



SOLID PRINCIPLES

TIM Pengelola MK Pola
Perancangan

CAPAIAN PEMBELAJARAN

Mahasiswa dapat mendefinisikan tiap prinsip SOLID dengan benar.

Mahasiswa dapat mengidentifikasi pelanggaran prinsip SOLID pada potongan kode nyata.

SOLID PRINCIPLES

SOLID adalah akronim dari **lima prinsip desain perangkat lunak berorientasi objek** yang diperkenalkan oleh **Robert C. Martin (Uncle Bob)**. Tujuannya adalah membuat software **mudah dipelihara, fleksibel, reusable, dan testable**.

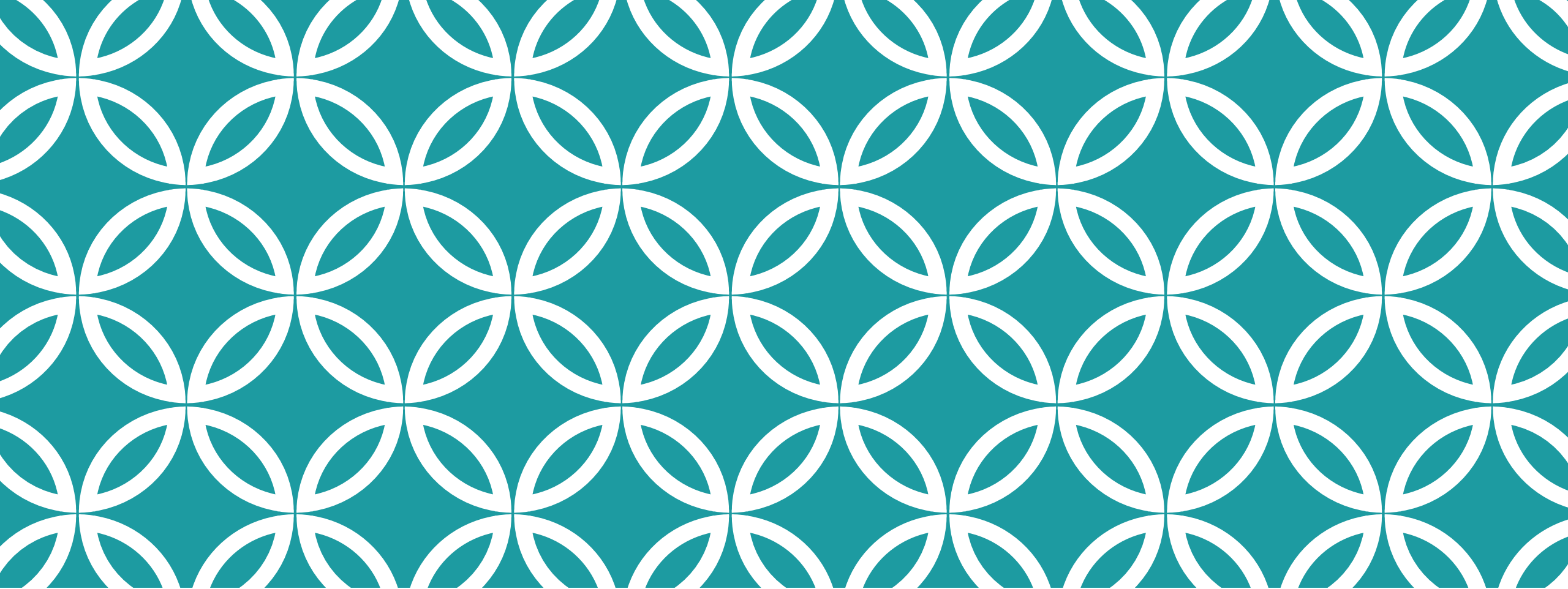
S — Single Responsibility Principle (SRP)

O — Open/Closed Principle (OCP)

L — Liskov Substitution Principle (LSP)

I — Interface Segregation Principle (ISP)

D — Dependency Inversion Principle (DIP)



**SRP (SINGLE RESPONSIBILITY
PRINCIPLE)** |

DEFINISI KLASIK

“A class should have only one reason to change.”

(Sebuah kelas seharusnya hanya memiliki satu alasan untuk berubah.)

Maksudnya: perubahan kebutuhan bisnis, teknis, atau organisasi **tidak boleh memaksa sebuah kelas berubah karena lebih dari satu alasan yang berbeda.**

Jika ada lebih dari satu alasan, berarti kelas tersebut punya **lebih dari satu tanggung jawab.**

PERSPEKTIF “ALASAN UNTUK BERUBAH”

Uncle Bob menggeser definisi dari “tanggung jawab” ke “alasan berubah”. Mengapa?

Kata “**responsibility**” bisa ambigu: apakah maksudnya “fungsi”? “peran”? “feature”?

Dengan menggantinya menjadi “**reason to change**”, kita bisa lebih jelas menilai apakah kelas melanggar SRP atau tidak.

➔ Jadi, **SRP berbicara tentang arah perubahan**: jika satu kelas harus berubah karena beberapa alasan yang datang dari aktor berbeda, maka kelas itu melanggar SRP.

PERSPEKTIF “AKTOR”

Dalam *Clean Architecture*, definisi diformalkan ulang:

“A module should be responsible to one, and only one, actor.”

Aktor = kelompok pemangku kepentingan (stakeholder) atau peran bisnis yang memicu perubahan.

Contoh aktor:

- **CFO (Accounting)** → ingin aturan perhitungan gaji berubah.
- **COO (HR)** → ingin format laporan jam kerja berubah.
- **CTO (DBA)** → ingin struktur penyimpanan karyawan berubah.

Jika satu kelas `Employee` menggabungkan ketiganya (`calculatePay()`, `reportHours()`, `save()`), maka kelas itu memiliki **tiga alasan berubah** (tiga aktor berbeda). Itu jelas melanggar SRP.

HUBUNGAN DENGAN KOHESI

Cohesion = tingkat keterkaitan tanggung jawab dalam satu modul.

SRP = *kohesi berdasar aktor.*



Modul dengan SRP tinggi berarti: semua metode di modul itu **melayani kebutuhan aktor yang sama.**

Modul yang melanggar SRP berarti: metode-metodenya melayani **aktor berbeda**, sehingga kohesi rendah.



CONTOH DEFINISI PRAKTIS

Mari bandingkan:

-  **Salah kaprah umum:**
“SRP berarti satu kelas hanya boleh punya satu fungsi.”
→ Ini terlalu sempit. Sebuah UserService boleh punya banyak fungsi (registerUser, authenticateUser, resetPassword) **asalkan semua fungsi itu melayani aktor yang sama** (misalnya: bagian user account).
-  **Benar menurut Uncle Bob:**
“SRP berarti satu kelas hanya melayani satu aktor.”
→ Jika ada fungsi yang melayani aktor berbeda, pisahkan ke kelas lain.

RINGKASAN DEFINISI SRP

Versi asli: *One reason to change.*

Versi aktor: *One actor per module.*

Makna praktis: Kelas dengan SRP → semua kodenya akan berubah **bersama-sama** saat ada permintaan perubahan dari **satu aktor**, dan tidak akan berubah bila aktor lain mengajukan permintaan.

KASUS : SISTEM KRS

```
public class KRSManager {  
    public void validateKRS(KRS krs) { /* validasi beban SKS */ }  
    public void saveKRS(KRS krs) { /* simpan ke database */ }  
    public void printKRS(KRS krs) { /* cetak laporan */ }  
}
```

Validasi KRS → tanggung jawab Bagian Akademik.

Simpan ke DB → tanggung jawab Tim Sistem Informasi (DBA).

Cetak laporan → tanggung jawab Mahasiswa/Dosen PA (untuk bimbingan).

→ Ada **tiga aktor berbeda**. Kelas ini melanggar SRP.
Kalau format cetak berubah → kelas ikut berubah. Kalau struktur database diubah → kelas ikut berubah lagi.

SOLUSI

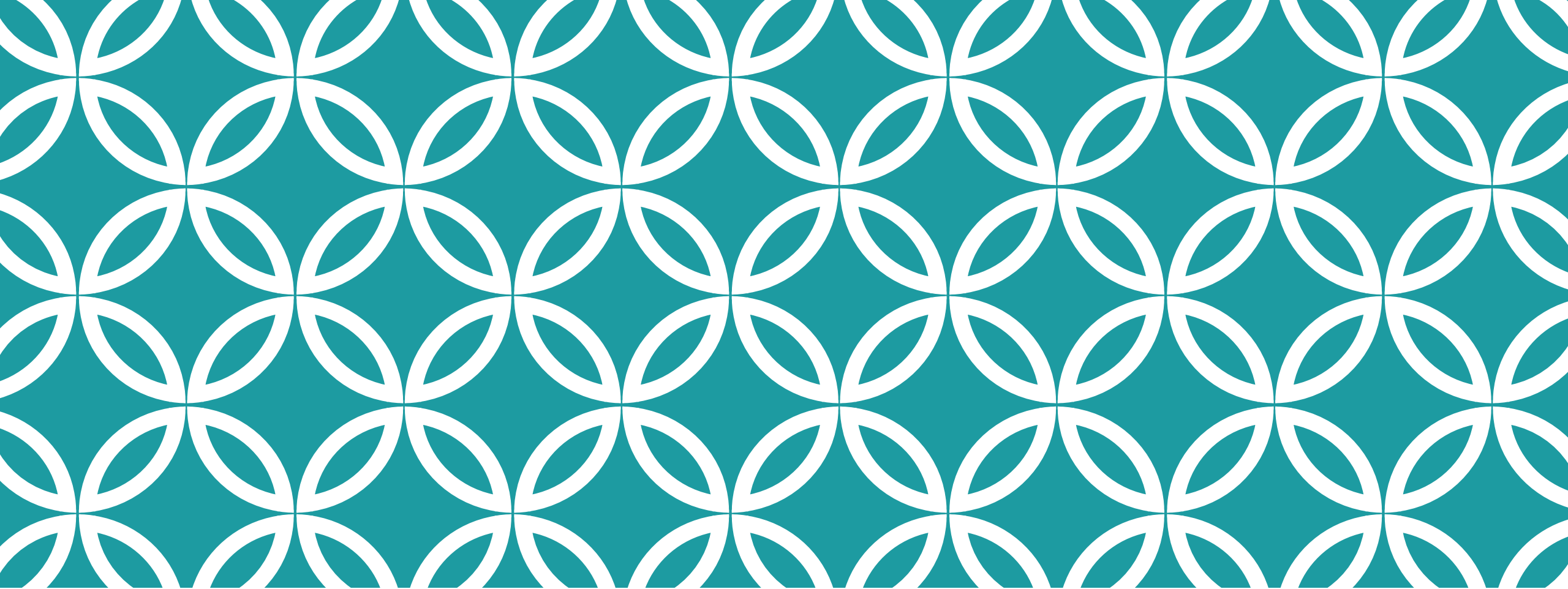
Pisahkan per tanggung jawab (aktor):

Sekarang:

- **KRSValidator** → hanya berubah bila aturan akademik berubah (aktor: Bagian Akademik).
- **KRSRepository** → hanya berubah bila teknologi DB berubah (aktor: DBA).
- **KRSPrinter** → hanya berubah bila format laporan berubah (aktor: Mahasiswa/Dosen PA).

➔ **Satu aktor = satu alasan berubah = satu kelas.**

```
public class KRSValidator {  
    public void validate(KRS krs) { /* validasi beban SKS */ }  
}  
  
public interface KRSRepository {  
    void save(KRS krs);  
}  
  
public class SqlKRSRepository implements KRSRepository { /* implementasi DB */  
  
public class KRSPrinter {  
    public void print(KRS krs) { /* cetak laporan */ }  
}
```



OCP (OPEN-CLOSE PRINCIPLE) |

DEFINISI

Definisi klasik (Bertrand Meyer, 1988):

- “Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.”

Makna:

- **Open for extension** → kita bisa menambahkan perilaku baru.
- **Closed for modification** → kita tidak perlu mengubah kode yang sudah stabil & diuji.

Dalam *Clean Architecture* Uncle Bob menekankan:

- Kode yang sudah **dideploy & berjalan stabil** harus diperlakukan sebagai **tertutup untuk modifikasi**.
- Namun tetap **terbuka untuk diperluas** lewat abstraksi, inheritance, atau komposisi.
- Caranya? **Gunakan polymorphism & dependency management**.
- OCP tidak bisa dicapai tanpa adanya **abstraksi**.

MENGAPA OCP PENTING?

Perubahan kebutuhan bisnis itu **pasti**.

Tanpa OCP, setiap kali ada fitur baru, developer harus **mengutak-atik kode lama** → rawan bug, mahal diuji ulang.

OCP membuat sistem lebih **stabil di inti**, namun **fleksibel di pinggiran**.

Cocok untuk sistem besar yang terus berkembang (ERP, LMS, sistem akademik, sistem pembayaran).



GEJALA PELANGGARAN OCP

Kode penuh if/else atau switch-case untuk mengatur banyak variasi perilaku.

Setiap kali ada fitur baru → kode lama harus diubah.

Risiko: perubahan pada kode lama bisa memicu bug di fitur yang sudah teruji.

SOLUSI & POLA YANG DIANJURKAN

Cara paling umum menerapkan OCP:

- **Polymorphism (inheritance)**
 - Base class/interface mendefinisikan kontrak.
 - Subclass bisa menambahkan perilaku baru tanpa menyentuh base class.
- **Strategy Pattern / Plug-in**
 - Pisahkan algoritme/variasi ke dalam strategi berbeda.
 - Kelas utama tidak perlu tahu detail implementasi.
- **Dependency Injection (DIP mendukung OCP)**
 - Kelas utama hanya tahu abstraksi. Implementasi dikirim dari luar.

CONTOH KASUS

✗ Versi yang melanggar OCP

Kalau ada metode pembayaran baru (misalnya Dana, OVO), kita harus **menambah if baru** di PaymentService.

Melanggar OCP: kode lama harus dimodifikasi.

```
public class PaymentService {  
    public void pay(String method) {  
        if (method.equals("paypal")) {  
            // PayPal API  
        } else if (method.equals("creditcard")) {  
            // CreditCard API  
        } else if (method.equals("gopay")) {  
            // GoPay API  
        }  
    }  
}
```

SOLUSI

PaymentService tidak berubah meskipun kita menambah DanaPayment atau OVOPayment.

Fitur baru cukup dibuat dengan **menambahkan kelas implementasi baru**.

➡ Inilah inti **OCP: extensible tanpa modifikasi inti**.

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

public class PayPalPayment implements PaymentMethod {
    public void pay(double amount) {
        System.out.println("Pay with PayPal: " + amount);
    }
}

public class CreditCardPayment implements PaymentMethod {
    public void pay(double amount) {
        System.out.println("Pay with Credit Card: " + amount);
    }
}

public class GoPayPayment implements PaymentMethod {
    public void pay(double amount) {
        System.out.println("Pay with GoPay: " + amount);
    }
}

public class PaymentService {
    public void process(PaymentMethod method, double amount) {
        method.pay(amount);
    }
}
```

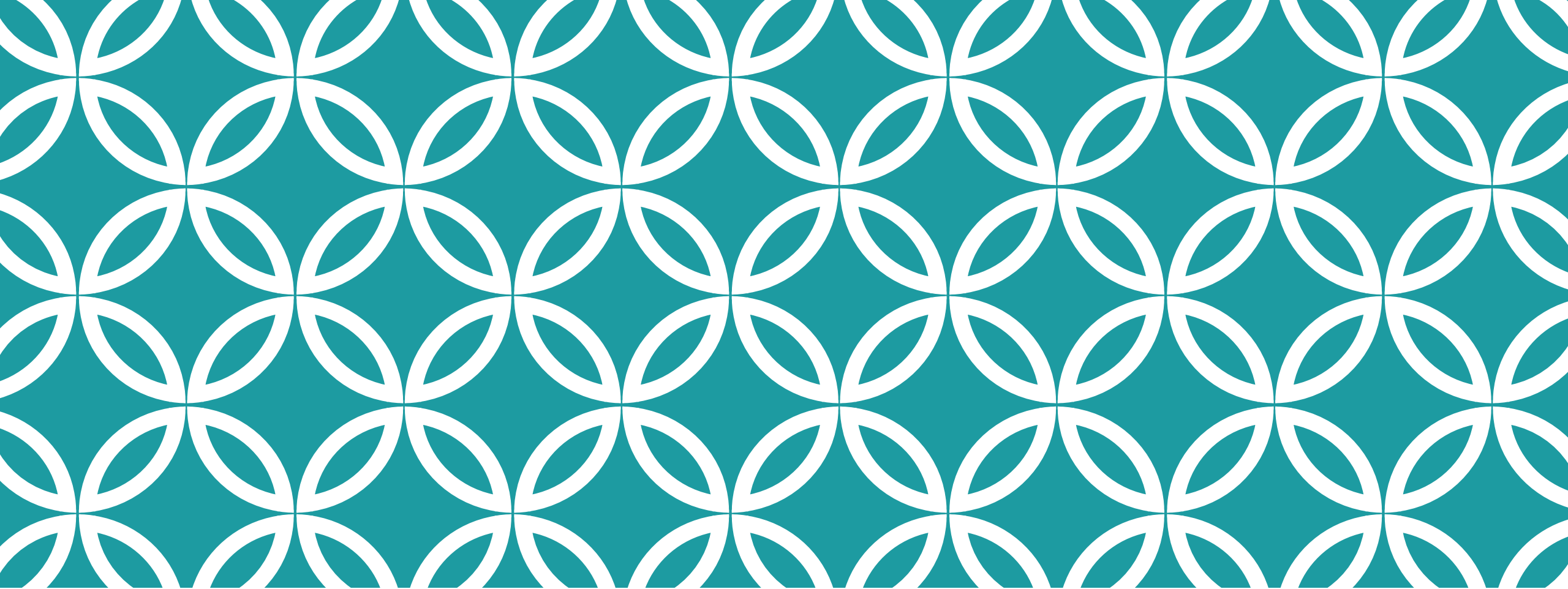
PANDUAN PRAKTIS

Checklist OCP:

- Apakah untuk menambah fitur baru Anda harus mengedit kode lama?
 - Jika ya → pelanggaran OCP.
- Apakah variasi perilaku di-hardcode di if/switch?
 - Jika ya → refactor ke polymorphism.
- Apakah kelas inti tetap stabil saat fitur baru ditambahkan?
 - Jika ya → sesuai OCP.

Resep Refactor:

- Identifikasi bagian kode yang sering berubah karena kebutuhan bisnis.
- Ekstrak ke interface/abstraksi.
- Pindahkan variasi implementasi ke kelas terpisah.
- Gunakan DI/factory untuk menghubungkan.



LSP (LISKOV SUBSTITUTION PRINCIPLE)



DEFINISI

Definisi (Barbara Liskov, 1987):

“If S is a subtype of T , then objects of type T may be replaced with objects of type S without altering any of the desirable properties of that program.”

Versi sederhananya:

“Subtypes must be substitutable for their base types.”

Makna: Jika kita punya kelas dasar Base, maka setiap subclass Derived harus bisa **digunakan di tempat Base** tanpa menyebabkan perilaku program rusak.

MENGAPA LSP PENTING?

OOP sangat bergantung pada **polymorphism**.

Jika subclass **merusak kontrak perilaku superclass**, maka polymorphism gagal.

LSP menjaga agar inheritance tidak sekadar “code reuse”, tapi juga **kontrak semantik** tetap terjaga.

Melanggar LSP → kode menjadi fragile, penuh pengecekan instanceof atau if, dan sulit diuji.

GEJALA PELANGGARAN LSP

Subclass **men-throw exception** atau meninggalkan method kosong pada method yang seharusnya valid di superclass.

Subclass **mengubah aturan** dari superclass (misalnya method `setWidth()` di `Rectangle` tapi subclass `Square` membuat perilakunya berbeda).

Banyak `if (obj instanceof Subclass)` → tanda bahwa subclass tidak benar-benar substitutable.



CONTOH KLASIK PELANGGARAN

✗ Square vs Rectangle

Square adalah Rectangle secara sintaks, tapi secara perilaku kontrak berubah.

Misalnya, jika fungsi test mengharapkan bisa mengatur $\text{width} \neq \text{height}$ di Rectangle, substitusi Square akan melanggar ekspektasi.

```
class Rectangle {  
    protected int width, height;  
    public void setWidth(int w) { width = w; }  
    public void setHeight(int h) { height = h; }  
    public int getArea() { return width * height; }  
}  
  
class Square extends Rectangle {  
    @Override  
    public void setWidth(int w) { width = height = w; }  
    @Override  
    public void setHeight(int h) { width = height = h; }  
}
```

SOLUSI & POLA UNTUK MENJAGA LSP

Redesign hierarki inheritance

- Jangan paksa hubungan “is-a” jika tidak benar-benar substitutable.
- Solusi: pisahkan Rectangle dan Square sebagai implementasi dari Shape abstrak.

Gunakan komposisi ketimbang inheritance jika perilaku tidak sejalan.

Gunakan kontrak/abstraksi yang lebih sempit (interface) agar subclass tidak dipaksa memenuhi hal yang tidak sesuai.



VERSI SESUAI LSP

Tidak ada kontradiksi perilaku.

Keduanya bisa digunakan sebagai Shape.

```
interface Shape {  
    int getArea();  
}  
  
class Rectangle implements Shape {  
    private int width, height;  
    public Rectangle(int w, int h) { width = w; height = h; }  
    public int getArea() { return width * height; }  
}  
  
class Square implements Shape {  
    private int side;  
    public Square(int s) { side = s; }  
    public int getArea() { return side * side; }  
}
```

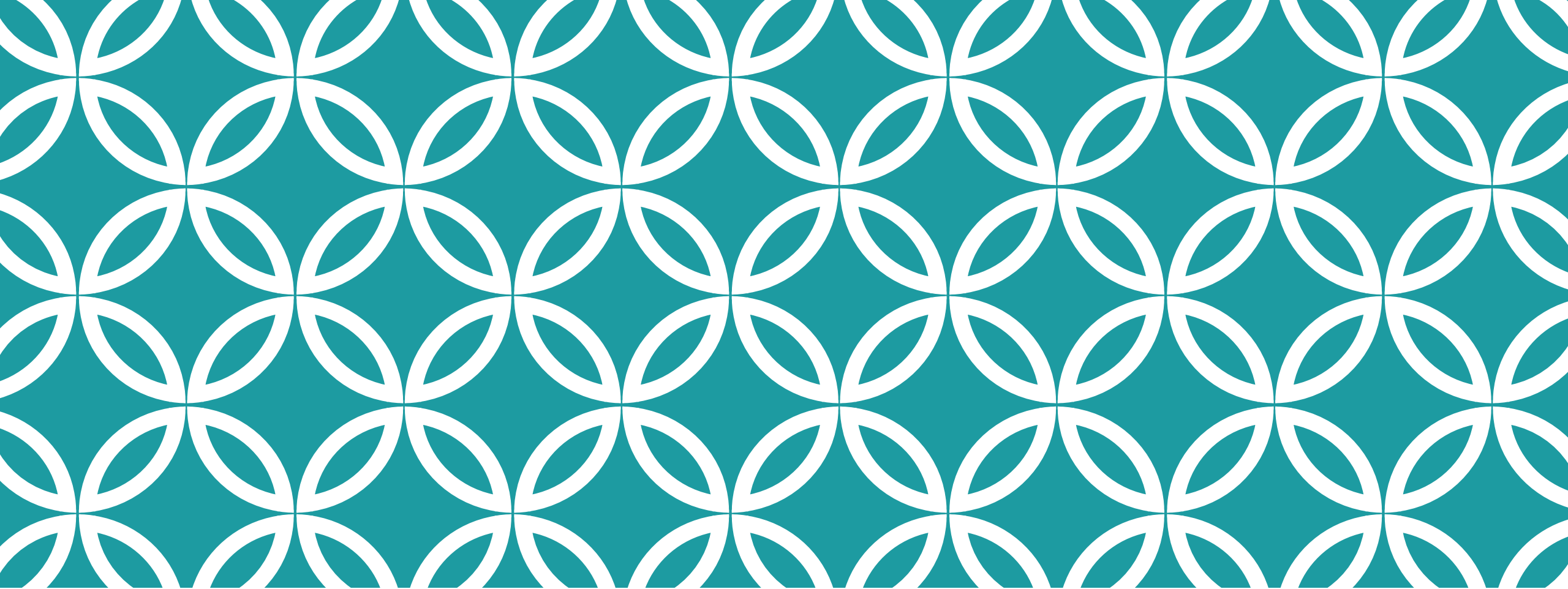
CONTOH KASUS RIIL

Sistem Akademik – Presensi Mahasiswa

- Kita punya kelas AttendanceMethod.
- ❌ Jika FaceRecognition mewarisi AttendanceMethod tapi method scanCard() dibiarkan kosong → melanggar LSP.
- ✅ Lebih baik AttendanceMethod dibuat interface sempit (verify(Student)) yang bisa diimplementasi oleh BarcodeScanner, RFIDScanner, FaceRecognition.

Sistem Pembayaran UKT

- ❌ Jika BankPayment superclass punya authorizeCard() tapi subclass EWalletPayment tidak relevan lalu melempar exception.
- ✅ Solusi: gunakan interface PaymentMethod dengan method pay(), bukan memaksa semua turunan punya method card-specific.



ISP (INTERFACE SEGREGATION PRINCIPLE)



DEFINISI

Definisi (Robert C. Martin / Uncle Bob):

- *“Clients should not be forced to depend on methods they do not use.”*

Makna:

- Jangan buat *interface gemuk* (fat interface) dengan banyak method yang tidak relevan untuk semua klien.
- Lebih baik pisahkan jadi **beberapa interface kecil dan spesifik** yang sesuai kebutuhan tiap klien.

Analogi sederhana:

Bayangkan ada **remote TV** dengan 50 tombol, tapi Anda hanya butuh tombol power dan volume. Remote jadi membingungkan. ISP ingin remote didesain sesuai kebutuhan → lebih ringkas, lebih fokus.

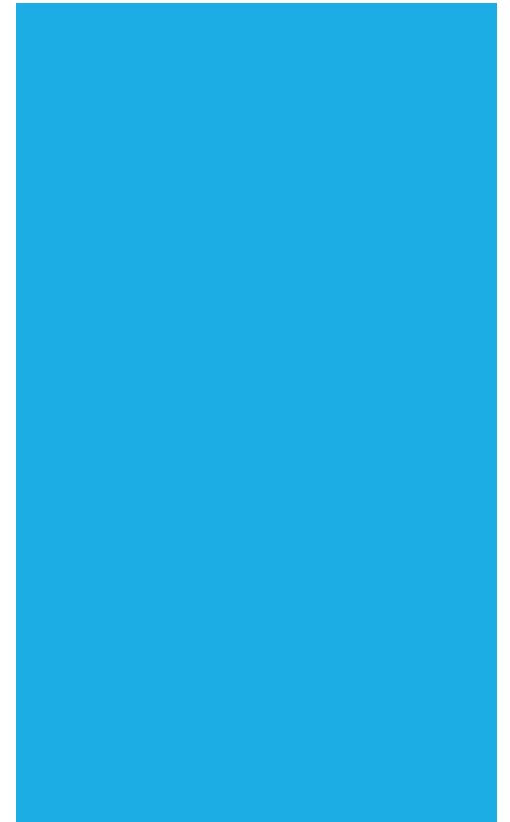
MENGAPA ISP PENTING

Tanpa ISP:

- Kelas klien dipaksa meng-implementasikan method yang tidak relevan.
- Jika interface berubah (misalnya ditambah method baru), semua implementasi ikut rusak.

Dengan ISP:

- Kode lebih modular, kohesif, dan mudah dipelihara.
- Perubahan hanya memengaruhi bagian yang benar-benar butuh.



GEJALA PELANGGARAN ISP

Interface memiliki **terlalu banyak method**.

Implementasi terpaksa menulis method kosong (throw new UnsupportedOperationException).

Banyak if (feature not supported) di dalam kelas.

Klien hanya memakai sebagian kecil dari interface besar.



CONTOH

✗ Versi yang Melanggar ISP

SimplePrinter hanya butuh print(), tapi dipaksa implement scan() dan fax() → pelanggaran ISP.

```
public interface IMachine {  
    void print(Document d);  
    void scan(Document d);  
    void fax(Document d);  
}  
  
public class SimplePrinter implements IMachine {  
    public void print(Document d) { /* ok */ }  
    public void scan(Document d) { /* tidak perlu */ }  
    public void fax(Document d) { /* tidak perlu */ }  
}
```

SOLUSI

SimplePrinter hanya bergantung pada IPrinter.

MultiFunctionPrinter bisa mengimplementasikan semua.

Modular, fleksibel, tidak ada beban berlebih.

```
public interface IPrinter {
    void print(Document d);
}

public interface IScanner {
    void scan(Document d);
}



public interface IFax {
    void fax(Document d);
}

public class SimplePrinter implements IPrinter {
    public void print(Document d) { /* ok */ }
}



public class MultiFunctionPrinter implements IPrinter, IScanner, IFax {
    public void print(Document d) { /* ... */ }
    public void scan(Document d) { /* ... */ }
    public void fax(Document d) { /* ... */ }
}
```

CONTOH RIIL

Sistem Akademik (Notifikasi Mahasiswa)

-  Interface INotification berisi: sendEmail(), sendSMS(), sendWhatsApp().
 - EmailNotification terpaksa mengisi method SMS/WA dengan kosong.
-  Solusi ISP: pisah jadi IEmailNotification, ISMSNotification, IWhatsAppNotification.
 - Setiap kelas implement hanya interface yang relevan.

Sistem Pembayaran UKT

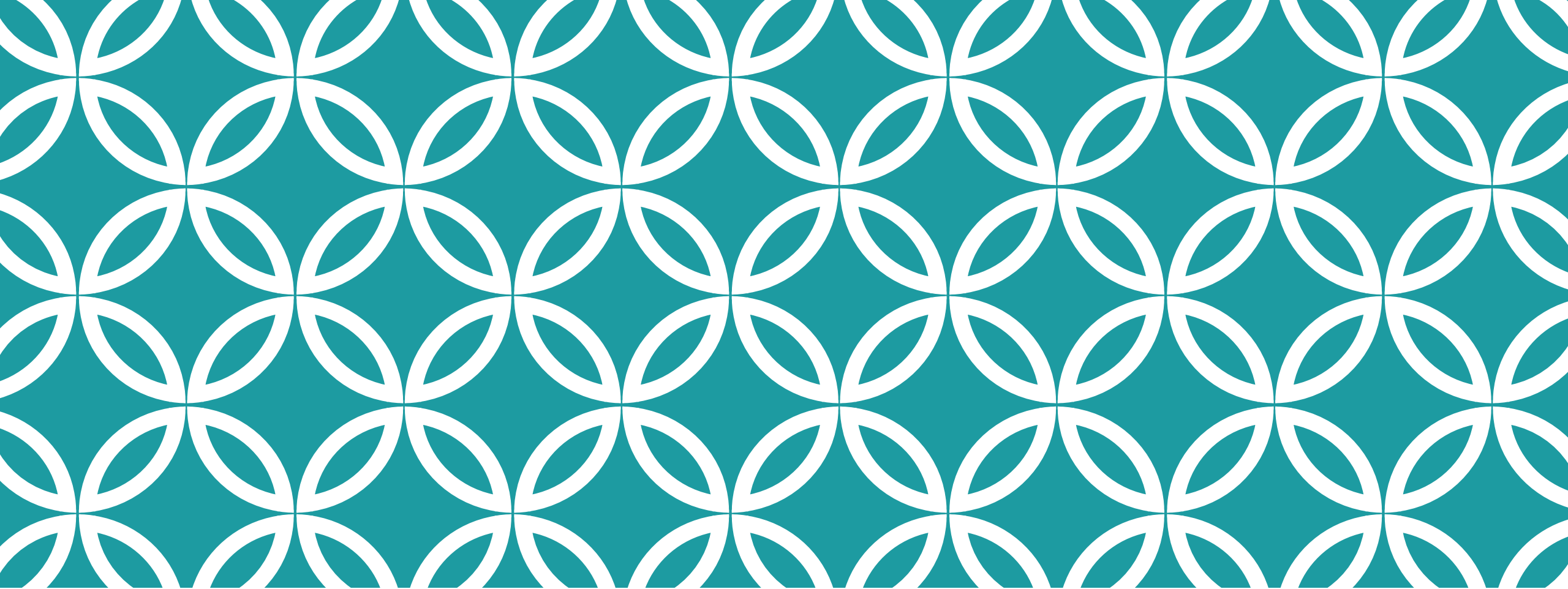
-  Interface IPayment dengan payWithBank(), payWithEWallet(), payWithCreditCard().
 - EWalletPayment harus implement payWithBank() meski tidak relevan.
-  Solusi ISP: buat IBankPayment, IEWalletPayment, ICardPayment.
 - Kelas implementasi hanya pilih interface sesuai metode pembayaran.

PANDUAN PRAKTIS

Apakah interface punya banyak method “kosong” di implementasi?
→ Langgar ISP.

Apakah semua klien benar-benar menggunakan semua method? Jika tidak → perlu pecah interface.

Gunakan interface kecil, fokus, dan sesuai aktor.



DIP (DEPENDENCY INVERSION PRINCIPLE)



DEFINISI

Definisi (Robert C. Martin):

“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.”

Makna:

- **High-level module** (aturan bisnis, use case) → tidak boleh bergantung langsung ke detail teknis (database, framework, library).
- **Keduanya harus bergantung pada abstraksi (interface/kontrak).**
- **Detail teknis** mengikuti arsitektur inti, bukan sebaliknya.

Analogi sederhana:

Bayangkan stopkontak (abstraksi) dan berbagai peralatan listrik (detail). Kita tidak mengubah rumah tiap kali membeli alat baru, cukup pastikan alat mengikuti standar stopkontak.

MENGAPA DIP PENTING?

Tanpa DIP: logika inti software rapuh karena terikat detail teknis (DB, UI, framework).

Dengan DIP:

- Logika bisnis tetap stabil meski DB/framework diganti.
- Aplikasi lebih mudah diuji (pakai mock/fake implementasi).
- Fleksibilitas tinggi untuk perubahan jangka panjang.

GEJALA PELANGGARAN DIP

Service langsung memanggil kode DB (`new MySQLRepository()`).

Logika bisnis tahu detail framework (SpringJpaRepository, EntityFramework).

Unit test sulit karena harus melibatkan DB nyata.

Modul “tingkat tinggi” ikut rusak jika modul “tingkat rendah” berubah.

SOLUSI & POLA YANG DIANJURKAN

Gunakan interface (abstraksi) sebagai kontrak.

High-level module hanya bergantung pada interface.

Low-level module (DB, API, UI) mengimplementasikan interface itu.

Dependency Injection / IoC Container untuk menghubungkan di runtime.



CONTOH KASUS

✗ Versi yang Melanggar DIP

OrderService (aturan bisnis) langsung bergantung ke detail MySQL.

Jika DB diganti Mongo/Postgres → OrderService ikut berubah.

```
public class OrderService {  
    private MySQLOrderRepository repo = new MySQLOrderRepository();  
  
    public void placeOrder(Order order) {  
        repo.save(order); // bergantung langsung ke MySQL  
    }  
}
```

SOLUSI

OrderService hanya tahu OrderRepository (abstraksi).

Implementasi detail bisa diganti tanpa menyentuh OrderService.

Mendukung unit testing → bisa pakai MockOrderRepository.

```
// Abstraksi
public interface OrderRepository {
    void save(Order order);
}

// High-level module
public class OrderService {
    private final OrderRepository repo;

    public OrderService(OrderRepository repo) {
        this.repo = repo;
    }



    public void placeOrder(Order order) {
        repo.save(order); // tidak peduli implementasi apa
    }
}

// Low-level implementations
public class MySQLOrderRepository implements OrderRepository {
    public void save(Order order) { /* simpan ke MySQL */ }
}



public class MongoOrderRepository implements OrderRepository {
    public void save(Order order) { /* simpan ke MongoDB */ }
}
```

KASUS RIIL

Sistem Akademik (KRS)

-  KRSService langsung bergantung ke SQLKRSRepository.
-  Dengan DIP: KRSService bergantung ke IKRSRepository.
 - Bisa pakai MySQL, PostgreSQL, bahkan InMemoryRepository untuk testing.

Sistem Pembayaran UKT

-  PaymentService langsung pakai BNIGateway.
-  Dengan DIP: PaymentService bergantung ke IPaymentGateway.
 - Bisa pilih BNI, Mandiri, atau E-Wallet tanpa ubah PaymentService.

HUBUNGAN DIP ↔ PRINSIP & ARSITEKTUR

DIP adalah inti Clean Architecture.

Dalam Clean Architecture ada *Dependency Rule*:

- Arah dependency harus mengalir **ke dalam (toward business rules)**.
- Lapisan dalam (Entities, Use Cases) → independen dari lapisan luar (DB, UI, Frameworks).

Dengan DIP:

- Use Case tidak tahu detail DB.
- Entities tidak tahu UI.
- Framework hanya “detail” yang bisa diganti.



PANDUAN PRAKTIS

Apakah service Anda langsung new repository/DB class? → Langgar DIP.

Apakah logika bisnis tahu framework spesifik (Spring, Hibernate)? → Langgar DIP.

Apakah modul inti bisa diuji tanpa DB/framework nyata? → Jika ya, Anda sudah mengikuti DIP.

Gunakan **abstraksi (interface)**, bukan implementasi detail.

KESIMPULAN

S → Single Responsibility: satu alasan berubah.

O → Open/Closed: terbuka untuk ekstensi, tertutup untuk modifikasi.

L → Liskov Substitution: subclass harus konsisten dengan superclass.

I → Interface Segregation: interface kecil, sesuai kebutuhan.

D → Dependency Inversion: bergantung pada abstraksi, bukan detail.