

## TuteLab 4 – Extended Courier Management System (CMS)

### The Scenario

The courier company has decided to expand and diversify its delivery fleet by adding aircraft, in addition to its vans and trucks. Hence, the company would like to extend (and rework if necessary) the Courier Management System designed in the previous TuteLab. Despite the now diverse range of transportation modes, the system should still operate in the manner introduced in the previous week.

#### *Aircraft*

Like other Vehicles an aircraft has a registration (for identification), and a year, make and model. The maintenance schedule, however, is significantly different. Instead of using the distance travelled as a service indicator, an aircraft requires maintenance after a certain number of flights, *or* after a certain number of flying hours. Like in the case of vehicles, the maintenance schedule should be protected from being modified arbitrarily.

As the system works with distance and not time, flying hours are estimated by dividing the distance travelled against a pre-determined average speed of the craft. (This average speed figure is modifiable.)

An aircraft is considered unsuitable for work if *either* the flight count, *or* service interval (the flying hour count) is exceeded.

An aircraft's wear and tear estimate can be calculated as \$30,000 + \$5 per kilometre travelled.

Looking back at the courier scenario (both van+truck, and van+truck+aircraft), some of your original design ideas from the previous week might not lend themselves to being extended or maintained.

When thinking of extending the scenario, consider two scenarios:

- a) As specified in TuteLabs 2 and 3 (i.e. land and air vehicles)
- b) A hypothetical situation involving non-vehicle delivery systems (e.g. shipping container etc.)

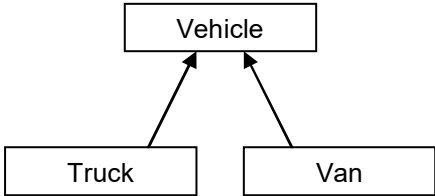
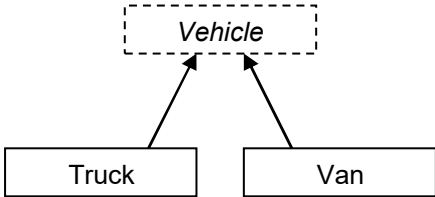
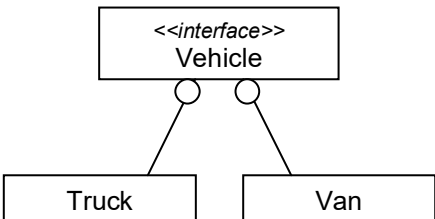
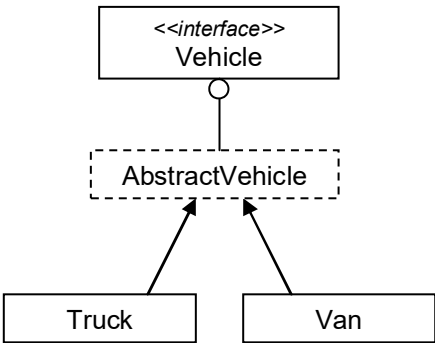
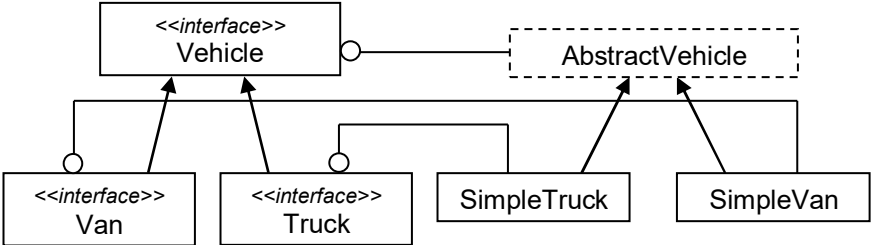
As with the previous tutelab you can use the provided an (updated) application driver class (CMSTestHarnessExtended.java) and OutputTraceExtended.txt to test your system in a structured manner. These files are again available on Canvas alongside this tutelab sheet. As before it places some constraints on your implementation in terms of the required classes, constructors and methods and some additional comments regarding the constructors etc.

**Design Hints .. below are some prototypical situations and partial design hierarchies. Note that these do not represent complete solutions (*continued overleaf*)**

For each of the alternatives A-E, try to answer the following questions to help you decide on the best approach for this scenario (keeping in mind that it could be extended in the future (again)!) )

- a) What are the advantages of this approach?
- b) What are the challenges or disadvantages of this approach?
- c) How can this approach simplify the process of extending the system in the future to deal with other type of vehicles, different maintenance schedules etc.?

**FINAL HINT:** If you haven't abstracted out the servicing functionality into a separate class/interface hierarchy then this is a good way to leverage most of the functionality of the previous specification since it is mostly the servicing that is different with Aircraft!

<p><b>A. Simple Inheritance Hierarchy</b></p>	 <pre> classDiagram     class Vehicle     class Truck     class Van     Vehicle &lt; -- Truck     Vehicle &lt; -- Van </pre>
<p><b>B. Inheritance with Abstract Class</b></p>	 <pre> classDiagram     class Vehicle     class Truck     class Van     Vehicle &lt; -- Truck     Vehicle &lt; -- Van     class Vehicle {         &lt;&lt;abstract&gt;&gt;     } </pre>
<p><b>C. Simple Interface Implementation</b></p>	 <pre> classDiagram     class Vehicle     class Truck     class Van     Vehicle &lt; -- Truck     Vehicle &lt; -- Van     class Vehicle {         &lt;&lt;interface&gt;&gt;     } </pre>
<p><b>D. Vehicle Interface with Abstract Vehicle</b></p>	 <pre> classDiagram     class Vehicle     class AbstractVehicle     class Truck     class Van     Vehicle &lt; -- AbstractVehicle     AbstractVehicle &lt; -- Truck     AbstractVehicle &lt; -- Van     class Vehicle {         &lt;&lt;interface&gt;&gt;     }     class AbstractVehicle {         &lt;&lt;abstract&gt;&gt;     } </pre>
<p><b>E. Full Parallel Interface Hierarchy "Shadow"</b></p>	 <pre> classDiagram     class Vehicle     class Van     class Truck     class SimpleTruck     class SimpleVan     class AbstractVehicle     Vehicle &lt; -- Van     Vehicle &lt; -- Truck     Van &lt; -- SimpleTruck     Van &lt; -- SimpleVan     Truck &lt; -- SimpleTruck     Truck &lt; -- SimpleVan     class Vehicle {         &lt;&lt;interface&gt;&gt;     }     class Van {         &lt;&lt;interface&gt;&gt;     }     class Truck {         &lt;&lt;interface&gt;&gt;     }     class AbstractVehicle {         &lt;&lt;abstract&gt;&gt;     } </pre>

