Transport Tycoon DDD exercises

This document is a contribution by me (https://twitter.com/tonyxzt) for the #fsadvent.

My post is about a DDD programming exercise (https://github.com/Softwarepark) in F# language.

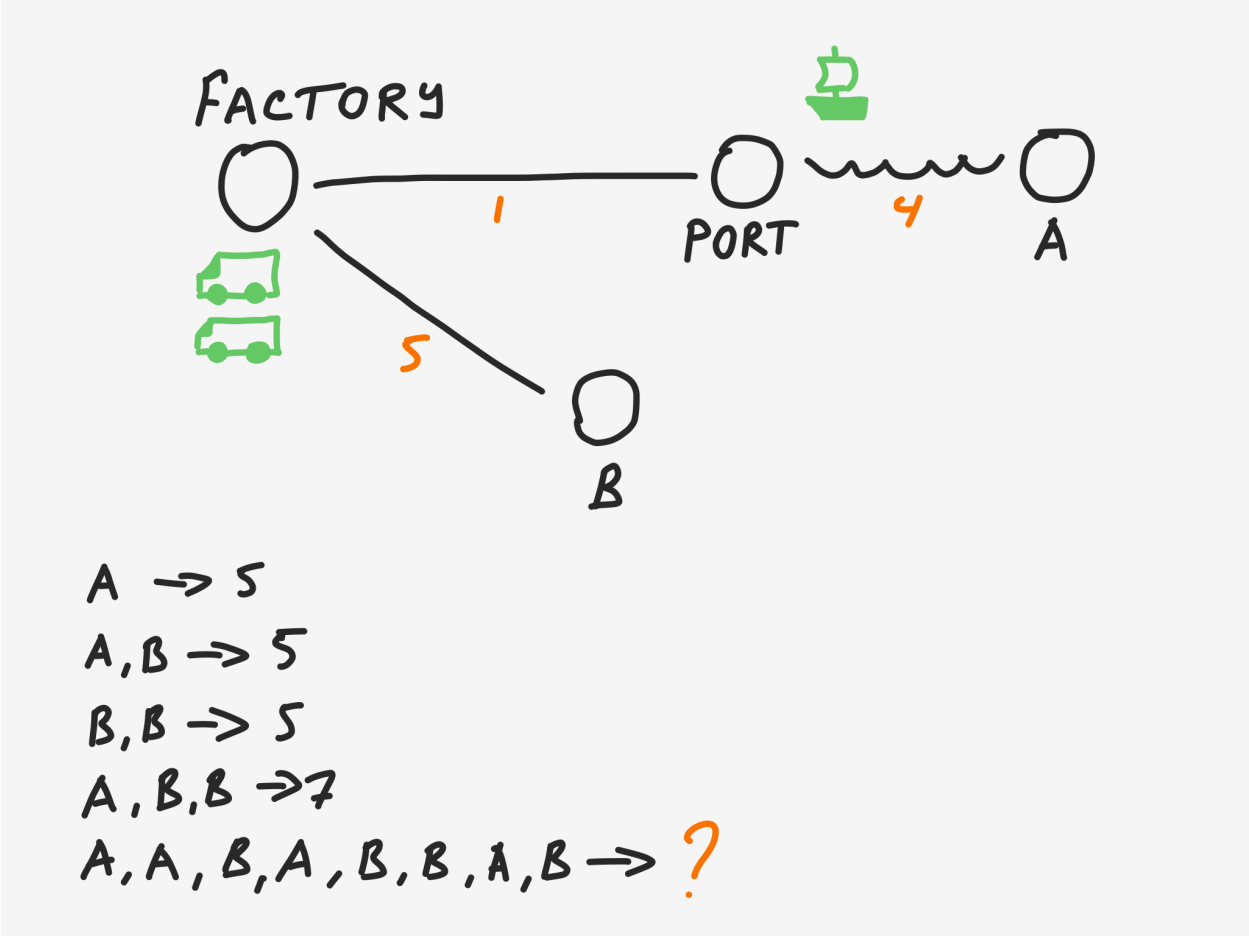
My solutions are:
Assignment 1
Assignment 2

I am still working on them, but I experienced some technical problems with my laptop so didn't have time to fix the solution as desired. I hope you don't mind. ;-)

A quick overview of the assignment.

There is a transport network with some cargo (containers) directed to some destinations. There are vehicles like trucks and ships that will bring cargo to their destinations.

The destinations can be A or B.
The first assignment is: given a sequence of destinations for cargos, like AAB, compute the time passed until all the cargos will be at their destinations, assuming there are two trucks starting from the factory and one ship starting from the port.
This is the network:



(IMG. from https://github.com/Softwarepark)

The second assignment requires to log the relevant events and to transform them into a specific JSON format, to feed an external batch process able to visualize such events.

There are also other requirements I'll not discuss here, about a latency time required for the ship to to load and drop cargos, and about the possibility for the ship to transport up to four cargos at one time, rather than just one.

The structure of the project.

The structure of the solution in F# is based on two separate projects (one for the test and one for the library) under the same solution. They are based on this how to: https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-fsharp-with-nunit

So there is a test project and a main project.

Editing the code on this solution is ok using either Jetbrain rider or VsCode with Ionide and test Explorer extensions.
VsCode may need a little bit more of configuration for debugging an test running respect to Rider which is more immediate.

A closer description of my solution with snaps of code:

There are four different locations:

```
type Location = FACTORY |  PORT | A | B
```

A “path” will indicate two locations, and the time needed to cover their distance

```
type Path = {StartNode: Location; TimeToTravel: int;  EndNode: Location }
```

(Note: "path" is what graph theory calls "arc").

A vehicle, after loading a cargo plan a tour given by a path to go and a path to return back.

```
type GoAndBackPaths = {GoPath:Path;BackPath:Path}
```

A vehicle has also a position that can be *moving* or *stationary*:

```
type Position = Moving of Moving  | Stationary of Location
```

Moving means being somewhere in a path:

```
type Moving = {Connection:Path; RelativePosition:int}
```

So the vehicle is defined as follows:

```
type Vehicle = {
        VehicleName:string;
        Cargo: Cargo option;
        Tour: GoAndBackPaths option;
        Position:Position
}
```

For a cargo, if the destination is A, then the list of the paths contains two paths, one from the factory to the port, and one from the port to A:

let factoryToPortPath = {StartNode=Factory;EndNode=Port;TimeToTravel=2}
let portToBPath = {StartNode=Port;EndNode=B;TimeToTravel=2}

The following will represent a cargo that is directed to A:

let cargo = {CargoName="cargo"; Paths = [factoryToPortPath;portToAPath]; NodeLocation = Some Factory }

Now that we have defined cargos, vehicles, directions, locations, paths, looks like the only thing that we need is a representation of the state of the world:

```
type World = {Vehicles: Vehicle list; StationaryCargos:Cargo list}
```

There are some stationary cargo that are somewhere and are directed somewhere. Any vehicle available will bring them toward the direction where they are directed.

Essentially we update the world by a “tick”, that makes the following actions:

. Make vehicle load cargos
. move vehicle to the direction given by the cargo they loaded, or to the direction for reaching back the base point (the factory).
. make vehicles drop the cargos when they reached the location.

The end of the task is reached when all the cargos are at their destination.

I’ll give a quick lock at some code relative to the “tick”, without going too much into the details.

```
let tick world =
       let (newVehicles,leftCargos) = tryLoadCargos world.Vehicles world.StationaryCargos
       let newVehicles2 = updateAllVehiclesPositions newVehicles
       let (newVehicles3,droppedCargos) = tryDropCargos newVehicles2
       let newVehicles4 = tryAlignVehiclePositions newVehicles3
       {Vehicles=newVehicles4;StationaryCargos=droppedCargos@leftCargos}
```

The “tryloadcargos” will return the vehicles eventually with cargo loaded, and the cargo that are left out because they cannot be loaded by any vehicle.
The “updateAllVehiclePositions” update the vehicle's position according to their direction.
After that, vehicles that reached some destination where the cargo are supposed to be dropped, will actually drop them, by the “tryDropCargos”.
The “tryAlignVehiclePositions” will just set the vehicle position or movement as needed. For instance switching the vehicle direction.

I don’t want to go further explaining the behavior of the function called by the update before I had the time to simplify them (for instance removing a too heavy use of recursion, where I can use List.fold instead)

About calculating the time for all the cargo reaching their destination I used the "unfold" function for generating sequences:

An example of using unfold is the following (from https://docs.microsoft.com/it-it/dotnet/fsharp/language-reference/sequences)

```
let seq1 =
    0 // Initial state
    |> Seq.unfold (fun state ->
        if (state > 20) then
            None
        else
            Some(state, state + 1))

printfn "The sequence seq1 contains numbers from 0 to 20."

for x in seq1 do
    printf "%d " x
```

I used the fold in a similar way as the previous example to create a sequence of all the state until the condition “all the cargos are at their destination” is reached:

```
let numberOfTicksForTheCargoBeingAtDestination (world:World) nCargos  =
        let newWorld = world |> tick
        let theSeq =
        (0,newWorld)
        |> Seq.unfold (fun (c,w) ->
                if (List.length
                        (w.StationaryCargos
                        |> List.filter
                                (fun x -> match x.NodeLocation with | Some X when (X = A|| X = B)
                        -> true | _ -> false )) =  nCargos) then
                None
                        else Some((c,w),(c+1,tick w))
                )
        Seq.length theSeq + 1
```

As we can see the condition to exit is that all the stationary cargos are in A or B. When the sequences of the state reaches such point then I just get the length of the sequence.

Excercise part 2

Part 2 essentially requires introducing the logging of events like *depart*, *arrive*.
Such events must be transformed in a predefined json format like follows:

```
{
  "event": "DEPART",      # type of log entry: DEPART or ARRIVE
  "time": 0,              # time in hours
  "transport_id": 0,      # unique transport id
  "kind": "TRUCK",        # transport kind
  "location": "FACTORY", # current location
  "destination": "PORT", # destination (only for DEPART events)
  "cargo": [              # array of cargo being carried
    {
      "cargo_id": 0,      # unique cargo id
      "destination": "A",# where should the cargo be delivered
      "origin": "FACTORY"# where it is originally from
    }
  ]
}
```

I decided to slightly change my record types, so that they would better fit the desired format, (for instance introducing the “transportId”), and then transforming them in an object based on the structure suggested for json. Then there is one more transformation into “transfer objects” with similar structure except that the types are all string or integers. This is needed to make the newtonsoft json serialization work as expected.

```
type DepartLogItem = {
        Event: EventType
        Time: int
        TransportId: int
        Kind: Kind
        Location: Location
        Destination: Location
        Cargo: Cargo list
}
```

For the aim of transforming them to json I have a similar record that uses only int and strings:

```
type DepartLogItemDto = {
        event: string
        time: int
        transport_id: int
        kind: string
        location: string
        destination: string
        cargo: CargoDto list
```

```
}
```

Using this kind of structure I can serialize them by using newtonsoft library obtaining exactly the format required by the assignment.

The log items are added to the “world” as long as the events are generated, so we have a new definition of the “world”:

```
type World =
{
        Vehicles: Vehicle list;
        StationaryCargos:Cargo list;
        Time: int;
        Logs: LogItem list;
}
```

Each time a log item must be produced then it will be appended to the Logs field.

To make the world update the number of times needed until all the vehicles are “stationary”, and then get the logs I again used the “unfold” function for sequences:

```
let showLogs (world:World) nVehicles  =
        let newWorld = world |> tick
        let theSeq =
        (1,newWorld) |> Seq.unfold (fun (c,w) ->
                if (List.length
                        (w.Vehicles |> List.filter
                                (fun x -> match x.Position with | Stationary _   -> true | _ ->
                        false )) =  nVehicles)
                then
                        None
                else
                        Some((c,w),(c+1,tick w))
        )
(snd (theSeq |> Seq.last)).Logs |> List.map (fun x -> LogItemToDto x)
```

The "showLogs" will tick the world and query the log. Fair enough for the exercise, even though a cleaner approach could be doing those two tasks separately or at least using memoization to store the already computed “world states” so that they can be reused without recomputing them.

Recap: I gave a quick view of the solution without going too much deeper into the single function because they need improvement I may talk about in a future doc.

The description of the domain in terms of records and types, using for instance union, option and records works well for instance compared to classical o.o. languages.

Any use of o.o. Language probably would make it heavier in terms of presence of more boilerplate code, whereas, and the lack of option type in classic o.o. Languages (like Java or C#) can be responsible for potentially more subtle runtime errors like null pointer exceptions.

Option types helps communicating information relevant for the domain, and prevent errors because the typical way of dealing with option type is by using pattern matching which is more “bullet proof” against “none” values.

(note: Ironically, I have been able to violate the “robustness” of the option type because somewhere in the code I used statement like object.Value rather than pattern matching for an option type. It will generate an error if object is “None”, in the same way it may happen in classic “option-less” languages.

Of course a better way to deal with such kind of object is pattern matching like:
match object with
| Some X -> ...
| None -> ...)

That’s all for now. Thanks for reading.