Practical project: Boat

420-D04-SU
Object-oriented programming I: Java I
AEC Programming, Networks and Security (LEA.5F)
AEC Programming and Web Technologies (LEA.5G)
AEC Video Game Programming (LEA.CU)

Evaluation weight: 20% of final grade

Due date: Morning of the final exam

Submission specifications

- Project name: BoatProject + name 1 + name 2 [+ name 3]
- Your submission must Include all of your source code (**.java** files)
- Important: Double check that the **.java** files actually contain your code, all of your code, and the most up-to-date version of your code
- Your submission should include a documentation text file (a readme file). In this file, you should:
    - List which parts of the project each student worked on
    - Describe the application's features. In particular, describe the additional features your team designed and developed, as well as any other relevant information, including any remaining bugs or problems with your code
- Finally, compress your entire Eclipse project folder, including the readme file, into a **.zip** file
- Submit this **.zip** file in the correct directory on the Dropbox drive

Description

Create a graphical Swing application that simulates the functionality of a boat.

The boat has a position and a velocity*, and moves gradually over time, based on the velocity. It also has a fuel tank, which has a maximum capacity, and the fuel level can be between 0 and the maximum capacity.

The boat should also have various possible states. These could be something like:

Off, Running, Refilling fuel, Capsized

Using the graphical user interface, the user should be able to start the boat, stop the boat, increase its speed, decrease its speed, change its direction, and to refill the fuel.

The user interface should also have some way of displaying the position and velocity*, as well as the fuel level and fuel capacity. It should also give relevant messages to the user as needed.

* Velocity = both a speed and a direction

Rules

The various actions should be valid or invalid at various times, depending on the boat's state/data. You must design this logic in a way that is reasonable and accounts for all possibilities.

Driving the boat too fast can be dangerous. While driving the boat above a certain speed, there should be a chance (per second) that the boat will capsize and sink. This probability should increase as the speed increases above the limit. If the boat capsizes, it becomes inoperable (cannot perform actions).

At a certain position/region, there should be a shoreline. All of the available fuel stations are on land, at the shoreline, so the boat must return to shore in order to refill its fuel. If the boat runs out of fuel out on the water, the boat should automatically stop and become inoperable.

Driving the boat at a higher speed should consume more fuel than driving the boat more slowly over the same distance.

When the speed is dangerously high, or when the fuel is dangerously low, a warning should be displayed to the user.

Additional features

In addition to these requirements, you are encouraged to add some features or functionalities to your application.

Every additional feature added must be documented and described in a readme file contained in the project submission.

Evaluation

This is a team project, and it must be produced by the entire team working together as a group. All work must be original, and must not be copied from any external sources, including any students in other teams or classes. All members in a team are expected to write a roughly equal amount of the code.

The project will be graded according to the following criteria:

Exactitude

- Does the program do what it is supposed to do, and meet all requirements?

Input validation

- The application should correctly validate all user input
- The application should handle any errors that can occur during input, including displaying user-friendly error messages to the user

User experience and visual presentation

- All text displayed to the screen should be well-arranged and written in proper English
- The user interface should be user-friendly: it should be easy to use and behave predictably
- The user interface should show the right information to the user at the right time

Architecture

- The application's architecture should be well-designed and appropriate to the application's needs
- The model and view must be completely separate and decoupled from each other
- Some form of model-view-controller architecture is strongly encouraged
- Using an event system is also recommended. This is likely to be useful for keeping your user interface updated and in sync with the model. For example, make your boat model an event source that notifies all listeners whenever a relevant event occurs in the boat

Structure

- The code should be divided into classes and methods, according to the relevant needs
- Each method should perform a single simple task (usually a few lines of code)
- Each class should only contain fields and methods that are closely related to each other
- If a class contains members that are not conceptually or logically related to each other, then the class should probably be split up in to two or more classes
- Unchanging values should be stored in constants, rather than hard-coded as literal values

Format and conventions

- The code should contain relevant comments
- The code should be properly indented
- The code should respect naming conventions
    - The names of variables and methods should be written in **camelCase**
    - The names of classes/interfaces/enums should be written in **UpperCamelCase** (Pascal case)
    - The names of constants should be written in **UPPER_CASE**