

Programmazione modulare

Programmazione in grande

Lo sviluppo di applicazioni di elevata complessità richiede:

- Progetti collettivi "⇒" suddivisione dei programmi in parti separatamente sviluppabili
- Alti costi di sviluppo "⇒" vita del software più lunga

Requisiti dei prodotti software

- Affidabilità
- Efficienza
- Modificabilità
- Estensibilità
- Riutilizzo

Modulo

Il concetto di **modulo** si propone come soluzione per risolvere i problemi introdotti dalla programmazione in grande e per supportare i requisiti di modificabilità, estensibilità e riutilizzo.

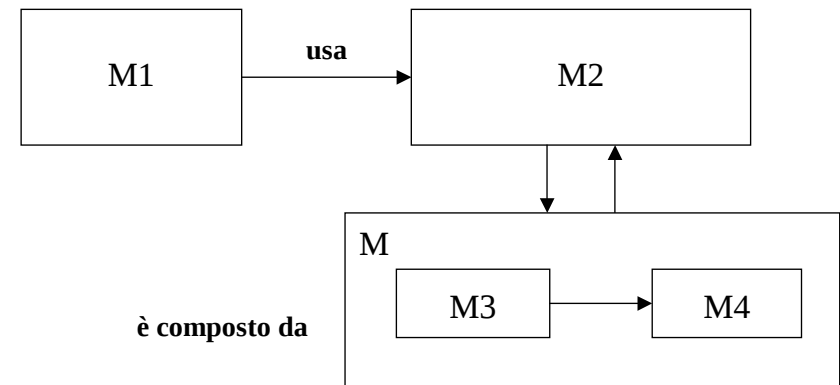
Architettura Modulare

È un sistema software costituito da un insieme di componenti (detti moduli) correlati tra di loro attraverso relazioni.

Relazioni

- “usa”
Un modulo M1 usa M2 se M1 utilizza meccanismi, tipi, o dati definiti all’interno di M2
- “è composto da”
È possibile definire moduli come aggregazioni di altri moduli componenti:
 - programmazione “top-down”
 - programmazione “bottom-up”

Esempio



Modulo

È il componente di base di una architettura modulare.

È costituito da due parti:

Interfaccia

È l'unica parte del modulo visibile e accessibile dall'esterno (cioè dagli altri moduli).

Può offrire all'esterno una vista astratta e parziale di ciò che il modulo effettivamente contiene.

È suddivisa in:

- EXPORT: entità (tipi, variabili, operazioni) **esportate**
- IMPORT: entità (tipi, variabili, operazioni) **importate**

Corpo

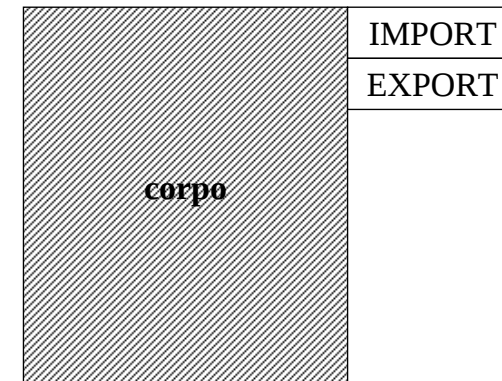
Contiene la realizzazione dei meccanismi che il modulo è in grado di eseguire.

Contiene la realizzazione di tipi e variabili locali al modulo.

Relazioni

- “usa”
Un modulo M1 usa M2 se M1 importa una (o più) entità esportate da M2.
- “è composto da”
È possibile definire moduli come aggregazioni di altri moduli componenti:
 - programmazione “top-down”
 - programmazione “bottom-up”

Moduli



Separazione tra corpo e interfaccia

information hiding

L'interfaccia (EXPORT) presenta una vista astratta di ciò che il modulo contiene: viene mostrato soltanto quello che è strettamente necessario ad altri moduli (cioè quello che viene importato da essi); i dettagli sono nascosti all'interno del corpo.

Information Hiding

Vantaggi

- La struttura interna di un modulo dipende in modo minimo dal “mondo esterno”
 - **riutilizzo**
 - possibilità di **sviluppo separato**
- L'architettura modulare non è influenzata dalla struttura interna del modulo
 - **modificabilità**
 - **estensibilità**
 - progetto separato dalla realizzazione ⇒ **rapida prototipazione**
- **Protezione**: è possibile impedire l'accesso a dati definiti in un modulo, nascondendoli all'interno del corpo
- **Astrazione** ⇒ leggibilità

Progettazione Modulare

Metodologie

Analizziamo i criteri da adottare nello sviluppo di applicazioni modulari:

- **information hiding**
Per ogni modulo bisogna valutare attentamente cosa mostrare e cosa nascondere.
È buona norma definire interfacce:
 - **minimali**
 - **stabili nel tempo**
- **low coupling – high cohesion**
Limitare al minimo le interazioni tra moduli.
⇒ riuso, modificabilità
Concentrare nello stesso modulo dati e funzioni più frequentemente usati in modo congiunto.
⇒ manutenibilità
- **progettazione per il cambiamento**
Strutturare il progetto in moduli che tengano conto delle eventuali future estensioni e modifiche
⇒ manutenibilità

Progettazione Modulare

Meccanismi

Esistono linguaggi di programmazione (linguaggi **modulari**) che offrono costrutti per la strutturazione dei programmi in moduli (ad esempio: ADA, Mesa, Modula2).

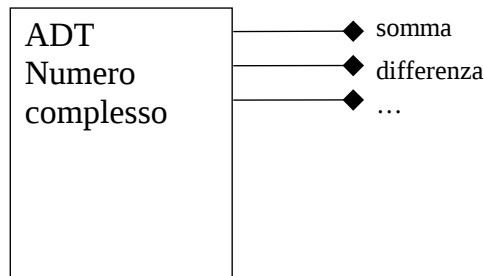
Caratteristiche di Linguaggi Modulari:

- definizione di moduli
- separazione tra interfaccia e corpo
- compilazione separata
- entità esportabili:
 - variabili
 - tipi
 - operazioni

Esempio:

Realizzazione del tipo di dato astratto (Abstract Data Type, o ADT) numero complesso.

Tipo di dato astratto: <S, Op, C>



Realizziamo l'ADT COMPLEX in uno pseudo-linguaggio modulare:

```
/*interfaccia*/

module interface COMPLEX
import  printf, scanf from stdio;
        sin, cos, sqrt, ... from math;
end import;
export type hidden complex; /*tipo "opaco" */
        typedef enum{ ReIm, ModArg; } rappr;
        complex crea(float A, float B, rappr R);
        complex somma(complex A, complex B);
        <altre operazioni...>
end export;
end interface

/*corpo */
module implementation COMPLEX
typedef struct { Float Re, Im} complex;

complex crea(float A, float B, rappr R) {
    complex C;
    if(R==ReIm) {
        C.Re=A;
        C.Im=B;
    }
    else{ <conversione>...}
    return C;
}

complex somma(complex A, complex B){
    complex C;
    C.Re=A.Re+B.Re;
    C.Im=A.Im+B.Im;
    return C;
}

<definizioni di altre funzioni>

end implementation
```

Osservazioni

- information hiding sui tipi: la rappresentazione concreta (dichiarazione) del tipo complex è nascosta $\Rightarrow$ astrazione
- information hiding sulle operazioni: la realizzazione (definizione) delle operazioni sul tipo (funzioni) è nascosta $\Rightarrow$ astrazione

I moduli che utilizzeranno COMPLEX:

- Non hanno la visibilità della dichiarazione del tipo complex (tipo *opaco*)
- Possono utilizzare il tipo complex per creare variabili
- Non hanno bisogno di conoscere come è realizzato il tipo complex
- Dispongono di alcune operazioni (quelle esportate dal modulo COMPLEX) che consentono di manipolare in modo astratto dati del tipo complex

Modularità in C

☞ Il linguaggio C **non** è un linguaggio modulare.

Però:

Esiste la possibilità di strutturare un programma su più file:

- file sorgenti compilabili separatamente
- file header

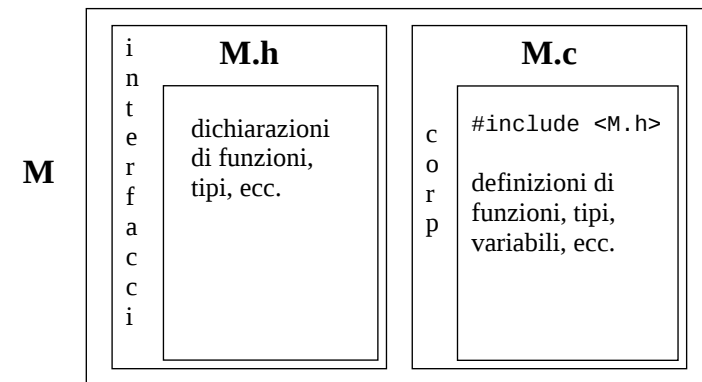
Idea di base:

“Mappare” ogni modulo M su una coppia di file:

- un file **header** (M.h) che rappresenta *l'interfaccia* del modulo
- un file **sorgente** (M.c) che rappresenta *il corpo* del modulo

Interfaccia

- **export:** tutte le dichiarazioni e le definizioni in M.h.
- **import:** direttive `#include <...>` in M.c.



Esempio: ADT numero complesso

Mappiamo il modulo sui due file:

- `complex.h` (interfaccia) e
- `complex.c` (corpo)

```
/*file complex.h - interfaccia del modulo */
#include <stdio.h>
#include <math.h>

typedef enum{ ReIm, ModArg; } rappr;
typedef struct { float Re, Im; } complex;
complex somma(complex A, complex B);
complex crea(float A, float B, rappr R);
...
```

```
/*file complex.c - corpo del modulo */
#include "complex.h"

complex crea(float A, float B, rappr R) {
    complex C;
    if(R==ReIm) {
        C.Re=A;
        C.Im=B;
    }
    else{ <conversione...> }
    return C;
}

complex somma(complex A, complex B) {
    complex C;
    C.Re=A.Re+B.Re;
    C.Im=A.Im+B.Im;
    return C;
}
<definizioni di altre funzioni>
```

Osservazioni

- Assenza di tipi opachi $\Rightarrow$ non si può realizzare un ADT in senso stretto (non c'è protezione)

Ad esempio:

```
/* file main.c */
#include "complex.h"

main() {
    complex A, B, C;
    ...
    C=somma(A,B); /* accesso in modo astratto*/
    ...
    A.Re=10.5; /* accesso diretto alla
rappresentazione */
}
```

- Import non selettivo

☞ Il C fornisce un debole supporto alla modularità: per l'organizzazione modulare di un programma è necessario un uso auto-disciplinato degli strumenti del linguaggio.

Nei linguaggi modulari l'uso dei moduli è un *obbligo*!

Programmi Modulari in C

Il collegamento tra moduli viene specificato attraverso un **file progetto** (estensione **.prj** in turboC o Djgpp) o Makefile (in UNIX).

File progetto

Per compilare un programma suddiviso in più file, viene creato il file progetto (estensione **.prj**) in cui si specificano quali file sorgenti (i corpi dei moduli) costituiscono l'applicazione.

Esempio: ADT complex

Contenuto di *file.prj*:

complex.c main.c

Ciascun file .c che compare nel file progetto aperto viene compilato separatamente (producendo un file oggetto, con estensione **.o**) e successivamente il linker collega i file oggetto in un unico file eseguibile.

Esempio (linguaggio C)

Gli identificatori esportati sono definiti o dichiarati in un file separato (file *header*, con estensione .h)

```
/* ELEMENT TYPE - file el.h */
typedef int el_type;
typedef enum {false, true; } boolean;
boolean isequal(el_type, el_type);
boolean isless(el_type, el_type);
void showel(el_type e);
```

Altri identificatori e la definizione delle funzioni esportate e altre funzioni sono riportate in un secondo file (file di codice con estensione .c)

```
/* ELEMENT TYPE - file el.c*/
#include "el.h"
#include <stdio.h>

boolean isequal(el_type e1, el_type e2) {
    return (e1==e2);
}

boolean isless(el_type e1, el_type e2) {
    return (e1<e2);
}

void showel(el_type e) {
    printf("%d", e);
}
```

Il collegamento tra moduli viene specificato anche in questo caso attraverso un file progetto (estensione **.prj**)

Un Esempio Completo in C

```
/* ELEMENT TYPE - file el.h */
typedef int el_type;
typedef enum {false, true; } boolean;
boolean isequal(el_type, el_type);
boolean isless(el_type, el_type);
void showel(el_type e);
```

```
/* ELEMENT TYPE - file el.c */
#include "el.h"
#include <stdio.h>

boolean isequal(el_type e1, el_type e2) {
    return (e1==e2);
}
boolean isless(el_type e1, el_type e2) {
    return (e1<e2);
}
eltype leggi() {
    eltype el;
    scanf("%d", &el);
    return el;
}
void showel(el_type e) { printf("%d",e); }
```

```
/* main file - inizio.c */
#include "el.h" /* utilizzo */

void main(void) {
    el_type Dato;
    Dato=leggi();
    showel(Dato);
}
```

Nel file **.prj**:

el.c inizio.c

Componenti software

Esistono diversi livelli di utilizzo di moduli per realizzare:

Librerie

Il modulo rende visibili procedure e funzioni che non fanno uso di variabili non locali (prive, cioè, di effetti collaterali). Il modulo offre una collezione di operazioni (ad esempio, funzioni matematiche).

Astrazioni di dato

Il modulo ha dati locali (nascosti) e rende visibili all'esterno i prototipi delle operazioni invocabili (procedure e funzioni) su questi dati locali, ma non gli identificatori dei dati. Attraverso una di queste operazioni si può assegnare un valore iniziale ai dati locali nascosti.

Tipo di dato astratto

Il modulo esporta un identificatore di tipo T ed i prototipi delle operazioni eseguibili su dati dichiarati di questo tipo. I moduli "clienti" dichiarano e controllano quindi il tempo di vita delle variabili di tipo T.