Lab08

Phạm Gia Huy- ITCSIU24034

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

a) Marking Responsibilities in the Original Code

The original CourseReportManager class (before refactoring) contained multiple responsibilities in its generateReport method:

- **Data Access (Load Data):** Loading students and grades from the "database".
- **Calculation (Statistics):** Calculating the average grade.
- **Formatting (Report):** Building the text report string.
- **Repository (Save to File):** Saving the report to a file.
- **Notification (Send Email):** Sending an email notification to the teacher.

b) Explanation of SRP Violation

The original `CourseReportManager` class violates the **Single Responsibility Principle (SRP)** because it has **multiple reasons to change**. The single `generateReport` method contains five distinct responsibilities: data access, calculation, formatting, persistence, and notification. For example, a change in how the average grade is calculated (a calculation change), or a change in the email content (a notification change), or a change in the report's file format (a persistence/formatting change) would all require modifying this single class. This coupling makes the class fragile and hard to maintain.

The refactored design respects the **Single Responsibility Principle (SRP)** because each new class now has only **one responsibility**.

c)

```java
package srp2;


import java.util.List;

public class CourseDataLoader {
    public List<String> loadStudents(String courseId) {
        System.out.println("Connecting to database...");
        System.out.println("Loading students for course " + courseId + "...");
        return List.of("An", "Binh", "Chi");
    }
```

```
    public List<Integer> loadGrades(String courseId) {
        return List.of(9, 7, 8);
    }
}

package srp2;

import java.util.List;

public class GradeCalculator {
    public double calculateAverage(List<Integer> grades) {
        double sum = 0;
        for (int g : grades) sum += g;
        return sum / grades.size();
    }
}

package srp2;

import java.util.List;

public class ReportFormatter {
    public String format(String courseId, List<String> students, List<Integer>
grades, double avg) {
        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");
        return report.toString();
    }
}

package srp2;

public class ReportStorage {
    public void save(String report) {
        System.out.println("Saving report.txt ...");
    }
}
```

```java
package srp2;

public class TeacherNotifier {
    public void notify(String report) {
        System.out.println("Sending email to teacher with report attached...");
        System.out.println(report);
    }
}
```

d)

```java
package srp2;

import java.util.List;

public class CourseReportManager {

    private CourseDataLoader loader = new CourseDataLoader();
    private GradeCalculator calculator = new GradeCalculator();
    private ReportFormatter formatter = new ReportFormatter();
    private ReportStorage storage = new ReportStorage();
    private TeacherNotifier notifier = new TeacherNotifier();

    public void generateReport(String courseId) {

        List<String> students = loader.loadStudents(courseId);
        List<Integer> grades = loader.loadGrades(courseId);

        double avg = calculator.calculateAverage(grades);

        String report = formatter.format(courseId, students, grades, avg);

        storage.save(report);

        notifier.notify(report);
    }
}
```

e)

```java
package srp2;

public class Main {
```

```
    public static void main(String[] args) {
        CourseReportManager manager = new CourseReportManager();
        manager.generateReport("C001");
    }
}
```

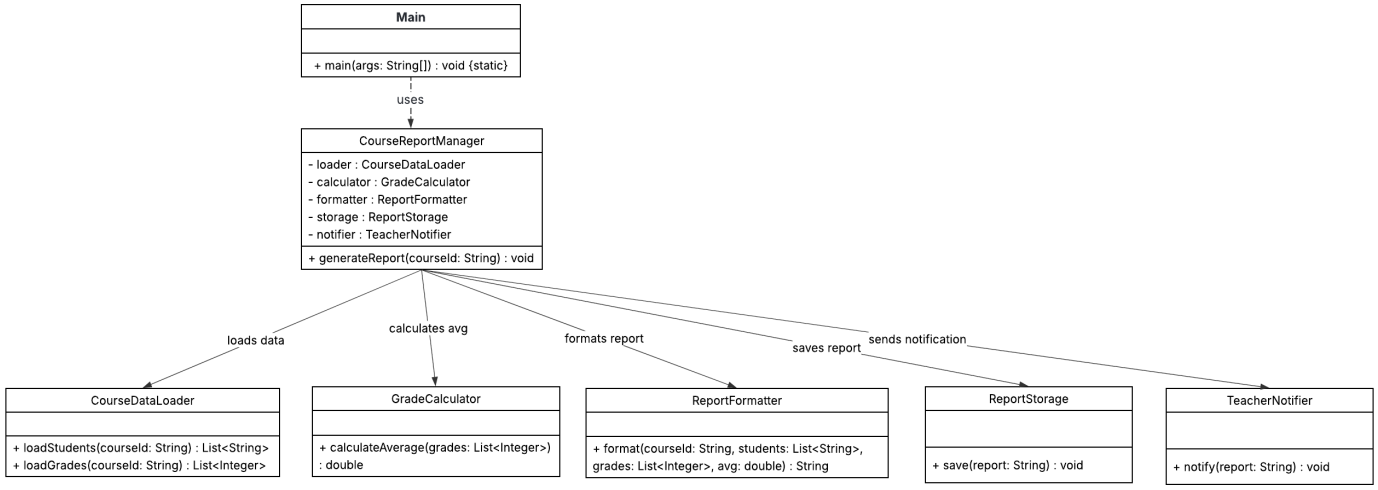
f) Explanation of How SRP is Now Respected

- CourseDataLoader handles only data access.
- GradeCalculator handles only statistics calculation.
- ReportFormatter handles only report structure/formatting.
- ReportStorage handles only saving)ư.
- TeacherNotifier handles only the notification process.

The CourseReportManager is now only a coordinator, responsible for orchestrating the workflow of these specialized classes. Changes to one domain (e.g., calculation logic) are now isolated to the corresponding class (GradeCalculator), preventing unnecessary changes in the other classes, which improves maintainability and robustness.

c) Write short explanations (2–4 sentences) for how your new design follows SOLID. (Specific to SRP)

The new design follows the SRP by successfully decoupling the process into specialized modules. Each new component, such as CourseDataLoader for data access or ReportStorage for persistence, is solely responsible for its designated task. This ensures that modifications to one area of functionality, such as how grades are loaded, will not affect unrelated areas, like how the report is formatted or sent.



Exercise 2 – OCP in a Discount System (ocp2)

a) Explain why the original design breaks the Open/Closed Principle (OCP)

The original DiscountService (before refactoring) broke the Open/Closed Principle (OCP) because it was **closed for extension but open for modification**. Adding a new user type, like "ALUMNI," required directly modifying the existing calculateDiscount method to insert a new else if condition. This modification introduces a risk of breaking existing discount logic and violates OCP, which states that software entities should be open for extension but closed for modification.

b)

```
package ocp2;


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

c)

```
package ocp2;

public class StudentDiscount implements DiscountPolicy {
    @Override
    public double discount(double totalPrice) {
        return totalPrice * 0.15;
    }
```

```
}
```

```
package ocp2;

public class TeacherDiscount implements DiscountPolicy {
    @Override
    public double discount(double totalPrice) {
        return totalPrice * 0.20;
    }
}
```

```
package ocp2;

public class VipDiscount implements DiscountPolicy {
    @Override
    public double discount(double totalPrice) {
        return totalPrice * 0.25;
    }
}
```

d)

```
package ocp2;

import java.util.Map;

public class DiscountService {

    private final Map<String, DiscountPolicy> policies;

    public DiscountService(Map<String, DiscountPolicy> policies) {
        this.policies = policies;
    }

    public double calculateDiscount(String userType, double totalPrice) {
        DiscountPolicy p = policies.getOrDefault(userType, price -> 0.0);
        return p.discount(totalPrice);
    }
}
```

e)

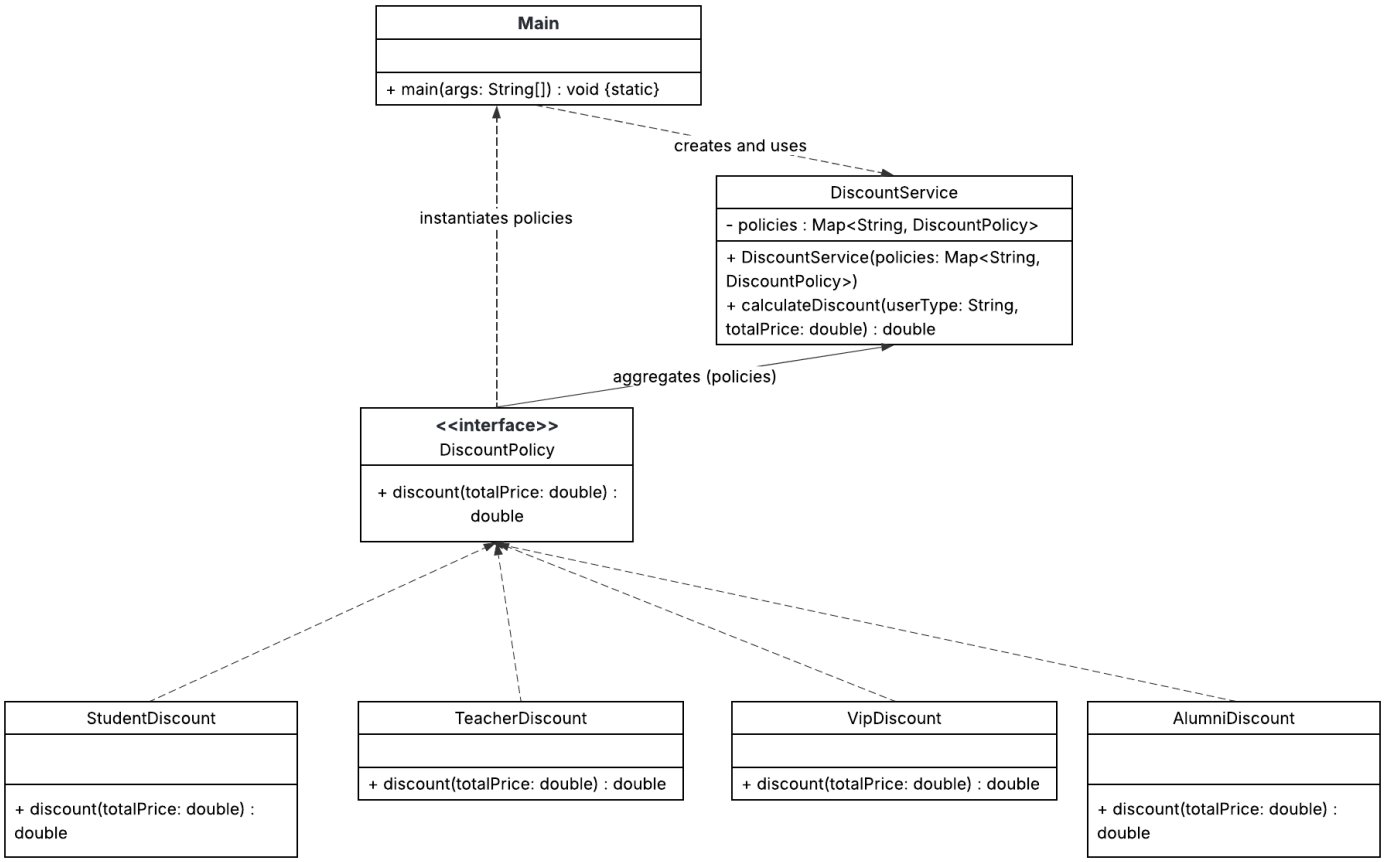
```java
package ocp2;

public class AlumniDiscount implements DiscountPolicy {
    @Override
    public double discount(double totalPrice) {
        return totalPrice * 0.10;
    }
}
```

f) Explain how the refactoring respects OCP

The refactored design respects the Open/Closed Principle (OCP) by moving the discount logic into specific classes that implement the DiscountPolicy interface. The DiscountService class is now **closed for modification** because its core logic no longer requires a chain of if-else statements. To add a new user type like ALUMNI, we simply create a new AlumniDiscount class (which is **open for extension**) and add it to the policies map in the application startup. This isolates new functionality without changing existing, tested code.

c) Write short explanations (2–4 sentences) for how your new design follows SOLID

This new design directly follows the **Open/Closed Principle (OCP)**. It uses the DiscountPolicy interface to abstract the discount calculation, allowing new discount types to be added via new classes (extension) without changing the core DiscountService logic (modification). Furthermore, the design indirectly supports the **Dependency Inversion Principle (DIP)** because the high-level DiscountService now depends on the abstraction (DiscountPolicy) rather than specific, concrete discount classes.



Exercise 3 – LSP in a Banking Example (lsp2)

a) Describe why FixedTermAccount violates the Liskov Substitution Principle (LSP)

The original design violated the **Liskov Substitution Principle (LSP)** because the FixedTermAccount subclass could not reliably substitute its parent class, BankAccount. Specifically, a client method expecting a BankAccount and calling its withdraw() method would break unexpectedly if an instance of FixedTermAccount was passed, because it explicitly threw an UnsupportedOperationException. This behavior changes the contract of the parent class, meaning the subtype is not truly substitutable for the base type without the client needing to know the specific subclass, which is the core violation of LSP.

b)

```
package lsp2;

public interface Account {
    void deposit(double amount);
```

```
    double getBalance();
}
```

```
package lsp2;

public interface Withdrawable {
    void withdraw(double amount);
}
```

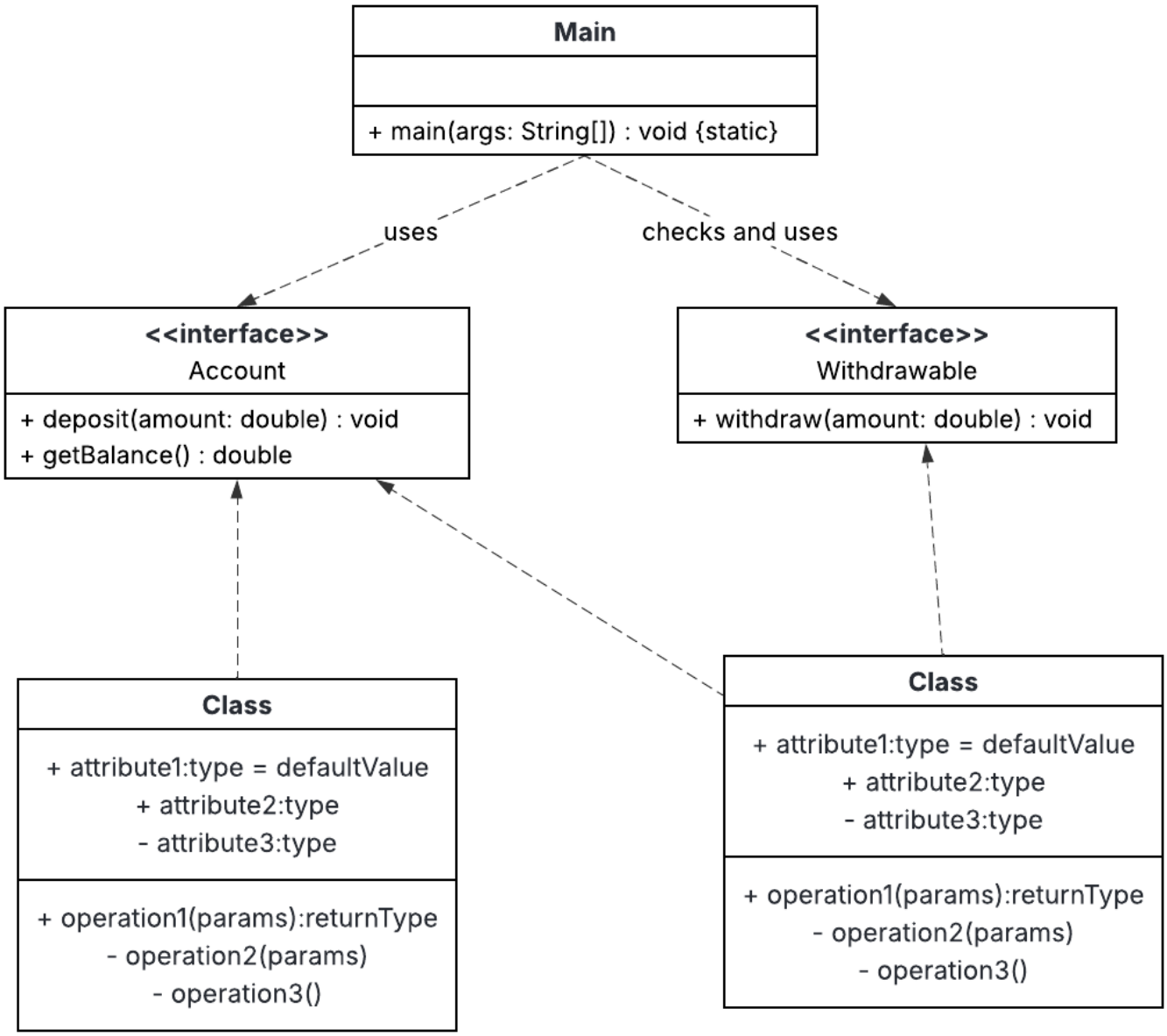
```
package lsp2;

import java.util.List;

public class Main {
    public static void main(String[] args) {

        Account a1 = new BankAccount();
        Account a2 = new FixedTermAccount();

        a1.deposit(1000);
        a2.deposit(500);

        List<Account> accounts = List.of(a1, a2);

        System.out.println("Balances:");
        for (Account acc : accounts) {
            System.out.println(acc.getBalance());
        }


        if (a1 instanceof Withdrawable w) {
            w.withdraw(200);
            System.out.println("a1 after withdraw: " + a1.getBalance());
        }


    }
}
```

e) Explain how your new design follows LSP

The new design follows the **Liskov Substitution Principle (LSP)** by separating the common capabilities from the optional ones through different interfaces. The base abstraction, Account, defines only common methods like deposit() and getBalance(), which all account types (including FixedTermAccount) can successfully implement. The behavior that might not be supported by all subtypes, specifically withdraw(), is moved to a separate interface, Withdrawable. This structure ensures that any client code operating on a list of Account objects will only call guaranteed methods, and any client needing to call withdraw() must explicitly check or depend on the Withdrawable interface, thus preventing unexpected runtime exceptions.

c) Write short explanations (2–4 sentences) for how your new design follows SOLID

This new design directly follows the **Liskov Substitution Principle (LSP)** and the **Interface Segregation Principle (ISP)**. By introducing smaller interfaces (Account and Withdrawable), clients can depend only on the minimal set of methods they require. This separation guarantees that any class substituting the Account interface will fully support its contract, thereby eliminating the LSP violation caused by the FixedTermAccount throwing exceptions for an unsupported withdraw operation.



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

a) Explain why the original design breaks the Interface Segregation Principle (ISP)

In the original design, the methods **showImage(String path)** and **playVideo(String path)** were not relevant for the LoginScreen. The LoginScreen was forced to implement these irrelevant methods from the monolithic Screen interface, resulting in implementation with UnsupportedOperationException. This forced implementation breaks the **Interface**

Segregation Principle (ISP) because clients should not be forced to depend on methods they do not use.

c)

```
package isp2;

public interface TextScreen {
void showText (String text);

}
```

```
package isp2;

public class TutorialScreen implements TextScreen, MediaSreen {
    @Override
    public void showText(String text) {
        System.out.println("TUTORIAL: " + text);
    }

    @Override
    public void showImage(String path) {
        System.out.println("Showing image: " + path);
    }

    @Override
    public void playVideo(String path) {
        System.out.println("Playing video: " + path);
    }



}
```

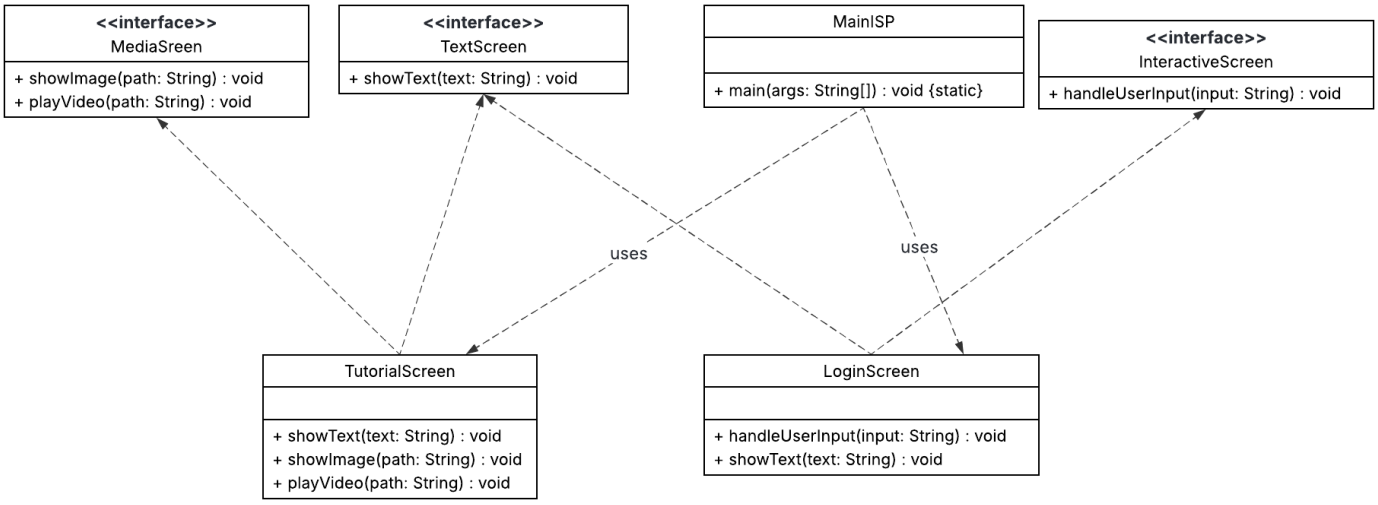
d) Explain how the new design respects ISP

The new design respects the **Interface Segregation Principle (ISP)** by breaking the large Screen interface into smaller, role-specific interfaces: TextScreen, InteractiveScreen, and MediaSreen . The LoginScreen now only implements TextScreen and InteractiveScreen, which are the only methods it needs. This segregation ensures that classes are not forced

to implement methods irrelevant to their function, eliminating the need for dummy or exception-throwing implementations.

c) Write short explanations (2–4 sentences) for how your new design follows SOLID

This new design directly follows the **Interface Segregation Principle (ISP)** by creating client-specific interfaces, thus reducing the burden on implementation classes. The high-level LoginScreen only depends on the minimal abstractions it requires (TextScreen, InteractiveScreen), ensuring that its code is clean and stable. This approach promotes better decoupling and adherence to the principle that "many client-specific interfaces are better than one general-purpose interface."



Exercise 5 – DIP in a Payment Module (dip2)

a) Explain why the original design violates the Dependency Inversion Principle (DIP)

The original design violated the **Dependency Inversion Principle (DIP)** because the high-level module, CheckoutService, directly depended on the low-level, concrete class, PaypalPayment. DIP states that high-level modules should not depend on low-level modules, but both should depend on abstractions. This direct dependency meant that the CheckoutService was tightly coupled to a single payment method, making it hard to switch to other methods like Credit Card or Cash without modification.

c)

```
package dip2;

public class PaypalPayment implements PaymentMethod {
    @Override
```

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


}
```

```
package dip2;

public class CreditCardPayment implements PaymentMethod {
    @Override
    public void pay(double amount)  {
        System.out.println("Paying " + amount + " with credit card");
    }

}
```

d)

```
package dip2;

public class CheckoutService {
    private final PaymentMethod paymentMethod;
    public CheckoutService(PaymentMethod paymentMethod) {
        this.paymentMethod = paymentMethod;
    }
    public void checkout(double amount) {
    // some logic...
    paymentMethod.pay(amount);
    }
}
```

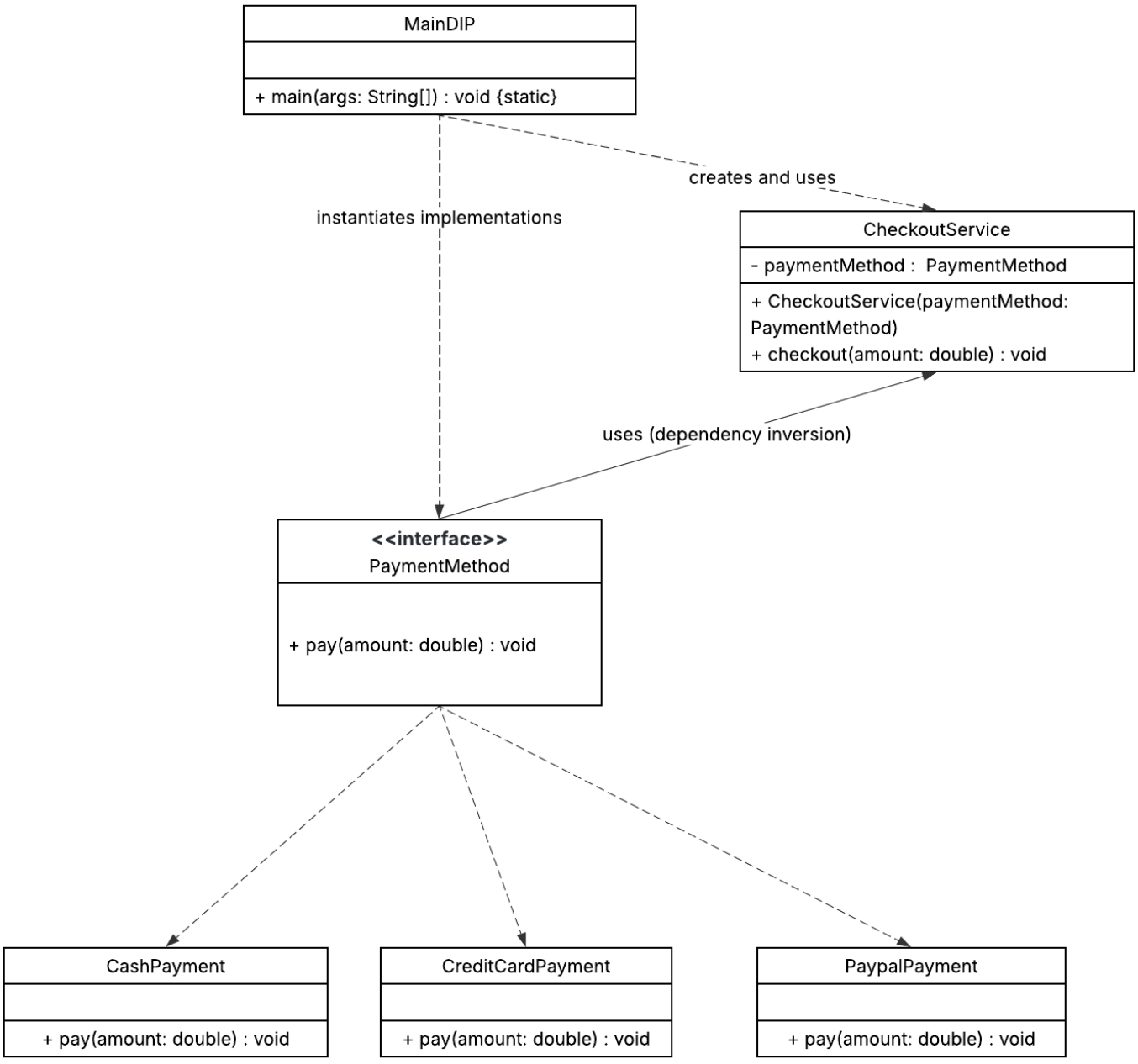
f) In 3–4 sentences, explain how this design now follows DIP

The new design now follows the **Dependency Inversion Principle (DIP)** because both the high-level CheckoutService and the low-level payment classes depend on the abstraction, PaymentMethod. CheckoutService no longer creates its dependencies but receives the abstract PaymentMethod via constructor injection. This inversion of control decouples the service from specific implementations, allowing it to easily support any new payment

method that implements the interface. The system is now flexible and open to extension without modification.

c) Write short explanations (2–4 sentences) for how your new design follows SOLID

This new design directly follows the **Dependency Inversion Principle (DIP)** by introducing the PaymentMethod interface. This abstraction ensures the high-level CheckoutService depends on the interface, not the concrete payment methods, effectively inverting the dependency. Furthermore, it supports the **Open/Closed Principle (OCP)** because new payment methods can be added by implementing the interface, without requiring any modification to the CheckoutService.



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

d) Design Explanation and Trade-offs (Mini Project)

Where SOLID Principles Appear in the Design:

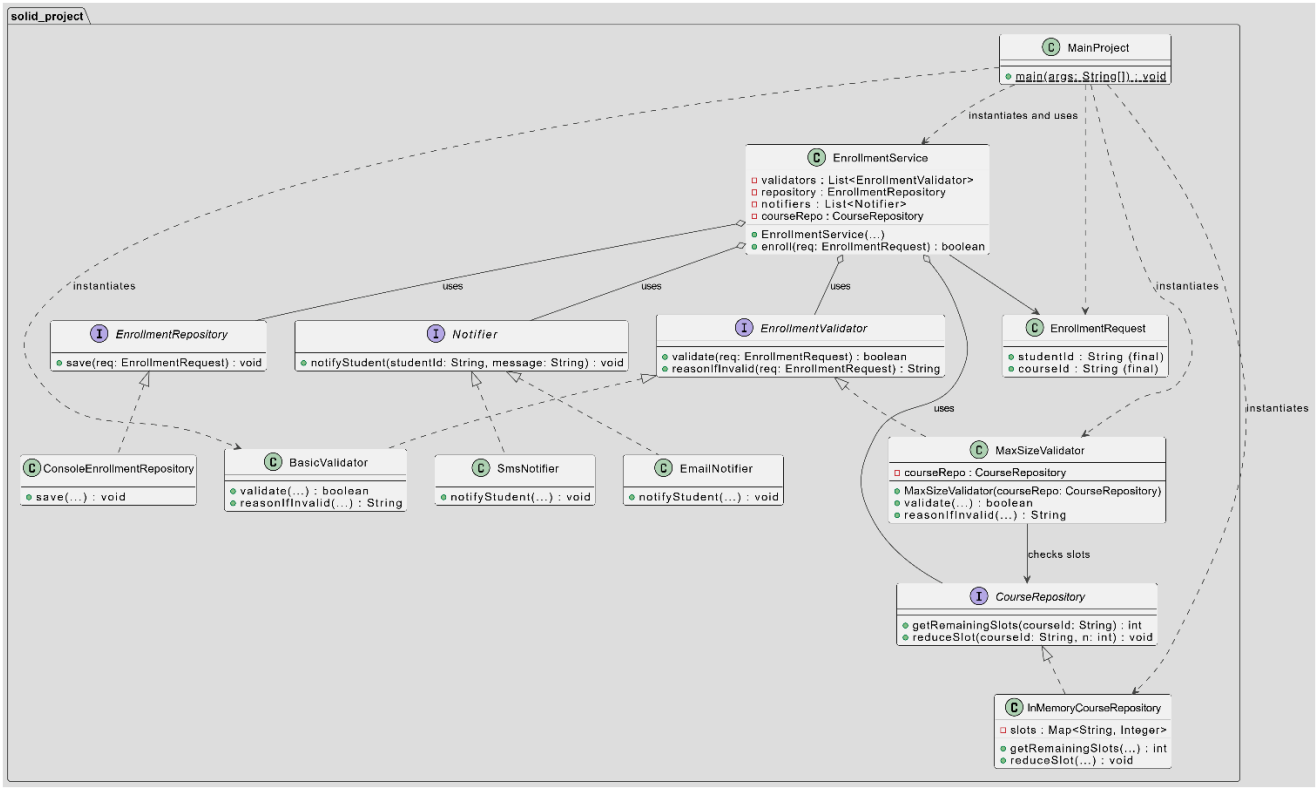
- **SRP (Single Responsibility Principle):** Each class handles a single, distinct task. EnrollmentService is solely a coordinator of the workflow. EnrollmentRepository is responsible only for data persistence , while Notifier implementations handle only

message delivery. MaxSizeValidator focuses exclusively on checking course capacity. This separation ensures that changing how data is saved does not risk breaking validation logic.

- **OCP (Open/Closed Principle):** The EnrollmentService is closed for modification but open for extension. We can add new validation rules (e.g., a PrerequisiteValidator) or new notification channels (e.g., WhatsAppNotifier) simply by creating new classes that implement the EnrollmentValidator or Notifier interfaces. The service logic itself remains untouched because it relies on lists of abstractions.
- **LSP (Liskov Substitution Principle):** The design ensures that any implementation of EnrollmentValidator (like BasicValidator or MaxSizeValidator) can be used interchangeably by the service without causing errors. The service iterates through validators uniformly, trusting that all implementations adhere to the interface contract.
- **ISP (Interface Segregation Principle):** Interfaces are kept small and focused. Notifier has only notifyStudent, and EnrollmentValidator has only validation-related methods. Classes are never forced to implement methods they do not use (e.g., a validator is not forced to implement save methods).
- **DIP (Dependency Inversion Principle):** The high-level EnrollmentService does not depend on low-level details like ConsoleEnrollmentRepository or SmsNotifier. Instead, it depends entirely on abstractions (EnrollmentRepository, Notifier, EnrollmentValidator). Dependencies are injected via the constructor, decoupling the business logic from specific infrastructure.

Trade-offs Made for Simplicity:

- **Persistence:** We used a ConsoleEnrollmentRepository that prints to the console instead of a real database to avoid the complexity of setting up SQL or JDBC for this lab.
- **Concurrency:** The InMemoryCourseRepository does not handle thread safety (locking) for slot reduction, which would be necessary in a real multi-user environment.
- **Notification Logic:** The system simply notifies all configured channels (Email and SMS) rather than implementing a complex preference system where users choose their preferred contact method.


solid_project
MainProject
main(args: String[]) : void
instantiates and uses
EnrollmentService
validators : List<EnrollmentValidator>
repository : EnrollmentRepository
notifiers : List<Notifier>
courseRepo : CourseRepository
EnrollmentService(...)
enroll(req: EnrollmentRequest) : boolean
instantiates
uses
uses
uses
uses
instantiates
EnrollmentRepository
save(req: EnrollmentRequest) : void
Notifier
notifyStudent(studentId: String, message: String) : void
EnrollmentValidator
validate(req: EnrollmentRequest) : boolean
reasonIfInvalid(req: EnrollmentRequest) : String
EnrollmentRequest
studentId : String (final)
courseId : String (final)
uses
instantiates
ConsoleEnrollmentRepository
save(...) : void
BasicValidator
validate(...) : boolean
reasonIfInvalid(...) : String
SmsNotifier
notifyStudent(...) : void
EmailNotifier
notifyStudent(...) : void
MaxSizeValidator
courseRepo : CourseRepository
MaxSizeValidator(courseRepo: CourseRepository)
validate(...) : boolean
reasonIfInvalid(...) : String
checks slots
CourseRepository
getRemainingSlots(courseId: String) : int
reduceSlot(courseId: String, n: int) : void
InMemoryCourseRepository
slots : Map<String, Integer>
getRemainingSlots(...) : int
reduceSlot(...) : void