28 - Principi SOLID

Con l'acrostico SOLID si identificano i cosiddetti "primi 5 principi" della progettazione object oriented. Lo scopo di essi è fornire una guida verso lo sviluppo di progetti che siano più "flessibili", ovvero progetti per i quali sia relativamente semplificato effettuare modifiche in termini di:

- manutenzione delle funzionalità esistenti;
- estensione delle funzionalità supportate.
  S ⇒ SRP (Single Responsibility Principle)
  O ⇒ OCP (Open-Closed Principle)
  L ⇒ LSP (Liskov Substitution Principle)
  I ⇒ ISP (Interface Segregation Principle)
  D ⇒ DIP (Dependency Inversion Principle)
  Parliamo di principi e non di tecniche o metodi in quanto non sono immediatamente applicabili, la loro applicazione richiede di valutare se sia o meno opportuno applicarli nei vari contesti concreti che si presentano.
  Quando un software cresce in termini di complessità, aumenta il rischio di creare codice fragile o difficile da mantenere. Alcune problematiche che SOLID aiuta a prevenire sono:
- **accoppiamento eccessivo**: le classi dipendono troppo l'una dall'altra, rendendo difficile apportare modifiche senza causare effetti collaterali;
- **ridondanza**: il codice viene duplicato perché manca modularità e riutilizzo;
- **rigidità**: è difficile aggiungere nuove funzionalità perché il codice esistente deve essere modificato in modo esteso;
- **opacità**: il codice diventa complesso e difficile da leggere.
  I principi SOLID favoriscono lo sviluppo di software scalabile e flessibile, capace di adattarsi facilmente ai cambiamenti richiesti nel tempo.

SRP - Single Responsibility Principle

(Principio della Responsabilità Unica)
Una classe, funzione o modulo dovrebbe essere **responsabile di un'unica cosa**. Si può dire anche che, ogni classe dovrebbe avere **un solo motivo per essere modificata**: se esistono più motivi distinti vuol dire che la classe si assume più responsabilità e quindi dovrebbe essere suddivisa in più componenti.
Quando una classe ha più responsabilità:

- diventa più difficile da mantenere;
- qualsiasi modifica a una delle responsabilità potrebbe causare effetti collaterali su altre funzionalità;
- diventa meno riutilizzabile perché accoppia logiche diverse.
  Il rispetto di questo principio porta a codice:
- più manutenibile;
- più facile da riutilizzare.
  Esempio: consideriamo una classe `Report` che:

1. genera un report;
2. lo salva su disco;

3. lo invia tramite email.
   Questa classe viola il principio SRP perché ha tre responsabilità distinte.
   Correzione:

- `ReportGenerator` → crea il report;
- `ReportSaver` → salva il report;
- `EmailSender` → invia il report.

Una classe che deve manipolare più risorse (in maniera exception safe) non deve prendersi carico anche della corretta gestione dell'acquisizione e rilascio; piuttosto dovrebbe delegare questi compiti ad altre classi gestore.

OCP - Open-Closed Principle

(Principio Aperto-Chiuso)
Le entità software (classi, moduli, funzioni) dovrebbero essere **aperte per estensioni**, ma **chiuse per modifiche**.
In altre parole, un software ben progettato deve rendere semplice l'aggiunta di funzionalità, senza che per fare ciò sia necessario modificare il codice esistente, per evitare di introdurre bug o effetti collaterali.
Per ottenere un progetto coerente con il principio OCP occorre individuare le parti del software che, probabilmente, saranno oggetto di modifiche in futuro e applicare ad esse opportuni costrutti di astrazione.
Esempio: confrontando le due varianti (`old_style` vs `oo_style`) del progetto "fattoria" le parti di codice che saranno oggetto di cambiamento è l'introduzione di nuovi animali, che dovranno essere utilizzabili nel codice utente. Notiamo che:

- `old_style` :
  - ha un'unica classe `Animale` che implementa tutti gli animali concreti;
  - l'aggiunta di nuovi animali, infatti, comporta la modifica della classe;
  - il codice è aperto alle estensioni, ma solo parzialmente chiuso alle modifiche;
    Pur non dovendo modificare il codice che genera le strofe, ogni estensione che aggiunge un animale rischia di rompere il codice preesistente.
- `oo_style` :
  - abbiamo una classe astratta `Animale` che fornisce l'interfaccia, senza dettagli implementativi;
  - il codice è aperto alle estensioni, in quanto per aggiungere un nuovo animale, basta creare una nuova classe che implementa (con derivazione pubblica "IS-A") l'interfaccia astratta;
  - il codice è chiuso alle modifiche: le aggiunte non hanno impatto sul codice che genera la strofa e nemmeno sulle classi degli altri animali.
    N.B. in entrambi i casi l'aggiunta di nuovi animali prevede modifiche al modulo `Maker.cc` , per consentire ai nuovi animali di essere utilizzati dal programma. Quindi la "chiusura alle modifiche" non può mai essere totale.

DIP - Dependency Inversion Principle

I moduli di alto livello non dovrebbero dipendere da quelli di basso livello; entrambi dovrebbero dipendere da astrazioni. Le astrazioni non devono dipendere dai dettagli, ma i dettagli devono dipendere dalle astrazioni.
In altre parole, invece di far dipendere il codice da implementazioni concrete, si dovrebbe basare su astrazioni, come interfacce o classi astratte. Questo principio fa una classifica delle dipendenze tra

moduli software, stabilendo che alcune sono ammesse (inevitabile e innocue), mentre altre sono da evitare (dannose).

- *dipendenze buone*: quelle verso i concetti astratti;
- *dipendenze cattive*: quelle verso i dettagli implementativi.
  Spesso il software viene progettato con un approccio *top-down*: partendo dal problema generale, lo si suddivide in sottoproblemi via via sempre più piccoli. Si arriva cosi a una stratificazione del codice, dove i moduli a livello più alto usano (quindi dipendono da) i moduli a livello più basso, creando cosi delle dipendenze (dall'astratto al concreto).
  Il principio DIP suggerisce di **invertire** le dipendenze sostituendolo con altre che non creano problemi, per farlo occorre individuare le **interfacce astratte**, che non dipendono da dettagli implementativi:
- i moduli ad alto livello vengono modificati per usare le interfacce astratte;
- i moduli a basso livello vengono modificati per implementare le interfacce astratte.
  In questo modo miglioriamo anche l'aderenza al principio OCP, in quanto adesso è possibile estendere il software senza influenzare i moduli ad alto livello.

LSP - Liskov Substitution Principle

(Principio di Sostituibilità di Liskov)
Gli oggetti di una classe derivata devono poter sostituire quelli della classe base senza alterare il comportamento corretto del programma.

In altre parole, il comportamento di una classe derivata deve essere coerente con quello della classe base. Le classi derivate non devono violare le aspettative degli utenti della classe base.
Il principio LSP definisce il cosiddetto **behavioral subtyping**: siccome la classe derivata dichiara di essere in relazione IS-A rispetto alla classe base, gli oggetti della classe derivata non solo devono fornire (sintatticamente) i metodi dichiarati dalla classe base, ma si devono anche comportare (aspetto semantico) come se fossero degli oggetti della classe base.
Le corrispondenze di comportamento tra le due classi non deve essere intesa in senso assoluto, ma limitata alle **aspettative legittime** che può avere un utente della classe base.
Le aspettative legittime sono quelle stabilite dai *contratti* (invarianti di classe, precondizioni e postcondizioni dei metodi) che la classe base ha stipulato con i suoi utenti.
Infatti, il principio LSP (e il behavioral subtyping) sono quindi in connessione stretta con la **programmazione per contratto**: quando la classe derivata dichiara di essere in relazione IS-A con la classe base, di fatto si impegna a rispettare tutti i contratti che la classe base ha stabilito con i suoi utenti.
Esempio: consideriamo l'esempio della "Fattoria": un animale concreto soddisfa il principio LSP se, quando implementa i tre metodi virtuali dell'interfaccia astratta Animale, rispetta il contratto stabilito da questa con i suoi utenti.
Un altro esempio: partiamo dalla classe `Rettangolo` e cerchiamo di ottenere la classe `Quadrato`

```
class Rettangolo {
    long lung;
    long largh;

    public:
    bool check_inv() const { return lung > 0 && larg > 0; }

    Rettangolo(long lunghezza, long larghezza)
```

```
        : lung(lunghezza), larg(larghezza) {
        if (!check_inv())
            throw std::invalid_argument("Dimensioni invalide");
    }

    long get_lunghezza() const { return lung; }
    long get_larghezza() const { return larg; }

    ////

    void set_lunghezza(long value) {
        if (value < = 0)
            throw std::invalid_argument("Dimensione invalida");
        lung = value;
    }

    void set_larghezza(long value) {
        if (value < = 0)
            throw std::invalid_argument("Dimensione invalida");
        larg = value;
    }

    long get_area() const { return lung * larg; }
}; //class Rettangolo
```

Per creare la classe `Quadrato`, si potrebbe pensare di implementare la classe usando l'ereditarietà pubblica e il polimorfismo dinamico, dunque le uniche modifiche da fare sono:

- dichiarare i metodi di rettangolo come virtual;
- dichiarare il distruttore di rettangolo virtuale
- derivare pubblicamente `Quadrato` da `Rettangolo`
- fare l'override dei metodi `set_larghezza` e `set_lunghezza`, per assicurarsi che quando si modifica una dimensione sia modificata anche l'altra, cosi da mantenere l'invariante della classe `Quadrato`
  Modifichiamo `Rettangolo`:

```
class Rettangolo {
    ///
    virtual bool check_inv() const { ... }
    virtual long get_lunghezza() const { ... }
    virtual long get_larghezza() const { ... }
    virtual long set_lunghezza(long value) { ... }
    virtual long set_larghezza(long value) { ... }
    virtual long get_area() const { ... }
    virtual ~Rettangolo() = default;
}; //class Rettangolo

class Quadrato : public Rettangolo {
public:
    Quadrato(long lato) : Rettangolo(lato, lato) {}
    bool check_inv() const override {
        return lung > 0 && lung == larg;
    }
    void set_lunghezza(long value) override {
```

```
        Rettangolo::set_lunghezza(value);
        Rettangolo::set_larghezza(value);
    }
    void set_larghezza(long value) override {
        set_lunghezza(value);
    }
}; //class Quadrato
```

Questo progetto viola la LSP, in quanto è falso che qualunque funzione che usa puntatori (o riferimenti) alla classe base Rettangolo possiamo passare invece puntatori (o riferimenti) alla classe derivata Quadrato e soddisfare le legittime aspettative dell'utente. Hanno la stessa interfaccia (aspetto sintattico) ma non sono equivalenti in quanto si comportano diversamente. Ad esempio, la classe Quadrato viola il contratto del metodo `Rettangolo::set_lunghezza`, in quanto nelle postcondizioni stabilisce che va a modificare solo la lunghezza, mentre l'overriding definito da Quadrato va a modificare anche la larghezza. Possiamo allora dire che la relazione Quadrato IS-A Rettangolo non è valida, in quanto un quadrato non si comporta come un rettangolo.

ISP - Interface Segregation Principle

(Principio di Segregazione delle Interfacce)
Un'interfaccia non dovrebbe costringere le classi a implementare metodi che non usano. Interfacce troppo grandi rendono le classi rigide e accoppiate. Cambiare un metodo inutilizzato in un'interfaccia può richiedere modifiche a tutte le classi che la implementano. In altre parole, si dovrebbe preferire tante interfacce piccole (*thin interfaces*) rispetto a poche interfacce grandi (*thick interfaces*). Può essere interpretato come una forma particolare del principio SRP, che si concentra sul caso specifico della progettazione delle interfacce.
L'applicazione di questo principio presuppone la possibilità di utilizzare l'ereditarietà multipla (di interfaccia).