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

January 11th, 2019

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 a load balancing component of a system, which creates and destroys instances of certain servers depending on the number of requests that are currently being handled by the system.
Requests can be of various different types. For simplicity, let us say that there are three request types, `Request1`, `Request2`, and `Request3`, corresponding to the different services that can be handled by the servers.
When instantiating a server, the load balancer can configure it so that it is able to handle some request types.
In total, each server can handle, at the same time, at most 10 requests of any type.

1. Define the Alloy signatures that are necessary to model servers and the requests that they handle.

2. Using the signatures defined, formalize, through Alloy predicates, **two** of the following operations (we will grade only the first two operations on your exam):

- `newRequest`: given the set of currently available servers and a new request, under the precondition that there is, among the available servers, one that is handling at most 9 requests and that can handle the type of the new request, the operation allocates the request to one such server.
- `newServer`: given the set of currently available servers and a new request, under the precondition that there is no available server that is handling at most 9 requests and can handle the type of the new request, the operation creates the instance of a new server that can handle the desired request, allocates the request to it and adds this new server to the set of available servers.
- `completeRequest`: given the set of currently available servers and a request that is currently being handled by one of the servers, the operation removes the request from the server.
- `cleanServers`: given the set of currently available servers, the operation removes from it all servers that are not handling any request.

Solution
A possible solution for the exercise is the following. Other ones are also possible, of course.

```
abstract sig RequestType {}
one sig RequestType1 extends RequestType {}
one sig RequestType2 extends RequestType {}
one sig RequestType3 extends RequestType {}

sig Request {
  type : one RequestType
}

sig Server {
  services : some RequestType,
  requests : set Request
}{
#requests =< 10
requests.type in services
}

sig AvailableServers{
  servers : set Server
}{ all disj s,s':servers | s.requests & s'.requests = none }
```

```
pred newRequest[ ss : AvailableServers, r : Request, ss': AvailableServers ]{
  ( not(r in ss.servers.requests) and
    some disj s, s' : Server | s in ss.servers and #s.requests =< 9
                          and r.type in s.services
                          and s'.services = s.services
                          and s'.requests = s.requests + r
                          and ss'.servers = ss.servers - s + s')
}

pred newServer[ ss : AvailableServers, r : Request, ss': AvailableServers ]{
  ( not(r in ss.servers.requests) and
    no s : Server | s in ss.servers and #s.requests =< 9 and r.type in s.services )
    and some s': Server | r.type in s'.services and
                        r in s'.requests and
                        ss'.servers = ss.servers + s’
}

pred completeRequest[ ss : AvailableServers, r : Request, ss': AvailableServers ]{
  ( one s : Server | s in ss.servers and r in s.requests
                   and some s': Server | s'.services = s.services
                                       and s'.requests = s.requests - r
                                       and ss'.servers = ss.servers  -  s + s')
}

pred cleanServers[ ss : AvailableServers, ss': AvailableServers ]{
  all s : Server | s in ss'.servers iff s in ss.servers and #s.requests > 0
}
```

Question 2 JEE (6 points)

A company offers a bike rental service. Bikes are distributed in the city based on where they have been left by the previous user. The company wants to develop a software system to allow users to rent and ride bikes. Rental fees are computed according to the usage time. In particular, through the system, a user should be able to:

1) Register.
2) Log in/log out.
3) Display the list of available bikes organized in different categories (e.g., mountain bike, city bike, racing bike) and their rental price.
4) Select a bike either from the list of available bikes (see point 3) or by specifying the bike ID (displayed on the bike).
5) Make a reservation for a bike. In response to this operation, the user receives a OTP (One-Time Password) to be used to unlock the bike. A reservation should last 20 minutes after which, if the user has not yet started the rental, it is automatically cancelled. The user should be able to see a countdown associated with his/her active reservation even if he/she logs out and then in again.
6) Start the rental by inserting the OTP (One-Time Password) received at reservation time on the bike numeric keypad.
7) Cancel the reservation.

8) Terminate the rental.
9) Access the list of all his/her rentals.
10) For each rental, get the total covered distance, the estimated calories spent and the saved CO_2.

You are asked to design the software system using JEE. In particular:

A. You should identify the required JEE entity/session beans that you wish to have in order to implement the system. For each bean you should specify the type (e.g., stateful session bean) and the requirements it is intended to fulfill (among those listed above). To ease our assessment of your solution, please use the table below.
B. Referring to the identification of stateful and stateless session beans, explain the motivations for your choices.
C. Specify **at least 2 methods** for each bean choosing some significant functionality of the bean. In particular, provide the signature of the method, a description of what it does, and list the JEE features the bean and/or the method exploits.

Bean name	**Bean type**	**Requirements it contributes to fulfill**

Solution

Point A

Bean name	**Bean type**	**Requirements it contributes to fulfill**
`UserEntity`	JPA entity	1, 2, 7, 8
`BikeEntity`	JPA entity	3, 4, 5, 6, 7, 8
`RentalEntity`	JPA entity	6, 8, 9, 10
`UserBean`	Stateless Session Bean	1, 2, 5, 7, 8, 9

`RentalBean`	Stateless Session Bean	6, 8, 10
`BikeManagementBean`	Stateless Session Bean	3, 4

Point B
All beans are defined as stateless because there is no need to maintain a temporary state for the duration of a session. The reservation time, which allows us to implement the countdown, in fact, should be available to clients also when the user logs out and then in again in the system (maybe even from a different device). This means that this information should last longer than the single session. We choose then to store this data as part of the rental entity and to offer the countdown operation through the UserBean (see below).

Point C
`UserEntity`:
This entity has various attributes, among which the most important are: user name, password and rental status (which is a Boolean value representing whether the bike is reserved/rented or not). The user name is also the primary key for a user and thus the corresponding attribute is annotated with the `@Id` JPA annotation.
Methods:
- `String getUsername()/void setUsername(String name)` : getter and setter for the user name.
- `void setPassword(String password)` : setter for the password of the user.
- `Boolean checkPassword(String password)` : checks that the password passed as parameter corresponds to the one stored in the entity. Note that this approach is better than offering a getter method for the password as it allows the entity to protect this piece of confidential data, in line with the encapsulation principle.
- `Boolean getRentalStatus()/void setRentalStatus (Boolean rentalStatus)` : getter and setter for the rental status of the user. We assume that this is true from the time the user reserves a bike to the time he/she releases it after rental, or the time in which the reservation expires.
- Other getters and setters for other personal information, e.g., payment information.

`RentalEntity`:
This entity has the following attributes:
- Id of the rental, which we assume to be automatically generated by the JPA using the `@GeneratedValue` annotation.
- The user who rents the bike. The relation between a rental and its user is specified using the `@ManyToOne` annotation of the JPA.
- The bike that has been rented. The relation between a rental and the corresponding bike is specified using the `@ManyToOne` annotation of the JPA.
- The reservation start time. We do not need to store the end time of the reservation because there will be a heartbeat message which will check, with a given time step, that the reservation countdown has not expired.
- The rental start and end time.
- The rental status. It includes also information about the reservation if the rental has not started yet.
- The OTP associated to this rental.
- Additional information about the journey (covered distance, spent calories and saved CO_2). These fields are not null only if the rental actually started.

Methods:
- `Integer getId()`: getter for the id (setter is not needed as the id is auto-generated).
- `UserEntity getUser()/void setUser(UserEntity user)` : getter and setter for the user who rented a bike.

- `BikeEntity getBike()/void setBike(BikeEntity bike)` : getter and setter for the bike with which the rental is associated.
- `Timestamp getReservationStartTime ()/void setReservationStartTime (Timestamp timestamp)` : getter and setter for the reservation starting time.
- `Timestamp getRentalStartTime ()/void setRentalStartTime (Timestamp timestamp)` : getter and setter for the rental starting time.
- `Timestamp getRentalEndTime()/void setRentalEndTime (Timestamp timestamp)` : getter and setter for the rental ending time.
- `Integer getRentalStatus()/void setRentalStatus (Integer status)` : getter and setter for the status of the rental.
- `Integer getOTP()/void setOTP(Integer generatedOTP)` : getter and setter for the OTP generated after the reservation of the bike.
- `String getRidingInfo()/void setRidingInfo (String ridingInfo)` : getter and setter for the additional information about the journey.

`BikeEntity`:
This entity has the following attributes:
- Id of the bike, which we assume to be automatically generated by the JPA using the `@GeneratedValue` annotation.
- Bike model.
- Rental price.
- Boolean that indicates whether the bike is available for renting or not.
- Current rental entity associated with the bike. The relation between a rental and its bike is specified using the `@OneToMany` annotation of the JPA.

Methods:
- `Integer getId()`: getter for the id of the bike (setter is not needed at it is auto-generated.
- `String getModel()/void setModel(String model)` : getter and setter for the model of the bike.
- `Double getRentalPrice()/void setRentalPrice(Double rentalPrice)` : getter and setter for the rental price of the bike.
- `Boolean isAvailable()/void setAvailability(Boolean availability)` : getter and setter for the availability state of the bike.
- `RentalEntity getCurrentRental()/void setCurrentRental(RentalEntity currentRental)` : getter and setter for the current rental of the bike.

Methods provided by `UserBean`:
- `Integer registerNewUser(String userName, String password)`. It creates a new `UserEntity` and uses an `EntityManager` to persist the new user into the database. It returns the user Id. If a user with the same username already exists it throws an exception.
- `Boolean loginUser(String userName, String password)`. It queries the database (using an `EntityManager`) looking for a user with the specified username. If this exists, it compares the corresponding password with the one received as argument. If they match, the method returns true. In any other case it returns false.
- `Boolean existsUser(String userName)`. It queries the database (using an `EntityManager`) looking for a user with the specified username and it returns true if this exists, false otherwise.
- `User getUserById(Integer userId)`. It queries the database (using an `EntityManager`) looking for a user with the specified id and it returns the `User` data structure if the corresponding entity exists, null otherwise. The `User` data structure contains all field of `UserEntity` excluding the sensible ones, the password in this case.

- `List<Rental> getRentals (Integer userId)`. It queries the database (using an `EntityManager`) for the list of all the `RentalEntity` associated with the user. It returns the `Rental` data structure that contains all fields of `RentalEntity` excluding the sensible ones.
- `void cancelReservation (Integer rentalId)`. It cancels a reservation by setting the proper fields in the `RentalEntity`.
- `void endRental (Integer rentalId)`. It allows the user to end the rental by setting the proper fields in the `RentalEntity`.
- `Integer makeReservation(Integer userId, Integer bikeId)`. If the bike corresponding to `bikeId` is available, it creates a new `RentalEntity` by querying the database (using an `EntityManager`) and setting the start rental timestamp associated with the rental. It also generates a OTP using the method below and updates the corresponding field in the `RentalEntity` by using the `setOTP` method. Finally, it updates in the `BikeEntity` the current rental, using the `setCurrentRental` method. This way, the user can log out and then in again without losing the generated OTP. OTP is finally returned to the caller. If `bike` is not available, this method throws an exception.
- `Integer generateOTP()`. It generates a OTP.
- `Timestamp checkCountdown(Integer rentalId)`. Given the `rentalId`, it queries the database to retrieve `RentalEntity` so that the start time of the reservation can be recovered. This method is periodically called to synchronize the remote countdown with the true one based on the start time of the reservation and the current time. If the time elapsed and no rental has started, then the reservation is canceled and the method returns -1. If a rental has started already, then this method throws an exception.

Methods provided by `BikeManagementBean`:
- `Bike getBikeById(Integer id)`. It queries the database (using an `EntityManager`) looking for a bike with the specified id and it returns the corresponding `Bike` data structure if the corresponding entity exists, null otherwise. `Bike` includes all fields of `BikeEntity` excluding the sensible ones.
- `List<Bike> getAvailableBikes()`. It queries the database (using an `EntityManager`) for all the available bikes and it returns the matching list of `Bike` if at least one exists, null otherwise.
- `List<Bike> getBikeByModel(String model)`. It queries the database (using an `EntityManager`) looking for a bike of the specified category and it returns the matching list of `Bike` if at least one exists, null otherwise.

Methods provided by `RentalBean`:
We assume that there could be no more than one bike reservation/rental per user at a time, so the pair user/bike is sufficient to retrieve the corresponding rental and to handle rental start and end. In particular, the , but just the bike, and the `RentalBean` will check for the rental currently under definition retrieving the proper `BikeEntity` field by means of `getCurrentRental` method. Different implementations could still be acceptable:
- `void startRental(Integer bikeId, Integer UserID, Integer OTP)`. Through the `bikeId`, it retrieves the `BikeEntity` and then the current reservation (`RentalEntity`) for the bike using the `getCurrentRental` method. At this point, it checks whether the OTP parameter corresponds to the one in the entity, if the time elapsed from the reservation is less than 20 mins and if the reservation is correctly associated to the user. If yes, it sets the start rental timestamp associated with the rental and changes the rental status.
- `String computeStatistics(Integer rentalId)`. It computes useful statistics (covered distance, spent calories and saved CO_2) and returns structured text.

- `void endRental(Integer bikeId, Timestamp endRental).` It queries the database (using an `EntityManager`) and sets the end rental timestamp associated with the rental.

Question 3: Design (3 points)

Consider the load balancing component of Question 1. Suppose that the load balancer has an availability of 98%, and that each server instance has an availability of 97%, independent of the services it offers.
The desired total availability of the overall system (load balancer plus instantiated servers) when handling a request is 99%.
Discuss possible strategies to obtain the desired availability.

Solution

The system that handles each request is made of the load balancer in series with the server(s) actually handling the requests.
Given that the availability of the load balancer is itself lower than the target, a single instance of the load balancer is not enough to achieve the target.
Hence, we need at least 2 replicas of the load balancer, which together have an availability of 1 - (1-0.98)*(1-0.98) = 99.96%.
Then, if each request is handled by 2 replicas of a server (i.e., the load balancer instantiates 2 replicas of each server), the availability of the servers handling each request is 1 - (1-0.97)*(1-0.97) = 99.91%, and the availability of the whole system (the series of the load balancers and of the servers) is 0.9991*0.9996=99.87%.
Of course, if the used servers are able to handle only a subset of the foreseen request types, the deployed solution should have at least two replicas for each request type.