EECS2030F: LAB 4 Due: Oct. 30, 2025 - 11:59 pm

The purpose of this lab is to ensure that you practice

A) creating a correct aggregation and composition relationship between objects.
B) Implementing inheritance relationships between superclass and subclasses.
C) Understanding object lifecycle and ownership in complex systems.

1. Lab Policies

a) **Academic Integrity**: Submit your own work. Do not copy code from classmates or online sources. All violations will be reported as academic misconduct.
b) **Submission Format**: Submit only the file: **SpaceMission.java**. Do not upload ZIP files or project folders.
c) **Work Environment**: Complete the methods and test your submission by using tester file**.** Your code should not contain a main method and Scanner Class objects and its methods. Ensure your code compiles without errors.
d) **Deadline:** Submit your **SpaceMission.java** file to the eClass course page by **Thursday, October 30 (11:59pm).** This is an extended deadline due to midterm on Oct. 29. No late submissions are accepted. Email submissions are not accepted.
e) Don't change the name of the file as we will not be able to test your code with our tester**.** It needs to be **SpaceMission.java** (no case change)
f) Your lab assignment is not graded during the weekly lab sessions scheduled. The lab sessions are meant to get your questions answered from TAs.

2. Downloading and importing the Starter Project

You have already done this step in previous labs. Follow the below listed steps to set up your working environment by:

a) Download the Eclipse Java project archive file from eClass: EECS2030_Lab4.zip
b) Launch Eclipse and browse to EECS2030-workspace (for instance or your own created workspace).
c) In Eclipse:
- Choose File->Import
- Under General, choose Existing Projects into workspace

- Choose Select archive file. Browse your compressed zip folder and attach it.
- Make sure that the EECS2030_Lab4 box is checked under Projects and you don't have the same project already in the workspace. Then Finish.
- You should see two files, one is called **SpaceMission.java** and one **SpaceMission.java**.

Note: You will see errors in both the files as the methods are not implemented. The moment you start implementing the code correctly, the errors will go away.

3. Important Notes:

To practice testing, we have only provided a set of incomplete test cases. Please make sure to add enough test cases to the tester that thoroughly tests your code. Look at the tester code, **SpaceMission Test.java**, and add more test cases to ensure comprehensive testing of your code. To do this, you can copy one of the methods, change the name of the method to avoid having a duplicate method name, and modify the body of the method to test your code with your selected input.

4. Javadoc generation

The Javadoc has been written for you. All you need to do is generate it as an HTML file to make navigation easier. To do this, click on the lab4 package, select Project -> Generate Javadoc. It will ask you for the location where you want to store the documentation. You can choose the default location or select your own folder. Enter the path, and then click on Finish. If you look at the location where you stored the documentation, you'll see a file called index.html. Clicking on this file will display the project documentation in your browser.

If you have any doubts on the Javadoc generation, please review lecture slides of week2.

5. Programming Tasks for this Lab

In this lab, you are going to implement a **Space Exploration Mission Management System** that demonstrates inheritance, composition, and aggregation relationships. You will create a superclass representing general spacecraft, two subclasses for specialized mission types, and additional classes that show both strong (composition) and weak (aggregation) object relationships.

Understanding the Class Hierarchy

The system is built around a central **Spacecraft superclass** that contains common attributes and behaviors shared by all types of space missions. This superclass defines fundamental properties such as mission identification, launch scheduling, fuel management, and operational status that every spacecraft needs regardless of its specific mission type. From this superclass, two specialized subclasses extend the basic spacecraft functionality. The **RoverMission subclass** represents spacecraft designed for planetary surface exploration, adding capabilities specific to ground-based operations such as terrain traversal, sample collection, and surface analysis. The **SatelliteMission subclass** represents spacecraft designed for orbital observation, adding capabilities for image capture, orbit management, and remote sensing from space.

Understanding Composition Relationship

Composition represents a strong "part-of" relationship where the component cannot exist without its container. The **PowerSystem** is an integral part of every spacecraft that manages all energy generation, storage, and consumption. Every spacecraft must have exactly one power system built into it at construction time. This power system tracks battery levels, monitors solar panel efficiency, and manages power consumption rates. The power system is so fundamentally integrated into the spacecraft that it cannot exist independently - when a spacecraft is destroyed or decommissioned, its power system ceases to exist as well. The power system is created internally within the spacecraft constructor and cannot be passed in from outside or shared with another spacecraft.

Understanding Aggregation Relationship

Aggregation represents a weaker "has-a" relationship where the component can exist independently of its container. The **Crew** represents the team of astronauts and specialists who operate spacecraft missions. Unlike the power system, crew members exist as independent entities who can be assigned to different missions throughout their careers. A crew can be assembled before a spacecraft is built, can be transferred between different spacecraft, and continues to exist even after a particular spacecraft mission ends. Multiple spacecraft missions might share crew members at different times, or a spacecraft might operate without any crew in unmanned mode. Crew objects are created independently outside the spacecraft and then assigned to missions as needed.

Implementation Requirements

For this lab, you will implement constructors, getters, and setters for all five classes (Spacecraft, RoverMission, SatelliteMission, PowerSystem, and Crew).

Task 1: Implementing PowerSystem class

Implement the PowerSystem class with an overloaded constructor and a copy constructor. This class represents the power management component that is composed within each spacecraft.

Requirements:

- Overloaded constructor.
- Copy constructor that creates a deep copy of another PowerSystem object
- Getters and setters for all four properties
- displayPowerInfo() is already implemented.
- Additional methods shown in UML

Important: PowerSystem objects should only be created within the Spacecraft class constructor, demonstrating composition. The PowerSystem cannot exist independently of a Spacecraft.

Task 2: Implementing Crew class

Implement the Crew class with an overloaded constructor and a copy constructor. This class represents personnel who can be assigned to spacecraft missions and demonstrates aggregation.

Requirements:

- Overloaded constructor.
- Copy constructor that creates a copy of another Crew object
- Getters and setters for all three properties
- displayCrewInfo() is already implemented.
- Additional methods shown in UML

Important: Crew objects are created independently outside of Spacecraft and passed to Spacecraft as a parameter, demonstrating aggregation. Crew can exist before, during, and after a Spacecraft's lifecycle.

Task 3: Implementing Spacecraft class (Superclass)

Implement the Spacecraft superclass with default, overloaded, and copy constructors. This is the parent class that will be extended by RoverMission and SatelliteMission.

Requirements:

- Default constructor that initializes all fields to default values and creates a new PowerSystem object internally with default values (e.g., batteryLevel=100.0, solarPanelEfficiency=95.0, powerConsumptionRate=5.0, maxCapacity=1000.0). The crew reference should be set to null.
- Overloaded constructor that accepts: missionID, launchDate, fuelCapacity, currentStatus, missionDuration, and a Crew object reference. Inside this constructor, create a new PowerSystem object with same specific values as default constructor.
- Copy constructor that performs deep copying for the PowerSystem and shallow copying for the Crew.
- Getters for all properties.
- Setters for all except for PowerSystem (composition - it cannot be replaced from outside)
- transmitData() is already implemented.
- displayInfo() is already implemented.
- Additional methods shown in UML

Task 4: Implementing RoverMission class (Subclass)

Implement the RoverMission subclass that extends Spacecraft. This class represents planetary surface exploration missions with specialized rover capabilities.

Requirements:

- Overloaded constructor.
- Must use super() to call the parent constructor for inherited properties
- Copy constructor that accepts another RoverMission object, uses super() to copy Spacecraft fields, and copies RoverMission-specific fields
- Getters and setters for all four additional properties
- analyzeTerrain(), deployInstruments() , driveDistance() is already implemented.
- Override the displayInfo() method from Spacecraft to include rover-specific information
- Additional methods shown in UML

Important: Demonstrate proper use of inheritance, constructor chaining with super(), and method overriding.

Task 5: Implementing SatelliteMission class (Subclass)

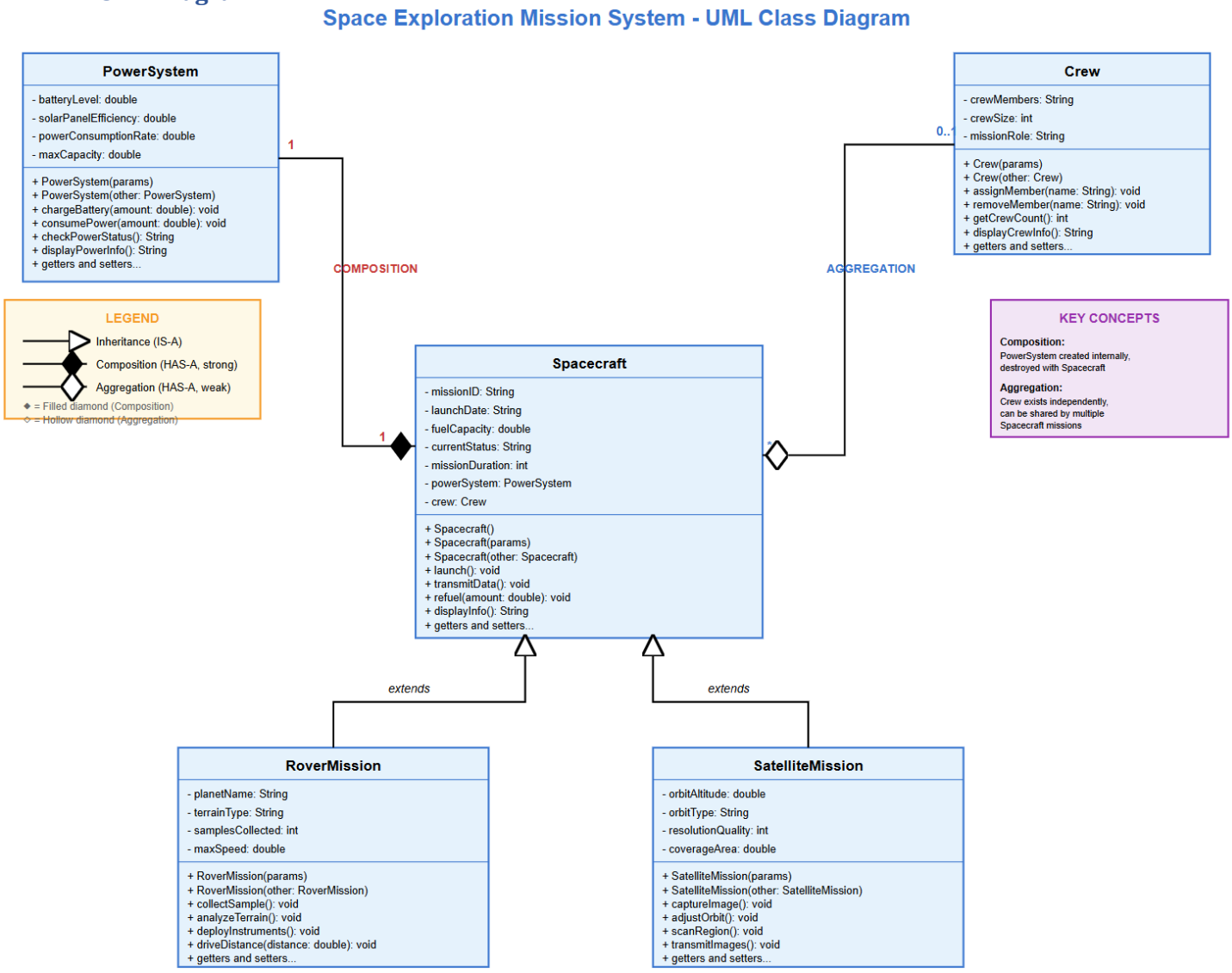
Implement the SatelliteMission subclass that extends Spacecraft. This class represents orbital observation missions with specialized satellite capabilities.

Requirements:

- Overloaded constructor that accepts all Spacecraft parameters plus SatelliteMission-specific parameters
- Must use super() to call the parent constructor for inherited properties
- Copy constructor that accepts another SatelliteMission object, uses super() to copy Spacecraft fields, and copies SatelliteMission-specific fields
- Getters and setters for all four additional properties
- captureImage(), adjustOrbit(), scanRegion(), transmitImages() is already implemented.
- Override the displayInfo() method from Spacecraft to include satellite-specific information

Important: Demonstrate proper use of inheritance, constructor chaining with super(), and method overriding.

UML Diagram:



6. Testing your code

A test case file has been provided to evaluate your methods. Your implementation should pass all the test cases if it correctly handles both normal and edge cases. **Do not modify or update the provided test cases in any way. You should create your own test cases too to thoroughly test your code.**

7. Submit

Submit only one file named " **SpaceMission.java** " via eClass by clicking on the lab link.

8. Marking Schema

You are graded based on the correctness of your code. For each method, there will be comprehensive test cases to examine the correctness of the code. All test cases carry the same weight.

Please note that in all the labs, even if the test cases are provided, we will test your code with a different set of test cases to ensure that you have tested your code completely.

Note:

- The program that does not compile will get zero marks.
- Don't change the method headers. The test cases will fail if you change the method headers.
- Your submission should strictly have the name as **SpaceMission.java**, otherwise it can not be used to test your assignment. It will have zero grades.
- No resubmissions are allowed. Therefore, please make sure you submit the correct file within due date.

****