♦ PROGETTAZIONE TOP-DOWN E BOTTOM-UP

Top-down e **bottom-up** sono strategie di elaborazione dell'informazione e di gestione delle conoscenze, riguardanti principalmente il software e, per estensione, altre teorie umanistiche le teorie dei sistemi.

Nel modello **top-down** è formulata una visione generale del sistema senza scendere nel dettaglio di alcuna delle sue parti. Poi ogni parte può essere nuovamente rifinita, specificando ulteriori dettagli finché la specifica completa è sufficientemente dettagliata da validare il modello.

In contrasto con il modello top-down c'è la progettazione **bottom-up**, nella quale parti individuali del sistema sono specificate in dettaglio. Queste parti vengono poi connesse tra loro in modo da formare componenti più grandi, che vengono a loro volta interconnessi fino a realizzare un sistema completo.

Resta tuttavia aperta una questione: come facciamo a suddividere un problema grosso in sotto-problemi di dimensioni inferiori? Esistono delle pratiche che facilitino l'analisi e la formalizzazione di problemi?
Queste pratiche esistono, e sono oggetto di studio di una scienza, detta **"Ingegneria del Software".**
Noi ci limitiamo ad analizzare i due approcci più in uso nella programmazione strutturata: **Top Down** e **Bottom Up**.
Il principio fondante della programmazione strutturata è che sono sufficienti appena tre strutture di controllo per realizzare qualunque tipo di programma: la *sequenza*, l'*iterazione* e la *selezione*. Le istruzioni possibili appartengono a loro volta a tre sole categorie: *input*, *output* ed *assegnamento*. È possibile applicare i costrutti fondamentali di sequenza, iterazione e selezione in ciascun sottoprogramma accessibile da un programma.

La **programmazione top-down** è uno stile di programmazione, fondamento dei tradizionali **linguaggi procedurali**, nel quale la progettazione inizia specificando parti complesse e suddividendole successivamente in parti più piccole. Eventualmente, i componenti sono specificati quanto basta per la codifica ed il programma viene anche scritto. Si segue quindi una scomposizione iterativa di un problema in sottoproblemi: si parte dall'alto (problema nella sua interezza) e si arriva in basso (problema visto come insieme di sottoproblemi elementari).

La tecnica per la scrittura di un programma mediante l'utilizzo dei metodi **top-down** indica di scrivere una procedura principale che indica dei nomi per le principali funzioni di cui avrà bisogno. In seguito, il gruppo di programmazione esaminerà i requisiti di ognuna di queste funzioni ed il processo verrà ripetuto. Queste sotto-procedure a comparto eseguiranno eventualmente azioni così elementari che porteranno ad una codifica semplice e concisa. Quando tutte le varie sotto-procedure sono state codificate, il programma è realizzato.

Questo è l'esatto opposto della **programmazione bottom-up** che è comune nei **linguaggi orientati agli oggetti** come C++ o Java. Essa ha l'obiettivo di individuare algoritmi utili e di generalizzarli in modo da creare un insieme di funzioni di libreria.

Ogni linguaggio che si rispetti comprende per default un buon numero funzioni di libreria, che coprono le esigenze più svariate. Nonostante la dimensione di tali librerie, esistono ancora oggi delle circostanze in cui ha un senso realizzare librerie per uso personale. La particolarità di una funzione di libreria è la sua versatilità: essa non si deve limitare a risolvere un particolare problema, ma una intera classe di problemi. Un tipico esempio di funzioni riutilizzabili sono le librerie di funzioni matematiche: il loro uso spazia dal calcolo scientifico a quello commerciale, dalla grafica 3D ai videogames.

Vantaggi della programmazione Top-down

- permette di concentrarsi in ogni momento del progetto sugli aspetti più significativi, rimandando a momenti successivi gli aspetti di maggior dettaglio;
- consente di dare una descrizione dell'algoritmo risolutivo piu' leggibile;
- essendo ciascun sottoproblema indipendente dagli altri, la sua risoluzione puo' essere modificata, se necessario, senza che si modifichi la struttura generale dell'algoritmo risolutivo (migliore manutenzione dei programmi);
- si puo' suddividere il compito di risolvere il problema tra piu' persone o gruppi di lavoro;
- la risoluzione di sottoproblemi puo' essere riutilizzata in altri problemi (riduzione dei costi di realizzazione di un programma).

Svantaggi

- La programmazione top-down può complicare la fase di test, dato che non esisterà un eseguibile finché non si arriverà quasi alla fine del progetto.
- La programmazione bottom-up agevola il test di unità, ma finché il sistema non si unisce non può essere testato nella sua interezza, e ciò causa spesso complicazioni verso la fine del progetto "Individualmente ci siamo, insieme falliamo."

Limiti della programmazione Top Down

Un tempo si pensava che l'approccio Top Down fosse la chiave per affrontare qualunque problema di programmazione. Ma a partire dagli anni '80 emersero almeno due grossi problemi: in primo luogo, l'approccio Top Down è ideale per problemi di dimensioni contenute, risolvibili con programmi da poche migliaia di righe di codice. Di contro, diventa quasi impraticabile su programmi di milioni di righe, sempre più comuni ai giorni nostri. In secondo luogo, la programmazione strutturata non offre un valido supporto alla progettazione di strutture di dati, uno degli aspetti più importanti della programmazione. La progettazione di strutture dati con linguaggi procedurali è un tema talmente complesso da scoraggiarne la trattazione anche solo in forma superficiale. I **linguaggi ad oggetti** offrono un approccio molto più intuitivo e naturale all'argomento in quanto la programmazione ad oggetti è basata largamente sul principio della composizione e dell'astrazione delle strutture dati.

Infine, la progettazione Top Down presuppone che il programmatore affronti ogni nuovo problema partendo da zero, cosa che nella realtà si verifica assai di rado. Pensiamo ad un architetto che progetta una casa: sebbene ogni casa sia diversa dalle altre, è anche vero che in fase di progettazione ciascuna di esse presenta lo stesso tipo di problematiche. Al momento di progettare una casa, l'architetto non è costretto ogni volta a considerare che la casa deve assolvere, tra gli altri, anche il compito di permettere l'igiene personale, e da questo dedurre che la soluzione più adatta sia di costruire una stanza apposta. Al contrario, egli sa con certezza che ogni casa deve possedere un bagno, e si limita a prendere delle decisioni sulla disposizione del bagno in rapporto agli altri locali, o a stabilire come disporre i vari elementi all'interno del bagno stesso. In conclusione, l'architetto ricorre ad un approccio basato sulla conoscenza di determinati schemi progettuali predefiniti, che vengono raffinati attraverso la composizione di oggetti preesistenti (come lavandini o la vasche da bagno). Questo approccio è attuabile ad ogni livello del progetto, dalla singola camera, all'appartamento fino ad arrivare a comprendere un intero edificio. A ciascuno di questi livelli molti requisiti sono ben noti, e pertanto non è necessario re inventarli.

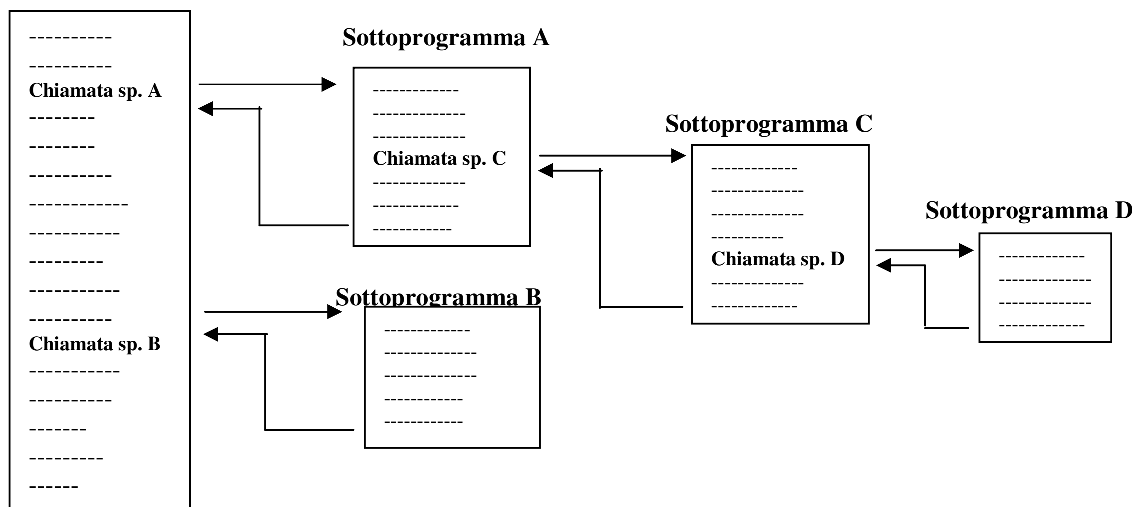
La programmazione ad oggetti, come vedremo, è basata largamente sul principio della composizione e dell'astrazione delle strutture dati. L'approccio Top Down è comunque sempre presente, anche se su scala minore, in determinati passaggi della stesura del codice.

♦ TECNICHE DEI SOTTOPROGRAMMI (METODOLOGIA TOP-DOWN)

La scomposizione di un *problema* P in *sottoproblemi*, che possono essere risolti in modo indipendente all'interno di P, permette di costruire programmi con strumenti informatici specifici che prendono il nome di **tecniche dei sottoprogrammi**.

Il programma che risolve il problema P sarà quindi costituito da:

- **un programma principale** (main), avente la funzione di risolvere globalmente il *problema* P,
- **uno o più sottoprogrammi** la cui funzione è quella di risolvere i *sottoproblemi* contenuti in P.



Un sottoprogramma può essere:

- una **funzione** se, quando termina, *restituisce* al programma chiamante **un valore**,
- una **procedura** se, quando termina, *non restituisce* al programma chiamante **nessun valore**.

Ogni sottoprogramma è identificato da un **nome** a cui un'istruzione del programma principale, detta **chiamata**, fa riferimento in ogni punto del programma in cui è richiesta l'esecuzione del sottoprogramma chiamato. Anche nei sottoprogrammi si possono trovare delle **chiamate** ad altri sottoprogrammi.
Durante l'elaborazione del programma principale il computer, ogni volta che incontra un'istruzione di chiamata ad un sottoprogramma, sospende l'elaborazione del programma principale ed esegue le istruzioni del sottoprogramma chiamato, per poi tornare ad elaborare il programma principale a partire dalla prima istruzione successiva a quella della chiamata.

In generale un sottoprogramma implementa un algoritmo che opera su alcuni *valori*, detti *dati di ingresso* o *parametri di ingresso*, che il sottoprogramma riceve dal programma principale (o dal sottoprogramma chiamante) mediante l'istruzione di chiamata. Dopo aver ricevuto i dati di ingresso, il sottoprogramma calcola dei *valori*, detti *dati di uscita* o *parametri di uscita*, e restituisce tali valori al programma principale (o al sottoprogramma chiamante). *Ci sono casi in cui il sottoprogramma non ha né dati di ingresso né dati di uscita.*

Il programma principale comunica con il sottoprogramma mediante una lista di variabili, dette **parametri attuali** o **effettivi,** che vengono specificati nella **chiamata al sottoprogramma** e destinati a contenere i *dati di ingresso* su cui il sottoprogramma deve operare.

Il sottoprogramma contiene all'inizio della sua definizione una lista di variabili, dette **parametri formali,** che servono a contenere i *dati di ingresso* del sottoprogramma che saranno riempiti con i valori dei *parametri attuali* del programma chiamante.

In C++ ogni **sottoprogramma** è **una funzione**.

In C++ anche il **programma principale** è **una funzione** che, in genere, non ha nessun dato di ingresso e nessun dato di uscita (void main ( )).

Esempio

```
int max (int , int );

void main( )
{ int N, M, X;
  cout<<"Inserisci un primo valore intero : " ;
  cin >> N;
  cout<<"Inserisci un secondo valore intero : " ;
  cin >> M;
  X= max (N , M);      // chiamata alla funzione di
                       // nome max  alla quale vengono
                       // passati i valori contenuti nei
                       // parametri  attuali  N  e  M
  cout<< "\n Il valore maggiore è : "<<X ;
     ⋮

}
```

```
// funzione max  che ha come
// parametri formali A e B

int max (int A, int B)
    {
     if   (A>=B)
           return A;
     else
          return B;
    }
```

♦ IMPLEMENTAZIONE DELLE FUNZIONI IN C++

In C++ ogni funzione ha bisogno di due parti:

- la **dichiarazione della funzione** per mezzo di **un prototipo**
- la **definizione della funzione** stessa

▪ Il **prototipo** di una funzione viene specificato nel seguente modo:

tipo_valore_restituito nome_funzione **(** ***tipo_param1*****,** ***tipo_param2*****, …..);**

e viene posto subito prima del programma principale (main).

Il ***tipo_valore_restituito*** indica il tipo del dato in uscita (restituito al programma chiamante) e può essere uno qualsiasi dei tipi di dati ammessi in C++ (ad es. *int* o *float* o *long int* ….); se non viene specificato è sottinteso ***int*** ; per indicare che la funzione non restituisce nessun valore si deve scrivere ***void*** (ad es. *void main ( )*).

Il ***nome_funzione*** è il nome che il programmatore assegna alla funzione e che usa nelle **chiamate**.

I ***tipo_param1*****,** ***tipo_param2, …..*** indicano l'elenco dei *tipi* dei *parametri di ingresso*. Quindi se i parametri di ingresso sono 3 devono essere specificati 3 tipi (ad es. *int*, *int*, *float*). Se non vi sono parametri di ingresso non si scrive nessun tipo.

▪ La **definizione della funzione** viene posta dopo la definizione della funzione *main,* cioè dopo il programma principale ed ha la seguente struttura:

tipo_valore_restituito nome_funzione (*tipo_param1* ***nome_param1***, *tipo_param2* ***nome_param2***, …..)
{
corpo della funzione
}

tipo_param1 ***nome_param1,*** *tipo_param2* ***nome_param2*** … sono i nomi dei *parametri di ingresso* preceduti ciascuno dal *tipo* che è già stato specificato nel prototipo della funzione.

Il ***corpo*** della funzione è composto da:

a) eventuali *dichiarazioni di variabili locali* alla funzione, cioè variabili usate e conosciute solo all'interno della funzione;
b) le istruzioni che realizzano la funzione;
c) una o più istruzioni del tipo: ***return*** *espressione* che restituiscono il *valore* (*parametro di uscita*) al programma chiamante; se la funzione ha come *tipo_valore_restituito void* si mette solo **return** senza che sia seguito da alcun valore.

Esempio di programma in C++ strutturato in sottoprogrammi

```
/* Date da tastiera le misure dei cateti di un triangolo rettangolo visualizza
   i valori dell'ipotenusa, dell'area e dell'altezza relativa all'ipotenusa */

#include <iostream>
#include <math>
using namespace std;

float ipotenusa (float , float);                  // Dichiarazione delle
float area (float , float);                       // funzioni
float altezza (float, float);

int main()                                        // Programma principale
{float c1, c2, ar, ip, alt;

 // richiesta dati
 do
   {cout << "Inserisci il primo cateto: ";
    cin >> c1;
   } while(c1<=0);
 do
   {cout << "Inserisci il secondo cateto: ";
    cin >> c2;
   } while(c2<=0);
 // Determina area, ipotenusa e altezza
 ar = area ( c1, c2 );

 ip = ipotenusa ( c1, c2 );

 alt= altezza ( ar, ip );

 // Visualizza i risultati
 cout <<"\nL'ipotenusa è: "<< ip ;
 cout <<"\nL'area è: "<< ar ;
 cout <<"\nL'altezza è: "<< alt << endl;
 system ("pause");
}

// Definizione di funzione
// Restituisce il valore dell'ipotenusa
// dati i valori dei cateti
float ipotenusa (float a, float b)
{float q, ipo;
 q   =a*a + b*b;
 ipo =sqrt(q);
 return ipo;
}

// Definizione di funzione
// Restituisce l'area del triangolo dati
// i cateti
float area (float c , float d)
{float a ;
 a = c*d/2;
 return a ;
}

// Definizione di funzione
// Restituisce l'altezza relativa all'ipotenusa
// data l'area e l'ipotenusa
float altezza (float area_tri, float ipo_tri)
{float h;
 h = area_tri * 2 /ipo_tri;
 return h;
}
```

Prototipi delle funzioni

Variabili locali al **main** e quindi conosciute solo nel main

Parametri attuali

Chiamate alle funzioni

Variabili locali alla funzione e quindi conosciute solo in ipotenusa

Parametri formali (***parametri di ingresso***)

(***parametri di uscita***)