

**Usman Institute of Technology**

**Department of Computer Science**

**Course Code: SE308**

**Course Title: Software Design and Architecture**

**Summer 2024**

**Lab 01**

**Objective:** To Understand Use Case Diagram and Analysis Level Class Diagram

**Student Information**

|  |  |
| --- | --- |
| Student Name | **Jawwad Bhatti** |
| Student ID | **20B-011-SE** |
| Date | **14-07-2024** |

**Assessment**

|  |  |
| --- | --- |
| Marks Obtained |  |
| Remarks |  |
| Signature |  |

**Usman Institute of Technology**

**Department of Computer Science**

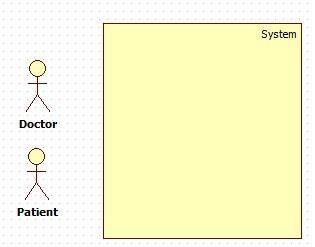
**SE308 - Software Design and Architecture**

**Lab 01**

**Elements of Use Case Diagram:**

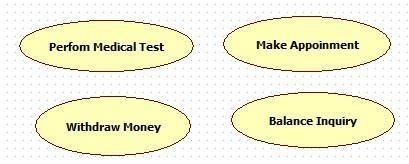
**Actors:**

An actor is a role that a user plays with respect to the system. Actors carry out use cases. A single actor may perform many use cases; on the other hand, a use case may have several actors performing it. An actor is shown as stick figure in use case diagram depicted “outside” the system boundary as shown in figure



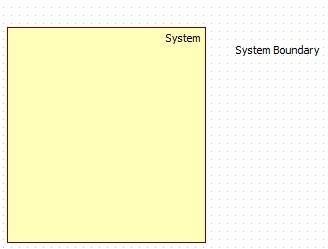
**Use Cases:**

Use cases in use case diagram are shown as an eclipse. It defines the distinct functionality of the system. To choose a business process as a likely candidate for modeling as use case, ensure that the business process is discrete in nature. As the first step in identifying use case, list the discrete business functions in requirements document. Each of this business function can be classified as a potential use case. A use case can be connected to many actors at a time.



**System Boundary:**

System boundary defines the scope of what a system will be. The system boundary in use case diagram defines the limits of a system. System boundary is shown as a rectangle spanning all the use cases.



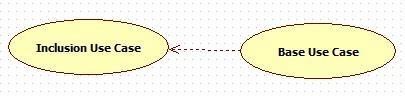
**Relationships in Use Case Diagram:**

A relationship between two use cases is basically a dependency between two use cases. Use cases share different kind of relationship

1. **Include:**

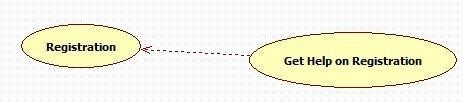
An include relationship is a relationship in which one use case (the base use case) includes the functionality of another use case (the inclusion use case).The include relationship supports the reuse of functionality in a use case.

As the following figure illustrates, an inclusive relationship is shown as a dashed line with an open arrow pointing from the base use case/parent use case to the inclusive use case/child use case. The keyword include is attached to the connector.



1. **Extends**

In an extend relationship between two use cases, the child use case adds to the existing functionality and characteristics of the parent use case. An extend relationship is depicted with a directed arrow having a dotted shaft, similar to the include relationship. The tip of the arrowhead points to the parent use case and the child use case is connected at the base of the arrow The keyword extends attached to the connector.



**Class Diagram**

* Class diagrams are visual representations of the static structure and composition of a particular system using the conventions set by the Unified Modeling Language (UML).
* System designers use class diagrams as a way of simplifying how objects in a system interact with each other.
* Using class diagrams, it is easier to describe all the classes, packages, and interfaces that constitute a system and how these components are interrelated.
* Since class diagrams are used for many different purposes, such as making stakeholders aware of requirements to highlighting your detailed design, you need to apply a different style in each circumstance

**Example**

* Simple class diagram may be used to show how an organization such as a convenient store chain is set up.
* Precisely detailed class diagrams can readily be used as the primary reference for translating the designed system into a programming code.

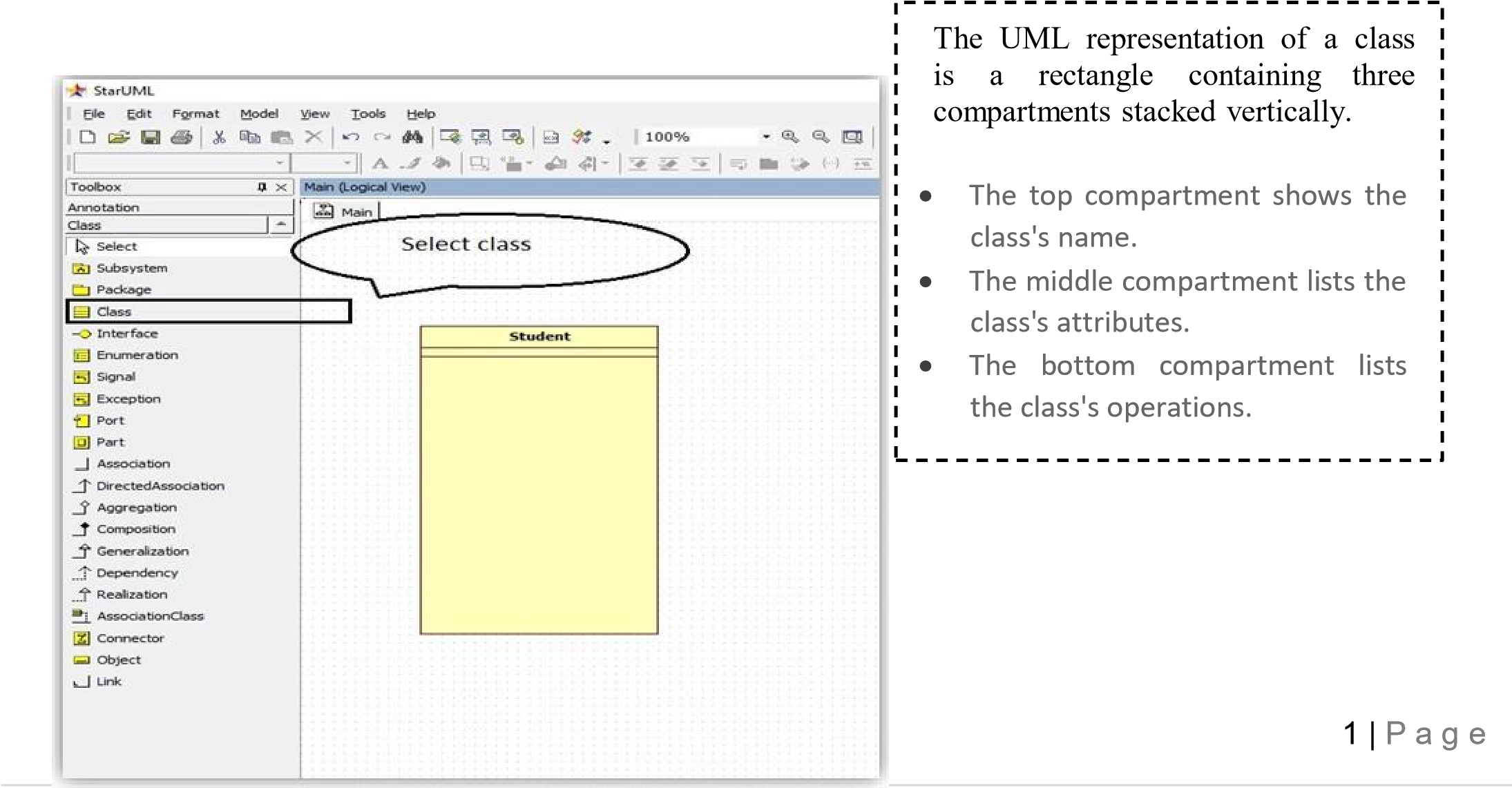
**Notation of Class Diagram**

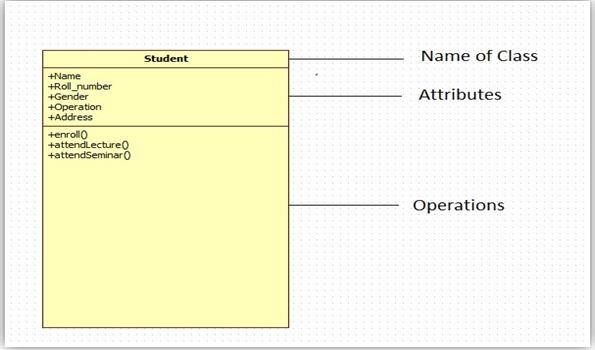
**1. Class**

* An object is any person, place, thing, concept, event, screen, or report applicable to your system. Objects both know things (they have attributes) and they do things (they have methods).
* A class is a representation of an object and, in many ways, it is simply a template from which objects are created.
* Classes form the main building blocks of an object-oriented application.

**Example**

Although thousands of students attend the university, you would only model one class, called Student, which would represent the entire collection of students.





**2. Attributes**

An attribute of a class represents a characteristic of a class that is of interest for the user.

The full format of the attribute text notation is:

**Visibility name: type multiplicity = default [property-string]**

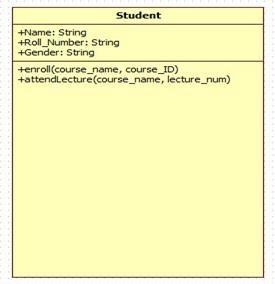
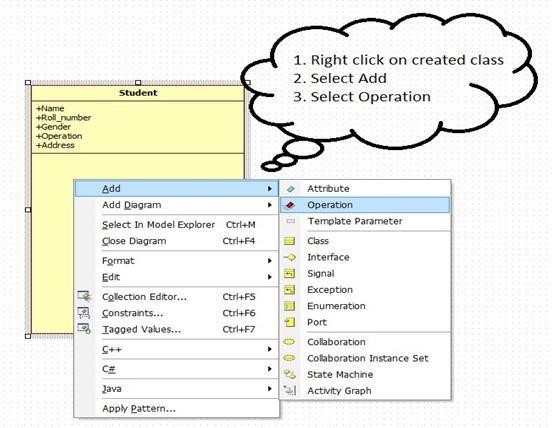
**3. Operations**

A UML operation is a declaration, with a name, parameters, return type, exceptions list, and possibly a set of constraints of pre and post conditions. But, it isn’t an implementation – rather, methods are implementation.

**4. Visibility**:

Use visibility markers to signify who can access the information contained within a class.

* Public +
* Private -
* Protected #
* Package



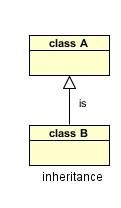
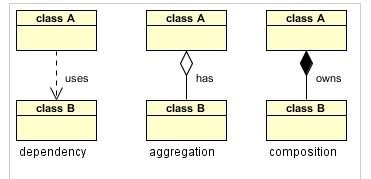
**Relationship**

Dependency: class A uses class B

Aggregation: class A has a class B

Composition: class A owns a class B

Inheritance: class B is a Class A (or class A is extended by class B)



**1. Association**

An association is a "using" relationship between two or more objects in which the objects have their own life time and there is no owner.

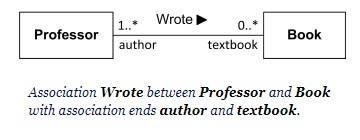
For **Example**: A patient may visit one or many doctors and same way, a doctor can be associated with multiple patients. If a patient dies, existence of doctor will not be vanished and similarly if doctor dies patient will remain patient.

Association is represented as thin line connecting two classes. Association can be unidirectional (shown by arrow at one end) or bidirectional (shown by arrow at both end) or without arrow.

**Multiplicity** defines how many instances can be associated at any given moment**.**

|  |  |  |
| --- | --- | --- |
| 0..1 | No instances or one instance | A flight seat can have no or one passenger only |
| 1 | Exactly one instance | An order can have only one customer |
| 0..\* or \* | Zero or more instances | A class can have zero or more students. |
| 1..\* | One or more instances (at least one) | A flight can have one or more passenger |

**Example**:



**2. Aggregation**

Aggregation is a special form of [association. It is](http://beginnersbook.com/2013/05/association/) also a relationship between two classes like association, however, it’s a **directional** association, which means it is strictly a **one way association, means unidirectional association.** It represents a **Has-A** relationship.

**For Example**: Consider two classes Student class and Address class. Each student must have an address so the relationship between student and address is a Has-A relationship. But if you consider its vice versa then it would not make sense as an Address doesn’t need to have a Student necessarily.

**NOTE:** Unarguably, Address is an attribute of a student, but here in this example I am breaking address into several fields i.e city, province and country. This is the reason for making address a class**.**

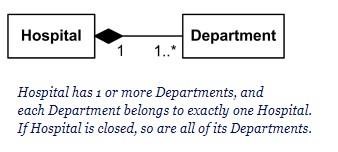
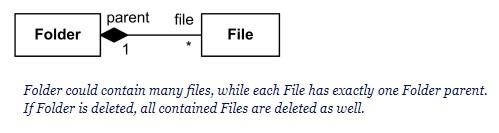
**3. Composition**

Composition is a special case of aggregation. In a more specific manner, a restricted aggregation is called composition. When an object contains the other object, if the contained object cannot exist without the existence of container object, then it is called composition.

For **Example**: Consider the same scenario with some modifications. In this scenario, a student has address and each student has different address (Please keep sibling relationship argument apart). So, when a student record is added his house number and street number will be entered. And if I delete the record of a particular student, then his/her record will be of no use.

**filled black diamond** at the aggregate (whole) end.

**Example:**



**4. Generalization**

In object oriented programming, the concept of IS-A is a totally based on Inheritance, which can be of two types Class Inheritance or Interface Inheritance. It is just like saying "A is a B type of thing". For example, Apple is a Fruit, Car is a Vehicle etc. Inheritance is uni-directional. For example House is a Building. But Building is not a House.

It is key point to note that you can easily identify the IS-A relationship. Wherever you see an extends keyword or implements keyword in a class declaration, then this class is said to have IS-A relationship.

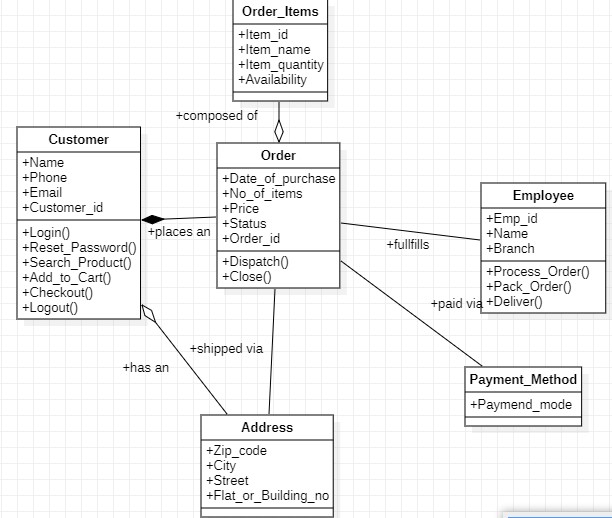
**Example**: Refer your theory lectures.

**Student Tasks**

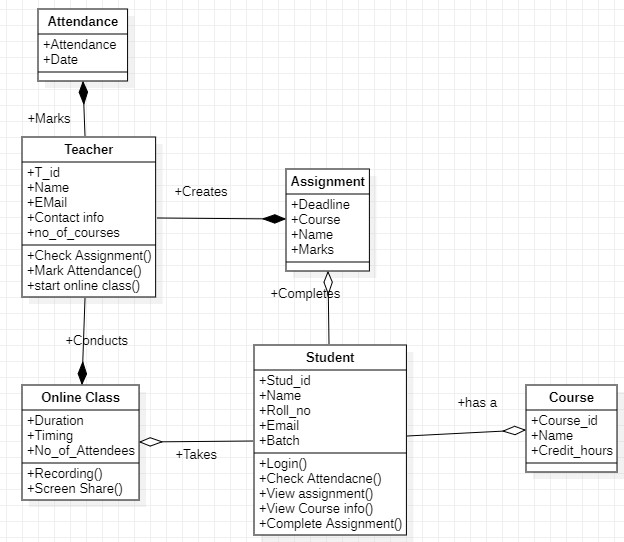
**Task 1: Create a UML Class diagram of the below scenario also include all possible relations**

1. An **Order** is ordered by a **Customer**.
2. An **Order** is fulfilled by an **Employee**.
3. An **Order** is paid via a **Payment Method**.
4. An **Order** is shipped via an **Address** belonging to the **Customer** who is the buyer.
5. An **Order** is composed of **Order Items**.

**Note:** Add appropriate attributes and methods according to the order management system



|  |  |
| --- | --- |
| **Task 2: Consider the scenario of LMS System and draw a Analysis level class diagram with its** | |
| **possible objects and their relation** |  |



Use Cases Diagrams Lab Session 1 Ta

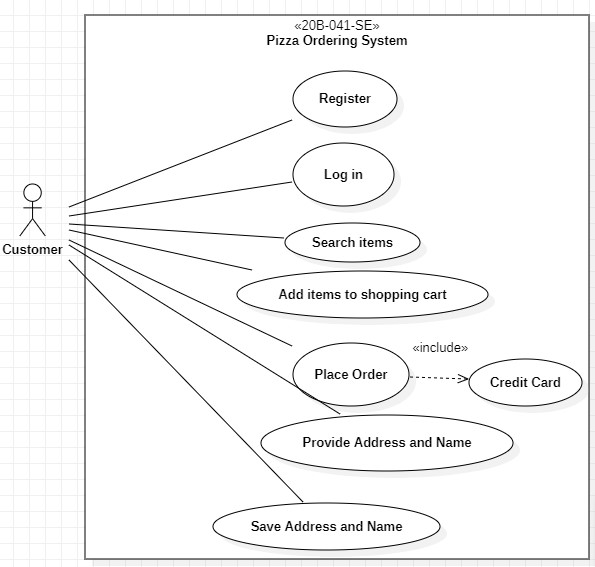
**Task 1 : The Pizza Ordering System**

The Pizza Ordering System allows the user of a web browser to order pizza for home delivery. To place an order, a shopper searches to find items to purchase, adds items one at a time to a shopping cart, and possibly searches again for more items.

When all items have been chosen, the shopper provides a delivery address. If not paying with cash, the shopper also provides credit card information.

The system has an option for shoppers to register with the pizza shop. They can then save their name and address information, so that they do not have to enter this information every time that they place an order.

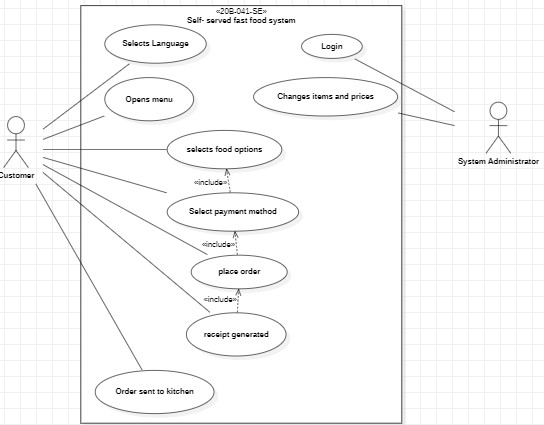
Develop a **use case diagram**, for a use case for placing an order, Place Order. The use case should show a relationship to two previously specified use cases, Identify Customer, which allows a user to register and log in, and Pay by Credit, which models credit card payments



Task 2: Self-Served Fast Food System

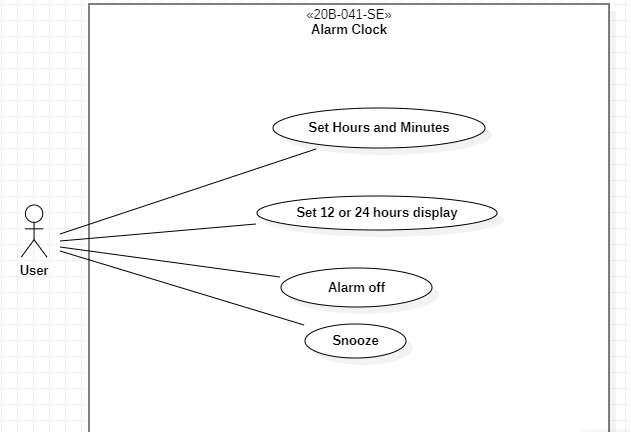
This self-service fast food restaurant will be equipped with a user-friendly touch screen, a credit/debit card reader, and software for completing the process at the backend. For this system there will be a system administrator who will have the rights to enter the menu with their current prevailing prices. He/she can enter anytime in the system by a secured system password to change the menu contents by adding or deleting an item or changing its price.

Now when the customer enters the restaurant, he will place his order with the help of the touch screen using the intuitive graphical user interface, right from the selection of language till the payment confirmation. He will select from the food options according to his choice and the system will display the payment amount he has to make once he has finished with his order. He will have the option of paying the bill by cash, debit card or a credit card. The user will slide his card and the system will check for the validity of the card and the payment will be made. A receipt will be printed containing the order number and the order will be sent in the kitchen for processing.



Task 3 : Alarm Clock

Suppose we want to develop software for an alarm clock. The clock shows the time of day. Using buttons, the user can set the hours and minutes fields individually, and choose between 12 and 24-hour display. It is possible to set one or two alarms. When an alarm fires, it will sound some noise. The user can turn it off, or choose to ’snooze’. If the user does not respond at all, the alarm will turn off itself after 2 minutes. ’Snoozing’ means to turn off the sound, but the alarm will fire again after some minutes of delay. This ’snoozing time’ is pre-adjustable

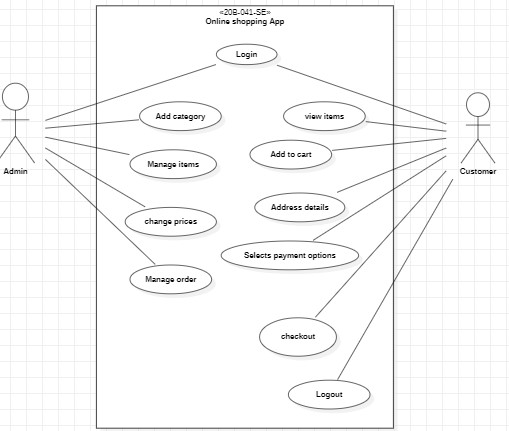


Use Cases Diagrams

Lab Session 1 Tasks

Task 4: Online Shopping

Create a use case diagram based on your own scenario



Task 5: Course Registration System

At the beginning of each semester students may request a course catalogue containing a list of course offerings for the semester. Information about each course, such as professor, department, and prerequisites will be included to help students make informed decisions.

The new on-line registration system will allow students to select four course offerings for the coming semester. In addition, each student will indicate two alternative choices in case a course offering becomes filled or canceled. No course offering will have more than ten students. No course offering will have fewer than three students. A course offering with fewer than three students will be canceled. Once the registration process is completed for a student, the registration system sends information to the billing system, so the student can be billed for the semester.

Professors must be able to access the on-line system to indicate which courses they will be teaching. They will also need to see which students signed up for their course offering.

For each semester, there is a period of time that students can change their schedules. Students must be able to access the on-line system during this time to add or drop courses. The billing system will credit all students for courses dropped during this period of time

