Object Oriented Programming with Java 1
Exam Project

Hilmar Simonsen <hilmars@setur.fo>
Niels-Christian Borbjerg <nielsb@setur.fo>

May 5, 2023

Contents

1 Background

The international railway company *Ironbend Train and Brain Railway Inc.* has presented a consulting assignment to solve an urgent IT problem. There are rumours in the company saying that there is total chaos when it comes to their rail vehicles. It seems to be pure luck if a train can depart for the right destination at the right time — if there is any train available at all. Nobody has a clear view of the logistics of the vehicles and the whole company is at jeopardy. In the long term, the company will need tools for administering the vehicle fleet and trains, but the goal for our assignment is to construct a prototype for an application to simulate the operations during a 24-hour period. Given the initial data for stations, trains, vehicles, and the timetable, you are to simulate the assembly of trains at the departure stations, the running of the trains and their disassembly at the destination stations. The expected outcome will be information on which trains arrived at their destination on time, which trains were delayed etc.

2 Problem description and specification

The overall goal is to construct the simulation prototype which *Ironbend Train and Brain Railway Inc.* is in dire need of.

2.1 Object-oriented modelling

The first phase in the project covers the creation of an object-oriented model that encompasses trains and vehicles. The model is based on a set of classes and relations between these classes which should be implemented.

The basis of the model is the following facts and descriptions:

1. There are two types of engines: diesel engines and electrical engines.
2. There are two types of passenger-carrying cars: coach and sleeping car.
3. There are two types of freight cars: open freight car and covered freight car.
4. Each train has at least one engine and a number of cars. A train can have more than one type of engines and cars
5. Every vehicle, i.e. engines, passenger cars and freight cars, has a unique id-number.
6. Every train connection has its own train number. The train number is a logical concept that relates to a communication between two stations at a certain point in time. In this concept lies:
   - Types of vehicles (engines and cars)
   - Order of vehicles
   - Departure station
   - Destination station
   - Departure time
   - Time of arrival at the destination

   E.g., train 859 departs from South Bend central at 16.13 and arrives in Tahoma City 19.43 every day and it always consists of a diesel engine followed by 3 coaches and 2 covered freight cars. On the other hand, *the train number does not say anything about exactly which specific vehicles (identified by their unique id) are assembled to make up the train. This can vary depending on which vehicles are available at the station on the occasion at hand.*
7. Every train passes through a series of states throughout its life cycle. These states are:
   - `NOT ASSEMBLED`
     The train exists only as a logical concept (see point 6 above). No vehicles are connected.

- `INCOMPLETE`
  The train is partly assembled but one or more vehicles are missing on the station.
- `ASSEMBLED`
  The train is successfully assembled.
- `READY`
  The train is complete and ready to leave.
- `RUNNING`
  The train has left the departure station heading for its destination.
- `ARRIVED`
  The train has arrived at its destination station.
- `FINISHED`
  The train has been disassembled at its destination station and added to the pool of available vehicles at that station. The vehicles are available for further use in other trains.

8. A train is assembled from available vehicles at the departure station. For each train the user should always be able to get information about:
   - Train number
   - Departure station and time
   - Destination station and time
   - Current state
   - Type of every vehicle that the train consists of (in the right order) — and if the train is fully or partially assembled the information for each vehicle should be extended with its id and other attributes (see 9. below)
   - Average speed in km/h calculated from departure time, destination time and distance.
9. The following attributes of the different vehicle types are part of the model:
   - Coach - number of chairs, internet yes/no
   - Sleeping car - number of beds
   - Open freight car – cargo capacity in tons and floor area in square metres
   - Covered freight car – cargo capacity in cubic metres
   - Electrical engine - max speed in km/h and max power in kW
   - Diesel engine - max speed in km/h and fuel consumption in litres/h

Start by carefully thinking about a vehicle hierarchy and how you want to model the relationship between different types of vehicles. Ideally you should also be able to both implement and reason about your design choices.

2.2 Applying the model for simulation

The second phase in the project is the development of the simulation application. The function of this prototype is described as follows:

- The running of a number of trains are simulated from one point in time to another point in time during the same day starting at 00:00 and ending at 23:59.
- In the simplest way the simulation time is advanced in 10-minutes intervals by the user striking a key.
- For each time interval the current simulation time is output as hh:mm to the screen. All changes in the states of the trains that have occurred during the last interval are also output to allow the user to monitor the chain of events. If no state changes have occurred, only the current time is output.
- As an alternative to advancing the time the user should always be able to get information about

  1. The timetable: trains with train numbers, stations and times
  2. A chosen train: train number, state, stations, times, vehicles
  3. A chosen station: trains at the stations with states and vehicles, available vehicle in the pool

  Information about a vehicle should always include id and type.
- Statistics is shown after completed simulation. This includes:
  - Which trains were successfully run without delays
  - Which trains were delayed and how much
  - Which trains never left their departure stations and why
  - The accumulated delays in minutes

Each station has a pool of vehicles available to assemble trains. The content of the pool will change during the simulation as departing trains are assembled and arriving trains are disassembled.

The life cycle of a train can be described by the states and the conditions for state changes.

- A train starts in the state `NOT ASSEMBLED`
- An attempt to assemble the train is made 30 minutes before its departure time. If there are more than one vehicle of the requested type available, the one with the lowest id should be chosen. If it is successfully assembled according to the specification its state is changed to `ASSEMBLED`. If, on the other hand, at least one vehicle is missing its state is changed to `INCOMPLETE`. New attempts to assemble the train shall then be made every 10 minutes until it succeeds, and the state is changed to `ASSEMBLED`, or the simulation is ended. For every failure to assemble the train, 10 minutes shall be added to its departure time and time for arrival.

- The state of a train is changed from `ASSEMBLED` to `READY` 10 minute before its current departure time.
- On departure the state is changed to `RUNNING`.
- On arrival to its destination the state of the train is changed to `ARRIVED`.
- 20 minutes after arrival the train is disassembled, and its vehicles are placed in the vehicle pool of the station. The train has now reached its final state which is `FINISHED`.

The possible state changes can be depicted in an UML state diagram as seen in 1.

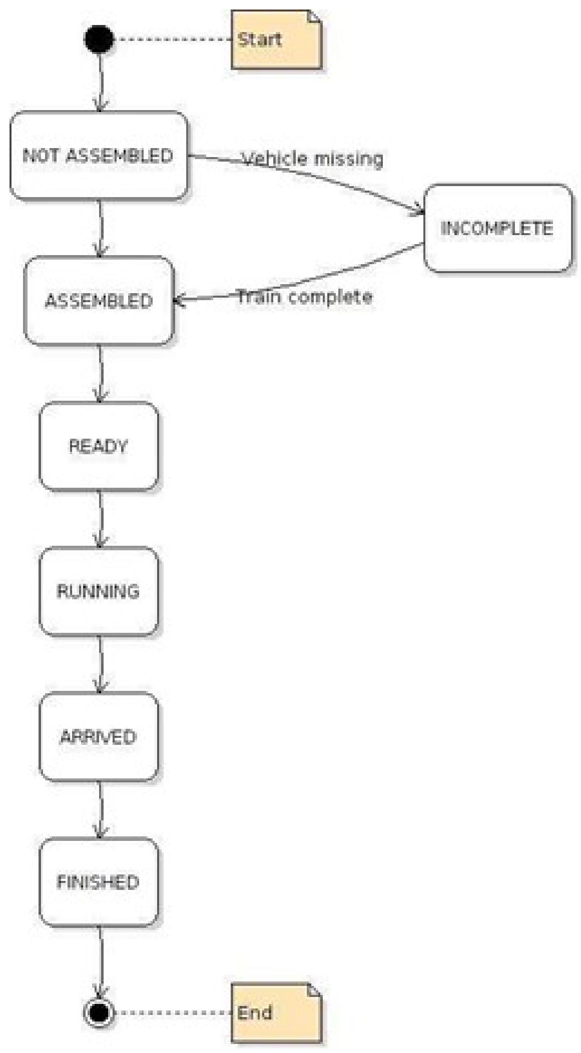

Figure 1: State changes during train life cycle.

If a train comes into the state `INCOMPLETE`, this means that it will be delayed. New attempts to complete the train must be made every 10 minutes until the train is complete and can enter the state `ASSEMBLED`, or the simulation is complete. For each failure to make the train complete, the times of departure and arrival are to be delayed by 10 minutes. The consequences of this is will be that some trains will arrive late to their destination and that some trains will never leave their destination.

The simulation data, i.e. all data concerning trains, stations, vehicles are read from the three provided data files at program start-up time. These files are described in section 2.3.

Information to be read is:

- The vehicles in the system, their id, type and attributes
- The stations in the system and which vehicles there are in their respective vehicle pool
- The trains with train number, departure station / time, destination station / time, number and types of vehicles, max speed and distance

The program must only know the names of these data files, no other simulation data must be hard-coded. On reading these files, the corresponding objects should be created dynamically.

The simulation spans one day and start at 00:00. In an ideal world without delays all trains have reached their destinations no later than 23:59. If a train due to a delay has started but not reached its destination at 23:59 the simulation should continue until the train has reached the `FINISHED` state. Put another way round: the simulation stops when: 1) all trains have reached the `FINISHED` state **OR** 2) time has passed 23:59 and no train is in the `RUNNING` or `ARRIVED` state.

If the simulation continues after 23:59 no trains are allowed to be assembled or started as we are only interested in trains that are departed during the day in question. The obtained results are presented when the simulation is finished.

This includes information on:

- Which trains reached their destinations on time
- Which trains were delayed (and why)
- Which trains could never leave their departure stations (and why)
- Total delay

2.3 Data files

The data used for simulation, i.e. data regarding trains, stations and schedule, are stored in the files `TrainMap.txt`, `TrainStations.txt` and `Trains.txt`. Two sets of these files are supplied — one for Windows and one for UNIX/Linux/OSX.

`TrainMap.txt`:

This file contains the distances between stations. Each row of the file contains three fields:

```
    stationName stationName2 distance
```

Distances are symmetric meaning if the entry `<Dunedin Luz 65>` is present in the file, then the converse case `<Luz Dunedin 65>` will not be present.

`TrainStations.txt`:

This file contains information on which vehicles are at what station at the start of the simulation. Each row in the file holds data for one stations in the format

```
stationName (id type param0 ...)(id type param0 ...)
```

Note that . . . indicates that zero or more parameters might follow, e.g. `(id type param0 param1 param2)`. More concretely `id` is the unique id, `type` is the type encoded as an integer from 0 to 5. Each type has different parameters representing different things:

```
coach : 0               // type 0 represents the coach car

    param0 : int        // number of chairs
    param1 : int        // internet access (0 = no access, 1 = access)

sleeping car : 1        // type 1 represents the sleeping car

    param0 : int        // number of beds

open freight car : 2    // type 2 represents the open freight car

    param0 : int        // cargo capacity in tonnes
    param1 : int        // floor area in square meters

covered freight car : 3 // type 3 represents covered freight car

    param0 : int        // cargo capacity in cubic meters

electrical engine : 4   // type 4 represents an electrical engine

    param0 : int        // max speed in km/h
    param1 : int        // power in kW

diesel engine : 5       // type 5 represents a diesel engine

    param0 : int        // max speed in km/h
    param1 : int        // fuel consumption in liters/h
```

`Trains.txt`:

This file describes the trains. Each row contains data for one train as follows:

```
trainId from to departTime arriveTime maxSpeed type ...
```

`trainId` is the train number representing the train connection, `from` is the station of departure, `to` is the destination station, `departTime` is the time of departure, `arriveTime` is the expected time of arrival at the destination station, `maxSpeed` is the maximal speed, while enumeration of `type` indicates the types of vehicles that make up the train encoded as intergers from zero to five, note again that . . . simply means zero or more types of vehicles can follow. The max speed is also given as an integer, but it should be treated as a double in calculations.

2.4 Extensions

The above constitutes the basis for the project. The specification should be extended with

1. User option for choosing start and end time of the simulation.
2. User option for changing the interval length.
3. User option for bypassing the fixed interval length and let the simulation run until the next state change has occurred.
4. User option for getting the current location of a vehicle given its id.
5. User option for getting a history of the movements of a chosen vehicle so far at any point in the simulation.
6. Implementation of calculated average speed of the trains. Let the average speed be affected by delays. Implement that a train increases its speed in proportion to its delay with the restrictions that it can never exceed its maximum speed and it may never reach its destination before the original arrival time. This means that a delayed train will get 10 minutes added to its departure time for each failed assembly attempt but the arrival time is recalculated.

The more extensions that is provided in your project solution the better — of course given that they do not jeopardize your ability to provide a decent base solution.

3 Report

Write a project report containing:

- A front page with names of the course, the project and yourselves
- A summary of the task
- An account of the used platform and your development tools
- A description of your solution. Focus on important implemtation and design decisions — how are classes related, how are they co-operating, how is responsibility distributed across classes, etc.. You can use UML-diagrams to illustrate this.
- Describe any shortcomings of your solution in regards to the above specifications.

4 Requirements and assessment

Your solution should

- Meet the description and specification as stated above and thereby produce a correct simulation result.

- Catch exceptions — no runtime crashes due to erroneous input.
- Be commented in a way that contributes to an understanding of what goes on.
  Example of a non-comment: `++i // increase i by one`
  Methods should be annotated with a Doc comment[1].
- Provide a project report that describes your program.

For assessment of the project before the oral exam attention will be allocated to:

- **Goal achievement**
  Does your solution comply with the description and with the requirements?
- **Structure**
  Division of the problem domain into classes with attributes and related methods. Assignment of responsibilities among classes and functions.
- **Usage of OOP and Java language related concepts**
  To what extent does your solution benefit from features like e.g. inheritance, polymorphism, generic collections and similar.
- **Source code**
  Consequent typography and indentation. Descriptive identifier names and meaningful commenting.
- **User friendliness**
  Informative and easy-to-use interface.
- **Project report**
  Structure, layout and content.

5 Submission

The Assignment is to be turned in online on BitBucket in your student repository in the folder "Project". Be sure to commit and push your solution and report to BitBucket.

- The Moodle Project submission is to be a text-file with the EXACT name: Project.txt
- The text file should contain ONLY the URL for your BitBucket project repository.

The exam project should be handed in no later than 23:59 on May 26th.

[1]See *here* for a guide on Doc formatting.