

✅ DIP (Dependency Inversion Principle) – Banking Domain Case Study

Definition of DIP (from SOLID)

"High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions."

🏦 Case Study Scenario: Payment Processing System in a Bank

A bank offers multiple ways to **process payments**:

- NEFT
- IMPS
- UPI
- RTGS

The business logic should be **independent** of the payment method. But developers hardcoded specific payment logic directly into the main payment service.

❌ Violation of DIP: High-Level Module Depends on Low-Level Details

```
class NEFTPayment {
    public void pay(String fromAcc, String toAcc, double amount) {
        System.out.println("Processing NEFT payment...");
    }
}

class PaymentService {
    private NEFTPayment neftPayment = new NEFTPayment();

    public void makePayment(String fromAcc, String toAcc, double amount) {
        neftPayment.pay(fromAcc, toAcc, amount);
    }
}
```

🔴 Problems:

- You can't switch to IMPS or UPI without **modifying PaymentService**.
 - Adding new payment methods = **changing core business code**.
 - Tight coupling → Poor testability, poor maintainability.
-

✅ Applying DIP – Refactored with Abstractions

Step 1: Define an abstraction (interface)

```
public interface PaymentMethod {  
    void pay(String fromAccount, String toAccount, double amount);  
}
```

Step 2: Create concrete implementations (low-level modules)

```
public class NEFTPayment implements PaymentMethod {  
    public void pay(String fromAccount, String toAccount, double amount) {  
        System.out.println("NEFT payment from " + fromAccount + " to " +  
toAccount);  
    }  
}  
  
public class UPIPayment implements PaymentMethod {  
    public void pay(String fromAccount, String toAccount, double amount) {  
        System.out.println("UPI payment from " + fromAccount + " to " +  
toAccount);  
    }  
}
```

Step 3: Inject abstraction into high-level business logic

```
public class PaymentService {  
    private final PaymentMethod paymentMethod;  
  
    // Dependency Injection via Constructor  
    public PaymentService(PaymentMethod paymentMethod) {  
        this.paymentMethod = paymentMethod;  
    }  
  
    public void process(String from, String to, double amt) {  
        paymentMethod.pay(from, to, amt);  
    }  
}
```

Step 4: Client Code

```
public class BankApp {  
    public static void main(String[] args) {  
        PaymentMethod method = new NEFTPayment(); // OR new UPIPayment();  
        PaymentService service = new PaymentService(method);  
  
        service.process("123456", "789012", 1000.0);  
    }  
}
```

Benefits of DIP in Action

Without DIP	With DIP (Abstraction)
Tight coupling	Loose coupling
Can't swap payment methods easily	Can plug-and-play new methods

Without DIP	With DIP (Abstraction)
Hard to unit test	Easy to test with mock <code>PaymentMethod</code>
Violates OCP, SRP	Supports all SOLID principles

BONUS: Extend with Strategy + DIP + Factory

You can inject `PaymentMethod` dynamically using a Factory or Spring `@Qualifier` and easily support runtime selection.

Analogy:

Imagine a **power socket (interface)** where you can plug any device (concrete implementation) — charger, toaster, laptop. The **wall (business logic)** doesn't change for each device.

Teaching Tip

Create an interface `NotificationService` with Email/SMS implementations and ask students to apply DIP into `TransactionAlertService`.

Relationship bw DI and DIP

Dependency Injection (DI) is indeed a **way to achieve the Dependency Inversion Principle (DIP)** in practical terms.

Clarifying the Relationship:

Concept	What it Means
DIP (Dependency Inversion Principle)	A design principle that promotes programming to abstractions (not to concrete implementations).
DI (Dependency Injection)	A technique (mechanism) to implement DIP by injecting dependencies rather than hardcoding them.

Analogy:

DIP is like saying:
“I want to play music through *any* audio system (speaker, headphones, Bluetooth). Just give me an audio interface.”

DI is like someone handing you the appropriate device when needed, instead of you building or choosing it yourself.

Without DI (Violation of DIP):

```
class PaymentService {  
    private final NEFTPayment payment = new NEFTPayment(); // tightly coupled  
}
```

With DIP + DI:

```
class PaymentService {  
    private final PaymentMethod payment;  
  
    // Constructor Injection (a form of DI)  
    public PaymentService(PaymentMethod payment) {  
        this.payment = payment;  
    }  
}
```

Summary:

- **DIP** is the **principle** (a guideline for better design).
 - **DI** is the **tool/technique** to realize that principle.
 - DI helps you inject **abstractions** into your code, aligning with DIP.
-

What are the other ways to achieve DIP beside DI

While **Dependency Injection (DI)** is the most **popular and practical** way to achieve the **Dependency Inversion Principle (DIP)**, it is **not the only way**. DIP is a *design principle*, and there are **other ways to follow it** besides using a DI framework or explicit injection.

Ways to Achieve DIP Besides DI

Technique	Description	Example
Factory Pattern	Use a Factory to create instances of abstractions, decoupling construction logic.	A <code>PaymentFactory</code> that returns <code>PaymentMethod</code> objects.
Service Locator Pattern	A central registry that returns dependencies by type or name.	<code>ServiceLocator.getService("Payment")</code> returns a <code>PaymentMethod</code> .
Plugin Architecture / SPI (Service Provider Interface)	Use Java SPI or plugin loaders to load implementations at runtime.	JDBC driver loading using <code>ServiceLoader</code> .
Event-Driven / Observer Pattern	Subscribers depend on interfaces, and concrete publishers fire events.	<code>AccountNotifier</code> implements <code>EventListener</code>

Technique	Description	Example
Reflection + Configuration	Use configuration files and reflection to instantiate classes dynamically.	Reading class name from a file and instantiating via <code>Class.forName()</code> .

1: DIP via Factory (No DI Framework)

```
interface MessageService {
    void sendMessage(String to, String message);
}

class EmailService implements MessageService {
    public void sendMessage(String to, String message) {
        System.out.println("Email to " + to + ": " + message);
    }
}

class MessageServiceFactory {
    public static MessageService getService() {
        return new EmailService(); // Could later return SMS, etc.
    }
}

class NotificationClient {
    private final MessageService service;

    public NotificationClient() {
        this.service = MessageServiceFactory.getService(); // Depends on abstraction
    }

    public void notifyUser() {
        service.sendMessage("user@example.com", "Hello!");
    }
}
```

✓ *No DI framework used, still adheres to DIP.*

Summary

Technique	Uses Abstraction	Concrete Decoupled	DIP Compliant
Constructor Injection	✓	✓	✓
Factory Pattern	✓	✓	✓
Service Locator	✓	✗ (hidden dependencies)	⚠ Partially
Plugin Architecture	✓	✓	✓
Reflection & Config	✓	✓	✓

✓ Best Practices:

- Use **DI (Constructor Injection)** when possible — clear, testable, and framework-supported (Spring, Jakarta EE).

- Use **Factory** when you need **custom creation logic**.
- Avoid overusing **Service Locator** — can become an **anti-pattern** if it hides dependencies.

Would you like a visual diagram or code comparison table of these approaches?

Code with Factory Pattern

Working example that applies the **Factory Pattern** to achieve the **Dependency Inversion Principle (DIP)** — **without using Dependency Injection** frameworks.

Use Case: Sending Notifications (Email or SMS)

We'll refactor a notification system where:

- High-level module (NotificationClient) depends on an **abstraction** (MessageService).
 - Concrete classes like EmailService and SMSService implement the abstraction.
 - A **Factory** decides which implementation to use — achieving DIP.
-

Step-by-Step Code with Factory Pattern

1 MessageService (Abstraction)

```
public interface MessageService {  
    void sendMessage(String to, String message);  
}
```

2 EmailService (Concrete Implementation)

```
public class EmailService implements MessageService {  
    @Override  
    public void sendMessage(String to, String message) {  
        System.out.println("✉ Email sent to " + to + " with message: " +  
message);  
    }  
}
```

3 SMSService (Another Concrete Implementation)

```
public class SMSService implements MessageService {  
    @Override  
    public void sendMessage(String to, String message) {  
        System.out.println("📱 SMS sent to " + to + " with message: " +  
message);  
    }  
}
```

```
}  
}
```

4 MessageServiceFactory (Factory Pattern)

```
public class MessageServiceFactory {  
    public static MessageService getService(String type) {  
        if ("email".equalsIgnoreCase(type)) {  
            return new EmailService();  
        } else if ("sms".equalsIgnoreCase(type)) {  
            return new SMSService();  
        }  
        throw new IllegalArgumentException("Unknown message service type: " +  
type);  
    }  
}
```


5 NotificationClient (High-Level Module Using Abstraction)

```
public class NotificationClient {  
    private final MessageService service;  
  
    public NotificationClient(String serviceType) {  
        this.service = MessageServiceFactory.getService(serviceType); // DIP via  
Factory  
    }  
  
    public void notifyUser(String to, String message) {  
        service.sendMessage(to, message);  
    }  
}
```

6 Main Class

```
public class Main {  
    public static void main(String[] args) {  
        NotificationClient client = new NotificationClient("email"); // or "sms"  
        client.notifyUser("raj@example.com", "Welcome to the system!");  
    }  
}
```

Output

 Email sent to raj@example.com with message: Welcome to the system!

How DIP Is Achieved

- NotificationClient depends on MessageService (interface), not concrete classes.

- Object creation is outsourced to `MessageServiceFactory`, so we don't directly new any implementation inside the client.
 - We can add more implementations without changing `NotificationClient`.
-

Extra Stuff

1. What Is JNDI in EJB?

- **JNDI (Java Naming and Directory Interface)** is a Java API that allows Java clients to discover and look up data and objects (like EJBs, `DataSource`s) via a name.
 - In EJB, objects like `javax.ejb.EJBHome` or `javax.sql.DataSource` are registered in a **naming service**.
 - Clients perform **JNDI lookups** to retrieve these objects.
-

2. Why JNDI Resembles a Factory?



Because:

- Both JNDI and Factory return an object instance (e.g., `MyService myService = MyFactory.getService()` or `InitialContext.lookup("jndi/myService")`).
- Both separate the object **creation logic** from the **usage logic**.

But they are **not exactly the same**.

3. Formal Name: Service Locator Pattern

What you're describing is actually the **Service Locator Pattern**, not Factory.

Feature	Factory Pattern	Service Locator Pattern (JNDI)
Purpose	Encapsulate object creation	Encapsulate JNDI or remote service lookup
Focus	Local object instantiation	Discovering and caching remote/local services
Example usage	<code>new SomeService()</code> or config-based	<code>ctx.lookup("java:global/myEjb")</code>
Typical in	Spring, core Java apps	EJB, legacy Java EE apps
Creational Pattern?	 Yes	 No (It's a structural/service access pattern)

4. Example: Using JNDI (Service Locator Pattern)


```

public class ServiceLocator {

    private static InitialContext context;

    static {
        try {
            context = new InitialContext();
        } catch (NamingException e) {
            throw new RuntimeException(e);
        }
    }

    public static Object getService(String jndiName) {
        try {
            return context.lookup(jndiName);
        } catch (NamingException e) {
            throw new RuntimeException("Service not found: " + jndiName);
        }
    }
}

```

Usage:

```

// Client code using service locator
MyRemoteEJB ejb = (MyRemoteEJB)
ServiceLocator.getService("java:global/MyApp/MyBean");
ejb.process();

```

5. Why Not Just Use Factory?

Because:

- **JNDI** services often live in a **different JVM or container** (like EJB on WebLogic or GlassFish).
- **Factories** usually instantiate or configure objects within the **same JVM**.

Downsides of JNDI & Service Locator

- Hard-coded names ("java:global/MyApp/MyBean")
- Not type-safe
- Verbose error handling
- Encourages global access

Modern Alternative

In modern Java:

- **Dependency Injection (DI)** has replaced much of this.
- Frameworks like **Spring** auto-wire beans using annotations like `@Autowired`, avoiding manual lookup.

- Even Jakarta EE (post EJB 3.1) prefers `@EJB` or `@Inject`.



Summary

Feature	JNDI-Based (Service Locator)	Factory Pattern
Design Pattern	Service Locator	Creational (Factory)
Intent	Discover and return services	Create and return objects
Usage	Remote objects (EJBs, JMS)	Any POJOs, local beans
Modern alt	CDI, Spring DI	Spring FactoryBeans

Would you like a full example comparing:

- `ServiceLocator` (JNDI)
- `Factory` (Pure Java/Spring)