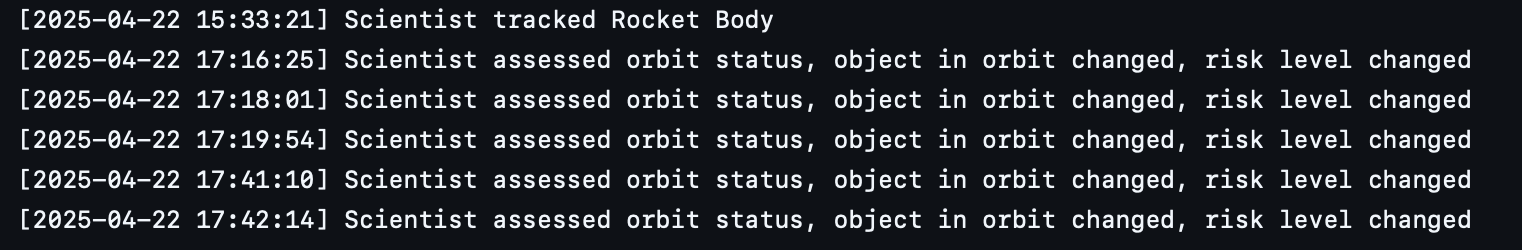
Additionally, we introduced a Factory Creational Design Pattern to streamline object creation, that the UserExt uses. Instead of relying on multiple constructors scattered across the program, we centralized object instantiation using a factory method that returns the correct type of User or more specifically to handle the dynamic creation of users. This approach increases flexibility, enhances maintainability, and ensures that new user types or object types can be added with minimal code changes. Overall, the system design promotes extensibility and clean abstraction. The system also follows the Model-View-Controller (MVC) architectural pattern. The Model includes the SpaceObject class and debris-processing logic, as well as the database to bolster the processing logic of the code base; the View is represented by the user-facing console output in UI.java; and the Controller is primarily handled by RunSimulation.java, which manages user interactions and application flow. This separation of concerns improves testability, readability, and modularity across the system.

**Testing**

We created a JUnit 5 test class `SysFuncTest.java` to verify the behavior of our custom exceptions. The test testInvalidMenuOptionException() checks that improper menu selections correctly trigger the InvalidMenuOptionException. This ensures the system’s role-based logic is enforced and that invalid operations don’t crash the system. We also have tests for the Logger Functionality, debris risk level calculation, and file reader with the jumble rso or columns. We also have conducted tests throughout the whole project, but those were more in general checks if the code can compile, we also did test runs on the whole project, and menu exception handling.

**Test Results**

Here we have the report of the debris:  


Here we also have the test result of the Logger:  


**Code Review**

Our team conducted a detailed code review using the provided checklist to ensure the project followed sound object-oriented design practices. The review confirmed that encapsulation was consistently applied attributes were private with public accessors where needed. Each class maintained a clear single responsibility, and naming conventions followed Java standards for clarity and consistency.

We verified that classes and methods were modular, focused, and reusable. The transition from multiple subclasses to a single SpaceObject implementation showed good use of abstraction while reducing redundancy. Additionally, the Logger class followed a centralized logging approach that enabled easy testing and verification of user activity. The introduction of the factory pattern to instantiate user roles and future object types was seen as a positive addition to support extensibility. Exception handling was properly implemented using custom exceptions (UnauthorizedAccessException), and unit testing provided verification for these cases. Overall, the project meets the checklist criteria for readability, maintainability, reusability, and correctness, with clear comments, Javadoc documentation, and robust user role handling.