Students' algorithm teaching tool
Analysis and Design Document

Student: Rares-Sebastian Moldovan
Group: 30433

Revision History

Date	Version	Description	Author
02/04/2018	1.0	First version of the documentation.	Rares-Sebastian Moldovan
23/04/2018	1.1	Completed Iteration 1.1	Rares-Sebastian Moldovan
27/05/2018	2.0	Final deliverable version.	Rares-Sebastian Moldovan

Table of Contents

I. Project Specification 4

II. Elaboration – Iteration 1.1 5

1. Domain Model 5

2. Architectural Design 6

2.1 Conceptual Architecture 6
2.2 Package Design 7
2.3 Component and Deployment Diagrams 7

III. Elaboration – Iteration 1.2 8

1. Design Model 8

1.1 Dynamic Behavior 8
1.2 Class Design 9

2. Data Model 10

3. Unit Testing 11

IV. Elaboration – Iteration 2 12

1. Architectural Design Refinement 12

2. Design Model Refinement 13

V. Construction and Transition 15

1. System Testing 15

2. Future improvements 16

VI. Bibliography 17

I. Project Specification

The "Students' algorithm teaching tool" uses Java API for implementing a web application that addresses the problem of sharing information about algorithms among the Computer Science department of the University.

The application has three types of users: system administrators, algorithm administrators and regular users.

- The system administrator is a teacher/student that takes care of administrative concerns such as deleting redundant information and assigning roles to the other users. He can perform CRUD operations on all accounts and all algorithm pages. Moreover, the administrator is responsible for generating reports about the user activity and ratings.
- The algorithm administrator is a teacher/student that takes care of a certain algorithm page. A regular user can take the role of an algorithm administrator if and only if this role is assigned to him by a system administrator. He can perform CRUD operations on the algorithm page for which he is responsible and must validate regular user inquiries on the algorithm page. He also takes care of generating reports about user activity and ratings in his algorithm page.
- The regular user has access to all algorithm pages in view-mode but cannot modify by his own the information on the page. In order to submit a new implementation/description for an algorithm, the user must send a request to the algorithm administrator and only after it is validated, the administrator will submit the information on the algorithm page. Moreover, the regular user (as well as the administrators) can rate the resource from 1 to 10 so that the system can keep a list of best-rated users for the entire system or for a certain algorithm page. The information itself will also be assigned an average rating that indicates how reliable this implementation/theoretical aspect is.

Regarding search strategies, a user can search information based on:

- Algorithm name.
- Disciplines.
- Users.
- Programming language.

Pages for disciplines will also be provided with fields that display all the available information regarding the corresponding algorithms of that discipline.

An algorithm page is characterized by the following fields:

- Title
- Associated discipline
- An algorithm administrator associated with a "Request to add information button" which opens a form for sending an inquiry for the administrator to accept.
- A text-field/document that presents the general theoretical aspects of the algorithm. This field introduces definitions, algorithm steps and examples in a purely theoretical manner. No actual implementation is allowed.

- A variable number of text fields which present implementations in different languages. Such algorithm information is characterized by the title of the technology used and the actual implementation. Any number of implementations is supported by the application for one algorithm.

II. Elaboration – Iteration 1.1

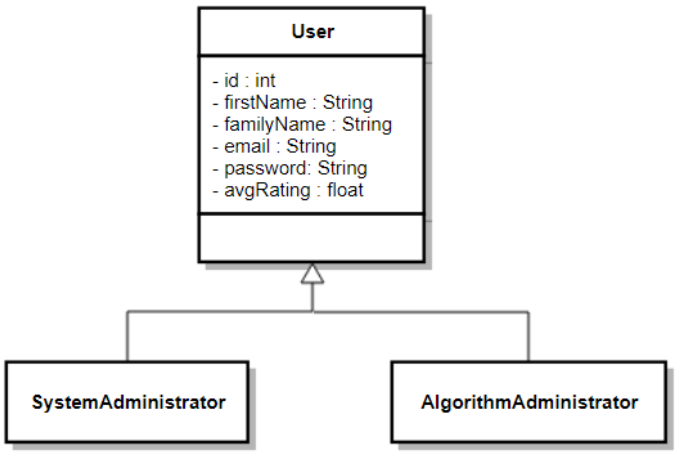
1. Domain Model

The key concepts in the domain model are:

- User – regular, algorithm administrator or administrator
- Algorithm
- Discipline
- Programming language
- Text information (theoretical or code).

These will become the entities in the database, as well as the entity classes in data layer of the application.

In the application, a clearer distinction between users will be highlighted by inheritance as follows:

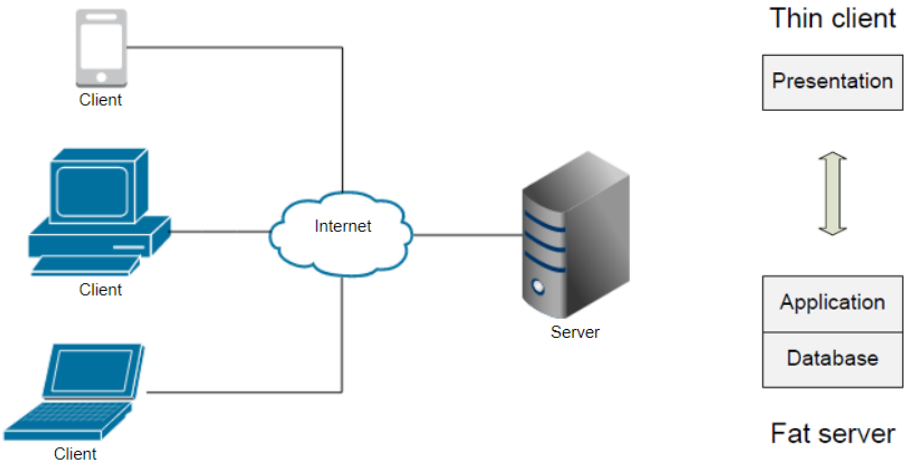


Moreover, the many to many relationships will not be kept in separate entities in the application. The application will make use of Collections to store these relationships.

- The algorithm entity will store a list of information objects associated with the page.
- The information entity will contain the actual user object.
- The algorithm entity will contain the actual user administrator object.

2. Architectural Design

2.1 Conceptual Architecture

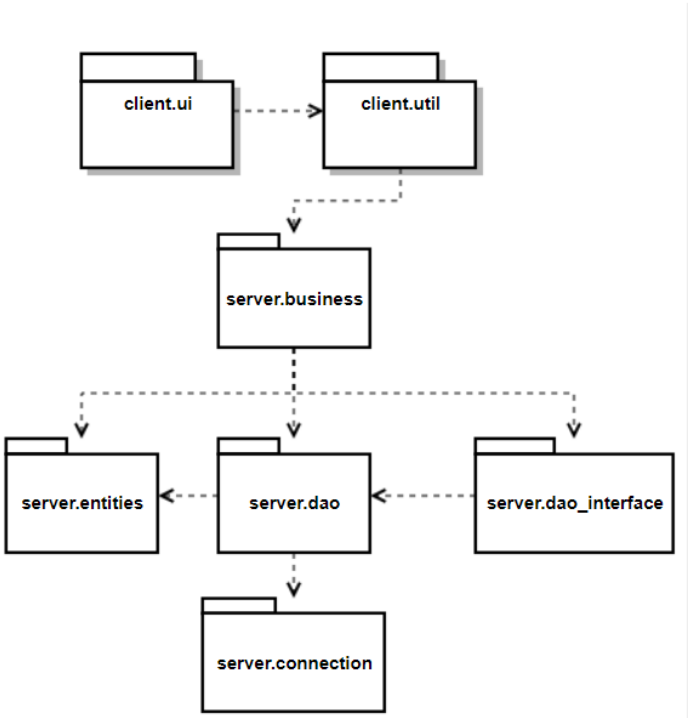


The system uses the client-server architecture. It is an obvious choice because of the need of many users and one processing unit which needs to provide services for each of these users.

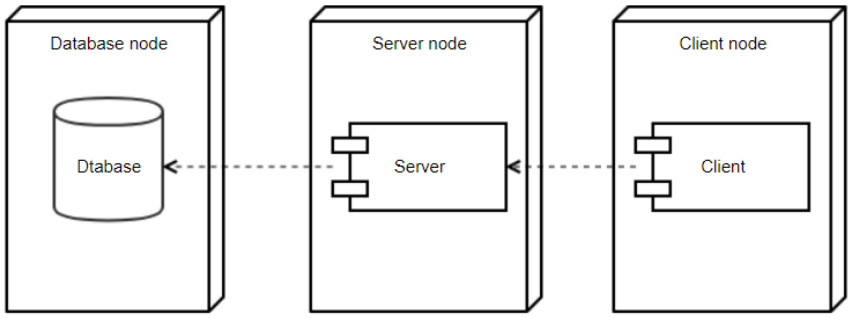
- The server will take care of accessing the database, as well as perform the business logic for the user commands. Therefore, it will be a Fat Server.
- The client will display the user interface and send commands to the server via the Internet connection. Therefore, it will be a Thin Client.

This type of approach is called **Sever-based processing**, since the server does the hard part while the client only displays data to the user and calls the server for any type of transaction that needs to be performed.

2.2 Package Design



2.3 Component and Deployment Diagrams

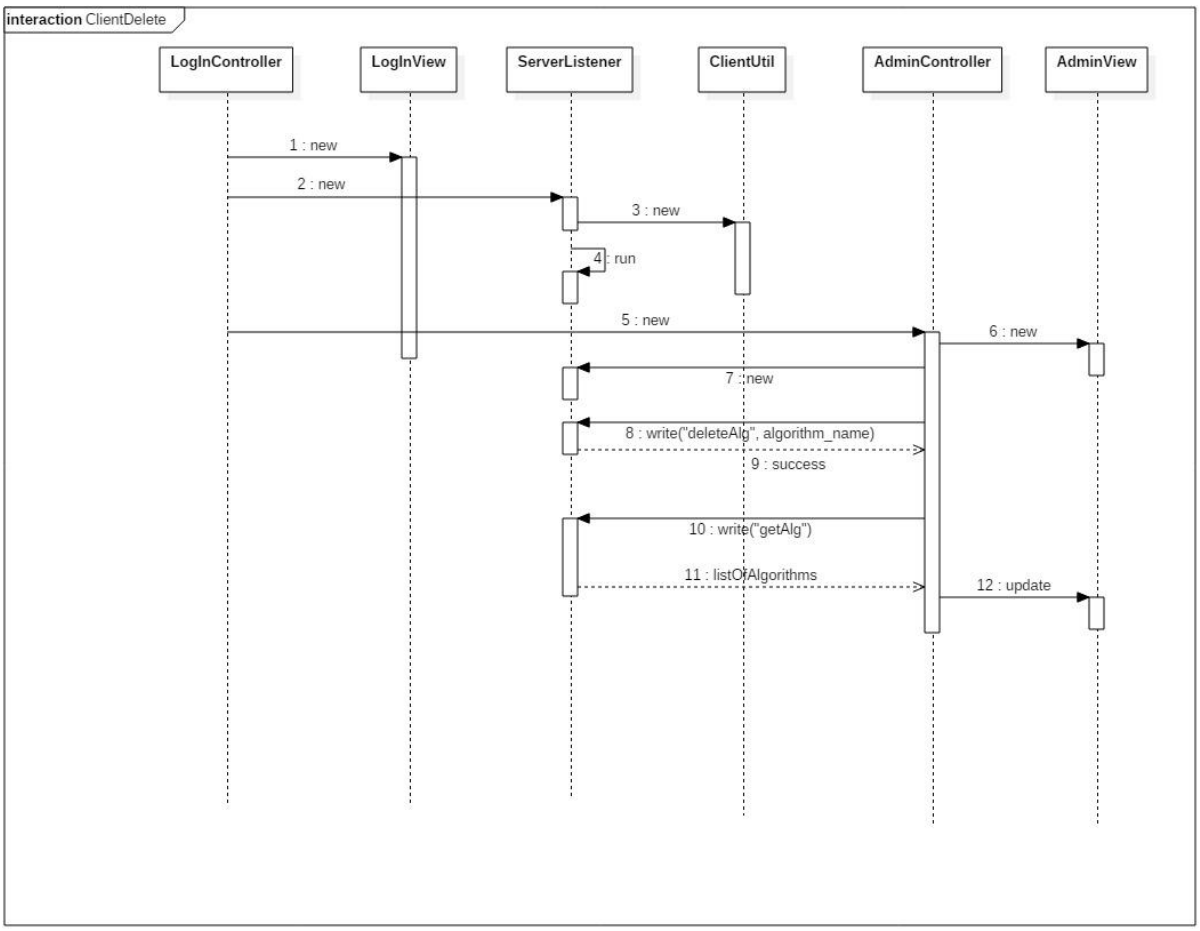


III. Elaboration – Iteration 1.2

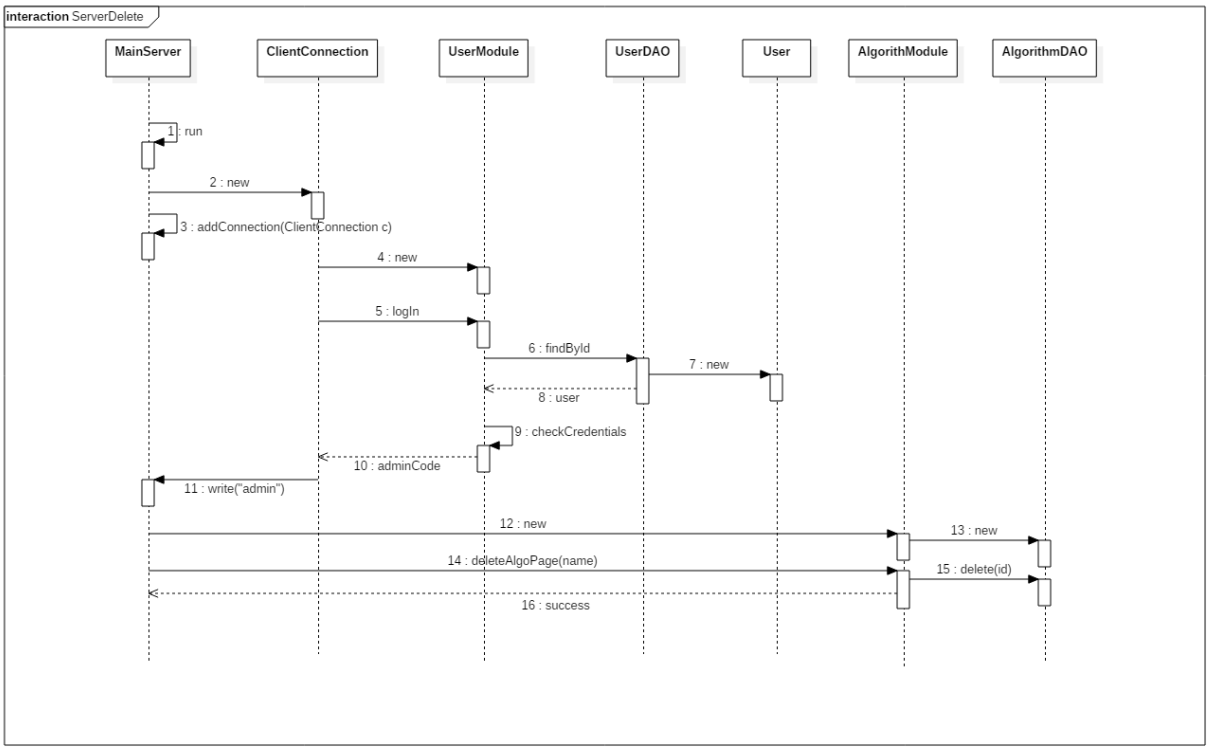
1. Design Model

1.1 Dynamic Behavior

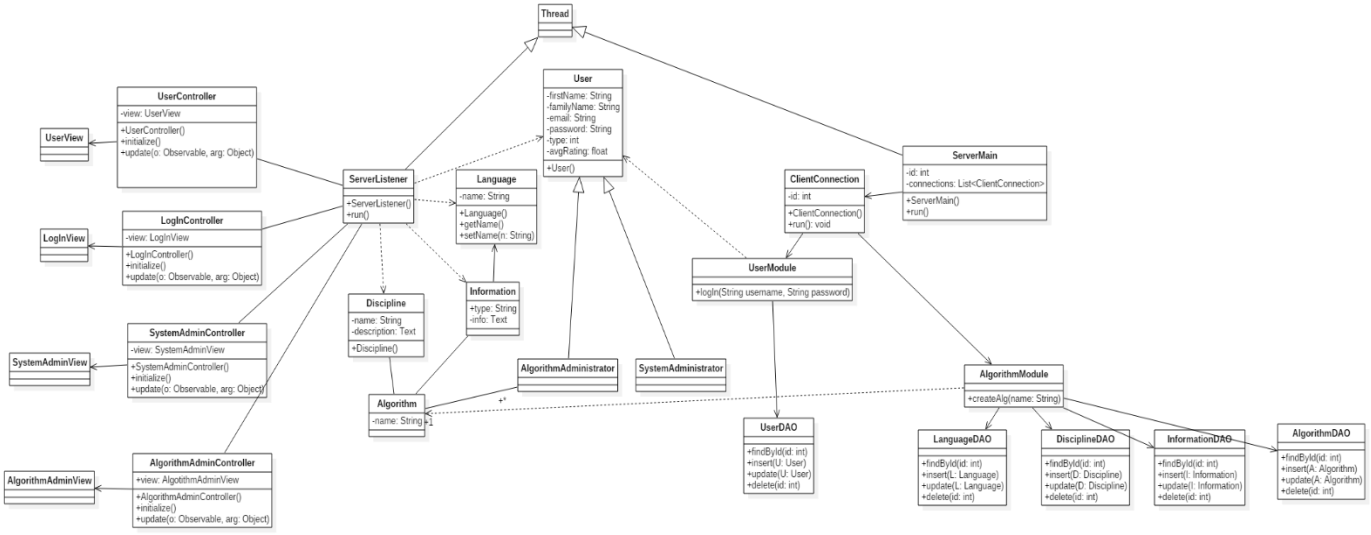
Sequence diagrams for administrator delete algorithm – Client side



Sequence diagram for administrator delete algorithm – Server side



1.2 Class Design



The main used design patterns are:

- MVC for client views
- DAO for persistence (database access)

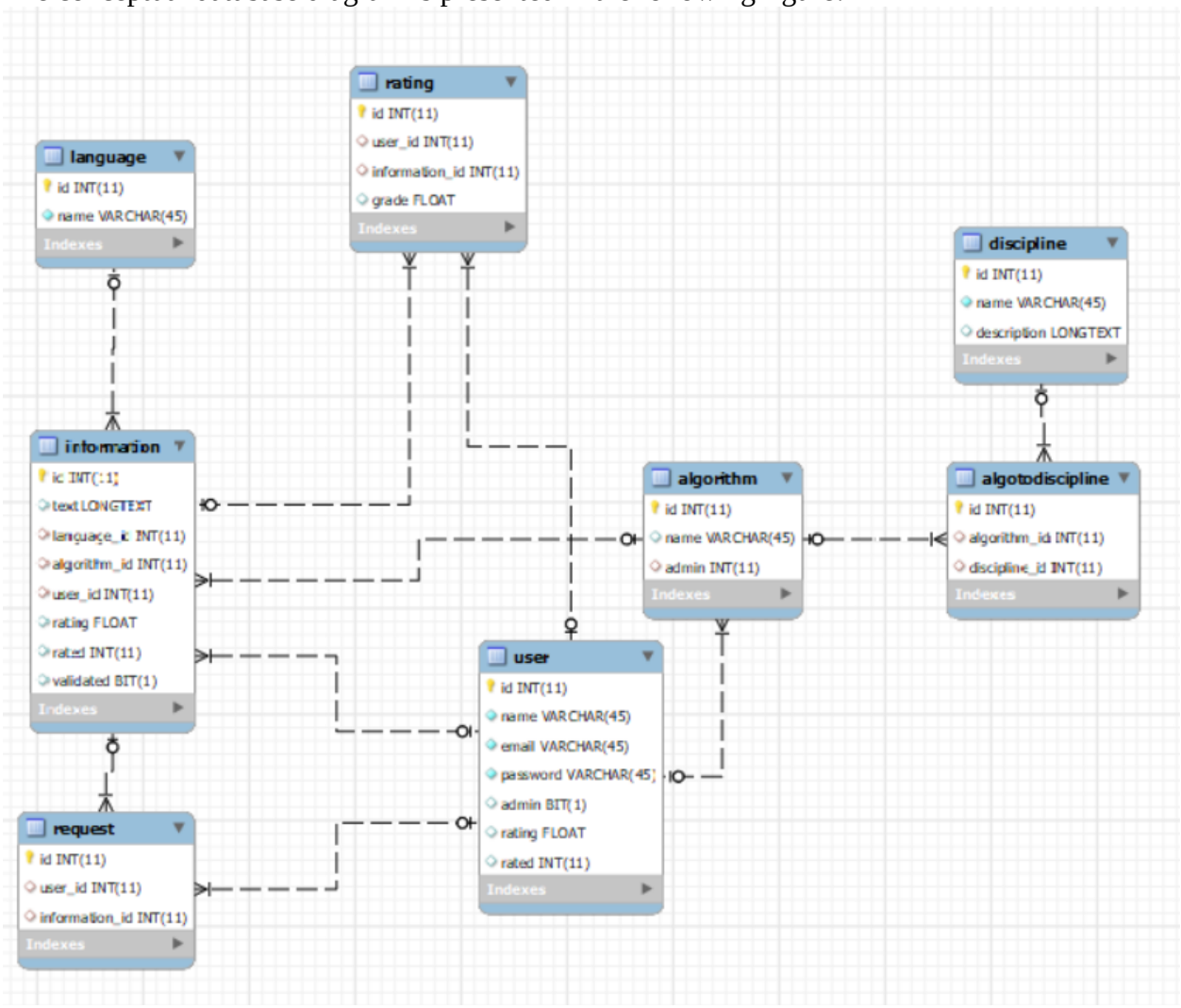
2. Data Model

The key concepts in the domain model are:

- User – regular, algorithm administrator or administrator
- Algorithm
- Discipline
- Programming language
- Text information (theoretical or code).
- Rating
- Request

These will become the entities in the database, as well as the entity classes in data layer of the application.

The conceptual database diagram is presented in the following figure:



The many to many relationships are kept in separate tables via foreign keys. The database access will be performed via an integrating framework such as Hibernate.

3. Unit Testing

The following are some relevant testing scenarios that need to be considered when performing the application testing.

- Register with an email twice

Test that the application does not accept a user to provide an email that has been already provided by a different user. Ideally, the application must display an error message to the client (user) that attempts to register with this type of flaw.

- Delete a contributing user account

Test that the application maintains information in the algorithm pages from users that were deleted from the database. The application should display an "N/A" message in the field that specifies the user that contributed with the information.

- Delete an algorithm administrator

Test the application for the deletion of an algorithm administrator. The issue of this deletion can be solved in several ways: set a system administrator as algorithm administrator (however, they must be default administrators on all algorithm pages) or force the system administrator to set a new algorithm administrator upon the deletion of the current one.

- Rating management

Several tests must be performed for the rating facilities. An average rating must be updated after each relevant transaction. After one user rates an information, both the rating of the information and the average rating of the user must be updated.

- Administrator acceptance

Test for the information to be stored in the database only when the algorithm administrator accepts it. Test for the rejected ones to be deleted from the database.

- System administrator deletion

Test for the case when a system administrator attempts to delete himself. The system should not allow such a deletion since the entire application would be affected by the lack of any system administrators.

- Large information

Test for large pieces of information, such as long texts , long pieces of code etc. Check for the system ability to store and retrieve the information as quickly as allowed by a maximum delay margin.

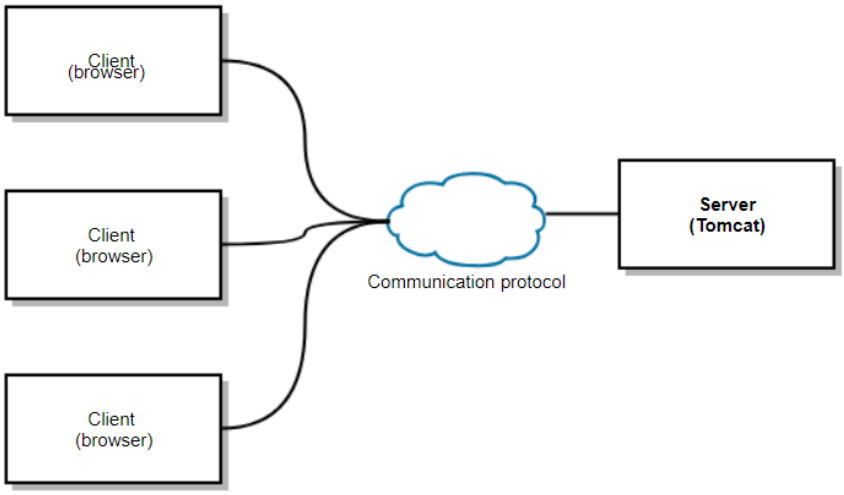
IV. Elaboration – Iteration 2

1. Architectural Design Refinement

The architectural design remains the same: Client-Server. However, the implementation relies on the Spring framework, therefore some key concepts can be displayed in the architectural design.

- Server – it receives requests in form of URIs of type server_address:port:URI. The server has a number of controllers which take care of satisfying these requests via services. Therefore, the server relies on controllers and services. Controllers are the components which take care of identifying the URI and calling the appropriate service methods such that the user request can be fulfilled.
- Client – it is represented by a browser application. Spring framework offers facilities for building up the server side of the application. Therefore, the client is already implemented by a third party (Google Chrome, Opera, Internet Explorer etc.). The browser is capable of receiving html information and displaying it.

In conclusion, the conceptual architecture does not change, but it is important to mention that the "Thin" client is represented by a browser.

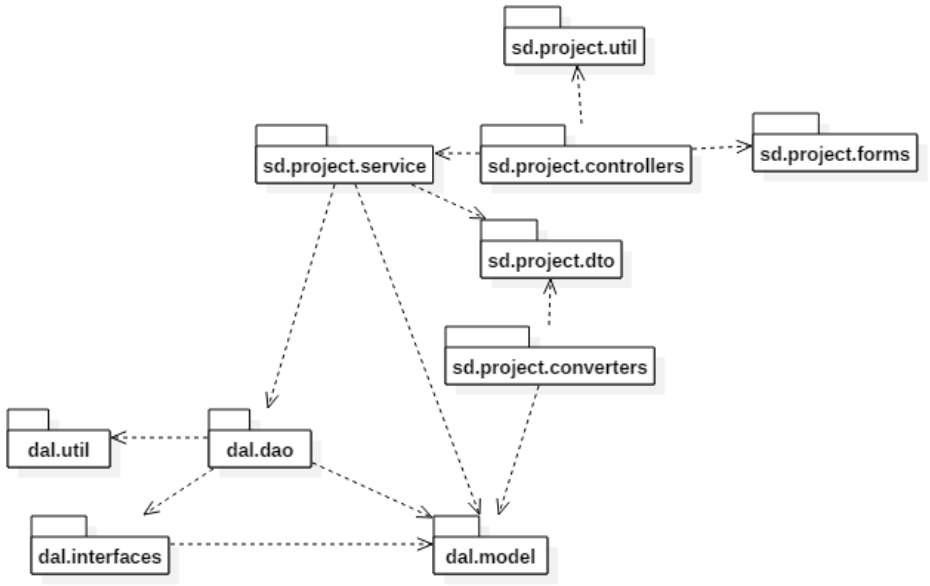


Because of the radical change in implementation (using a framework), the package diagram also needed to be changed.

There are two types of packages:

- Data access packages– they hold classes and interfaces related to persistence, such as DAO interfaces and objects, database connection settings (Hibernate based) and main model of the application (entities).
- Spring packages – they classes needed for constructing a Spring application, such as controllers and services. Moreover, this package also holds the DTOs (data transfer objects) needed for communication between java code and html. Besides these objects, packages for HTML form control are also used, as well as a util package for email construction, configuration and sending.

The package design is validated by the lack of cycles in the direct graph. Therefore, the direct graph constructed by the packages and their dependencies is a DAG (direct acyclic graph), which thoroughly respect the responsibility assignment principle.
Moreover, in the original design the DAO objects did not commit to any type of interface. In order to convey to SOLID principles (especially the dependency inversion principle), a package for DAO interfaces was provided. Therefore, the lower level module of the spring application which relies on the data access module, will not depend on the actual implementation, but rather on the abstract representation of the DAO functionality.



2. Design Model Refinement

The class diagram changes drastically due to the Spring-based implementation imposed. Although the listener and responding threads are no longer of concern when using the Spring framework, the key concepts remain the same.

The entity model is indeed the same, but this time we talk about two representations of the entities:

- Domain model – entities inferred by the database. Through them, basic CRUD operations are performed on the database. By using Hibernate framework, these entities also declare the relations between them through annotations.
- DTO – entities passed between the view and the business. The view needs to display some characteristics of the entities, but not all. Therefore, the DTOs will not keep the redundant relations imposed by Hibernate. For example, in a one-to-many mapping, the DTO will keep only the one element needed in the left side of the relation and eliminate the list required in the right side of the relation.

Two additional design patterns were implemented:

- **Façade design pattern**

This DP focuses on providing a unified interface for a set of interfaces or implementations. Therefore, Façade is able to provide one class through which other components can call all the methods of the interface set.

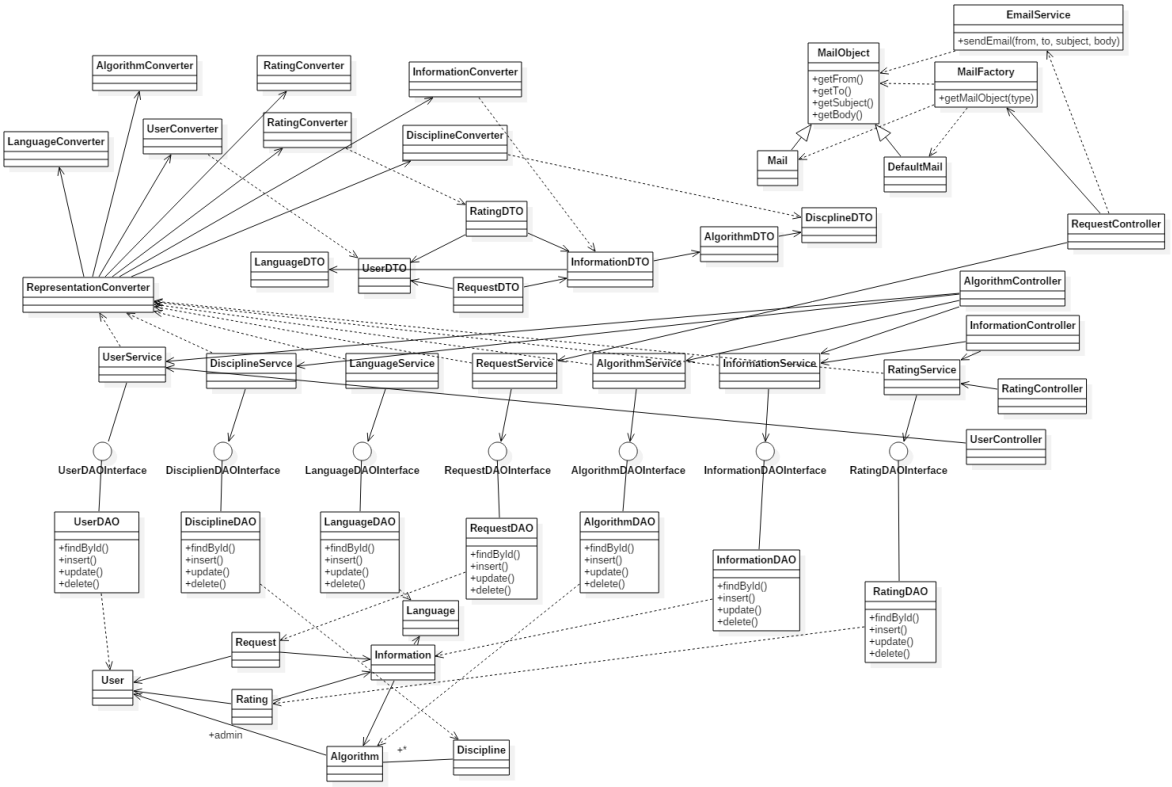
Usage: This design pattern was used for providing a single class able to convert between entity-DTO representations. Although each entity has associated a converter class, the class *RepresentationConverter* provides methods for all transformations by calling the methods of the dedicated classes.

On the class diagram, this design pattern is illustrated by the association between all the converter classes and the main static class which is used by all the services for conversions.

- **Factory method design pattern**

This DP focuses on providing methods for object-construction based on some selector chosen by the caller (the object which calls the method). It usually works with inheritance (class-based or interface implementation) by declaring the return type of the factory method as an abstract and returning objects of subclasses of that abstract class/interface.

Usage: This design pattern was used for differentiating between two types of emails: the default email and the normal email. A factory class for constructing such type of emails was used. The factory method returns an object of type MailObject and this class has as children (subclasses) the two specified type of mails: *Mail* and *DefaultMail*



V. Construction and Transition

1. System Testing

The system testing was performed using the most relevant scenarios. These scenarios were drawn from the system specification and the use cases.

First of all, the use cases were taken and tested. In order to assure that a use case test is passed, each use case test was split into more relevant cases. Of course, the testing helped in both correcting errors and understanding the system limitations.

Some of the most problematic tests regarded deletions. These tests yielded that some operations do not make sense, such as deleting a user after it provided some information. If one user is deleted from the system and he/she had contributed with something in the database, problems can arise. Since the contract of the domain model supposes the existence of a user provider for any type of information, this deletion does not make sense and may not be supported by the system in future releases. However, it was left in this version to showcase the implementation of all CRUD operations on the basic resources.

The most relevant testing scenarios were:

- CRUD operations on users.
- CRUD operations on algorithms, languages and disciplines.
- Rating submission (testing for correct updating of over-all rating of both users and information).
- Request submission with subcases: request accepted and request denied.
- Email sending with cases: default mail and simple mail. It is very important to note that this testing scenario uncovered a problem of Gmail regarding third-party software mail senders. In order to receive such an email from the Spring application, some properties of the Gmail account needed to be changed (properties regarding security and acceptance of third-party email senders).
- GUI extensive testing
  This testing brought up the most problems. They can also be seen as the "R" from the CRUD. One of the main purposes of the graphical user interface is to be the means of communicating the user what the database looks like (of course, in a refined manner and allowing specific type of users to see specific content).
  The most problems arose with transitions between html pages and, as unlikely as it might have sounded before, for operations that keep the user on the same page.
  For example, after the user logs in (the basic user/algorithm administrator), he/she can search the algorithms by name, discipline and administrator. Major issues arose when the content of the table which displayed the searching results changed.

2. Future improvements

The application has a lot of problems regarding the user interface. Besides being a little non-attractive due to my inexperience with HTML, the functionality is also suffering out of this inexperience.
For example, direct links to other pages could have been more easily implemented using hyperlink or table button clicks. However, dealing with such events supposes knowledge about Javascript or JQuery which is beyond the purpose of this project. Moreover, the log in page suffers of the displaying of the password. Normally, the password should be covered by a string of "*", as all applications these days provide this functionality.
Regarding functional improvements, some aspects could be significantly improved:

1. **Rating improvement**
   The rating system is as easy as it sounds. One user is able to rate an information only one time and the specific information/user ratings are updated. However, many websites and applications allow users to change their mind about ratings. This is not hard to implement due to the fact that ratings are kept in a separate table. Moreover, the functionality provided by the application in the current state could deal with the lack of such an explicit rating table by just updating the rating fields of users/information. The idea of the many-to-many table was to provide such a "Command"-like structure of rating management. Rating could be done and "undone".

2. **Extending algorithm administrator capabilities**
   The algorithm administrator has only one additional capability besides the regular user up to this point. Basically, he allows information to be submitted on its algorithms. However, information could be more flexible through updating/deletion. The deletion process may destroy the rating system and could arise difficult matters such as "What happens with the rating of the submitting user if an information is deleted?". Of course, solutions do exist for dealing with such situations.

3. **User communication**
   If we were to take a look at some websites which deal with providing such algorithm implementations (a good example would be Stack Overflow), the users are able to comment on the solutions and even ask questions. Therefore, each information page could provide fields for inter-user communication. This communication could be easily done via email messages as well. However, the website could provide a window for discussion on each information.

4. **Page updating without refresh**
   A more challenging task would be to enable the users to see modifications in real time. For example, if one user were to be on an algorithm page while an information is just being accepted by the algorithm administrator, he should be able to see the table updating in real time.
   In order to provide such a functionality, the Observer design pattern could be used, as well as web sockets.

VI. Bibliography

1. Spring boot
   https://projects.spring.io/spring-boot/
   https://spring.io/guides/gs/spring-boot/
   http://www.mkyong.com/tutorials/spring-boot-tutorials/

2. Thymeleaf
   http://www.mkyong.com/spring-boot/spring-boot-hello-world-example-thymeleaf/
   https://www.thymeleaf.org/doc/tutorials/2.1/usingthymeleaf.html

3. HTML
   https://www.w3schools.com/html/

4. CSS
   https://www.w3schools.com/Css/

5. Hibernate
   https://www.tutorialspoint.com/hibernate/index.htm
   https://www.journaldev.com/3793/hibernate-tutorial

6. Various problems
   http://forum.thymeleaf.org/
   http://forum.spring.io/forum/spring-projects/boot
   https://stackoverflow.com/