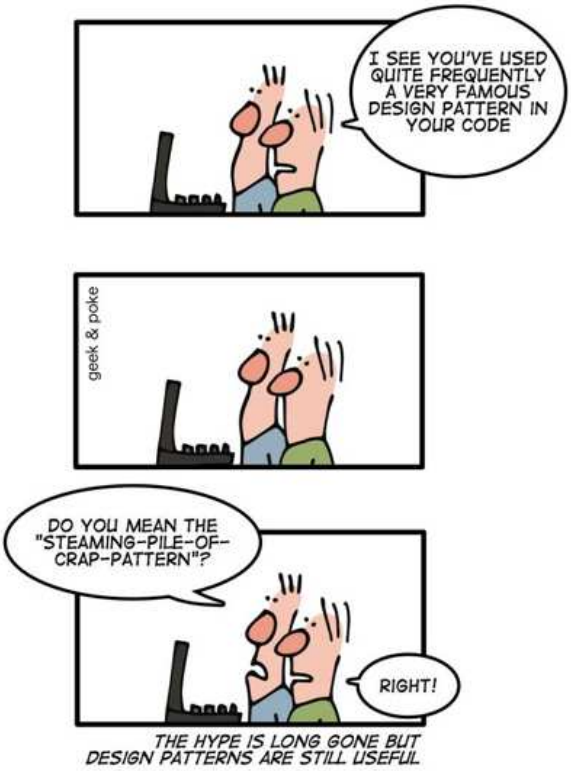
Courses Management App

Guidelines and Hints for the development of the project



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.

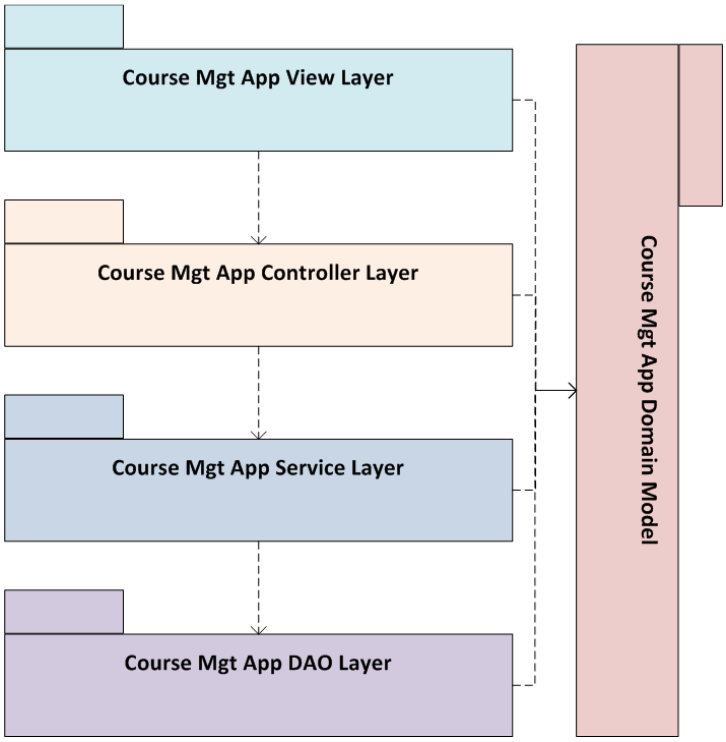


FIGURE 1 APPLICATION ARCHITECTURE.

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

- The core of the proposed architecture is the **domain model**. The domain 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.** The DAO 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**. DAO is a layer of Data Mappers that move data between domain objects and the database, while keeping them independent of each other and the mapper itself.

- 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 Domain Model

A simple domain model that covers the needs of the application consists of two classes. The **Course** class defines the data representation and the domain logic for the management of courses, while the **StudentRegistration** class defines the data representation and the domain logic for the management of student registrations. The Course class has a number of attributes (see the **requirements document**) for the id, the name, the description of the course, the instructor of the course - for convenience **this can be the instructor's login name** - etc. Moreover, the Course class has a list of **StudentRegistration** objects that corresponds to a **one-to-many relation** between a course and the student registrations of the course.

The **database schema** of the application should define corresponding tables, **courses**, **student_registrations**. The student_registrations table has a foreign key to the primary key of the courses table, so as to realize the one-to-many relation between a course and the student registrations of the course.

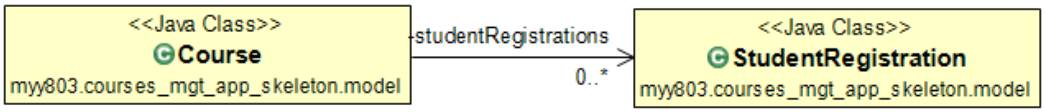


FIGURE 2 DOMAIN MODEL (THESE CLASS DIAGRAMS ARE ONLY AN INCOMPLETE SKETCH THAT SHOULD BE FURTHER ELABORATED BY EACH TEAM).

3.3 Data Access Objects Layer

The DAO layer consists of two interfaces, **CourseDAO** and **StudentRegistrationDAO**, that define the database operations which map domain objects to database table row data. The interfaces can have many different implementations for various underlying database management systems and technologies. In the first release of the application, we assume at least two implementations, **CourseDAOImpl** and **StudentRegistrationDAOImpl** that rely on **Spring Java Persistent Access (JPA) Hibernate and provide access to MySQL**. The rest of the application code and specifically the Service layer uses the DAO interfaces for accessing the application database. The fact that the **Service layer is independent from the specific implementations of the DAO interfaces** allow to easily **extend the application** with **other database management systems and technologies** in the future and use them interchangeably. Essentially, here we employ the **GoF Strategy pattern (see the lecture slides on Software Design)**. This way we satisfy the **maintainability requirements** given in the application requirements document.

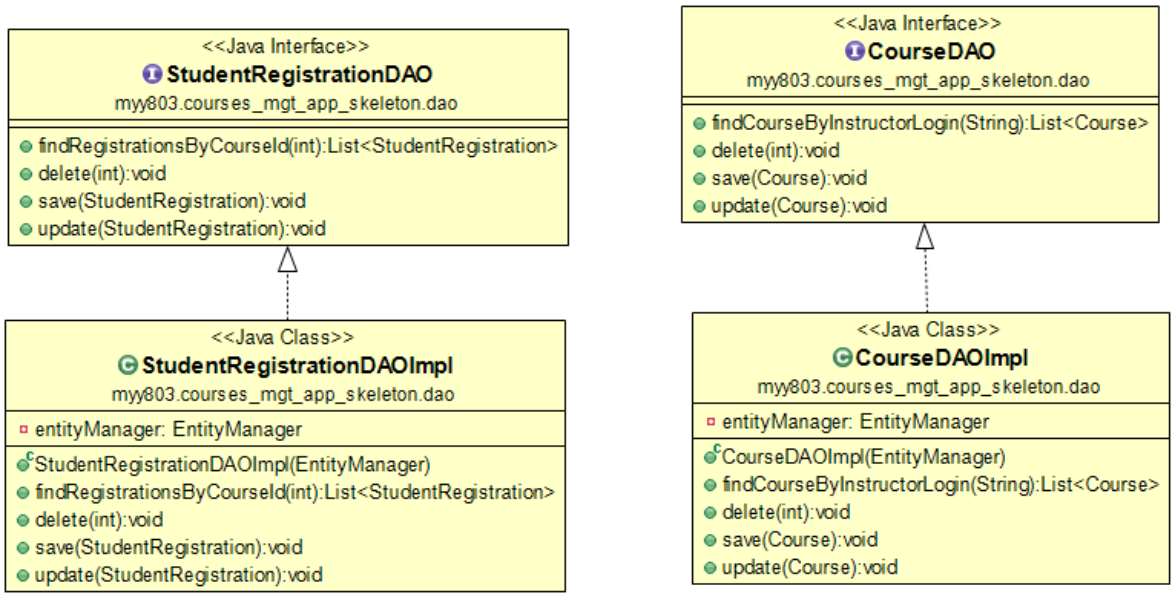


FIGURE 3 DAO LAYER (THESE CLASS DIAGRAMS ARE ONLY AN INCOMPLETE SKETCH THAT SHOULD BE FURTHER ELABORATED BY EACH TEAM).

3.4 Service Layer

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

For the calculation of course statistics, the **CourseServiceImpl** class has **a list of Statistic objects. Statistic** is an interface that can have different implementations one for each statistic (e.g. mean, max, min, skewness, etc.) Again here we employ the **GoF Strategy pattern**. This way we satisfy the **maintainability requirements** given in the application requirements document for the **easy extension of the application with new statistics** in the future.

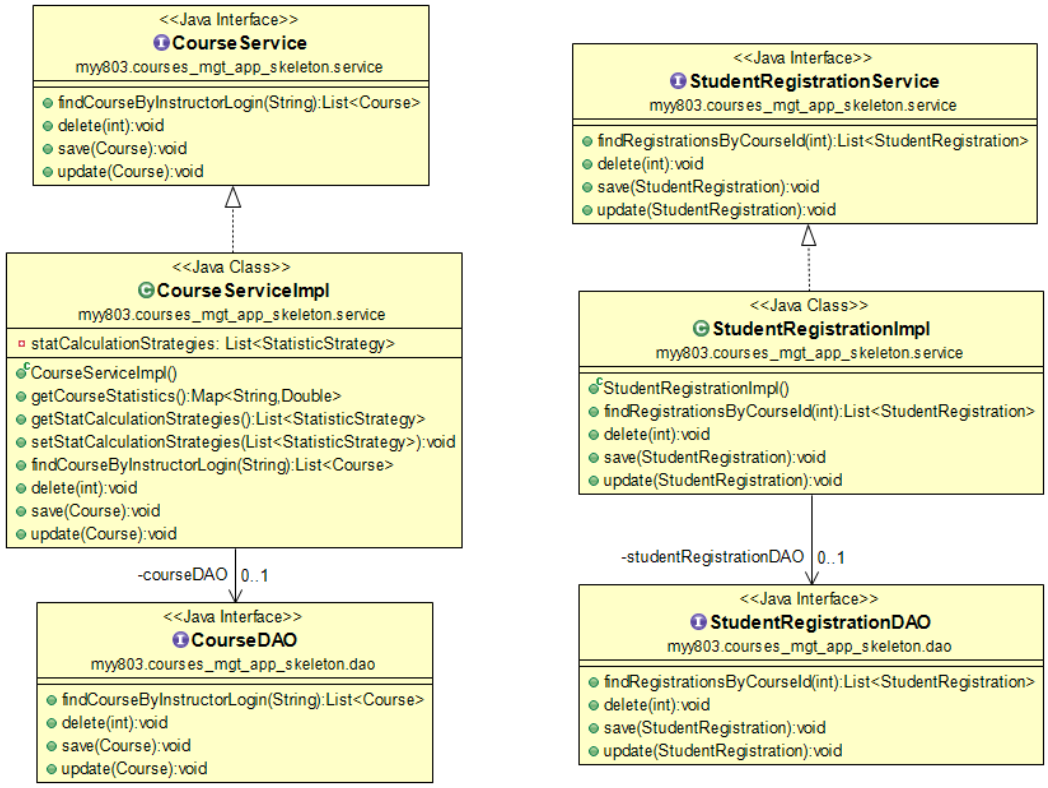


FIGURE 4 SERVICE LAYER (THESE CLASS DIAGRAMS ARE ONLY AN INCOMPLETE SKETCH THAT SHOULD BE FURTHER ELABORATED BY EACH TEAM).

In the first release of the application, we assume that the classes that implement the Statistic interface are based on the **Apache Math API**. The core class of the Apache Math API is called **DescriptiveStatistics.** In a sense, each implementation of the Statistic interface is a **GoF Adapter (see the lecture slides on Software Design)** for the **DescriptiveStatistics** class**.** Specifically, each class that implements the Statistic interface uses a **DescriptiveStatistics object** to **create a dataset** that contains the students grades and then calculate the corresponding statistic. So, the statistics calculation process has **a common part** for all the different Statistic implementations. To avoid duplicate code in the classes that implement the Statistic interface we can employ the **TemplateMethod** GoF pattern **(see the lecture slides on Software Design)**. Specifically, the idea here is to define a common **abstract class**

TemplateStatistic for all the classes that implement the Statistic interface. The template method in the abstract class creates the data set and then calls an abstract method that calculates the statistic. The abstract method is concretely defined in each concrete subclasses of the abstract class to calculate the corresponding statistic.

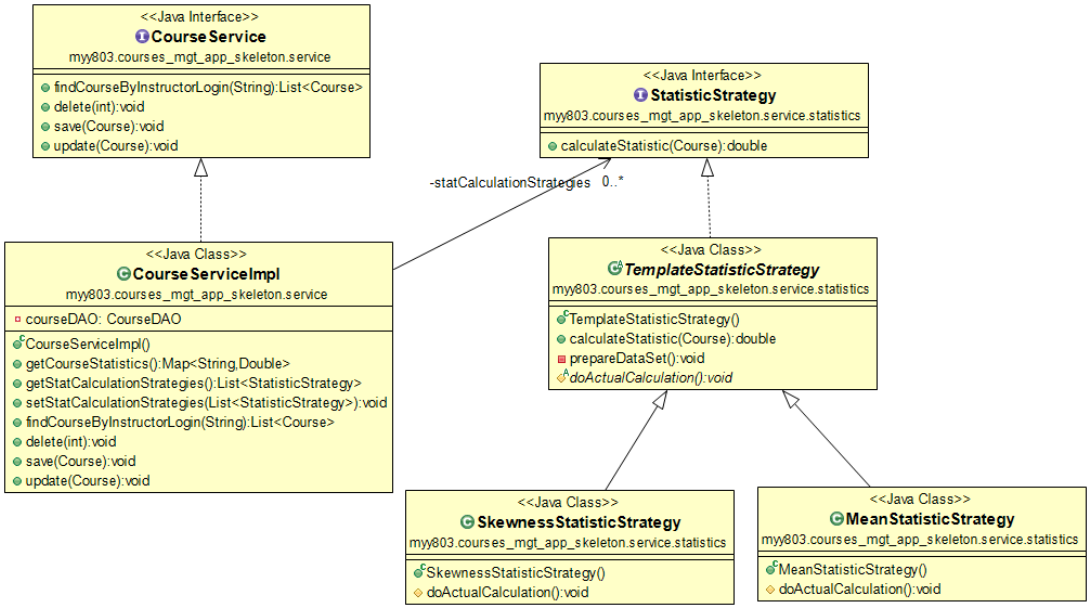


FIGURE 5 SERVICE LAYER 2 (THESE CLASS DIAGRAMS ARE ONLY AN INCOMPLETE SKETCH THAT SHOULD BE FURTHER ELABORATED BY EACH TEAM).

3.5 MVC- Controller & View Layers

In the simplest case, the Controller layer consists of a controller class **CourseMgtAppController** that uses 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 for the manipulation of courses and student registrations. There is a need for at least two templates for showing and manipulating courses and registrations and two forms for the addition/update of new courses and registrations.

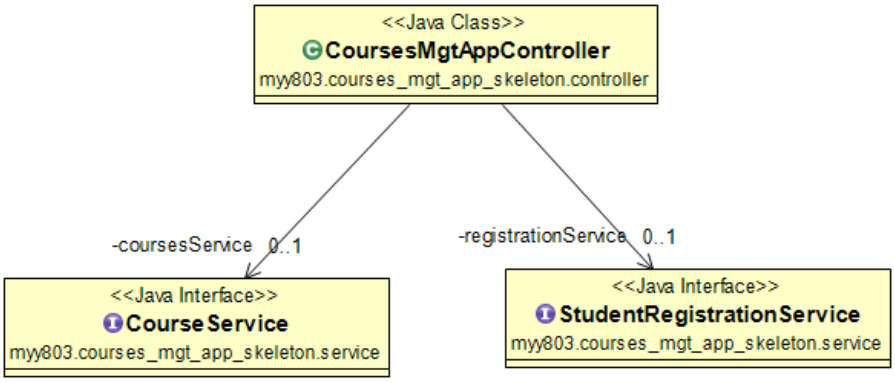


FIGURE 6 CONTROLLER LAYER (THESE CLASS DIAGRAMS ARE ONLY AN INCOMPLETE SKETCH THAT SHOULD BE FURTHER ELABORATED BY EACH TEAM)

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.