



UML – Class diagrams, Sequence/Communication Diagrams.

By Apram Cherra and Paul Swan

Introduction to UML

Unified Modeling Language (UML)

UML standardized diagrams are used to describe, model, and design software systems.

Importance of UML

UML provides a standardized way of modeling software systems.

It helps in improving communication among stakeholders.

It helps in identifying potential design issues.

It helps in reducing development time and cost.

Types of UML diagrams

Class Diagram

Sequence Diagram

Communication Diagram

Domain Model

Use Case Diagram

Activity Diagram

State Diagram.

And others...

Class Diagram

A class diagram represents the static structure of a software system.

It shows the classes, interfaces, attributes, and operations of the system, as well as the relationships between them.

Purpose of Class Diagram

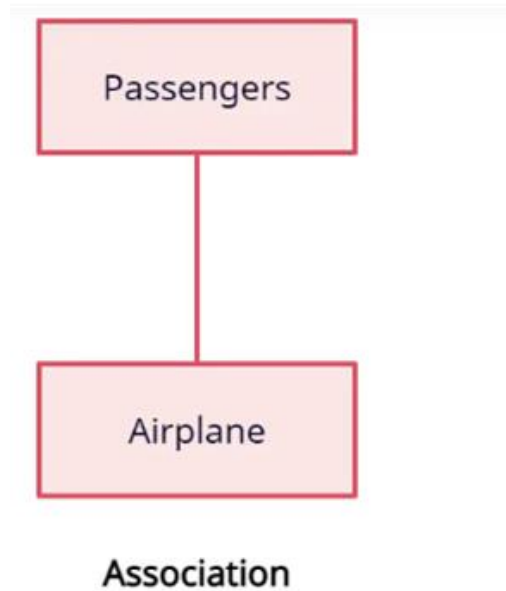
A class diagram provides a detailed view of a system's structure, including its objects and their behavior.

Can be easier to read.

Helpful when planning projects.

Associations

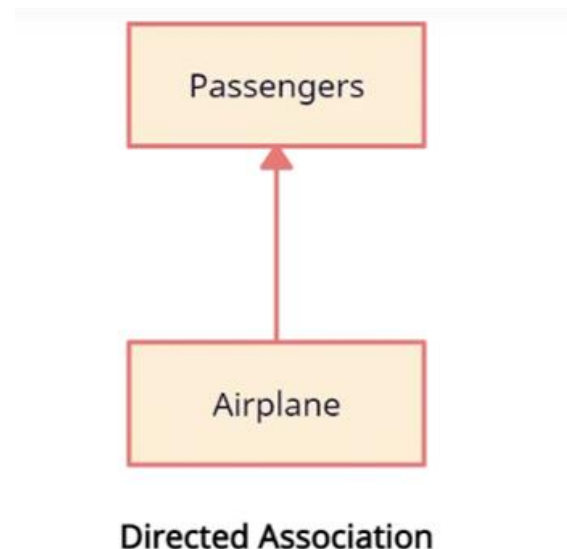
Associations: Relationships between two classes. Associations are represented by lines connecting the classes.



Association Types: Directed Association

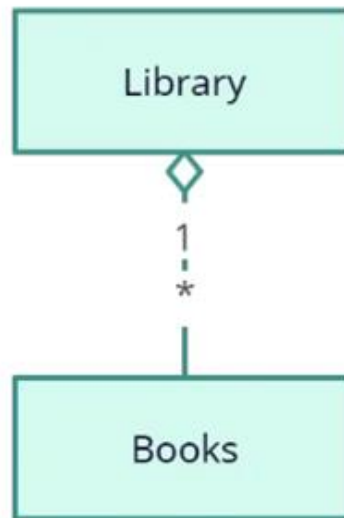
A relationship between two objects, where one object is the "source" or "sender" and the other object is the "target" or "receiver".

This means that the association is one-way. Changes made to one object do not necessarily affect the other object.



Association Types: Aggregation

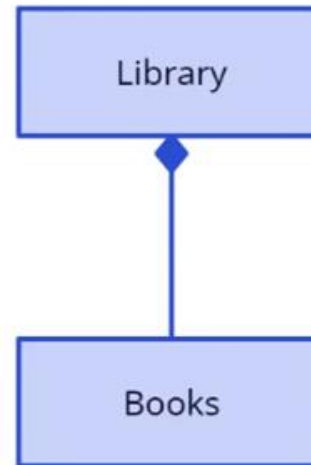
A association where one object is composed of one or more other objects. The objects that make up the aggregate can exist independently of the aggregate itself and may be shared between multiple aggregates.



Aggregation

Association Types: Composition

Where one object is composed of one or more other objects, but the lifetime of the composed objects is tied to the lifetime of the containing object. This means that when the containing object is destroyed, so are its composed objects.



Composition

Association Types: Inheritance/Generalization

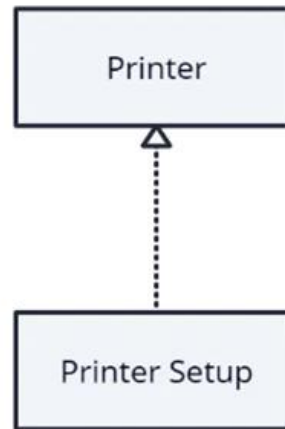
A type of association where one class (the subclass) inherits the properties and methods of another class (the superclass). This allows the subclass to reuse the code and behavior of the superclass, while also adding its own unique features.



Inheritance

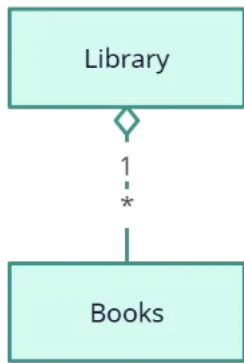
Association Types: Realization

A type of association where a class or object implements an interface, which specifies a set of methods that the class or object must implement. This allows for multiple classes or objects to share a common interface, while also allowing each one to implement the interface in its own way.

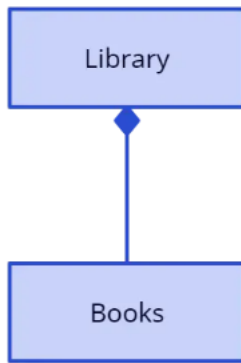


Realization

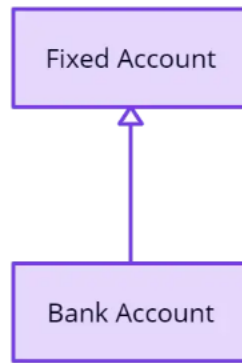
Elements of a class diagram



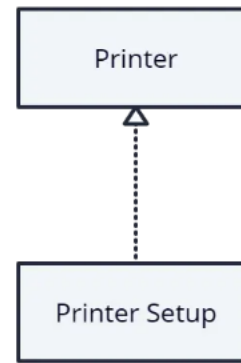
Aggregation



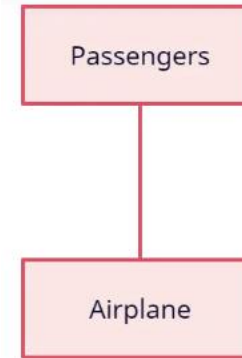
Composition



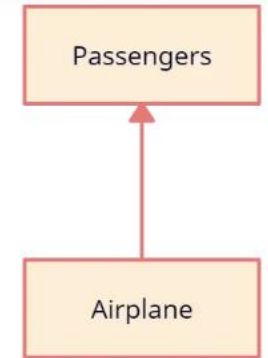
Inheritance



Realization

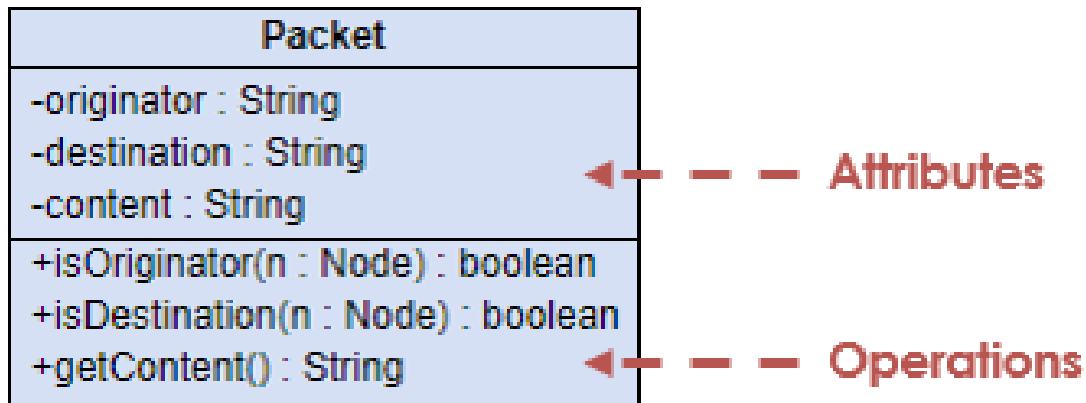


Association



Directed Association

Classes - At the Core of Class Diagrams



- Class: A blueprint or template for creating objects. Classes are represented by rectangles with the class name inside.
- Attributes: Characteristics or properties of a class, such as its name, type, and visibility. Attributes are represented by small rectangles with the attribute name and type inside, connected to the class with a line.
- Operations: Actions or behaviors that a class can perform, such as methods or functions. Operations are represented by rectangles with the operation name, parameters, and return type inside, connected to the class with a line.

Visibility



Mark	Visibility type
+	Public
#	Protected
-	Private
~	Package

Not required to show for classes

Required for attributes and operations.

Multiplicity

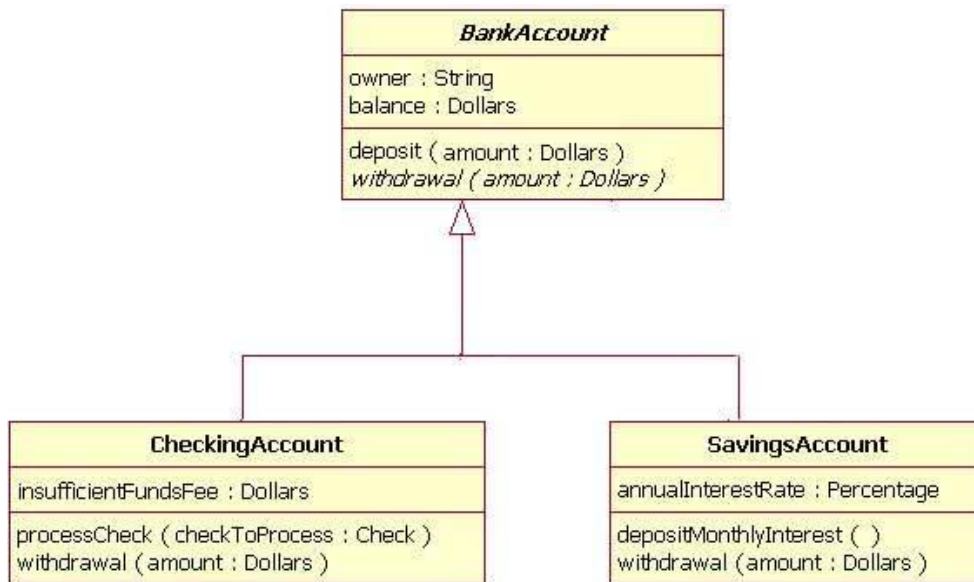
The multiplicity value represents the number of instances a class can have.



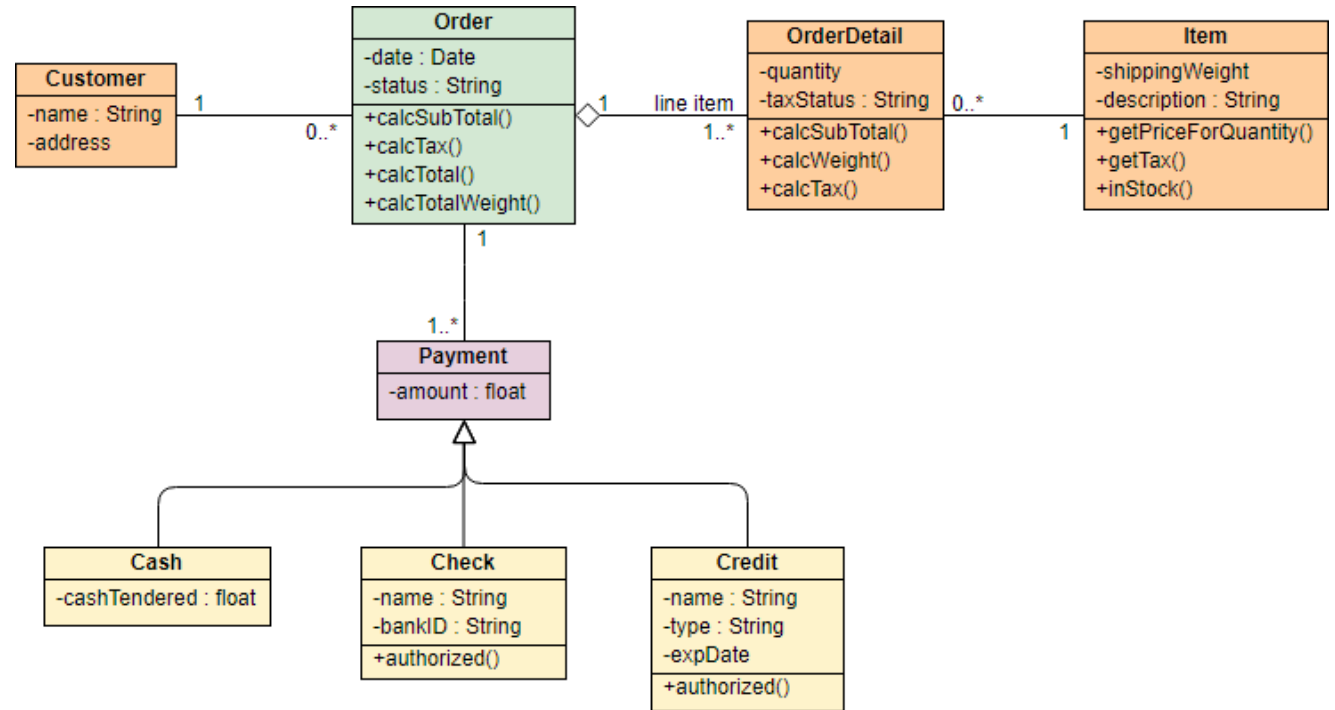
The multiplicity value of `0..1` next to the **Plane** class indicates that a **Flight** instance can have either one instance of a **Plane** associated with it or none.

how to indicate abstract methods and interfaces

The italic text in the BankAccount indicates the class is an abstract class and the withdrawal method is an abstract operation.



Class Diagram Example.



Communication Diagrams

- It shows the interactions between objects, the order in which they occur, and the messages exchanged between them.

Purpose of Communication Diagrams

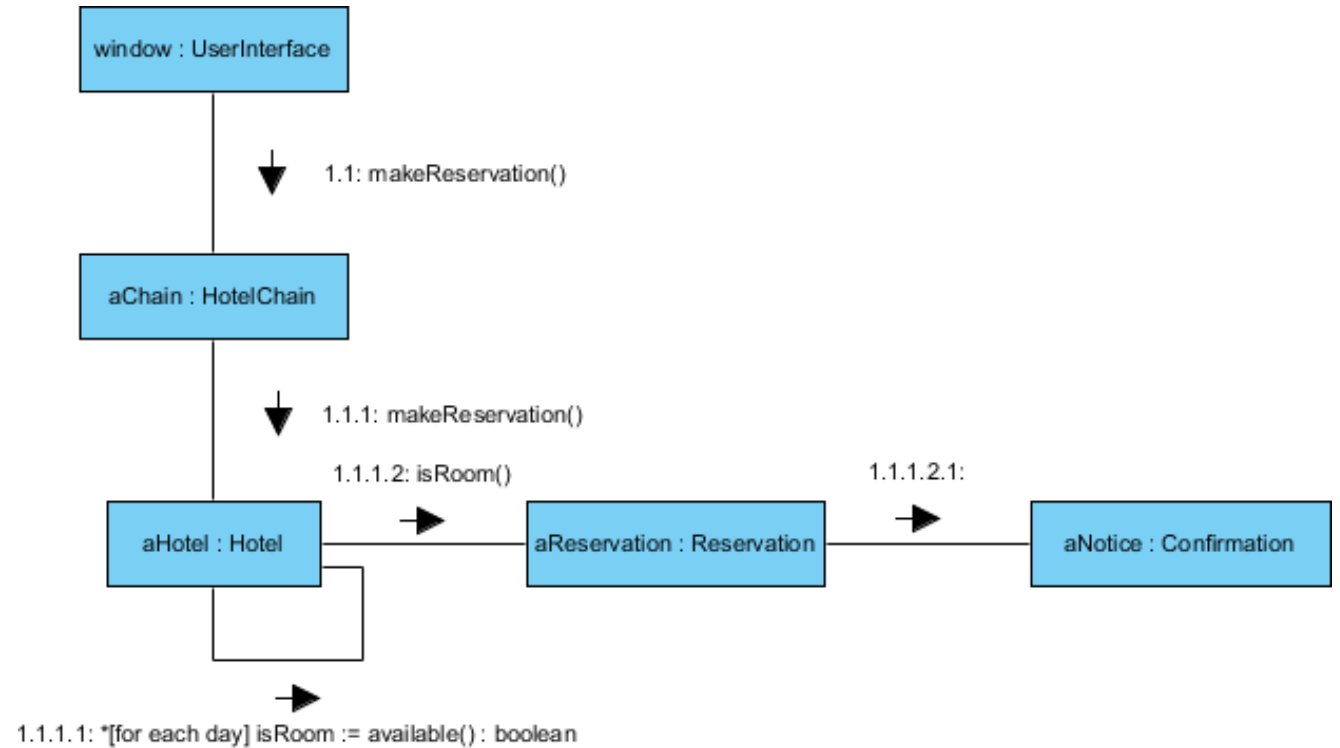
- Model the timing and sequencing of messages (such as events) exchanged between participants
- Clarify the relationships and dependencies between objects or components in a system
- Finding Message Related Errors:
 - Message content errors: where messages contain incorrect or incomplete information, leading to incorrect processing or decision-making
 - Communication errors: where messages are lost, duplicated, or not delivered to the intended recipient, causing problems with the system's functionality or reliability

Elements of Communication Diagrams

Labeled Messages.

Loops are messages sent by the object to itself.

Communication Diagram Example.



Sequence Diagrams

A sequence diagram represents the dynamic behavior of a software system in a single use case.

It shows the interactions between objects and the order in which they occur, as well as the messages exchanged between them.

Purpose of Sequence Diagrams

The purpose of a sequence diagram is to provide a detailed view of the system's behavior, including:

- The flow of control between objects, i.e., how messages are passed back and forth.

- The flow of data between objects, i.e., what information is being exchanged.

Sequence diagrams are useful in identifying potential issues in the system's design, such as:

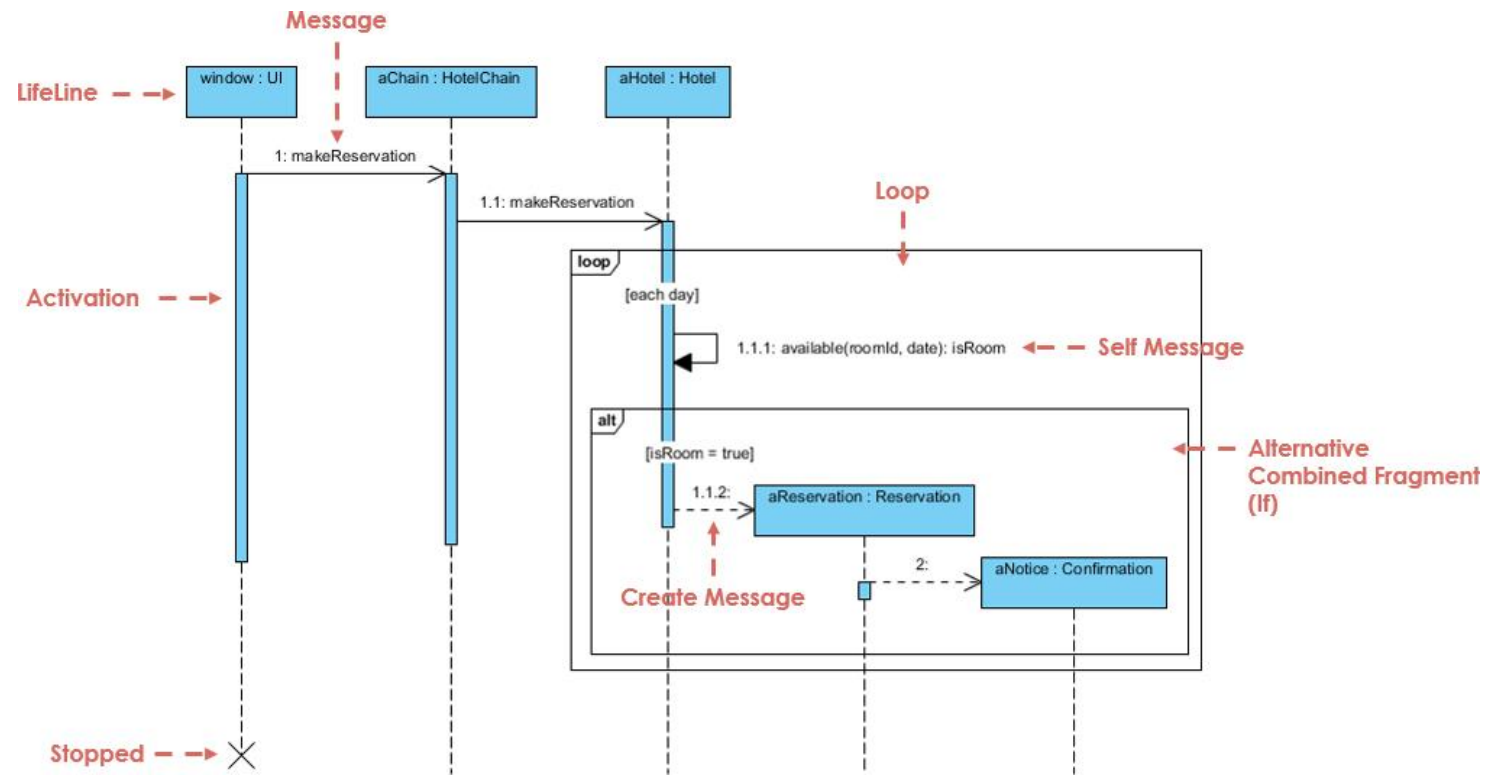
- Incorrect sequencing, i.e., when objects are not communicating with each other in the correct order.

- Missing interactions, i.e., when important messages or data are not being passed between objects.

Elements of Sequence Diagrams

- 1.Objects: Objects represent instances of classes or components in the system. They are depicted as rectangles with the object name at the top, optionally followed by a colon and the class name. Objects can be self-referential, meaning they can send messages to themselves.
- 2.Lifelines: Lifelines represent the existence of objects over time. They are depicted as vertical lines that extend downward from the object they represent. Lifelines can have activation bars that indicate the time period during which an object is executing an operation.
- 3.Messages: Messages represent the interactions between objects. They are depicted as arrows that go from the sender object to the receiver object, with the message name above the arrow. Messages can have optional parameters and return values.
- 4.Activations: Activations represent the time periods during which an object is executing an operation. They are depicted as rectangles that overlap with the lifeline of the object. The length of an activation bar represents the duration of the operation.

Sequence Diagram Example



Sequential vs Communication Diagrams

SEQUENTIAL

Show the flow of control or sequence of actions in a system or process.

Emphasize the order in which actions occur.

COMMUNICATION

Show the interactions between objects or components in a system.

Emphasize the messages or data being exchanged between objects or components.

```
graph TD
    subgraph Portal [Patient Doctor Pairing Portal]
        direction TB
        subgraph Patient_UseCases
            P_Register((Register))
            P_Login((Login))
            P_ViewDoctors((View Doctors))
            P_ManageAppointment((Manage Appointment))
            P_CancelAppointment((Cancel Appointment))
            P_MakePayment((Make Payment))
            P_BookWaitingList((Book in Waiting List))
        end
        subgraph Doctor_UseCases
            D_Login((Login))
            D_CreatePrescription((Create Prescription))
            D_ManagePatient((Manage Patient))
            D_ViewCalendar((View Calendar))
        end
        subgraph Admin_UseCases
            A_ManageUsers((Manage Users))
            A_DeleteStaffs((Delete Staffs))
            A_UpdateStaffs((Update Staffs))
            A_AddStaff((Add Staff))
            A_SendReminder((Send Reminder))
            A_ViewAppointments((View Appointments))
            A_UpdateRecord((Update Record))
            A_ViewPatient((View Patient))
        end
        subgraph Shared_UseCases
            S_SendMessages((Send Messages))
            S_AddAppointment((Add Appointment))
            S_ChangeAppointment((Change Appointment))
            S_SearchAppointment((Search Appointment))
        end
    end

    Patient((Patient)) --- P_Register
    Patient --- P_Login
    Patient --- P_ViewDoctors
    Patient --- P_ManageAppointment
    Patient --- P_CancelAppointment
    Patient --- P_MakePayment
    Patient --- P_BookWaitingList

    Doctor((Doctor)) --- D_Login
    Doctor --- D_CreatePrescription
    Doctor --- D_ManagePatient
    Doctor --- D_ViewCalendar

    Admin((Admin)) --- A_ManageUsers
    Admin --- A_DeleteStaffs
    Admin --- A_UpdateStaffs
    Admin --- A_AddStaff
    Admin --- A_SendReminder
    Admin --- A_ViewAppointments
    Admin --- A_UpdateRecord
    Admin --- A_ViewPatient

    P_Register -.->|«include»| P_Login
    P_ManageAppointment -.->|«include»| P_Login
    P_ManageAppointment -.->|«include»| P_CancelAppointment
    P_ManageAppointment -.->|«include»| P_MakePayment
    P_ManageAppointment -.->|«include»| P_BookWaitingList
    P_ManageAppointment -.->|«include»| S_SendMessages
    P_ManageAppointment -.->|«include»| S_AddAppointment
    P_ManageAppointment -.->|«include»| S_ChangeAppointment
    P_ManageAppointment -.->|«include»| S_SearchAppointment
    P_ManageAppointment -.->|«include»| A_ManageUsers
    P_ManageAppointment -.->|«include»| A_DeleteStaffs
    P_ManageAppointment -.->|«include»| A_UpdateStaffs
    P_ManageAppointment -.->|«include»| A_AddStaff
    P_ManageAppointment -.->|«include»| A_SendReminder
    P_ManageAppointment -.->|«include»| A_ViewAppointments
    P_ManageAppointment -.->|«include»| A_UpdateRecord
    P_ManageAppointment -.->|«include»| A_ViewPatient

    D_ManagePatient -.->|«include»| S_SearchAppointment
    D_ManagePatient -.->|«include»| A_UpdateRecord
    D_ManagePatient -.->|«include»| A_ViewPatient

    A_ManageUsers -.->|«include»| A_DeleteStaffs
    A_ManageUsers -.->|«include»| A_UpdateStaffs
    A_ManageUsers -.->|«include»| A_AddStaff
    A_ManageUsers -.->|«include»| S_SendMessages
    A_ManageUsers -.->|«include»| S_AddAppointment
    A_ManageUsers -.->|«include»| S_ChangeAppointment
    A_ManageUsers -.->|«include»| S_SearchAppointment
    A_ManageUsers -.->|«include»| A_ViewAppointments
    A_ManageUsers -.->|«include»| A_UpdateRecord
    A_ManageUsers -.->|«include»| A_ViewPatient
```

You will be developing a website that helps patients seek doctors needed and also help doctors identify patients in need. This project aims to link individuals with healthcare professionals. Your client would like to create a web app which attempts to link patients with healthcare professionals through short message service (SMS)

What we have here is a use case diagram for that entire system. But what we will be doing is focusing on a couple of these bubbles and seeing how we can create a sequence diagram for those use cases.

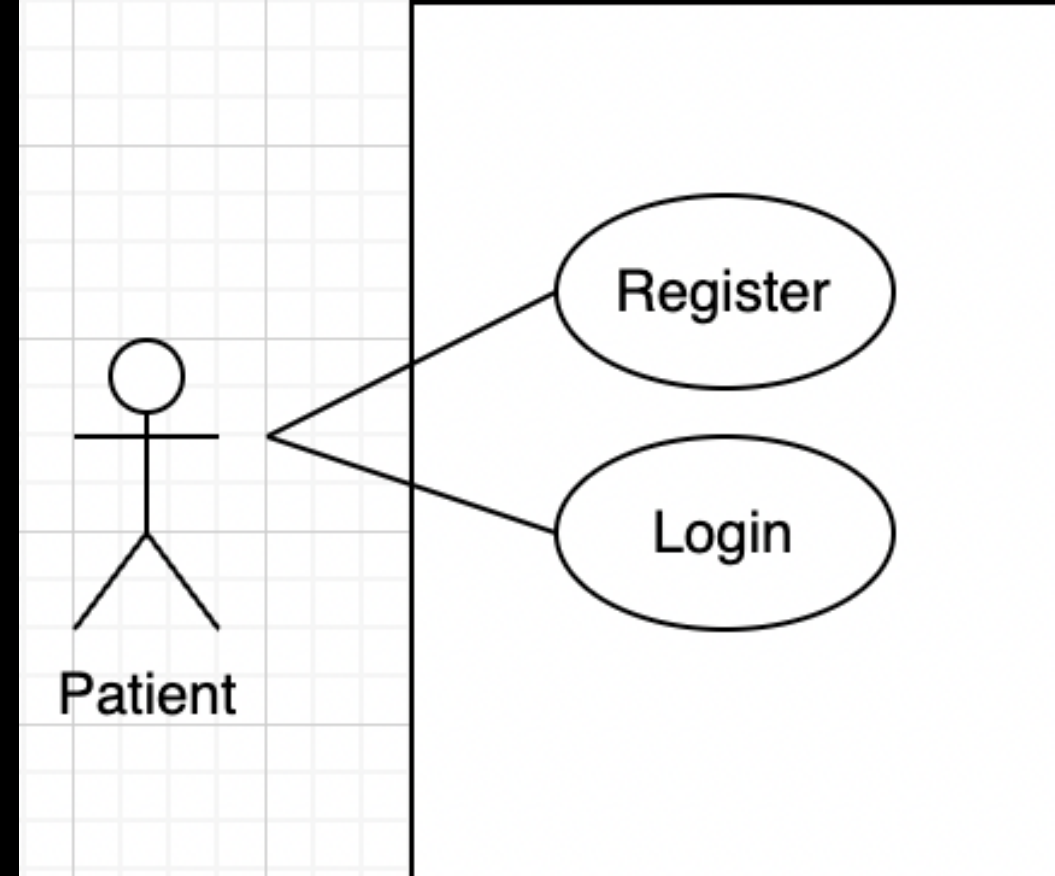
Let's first take a look at these two use cases first.

How can we create a sequence diagram for these use cases?

Just think about the interaction that will occur between the actor (user) and the web page, and database for registration.

Focus on the order of events.

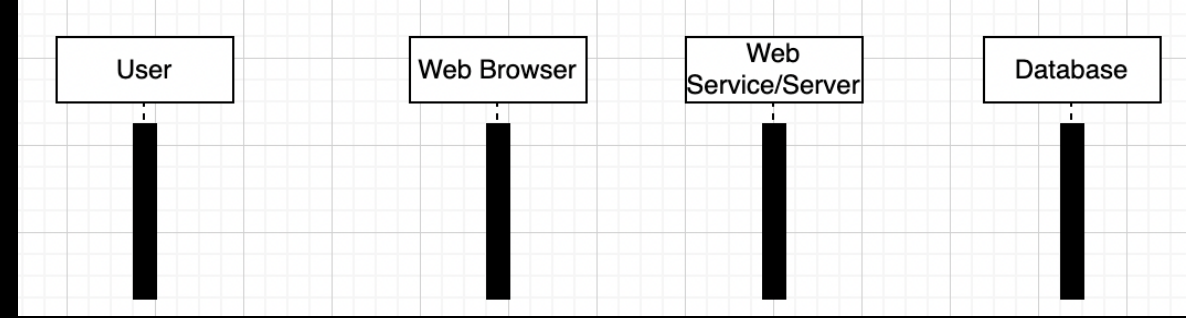
Let's start with registration.



We can start by listing all objects.

Now we need to focus on how these objects will interact with each other.

First trying explaining in your own words the sequence of events that would occur between these objects.



There is no right or wrong answer. But one way to go about this would be....

First, user would send a request to be registered.

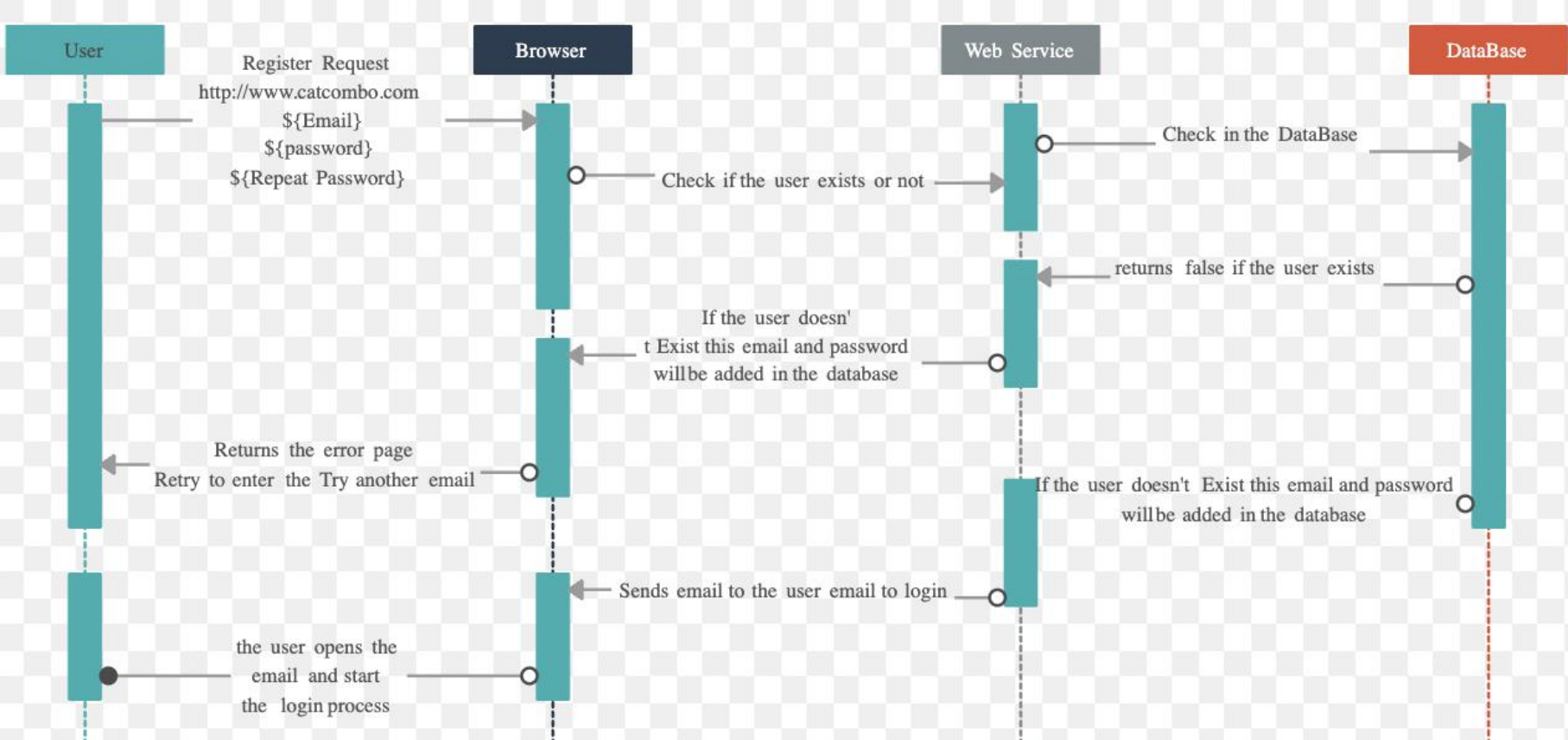
Browser send request to the service so they can check the database if that user already exists or not.

Database will return false if user exists, if doesn't, email and password will be added to database.

Now depending on what happened, the web service will send a web page for the browser to return to the user.

Some services may send a confirmation email, it's up to you if you want to convey this.

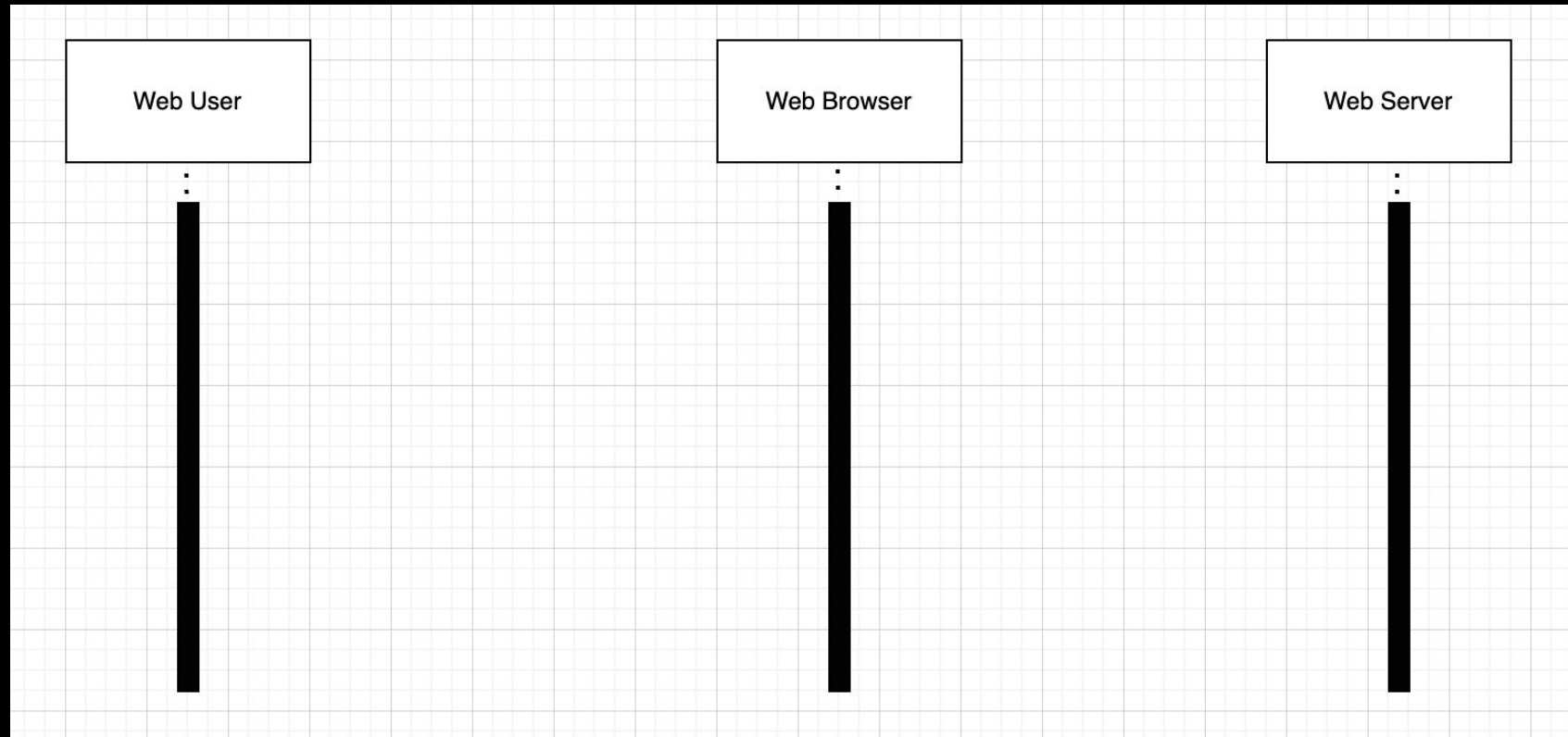
The sequence diagram for the registration scenario would look something like this depending on what you accounted for. We are thinking about each interaction that could happen during the registration process between different actors, machines, etc.



Now let's try creating a sequence diagram for the login use case.

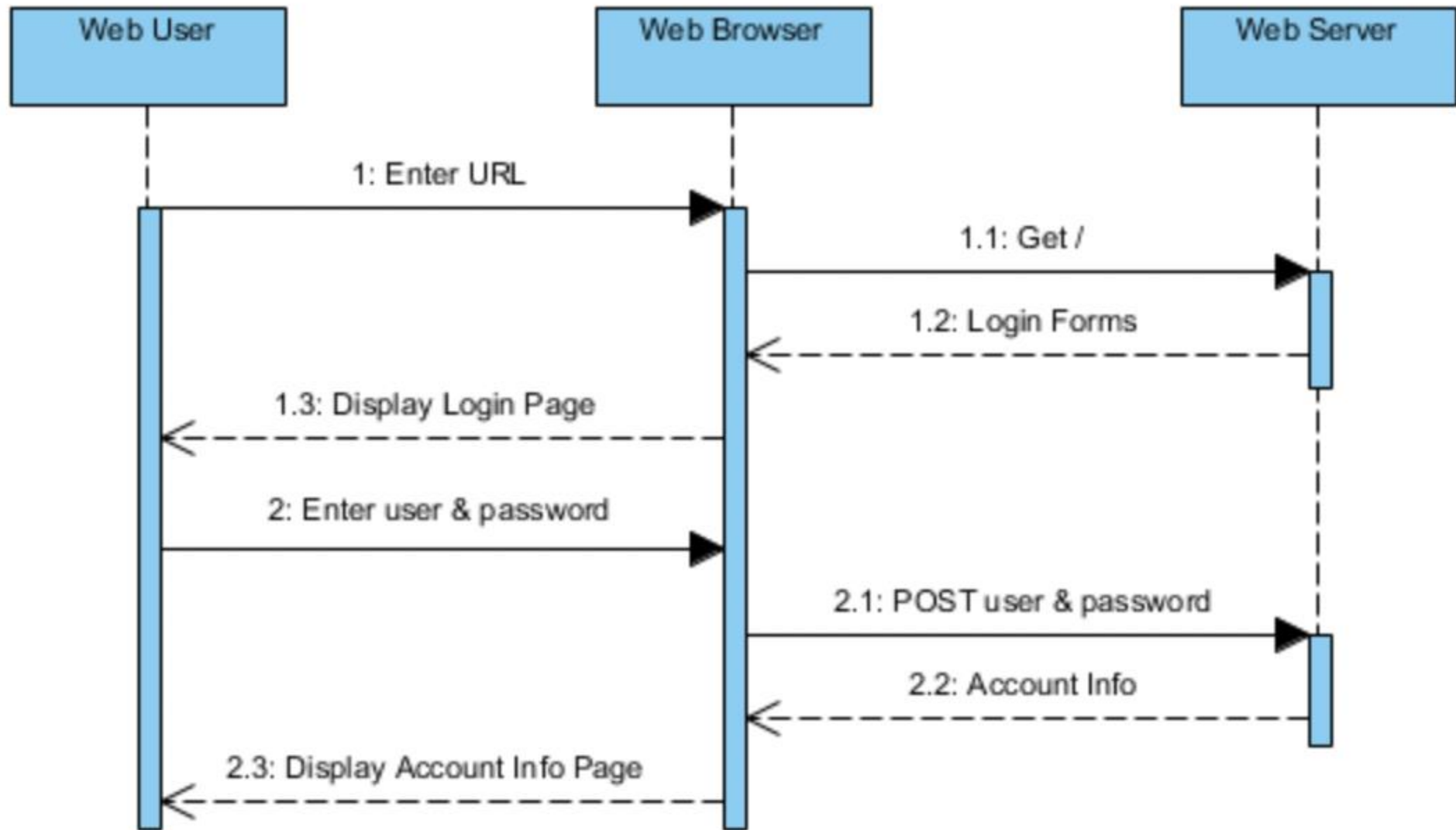
This is how I would start my sequence diagram.

Now, the steps will be provided, try to create the sequence of events, and interaction yourself to complete the sequence diagram for the login use case.



Let's first list out all the steps and then create a diagram.

- First we know once we have our web browser open, of course to visit any website, we must enter in the URL.
- Once we have entered the URL, the browser will connect to web server to fetch correct website.
- The webserver will send the login form to the web browser.
- The browser will now display the login page to the web user.
- The web user will now enter login information.
- The web browser will send this information to the server to confirm login.
- The web server will send account information to the web browser.
- Finally the browser will display account information to the web user.
- This would be the end of the interaction between a user entering in a URL to a login based webpage.



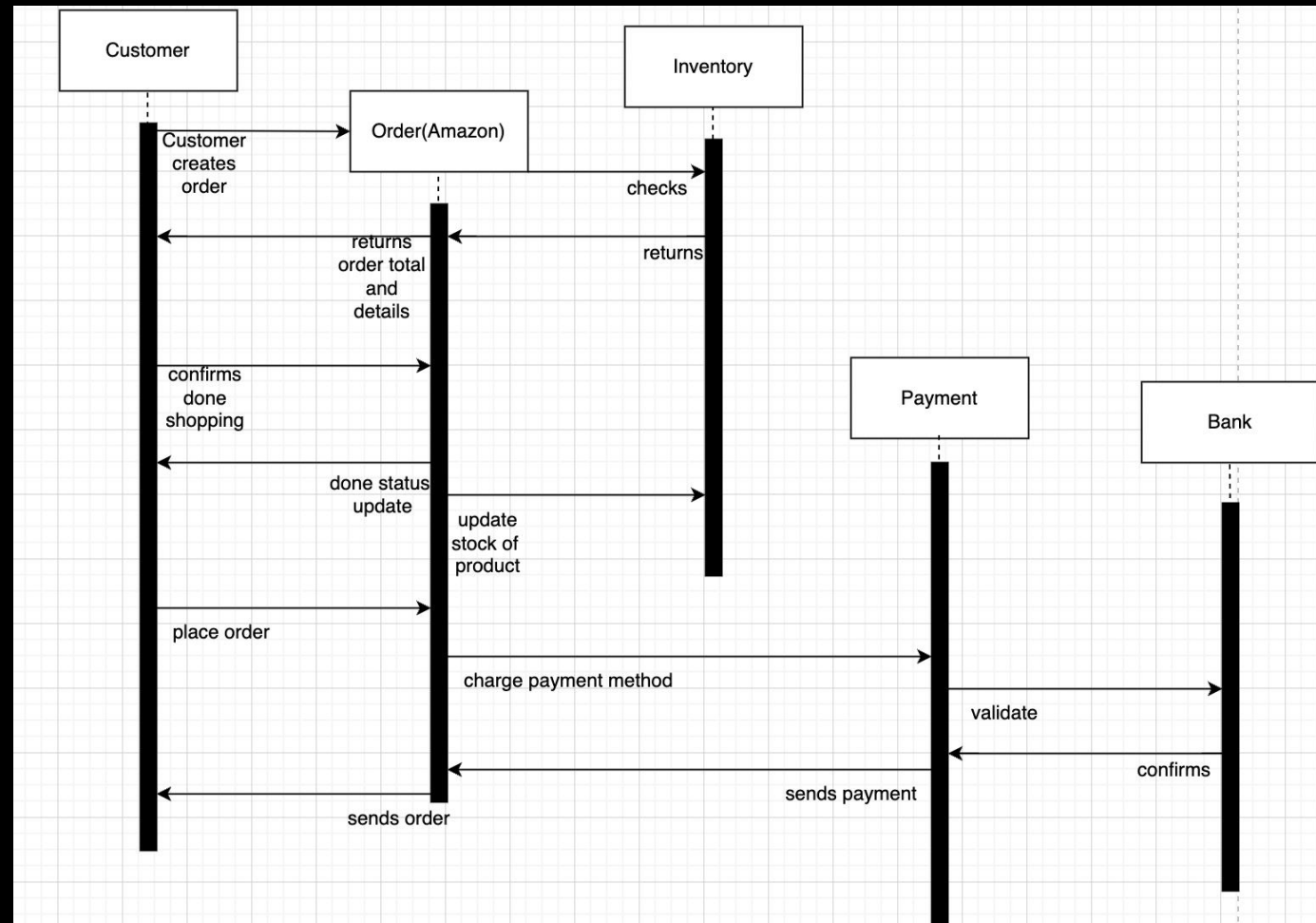
Once again, each diagram can be different for the same scenario.

Of course of different scenarios, there may be less, or more interactions between classes, there may be more classes in play.

Let's take a look at the following sequence diagram and try guessing the scenario.

Sequence Diagrams Example.

Scenario: Draw a UML sequence diagram that represents the behavior of ordering from an online shopping system such as amazon.

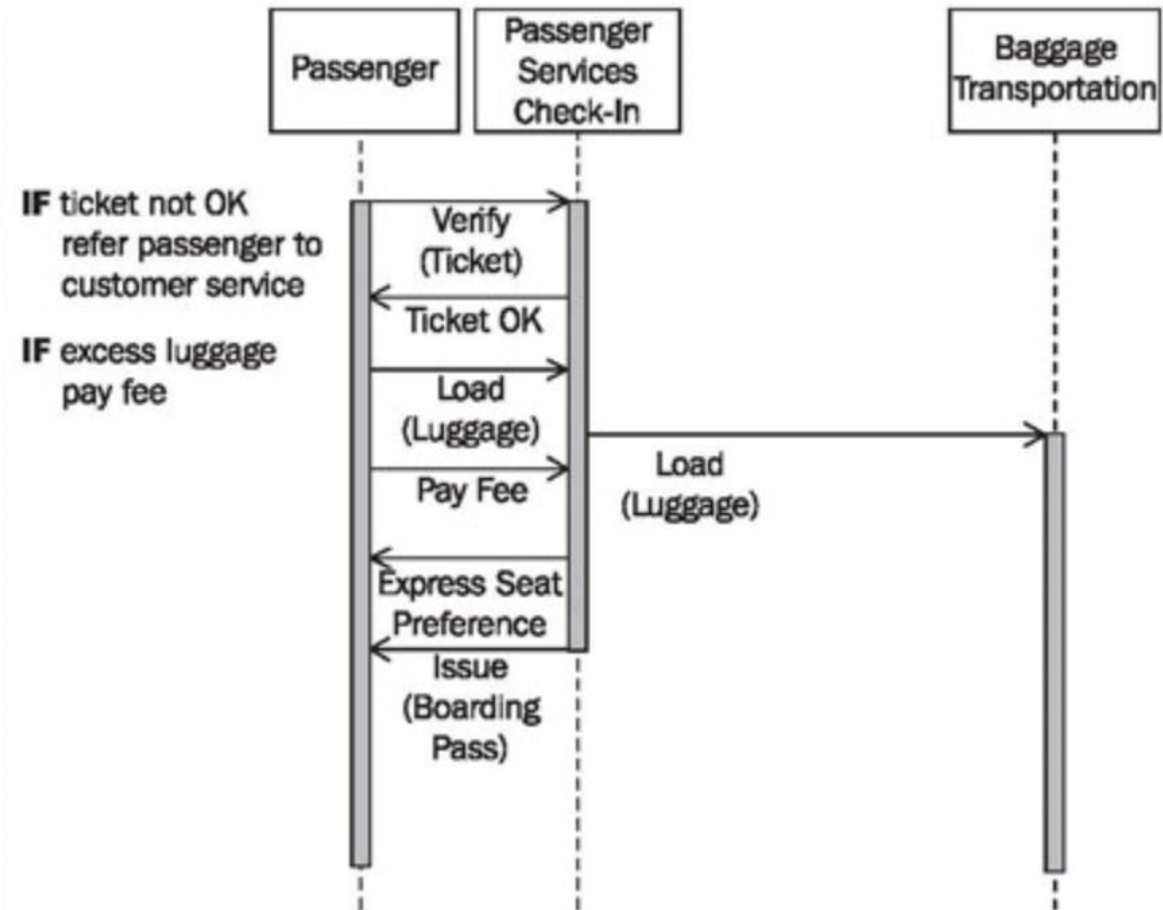


Now that we have gone over some sequence diagrams. Try one out for yourself!

Create a UML sequence diagram for the following scenario:

A passenger service check-in that checks the weight of the passengers baggage before sending the baggage to "Baggage Transportation".

Things to think about. You have passengers. They have baggages. There is a system in place to check the baggage before sending it off the transportation services. Try creating a UML sequence diagram.



We were able to use the use case diagram, take an individual use case, and create a sequence diagram for that case.

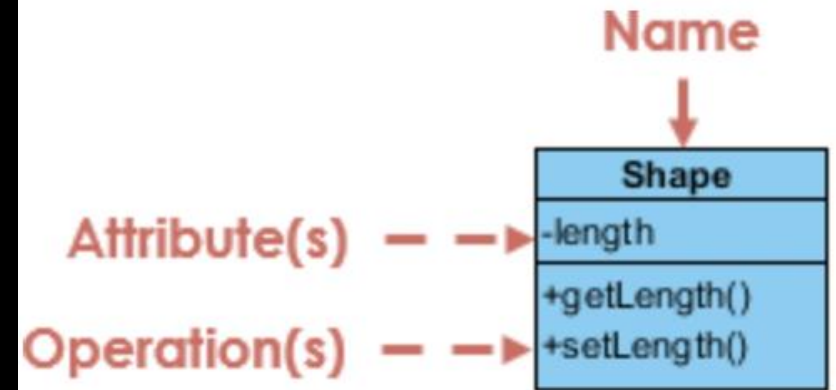
Now what about a class diagram? A use case diagram will display all potential cases an actor can face inside our system. A class diagram aims to show all possible classes the system may need. By examining what we want our actors to be able to do in our system, we can think about classes that need to be created to help satisfy this need.

Recall the UML Class Notation

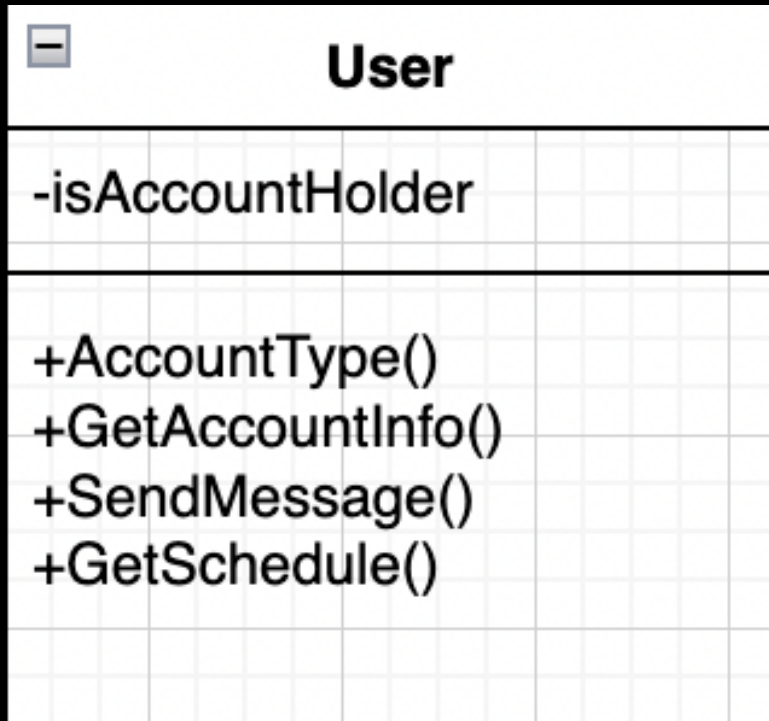
First partition contains class name.

Second partition contain attributes. Attributes describe values the class can potentially hold.

The third partition contains operations. These are the services provided by the class.




For example...



This is what the "User" class in the Class Diagram would look like for the patient doctor web portal.


In the third partition, you can see these are all the operation we expect this class to perform. Operation are implemented via methods.

 User
-isAccountHolder
+AccountType() +GetAccountInfo() +SendMessage() +GetSchedule()

So we will also have to update our class diagram with the following class to ensure we convey our point.

In User we say
"AccountType()"

So, a method? But depending on how YOU would implement this, you may create a whole new class to obtain information regarding an account.

 Account Info
-AccountType
+LoginID() +Speciality()
+setName() +getAccountType()
+getPaymentInfo()
+setNumber()
+Insurence()

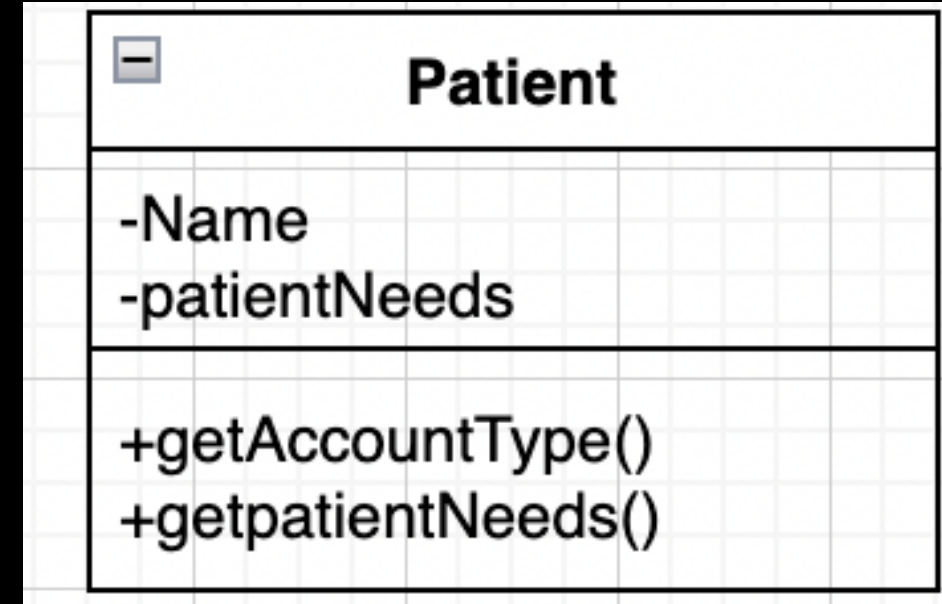
Think about user...

There will be visitors, these are users who don't have an account, yet want to view or navigate our service.

The second is patients, these are users who have signed up for our service.

Doctors are also a type of user, they are using our service.

All these users will need their own class, while they are all users, they each will have unique functionality which we will implement in their respective class.



We can create the following classes because we know we will have patient and visitors, both being users and both may have common code and functionally, however, patients are users who actually have an account so they of course would be able to do more on our website. Whatever patient can do that is unique to just patient we will add to our patient class diagram.

For example, we need to know or find out a patient's needs, this is unique to patient, whereas we don't really care about visitor, we even limit their view.

Now let's look at doctors.

Patients connect with doctors via health insurance providers. Patients might also want to view the doctors credentials. We need to be ready for these cases.

Doctors will also have accounts, which will also hold information much like most users.

Add what is unique to doctor.

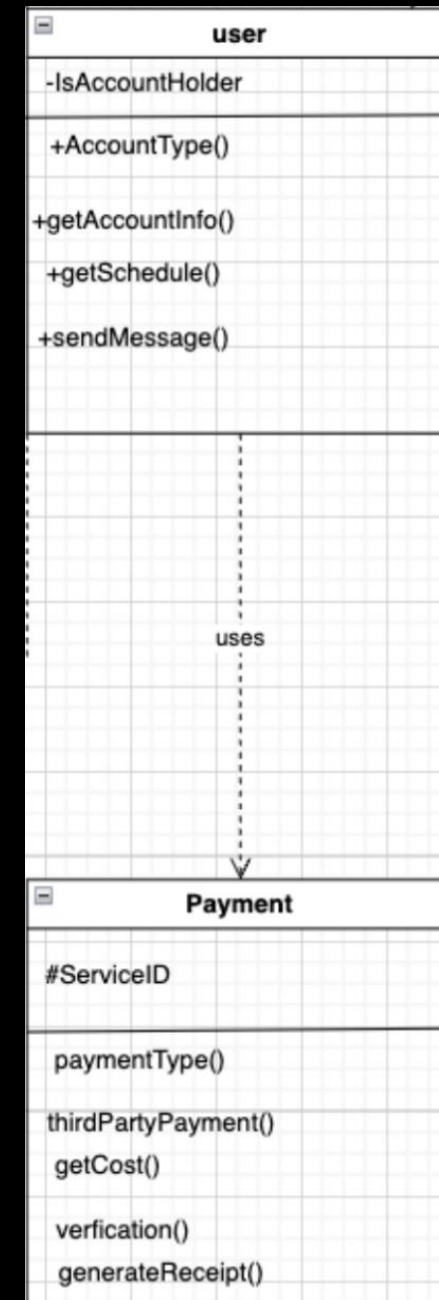
Doctor			
-accountType			
-doctorInfo			
-doctorBackGround			
+GetaccountType()			
+GetdoctorInfo()			
+GetdoctorBackGround()			

Patients will have to pay for their visit.

We will end up needing to development a payment class, this class will be used by our User class to allow any type of user to make a payment. For example maybe a doctor needs to buy more medication (just an example).

So now we must create a payment class. Payment will have nothing to do with our different types of users, however all users will be able to use this payment class. Therefore we must label the relation appropriately.

The payment class will continue to exist without user class, but doctor, patient, and visitor basically extend from user class so these class will be effected with the deletion of user. Payment is not effected at all. Therefore User uses Payment class.



Our users must be able to send messages, view their schedule. So we know we will need to provided classes such as schedule and messages which our users will be using those classes.

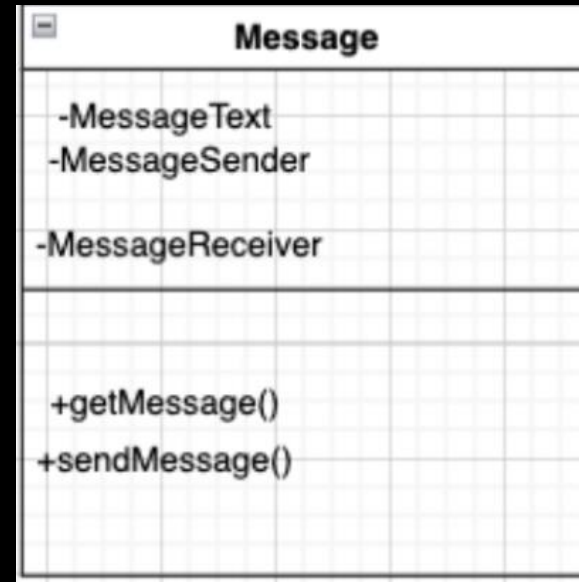
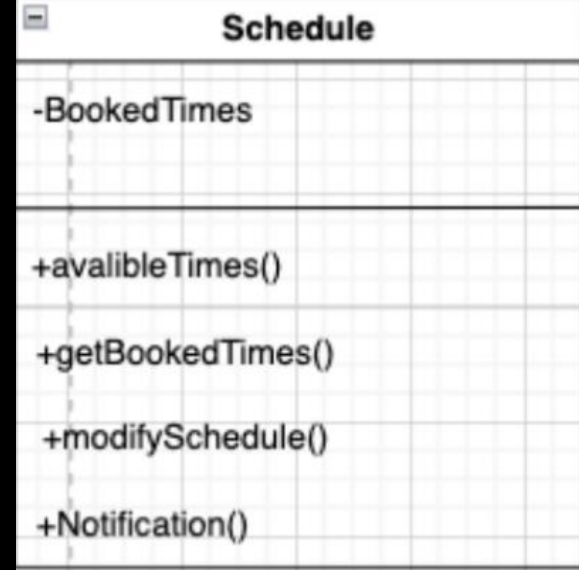
user	
-IsAccountHolder	
+AccountType()	
+getAccountInfo()	
+getSchedule()	
+sendMessage()	

We must also create classes for those two particular functions.

Doctors, Admins, and Patients may all receive messages, but we need to make sure the right message reaches the right people. For this we create a messages class.

This allow promotes making our code reusable. We want only one messages call that can handle and manage all types of messages all types of users.

Same for schedule. For example, a patient may only have one visit on their schedule since they'll only see one doctor, however, doctors we multiple patients. We want to reuse the code that would allow patient and doctor to view their own respectable schedules.

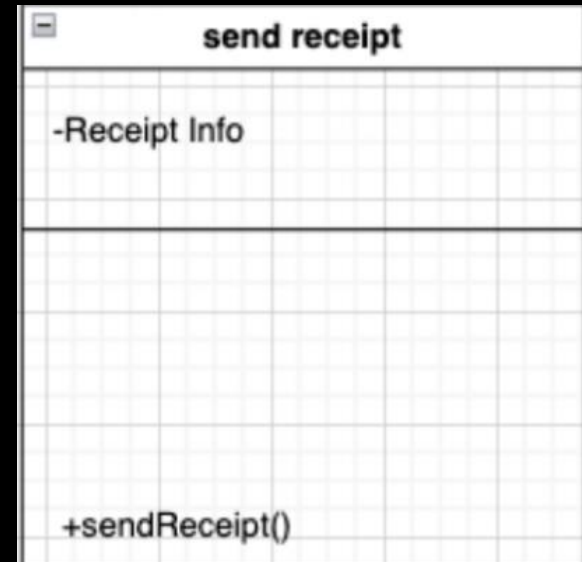
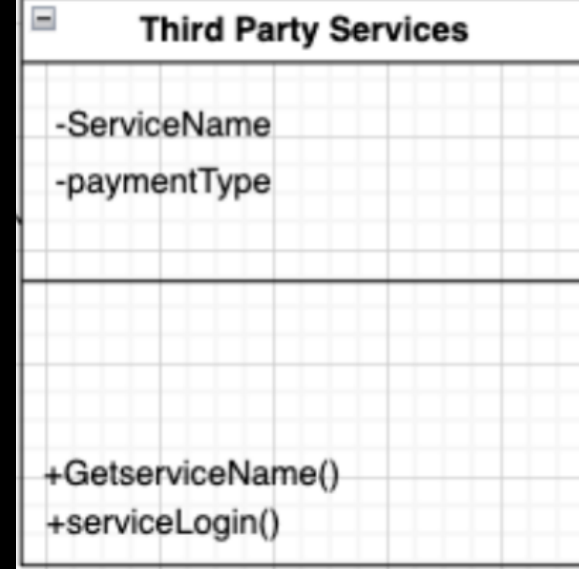


Include as many details into your diagram as you can.

We know with payment, you can also have third party services such as paypal. So we need to include a class which includes the addition of third party services. This class will have a direct relation with the payment class.

Normally when you schedule something, you might get a notification regarding your appointment, so we can include that as well.

When you purchase something, you are normally emailed a receipt, so a receipt class will also be necessary.



Now that we have thought about almost everything. How would we connect the different classes in one class diagram?

Let's first think about what classes were created due to the existence of another class. Or classes that have "shared ownership relation". We have class user, and because of this, we needed to create, class, patient, visitor, and doctor to help differentiate all the different types of users we may have. These will be symbolized with a clear arrow symbol.

Another type of arrow we will use is a black diamond arrow, this will represent classes that are strongly independent.

End Result:

