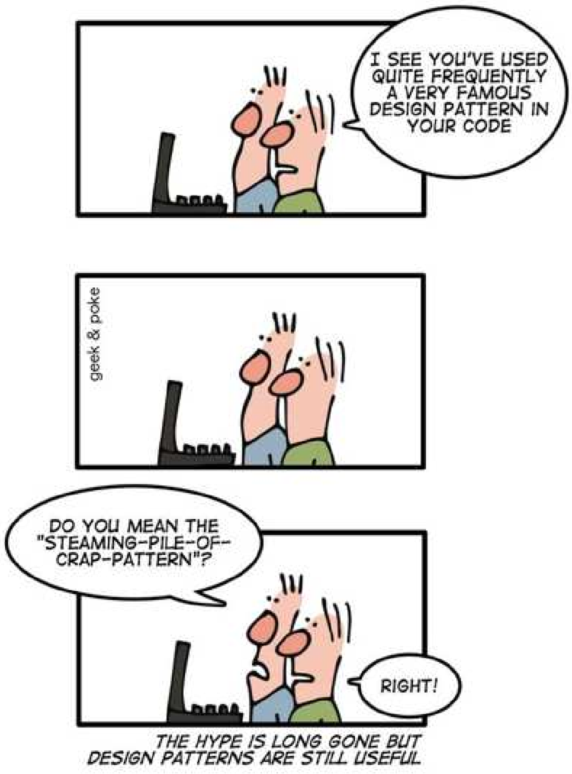
Diploma Projects Management App

Guidelines and Hints for the development of the project


I SEE YOU'VE USED QUITE FREQUENTLY A VERY FAMOUS DESIGN PATTERN IN YOUR CODE
geek & poke
DO YOU MEAN THE "STEAMING-PILE-OF-CRAP-PATTERN"?
RIGHT!
THE HYPE IS LONG GONE BUT DESIGN PATTERNS ARE STILL USEFUL

1 Introduction

This document provides some basic but important guidelines for the development of the project. We begin with general development tips that should be followed. Then, we provide design directions concerning the project.

2 General Guidelines - DOs and DON'Ts

- Classes
  - ✓ Make **classes small** and cohesive - A single well defined responsibility for a class
  - ✓ Don't break **encapsulation** by making the data representation public
  - ✓ **Class names** are important – use descriptive names for the concepts represented by the classes
  - ✓ Use **Noun & Noun phrases** for class names
  - ✓ **See here for more - http://www.cs.uoi.gr/~zarras/soft-devII.htm**
- Methods
  - ✓ Make **methods small** – A method must **do one thing**
  - ✓ **Method names** are important – use descriptive names for the concepts represented by the methods
  - ✓ Use **Verb & Verb phrases** for method names
  - ✓ **See here for more - http://www.cs.uoi.gr/~zarras/soft-devII.htm**
- Fields
  - ✓ Make **fields private** – A method must do one thing
  - ✓ **Field names** are important – use descriptive names for the concepts represented by the fields
  - ✓ Use **Noun & Noun phrases** for field names
- For naming see here **http://www.cs.uoi.gr/~zarras/soft-devII-notes/2-meaningful-names.pdf**
- Follow the standard Java Coding Style **http://www.cs.uoi.gr/~zarras/soft-devII-notes/java-programming-style.pdf**

3 Application Design and Related Design Patterns

3.1 Architecture

To facilitate the maintenance and enable future extensions of the application we assume an architecture that conforms with the **layered architectural style (see the lecture slides on Software Design)**. Maintainability and extensibility are further promoted by the fact that the **Spring framework** heavily relies on the **Model View Controller (MVC) pattern (see the lecture slides on Software Design)** for the development of Web applications. MVC allows the clear separation of three different concerns: the view of the application that is responsible for the user interaction (UI) with the application, the domain model that represents the data handled by the application and the business logic, and the controller that takes user input, manipulates the domain model data and updates the view.

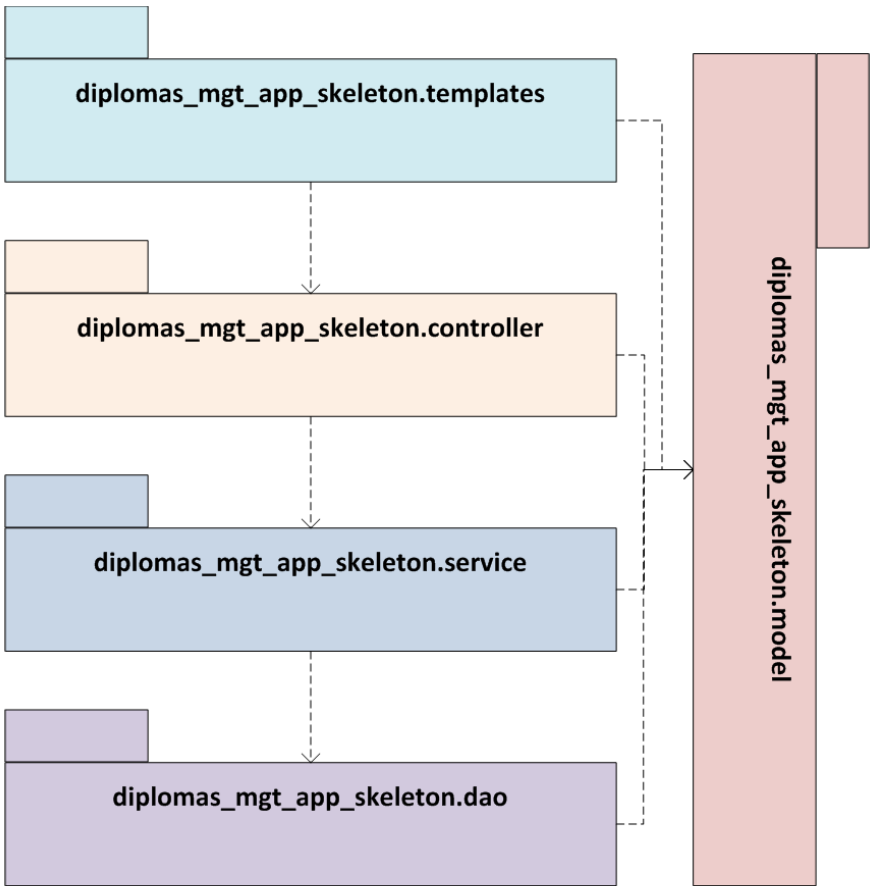

diplomas_mgt_app_skeleton.templates
diplomas_mgt_app_skeleton.controller
diplomas_mgt_app_skeleton.service
diplomas_mgt_app_skeleton.dao
diplomas_mgt_app_skeleton.model

FIGURE 1 APPLICATION ARCHITECTURE.

The following list describes briefly the basic components of the architecture (Figure 1):

- The core of the proposed architecture is the **model**. The model consists of the basic classes that define the representation of the data handled by the application and the domain logic that is needed for the data manipulation. All the different layers of the application rely on the data model. Essentially, here we apply the **Domain Model pattern** from Martin Fowler's catalog of **Enterprise Application Architecture (EAA) patterns** (see https://martinfowler.com/eaaCatalog/).

- The first layer of the proposed architecture is the **Data Access Objects (DAO) layer.** This layer consists of classes that perform basic database operations which map the application data, stored in the table rows of the database to corresponding in memory objects of the domain model classes. Basically, here we apply the **Data Mapper pattern** from **Martin Fowler's catalog of EAA patterns**.

- The second layer of the proposed architecture is the **Service layer** which defines a set of services provided by the application to the users and coordinates the application's response in each service operation. Essentially, here we apply the **Service Layer pattern** from Martin Fowler's catalog of EAA patterns. Often many of the provided service operations correspond directly to respective DAO operations. However, the services may also provide more complex operations that involve several DAO and domain objects.

- The third layer of the application is the **Controller layer**. This layer consists of one or more controller classes which take user input from the view of the application, manipulate domain model objects, and update the view of the application. Hence, here we apply the well-known **MVC pattern**.

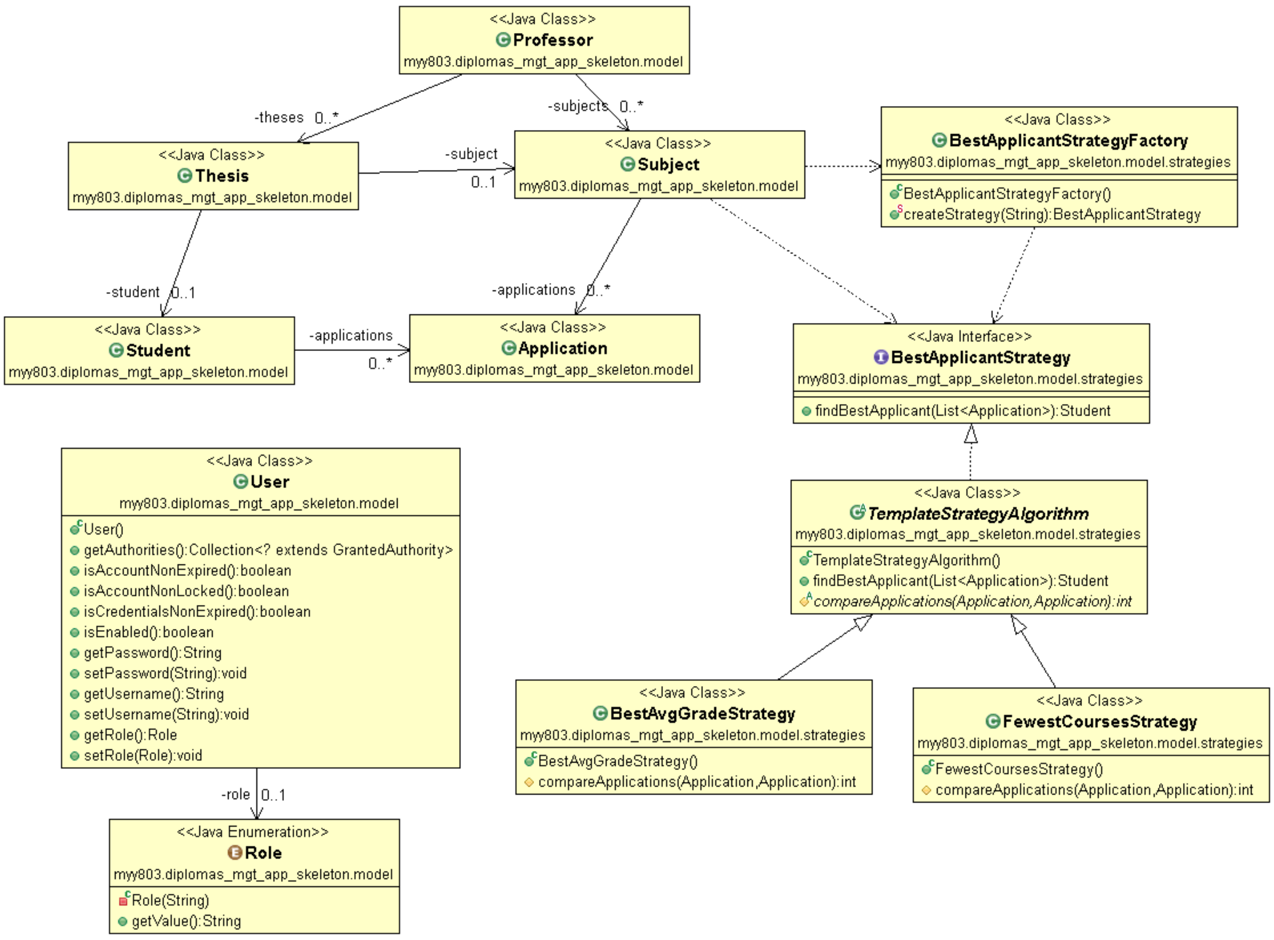
- Last, the top layer of the application is the **View layer** that realizes the user interaction (UI) with the application in collaboration with the Controller layer. Typically, in a Web based enterprise application this layer consists of a set of static and dynamic Web pages. The dynamic Web pages are also known as template views (see **Template View pattern** from Martin Fowler's catalog of EAA patterns). A dynamic HTML page renders data from domain model objects into HTML based on embedded markers. The application controller(s) is (are) responsible for passing the appropriate domain model objects to the view layer.

3.2 Model

A domain model that can be used to satisfy the requirements of the application is given in Figure 2. User and Role are specific classes we have to implement for the realization of the user registration and login actions. User should implement the Spring UserDetails interface for this reason (see the https://github.com/zarras/myy803_springboot_web_app_tutorials/tree/master/sb_tutorial_7_signup_signin for more details).

Professor has a list of Subject objects that correspond to available diploma thesis subjects. To ease the management of Subject objects in the database this association can be bidirectional in the sense that a Subject object is characterized by the supervisor and a list of Application objects that correspond to the applications of students who are willing to take over the subject. Professor also has a list of Thesis objects that correspond to the diploma thesis subjects he actually supervises. Again to ease the management of Thesis subject this association can be bidirectional. Each Thesis object is characterized by the supervisor, the subject and different grades that serve for calculating the total grade of the student. Student has a list of Application objects for subjects that the student is interested in. The student is also characterized by profile information that serves for the evaluation of the student applications.

The database schema of the application should define corresponding tables for the classes of the application with foreign keys that correspond to the class associations.

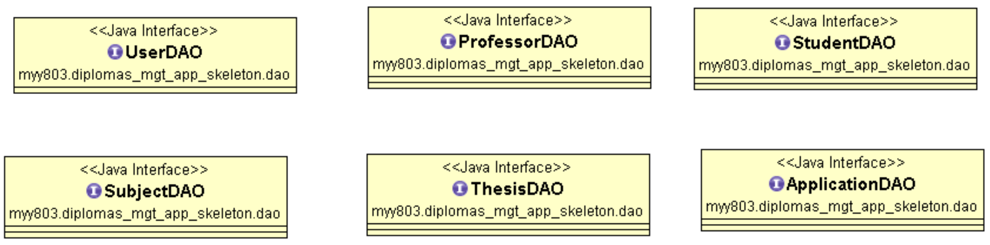


<<Java Class>>
Professor
myy803.diplomas_mgt_app_skeleton.model
-theses 0..*
-subjects 0..*
<<Java Class>>
Thesis
myy803.diplomas_mgt_app_skeleton.model
-subject
0..1
<<Java Class>>
Subject
myy803.diplomas_mgt_app_skeleton.model
<<Java Class>>
BestApplicantStrategyFactory
myy803.diplomas_mgt_app_skeleton.model.strategies
BestApplicantStrategyFactory()
createStrategy(String):BestApplicantStrategy
-student 0..1
-applications 0..*
<<Java Class>>
Student
myy803.diplomas_mgt_app_skeleton.model
-applications
0..*
<<Java Class>>
Application
myy803.diplomas_mgt_app_skeleton.model
<<Java Interface>>
BestApplicantStrategy
myy803.diplomas_mgt_app_skeleton.model.strategies
findBestApplicant(List<Application>):Student
<<Java Class>>
User
myy803.diplomas_mgt_app_skeleton.model
User()
getAuthorities():Collection<? extends GrantedAuthority>
isAccountNonExpired():boolean
isAccountNonLocked():boolean
isCredentialsNonExpired():boolean
isEnabled():boolean
getPassword():String
setPassword(String):void
getUsername():String
setUsername(String):void
getRole():Role
setRole(Role):void
<<Java Class>>
TemplateStrategyAlgorithm
myy803.diplomas_mgt_app_skeleton.model.strategies
TemplateStrategyAlgorithm()
findBestApplicant(List<Application>):Student
compareApplications(Application,Application):int
<<Java Class>>
BestAvgGradeStrategy
myy803.diplomas_mgt_app_skeleton.model.strategies
BestAvgGradeStrategy()
compareApplications(Application,Application):int
<<Java Class>>
FewestCoursesStrategy
myy803.diplomas_mgt_app_skeleton.model.strategies
FewestCoursesStrategy()
compareApplications(Application,Application):int
-role 0..1
<<Java Enumeration>>
Role
myy803.diplomas_mgt_app_skeleton.model
Role(String)
getValue():String

FIGURE 2 DOMAIN MODEL (THESE CLASS DIAGRAMS ARE ONLY AN INCOMPLETE SKETCH THAT SHOULD BE FURTHER ELABORATED BY EACH TEAM).

3.3 Data Access Objects Layer

The DAO layer for the model of the application is illustrated in Figure 2. For each class the DAO layer defines a respective JPA repository that facilitates the retrieval and the storage of model object from and to the underlying database.

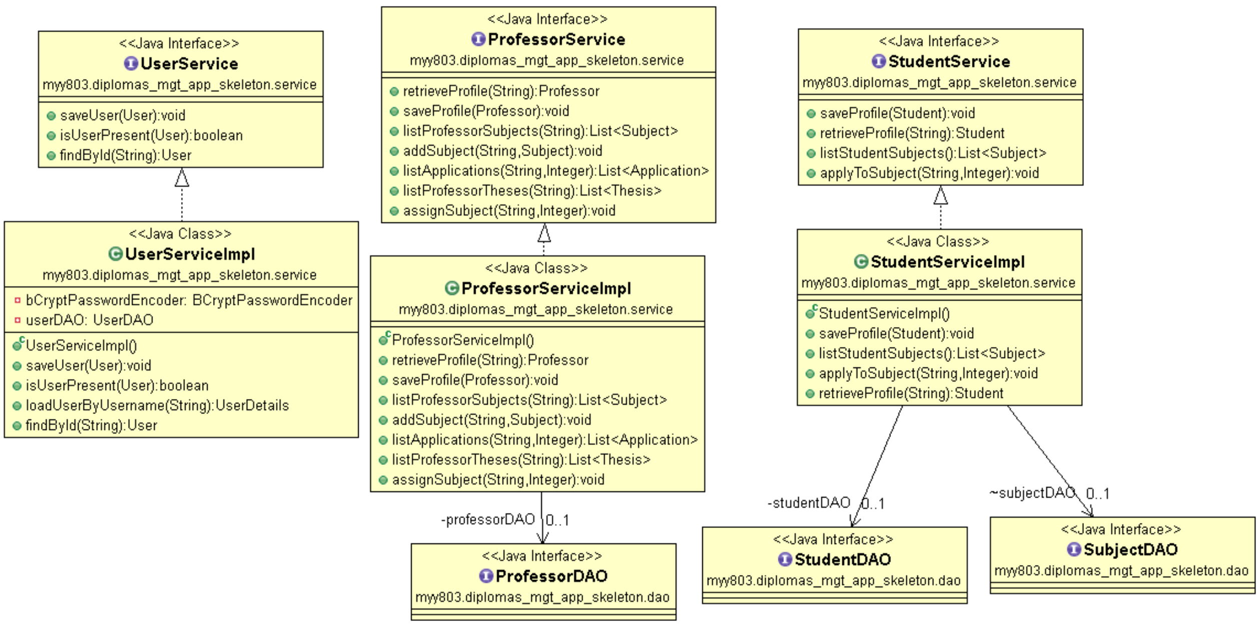


<<Java Interface>>
UserDAO
myy803.diplomas_mgt_app_skeleton.dao
<<Java Interface>>
ProfessorDAO
myy803.diplomas_mgt_app_skeleton.dao
<<Java Interface>>
StudentDAO
myy803.diplomas_mgt_app_skeleton.dao
<<Java Interface>>
SubjectDAO
myy803.diplomas_mgt_app_skeleton.dao
<<Java Interface>>
ThesisDAO
myy803.diplomas_mgt_app_skeleton.dao
<<Java Interface>>
ApplicationDAO
myy803.diplomas_mgt_app_skeleton.dao

FIGURE 3 DAO LAYER (THESE CLASS DIAGRAMS ARE ONLY AN INCOMPLETE SKETCH THAT SHOULD BE FURTHER ELABORATED BY EACH TEAM).

3.4 Service Layer

In a similar vein, the Service layer consists of interfaces, that define the operations of the services that are provided by the application to the users. The interfaces are realized by corresponding implementation classes**.**



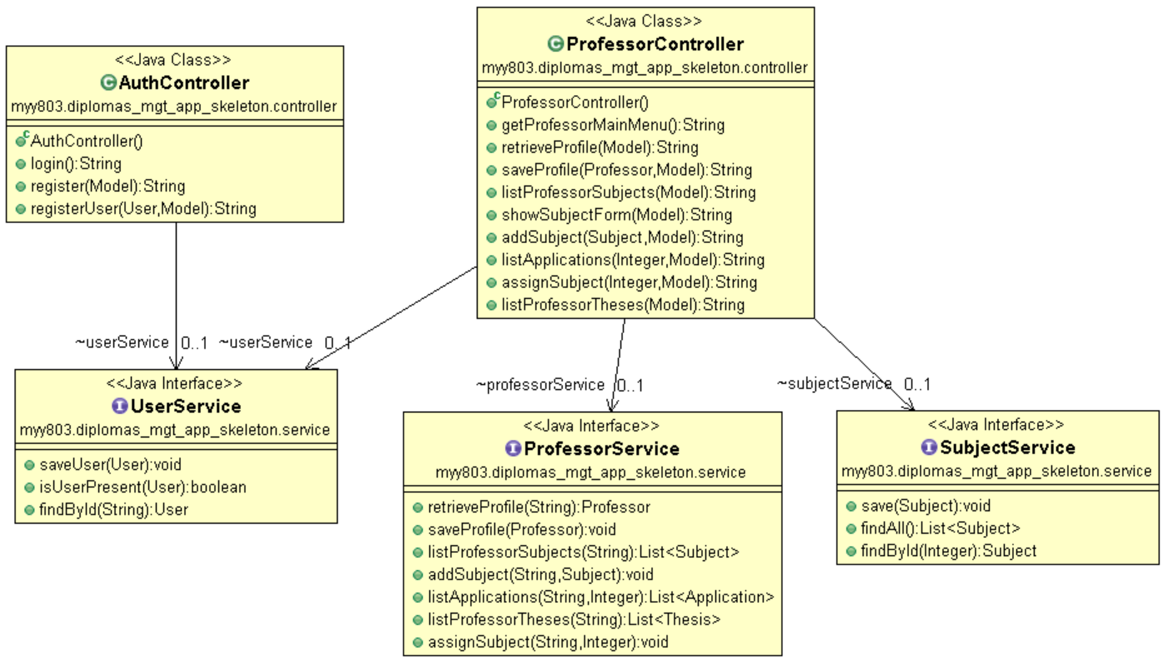
<<Java Interface>>
UserService
myy803.diplomas_mgt_app_skeleton.service
saveUser(User):void
isUserPresent(User):boolean
findById(String):User
<<Java Class>>
UserServiceImpl
myy803.diplomas_mgt_app_skeleton.service
bCryptPasswordEncoder: BCryptPasswordEncoder
userDAO: UserDAO
UserServiceImpl()
saveUser(User):void
isUserPresent(User):boolean
loadUserByUsername(String):UserDetails
findById(String):User
<<Java Interface>>
ProfessorService
myy803.diplomas_mgt_app_skeleton.service
retrieveProfile(String):Professor
saveProfile(Professor):void
listProfessorSubjects(String):List<Subject>
addSubject(String,Subject):void
listApplications(String,Integer):List<Application>
listProfessorTheses(String):List<Thesis>
assignSubject(String,Integer):void
<<Java Class>>
ProfessorServiceImpl
myy803.diplomas_mgt_app_skeleton.service
ProfessorServiceImpl()
retrieveProfile(String):Professor
saveProfile(Professor):void
listProfessorSubjects(String):List<Subject>
addSubject(String,Subject):void
listApplications(String,Integer):List<Application>
listProfessorTheses(String):List<Thesis>
assignSubject(String,Integer):void
-professorDAO
0..1
<<Java Interface>>
ProfessorDAO
myy803.diplomas_mgt_app_skeleton.dao
<<Java Interface>>
StudentService
myy803.diplomas_mgt_app_skeleton.service
saveProfile(Student):void
retrieveProfile(String):Student
listStudentSubjects():List<Subject>
applyToSubject(String,Integer):void
<<Java Class>>
StudentServiceImpl
myy803.diplomas_mgt_app_skeleton.service
StudentServiceImpl()
saveProfile(Student):void
listStudentSubjects():List<Subject>
applyToSubject(String,Integer):void
retrieveProfile(String):Student
-studentDAO
0..1
~subjectDAO
0..1
<<Java Interface>>
StudentDAO
myy803.diplomas_mgt_app_skeleton.dao
<<Java Interface>>
SubjectDAO
myy803.diplomas_mgt_app_skeleton.dao

FIGURE 4 SERVICE LAYER (THESE CLASS DIAGRAMS ARE ONLY AN INCOMPLETE SKETCH THAT SHOULD BE FURTHER ELABORATED BY EACH TEAM).

For the assignment of diploma thesis subjects to students with different alternative strategies we use the GoF Strategy pattern. To this end we define a general SubjectAssigmentStrategy interface tha has different implementations. SubjectServiceImpl has a list of SubjectAssigmentStrategy, one for each alternative strategy. This way we satisfy the **maintainability requirements** given in the application requirements document for the **easy extension of the application with new assignment strategies** in the future. Note that the different strategy implementations may also have some common steps, which give room for the use of the GoF Template Method pattern.

3.5 MVC- Controller & View Layers

This layer consists of different controllers for Student and Professor that provide access to the respective functionalities. The controllers use service layer objects for the manipulation of domain objects and the update of the views. For the realization of the View layer we can employ the Thymeleaf template engine to define template views.



<<Java Class>>
AuthController
myy803.diplomas_mgt_app_skeleton.controller
AuthController()
login():String
register(Model):String
registerUser(User,Model):String
<<Java Class>>
ProfessorController
myy803.diplomas_mgt_app_skeleton.controller
ProfessorController()
getProfessorMainMenu():String
retrieveProfile(Model):String
saveProfile(Professor,Model):String
listProfessorSubjects(Model):String
showSubjectForm(Model):String
addSubject(Subject,Model):String
listApplications(Integer,Model):String
assignSubject(Integer,Model):String
listProfessorTheses(Model):String
~userService 0..1
~userService 0..1
~professorService 0..1
~subjectService 0..1
<<Java Interface>>
UserService
myy803.diplomas_mgt_app_skeleton.service
saveUser(User):void
isUserPresent(User):boolean
findById(String):User
<<Java Interface>>
ProfessorService
myy803.diplomas_mgt_app_skeleton.service
retrieveProfile(String):Professor
saveProfile(Professor):void
listProfessorSubjects(String):List<Subject>
addSubject(String,Subject):void
listApplications(String,Integer):List<Application>
listProfessorTheses(String):List<Thesis>
assignSubject(String,Integer):void
<<Java Interface>>
SubjectService
myy803.diplomas_mgt_app_skeleton.service
save(Subject):void
findAll():List<Subject>
findById(Integer):Subject

FIGURE 5 CONTROLLER LAYER (THESE CLASS DIAGRAMS ARE ONLY AN INCOMPLETE SKETCH THAT SHOULD BE FURTHER ELABORATED BY EACH TEAM)

4 Tests

Concerning the tests of the application logic we need to develop the following:

1. A set of unit tests for the domain model classes.
2. A set of integration tests for the DAO layer.
3. A set of integration tests for the Service layer.

4. A set of acceptance tests for the Controller layer.

For the development of the tests we can rely on the SpringBoot testing and mocking facilities.