

Appunti di Progettazione del Software

Canale 2 · Prof. Massimo Mecella

Capitolo 2: Modularizzazione e orientamento agli oggetti

Autore: Tommaso C.

Appunti tratti dalle lezioni in aula e dal materiale del corso

Argomenti: Moduli, servizi e interfaccia · Principi della modularità

Un esempio di cattiva e buona modularizzazione

Approccio procedurale e a oggetti · Classi e oggetti

Ereditarietà e polimorfismo

Indice

1	La modularizzazione	1
2	I principi della modularità	2
3	Un esempio: la ricerca in una lista	4
3.1	La versione modularizzata male	4
3.2	La versione modularizzata bene	6
4	Approccio procedurale e approccio a oggetti	7
5	Classi e oggetti	8
6	Ereditarietà e polimorfismo	9

1 La modularizzazione

Definizione – Modularizzazione e modulo

La **modularizzazione** è il principio secondo cui il software si struttura in unità, dette **moduli**. Un modulo è un'unità di programma che:

- ha un **obiettivo chiaro**, descrivibile con una frase come “il modulo fa X”;
- ha relazioni strutturali con altri moduli;
- offre agli altri moduli un insieme ben definito di **servizi**;
- può usare i servizi di altri moduli.

Invece di un unico blocco, come un **main** di diecimila righe, il programma si divide in moduli. È il principio che rende possibili le qualità del software del Capitolo 1, in particolare la strutturazione e la modularità.

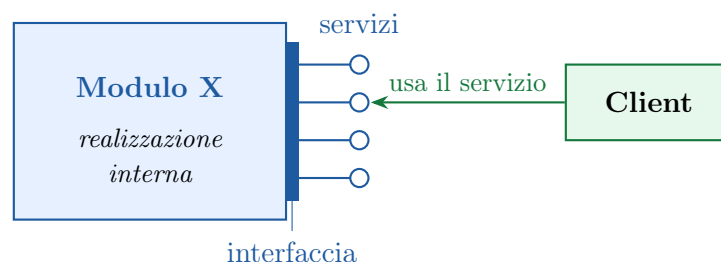
Definizione – Servizio, interfaccia, client e server

Un **servizio** è un'operazione elementare che un modulo offre agli altri. L'insieme dei servizi offerti è l'**interfaccia** del modulo. Il modulo che offre un servizio è il **server** (o **servant**); quello che lo chiede è il **client**.

Nota – Server e servant

Server indica anche la macchina fisica che offre servizi. Chi vuole distinguere usa *servant* (servente) per il modulo software e *server* per la macchina; in questo corso i due termini sono sinonimi. *Client* indica invece sia il software sia la macchina che chiede i servizi: un'alternativa non c'è.

Un modulo non ha un ruolo fisso: è server nelle relazioni in cui offre un servizio e client in quelle in cui lo chiede. Su questa relazione si fonda l'**architettura client-server** del software: un modulo chiede, un altro risponde, e la computazione avanza lungo la catena di richieste e risposte. Per capire che cosa fa un programma bisogna saperla seguire.



Un modulo: l'interfaccia (la barra) espone i servizi, disegnati come lecca-lecca (lollipop); il client si collega al servizio che usa, e la realizzazione interna non si vede.

Osservazione – Che cosa, non come

Il client deve sapere *che cosa* fa un modulo, cioè la sua interfaccia, non *come* lo fa. Finché l'interfaccia resta la stessa, la realizzazione interna può cambiare, per esempio con un algoritmo migliore, senza toccare i client.

Su questo si regge la manutenzione. Se un intervento cambia l'interfaccia di un modulo, vanno cambiati i suoi client; ciascuno di loro può essere server di altri moduli, che a loro volta vanno cambiati: è un effetto domino. Con interfacce ben progettate e moduli dall'obiettivo preciso, l'intervento resta confinato nel modulo da correggere.

2 I principi della modularità

Definizione – I quattro principi della modularità

- **Alta coesione** (o unitarietà, incapsulamento): il modulo corrisponde a un'unità concettuale ben definita e ne contiene tutti e soli gli aspetti.
- **Basso accoppiamento**: il modulo comunica con il minor numero possibile di moduli e scambia con loro solo l'informazione necessaria.
- **Interfacciamento esplicito**: è chiaro come si usa il modulo, cioè quali servizi offre e quali chiede.
- **Information hiding**: ciò che non serve agli altri moduli resta privato del modulo.

In breve: omogeneità interna, indipendenza dagli altri moduli, chiare modalità d'uso, poco rumore nella comunicazione.

Alta coesione. Un modulo rappresenta un solo concetto, che può essere un dato (il cliente, il conto corrente) o un gruppo di operazioni (per esempio solo operazioni matematiche). In un'applicazione bancaria un modulo raccoglie le operazioni tipiche del conto corrente, un altro quelle del cliente. Un modulo unico per clienti e conti mescolerebbe due concetti, e ogni intervento di manutenzione dovrebbe prima separarli.

Basso accoppiamento. Un modulo isolato non serve a niente, ma non è ben strutturata nemmeno un'architettura in cui tutti parlano con tutti. Ogni comunicazione ha un costo, in memoria, tempo di CPU e lavoro di sviluppo: un modulo parla solo con i moduli che gli servono e rende esplicite solo le informazioni che servono agli altri.

Interfacciamento esplicito. Un modulo si usa solo invocando i suoi servizi, con i dati passati come parametri. In C lo si ottiene in parte con i file `.h`, che specificano le operazioni, separati dai file `.c`, che le realizzano.

Information hiding. Un modulo che ordina dei dati offre il servizio `ordina`: riceve i dati e li restituisce ordinati. Se usa bubble sort o merge sort, e con quali variabili di lavoro, è un dettaglio della realizzazione che i client non vedono.

Osservazione – Le variabili globali

Una variabile globale viola due principi:

- l'**information hiding**, perché un dato che serve a un solo modulo diventa visibile a tutti;
- l'**interfacciamento esplicito**, perché chiunque può modificarla con un *side effect*: nasce una dipendenza implicita, difficile da scoprire leggendo il codice.

Il C permette di scrivere tutto in un `main` e di usare variabili globali: non impedisce la cattiva modularizzazione. Nei linguaggi orientati agli oggetti, come Java, il codice sta sempre dentro una classe, cioè dentro un modulo: è il linguaggio stesso a indurre un progetto modulare, ed è una delle ragioni della loro diffusione.

I principi si riconoscono anche in situazioni della vita quotidiana.

Principio	Esempio
Alta coesione, violata	allo stesso sportello postale si pagano i bollettini e si ritirano le pensioni: servizi disomogenei
Basso accoppiamento, violato	per alcuni servizi dell'anagrafe serve una marca da bollo, che si compra dal tabaccaio: se il tabaccaio è chiuso, si blocca anche l'anagrafe
Interfacciamento esplicito	cartelli chiari dicono quali servizi offre ciascuno sportello
Information hiding	chi usa una carta di credito non sa che cosa è memorizzato nella banda magnetica o nel chip: gli bastano la carta e il PIN

L'information hiding permette di cambiare la realizzazione. Il telefono usa la SIM solo attraverso la sua interfaccia, senza sapere come è fatta dentro: per questo la SIM fisica si è potuta sostituire con la eSIM, interamente digitale.

Osservazione – Modularità ed efficienza

La modularità può costare efficienza: uno sportello per ogni servizio richiede più personale di uno sportello che fa tutto. Come per le qualità in contrasto del Capitolo 1, si sceglie secondo l'obiettivo. Nel software di controllo di un treno conta prima l'efficienza, e si usa spesso il C, vicino alla macchina; i linguaggi a oggetti sono in genere un po' più lenti.

3 Un esempio: la ricerca in una lista

Il servizio **cerca** dice se un intero compare in una lista collegata. La lista è fatta di nodi **Lista**, ciascuno con un campo **info**, il valore, e un campo **next**, il nodo successivo; la lista vuota è **null**. Il client legge la lista e l'intero da cercare, chiama il servizio e stampa se il valore è presente.

Il codice è in Java, ma per ora si legge come il C: **public static** si può ignorare, **boolean** è il tipo dei valori vero e falso, e **Server.alfa** è la variabile **alfa** del modulo **Server**.

3.1 La versione modularizzata male

```
public class Server {
    public static int alfa;                // (1)

    public static boolean cerca(Lista x) { // (2)
        do {                               // (3)
            if (x.info == alfa) return true;
            x = x.next;
        } while (x != null);
        return false;
    }
}
```

```

public class Client {
    private static Lista leggiListaCollegataInteri() { /* ... */ }
    private static int leggiIntero() { /* ... */ }

    public static void main(String[] args) {
        Lista s = leggiListaCollegataInteri();
        if (s != null) {                                // (4)
            Server.alfa = leggiIntero();                // (1)
            if (Server.cerca(s))
                System.out.print(Server.alfa + " presente");
            else
                System.out.print(Server.alfa + " non presente");
        }
    }
}

```

Punto	Difetto	Principio
(1)	il valore da cercare passa per la variabile globale alfa : il client deve sapere che va assegnata prima di chiamare cerca	alto accoppiamento
(2)	cerca riceve solo la lista: dalla sua intestazione non si capisce che il valore cercato è alfa	interfacciamento non esplicito
(3)	il ciclo legge x.info prima di controllare x : con la lista vuota la ricerca fallisce	bassa coesione: il controllo del server resta al client
(4)	il client deve controllare che la lista non sia vuota	alto accoppiamento

Il punto (3) rende la ricerca poco robusta: funziona solo se il client ha fatto la sua parte, e un client che passa una lista vuota la manda in errore. Ogni regola d'uso nascosta va poi scritta nella documentazione ("prima di chiamare **cerca** assegna **alfa**; non passare una lista vuota") e letta da chi usa il modulo. Su migliaia di moduli è un costo enorme, e ogni regola dimenticata diventa un errore.

3.2 La versione modularizzata bene

```
public class Server {  
    public static boolean cerca(Lista x, int alfa) {  
        while (x != null) {  
            if (x.info == alfa) return true;  
            else x = x.next;  
        }  
        return false;  
    }  
}
```

```
public class Client {  
    private static Lista leggiListaCollegataInteri() { /* ... */ }  
    private static int leggiIntero() { /* ... */ }  
  
    public static void main(String[] args) {  
        Lista s = leggiListaCollegataInteri();  
        System.out.print("Inserisci un intero: ");  
        int alfa = leggiIntero();  
        if (Server.cerca(s, alfa))  
            System.out.print(alfa + " presente");  
        else  
            System.out.print(alfa + " non presente");  
    }  
}
```

L'intestazione `cerca(Lista x, int alfa)` dice da sola come si usa il servizio: il codice si **autodocumenta**. Il server controlla la lista vuota, che non contiene nessun valore, e restituisce **false**; il client legge i dati, chiama il servizio e stampa il risultato, senza sapere nulla di come si svolge la ricerca.

Osservazione – Stesso risultato, qualità diversa

Su una lista non vuota i due programmi danno lo stesso risultato. La seconda versione ha però alta coesione, basso accoppiamento e interfacciamento esplicito, ed è più robusta: anche un programma molto piccolo si può modularizzare male.

4 Approccio procedurale e approccio a oggetti

L'approccio a oggetti rovescia il rapporto fra procedure e dati dell'approccio procedurale.

	Approccio procedurale	Approccio a oggetti
Si parte da	le procedure: come si svolgono le operazioni	gli oggetti da manipolare, cioè i concetti del dominio
Poi si definiscono	le strutture dati che servono alle procedure	le operazioni su ciascun oggetto
Più adatto a	software algoritmico, con forti vincoli di efficienza (calcolo fisico)	software gestionale, che rappresenta in digitale una parte del mondo (banche, assicurazioni, commercio elettronico)

Nell'approccio procedurale, come in C, una **procedura** è un blocco di codice che non restituisce un valore (`void f(...)`), una **funzione** un blocco che lo restituisce (`int f(...)`); spesso si chiamano entrambe procedure.

Esempio – Il trasferimento bancario

Un'applicazione bancaria a oggetti comincia dai concetti della banca: il cliente, il conto corrente, il trasferimento, il deposito. Del trasferimento non si pensa prima l'algoritmo, ma che cosa serve: due conti, il mittente e il destinatario, che nel programma saranno due oggetti.

Il software gestionale si presta all'approccio a oggetti perché il programma costruisce un mondo virtuale che rispecchia quello reale: gli oggetti del sistema sono i concetti del dominio, e chi programma li riconosce subito. Chi progetta deve comunque conoscere entrambi gli approcci e scegliere secondo il problema.

Definizione – Approccio orientato agli oggetti

L'approccio **orientato agli oggetti** (*object orientation*, OO):

- modella la realtà di interesse nel modo più diretto e astratto possibile, con **classi** (categorie di oggetti), **oggetti** e relazioni;
- costruisce il programma in moduli, e ogni classe è un modulo;
- lega gli aspetti strutturali, i dati, a quelli comportamentali, le operazioni;
- fa avanzare la computazione con oggetti che si scambiano messaggi, cioè richieste di servizi, come client e server.

5 Classi e oggetti

Definizione – Oggetto, classe, istanza

Un **oggetto** è un'entità che contiene dati e su cui si eseguono operazioni: mette insieme dato e operazione. Una **classe** raggruppa oggetti simili e ne definisce le caratteristiche comuni:

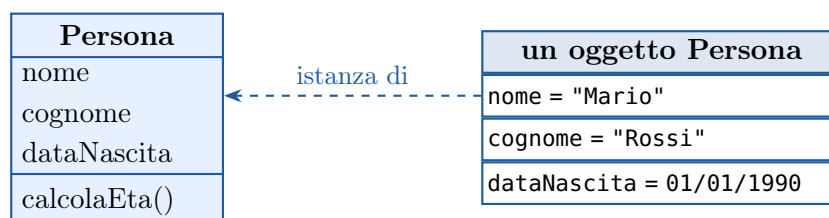
- gli **attributi** (o proprietà), le caratteristiche strutturali, cioè i dati di ogni oggetto;
- i **metodi** (o operazioni), le caratteristiche comportamentali, cioè i servizi che la classe offre.

Ogni oggetto è un'**istanza** di una classe.

Astrazione. La classe nasce da un'**astrazione**: si trascurano i dettagli e si cercano le caratteristiche comuni. Tutte le persone hanno un nome, un cognome, un colore degli occhi: i valori cambiano da una persona all'altra, il tipo di caratteristica è lo stesso per tutte. L'insieme di queste caratteristiche definisce la classe Persona.

Attributi e metodi. La classe Persona ha gli attributi **nome**, **cognome** e **dataNascita** e il metodo **calcolaEta()**. Un metodo è una procedura o una funzione, con un tipo restituito e una **segnatura**: il nome e il numero e il tipo dei parametri. A differenza di una funzione del C, però, appartiene a una classe e si invoca solo su un oggetto di quella classe: le operazioni stanno con i dati che trattano, e coesione e information hiding ne guadagnano.

Un oggetto, nella memoria dell'elaboratore, è una zona organizzata secondo la struttura della sua classe: un oggetto Persona ha un campo per il nome, uno per il cognome e uno per la data di nascita, ciascuno con il proprio valore.



La classe Persona e una sua istanza: l'oggetto ha un campo per ogni attributo della classe, con i propri valori.

Osservazione – Oggetti e **struct** del C

In C un **typedef** di una **struct** raggruppa più dati sotto un solo nome, ma nulla lega a quel tipo le funzioni che lo trattano: quali siano le sue operazioni lo sanno solo chi programma e la documentazione. In una classe dati e metodi sono definiti insieme, e un oggetto supporta solo i metodi della sua classe: invocarne un altro è un errore che segnala il compilatore.

Osservazione – Esistono solo gli oggetti

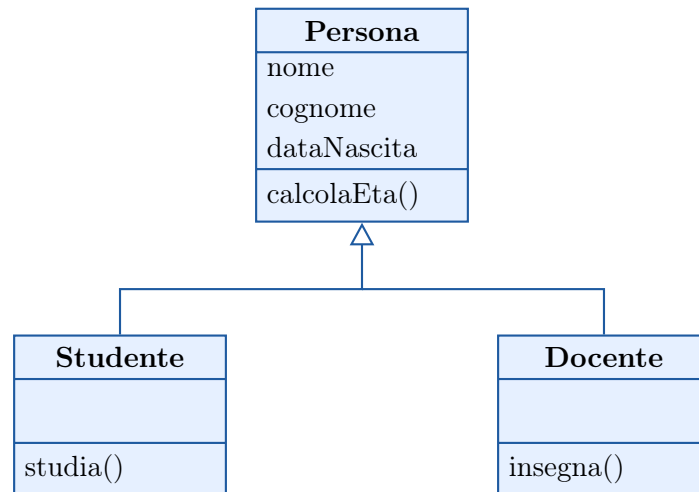
Nella realtà esistono solo gli oggetti, e anche nell'applicazione in esecuzione ciò che lavora sono le istanze. Le classi sono lo strumento con cui si progetta: descrivono la struttura che i loro oggetti seguiranno. Si progetta pensando alle classi, non ai singoli oggetti.

6 Ereditarietà e polimorfismo

Definizione – Ereditarietà

L'**ereditarietà** permette di definire una classe come caso particolare di una più generale: la **sottoclasse** riceve (*eredita*) attributi e metodi della classe generale, senza riscriverli, e aggiunge solo le caratteristiche proprie. Le classi formano così una **gerarchia di ereditarietà**, un albero con la classe più generale alla radice.

Studenti e docenti sono persone. Gli attributi **nome**, **cognome** e **dataNascita**, con il metodo **calcolaEta()**, si scrivono una volta sola in **Persona**, e **Studente** e **Docente** li ereditano. In **Studente** si aggiunge solo ciò che è proprio degli studenti, come il metodo **studia()**; in **Docente** il metodo **insegna()**. Ereditare vuol dire riusare, non tramandare nel tempo.



*Una gerarchia di ereditarietà: **Studente** e **Docente** ereditano attributi e metodi di **Persona** e aggiungono ciascuna il proprio metodo. La freccia va dalle sottoclassi alla classe più generale.*

Definizione – Polimorfismo

Il **polimorfismo** è la possibilità di chiedere la stessa operazione, con lo stesso nome, a oggetti di classi diverse: ciascuno la esegue con il metodo della propria classe.

Chi chiede l'operazione non deve sapere di quale classe sia l'oggetto né come l'operazione sia realizzata. Se più classi di docenti realizzano ciascuna a modo proprio il metodo `insegna()`, un programma può chiedere `insegna()` a ogni docente di un elenco, e ognuno risponde come prevede la sua classe.

Riepilogo

- **Modulo:** unità di programma con un obiettivo chiaro, che offre servizi (server) e ne usa altri (client).
- **Interfaccia:** l'insieme dei servizi offerti; il client sa che cosa fa il modulo, non come.
- **Principi della modularità:** alta coesione, basso accoppiamento, interfacciamento esplicito, information hiding.
- **Variabili globali:** violano information hiding e interfacciamento esplicito.
- **Modularità ed efficienza:** possono essere in contrasto; si sceglie secondo l'obiettivo.
- **Procedurale e a oggetti:** si parte dalle procedure oppure dagli oggetti del dominio.
- **Classe e oggetto:** la classe definisce attributi e metodi, l'oggetto ne è un'istanza.
- **Ereditarietà e polimorfismo:** la sottoclasse riusa ciò che è comune; la stessa operazione si esegue con il metodo della propria classe.