Progettazione – parte II

Leggere sez. 4.2.1 - 4.2.4

Ghezzi et al

Modularizzazione

- Modulo: componente ben definito di un sistema software
- Fornitore di risorse computazionali o di servizi
  - Dati
  - Funzioni
  - Oggetti

Sommario

- Come strutturare un sistema software in moduli
- Notazione per la descrizione della struttura modulare di un sistema
- Proprietà desiderabili per la struttura
- Interfaccia e implementazione
- Tipi di moduli

Insiemi e relazioni

- S insieme di moduli

$$S = \{M_1, M_2, \ldots, M_n\}$$

- Relazione binaria r su S

$$r \subseteq S \times S$$

- Notazione

$$M_j \; r \; M_k \leftrightarrow (M_j, M_k) \in r$$

Relazioni fra moduli

- Le ipotizziamo *anti-riflessive*

$$\forall i \in 1..n \; (M_i, M_i) \notin r$$

- *Chiusura transitiva* r^+ di r:

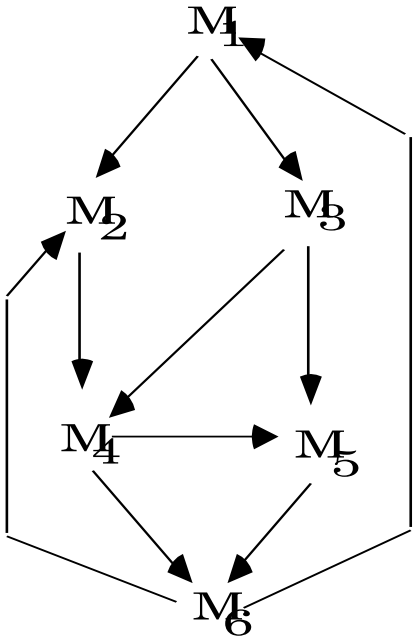
$$M_i \; r^+ \; M_j \text{ se e solo se}$$

  - $M_i \; r \; M_j$ oppure
  - $\exists \; M_k \in S$ t.c. $M_i \; r \; M_k$ e $M_k \; r^+ \; M_j$

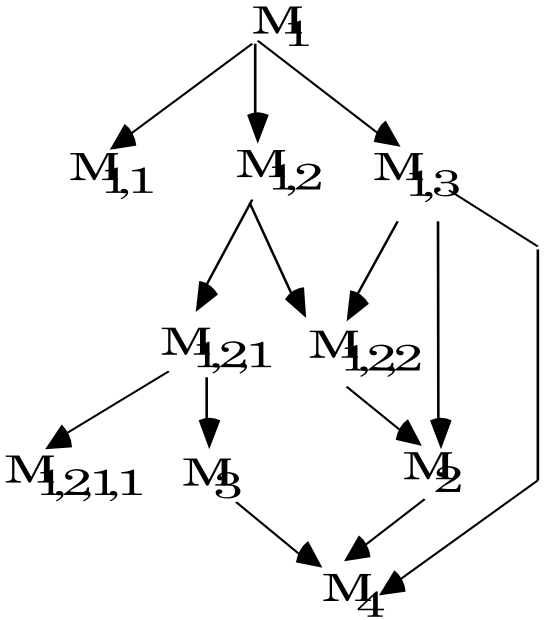
- *Gerarchia*: relazione r tale che per nessuna coppia (M_i, M_j) vale $M_i \; r^+ \; M_j$ e $M_j \; r^+ \; M_i$

Relazioni: rappresentazione come grafi

- Freccia da M_j a M_k significa M_j r M_k
- Una gerarchia è rappresentata da un grafo diretto aciclico (DAG)



a)



b)

Tipi di relazioni fra moduli

- Dipendenza funzionale (USES)
  - evidenzia la dipendenza di un modulo dalle funzionalità esportate da un altro
- Scomposizione (IS_COMPONENT_OF e derivate):
  - indicano come un modulo si suddivide in altri di dimensioni minori

La relazione USES

- Def: M_j USES M_k se e solo se M_j richiede M_k per il suo funzionamento
- M_j utilizza una risorsa (funzione di calcolo, struttura dati, etc.) resa disponibile da M_k.
- M_j è detto *client*, M_kè detto *server*.
- Relazione “statica”: sussiste anche se a tempo di esecuzione M_j non usa M_k.

Proprietà desiderabili di USES

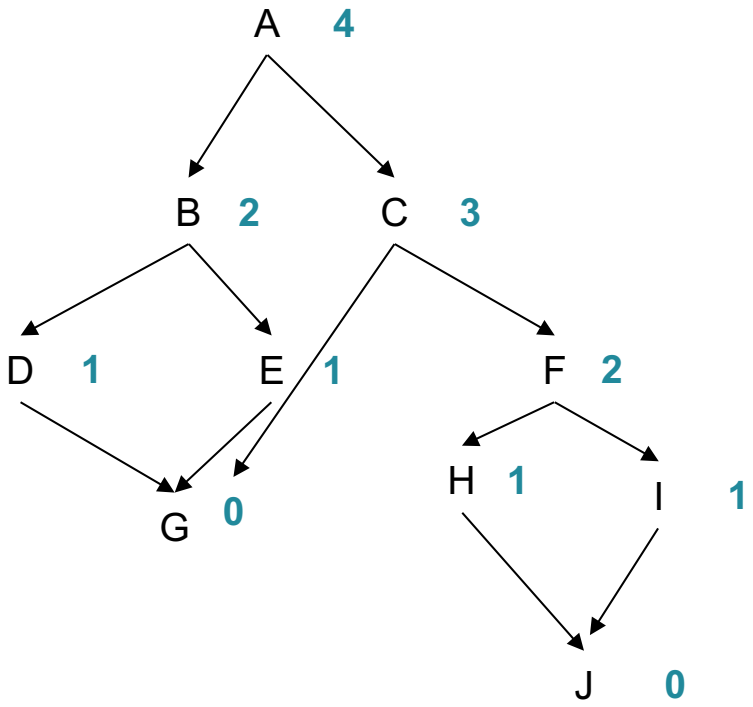
- E’ utile che USES sia una gerarchia
- Un sistema strutturato gerarchicamente è più facile da
  - capire
  - implementare
  - testare

  partendo dalle foglie e andando verso l’alto.

Livelli di astrazione

- Una gerarchia può essere interpretata a livelli di astrazione
- Definizione ricorsiva:
  - M_i ha livello 0 se per nessun M_j si ha M_i r M_j
  - Sia k il massimo livello di tutti gli M_j tali che M_i r M_j. Allora M_i ha livello k+1

Livelli di astrazione



Macchina astratta

- I moduli di livello i forniscono una *macchina astratta* ai moduli di livello maggiore di i.
- Macchina astratta: insieme di servizi a cui si accede astraendo dalla loro implementazione

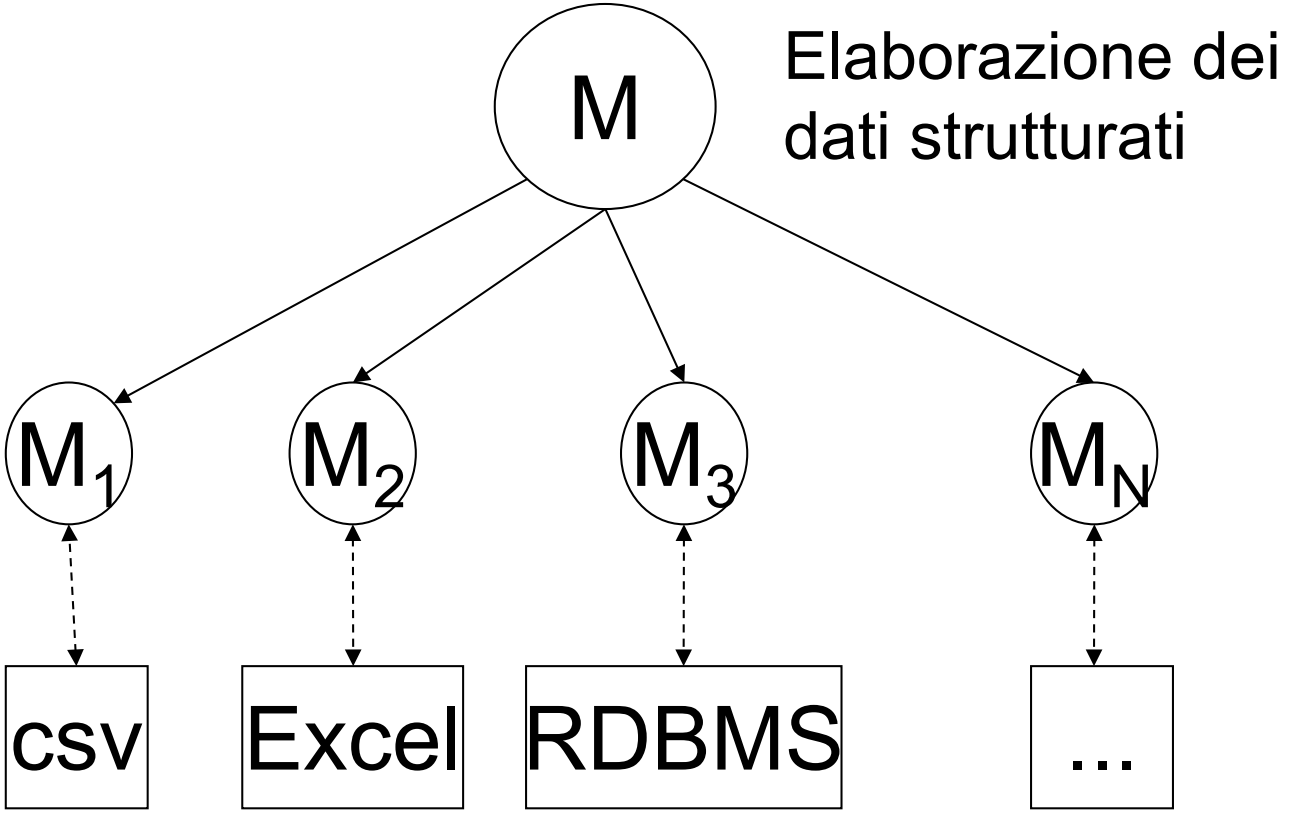
Proprietà desiderabili di USES

- Archi uscenti: Fan-Out (meglio basso)
- Archi entranti: Fan-In (meglio alto)
- Che tipi di USES?
  - Modifica dei dati pubblici di un modulo (*da evitare!)*
  - Scambio di flag (poco leggibile)
  - Chiamate di funzioni, passaggio di parametri (OK)
  - Chiamata di funzione remota (Java RMI)

Esempio

- Sistema che elabora dati strutturati (record)
- Dati immagazzinati in diverse modalità:
  - csv, Excel, RDBMS, ...
- Moduli per fornire macchina astratta con funzionalità di apertura, lettura, scrittura, chiusura a quelli di livello superiore.
- Famiglia di prodotti

Esempio


M
Elaborazione dei dati strutturati
M_1
M_2
M_3
M_N
csv
Excel
RDBMS
...

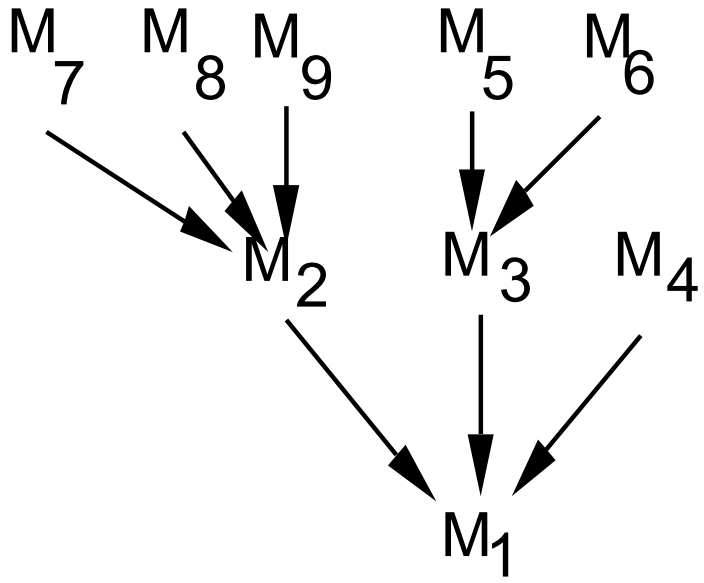
IS-COMPONENT-OF, COMPRISES

- Definisce come un modulo è composto da altri
- M_j IS-COMPONENT-OF M_k se e solo se M_k è ottenuto aggregando vari moduli, uno dei quali è M_j.
- COMPRISES: relazione inversa
- M_k COMPRISES M_j se e solo se M_j IS-COMPONENT-OF M_k

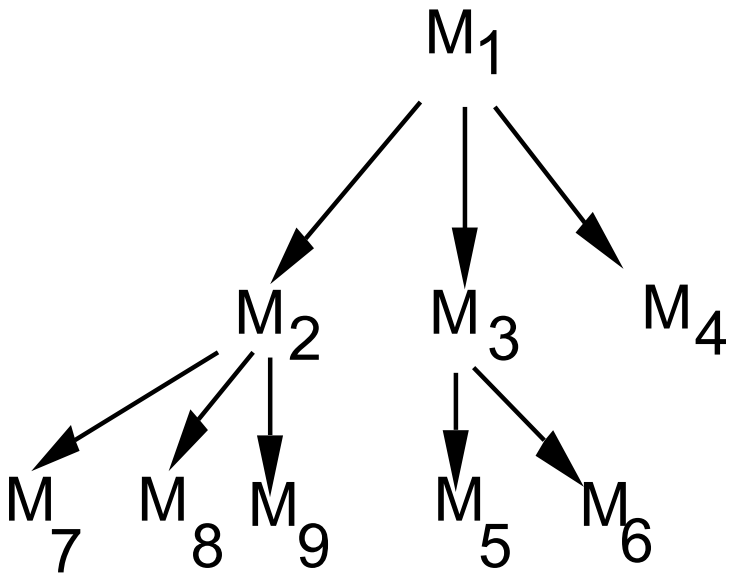
IS-COMPOSED-OF, IMPLEMENTS

- Dato un insieme S = {M_1, ..., M_n} di moduli, e definito
- $M_{S,j}$={M_k|$M_k \in$S e M_k IS-COMPONENT-OF M_j}
- Si dice che M_j IS-COMPOSED-OF $M_{S,j}$ e che $M_{S,j}$ IMPLEMENTS M_j

Rappresentazione grafica



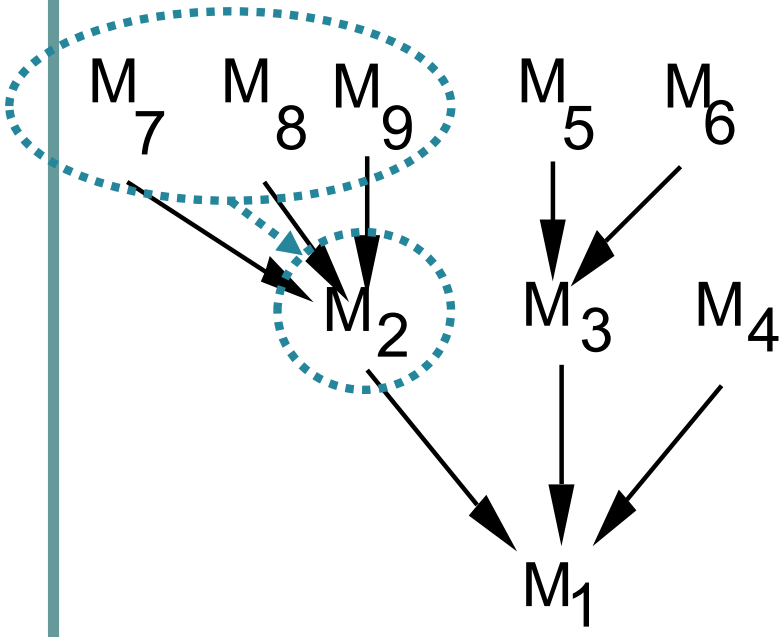
(IS_COMPONENT_OF)



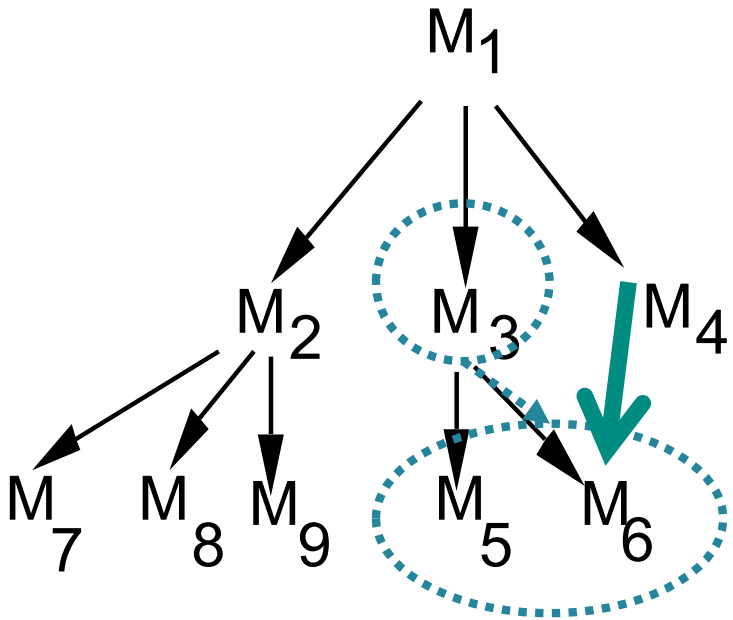
(COMPRISES)

- M1 composto da M2, M3 E M4
- Entrambe sono gerarchie

Rappresentazione grafica



(IMPLEMENTS)



(IS-COMPOSED-OF)

- Se M6 è anche componente di M4

Interfaccia e implementazione

Interfaccia e implementazione

- Un modulo *esporta* i servizi che rende disponibili ai suoi client
- L'insieme dei servizi esportati da un modulo è la sua *interfaccia*
- Il modo in cui l'interfaccia è realizzata (*implementazione*) è un "segreto" del modulo, cioè è nascosto ai client (o è *incapsulato* nel modulo)

Interfaccia e implementazione

- La distinzione fra interfaccia e implementazione è un principio fondamentale della progettazione software.
- Permette a chi implementa il modulo di concentrarsi solo su esso e di astrarre dall'implementazione dei server
- L'interfaccia è un contratto fra l'implementatore e l'utente del server.
- L'implementazione può cambiare senza conseguenze per i client.

Come progettare l'interfaccia di un modulo?

- Compromesso fra
  - necessità di esportare più servizi per incrementare la funzionalità
  - necessità di esportare meno servizi per mantenere semplice l'interfaccia
- E' meglio non esportare qualcosa che può cambiare.

Esempio 1

- Interprete di un linguaggio di programmazione
- Modulo SYMBOL_TABLE per la gestione delle variabili
- Fornisce operazioni per:
  - creare una nuova variabile (CREATE)
  - assegnare un valore a una variabile (PUT)
  - leggere il valore di una variabile (GET)

Esempio 1

- Come valore di controllo, GET e PUT restituiscono anche un puntatore alla variabile che contiene:
  - l'indirizzo della variabile se l'operazione è andata a buon fine
  - NULL se la variabile non esiste

Esempio 1

- Ora il client ha accesso all'indirizzo fisico della variabile.
- Può accedere direttamente all'area di memoria. Questo è un problema se
  - l'indirizzo della variabile cambia a tempo di esecuzione (accesso ripetuto)
  - cambia l'implementazione della struttura dati che rappresenta il valore

Esempio 2

- Sistema di controllo di un impianto
- Raccoglie dati fisici e
  - li archivia in uno storico
  - a seconda dei valori restituisce segnali di controllo per l'impianto
- Si prevede di voler integrare, in futuro, funzionalità di interrogazione e analisi statistica sui dati

Esempio 2

- Modulo per l'acquisizione dell'input che nasconde il modo in cui i dati vengono raccolti e restituisce i valori
- Modulo per l'accesso allo storico che nasconde la rappresentazione dei dati
- Modulo per l'interpretazione delle interrogazioni che ne restituisce una versione astratta

Progetto per sviluppo successivo

- Sviluppo incrementale:
  - Identificare subito un nucleo di sottoinsiemi utili (e urgenti) dell'applicazione
- Definire con precisione le interfacce delle parti che vengono rimandate!
- Possibilità di sviluppare versioni semplificate di alcune parti

Prototipi

- Una struttura modulare ben progettata facilita la creazione di prototipi
- Inizialmente, si possono ottimizzare i moduli che si vuole mostrare ai clienti e lasciare gli altri in versione prototipale.
- Successivamente, si possono migliorare gli altri moduli senza toccare quelli già ottimizzati.

Esempio

- DBMS con funzionalità di interrogazione innovative
- Prima sviluppare a fondo l'interfaccia utente, per mostrarla ai clienti e ottenere feedback
- Poi ottimizzare la rappresentazione e l'accesso ai dati.

Notazioni per la progettazione

Notazioni per la progettazione

- Necessità di notazioni non ambigue
- Non è ancora emersa una notazione standard
- UML, per Object-Oriented Design
- Due notazioni:
  - TDN, testuale
  - GDN, grafica

TDN

- Ispirata a linguaggi di programmazione modulari come Ada o Modula-2
- Parzialmente non specificata
- Un modulo può esportare qualsiasi risorsa:
  - variabile
  - tipo
  - procedura
  - funzione...

TDN

- I commenti forniscono
  - informazioni di natura semantica sui servizi esportati
  - la definizione del *protocollo*, ossia delle regole che il modulo client deve seguire per utilizzare i servizi
    - inizializzazione
    - propedeuticità
  - informazioni di natura non semantica

TDN

- **uses** indica i moduli server
- **exports** indica le risorse esportate (interfaccia)
- **implementation** definisce ad alto livello l'implementazione: utile per
  - comprendere il funzionamento
  - guidare l'implementazione effettiva
- commenti informali sulle parti incapsulate e sulla motivazione

Esempio

module X
uses Y, Z
exports var A : integer;
type B : **array** (1. .10) **of** real;
procedure C (D: **in out** B; E: **in** integer; F: **in** real);
Here is an optional natural-language description of what A, B, and C actually are, along with possible constraints or properties that clients need to know; for example, we might specify that objects of type B sent to procedure C should be initialized by the client and should never contain all zeroes.
implementation
If needed, here are general comments about the rationale of the modularization, hints on the implementation, etc.
is composed of R, T
end X

Esempio

```
module R
uses Y
exports var K : record . . . end;
        type B : array (1..10) of real;
        procedure C (D: in out B; E: in integer; F: in real);
implementation
        .
        .
        .
end R

module T
uses Y, Z, R
exports var A : integer;
implementation
        .
        .
        .
end T
```

Altre caratteristiche

- Import selettivo

  uses M **imports** (E_1,E_2)

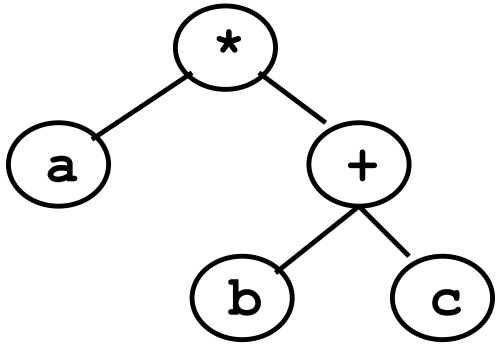
- Riferimento a entità esportate. Se E è esportata da M, il client può riferisi ad essa:
  - M.E (dot notation)
  - Semplicemente E se non c'è ambiguità

Esempio: compilatore MINI

module COMPILER
exports procedure MINI (PROG: in file of char;
CODE: out file of char);
MINI is called to compile the program stored in PROG and produce the object code in file CODE
implementation
A conventional compiler implementation. ANALYZER performs both lexical and syntactic analysis and produces an abstract tree, as well as entries in the symbol table; CODE_GENERATOR generates code starting from the abstract tree and information stored in the symbol table. MAIN acts as a job coordinator.
is composed of ANALYZER, SYMBOL_TABLE, ABSTRACT_TREE_HANDLER, CODE_GENERATOR, MAIN
end COMPILER

Albero sintattico astratto

a*(b+c)



Notazione polacca posfissa: abc+*

Altri moduli

```
module MAIN
uses ANALYZER, CODE_GENERATOR
exports procedure MINI (PROG: in file of char;
                        CODE: out file of char);
...
end MAIN

module ANALYZER
uses SYMBOL_TABLE, ABSTRACT_TREE_HANDLER
exports procedure ANALYZE (SOURCE: in file of char);
SOURCE is analyzed; an abstract tree is produced
by using the services provided by the tree handler,
and recognized entities, with their attributes, are
stored in the symbol table.
...
end ANALYZER
```

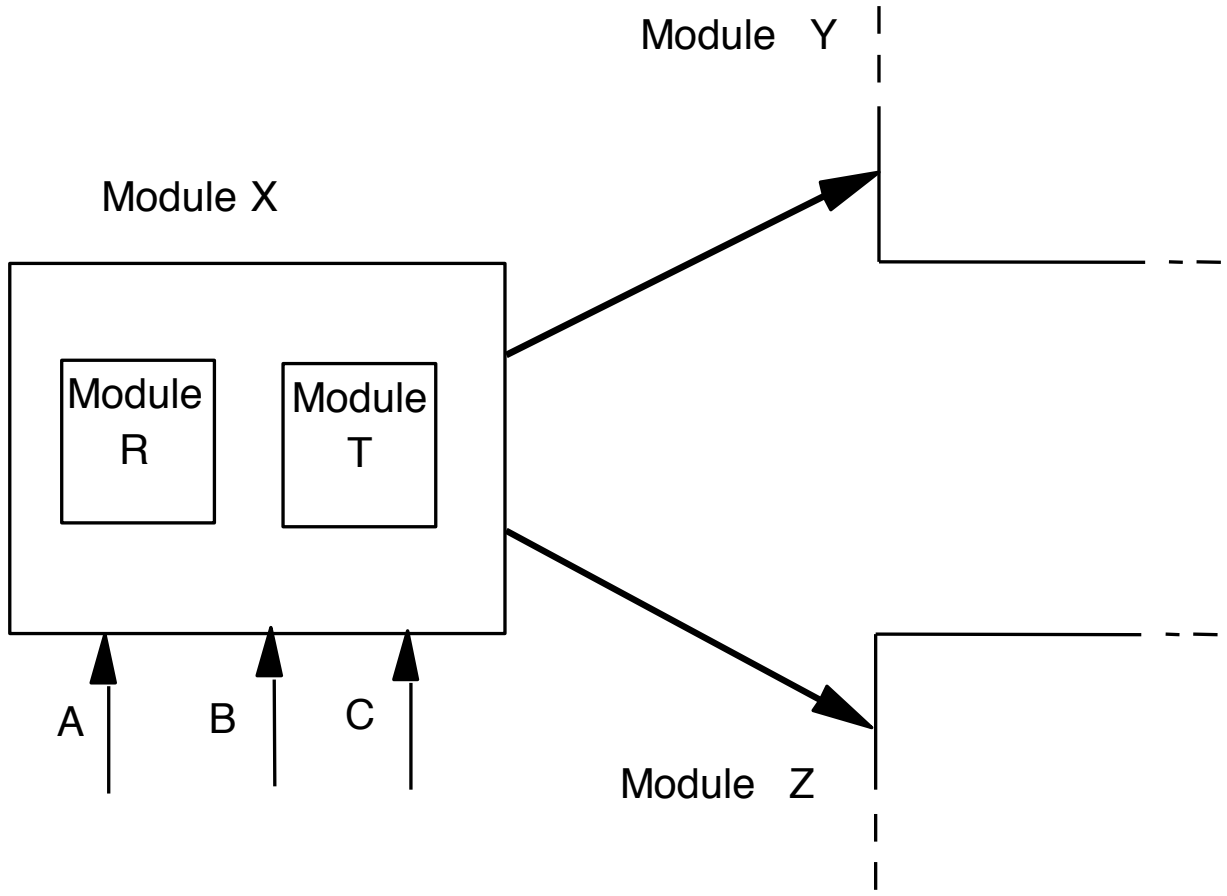
Altri moduli

module CODE_GENERATOR
uses SYMBOL_TABLE, ABSTRACT_TREE_HANDLER
exports procedure CODE (OBJECT: out file of char);
The abstract tree is traversed by using the operations exported by the ABSTRACT_TREE_HANDLER and accessing the information stored in the symbol table in order to generate code in the output file.
...
end CODE_GENERATOR

GDN

- Notazione grafica corrispondente a TDN
- Modulo indicato da un rettangolo
- Le risorse esportate sono indicate su frecce entranti
- Il rettangolo corrispondente a un modulo contiene i suoi componenti

Esempio



Esempio

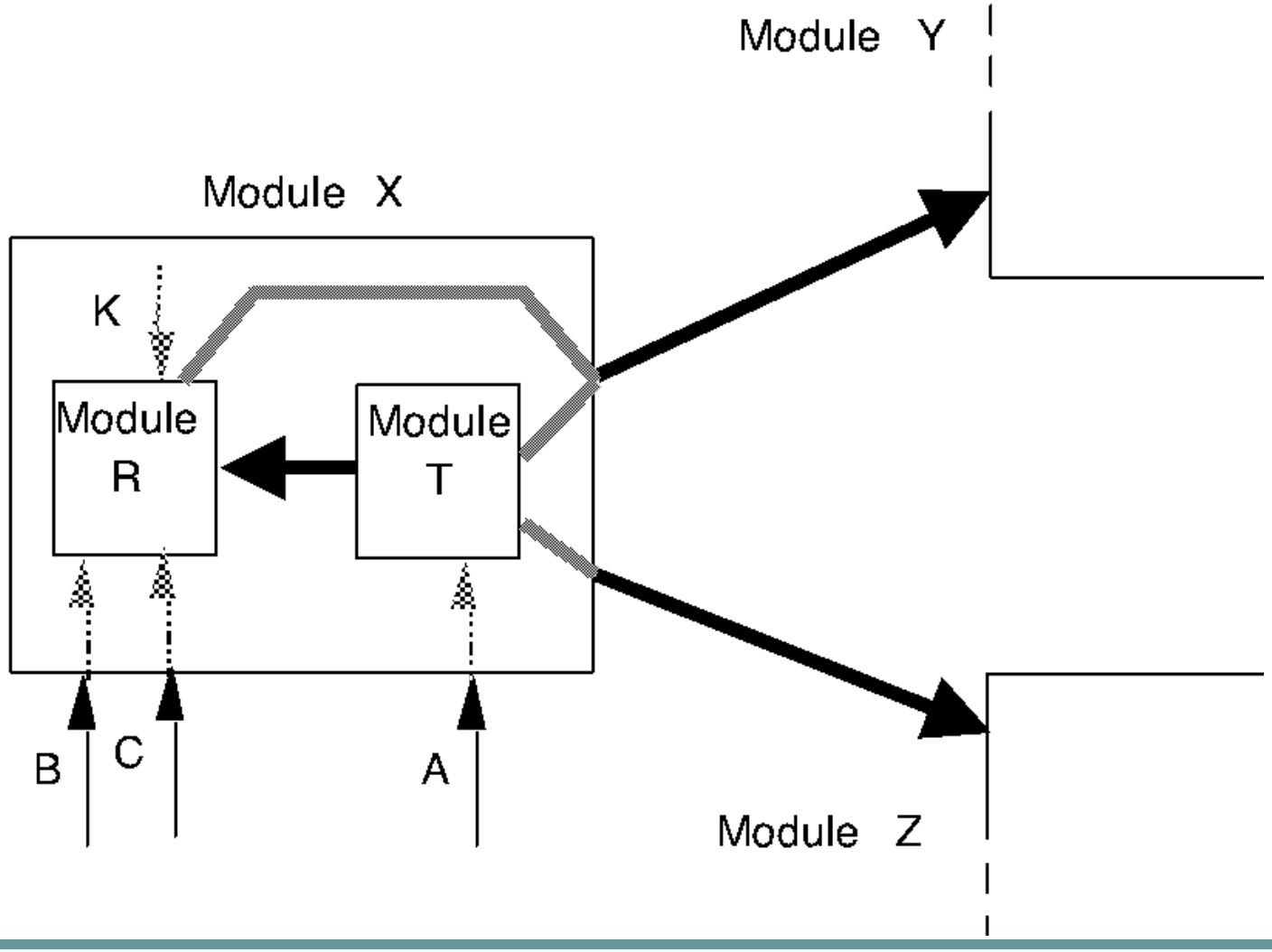
module X
uses Y, Z
exports var A : integer;
type B : **array** (1. .10) **of** real;
procedure C (D: **in out** B; E: **in** integer; F: **in** real);
Here is an optional natural-language description of what A, B, and C actually are, along with possible constraints or properties that clients need to know; for example, we might specify that objects of type B sent to procedure C should be initialized by the client and should never contain all zeroes.
implementation
If needed, here are general comments about the rationale of the modularization, hints on the implementation, etc.
is composed of R, T
end X

Esempio

```
module R
uses Y
exports var K : record . . . end;
        type B : array (1..10) of real;
        procedure C (D: in out B; E: in integer; F: in real);
implementation
        .
        .
        .
end R

module T
uses Y, Z, R
exports var A : integer;
implementation
        .
        .
        .
end T
```

Esempio



Categorie di moduli

- Astrazioni procedurali: funzionalità di calcolo
  - funzioni
  - procedure

  rese disponibili ad altri moduli.
- Incapsulano algoritmi
- Esempi: ordinamento, trasformata di Fourier, traduzione...

Categorie di moduli

- Librerie
  - Insiemi di astrazioni procedurali correlate (es. libreria di operazioni su matrici)
  - (solitamente) stateless
- Pool comuni di dati
  - Dati necessari ad altri moduli (es. costanti di configurazione)
  - Basso livello

Categorie di moduli

- Oggetti astratti (astrazioni di dato)
  - Insieme di funzioni, come per le librerie, ma
  - stato interno accessibile ai client solo attraverso funzioni di accesso
- Tipi di dati astratti (ADT)
  - Modelli per la creazione di istanze di dati astratti

Esempio –stack

- Valutatore di espressioni in RPN (notazione polacca inversa, o postfissa)

  a*(b+c) → abc+*

- Si può implementare per mezzo di una pila. Se le operazioni sono binarie: per input
  - operando → push
  - operatore →
    1. pop di due operandi (i successivi nello stack)
    2. push del risultato

Esempio

- Oggetto astratto STACK

  exports

  procedure PUSH (VAL: in integer);

  procedure POP_2 (VAL1, VAL2:
  out integer);

- La struttura dati (stato) è nascosta e può cambiare senza conseguenze per il client

Esempio

- Difetti del progetto
  - suppone che l'espressione da valutare sia corretta: l'espressione "1+" causerebbe un errore a runtime
    - correzione: funzione EMPTY
  - non supporta le operazioni non binarie
    - correzione: esportare una funzione POP che preleva un solo valore dallo stack

Tipi di dati astratti

- L'applicazione può richiedere più esemplari di un determinato oggetto astratto *con stati indipendenti*.
- Il tipo di dato astratto permette al client di creare un numero arbitrario di istanze, ognuna delle quali è un oggetto astratto
- Corrispondono alle *classi* in C++, Java, .NET.

Esempio

- Simbolo “?”: i dettagli del tipo esportato sono nascosti nell’implementazione

```
module  STACK_HANDLER
exports
        type  STACK = ?;
        This is an abstract data-type module; the data structure
        is a secret hidden in the implementation part.
        procedure   PUSH (S: in out  STACK ; VAL: in  integer);
        procedure   POP (S: in out  STACK ; VAL: out  integer);
        function  EMPTY (S: in  STACK) : BOOLEAN;
        .
        .
        .
end  STACK_HANDLER
```

Assegnamento e uguaglianza

- Operatori usati spesso
- Anzichè annotare l'esportazione come per le altre funzioni:

type A_TYPE: ? (:=, =);

Il client può applicare assegnamento o uguaglianza a variabili del tipo A_TYPE

Esempio

- Area di servizio
- Vari servizi disponibili
  - Benzina/diesel
  - Metano
  - Lavaggio
- A ogni servizio è associata una coda (FIFO)

Esempio

```
module FIFO_CARS
uses CARS
exports
type QUEUE : ?;
procedure ENQUEUE (Q: in out QUEUE ; C: in CARS);
procedure DEQUEUE (Q: in out QUEUE ; C: out CARS);
function IS_EMPTY (Q: in QUEUE) : BOOLEAN;
function LENGTH (Q: in QUEUE) : NATURAL;
procedure MERGE (Q1, Q2 : in QUEUE ; Q : out QUEUE);
This is an abstract data-type module representing
queues of cars, handled in a strict FIFO way;
queues are not assignable or checkable for equality,
since ":=" and "=" are not exported.
…
end FIFO_CARS
```

Una breve sintesi …

Oggetti astratti

- Astrazioni di dato
- Classi con parte statica
- Esemplari di un tipo di dato astratto (oggetti, istanze)

Tipi di dato astratto

- Classi con parte non statica

Moduli generici

- Molte strutture dati prevedono
  - un tipo “componente”
  - operazioni parametriche rispetto al componente
- Esempio: pila
  - Push, pop sono disponibili indipendentemente dal tipo del singolo dato
  - Definire un tipo pila per ogni tipo del singolo dato è scomodo e porta a errori

Moduli generici

- Definiti parametricamente rispetto a tipi.
- Es. pila di <T>

```
generic module GENERIC_STACK_2 (T)
      . . .
 exports
    procedure PUSH (VAL : in T);
    procedure POP_2 (VAL1, VAL2 : out T);
 ...
end GENERIC_STACK_2
```

Istanziazione

- Il client non può utilizzare direttamente il tipo generico: deve specificare T

  module INTEGER_STACK_2 **is**
  GENERIC_STACK_2 (INTEGER)

- *template* in C++
- *generic* in Java, .NET

Genericità ristretta

- Se è necessario che il tipo parametro T supporti un'operazione OP:

generic module M (T) **with** OP(T)
  uses …
    …
  end M

- Esempio: l'ordinamento richiede ≤
- Istanziazione

  module M_A_TYPE **is** M(A_TYPE) PROC(M_A_TYPE)

Java Collection Framework

Java Collections Framework (JCF) fornisce il supporto a qualunque tipo di *struttura dati*

- Interfacce (insieme, lista, mappa associativa …)
- una classe Collections che definisce algoritmi polimorfi (funzioni statiche)
- classi che forniscono implementazioni dei vari tipi di strutture dati specificati dalle interfacce

Obiettivo: strutture dati per "elementi generici"