Software Testing & Quality Assurance Lab
(IT-662)

Submitted in partial fulfillment of the requirements for the award of the degree

Of

Masters of Computer Application (SE)

Submitted To

Ms. Sanchi Kalra Dang

Lab Instructor

USIC&T

Submitted By

Khushal Sachdeva

Enroll. No. 00516404524

Semester - II (Section-1)

University School of Information , Communication & Technology

Guru Gobind Singh Indraprastha University

Sector-16C , Dwarka New Delhi - 110078

(2024-2026)

INDEX

Practical - 1

Aim: To perform Requirements Elicitation & Use Case Modeling to represent the project system's structure and behavior.

1. Problem Statement: Training and Placement Cell Portal

The Training and Placement Cell (TnP) at USICT College plays a crucial role in bridging the gap between students and potential employers. However, the current manual processes and communication gap create inefficiencies, delays, and confusion, leading to a suboptimal experience for all stakeholders involved.

1. Inefficient Communication: Communication between students, coordinators, and recruiters is often fragmented, leading to missed updates and opportunities.

2. Tracking Placement Progress: Students lack a clear, real-time overview of their status in the placement process, causing uncertainty and anxiety.

3. Recruiter Management: Recruiters face challenges in registering and managing their placement sessions, job openings, and interactions with students.

4. Manual Updates and Processes: Coordinators struggle with manually managing, coordinating with students and updating the status of multiple students and recruiters, leading to potential errors and delays.

2. Suitable SDLC model for the system to be developed.

For developing the Training and Placement Cell Portal, the **Agile Software Development** process model would be a suitable choice. Here's why:

1. **Iterative Development**: Agile emphasizes delivering small, functional components over time. This is important for the portal because you can develop key features like student dashboards, recruiter portals, and coordinator tools incrementally. This allows you to gather feedback and make improvements at each stage, reducing the risk of building features that don't align with user needs.
2. **Flexibility**: Given the complexity of the portal with different user roles and functionalities, Agile allows for adaptability. If student or recruiter feedback indicates a need for new features or adjustments, it can be addressed in the next sprint. This is harder to achieve in models like Waterfall, which lack flexibility after requirements are defined.
3. **Collaboration and Communication**: Agile promotes constant communication among all stakeholders, which aligns with the key challenges we're solving — especially ensuring efficient communication between students, coordinators, and recruiters.
4. **Continuous Improvement**: Agile allows for ongoing testing and integration, so issues can be identified and fixed throughout the development cycle, leading to a more polished final product.
5. **User-Centered Design**: Agile encourages a focus on delivering usable features with each iteration, which means that both student and recruiter-specific functionalities can be validated early.
6. **Risk Mitigation**: Continuous testing and feedback throughout the development lifecycle reduce the chance of failure. By catching issues early and refining functionality iteratively, Agile ensures that the system is always moving toward a production-ready state.

This model will enable our portal to evolve effectively in response to real user needs and provide a more streamlined, efficient development process.

Benefits

1. Agile is better suited for complex systems because it offers structured planning and clear prioritization through **backlog management, sprints, and user stories**.
2. Agile, prioritizes **building a fully functional system** in incremental steps. The focus is on delivering working features with each iteration, ensuring that even early versions of the software are production-ready.
3. While our project involves multiple user-specific features (for students, recruiters, coordinators), **Agile can handle evolving requirements** through regular feedback loops, while still maintaining focus on delivering usable functionality with each iteration.
4. Agile encourages frequent communication and collaboration between the development team and all stakeholders through daily stand-ups, reviews, and retrospectives.
5. Agile incorporates continuous testing and quality control throughout the development process, ensuring that bugs and issues are caught early and addressed immediately.
6. While the Evolutionary Model offers flexibility, Agile provides a **more structured and balanced approach** that can handle the complexity of our system, ensure better communication, and deliver a fully functional product in each iteration. This will help us ensure that students, recruiters, and coordinators get a seamless and efficient experience from the outset.

3. Software Requirement Specification Document (SRS)

1. Introduction

1.1 Purpose

The purpose of this SRS document is to outline the functional and non functional requirements for the development of the Training and Placement Cell (TnP) Portal for USICT College. This portal aims to bridge the communication gap between students, recruiters, and coordinators, streamline placement processes, and provide an efficient, user-friendly platform for all stakeholders.

1.2 Scope

The TnP Portal will serve as a centralized platform for final-year students seeking internships and full-time job opportunities. The system will allow recruiters to post job openings, manage applications, and interact with students. Placement coordinators will use the portal to schedule placement activities, track student progress, and communicate with both students and recruiters. The portal will also replace manual processes, improving overall efficiency.

1.3 Definitions, Acronyms, and Abbreviations

- **TnP**: Training and Placement
- **SRS**: Software Requirements Specification
- **USICT**: University School of Information, Communication & Technology
- **SDLC**: Software Development Life Cycle
- **API**: Application Programming Interface
- **FR**: Functional Requirement
- **NFR**: Non-Functional Requirement

1.4 Overview

This SRS document includes the overall system architecture, functional and non-functional requirements, external interface requirements, system constraints, and use cases for the TnP Portal. It also describes the system's expected performance, scalability, and usability features.

2. Overall Description

2.1 System Overview

The TnP Portal is designed to manage interactions between students, recruiters, and placement coordinators. The system will feature:

- A student dashboard to track job listings, application status and its progress tracking.
- A recruiter portal to manage job postings, review student applications, and schedule interviews.
- Placement coordinator manages placement events, tracks student's progress, and verifies student profiles.

2.2 Product Perspective

The portal will be developed as a web application, operating independently of any existing systems but with the potential for future integrations (e.g., LinkedIn, document verification systems). It will feature role-based access, ensuring students, recruiters, and placement coordinators have tailored views and functionalities.

2.3 User Classes and Characteristics

1. **Students**: Final-year students looking for internships or full-time jobs. They need access to job postings, application tracking, and notifications.
2. **Recruiters**: Company representatives who post job listings, manage student applications, and schedule interviews.
3. **Placement Coordinators**: Faculty responsible for managing the entire placement process, from job postings to tracking students' placement progress.

2.4 Operating Environment

- **Browsers**: Chrome, Firefox, Safari (latest versions)
- **Platforms**: Desktop, tablet, mobile devices
- **Operating Systems**: Windows, macOS, Linux, iOS, Android

2.5 Constraints

- Must be scalable to support up to 1000 simultaneous users.
- Must be secure, protecting user data with encryption and multi-factor authentication.
- Compliance with accessibility standards (e.g., WCAG 2.0).

2.6 Assumptions and Dependencies

- Users will have stable internet access.
- Integration with third-party services like LinkedIn and document verification systems may be required.
- System downtime should not exceed 0.1% during placement seasons.

3. System Features

3.1 Student Dashboard

Description:

- Students can view job and internship listings filtered by eligibility, company, package, etc.
- Students can upload and manage resumes and certificates.
- Notifications are sent for new job listings and application status changes.

Functional Requirements:

- **FR1**: The system shall allow students to view job/internship listings filtered by eligibility criteria.
- **FR2**: The system shall enable students to upload and manage documents (e.g., resumes, certificates).
- **FR3**: The system shall notify students of job application status and deadlines.
- **FR4:** The system shall track the placement progress of students through the following stages:
  1. Apply
  2. Resume Shortlisting
  3. Task Submission
  4. Task-Based Shortlisting
  5. Role based Interview
  6. HR Interview
  7. Result Awaited
  8. Selected / Not Selected

3.2 Recruiter Portal

Description:

- Recruiters can register, post job openings, and manage applications.
- Recruiters can schedule interviews and communicate with shortlisted candidates.

Functional Requirements:

- **FR5**: The system shall allow recruiters to post job openings with eligibility criteria.
- **FR6**: The system shall allow recruiters to manage and shortlist student applications.
- **FR7**: The system shall enable recruiters to schedule interviews with shortlisted students.

3.3 Placement Coordinator Tools

Description:

- Coordinators can verify and manage student profiles and documents.
- Coordinators can schedule placement activities like tests and interviews.
- Coordinators can track the placement progress of students and generate reports.

Functional Requirements:

- **FR8**: The system shall allow coordinators to verify student profiles and documents.
- **FR9**: The system shall allow coordinators to schedule placement activities (e.g., interviews, tests).
- **FR10**: The system shall allow coordinators to track placement statistics and generate reports.

3.4 Integrated Communication

Description:

- The system provides communication tools, including email notifications and in-portal messaging between students, recruiters, and coordinators.

Functional Requirements:

- **FR11**: The system shall provide email notifications for important updates (e.g., job postings, interview schedules).
- **FR12**: The system shall enable real-time messaging between students, recruiters, and coordinators.

3.5 Reports and Analytics

Description:

- The system can generate reports on placement statistics, student engagement, and company participation.

Functional Requirements:

- **FR13**: The system shall generate reports on student placement performance.
- **FR14**: The system shall visualize placement trends over time.

4. External Interface Requirements

4.1 User Interfaces

- **Responsive Design**: The portal must be accessible on desktop, tablet, and mobile devices.
- **Accessibility**: The system must comply with WCAG 2.0 accessibility standards.

4.2 Software Interfaces

- Integration with LinkedIn for profile synchronization.
- Integration with third-party document verification services.

4.3 Hardware Interfaces

- No special hardware requirements are needed apart from standard desktop or mobile devices.

4.4 Communication Interfaces

- Email, push notifications, and SMS (optional) for key updates.

5. Non-Functional Requirements

5.1 Performance

- The system shall handle up to 1000 simultaneous users with an average response time of less than 3 seconds.

5.2 Security

- All sensitive data shall be encrypted using standard encryption algorithms (e.g., AES-256).
- Multi-factor authentication (MFA) will be required for students, recruiters, and coordinators during login.

5.3 Usability

- The user interface shall be intuitive and easy to navigate, with a focus on user-friendly design for all user classes.

5.4 Availability

- The system must have 99.9% uptime, especially during peak placement seasons.

5.5 Scalability

- The system should be able to handle a growing number of users and job listings without degradation in performance.

5.6 Maintainability

- The system should be modular and maintainable with minimal downtime during updates.

5.7 Reliability

- The system must ensure accurate tracking and updating of student placement status.

5.8 Portability

- The system should be accessible across different operating systems (Windows, macOS, Linux) and devices (desktops, tablets, smartphones).

6. Use Cases

Use Case 1: Student Applies for a Job

Primary Actor: Student
Description: A student logs in, views job listings, applies for a job, and is notified of updates.
Precondition: The student has an active profile.
Postcondition: The job application is successfully submitted.

Use Case 2: Recruiter Posts a Job

Primary Actor: Recruiter
Description: A recruiter logs in, posts a job with specific eligibility criteria, and students are notified.
Precondition: The recruiter has an active account.
Postcondition: The job listing is posted and visible to eligible students.

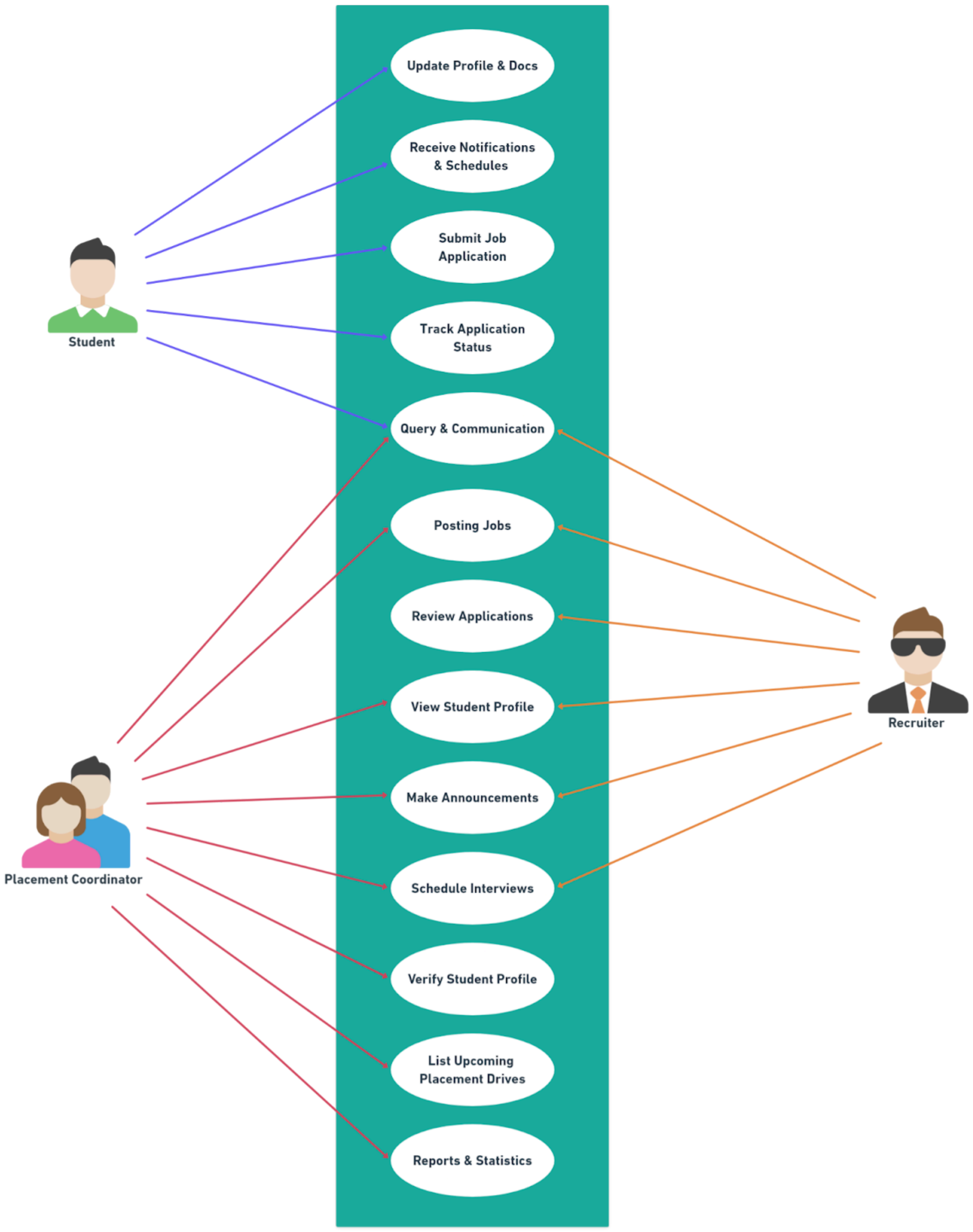
Use Case 3: Coordinator Schedules an Interview

Primary Actor: Coordinator
Description: The coordinator schedules interviews for shortlisted students, and both students and recruiters are notified.
Precondition: Students have been shortlisted.
Postcondition: Interview schedules are created and notifications are sent.

7. Appendices

- **Technology Stack**:
  - Frontend: React.js, Next.js, TypeScript, Tailwind CSS
  - Backend: Node.js, Express.js
  - Database: MongoDB
- **Third-Party Integrations**: LinkedIn, document verification services, SMS/Email notification systems.

Use Case Diagram



Practical - 2

Aim: To Identify bad smells in a project.

Project: Hospital Management System

// Doctor.java

```
package HospitalManagement;
public class Doctor {
    String name;                //Name of doctor
    String qualification;       //Qualification of doctor
    String experience;       //Experience of doctor
    String salary;              //Given Salary of doctor
    public Doctor() {   //Default Constructor
        name = null;
        qualification = null;
        experience = null;
        salary = null;
    }
    public Doctor(String drName,String quali,String expi,String sal){
        //sets info
        name = drName;
        qualification = quali;
        experience = expi;
        salary = sal;
    }
    public String toString() {
        return name + ", " + qualification + ", " + experience + ", " + salary+".";
    }
}
```

// Patient.java

```
package HospitalManagement;
public class Patient {
    String nameP;       //Name of patient
    String age;       //Age of patient
    String Illness;     //Disease
    String bloodGroup;  //Blood Group pf patient
    public Patient() {  //Default Constructor
        nameP = null;
        age = null;
        Illness = null;
        bloodGroup = null;
    }
    public Patient(String ptName,String ag,String ill,String bg){
        //sets info
        nameP = ptName;
        age = ag;
        Illness = ill;
        bloodGroup = bg;
    }
    public String toString() {
        return nameP + ", " + age + ", " + Illness + ", " + bloodGroup+".";
    }
}
```

// Main.java

```
package HospitalManagement;
import java.util.ArrayList;
import java.util.Scanner;
public class Main {
    static ArrayList<Patient> pt=new ArrayList<Patient>();
    static ArrayList<Doctor> dr=new ArrayList<Doctor>();
    public static void main(String[] args) {
        System.out.println("*************************************************");
        System.out.println("*            WELCOME TO MY HOSPITAL             *");
        System.out.println("*************************************************");
        System.out.println("Enter Choice\n");
        System.out.println("1. Add Doctor");
        System.out.println("2. Add Patient");
        System.out.println("3. Show Staff");
        System.out.println("4. Show Patient");
        System.out.println("5. Discharge Patients");
        Scanner ch = new Scanner(System.in);
        while (true) {
            System.out.print("Enter your choice: ");
            int x = ch.nextInt();
            switch (x) {
                case 1 : {
                    Scanner setD = new Scanner(System.in);
                    System.out.println("Enter information of Doctor");
                    System.out.println("---------------------------");
                    System.out.print("Name of Doctor: ");
                    String tempName = setD.nextLine();
                    System.out.print("Enter Qualification: ");
                    String tempQ = setD.nextLine();
                    System.out.print("Enter Experience: ");
                    String tempEx = setD.nextLine();
                    System.out.print("Enter Salary given to Doctor: ");
                    String tempSal = setD.nextLine();
                    dr.add(new Doctor(tempName, tempQ, tempEx, tempSal));
         System.out.println("-----------------------------------------------------");
```

```
                    break;
                }
                case 2 : {
                    Scanner setP = new Scanner(System.in);
                    System.out.println("Enter information of Patient");
                    System.out.println("-----------------------------");
                    System.out.print("Name of Patient: ");
                    String tempNameP = setP.nextLine();
                    System.out.print("Enter Age: ");
                    String tempAg = setP.nextLine();
                    System.out.print("Enter Illness: ");
                    String tempIll = setP.nextLine();
                    System.out.print("Enter Blood Group: ");
                    String tempBg = setP.nextLine();
                    pt.add(new Patient(tempNameP, tempAg, tempIll, tempBg));
System.out.println("-----------------------------------------------------");
                    break;
                }
                case 3 : {
                    System.out.println("NAME, QUALIFICATIONS, EXPERIENCE, SALARY: ");
                    for (int k = 0; k < dr.size(); k++) {
                        System.out.print(k+1+".");
                        System.out.println(dr.get(k));
                    }
                    break;
                }
                case 4 : {
                    System.out.println("NAME, AGE, ILLNESS, BLOOD GROUP:    ");
                    for (int k = 0; k < pt.size(); k++) {
                        System.out.print(k+1+".");
                        System.out.println(pt.get(k));
                    }
                    break;
                }
                case 5 : System.out.println("Coming Soon....");
                break;
                default : System.out.println("Wrong Choice\n   Try Again!");
            }
        }
    }
}
```

Here are some code smells in the project:

Application-Level Smells

Bad Smell	Description	Example in Code	Refactoring
Primitive Obsession	Using primitive types (e.g., String) for numeric fields instead of proper data types.	Patient and Doctor classes store age and salary as String instead of int or double.	**Encapsulate Fields** Use int age and double salary with validation in setters.
Data Clump	Related data (e.g., patient/doctor details) stored redundantly.	Main class uses ArrayList<Patient> and ArrayList<Doctor> without encapsulation.	**Introduce Repository Class** Create a HospitalRepository class to centralize data storage and retrieval logic.
Duplicate Code	Similar input-handling logic duplicated for doctors and patients.	case 1 (Add Doctor) and case 2 (Add Patient) in Main.java use nearly identical code for input collection.	**Extract Method** Create a shared method (e.g., promptUserForDetails()) to reduce redundancy.

Method-Level Smells

Bad Smell	Description	Example in Code	Refactoring
Long Method	The main method handles UI, logic, and data storage, making it overly complex.	Main.java contains a large while loop with nested switch cases and input/output logic.	Extract Classes Split into HospitalUI (handles menus) and HospitalController (manages logic).
Incomplete Error Handling	Missing validation for user inputs (e.g., age as a number).	No checks for invalid inputs like non-numeric age or salary in Main.java.	Add Validation Use try-catch blocks or regex to validate inputs (e.g., age must be numeric).
Dead Code	Unimplemented functionality.	case 5 (Discharge Patients) prints "Coming Soon..." but lacks logic.	Implement or Remove Either add discharge logic or remove the placeholder code.

Class-Level Smells

Bad Smell	Description	Example in Code	Refactoring
Large Class	Main class violates Single Responsibility Principle by handling multiple roles.	Main manages UI, data storage, and business logic.	Separate Concerns Divide into HospitalUI, HospitalService, and HospitalRepository classes.
Feature Envy	Main class directly accesses and manipulates Patient/Doctor data.	Main adds/retrieves objects from ArrayList instead of delegating to a dedicated class.	Move Method Shift data management to a HospitalService class.
Shotgun Surgery	Adding a new feature requires modifying multiple sections of Main.	To add a "Search Patient" feature, changes would be needed in UI, logic, and data layers within Main.	Decouple Logic Use a modular design (e.g., MVC) to isolate layers.

Bloaters

Smell	Impact on Code	Example
Long Parameter List	Methods with excessive parameters reduce readability.	Doctor constructor: Doctor(String drName, String quali, String expi, String sal).
Primitive Obsession	Over-reliance on primitives limits data integrity.	Storing bloodGroup as String without validation (e.g., "AB+", "O-").

Refactoring:

- Introduce Data Objects (e.g., BloodGroup class with validation).
- Use Builder Pattern for constructing Doctor/Patient objects.

Dispensables

Smell	Description	Example
Speculative Generality	Unused or placeholder code.	Case 5 (Discharge Patients) has no implementation.
Dead Code	Redundant variables or methods.	Multiple Scanner instances (setD, setP) in Main.java.

Refactoring:

- Remove Redundant Code (e.g., use a single Scanner instance).
- Delete Unused Features or implement them fully.

Couplers

Smell	Description	Example
Inappropriate Intimacy	Main class tightly coupled with Patient/Doctor data structures.	Directly modifying ArrayList<Patient> and ArrayList<Doctor> in Main.java.
Message Chains	Deep access to nested data.	pt.get(k).toString() in case 4 assumes Patient fields are accessible.

Refactoring:

- Hide Data Access behind service classes (e.g., HospitalService.getPatientDetails()).

Object-Orientation Abusers

Smell	Description	Example
Switch Statements	Overuse of switch in Main.java for menu handling.	Large switch (x) block in Main's loop.
Temporary Field	Variables like Scanner ch are declared at class level unnecessarily.	static Scanner ch = new Scanner(System.in); in Main.java.

Refactoring:

- Replace Conditional with Polymorphism (e.g., use command pattern for menu options).
- Limit Variable Scope (e.g., declare Scanner inside methods).

Refactoring Recommendations

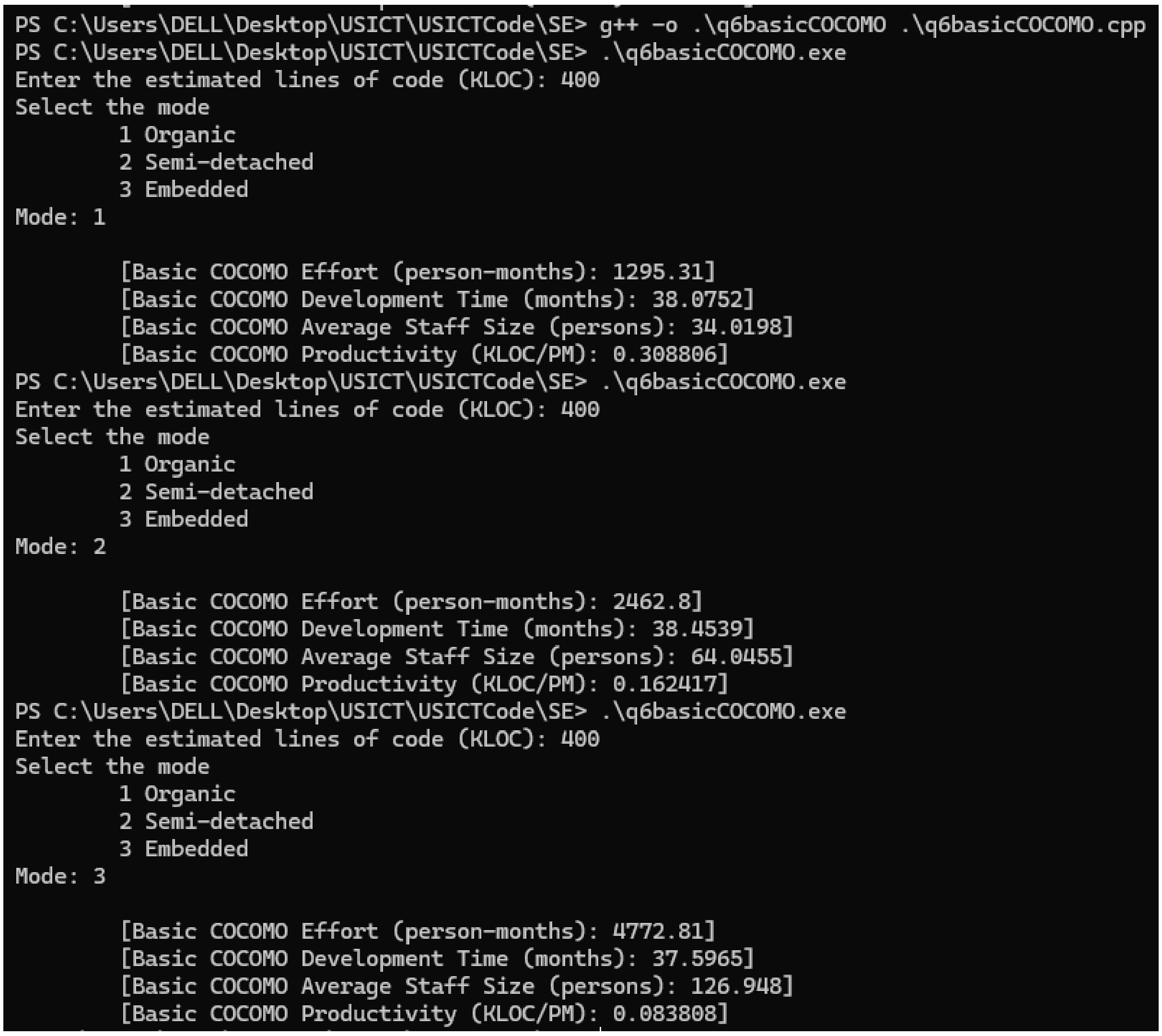
1. Separate UI and Business Logic
   - Create HospitalUI for menus and HospitalController for logic.
2. Encapsulate Data Storage
   - Introduce HospitalRepository to manage ArrayList operations.
3. Validate Inputs
   - Ensure age, salary, and bloodGroup follow valid formats.
4. Implement Discharge Feature
   - Add logic to remove patients from the list and update records.
5. Adopt MVC Pattern
   - Decouple Model (Patient, Doctor), View (HospitalUI), and Controller (HospitalController)**.**

Practical - 3

Aim: To determine cost estimation of project using COCOMO Model

Basic COCOMO Model

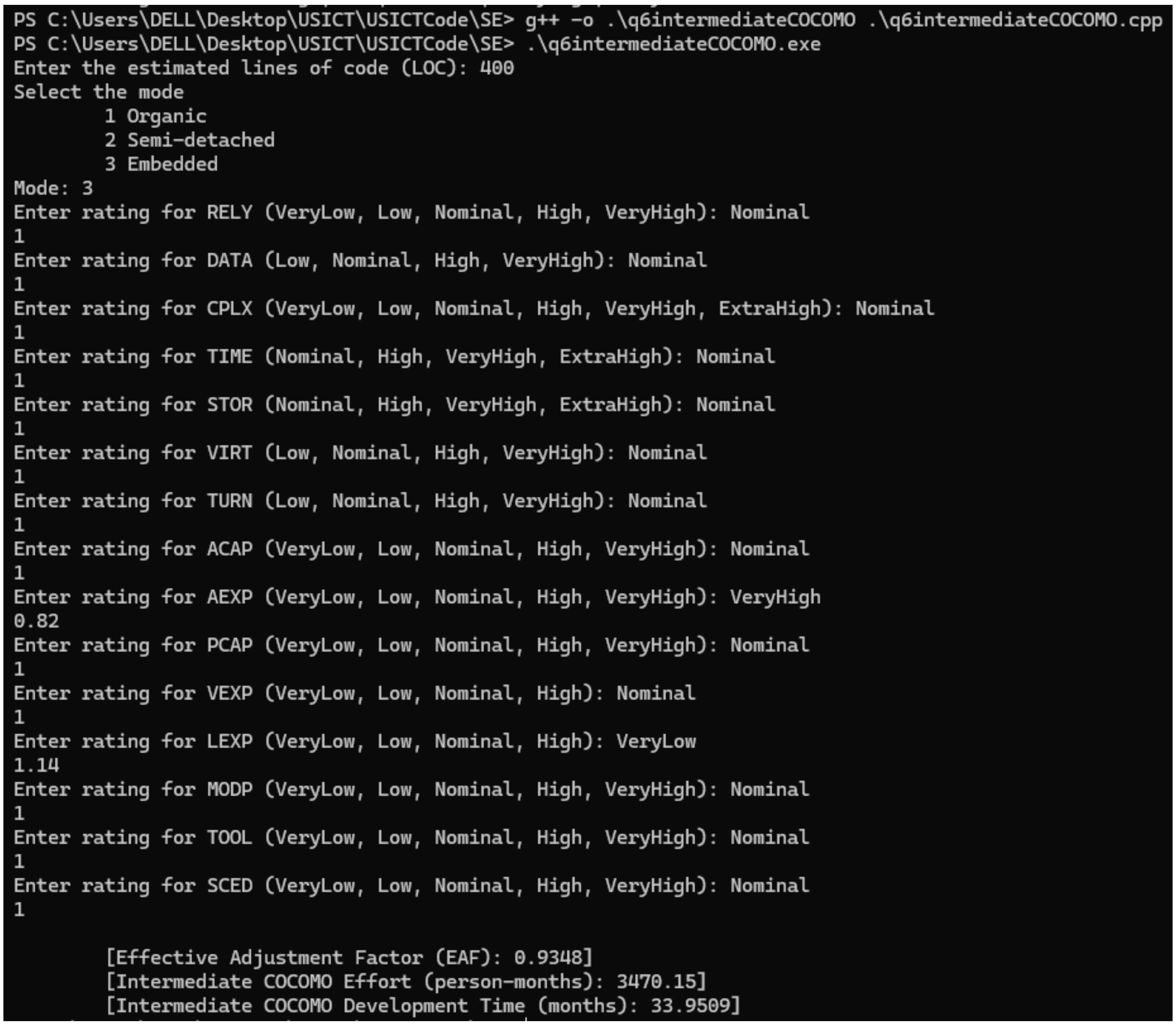
```
#include <iostream>
#include <cmath>
using namespace std;

pair<float, float> basicCOCOMO(int linesOfCode, int mode)
{
    float a, b, c, d;
    // mode => 1: organic, 2: semi-detached, 3: embedded
    if (mode == 1)
    {
        a = 2.4;
        b = 1.05;
        c = 2.5;
        d = 0.38;
    }
    else if (mode == 2)
    {
        a = 3.0;
        b = 1.12;
        c = 2.5;
        d = 0.35;
    }
    else if (mode == 3)
    {
        a = 3.6;
        b = 1.20;
        c = 2.5;
        d = 0.32;
    }
    else
    {
        cout << "Invalid mode!" << endl;
        return {0, 0};
    }
    float effort = a * pow(linesOfCode, b);
    return {effort, c * pow(effort, d)};
}

int main()
{
    int linesOfCode, mode;
    cout << "Enter the estimated lines of code (KLOC): ";
    cin >> linesOfCode;
    cout << "Select the mode\n\t1 Organic\n\t2 Semi-detached\n\t3 Embedded\nMode: ";
    cin >> mode;

    float effort = basicCOCOMO(linesOfCode, mode).first;
    float time = basicCOCOMO(linesOfCode, mode).second;
    cout << "\n\t[Basic COCOMO Effort (person-months): " << effort << "]" << endl;
    cout << "\t[Basic COCOMO Development Time (months): " << time << "]" << endl;
    cout << "\t[Basic COCOMO Average Staff Size (persons): " << effort/time << "]"
<< endl;
    cout << "\t[Basic COCOMO Productivity (KLOC/PM): " << linesOfCode/effort << "]"
<< endl;
    return 0;
}
```



```
PS C:\Users\DELL\Desktop\USICT\USICTCode\SE> g++ -o .\q6basicCOCOMO .\q6basicCOCOMO.cpp
PS C:\Users\DELL\Desktop\USICT\USICTCode\SE> .\q6basicCOCOMO.exe
Enter the estimated lines of code (KLOC): 400
Select the mode
        1 Organic
        2 Semi-detached
        3 Embedded
Mode: 1

        [Basic COCOMO Effort (person-months): 1295.31]
        [Basic COCOMO Development Time (months): 38.0752]
        [Basic COCOMO Average Staff Size (persons): 34.0198]
        [Basic COCOMO Productivity (KLOC/PM): 0.308806]
PS C:\Users\DELL\Desktop\USICT\USICTCode\SE> .\q6basicCOCOMO.exe
Enter the estimated lines of code (KLOC): 400
Select the mode
        1 Organic
        2 Semi-detached
        3 Embedded
Mode: 2

        [Basic COCOMO Effort (person-months): 2462.8]
        [Basic COCOMO Development Time (months): 38.4539]
        [Basic COCOMO Average Staff Size (persons): 64.0455]
        [Basic COCOMO Productivity (KLOC/PM): 0.162417]
PS C:\Users\DELL\Desktop\USICT\USICTCode\SE> .\q6basicCOCOMO.exe
Enter the estimated lines of code (KLOC): 400
Select the mode
        1 Organic
        2 Semi-detached
        3 Embedded
Mode: 3

        [Basic COCOMO Effort (person-months): 4772.81]
        [Basic COCOMO Development Time (months): 37.5965]
        [Basic COCOMO Average Staff Size (persons): 126.948]
        [Basic COCOMO Productivity (KLOC/PM): 0.083808]
```

Intermediate COCOMO Model

```
#include <iostream>
#include <vector>
#include <cmath>
#include <string>
using namespace std;

float getWeight(vector<pair<string, float>> &weights, string rating)
{
    for (const auto &w : weights)
    {
        if (w.first == rating)
        {
            cout << w.second << endl;
            return w.second;
        }
    }
    cout << "Invalid rating!" << endl;
    return 1.0;
}

// Function to calculate EAF based on user input for all cost drivers
float calculateEAF()
{
    vector<pair<string, vector<pair<string, float>>>> costDrivers = {
        {"RELY", {{"VeryLow", 0.75}, {"Low", 0.88}, {"Nominal", 1.00}, {"High", 1.15}, 
{"VeryHigh", 1.40}}},
        {"DATA", {{"Low", 0.94}, {"Nominal", 1.00}, {"High", 1.08}, {"VeryHigh", 1.16}}},
        {"CPLX", {{"VeryLow", 0.70}, {"Low", 0.85}, {"Nominal", 1.00}, {"High", 1.15}, 
{"VeryHigh", 1.30}, {"ExtraHigh", 1.65}}},
        {"TIME", {{"Nominal", 1.00}, {"High", 1.11}, {"VeryHigh", 1.30}, {"ExtraHigh", 1.66}}},
        {"STOR", {{"Nominal", 1.00}, {"High", 1.06}, {"VeryHigh", 1.21}, {"ExtraHigh", 1.56}}},
        {"VIRT", {{"Low", 0.87}, {"Nominal", 1.00}, {"High", 1.15}, {"VeryHigh", 1.30}}},
        {"TURN", {{"Low", 0.87}, {"Nominal", 1.00}, {"High", 1.07}, {"VeryHigh", 1.15}}},
        {"ACAP", {{"VeryLow", 1.46}, {"Low", 1.19}, {"Nominal", 1.00}, {"High", 0.86}, 
{"VeryHigh", 0.71}}},
        {"AEXP", {{"VeryLow", 1.29}, {"Low", 1.13}, {"Nominal", 1.00}, {"High", 0.91}, 
{"VeryHigh", 0.82}}},
        {"PCAP", {{"VeryLow", 1.42}, {"Low", 1.17}, {"Nominal", 1.00}, {"High", 0.86}, 
{"VeryHigh", 0.70}}},
        {"VEXP", {{"VeryLow", 1.21}, {"Low", 1.10}, {"Nominal", 1.00}, {"High", 0.90}}},
        {"LEXP", {{"VeryLow", 1.14}, {"Low", 1.07}, {"Nominal", 1.00}, {"High", 0.95}}},
        {"MODP", {{"VeryLow", 1.24}, {"Low", 1.10}, {"Nominal", 1.00}, {"High", 0.91}, 
{"VeryHigh", 0.82}}},
        {"TOOL", {{"VeryLow", 1.24}, {"Low", 1.10}, {"Nominal", 1.00}, {"High", 0.91}, 
{"VeryHigh", 0.83}}},
        {"SCED", {{"VeryLow", 1.23}, {"Low", 1.08}, {"Nominal", 1.00}, {"High", 1.04}, 
{"VeryHigh", 1.10}}}};

    float eaf = 1.0;
    string rating;

    for (auto &driver : costDrivers)
    {
        cout << "Enter rating for " << driver.first << " (";
        int index = 0;
        int total = driver.second.size();
        for (auto &r : driver.second)
        {
            cout << r.first;
            if (index < total - 1)
            { // Add comma if not the last element
                cout << ", ";
            }
            index++;
        }
        cout << "): ";
        cin >> rating;
        eaf *= getWeight(driver.second, rating);
    }

    return eaf;
}

pair<float, float> intermediateCOCOMO(int linesOfCode, int mode, float eaf)
{
    float a, b, c, d;
    if (mode == 1)
    { // Organic
        a = 3.2;
        b = 1.05;
        c = 2.5;
        d = 0.38;
    }
    else if (mode == 2)
    { // Semi-detached
        a = 3.0;
        b = 1.12;
        c = 2.5;
        d = 0.35;
    }
    else if (mode == 3)
    { // Embedded
        a = 2.8;
        b = 1.20;
        c = 2.5;
        d = 0.32;
    }
    else
    {
        cout << "Invalid mode!" << endl;
        return {0, 0};
    }
    float effort = eaf * a * pow(linesOfCode, b);
    return {effort, c * pow(effort, d)};
}

int main()
{
    int linesOfCode, mode;
    cout << "Enter the estimated lines of code (LOC): ";
    cin >> linesOfCode;
    cout << "Select the mode\n\t1 Organic\n\t2 Semi-detached\n\t3 Embedded\nMode: ";
    cin >> mode;

    float eaf = calculateEAF();
    cout << "\n\t[Effective Adjustment Factor (EAF): " << eaf << "]" << endl;
    float effort = intermediateCOCOMO(linesOfCode, mode, eaf).first;
    float time = intermediateCOCOMO(linesOfCode, mode, eaf).second;
    cout << "\t[Intermediate COCOMO Effort (person-months): " << effort << "]" << endl;
    cout << "\t[Intermediate COCOMO Development Time (months): " << time << "]" << endl;

    return 0;
}
```



```
PS C:\Users\DELL\Desktop\USICT\USICTCode\SE> g++ -o .\q6intermediateCOCOMO .\q6intermediateCOCOMO.cpp
PS C:\Users\DELL\Desktop\USICT\USICTCode\SE> .\q6intermediateCOCOMO.exe
Enter the estimated lines of code (LOC): 400
Select the mode
        1 Organic
        2 Semi-detached
        3 Embedded
Mode: 3
Enter rating for RELY (VeryLow, Low, Nominal, High, VeryHigh): Nominal
1
Enter rating for DATA (Low, Nominal, High, VeryHigh): Nominal
1
Enter rating for CPLX (VeryLow, Low, Nominal, High, VeryHigh, ExtraHigh): Nominal
1
Enter rating for TIME (Nominal, High, VeryHigh, ExtraHigh): Nominal
1
Enter rating for STOR (Nominal, High, VeryHigh, ExtraHigh): Nominal
1
Enter rating for VIRT (Low, Nominal, High, VeryHigh): Nominal
1
Enter rating for TURN (Low, Nominal, High, VeryHigh): Nominal
1
Enter rating for ACAP (VeryLow, Low, Nominal, High, VeryHigh): Nominal
1
Enter rating for AEXP (VeryLow, Low, Nominal, High, VeryHigh): VeryHigh
0.82
Enter rating for PCAP (VeryLow, Low, Nominal, High, VeryHigh): Nominal
1
Enter rating for VEXP (VeryLow, Low, Nominal, High): Nominal
1
Enter rating for LEXP (VeryLow, Low, Nominal, High): VeryLow
1.14
Enter rating for MODP (VeryLow, Low, Nominal, High, VeryHigh): Nominal
1
Enter rating for TOOL (VeryLow, Low, Nominal, High, VeryHigh): Nominal
1
Enter rating for SCED (VeryLow, Low, Nominal, High, VeryHigh): Nominal
1

        [Effective Adjustment Factor (EAF): 0.9348]
        [Intermediate COCOMO Effort (person-months): 3470.15]
        [Intermediate COCOMO Development Time (months): 33.9509]
```

Practical - 4

Aim: To write a program and generate test cases using Functional Testing Techniques (Boundary Value Analysis (BVA), Equivalence Class Partitioning, Decision Table, and Cause-Effect Graphing)

Program: Online Shopping Discount System

A system gives discount based on the amount of an online order:

Order Amount	**Discount**
Below ₹1000	No Discount
₹1000 to ₹5000	10% Discount
Above ₹5000	20% Discount
Negative values	Error

```
#include <iostream>
#include <string>

std::string getDiscountMessage(double amount) {
    if (amount < 0) return "Error: Invalid order amount";
    if (amount < 1000) return "No Discount";
    if (amount <= 5000) return "10% Discount";
    return "20% Discount";
}

int main() {
    double amount;
    std::cout << "Enter order amount: ₹";
    std::cin >> amount;
    std::cout << getDiscountMessage(amount) << std::endl;
    return 0;
}
```

1. Boundary Value Analysis (BVA)

BVA focuses on testing the values at the edges of input ranges, as errors often occur at the boundaries. We test **just below**, **at**, and **just above** the boundaries: ₹1000 and ₹5000.

Test Case ID	Input (₹)	Expected Output
TC_BVA_01	999	No Discount
TC_BVA_02	1000	10% Discount
TC_BVA_03	1001	10% Discount
TC_BVA_04	4999	10% Discount
TC_BVA_05	5000	10% Discount
TC_BVA_06	5001	20% Discount

2. Equivalence Class Partitioning (ECP)

We divide input into valid and invalid partitions. ECP divides input data into partitions of valid and invalid classes. Instead of testing every possible value, one representative from each class is selected.

Partition	Example Input (₹)	Expected Output
Invalid (Negative)	-500	Error
Valid (<1000)	700	No Discount
Valid (1000-5000)	3000	10% Discount
Valid (>5000)	6000	20% Discount

3. Decision Table Testing

This technique uses a table format to represent different combinations of inputs (conditions) and their corresponding actions (outputs). It helps ensure all possible input scenarios are considered, especially when multiple conditions affect the outcome.

We define **conditions** and **actions** based on business rules:

Conditions:

- C1: Amount < 0
- C2: Amount < 1000
- C3: Amount ≤ 5000

Actions:

- A1: Error
- A2: No Discount
- A3: 10% Discount
- A4: 20% Discount

Test Case	C1	C2	C3	Action	Expected Output
TC_DT_01	T	-	-	A1	Error
TC_DT_02	F	T	T	A2	No Discount
TC_DT_03	F	F	T	A3	10% Discount
TC_DT_04	F	F	F	A4	20% Discount

4. Cause-Effect Graphing

Cause-effect graphing visualizes the relationship between inputs (causes) and outputs (effects). It helps identify logical conditions that must be satisfied for certain actions to occur, and is particularly useful for complex systems with multiple inputs.

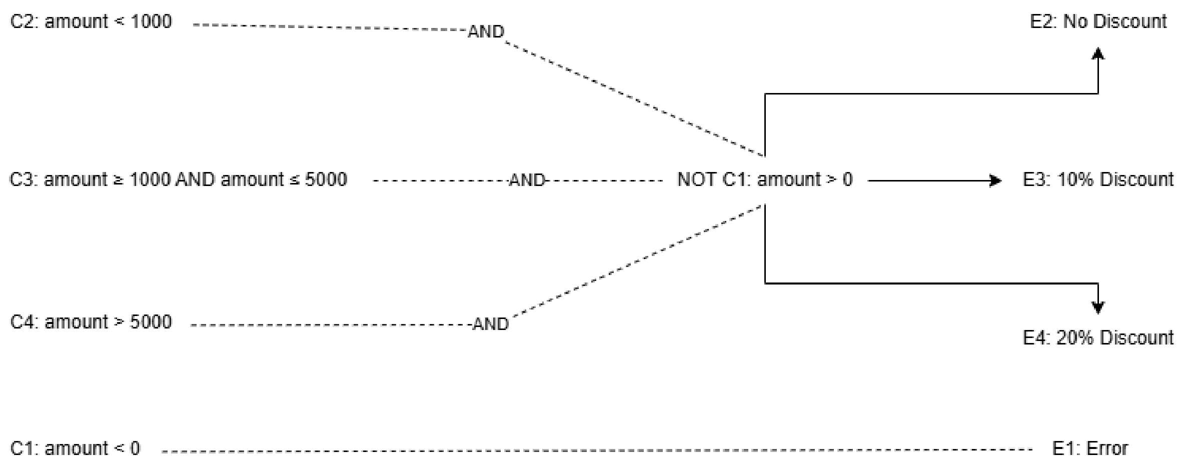
Causes:

- C1: Amount < 0
- C2: 0 ≤ Amount < 1000
- C3: 1000 ≤ Amount ≤ 5000
- C4: Amount > 5000

Effects:

- E1: Error
- E2: No Discount
- E3: 10% Discount
- E4: 20% Discount

Test Case	Cause Activated	Effect
TC_CEG_01	C1	E1 (Error)
TC_CEG_02	C2	E2 (No Discount)
TC_CEG_03	C3	E3 (10% Discount)
TC_CEG_04	C4	E4 (20% Discount)



Practical - 5

Aim: To write a program and generate test cases using Control Flow Testing(statement coverage, branch coverage, and path coverage)
Program: Grading System

```
#include <iostream>
using namespace std;

1. void checkGrade(int marks) {
2.     if (marks >= 90)
3.         cout << "Grade A" << endl;
4.     else if (marks >= 75)
5.         cout << "Grade B" << endl;
6.     else if (marks >= 50)
7.         cout << "Grade C" << endl;
8.     else cout << "Fail" << endl;
9.     cout << "Evaluation Complete" << endl;
10. }
```

Control Flow Graph (CFG)

```
  [1] Start
      |
   [2] if (marks >= 90)?
    / \
  T/   \F
[3]    [4] if (marks >= 75)?
        / \
      T/   \F
    [5]    [6] if (marks >= 50)?
              / \
            T/   \F
          [7]   [8]
                 |
              [9] "Evaluation Complete"
                 |
              [10] End
```

Nodes:

- [1] Start
- [2] if (marks >= 90)
- [3] "Grade A"
- [4] else if (marks >= 75)
- [5] "Grade B"
- [6] else if (marks >= 50)
- [7] "Grade C"
- [8] "Fail"
- [9] "Evaluation Complete"
- [10] End

- **Statement Coverage**

We want every statement (print and conditions) to be executed at least once.

Test Case	Input	Statements Covered
TC1	95	[1,2(T),3,9,10]
TC2	80	[1,2(F),4(T),5,9,10]
TC3	60	[1,2(F),4(F),6(T),7,9,10]
TC4	30	[1,2(F),4(F),6(F),8,9,10]

$$\text{Statement Coverage} = \frac{\text{Number Of Executed Statements}}{\text{Total Statements}} * 100$$

→ 100% Statement Coverage

- **Branch Coverage**

We need both **True and False** outcomes for each decision point.

- if (marks >= 90) → TC1 (True), TC2 (False)
- if (marks >= 75) → TC2 (True), TC3 (False)
- if (marks >= 50) → TC3 (True), TC4 (False)

→ **100% Branch Coverage** using same 4 test cases

- **Path Coverage**

  We list all unique logical paths:

Path	Description	Covered By
P1	1 → 2(T) → 3 → 9 → 10	TC1 (95)
P2	1 → 2(F) → 4(T) → 5 → 9 → 10	TC2 (80)
P3	1 → 2(F) → 4(F) → 6(T) → 7 → 9 → 10	TC3 (60)
P4	1 → 2(F) → 4(F) → 6(F) → 8 → 9 → 10	TC4 (30)

All 4 paths tested → 100% Path Coverage

Summary:

Metric	Result
Statement Coverage	100%
Branch Coverage	100%
Path Coverage	100%
Unreachable Code	None
Missing Test Cases	None

Practical - 6

Aim: To write a program and Identify DU (Define-Use) paths in the code.

Program: Sum of two numbers

```
#include <iostream>
using namespace std;

1. void compute(int a, int b) {
2.     int sum = a + b;           // D: sum, U: a, b (C-Use)
3.     int result = 0;            // D: result
4.     if (sum > 10) {            // U: sum (P-Use)
5.         result = sum * 2;      // U: sum (C-Use), D: result
6.     } else {
7.         result = sum + 5;      // U: sum (C-Use), D: result
       }
8.     cout << "Result: " << result << endl;  // U: result (C-Use)
   }
```

Variables: a, b, sum, result

Key Terms:
1. Definition (D): A variable is assigned a value

2. Use (U): A variable's value is used in computations or conditions.
C-use: Computational Use

- The variable is used in a **computation or expression**, not in a condition.
- Often occurs in **assignments, arithmetic operations, function calls**, etc.

P-use: Predicate Use

- The variable is used in a **decision or condition**, like in `if`, `while`, `for`, `switch`.
- It **affects the control flow** of the program.

3. DU Chain: A sequence of execution where a defined variable is used before being redefined or going out of scope.

DU Chains

Variable	Definition Line	Use Type	Use Line	Path Type
`a`	param	C-Use	Line 2	Simple DU Path
`b`	param	C-Use	Line 2	Simple DU Path
`sum`	Line 2	P-Use	Line 4	Extended DU Path
`sum`	Line 2	C-Use	Line 5	Extended DU Path
`sum`	Line 2	C-Use	Line 7	Extended DU Path
`result`	Line 3	C-Use	Line 8	Extended DU Path
`result`	Line 5	C-Use	Line 8	Simple DU Path
`result`	Line 7	C-Use	Line 8	Simple DU Path

→ **Simple DU Path (Direct Definition to Use)**
A variable is defined and later used without any redefinition in between.

→ **Extended DU Path (Definition to Use Across Multiple Statements)**
A variable is defined in one block and used in another without redefinition.

→ **Loop DU Path (Definition Used in a Loop)**
A variable is defined outside or within a loop and used repeatedly.

Practical - 7

Aim: Write a program and introduce mutations into it and check if test cases detect them.

Program: Multiply and Compare

```
#include <iostream>
using namespace std;

bool isProductGreater(int a, int b, int threshold) {
    int product = a * b;
    return product > threshold;
}
```

(i) Test Cases for Original Code

Test Case	Input (a,b, threshold)	Expected Output
TC1	2,3,5	true
TC2	2,2,5	false
TC3	1,5,5	false
TC4	3,3,8	true

(ii) Introduce Mutants

- **Mutant 1 (Change * to +)**

```
int product = a + b;  // M1
```

Test Case TC1:

- **Original:** 2 * 3 = 6 → 6 > 5 → true
- **Mutant:** 2 + 3 = 5 → 5 > 5 → false

  Detected → Killed

- **Mutant 2 (Change > to >=)**

```
return product >= threshold;  // M2
```

Test Case TC3:

- **Original:** 1 * 5 = 5 → 5 > 5 → false
- **Mutant:** 5 >= 5 → true

  Detected → Killed

- **Mutant 3 (Remove return line)**

```
return true;  // M3 (removal of logic)
```

Any test case expecting false (e.g., TC2 or TC3) will fail.

Detected → Killed

(iii) Mutation Analysis

Mutant	Description	Killed By	Status
M1	* → +	TC1	Killed
M2	> → >=	TC3	Killed
M3	Logic removed	TC2	Killed

(iv)Mutation Score

$$\text{Mutation Score} = \frac{\text{Killed Mutants}}{\text{Total Mutants}} \times 100 = (3 / 3) \times 100 = 100\%$$

→ High mutation score indicates that the test suite is strong and effectively detects logical changes.

Practical - 8

Aim: Write a program with its test cases and prioritise different test cases for Regression Testing.

Regression Testing

It is a type of software testing that ensures new changes or bug fixes in the code have not introduced new errors or broken existing functionality. It involves re-running previously passed test cases to verify that everything still works correctly after updates.
This testing is especially important after:

- Adding new features
- Fixing bugs
- Refactoring or optimizing code

By carefully prioritizing test cases, regression testing helps save time while maintaining high software quality.

Program: User Login System

```
#include <iostream>
#include <string>
using namespace std;

bool login(string username, string password) {
    if (username == "admin" && password == "12345") {
        return true;
    }
    return false;
}
```

Update:

A developer adds a new password policy check (min 5 characters):

```
if (password.length() < 5) {
    return false;
}
```

Now the updated function is:

```
bool login(string username, string password) {
    if (password.length() < 5) {
        return false;
    }
    if (username == "admin" && password == "12345") {
        return true;
    }
    return false;
}
```

Test Cases (Before & After Change)

Test Case ID	Input	Expected Output	Affected	Description
TC1	("admin", "12345")	true	Yes	Valid Login
TC2	("admin", "wrongpass")	false	Yes	Invalid password
TC3	("user", "12345")	false	Yes	Invalid username
TC4	("admin", "123")	false	Yes	Password too short
TC5	("", "")	false	Yes	Empty credentials
TC6	("admin", " ")	false	Yes	Password with spaces
TC7	("ADMIN", "12345")	false	No	Case sensitivity test

Prioritize Test Cases for Regression Testing

We'll prioritize using the Risk + Impact approach:

Priority	Test Case ID(s)	Reason for Priority
High	TC1, TC4, TC5	Tests core logic + new feature + potential failure case
Medium	TC2, TC3	Tests variations on valid credentials
Low	TC6, TC7	Rare edge cases, lower impact

Regression Testing Strategy

We'll use Selective + Prioritized Regression Testing:

- Retest high-priority cases first: TC1, TC4, TC5
- Then cover medium-priority ones: TC2, TC3
- Optionally retest low-priority cases: TC6, TC7

This minimizes effort while maximizing bug detection around the new feature.

Conclusion

- The test suite was created to cover all important functional and edge cases.
- Test cases were prioritized to focus on those most likely to detect defects after code changes.
- This approach ensures efficient and focused regression testing without rerunning the entire suite unnecessarily.

Practical - 9

Aim: Write a program and demonstrate Static and Dynamic Slice-based Testing.

Program Slicing

Program slicing is the process of extracting parts of the program that affect or are affected by a particular variable at a certain point.

Program: Calculate Grade

```
#include <iostream>
using namespace std;
char calculateGrade(int marks) {
    string status = "Fail";
    char grade;
    if (marks >= 90) {
        grade = 'A';
        status = "Pass";
    } else if (marks >= 75) {
        grade = 'B';
        status = "Pass";
    } else if (marks >= 50) {
        grade = 'C';
        status = "Pass";
    } else {
        grade = 'F';
    }
    cout << "Status: " << status << endl;
    return grade;
}
```

We'll perform slicing based on target: grade (what affects the grade returned)

(i) Static Program Slice for grade
Based on the program's source code without running it. It analyzes the entire program code to find all statements that might affect the value of a variable, regardless of input values or execution paths.
Static Slice (Statements affecting grade):

```
char grade;
if (marks >= 90) {
    grade = 'A';
} else if (marks >= 75) {
    grade = 'B';
} else if (marks >= 50) {
    grade = 'C';
} else {
    grade = 'F';
}
return grade;
```

Test Cases for Static Slice

These test cases aim to **exercise all paths** that can affect grades.

Test Case	Input (marks)	Expected Grade	Covered Branch
TC1	92	A	`marks >= 90`
TC2	80	B	`marks >= 75`
TC3	65	C	`marks >= 50`
TC4	40	F	`else`

These cases **fully test** the static slice for grade.

(ii) Dynamic Slicing

It analyzes a specific execution of the program, using specific input values, and includes only the statements that were actually executed and affected the variable.

Input: marks = 92
Executed Statements:

- **char grade;**
- **if (marks >= 90) → true**
- **grade = 'A';**
- **return grade;**

Dynamic Slice for Input 92:

```
char grade;
if (marks >= 90) {
    grade = 'A';
}
return grade;
```

Input: marks = 65
Executed Statements:

- **char grade;**
- **if (marks >= 90) → false**
- **else if (marks >= 75) → false**
- **else if (marks >= 50) → true**
- **grade = 'C';**
- **return grade;**

Dynamic Slice for Input 65:

```
char grade;
else if (marks >= 50) {
    grade = 'C';
}
return grade;
```

Summary Table

Type	Includes All Possibilities?	Input-Dependent	Slice Size
Static Slice	Yes	No	Large
Dynamic Slice	No (only actual path)	Yes	Small

Conclusion

- **Static slicing** helps understand all logic affecting a variable — great for maintenance and completeness.
- **Dynamic slicing** is ideal for debugging and analyzing behavior for specific inputs.

Practical - 10

Aim: Write a program and perform Agile Testing by creating User Stories & Scenarios

Agile testing is an iterative, collaborative approach to software quality assurance that aligns closely with Agile development principles—emphasizing frequent feedback, continuous integration, and rapid adaptation to change. Instead of waiting until the end of a development cycle, Agile testers work alongside developers from the start, writing and executing small, automated test suites against each new feature as it's delivered. This enables teams to catch defects early, validate requirements through acceptance criteria and user stories, and continuously refine both the product and the test suite. By embracing practices such as test-driven development (TDD), behavior-driven development (BDD), and continuous regression testing, Agile testing ensures that quality is built in at every sprint, reduces the risk of large defect backlogs, and fosters a shared responsibility for quality across the entire team.

Program: Simple Banking App – Balance Inquiry Feature

User Story 1: As a user, I want to **check my account balance**, so that I can know how much money I have available.

1.1 Acceptance Criteria

The feature is considered **done** when:

- User can enter a valid userId and receive the correct balance.
- If the userId is invalid, the system should return -1 or an error message.
- The function should execute within 1 second (performance check).
- Code is tested for both valid and invalid user IDs.
- Covered with unit and exploratory testing.

1.2 Test Scenarios

Scenario ID	Description	Input	Expected Output
TS01	Valid user ID → existing account	`"user123"`	`1500.75`
TS02	Valid user ID → another account	`"user456"`	`250.50`
TS03	Invalid user ID	`"user789"`	`-1`
TS04	Empty string as input	`""`	`-1`
TS05	Very large input	`"user999999999"`	`-1`

User Story 2: As a user, I want to **deposit money into my account**, so that I can increase my available balance.

2.1 Acceptance Criteria

The feature is considered **done** when:

- Valid userId and positive amount increase balance.
- Invalid userId or negative/zero amount returns false.
- Operation is completed within 1 second.
- Tested with multiple userIds and edge values (e.g., 0, very large amounts).

2.2 Test Scenarios

Scenario ID	Description	Input	Expected Output
TS06	Valid deposit to existing account	"user123", 500.00	true, new balance: 2000.75
TS07	Deposit to non-existing account	"user456", 100.00	false
TS08	Deposit zero amount	"user123", 0.00	false
TS09	Deposit negative amount	"user123", -50.00	false
TS10	Very large deposit amount	"user123", 1e9	true, balance updated accordingly

Code Implementation (C++)

```
#include <iostream>
#include <unordered_map>
using namespace std;

class Bank {
private:
    unordered_map<string, double> accounts;

public:
    Bank() {
        // Pre-filled dummy accounts
        accounts["user123"] = 1500.75;
        accounts["user456"] = 250.50;
    }

    double getBalance(string userId) {
        if (accounts.find(userId) != accounts.end()) {
            return accounts[userId];
        } else {
            return -1;  // user not found
        }
    }
    bool deposit(string userId, double amount) {
       if (accounts.find(userId) != accounts.end() && amount > 0) {
           accounts[userId] += amount;
           return true;
       }
       return false;
    }
};
```

Agile Testing Process

Day 1: User Story & Acceptance Criteria Defined
Product owner and tester define:

- Function: getBalance() & deposit()
- Criteria: handles valid & invalid inputs
- Collaboratively define test coverage areas.
- Identify edge cases (e.g., zero, negative, large amounts).

Day 2-4: Developer Writes Code, Tester Writes Unit Tests
Example unit test:

```
Bank bank;
assert(bank.getBalance("user123") == 1500.75);
assert(bank.getBalance("user456") == 250.50);
assert(bank.getBalance("user789") == -1);

assert(bank.deposit("user123", 100.0) == true);
assert(bank.getBalance("user123") == 1600.75);
assert(bank.deposit("user789", 100.0) == false); //non-existent
assert(bank.deposit("user123", 0.0) == false);
assert(bank.deposit("user123", -50.0) == false);
```

Day 5: Run Automated Tests

- Unit tests pass
- Performance test: Response < 1 second
- Test behavior on multiple deposits in succession.
- Try depositing boundary values (e.g., `DBL_MAX`, `0.01`).
- Simulate API or UI interaction (if integrated).

Day 6-7: Exploratory Testing

Tester tries:

- Special characters as user ID
- Empty or null strings
- Very long string input

Bug found: No validation for extremely long strings → noted for future enhancement.

Day 8: Review, Feedback & Regression

- Peer code and test review.
- Tester validates coverage against Acceptance Criteria.
- Regression tests all previously working features.

Day 9: Edge Case & UI Testing

Testers simulate mobile UI inputs (if applicable via frontend)

Day 10: Sprint Demo & Review

- Working feature demonstrated
- Stakeholders approve
- Story marked "Done"

Summary

Agile Concept	Applied To
User Story 1	"As a user, I want to check my balance, so that I can know how much money I have available."
User Story 2	"As a user, I want to deposit money into my account, so that I can increase my available balance."
Acceptance Criteria	Valid/invalid user ID handling, positive/negative input validation, response under 1 sec, balance updated correctly
TDD (Test-Driven Development)	Unit tests written in sync with development: getBalance() and deposit() tested thoroughly
Test Coverage	Valid & invalid inputs, empty string, large input, zero & negative deposits, large amounts
Exploratory Testing	Special characters, null/empty strings, very long userId — revealed missing validation
Performance Testing	Verified all operations execute under 1 second using simple test timing methods
Regression Testing	getBalance and deposit features retested after each enhancement or bug fix
Continuous Integration	Tests are run iteratively at each step of development and feedback is incorporated immediately
Sprint Review	Final demo to stakeholders on Day 10 — all test scenarios passed, accepted as “Done”

Practical - 11

Aim: To Conduct a Survey of following Testing Tools:

i. JUnit ii. Selenium iii. JMeter iv. LoadRunner
v. WinRunner vi. Silk Test vii. Cactus

Tool	Type	Purpose / Use Case	Key Features	Usage in Agile or Structural Testing
JUnit	Unit Testing (Java)	Unit testing of Java methods and classes	Lightweight, annotations, assertions, integrated in IDEs	Used in Agile TDD for writing fast-running unit tests
Selenium	Functional / UI Testing	Automate web browsers for end-to-end testing	WebDriver API, cross-browser testing, supports multiple languages	Used in Agile regression, continuous testing, CI
JMeter	Performance / Load Testing	Load and performance testing for web & server applications	Simulates multiple users, HTTP/SOAP/REST, graphs, CLI-supported	Structural non-functional testing; fits Agile CI/CD
LoadRunner	Performance / Stress Testing	Measures performance under load for enterprise apps	Protocol-based testing, extensive metrics, advanced analytics	Suited for large-scale structural performance testing
WinRunner	Functional Testing (Retired)	GUI testing of Windows applications (by HP)	Record-and-playback, integrated test scripts, defect logging	Now deprecated; replaced by UFT (Unified Functional Testing)
Silk Test	Functional and Regression Testing	Automated testing of applications on desktop/web	Supports Java, .NET, cross-browser tests, stable scripting	Often used in Agile for UI automation in enterprise apps
Cactus	In-Container Testing (Java EE)	Unit testing of server-side Java components	Focused on servlet, EJB, tag libraries, runs in-container	Agile-compatible for backend/service-side unit testing

Highlights Per Tool

1. **JUnit**
    - **Type:** Unit Testing
    - **Use:** TDD in Agile (write test first, then code)
    - **Example:** Structural Testing – Statement/Branch Coverage

2. **Selenium**
    - **Type**: Web Automation
    - **Use:** Regression testing in Agile sprints
    - **Example:** Automating login page, cart flow

3. **JMeter**
    - **Type:** Load/Stress Testing
    - **Use:** Performance analysis before release
    - **Example:** Test user concurrency on a web app

4. **LoadRunner**
    - **Type:** Enterprise Performance Testing
    - **Use:** Suitable for banking, telecom systems
    - **Example:** 5000-user simulation on transaction system

5. **WinRunner** ***(Legacy)***
    - Discontinued (Replaced by UFT)
    - Useful historically for GUI regression testing

6. **Silk Test**
    - **Type**: Functional/Regression
    - **Use:** Automates complex test scenarios across platforms
    - Useful for Agile regression automation

7. **Cactus**
    - **Type:** Unit Testing Java EE (In-container)
    - **Use:** Validates server-side logic (EJBs, servlets)
    - Agile + Structural testing combo for backend apps

Usage Matrix

Tool	Unit Testing	Functional Testing	Performance Testing	Structural Testing	Agile Friendly
JUnit	Yes	No	No	Yes	Yes
Selenium	No	Yes	No	No	Yes
JMeter	No	No	Yes	Yes	Yes
LoadRunner	No	No	Yes	Yes	Partial
WinRunner	No	Yes	No	Yes	Obsolete
Silk Test	No	Yes	Limited	Partial	Yes
Cactus	Yes	No	No	Yes	Yes

Conclusion:

This survey highlights that each tool serves a specific purpose—ranging from unit testing (JUnit, Cactus), functional UI testing (Selenium, Silk Test), to performance testing (JMeter, LoadRunner). Choosing the right tool depends on the application domain, test requirements, scalability needs, and budget constraints.