**Khushi Desai - 21BCP264**

**Computer Engineering**

**Pandit Deendayal Energy University**

**Educational Initiatives**

**Exercise 2:**

**Satellite Command System Programming Exercise**

* **Project Structure:**

Satellite.java

SatelliteCommandSystem.java

MyLogger.java

|  |  |
| --- | --- |
| **1.** | **Satellite.java:** |
|  | This class represents the satellite and its functionalities. |
|  | public class Satellite {      private String orientation;      private String solarPanels;      private int dataCollected;      public Satellite() {          this.orientation = "North";          this.solarPanels = "Inactive";          this.dataCollected = 0;      }      public void rotate(String direction) {          if (isValidDirection(direction)) {              this.orientation = direction;              MyLogger.log("Satellite rotated to " + this.orientation + ".");          } else {              throw new IllegalArgumentException("Invalid direction: " + direction);          }      }      public void activatePanels() {          this.solarPanels = "Active";          MyLogger.log("Solar panels activated.");      }      public void deactivatePanels() {          this.solarPanels = "Inactive";          MyLogger.log("Solar panels deactivated.");      }      public void collectData() {          if (this.solarPanels.equals("Active")) {              this.dataCollected += 10;              MyLogger.log("Data collected. Total: " + this.dataCollected + " units.");          } else {              throw new IllegalStateException("Cannot collect data. Solar panels are inactive.");          }      }      public void displayStatus() {          System.out.println("Current Status:");          System.out.println("Orientation: " + this.orientation);          System.out.println("Solar Panels: " + this.solarPanels);          System.out.println("Data Collected: " + this.dataCollected + " units\n");      }      private boolean isValidDirection(String direction) {          return direction.equals("North") || direction.equals("South") ||                 direction.equals("East") || direction.equals("West");      }  } |
| **2.** | **MyLogger.java** |
|  | This class handles logging throughout the application. |
|  | import java.util.logging.Level;  import java.util.logging.Logger;  public class MyLogger {      private static final Logger logger = Logger.getLogger(MyLogger.class.getName());      public static void log(String message) {          logger.log(Level.INFO, message);      }  } |
| **3.** | **SatelliteCommandSystem.java** |
|  | This class manages user input and controls the main application flow. |
|  | import java.util.Scanner;  public class SatelliteCommandSystem {      public static void main(String[] args) {          Scanner scanner = new Scanner(System.in);          Satellite satellite = new Satellite();          String command;          System.out.println("Welcome to the Satellite Command System");            do {              satellite.displayStatus();              System.out.println("Enter command (rotate, activate, deactivate, collect, exit):");              command = scanner.nextLine().trim().toLowerCase();              try {                  switch (command) {                      case "rotate":                          System.out.println("Enter direction (North, South, East, West):");                          String direction = scanner.nextLine().trim();                          satellite.rotate(direction);                          break;                      case "activate":                          satellite.activatePanels();                          break;                      case "deactivate":                          satellite.deactivatePanels();                          break;                      case "collect":                          satellite.collectData();                          break;                      case "exit":                          MyLogger.log("Exiting the Satellite Command System.");                          break;                      default:                          MyLogger.log("Invalid command: " + command);                          System.out.println("Unknown command. Please try again.");                  }              } catch (IllegalArgumentException | IllegalStateException e) {                  MyLogger.log("Error: " + e.getMessage());                  System.out.println("Error: " + e.getMessage());              } catch (Exception e) {                  MyLogger.log("An unexpected error occurred: " + e.getMessage());                  System.out.println("An unexpected error occurred. Please try again.");              }          } while (!command.equals("exit"));          scanner.close();      }  } |

* **Key Features:**

1. **SOLID Principles**:
   * **Single Responsibility Principle**: Each class has its own responsibility (e.g., Satellite for state management, Logger for logging).
   * **Open/Closed Principle**: New features can be added by extending classes rather than modifying existing code.
2. **Error Handling**:
   * Specific exceptions (e.g., IllegalArgumentException, IllegalStateException) are used for better clarity on issues.
   * A general exception handler captures any unexpected issues.
3. **Logging**:
   * Java’s built-in logging framework is used to log actions and errors.
4. **Input Validation**:
   * Commands and parameters are validated to ensure valid actions are taken.

* **Code Explanation**

**1. Architecture and Class Interaction**

The project is structured around a few core classes that work together to create a clean and maintainable application:

**Satellite Class**:

This class encapsulates all the properties and behaviors of the satellite, including its orientation, solar panel status, and the amount of data collected. It provides methods to change the state of the satellite (like rotate, activatePanels, deactivatePanels, and collectData) and to display the current status.

**MyLogger Class**:

This class handles all logging functionalities. It centralizes the logging logic, making it easier to modify or extend the logging mechanism in the future without touching the core application logic.

**SatelliteCommandSystem Class**:

This is the entry point of the application. It manages user input and orchestrates commands by calling the appropriate methods from the Satellite class. It acts as a controller, ensuring that user interactions are correctly routed to the satellite's functionality.

The interaction flow is straightforward:

The SatelliteCommandSystem takes user commands.

Based on the command, it invokes methods on the Satellite instance.

Actions performed by Satellite are logged via the MyLogger class.

**2. Importance of SOLID Principles**

**Single Responsibility Principle (SRP)**:

Each class has a specific purpose. The Satellite class manages satellite states, the MyLogger class manages logging, and the SatelliteCommandSystem handles user interactions. This separation allows changes in one class without affecting others, improving maintainability.

**Open/Closed Principle (OCP)**:

The architecture allows for future extensions without modifying existing code. For example, if you wanted to add a new command (like resetting the satellite), you could do so by adding new methods to the Satellite class without changing the existing methods.

**Liskov Substitution Principle (LSP)**:

Although not explicitly demonstrated here, the architecture allows subclasses to be substituted for their parent classes. If we were to create a specialized version of Satellite, it could be used in place of the base class without breaking functionality.

**Interface Segregation Principle (ISP)**:

The design promotes small, focused classes. If we needed to introduce more commands, we could create additional classes for those specific functionalities rather than overloading the Satellite class.

**Dependency Inversion Principle (DIP)**:

By depending on abstractions (like logging), rather than concrete implementations, the system is more flexible. If you decide to change the logging mechanism (e.g., to a file-based logger), you only need to modify the MyLogger class.

**3. Logging and Error Handling**

**Logging**:

The logging mechanism uses Java's built-in logging framework, which records important events and state changes. This helps in monitoring the application’s behavior during execution and is useful for debugging. For instance, every time a command is executed, an entry is logged, providing an audit trail of actions taken by users.

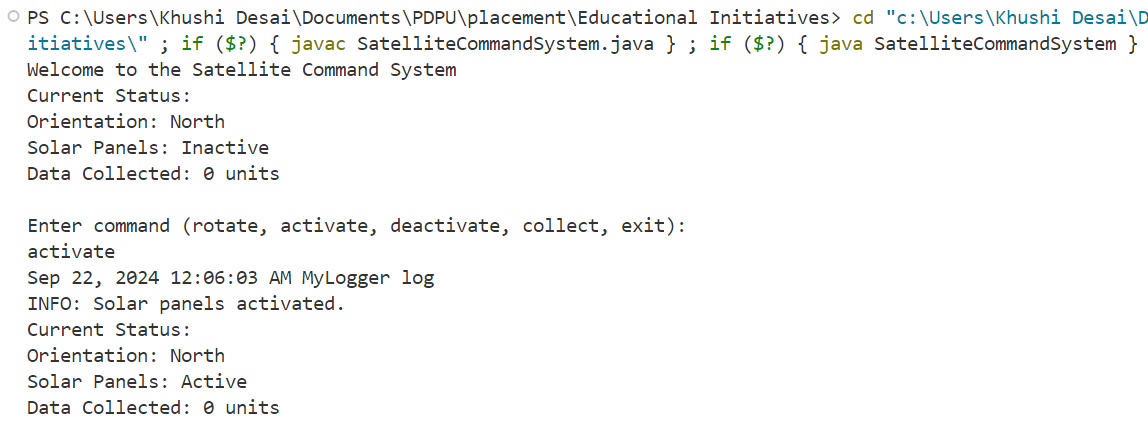
**Error Handling**:

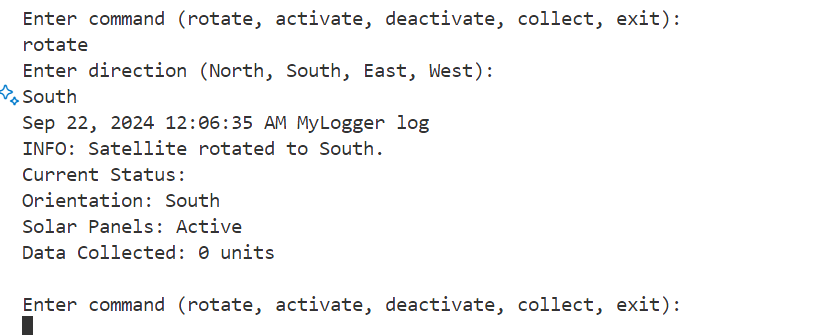
Specific exceptions (IllegalArgumentException, IllegalStateException) provide clarity about what went wrong and why. This structured approach allows users to receive informative feedback. For example, if a user tries to collect data while the solar panels are inactive, the system throws an exception with a clear message, guiding the user on what to correct.

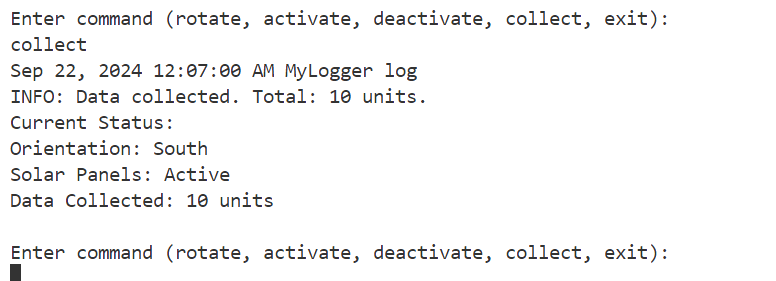
**User Experience**:

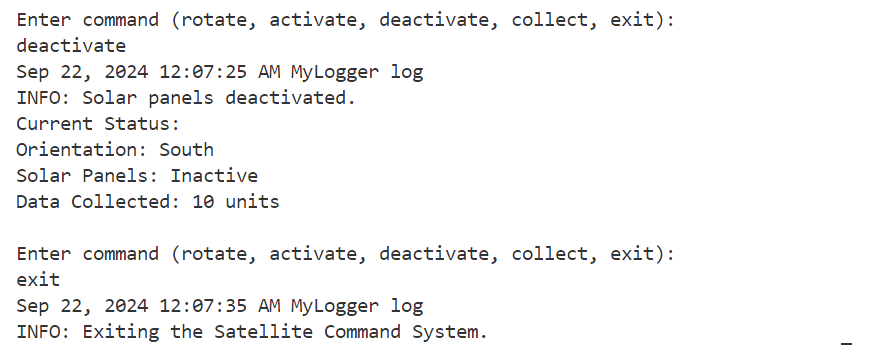
The combination of logging and error handling enhances the user experience by preventing the application from crashing unexpectedly. Instead, users receive helpful feedback that informs them of the issues. This not only improves usability but also builds user trust in the application’s reliability.

* **Output:**









* **Conclusion:**

In summary, this project exemplifies a well-structured application that adheres to good software design principles. The careful consideration of architecture through class responsibilities, the adherence to SOLID principles for maintainability, and the implementation of robust logging and error handling make it a reliable and user-friendly system.