

LAB MANUAL

SOFTWARE CONSTRUCTION & DEVELOPMENT



**DEPARTMENT OF SOFTWARE ENGINEERING
FACULTY OF ENGINEERING & COMPUTING
NATIONAL UNIVERSITY OF MODERN LANGUAGES
ISLAMABAD**

Preface

This lab manual has been prepared to facilitate the students of software engineering in understanding and implementing Software Construction fundamentals. The students will learn to understand and develop Object oriented design models, refine designs to reflect implementation details, demonstrate good and bad coding practices and analyze major software development activities. After completing this course, students will be able to practically implement the concepts of software construction and development in real-life applications.

Tools/ Technologies

- Visual Paradigm for UML
- Smart Draw
- Eclipse

TABLE OF CONTENTS

Preface.....	2
Tools/ Technologies.....	2
LAB 1: Introduction to Visual Paradigm	4
LAB 2: Structural and Behavioural Diagrams	6
LAB 3: Package Diagram.....	10
LAB 4: Object Diagram	12
LAB 5: Communication Diagram	14
LAB 6: Timing Diagram	16
LAB 7: Petri Nets.....	18
LAB 8: Clean and Bad Code (Naming Guidelines)	20
LAB 9: Good Code and Bad Code (Commenting Guidelines)	22
LAB 10: Refactoring.....	24
LAB 11: Extract Method.....	26
LAB 12: Exception Handling.....	27
LAB 13: Test Driven Development	29
LAB 14: Git & Git Hub	32
Open Ended Lab (OEL)	35

LAB 1: Introduction to Visual Paradigm

Objectives

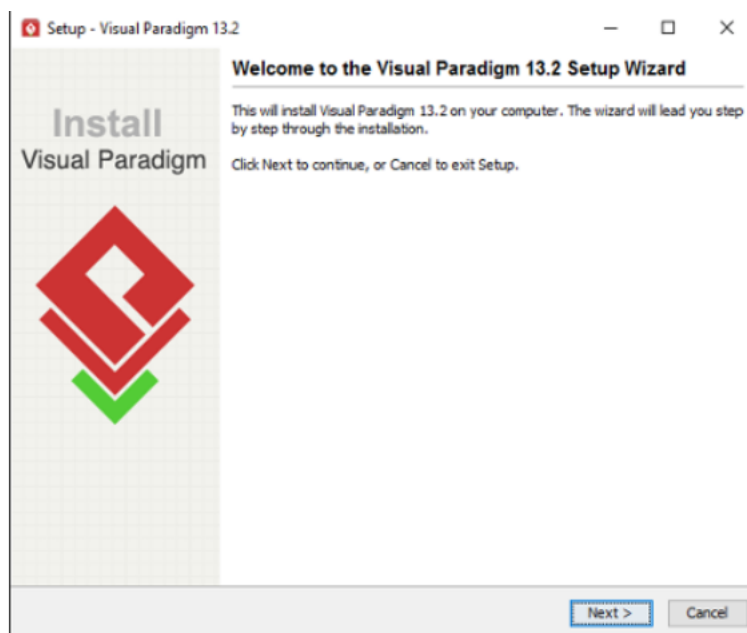
The purpose of this lab is to give a basic overview of Visual Paradigm. The manual discusses how to install and use Visual Paradigm.

Installation of MS Visio:

Software is available at:

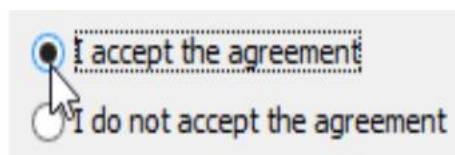
<https://www.visual-paradigm.com/download/>

Step 1: Run the setup.exe file. The setup wizard appears as below.



Step 2: Click **Next** to proceed to the **License Agreement** page.

Step 3: Read through the license agreement carefully. Make sure you accept the terms before continuing with the installation. If you accept the agreement, select **I accept the agreement** and click **Next** to proceed to the **Select Destination Directory** page.



The License Agreement

Step 4: Specify the directory for installing Visual Paradigm. Click **Next** to proceed to the next page.

Step 5: Specify the name of the Start Menu folder that will be used to store the shortcuts.

Step 6: Keep **Create shortcuts for all users** checked if you want the shortcut to be available in all the user accounts in the machine. Click **Next** to proceed.

In the **File Association** page, keep Visual Paradigm **Project (*.vpp)** checked if you want your system able to open the project file upon direct execution (i.e. double click). Click **Next** to start the file copying process.

Step 7: Upon finishing, you can select whether to start Visual Paradigm or not. Keep Visual Paradigm selected and click **Finish** will run Visual Paradigm right away.

LAB 2: Structural and Behavioural Diagrams

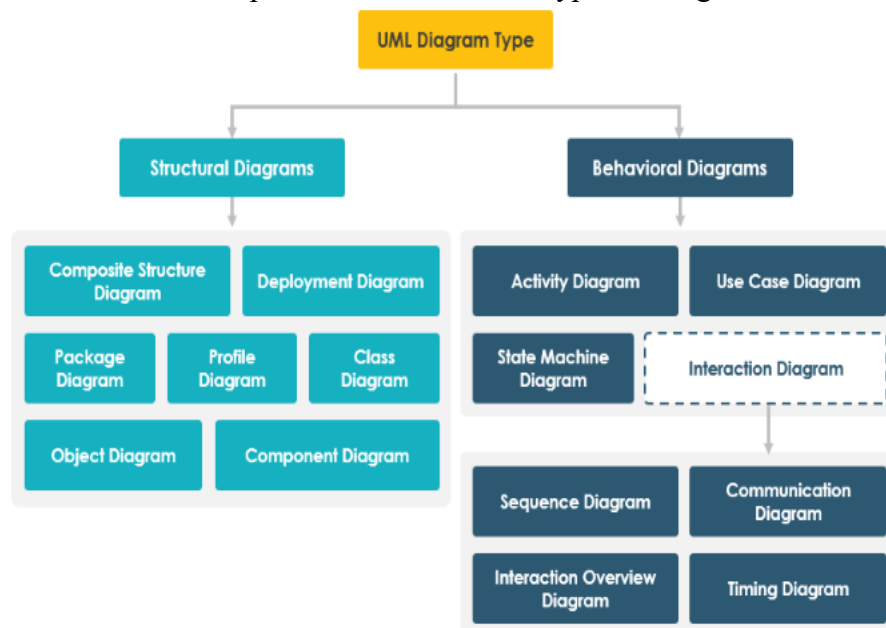
Objective

In today's lab, students will learn about structural and behavioral diagrams. First an introduction will be given; next students will practice an exercise related to use case diagrams.

Theoretical Description

- **Structural (or Static) view:** emphasizes the static structure of the system using objects, attributes, operations and relationships. It includes class diagrams and composite structure diagrams.
- **Behavioral (or Dynamic) view:** emphasizes the dynamic behavior of the system by showing collaborations among objects and changes to the internal states of objects. This view includes sequence diagrams, activity diagrams, and state machine diagrams.

Here's a brief description of each of the 14 types of diagrams and their categorization:



Distinguishing Between structure and behavior diagrams:

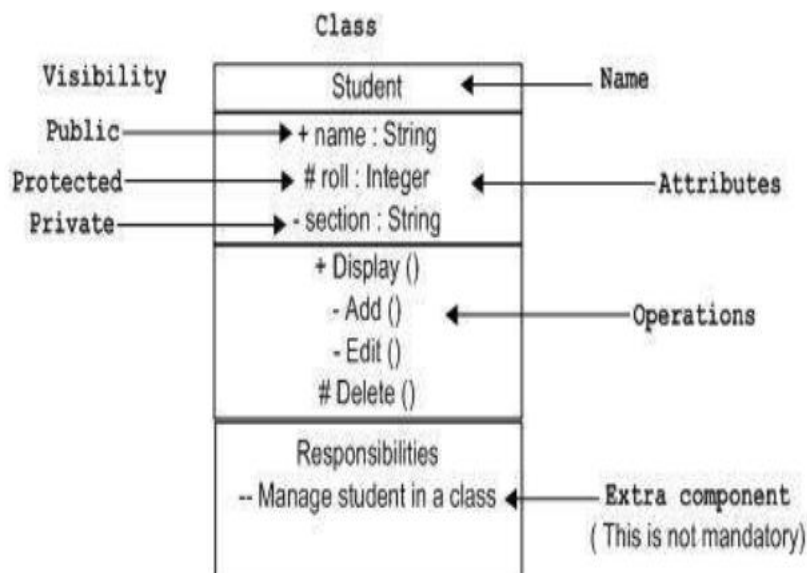
Structure diagrams provide a static view of a system, emphasizing its components, relationships, and organization, while behavior diagrams offer a dynamic view, focusing on the runtime behavior, interactions, and processes within the system. These two categories of diagrams serve distinct purposes and are essential for comprehensively modeling and documenting software systems, addressing both their static and dynamic aspects.

Here's a table that categorizes each of the 14 types of UML diagrams under the two categories, along with a brief example for each:

Structure Diagrams (Static Modeling):

Diagram Type	Description	Example
Class Diagram	Represents static class structure and relationships.	Example: Modeling a library system with classes like Book, Author, and Library.
Object Diagram	Shows instances and their relationships at a specific moment.	Example: Displaying specific Book and Member objects in a library system.
Package Diagram	Organizes elements into packages or namespaces.	Example: Grouping related classes into a LibraryManagement package.
Component Diagram	Depicts physical or logical system components and their connections.	Example: Illustrating software components like databases, web servers, and client applications in a web system.
Composite Structure Diagram	Details internal structure of a class with parts, ports, and connectors.	Example: Showing the internal structure of a computer system with components like CPU, RAM, and motherboard.
Deployment Diagram	Displays physical deployment of components on nodes or servers.	Example: Representing how web server software components are deployed on physical servers.

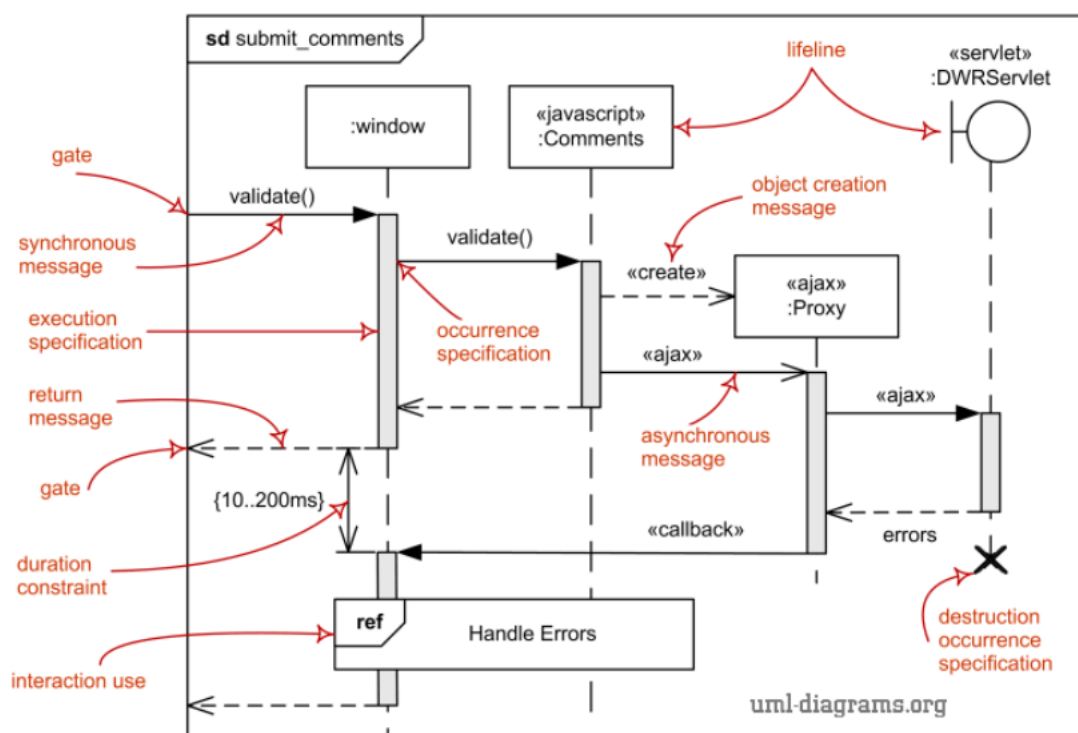
A sample structure diagram is attached for your reference.



Behavior Diagrams (Dynamic Modeling):

Diagram Type	Description	Example
Use Case Diagram	Defines actors and their interactions with the system through use cases.	Example: Modeling how a customer interacts with an ATM system to withdraw cash.
Activity Diagram	Depicts workflows, processes, and actions in a system, including branching and parallelism.	Example: Illustrating the steps involved in processing an online order.
State Machine Diagram	Represents the behavior of an object or system as a finite state machine with states and transitions.	Example: Modeling the states and transitions of a traffic signal system.
Sequence Diagram	Displays interactions between objects or components over time through messages.	Example: Showing the sequence of messages between a user and a database system during a login process.
Communication Diagram	Focuses on object interactions and their collaborations in a system.	Example: Visualizing how objects in a chat application exchange messages.
Interaction Overview Diagram	Combines elements of activity and sequence diagrams to provide an overview of complex interactions.	Example: Simplifying a complex order processing workflow in a retail system.
Timing Diagram	Specifies timing constraints of interactions, including lifelines and events.	Example: Showing the timing of data transmission between devices in a network.

A sample behavioral diagram is attached for reference



Lab Task:

1. Create class diagram to model the structural view of online bookstore system.
 - Identify the main classes
 - Define attributes and methods for each class.
 - Establish relationships (associations, inheritance, and dependencies) between classes.
 - Use appropriate visibility indicators (public, private, protected)
2. Create a sequence diagram to model the dynamic view by for the process of placing an order in the online bookstore system.
 - Identify the key objects involved in placing an order.
 - Model the interactions between these objects when a customer places an order.
 - Ensure to include method calls, returns, and object creation/destruction

LAB 3: Package Diagram

Objective

In today's lab, students will learn how to create Package diagram. The students will practice an exercise related to package diagram.

Theoretical Description

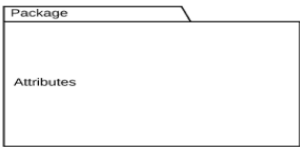
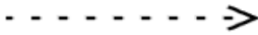
Package diagram, a kind of structural diagram, shows the arrangement and organization of model elements in middle to large scale project. Package diagram can show both structure and dependencies between sub-systems or modules, showing different views of a system, for example, as multi-layered application - multi-layered application model.

Package diagrams are used to structure high level system elements. Packages are used for organizing large system which contains diagrams, documents and other key deliverables.

Basic Concepts of Package Diagram

Package diagram follows hierarchal structure of nested packages. Atomic module for nested package is usually class diagrams.

Fully qualified name of a package has the following syntax.

Symbol Image	Symbol Name	Description
	Package	Groups common elements based on data, behavior, or user interaction
	Dependency	Depicts the relationship between one element (package, named element, etc) and another

Case Study:

The Travel Booking System is an integrated platform that enables users to book flights, hotels, and car rentals for their trips. It is structured into several main packages to efficiently manage its functionalities. The primary package, Travel Booking System, encompasses all aspects of the system. Within this, the Flights package includes all components related to flight bookings, the Hotels package covers hotel reservations, and the Car Rentals package deals with car hire services. User Profiles manages user accounts and profiles, while the Payment package handles all payment processing for bookings. Additionally, the Authentication package manages user authentication and authorization. There are specific dependencies within the system: Flights, Hotels, and Car Rentals all depend on the Payment package for processing transactions. User

Profiles relies on the Authentication package for user verification. Moreover, Flights, Hotels, and Car Rentals may need to interface with external services or APIs to retrieve availability and pricing information, ensuring that users receive up-to-date details for their travel plans.

Lab Task

Create a package diagram for the Travel Booking System that clearly identifies the main package, sub-packages, and their dependencies.

LAB 4: Object Diagram

Objective

In this lab, students will learn how to identify real-world objects, their attributes and how to model all this using Object Model.

Theoretical Description

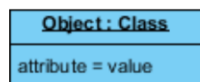
Object diagrams are derived from class diagrams so object diagrams are dependent upon class diagrams. Object diagrams represent an instance of a class diagram. The basic concepts are similar for class diagrams and object diagrams. Object diagrams also represent the static view of a system but this static view is a snapshot of the system at a particular moment.

Object Model Syntax

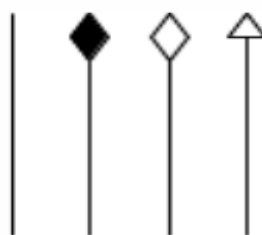
- **Object Structure:** Every object is actually symbolized like a rectangle, that offers the name from the object and its class underlined as well as divided with a colon.



- **Object Attributes Structure:** Similar to classes, you are able to list object attributes inside a separate compartment. However, unlike classes, object attributes should have values assigned for them.



- **Links Structure:** Links tend to be instances associated with associations. You can draw a link while using the lines utilized in class diagrams.



Case Study:

Consider social media network system, It is designed to facilitate interactions among users through posts, comments, and friendships. The primary classes within this system include User, Post, Comment, and Friendship.

Each User object possesses attributes such as name, username, and dateOfBirth, capturing essential personal information. Users interact with the platform by creating Post objects, which consist of content and a timestamp, documenting the activity and thoughts of the user at specific moments. Posts form the core of user-generated content within the network.

In addition to creating posts, users can engage with content by adding Comment objects to posts. Each comment is associated with a specific post, allowing users to express their opinions and participate in discussions.

The social aspect of the network is further enhanced by the Friendship objects. These objects represent the connections between different users, enabling them to build a network of friends. Friendships facilitate more personalized interactions and enable users to view and interact with their friends' posts and comments.

Lab Task

Develop a Object model for the given Social Media Network. Identify the objects, attributes, and associations explicitly. This lab task submission must include:

1. List of possible objects
2. Attributes values for each object
3. An object model showing the objects, its attributes and association between them.

LAB 5: Communication Diagram

Objective

The aim of this lab is to introduce students to the concept of UML communication diagrams. Scenario shall be given to the user for which they shall have to create a communication diagram.

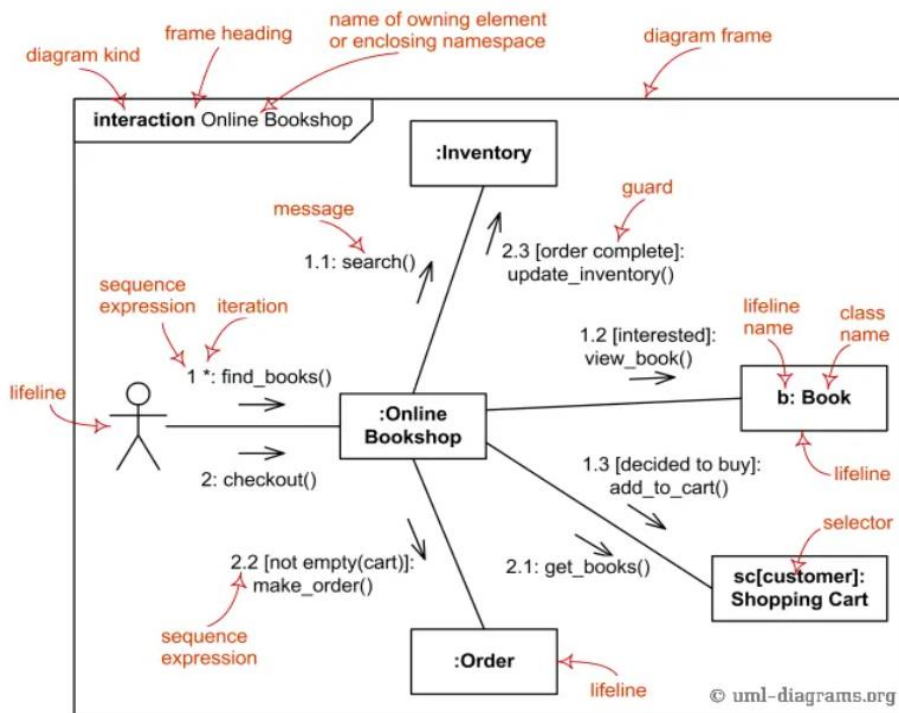
Theoretical Description

Communication diagrams, like the sequence diagrams is a kind of interaction diagram, shows how objects interact. A communication diagram is an extension of object diagram that shows the objects along with the messages that travel from one to another. In addition to the associations among objects, communication diagram shows the messages the objects send each other.

- Model message passing between objects or roles that deliver the functionalities of use cases and operations
- Model mechanisms within the architectural design of the system
- Capture interactions that show the passed messages between objects and roles within the collaboration scenario
- Model alternative scenarios within use cases or operations that involve the collaboration of different objects and interactions
- Support the identification of objects (hence classes), and their attributes (parameters of message) and operations (messages) that participate in use cases

Symbols:

Refer to the attached example for the symbols used in the communication diagram.



Case study:

Consider a coffee shop environment where various elements collaborate to serve customers efficiently. Customers place their coffee orders with the Baristas, specifying their preferences. Baristas take these orders and relay them to the Coffee Machines. The Coffee Machines brew coffee based on the orders received. Once ready, they notify the Baristas. Baristas prepare the coffee by adding milk, sugar, or other requested ingredients. They then serve the completed orders to the Customers. The Cash Register keeps track of the orders and handles payments. After receiving payment, it notifies the Order Queue to prepare the next order. The Order Queue organizes the orders and ensures they are prepared in the right sequence. It communicates with the Baristas and Coffee Machines to maintain a smooth workflow.

Lab Task

Create the communication diagram for the above-mentioned scenario.

LAB 6: Timing Diagram

Objective

To model specific behavioral modeling diagram that focuses on timing constraints.

Theoretical Description

Timing diagrams focus on conditions changing within and among lifelines along a linear time axis. Timing Diagrams describe behavior of both individual classifiers and interactions of classifiers, focusing attention on time of occurrence of events causing changes in the modeled conditions of the Lifeline

State Timeline Representation

Changes from one state to another are represented by a change in the level of the lifeline. For the period of time when the object is a given state, the timeline runs parallel to that state. A change in state appears as a vertical change from one level to another. The cause of the change, as is the case in a state or sequence diagram, is the receipt of a message, an event that causes a change, a condition within the system, or even just the passage of time.

Value lifeline Representation

The figure below shows an alternative notation of UML Timing diagram. It shows the state of the object between two horizontal lines that cross with each other each time the state changes.

Basic Concepts of Timing Diagrams

Major elements of timing UML diagram - lifeline, timeline, state or condition, message, duration constraint, timing ruler.

Lifeline

A lifeline in a Timing diagram forms a rectangular space within the content area of a frame. Lifeline is a named element which represents an individual participant in the interaction. It is typically aligned horizontally to read from left to right.

State Timeline in Timing Diagram

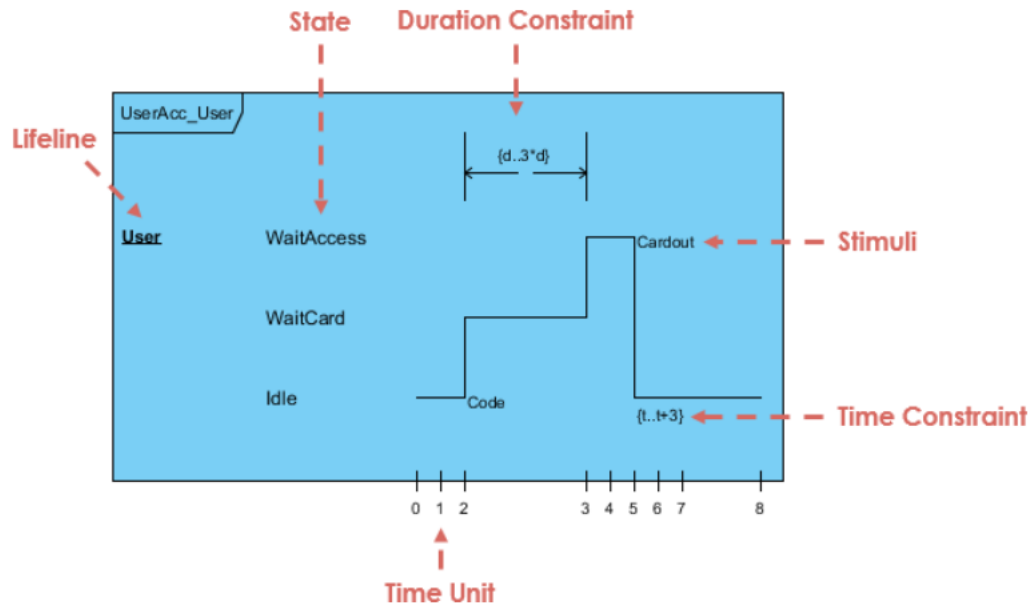
A state or condition timeline represents the set of valid states and time. The states are stacked on the left margin of the lifeline from top to bottom.

Multiple Compartments

It is possible to stack several life lines of different objects in the same timing diagram. One life line above the other. Messages sent from one object to another can be depicted using simple arrows. The start and the end points of each arrow indicate when each message was sent and when it was received.

State Lifeline

A state lifeline shows the change of state of an item over time. The X-axis displays elapsed time in whatever units are chosen while the Y-axis is labelled with a given list of states. Refer to the attached example for the symbols used in the Timing diagram.



Case Study: Multiplayer Game Match in a Gaming App

Consider a multiplayer online game where players join a match, and various game events occur in a time-ordered sequence. Details of timing diagram to illustrate the sequence of events during a match.

Participants:

- ✓ Players: Represented as Player1, Player2, Player3, etc.
- ✓ Game Server: Responsible for managing the game match and broadcasting events.

Events and Actions:

- ✓ Match Start: Players join the match, and the game server initiates the match.
- ✓ Gameplay Events: Players perform actions such as moving characters, shooting, and interacting with objects.
- ✓ Scoring: Players score points during the game based on their actions and achievements.
- ✓ Game Over: The match ends when certain conditions are met, and a winner is determined.
- ✓ Post-Match: Players view post-match statistics, scores, and can choose to play another match or exit.

Lab Task

Draw the Timing diagram for the above mention case study.

LAB 7: Petri Nets

Objective

The objective of the Petri nets lab is to understand and apply the modeling, analysis, and simulation of concurrent, distributed systems using Petri nets.

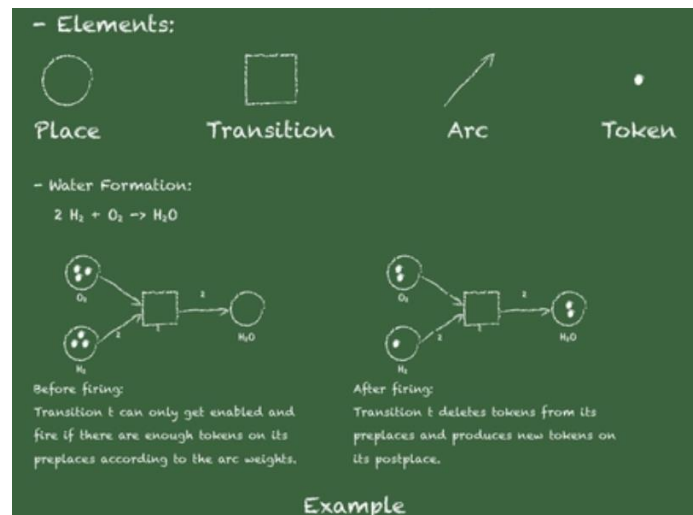
Theoretical Description

- Petri nets describe complex procedures and model the workings of a system.
- Petri nets use elements like places, transitions, arc & token to describe this complex procedure.
- A Petri net is also known as a place/transition net.
- Petri Net (PN) is a bipartite graph.

PN contains the following elements.

- Place (where you store something or It represents some sort of conditions that has to be satisfied).
- Transition (represents some event that occur/ or some processing activity that takes place).
- Tokens (represented by dots) are located at places in PNs. When a transition fires, it removes a token from each input place of the transition, and puts a token to each output place of the transition. Tokens are values that circulate through the graph.

Refer to the attached example for the symbols used in the Timing diagram.



Case study: Vending Machine for Beverages

Places:

- ✓ Idle: Represents the initial state of the vending machine.
- ✓ Beverage Selection: Represents the state where the customer selects a beverage.
- ✓ Payment: Represents the state where the customer makes a payment.
- ✓ Dispensing: Represents the state when the machine dispenses the selected beverage.
- ✓ Change: Represents the state where change is returned to the customer.
- ✓ Out of Stock: Represents the state when the selected beverage is unavailable.

Transitions:

- ✓ Customer Selection: Transition for the customer selecting a beverage.
- ✓ Payment Received: Transition for the customer making a payment.
- ✓ Dispense Beverage: Transition for the machine dispensing the beverage.
- ✓ Return Change: Transition for returning change.
- ✓ Out of Stock Notification: Transition to notify the customer when a beverage is out of stock.

Arcs (Connections):

Tokens (resources) move from one place to another through transitions based on the actions of the customer and the state of the vending machine.

Initial Marking:

Initially, the "Idle" place has a token, representing the vending machine being ready for use. The "Out of Stock" place may contain tokens for beverages that are unavailable.

Token Movement:

- ✓ Initially, the vending machine is in the "Idle" state, waiting for a customer to make a selection.
- ✓ When the customer selects a beverage, the "Customer Selection" transition is enabled, and a token moves to "Beverage Selection."
- ✓ After the customer makes a payment, the "Payment Received" transition is enabled, and a token moves to the "Payment" place.
- ✓ When the payment is received, the vending machine can dispense the beverage, and the "Dispense Beverage" transition is enabled, moving a token to "Dispensing."
- ✓ After dispensing, if there's change to return, the "Return Change" transition is enabled, and a token moves to the "Change" place.

If a beverage is out of stock, the "Out of Stock Notification" transition is enabled, and a token moves to "Out of Stock," notifying the customer and returning them to the "Beverage Selection" state

Lab Task

Draw Petri nets for the above-mentioned scenario.

LAB 8: Clean and Bad Code (Naming Guidelines)

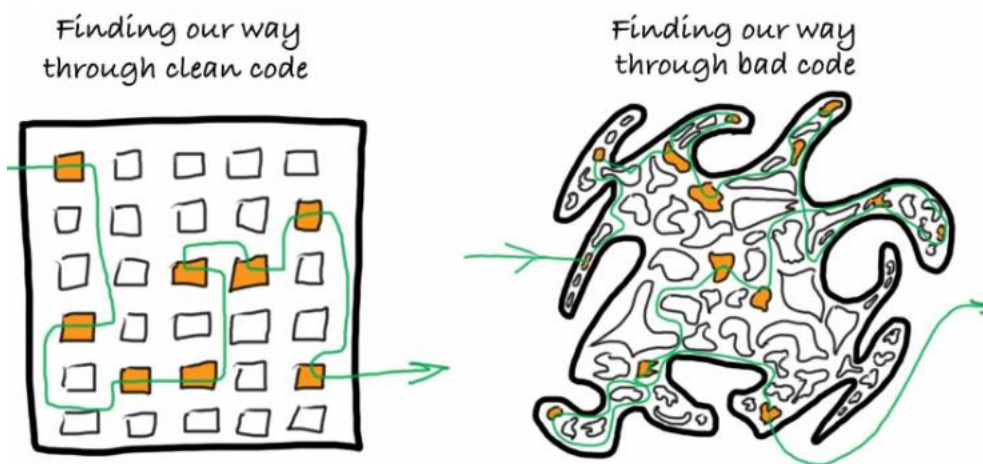
Objective

In this lab students will understand and apply clean code-naming guidelines to enhance code readability and maintainability.

Theoretical Description

Clean Code

Clean code is code that is easy to understand and easy to change. It is simple and direct. It reads like well-written prose. It never obscures the designer's intent but rather is full of crisp abstractions and straightforward lines of control.



Meaningful Names:

Names are everywhere in software. We name our variables, functions, classes, and packages. We name our source files and the directories that contain them. We name our jar files and war files. We name and name and name. Because we do so much of it, we'd better do it well. What follows are some simple rules for creating good names.

- Use Intention-Revealing Names
 - ✓ It is easy to say that names should reveal intent.
 - ✓ The name of a variable, function, or class, should answer all the big questions.
 - ✓ It should tell you why it exists, what it does, and how it is used. If a name requires a comment, then the name does not reveal its intent.
- Use pronounceable names
- Use Searchable Names
 - ✓ You need to search the code to see the different fields used.
 - ✓ For example searching and replacing a number value in an application can be a nightmare if the same value was used in line for different use cases.
 - ✓ Customer_name, employee_name, admin_name
- Don't Be Cute/Don't Use Offensive Words

- ✓ Don't use funny names or any sort of slang while naming your items.
- ✓ Don't use culture dependent jokes.
- ✓ Again , focus on the clarity. Say what you mean , Mean what you say.
- Pick One Word per Concept
 - ✓ Use the same concept across the codebase.
- Don't pun.
 - ✓ Don't use same word for two different purposes.

Lab Task

1. Implement a car rental system using Naming conventions and follow Hungarian Notation as standard. Include fields for car details, rental duration, and customer information and methods for editing and printing these details.
2. Implement Zakat Calculator using Naming conventions and follow snake case Notation as standard

LAB 9: Good Code and Bad Code (Commenting Guidelines)

Objective

To understand and apply clean code and bad code commenting guidelines to improve code clarity and maintainability.

Theoretical Description

The commenting guidelines are as follows:

Don't be redundant:

- Avoid repeating information that is already clear from the code itself. Redundant comments clutter the code and do not add value.
Example: `int age = 25; // age is 25` (The comment is redundant as it repeats what the code already states).

Don't add obvious noise:

- Avoid comments that state the obvious or add no new information. These comments are unnecessary and make the code harder to read.
Example: `i++; // increment i by 1` (It is clear from the code that `i` is being incremented).

Don't use closing brace comments:

- Avoid comments that indicate the end of a code block, such as `// end if`. These comments are often unnecessary if the code is well-structured and properly indented.
- Example:

```
if (condition) {  
    // some code  
} // end if
```

Don't comment out code. Just remove:

- Instead of commenting out old or unused code, remove it. If you need to reference old code, use version control systems.
- Example: `// int oldValue = computeOldValue();`

Use as explanation of intent:

- Use comments to explain the purpose or intent behind a piece of code, especially if it is not immediately obvious.
Example: `// Calculate the total price after applying discount`

Use as clarification of code:

- Use comments to clarify complex or non-obvious code logic, helping future readers understand what the code is doing.
Example: `// Convert date to UTC to ensure consistency across time zones`

Use as warning of consequences:

- Use comments to warn about potential side effects or important consequences of the code, which might not be immediately apparent.
Example: `// Warning: This operation will delete all user data`

A sample Warning comment is attached for your reference

```
// Should not take more than 10000 Logs due to performance issue  
public File createLogFile(final String filePath, final int logs) {  
    final File file = new File(filePath);  
  
    return adjustFile(file, FileType.LOGS, logs);  
}
```

Lab Task

1. Add to do comments and Legal Comments in Car Rental system.
2. Improve Zakat Calculator with clear comments explaining calculation logic and important considerations for better code understanding.
3. Give example of amplification comments.

LAB 10: Refactoring

Objective

In this lab, students will improve code quality and maintainability by applying refactoring.

Theoretical Description

Refactoring:

Refactoring is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour.

Three strikes and you refactor

1. You do something the first time
2. You do similar thing again with regret
3. On the third time – start refactoring

Code Smells:

Code smells are indicators of problems that can be addressed during refactoring. Code smells are easy to spot and fix, but they may be just symptoms of a deeper problem with code.

Here's an overview of common code smells and their types:

1. **Long Method:**
Methods that are excessively long and perform too many tasks.
Example: A method that spans over 100 lines, making it hard to understand and maintain.
2. **Long Class:**
Classes that are too large, containing numerous methods and data fields.
Example: A class with hundreds of lines of code and many member variables.
3. **Code Duplication:**
Identical or very similar code fragments repeated throughout the codebase.
Example: The same block of code appearing in multiple places without being encapsulated into a reusable function or class.
4. **Large Parameters:**
Methods that take a large number of parameters.
Example: A method with 10 or more parameters, which can lead to confusion and makes the method difficult to use correctly.
5. **Dispensable:**

Code that is unnecessary, redundant, or does not contribute to the functionality.

Example: Unused variables, commented-out code, or methods that are never called.

Addressing these code smells improves code maintainability, readability, and reduces the risk of introducing bugs during development.

Lab Task

Identify the smell in the given code snippet and remove using refactoring.

```
public class ShapeAreaCalculator {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Select a shape to calculate area:");
        System.out.println("1. Square");
        System.out.println("2. Rectangle");
        System.out.println("3. Triangle");
        int choice = scanner.nextInt();
        double area = 0.0;
        switch (choice) {
            case 1:
                System.out.print("Enter the side length of the square: ");
                double sideLength = scanner.nextDouble();
                area = sideLength * sideLength;
                break;
            case 2:
                System.out.print("Enter the length of the rectangle: ");
                double length = scanner.nextDouble();
                System.out.print("Enter the width of the rectangle: ");
                double width = scanner.nextDouble();
                area = length * width;
                break;
            case 3:
                System.out.print("Enter the base length of the triangle: ");
                double baseLength = scanner.nextDouble();
                System.out.print("Enter the height of the triangle: ");
                double height = scanner.nextDouble();
                area = 0.5 * baseLength * height;
                break;
            default:
                System.out.println("Invalid choice. Please select a valid option.");
        }
        System.out.println("Area: " + area);
        scanner.close();
    }
}
```

LAB 11: Extract Method

Objective

The objective of this lab is to apply extract method to improve code maintainability.

Theoretical Description

The Extract Method technique involves taking a sequence of statements from within a method and moving them into a new method. This helps to clarify the purpose of the code, reduce code duplication, and improve the ability to reuse and maintain the codebase effectively.

The more lines found in a method, the harder it's to figure out what the method does. This is the main reason for this refactoring. Besides eliminating rough edges in your code, extracting methods is also a step in many other refactoring approaches.

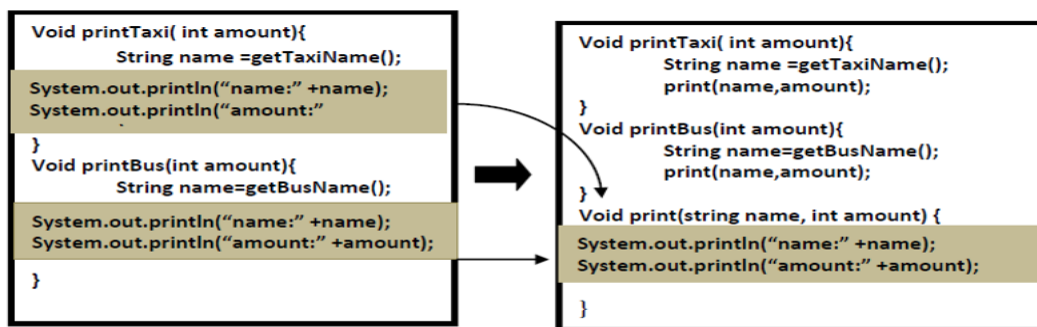


Figure 1. One Scenario for Extract Method Refactoring

Lab Task

Analyze the code and use extract method to improve its structure.

```

public class PayrollCalculator {
    public static void main(String[] args) {
        String employeeName1 = "John"; double hourlyWage1 = 15.0;
        int hoursWorked1 = 45;
        System.out.println("Employee: " + employeeName1);
        System.out.println("Monthly Salary: $" + hourlyWage *
hoursWorked;
        System.out.println();
        String employeeName2 = "Alice";
        double hourlyWage2 = 20.0;
        int hoursWorked2 = 35;
        System.out.println("Employee: " + employeeName2);
        System.out.println("Monthly Salary: $" + hourlyWage *
hoursWorked;
        System.out.println(); }
  
```

LAB 12: Exception Handling

Objective

The objective of this lab is to allow students to effectively identify and handle runtime errors.

Theoretical Description

The Exception Handling in Java is one of the powerful mechanisms to handle the runtime errors so that the normal flow of the application can be maintained. Exception Handling is a mechanism to handle runtime errors such as Class Not Found Exception, IO Exception, SQL Exception, Remote Exception, etc. The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions. Let's consider a scenario:

```
statement 1;  
statement 2;  
statement 3;  
statement 4;  
statement 5;//exception occurs  
statement 6;  
statement 7;  
statement 8;  
statement 9;  
statement 10;
```

There are different types of Exceptions:

Checked Exception

The classes that directly inherit the Throwable class except Runtime Exception and Error are known as checked exceptions. For example, IO Exception, SQL Exception, etc. Checked exceptions are checked at compile-time.

Unchecked Exception

The classes that inherit the Runtime Exception are known as unchecked exceptions. For example Arithmetic Exception,. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

Error

Error is irrecoverable. Some example of errors are Out Of Memory Error.

Java provides five keywords that are used to handle the exception. The following table describes each.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Lab Tasks

1. Write a program that handles division by zero using try-catch for Arithmetic Exception.
2. Create a program that handles accessing an invalid array index using try-catch for Array Index Out of Bounds Exception.
3. Develop a program that includes multiple catch blocks to handle different exceptions such as Null Pointer Exception and Arithmetic Exception.

LAB 13: Test Driven Development

Objective

The aim of this lab is to introduce students to the concept of test-driven development.

Theoretical Description

Test-Driven Development (TDD) is a software development process that emphasizes writing tests before writing the actual code. It follows a cycle of writing a test, running the test (which initially fails because the code isn't implemented yet), implementing the code to make the test pass, and finally refactoring the code while ensuring all tests continue to pass. This approach aims to improve code quality, maintainability, and reliability by ensuring that all code is tested and meets the specified requirements from the outset.



Test Driven Development (TDD) Examples

Calculator Function: When building a calculator function, a TDD approach would involve writing a test case for the “add” function and then writing the code for the process to pass that test. Once the “add” function is working correctly, additional test cases would be written for other functions such as “subtract”, “multiply” and “divide”.

User Authentication: When building a user authentication system, a TDD approach would involve writing a test case for the user login functionality and then writing the code for the login process to pass that test. Once the login functionality works correctly, additional test cases will be written for registration, password reset, and account verification.

E-commerce Website: When building an e-commerce website, a TDD approach would involve writing test cases for various features such as product listings, shopping cart functionality, and checkout process. Tests would be written to ensure the system works correctly at each process stage, from adding items to the cart to completing the purchase.

TDD Implementation:

Here in this Test-Driven Development example, we will define a class password. For this class, we will try to satisfy following conditions.

A condition for Password acceptance:

- The password should be between 5 to 10 characters.

First in this TDD example, we write the code that fulfills all the above requirements.

```
package Prac;

import org.testng.Assert;
import org.testng.annotations.Test;

public class TestPassword {
    @Test
    public void TestPasswordLength() {
        PasswordValidator pv = new PasswordValidator();
        Assert.assertEquals(true, pv.isValid("Abc123"));
    }
}
```

Annotations and their purposes:

- `@Test`: Needed for TestNG
- `TestPasswordLength()`: We can not run test because this class is not created yet
- `pv.isValid("Abc123")`: This is main validation test

Scenario 1: To run the test, we create class PasswordValidator ();

```
package Prac;

public class PasswordValidator {
    public boolean isValid(String Password)
    {
        if (Password.length() >= 5 && Password.length() <= 10)
        {
            return true;
        }
        else
            return false;
    }
}
```

Main condition checking length of password. If meets return true otherwise false.

Output:

```

<terminated> TestPassword [TestNG] C:\Program Files\Java\jre1.8.0_77\bin\javaw.exe (Jul 25, 2016, 2:10:22 PM)
[TestNG] Running:
  C:\Users\kanchan\AppData\Local\Temp\testng-eclipse--571370159\testng-customsuite.xml

PASSED: TestPasswordLength
=====
      Default test
      Tests run: 1, Failures: 0, Skips: 0
=====

=====
Default suite
Total tests run: 1, Failures: 0, Skips: 0
=====

[TestNG] Time taken by org.testng.reporters.EmailableReporter2@1b40d5f0: 202 ms
[TestNG] Time taken by org.testng.reporters.XMLReporter@28f67ac7: 63 ms
[TestNG] Time taken by org.testng.reporters.jq.Main@546a03af: 78 ms
[TestNG] Time taken by org.testng.reporters.JUnitReportReporter@5a01ccaa: 2 ms
[TestNG] Time taken by [FailedReporter passed=0 failed=0 skipped=0]: 1 ms
[TestNG] Time taken by org.testng.reporters.SuiteHTMLReporter@2b80d80f: 10 ms

```



Result of test as Passed

Lab Task**1. Setup Initial Test Structure**

Create a basic test structure for a simple function or class. Write a test that verifies a basic functionality (e.g., a function that adds two numbers).

2. Write a Failing Test

Write a test that initially fails because the code under test is not yet implemented. Write a test case that expects a certain behaviour from the code (e.g., expecting a function to return a specific result).

3. Implement Minimum Code to Pass Test

Implement the minimum code necessary to make the failing test pass. Write the code to fulfill the requirements of the test case, ensuring it passes the test without adding unnecessary complexity.

4. Refactor Code

Refactor the code to improve design, readability, and maintainability. Review the implemented code and refactor it while ensuring all tests continue to pass. Remove duplication, improve naming, and apply best practices.

LAB 14: Git & Git Hub

Objective

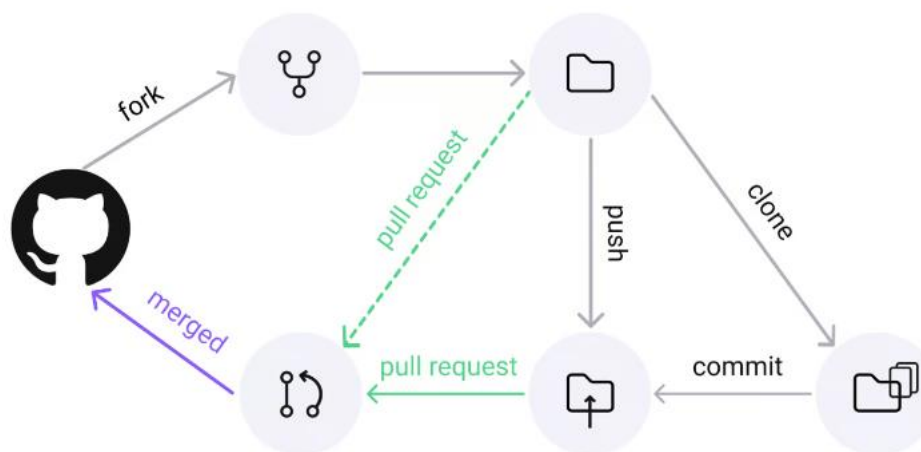
The objective of this lab is to familiarize students with the fundamentals of version control using Git and collaboration using GitHub.

Theoretical Description

Git is a distributed version control system that allows multiple developers to collaborate on projects efficiently. It tracks changes to files, enables versioning, and facilitates team coordination by managing branches, merging code, and providing mechanisms for code review and rollback. GitHub, on the other hand, is a platform built around Git that provides hosting for Git repositories, collaboration tools, issue tracking, and continuous integration capabilities, making it a powerful tool for software development teams to work together.

GitHub is a web-based platform built around Git, providing hosting for Git repositories and offering additional collaboration features. It enhances Git by adding social and project management functionalities, including issue tracking, pull requests for code review, team collaboration tools, and integration with continuous integration/deployment (CI/CD) pipelines. GitHub serves as a central hub for developers to share code, contribute to open-source projects, and manage software development projects effectively.

Note: For this you need a GitHub account and Internet access. You don't need to know how to code, use the command line, or install Git (the version control software that GitHub is built on).



Here are descriptions of some general terms used with Git:

1. **Repository (Repo):**

A repository is a storage location where your project's files and revision history are stored. It can be local (on your computer) or remote (on a server, like GitHub). Git repositories contain all the files and directories associated with your project, along with metadata about changes to those files.

2. **Commit:**

A commit is a snapshot of changes made to files in a repository at a specific point in time. Each commit has a unique identifier (hash) and includes the changes made, along with a commit message that describes the changes. Commits are used to track the history of changes and facilitate collaboration among developers.

3. **Branch:**

A branch is a separate line of development within a repository. It allows you to work on new features or bug fixes without affecting the main codebase (typically referred to as the main or master branch). Branches are lightweight and can be created, merged, or deleted as needed to support parallel development.

4. **Merge:**

Merging is the process of combining changes from one branch (source branch) into another branch (target branch). It integrates the changes made in the source branch with the target branch, creating a new commit that reflects the combined history of both branches. Merging is essential for integrating new features, resolving conflicts, and maintaining a cohesive codebase.

5. **Pull Request (PR):**

A pull request is a GitHub feature that allows developers to propose changes to a repository and request that those changes be reviewed and merged into the main branch. It includes a comparison of the changes made, discussions, and automated tests (if configured). Pull requests facilitate code review, collaboration, and quality control before merging changes into the main codebase.

6. **Clone:**

Cloning is the process of creating a copy of a remote repository onto your local machine. It allows you to work on the repository locally, make changes, and synchronize those changes with the remote repository. Cloning establishes a connection between your local environment and the remote repository, enabling collaboration and version control.

7. **Push:**

Pushing refers to sending committed changes from your local repository to a remote repository (like GitHub). It updates the remote repository with your local commits, making your changes accessible to others and enabling collaboration. Pushing is essential for sharing work and synchronizing changes across multiple developers.

8. **Pull:**

Pulling is the process of fetching and integrating changes from a remote repository into your local repository. It updates your local repository with changes made by others, allowing you to stay up-to-date with the latest codebase. Pulling is necessary before pushing your own changes to ensure your repository reflects the current state of the remote repository.

Lab Task

1. Create a new repository on GitHub. Log in to GitHub, navigate to your profile, click on "Repositories," and then click on "New." Enter a name for your repository, choose visibility options, and click "Create repository."
2. Initialize a Git repository locally, add files to it. Open your terminal or Git Bash, navigate to your project directory (`cd path/to/your/project`), initialize a Git repository (`git init`), add your files (`git add .` or `git add <file>` for specific files).

3. Commit the added files with a descriptive message. Commit your changes (`git commit -m "Initial commit"`). Replace "Initial commit" with a meaningful message describing the changes made in this commit.
4. Push your local repository to GitHub. Link your local repository to the remote GitHub repository (`git remote add origin <repository-url>`), push changes to GitHub (`git push -u origin main` or `git push -u origin master` depending on your branch name).

Open Ended Lab (OEL)

Course: Software Construction & Development

Total Marks: 15

OEL Title: Analysis and Design in Software Construction

OEL Level: 1

Objectives:

The objective of this lab is to enable students to develop advanced skills in analyzing and designing software systems by applying modern software construction principles. Students will explore key aspects of software behavior, synchronization, timing, and modeling in the context of real-world problems. The lab aims to enhance problem-solving abilities, develop critical thinking, and strengthen their understanding of software design and development processes.

Background Information:

In software construction, **static and dynamic behavioural models** play a crucial role in representing the structural and behavioural aspects of a system.

1. Static Behavioural Models:

Static models focus on the system's structure at a specific point in time, providing a snapshot of components, their relationships, and organization. These models are stable and do not depict system interactions or changes over time. Examples include:

- **Package Diagram:** Represents the organization of system components into logical groups to manage dependencies and modularity.
- **Object Diagram:** Illustrates the system's objects at a particular point in time, showing their relationships and attributes.

2. Dynamic Behavioural Models:

Dynamic models emphasize the system's behaviour over time, capturing interactions, processes, and state transitions. These models show how components respond to events or user actions and are crucial for understanding real-time functionality. Examples include:

- **Communication Diagram:** Depicts interactions between objects or components, focusing on message exchanges and collaboration.
- **Timing Diagram:** Represents the temporal behaviour of objects or components, showing changes in state or events over time.
- **Petri Net:** Models concurrent processes and resource allocation using tokens to represent state transitions and events.

By combining static and dynamic models, developers gain a comprehensive understanding of both the system's structural organization and its behavioural dynamics, enabling effective analysis and design.

Problem:

1. Model the Petri net for the given scenario. [10 Marks]

Consider the coffee machine basic functionality. Initially, the coffee machine is in the Idle position. You can add water or coffee by triggering the Add Water or Add Coffee buttons, which move tokens from Idle to the respective state. When both water and coffee are available, you can press the Start Brewing button, which transitions to the Brewing. The Brewing simulates the brewing process. When the brewing process is complete, the Brewing Complete action moves the system back to the Idle. For the given scenario, also draw the input/output matrix and transition firing table.

2. Model the communication diagram for the following scenario. [5 Marks]

Ahmad wants to select a workout plan through Fitness App. He accesses his daily nutritional intake data through the App. The App communicates with the Nutrition Tracker to fetch and display nutrition information. Ahmad logs his meals and nutritional intake through the App Interface. The Nutrition Tracker receives and records this data. Ahmad shares his workout and nutrition achievements with friends through the Social Sharing feature. The Social Sharing component communicates with the Database to store shared posts.

Requirements:

Solution Criteria for Petri-Nets:

- **State Representation:**
The solution must clearly define all system states (such as Idle, Add Water, Add Coffee, Brewing, Brewing Complete) in a way that reflects the real-world process.
- **Transition Logic:**
Transitions between states must be clearly defined based on user actions (e.g., pressing buttons) and system conditions. The transitions should ensure the system behaves as expected under all scenarios.
- **Token Movement:**
Tokens must move logically between states based on the actions taken, reflecting the system's status and current activity.
- **Input/Output Matrix:**
The solution must include a matrix that clearly defines the system's inputs and corresponding state transitions, showing the output at each stage.
- **Firing Table:**
A firing table must accurately represent the system's reaction to each action and transition, providing a detailed mapping of the system's behavior at each step.

Solution Criteria for Communication Diagram

- **Clear Object Representation:**
The solution must define distinct objects with clearly assigned responsibilities corresponding to real-world components.

- **Message Flow Representation:**
The communication between objects must be shown clearly with a logical flow of messages, accurately depicting how information is passed between objects .
- **Interaction Clarity:**
Interactions between objects must be depicted in a structured manner, with each message appropriately labelled to show the correct sequence of actions.
- **Logical Sequence:**
The solution must ensure that all interactions between components follow a logical sequence that represents the flow of operations in the system.
- **Component Interaction:**
The diagram must clearly show how components interact and depend on each other to achieve the user's goal