

DEPARTMENT OF SOFTWARE ENGINEERING

NATIONAL UNIVERSITY OF MODERN LANGUAGES | ISLAMABAD

SOFTWARE DESIGN AND ARCHITECTURE (SESD-202)

LAB MANUAL

SOFTWARE DESIGN AND ARCHITECTURE

(COURSE CODE)

**OBJECTIVES**

* To acquaint students with better understanding of Design and Analysis
* To familiarize students with the importance of design in developing a software
* To equip students with hands-on experience of tracing requirements to design and then design to code

**INTRODUCTION**

This lab is about software design and architecture, which is the core course for learning how to design your software/system. In this lab students will learn about designing and architecture of software from basic and they will also learn the tools on which they can design the architecture or basic UML of software system.

**DESIGN SKILLS / TECHNIQUES PRACTICED**

Place content related to design skills / techniques practiced in your course and lab here.

**SOFTWARE TOOLS / TECHNOLOGY INVOLVED**

* MS Visio

**EFFECTIVENESS**

This lab provides hands-on experience of designing different level of a software from a very basic to expert level. In this lab students will able to design different software, understand the system with real-world environment. The last but the main objective of this lab is to make students able to trace the requirements they are given with the design and also trace the design to their code.

SOFTWARE DESIGN AND ARCHITECTURE (COURSE CODE) 

DEPARTMENT OF SOFTWARE ENGINEERING

NATIONAL UNIVERSITY OF MODERN LANGUAGES ISLAMABAD

**LAB OUTLINE**

|  |  |
| --- | --- |
| **Week #** | **Topics to be covered** |
| **1** | Introduction to UML (Unified Modeling Language), Static and Dynamic UML Diagrams, Introduction to Domain Model |
| **2** | Domain Model, Vocabulary, Notations, and Roles | Goals, Domain Model Relationships | Diagram, Examples |
| **3** | Introduction to Use Case, Vocabulary, Notations, Functional Requirements, Use case Diagram with Examples |
| **4** | Use Case Description, User interaction with the system, Tabular Notations, Examples |
| **5** | Class Diagram, Relationships, Cardinality, Functions and attributes, Class Diagram Examples |
| **6** | Sequence Diagram, Notations, Lifelines, Examples |
| **7** | Continuation of Sequence Diagram, If-else statements, parallel states, More Examples |
| **8** | Deployment Diagram, Notations, Examples |
| **9** | Introduction to Architecture Design, Design Patterns, Architecture Styles, High level and low level designs, Examples |
| **10** | Gang of Four (GoF) design patterns, examples |
| **11** | Style Attributes, factory Fasade, Singleton, Examples |
| **12** | MVC (Model View Controller), Real world scenarios Examples |
| **13** | System according to MVC-Design level |
| **14** | Revision |
| **15** | Revision |
| **16** | Lab Exam |

TABLE OF CONTENTS

[**LAB 1: Introduction to UML and Domain Model 1**](#_heading=h.1t3h5sf)

[**Purpose** 1](#_heading=h.2s8eyo1)

[**Theoretical Description** 1](#_heading=h.17dp8vu)

[**LAB REPORT-01** 2](#_heading=h.3rdcrjn)

[**LAB 2: Domain Model 3**](#_heading=h.26in1rg)

[**Purpose** 3](#_heading=h.lnxbz9)

[**Theoretical Description** 3](#_heading=h.35nkun2)

[**Lab Report-02** 4](#_heading=h.1ksv4uv)

[**LAB 3: Use Case Diagram 5**](#_heading=h.44sinio)

[**Purpose** 5](#_heading=h.2jxsxqh)

[**Theoretical Description** 5](#_heading=h.z337ya)

[**Lab Report-03** 6](#_heading=h.3j2qqm3)

[**LAB 4: Use Case Description 7**](#_heading=h.1y810tw)

[**Purpose** 7](#_heading=h.4i7ojhp)

[**Theoretical Description** 7](#_heading=h.2xcytpi)

[**Lab Report-04** 9](#_heading=h.1ci93xb)

[**LAB 5: Class Diagram 10**](#_heading=h.3whwml4)

[**Purpose** 10](#_heading=h.2bn6wsx)

[**Theoretical Description** 10](#_heading=h.qsh70q)

[**Lab Report-05** 12](#_heading=h.3as4poj)

[**LAB 6: Sequence Diagram 13**](#_heading=h.1pxezwc)

[**Purpose** 13](#_heading=h.49x2ik5)

[**Theoretical Description** 13](#_heading=h.2p2csry)

[**Lab Report-06** 16](#_heading=h.147n2zr)

[**LAB 7: Sequence Diagram (Continued) 17**](#_heading=h.3o7alnk)

[**Purpose** 17](#_heading=h.23ckvvd)

[**Theoretical Description** 17](#_heading=h.ihv636)

[**Lab Assignment** 17](#_heading=h.32hioqz)

[**LAB 8: Deployment Diagram 18**](#_heading=h.1hmsyys)

[**Purpose** 18](#_heading=h.41mghml)

[**Theoretical Description** 18](#_heading=h.2grqrue)

[**Lab Report-08** 19](#_heading=h.vx1227)

[**LAB 9 | 10: Software Architecture, Gang of Four(GoF), Design Patterns 20**](#_heading=h.3fwokq0)

[**Purpose** 20](#_heading=h.1v1yuxt)

[**Theoretical Description** 20](#_heading=h.4f1mdlm)

[**Lab Report-09|10** 24](#_heading=h.2u6wntf)

[**LAB 10 | 11: Model View Controller (MVC) 25**](#_heading=h.19c6y18)

[**Purpose** 25](#_heading=h.3tbugp1)

[**Theoretical Description** 25](#_heading=h.28h4qwu)

[**Lab Report-11|12** 26](#_heading=h.nmf14n)

# **LAB 1: Introduction to UML and Domain Model**

## 

## **Purpose**

The purpose of this lab is to build the understanding of design and importance of designing in the development of a software.

## **Theoretical Description**

This section contains theoretical description of some basic concepts of UML and basics of Domain Model.

**WHAT IS UML?**

Unified Modeling Language (UML) is a general purpose modelling language. The main aim of UML is to define a standard way to visualize the way a system has been designed. It is quite similar to blueprints used in other fields of engineering.

UML is not a programming language; it is rather a visual language. We use UML diagrams to portray the behavior and structure of a system. UML helps software engineers, businessmen and system architects with modelling, design and analysis. The Object Management Group (OMG) adopted Unified Modelling Language as a standard in 1997. It’s been managed by OMG ever since.  International Organization for Standardization (ISO) published UML as an approved standard in 2005. UML has been revised over the years and is reviewed periodically.

**NEED OF UML:**

Complex applications need collaboration and planning from multiple teams and hence require a clear and concise way to communicate amongst them. Businessmen do not understand code. So UML becomes essential to communicate with non-programmers’ essential requirements, functionalities and processes of the system. A lot of time is saved down the line when teams are able to visualize processes, user interactions and static structure of the system.

UML is linked with object orienteddesign and analysis. UML makes the use of elements and forms associations between them to form diagrams.

**DOMAIN MODEL**

A **domain model** is a visual representation of conceptual classes or real - situation objects in a domain. Domain models have also been called **conceptual models**, **domain object models,** and **analysis object models.**

**VOCABULARY OF ORDER PROCESSING SYSTEM:**

|  |  |
| --- | --- |
| **ROLE** | **OPERATION** |
| Customer | Customer id,Customer Name,Phone Number,Address. |
| Order | Order id,Customer id,Customer Name,Product  id,Amount,OrderDate. |
| Stock | Product id,Quantity,Shop Number. |
| Product | Product id,Product Type,Product Price. |
| Manager | Manage Bill,Maintain Record,Assign Duties,Hire  Staff,Check ProductExpiryDate,Make Ordering List. |
| Supplier | Provide Quality Product |

## **LAB REPORT-01**

* Write an introduction to UML.
* Which diagrams comes under static and dynamic UML diagrams
* Identify the actors and their functions/roles involves in “Hospital Management System”/ “Student Management System”

# **LAB 2: Domain Model**

## **Purpose**

The purpose of this lab is to develop the understanding of system are see its compatibility with real world. In this lab students will able to design domain model of any given system on designing tool.

## **Theoretical Description**

A domain model is a representation of real-world conceptual classes, not of software components. Domain modeling is a technique used to understand the project problem description and to translate the requirements of that project into software components of a solution. The software components are commonly implemented in an object oriented programming language. A domain model contains conceptual classes, associations between conceptual classes, and attributes of a conceptual class. Informally, a conceptual class is an idea, thing, or object.

Relationships that are occurred within these conceptual classes are: one to one, one to many, and many to many.

The goal of domain model is to understand the real-world circumstances, understanding the vocabulary of the system, actors involve to the system, what action or roles they are going to perform. All these things are the part of domain model.

There are certain conventions that can be used to identify classes

**Using natural language analysis**

* Pioneered by Russell Abbott (1983), popularized by Grady Booch
* Not perfect, but coupled with other techniques, it's a good start
* This can be done from a general problem description, or better, from a use case or scenario
* Map parts of speech to object model components.
  + nouns usually map to classes, objects, or attributes
  + verbs usually map to operations or associations

|  |  |  |
| --- | --- | --- |
| **Part of speech** | **model component** | **Examples** |
| Proper noun | Instance (object) | Alice, Ace of Hearts |
| Common noun | Class (or attribute) | Field Officer, PlayingCard, value |
| Doing verb | Operation | Creates, submits, shuffles |
| Being verb | Inheritance | Is a kind of, is one of either |
| Having verb | Aggregation/Composition | Has, consists of, includes |
| Modal verb | Constraint | Must be |
| Adjective | Helps identify an attribute | a *yellow* ball (i.e. color) |

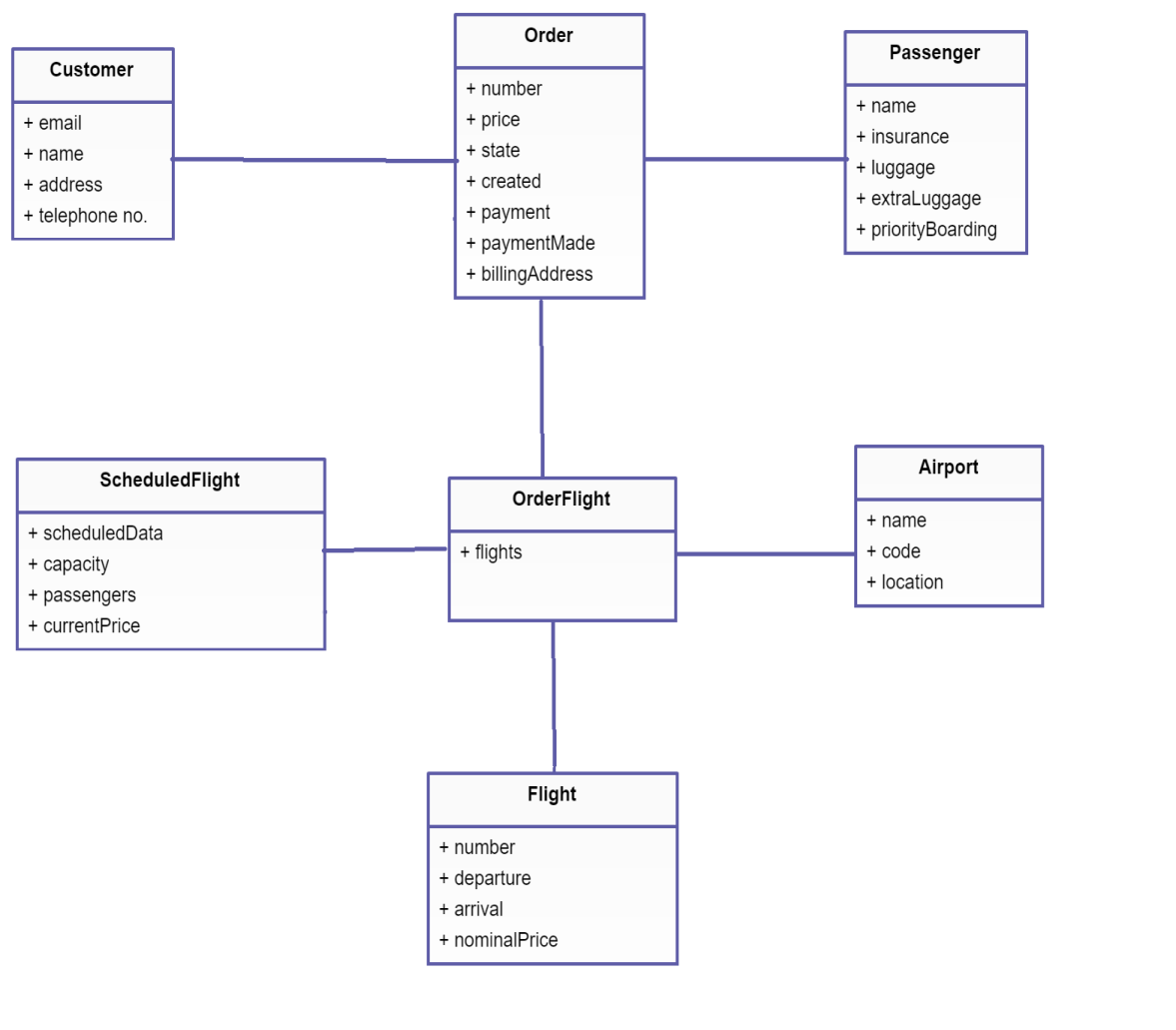
For practice view below link

<https://www.ooportal.com/problem-analysis/module2/identify-classes-objects.php>

The table below represents the vocabulary which includes roles/actors and operations/functions of “Flight Reservation System”

|  |  |
| --- | --- |
| **Roles** | **Operations** |
| Customer | Customer Name, ID, email, age, contact no |
| Passenger | Passenger Name, Passport no, age, Contact, trip info |
| Airplane | Airplane Model, Capacity, Age, Maintenance record, |
| Flight | Flight no, Duration, Cargo info, passenger info |
| Schedule | Flight route, arrival, destination, stay |
| airport | Airport name, code, location, schedule |

The figure below represents the domain model for “Flight Reservation System”



## **Lab Report-02**

1. Make the complete domain model of “Online Shopping System” or “Order Processing System”

# **LAB 3: Use Case Diagram**

## **Purpose**

The purpose of this lab is to identify the functional requirements of the system, primary and secondary actors of the system and make use case model for it.

## **Theoretical Description**

**Basic Use Case Diagram Symbols and Notations**

The Use Case diagram captures systembehavior from the user's point of view. The purpose of the diagram is to describe the features of the system that the client or we expect. The diagram describes what the system should be able to dobutdoes not say how the system will do it**.** Therefore, it's usually the first diagram we create when designing an information system. It's important to agree on what our system (or application, game, anything) should do at first. Only then it makes sense to ask how we're going to actually do it.

* **System**

Draw your system's boundaries using a rectangle that contains use cases. Place actors outside the system's boundaries.



* **Use Case**

Draw use cases using ovals. Label the ovals with verbs that represent the system's functions.

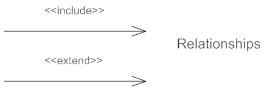


* **Actors**

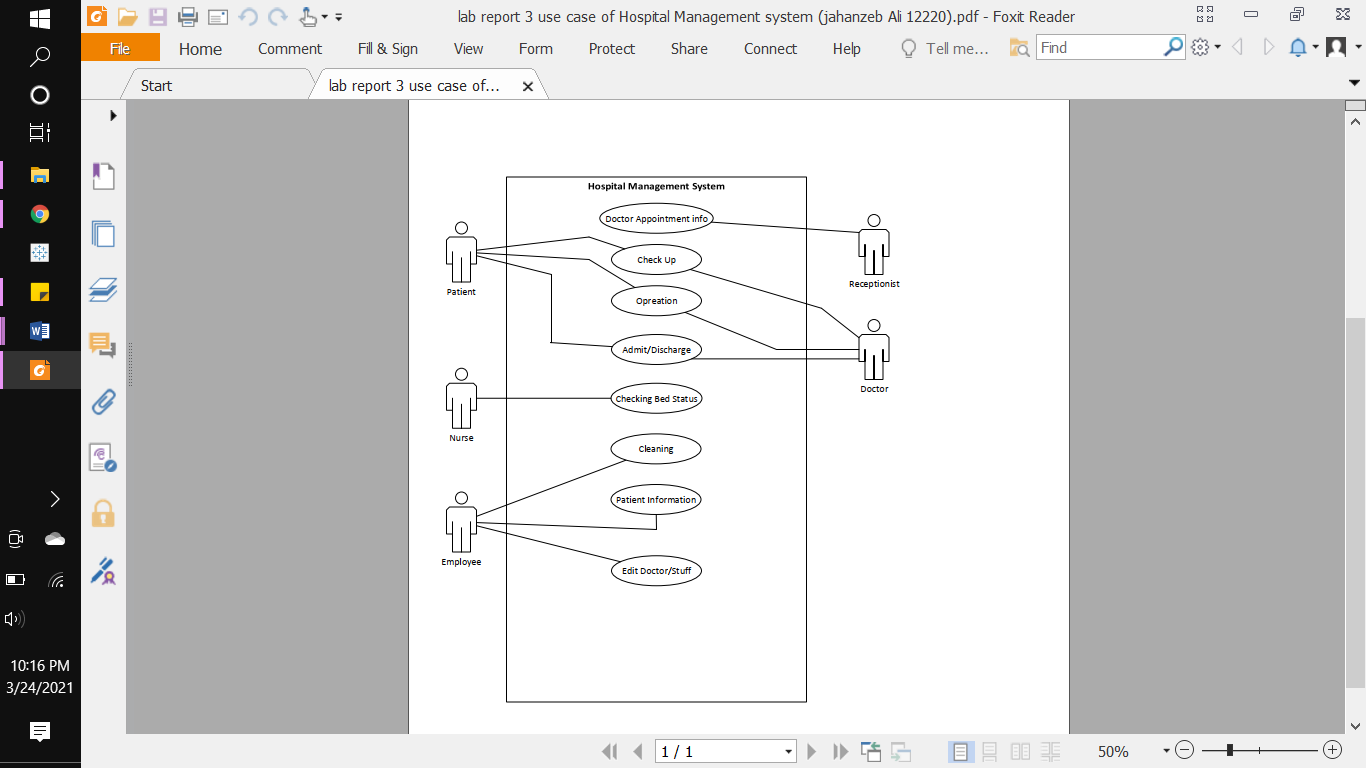
Actors are the users of a system. When one system is the actor of another system, label the actor system with the actor stereotype.



* **Relationships**  
  Illustrate relationships between an actor and a use case with a simple line. For relationships among use cases, use arrows labeled either "uses" or "extends." A "uses" relationship indicates that one use case is needed by another in order to perform a task. An "extends" relationship indicates alternative options under a certain use case.



**USE CASE MODEL for “Hospital Management System”**



## **Lab Report-03**

1. Make Use Case Model for “Facebook” or “Student Management System”

# **LAB 4: Use Case Description**

## **Purpose**

The purpose of this lab is to describe the use case, their interaction with the system and response of the systems with alternate scenarios.

## **Theoretical Description**

A use case description is a text-based narrative of a functionality comprised of detailed, step-by-step interaction between the actor and the system. It describes the outcomes of an action taken to accomplish a specific goal.

Use case description involves these main components:

1. Use Case Number | Use Case Name
2. Actor involve
3. Goal of the use case
4. Pre-requisite of the use case
5. Description of use case which includes user action and system response
6. Alternate scenario of the use case

Let’s consider an example of ATM machine for elaborate the use case description.

For an ATM machine scenario these use cases are involved:

* + Validate PIN
  + Withdrawal | Transaction
  + Deposit
  + Check Balance

|  |  |  |
| --- | --- | --- |
| **UC-01** | **Validate PIN** | |
| **Actor** | Customer | |
| **Goal** | Verify PIN number | |
| **Pre-requisite** | User must have a valid PIN code | |
| **Description** | **User Action** | **System Response** |
| * User enters their card * User Enter a valid PIN number | * System accepts card and ask for PIN * Verify entered PIN number |
| **Alternate** | * If a user enters wrong PIN, then re-enter PIN code * If a user enters wrong PIN for 3times, then card is blocked | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **UC-02** | **Withdraw Cash** | | | |
| **Actor** | Customer | | | |
| **Goal** | User must enter a valid amount of Cash | | | |
| **Pre-requisite** | UC-01 | | | |
| **Description** | **User Action** | | **System Response** | |
| * After enter a valid PIN code * user enter a valid amount for withdrawal | | * System after verifying PIN, ask for withdrawal amount * If the amount is valid and available in account, cash is transacted. | |
| **Alternate** | * User must re-enter a valid amount of cash if enter wrong | | | |
| **UC-03** | | **Deposit Cash** | | |
| **Actor** | | Customer | | |
| **Goal** | | User must have cash to deposit | | |
| **Pre-requisite** | | UC-01 | user must have an account | | |
| **Description** | | **User Action** | | **System Response** |
| * User have a valid account * Give cash to deposit | | * Verify account * Cash Deposited |
| **Alternate** | | * If a user provided wrong account number then he/she must provide valid account number | | |

|  |  |  |
| --- | --- | --- |
| **UC-04** | **Check Balance** | |
| Actor | Customer | |
| Goal | User needs to check their account balance | |
| Pre-requisite | UC-01 | |
| Description | **User Action** | **System Response** |
| * User Enter a Valid PIN code * Enter desired query | * Verifies PIN code * System grants access to that requested query |

## **Lab Report-04**

* Write down the use case description for “Flight Reservation System” or “Online Food Ordering System”

# **LAB 5: Class Diagram**

## **Purpose**

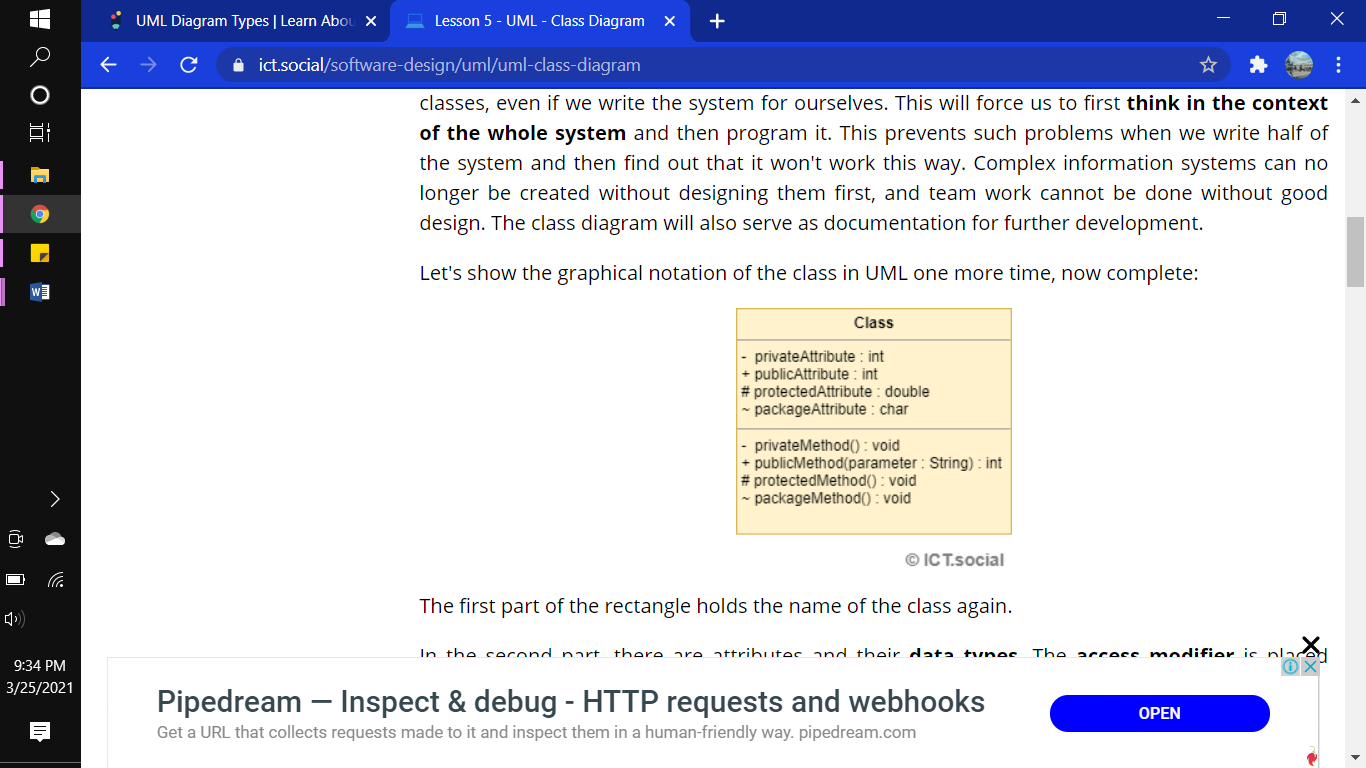
The purpose of this lab is to identify the syntax class diagram attributes and functions of each conceptual class.

## **Theoretical Description**

Class diagrams are the main building block of any object-oriented solution. It shows the classes in a system, attributes, and operations of each class and the relationship between each class.

In most modeling tools, a class has three parts. Name at the top, attributes in the middle and operations or methods at the bottom. In a large system with many related classes, classes are grouped together to create class diagrams. Different relationships between classes are shown by different types of arrows.

The below figure shows how a class diagram can be created.



The first part of the rectangle holds the name of the class again.

In the second part, there are attributes and their **data types**. The **access modifier** is placed before each attribute. We have 4 options:

* **- (minus)** - Private attribute.
* **+ (plus)** - Public attribute.
* **# (hash cross)** - Protected attribute.
* **~ (tilde)** - Attribute visible from within the package.

Four types of **relations** occur between different classes. These relations are:

* **Association**

An association is the basic relationship between two entities. These entities can exist independently of each other. We draw it as a simple solid line. The association is bi-directional by default. That means that the first entity has a reference to the other, and the other to the first. We can change this behavior by adding a simple arrow specifying the direction of the relationship. Only the instance from which the arrow points stores the reference to the other entity in these cases.

* **Aggregation**

Aggregation represents the relationship between a whole and its parts. We draw it as a solid line with an empty diamond shape. The diamond is drawn at the class representing the whole (e.g. an article section). From the implementation point of view, this is the entity that holds the item collection. An entity representing the part can exist independently and be part of other collections.

* **Composition**

A composition is similar to aggregation, but it represents a stronger relationship. The entity representing the part has no sense without the entity representing the whole. If the entity representing the whole is removed, its parts are automatically removed as well. We draw the composition relationship like the aggregation, but the diamond shape is filled. The multiplicity of the entity representing the whole must always be 1. This relationship is confusing and I'd rather avoid it and replace it with aggregation.

* **Generalization**

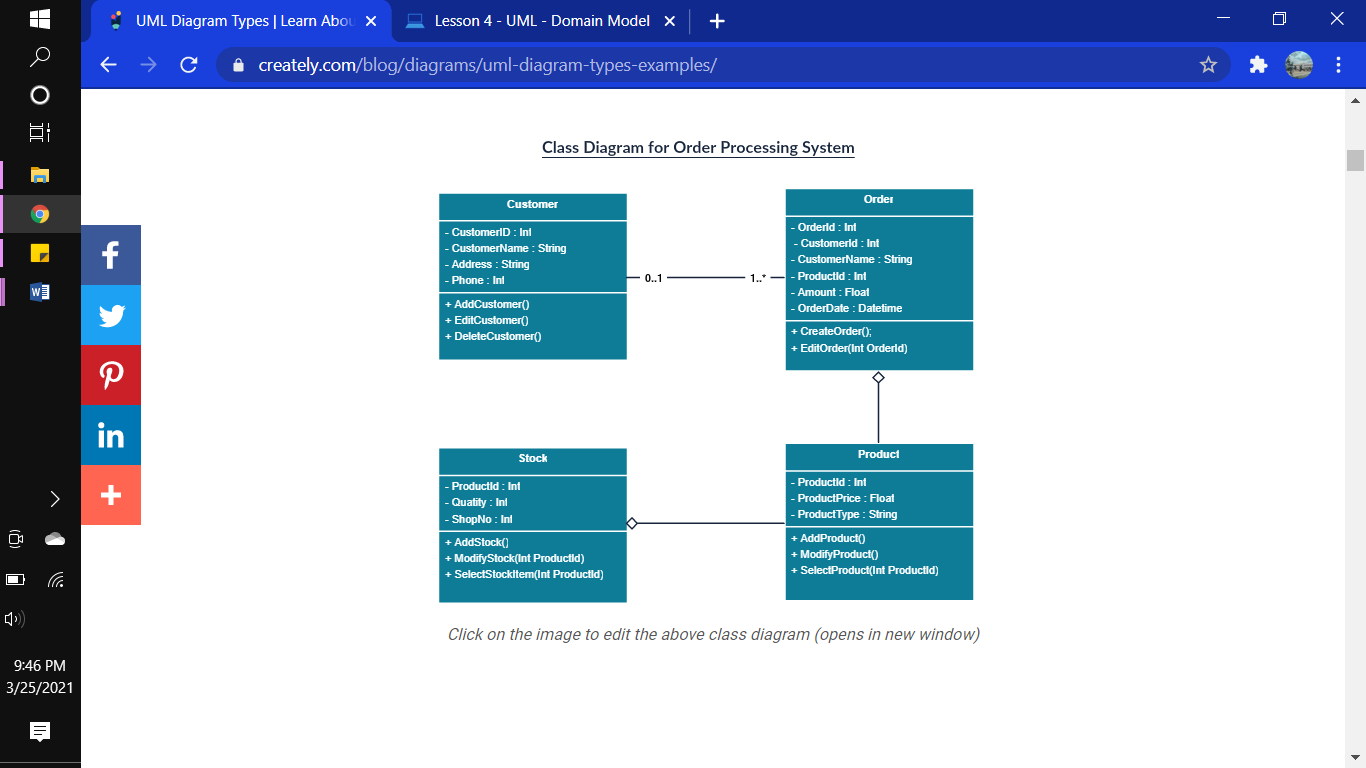
In terms of implementation, it represents inheritance. One entity inherits the properties and behavior from another entity. We've already seen this kind of relationship in the use case diagram. We draw the generalization as a solid line with an empty arrow on one side (a triangle if you like). The arrow is on the side of the entity from which it's inherited.

* **Multiplicity**

We can specify the multiplicity for association, aggregation, and composition (for composition on one side only). Let's go back to the example with a section and an article: We read the multiplicity here as follows: A section can contain any number of articles (this is shown by the asterisk at the Article class). An Article belongs to 1 or more sections (it's shown by 1..\* at the Section). Let's list available multiplicity syntax:

1. **1 (number)** - Indicates a specific value (1 in this example).
2. **\* (asterisk)** - Indicates any number (even 0). Instead of an asterisk, we can find the N symbol in some diagrams.
3. **1..\* (interval)** - We can specify an interval with 2 dots. Then we use the symbols we already known, such as: 2..6 or 1..\* or 0..1.

Let’s take a simple example of ‘Order Processing System’ for making class diagram



## **Lab Report-05**

1. Make a class diagram for “POS”.

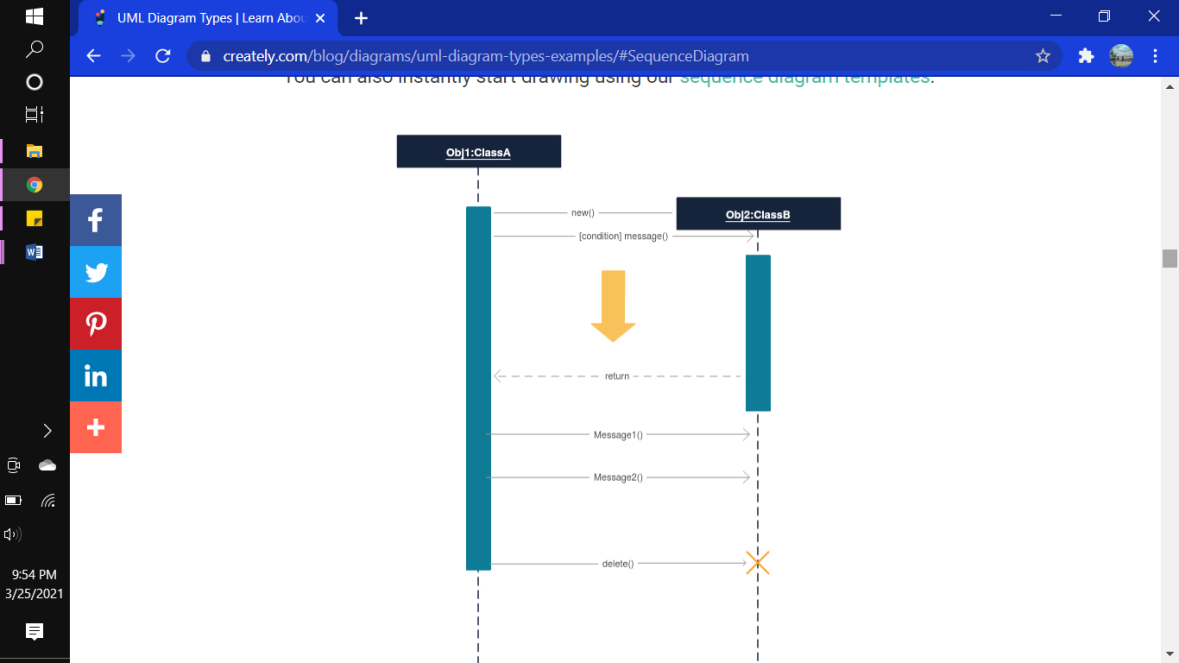
# **LAB 6: Sequence Diagram**

## **Purpose**

The purpose of this lab is to make sure that the students learn about sequence diagram, dynamic responses of the system.

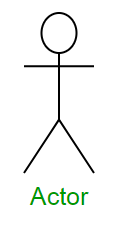
## **Theoretical Description**

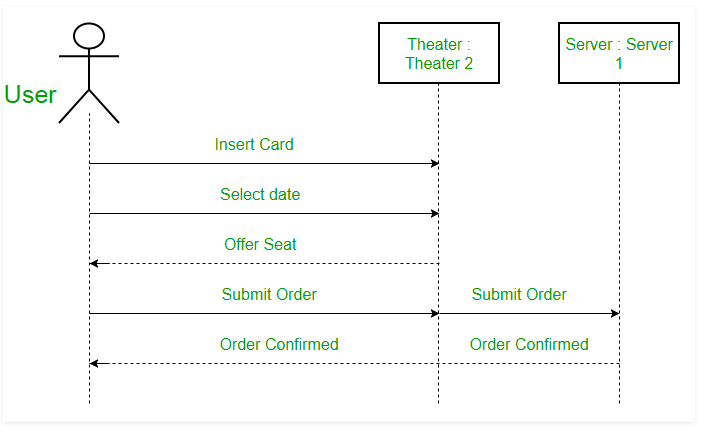
Sequence diagrams in UML show how objects interact with each other and the order those interactions occur. It’s important to note that they show the interactions for a particular scenario. The processes are represented vertically and interactions are shown as arrows.The diagram below shows how to make a sequence diagram.

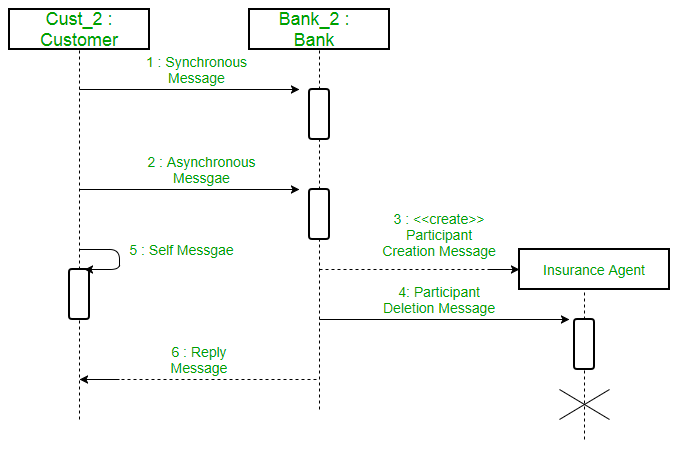


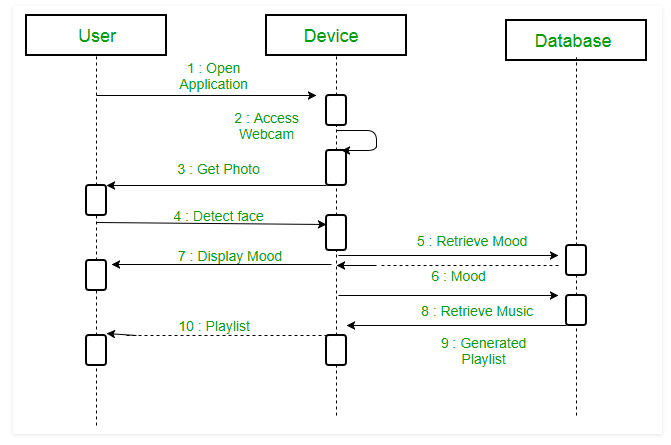
To understand what a sequence diagram is, you should be familiar with its symbols and components. Sequence diagrams are made up of the following icons and elements:

**Sequence Diagram Notations –**

1. **Actors** – An actor in a UML diagram represents a type of role where it interacts with the system and its objects. It is important to note here that an actor is always outside the scope of the system we aim to model using the UML diagram.  
     
   We use actors to depict various roles including human users and other external subjects. We represent an actor in a UML diagram using a stick person notation. We can have multiple actors in a sequence diagram.  
   For example – Here the user in seat reservation system is shown as an actor where it exists outside the system and is not a part of the system.
2. **Lifelines –** A lifeline is a named element which depicts an individual participant in a sequence diagram. So basically each instance in a sequence diagram is represented by a lifeline. Lifeline elements are located at the top in a sequence diagram. The standard in UML for naming a lifeline follows the following format – Instance Name : Class Name  
   We display a lifeline in a rectangle called head with its name and type. The head is located on top of a vertical dashed line (referred to as the stem) as shown above. If we want to model an unnamed instance, we follow the same pattern except now the portion of lifeline’s name is left blank.



1. **Messages –** Communication between objects is depicted using messages. The messages appear in a sequential order on the lifeline. We represent messages using arrows. Lifelines and messages form the core of a sequence diagram.  
   Messages can be broadly classified into the following **categories**:
   * **Synchronous messages –** A synchronous message waits for a reply before the interaction can move forward. The sender waits until the receiver has completed the processing of the message. The caller continues only when it knows that the receiver has processed the previous message i.e. it receives a reply message. A large number of calls in object oriented programming are synchronous. We use a solid arrow head to represent a synchronous message.
   * **Asynchronous Messages –** An asynchronous message does not wait for a reply from the receiver. The interaction moves forward irrespective of the receiver processing the previous message or not. We use a lined arrow head to represent an asynchronous message.  
     

A sequence diagram for an emotion based

## **Lab Report-06**

* Make sequence diagram for “”

# **LAB 7: Sequence Diagram (Continued)**

## **Purpose**

The purpose of this lab is to extend and elaborate sequence diagram.

## **Theoretical Description**

In this part of extended sequence diagram we will cover if-else statements in sequence diagram and parallel sequences.

## **Lab Assignment**

1. Make sequence diagram for “”

# **LAB 8: Deployment Diagram**

## **Purpose**

The purpose of this lab is to develop the understanding of system with server and on deployed machine.

## **Theoretical Description**

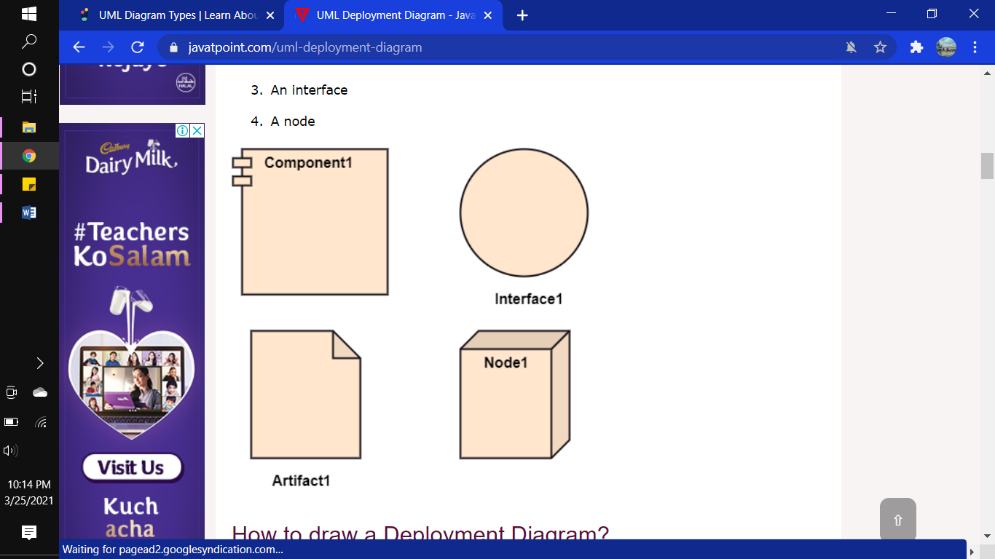
A deployment diagram shows the hardware of your system and the software in that hardware. Deployment diagrams are useful when your software solution is deployed across multiple machines with each having a unique configuration.

The deployment diagram visualizes the physical hardware on which the software will be deployed. It portrays the static deployment view of a system. It involves the nodes and their relationships.

It ascertains how software is deployed on the hardware. It maps the software architecture created in design to the physical system architecture, where the software will be executed as a node. Since it involves many nodes, the relationship is shown by utilizing communication paths.

The deployment diagram consists of the following notations:

1. A component
2. An artifact
3. An interface
4. A node



Now let’s discuss how to make a deployment diagram:

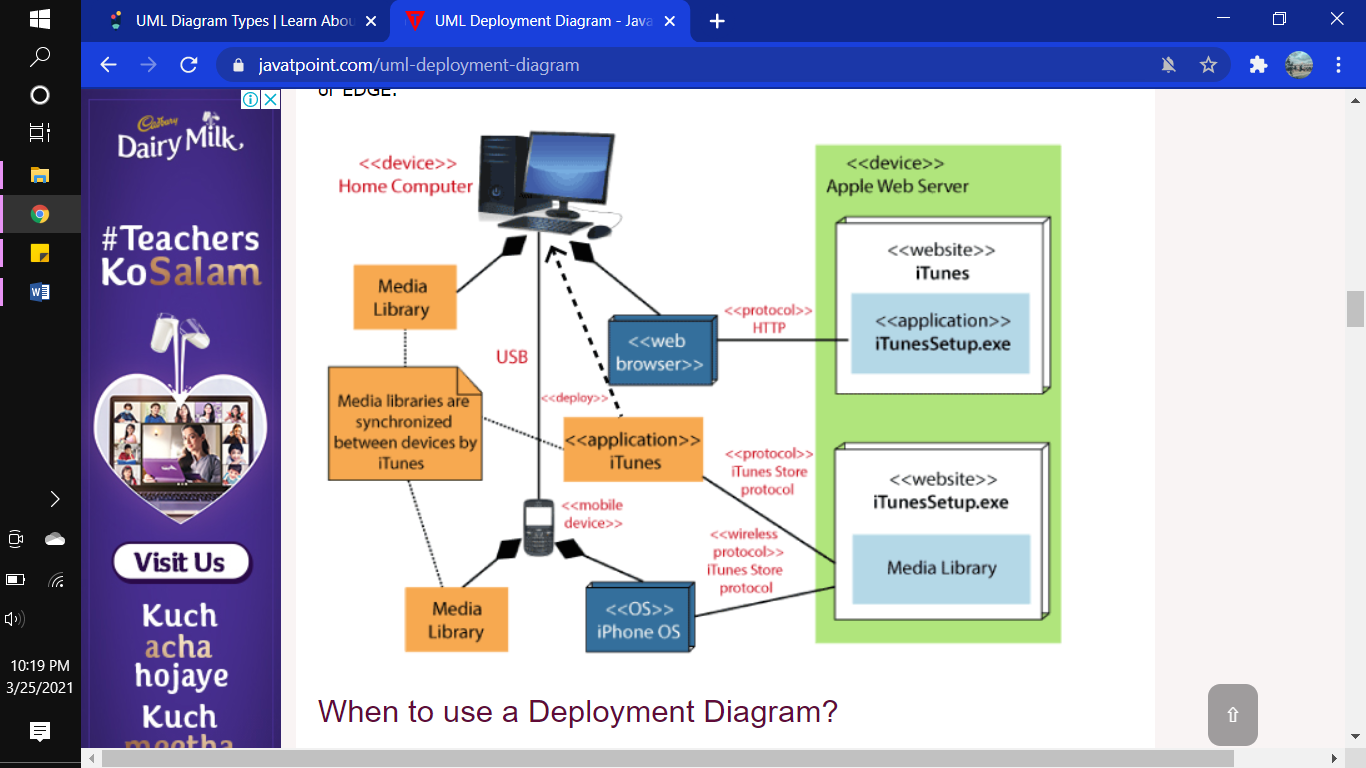
The deployment diagram portrays the deployment view of the system. It helps in visualizing the topological view of a system. It incorporates nodes, which are physical hardware. The nodes are used to execute the artifacts. The instances of artifacts can be deployed on the instances of nodes.

Since it plays a critical role during the administrative process, it involves the following parameters:

1. High performance
2. Scalability
3. Maintainability
4. Portability
5. Easily understandable

One of the essential elements of the deployment diagram is the nodes and artifacts. So it is necessary to identify all of the nodes and the relationship between them. It becomes easier to develop a deployment diagram if all of the nodes, artifacts, and their relationship is already known.

Consider the example of Apple itunes application.  iTunes application can easily interconnect with the Apple iTunes store. Users can purchase and download music, video, TV serials, etc. and cache it in the media library. Devices like Apple iPod Touch and Apple iPhone can update its own media library from the computer with iTunes with the help of USB or simply by downloading media directly from the Apple iTunes store using wireless protocols, for example; Wi-Fi, 3G, or EDGE.



## **Lab Report-08**

* Make deployment diagram of system of your choice

# **LAB 9 | 10: Software Architecture, Gang of Four(GoF), Design Patterns**

## **Purpose**

The purpose of this lab is to explain the architectural part of the software.

## **Theoretical Description**

Design patterns represent the best practices used by experienced object-oriented software developers. Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

**Gang of Four (GOF)** authors design patterns are primarily based on the following principles of object orientated design.

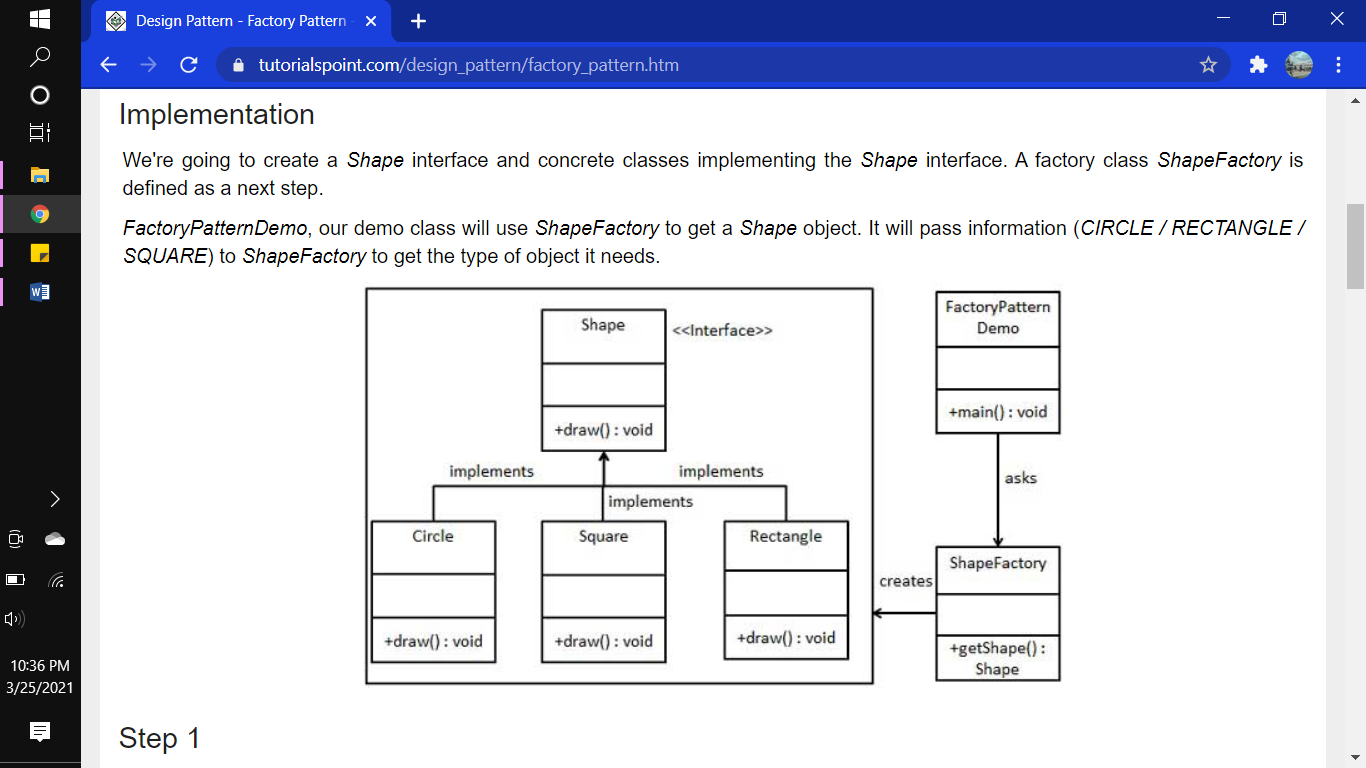
* Program to an interface not an implementation
* Favor object composition over inheritance

In the design patterns, there are 23 design patterns which can be classified in three categories: Creational, Structural and Behavioral patterns. Foremost used design patterns are explained below:

* **Factory**

Factory pattern is one of the most used design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.



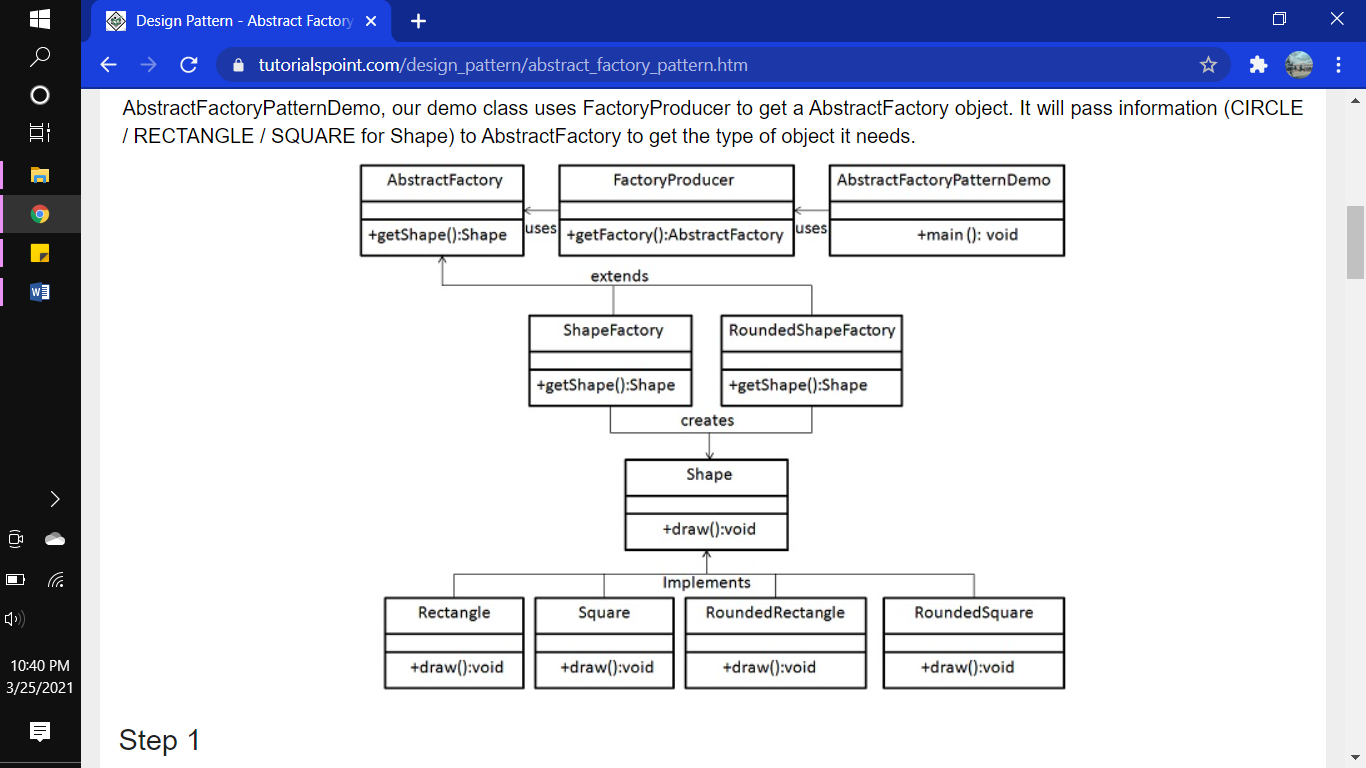
* **Abstract**

Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called as factory of factories. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.

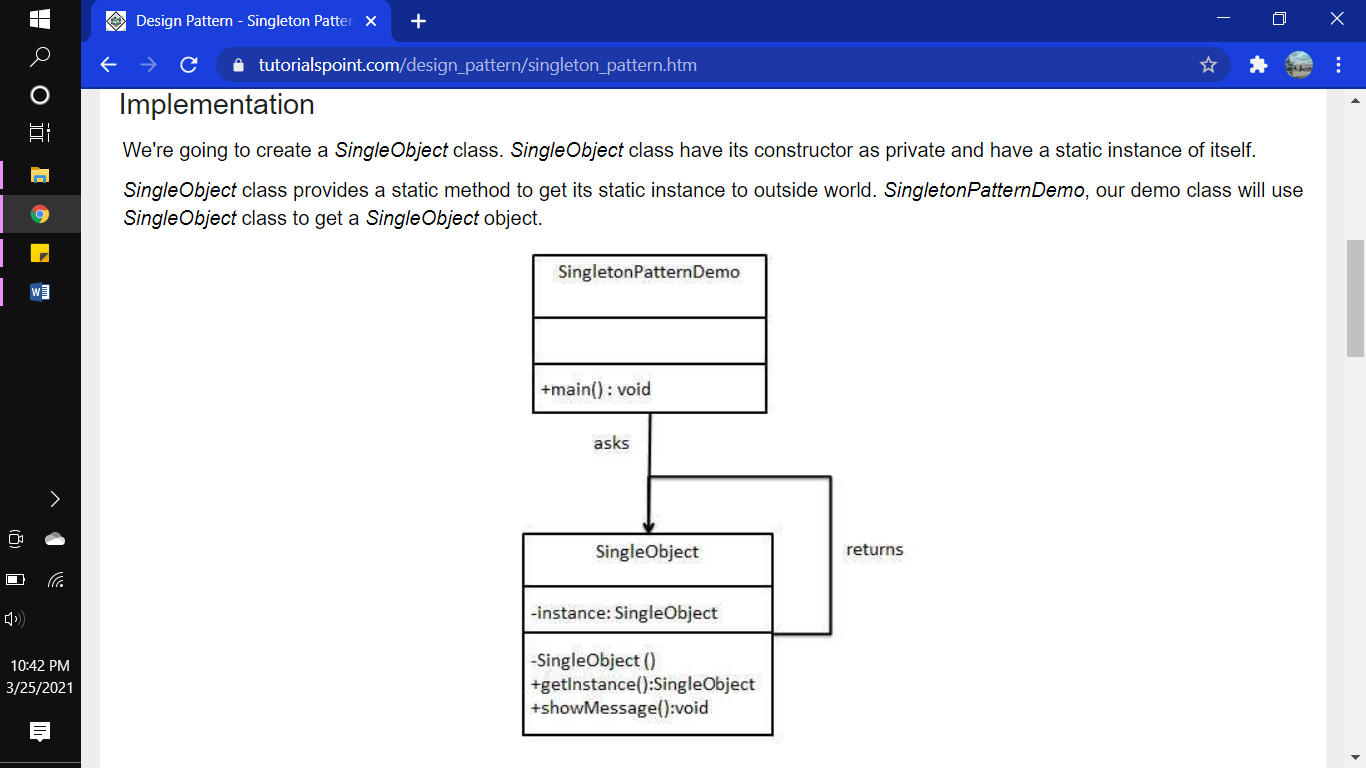
We are going to create a Shape interface and a concrete class implementing it. We create an abstract factory class AbstractFactory as next step. Factory class ShapeFactory is defined, which extends AbstractFactory. A factory creator/generator class FactoryProducer is created.

The figure below explained how to create an abstract design pattern for any system.



* **Singleton**

Singleton pattern is one of the simplest design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object. This pattern involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class. *SingleObject* class provides a static method to get its static instance to outside world. *SingletonPatternDemo*, our demo class will use *SingleObject* class to get a *SingleObject* object.



* **Adapter**

Adapter pattern works as a bridge between two incompatible interfaces. This type of design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces.

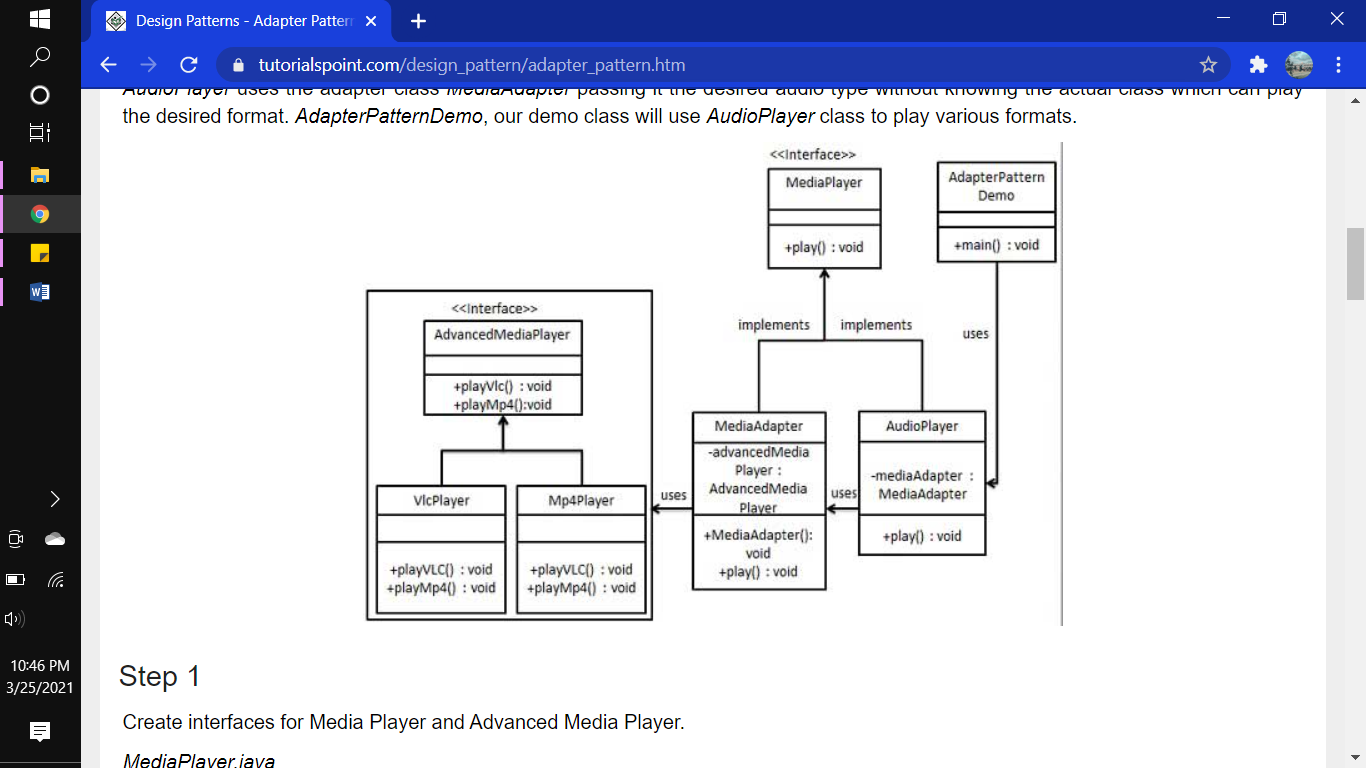
This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces. A real life example could be a case of card reader which acts as an adapter between memory card and a laptop. You plug in the memory card into card reader and card reader into the laptop so that memory card can be read via laptop.

We have a *MediaPlayer* interface and a concrete class *AudioPlayer* implementing the *MediaPlayer* interface. *AudioPlayer* can play mp3 format audio files by default.

We are having another interface *AdvancedMediaPlayer* and concrete classes implementing the *AdvancedMediaPlayer* interface. These classes can play vlc and mp4 format files.

We want to make *AudioPlayer* to play other formats as well. To attain this, we have created an adapter class *MediaAdapter* which implements the *MediaPlayer* interface and uses *AdvancedMediaPlayer* objects to play the required format.

*AudioPlayer* uses the adapter class *MediaAdapter* passing it the desired audio type without knowing the actual class which can play the desired format. *AdapterPatternDemo*, our demo class will use *AudioPlayer* class to play various formats.



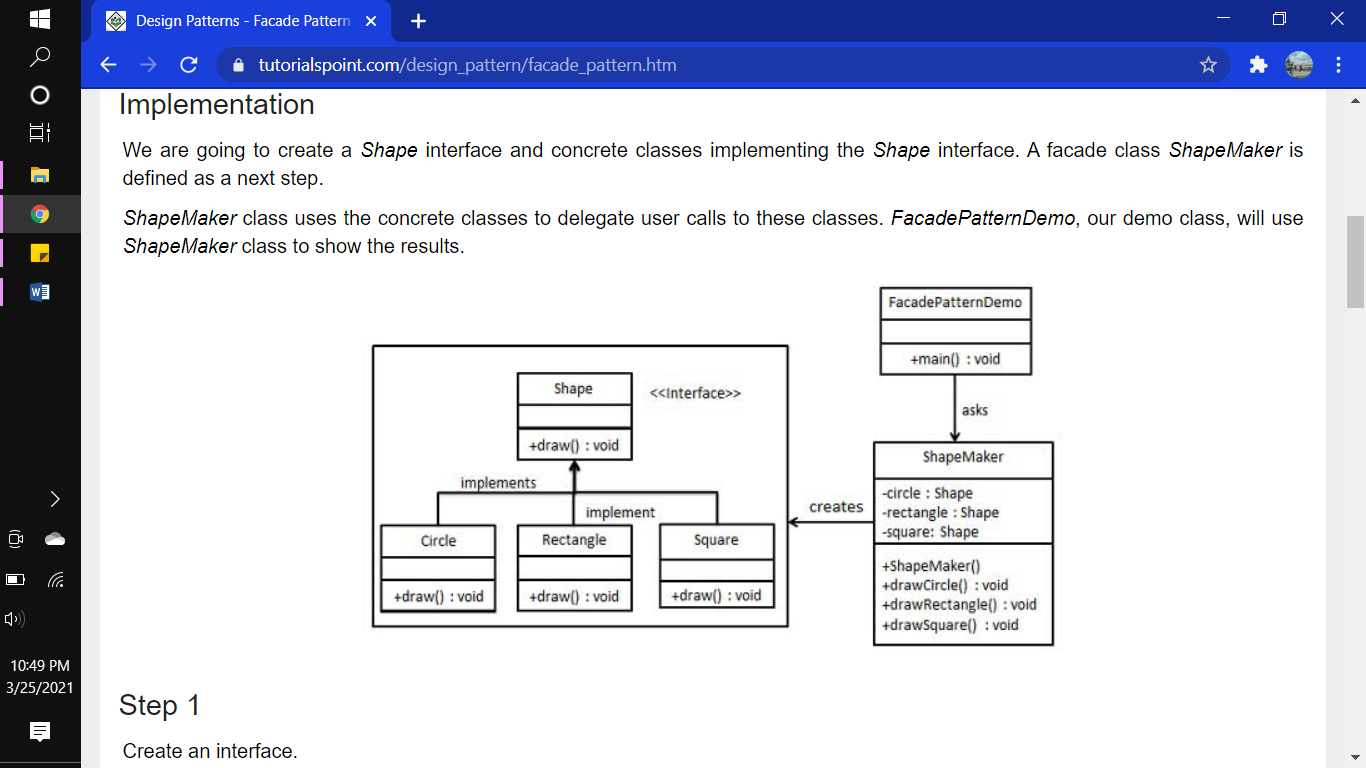
* **Façade**

Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. This type of design pattern comes under structural pattern as this pattern adds an interface to existing system to hide its complexities.

This pattern involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.

We are going to create a *Shape* interface and concrete classes implementing the *Shape* interface. A facade class *ShapeMaker* is defined as a next step.

*ShapeMaker* class uses the concrete classes to delegate user calls to these classes. *FacadePatternDemo*, our demo class, will use *ShapeMaker* class to show the results.



## **Lab Report-09|10**

# **LAB 10 | 11: Model View Controller (MVC)**

## **Purpose**

The purpose of this lab is to explain the concept of MVC in the systems.

## **Theoretical Description**

The Model-View-Controller (MVC) is an architectural pattern that separates an application into three main logical components: the model, the view, and the controller. Each of these components are built to handle specific development aspects of an application. MVC is one of the most frequently used industry-standard web development framework to create scalable and extensible projects.

Three major components of MVC are:

* **Model**

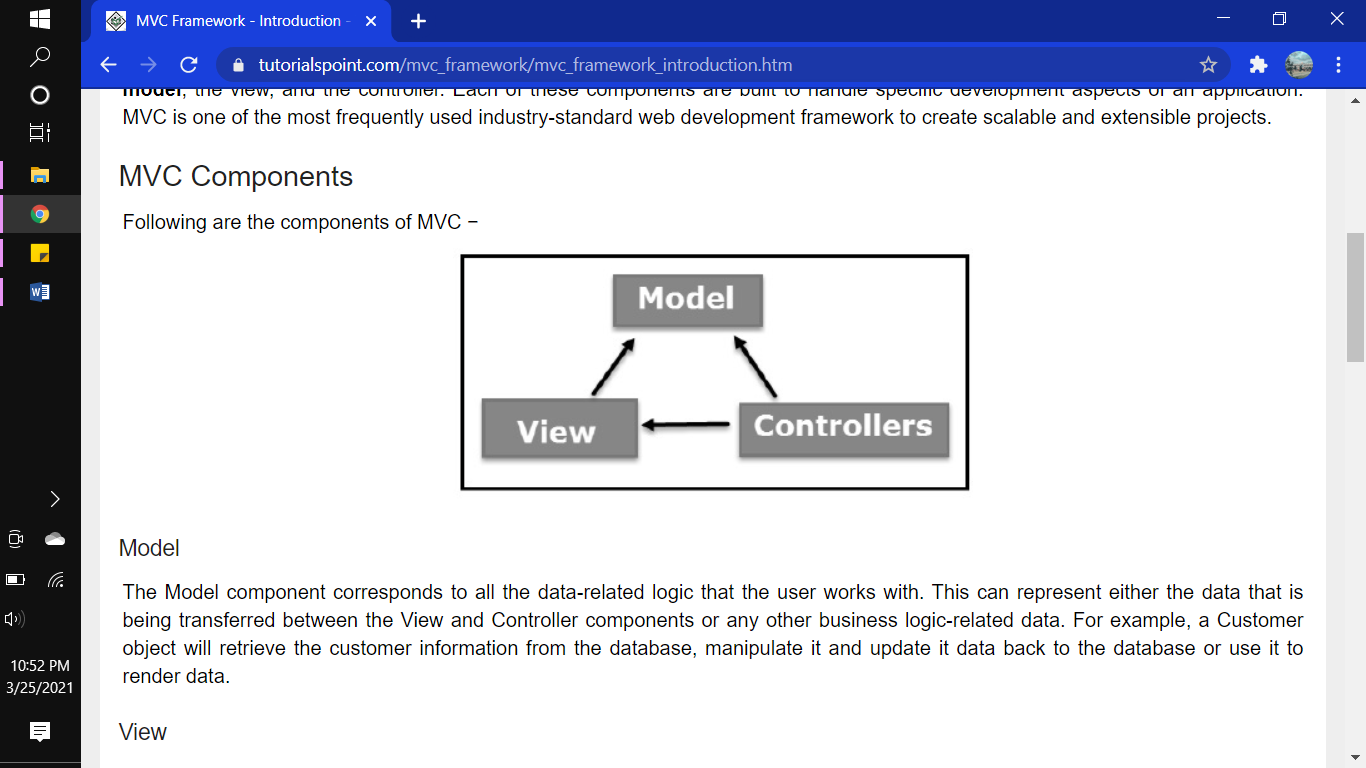
The Model component corresponds to all the data-related logic that the user works with. This can represent either the data that is being transferred between the View and Controller components or any other business logic-related data. For example, a Customer object will retrieve the customer information from the database, manipulate it and update it data back to the database or use it to render data.

* **View**

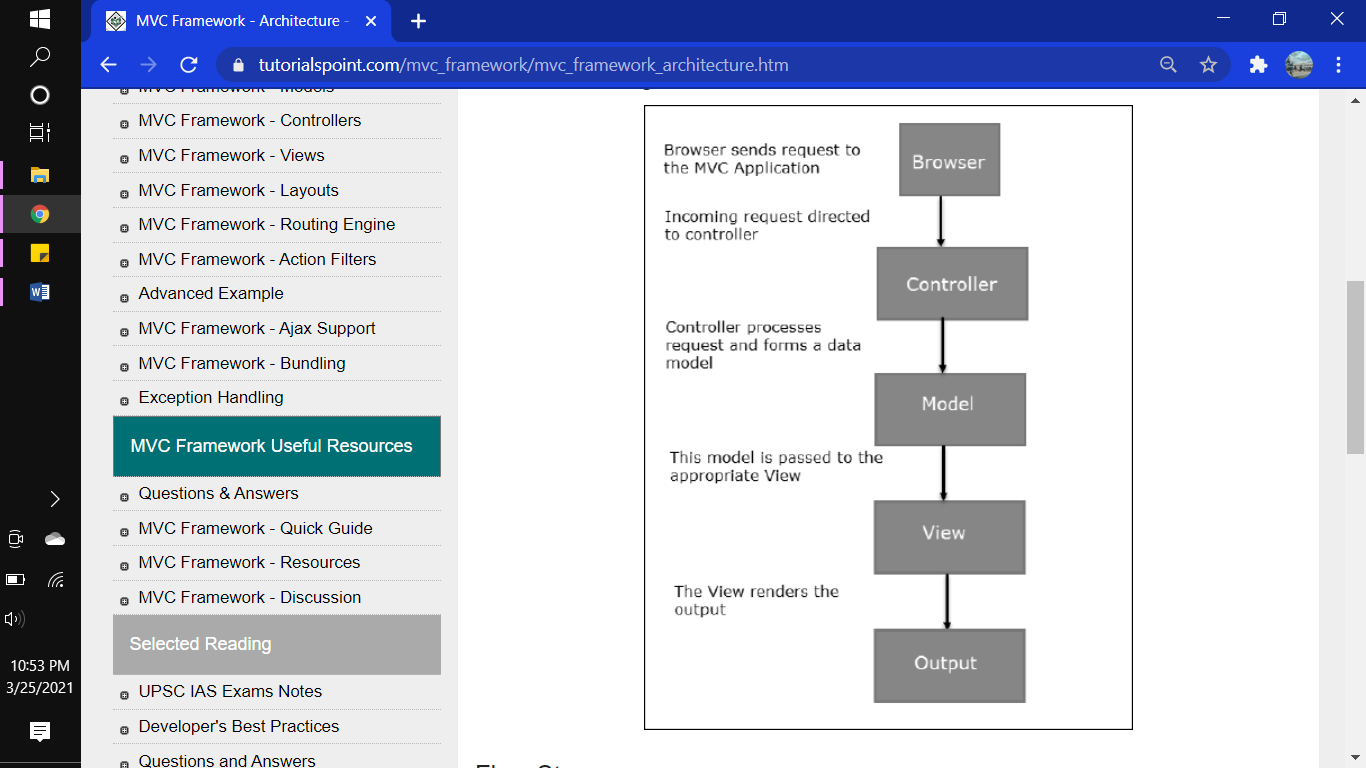
The View component is used for all the UI logic of the application. For example, the Customer view will include all the UI components such as text boxes, dropdowns, etc. that the final user interacts with.

* **Controller**

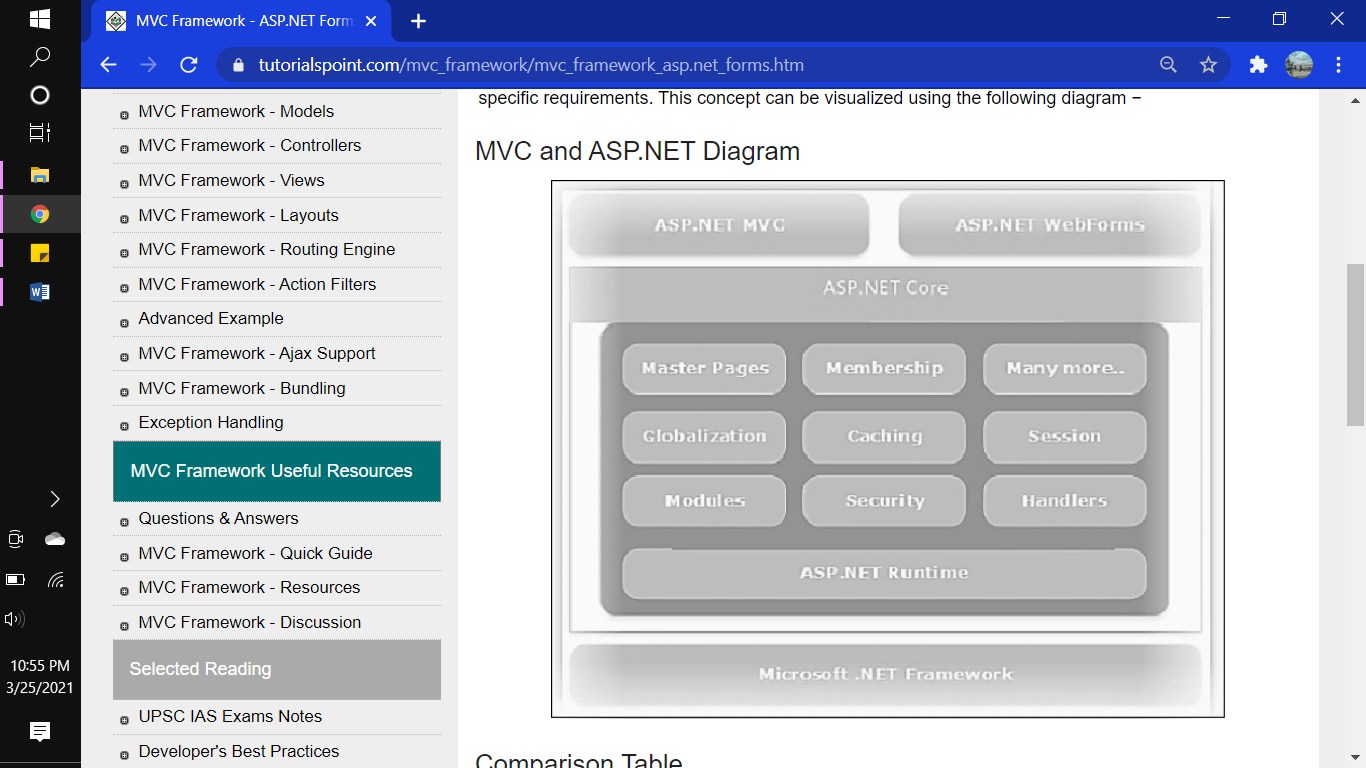
Controllers act as an interface between Model and View components to process all the business logic and incoming requests, manipulate data using the Model component and interact with the Views to render the final output. For example, the Customer controller will handle all the interactions and inputs from the Customer View and update the database using the Customer Model. The same controller will be used to view the Customer data.



Now let us take a look at how the execution of an MVC application takes place when there is a certain request from the client. The following diagram illustrates the flow.



MVC and ASP.NET Web Forms are inter-related yet different models of development, depending on the requirement of the application and other factors. At a high level, you can consider that MVC is an advanced and sophisticated web application framework designed with separation of concerns and testability in mind. Both the frameworks have their advantages and disadvantages depending on specific requirements. This concept can be visualized using the following diagram



Model-View-Controller (MVC) is a software architectural pattern for implementing user interfaces. It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user.

## **Lab Report-11|12**