
INDICE RAPIDO

---

1. DEFINIZIONI UFFICIALI E FORMULAZIONI D'ESAME {#definizioni}

Scopo generale dei principi SOLID

Citazione di Robert Martin (2000):

> "Creare codice comprensibile, leggibile e testabile su cui molti sviluppatori possano lavorare in modo collaborativo, aiutando a gestire le dipendenze, in modo flessibile e robusto."

Obiettivi architetturali perseguiti:

- Modularità
- Affidabilità
- Riusabilità
- Semplicità
- Basso accoppiamento
- Flessibilità
- Robustezza
- Coesione

S - Single Responsibility Principle (SRP)

Definizione ufficiale:

> "Una classe dovrebbe avere una, e una sola, ragione per cambiare."

Identificazioni equivalenti:

- Noto anche come **COESIONE** ⚠️ (questo è fondamentale per le crocette)
- "Un modulo deve avere correlazione funzionale degli elementi"
- "Un modulo dovrebbe avere un solo motivo per cambiare"

Concetti chiave:

- Le responsabilità sono **accoppiate**: modifiche a una responsabilità possono compromettere altre
- Necessario **analizzare tutto il contesto**, non solo la singola classe isolata
- Maggiori responsabilità = maggiore probabilità di cambiamenti
- Le modifiche a una responsabilità possono inibire altre funzionalità

Quali qualità architetturali persegue SRP:

- Modularità
- Affidabilità
- Riusabilità
- Semplicità
- Basso accoppiamento

O - Open-Closed Principle (OCP)

Definizione ufficiale:

> "Le entità software devono essere aperte all'estensione, ma chiuse alle modifiche."

Interpretazione operativa:

- Si deve poter **estendere il comportamento aggiungendo nuovo codice**
- **NON modificando codice vecchio**
- L'**astrazione è la chiave** (interfacce/classi astratte)

Euristiche di implementazione (dettami):

1. Rendere **private le variabili membro**
2. **Evitare variabili globali**
3. Fare **attenzione all'uso di RTTI** (Run Time Type Identification)
4. Definire **metodi virtuali** per prevedere estensibilità

Nota critica per crocette:

- È **possibile** che un gruppo di classi sia Open-Close anche usando variabili globali, **nei modi e termini permessi dal linguaggio** ✓
- È **impossibile** ottenere chiusura completa del software intero
- Servono **scelte strategiche** per garantire modularità, flessibilità, incapsulamento

Quali qualità architetturali persegue OCP:

- Modularità
- Flessibilità
- Incapsulamento

Violazioni tipiche:

- Cascate di if/switch su tipi concreti
- Modifica diretta di classi esistenti per nuove funzionalità

L - Liskov Substitution Principle (LSP)

Definizione ufficiale:

> "Le classi derivate devono essere sostituibili con le loro classi base."

Formulazione alternativa (più tecnica):

> "Le funzioni che utilizzano puntatori o riferimenti a classi base devono essere in grado di utilizzare oggetti di classi derivate senza saperlo."

Concetti fondamentali:

- È alla **base della programmazione ad oggetti**
- Collegato a **Design by Contract** ⚠️ (questo è esplicito nei quiz)
- Violare LSP implica violare OCP

Regole sulle pre/postcondizioni:

- **Precondizioni** nelle classi derivate: **uguali o più deboli** (fanno meno assunzioni)
- **Postcondizioni** nelle classi derivate: **uguali o più forti** rispetto alla classe base

Quali qualità architetturali persegue LSP:

- Modularità
- Flessibilità
- Coesione

Implicazioni pratiche:

- Non sovrascrivere le pre- e post-condizioni della classe base
- Evitare ridefinizione di comportamenti pubblici estrinseci
- Attenersi alle specifiche date dal codice

I - Interface Segregation Principle (ISP)

Definizione ufficiale:

> "Creare interfacce a grana fine che siano specifiche per il client."

Formulazione negativa:

> "I client non devono essere forzati ad implementare un'interfaccia che non usano."

Concetti chiave:

- Riduce l'**accoppiamento**
- Interfacce devono essere **coese** e **specializzate**
- Evitare "fat interfaces" con metodi non utilizzati da tutti i client

Strategie di implementazione:

- **Ereditarietà multipla** (se disponibile nel linguaggio)
- **Pattern Adapter** per separare interfacce
- Delegazione di funzionalità

Quali qualità architetturali persegue ISP:

- Modularità
- Flessibilità
- Basso accoppiamento
- Coesione
- Riusabilità

D - Dependency Inversion Principle (DIP)

Definizione ufficiale:

> "Si dovrebbe dipendere dalle astrazioni, non dalle concretizzazioni."

Formulazione estesa:

- I moduli di **alto livello** non dovrebbero dipendere da quelli di **basso livello**

- **Entrambi** dovrebbero dipendere da **astrazioni**
- Le astrazioni non dovrebbero dipendere dai dettagli
- I dettagli dovrebbero dipendere dalle astrazioni

Modi per perseguirlo:

- Inserimento delle **policy decisionali** nei moduli alto livello
- Utilizzo del **pattern Template Method**
- Utilizzo di **interfacce per ogni livello** di architettura

Quali qualità architetturali persegue DIP:

- Modularità
- Flessibilità
- Basso accoppiamento
- Coesione
- Sicurezza rispetto ai malfunzionamenti
- Semplicità
- Robustezza

---

2. ANALISI DETTAGLIATA PER PRINCIPIO {#analisi-principi}

Single Responsibility Principle - Analisi approfondita

Domanda operativa chiave: "Che cos'è davvero una responsabilità?"

- Un **asse di cambiamento** è tale **solo se i cambiamenti si verificano effettivamente**
- Il **contesto dell'applicazione** è fondamentale
- Rischio: aggiungere complessità inutile con separazioni premature

Esempio classico: Rectangle

```
GeometricApplication    →    Shape
  +draw()                       ↑
                           Rectangle
                           +x: int
                           +y: int
                           +draw()
                           +area()
```

Analisi:

- `draw()` usato solo dalla GUI

- `area()` usato dall'applicazione di calcolo
- **Due responsabilità diverse in una classe**
- Accoppiamento: cambiamenti grafici possono impattare logica di calcolo

Soluzione corretta:

```
GeometricApplication → GeometricShape
  +draw()                 +area()
                             ↑
                         Rectangle
                         +x: int
                         +y: int
                         +area()

GUI → GraphicShape
        +draw()
           ↑
    GraphicRectangle
       +draw()
```

Separazione completa: grafica vs calcolo.

Esempio modem: context-dependent responsibilities

```
Interfaccia Modem iniziale:
  +dial()
  +hangup()
  +send()
  +receive()
```

Domanda: le responsabilità di connessione e trasmissione dati devono essere separate?

Risposta: Dipende dal contesto:

- Se cambiano **indipendentemente** → separare
- Se cambiano **sempre insieme** → tenere unite

Soluzione con separazione:

```
ModemImplementation
    ↓              ↓
DataChannel    Connection
+send()        +dial()
+receive()     +hangup()
```

I client dipendono dalle interfacce, non dall'implementazione accoppiata.

Relazione con Hexagonal Architecture:

- SRP rafforza separazione business logic da dettagli tecnici
- Business logic testabile indipendentemente
- Uso di porte e adattatori

Open-Closed Principle - Analisi approfondita

Problema classico: calcolo area figure geometriche

Versione SBAGLIATA (viola OCP):

```
class AreaCalculator {
    double calculateArea(Shape[] shapes) {
        double area = 0;
        for (Shape s : shapes) {
            if (s instanceof Circle) {
                Circle c = (Circle) s;
                area += Math.PI * c.radius * c.radius;
            } else if (s instanceof Rectangle) {
                Rectangle r = (Rectangle) s;
                area += r.width * r.height;
            }
            // Aggiungere Triangle richiede MODIFICA di questo codice
        }
        return area;
    }
}
```

Problemi:

- Ogni nuova figura richiede **modifica** di `calculateArea`
- Uso di **RTTI** (`instanceof`)
- **Cascata di if/else**
- Non estensibile senza modifiche

Versione CORRETTA (rispetta OCP):

```
abstract class Shape {
    abstract double area();
}

class Circle extends Shape {
    double radius;
    @Override
```

```
    double area() {
        return Math.PI * radius * radius;
    }
}

class Rectangle extends Shape {
    double width, height;
    @Override
    double area() {
        return width * height;
    }
}

class AreaCalculator {
    double calculateArea(Shape[] shapes) {
        double area = 0;
        for (Shape s : shapes) {
            area += s.area();  // Polimorfismo
        }
        return area;
    }
}
```

Aggiunta di Triangle:

```
class Triangle extends Shape {
    double base, height;
    @Override
    double area() {
        return 0.5 * base * height;
    }
}
// AreaCalculator NON cambia!
```

Meccanismi di chiusura:

1. **Astrazione tramite ereditarietà e polimorfismo**
2. **Data-driven closure** (configurazioni esterne)

Nota strategica:

- Impossibile avere programma 100% chiuso
- Scegliere strategicamente **quali dimensioni di cambiamento** rendere OCP
- Bilanciare flessibilità e complessità

Liskov Substitution Principle - Analisi approfondita

Esempio classico: Square/Rectangle paradox

Setup iniziale:

```
class Rectangle {
    protected int width;
    protected int height;

    void setWidth(int w) { width = w; }
    void setHeight(int h) { height = h; }
    int area() { return width * height; }
}

class Square extends Rectangle {
    @Override
    void setWidth(int w) {
        width = w;
        height = w;  // Mantiene invariante quadrato
    }

    @Override
    void setHeight(int h) {
        width = h;
        height = h;  // Mantiene invariante quadrato
    }
}
```

Test che rivela violazione LSP:

```
void testRectangle(Rectangle r) {
    r.setWidth(5);
    r.setHeight(4);
    assert r.area() == 20;  // FALLISCE con Square!
}

Rectangle rect = new Rectangle();
testRectangle(rect);  // OK

Rectangle square = new Square();
testRectangle(square);  // FALLISCE: area = 16, non 20
```

Perché viola LSP:

- Client di `Rectangle` assume che `setWidth` e `setHeight` siano **indipendenti**
- `Square` **rafforza le precondizioni** (width == height sempre)

- `Square` **non è sostituibile** a `Rectangle` in tutti i contesti

Design by Contract:

- **Precondizioni** (cosa deve essere vero prima): nella derivata devono essere **uguali o più deboli**
- **Postcondizioni** (cosa deve essere vero dopo): nella derivata devono essere **uguali o più forti**
- **Invarianti** (cosa deve essere sempre vero): devono essere preservate

Soluzione corretta:

```
interface Shape {
    int area();
}

class Rectangle implements Shape {
    private int width, height;
    Rectangle(int w, int h) {
        width = w; height = h;
    }
    public int area() { return width * height; }
}

class Square implements Shape {
    private int side;
    Square(int s) { side = s; }
    public int area() { return side * side; }
}
```

Non c'è più ereditarietà Rectangle→Square, quindi no violazione LSP.

Regola generale:

- Non sovrascrivere comportamenti pubblici estrinseci
- Rispettare le specifiche della classe base
- Se una sottoclasse richiede **assunzioni più forti**, viola LSP

Interface Segregation Principle - Analisi approfondita

Problema: Fat Interface

```
interface Worker {
    void work();
    void eat();
```

```
    void sleep();
}

class HumanWorker implements Worker {
    public void work() { /* lavora */ }
    public void eat() { /* mangia */ }
    public void sleep() { /* dorme */ }
}

class RobotWorker implements Worker {
    public void work() { /* lavora */ }
    public void eat() { /* NON SERVE */ }
    public void sleep() { /* NON SERVE */ }
}
```

Problema:

- `RobotWorker` forzato a implementare metodi inutili
- Accoppiamento non necessario
- Violazione ISP

Soluzione con interfacce segregate:

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

interface Eatable {
    void eat();
}

interface Sleepable {
    void sleep();
}

class HumanWorker implements Workable, Eatable, Sleepable {
    public void work() { /* lavora */ }
    public void eat() { /* mangia */ }
    public void sleep() { /* dorme */ }
}

class RobotWorker implements Workable {
    public void work() { /* lavora */ }
}
```

Benefici:

- Ogni classe implementa solo ciò che serve

- Interfacce **coese** e **specializzate**
- Riduzione accoppiamento

Con Adapter (se ereditarietà multipla non disponibile):

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

class HumanWorker implements Workable {
    private EatingBehavior eater = new HumanEater();
    private SleepingBehavior sleeper = new HumanSleeper();

    public void work() { /* lavora */ }
    public void eat() { eater.eat(); }
    public void sleep() { sleeper.sleep(); }
}
```

Dependency Inversion Principle - Analisi approfondita

Problema: dipendenza diretta da dettagli

```
// Modulo ALTO livello
class BusinessLogic {
    private MySQLDatabase db = new MySQLDatabase();

    void saveData(String data) {
        db.insert(data);
    }
}

// Modulo BASSO livello
class MySQLDatabase {
    void insert(String data) { /* SQL specifico */ }
}
```

Problemi:

- `BusinessLogic` dipende da dettaglio concreto (`MySQLDatabase`)
- Cambiare database richiede modifica di `BusinessLogic`
- Testing difficile (serve MySQL reale)

Soluzione con DIP:

```java
// Astrazione (nessuna dipendenza da dettagli)
interface Database {
    void insert(String data);
}

// Modulo ALTO livello dipende da astrazione
class BusinessLogic {
    private Database db;

    BusinessLogic(Database database) {
        this.db = database;
    }

    void saveData(String data) {
        db.insert(data);
    }
}

// Moduli BASSO livello dipendono da astrazione
class MySQLDatabase implements Database {
    public void insert(String data) { /* SQL specifico */ }
}

class PostgreSQLDatabase implements Database {
    public void insert(String data) { /* PostgreSQL specifico */ }
}

class MockDatabase implements Database {
    public void insert(String data) { /* per test */ }
}
```

Benefici:

- Policy nei moduli alto livello
- Dettagli intercambiabili
- Testabilità completa

Dependency Injection:

```java
// Constructor injection (dipendenze forti)
BusinessLogic logic = new BusinessLogic(new MySQLDatabase());

// Setter injection (dipendenze deboli)
BusinessLogic logic = new BusinessLogic(defaultDb);
logic.setDatabase(new PostgreSQLDatabase());
```

Preferenza: sempre **constructor injection** per dipendenze necessarie.

3. QUIZ STORICI CON SOLUZIONI E STRATEGIE {#quiz}

Quiz 1: Identificazione equivalenze SRP

Domanda: "Si selezionino le affermazioni corrette fra le seguenti relative ai SOLID principles."

a. È possibile che un gruppo di classi sia Open-Close anche se si utilizzano delle variabili globali, nei modi e termini permessi dal linguaggio di programmazione di riferimento.

b. Usando il Liskov Substitution Principle è associabile al concetto di Design by Contract.

c. Un insieme di classi può essere definito Open-Close rispetto a qualsiasi tipo di modifica richiesta.

d. Il Single Responsibility Principle è anche noto con il nome di coesione.

Risposte corrette: b, d

Analisi opzione a:

- ❌ FALSA
- Variabili globali creano accoppiamento forte
- Violano information hiding
- Rendono difficile OCP (modifiche globali impattano ovunque)

Analisi opzione b:

- ✓ CORRETTA
- LSP richiede rispetto di pre/postcondizioni
- Design by Contract formalizza contratti tra caller e callee
- Connessione diretta e fondamentale

Analisi opzione c:

- ❌ FALSA
- È **impossibile** essere OCP rispetto a **qualsiasi** modifica
- OCP richiede scelte strategiche su **quali** dimensioni di cambiamento supportare
- Parola chiave "qualsiasi" rende la frase falsa

Analisi opzione d:

- ✓ CORRETTA
- SRP ≡ Coesione (equivalenza esatta)
- Questa è la definizione più importante da ricordare

Strategia generale:

- Prestare attenzione a parole assolute: "qualsiasi", "sempre", "mai"
- SRP = coesione è equivalenza da memorizzare
- LSP ↔ Design by Contract è connessione fondamentale

Quiz 2: Valutazione coesione sistema

Domanda: "Sia dato il seguente diagramma delle classi. Si prenda in considerazione un sistema costituito unicamente dai tipi riportati nel diagramma. È possibile affermare che il sistema abbia il massimo grado di coesione possibile?"

O Vero
✓ Falso

Risposta corretta: Falso

Analisi:

- Coesione massima richiede che **ogni elemento sia necessario** e **nessuno sia superfluo**
- Dalla struttura non si può dedurre:
  - Se A, C, D usano effettivamente B
  - Se le dipendenze sono funzionalmente necessarie
  - Se ci sono responsabilità separate o accoppiate
- **Senza contesto applicativo**, non si può affermare coesione massima
- Principio generale: **bisogna analizzare tutto il contesto**, non la singola struttura

Strategia:

- Domande su "massimo grado" o "sempre" richiedono verifica completa
- Struttura UML da sola non determina rispetto principi SOLID
- Serve contesto: come vengono usate le classi?

Quiz 3: Rectangle e SRP

Domanda: "Sia data la seguente struttura, utilizzata da un'applicazione che visualizza figure geometriche su uno schermo."

```
GeometricApplication  →  Shape
   +draw()                  ↑
                        Rectangle
                        +x: int
                        +y: int
                        +draw()
                        +area()
```

"La classe Rectangle soddisfa il Single Responsibility Principle."

✓ Vero
O Falso

Risposta corretta: Vero

Analisi dettagliata:

- Domanda **insidiosa**: la descrizione dice "visualizza figure geometriche"
- Contesto: applicazione grafica
- In questo contesto, `draw()` e `area()` fanno parte della **stessa responsabilità** (renderizzazione geometrica)
- `area()` potrebbe essere necessario per calcoli di rendering (es. ordinamento z-buffer, culling)
- **Chiave:** il contesto determina se c'è una o più responsabilità

Confronto con esempio classico:

- Nell'esempio classico Cardin: due applicazioni separate (GUI + calcolo)
- In quel caso: due responsabilità distinte
- Qui: unica applicazione grafica → unica responsabilità

Lezione strategica:

- **Il contesto è tutto**
- Stessa struttura può rispettare o violare SRP a seconda dell'uso
- Leggere attentamente la descrizione del sistema

Quiz 4: OCP e variabili globali

Domanda implicita: "È possibile che un gruppo di classi sia Open-Close anche se si utilizzano delle variabili globali?"

Risposta: Sì, "nei modi e termini permessi dal linguaggio di programmazione di riferimento"

Analisi:

- Variabili globali generalmente **danneggiano** OCP
- MA esistono contesti dove sono gestite in modo controllato:
  - Singleton pattern ben implementato
  - Costanti globali immutabili
  - Configuration objects read-only
- Frase critica: "**nei modi e termini permessi**"
- Riconosce che linguaggi diversi hanno meccanismi diversi

Strategia:

- Non cadere in assolutismi
- Qualificatori come "nei termini permessi" cambiano la risposta
- OCP non è binario (sì/no) ma graduato

---

4. ESEMPI DI CODICE E DIAGRAMMI UML {#esempi}

Esempio completo: sistema di notifiche

Scenario: Sistema che invia notifiche via Email, SMS, Push.

Versione 1: Viola SRP, OCP, DIP

```
class NotificationService {
    private String smtpServer = "smtp.example.com";
    private String smsGateway = "api.sms.com";

    void sendNotification(User user, String message, String type) {
        if (type.equals("email")) {
            // Logica SMTP diretta
            connect(smtpServer);
            send(user.email, message);
        } else if (type.equals("sms")) {
            // Logica SMS diretta
            httpPost(smsGateway, user.phone, message);
        } else if (type.equals("push")) {
            // Logica push diretta
            pushToDevice(user.deviceId, message);
        }
```

```
    }
}
```

Problemi:

1. **Viola SRP**: gestisce 3 tipi di notifiche (3 responsabilità)
2. **Viola OCP**: aggiungere Slack richiede modificare `sendNotification`
3. **Viola DIP**: dipende da dettagli concreti (SMTP, SMS API)
4. Difficile da testare
5. Accoppiamento alto

Versione 2: Rispetta tutti i principi SOLID

```
// DIP: Astrazione
interface NotificationChannel {
    void send(User user, String message);
}

// ISP: Interfacce specifiche se servono comportamenti diversi
interface BatchNotificationChannel extends NotificationChannel {
    void sendBatch(List<User> users, String message);
}

// OCP: Implementazioni estendono senza modificare
class EmailNotification implements NotificationChannel {
    private EmailService emailService;

    EmailNotification(EmailService service) {
        this.emailService = service;
    }

    public void send(User user, String message) {
        emailService.sendEmail(user.getEmail(), message);
    }
}

class SmsNotification implements NotificationChannel {
    private SmsService smsService;

    SmsNotification(SmsService service) {
        this.smsService = service;
    }

    public void send(User user, String message) {
        smsService.sendSms(user.getPhone(), message);
    }
```

```
}

class PushNotification implements NotificationChannel {
    private PushService pushService;

    PushNotification(PushService service) {
        this.pushService = service;
    }

    public void send(User user, String message) {
        pushService.push(user.getDeviceId(), message);
    }
}

// SRP: Responsabilità unica = orchestrare notifiche
class NotificationService {
    private NotificationChannel channel;

    // DIP: Dependency injection
    NotificationService(NotificationChannel channel) {
        this.channel = channel;
    }

    void sendNotification(User user, String message) {
        channel.send(user, message);
    }
}

// Uso
NotificationChannel email = new EmailNotification(new SmtpEmailService());
NotificationService service = new NotificationService(email);
service.sendNotification(user, "Hello!");

// OCP: Aggiungere Slack NON richiede modifiche esistenti
class SlackNotification implements NotificationChannel {
    public void send(User user, String message) {
        // Implementazione Slack
    }
}
```

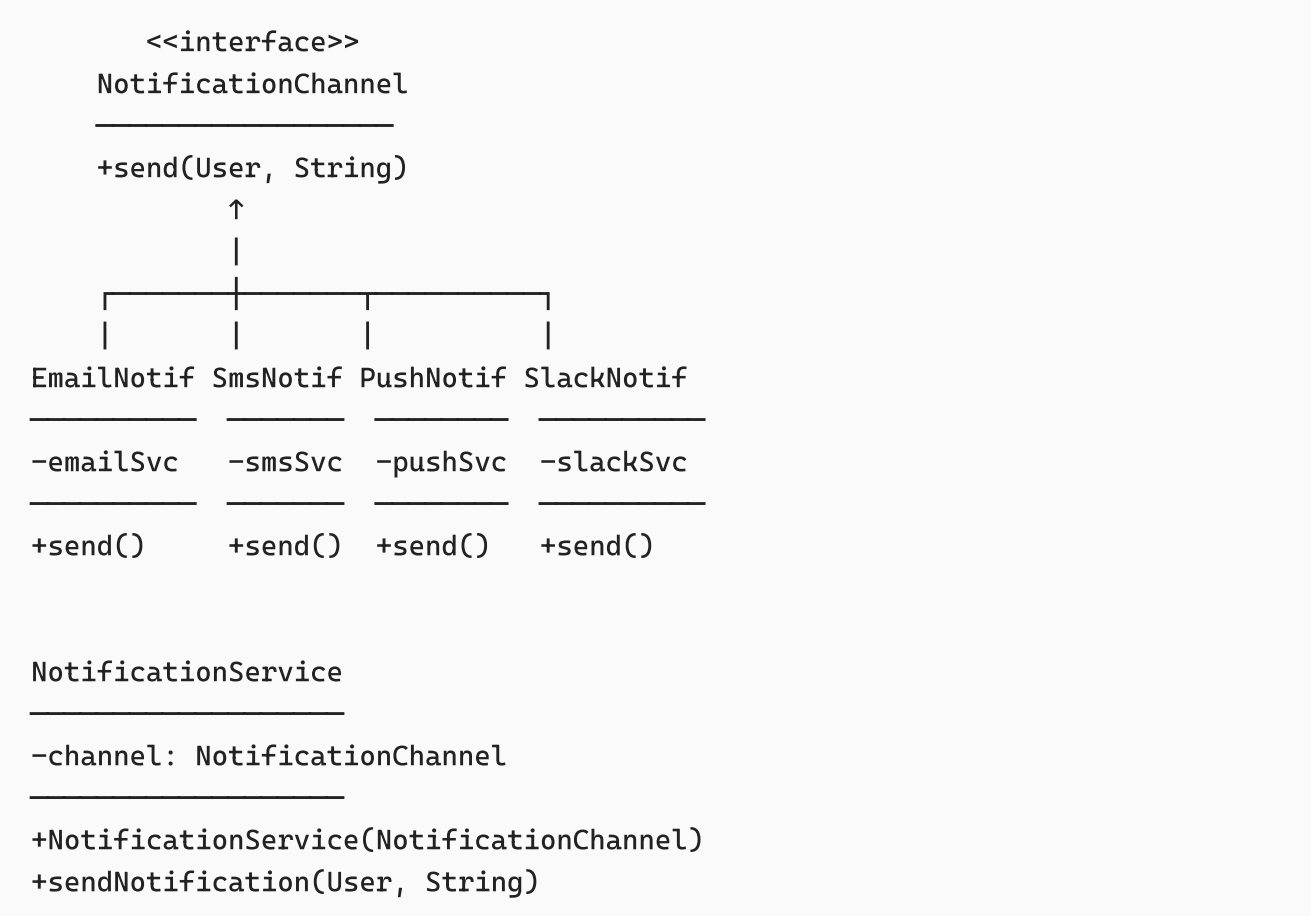
Analisi rispetto SOLID:

1. **SRP**:
   - `NotificationService` = orchestrazione
   - `EmailNotification` = dettagli email
   - `SmsNotification` = dettagli SMS
   - Ogni classe: una responsabilità
2. **OCP**:

- Aggiungere `SlackNotification` = estensione (nuovo codice)
- NON modifica `NotificationService` o altre classi

3. **LSP**:
   - Tutte le implementazioni sostituibili via `NotificationChannel`
   - Nessuna viola contratto di `send()`
4. **ISP**:
   - `NotificationChannel` = interfaccia minima
   - `BatchNotificationChannel` = estensione opzionale
   - Client usano solo ciò che serve
5. **DIP**:
   - `NotificationService` dipende da `NotificationChannel` (astrazione)
   - Non dipende da `EmailNotification` (concretizzazione)
   - Dependency injection via constructor

Diagramma UML: sistema notifiche



Relazioni:

- `NotificationService` **dipende** da `NotificationChannel` (DIP)
- `EmailNotification` **implementa** `NotificationChannel` (OCP)
- Client può sostituire qualsiasi implementazione (LSP)

5. ERRORI COMUNI E DISTINZIONI CRITICHE {#errori-comuni}

Errore 1: "SOLID si applica alla singola classe"

SBAGLIATO:

> "Guardo la classe Rectangle. Ha draw() e area(). Due metodi = due responsabilità = viola SRP."

CORRETTO:

> "Guardo il contesto: chi usa Rectangle? Se una sola applicazione grafica usa entrambi i metodi per rendering, è SRP compliant. Se due applicazioni diverse (GUI e calcolo scientifico) usano metodi diversi, viola SRP."

Lezione:

- SOLID richiede **analisi del contesto**
- Stessa struttura può rispettare o violare principi a seconda dell'uso

---

Errore 2: "OCP significa codice immutabile"

SBAGLIATO:

> "OCP dice che non devo mai modificare il codice esistente."

CORRETTO:

> "OCP dice di estendere comportamento aggiungendo codice, non modificando il vecchio. Ma chiusura completa è impossibile; faccio scelte strategiche su quali dimensioni di cambiamento rendere OCP."

Esempio di modifica legittima:

- Bug fix nel codice esistente
- Refactoring per performance
- Cambio requisiti fondamentali

Lezione:

- OCP non è assoluto
- Bilanciare flessibilità e complessità

Errore 3: "Ereditarietà IS-A garantisce LSP"

SBAGLIATO:

> "Square IS-A Rectangle matematicamente, quindi Square estende Rectangle è corretto."

CORRETTO:

> "LSP ≠ relazione IS-A concettuale. LSP = sostituibilità comportamentale. Square rafforza precondizioni (width==height), quindi viola LSP anche se matematicamente corretta."

Lezione:

- IS-A concettuale ≠ IS-A comportamentale
- LSP richiede sostituibilità in **tutti i contesti d'uso**

Errore 4: "Più interfacce = sempre meglio"

SBAGLIATO:

> "ISP dice di avere tante interfacce piccole, quindi creo un'interfaccia per ogni metodo."

CORRETTO:

> "ISP dice di evitare fat interfaces. Ma interfacce troppo granulari creano complessità. Bilancio tra coesione (metodi correlati insieme) e segregazione (client non dipendono da inutile)."

Esempio:

```
// ESAGERATO (troppo granulare)
interface Readable { String read(); }
interface Writable { void write(String s); }
interface Closeable { void close(); }
interface Flushable { void flush(); }

// BILANCIATO
interface InputSource { String read(); void close(); }
interface OutputSink { void write(String s); void flush(); void close(); }
```

Lezione:

- ISP non significa "un metodo per interfaccia"
- Cercare coesione funzionale

Errore 5: "DIP = Dependency Injection"

SBAGLIATO:

> "Se uso dependency injection (DI), rispetto DIP."

CORRETTO:

> "DI è un meccanismo per implementare DIP, ma DIP è il principio: dipendere da astrazioni. Posso fare DI con classi concrete (viola DIP) o fare DIP senza DI framework (manual wiring)."

Esempio viola DIP anche con DI:

```
class Service {
    private MySQLDatabase db;

    Service(MySQLDatabase database) {  // DI, ma classe concreta!
        this.db = database;
    }
}
```

Esempio rispetta DIP:

```
class Service {
    private Database db;  // Interfaccia

    Service(Database database) {
        this.db = database;
    }
}
```

Lezione:

- DIP = principio architetturale
- DI = tecnica implementativa

Errore 6: "SOLID solo per OOP"

SBAGLIATO:

> "SOLID è per Java/C++. In linguaggi funzionali non si applica."

CORRETTO:

> "SOLID esprime principi universali:

> - SRP → Funzioni/moduli con responsabilità singola
> - OCP → Higher-order functions, pattern matching estensibile
> - LSP → Rispetto contratti funzioni
> - ISP → Funzioni con pochi parametri necessari
> - DIP → Parametrizzazione su funzioni (astrazione)"

Esempio funzionale (Haskell-style):

```haskell
-- DIP: Parametrizzo su funzione (astrazione)
processData :: (a -> b) -> [a] -> [b]
processData transform = map transform

-- Client sceglie trasformazione concreta
result = processData (\x -> x * 2) [1,2,3]
```

Lezione:

- SOLID = principi di design modulare
- Applicabili a qualsiasi paradigma

Errore 7: "Seguire tutti i principi SOLID sempre"

SBAGLIATO:

> "Devo applicare tutti i 5 principi a ogni classe."

CORRETTO:

> "SOLID offre linee guida, non leggi assolute. A volte violare un principio è pragmatico:
>
> - Script piccoli: OCP potrebbe essere overkill
> - Prototipi: flessibilità futura non prioritaria
> - Performance critical: indirezione (DIP) può costare"

Trade-offs:

- SOLID aumenta flessibilità ma anche complessità iniziale
- Valutare ROI: il codice cambierà spesso?

Lezione:

- SOLID = strumenti, non dogmi
- Bilanciare principi con pragmatismo

---

6. CHECKLIST PRE-ESAME {#checklist}

Definizioni da sapere a memoria

- [ ] **SRP**: "Una classe dovrebbe avere una, e una sola, ragione per cambiare"
- [ ] **SRP = Coesione** (equivalenza fondamentale per crocette)
- [ ] **OCP**: "Aperte all'estensione, chiuse alle modifiche"
- [ ] **OCP** meccanismo: astrazione (interfacce/classi astratte)
- [ ] **LSP**: "Classi derivate sostituibili con classi base"
- [ ] **LSP ↔ Design by Contract** (connessione esplicita per crocette)
- [ ] **LSP** pre/postcondizioni: pre più deboli, post più forti nelle derivate
- [ ] **ISP**: "Interfacce specifiche per client, non fat interfaces"
- [ ] **DIP**: "Dipendere da astrazioni, non concretizzazioni"
- [ ] **DIP** policy: moduli alto livello non dipendono da basso livello

Concetti da chiarire

- [ ] Differenza SRP vs accoppiamento
- [ ] Perché violazione LSP implica violazione OCP
- [ ] Impossibilità di chiusura completa in OCP
- [ ] Relazione tra DIP e Dependency Injection (DIP = principio, DI = tecnica)
- [ ] ISP e ereditarietà multipla vs Adapter
- [ ] SOLID context-dependent (analisi contesto, non singola classe)

Quiz pattern da riconoscere

- [ ] "SRP è anche noto come..." → Coesione
- [ ] "LSP è associabile a..." → Design by Contract
- [ ] "OCP con variabili globali..." → Possibile nei termini del linguaggio
- [ ] "OCP rispetto a qualsiasi modifica..." → FALSO (impossibile)
- [ ] "Massimo grado di coesione..." → Richiede analisi contesto completo

Esempi di codice da saper analizzare

- [ ] Rectangle con draw() e area(): SRP sì/no? → Dipende dal contesto
- [ ] AreaCalculator con if/instanceof: viola OCP
- [ ] AreaCalculator con polimorfismo: rispetta OCP
- [ ] Square extends Rectangle: viola LSP
- [ ] Worker interface con eat/sleep per Robot: viola ISP
- [ ] BusinessLogic con MySQLDatabase concreta: viola DIP
- [ ] NotificationService con dependency injection su interfaccia: rispetta DIP

Strategie per crocette

- [ ] Leggere **tutto il contesto** della domanda (descrizione sistema)
- [ ] Attenzione a parole assolute: "sempre", "mai", "qualsiasi" (spesso false)
- [ ] Qualificatori cambiano risposta: "nei modi e termini permessi"
- [ ] "Massimo grado" richiede verifica completa (spesso FALSO)
- [ ] Struttura UML da sola non determina SOLID (serve contesto d'uso)

Domande teoria aperta possibili

- [ ] "Spiegare i principi SOLID e il loro scopo"
- [ ] "Quale principio SOLID viene violato in questo codice? Perché?"
- [ ] "Refactorare questo codice per rispettare OCP"
- [ ] "Differenza tra DIP e Dependency Injection"
- [ ] "Perché LSP è collegato a Design by Contract?"
- [ ] "Quando è accettabile violare un principio SOLID?"

Collegamenti con altri concetti

- [ ] SOLID → Coesione/Accoppiamento (SRP/ISP/DIP riducono accoppiamento)
- [ ] SOLID → Design Patterns (Strategy = OCP, Template Method = DIP)
- [ ] SOLID → Testabilità (DIP fondamentale per mock/stub)
- [ ] SOLID → Manutenibilità (tutti i principi facilitano modifica)
- [ ] SOLID → Hexagonal Architecture (separazione core vs adattatori)

Errori da evitare assolutamente

- [ ] Non dire "SRP = un metodo per classe" (è responsabilità, non granularità)
- [ ] Non dire "OCP = codice immutabile" (estensione, non immutabilità)
- [ ] Non confondere IS-A concettuale con sostituibilità LSP
- [ ] Non credere che SOLID si applichi solo a OOP
- [ ] Non dimenticare: **il contesto è fondamentale**

RISORSE SUPPLEMENTARI

Libri consigliati da Cardin

- **"Clean Code"** di Robert Martin
- **"Agile Software Development: Principles, Patterns, and Practices"** di Robert Martin

Pattern correlati a SOLID

- **Strategy Pattern**: implementa OCP (estensione via nuove strategie)
- **Template Method**: implementa DIP (policy in classe astratta, dettagli in derivate)

- **Adapter Pattern**: implementa ISP (adatta interfacce incompatibili)
- **Dependency Injection**: tecnica per DIP

Collegamenti architetturali

- **Hexagonal Architecture**: enfatizza SRP e DIP (core vs adattatori)
- **Layered Architecture**: rischia di violare DIP se layer superiori dipendono da implementazioni inferiori
- **Microservizi**: ciascun servizio dovrebbe rispettare SRP

NOTE FINALI

Approccio all'esame:

1. Leggere domanda COMPLETA (descrizione contesto)
2. Identificare principio/i coinvolti
3. Analizzare contesto d'uso (non solo struttura)
4. Verificare parole chiave ("sempre", "qualsiasi", "massimo")
5. Applicare definizioni memorizzate
6. Controllare coerenza risposta

Priorità di studio:

1. **Critici**: SRP=Coesione, LSP↔Design by Contract, OCP astrazione
2. **Importanti**: DIP vs DI, ISP motivazione, violazioni comuni
3. **Utili**: Pattern correlati, esempi codice, trade-offs

Attitudine mentale:

- SOLID = linee guida, non leggi
- Contesto determina rispetto principi
- Bilanciare purezza teorica e pragmatismo
- Comprendere "perché" oltre al "cosa"

DOCUMENTO CREATO PER RIPASSO COMPLETO SOLID
Ingegneria del Software - UniPD
Prof. Tullio Vardanega e Riccardo Cardin