Introduzione

I principi **SOLID** sono un insieme di cinque principi di progettazione orientata agli oggetti che aiutano il programmatore a gestire le dipendenze all'interno di un sistema. L'obiettivo è creare architetture software con accoppiamento lasco e alta coesione.

Acronimo SOLID:

Lettera	Principio	Definizione
S	Single Responsibility Principle	A class should have one, and only one, reason to change
O	Open-Closed Principle	You should be able to extend a classes behavior, without modifying it
L	Liskov Substitution Principle	Derived classes must be substitutable for their base classes
I	Interface Segregation Principle	Make fine grained interfaces that are client specific
D	Dependency Inversion Principle	Depend on abstractions, not on concretions

Importanza relativa:
I primi tre principi (**S.O.L.**) sono fondamentali. Il rispetto di questi garantisce molto probabilmente anche la conformità agli altri due (**I** e **D**). Tutte le componenti devono avere un accoppiamento il più lasco possibile.

---

1. SINGLE RESPONSIBILITY PRINCIPLE (SRP)

1.1 Definizione

> **A class should have one, and only one, reason to change.**

Una classe deve avere una ed una sola singola responsabilità; una classe deve avere un solo motivo per cambiare.

1.2 Concetto di Responsabilità

Responsabilità = Motivo di cambiamento

- Una responsabilità è un asse di cambiamento solo se i cambiamenti effettivamente avvengono
- Il contesto dell'applicazione è importante
- **Needless complexity:** Non separare prematuramente responsabilità che potrebbero non cambiare mai

1.3 Coesione (Cohesion)

Il SRP è anche conosciuto come **coesione funzionale:**

- Relazione funzionale tra gli elementi di un modulo
- Un modulo dovrebbe avere solo un motivo per cambiare

Responsabilità accoppiate:

- I cambiamenti a una responsabilità possono compromettere o inibire l'abilità della classe di soddisfare le altre
- Design fragile che si rompe in modi inaspettati
- Ricompilazione, test, deploy non necessari

1.4 Esempio: Interface Modem

```
public interface Modem {
    public void dial(String pno);
    public void hangup();
    public void send(char c);
    public char recv();
}
```

Analisi delle responsabilità:

1. **Connection management:** `dial()`, `hangup()`
2. **Data communication:** `send()`, `recv()`

Domanda: Queste due responsabilità dovrebbero essere separate?

Risposta: Dipende da come l'applicazione cambia:

- Se la gestione della connessione e la comunicazione cambiano per ragioni diverse → separare
- Se cambiano sempre insieme → mantenerle unite

1.5 Soluzione: Separazione Responsabilità

```
// Interfaccia per gestione connessione
public interface Connection {
```

```
    void dial(String pno);
    void hangup();
}

// Interfaccia per comunicazione dati
public interface DataChannel {
    void send(char c);
    char recv();
}

// Implementazione che mantiene entrambe
public class ModemImplementation implements Connection, DataChannel {
    // Implementazione dei metodi
}
```

Vantaggi:

- Le responsabilità sono separate nelle interfacce
- I client dipendono solo dall'interfaccia che utilizzano
- Evita l'accoppiamento tra client che usano funzionalità diverse
- Le responsabilità sono ancora accoppiate in `ModemImplementation`, ma i client non devono preoccuparsene

1.6 Quando Applicare SRP

Bisogna analizzare tutto il contesto, non solo la singola classe:

- Maggiori sono le responsabilità di una classe, maggiore sarà la probabilità che ci sarà bisogno di doverla cambiare
- Si punta ad avere oggetti piccoli e concisi, facili anche da testare
- Evitare classi che fanno troppo (God objects)

Esempio negativo: Una classe che:

- Fa calcoli aritmetici
- Disegna nella GUI
- Si connette al database per salvare dati

1.7 Vantaggi

- Facilita il testing (classi più piccole)
- Riduce l'accoppiamento
- Migliora la leggibilità
- Facilita il riutilizzo
- Riduce gli effetti domino delle modifiche

---

2. OPEN-CLOSED PRINCIPLE (OCP)

2.1 Definizione

> **You should be able to extend a classes behavior, without modifying it.**

Le entità software (classi, moduli, funzioni) dovrebbero essere:

- **Aperte per l'estensione:** Si può aggiungere nuovo comportamento
- **Chiuse per la modifica:** Il codice esistente non cambia

2.2 Fondamento: Astrazione

Software entities should be open for extension, but closed for modification.

L'astrazione è la chiave:

- I tipi astratti sono la parte fissa
- Le classi derivate sono i punti di estensione
- Si utilizza polimorfismo e interfacce

2.3 Esempio Violazione OCP

```
public static void drawAll(Shape[] shapes) {
    for (Shape shape : shapes) {
        switch (shape.shapeType) {
            case Square:
                ((Square) shape).drawSquare();
                break;
            case Circle:
                ((Circle) shape).drawCircle();
                break;
        }
    }
}
```

Problemi:

- Non conforme al principio open-closed
- Non può essere chiuso rispetto a nuovi tipi di shape
- Per estendere la funzione, devo modificarla
- **Cascade of changes:** Un cambiamento richiede modifiche multiple

2.4 Soluzione Conforme OCP

```
// Astrazione
public abstract class Shape {
```

```
    public abstract void draw();
}

// Implementazioni concrete
public class Square extends Shape {
    public void draw() {
        // disegna quadrato
    }
}

public class Circle extends Shape {
    public void draw() {
        // disegna cerchio
    }
}

// Funzione conforme OCP
public static void drawAll(Shape[] shapes) {
    for (Shape shape : shapes) {
        shape.draw();
    }
}
```

Vantaggi:

- Per estendere il comportamento, basta aggiungere una nuova derivata di `Shape`
- Nessuna modifica al codice esistente
- I programmi conformi a OCP non subiscono "cascade of changes"

2.5 Esempio Pratico: Calcolo Aree

Versione NON conforme:

```
class ShapeManager {
    public double calculate(Object shape) {
        double area = 0;
        if (shape instanceof Rectangle) {
            Rectangle rectangle = (Rectangle) shape;
            area = rectangle.width * rectangle.height;
        } else if (shape instanceof Circle) {
            Circle circle = (Circle) shape;
            area = circle.radius * circle.radius * Math.PI;
        }
        return area;
    }
}
```

Versione conforme OCP:

```java
// Interfaccia
interface Area {
    double getArea();
}

// Implementazioni
class Rectangle implements Area {
    private double width;
    private double height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public double getArea() {
        return width * height;
    }
}

class Circle implements Area {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double getArea() {
        return radius * radius * Math.PI;
    }
}

// Manager conforme OCP
class ShapeManager {
    public double calculate(Area shape) {
        return shape.getArea();
    }
}
```

2.6 Limiti e Considerazioni

Nessun programma può essere 100% chiuso:

- La chiusura deve essere strategica
- Identificare gli assi di cambiamento più probabili

Closure può essere ottenuta:

1. **Tramite astrazione:** Interfacce e polimorfismo
2. **Data-driven:** Informazioni configurate in strutture esterne

Esempio limite: Cosa succede se vogliamo disegnare le shape in un ordine specifico che dipende dal tipo? → Potrebbe essere necessario un approccio data-driven.

2.7 Convenzioni derivate da OCP

Make all member variables private:

- Quando le variabili membro cambiano, ogni funzione che dipende da esse deve essere cambiata
- **Encapsulation**

No global variables (ever):

- Nessun modulo che dipende da una variabile globale può essere chiuso rispetto a qualunque altro modulo che potrebbe scrivere quella variabile
- Pochissimi casi possono disobbedire (es. `cin`, `cout`)

RTTI is dangerous:

- L'esempio di Shape mostra il modo sbagliato di usare RTTI
- Ma ci sono anche casi buoni...

---

3. LISKOV SUBSTITUTION PRINCIPLE (LSP)

3.1 Definizione

> **Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.**

Le classi derivate devono essere sostituibili alle loro classi base.

Versione semplificata (caso speciale del LSP reale):

- Le funzioni che usano puntatori o riferimenti a classi base devono poter usare oggetti di classi derivate senza saperlo
- Violando questo principio si viola anche OCP

3.2 Fondamento: Astrazione e Polimorfismo

Alla base di OOD e OCP:

- Quali sono le caratteristiche delle migliori gerarchie di ereditarietà?
- Quali sono le trappole?

Domanda chiave: Cosa rende una buona gerarchia di ereditarietà?

3.3 Esempio Classico: Square-Rectangle

```
public class Rectangle {
    private double width;
    private double height;

    public void setWidth(double w) {
        width = w;
    }

    public void setHeight(double h) {
        height = h;
    }

    public double getWidth() { return width; }
    public double getHeight() { return height; }
}

public class Square extends Rectangle {
    @Override
    public void setWidth(double w) {
        super.setWidth(w);
        super.setHeight(w); // Mantiene lati uguali
    }

    @Override
    public void setHeight(double h) {
        super.setHeight(h);
        super.setWidth(h); // Mantiene lati uguali
    }
}
```

Test che viola LSP:

```
public void f(Rectangle r) {
    r.setWidth(42);
}

@Test
public void testF() {
    Rectangle r = new Square();
    r.setHeight(15);
    f(r);
```

```
    // Questo test NON passerà!!!
    assertEquals(15, r.getHeight());
}
```

Problema:

- Se passiamo un riferimento a un oggetto `Square` in questa funzione, l'altezza cambierà anche se non dovrebbe
- Questa è una chiara violazione di LSP
- La funzione `f` non funziona per le derivate dei suoi argomenti

3.4 Design by Contract

Il problema è il comportamento estrinseco pubblico:

- Il comportamento da cui i client dipendono
- La relazione tra `Square` e `Rectangle` NON è una relazione IS-A in OOD

Regola per il contratto:

> ...when redefining a routine [in a derivative], you may only replace its precondition by a weaker one, and its postcondition by a stronger one.

Postcondizioni Rectangle.setWidth(double w):

```
assert((width == w) && (height == old.height));
```

Questa postcondizione è violata in `Square`.

3.5 Validazione del Modello

Un modello, visto in isolamento, non può essere validato in modo significativo.

La validità di un modello può essere espressa solo in termini dei suoi client.

```
public void f(Rectangle r) {
    r.setWidth(42);
}
```

Se questa funzione è tipica del modo in cui i client usano `Rectangle`, allora `Square` non è sostituibile e viola LSP.

3.6 Esempio Pratico Corretto

Problema: Rectangle con lati modificabili vs Square con lati uguali

Soluzione: Non usare ereditarietà

```
interface Shape {
    double getArea();
}

class Rectangle implements Shape {
    private double width;
    private double height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    public void setWidth(double w) { width = w; }
    public void setHeight(double h) { height = h; }

    public double getArea() {
        return width * height;
    }
}

class Square implements Shape {
    private double side;

    public Square(double side) {
        this.side = side;
    }

    public void setSide(double s) { side = s; }

    public double getArea() {
        return side * side;
    }
}
```

3.7 Precondizioni e Postcondizioni

In una classe derivata:

- Le **precondizioni** devono essere più deboli (o uguali) di quelle della classe base
- Le **postcondizioni** devono essere più forti (o uguali) di quelle della classe base

Esempio:

- Base: `setWidth(w)` → postcondizione: `width == w && height == old.height`
- Derivata Square: `setWidth(w)` → postcondizione violata perché modifica anche `height`

3.8 Implicazioni

Violazione LSP implica violazione OCP:

- Una funzione che usa un puntatore o riferimento a una classe base, ma deve conoscere tutte le derivate di quella classe
- Non può essere chiusa rispetto a nuove derivate

---

4. INTERFACE SEGREGATION PRINCIPLE (ISP)

4.1 Definizione

> **Clients should not be forced to depend upon interfaces that they do not use.**

I client non dovrebbero essere forzati a dipendere da interfacce che non usano.

Obiettivo: Creare interfacce con pochi metodi e molto specifiche.

4.2 Problema: Fat Interfaces

Ridurre l'accoppiamento significa dipendere da interfacce, non da implementazioni:

- Il rischio è dipendere da interfacce "grasse" o "inquinate"
- Le fat interfaces non sono coese
- I metodi possono essere raggruppati in gruppi di funzioni
- I client devono vedere solo la parte che interessa loro

4.3 Esempio: TimedDoor

Scenario iniziale:

```
interface Door {
    void lock();
    void unlock();
    boolean isDoorOpen();
}

class TimedDoor implements Door {
    // Necessita di suonare un allarme dopo timeout
    // Ma come?
}
```

Prima soluzione (ERRATA):

```
interface Door {
    void lock();
    void unlock();
    boolean isDoorOpen();
    void timeOut(); // ← Inquina l'interfaccia!
}

interface TimerClient {
    void timeOut();
}

class TimedDoor implements Door, TimerClient {
    // Implementazione
}
```

Problemi:

- `Door` ora dipende da `TimerClient`
- Non tutte le varietà di `Door` necessitano timing
- Le applicazioni che usano quelle derivate devono importare la definizione di `TimerClient`, anche se non è usata
- L'interfaccia di `Door` è stata inquinata con un'interfaccia che non richiede
- Ogni volta che viene aggiunta una nuova interfaccia alla classe base, quella deve essere implementata nelle classi derivate
- Implementazioni di default violano il Liskov Substitution Principle (LSP)

4.4 Soluzione 1: Separation by Delegation (Adapter)

Object form of the Adapter design pattern:

```
interface Door {
    void lock();
    void unlock();
    boolean isDoorOpen();
}

interface TimerClient {
    void timeOut();
}

class TimedDoor implements Door {
    // Implementazione Door
}

class DoorTimerAdapter implements TimerClient {
    private TimedDoor door;
```

```
    public DoorTimerAdapter(TimedDoor door) {
        this.door = door;
    }

    public void timeOut() {
        door.soundAlarm();
    }
}
```

Vantaggi:

- `DoorTimerAdapter` traduce una `Door` in un `TimerClient`
- I client di `Door` e `TimerClient` non sono più accoppiati

4.5 Soluzione 2: Separation through Multiple Inheritance

Class form of the Adapter design pattern:

```
interface Door {
    void lock();
    void unlock();
    boolean isDoorOpen();
}

interface TimerClient {
    void timeOut();
}

class TimedDoor implements Door, TimerClient {
    public void lock() { /* ... */ }
    public void unlock() { /* ... */ }
    public boolean isDoorOpen() { /* ... */ }

    public void timeOut() {
        soundAlarm();
    }

    private void soundAlarm() { /* ... */ }
}
```

Vantaggi:

- Il client può usare lo stesso oggetto attraverso interfacce diverse e separate
- Possibile solo quando l'ereditarietà multipla è supportata
- Meno tipi usati rispetto alla soluzione con delegation

4.6 Esempio Pratico: Worker

Problema:

```
interface Worker {
    void work();
    void sleep();
}

class Human implements Worker {
    public void work() { /* ... */ }
    public void sleep() { /* ... */ }
}

class Robot implements Worker {
    public void work() { /* ... */ }
    public void sleep() {
        // Robot non dorme! Violazione ISP
        throw new UnsupportedOperationException();
    }
}
```

Soluzione:

```
interface Workable {
    void work();
}

interface Sleepable {
    void sleep();
}

class Human implements Workable, Sleepable {
    public void work() { /* ... */ }
    public void sleep() { /* ... */ }
}

class Robot implements Workable {
    public void work() { /* ... */ }
}
```

4.7 Vantaggi ISP

- Interfacce coese e specifiche per client
- Riduce l'accoppiamento tra client diversi
- Facilita l'evoluzione del sistema
- Evita ricompilazioni non necessarie

---

5. DEPENDENCY INVERSION PRINCIPLE (DIP)

5.1 Definizione

> **High level modules should not depend upon low level modules. Both should depend upon abstractions.**
>
> **Abstractions should not depend upon details. Details should depend upon abstractions.**

Dipendere dalle astrazioni, non dalle concretizzazioni.

5.2 Motivazione

I moduli ad alto livello contengono le decisioni di policy importanti:

- Sono questi moduli che vogliamo riutilizzare
- Template method design pattern
- In applicazioni a livelli, ogni livello dovrebbe esporre un adeguato livello di astrazione (interfaccia)

Implementazione naive: Può forzare dipendenze sbagliate tra moduli.

5.3 Esempio Problema: Copy

Struttura tradizionale (SBAGLIATA):

Problema:

- Il livello Policy è sensibile ai cambiamenti fino al livello Utility
- Dipendenze transitivas non volute

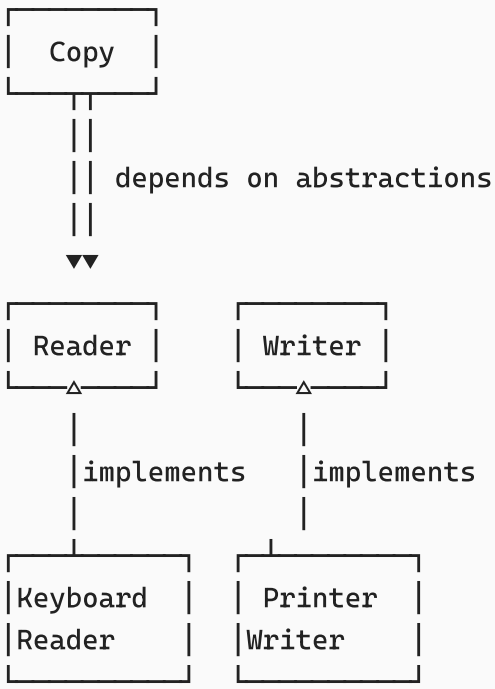
Esempio concreto:

```
class KeyboardReader {
    public char read() { /* ... */ }
}

class PrinterWriter {
    public void write(char c) { /* ... */ }
}

class Copy {
    private KeyboardReader reader;
    private PrinterWriter writer;

    public void copy() {
        char c;
        while ((c = reader.read()) != EOF) {
            writer.write(c);
        }
    }
}
```

Problema: `Copy` dipende da dettagli concreti, non può essere riutilizzata.

5.4 Soluzione: Dependency Inversion

Struttura corretta:



Implementazione:

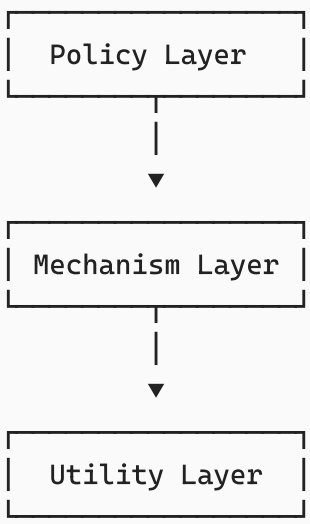
```
// Astrazioni
interface Reader {
    char read();
}

interface Writer {
    void write(char c);
}

// Policy ad alto livello
class Copy {
    private Reader reader;
    private Writer writer;

    public Copy(Reader r, Writer w) {
        this.reader = r;
        this.writer = w;
    }

    public void copy() {
        char c;
        while ((c = reader.read()) != EOF) {
            writer.write(c);
        }
    }
}

// Dettagli concreti
class KeyboardReader implements Reader {
    public char read() { /* ... */ }
}

class PrinterWriter implements Writer {
    public void write(char c) { /* ... */ }
}
```

Risultato:

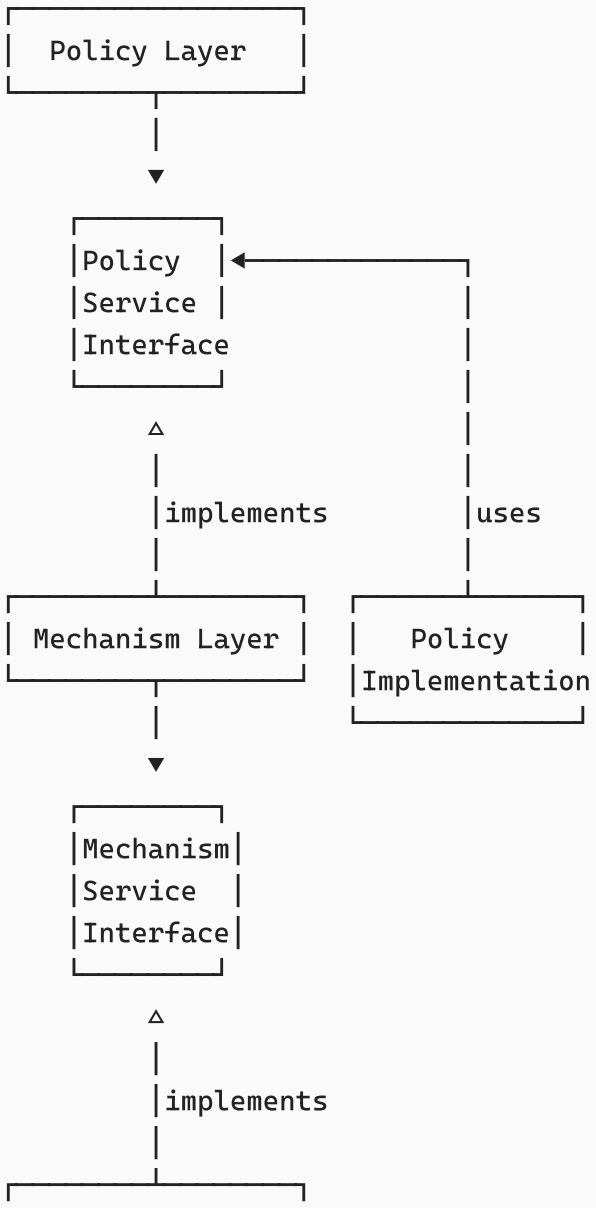
- Abbiamo eseguito **dependency inversion**
- Le dipendenze sono state invertite
- La classe `Copy` dipende da astrazioni
- I reader e writer concreti dipendono dalle stesse astrazioni
- Ora possiamo riutilizzare `Copy` indipendentemente da `KeyboardReader` e `PrinterWriter`

5.5 Architettura a Livelli con DIP

Prima (SENZA DIP):



Dopo (CON DIP):



```
|  Utility Layer  |
└─────────────────┘
```

Vantaggi:

- Ogni livello superiore usa il livello inferiore attraverso un'interfaccia astratta
- Nessun livello dipende da un altro livello
- Policy è indipendente dai dettagli implementativi

5.6 Esempio Pratico

Senza DIP (accoppiamento forte):

```
class LightBulb {
    public void turnOn() { /* ... */ }
    public void turnOff() { /* ... */ }
}

class Switch {
    private LightBulb bulb;

    public Switch(LightBulb bulb) {
        this.bulb = bulb;
    }

    public void operate() {
        // dipende da LightBulb concreta
        if (/* switch is on */) {
            bulb.turnOn();
        } else {
            bulb.turnOff();
        }
    }
}
```

Con DIP (accoppiamento lasco):

```
// Astrazione
interface Switchable {
    void turnOn();
    void turnOff();
}

// Dettagli
class LightBulb implements Switchable {
    public void turnOn() { /* ... */ }
    public void turnOff() { /* ... */ }
}
```

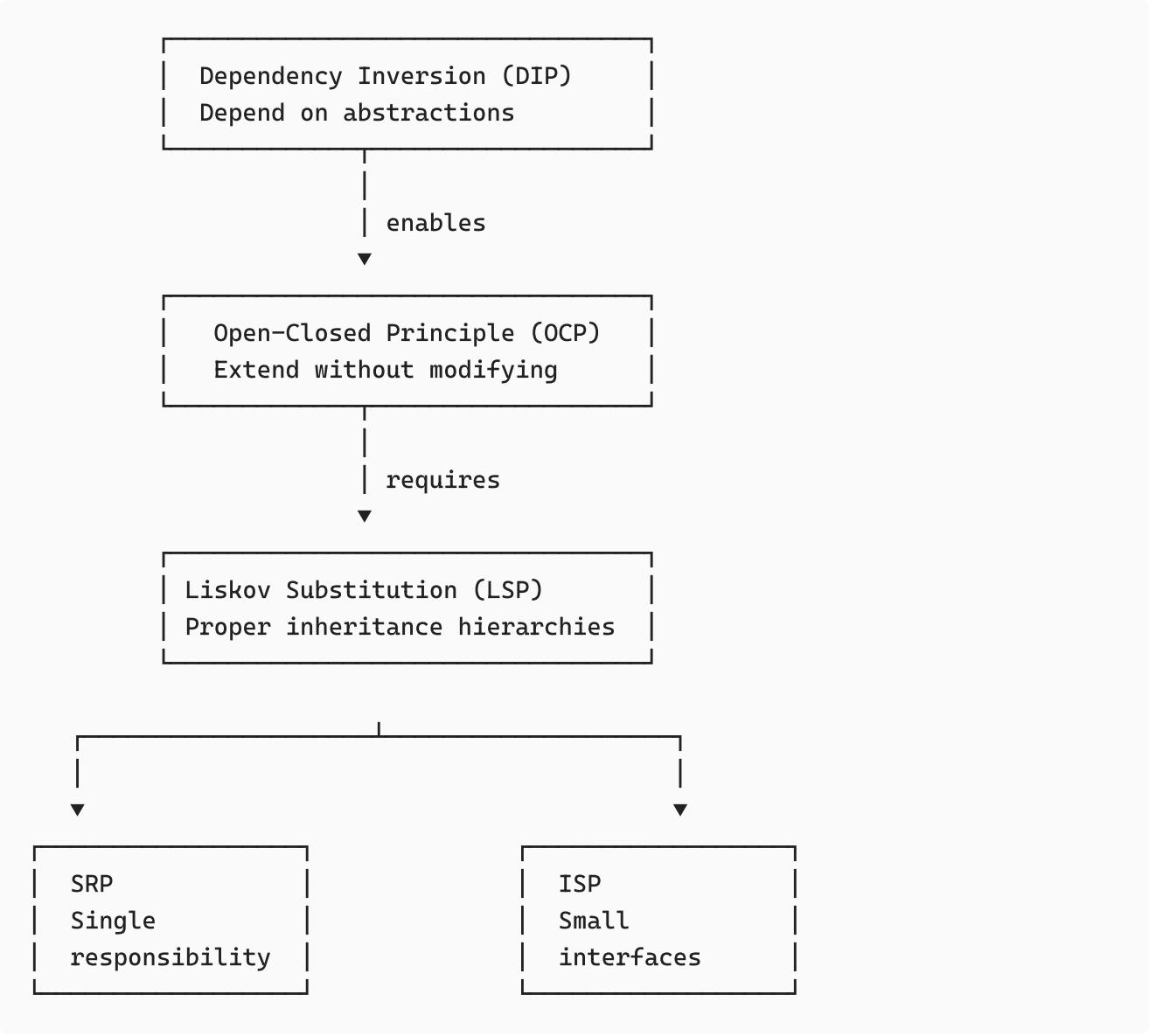
```
class Fan implements Switchable {
    public void turnOn() { /* ... */ }
    public void turnOff() { /* ... */ }
}

// Policy
class Switch {
    private Switchable device;

    public Switch(Switchable device) {
        this.device = device;
    }

    public void operate() {
        if (/* switch is on */) {
            device.turnOn();
        } else {
            device.turnOff();
        }
    }
}
```

Tabella Riepilogativa SOLID

Principio	Definizione	Obiettivo	Violazione implica
SRP	Una classe = una responsabilità	Coesione alta	Modifiche a cascata, testing difficile
OCP	Aperto estensione, chiuso modifica	Estensibilità senza modifica	Modifiche al codice esistente
LSP	Derivate sostituibili alle basi	Polimorfismo corretto	Violazione OCP, comportamento inatteso
ISP	Interfacce client-specific	Dipendenze minime	Accoppiamento non necessario
DIP	Dipendere da astrazioni	Inversione delle dipendenze	Accoppiamento a dettagli concreti

---

Interrelazioni tra Principi



Flusso logico:

1. **DIP** fornisce le basi (astrazione)
2. **OCP** è abilitato da DIP (estensione tramite astrazione)
3. **LSP** garantisce che OCP funzioni (sostituibilità)
4. **SRP** e **ISP** supportano tutti i precedenti (coesione e interfacce specifiche)

Best Practices e Linee Guida

Applicazione dei Principi

SRP:

- Identificare responsabilità multiple
- Considerare il contesto dell'applicazione
- Non separare prematuramente (YAGNI - You Aren't Gonna Need It)

OCP:

- Identificare assi di cambiamento probabili
- Usare astrazione per punti di estensione
- Accettare che 100% closure è impossibile

LSP:

- Design by contract
- Testare sostituibilità
- Precondizioni più deboli, postcondizioni più forti

ISP:

- Role interfaces
- Interfacce client-specific
- Preferire molte interfacce piccole a una grande

DIP:

- Depend on abstractions
- Dependency Injection
- Layered architecture con interfacce

Codice Smell che Violano SOLID

Smell	Principio Violato	Descrizione
God Class	SRP	Classe con troppe responsabilità
Switch Statements on Type	OCP, LSP	Controllo del tipo invece di polimorfismo
Refused Bequest	LSP	Sottoclasse non usa metodi ereditati
Feature Envy	SRP	Metodo usa più metodi di un'altra classe
Fat Interface	ISP	Interfaccia con troppi metodi
Tight Coupling	DIP	Dipendenza diretta da classi concrete

Riferimenti

Libri:

- Robert C. Martin, "Agile Principles, Patterns, and Practices in C#", 2006, Prentice Hall
  - Chap. 8: The Single-Responsibility Principle (SRP)
  - Chap. 9: The Open/Closed Principle (OCP)

- Chap. 10: The Liskov Substitution Principle (LSP)
- Chap. 11: The Dependency-Inversion Principle (DIP)
- Chap. 12: The Interface Segregation Principle (ISP)

Online:

- The Principles of OOD: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
- Single-Responsibility Principle done right: http://blog.rcard.in/solid/srp/programming/2017/12/31/srp-done-right.html
- The Secret Life of Objects: Inheritance: http://blog.rcard.in/design/programming/oop/fp/2018/07/27/the-secret-life-of-objectspart-2.html