5 SOLID Principles: Assignment — Dependency Inversion Principle (DIP) in a Notification Module

**Objective** Identify tight coupling between high- and low-level code, refactor so both depend on an abstraction, document the pattern you applied, and reflect on the design change.  
 **Concept recap** DIP means ① high-level policy modules depend only on abstractions, ② details (concrete classes) depend on those abstractions. Result: flexible, testable, maintainable code.

#### **Starter code (keep unchanged in src/main/java/legacy/)**

java

CopyEdit

package legacy;

public class EmailSender {

public void sendEmail(String msg){ System.out.println("Sending email: "+msg); }

}

package legacy;

public class NotificationService {

private final EmailSender emailSender;

public NotificationService(){ this.emailSender = new EmailSender(); }

public void notifyUser(String msg){ emailSender.sendEmail(msg); }

}

#### **Tasks**

1 Analyse the violation: in analysis/dip\_problems.md explain why NotificationService is tightly coupled and list three risks that follow.  
 2 Write a failing JUnit test (NotificationServiceUsesEmailSenderTest) that asserts the concrete dependency exists (e.g., via reflection or behaviour).  
 3 Refactor under src/main/java/clean/  
 • Create abstraction MessageSender.  
 • Implement EmailSender and a new SmsSender that both realise the interface.  
 • Modify NotificationService to accept a MessageSender through its constructor (constructor injection).  
 4 Extend the system: add PushSender without touching existing classes; show its use in Main.java.  
 5 Reflection in reflection.md  
 • What concrete changes were needed in high-level code after DIP?  
 • Which architecture pattern did you apply (answer: *Strategy via Dependency Injection*)?  
 • How does the new design improve testability and flexibility?

#### **Deliverables**

bash

CopyEdit

analysis/dip\_problems.md

src/main/java/clean/\*\* ← abstractions & senders

src/test/java/\*\* ← failing then passing tests

src/main/java/Main.java ← demo: Email, SMS, Push

reflection.md

README.md ← build/run instructions

#### **Solution reference (clean module)**

java

CopyEdit

package clean;

public interface MessageSender { void sendMessage(String msg); }

package clean;

public class EmailSender implements MessageSender {

public void sendMessage(String msg){ System.out.println("Sending email: "+msg); }

}

package clean;

public class SmsSender implements MessageSender {

public void sendMessage(String msg){ System.out.println("Sending SMS: "+msg); }

}

package clean;

public class PushSender implements MessageSender {

public void sendMessage(String msg){ System.out.println("Sending push: "+msg); }

}

package clean;

public class NotificationService {

private final MessageSender sender;

public NotificationService(MessageSender sender){ this.sender = sender; }

public void notifyUser(String msg){ sender.sendMessage(msg); }

}

package clean;

public class Main {

public static void main(String[] args){

new NotificationService(new EmailSender()).notifyUser("Hello via email");

new NotificationService(new SmsSender()).notifyUser("Hello via SMS");

new NotificationService(new PushSender()).notifyUser("Hello via push");

}

}

*Architecture pattern used*: **Strategy Pattern combined with Constructor-based Dependency Injection**—the high-level service selects behaviour at runtime by receiving a MessageSender strategy, satisfying the Dependency Inversion Principle.