

Concordia University

Department of Computer Science and Software Engineering

**Kurosh Farsimadan (40047131)**

**DESIGN DOCUMENT FOR DISTRIBUTED COURSE  
REGISTRATION SYSTEM (DCRS)**

Instructor: Professor R. Jaykumar

## TABLE OF CONTENTS

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>UML FOR DCRS .....</b>                                | <b>3</b>  |
| 1.1      | SOEN CLASS DIAGRAM .....                                 | 3         |
| 1.2      | INSE CLASS DIAGRAM .....                                 | 3         |
| 1.3      | COMP CLASS DIAGRAM.....                                  | 4         |
| <b>2</b> | <b>DATA STRUCTURE.....</b>                               | <b>4</b>  |
| <b>3</b> | <b>SYSTEM ARCHITECTURE .....</b>                         | <b>6</b>  |
| 3.1      | SERVER ARCHITECTURE AND LOGIC .....                      | 7         |
| 3.2      | CLIENT ARCHITECTURE AND LOGIC.....                       | 7         |
| <b>4</b> | <b>TEST CASES .....</b>                                  | <b>8</b>  |
| 4.1      | TEST CASE 1: ENROLL MULTIPLE STUDENTS INTO A COURSE..... | 8         |
| 4.2      | TEST CASE 2: ADD TWO SIMILAR COURSES .....               | 9         |
| 4.3      | TEST CASE 3: ENROLL TO A NEWLY ADDED COURSE .....        | 9         |
| <b>5</b> | <b>OTHER.....</b>  | <b>10</b> |
| 5.1      | FUTURE CONSIDERATIONS .....                              | 10        |
| 5.2      | MOST IMPORTANT PART IN THIS PROJECT.....                 | 10        |
| 5.3      | MOST DIFFICULT PART IN THIS PROJECT .....                | 11        |
| 5.4      | CODING STANDARDS AND PATTERNS .....                      | 11        |
| 5.5      | JAVADOCS.....  | 11        |
|          | <b>REFERENCES .....</b>                                  | <b>11</b> |

# 1 UML FOR DCRS

In this chapter, I have attached the class diagram of each project component. By component I mean an Eclipse generated package. There are about 4 components in the project as listed below. Each component is responsible either for the client or the university departments as shown in the figure 1.

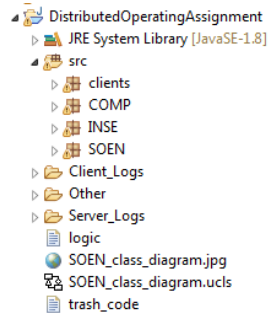


Figure 1. Project hierarchy

The figures were made using Object Aid UML autogenerator [1], which helped in fast UML generation for my project. The orange method names mean “protected” and the green ones mean “public”. The original pictures are in the Java project folder named “Class\_Diagrams”.

## 1.1 SOEN class diagram

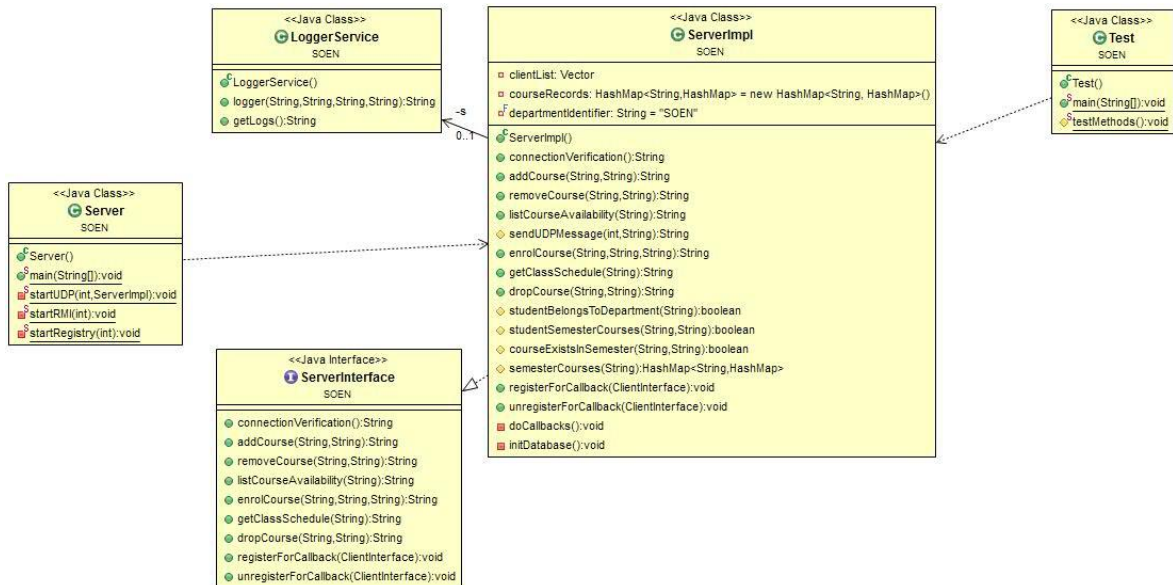


Figure 2. SOEN class diagram

## 1.2 INSE class diagram

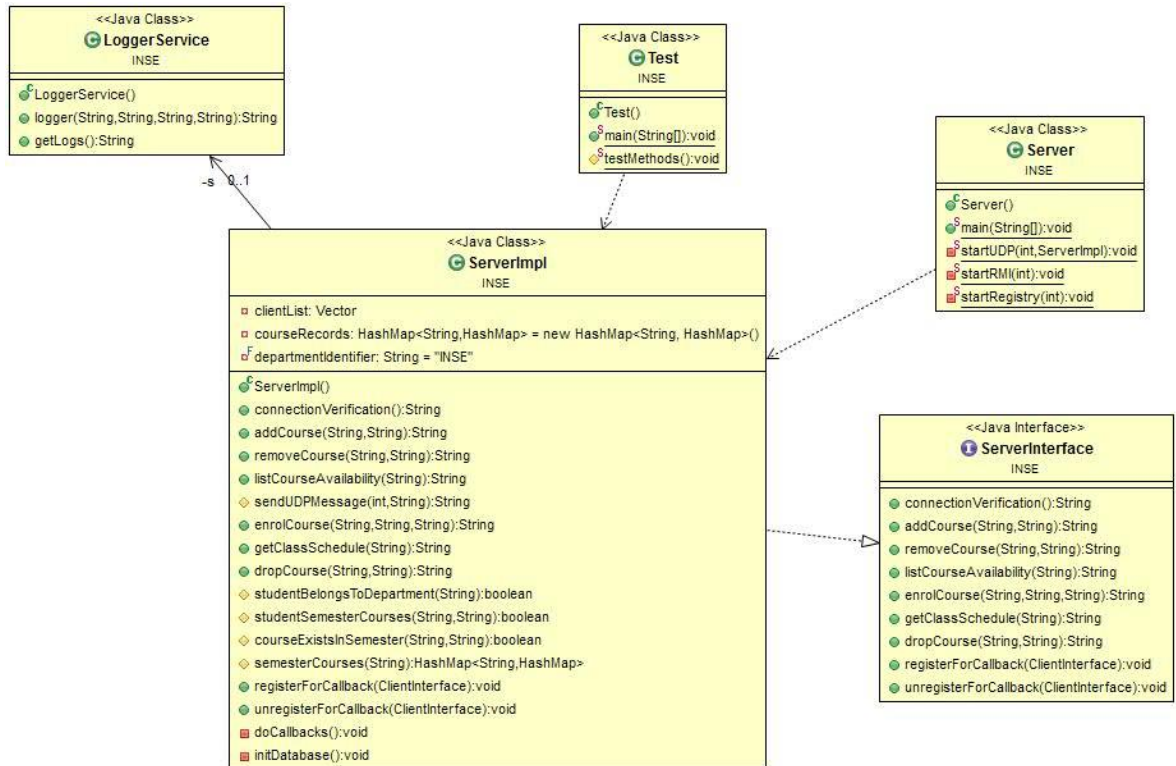


Figure 3. INSE class diagram

### 1.3 COMP class diagram

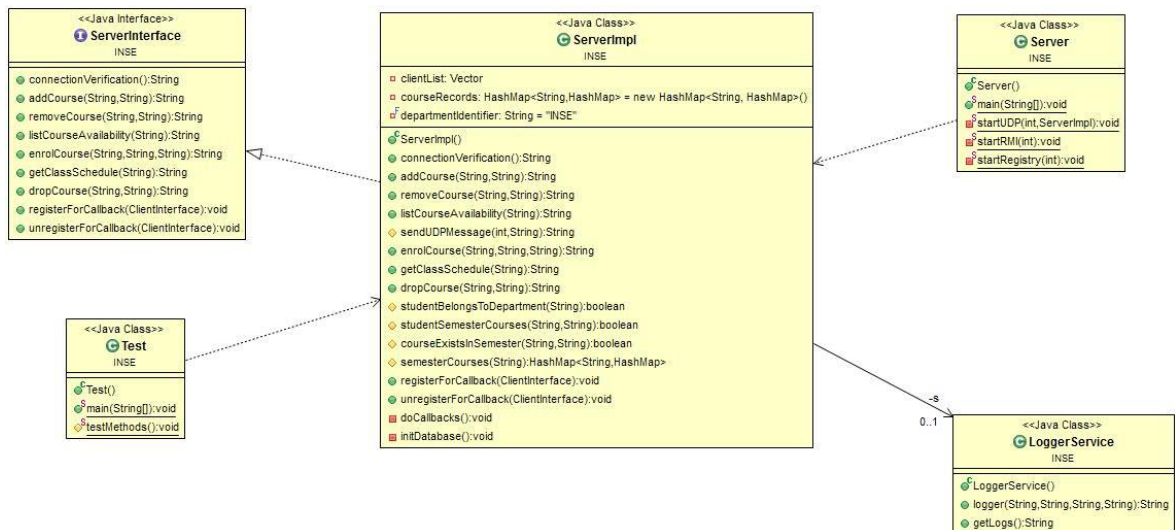


Figure 4. COMP class diagram

## 2 DATA STRUCTURE

The main data structure in the implementation of the whole project and mainly the university department servers were HashMaps. The primary variable in holding the shared semester, course, student, and course information data is location in each “ServerImpl” class and the name of this variable is “courseRecords” as seen in the below figure 5. It is initialized as a class attribute in the top of the class

```
package INSE;

import java.io.IOException;

public class ServerImpl extends UnicastRemoteObject implements ServerInterface {

    private Vector clientList;

    private HashMap<String, HashMap> courseRecords = new HashMap<String, HashMap>();

    // So that we can only change one field if the server is different (different
    // identifier)
    private final String departmentIdentifier = "INSE";
```

Figure 5. Main shareable variable

The data structure is logically structured as follows in textual format, which is not similar to JSON or XML, but hierarchically it could be similar to aforementioned standards and technologies:

```
{WINTER={INSE349={Capacity={TotalCapacity=3,Registered=1},RegisteredStudents={I
NSES1234=Kurosh Farsimadan}, Information={Details=This is testing material}},
INSE1234={Capacity={TotalCapacity=3,Registered=1},RegisteredStudents={INSES1234
=Kurosh Farsimadan}, Information={Details=This is testing material}}}}
```

In essence, the data structure is hierarchically shown as the following (in HashMaps, the order does not matter) with their corresponding meaning or problem that they solve:

- Course records (variable)
  - o Semester 1 (WINTER)
    - Course 1 (INSE349)
      - Course information (multiple HashMaps)
        - o Capacity
          - Total capacity (for available seats)
          - Registered (how many has registered so far)
        - o Information (string description of the course)

- Registered students (students who have registered in the course)
  - Course 2 (INSE1234)
    - Semester 2 (FALL)
    - Semester 3 (SUMMER)

In modifying the primary data structure, temporary HashMaps were heavily used as can be seen in the source code. Please see the discussion for the concerns and problems in this design document.

### 3 SYSTEM ARCHITECTURE

The DCRS application is made of 4 components. Each component could be started separately as they mock standalone applications. However, in the current solution with only one instance of COMP, SOEN, and INSE servers being showcased, each department server must be up and running for them to be able to communicate with each other using UDP protocol in retrieving other department courses.

Similarly, the client is dependent on these three server instances as the logic does not take into account the case where one of the department servers is down. Hence, if the user of the client wants to invoke a remote object method in SOEN server, then the SOEN server must be up and running as seen in the figure 6.

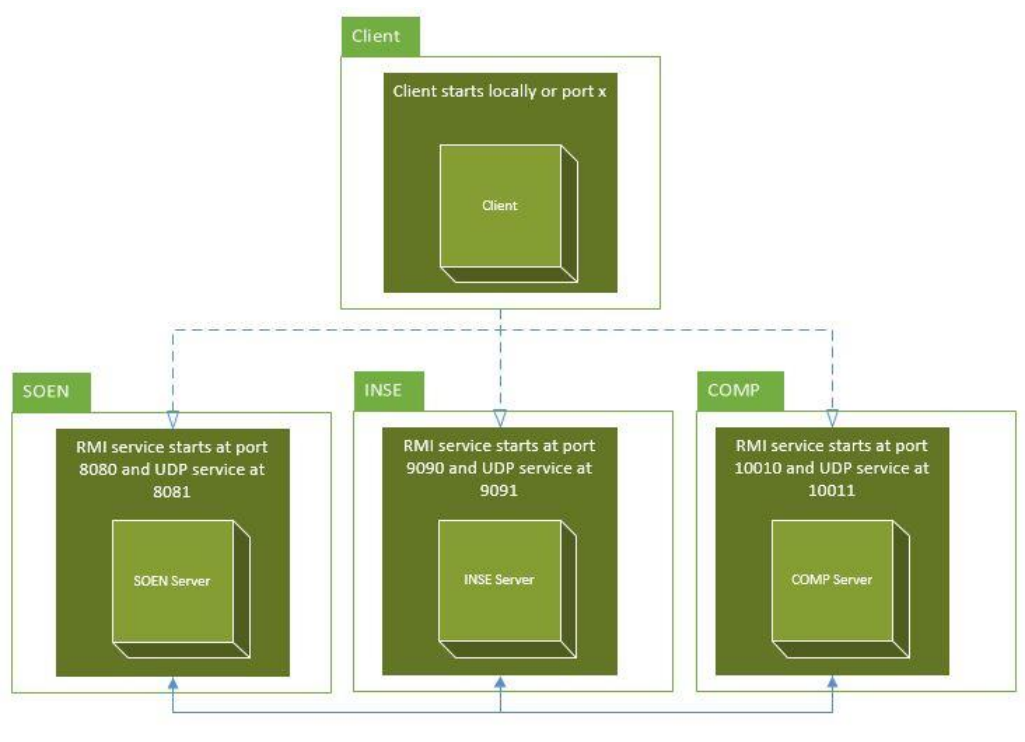


Figure 6. Software architecture

### 3.1 Server architecture and logic

When any of the servers is started, they will start on specific port as defined in the “Server” class. This static port lookup could be dynamic, but as a result of time constrain, each server must have a defined port. Each university department server has their own specific ports on which they will start like the following:

- SOEN
  - o RMI service starts at 8080
  - o UDP service starts at 8081
- INSE
  - o RMI service starts at 9090
  - o UDP service starts at 9091
- COMP
  - o RMI service starts at 10010
  - o UDP service starts at 10011

At the current moment, the server logic is not created in a way that it would look up for alternative servers in case if one of the two other department servers will go down. This fact was acknowledged while I was creating the servers. The solution for one instance for each server was to implement a timeout (currently it is at 10 seconds for each department server) when doing a UDP call with an appropriate error message. Both the RMI and UDP service is started in the “Server” class.

Each server saves each request made to it into their own respective folders e.g. “SOEN\_Server\_Logs” with the necessary details so that the request could be traced back to a specific time and user.

### 3.2 Client architecture and logic

For demo purposes, the client side architecture is not something that could be cheered as following the state-of-the-art object-oriented patterns and designs, but it has a fairly good error handling and logic.

The client at the current moment is just a locally started Java console input and output program with the following flow:

- Ask the user for person ID
  - o The system checks whether the user is an advisor or student
  - o The system does not check whether the student exists since this was not part of the server requirement
  - o If the person id is not recognizable i.e. if it not in the format of COMPA or COMPS for example, then the user is prompted to give the person ID again.
- The system will automatically check to which department the person belongs to and passes the necessary variable values to client handler method
- The client handler method will showcase the person specific methods based whether or not the person is an advisor or student

- The person must choose one of the available method that is shown to him
- The system will then check on which department the person belongs to and makes a call to the specific server and method
- After the message from the server has been retrieved, the message will be saved into a log file

Client saves each request it made to it into a folder called “Client\_Logs” with the necessary details so that the request could be traced back to a specific time and user.

## 4 TEST CASES

In designing test cases, I had many different options available for me like JUnit. I came to a conclusion that for a small project like this, manual testing will be sufficient. However, I did add one test case (test case 2), which mocks JUnit e.g. we want to automate testing by calling certain service methods with specific sets of messages in return.

Maybe even a logger could be used for the testing for failed test cases in a particular day or hour. Because there is a page limit, I will only showcase 3 test cases. The rest of the test cases can be seen in SOEN packages “Test” class. To some degree, some test cases must be chained if we want to test the whole chain.

In testing the results for different clients so that the thread and synchronization logic is properly done, I conducted this manually.

### 4.1 Test case 1: Enroll multiple students into a course

The primary purpose of this test case is to test whether the server works as it should i.e. the student data is saved into the HashMap and a proper message is returned whether or not the enrollment was successful.

When trying to enroll multiple students as shown in the below figure 7, the system should return the proper error.

```
protected static void testMethods() {
    ServerImpl exportedObj;
    try {
        exportedObj = new ServerImpl();

        System.out.println("Testing begins...");

        System.out.println(exportedObj.enrolCourse("SOENS3355", "SOEN1234", "WINTER"));
        System.out.println(exportedObj.enrolCourse("SOENS3355", "SOEN1234", "WINTER"));
        System.out.println(exportedObj.enrolCourse("SOENS3355", "SOEN1234", "WINTER"));
        System.out.println(exportedObj.enrolCourse("SOENS3355", "SOEN1234", "WINTER"));
        System.out.println(exportedObj.enrolCourse("SOENS3322", "SOEN1234", "WINTER"));
        System.out.println(exportedObj.enrolCourse("SOENS32322", "SOEN1234", "WINTER"));
        System.out.println(exportedObj.enrolCourse("SOENS2255", "SOEN1234", "FALL"));
        System.out.println(exportedObj.enrolCourse("SOENS3355", "SOEN123444", "WINTER"));
        System.out.println(exportedObj.addCourse("SOEN883", "WINTER"));
    }
}
```

Figure 7. Code snippet for test case 1

For the above server service call, we get a response as shown in figure 8, which is the expected result as each server initiates and populates two courses (SOEN349 and SOEN1234 for example) with one student. We can see that we are able to register two students, but the other requests are not valid.



```

Test (3) [Java Application] C:\Program Files\Java\jre1.8.0_181\bin\javaw.exe (11.10.2018 klo 17.08.50)
Testing begins...
The student SOENS3355 is registered for the course SOEN1234
The student SOENS3355 is already registered for the course SOEN1234
The student SOENS3355 is already registered for the course SOEN1234
The student SOENS3355 is already registered for the course SOEN1234
The student SOENS3322 is registered for the course SOEN1234
The course SOEN1234 has no room left
Semester not found in the database
Course SOEN123444 not found in the database

```

Figure 8. Console output for test case 1

This message returning is an important part of any component or service whether it is an RMI, UDP, API web service returning JSON or XML, or a queue.

## 4.2 Test case 2: Add two similar courses

In this test case, we want to test whether we are allowed to add a new course for a particular semester and if we are able to add another similar course for the same semester as shown in the below figure 9.

```

System.out.println("");

String message = exportedObj.addCourse("SOEN883", "WINTER");
if (message.toString().equalsIgnoreCase("A new course SOEN883 has been added for WINTER semester")) {
    System.out.println("Adding a new course was successful with the message: " + message);
} else {
    System.out.println("Test case failed with the message: " + message);
}

System.out.println("");

message = exportedObj.addCourse("SOEN883", "WINTER");
if (message.toString().equalsIgnoreCase("A new course SOEN883 has been added for WINTER semester")) {
    System.out.println("Adding a new course was successful with the message: " + message);
} else {
    System.out.println("Test case failed with the message: " + message);
}

System.out.println("");

```

Figure 9. Code snippet for test case 2

The console output is like the following with the expected results.

```

Test (3) [Java Application] C:\Program Files\Java\jre1.8.0_181\bin\javaw.exe (11.10.2018 klo 18.36.26)

Adding a new course was successful with the message: A new course SOEN883 has been added for WINTER semester
Test case failed with the message: A course SOEN883 already exists for WINTER semester

```

Figure 10. Console output for test case 2

## 4.3 Test case 3: Enroll to a newly added course

In this test case, we want to test if we can enroll to a newly added course as shown in test case 2. See the below figure for the test case scope.

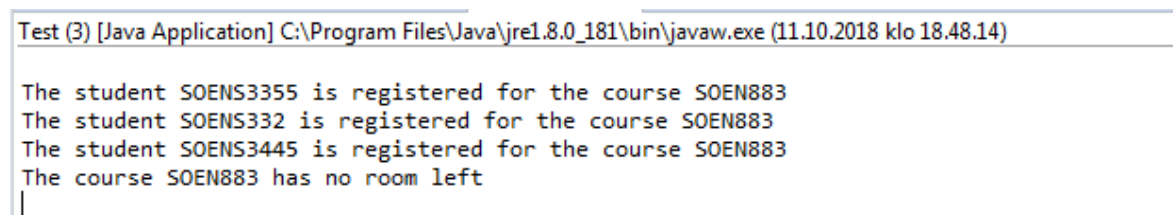
```
System.out.println("");

System.out.println(exportedObj.enrolCourse("SOENS3355", "SOEN883", "WINTER"));
System.out.println(exportedObj.enrolCourse("SOENS332", "SOEN883", "WINTER"));
System.out.println(exportedObj.enrolCourse("SOENS3445", "SOEN883", "WINTER"));
System.out.println(exportedObj.enrolCourse("SOENS3245", "SOEN883", "WINTER"));

System.out.println("");
```

Figure 11. Code snippet for test case 3

The console output is as expected as shown in the below figure.



The screenshot shows a Java console window titled "Test (3) [Java Application] C:\Program Files\Java\jre1.8.0\_181\bin\javaw.exe (11.10.2018 klo 18.48.14)". The output text is as follows:

```
The student SOENS3355 is registered for the course SOEN883
The student SOENS332 is registered for the course SOEN883
The student SOENS3445 is registered for the course SOEN883
The course SOEN883 has no room left
|
```

Figure 11. Console output for test case 3

## 5 OTHER

### 5.1 Future considerations

There are a lot of “if only” and “I wish I have included this” scenarios in this project. In the future, all of the servers need to be refactored to follow the object-oriented design principles. For example, Student methods could be in their own classes have its own interface for the client with the advisor methods.

Also, the servers should start dynamically in the range of determined available ports so that they would not need to be hardcoded into the Servers. This means that servers must find other server instances dynamically.

The servers should also be able to dynamically discover other servers when one instance of for example the SOEN is down. This way, there would never be service interruptions.

Furthermore, for data validity, we should cross check the data from example the SOEN department by retrieving the data from at least 3 different SOEN department servers.

There are a lot of other concerns and future development ideas, but these were the main ones for starters.

### 5.2 Most important part in this project

The most important part of this assignment for me was to design each department server in a way that allows the RMI and UDP services to run under the same program / package.

Also, the proper threading and synchronization is needed for the whole application to run as expected so that the data is not corrupt.

### **5.3 Most difficult part in this project**

The most difficult part was to handle the HashMap. I usually do not use in-memory structures and HashMaps are bad (or good depending on the developer) as they are hard to maintain (HashMap referencing). Otherwise, the whole project was fairly straightforward and easily testable both automatically and manually.

Another difficult part was to manage all three different instances of the department servers. My problems will be explained in the following sub-chapter, but I believe that although it is good to test the whole application once in a while, it is still better to develop one good server application and then copy and modify the needed configurations.

### **5.4 Coding standards and patterns**

Although I tried to create the project in a clean and orderly fashion, there are some places where I could have used better patterns. In try-catch error throwing, I could have used a class with specific error messages for specific cases. In message returning, I could have used a class or method that would return the specific messages based on the outcome.

Another problem that I had was related to the department specific server configurations. This could have been easily avoided if proper object-oriented principles would have been followed with a separate configuration file.

All in all, the project serves as a demo or proof-of-concept.

### **5.5 Javadocs**

The auto generated Java docs are inside the “doc” folder. The java documentation automatically retrieves the necessary documents and could be used in a similar fashion to WSDL or Swagger UI for the possible users of the system. Currently the javadoc only stands as a demonstration and the comments have not been written in an order that could help the user to understand the available services.

## **REFERENCES**

1. <http://objectaid.com/class-diagram>