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 1 (WE1)

June 14th, 2024

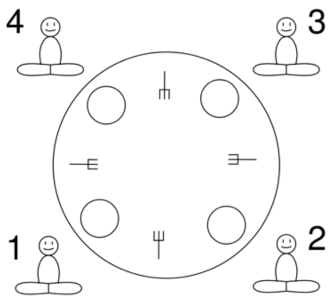
Last name, first name and Id number (matricola):

Number of paper sheets you are submitting as part of the exam:

Notes

A. Write your name and Id number (matricola) on each piece of paper that you hand in.
B. You may use a pencil.
C. Unreadable handwriting is equivalent to not providing an answer.
D. The use of any electronic devices (computer, smartphone, camera, etc.) is strictly forbidden, except for an ebook reader or a plain calculator.
E. The exam is composed of three exercises. Read carefully all points in the text.
F. Total available time for WE1: 1h and 30 mins

Question 1 – Alloy (7 points)
The figure depicts philosophers seated around a round table, with a fork placed between each pair of adjacent philosophers. These philosophers want to eat spaghetti from their plates. In the example illustrated, there are four philosophers. Each philosopher requires both the left fork and the right fork to eat the spaghetti. The philosophers have agreed on the following protocol each of them can execute individually:



- Think about philosophy.
- When hungry, do the following:
    - Take the left fork.
    - Take the right fork and start eating.
    - Return both forks simultaneously and repeat from the beginning.

Consider the following Alloy fragment:

```
abstract sig Status {}
one sig Thinking extends Status {}
one sig Eating extends Status {}
one sig GettingForks extends Status {}

sig Philosopher {
  status: Status,
  leftFork: Fork,
  rightFork: Fork
}{ leftFork not in rightFork }

sig Fork {
  gotBy: lone Philosopher
}{ #gotBy > 0 implies (this in (gotBy.leftFork + gotBy.rightFork)) }
```

It describes philosophers having a state and two forks, one located at his/her left hand side and the other at his/her right hand side. A fork can be available (i.e., on the table) or under the control of the left or the right philosopher.
Answer to the questions below.

Alloy_1 (1 point)
Modify the signatures above to consider that, according to the described protocol, part of the information associated with philosophers and forks may change over time. Provide a justification for your answer.

Alloy_2 (2 points)
Add one or more facts to ensure that when a philosopher is eating, he/she has two forks and when a philosopher is thinking, he/she does not have any fork.

Alloy_3 (4 points)
Write the predicates `gettingLeftFork`, `gettingRightForkAndEat`, `moveToThinking` that, given a philosopher, model, respectively, the operations: 1) acquiring the left fork, 2) acquiring the right fork and start eating, and 3) releasing the acquired forks and getting back to thinking.

Solution

Alloy_1
Fields `status` and `gotBy` should be defined as mutable through keyword `var`, as shown below. In fact, the status of a philosopher can vary overtime according to the protocol described above. Similarly, a fork can be available (on the table) at a certain time instant and in the hands of a specific philosopher in another time instant. In this last case, the gotBy field relates the fork with the specific philosopher under

consideration. The other fields, leftFork and rightFork, as well as the signatures Philosopher and Fork do not evolve overtime.
Given that `gotBy` is now variable, the constraint defined in the `Fork` signature about this field must be true at any time. Therefore, the `always` operator is needed (see below).

```
abstract sig Status {}
one sig Thinking extends Status {}
one sig Eating extends Status {}
one sig GettingForks extends Status {}

sig Philosopher {
  var status: Status,
  leftFork: Fork,
  rightFork: Fork
}{ leftFork not in rightFork}

sig Fork {
  var gotBy: lone Philosopher
}{always(#gotBy > 0 implies (this in (gotBy.leftFork + gotBy.rightFork)))}
```

Alloy_2
```
fact eatingOnlyIfHandlingForks {
always(  all p: Philosopher | p.status = Eating implies
       ((p.leftFork).gotBy = p and (p.rightFork).gotBy = p))

fact thinkingWithoutForks {
always(  all p: Philosopher | p.status = Thinking implies
       (p not in ((p.leftFork).gotBy + (p.rightFork).gotBy)))
}
```
Note the usage of the always operator which is needed to state that the formulas must be true at any time.

Alloy_3
```
pred gettingLeftFork(p: Philosopher) {
  //precondition
  p.status = Thinking
  (p.leftFork).gotBy = none

  //postcondition
  (p.leftFork).gotBy' = p
  p.status' = GettingForks
}

pred gettingRightForkAndEat(p: Philosopher) {
  //precondition
  p.status = GettingForks
  (p.rightFork).gotBy = none

  //postcondition
  (p.rightFork).gotBy' = p
  p.status' = Eating
}

pred moveToThinking(p: Philosopher) {
  //precondition
  p.status = Eating
  //postcondition
  p.status' = Thinking
  p not in (p.rightFork).gotBy'
  p not in (p.leftFork).gotBy'
}
```

Note that, adopting a different interpretation of the transition to thinking, in the `moveToThinking` predicate we could also replace the last two constraints with the following ones:

```
  (p.rightFork).gotBy' = none
  (p.leftFork).gotBy' = none
```

In this second case we are stating that when the philosopher releases the forks, in that time instant, such forks are not in the hands of anyone else. On the contrary, in the original predicate, we adopt a less strict interpretation, and we state only that the forks, in the time instant in which the philosopher enters the Thinking state, are not anymore in his/her hands. This implies also that in that same time instant they might be already in the hands of someone else.

Question 2 – Project Management (5 points)

Consider the following description of ProTennisTracker (PTT), a software system under development.
PTT keeps track of scores of tennis matches. Different categories of users can interact with PTT after successful authentication. The *referee* of a tennis match updates the score of the match when a player wins a point; the information is collected and stored in an internal repository that maintains information on matches. In addition, PTT gathers information from *external speed detection devices* that measure the speed of the serves hit for each point. This information goes into another repository that keeps further information about individual points. The system also relies on *human monitors* who interact with PTT by tagging each point with information about how the point was won. Information on matches and points gathered by the system is made available to *regular users*, who can follow the status of matches in real-time. Regular users can be notified about certain events, such as incoming matches and results (information maintained by PTT), as well as news about players, or match cancelation due to disruptive events (generated by external applications).

Assume that the software system must be delivered in two incremental releases:

1) PTTv1: includes all features that allow interaction with referees, human monitors, and regular users without notifications.
2) PTTv2: in addition to all features of PTTv1, it also includes integration with external speed detection devices and a fully functioning notification subsystem for regular users.

PM _1: We decide to adopt the Function Points (FP) approach for effort estimation. For each version (PTTv1 and PTTv2), draw a high-level schema that illustrates all the function types included in that version. Then, estimate the FP number needed to develop PTTv1 and the FP increment required to implement PTTv2, assuming that the previous one is available.

Estimates must be calculated based on the following weights.

function type	simple	medium	complex
Input	3	4	6
Output	4	5	7
Inquiry	3	4	6
ILF	7	10	15
EIF	5	7	10

Assume also that developers consider the entire system to be simple, except for the integration with external speed detection devices, which they regard as medium complexity.

PM _2: The developing team has to decide whether developing the system using *C* or *Java*. The two options lead to the following considerations.

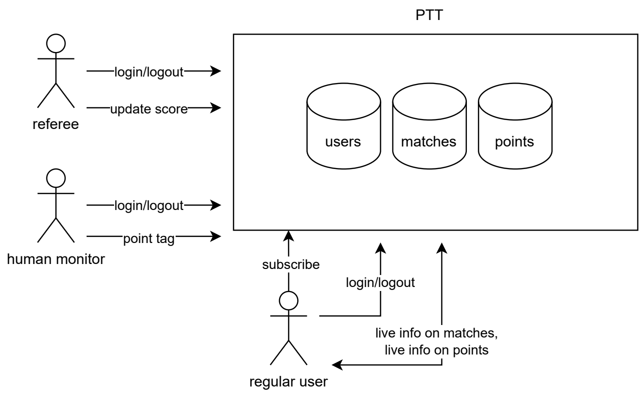
- *C*: is the preferred choice of the team, however speed detection devices cannot be easily integrated. The level of complexity of such integration changes from medium to complex. The average number source statements per FP is 97 in this case.
- *Java*: the team is not familiar with Java and this increases the complexity of some functions, however the integration of speed detectors is simpler in this case. The level of complexity of such integration becomes simple, while all the function types related to the notification subsystem become complex. The average number source statements per FP is 53 in this case.

Assuming that the development team has 10 developers who can write on average 100 lines of code per day, estimate the time to delivery for PTTv1 and then PTTv2. What is the option that minimizes the time to delivery of the whole system?

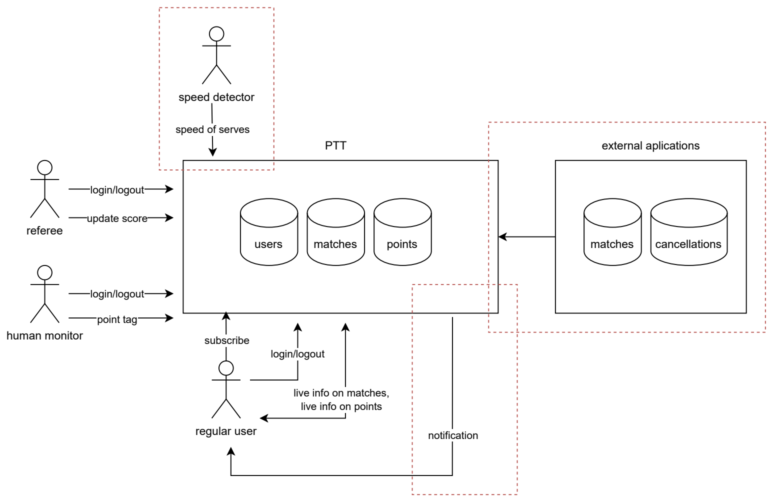
Solution

PM_1

Schema of PTTv1



Schema of PTTv2



FP estimates

PTTv1: 4 x simple input, 2 x simple inquiry, 3 x simple ILF = 4*3+2*3+3*7 = 39 FP
PTTv2: 2 x medium input, 2 x simple EIF, 1 x simple output = 2*4+2*5+1*4 = 22 FP

PM_2

C language

PTTv1:
- 4 x simple input, 2 x simple inquiry, 3 x simple ILF = 4*3+2*3+3*7 = 39 FP
- 39 FP x 97 = 3783 LoC / 1000 ~ 4 working days

PTTv2:
- 2 x ~~medium~~ complex input, 2 x simple EIF, 2 x simple output = 2*6+2*7+1*5 = 31 FP
- 31 FP x 97 = 3007 LoC / 1000 ~ 3 working days

Java language

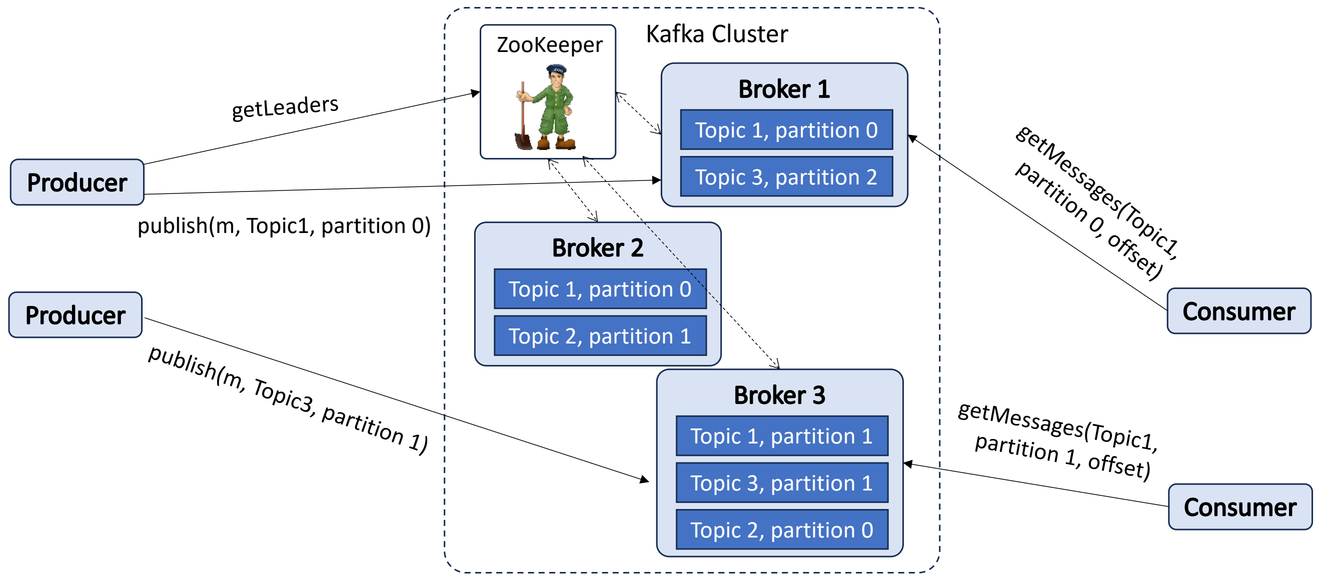
PTTv1:
- 4 x simple input, 2 x simple inquiry, 3 x simple ILF = 4*3+2*3+3*7 = 39 FP
- 39 FP x 53 = 2067 LoC / 1000 ~ 2 working days

PTTv2:
- 2 x ~~medium~~ simple input, 2 x ~~simple~~ complex EIF, 2 x ~~simple~~ complex output = 2*3+2*10+2*7 = 40 FP
- 40 FP x 53 = 2120 LoC / 1000 ~ 2.1 working days

The choice that minimizes time to delivery is Java.

Question 3 – Availability (4 points)

Consider the following instance of a Kafka cluster. Suppose that the servers on which the 3 *Brokers* are hosted have the following availability values.
- Server Broker 1: 95%
- Server Broker 2: 98%
- Server Broker 3: 99%



As illustrated by the schema, *Consumers*, retrieve various messages concerning certain topics. Messages are obtained through the `getMessages` operation, which can retrieve messages from any broker handling a topic and partition. For the purpose of this exercise, assume that the messages have already been published. Assume also that the schema shows all the existing partitions for the depicted topics.

Availability_1: What is the total availability of an operation that needs to retrieve 2 messages related to Topic 1, where the first belongs to partition 1, and the second to partition 0?

Availability_2: What is the total availability of an operation that needs to retrieve 2 messages related to Topic 2, where the first belongs to partition 0, and the second to partition 1?

Availability_3: Assuming that each broker can handle only a single partition for each topic, can we reconfigure the cluster so that the operation of point Availability_2 has a total availability of 99%?

Solution

Availability_1

This is the series of the availability of partition 1 (which is not replicated, so the availability of the partition is the availability of the corresponding server) and partition 0 (which is replicated, so its availability is the parallel of the corresponding servers).

A = 0.99*(1-(1-0.95)*(1-0.98)) = 0.989

Availability_2

This is the series of the availability of partition 0 (which is not replicated, so the availability of the partition is the availability of the corresponding server) and partition 1 (also not replicated).

A = 0.99*0.98 = 0.97

Availability_3

If the number of brokers and servers cannot be increased, the only move we can try, given the constraint of not having partitions of the same topic in the same broker, is to replicate only one of the partitions on two brokers. Considering that the weakest brokers are Broker 1 and Broker 2, we could replicate Topic2, Partition 1 in both of them leaving everything else as in the figure. This would allow us to stay close to the required 99%, which, however, will remain an upper bound for our system, in fact, we will have:

A = 0.99*(1-(1-0.95)*(1-0.98)) = 0.989

Which is slightly lower than 99%.
In case other servers and brokers can be introduced, we could fulfill the constraint.