INDEX

Problem No: 01

Problem Name: Write a program in "JAVA" or "C" to develop a simple calculator that would be able to take a number, an operator (addition/subtraction/multiplication/ division/modulo) and another number consecutively as input and the program will display the output after pressing "=" sign. Sample input: 1+2; 8%4; Sample output: 1+2=3; 8%4=0.

Objective:

To Develop a simple calculator program in C or Java that accepts a single expression containing two numbers and an operator, then evaluates the expression and displays the result.

Theory: A calculator is a tool, either physical or software-based, that performs arithmetic operations on numbers. In programming, a simple calculator can be created by accepting two operands and an operator as input, then executing the appropriate arithmetic operation based on the operator.

The basic arithmetic operations are:
• **Addition (+):** Computes the sum of two numbers.
• **Subtraction (-):** Computes the difference by subtracting the second number from the first.
• **Multiplication (*):** Calculates the product of two numbers.
• **Division (/):** Divides the first number by the second (must handle division by zero).
• **Modulo (%):** Returns the remainder when the first number is divided by the second.

In programming languages such as Java and C, these functions are implemented using built-in operators. The program must:

1. Accept a user input expression like 1+2.
2. Parse this input to extract the operator and both numbers.
3. Execute the relevant arithmetic operation.
4. Show the result in a format like 1+2=3.

This experiment enables understanding of:
• How to accept and analyze input from the user.
• How to apply conditional logic (if-else or switch-case) to determine the operation.
• How to manage exceptions such as division by zero.
It also reinforces core programming concepts and arithmetic processing.

Algorithm:

1. **Start**
2. Read the input expression (e.g., 1+2) as a string.
3. Parse the first number, operator, and second number from the input.
4. Use a conditional structure (if or switch) to:
    - Perform addition if operator is +
    - Perform subtraction if operator is -
    - Perform multiplication if operator is *
    - Perform division if operator is /
    - Perform modulo if operator is %
5. Calculate the result based on the operator.
6. Display the full expression with result (e.g., 1+2=3).
7. **End**

Source Code:

```
#include <stdio.h>

int main() {
    int num1, num2, result;
    char operator, equalSign;

    printf("Enter an expression (e.g., 1+2=): ");
    scanf("%d%c%d%c", &num1, &operator, &num2, &equalSign);

    if (equalSign != '=') {
        printf("Error: Expression must end with '='\n");
        return 1;
    }

    switch (operator) {
        case '+':
            result = num1 + num2;
            printf("%d%c%d=%d\n", num1, operator, num2, result);
            break;
        case '-':
            result = num1 - num2;
            printf("%d%c%d=%d\n", num1, operator, num2, result);
            break;
        case '*':
            result = num1 * num2;
            printf("%d%c%d=%d\n", num1, operator, num2, result);
            break;
        case '/':
            if (num2 == 0) {
                printf("Error: Division by zero is not allowed.\n");
            } else {
                result = num1 / num2;
                printf("%d%c%d=%d\n", num1, operator, num2, result);
            }
            break;
        case '%':
            if (num2 == 0) {
                printf("Error: Modulo by zero is not allowed.\n");
            } else {
                result = num1 % num2;
                printf("%d%c%d=%d\n", num1, operator, num2, result);
            }
            break;
        default:
            printf("Error: Invalid operator.\n");
    }

    return 0;
}
```

Input: Sample input: 1+2=; 8%4=

Output:

Sample output: 1+2=3; 8%4=0.

Problem No: 02

Problem Name: Write a program in "JAVA" or "C" that will take two 'n' integers as input until a particular operator and produce 'n' output. Sample input: 4 5 7 8 20 40 +; Sample output: 9 15 60.

Objective:

To develop a C program that accepts two sequences of equal length integers followed by an arithmetic operator, performs the specified operation element-wise on the paired integers, and outputs the resulting sequence. This exercise aims to enhance skills in array handling, user input parsing, conditional statements, and basic arithmetic operations in C programming.

Theory:

This experiment focuses on array handling and performing element-wise operations, which are frequently used in scientific and statistical computing.

Key Concepts Covered:

1. **Arrays:**
   Arrays store multiple values of the same data type. In this program, two integer arrays are utilized to hold `n` values each.
2. **Arithmetic Operators:**
   These operators include:
   • Addition (+)
   • Subtraction (-)
   • Multiplication (*)
   • Division (/)
   • Modulo (%)
3. **Looping Constructs:**
   Loops such as `for` or `while` are vital for iterating through arrays and applying operations to each element.
4. **Input Parsing:**
   The program processes a single line of input, separating integers and the operator using string and input handling techniques.
5. **Element-Wise Operations:**
   Each element in the first array is combined with the corresponding element in the second array using the specified operator.

Algorithm:

1. **Start**
2. Initialize two arrays to store the two sequences of n integers.
3. Read input values until an operator (+, -, *, /, %) is encountered.
4. Count the number of integers and divide them into two equal parts.
5. Store the first half in array A and the second half in array B.
6. Identify the operator at the end of input.
7. For each index i from 0 to n-1:
   - Perform the operation A[i] op B[i]:
     - + → addition
     - - → subtraction
     - * → multiplication
     - / → division (check for divide-by-zero)
     - % → modulo
8. Print the resulting n outputs.
9. **End**

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    int numbers[100];
    int count = 0;
    char input[1000], *token;
    char operator;

    printf("Enter integers followed by an operator (e.g., 4 5 7 8 20 40 +):\n");
    fgets(input, sizeof(input), stdin);
    token = strtok(input, " \n");
    while (token != NULL) {
        if (strchr("+-*/%", token[0]) && strlen(token) == 1) {
            operator = token[0];
            break;
        }
        numbers[count++] = atoi(token);
        token = strtok(NULL, " \n");
    }
    if (count % 2 != 0) {
        printf("Invalid input: odd number of integers.\n");
        return 1;
    }

    for (int i = 0; i < count; i += 2) {
        int a = numbers[i];
        int b = numbers[i + 1];
        int result;
        switch (operator) {
            case '+': result = a + b; break;
            case '-': result = a - b; break;
            case '*': result = a * b; break;
            case '/': result = (b != 0) ? a / b : 0; break;
            case '%': result = (b != 0) ? a % b : 0; break;
            default:
                printf("Invalid operator.\n");
                return 1;
        }
        printf("%d ", result);
    }
    printf("\n");
    return 0;
}
```

Input: Enter integers followed by an operator:

4 5 7 8 20 40 +

Output:

9 15 60.

Problem No: 03

Problem Name: Write a program in “JAVA” or “C” to check whether a number or string is palindrome or not.
N.B: Your program must not take any test case number such as 1 or 2 for the desired cases from the user. Program user will insert a number or string as input directly and the program will display the exact result in the output console.

Objective:

To develop a C program that takes a single input (either a number or a string) and checks whether it is a palindrome, without requiring the user to specify the input type or number of test cases. This enhances understanding of string manipulation, conditional logic, and palindrome detection in C.

Theory: A **palindrome** is a number, word, or sequence that reads the same forward and backward. Examples include words like "madam" or numbers like 121. This concept is frequently used in string and number manipulation tasks in programming.

For Strings:

- A string is a palindrome if the characters at the beginning and end match and continue to match as we move inward.
- For example, "level" is a palindrome because the first and last letters are the same, and the second and second-last are the same.
- **For Numbers:**
- A number is treated as a sequence of digits. To check if a number is a palindrome, it can either be converted to a string or reversed mathematically and compared to the original number.

Algorithm:

1. **Start**
2. Take a string input from the user (treat both numbers and strings as character arrays).
3. Determine the length of the input.
4. Initialize two pointers: one at the start, one at the end.
5. While start < end:
    - Compare characters at both pointers.
    - If any pair doesn't match, it's **not a palindrome** → stop.
    - Move the pointers towards the center.
6. If all characters match, it's a **palindrome**.
7. Display the result.
8. **End**

Source Code:

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int isPalindrome(char str[]) {
    int left = 0;
    int right = strlen(str) - 1;

    while (left < right) {
        if (str[left] != str[right])
            return 0;
        left++;
        right--;
    }
    return 1;
}

int main() {
    char input[100];

    printf("Enter a number or string: ");
    scanf("%s", input);

    // Optional: Convert to lowercase if case-insensitive check is needed
    for (int i = 0; input[i]; i++) {
        input[i] = tolower(input[i]);
    }

    if (isPalindrome(input)) {
        printf("'%s' is a palindrome.\n", input);
    } else {
        printf("'%s' is not a palindrome.\n", input);
    }

    return 0;
}
```

Input: Enter a number or string: Pranto Mahmud

Enter a number or string: 4004

Output: Pranto Mahmud is not a palindrome.

4004 is a palindrome.

Problem No: 04

Problem Name: Write down the ATM system specifications and report the various bugs.

Objective:

To outline the functional and non-functional requirements of an Automated Teller Machine (ATM) system and highlight possible bugs or operational issues, with the goal of enhancing its reliability, security, and overall user experience.

Theory:

1. System Overview

An Automated Teller Machine (ATM) provides customers with banking services such as cash withdrawals, deposits, balance inquiries, and fund transfers without the need for human interaction.

2. Core Functional Requirements

- **Authentication**
    - Card insertion
    - PIN verification (usually 4 or 6 digits)
- **Transaction Services**
    - Cash Withdrawal
    - Balance Inquiry
    - Mini Statement
    - Fund Transfer (within bank or inter-bank)
    - Deposit (cash or check)
- **User Interface**
    - Multilingual support
    - Touchscreen or button-based navigation
    - Receipt printing option
- **Security**
    - Encrypted PIN and transaction processing
    - Session timeout
    - Camera surveillance (hardware)
- **Admin Services**
    - Cash loading and denomination management
    - System diagnostics
    - Logs of usage and errors

3. Non-Functional Requirements

- **Availability**: 24/7 operation
- **Performance**: Transactions processed within a few seconds
- **Scalability**: Support for integration with multiple banks and accounts
- **Reliability**: Must handle power failures gracefully (e.g., UPS backup)
- **Maintainability**: Logs, alerts, and remote diagnostics

4. Hardware Interfaces

- Card Reader (EMV chip/magnetic strip)
- PIN Pad / Touchscreen
- Cash Dispenser

- Receipt Printer
- Deposit Module
- Network Modem or Router

5. Software Interfaces

- Bank Core System API
- Encryption Libraries
- Operating System (typically Linux or a secure embedded OS)
- Database for local logging
-

Common ATM System Bugs

Bugs can significantly impact ATM reliability and security. Here are frequent issues:

1. Functional Bugs:

- **Incorrect Balance Display:** Balance not updating immediately after transactions.
- **Cash Dispensing Errors:** Account debited, but no cash dispensed, or incorrect denominations.
- **Deposit Discrepancies:** Mismatch between deposited amount and credited amount.
- **PIN Lockout Failure:** Incorrect PIN attempts not leading to card lockout.
- **Receipt Inaccuracies:** Wrong transaction details or dates on receipts.

2. Security Bugs:

- **Skimming Vulnerabilities:** Physical design flaws allowing card data theft.
- **Weak Data Encryption:** Vulnerable communication leading to sensitive data exposure (e.g., PINs).
- **Session Hijacking:** Previous user's session remaining active after card removal.
- **Unauthorized Access:** Unsecured physical ports or service panels.

3. Performance & Usability Bugs:

- **Slow Transactions:** Excessive lag in processing user requests.
- **System Freezes/Crashes:** Application unresponsiveness or unexpected reboots.
- **Unclear Error Messages:** Cryptic error codes instead of user-friendly explanations.
- **Touchscreen Unresponsiveness:** Inconsistent or delayed response to input.

4. Hardware & Integration Bugs:

- **Cash Dispenser Jams:** Frequent jamming of banknotes.
- **Card Reader Malfunctions:** Difficulty reading cards or frequent "re-insert card" prompts.
- **Offline Synchronization Issues:** Data inconsistencies when the ATM reconnects after being offline.

Problem No: 05

Problem Name: Write a program in "JAVA" or "C" to find out the factorial of a number using while or for loop. Also verify the results obtained from each case.

Objective:

To write a "JAVA" or "C" program that calculates the factorial of a positive integer using a while or for loop and verifies the result by comparing it with the output from a different loop-based method.

Theory:

The factorial of a number n (denoted as n!) is the product of all positive integers from 1 to n. It is mathematically defined as:

$n! = n \times (n - 1) \times (n - 2) \times ... \times 2 \times 1$

In programming, a factorial can be computed using iterative methods such as while or for loops. This program uses a while loop for the main calculation and a for loop to verify the result. Factorials are often used in mathematical computations like permutations, combinations, and series expansions.

Algorithm:

1. Start the program.
2. Declare an integer variable n and two unsigned long long variables for storing factorial results.
3. Prompt the user to enter a positive integer.
4. Check if n is non-negative.
5. Use a while loop to compute the factorial:
    - Initialize fact = 1 and i = 1.
    - While i <= n, multiply fact by i and increment i.
6. Use a for loop to compute the factorial again for verification.
7. Print both results.
8. Compare and display whether both results match.
9. End the program.

Source Code:

```
#include <stdio.h>

int factorialForLoop(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}

int factorialWhileLoop(int n) {
    int result = 1;
    int i = 1;
    while (i <= n) {
        result *= i;
        i++;
    }
    return result;
}

int main() {
    int number;
    printf("Enter a positive integer: ");
    scanf("%d", &number);

    if (number < 0) {
        printf("Factorial is not defined for negative numbers.\n");
        return 1;
    }

    int factFor = factorialForLoop(number);
    int factWhile = factorialWhileLoop(number);

    printf("Factorial using for loop: %d\n", factFor);
    printf("Factorial using while loop: %d\n", factWhile);

    if (factFor == factWhile) {
        printf("✓Both results match. Factorial is %d.\n", factFor);
    } else {
        printf("✗Mismatch in results! Check the logic.\n");
    }

    return 0;
}
```

Input:

Enter a positive integer: 5

Output:

Factorial using for loop: 120
Factorial using while loop: 120
Both results match. Factorial is 120.

Problem No: 06

Problem Name: Write a program in "JAVA" or "C" that will find sum and average of array using do while loop and 2 user-defined function.

Objective:

To learn how to calculate the sum and average of array elements using a do-while loop.

Theory:

In C programming, arrays are used to store multiple values of the same data type. Finding the **sum and average** of array elements is a common operation in data processing and numerical computations.

The **`do-while` loop** is an exit-controlled loop that ensures the body of the loop executes at least once before the condition is tested. This loop is particularly useful when the number of iterations is known and we want the statements inside the loop to run at least once.

In this experiment, the program takes user input to form an array of integers. It then calculates the **sum** using a `do-while` loop in a user-defined function. Another user-defined function is used to calculate the **average** from the computed sum and number of elements. This practice helps reinforce concepts like:

- Array traversal
- Use of `do-while` loops
- Function creation and reuse
- Input/output operations
- Basic arithmetic processing

Algorithm:

1. Start the program.
2. Input the size of the array from the user (`n`).
3. Declare an array of size `n`.
4. Use a `do-while` loop to take `n` elements as input from the user.
5. Call the user-defined function `calculateSum()`:
    - Use a `do-while` loop inside this function to compute the sum of array elements.
    - Return the sum.
6. Call the second function `calculateAverage()`:
    - Use the sum and size as parameters.
    - Return the average as a float.
7. Display the sum and average.
8. End the program.

Source Code:

```
#include <stdio.h>

// Function to calculate the sum of array elements
int calculateSum(int arr[], int size) {
    int sum = 0, i = 0;
    do {
        sum += arr[i];
        i++;
    } while (i < size);
    return sum;
}

// Function to calculate the average of array elements
float calculateAverage(int sum, int size) {
    return (float)sum / size;
}

int main() {
    int n;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];

    printf("Enter %d elements:\n", n);
    int i = 0;
    do {
        printf("Element %d: ", i + 1);
        scanf("%d", &arr[i]);
        i++;
    } while (i < n);

    int sum = calculateSum(arr, n);
    float avg = calculateAverage(sum, n);

    printf("\nSum of array elements = %d\n", sum);
    printf("Average of array elements = %.2f\n", avg);

    return 0;
}
```

Input:

Enter the number of elements: 5
Enter 5 elements:
Element 1: 3
Element 2: 5
Element 3: 6
Element 4: 7
Element 5: 4

Output:

Sum of array elements = 25
Average of array elements = 5.00

Problem No: 07

Problem Name: Write a simple "JAVA" program to explain classNotFound Exception and endOfFile(EOF) exception.

Objective:

To write a simple Java program that demonstrates the use of exception handling for ClassNotFoundException and EOFException, showing how these exceptions occur during dynamic class loading and file input operations, and how to handle them gracefully using try-catch blocks.

Theory:

In Java, **exceptions** are events that disrupt the normal flow of a program. They can be caught and handled using try-catch blocks. This experiment focuses on two specific **checked exceptions**:

1. **ClassNotFoundException**
   This exception occurs when the Java Virtual Machine (JVM) attempts to load a class using methods like Class.forName() and the specified class is not found in the classpath. It typically happens in dynamic class loading or JDBC operations.
2. **EOFException (End Of File Exception)**
   This exception is thrown when input operations are attempted beyond the end of a file or stream. It usually occurs during deserialization or when reading primitive data types using classes like ObjectInputStream or DataInputStream.

Algorithm:

1. Start the program.
2. Use a try block to:
   - Attempt to load a class using Class.forName("NonExistentClass").
   - Catch the ClassNotFoundException and print an appropriate message.
3. In another try block:
   - Create a file and write a single integer using ObjectOutputStream.
   - Read the integer back using ObjectInputStream.
   - Attempt to read again from the file, which triggers an EOFException.
   - Catch and handle EOFException and IOException.
4. Display appropriate exception messages.
5. End the program.

Source code:

```
import java.io.*;

public class ExceptionDemo {

   public static void main(String[] args) {
      // Demonstrating ClassNotFoundException
      try {
         Class.forName("NonExistentClass");
      } catch (ClassNotFoundException e) {
         System.out.println("ClassNotFoundException caught:");
         System.out.println(e.getMessage());
      }

      // Demonstrating EOFException
      try {
         // Create a file and write some data
         FileOutputStream fos = new FileOutputStream("data.txt");
         ObjectOutputStream oos = new ObjectOutputStream(fos);
         oos.writeInt(100);
         oos.close();

         // Now try to read more data than available
         FileInputStream fis = new FileInputStream("data.txt");
         ObjectInputStream ois = new ObjectInputStream(fis);

         System.out.println("Read Integer: " + ois.readInt());

         // Trying to read again causes EOFException
         System.out.println("Read again (will cause EOFException):");
         System.out.println(ois.readInt()); // Nothing more to read

         ois.close();

      } catch (EOFException e) {
         System.out.println("EOFException caught: End of file reached.");
      } catch (IOException e) {
         System.out.println("IOException caught: " + e.getMessage());
      }
   }
}
```

Output:

ClassNotFoundException caught:

NonExistentClass

Read Integer: 100

Read again (will cause EOFException):

EOFException caught: End of file reached.

Problem No: 08

Problem Name: Write a program in "JAVA" or "C" that will read a input.txt file containing n positive integers and calculate addition, subtraction, multiplication and division in separate output.txt file. Sample input: 5 5 9 8; Sample output: Case-1:10 0 25 1; Case-2: 17 1 72 1.

Objective:

To develop a program in **Java** or **C** that reads n positive integers from an input.txt file, performs pairwise **addition**, **subtraction**, **multiplication**, and **division**, and writes the results for each pair to an output.txt file in a structured format.

Theory:

File handling in C allows programs to read data from files and write data to files, enabling persistent data storage. This program reads n positive integers from an input file input.txt and performs arithmetic operations (addition, subtraction, multiplication, and division) on pairs of numbers.

- The program assumes the input contains pairs of numbers. For each pair:
  - Addition: sum of the two numbers.
  - Subtraction: difference (first number minus second).
  - Multiplication: product of the two numbers.
  - Division: integer division (first number divided by second).
- The results for each pair are written to an output file output.txt with appropriate labeling.
- Basic file operations in C involve using fopen, fprintf, fscanf, and fclose.

Algorithm:

1. Start the program.

2. Open the input file `input.txt` in read mode.

3. Open the output file `output.txt` in write mode.

4. Repeat for each line in the input file:

   - Read the line using `fgets()`.
   - Tokenize the line to extract integers using `strtok()`.
   - Store the integers in an array.

5. Initialize `sum = 0`, `sub = first number`, `mul = 1`, `div = first number`.

6. Loop through the array:

   - Add each number to `sum`.
   - Subtract numbers from `sub` (starting from second number).
   - Multiply each number to get `mul`.
   - Divide `div` by each subsequent number (check for zero).

7. Write results to `output.txt` in the format:
```
Case-x: sum sub mul div
```

8. Close both input and output files.

9. End the program.

Source Code:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
   FILE *fin = fopen("input.txt", "r");
   FILE *fout = fopen("output.txt", "w");

   if (fin == NULL) {
      printf("Error opening input.txt\n");
      return 1;
   }
   if (fout == NULL) {
      printf("Error opening output.txt\n");
      fclose(fin);
      return 1;
   }
   int numbers[1000];
   int count = 0;
   while (fscanf(fin, "%d", &numbers[count]) == 1) {
      count++;
   }
   if (count % 2 != 0) {
      fprintf(fout, "Error: Odd number of integers in input file.\n");
      fclose(fin);
      fclose(fout);
      return 1;
   }
   for (int i = 0, case_num = 1; i < count; i += 2, case_num++) {
      int a = numbers[i];
      int b = numbers[i + 1];
      int add = a + b;
      int sub = a - b;
      int mul = a * b;
      int div = (b != 0) ? (a / b) : 0;  // handle division by zero gracefully
      fprintf(fout, "Case-%d: %d %d %d %d\n", case_num, add, sub, mul, div);
   }
   fclose(fin);
   fclose(fout);

   printf("Processing complete. Check output.txt\n");

   return 0;
}
```

Input:

Enter the number: 5 5 9 8

Output:

Case-1: 10 0 25 1
Case-2: 17 1 72 1

Problem No: 09

Problem Name: Explain the role of software engineering in Biomedical Engineering and in the field of Artificial Intelligence and Robotics.

Objective:

To understand and explain the crucial role of software engineering in advancing **Biomedical Engineering** and its applications, as well as its impact on the development of **Artificial Intelligence (AI)** and **Robotics**, highlighting how software solutions improve healthcare technologies, intelligent systems, and automation.

Theory:

1. Role of Software Engineering in Biomedical Engineering

A. Medical Software Development

Software engineers develop applications and systems used in hospitals, clinics, and diagnostic labs.
Examples include:

- **Electronic Health Records (EHR) systems**
- **Medical imaging software** (e.g., MRI, CT scan analysis)
- **Patient monitoring systems**
- **Decision support systems** for doctors

B. Biomedical Device Control

- Many biomedical devices (like pacemakers, infusion pumps, or prosthetics) are software-driven.
- Software engineers write the **embedded software** that controls these devices.

C. Data Analysis and Signal Processing

- Biomedical signals (like ECG, EEG, EMG) require specialized software to:
    - Filter noise
    - Detect patterns or anomalies
    - Aid in diagnosis (e.g., detecting arrhythmia from ECG)
- Software engineering ensures this is done **efficiently, accurately, and safely**.

D. Regulatory and Safety Compliance

- Software in biomedical systems must meet **strict regulatory standards** (e.g., FDA, ISO 13485).
- Software engineers must document development and perform **extensive testing and validation**.

E. Modeling and Simulation

- Tools like MATLAB or Simulink are used for:
    - Simulating biological systems
    - Testing control algorithms before implementation in physical devices

2. Role of Software Engineering in Artificial Intelligence and Robotics

A. Algorithm Development

- AI systems rely on **complex algorithms** for:
    - Machine learning
    - Natural language processing
    - Computer vision
- Software engineers implement, optimize, and maintain these algorithms in real-world applications.

B. Robotics Control Systems

- Robots need **real-time control systems** to move, sense, and respond.
- Software engineers write:
    - **Firmware** for motor and sensor control
    - **Middleware** like ROS (Robot Operating System)
    - **High-level logic** for navigation, object recognition, etc.

C. Simulation and Testing

- Before deploying AI models or robots in the real world, software engineers:
    - Use **simulators** (e.g., Gazebo, Unity, Webots)
    - Perform **unit testing**, **integration testing**, and **stress testing**

D. Ethics and Safety

- Engineers must design systems that are:
    - Safe to operate
    - Explainable and unbiased (especially in AI)
    - Fail-safe in case of hardware or software malfunction

E. Human-Robot Interaction (HRI)

- Software engineers build interfaces that allow humans to interact naturally with robots using:
- Voice commands

Problem No: **10**

Problem Name: Study the various phases of Water-fall model. Which phase is the most dominated one?

Objective:

To study and understand the different phases of the **Waterfall Software Development Model** and identify which phase typically dominates or requires the most effort in the overall software development process.

Theory:

The **Waterfall Model** is one of the earliest and most straightforward software development life cycle (SDLC) models. It is a **linear and sequential approach**, where each phase must be completed before the next begins. Here's an overview of its **phases** and an analysis of the **most dominant phase**.

Phases of the Waterfall Model

1. Requirement Analysis

- Involves gathering and documenting all user and system requirements.
- Outputs a **Software Requirements Specification (SRS)** document.
- No coding or designing happens here—focus is on what the system should do.

2. System Design

- Based on the SRS, system architecture and design are created.
- Includes:
    - High-level design (HLD) – overall system architecture
    - Low-level design (LLD) – module-level details
- Determines programming language, data structures, UI design, etc.

3. Implementation (Coding)

- Developers write the actual source code according to the design documents.
- This phase turns design into a working product (software application).

4. Integration and Testing

- Individual modules are integrated and tested as a complete system.
- Focus is on:
    - Unit testing
    - System testing
    - Acceptance testing
- Objective: detect and fix bugs, verify functionality.

5. Deployment

- The fully tested product is delivered to the customer or deployed in a live environment.

6. Maintenance

- Ongoing support:
    - Bug fixes
    - Updates or enhancements
    - Performance improvements
- Longest-lasting phase in many real-world projects.

Most Dominant Phase: Requirement Analysis

- **Foundation of the entire project**: If requirements are wrong, everything built after will be flawed.
- It **influences all other phases**: design, implementation, testing, etc.
- Waterfall assumes that requirements are fixed early; so **thorough, accurate analysis** is critical.
- Errors in this phase are **costlier to fix later** than those made in coding or testing.

"Poor requirements gathering is the leading cause of software project failure."

Alternative Perspective:

Some may argue that **Maintenance** dominates in terms of time and effort in the long run, especially for large systems. However, from a **process-centric and planning viewpoint**, **Requirement Analysis** holds the most **influence** and is considered the **most critical (and dominant)** in the Waterfall model

Problem No: 11

Problem Name: Using COCOMO model estimate effort for specific problem in industrial domain.

Objective:

COCOMO is a software estimation model developed by **Barry Boehm** to predict the cost, effort, and schedule of a software project. It uses mathematical formulas based on historical project data.

There are three types of COCOMO models:

1. **Basic COCOMO** – Uses only size (KLOC) to estimate effort.
2. **Intermediate COCOMO** – Adds cost drivers (factors like reliability, complexity, experience).
3. **Detailed COCOMO** – Includes all features of intermediate and more breakdown by phase.

For an **industrial domain**, we commonly use the **Intermediate COCOMO** model to account for complexity and real-world factors.

Theory:

The COCOMO (Constructive Cost Model) model is used to estimate the effort, cost, and schedule for software projects based on the size of the software and various project characteristics. To estimate

The **Basic COCOMO Model** estimates:

- **Effort (in person-months)** using the formula:

(PM)Effort (PM)=a×(KLOC)b

Where:

- KLOC = Thousands of Lines of Code
- a, b = constants based on the project type:
    - **Organic** (simple, small teams, familiar domain)
    - **Semi-detached** (medium-sized, mixed experience)
    - **Embedded** (complex, tightly coupled with hardware or regulations)

Problem Scenario: Industrial Process Monitoring System

Let's assume you're developing a **real-time industrial monitoring system** (e.g., to monitor pressure, temperature, etc., in a chemical plant).

Assumptions:

- Estimated size: **50 KLOC**
- Project Type: **Embedded** (since it's safety-critical and hardware-interfacing)
- Basic COCOMO constants for Embedded:
    - a = 3.6, b = 1.20

COCOMO Effort Estimation

Effort=3.6×(50)1.20

Let's calculate:

(50)1.20≈109.4 person-monthsEffort=3.6×109.4≈393.8 person-months

Development Time (Optional)

You can also estimate development time:

(months)Time (months)=c×(Effort)d

For Embedded: c = 2.5, d = 0.32

monthsTime=2.5×(393.8)0.32≈2.5×7.2≈18 months

Using the COCOMO model, the estimated effort for a 25 KLOC industrial (embedded) project is approximately 195.5 person-months, and it would take about 14.7 months to complete with an average-size team. These estimates help in planning resources, budgeting, and scheduling in industrial software projects.

Problem No: 12

Problem Name: Identify the reasons behind software crisis and explain the possible solutions for the following scenario:

Case 1: "Air ticket reservation software was delivered to the customer and was installed in an airport 12.00 AM (mid night) as per the plan. The system worked quite fine till the next day 12.00 PM (noon). The system crashed at 12.00 PM and the airport authorities could not continue using software for ticket reservation till 5.00 PM. It took 5 hours to fix the defect in the software".

Case 2: "Software for financial systems was delivered to the customer. Customer conformed the development team about a mal-function in the system. As the software was huge and complex, the development team could not identify the defect in the software".

Objective:

The term **"Software Crisis"** refers to the set of problems faced during the early days of software engineering and still relevant today in large or critical systems. These include difficulties in software development such as project overruns, low quality, poor maintainability, and system failures.

Theory:

The term "Software Crisis" refers to the set of problems faced during the early days of software engineering and still relevant today in large or critical systems.

Reasons Behind the Software Crisis

1. **Poor Requirements Analysis**
   Inadequate understanding of what the software should do leads to incorrect or incomplete functionality.
2. **Lack of Design Standardization**
   Ad hoc designs result in unscalable and hard-to-maintain systems.
3. **Inadequate Testing**
   Systems are released without rigorous testing, especially in edge cases or under real-time conditions.
4. **Complexity of Software**
   As systems become larger, managing interdependencies becomes very difficult.
5. **Lack of Proper Documentation**
   When code is not well-documented, debugging and maintenance become harder.
6. **Underestimation of Project Time and Resources**
   Unrealistic timelines and insufficient staffing lead to rushed, buggy code.
7. **Poor Maintenance Planning**
   Post-delivery issues are not addressed efficiently due to the lack of ongoing support structures.

Case Analysis and Solutions

Case 1: Airport Ticket Reservation System Crash

Issue Summary:

- The system worked fine for 12 hours (from midnight to noon) and then crashed.
- System was unusable for 5 hours.

- This implies a **runtime defect** related to time handling or load handling.

Possible Root Causes:

- **Time Format Bug**: A 12-hour vs 24-hour time format issue (e.g., confusion between AM/PM or time rollover).
- **Memory Leak**: A resource leak that accumulated until system resources were exhausted.
- **Load Threshold Exceeded**: System not tested under real-world traffic, leading to overload.
- **Date-Time Overflow Bug**: A bug in a counter or timestamp function.
- **Insufficient Logging and Monitoring**: Delayed response time in fixing the bug.

Solutions:

1. **Rigorous Boundary Testing**
    - Test edge cases like 11:59 AM to 12:01 PM transitions.
    - Validate AM/PM vs 24-hour handling.
2. **Load and Stress Testing Before Deployment**
    - Simulate real-world ticketing loads for extended periods.
3. **Automated Monitoring and Logging**
    - Deploy monitoring tools that alert immediately on abnormal behavior.
4. **Fallback and Redundancy Planning**
    - Deploy redundant systems or failover options to avoid total outages.
5. **Hotfix Framework in Place**
    - Ensure modular patching is possible without system downtime.

Case 2: Financial System Malfunction – Hard to Identify Defect

Issue Summary:

- The system is **large and complex**.
- A defect was reported, but developers were **unable to locate it** easily.

Possible Root Causes:

- **Poor Modularity**: Large monolithic codebase makes defect tracing difficult.
- **Lack of Logging/Tracing**: Inability to reproduce the issue or track its source.
- **Inadequate Documentation**: Developers can't understand how different components interact.
- **Code Complexity**: High cyclomatic complexity or tangled logic.
- **Insufficient Unit/Integration Tests**: Hard to pinpoint which module fails.

Solutions:

1. **Modular Design and Architecture**
    - Follow principles like **Separation of Concerns** and **Single Responsibility**.
    - Use microservices where appropriate.

2. **Advanced Debugging Tools**
    - Use profilers, debuggers, and log analyzers to trace defects.
3. **Detailed Logging and Auditing**
    - Implement application-wide logging at various levels (info, warn, error).
4. **Test-Driven Development (TDD)**
    - Writing tests before code improves defect identification and localization.
5. **Code Refactoring and Documentation**
    - Periodically refactor the code to reduce complexity.
    - Maintain updated architecture diagrams and module interaction maps.
6. **Team Knowledge Sharing**
    - Conduct walkthroughs and knowledge transfer sessions regularly.

Conclusion

The software crisis emerges mainly from **lack of proper planning, testing, and design practices**. In both cases, adopting **engineering best practices, rigorous testing, modular architecture, and active monitoring** can prevent or mitigate such failures.