

# ESOF 423 - Software Engineering Applications

## A1 - Software Testing Assignment

Due on Friday, 02/22 11:59 pm. Worth 40% of assignment points

This assignment consists of two parts.

1. Part 1: Testing a Java application called *Foo's Participation System* (25%).
2. Part 2: Testing a Web application using Selenium (15%)

### Part1: Testing a Java application

(Originally developed by Wishnu Prasetya)

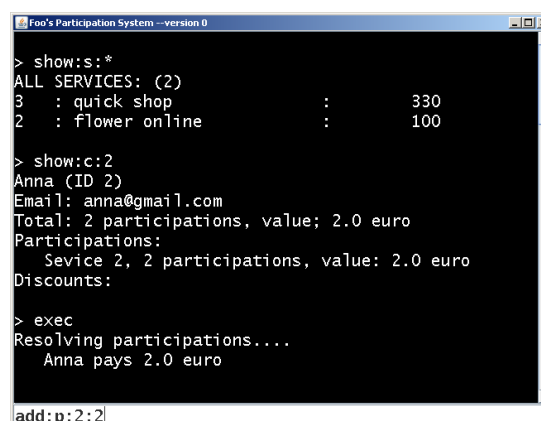
For this exercise you get a small Java application called *Foo's Participation System*. You will get its source code, along with its Javadoc documentation in the *foo.zip* file provided with this assignment. Your task is to test this application; this will be explained in details later.

#### The Application

This Foo System is not a real business system; it is just a simulation for your training purpose. It manages a set of customers, services, and customers' participations on services. It provides basic functionalities, e.g. to add and remove customers, services, etc. A 'service' here represents some business, e.g. an online shop, or a pizza restaurant. The application provides a functionality to let a customer add/buy a participation to a service. In the real life this means that the customer would be charged for daily contribution to the services he participates in. In exchange he will get a proportional share of the services' profit. He can drop participation at any time.

Currently profit calculation is not included in the system, but it does calculate the contribution cost for each customer. Furthermore, there is a functionality to award discount tokens to customers; these give them reduction on the contribution cost they have to pay. The application provides a functionality to 'resolve', intended to be invoked by the application owner at the end of every day. It will calculate the contribution cost of each customer, taking the discount tokens into account. This will also consume relevant tokens.

A screen shot of the application is provided below. You interact with it through a provided client program, by typing a command in the command field at the bottom; it will cause the corresponding functionality to be invoked. See the Appendix for the list of available commands.



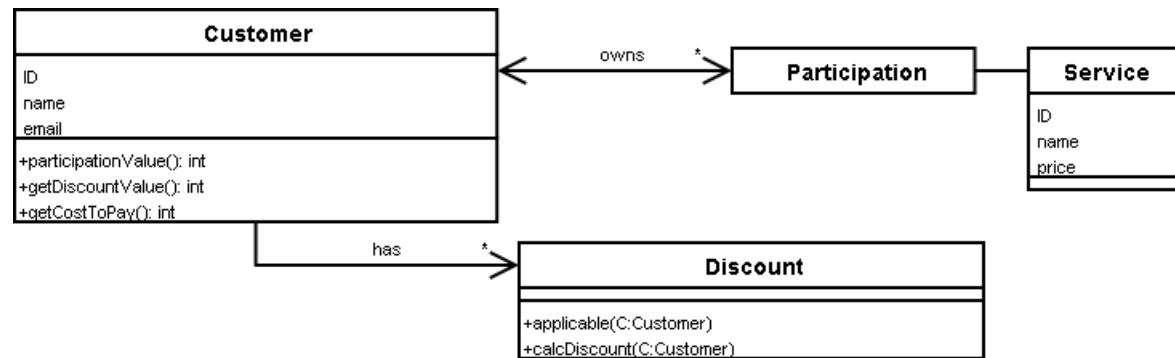
```
Foo's Participation System --version 0
> show:s:*
ALL SERVICES: (2)
3 : quick shop           : 330
2 : flower online       : 100

> show:c:2
Anna (ID 2)
Email: anna@gmail.com
Total: 2 participations, value; 2.0 euro
Participations:
  Service 2, 2 participations, value: 2.0 euro
Discounts:

> exec
Resolving participations...
  Anna pays 2.0 euro

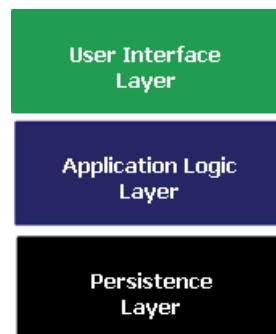
add:p:2:2
```

The class diagram below summarizes the relations between customers, services, participations, and discounts. Administrating these entities, and doing calculation over them is *the* purpose of our Foo System.



Administrating those data is an on-going task in the Foo company. So, we need to save the data when we exit the application, so that we can get them back the next time we start it. Actually we need to regularly save the data, so that we don't lose critical business information in case the application, for whatever reason, crashes.

Although simple, this application is not trivial either. To make it modular, I have split it in layers stacked as in the typical three tiered architecture shown below. The brain of this application is in the *application logic* layer; it is responsible for doing the actual administration of the entities in the class diagram above. The persistence layer is responsible for saving these data safely in a database, and for handling queries on those data. It uses an OO database called db4o. The user interface is implemented in a separate layer.



## Calculating Contribution

Each instance of the class `Discount` represents a discount token. `Discount` itself is an abstract class, promising two methods; see below. If  $t$  is a discount token, then:

- given a customer  $c$ ,  $t.\text{applicable}(c)$  returns true if and only if the customer  $c$  is at this moment eligible to be given the discount promised by  $t$ .
- $t.\text{calcDiscount}(c)$  returns the cost reduction (in eurocent) this token  $t$  would give to  $c$ .

A customer  $c$  can own a set  $P$  of participations and a set  $D$  of discount tokens. His total contribution cost before discount is:

$$R = \sum_{p \in P} p.\text{service}.\text{price}$$

This  $R$  is also called  $c$ 's *participation value*. Let  $D'$  be the subset of  $D$ , consisting of those discount tokens that are applicable on  $c$ . The total discount value of  $c$  is:

$$D = \min\left(\sum_{t \in D'} t.\text{calcDiscount}(c), R\right)$$

The contribution cost that he daily owes to the Foo company is then just:

$$C = R - D$$

The class `Customer` include three methods: `participationValue`, `getDiscountValue`, and `getCostToPay`, to calculate  $R$ ,  $D$ , and  $C$  respectively.

## Discounts

Currently two concrete subclasses of `Discount` are provided:

- `Discount_1000`. This discount is applicable on a customer whose total participation value exceeds 1000 euro.  
This discount gives 50 euro discount to every 1000 euro participation of a customer. E.g. if customer  $c$  owns participations worth 2000 euro, and he has one instance of `Discount_1000`, this will give him 100 euro discount. If he has two instances of this class, he will get 200 euro discount.
- `Discount_5pack`. This discount is applicable on a customer who owns participations in at least 5 services, each with the value of at least 100 euro. It will give 10 euro discount.

## Setting up your test environment

Download the zip of Foo, and unzip.

- The source code is in the subdirectory `src` of Foo's zip.
- The needed `db4o` library can be found in the `libs` subdirectory, provided as a jar file.
- Javadoc documentation is provided in the `docs` subdirectory.
- Some templates of JUnit test-classes are provided in the `mytest` subdirectory.

The best way to setup your testing environment is through Eclipse. Make an *Eclipse* project and make the source code of Foo part of your project.

You need to compile the source code first. Include the `db4o`'s jar in your class path.

Once compiled, running the class `ParticipationSystem` will launch Foo; don't forget to include `db4o`'s jar in your class path.

Use JUnit to do your testing. Use Emma to see the coverage of your tests. JUnit is already integrated in Eclipse. Emma can be installed as a plugin of Eclipse (this should have been done on our labs PCs).

## What to do:

Test the methods (10) listed below. You don't have to do them all, but you can only get your full point (10 pt) if you do.

1. from the class `Customer`:
  - (a) `getParticipationValue`

- (b) `getDiscountValue`
  - (c) `getCostToPay`
  - (d) `getParticipationGroups`
2. from the class `Discount_1000`:
    - (a) `applicable`
    - (b) `calcDiscount`
  3. from the class `Discount_5pack`:
    - (a) `applicable`
    - (b) `calcDiscount`
  4. from the class `ApplicationLogic`:
    - (a) `removeService`
    - (b) `resolve`

They should be tested whether they meet their (informal) specifications as described in the provided JavaDoc documentation for each. Your test should be adequate —we will use Emma’s branch coverage as our coverage criterion.

## Grading

Each method from the list above is worth 1pt, and is graded as follows:

- Your tests should give full (100%) coverage on the method. Else, obviously your tests are incomplete. For this you will get no point (0).
- It is important that you provide a good test suite against the method’s specification. If you only partially check the specification, you may miss fatal bugs. So, you will only get 0.5 pt if your tests are only partial. If your test expectations are really trivial (these are test expectations that will never reveal any bug), you get no point.

There are some templates of JUnit test classes that you can use as starting point. See the directory `mytest`.

## Bonus: Finding error

0.2 pt for each error you find, to a max. of 1 pt bonus.

## What to submit:

Submit all the \*.java files that you develop for writing JUnit test cases. We will be executing them by putting it in the *foo*’s package structure. So make sure to import all the relevant packages.

### • Appendix - Foo's commands

- `help`    print a list of available commands.
- `show:c:*`    list all customers.
- `show:s:*`    list all services.
- `show:c:id`    show the information of the customer with the given *id*. The *id* is an integer.
- `show:s:id`    show the information of the service with the given *id*. The *id* is an integer.

- `add:c:name:email` add a customer with the given name and email.
- `add:s:name:price` add a service with the given name and daily contribution price (in euro-cent).
- `add:p:cid:sid` add one unit of participation to the service *sid* to the costumer *cid*.
- `rem:c:id` remove the customer with the given *id*.
- `rem:s:id` remove the service with the given *id*.
- `rem:p:cid:sid` drop one unit of participation on the service *sid* owned by customer *cid*.
- `exec` calling `resolve`.
- `clear` clear the screen.
- To quit just close the window.

## Part 2 - Testing a Web application using Selenium.

This part of the assignment is about getting to get a deeper understanding of the Selenium IDE tool and the power of test automation. It is important that you read the question and answer accordingly.

### What do to:

1. First, **navigate to** [https://en.wikipedia.org/wiki/Main\\_Page](https://en.wikipedia.org/wiki/Main_Page)
2. **Create** a test case for each the following functionalities:
  - 2.1 **Create** a new user account
  - 2.2 **Login** to a user account
  - 2.3 **Logout** from a user account
3. **Save the test cases** as a test suite (i.e. one file) and call the file "*test-suite-wikipedia*"
4. For each of the three(3) test cases in [2], **briefly explain why/why not** test automation was not possible.
5. Submit the \*.side file you created and the answer to question 4 in a pdf file.