Principi progettazione object oriented

L'acrostico SOLID

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

- manutenzione delle funzionalità esistenti;
- estensione delle funzionalità supportate.

S $\to$ SRP (Single Responsibility Principle) O $\to$ OCP (Open-Closed Principle) L $\to$ LSP (Liskov Substitution Principle) I $\to$ ISP (Interface Segregation Principle) D $\to$ DIP (Dependency Inversion Principle)

Torna all'indice

Note importanti

I principi SOLID non sono stati "inventati" o "proposti" contemporaneamente: l'acrostico ha solamente fornito un nome facilmente memorizzabile per concetti o metodologie che sono stati sviluppati in tempi diversi da persone diverse e, abbastanza spesso, con nomi diversi. L'uso sistematico dei TLA (Three Letter Acronyms) per riferirsi ai singoli principi può essere etichettato come una "moda" dei tempi.

L'ordine in cui sono elencati i principi NON corrisponde ad una qualche relazione di priorità tra questi, ma è solo funzionale a creare l'acrostico. Di conseguenza, verranno presentati in un ordine diverso.

Si parla di "principi" e non di "tecniche" o "metodi", perché non sono immediatamente applicabili (e a maggior ragione non è opportuno immaginare che vengano applicati in maniera sistematica e/o automatica).

La loro applicazione richiede uno sforzo per valutare se sia o meno opportuno applicarli nei vari contesti concreti che si presentano, in quanto ogni beneficio (conseguente all'adozione di una modifica di progetto che favorisca uno dei principi) spesso comporta anche un corrispondente costo (in termini di comprensibilità del codice da parte del programmatore e/o in termini di modifiche da applicare a codice esistente).

In linea di massima, il progettista dovrebbe individuare quelle parti del software che:

- sono in contrasto con alcuni di questi principi;
- in futuro potrebbero beneficiare dall'applicazione di una ristrutturazione del codice in linea con i principi;

e quindi applicare le azioni correttive del caso.

La pretesa di applicare i principi sistematicamente, a tutte le porzioni di codice, indipendentemente da ogni valutazione di opportunità, tipicamente porta a progetti oltremodo complicati, che violano altri principi di progettazione. A titolo di esempio, si ricorda il principio KISS (Keep It Simple, Stupid) che suggerisce di evitare ogni tipo di complicazione non strettamente necessaria.

Torna all'indice

SRP (Single Responsibility Principle)

Si tratta di un principo di validità generale (cioè, non è limitato al caso della progettazione object oriented) che dice, intuitivamente, che ogni porzione di software che progettiamo e implementiamo (una classe, una funzione, ecc.) dovrebbe avere in carico una sola responsabilità.

A volte si dice che ogni classe dovrebbe avere un solo "motivo per essere modificata": se esistono più motivi distinti, questo è indice che la classe si assume più responsabilità e quindi dovrebbe essere suddivisa in più componenti, ognuno

dei quali caratterizzato da una singola responsabilità.

Il rispetto del principio porta a codice più manutenibile e, in linea di massima, più facile da riutilizzare.

Esempio: Una classe che deve manipolare una pluralità di risorse (in maniera exception safe) non dovrebbe prendersi carico direttamente della corretta gestione dell'acquisizione e rilascio delle singole risorse. Piuttosto, dovrebbe *delegare* questo compito ad opportune classi gestore (intuitivamente, una classe gestore per ogni tipologia distinta di risorsa) e focalizzarsi sull'uso appropriato delle risorse.

Torna all'indice

OCP (Open-Closed Principle)

Il principio "aperto-chiuso" è forse il più conosciuto dei principi SOLID. Ne sono state proposte due varianti: la prima si fa risalire al lavoro di Bertand Meyer (1988); la seconda (che è quella adottata nei principi SOLID) fu proposta da Robert C. Martin nel 1996, ma è di fatto una riformulazione di principi di progettazione proposti molti anni prima sotto altri nomi.

> Usando le parole di Martin: *"SOFTWARE ENTITIES (CLASSES, MODULES, FUNCTIONS, ETC.) SHOULD BE OPEN FOR EXTENSION, BUT CLOSED FOR MODIFICATION."*

Il principio dice che:

1. il software dovrebbe essere "aperto alle estensioni"
2. il software dovrebbe essere "chiuso alle modifiche"

In altre parole, un software progettato bene dovrebbe rendere semplice l'aggiunta di nuove funzionalità (scrivendo nuovo codice), senza che per fare ciò sia necessario modificare il codice esistente (chiusura alle modifiche).

Si noti che, nella sua enunciazione, il principio OCP non fornisce una metodologia esplicita per ottenere gli obiettivi che si propone, ragione per cui potrebbe sembrare inutile. In ogni caso, il principio può servire come linea guida per valutare quale, tra diverse alternative di progetto, soddisfa meglio i requisiti di apertura (alle estensioni) e chiusura (alle modifiche). In realtà, nei lavori che trattano del principio OCP, si dice esplicitamente che l'operazione chiave per ottenere un progetto aderente al principio suddetto è l'individuazione di quelle parti del software che, con probabilità alta, saranno oggetto di modifica in futuro e l'applicazione a queste parti di opportuni costrutti di astrazione (a volte detti costrutti per ottenere "information hiding").

Esempio: usiamo il principio OCP per confrontare le due varianti di progetto (old-style vs oo-style) del nostro esercizio Fattoria.

Iniziamo con il valutare quali parti del codice saranno (probabilmente) oggetto di cambiamento: come si era detto informalmente quando si era introdotto il problema, si prevede l'introduzione di nuovi animali, con caratteristiche più o meno diverse da quelli esistenti, che dovranno comunque essere utilizzabili nel "codice utente" (il codice che produce le strofe della canzoncina). Dobbiamo quindi valutare se il codice è aperto alle estensioni (aggiunta di nuovi animali) pur restando chiuso alle modifiche (il codice che genera la strofa della canzone e il codice degli animali preesistenti non dovrebbero essere modificati in seguito all'introduzione di nuovi animali).

Torna all'indice

Variante old-style

Nella variante old-style, esiste una unica classe Animale che implementa tutti gli animali concreti: l'aggiunta di un nuovo animale comporta la modifica di questa classe e, di conseguenza, la possibilità di modificare (magari inavvertitamente) il comportamento di uno degli animali preesistenti. In altre parole, il codice è aperto alle estensioni, ma è solo parzialmente chiuso alle modifiche: pur essendo vero che non dobbiamo modificare il codice che genera la strofa, ogni estensione che aggiunge un animale rischia di rompere il codice preesistente (inoltre, il codice che genera la strofa va comunque ricompilato, perché dipende direttamente dai dettagli implementativi della classe Animale).

Torna all'indice

Variante oo-style

Nella variante oo-style, invece, abbiamo una classe astratta Animale; questa fornisce l'interfaccia, ma non mostra alcun dettaglio implementativo. Il codice è aperto alle estensioni: per aggiungere un animale è sufficiente creare una nuova classe che implementa (mediante derivazione pubblica "IS-A") l'interfaccia astratta. Il codice è anche chiuso alle modifiche, perché queste aggiunte non hanno nessun impatto sul codice che genera la strofa e nemmeno sulle classi che implementano tutti gli altri animali. Quando si aggiunge un animale, non c'è nemmeno bisogno di ricompilare il codice che genera la strofa: deve essere solo ricollegato. In linea di principio, team di sviluppatori diversi potrebbero generare varianti diverse degli animali, senza dovere condividere il codice sorgente (solo l'interfaccia astratta).

Quindi, fissato il tipo di modifica "aggiunta di nuovi animali concreti", possiamo affermare che il progetto oo-style soddisfa il principio OCP in misura maggiore rispetto al progetto old-style.

> In entrambi i casi, l'aggiunta di nuovi animali concreti prevede piccole modifiche al modulo Maker.cc, per consentire ai nuovi animali di essere utilizzati dal programma. In altre parole, la "chiusura alle modifiche" non può mai essere totale: se vogliamo l'estendibilità devono sempre esistere dei punti in cui questa viene resa possibile; chiaramente, un buon progetto dovrebbe "confinare" il codice soggetto a modifiche in una zona ben delimitata e, per quanto possibile, piccola.

Torna all'indice

DIP (Dependency Inversion Principle)

Nella formulazione di Martin, il principio di inversione delle dipendenze viene enunciato in questo modo:

> "*I moduli di alto livello non devono dipendere da quelli di basso livello: entrambi devono dipendere da astrazioni. Le astrazioni non devono dipendere dai dettagli. Sono i dettagli che dipendono dalle astrazioni.*"

Il principio opera una classificazione sulle dipendenze tra moduli software (classi, funzioni, ecc.), stabilendo che alcune di queste dipendenze sono ammesse (in quanto inevitabili e tutto sommato innocue), mentre altre dipendenze sono da evitare (in quanto dannose).

Intuitivamente, le dipendenze "buone" sono quelle verso i concetti astratti; le dipendenze "cattive" sono quelle verso i dettagli implementativi.

Rimane da capire come mai il nome del principio parla di "inversione" delle dipendenze: è un punto importante, in quanto mette in evidenza l'aspetto metodologico nello sviluppo del software.

L'osservazione chiave è che, molto spesso, il software viene progettato e sviluppato seguendo un approccio top-down: partendo dal problema generale da risolvere, lo si suddivide in sottoproblemi più piccoli; la soluzione del problema generale si ottiene effettuando una opportuna composizione delle soluzioni dei sottoproblemi. Il processo viene ripetuto sui sottoproblemi, arrivando ad una stratificazione del codice, nella quale i moduli a livello più alto usano (e quindi *dipendono* da) i moduli a livello più basso. Si creano quindi naturalmente delle dipendenze (dall'astratto verso il concreto) che il principio DIP classifica come "cattive". Il principio DIP suggerisce quindi di "invertire" queste dipendenze, sostituendole con altre che invece non creano problemi (perché vanno dal concreto verso l'astratto).

A tale scopo, si individano alcune interfacce astratte, che non dipendono dai dettagli implementativi: i moduli di alto livello vengono modificati per usare (dipendere da) le interfacce astratte; analogamente, i moduli a basso livello vengono modificati per implementare (dipendere da) le interfacce astratte (realizzando quindi l'inversione). Complessivamente, si è eliminata la dipendenza dei moduli a alto livello dai moduli a basso livello. In particolare, si è migliorata anche l'aderenza del progetto al principio OCP, in quanto è ora possibile estendere il software, per esempio, consentendo la scelta di implementazioni alternative dell'interfaccia astratta senza influenzare i moduli ad alto livello.

Siccome il DIP si concentra sulle dipendenze tra moduli e queste dipendenze sono spesso rispecchiate dalla suddivisione in file del software (in particolare, dalle inclusioni di header file), a volte si dice che il principio DIP può essere visto come una reinterpretazione del principio OCP che si concentra sugli aspetti "sintattici".

Torna all'indice

LSP (Liskov Substitution Principle)

Questo principio prende il nome da Barbara Liskov, che nel 1987 aveva enunciato la nozione di "sostituibilità" per tipi di dato (nozione poi formalizzata in un articolo scritto insieme a Jeannette Wing).

Intuitivamente, si dice che S è un sottotipo di T se ad ogni modulo che usa un oggetto t di T è possibile passare (invece) un oggetto s di S ottenendo comunque un risultato equivalente, cioè un risultato che soddisfa le legittime aspettative dell'utente.

Martin ha riformulato il principio (in modo abbastanza grossolano) in questi termini:

> "*Ogni funzione che usa puntatori o riferimenti a classi base deve essere in grado di usare oggetti delle classi derivate senza saperlo.*"

Più propriamente, quello che si sta definendo è il cosiddetto "behavioral subtyping": le classi derivate (il sottotipo S) devono soddisfare le legittime aspettative degli utenti che accedono ad esse usando puntatori o riferimenti alle classi astratte (il tipo T). In altre parole, siccome S dichiara di essere in relazione "IS-A" rispetto a T, gli oggetti di tipo S non solo devono fornire (sintatticamente) i metodi forniti dalla classe base T, ma si devono anche comportare (behavior, aspetto semantico) come se fossero degli oggetti di tipo T.

Quindi il principio LSP può essere visto come una reinterpretazione del principio OCP che si focalizza sugli aspetti semantici (in precedenza avevamo notato che il DIP si focalizza solo sugli aspetti sintattici).

La corrispondenza del behavior non deve però essere intesa in senso assoluto (cioè, S non deve necessariamente essere identico a T): essa è limitata a quelle che sono le "aspettative legittime" che può avere un utente della classe T. Quali sarebbero queste legittime aspettative? Sono quelle stabilite dal *contratto* (precondizioni, invarianti e postcondizioni) che la classe T ha sottoscritto con i suoi utenti. Il principio LSP (e il behavioral subtyping) sono quindi in connessione stretta con la programmazione per contratto. Quando la classe derivata S dichiara di essere in relazione "IS-A" con la classe base T, di fatto si impegna a rispettare il contratto che T 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, rispetta il contratto stabilito da questa con i suoi utenti. Il contratto, purtroppo, non è stato stabilito esplicitamente, ma esiste e va rispettato, altrimenti l'utente si troverebbe di fronte a comportamenti erronei. Per esempio, è stato detto che il nome dell'animale deve essere il suo "nome comune" e che il genere (che serve a stabilire l'articolo indeterminativo da usare nella strofa) deve corrispondere al genere del nome comune (es., maschile per il cane, femminile per la volpe). Una classe concreta che, invece, fornisse il genere dell'animale (es., femminile per un cane femmina e maschile per una volpe maschio) violerebbe il principio LSP (e il contratto della classe base Animale).

Pur non essendo frequenti, le violazioni del principio LSP sono pericolose, perché non esistono modi semplici per rilevarle.

Un esempio classico di violazione del principio LSP è il seguente. Supponiamo che esista una classe Rettangolo, con la seguente interfaccia:

```
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 lungh * larg; }
};
```

Ad un certo punto viene richiesto di creare la classe Quadrato, dotata di una interfaccia simile. Siccome un quadrato è un tipo particolare di rettangolo, un programmatore (pigro) potrebbe pensare di implementare la classe quadrato usando l'ereditarietà pubblica e il polimorfismo dinamico. Le uniche accortezze tecniche sono quelle di dichiarare i metodi di Rettangolo come virtual (non puri, in quanto è una classe base concreta), di aggiungere il distruttore virtual e, infine, di fare l'override dei metodi `set_larghezza` e `set_lunghezza`, per assicurarsi che quando si modifica una dimensione sia modificata anche l'altra, così da mantenere l'invariante della classe Quadrato.

```
class Quadrato : public Rettangolo {
public:
 bool check_inv() const {
  return lung > 0 && lung == larg;
 }

 Quadrato(long lato) : Rettangolo(lato, lato) { }

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

Questo progetto viola il principio LSP. In particolare, NON è vero che a qualunque funzione che usa puntatori/riferimenti alla classe base (Rettangolo) noi possiamo passare invece puntatori/riferimenti alla classe derivata (Quadrato) e soddisfare le legittime aspettative dell'utente. In altre parole, NON è vero che un Quadrato "IS-A" Rettangolo, in quanto pur avendo la stessa interfaccia, non è equivalente dal punto di vista semantico.

Esempio: l'utente che usa la classe Rettangolo si aspetta che questo codice (in assenza di overflow) sia corretto, ovvero che l'asserzione sia sempre soddisfatta:

```
void raddoppia_area(Rettangolo& r) {
 long a_prima = r.get_area();
 long i = r.get_lunghezza();
 r.set_lunghezza(2 * i); // raddoppia la lunghezza
 long a_dopo = r.get_area();
 assert(area_dopo == 2 * a_prima);
 // .. altro codice
}
```

Se però alla funzione viene passato un riferimento a un Quadrato, il metodo `set_lunghezza()` raddoppierà sia la lunghezza che la larghezza e, di conseguenza, avremo una violazione dell'asserzione (perché `a_dopo == 4 * a_prima`).

Torna all'indice

Cosa è successo?

Semplicemente, la classe Quadrato ha violato il contratto (stabilito dalla classe Rettangolo) del metodo set_lunghezza. Il contratto stabilisce (nelle sue post-condizioni) che il metodo modifica *solo* la lunghezza del rettangolo, mentre l'overriding definito nella classe Quadrato modifica sia la lunghezza che la larghezza.

Si potrebbe obiettare: ma un quadrato deve avere i lati uguali. L'obiezione è sensata, ma sta ad indicare che la classe Quadrato NON può essere in relazione IS-A con la classe Rettangolo, ovvero che un Quadrato NON è un Rettangolo. Quando facciamo questa affermazione, chiaramente, non stiamo ragionando in puri termini geometrici, ma stiamo piuttosto considerando l'aspetto "behavioral" dei corrispondenti tipi di dato: un Quadrato non è un Rettangolo perché

esistono dei contesti (vedi la funzione di sopra) in cui un Quadrato NON si comporta come si comporterebbe un Rettangolo.

La classe base stabilisce un contratto per ognuno dei suoi metodi, in termini di pre-condizioni e post-condizioni. Cosa deve fare la classe derivata per soddisfare tale contratto? Non è necessario che il contratto stabilito dalla classe derivata sia identico, ma la classe derivata deve fornire come minimo tutte le garanzie fornite dalla classe base. Quindi, la classe derivata può *indebolire* le precondizioni (cioè fornire una implementazione per più casi rispetto a quelli previsti dalla classe base) e può *rafforzare* le postcondizioni (cioè fornire all'utente garanzie ulteriori oltre a quelle garantite dalla classe base). Nel caso analizzato nell'esempio, le post-condizioni dei metodi `set_lunghezza` e `set_larghezza` sono state modificate (rendendole incompatibili, non rafforzandole), da cui la violazione del contratto e, di conseguenza, del principio di sostituibilità di Liskov.

Torna all'indice

ISP (Interface Segregation Principle)

Il principio di separazione delle interfacce dice che l'utente non dovrebbe essere forzato a dipendere da parti di una interfaccia che non usa. Di conseguenza, il progettista di una interfaccia dovrebbe fare il possibile per *separare* quelle porzioni che potrebbero essere usate separatamente le une dalle altre, ovvero a preferire tante interfacce "piccole" (thin interfaces) rispetto a poche interfacce "grandi" (fat interfaces).

Aderendo a questo principio, si ottengono i seguenti benefici:

1. l'implementatore può implementare separatamente le interfacce piccole, evitando che un errore su una di queste si propaghi sulle altre;
2. l'implementatore può decidere di implementare solo alcune delle interfacce piccole ottenute dalla separazione dell'interfaccia grande;
3. se una delle interfacce piccole dovesse cambiare, l'utente che non la usa (perché usa le altre) non ne è minimamente influenzato; al contrario, adottando una sola interfaccia grande, una modifica su una sua parte influenza anche gli utenti che NON fanno alcun uso di quella parte.

Dal punto di vista tecnico, l'applicazione del principio ISP presuppone la possibilità di utilizzare l'ereditarietà multipla (di interfaccia) e, in effetti, può essere considerato l'esempio più frequente di uso appropriato dell'ereditarietà multipla. E' quindi importante studiare i corrispondenti aspetti tecnici dal punto di vista del linguaggio.

Si noti infine che il principio ISP può essere interpretato come una forma particolare del principio SRP, che si concentra al caso specifico della progettazione delle interfacce.

Torna all'indice