Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano

Prof. Matteo Camilli, Elisabetta Di Nitto, and Matteo Rossi

20133 Milano (Italia)
Piazza Leonardo da Vinci, 32
Tel. (39) 02-2399.3400
Fax (39) 02-2399.3411

Software Engineering 2 – Written Exam 2 (WE2)

January 10th, 2024

Last Name, First Name
Id number (Matricola)
Number of paper sheets you are submitting as part of the exam

Notes

A. You must write your name and student ID (matricola) on each piece of paper you hand in.
B. You may use a pencil.
C. Incomprehensible handwriting is equivalent to not providing an answer.
D. The use of any electronic apparatus (computer, cell phone, camera, etc.) is strictly forbidden, except for an ebook reader.
E. The exam is composed of 2 parts, one focusing on requirements, and one focusing on design. Read carefully all points in the text.
F. Total available time for WE2: 1h and 30 mins

System Description: TrafficZoneMonitor (TZM)

The municipality of a big city needs to monitor the vehicles circulating in the city center (think "Area C" in Milan), and for this it wants to create a new system, called **TrafficZoneMonitor (TZM)**. The system detects vehicles circulating in the City Center Area (CCA for short) in 2 possible ways:

- Through "gates" installed on the streets through which vehicles can enter the CCA. Each time a vehicle goes through a gate (which is assumed to be already available in the city), the gate reads the license plate of the vehicle, and provides TZM with the corresponding information (license plate number and time of passage of the vehicle), as further discussed later.
- Through monitoring devices installed on vehicles (not every vehicle is equipped with such devices). Each device has a GPS, through which it periodically collects the position of the vehicle. Each device also provides, as discussed later, the collected information (license plate number of the vehicle, position, timestamp) to TZM.

Users of the CCA must register with the system before accessing it. Accessing the CCA is free for residents of the area, but for non-residents it has a cost, which is computed by TZM. There are two different kinds of tariffs: for vehicles that do not have monitoring devices installed, and for those that do. Vehicles that do not have the monitoring device installed pay a daily fee for each day that they enter the CCA, irrespective of the amount of time the vehicle spends in the CCA, or how many times they enter the CCA during the day. Vehicles that have the monitoring device installed, instead, pay a daily fee according to how many kilometers they drive in the CCA in that day. Fees are due only when the CCA is in operation, which occurs only between 7am and 9pm each day.

TZM uses the information gathered from gates and monitoring devices to compute, each day after the CCA stops operating, how much is due by every non-resident user of the CCA. At the end of each day, TZM sends an invoice to each non-resident user whose vehicle has circulated in the CCA, notifying the user of how much he/she has to pay. Payment is handled by a different system, separate from TZM, but TZM must provide a way for the payment system to retrieve the information about the invoices to be paid (TZM does not keep track of whether a generated invoice has been paid or not).

Consider the following goals for the TZM system:

- **G1**: The municipality needs to keep track of how many vehicles (of non-resident users) are circulating in the CCA every day.
- **G2**: The municipality needs to determine how much each non-resident user of the CCA needs to pay for each day of use.

Part 1 Requirements (7 points)

RASD_Q1 (2 points)
Identify the actors relevant for the TZM system and explain how they interact with the system.

RASD_Q2 (2 points)
Define suitable world and machine phenomena for the TZM system, focusing on those relevant for goal G1.

RASD_Q3 (3 points)
Define suitable Domain Assumptions and Requirements related to goal G2.

Part 2 Design (7 points)

DD_Q1 (2 points)
Focus on how TZM could collect data from gates and monitoring devices. Gates (which, as mentioned in the overview of the system, are already installed in the city) are equipped with an embedded computer, which can be normally programmed. Similarly, monitoring devices can be programmed as regular mobile devices (e.g., Android-based boards).

Discuss what strategies and approaches are most suitable for TZM to collect data from gates and monitoring devices (synchronous, asynchronous, push, pull, publish-subscribe, etc.). You can use UML Sequence Diagrams to illustrate your points. Would you use the same strategy for both gates and monitoring devices?

DD_Q2 (2 points)
Suppose we want to structure TZM using a microservices-based approach. We identify two microservices, *UserManager* and *VehicleManager*, which store information about users and vehicles, respectively, and allow other services to retrieve that information. More precisely, the user information concerns their personal data (name, address, etc.), and the license plate number of the vehicles they own; the vehicle information, instead, concerns the identifier of the vehicle owner, plus static data regarding the vehicle model, production year, etc. None of the above-mentioned microservices has dynamic information regarding the detected positions of vehicles, etc.

Identify 3 additional microservices that are used to realize goal G2, and describe their purpose (notice that 3 microservices, in addition to *UserManager* and *VehicleManager*, are not necessarily enough to cover all functions involved in goal G2).

DD_Q3 (3 points)
Define, for each of the microservices identified at point DD_Q2, their interface, in terms of provided operations and related parameters (you can use a UML Class Diagram, if useful, to define suitable types for the operation parameters).

Solutions

RASD_Q1
The actors TZM is interacting with are the following ones:

- Gates, which read license plates of vehicles and communicate them to TZM.
- Monitoring devices, which detect the position of vehicles, and communicate them to TZM.
- Users, who register to ZTM, provide data about vehicle, and receive invoices from TZM (and pay them through an external system). To be more precise, one could separate Users in 2 categories: residents and non-residents. This would be reasonable if also residents could interact with the system, for example to inform it when they change vehicle. In this case, "non-resident" Users would simply be what are called "Users" here.
- External Payment System, which needs to retrieve invoices from TZM.
- Municipality Officers, who use TZM to check data regarding accesses to the CCA.

RASD_Q2
World phenomena (not shared)
(W1) Vehicle passes through gate
(W2) Vehicle drives in the CCA
(W3) Gate reads license plate
(W4) Monitoring Device detects position of vehicle

Shared phenomena controlled by the world
(SW1) Gate provides TZM with information about detected license plate (timestamp, license plate number)
(SW2) Monitoring Device provides TZM with information about vehicle position (timestamp, geographic coordinates, license plate number)
(SW3) Municipality Officer asks TZM to provide statistics about accesses to CCA

Shared phenomena controlled by the machine
(SM1) TZM displays to Municipality Officer the statistics regarding accesses to the CCA

RASD_Q3

Domain Assumptions:

DA1: Vehicles accessing the CCA are correctly associated with users
DA2: Gates correctly detect the license plate numbers of vehicles going through them
DA3: Monitoring Devices correctly detect the position of vehicles
DA4: Non-resident vehicles that have entered the CCA exit it before the next operating period starts

Notice that DA1 might not be an assumption, but a property that can be checked by the system, if there is, for example, an external system, such as a registrar (think of the PRA in Italy), that can provide information about vehicles owned by people.

Requirements:

R1: TZM shall allow users to register with TZM and provide their personal data (name, address, etc.)
R2: TZM shall allow users to provide (create, modify, delete) data (license plate number, vehicle model, etc.) about the vehicles they own
R3: TZM shall collect, before the end of each day, data from Gates, regarding the information of the vehicles detected by the Gate during that day
R4: TZM shall collect, before the end of each day, data from monitoring devices regarding the position of the corresponding vehicles during that day

R5: At the end of each day, TZM will compute the amount due by each vehicle detected by the Gates during the operating hours of the CCA
R6: At the end of each day, TZM will compute the amount due by each vehicle that has transited through the CCA during its operating hours
R7: At the end of each day, TZM will send an invoice to each User associated with a vehicle detected by the Gates during the operating hours of the CCA
R8: At the end of each day, TZM will send an invoice to each User associated with a vehicle equipped with a Monitoring Device that has transited through the CCA during its operating hours
R9: TZM will allow the External Payment System to retrieve the information of each invoice

DD_Q1
We can imagine that the number of gates is fairly limited (probably in the order of the few tens of elements, depending on the size of the CCA), and it does not change frequently, if at all (the number of streets in the CCA is surely very stable). The number monitoring devices, on the other hand, is very big, and surely changes rather frequently. In addition, given the hardware installed on gates and that of monitoring devices, we can imagine that, from a technological point of view, services offering suitable APIs and accessible by the TZM system can be easily deployed on gates. Monitoring devices, on the other hand, are essentially mobile devices, which can invoke services, but can hardly host them given their computational capabilities and connectivity characteristics and also their usual patterns of usage.
Given the above considerations, we can imagine to adopt, for the retrieval by the TZM system of the vehicle information detected by gates, a "pull" approach, realized through a service hosted by each gate, which provides a suitable API to TZM that offers data retrieval primitives.
The retrieval of data from mobile devices, on the other hand, is better achieved through a "push" mechanism, whereby each monitoring device periodically sends the vehicle position to the TZM system. To increase the flexibility of the communication mechanism (which is a very desirable property for TZM, since the monitoring devices sending data can change quite frequently), we can imagine to adopt an asynchronous communication style between monitoring devices and the TZM system, which can be achieved for example through a message queue or an event-driven middleware like Kafka.

DD_Q2
The macro functions involved in goal G2 are (in addition to the handling of the static information regarding users and vehicles, which is the focus of the already-identified microservices *UserManager* and *VehicleManager*):

- Data collection from gates and monitoring devices
- Computation of fees to be paid by users
- Generation and storage of invoices

Given that, as discussed in point DD_Q1, collection of data from gates and from monitoring devices is better done through different strategies, we can identify 2 separate microservices, *DataCollectionGate* and *DataCollectionMonDevice*. The data collected by these microservices is used to compute the fees to be paid by users. To limit the focus of *DataCollectionGate* and *DataCollectionMonDevice* to the collection of data, we can introduce a separate microservice, *VehicleDataManager*, whose purpose is to receive dynamic data related to vehicles from *DataCollectionGate* and from *DataCollectionMonDevice* and make the data available to the functions related to fee computation and invoice handling.
Finally, we can introduce separate microservices, *FeeGenerator* and *InvoiceManager*, to compute user fees and to generate invoices and make them available to the External Payment System.

DD_Q3
Instances of *DataCollectionGate* essentially act as clients to the services provided by gates. They offer the following operation

```
    void getDataFromGate(Gate g, Time from, Time to)
```
that, when invoked, retrieves the data from gate `g` regarding the vehicles detected between time `from` and time `to`. Additional primitives could be defined, for example to retrieve the last `n` amount of data, or the data of the last `n` hours, and so on.
These operations do not return the collected data, but, rather, send it to the *VehicleDataManager* service to be made available to other services.

Instances of *DataCollectionMonDevice*, instead, receive data from monitoring devices, and provide the following operation
```
    void receivePositionData(Vehicle v, GeoCoordinates pos, Time ts)
```
If the communication between monitoring devices and the *DataCollectionMonDevice* instances is done in an asynchronous manner, `receivePositionData` would actually correspond to a primitive of the chosen framework (e.g., the pushing of an event to a subscriber). Like *DataCollectionGate*, *DataCollectionMonDevice*, after it receives some vehicle data, sends it to the *VehicleDataManager*.

VehicleDataManager offers 2 operations (and possible variations thereof), one to receive vehicle data from other microservices (i.e., *DataCollectionGate* and *DataCollectionMonDevice*), and one to provide the stored data to other microservice (and in particular to *FeeGenerator*):
```
    void receiveVehicleData(Vehicle g, VPos d, Time ts)
```
where `VPos` can be either the vehicle geocoordinates, or the gate at which the vehicle was detected at time `ts`.
```
    VehicleList detectedVehicles(Date d)
    VehicleDataList getVehicleData(Vehicle v, Time from, Time to)
```
where `detectedVehicles` returns the list of vehicles that have been detected in the CCA (either by gates or through the position collected from monitoring devices) during day `d`, and `getVehicleData` returns the data concerning vehicle `v` that has been collected between time `from` and time `to`.

`FeeGenerator` offers the following operations:
```
    void generateInvoices(Date d)
    void generateInvoice(Vehicle v, Date d)
```
which generate, respectively, all invoices related to day `d`, and those concerning vehicle `v` for day `d`. Both operations, after creating the invoice(s), send it to *InvoiceManager* to make it available to interested parties.

Finally, *InvoiceManager* provides the following operations
```
    Invoice getInvoice(User u, Date d)
    Void receiveInvoice(Invoice i)
```
where `getInvoice` returns the invoice concerning user `u` related to day `d`, whereas `receiveInvoice` is used to store a new invoice in the *InvoiceManager*.