OOP Lab 8: Applying the SOLID Principles (Practice Lab)

Object-Oriented Programming Laboratory

1. Goals

This lab continues Lab 7. After this lab, you should be able to:

- Recognize code that violates one or more SOLID principles.
- Explain *which* principle is violated and *why*.
- Refactor a small design into one that better follows SOLID.
- Apply SOLID in a slightly bigger, more realistic example (mini project).

2. General Instructions

- You should **reuse** your understanding from Lab 7 (SRP, OCP, LSP, ISP, DIP).
- For each exercise:
  a) Read the given code and answer the conceptual questions.
  b) Refactor the code into a better design (create new classes / interfaces when needed).
  c) Write **short explanations** (2–4 sentences) for how your new design follows SOLID.
- Put each exercise in a separate package: `srp2`, `ocp2`, `lsp2`, `isp2`, `dip2`, `solid_project`.

3. Exercise 1 – SRP in a "God" Class (`srp2`)

Consider the following class from a (very simple) e-learning system:

```
package srp2;

import java.util.List;

public class CourseReportManager {

   public void generateReport(String courseId) {
      // 1. load data from "database"
      System.out.println("Connecting to database...");
      System.out.println("Loading students and grades for course " + courseId + "...");

      // imagine we loaded this:
      List<String> students = List.of("An", "Binh", "Chi");
      List<Integer> grades = List.of(9, 7, 8);
```

```
        // 2. calculate statistics
        double total = 0;
        for (int g : grades) {
            total += g;
        }
        double avg = total / grades.size();

        // 3. format report
        StringBuilder report = new StringBuilder();
        report.append("Course ID: ").append(courseId).append("\n");
        report.append("Students and grades:\n");
        for (int i = 0; i < students.size(); i++) {
            report.append("- ").append(students.get(i))
                  .append(": ").append(grades.get(i)).append("\n");
        }
        report.append("Average grade: ").append(avg).append("\n");

        // 4. save to file
        System.out.println("Saving report.txt ...");

        // 5. send notification email to teacher
        System.out.println("Sending email to teacher with report attached...");
        System.out.println(report);
    }
}
```

Tasks

a) In comments inside the code, clearly mark the different responsibilities (data access, calculation, formatting, persistence, notification).

b) Explain in 3–4 sentences why this class violates the Single Responsibility Principle (SRP).

c) Propose a new design and implement it. At minimum, you should have classes such as:

- `CourseDataLoader` (load data from "database").
- `GradeCalculator` (calculate average / statistics).
- `ReportFormatter` (build the text report).
- `ReportStorage` (save report).
- `TeacherNotifier` (send notification).

d) Refactor `CourseReportManager` so that it only **coordinates** these classes.

e) Write a short `main` method to show how to use your new design.

f) In 3–4 sentences, explain how SRP is now respected.

4. Exercise 2 – OCP in a Discount System (`ocp2`)

We have a simple discount calculator:

```
package ocp2;

public class DiscountService {

   public double calculateDiscount(String userType, double totalPrice) {
      if (userType.equals("STUDENT")) {
         return totalPrice * 0.15;
      } else if (userType.equals("TEACHER")) {
         return totalPrice * 0.20;
      } else if (userType.equals("VIP")) {
         return totalPrice * 0.25;
      } else {
         return 0;
      }
   }
}
```

Tasks

a) Explain in 2–3 sentences why this design breaks the Open/Closed Principle (OCP).

b) Introduce an interface:

```
public interface DiscountPolicy {
   double discount(double totalPrice);
}
```

c) Create concrete policies:

- `StudentDiscount`, `TeacherDiscount`, `VipDiscount`.

d) Refactor `DiscountService` so that:

- It **does not** use `if-else` on string constants.
- It can receive a `DiscountPolicy` (or a `Map<String, DiscountPolicy>`) from outside.

e) Add a new user type `"ALUMNI"` with its own discount, **without** modifying the logic inside `DiscountService`.

f) Explain in 3–4 sentences how your refactoring respects OCP.

5. Exercise 3 – LSP in a Banking Example (`lsp2`)

We have a simple hierarchy for bank accounts:

```
package lsp2;

public class BankAccount {

   protected double balance;

   public void deposit(double amount) {
```

```
        if (amount <= 0) throw new IllegalArgumentException();
        balance += amount;
    }

    public void withdraw(double amount) {
        if (amount <= 0) throw new IllegalArgumentException();
        if (amount > balance) throw new IllegalArgumentException("Not enough money");
        balance -= amount;
    }

    public double getBalance() {
        return balance;
    }
}

public class FixedTermAccount extends BankAccount {

    @Override
    public void withdraw(double amount) {
        throw new UnsupportedOperationException("Cannot withdraw from fixed term account unti
    }
}
```

Tasks

a) Describe in 3–4 sentences why `FixedTermAccount` violates the Liskov Substitution Principle (LSP).
(Hint: client code that uses `BankAccount` and calls `withdraw()` might break.)

b) Design a better hierarchy. For example, you may:

- Introduce an `Account` interface with behaviors that all accounts share.
- Separate "withdrawable" and "non-withdrawable" accounts using different abstractions.

c) Implement your improved design in `lsp2` package.

d) Write a small `main` method that:

- Stores multiple accounts in a list.
- Calls common methods in a way that respects LSP (no unexpected exceptions).

e) Add 3–4 sentences explaining how your new design follows LSP.

6. Exercise 4 – ISP in a User Interface Module (`isp2`)

We have an interface for a "screen" in an app:

```
package isp2;

public interface Screen {

    void showText(String text);
```

```
    void showImage(String path);

    void playVideo(String path);

    void handleUserInput(String input);
}
```

Now we have a very simple `LoginScreen`:

```
public class LoginScreen implements Screen {

    @Override
    public void showText(String text) {
        System.out.println("LOGIN: " + text);
    }

    @Override
    public void showImage(String path) {
        // not needed
        throw new UnsupportedOperationException("Login does not use images");
    }

    @Override
    public void playVideo(String path) {
        // not needed
        throw new UnsupportedOperationException("Login does not play videos");
    }

    @Override
    public void handleUserInput(String input) {
        System.out.println("Handling login input: " + input);
    }
}
```

Tasks

a) Explain in 2–3 sentences which methods are not relevant for `LoginScreen` and why this breaks the Interface Segregation Principle (ISP).

b) Propose a set of smaller, more focused interfaces. For example:

```
public interface TextScreen {
    void showText(String text);
}

public interface InteractiveScreen {
    void handleUserInput(String input);
}

public interface MediaScreen {
    void showImage(String path);
    void playVideo(String path);
}
```

c) Refactor:

- Make `LoginScreen` implement only the interfaces it really needs (e.g., `TextScreen`, `InteractiveScreen`).
- Create another class `TutorialScreen` that can show text, images, and videos.

d) Explain in 2–3 sentences how the new design respects ISP.

7. Exercise 5 – DIP in a Payment Module (`dip2`)

We have a payment service:

```
package dip2;

public class PaypalPayment {

   public void pay(double amount) {
      System.out.println("Paying " + amount + " with PayPal");
   }
}

public class CheckoutService {

   private final PaypalPayment payment = new PaypalPayment();

   public void checkout(double amount) {
      // some logic ...
      payment.pay(amount);
   }
}
```

Problems

- `CheckoutService` depends directly on `PaypalPayment`.
- It is hard to switch to other payment methods (Credit Card, Cash, etc.).

Tasks

a) Explain in 3–4 sentences why this design violates the Dependency Inversion Principle (DIP).

b) Introduce an abstraction:

```
public interface PaymentMethod {
   void pay(double amount);
}
```

c) Make:

- `PaypalPayment` implement `PaymentMethod`.
- New classes `CreditCardPayment` and `CashPayment` also implement `PaymentMethod`.

d) Refactor `CheckoutService` so that it depends on `PaymentMethod` (constructor injection):

```
public class CheckoutService {

   private final PaymentMethod paymentMethod;

   public CheckoutService(PaymentMethod paymentMethod) {
      this.paymentMethod = paymentMethod;
   }

   public void checkout(double amount) {
      // some logic...
      paymentMethod.pay(amount);
   }
}
```

e) Write a `main` method to:

- Create three `CheckoutService` objects: one with PayPal, one with credit card, one with cash.
- Call `checkout` on each.

f) In 3–4 sentences, explain how this design now follows DIP.

8. Mini Project – Combining SOLID in a Small Feature (`solid_project`)

In this part, you will design a small feature for an e-learning system: **"Course Enrollment and Notification"**.

Scenario

When a student enrolls in a course:

- The system must validate the student and the course.
- The system must store the enrollment in some "repository".
- The system must send a notification to the student (email or SMS).
- In the future, the system might support different notification channels, different validation rules (e.g., max class size), and different report formats.

Tasks

a) Draw (on paper or a diagram tool) a small class diagram for your design. Try to apply:

- SRP for validation, storage, and notification.
- OCP for adding new notification types or validation rules.
- DIP for depending on abstractions (interfaces) instead of concrete classes.

b) Implement your design in package `solid_project`. Suggested classes/interfaces (you may change names):

- `EnrollmentValidator` / `EnrollmentRule`.

- `EnrollmentRepository`.
- `Notifier` (interface), `EmailNotifier`, `SmsNotifier`.
- `EnrollmentService` (coordinates everything).

c) Write a `main` method that:

- Creates an `EnrollmentService` with:
  - at least one validator,
  - at least one notifier,
  - a repository (just print to console).
- Enrolls a few example students.

d) In your report (Section 9), write a short explanation (0.5 page) describing:

- Where SRP, OCP, LSP (if used), ISP, and DIP appear in your design.
- Which trade-offs you made to keep the design simple for this lab.

9. What to Submit

Each student should submit:

- **Short report (PDF)** with:
  - Explanations for each exercise (1–5): which principle was violated, how your design fixes it.
  - For the mini project: a brief description of your design and where SOLID is applied.
- **Source code** for:
  - `srp2`, `ocp2`, `lsp2`, `isp2`, `dip2`, `solid_project`.
- (Optional) Your class diagram for the mini project (picture or PDF).