

By: Eman A. Hjazi

## *Software Design Principles: SOLID, DRY, and KISS principles*

# Software Design Principles

KISS

SOLID

YAGNI

DRY

Notes [By Linkedin: Eman A. Hjazi](#)

## Introduction to Design Principles

Design Principles هي أسلوب تفكير أو مجموعة من القواعد والمبادئ التي يستخدمها المبرمج لبناء كود برمجي بطريقة صحيحة. الغرض منها ليس فقط بناء الكود، ولكن أيضاً لمساعدة المبرمج عندما يخطر في أي مشروع ليكون على دراية أو توقع بأن الكود الذي سيجهده أمامه مبني وفق مبدأ معين .

- **Design Principles vs. Design Patterns:**

من المهم التفريق بينم

- **Design Patterns:** Templates for solving specific, recurring problems (e.g., Singleton, Factory). حلول تم بناؤها علشان تحل مشكلة برمجية معينة
- **Design Principles:** General guidelines for writing clean, maintainable code, applicable across projects. معايير تنطبق على أي مشروع برمجي

Example: You may intuitively apply principles like DRY without realizing they are formal design principles.

البعض قد يطبق مبادئ التصميم بشكل بديهي دون أن يدرك أنها مبادئ تصميم قائمة بحد ذاتها.

هذه المبادئ يمكن أن تكون دليلاً للبعض لمعرفة من أين يبدأ التفكير عند الانخراط في أي مشروع برمجي

- The **SOLID** as the most well-known set of principles, often asked about **in job interviews, especially for backend or full-stack roles**. Other principles like DRY and KISS are also covered as foundational concepts.

## 1- DRY (Don't Repeat Yourself)

**DRY** is a principle aimed at reducing code duplication to improve maintainability and readability.

- **Core Idea:** Avoid repeating the same logic in multiple places. Instead, encapsulate it in a function or method for reuse.
- **Example:**

---

  - **Scenario:** Two functions, **printUserData** and **printAdminData**, both print a user's name, and email.
  - **Issue:** Duplicated logic for printing name and email.
  - **Solution:** Use the existing function **printUserData** inside **printAdminData** to avoid repeating the same printing code, ensuring code reuse and easier maintenance.

### // ✓ 1. Before DRY

```
void printUserData(User user) {  
    System.out.println("Name: " + user.name);  
    System.out.println("Email: " + user.email);  
}  
  
void printAdminData(Admin admin) {  
    System.out.println("Name: " + admin.name);  
    System.out.println("Email: " + admin.email);  
}  
  
// ♦ Issue: Code duplication - same logic repeated in both methods.
```

By: Eman A. Hjazi

## // ✔ 2. After DRY (Method Reuse)

```
void printUserData(User user) {
    System.out.println("Name: " + user.name);
    System.out.println("Email: " + user.email);
}

void printAdminData(Admin admin) {
    printUserData(admin); // assuming Admin extends User
}

// ♦ Improvement: Reduced repetition by reusing code.
```

## // 3. Using One Method with instanceof Check

```
void printData(Object obj) {
    if (obj instanceof User) {
        User user = (User) obj;
        System.out.println("Name: " + user.name);
        System.out.println("Email: " + user.email);
    } else if (obj instanceof Admin) {
        Admin admin = (Admin) obj;
        System.out.println("Name: " + admin.name);
        System.out.println("Email: " + admin.email);
    } else {
        System.out.println("Unknown object");
    }
}

// ♦ Note: This approach is less preferred as it violates OOP principles
and can lead to code that is harder to maintain.
```

### // ✔ 3. Best Practice – Abstraction Layer (OOP)

```
abstract class Person {  
    String name;  
    String email;  
    Person(String name, String email) {  
        this.name = name;  
        this.email = email;  
    }  
    void printData() {  
        System.out.println("Name: " + name);  
        System.out.println("Email: " + email);  
    }  
}  
  
class User extends Person {  
    User(String name, String email) {  
        super(name, email);  
    }  
}  
  
class Admin extends Person {  
    Admin(String name, String email) {  
        super(name, email);  
    }  
}
```

**By: Eman A. Hjazi**

```
// Usage
public class Main {
    public static void main(String[] args) {
        User user = new User("Alice", "alice@example.com");
        Admin admin = new Admin("Bob", "bob@example.com");

        user.printData();
        admin.printData();
    }
}

/*💎 Benefit:
Code is reusable
Clean and follows OOP principles
Only one method (printData) needed in the abstract class

*/
```

**DRY** is often applied intuitively by developers, but highlighting it as a principle reinforces its importance.

---

• **Benefits DRY:**

- Reduces code duplication.
- Simplifies maintenance (update logic in one place).
- Enhances readability.

## 2- KISS (Keep It Simple, Stupid)

**KISS** complements DRY by advocating for simplicity in code design.

- **Core Idea:** Write code as simply as possible to solve the problem at hand, avoiding unnecessary complexity.

◦ مبدأ مكمل لمبدأ DRY، يدعو إلى عدم تعقيد الأمور قدر الإمكان في كتابة الكود وتطبيق الحلول بأبسط قدر ممكن من البساطة.

◦ **مثال:** إذا كانت لديك عملية (مثل تصفية قائمة الأرقام إلى زوجية أو فردية) تستخدم لمرة واحدة فقط، فليس هناك داعٍ لإنشاء دوال منفصلة مثل `getOdds` و `getEvens` لها. يمكنك استخدامها مباشرة في مكانها. تنشأ دوال منفصلة فقط إذا كان الكود سيستخدم أكثر من مرة لتقليل التكرار.

- **Example:**

- Scenario: Filtering a list of numbers into even and odd numbers.

### Overcomplicated Approach: Not KISS

```
function getEvens(numbers) {  
  return numbers.filter(num => num % 2 === 0);  
}  
function getOdds(numbers) {  
  return numbers.filter(num => num % 2 !== 0);  
}  
console.log(getEvens(numbers));  
console.log(getOdds(numbers));
```

**Issue:** Two separate functions are created, but if used only once, they add unnecessary abstraction.

### Simplified Approach: KISS

```
console.log(numbers.filter(num => num % 2 === 0));  
console.log(numbers.filter(num => num % 2 !== 0));
```

**Benefit:** Direct use of filter avoids extra functions when the logic is used only once.

اللجوء إلى الدوال المنفصلة يكون منطقيًا فقط إذا كان هناك استدعاء متكرر لنفس كتلة الكود.

**By: Eman A. Hjazi**

- **When to Abstract:**

---

- Create functions like `getEvens` or `getOdds` only if the logic is reused multiple times.
- Example: If filtering evens/odds is needed in multiple parts of the codebase, abstraction makes sense.

- **Key Takeaway:**

---

- Simplicity reduces maintenance overhead.
- Avoid over-engineering by assessing whether abstraction is necessary.

**KISS** is intuitive but requires conscious effort to avoid overcomplicating solutions.



### 3. SOLID Principles

- **SOLID** is an acronym for **five principles** that ensure robust, scalable, and maintainable object-oriented code. **These principles are critical for backend, full-stack, or software engineering roles and are often tested in interviews.**

◦ هي خمسة مبادئ تشكل معًا أساس البرمجة كائنية التوجه (OOP)، وكل مبدأ فيها يبني على الذي يسبقه.

#### Overview of SOLID



Each principle builds on the previous one, **forming a cohesive framework for writing high-quality code**. Two key concepts underpin SOLID: **cohesion** and **coupling**.

## Cohesion vs Coupling – Explained with Code

### Cohesion:

The degree to which elements within a component (e.g., a class,function,model) belong together logically.

° يعني مدى ترابط وتجانس المحتويات الموجودة داخل الكومبوننت (الوحدة الواحدة أو الكلاس) مع الهدف الذي أنشئ لأجله.

**High Cohesion = Class does one clear job**

- A class contains only methods and attributes related to its specific purpose.

**Low Cohesion = Class does too many unrelated things**

- A class handles unrelated tasks, violating SRP.

### Goal:

*Achieve **high cohesion** to ensure each component has a clear, singular purpose.*

### ✗ Low Cohesion Example:

```
class DocumentManager {  
    void printDocument() { /* print logic */ }  
    void saveDocument() { /* save logic */ }  
    void sendEmail() { /* unrelated responsibility */ }  
    void connectToNetwork() { /* unrelated responsibility */ }  
}
```

### Issue:

This class mixes printing, saving, emailing, and networking = **violates Single Responsibility Principle (SRP) = Low Cohesion**

## ✓ High Cohesion Example:

```
class DocumentPrinter {
    void printDocument() { /* print logic */ }
}

class DocumentSaver {
    void saveDocument() { /* save logic */ }
}
```

### Benefit:

Each class handles one focused task → **High Cohesion**

Easier to maintain, test, and reuse.

## Coupling

- **Definition:** The degree of dependency between components (e.g., classes or modules).
- **Tight Coupling:** High dependency, where one component cannot function without another.

**Tight Coupling = Strong dependency between classes**

### السيناريو:

تخيل أن لديك مطعمًا، والمطعم يعتمد على مورد واحد فقط للخبز:

### Tight Coupling اعتماد قوي :-

- المطعم يتعامل فقط مع مخبز أبو علي.
  - كل مرة يريد خبز، يتصل مباشرة بأبو علي بالاسم ورقم الهاتف.
  - إذا أغلق مخبز أبو علي، يتوقف المطعم عن العمل تمامًا.
- هذا يسمى – **Tight Coupling** المطعم مربوط مباشرة بمورد واحد، ولا يستطيع التكيف بسهولة.

### Loose Coupling اعتماد ضعيف

- المطعم لا يتصل بأبو علي مباشرة.
  - بل يستخدم واجهة "مزود خبز" (BreadProvider).
  - يمكن تغيير المورد وقت الحاجة (أبو علي، أو مخبز جديد) دون تغيير طريقة طلب الخبز داخل المطعم.
- هذا يسمى – **Loose Coupling** النظام أكثر مرونة وسهولة في الصيانة والتوسيع

### ✗ Tight Coupling Example:

```
class AbuAliBakery {
    public String getBread() {
        return "Bread from Abu Ali Bakery";
    }
}

class Restaurant {
    AbuAliBakery bakery = new AbuAliBakery (); // Tight coupling

    void serveBread() {
        System.out.println(bakery.getBread());
    }
}
```

Issue : إذا تغيرت طريقة توفير الخبز، يجب تعديل كود Restaurant.

### ✓ Loose Coupling

Low dependency, where components can operate independently.

**Loose Coupling = Minimal dependency; classes work independently**

```
// Step 1: Abstraction
interface BreadProvider {
    String getBread();
}

// Step 2: Concrete implementations
class AbuAliBakery implements BreadProvider {
    public String getBread() {
        return "Bread from Abu Ali Bakery";
    }
}

class NewBakery implements BreadProvider {
    public String getBread() {
        return "Bread from New Bakery";
    }
}
```

By: Eman A. Hjazi

```
// Step 3: Restaurant depends on abstraction
class Restaurant {
    BreadProvider provider;

    Restaurant(BreadProvider provider) {
        this.provider = provider;
    }

    void serveBread() {
        System.out.println(provider.getBread());
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        BreadProvider provider = new AbuAliBakery(); // Or: new
NewBakery()
        Restaurant restaurant = new Restaurant(provider);
        restaurant.serveBread();
    }
}
```

### Goal:

*Achieve loose coupling to reduce dependencies and improve flexibility.*

✓ الفوائد في الحالة الثانية:

- يمكنك تغيير المورد بسهولة.
  - Restaurant لا يعرف تفاصيل المورد.
  - الكود مرن، قابل للتوسعة، ويطبق مبدأ الـ Dependency Inversion.
  - الاعتمادية القليلة (Low Coupling) أو الكابلينج المفكوك (Loose Coupling) هو المطلوب، ويعني أن الكائنات ليست معتمدة بشكل كبير على كائنات أخرى في دورة حياتها أو تعاملها.
- مثال: علاقة الشارع بالسيارة (اعتمادية منخفضة) أفضل من علاقة سكة الحديد بالقطار (اعتمادية عالية)، حيث يمكن للسيارة أن تسير بدون شارع مُعَبَّد بالضرورة، بينما القطار لا يعمل بدون سكة حديد.

## S: Single Responsibility Principle (SRP)

**Core Idea:** Each class should only have one job.

A class should have only one reason to change, meaning it should have a single responsibility.

«ينص على أن الكلاس أو الكومبوننت يجب أن يكون له مسؤولية واحدة فقط أو "سبب واحد للتغيير".

- مثال: كلاس **Square** يحتوي على دوال حسابية (**calculateArea, calculatePerimeter**) ودوال طباعة/عرض (**draw, rotate**). هذا الكلاس يخالف SRP لأنه يحتوي على مسؤوليتين مختلفتين (العمليات الحسابية وال rendering). الحل هو فصل الدوال المتعلقة بال rendering إلى كلاس آخر مثل **SquareUI**، ليصبح لكل كلاس مسؤولية واحدة فقط

### Example:

#### Bad Design:

```
class Square {  
    private int side;  
    public int calculateArea() { return side * side; }  
    public int calculatePerimeter() { return 4 * side; }  
    public void draw() { /* Render square */ }  
    public void rotate() { /* Rotate square */ }  
}
```

- **Issue:** The Square class handles two responsibilities:
  1. Mathematical operations (calculateArea, calculatePerimeter).
  2. Rendering operations (draw, rotate).
- This violates SRP, leading to low cohesion.

By: Eman A. Hjazi

### Good Design:

```
class Square {  
    private int side;  
    public int calculateArea() { return side * side; }  
    public int calculatePerimeter() { return 4 * side; }  
}  
  
class SquareUI {  
    public void draw() { /* Render square */ }  
    public void rotate() { /* Rotate square */ }  
}
```

**Benefit:** Separating responsibilities into Square (math) and SquareUI (rendering) achieves high cohesion and adheres to SRP.

### • Key Takeaway:

- High cohesion ensures a class focuses on one task.
- Violating SRP leads to maintenance issues, as changes in one responsibility may affect unrelated functionality.
- مثال آخر على الخرق: كلاس Student يحتوي على معلومات الطالب وعملية حفظه في قاعدة البيانات. هذا خرق لـ SRP، لأنه يجب فصل عملية الحفظ إلى كلاس مسؤول عن التخزين، مثل StudentRepository.

*In Single Responsibility we aim for High Cohesion*

*In Single Responsibility we aim for Loose Coupling*

**ملاحظة هامة:** رغم أهميته، يجب تطبيق هذا المبدأ بـ "المعقول"؛ فالفصل المبالغ فيه للمسؤوليات (SRP) يمكن أن يؤدي إلى تعقيد هيكلية المشروع وتعدد الكلاسات بشكل يصعب إدارته وتعديله.

## O: Open/Closed Principle (OCP)

**Core Idea:** Classes should be **open for extension** but **closed for modification**.

- **Open for Extension:** New functionality can be added via new classes or methods.
- **Closed for Modification:** Existing code should not be altered unless fixing bugs or aligning with SRP.
- You can **extend** the behavior of a class without changing its existing code.
- Existing classes should not be modified when adding new features, **unless:**
  - You're fixing a bug.
  - You're adding a new behavior that **fits the same responsibility** (SRP).

بتجنب التعديل "عند إضافة ميزات"، لأن ذلك قد يكسر سلوكيات موجودة سابقاً.

▪ يعني ذلك أنه إذا أردت إضافة ميزة جديدة (New Feature) ، يجب أن تضيفها كتوسع أو إضافة للكود الموجود، وليس بتعديل الكود الحالي.

▪ **الاستثناءات للتعديل:** يُسمح بتعديل الكود الموجود فقط في حالتين:

- إصلاح الأخطاء (Bugs)
- أو إضافة ميزات جديدة إذا كانت تتدرج تحت نفس مسؤولية الكلاس مثل إضافة دالة استرجاع للبيانات في StudentRepository الذي هو مسؤول عن العمليات على قاعدة البيانات

**التعديل يصبح خرقاً لـ OCP إذا:**

✗ أضفت سلوكيات مختلفة لا تخص مسؤولية الكلاس

✗ عدّلت منطقاً قد يستخدم من قبل عملاء آخرين مما يؤدي إلى مشاكل في النظام

▪ تعديل منطق مشترك قد يُستخدم من قبل عدة عملاء (Clients) يؤدي إلى كسر سلوكهم الأصلي، حتى لو كانت نيتك "تحسينه" أو "تطويره".

▪ الحل الصحيح: توسعة الكود بدلاً من تعديله، عبر الوراثة أو الواجهات أو الاستراتيجيات المناسبة.

**تابع مثال لتوضيح هذا النقطة**



## Common Pitfall:

### ◆ Modifying Shared Logic May Break Existing Clients

- **Clients** = Any other code that uses a class or method.
- When you **change an existing method's behavior**, you **risk breaking** code that **depends on the old behavior**.
- Even if you're "improving" the logic, you might unintentionally cause bugs elsewhere.

## Real-World Example:

### Original Version (used by many clients):

```
class DiscountCalculator {  
    public double calculateDiscount(Customer customer) {  
        return 0.1; // Always 10% discount  
    }  
}
```

This logic is **stable** and expected by all clients.

For example, billing, reporting, or frontend UI might rely on this 10% behavior.

### Bad Design (modifies shared logic): Modified Version (violates OCP):

```
class DiscountCalculator {  
    public double calculateDiscount(Customer customer) {  
        if (customer.isPremium()) {  
            return 0.2;  
        }  
        return 0.05;  
    }  
}
```

### ● Problem:

- The method **no longer returns 10% by default**.
- Some clients now receive 5% or 20%.
- This **breaks expected behavior** in systems that relied on the original 10%.
- You unintentionally **broke other clients** just by "improving" logic.

By: Eman A. Hjazi

### Good Design (OCP-compliant extension)

---

```
interface DiscountStrategy {
    double calculate(Customer customer);
}

class FixedDiscount implements DiscountStrategy {
    public double calculate(Customer customer) {
        return 0.1;
    }
}

class TieredDiscount implements DiscountStrategy {
    public double calculate(Customer customer) {
        if (customer.isPremium()) return 0.2;
        return 0.05;
    }
}
```

Now, clients can **choose** the appropriate strategy.

No existing logic is changed = **no risk of breaking other clients.**

---

**By: Eman A. Hjazi**

**Example:**

---

**Bad Design:**

```
class Student {  
    private int id;  
    public void save() { /* Save to database */ }  
    public void setStudentId(int id) { this.id = id; }  
    public int getStudentId() { return id; }  
}
```

**Issue:** The save method inside Student violates SRP (handling database operations) and OCP (modifying Student to add new save logic requires changing the class).

**Good Design:**

```
class Student {  
    private int id;  
    public void setStudentId(int id) { this.id = id; }  
    public int getStudentId() { return id; }  
}  
  
class StudentRepository {  
    public void save(Student student) { /* Save to database */ }  
}
```

**By: Eman A. Hjazi**

### **Benefit:**

---

- Student focuses on data (SRP).
- StudentRepository handles persistence, allowing new features (e.g., listAllStudents) to be added without modifying Student (OCP).

### **Caveats:**

---

- Over-separating responsibilities can lead to excessive complexity (e.g., too many classes).
- Balance is key: follow conventions like CRUD operations in repositories but avoid over-engineering for minor features.

### **Key Takeaway:**

---

- Extend functionality through new components, not by altering existing ones.
- Maintain SRP to ensure modifications align with a single responsibility.
- **Don't change shared logic** that other parts of the system depend on.
- Use **strategy, interfaces, or inheritance** to safely extend features.
- OCP helps you **protect existing behavior** and **avoid accidental regressions**.

## L: Liskov Substitution Principle (LSP)

### Core Idea:

LSP states that objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program.

ينص مبدأ LSP على أن الكائنات من النوع الأب يجب أن تكون قابلة للاستبدال بالكائنات من الأنواع الفرعية دون أن يتغير سلوك النظام.

- It ensures **safe polymorphism**. Subclasses must behave in a way that **does not surprise or break clients** relying on the parent type.

يستغل هذا المبدأ ميزة تعدد الأشكال (Polymorphism) في البرمجة الكائنية. ينص على أن الكائنات من الفئات المشتقة (subtypes) يجب أن تكون قابلة للاستبدال بكائنات من فئاتها الأساسية (base types) دون تغيير صحة البرنامج.

الهدف هو تجنب استخدام عبارات شرطية (if-conditions) متعددة للتعامل مع أنواع مختلفة من الكائنات، وبدلاً من ذلك، التعامل معها من خلال طبقة تجريد مشتركة (Abstraction Layer)، مثل الواجهات (Interfaces) أو الفئات المجردة (Abstract Classes).

### • Example:

- Scenario: A User interface with Admin and Student implementations.
  - مثال: بدلاً من التعامل مع كائنات Admin و Student بشكل منفصل داخل حلقة تكرارية باستخدام جمل if الشرطية الكثيرة، يمكن التعامل معهما من خلال طبقة تجريد مشتركة مثل User.
  - التحدي: إذا أضفت دالة خاصة بـ Admin فقط (ليست موجودة في User) وحاولت استدعاءها أثناء التعامل مع الكائن كـ User ضمن حلقة تكرارية، فقد يحدث خطأ **هذا يقودنا إلى مبدأ ISP.**

```
interface User {  
    void accessDashboard();  
}  
  
class Admin implements User {  
    public void accessDashboard() { /* Admin-specific logic */ }  
    public void manageUsers() { /* Admin-only logic */ }  
}  
  
class Student implements User {  
    public void accessDashboard() { /* Student-specific logic */ }  
}
```

**By: Eman A. Hjazi**

```
// Usage
List<User> users = Arrays.asList(new Admin(), new Student());
for (User user : users) {
    user.accessDashboard(); // Works fine
    // user.manageUsers(); //X Compile-time ERROR: method not in User
interface
}
```

**Issue:** If we treat `Admin` and `Student` as `User` and try to call `manageUsers()`, it will break substitutability — violating LSP.

- **Solution:** Ensure subclasses only implement methods that align with the parent interface's contract.

#### • Key Takeaway:

- Subclasses must respect the **expected behavior** of the parent class/interface.
- Avoid adding methods to subclasses that break substitutability.

## I: Interface Segregation Principle (ISP)

**Core Idea:** Clients should not be forced to depend on interfaces they do not use. Split large interfaces into smaller, specific ones.

- يكمل عمل LSP ، وينص على أن "العملاء (Clients) لا يجب أن يُجبروا على الاعتماد على واجهات (Interfaces) لا يستخدمونها."  
بمعنى آخر، يجب تجنب (Fat Interfaces) التي تحتوي على عدد كبير من الدوال، ويتم تقسيمها إلى واجهات أصغر وأكثر تحديداً.
- هذا المبدأ هو تطبيق لمبدأ SRP ولكن على مستوى الواجهات؛ فبدلاً من إجبار **PDFReport** على تطبيق دوال خاصة بـ **ExcelExport** (حتى لو كانت فارغة)، يمكن فصل هذه الدوال إلى واجهة خاصة بـ **ExcelExport** فقط.
- ظهور الدوال الفارغة (Empty Method Implementations) في الكود هو مؤشر على خرق هذا المبدأ) أو SRP بسبب سوء التصميم الأساسي.

### Example:

- Bad Design:

```
interface ReportGenerator {  
    void generatePDF();  
    void generateExcel();  
}  
  
class PDFReport implements ReportGenerator {  
    public void generatePDF() { /* generate PDF */ }  
    public void generateExcel() {  
        // Not supported: throw exception or leave empty - bad design!  
    }  
}
```

**Violation:** PDFReport مضطر لتنفيذ generateExcel() بدون سبب.

- Good Design:

```
interface PDFExportable {  
    void generatePDF();  
}  
  
interface ExcelExportable {  
    void generateExcel();  
}
```

**By: Eman A. Hjazi**

```
class PDFReport implements PDFExportable {  
    public void generatePDF() { /* generate PDF */ }  
}
```

**Benefit:** Smaller interfaces ensure classes only implement relevant methods, adhering to SRP at the interface level.

**Key Takeaway:**

- Split interfaces to avoid forcing unnecessary implementations.
- Aligns with SRP by ensuring responsibilities are separated at both class and interface levels.



## D: Dependency Inversion Principle (DIP)

### Core Idea:

High-level modules **should not depend** on low-level modules.  
**Both** should depend on **abstractions** (interfaces or abstract classes).  
**Abstractions should not depend on details.**  
**Details should depend on abstractions.**

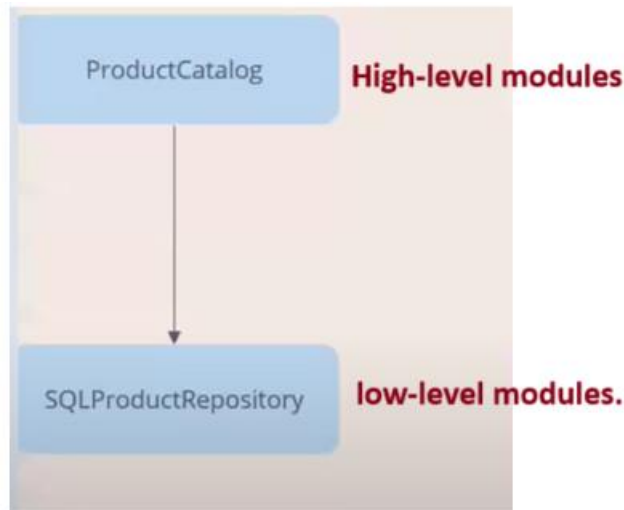
- يُعتبر هذا المبدأ تنويجاً للمبادئ السابقة ويُحققها بشكل طبيعي عند تطبيقه.
- **(High-level modules)** يجب ألا تعتمد على **(low-level modules)** كلاهما يجب أن يعتمد على التجريدات **abstraction**
- "التجريدات **(Abstractions)** يجب ألا تعتمد على التفاصيل **(details)** التفاصيل يجب أن تعتمد على التجريدات"

▪ الهدف: تحقيق كابلينج منخفض (Low Coupling) وكوهيجن عالٍ (High Cohesion).

**High-level modules** : هي الكلاسات التي تحتوي في داخلها على كائنات أخرى أو تستدعيها.

مثال **(ProductCatalog)** كانت بدخلها عملية انشاء اوبجكت لل low level modules الي هو **SqlProductRepository**

**low-level modules** هي الموديول أو المكون الذي يتم استخدامه أو استدعاؤه داخل **High-level modules** **(SQLProductRepository)**



**كيفية التطبيق:** بدلاً من أن تعتمد High-level modules على low-level (Concrete implementations) modules يجب أن تعتمد على (Interfaces) أو (Abstract Classes) وهذا يعني أن الاعتمادية تنعكس: بدلاً من أن تكون على المكونات، تصبح على Abstraction layer .

By: Eman A. Hjazi

### What is a High-Level Module?

- **Definition:**

A **High-Level Module** is a class that contains or **uses other classes** inside it. It focuses on the **business logic** or application rules.

- **Example:**

In the following code, **ProductCatalog** is a high-level module because it **uses SQLProductRepository** to get product names.

### What is a Low-Level Module?

**Definition:**

A **Low-Level Module** is a class that is **used by other classes**. It focuses on technical details like **reading from a database**, making HTTP calls, or saving files.

Example:

**SQLProductRepository** is a low-level module because it provides the actual logic for getting the product names.

### What are Abstractions?

---

**Definition:**

An **abstraction** is a general idea that hides internal details and exposes only what is necessary.

In code, abstractions are usually created using **interfaces** or **abstract classes**.

Example:

```
public interface ProductRepository {  
    List<String> getAllProductNames();  
}
```

---

This interface tells us *what* the class should do, not *how* it does it.

**By: Eman A. Hjazi**

Why should abstractions not depend on details?

**Answer:**

According to the **Dependency Inversion Principle (DIP)**:

**Abstractions should not depend on details.**

**Details (like concrete classes) should depend on abstractions.**

---

Why?

- It **reduces coupling** between components.
- It makes your system **easier to change or extend**.
- You can **swap implementations** (e.g., switch from SQL to NoSQL) without changing the high-level code.

Final Practical Code with Explanation

**Bad Example (Without Abstraction):**

---

```
// High-Level Module
public class ProductCatalog {
    public void listAllProducts() {
        SQLProductRepository sqlProductRepository = new
SQLProductRepository(); // tightly coupled
        List<String> allProductNames =
sqlProductRepository.getAllProductNames();

        for (String name : allProductNames) {
            System.out.println(name);
        }
    }
}

// Low-Level Module
import java.util.Arrays;
import java.util.List;

public class SQLProductRepository {
    public List<String> getAllProductNames() {
        return Arrays.asList("soap", "toothpaste");
    }
}
```

**By: Eman A. Hjazi**

Here, **ProductCatalog** is tightly coupled to the concrete class **SQLProductRepository**.

### • قبل تطبيق DIP :

#### المشاكل :

- اقتران عالي **ProductCatalog** (**Tight Coupling**): معتمد مباشرة على **SQLProductRepository**، فلو بدك تغير لـ **MongoDB** ، لازم تعدل الكود.
- يخرق **DIP**: الكلاس عالي المستوى (**ProductCatalog**) معتمد على كلاس منخفض المستوى (**SQLProductRepository**) بدل أبستراكشن.
- يخرق **OCP**: لو بدك تضيف وظيفة زي **save**، لازم تعدل **ProductCatalog**، وهذا مو مرن.
- يخرق **SRP**: لو أضفت وظائف قاعدة بيانات (زي **save** داخل **ProductCatalog**، بيصير عنده أكثر من مسؤولية.

By: Eman A. Hjazi

## Good Example (With Abstraction):

```
// High-level module
public class ProductCatalog {
    private ProductRepository repo;

    public ProductCatalog(ProductRepository repo) {
        this.repo = repo;
    }

    public void listAllProducts() {
        for (String name : repo.getAllProductNames()) {
            System.out.println(name);
        }
    }
}

// Abstraction
public interface ProductRepository {
    List<String> getAllProductNames();
}

// Low-level module
public class SQLProductRepository implements ProductRepository {
    public List<String> getAllProductNames() {
        return Arrays.asList("soap", "toothpaste");
    }
}
```



### بعد تطبيق DIP :

- إنشاء طبقة تجريد (Abstraction Layer) تم إنشاء واجهة (Interface) باسم ProductRepository
- التنفيذ يعتمد على التجريد: كلاس SQLProductRepository الآن يقوم بتطبيق (implements) الواجهة. ProductRepository.
- الوحدة عالية المستوى تعتمد على التجريد: كلاس ProductCatalog لم يعد ينشئ SQLProductRepository مباشرة. بدلاً من ذلك، أصبح يستقبل كائنًا من نوع ProductRepository (الواجهة) عبر الكونستركتر الخاص به. (Constructor)
- فصل مسؤولية الإنشاء: (Creation) يتم إنشاء الكائن الفعلي SQLProductRepository في مكان آخر خارج ProductCatalog ، وغالبًا ما يتم ذلك باستخدام "فاكتوري باترن (Factory Pattern)" أو بواسطة Dependency Injection Container في الأطر العمل الحديثة.
- النتائج: أصبح ProductCatalog معتمدًا على واجهة (تجريد) بدلاً من تنفيذ ملموس. هذا يسمح بتبديل نوع قاعدة البيانات (مثل من SQL إلى NoSQL) دون الحاجة لتعديل ProductCatalog نفسه، مما يجعله أكثر مرونة وأقل اعتمادية. كما أنه يحقق SRP لـ ProductCatalog لأنه أصبح مسؤولاً فقط عن عرض المنتجات، وليس عن كيفية جلبها أو إنشاء الكائن المسؤول عن ذلك

#### • Benefit:

- ProductCatalog depends on the ProductRepository interface (abstraction), not the concrete SQLProductRepository.
- SQLProductRepository implements the interface, ensuring loose coupling.
- The ProductFactory handles object creation, further reducing coupling.

Concept	Description
<b>Abstraction</b>	Hides details and exposes behavior (via interface/abstract class).
<b>Why not depend on details?</b>	To reduce tight coupling, and make code flexible and maintainable.
<b>Correct approach</b>	High-level and low-level modules should depend on abstractions.

## "Dependency Injection" VS "Dependency Inversion"

Aspect	Dependency Inversion Principle (DIP)	Dependency Injection (DI)
Definition	A design principle that promotes dependence on abstractions instead of concrete implementations.	A design pattern that provides dependencies from the outside.
Purpose	Reduces coupling by ensuring high-level modules do not depend on low-level modules directly.	Provides a mechanism to supply dependencies dynamically.
Concept Type	Principle (part of SOLID)	Pattern (used to implement DIP)
Focus	What should depend on what	How dependencies should be provided
Implementation	Achieved by defining interfaces/abstractions	Achieved through Constructor, Property, or Method Injection

- DIP = "أصمم الكلاس ليتعامل مع واجهات، وليس كائنات جاهزة"
- DI = "أرسل له الكائن الجاهز من الخارج بدل ما ينشئه بنفسه"


في الكود السابق : ProductRepository يهتم تمريره لـ ProductCatalog عن طريق الكونستركتر، وهذه هي Dependency Injection.

- الكونستركتر ProductCatalog(ProductRepository repo) يياخذ أوبجكت من نوع ProductRepository، وهذا هو DI (حقن الاعتمادية)، لأن الأوبجكت يهتم تمريره من برا بدل ما ينشئ داخل الكلاس .  
هذا ما يُسمى بـ Constructor Injection

الفاكتور (ProductFactory) يساعد في فصل عملية إنشاء الأوبجكتس، وهيك بنخلي الكود أنظف وأسهل للصيانة.

By: Eman A. Hjazi

```
public class ECommerceMainApplication {  
    public static void main(String[] args) {  
        ProductRepository productRepository =  
            ProductFactory.create();  
        ProductCatalog productCatalog = new  
            ProductCatalog(productRepository);  
        productCatalog.listAllProducts();  
    }  
}  
  
public class ProductFactory {  
    public static ProductRepository create() {  
        return new SQLProductRepository();  
    }  
}
```

A vertical arrow points from the `create()` method call in the `main` method of `ECommerceMainApplication` to the `create()` method definition in the `ProductFactory` class.



**All Comments and Remarks in Code or from Instructor**

• **Balancing Principles:**

- Over-applying SOLID (e.g., excessive class splitting) can lead to complexity.
- Follow team conventions (e.g., limit class size to 200-300 lines before splitting).

• **Legacy Systems:**

- Older systems may violate SOLID, leading to empty method implementations or fat interfaces.
- Refactor gradually, focusing on separating responsibilities and reducing coupling.

• الديزاين برنسبلز ليست ديزاين باترنز؛ الأولى أسلوب تفكير وقواعد، والثانية أنماط حلول لمشاكل متكررة.  
• مبادئ SOLID تبني على بعضها البعض، وتطبيق أحدها يؤدي بشكل أو بآخر إلى تطبيق الآخر.  
**High Cohesion** شيء جيد، و **Low (Loose) Coupling** شيء جيد

• تطبيق مبدأ OCP: لا يتم التعديل على الكود إلا في حالة وجود خطأ (Bug)، أو إذا كانت الإضافة الجديدة (Feature) تتوافق مع المسؤولية الواحدة للكلاس (Single Responsibility)

• تطبيق SRP المفرط (الفصل الزائد عن الحد) يمكن أن يؤدي إلى تعقيد هيكلية المشروع، وتزايد عدد الكلاسات والباكجز بشكل يصعب إدارته وتعديله. يجب تطبيق الفصل "بالمعقول"

LSP بمفرده لا يكفي لضمان تصميم مرن، ويجب أن يكمل عمله مبدأ ISP لتجنب مشاكل مثل الدوال الفارغة أو الأخطاء.

• ظهور Empty Method Implementations (تطبيق دوال فارغة) في الكود هو مؤشر على وجود عيوب في التصميم الأساسي، وأن المبرمج السابق لم يتبع مبادئ SOLID.

• مبدأ Dependency Inversion يجعل تحقيق مبادئ Cohesion و Coupling تحصيل حاصل

• في الأطر العمل (Frameworks) مثل Spring و ASP.NET، يتم تطبيق جزء كبير من مبدأ DIP و Dependency Injection تلقائيًا؛ لذا، لا يضطر المبرمج لبناء الفاكثوري والتحكم بالإنشاء يدويًا، بل يتبع الكونفشن (Convention) الخاص بالإطار العمل.

▪ "Dependency Injection" و "Dependency Inversion" مفهومان مختلفان، ويجب التفريق بينهما