

Algorithms and Data Structures

Abstract Data Types and Linked Lists in a Trains Configuration System

Assignment-1

Version: 25 August 2020



Introduction

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

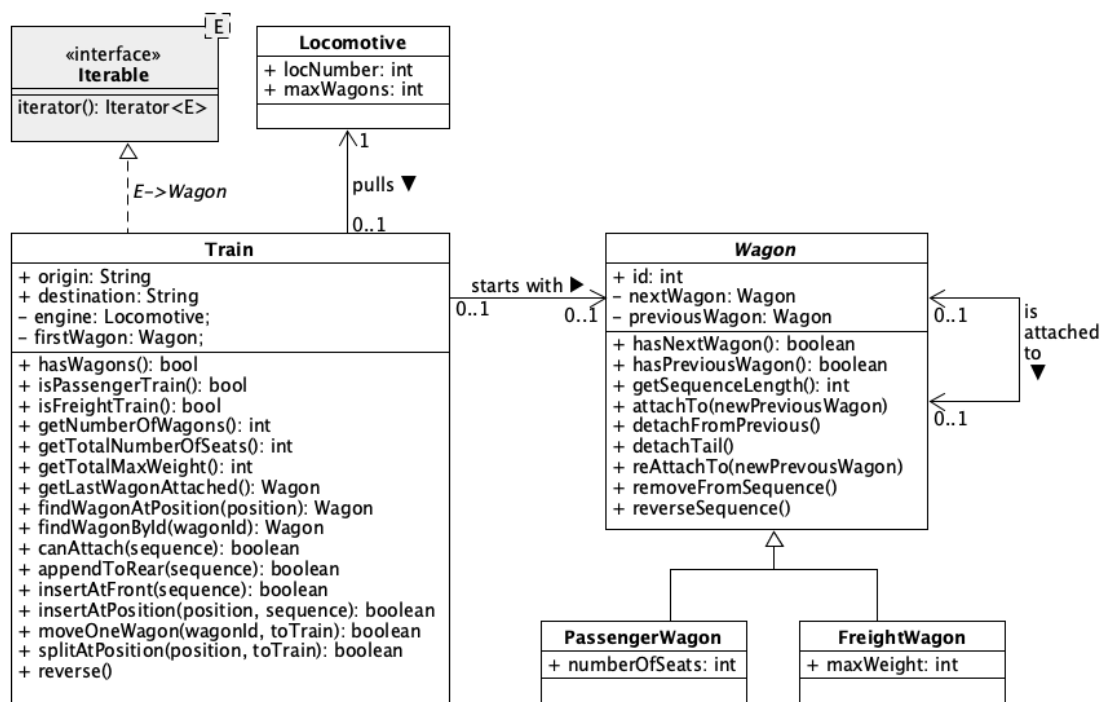
A train always has a locomotive (the engine) to pull the train. A Wagon can be hooked directly to the locomotive or to another wagon. Of course, wagons can also be detached from a train and they can be moved from one train to another. Not only single wagons can be moved around, also complete sequences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

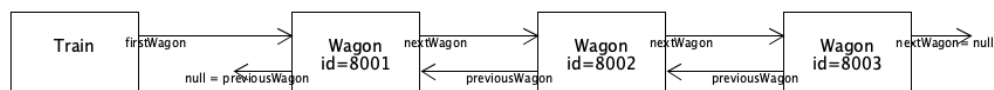
Engines have a capacity: that is the maximum number of wagons it can pull. So, the total number of wagons in the train may never exceed the capacity of its engine.

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.
- ii. Wagons cannot be instantiated; only PassengerWagons and FreightWagons can be instantiated.
- iii. The arrow tips and 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 representation of the train uses a 'doubly linked list' to represent the sequence of connected Wagons as in the picture below:



Every wagon in this list knows its successor and predecessor in the train.

For that, your code should at all times sustain the 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)

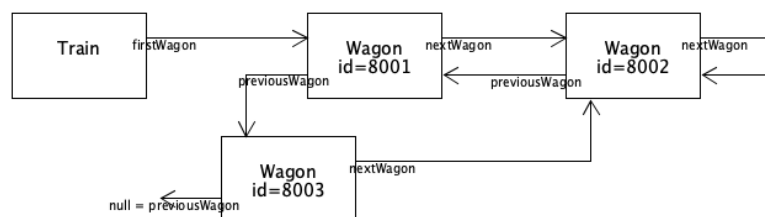
In plain English this says:

“Every wagon is the predecessor of its successor, if any” and

“every wagon is the successor of its predecessor, if any”

All your method implementations should sustain these invariants at completion of the method.

If you fail to do so, you may end up with impossible train configurations like for example:



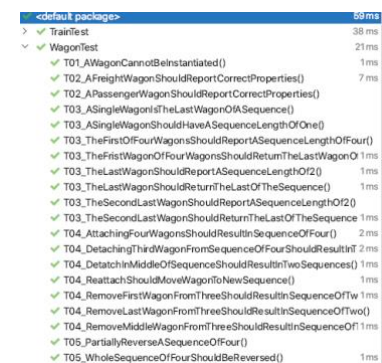
- v. In the Wagon class you find different variations of attach en detach methods:
attachTo(newPreviousWagon) should attach this wagon to a new previous wagon, but throw an exception if this wagon already has a predecessor or the newPreviousWagon already has a successor.
reAttachTo(newPreviousWagon) should first detach the predecessor of this wagon and the successor of newPreviousWagon, if any, before making the new attachment.

Both of these methods you will find useful to help out with a robust implementation of other Wagon and Train methods. The comments in the starter project explain specific pre- and post conditions of all methods to implement.

- vi. A train implements the Iterable Interface and should be able to iterate all its wagons.

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. Apply at least once a recursive solution.
4. Implement the Iterable interface on a Train, and use it at least once in your code by means of the syntax:
for (... : ...) { ... }
5. 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 complexity of your code.)
6. Leverage appropriate encapsulation of data and methods with useful parameters.
7. Adhere to HvA coding conventions and provide useful in-line comments.
8. Ensure that all provided unit tests run fine.
(The unit tests start with a prefix code. This prefix indicates interdependencies between methods. Focus on resolving the issues with earlier unit tests before worrying about the later tests.)
9. 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 with any realistic alternative test data, if attempted)
10. 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...)
11. Work incrementally and check-in intermediate results of your work into the Git repository. Both students in your team should be seen to contribute from their personal HvA account.



<default package>	59 ms
TrainTest	38 ms
WagonTest	21 ms
T01_AWagonCannotBeInstantiated()	1 ms
T02_AFreightWagonShouldReportCorrectProperties()	7 ms
T02_APassengerWagonShouldReportCorrectProperties()	
T03_ASingleWagonShouldHaveASequenceOfOne()	
T03_TheFirstWagonOfFourWagonsShouldReturnTheLastWagonOfFour()	1 ms
T03_TheFirstWagonOfFourWagonsShouldReturnTheLastWagonOfTwo()	1 ms
T03_TheLastWagonShouldReturnTheLastOfTheSequence()	1 ms
T03_TheSecondLastWagonShouldReturnTheLastOfTheSequence()	1 ms
T03_TheSecondLastWagonShouldReturnTheLastOfTheSequence()	1 ms
T04_AttachingFourWagonsShouldResultInSequenceOfFour()	2 ms
T04_DetachingThirdWagonFromSequenceOfFourShouldResultInTwoSequences()	1 ms
T04_DetachInMiddleOfSequenceShouldResultInTwoSequences()	1 ms
T04_ReattachShouldMoveWagonToNewSequence()	1 ms
T04_RemoveFirstWagonFromThreeShouldResultInSequenceOfTwo()	1 ms
T04_RemoveLastWagonFromThreeShouldResultInSequenceOfTwo()	1 ms
T04_RemoveMiddleWagonFromThreeShouldResultInSequenceOfTwo()	1 ms
T05_PartiallyReverseASequenceOfFour()	1 ms
T05_WholeSequenceOfFourShouldBeReversed()	1 ms

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 Wagon iterator. Those are taken into account for the 'Good' grade of the full solution.