**Question 1 Alloy (7 points)**

Consider the following Alloy signatures:

```
abstract sig Boolean {}
one sig True extends Boolean {}
one sig False extends Boolean {}

sig TCPData {
  processed: one Boolean
}

sig TCPServer {
  received: set TCPData
}
```

`TCPData` represents information being transferred through a TCP network and `TCPServer` represents servers able to receive such data. When `processed` is true, this means that the corresponding `TCPData` has been processed by a server. Of course, this happens only if the server receives such data.
Assume to observe the final state of the system composed of some data and some servers. In this final state, all data have been received by the corresponding servers (there are no data at the sources nor in transit) and, possibly, processed. Under this assumption, extend the above Alloy model as follows:

a) Model a TCP channel that maps `TCPData` to `TCPServers` through a routing relation. For the sake of simplicity, focus only on the destination of the routing, not on the source and make sure that the channel can route each `TCPData` to at most one `TCPServer`. Keep in mind that you are representing the situation you observe in the final state, so, at the end of the routing activity. This means that the routing relation indicates to which server a channel has routed a certain piece of data.
b) Model as a predicate the following domain assumption (which is enforced thanks to the guarantees offered by the TCP protocol): *data routed toward servers have been received by them*.
c) Model as a predicate the following requirement: *all data received by servers have been processed.*
d) Model as a predicate the following goal: *data routed toward servers should have been processed*.
e) Define the following two assertions:
    a. `GoalFulfillment` that aims at checking whether the goal is fulfilled when the requirement and the assumption are true.
    b. `GoalNeedsAssumptions` to check the following: even if the requirement holds, the goal cannot be satisfied if the assumption does not hold.

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

```
sig TCPChannel {
  routed: TCPData -> lone TCPServer
}

pred TCPChannelAssumption {
  all d: TCPData, s: TCPServer |
          d in TCPChannel.routed.s implies d in s.received
}

pred TCPServerRequirement {
  all d: TCPData, s: TCPServer |
          d in s.received implies d.processed = True
}
```

```
pred Goal {
  all d: TCPData |
     d in (TCPChannel.routed).TCPServer implies d.processed = True
}

assert GoalFulfillment {
  TCPServerRequirement and TCPChannelAssumption implies Goal
}

assert GoalNeedsAssumption {
  TCPServerRequirement implies
    (not TCPChannelAssumption implies not Goal))
}
```

**Question 2 Testing (6 points)**

Consider the following C program (lines are numbered for your convenience, please refer to these numbers in your solution):

```
1  main() {
2    int a, h, f, q;

3    scanf("%d", &a);
4    scanf("%d", &q);
5    h = q - 2;
6    while (a > 0){
7      if (q == h + 2)
8        f = a;
9      else if (a > f)
10        f = a;
11     scanf("%d", &a);
12     h = h+1;
13   }
14   printf(%d", f);
15 }
```

1. Draw the control flow graph of the program for def-use analysis (highlight in the graph where variables are defined and used) and write the def-use pairs in the table below.
Def-use analysis shows that there some potential problems in the program; explain what these problems are.

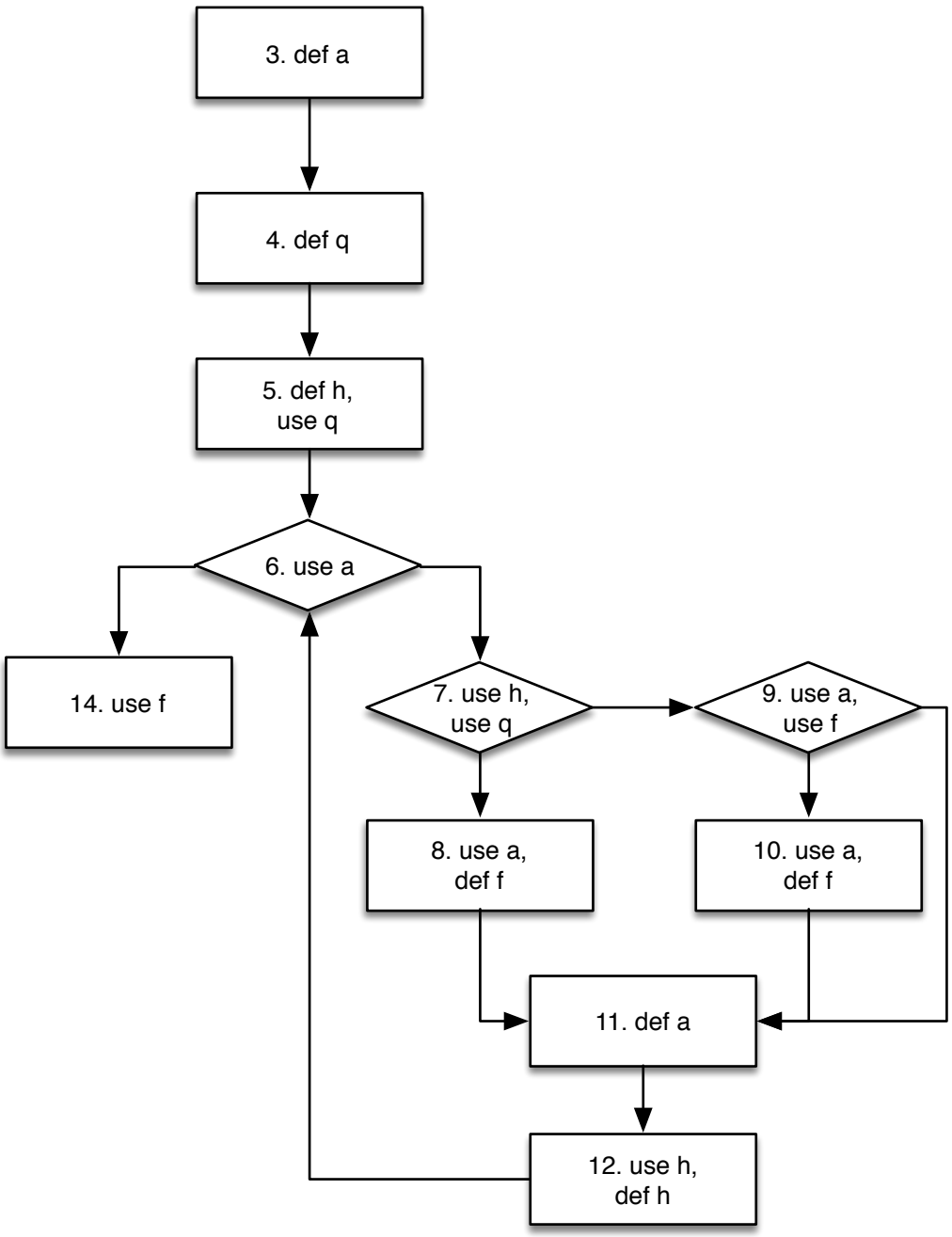
2. Use symbolic execution to show whether the problems highlighted by def-use analysis can occur or not during execution of the program.

| **variable** | **⟨ def, use ⟩ pairs** |
|---|---|
| | |
| | |

| | |
|---|---|
| | |
| | |

**Solution**

1.
The control flow graph of the program for def-use analysis is the following:




| **variable** | **⟨ def, use ⟩ pairs** |
|---|---|

| | |
|---|---|
| a | ⟨3, 6⟩, ⟨3, 8⟩, ⟨3, 9⟩, ⟨3, 10⟩, ⟨11, 6⟩, ⟨11, 8⟩, ⟨11, 9⟩, ⟨11, 10⟩ |
| h | ⟨5, 7⟩, ⟨5, 12⟩, ⟨12, 7⟩, ⟨12, 12⟩ |
| f | ⟨??, 9⟩, ⟨??, 14⟩, ⟨8, 9⟩, ⟨8, 14⟩, ⟨10, 9⟩, ⟨10, 14⟩ |
| q | ⟨4, 5⟩, ⟨4, 7⟩ |

Def-analysis highlights 2 potential uses of variable `f` (at lines 9 and 14) before it is initialized.

2.
The paths leading to the potential uses of `f` before its initialization are the following: 3, 4, 5, 6, 14 and 3, 4, 5, 6, 7, 9.
Symbolic execution through path 3, 4, 5, 6, 14 gives the following result:
3: a = A
4: q = Q
5: h =Q-2
6: $A \leq 0$
Hence, the path is indeed feasible with condition $A \leq 0$, so this is a real potential problem that can occur in the program.

Path 3, 4, 5, 6, 7, 9, instead, is not feasible. In fact, symbolic execution gives the following result:
3: a = A
4: q = Q
5: h =Q-2
6: $A > 0$
7: $Q \neq Q-2+2$
which yields the contradictory condition $Q \neq Q$. Hence, this is a false positive that def-use analysis produces.

**Question 3, Alternative a: Project management (3 points)**

A project has been set up to build an application composed of three software components.
The budget assigned to the project is 10000 euros
At a planned review the project manager (PM) calculates the following parameters according to the Earned Value Analysis:

- Earned Value (EV) is 800 euros.
- Planned Value (PV) is 1500 euros.
- Actual Cost (AC) is 1000 euros.

As PM of the project please answer to the following questions:

1. Inform the stakeholders about the situation of the project in terms of schedule and cost.
2. Estimate the budget at completion (EAC) of the project considering the following two options:
    a. The project will progress spending at the actual rate.

b. The project will progress spending at the original rate.

Please provide a short motivation to your answers.

**Solution**

1.
The project is running over budget because the value produced is less than the cost (EV < AC); in addition, it is running behind schedule because the value produced is less than the value planned (EV < PV).

2.
The cost performance index of the running project is CPI = EV/AC = 0.8

a. using the actual rate EAC = BAC/CPI = 10000/0.8 = 12.500 euros
b. continuing at the original rate EAC = AC + (BAC – EV) = 1000 + (10000-800) 10200 euros

**Question 3, Alternative b: Project management (1.5 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) Log in.
2) Display the list of available bikes organized in different categories (e.g., mountain bike, city bike, racing bike) and their rental price.
3) Make a reservation for a bike from the list and obtain back a OTP (One-Time Password) to be used to unlock the bike.
4) Start the rental by inserting the OTP (One-Time Password) received at reservation time on the bike numeric keypad.
5) Terminate the rental.

Referring to the Function Points analysis, how would you classify function type number 3 (make reservation)? Which complexity would you assign to it? Provide a short motivation for both answers.

**Solution**

Make a reservation can be classified as an external input. The OTP may be sent back to the user as a reply to the make reservation request or through a different mechanism, for instance, by sending an SMS or an email message to the user. In the second case, it should be accounted as a different function type offered by the system, in particular, as an external output. Both function types can be considered of simple complexity. In particular, make reservation requires the interaction with two related entities in the database, a reservation and a bike. Sending the OTP requires the interaction with an external email or SMS delivery service. In the case the OTP is returned as a result of make reservation, we could assign to this function type a medium complexity to account for the OTP generation.

**Questions 3, 4 and 5, Planning (5 points), Design (5 points) and Testing (5 points)**

Design a simple project management tool for a company. The application should:

3. Handle information about projects (name, purpose, budget, tasks) and people (name, skills, employee number, role, department, allocated projects, allocated tasks).
4. Enable people to log-in/out.
5. Allow people with role "project manager" to allocate other people to projects and tasks.
6. Allow people to visualize the tasks they are working on and input the level of accomplishment of these tasks.
7. Compute the set of people allocated to a specific project/task.
8. Support people in requesting help across the company based on skills (the user selects a set of skills and the system suggests people with these skills).

**A)** Estimate the size of the project defined above, using Function Points and assuming that the application is going to be developed in JAVA (1 FP = 53 SLOC). To apply Function Points you can use the table below:

| **Function Types** | **Weights** | | |
|---|---|---|---|
| | Simple | Medium | Complex |
| N. Inputs | 3 | 4 | 6 |
| N.Outputs | 4 | 5 | 7 |
| N. Inquiry | 3 | 4 | 6 |
| N. ILF | 7 | 10 | 15 |
| N. ELF | 5 | 7 | 10 |

**B)** Define:
- The Actors involved in the application
- The Use Cases describing the main functionality of the application: it is not necessary to detail all the use cases. Rather, draw them in one or more use case diagrams and provide the details of the most important one.
- A diagram highlighting the key components of the system architecture and the way they are connected. Explain the role of each component.

**C) Identify possible acceptance test cases** for the system focusing on functional aspects. While defining these cases specify the preconditions (if any) that should be true before executing the tests, the inputs provided during the test, and the expected outputs. Finally, explain why you think the selected tests are important for the specific system being considered.

**Solution (Part A)**

Our problem is the following:

*Design a simple project management tool for a company. The application should:*

1. *Handle information about projects (name, purpose, budget, tasks) and people (name, skills, employee number, role, department, allocated projects, allocated tasks).*
2. *Enable people to log-in/out.*
3. *Allow people with role "project manager" to allocate other people to projects and tasks.*
4. *Allow people to visualize the tasks they are working on and input the level of accomplishment of these tasks.*
5. *Compute the set of people allocated to a specific project/task.*
6. *Support people in requesting help across the company based on skills (the user selects a set of skills and the system suggests people with these skills).*

ILFs: (people, projects, tasks and skills. We can assume that all have simple structure as they are basic data structure containing a relatively small number of instances) 4*7 = **28**
ELFs: (we assume that this software is a standalone one. If we had assumed that this was connected, for instance, with the human resources software for managing exployee, we would have include here at least 1 ELF) **0**
EIs:

- Login – simple
- Logout – simple
- Project creation – simple
- Task creation and allocation to project – medium as it requires the usage of 2 entities
- Input level of accomplishment of a task – simple
- People assignment to task – medium as it requires the usage of at least 2 entities

3*4 + 4*2 = **20**
EIQs:

- Help request – medium as it needs to analyse a potentially large number of people from the whole company
- Get the set of people allocated to a specific project/task - simple
- Visualize user's tasks - simple

1*4+2*3 = **10**
EOs: 0 (application does not allow output)
TOTAL: **28+20+10 = 58**

**Assuming 53 LOC per each FP we have 58*53= 3074 SLOC**

**Solution (Part B)**

B.1)
Actors Involved are:

Project Manager (PM), responsible for project and task artifacts with all connected relations
Person/Developer, responsible for his/her own tasks and their progress status
Helper (anyone), responsible for providing help based on owned skills/expertise

B.2)
Use-Cases: we have one per each of the EI or EQ identified in the FP analysis

- Login
- Logout
- Create project

- Create task and allocate it to the project
- Input level of accomplishment of a task
- Assign people to task
- Request help
- Get the set of people allocated to a specific project/task
- Visualize user's tasks

The most important use case is probably "Assign people to task" as it requires specific attention not to overload people. So, during the assignment, the system will have to double check that the selected person can actually be allocated to the selected task given the availability of the person and the foreseen effort associated to the task. The description does not say anything about what the system should do in case the assignment is not possible. The option we choose to adopt in this solution is that if a task t requires x hours effort and person p can be available to the project for y hours with y<x in the timeframe of the task, then the person is allocated for y hours and the system informs the project manager that he/she needs to find another person to complete the allocation.

**Use case name**: Assign people to task
**Participating actors**: project manager, developer
**Entry condition**: the project manager has created a project and at least a task in the project
**Flow of events**

1. The project manager selects the project and the task to be allocated
2. The project manager selects the software engineer to be allocated to the task
3. The system performs the assignment and evaluates its compatibility. If the software engineer is available for less time than needed, the system informs the manager that the allocation to the task should be completed and the use case continues from point 2. Otherwise the system informs the manager than the allocation has been finalized.

**Exit condition**
The software engineer is allocated to the task.
**Exceptions**: if no software engineer is available for allocation, the use case terminates with a warning for the manager.

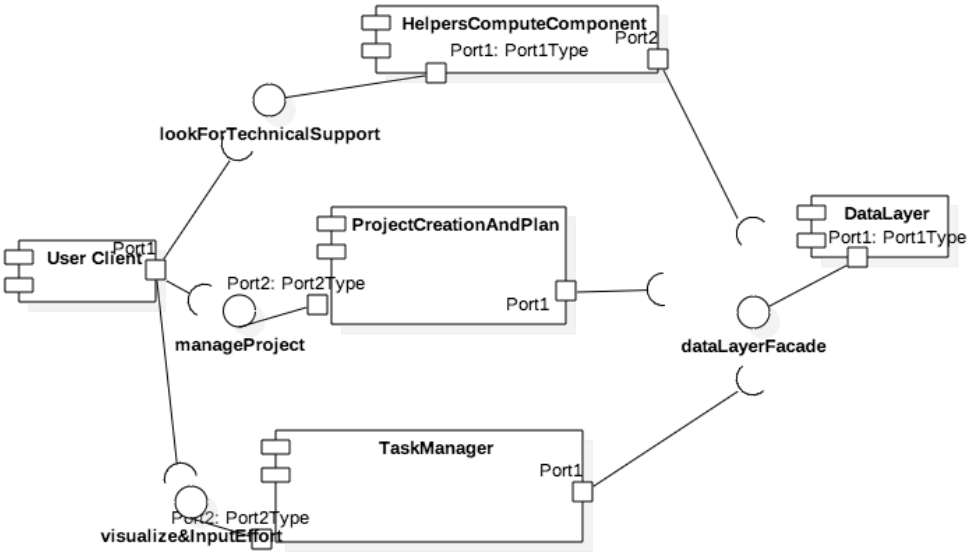
Arguably, the most important use-case is Look for technical support / help:

**Use case name**: Look for technical support
**Participating actors**: project manager or developer, helper
**Entry condition**: the project manager or a developer realizes that there is a need for help
**Flow of events**

- The developer polls the system to get suggestions on possible competent helping-hands
- The system evaluates tasks and completion of remaining company members to identify people who have the skills and some spare time to help.
- The system offers to the developer a list of people he/she could contact to get help.
- The system informs the helper that his/her name and skills have been communicated to a colleague.

**Exit condition**
The use case terminates when the system has computed the list of people. This list can be empty is none of the company members fulfills the requirements in terms of skills and free time.
**Exceptions**: none

B.3)
Architecture:




UserClient is used by all system users and provides interfaces on a role basis. Project managers will be able, through the features offered by the component ProjectCreationAndPlan, to create and manage projects, dividing work into tasks and allocating said tasks to other developers. Developers will be able to visualize their tasks and update the effort they spent on tasks. To enable these functions, the User Client will interact with TaskManager.
Finally, both project managers and developers will be able to exploit the feature offered by the HelpersComputeComponent to identify those people in the company that can provide some help to them. Since the text does not specify this further, for the sake of simplicity, we assume that the interaction with such people occurs outside the system.

**Solution Part C**

**The exercise is about identifying possible acceptance test cases**
- focusing on functional aspects.
- specify the preconditions (if any) that should be true before executing the tests,
- the inputs provided during the test,
- and the expected outputs.

Finally, to explain why you think the selected tests are important for the specific system being considered.

Given the most critical use-cases specified as part of exercise 3.B, it is reasonable to assume that those same use-cases should become scenarios to derive acceptance tests by means of black-box testing. This means that every of the following use-cases should be formulated into an acceptance test specification:

1. **Project staff allocation:**
    - PRECOND – at least one project is inserted and at least one task is ready to be allocated to at least one developer; also, at least one project manager is allocated to the project; finally, the project should be directly related to the project manager following an “instantiation” relation, i.e. the project manager created the project himself and has “ownership” rights;
    - INPUTS – project manager inputs the name of the person or skill needed for the task and should be able to specify the allocation;
    - OUTPUTS – system returns new development network structure that takes the allocation into account;
2. **Task status visualization and update**

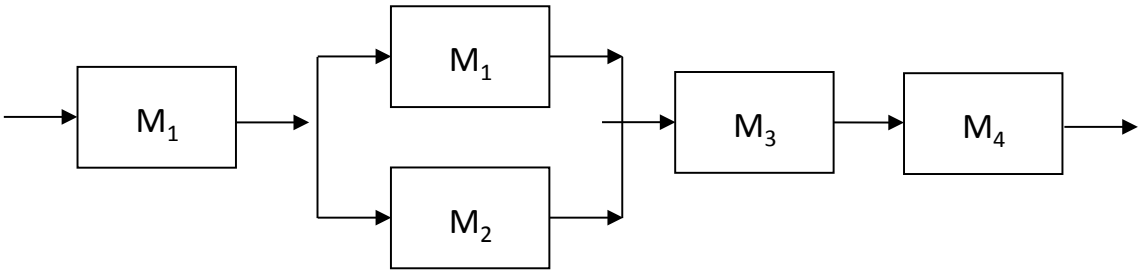
- PRECOND: at least one developer is working on a task X to which he was allocated; system is currently showing old status or <no-status> for task X;
- INPUTS: task X.ID is used to retrieve the status related to the task and the person responsible for it, if the person coincides with the requestor then modification should be permitted;
- OUTPUTS: a new task status should be presented via the system;

3. **Login-logout**
   - PRECOND: system is live; security check component is live;
   - INPUTS: employee credentials;
   - OUTPUTS: visualization of employee tasks;
4. **Get the set of people allocated to a specific project/task**
   - PRECOND: there is at least one project with at least one task and at least one task allocation per tasks present;
   - INPUTS: a user recognized as "project manager" requires to visualize the set of people allocated to a specific project or a specific task;
   - OUTPUTS: graph representation of project/task allocation;
5. **Technical support / help**
   - PRECOND: task-allocations have been done, project is started (i.e., task status updates are present and progress can be computed)
   - INPUT: "need-help-on-task" request;
   - OUTPUT: list of possible candidates for lending a helping hand, matched based on skills and current availability;

These tests are critical since they ensure that key functionalities of the core business logic to be exhibited by the system are actually taken care of. User-acceptance tests performed around said functionalities will make sure that the system performs its intended function in the way the user expects it to.

**Question 2 Architecture and availability (4 points) (13/02/2014)**

Consider the architecture shown in the figure below.




Knowing that the availabilities of components are the following:

| Component | Availability |
|---|---|
| M1 | 0.99 |
| M2 | 0.98 |
| M3 | 0.98 |
| M4 | 0.97 |

1. Compute the availability of the proposed architecture. Given that you are not allowed to use computers and other devices, we do not expect to see precise calculations, but we would like to see the steps you follow in the calculation.

2. Given the following non-functional requirement: “the system must offer an availability equal or greater to 0.98, explain if the architecture as is allows us to fulfill the requirement and, if not, propose proper modifications. Motivate your answers.

**Solution**
**Point 1**
Let’s call $A_5$ the availability of the parallel block composed of $M_1$ and $M_2$.
$A_5=1-(1-A_1)(1-A_2) = 1-(1-0.99)(1-0.98) = 1-0.01*0.02=1-1/100*2/100=1-2/10000=1-0.0002=0.9998$
$Afin=A_1*A_5*A_3*A_4=0.99*0.9998*0.98*0.97=0.9409$

**Point 2**
Even without performing the calculation it emerges that the overall availability is lower than 0.98 because of the presence of a weaker component.
In order to fulfill the requirements we can parallelize $M_4$. In this case, the availability of the last block will be $1-(1-A_4)^2=0.9991$, which leads to a total availability of $0.99*0.9998*0.98*0.9991=0.9691$.
To further increase the availability, we should focus on the next weak element in the chain, that is, $M_3$. We can parallelize it thus achieving $1-(1-A_3)^2= 1-(1-0.98)^2= 0.9996$. This allows us to achieve the availability: $0.99*0.9998*0.9996*0.9991=0.9885$.

**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:

6) Register.
7) Log in/log out.
8) Display the list of available bikes organized in different categories (e.g., mountain bike, city bike, racing bike) and their rental price.
9) Select a bike either from the list of available bikes (see point 3) or by specifying the bike ID (displayed on the bike).
10) 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.
11) Start the rental by inserting the OTP (One-Time Password) received at reservation time on the bike numeric keypad.
12) Cancel the reservation.
13) Terminate the rental.
14) Access the list of all his/her rentals.
15) 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.

**Question 3 Function Points and testing (6 points) 28/06/2017**

A company managing gasoline stations aims at developing an information system.

This system visualizes the status of all gasoline stations in terms of quantity of gasoline available and sales. Such data are acquired from sensors located within the gasoline tanks and from the cash registers.

The system, also manages the process through which a customer refuels his/her car: the customer selects a pump and an amount to pay, inserts his/her credit card to enable the payment, the system after getting the confirmation of the transaction from the bank, enables the selected pump, the customer refuels the car and, optionally, inserts a fidelity card. In this case, the points corresponding to the gasoline purchase have to be added to this card.

Finally, the system alerts the supplying department about the need to buy new gasoline when it goes below a certain threshold.

**A)** Identify: **Internal Logic Files**, **External Inputs** and, if they exist in this case, **External Interface Files**. Define the complexity of each of them providing a motivation.

**B)** You are planning the system testing activities for this piece of software. Define the main test cases that you think are needed in this case. For each test case define the inputs, the expected system state before the execution of the test and the expected result.

**Solution**

**A)**
ILF
GasolineStation: Simple complexity since the structure of this data is simple and we can assume to have a limited number of gasoline stations to manage.
Tank: Simple complexity as the structure of this data is very simple
User (includes the fidelity card): Average complexity as the data structure is simple but we can have a relatively large number of users.

EIF
Streams generated by tank sensors: Simple complexity as it can be seen as a single data transmitted by the sensor when the gasoline goes below a certain threshold.
Data acquired by cash register: Average complexity as they
Confirmation of credit card transaction

External inputs
Refuel request by user

**B)**
**Refuel request**
*Input*: the customer selects the pump to use and the amount to pay
*Expected system state*: the pump is free and available
*Expected result*: the pump is reserved, the customer is asked to perform the payment

**Finalize payment and refuel**
*Input*: the customer inserts the credit card
*Expected system state*: the pump is reserved and the system knows the amount selected by the customer
*Expected result*: if the credit card is valid and the payment finalized, the system enables the pump to release the required amount of gas

**Visualize the status of all gasoline stations**

*Input*: the user selects the menu to visualize the status of the stations
*Expected system state*: before the execution: any
*Expected result*: a list of station with the corresponding gas level is shown

**Alert the supplying department about the need to buy new gasoline**
*Input*: a measurement of the gas level goes below a certain threshold
*Expected system state*: before the execution: any
*Expected result*: the supplying department is alerted