Case Study

We will apply the UML concepts that we will be discussing through the coming weeks and design an entire real world application. Each session in the coming weeks will be rounded off with designing the case study application incrementally using each of the UML diagrams.

For our case study, we will be the architects assigned the task of constructing the design elements for a system that can be used to manage coursees/classes for an organization that specializes in providing training. Let us name the system that we will be designing as the Courseware Management System. The organization offers a variety of courses in a variety of areas such as learning management techniques and understanding different software languages and technologies. Each course is made up of a set of topics. Tutors in the organization are assigned courses to teach according to the area that they specialize in and their availability. The organization publishes and maintains a calendar of the different courses and the assigned tutors every year. There is a group of course administrators in the organization who manage the courses including course content, assign courses to tutors, and define the course schedule. The training organization aims to use the Courseware Management System to get a better control and visibility to the management of courses as also to streamline the process of generating and managing the schedule of the different courses.

Now that we have our problem statement defined, we can proceed to the next step—analyzing and elaborating on the requirements and then designing the Courseware Management System in the coming weeks.

Use Case Diagram:

Analyze the problem statement to identify the potential actors and use cases of the system. First, let us list the potential actors. A quick look at the problem statement shows up the following terms and entities specific to the system:

- Courses and Topics that make up a course
- Tutors who teach courses
- Course administrators who manage the assignment of the courses to tutors
- Calendar or Course Schedule is generated as a result of the
- Students who refer to the Course schedule or Calendar to decide which courses they wish to take up for study

**Identifying Actors of the Courseware Management System**

Out of the preceding list, one thing is clear. There are certain terms and entities in the list that identify that they perform certain roles or business processes. We will discuss what these business processes are after we complete our analysis for identifying actors. For now, focus on identifying the actors in the system. From the preceding list, you can see that there are some entities that perform an action and some that form the target for the action. The entities that perform action will be the actors for the Courseware Management System. In the above list, the actors that we can identify are:

- Tutors
- Course administrators
- Students

But, because students are not the potential active participants for this system, we will drop them from the list of actors. Similarly, tutors are not active participants from our system’s perspective, and hence, we will exclude tutors from our list if roles. Yet, we will still record them in our use case model since we do not wish to lose this business information. Our final list of primary actors has now come down to only one:

- Course administrators

**Identifying Use Cases of the Courseware Management System**

Next, let us identify the potential business processes in the Courseware Management System. The primary business flows in the system are:

- Manage courses
- Manage course assignments

As we analyze the problem statement further, we can determine some discrete processes within these primary business flows. To manage courses, the actor needs to have the ability to view existing courses, manage the course information for a course, such as duration and so forth, and also manage the
addition or removal of topics for a course. So, within the "Manage courses" use case, we can identify the following sub processes:

- View courses
- Manage topics for a course
- Manage course information

And similarly, the "Manage course assignment" use case can be refined into smaller discrete processes such as viewing the course calendar, viewing tutors, managing the tutor information of tutors working for the organization, and of course, assigning courses to tutors. Now, the use cases that we have identified within the "Manage course assignment" use case are:

- View course calendar
- View tutors
- Manage tutor information
- Assign courses to tutors

Our final list of use cases for the courseware management system will now be:

- View courses
- Manage topics for a course
- Manage course information
- View course calendar
- View tutors
- Manage tutor information
- Assign courses to tutors

Use Case Diagram
Class Diagram:

The UML class diagram of our Courseware Management System case study can be built after a careful analysis of the requirements. In the previous article, we identified the primary actors and use cases in the use case model of the case study. Because we did much of the groundwork of our analysis while building the use case model, we will use those analysis steps as the basis for identifying the classes and interfaces of this system.

Let us recap the analysis that was performed when the use case model was designed. The following terms and entities specific to the system were identified from the problem statement:

- Courses and Topics that make up a course
- Tutors who teach courses
- Course administrators who manage the assignment of the courses to tutors
- Calendar or Course Schedule is generated as a result of the
- Students who refer to the Course schedule or Calendar to decide which courses for which they wish to sign up

The potential actors of the system were:

- Tutors
- Course administrators
- Students

And the use cases of the system were:

- View courses
- Manage topics for a course
- Manage course information
- View course calendar
- View tutors
- Manage tutor information
- Assign courses to tutors

**Identifying classes of the Courseware Management System**

As you did in use case modeling, you will identify the classes and interfaces using an incremental approach.

1. Identify the "active" entities in the system

The basic rule that we learned until now for identifying classes and interfaces is that classes and interfaces reflect important entities of the business domain of the system being modeled. We will apply this rule to determine classes and interfaces of the case study system. At first glance, the actors identified in the use case appear to be prime candidates for being listed as potential classes. Even though we had excluded Students and Tutors from our final list of actors, we will still include them in our list as potential classes. So, our first list of classes in the system appears to be:
2. Identify business domain ("passive") entities in the system

But these are the "active" entities of the system. We had also identified "passive" elements in the system as well in the analysis for our use case model. These entities reflect the business domain and hence are potential classes for our system.

3. Entities that reflect the business terms are also called business domain classes or just "domain classes." Some of the business domain classes hold transient data and some hold persistent data for the application. Normally, such business domain classes map to either one or many database tables.

4. For example, in our case study, the Course class can be modeled as a database table `cms_course`. The data in this table for a particular course will be represented by an instance of the Course class and made available to the rest of the application.

Our two-step process has definitely yielded promising results! We have covered all the relevant items in our analysis. So, let us list the list of classes and interfaces that we have identified in the Courseware Management System.

- Course Administrator
- Tutor
- Student
- Course
- Topic
- CourseCalendar
Categorize and map the use cases and any relevant business functionality to either the passive or active entities. These will become the business methods of the classes in the system.

Classes encapsulate functionality. The classes that we have identified for the Courseware Management System also provide business functionality related to the application. The functionality encapsulated by these classes is distinct in nature and differs from each class. Recall from our use case model, that, along with actors, we had identified a set of use cases that the actors interacted with. Let us try to associate them with our classes. Because our primary actor is the course administrator and the use cases were related to this actor, we can directly map the use cases to the CourseAdministrator class as methods.

<table>
<thead>
<tr>
<th>ClassName</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>CourseAdministrator</td>
<td>viewCourses()</td>
</tr>
<tr>
<td></td>
<td>manageCourse()</td>
</tr>
<tr>
<td></td>
<td>manageTopic()</td>
</tr>
<tr>
<td></td>
<td>viewCourseCalendar()</td>
</tr>
<tr>
<td></td>
<td>viewTutors()</td>
</tr>
<tr>
<td></td>
<td>manageTutorInformation()</td>
</tr>
<tr>
<td></td>
<td>assignTutorToCourse()</td>
</tr>
</tbody>
</table>

In addition to this, we also can determine some implicit functionality of classes that reflect business entities. For example, what functionality should the Course class provide? Intuitively, we would define the Course class to provide functionality to view all courses in the system, ability to create new courses or modify information of existing courses, view the details of a particular course, or even remove a course from the system. We expect the Course class to provide such business functionality because the Course class reflects a business entity in the system. Hence, these become the methods exposed by the Course class. So, we can now refine the class diagram and add methods to each of these classes.

To cut a long story short, each of the classes that reflect business entities will provide similar implicit business functionality. Let us list all such "implicit" functionality for each of these classes.
<table>
<thead>
<tr>
<th>ClassName</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Course</td>
<td>viewAllCourses()</td>
</tr>
<tr>
<td></td>
<td>viewCourseInformation()</td>
</tr>
<tr>
<td></td>
<td>createCourse()</td>
</tr>
<tr>
<td></td>
<td>modifyCourse()</td>
</tr>
<tr>
<td></td>
<td>removeCourse()</td>
</tr>
<tr>
<td>Topic</td>
<td>viewAllTopics()</td>
</tr>
<tr>
<td></td>
<td>viewTopicInformation()</td>
</tr>
<tr>
<td></td>
<td>createTopic()</td>
</tr>
<tr>
<td></td>
<td>modifyTopic()</td>
</tr>
<tr>
<td></td>
<td>removeTopic()</td>
</tr>
<tr>
<td>Tutor</td>
<td>viewTutorInformation()</td>
</tr>
<tr>
<td></td>
<td>createTutor()</td>
</tr>
<tr>
<td></td>
<td>modifyTutor()</td>
</tr>
<tr>
<td></td>
<td>removeTutor()</td>
</tr>
<tr>
<td>CourseCalendar</td>
<td>viewCourseCalendar()</td>
</tr>
<tr>
<td>Student</td>
<td>viewAllStudents()</td>
</tr>
<tr>
<td></td>
<td>viewStudentInformation()</td>
</tr>
</tbody>
</table>

Revisit the class diagram and revise it by identifying shared features and/or common functionality between classes or interfaces. These will translate into reusable pieces of code for your system. To some extent, we can say that CourseAdministrator, Tutor, and Student are essentially users of the system. Hence, we can define a shared parent class named User and define basic functionality like for example, authentication, in the User class that can be inherited by the CourseAdministrator, Tutor, and Student classes. It is left to the design expertise to identify reusable classes/functionality.

This completes our analysis of the problem statement to define the classes for the Courseware Management System.
Identifying relationships between the classes of the Courseware Management System

The next step after defining the classes of the Courseware Management System is to define the relationships and dependencies between these classes and interfaces. To define the relationships between the classes, we need to analyze the interconnections between the classes—whether implicit or explicit. Relationship analysis can be broken up into three steps:

1. Identify relationships between "active" entities

   Active entities normally share generalization relationships ("is-a"). Essentially, the common attributes and functionality between classes are defined in a common parent class. All the related child classes inherit functionality from the parent class. Apart from generalization, a few active entities can also be interconnected by a realization relationship. Recall that elements in a realization relationship implement declared functionality as a "contract." For example, a set of classes may implement functionality declared as methods in an interface, and this can be modeled as a realization relationship between the interface and the classes implementing the interface.

   In our case study, we do not find an example of inheritance relationship between the active entities such as Student, Tutor, and CourseAdministrator or any realization relationships.

2. Identify relationships between "passive" business entities

   Passive business entities frequently share plain association or aggregation relationships ("has-a"). This is especially true because these business entities are non-transactional in nature and reflect data more than behavior. It is by far quite intuitive to identify aggregation as well as its variations—composition relationships for passive business entities.
Some of the classes in our case study do exhibit aggregation relationships. Because a set of topics makes up a course, we can define an aggregation relationship between the Course and Topic classes. Moreover, we can define this as a directed aggregation, meaning that you can check for the topics of a course but not vice versa. Similarly, we can define a plain association relationship between the Course and Tutor classes and Course and Student classes.

Identify relationships between "active" and "passive" entities

Relationships between active and passive entities can easily be represented using directed association. The directed association, a variation of the "vanilla" association relationship, provides easy identification of which is the container class and which is the contained class. The CourseAdministrator class can be modeled to have a directed association with the Course class. This association can be named as "manages" because the course administrator manages courses as a business activity. In addition to this, because the course administrator also manages the tutor information and topic information, we can model a directed relationship named as "manages" between the CourseAdministrator and the Course and Topic classes, respectively. We can enhance the readability of the association between CourseAdministrator and the Course, Tutor, and Topic classes by defining the multiplicity for the association—one to many, one to one, many to many, and so forth.
We have completed identifying the classes for the Courseware Management System and established the relationships among the classes. Take a look at the class diagram in Figure 4.2.1. The class diagram of the Courseware Management System includes all the classes and their relationships that we identified during our analysis of the problem statement.