Algorithms and Data Structures

Abstract Data Types and Linked Lists in a Trains Configuration System

Assignment-1

Version: September 5th, 2023

Introduction

In this assignment you will work on an application that deals with trains.

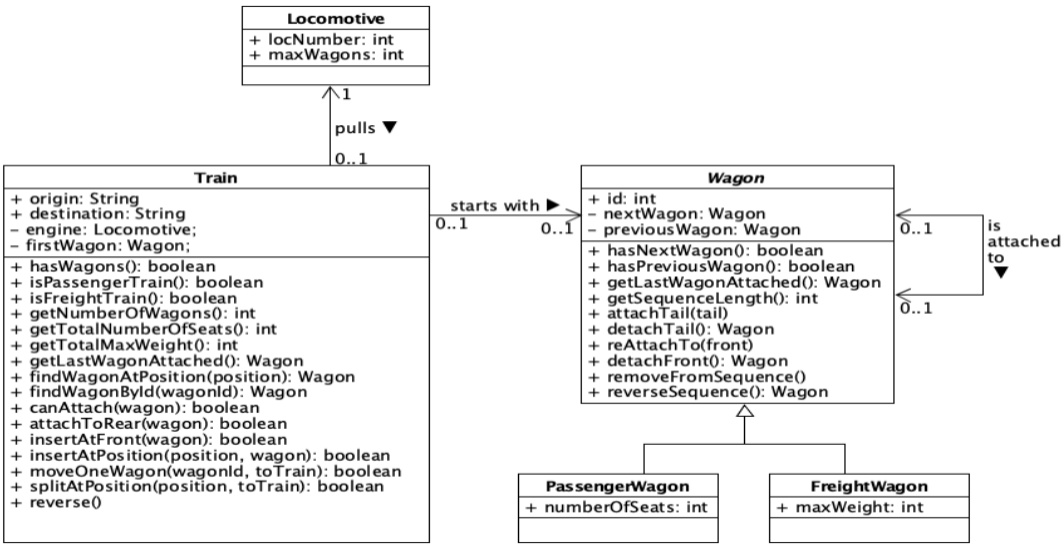
A train always has a locomotive (the engine) to pull the train. A Wagon can be hooked directly to the locomotive or to another wagon. Of course, wagons can also be detached from a train and they can be moved from one train to another. Not only single wagons can be moved around, also complete sequences of multiple wagons can be attached or split-off in one go.

Trains can also be reversed; that is when the engine is removed from the head of its sequence of wagons and reattached at the tail. By that the complete sequence of all wagons in the train reverses. This is an operation that you can frequently observe at the end-station of a service.

In this assignment we handle both passenger trains and freight trains. Passenger trains can only contain passenger wagons; freight trains only freight wagons. Each passenger wagon has a specific number of seats; each freight wagon has a maximum weight that can be loaded.

Engines have a capacity: that is the maximum number of wagons it can pull. So, the total number of wagons in the train may never exceed the capacity of its engine.

In the UML class diagram below, you find the design of our Trains Configuration System. It is your task to provide a Java implementation of this design, using the starter project template that has been provided as well.



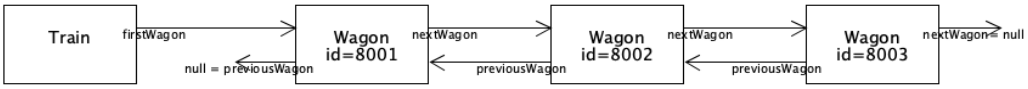
First, we provide some clarifications about the design:

i. Getters and setters are not shown in the class diagram. You should implement them as required and replace public class properties with private member variables and public getters and setters. (private class properties should have a private setter or no setter.)

ii. Wagons cannot be instantiated; only PassengerWagons and FreightWagons can be instantiated.

iii. The arrow tips at the end of the association between two classes indicate that the source object instance has a reference to the target object instance. I.e. a train knows its first wagon (by the firstWagon instance variable), but the first wagon does not know its train (there is no train instance variable in Wagon, and no arrow tip on the association from Wagon to Train.

iv. The data structure of a train uses a custom 'doubly linked list' representation to realize the sequence of connected Wagons as in the picture below:



Every wagon in this list knows its successor and predecessor in the train.
More precisely we can formulate 'representation invariants' of this data structure, i.e:

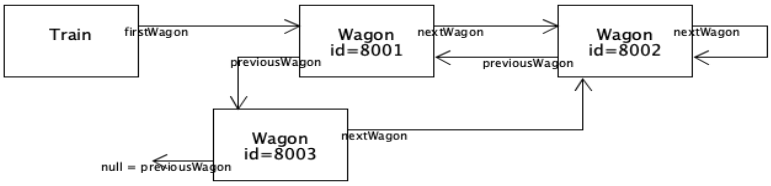
for all wagons w:

(w.nextWagon == null || w.nextWagon.previousWagon == w) and
(w.previousWagon == null || w.previousWagon.nextWagon == w)

In plain English this says:

"Every wagon is the predecessor of its successor, if any" and
"every wagon is the successor of its predecessor, if any"

All your public method implementations may assume validity of this representation invariants at the start of the method and should sustain these invariants at completion of the method irrespective of what changes in connections you need to make. If you fail to do so, you may end up with impossible train configurations like for example:

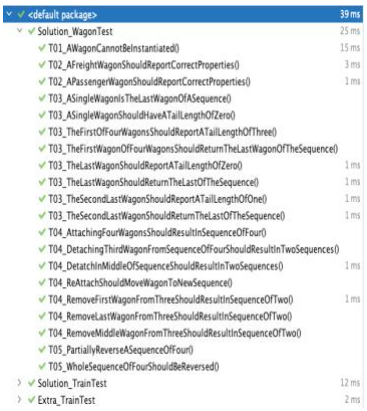


v. In the Wagon class you find different variations of attach and detach methods, a.o.:
attachTail(tail) should attach the wagon *tail* including all connected wagons behind the current wagon, but throw an exception if *tail* already has a predecessor or if the current wagon already has wagons behind itself. The message of the exception shall be clear about which two wagons already have a conflicting connection, e.g. it shall say something like:
"Wagon8007 has already been attached to Wagon8001"
or *"Wagon8001 is already pulling Wagon8007"*
detachTail() detaches the tail sequence of the current wagon and returns the head wagon of that tail sequence.
reAttachTo(front) attaches the current wagon behind the provided *front* wagon as its new tail. But before making such new attachment it should first detach the predecessor of the current wagon and the successor of the *front* wagon, if any, such that no conflicts arise.
So this method will not throw any exception, but rearrange the connections.
detachFront() detaches the current wagon from its predecessor, and returns that predecessor, if any.
All these methods you will find useful to help out with robust implementations of other Wagon and Train methods. The comments in the starter project explain further.

vi. A train has an engine which pulls a sequence of wagons which starts at its 'firstWagon' reference. A train is an example of an implementation of a single-ended, doubly linked list.
All train methods also shall sustain the representation invariant and throw no exceptions.
They rearrange wagon connections as needed to rearrange their composition.
The comments above each method in the starter project explain further.

Assignment

1. Complete all implementations of all methods in the starter project, such that you comply with the requirements and the design. (Also sustain the representation invariants of the design.)
2. Ensure Trains and Wagons can be printed and produce output as in the sample below.
3. Select appropriate data structures and algorithms that support efficient execution while still showing well readable code. (Specifically, avoid use of unnecessary, untransparent if-then-else structures, i.e. minimize cyclometric and cognitive complexity of your code.)
4. Leverage appropriate encapsulation of data and methods with useful parameters.
5. Adhere to HvA coding conventions and provide useful comment lines.
6. Ensure that all provided unit tests run fine.
   (The unit tests start with a prefix code. This prefix indicates some dependency between methods. Focus on resolving the issues with earlier unit tests before worrying about the later tests.)



7. Add additional unit test as you find appropriate to assure quality of your solution. (The provided unit tests only verify a subspace of your problem domain. Your implementation should also pass tests of other realistic situations within the boundaries of this casus.)
8. Provide a report that shows, explains, and justifies the seven most relevant code snippets of your solution (relevant from the perspective of complexity and smartness of your solution…)
   Explain how you have sustained the doubly linked list representation invariant in one of the more complex rearrangement methods.
   Apply the use of a loop-invariant in the explanation of one of your more complex rearrangement methods that involves a loop.
9. **First commit and push the starter project and work incrementally from there, also committing intermediate progress of your work into the Git repository. Both students in your team should be seen to contribute from their personal HvA account.**

Sample output of TrainsMain

```
Welcome to the HvA trains configurator
[Loc-24531][Wagon-8001][Wagon-8002][Wagon-8003][Wagon-8004][Wagon-8005][Wagon-8006][Wagon-8007] with 7
wagons from Amsterdam to Paris
Total number of seats: 258

Configurator result:
[Loc-24531][Wagon-8003][Wagon-8002][Wagon-8001][Wagon-8004] with 4 wagons from Amsterdam to Paris
Total number of seats: 126
[Loc-63427][Wagon-8007][Wagon-8006][Wagon-8005] with 3 wagons from Amsterdam to London
Total number of seats: 132
```

Grading

At DLO you find the rubrics for the grading of this assignment. There are three grading categories: Solution (50%), Report (30%) and Code Quality (20%). Your Solution grade must be sufficient, before the grading of Report and Code Quality is taken into account. Similarly, your Report grade must be sufficient before the code quality grade is granted.

For a 'Sufficient' score of a basic solution you must succeed with implementations of all methods subject to above requirements, except for the implementation of the *reverse()* and *reverseSequence()* methods and the explanation of the loop-invariant. Those are taken into account for the 'Good' grade of the full solution.