EECS 3311

Software Design

Lab 2 due July 21, 2023

Problem Description

Consider an aircraft which has three GPS receivers obtaining at real time the position of the aircraft in terms of latitude and longitude values. The autopilot software of the aircraft sends requests to obtain the coordinates from the three GPS receivers. There are two brands of GPS receivers (two Honeywell and one Rockwell). The current system employs a majority voting strategy to determine the coordinate to use (i.e. if two of the GPS readings are within a tolerance limit, then any of the two readings is used (at random), but if all three readings are above the tolerance limit, then an error message is generated and the reading of the first GPS is used). Once the current coordinates are obtained the appropriate modules to control surfaces of the airplane are notified, so that the airplane can be steered to the right direction to reach the next waypoint which is stored in the autopilot.

Task:

Design the described system as a collection of classes. Apply the following design patterns: Proxy, Observer, Façade, Strategy.

What to submit:

A zipped folder containing the following items:

1. **(60% of total) An archive file with the implementation of the system** in the form of a project which can be imported to Eclipse. **Your implementation should use four design patterns worth 15% each.** Do not forget to include the client code to run the system. **Make sure you "tag" with a comment the use of a pattern in your code (e.g. "Using the Factory Method pattern").**
2. **(20% of total) A very brief document** with comments where (in which class / package) you used which design pattern and why. Use short but explanatory sentences. As well, this portion of the grade will also encompass **whether your implementation passes our testing**.
3. **(20% of total) A class diagram** (you may use Microsoft Visio or https://app.diagrams.net/ to draw it), clearly showing the used design patterns.

Assumptions:

1. Consider the longitude and latitude values that the receiver units will output to be **integer** numbers from 0-1
2. The GPS receiver units will randomly generate the longitude and latitude values based on the criteria in assumption 1.
3. For the majority voting strategy, instead of using tolerances (because we have integer numbers), if at least two GPS units have the same reading, then that is the reading accepted. If all three of the units disagree, then an error message is generated and we use the reading from the first GPS unit.
4. Consider that the messages to modules that move the control surfaces (i.e. the rudder, the ailerons, the elevator) will steer the aircraft in a proper way (i.e. the control surface actuator hardware's response to the messages are not our responsibility).

Process:

1. The Autopilot object sends a message (do this on a loop 100 times – this is already included as NavigationServer.java as your starter code) to the Proxy component (see Proxy pattern).
2. The NavProxy object calls the Navigation Façade
3. The Navigation Façade does the following sequence of steps (see Façade pattern)
    a. Send a request to GPSReceiver object #1 to get the first reading
    b. Send a request to GPSReceiver object #2 to get the second reading
    c. Send a request to GPSReceiver object #3 to get the third reading

    Don't forget, two of the GPSReceivers are Honeywell brand, and the other one is Rockwell brand.

    d. Apply the comparison strategy/strategies to get the resulting coordinates (see the Strategy pattern)
    e. Send the coordinates to the GPS Data Subject (see Step 4 below)
    f. Inform the Autopilot of the resulting coordinates *(however, this does not update the Autopilot's nextLat and nextLon)*
4. The GPS Data Subject notifies the Actuators (which move the control surfaces) with the current latitude and longitude values, and the desired path's latitude and longitude values (see the Observer Pattern).
    a. Note: per assumption 4, the actuator just feeds its output to the hardware it is connected to, and has no responsibility of actually steering the plane.

Code Structure:

The starter code provided already has several incomplete classes that are organized into packages.

- autopilotModule
  - AutoPilot class (partially provided)
    - Note: nextLat and nextLon are Coordinates that the client (i.e. NavigationServer) has fed to the autopilot. The application itself should not be updating it.
    - The Autopilot class' navigate method shall return the coordinates that it got from the façade via the proxy.
  - Coordinates class (fully provided)
- client
  - NavigationServer class (fully provided). This is the main entry point for running the application.
    - By default, the next latitude and longitude values are 5 and 2 respectively, as hardcoded into this class. Per assumption 4, **you don't need to change these values, nor does the project need to try to "steer" the plane into the desired path** (as this would be the responsibility of the hardware that receives the output of our actuators).
- controlSurfacesModule
  - GPSDataSubject class
  - IControlActuator interface (fully provided)
  - You need to complete the implementations of RudderActuator, AileronActuator, ElevatorActuator classes (issue to move the control surfaces based on the signals the autopilot gets and its target waypoint) which implement the IControlActuator interface.
  - Note that you somehow need to get the nextLat and nextLon values that were given to the AutoPilot, in order to use them in the classes above.
- coordinateComparisonModule
  - ICompareCoordsStrategy interface (fully provided)
    - There is a compareCoords method
  - You need to complete the implementations of two coordinate comparison strategy classes (TwoThreeVoting, FirstIsBest) which implement the ICompareCoordsStrategy interface (both are partially provided)
    - **For TwoThreeVoting, if there is no agreement, please return null**
- navigationModule
  - NavProxy class (partially provided)
  - NavigationFacade class (partially provided, please finish the implementation)
    - The shell of the constructor is provided, you will need to uncomment lines 26-29
    - Please have a method with the signature

      ```
      public Coordinates compareGPSData(List<Coordinates> data)
      ```

which will return the Coordinates object from the prevailing strategy that this method helps to choose (see Step 3d in Process)

- GPSReceiverModule
    - GPSReceiver
        - This will be either a class or interface; you need to decide what works best for your design
        - Keep in mind that we are using two brands of GPSReceiver, Honeywell and Rockwell
        - GPSReceivers "obtain" the coordinate reading by generating random **integer** values (ranging from 0-1 inclusive)
        - Fulfill the rest of the implementation

Hints:

- You are allowed to import from java.util.*; in fact, there are some imports of this already.
- You are allowed to import any of the interfaces/classes/packages in your project to the other classes/interfaces across the project.
- You should write unit tests for your classes to validate your implementation, but they are not mandatory to include in your submission.
- A small sample output has been provided with the starter code so that you know what to expect when you run the main class in NavigationServer. **We will be testing against having appropriate outputs in alignment with the sample output's format (for example: the voting strategy should print out text that includes "AGREEMENT WITH GPS2 AND GPS3" if GPS 2 and GPS 3 readings are in agreement).**

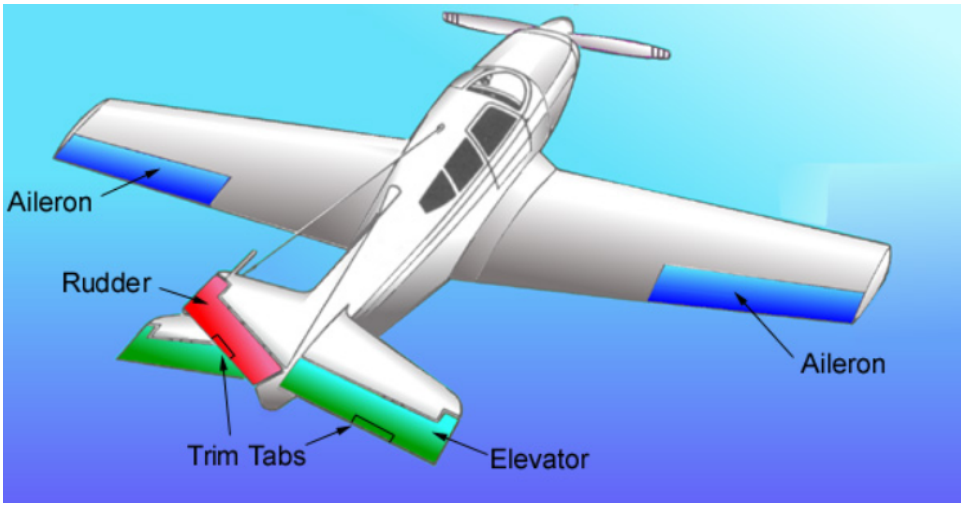

Fig. 1 Schematic for Rudder, Aileron, Elevator (ignore Trim Tabs)