Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano

Prof. 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 II

February 14th, 2020

Last Name
First Name
Id number (Matricola)

Note

1. The exam is not valid if you do not fill in the above data.
2. Write your answers on these pages. Extra sheets will be ignored. You may use a pencil.
3. Incomprehensible hand-writing is equivalent to not providing an answer.
4. The use of any electronic apparatus (computer, cell phone, camera, etc.) is strictly forbidden.
5. You cannot keep a copy of the exam when you leave the room.
6. The exam is composed of three exercises. Read carefully all points in the text!
7. **Total available time: 2h**

Scores of each question:

Question 1	(MAX 7)	_________
Question 2	(MAX 6)	_________
Question 3	(MAX 3)	_________

Question 1 Alloy (7 points)

Consider *TSPmonitor*, a crowd-based application that allows users to report on the status of Public Transport in the city. In particular, the application should allow citizens to provide short reports on the status of the bus, tram, and metro lines they take. The reports created through the application include the level of crowdedness of the line, the comfort of the trip, and the punctuality of the service. Reports created through the application are stored for statistics computation and data mining purposes. In addition, if the user allows it, reports are automatically transformed into tweets that make use of predefined hashtags.
The application can also notify users who request it about changes in the detected status of lines (e.g., when a line becomes crowded), and it can suggest alternative paths when problems strike their preferred lines. Also, thanks to its mining capabilities, the application can provide forecasts on the status of lines in the next 3 days.
Consider the following (simplified) domain elements and world and machine phenomena:

- Actual circulating trains, where each train belongs to a line and has an actual crowdedness level and a detected crowdedness level (where the crowdedness level, actual or detected, can be "sparse", "normal", "crowded", "overflowing").
- Travelers, who take trains.
- Users of *TSPmonitor*, who have a list of preferred train lines, which they regularly take.
- Users create reports on the crowdedness level of trains. For simplicity, we consider that each user produces at most one report per train.
- Users receive notifications from the system, where each notification concerns only trains of their preferred lines, and it contains the detected crowdedness level of the train. For simplicity, we consider that each user receives only one notification per train.

A. Define suitable signatures (and constraints thereof) that capture the elements and phenomena listed above.
B. Formalize through a predicate the following goal: users are informed about the actual crowdedness level of the trains of their preferred lines.
C. Formalize through a predicate the following domain assumption: users report the actual level of crowdedness of the train they are on.
D. Formalize through a predicate the following requirement: if all reporting users agree on the crowdedness level of a train, the detected crowdedness level of that train is defined and it corresponds to the one reported by the users.

Solution

```
sig Line {}

sig Train {
  status: Crowdedness,
  detectedStatus: lone Crowdedness,
  line: Line,
  passengers: set Traveler
}

abstract sig Crowdedness {}
one sig Sparse extends Crowdedness {}
one sig Normal extends Crowdedness {}
one sig Crowded extends Crowdedness {}
one sig Overflowing extends Crowdedness {}
```

```
sig Report {
  train: Train,
  status: Crowdedness,
}

sig Notification {
  train: Train,
  status: Crowdedness,
}

sig Traveler {}
sig User extends Traveler {
  preferred: set Line,
  myReports: set Report,
  myNotifications: set Notification,
}{ myNotifications.train.line in preferred
   all disj n1, n2: myNotifications | n1.train != n2.train
   all disj r1, r2: myReports | r1.train != r2.train }

pred Goal {
  all u: User, t: Train | t.line in u.preferred implies
                          ( one n:u.myNotifications | n.status = t.status )
}

pred Assumption {
  all u: User, r: u.myReports | r.train.status = r.status and u in r.train.passengers
}

pred Requirement {
  all t: Train, s: Crowdedness |
      t.detectedStatus = s iff
           ( no disj u1, u2: User | some r1, r2: Report | r1 in u1.myReports and
                                                         r2 in u2.myReports and
                                                         r1.train = t and r2.train = t and
                                                         r1.status != r2.status )
           and
           ( some u: User, r: Report | r in u.myReports and r.train = t and r.status = s )
}

// Alternative formalization for User, goal, assumption, requirement
// Notice that in this case signatures Report and Notification are no longer needed
sig User extends Traveler {
  preferred: set Line,
  myReports: Train -> lone Crowdedness,
  myNotifications: Train -> lone Crowdedness,
}{ (myNotifications. Crowdedness).line in preferred }

pred Goal {
  all u: User, t: Train | t.line in u.preferred implies t.(u.myNotifications) = t.status
}

pred Assumption {
  all u: User, t: (u.myReports).Crowdedness | t.(u.myReports) = t.status and
```

```
                                                   u in t.passengers
}

pred Requirement {
  all t: Train, s: Crowdedness |
      t.detectedStatus = s iff
           ( no disj u1, u2: User | #t.(u1.myReports) = #t.(u2.myReports)
                                    and
                                    t.(u1.myReports) != t.(u2.myReports) )
           and
           ( some u: User | t.(u.myReports) = s )
}
```

Question 2 JEE (6 points)
Alphabet is a website that allows users to choose from a list of freelancers for a *Translation Service*. Freelancers offer different types of *Translation Services*, including technical translation, legal, medical and financial translation in various languages.
We focus on the functions offered to clients. The system allows clients to:

1. Register.
2. Log in.
3. View the list of available freelancers in a selected date, categorized based on the type of the *Translation Service* offered.
4. View each freelancer's profile that displays his/her reputation (total completed *Translation Services*, and a rating of 1 to 10 stars given by her/his previous clients), and cost of the *Translation Service* per 1000 words (which is not negotiable).
5. Select the desired freelancer for a specific translation package offered by her/him.
   When a client selects a freelancer, she/he specifies also the length of the manuscript to be translated and the anticipated delivery date; this initiates a new *Translation Service*, which is thus created and its status is "initiated". The freelancer then has up to three days to respond to such proposal, by either accepting or rejecting it. In the case of approval of the proposal, the status of the *Translation Service* is updated to "activated" and the client must send the manuscript within three days (if the client fails to send the manuscript within three days, the status of the *Translation Service* turns to "cancelled"). After the proposal is accepted and the manuscript is delivered, the *Translation Service* automatically starts and the countdown to the delivery date begins. If the freelancer rejects the proposal, then the *Translation Service* automatically terminates, and its status is updated to "rejected".
6. Cancel any proposed and ongoing *Translation Services* (in the latter case, any compensation of the freelancers would be paid according to Alphabet's agreed cancellation terms and conditions).
7. View the list of all her/his *Translation Services* and their status (initiated, rejected, activated, cancelled, countdown to the delivery date, completed).
8. Give a score to the freelancer who has completed a *Translation Service*.

A. Which JEE API is commonly used for the development and management of the data layer?

Using the following table, identify the main data **entities**, their main **attributes** (including their types and corresponding annotations, if any) and their **relationships** to each other using the features of such API, and specify the **multiplicity of the relationship**.

Entity	Attributes	Relationship With	Multiplicity

B. Give two examples of possible **bidirectional relations** among those identified in part A). Justify your choice by explaining the assumptions behind it and the purpose of making the relations bidirectional.

C. Identify the required Session Bean Entities, specify their types, and for each Bean list the requirements that it satisfies.

Entity	Type	Requirements

D. Referring to the identified types of beans, explain the motivations for your choices.

Solution

Part A

The API suitable for the data layer is Java Persistence API

Entity	Attributes	Relationship With	Multiplicity
Client	clientID, String, @Id, @GeneratedValue name, String, @NotNull password, String, @NotNull ------------------------------------ translationService, Collection/Set<TranslationService>, @OneToMany	Translation-Service	@One -to- Many
Freelancer	freelancerID, String, @Id, @GeneratedValue name, String, @NotNull rate, Integer Total_Completed_Translation_Service, Integer ------------------------------------ translationService, Collection/Set<TranslationService>, @OneToMany	Translation-Service	@One -to- Many
Translation-Service	translation_ServiceID, String, @Id, @GeneratedValue type, String/Enumerated (i.e., TextType + @Enumerated(EnumType.STRING) cost, Integer proposed_Date, Date acceptedDate_Date start _Date, Date delivery_Date,Date status, Enumerated (i.e., Status + @Enumerated(EnumType.STRING) ------------------------------------ client, Client, @ManyToOne freelancer, Freelancer, @ManyToOne	Client Freelancer	@Many -to- One @ Many -to- One

** It is also possible to model the Reputation as an Entity (with Rate and Total_Completed_Translation_Service as its attributes), that it is in One-To-One relation with a Freelancer

Part B

The relation between Client Entity and TranslationService Entity is Bidirectional because the system needs to retrieve all the instances of TranslationService entity associated with a given instance of the client (to address function 7), as well as retrieve the particular instance of client associated with a given instance of TranslationService (to generate the bill and to let the client score the freelancer as described in functions 4,8)

The relation between Freelancer Entity and TranslationService Entity could be Bidirectional as well. The direction from a TranslationService instance to a Freelancer instance is necessary to record a particular TranslationService information, to track Freelancer to deliver the order and to update the availability of freelancers. In addition, the direction Freelancer to TranslationService is needed to retrieve all accepted TranslationServices for by each Freelancer to be shown in their profile and to compute the reputation of that freelancer.

Part C

Entity	Type	Requirements
ClientManager Bean	Stateless	1, 2, 7
FreelancerManager Bean	Stateless	3,4,8
TranslationServiceManager Bean	Either Message Driven or Stateless	5,6

Part D

None of beans need to maintain the conversational state of the clients, so accordingly we can model all of them as Stateless beans. In other words, a specific "state of the client" is not required for invocations of any methods of the beans. Indeed, each method is not client-specific, it performs a generic tasks for any client, for example to register a new user, to generate a new translation service, or to update the status of a translation service (the information which is required to perform such tasks is either persisted-and so fetched from the database-or it is included in the stateless communication between user and system).

However, it is also possible to define the TranslationServiceManager Bean as message driven. This could be done if one decides to design the system as a message-based architecture where method invocations/interactions among components (for instance, among clients and freelancers) are in term of message exchanges. For example, when the user selects a translation service, the system will notify the corresponding freelancer; similarly, whenever the status of a translation service changes, the system notifies the corresponding user.

Question 3: Project management (3 points)

The `HR` (Human Resources) application is in charge of managing information regarding employees of a company. The managed pieces of information concern user data, payslips and absences. Each absence must be classified based on the following types: illness, work permit, holiday.

Payslips are computed by an external application, `Payroll`, that is connected to the `HR` application. The pieces of information managed by `Payroll` are:

- Salaries
- Taxes
- Benefits
- Working days
- Payslips

Payslips are accessible from HR. Moreover, HR allows employees to:

1. Login / Logout
2. Retrieve payslips for each month/year
3. Insert / delete absence records. On insertion, the correct absence type must be included.

Calculate the function points for the HR application. Motivate your choices.

Refer to the following table to associate weights to the function types:

Function types	**Weights**		
	Simple	Medium	Complex
External Inputs	3	4	6
External Outputs	4	5	7
External Inquiry	3	4	6
Internal Logic Files	7	10	15
External Interface Files	5	7	10

Solution:
1 EIF (complex) for the payslips (10 fps)
3 ILF (simple) for users, absence, absence type (3*7=21fps)
2 External inputs (simple) for login / logout (2x3=6 fps)
1 External inquiry for the payslip retrieval (simple, 2 files accessed users, payslips) 3fps
1 External input for the insert absence (medium, 3 files accessed users, absence, absence type) 4fps
1 External input for the delete absence (simple, 2 files accessed users, absence) 3fps
FPS = 47

Notes: since we are focusing on HR, the Payroll pieces of data that are not used by HR are not to be considered in the function points computation. So, there is only one ELF from the HR perspective, that is, payslips. Insert absence is more complex than delete because it involves the usage of absence type.