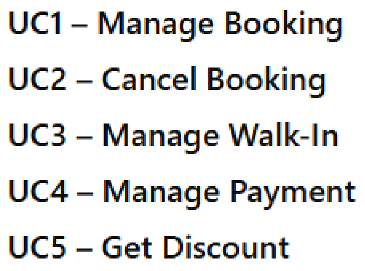
ANSWER KEYS FINAL[MCQS]

Question 1:

1.	A
2.	A
3.	A
4.	A
5.	A
6.	A
7.	A
8.	A
9.	A
10.	A
11.	A
12.	A
13.	A
14.	A
15.	A
16.	A
17.	A
18.	A
19.	A
20.	A
21.	B
22.	B
23.	B
24.	B
25.	B
26.	B
27.	B
28.	B
29.	B
30.	B
31.	B
32.	B
33.	B
34.	B
35.	B
36.	C
37.	C
38.	C
39.	C
40.	C
41.	C
42.	C
43.	C

44.	C
45.	C
46.	C
47.	C
48.	C
49.	C
50.	C
51.	D
52.	D
53.	D
54.	D
55.	D
56.	D
57.	D
58.	D
59.	D
60.	D

QUESTION 2



UC1 – Manage Booking

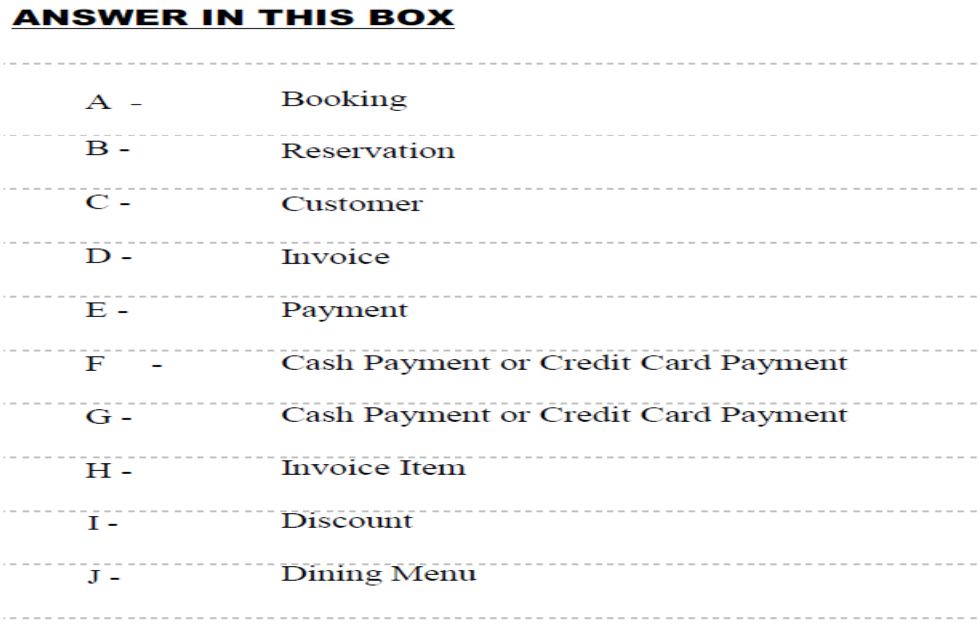
UC2 – Cancel Booking

UC3 – Manage Walk-In

UC4 – Manage Payment

UC5 – Get Discount

.



ANSWER IN THIS BOX

A -	Booking
B -	Reservation
C -	Customer
D -	Invoice
E -	Payment
F -	Cash Payment or Credit Card Payment
G -	Cash Payment or Credit Card Payment
H -	Invoice Item
I -	Discount
J -	Dining Menu

QUESTION 3:

Solution:

```
User                    EmployeeForm(UI)                EmployeeController

         |                            |                                  |
         | --- clicks Find ---------->                                   |
         |                            |--- getEmployee(empNo) --------->|
         |                            |                                  |
         |                            |<-- Employee / Not Found --------|
         |<------ shows record ------|                                   |
         |                            |                                  |
         |                            |                                  |
         | --- clicks Add ----------->                                   |
         |                            |--- addEmployee(data) ---------->|
         |                            |                                  |
         |                            |<------ success/failure ---------|
         |<------ shows message -----|                                 |
         |                           |                                 |
         |                           |                                 |
         | --- clicks Update -------->                                 |
         |                           |--- updateEmployee(data) ------->|
         |                           |                                 |
         |                           |<------ success/failure ---------|
         |<------ shows message -----|                                 |
         |                           |                                 |
```

✅ 1. DBConnection.java

```
import java.sql.*;

public class DBConnection {

    private static final String URL = "jdbc:mysql://localhost:3306/testdb";
    private static final String USER = "root";
    private static final String PASS = "yourpassword";

    public static Connection getConnection() throws SQLException {
        return DriverManager.getConnection(URL, USER, PASS);
    }
}
```

✅ 2. Employee.java (Model Data Class)

```
public class Employee {
    private int empno;
    private String empname;
    private String job;
    private double salary;
    private int deptno;

    // Constructor
    public Employee(int empno, String empname, String job, double salary, int deptno) {
        this.empno = empno;
        this.empname = empname;
        this.job = job;
        this.salary = salary;
        this.deptno = deptno;
    }

    // Getters
    public int getEmpno() { return empno; }
    public String getEmpname() { return empname; }
    public String getJob() { return job; }
    public double getSalary() { return salary; }
    public int getDeptno() { return deptno; }
}
```

✅ 3. EmployeeDAO.java (Model – Database Access Layer)

```
import java.sql.*;

public class EmployeeDAO {

    // FIND EMPLOYEE
    public Employee findEmployee(int empno) throws SQLException {
        String sql = "SELECT * FROM employees WHERE empno = ?";
        Connection con = DBConnection.getConnection();
        PreparedStatement ps = con.prepareStatement(sql);
        ps.setInt(1, empno);
        ResultSet rs = ps.executeQuery();

        if (rs.next()) {
            return new Employee(
                rs.getInt("empno"),
                rs.getString("empname"),
                rs.getString("job"),
                rs.getDouble("salary"),
                rs.getInt("deptno")
            );
        }
        return null;
    }

    // ADD EMPLOYEE
    public boolean addEmployee(Employee emp) throws SQLException {
        String sql = "INSERT INTO employees VALUES (?, ?, ?, ?, ?)";
        Connection con = DBConnection.getConnection();
        PreparedStatement ps = con.prepareStatement(sql);
        ps.setInt(1, emp.getEmpno());
        ps.setString(2, emp.getEmpname());
        ps.setString(3, emp.getJob());
        ps.setDouble(4, emp.getSalary());
        ps.setInt(5, emp.getDeptno());
        return ps.executeUpdate() > 0;
    }

    // UPDATE EMPLOYEE
    public boolean updateEmployee(Employee emp) throws SQLException {

        String sql = "UPDATE employees SET empname=?, job=?, salary=?, 
deptno=? WHERE empno=?";

        Connection con = DBConnection.getConnection();

        PreparedStatement ps = con.prepareStatement(sql);

        ps.setString(1, emp.getEmpname());

        ps.setString(2, emp.getJob());

        ps.setDouble(3, emp.getSalary());

        ps.setInt(4, emp.getDeptno());

        ps.setInt(5, emp.getEmpno());

        return ps.executeUpdate() > 0;

    }

}
```

✓ 4. EmployeeController.java (Controller Class)

```
import javax.swing.*;


public class EmployeeController {


    private EmployeeDAO dao;


    public EmployeeController() {

        dao = new EmployeeDAO();

    }


    // FIND

    public Employee findEmployee(int empno) {

        try {

            return dao.findEmployee(empno);
        } catch (Exception e) {
            JOptionPane.showMessageDialog(null, "Error finding employee: " + e.getMessage());
            return null;
        }
    }

    // ADD
    public void addEmployee(Employee emp) {
        try {
            if (dao.addEmployee(emp)) {
                JOptionPane.showMessageDialog(null, "Employee Added Successfully!");
            } else {
                JOptionPane.showMessageDialog(null, "Failed to Add Employee.");
            }
        } catch (Exception e) {
            JOptionPane.showMessageDialog(null, "Error: " + e.getMessage());
        }
    }

    // UPDATE
    public void updateEmployee(Employee emp) {
        try {
            if (dao.updateEmployee(emp)) {
                JOptionPane.showMessageDialog(null, "Employee Updated Successfully!");
            } else {
                JOptionPane.showMessageDialog(null, "Update Failed.");
            }
        } catch (Exception e) {
            JOptionPane.showMessageDialog(null, "Error: " + e.getMessage());
        }
    }
}
```

✓ 5. EmployeeForm.java (View – GUI Form)

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class EmployeeForm extends JFrame {

    private JTextField txtEmpno, txtName, txtJob, txtSalary, txtDeptno;
    private JButton btnFind, btnAdd, btnUpdate;

    private EmployeeController controller;

    public EmployeeForm() {
        controller = new EmployeeController();

        setTitle("Employee Form");
        setSize(400, 300);
        setLayout(new GridLayout(7, 2));
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Initialize fields
        txtEmpno = new JTextField();
        txtName = new JTextField();
        txtJob = new JTextField();
        txtSalary = new JTextField();
        txtDeptno = new JTextField();
        btnFind = new JButton("Find");
        btnAdd = new JButton("Add");
        btnUpdate = new JButton("Update");

        // Add UI components
        add(new JLabel("Employee No:"));
        add(txtEmpno);

        add(new JLabel("Employee Name:"));
        add(txtName);

        add(new JLabel("Job:"));
        add(txtJob);

        add(new JLabel("Salary:"));
        add(txtSalary);

        add(new JLabel("Dept No:"));
        add(txtDeptno);

        add(btnFind);
        add(btnAdd);
        add(btnUpdate);

        // Button handlers
        btnFind.addActionListener(e -> findEmployee());
        btnAdd.addActionListener(e -> addEmployee());
        btnUpdate.addActionListener(e -> updateEmployee());

        setVisible(true);
    }
```

```java
// ================= BUTTON EVENTS =================

private void findEmployee() {
    int empno = Integer.parseInt(txtEmpno.getText());
    Employee emp = controller.findEmployee(empno);

    if (emp != null) {
        txtName.setText(emp.getEmpname());
        txtJob.setText(emp.getJob());
        txtSalary.setText(String.valueOf(emp.getSalary()));
        txtDeptno.setText(String.valueOf(emp.getDeptno()));

        JOptionPane.showMessageDialog(this, "Record found!");
    } else {
        JOptionPane.showMessageDialog(this, "No record found!");
    }
}

private void addEmployee() {
    Employee emp = new Employee(
        Integer.parseInt(txtEmpno.getText()),
        txtName.getText(),
        txtJob.getText(),
        Double.parseDouble(txtSalary.getText()),
        Integer.parseInt(txtDeptno.getText())
    );

    controller.addEmployee(emp);
}

private void updateEmployee() {
    Employee emp = new Employee(
            Integer.parseInt(txtEmpno.getText()),
            txtName.getText(),
            txtJob.getText(),
            Double.parseDouble(txtSalary.getText()),
            Integer.parseInt(txtDeptno.getText())
        );

        controller.updateEmployee(emp);
    }

    public static void main(String[] args) {
        new EmployeeForm();
    }
}
```

QUESTION 4:

Here are the most suitable design patterns for each scenario:

1. **Facade Pattern** – To provide a simplified interface to complex subsystems like inventory, payment, and shipping.
2. **Adapter Pattern** – To make the new database system compatible with the legacy interface without changing existing client code.
3. **Facade Pattern** – To simplify the view of a large subsystem (25 classes) by exposing only the essential methods to clients.
4. **Adapter Pattern** – To wrap Franz's existing classes (`FranzCPU` and `FranzHardDisk`) so they can fit into your own `Part` hierarchy.
5. **Factory Pattern** – To create the appropriate payment processor object at runtime based on user selection without changing client code.

QUESTION 5:

```
 public class Logger {
private static Logger instance;
// Other logger attributes and methods

private  Logger() {
// Initialize logger
this.logFile = "app.log";
}

public static synchronized Logger getInstance() { if (instance == null) {
instance = new Logger();
}
return instance;
}

public void log(String message) {
// Log the message
}

// Other logging-related methods
```

```
public class Main {

    public static void main(String[] args) {

        Logger logger = Logger.getInstance();

        logger.info("Application started");

        logger.debug("Debugging information");

        logger.error("An error occurred");

    }

}
```