- ***Lezione 32***

In fase di progettazione dettagliata il flow chart è rappresentato da un grafo che stavolta fa riferimento a un singolo modulo.

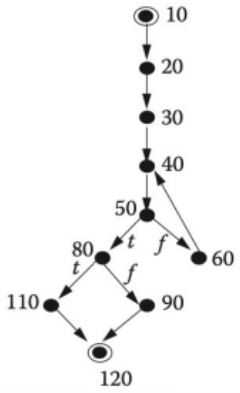
Tipicamente questi grafi di flusso hanno una struttura costituita da strutture di controllo di base e avanzate, noi ci focalizzeremo in questa fase di introduzione di metriche sulle strutture di controllo di base (sequenza, selezione, iterazione).

Un flow chart è quindi un grafo diretto dove ogni nodo corrisponde a una istruzione.
Esistono diversi tipi di nodi:

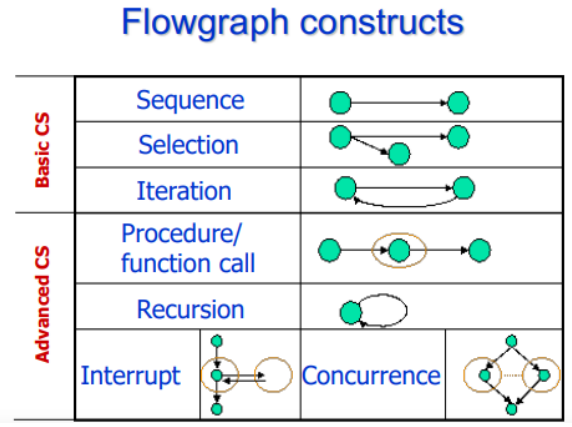
- Nodo Procedurale: nodo con un solo arco uscente (significa che quando il nodo termina la propria esecuzione esiste uno e un solo nodo successivo)
- Nodo Predicato: nodi con numero di archi uscenti != 1 (es. istruzioni decisionali, terminata l'esecuzione del nodo si "sceglie" uno e un solo nodo dei tanti possibili, non è concorrente è decisionale)
- Nodo Start: nodi con numero di archi entranti = 0
- Nodo End: nodi che ha archi in ingressso ma non in uscita

Mentre il nodo rappresenta i vari statement, gli archi rappresentano il vero e proprio flusso di controllo (ciò che avviene nell'eseguire le istruzioni).

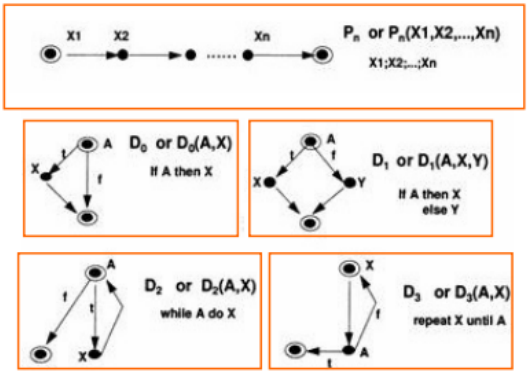
```
10  INPUT P
20  Div = 2
30  Lim = INT(SQR(P))
40  Flag = P/Div - INT(P/Div)
50  IF Flag = 0 OR Div = Lim THEN 80
60  Div = Div + 1
70  GO TO 40
80  IF Flag <>0 OR P>4 THEN 110
90  PRINT Div; "Smallest factor of"; P; "."
100 GO TO 120
110 PRINT P; " is prime"
120 END
```



10, 20, 30, 40 statement procedurali, 10 nodo start, 50 e 80 predicato etc…
Come detto, se il codice è ben strutturato, il corrispondente grafo di flusso esibisce dei costrutti di base (sequenza, iterazione, selezione).
Per quanto riguarda le strutture avanzate che non vedremo l'invocazione a funzione rappresenta il passaggio da un nodo a un nuovo sottografo che poi viene eseguito a parte, la ricorsione una chiamata del modulo a se stesso, l'interrupt interruzione e concorrenza rappresentata da pallini gialli



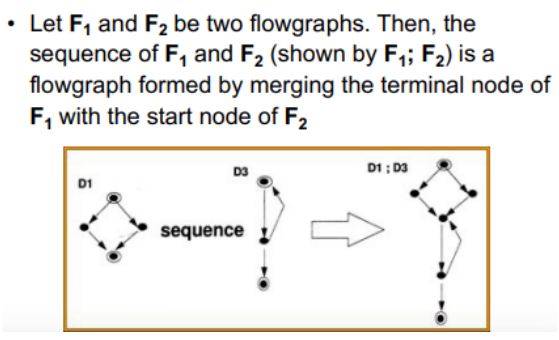
Limitandoci alle strutture base, è possibile individuare dei grafi di flusso comuni a tutti i programmi strutturati.



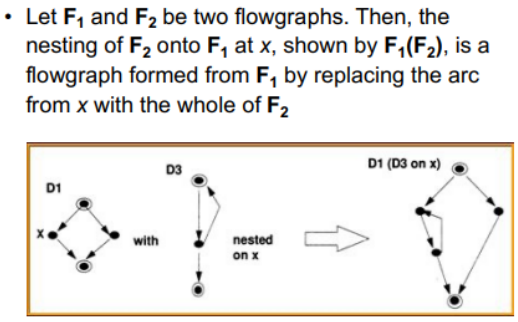
Pn come sequenza di n statement procedurali, D_0 o $D_0(A, X)$ l'if (dove A condizione e X codice da eseguire se la condizione è vera), poi D_1 o $D_1(A, X, Y)$ if then else, while do come D_2 e repeat until D_3. La differenza tra while do e repeat until è che con il repeat until l'istruzione che si effettua se la condizione A è vera (X) si esegue anche alla prima botta senza verificare subito la condizione.
Un grafo di flusso ben strutturato a livello di sottofase di progettazione dettagliata è costituito solo ed esclusivamente da questi sottografi di base, opportunamente combinati.

Due possibili operazioni per combinarle:

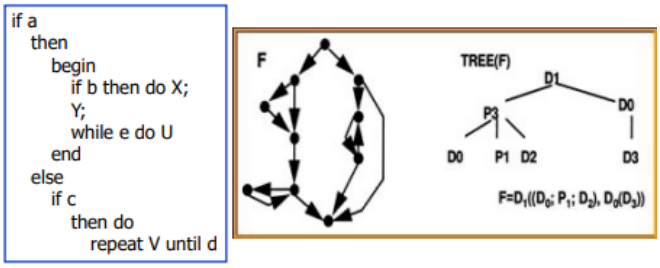
- *Sequenziamento:* dati due flow graph F_1 e F_2. Mettiamo i due flow graph in sequenza rendendo il nodo finale di F_1 il nodo iniziale di F_2, indichiamo il tutto con il simbolo; $(F_1; F_2)$.



- *Nesting:* in questo caso, dati sempre F_1 e F_2 flowgraphs, il nesting di F_2 in F_1 rispetto a X rappresenta il fatto che l'intero "codice" descritto da F_2 sostituisce la condizione X di F_1 → si sostituisce l'arco uscente da X con F_2. $F_1(F_2)$

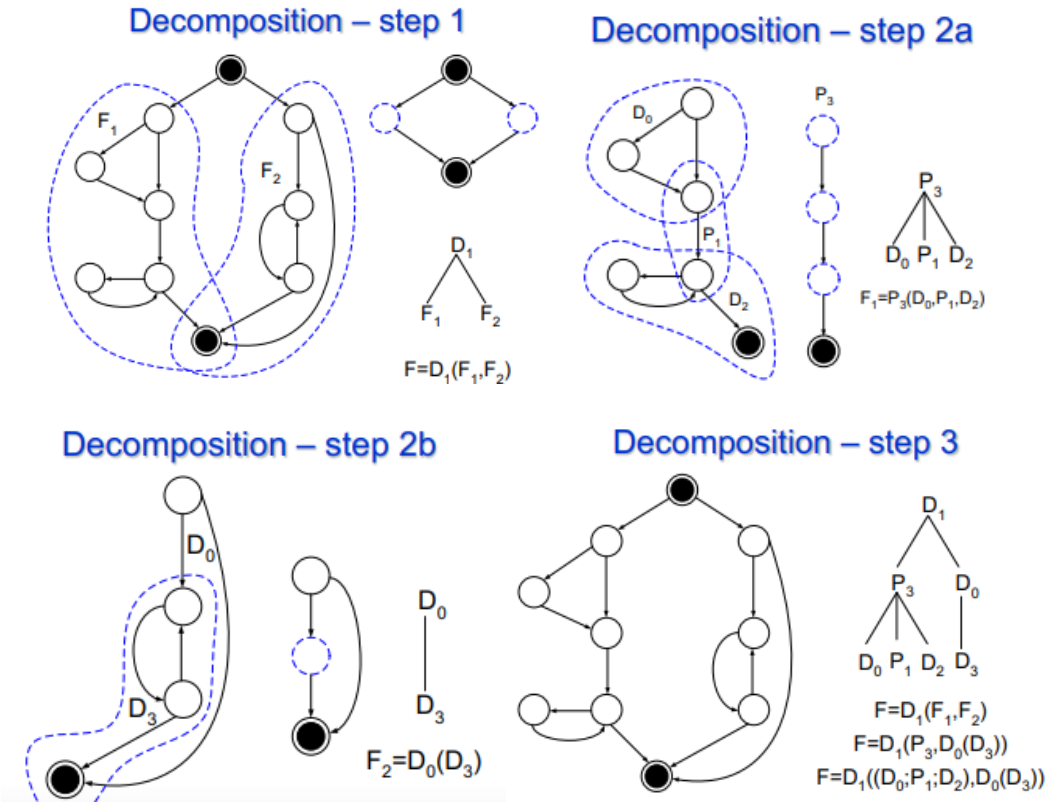


In generale, un Prime Flowgraph rappresenta un flowgraph che non può essere ulteriormente decomposto secondo il sequencing e il nesting.
Quindi P_1 (non P_n!), D_0, D_1, D_2 , D_3 sono tutti Prime Flowgraphs e sono dette D-structures.
Ciò che vogliamo fare a questo punto è verificare quanto è D-strutturato il grafo di flusso identificato in fase di progettazione dettagliata, per farlo utilizzeremo il Prime Decomposition Theorem (Fenton-Willy).
Il teorema afferma che ogni flow graph ha un'unica decomposizione in una gerarchia di flow graph primitivi (prime), detta "albero di decomposizione".
Ciò che si fa in pratica quindi è decomporre il codice in un opportuno sequenziamento di primitive di base (D-strutture) di modo che si possa capire quanto il nostro flow graph è d-strutturato.
Vediamo nell'esempio come si passa da codice a flowgraph e poi ad albero di decomposizione.



La prima istruzione è if then else D_1 → radice D_1. If true tre istruzioni → P_3 di cui un if then D_0, un singolo nodo procedurale P_1 e infine una while D_2.
D'altra parte se la radice è falsa abbiamo un if then D_0 a cui è annidata una repeat until D_3.

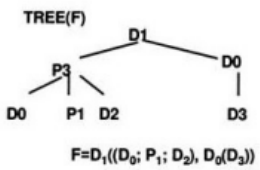
$$F = D_1((D_0; P_1; D_2), D_0(D_3))$$



Determinare l'albero di decomposizione di un grafo di flusso permette di definire le nostre metriche per capire se il codice risulta ben strutturato.
Daremo esempio di due metriche: la Depht of Nesting e la D-structureness. Esse si definiscono per le primitive di base e poi per applicazioni di sequencing e nesting.
La Depth of Nesting *n (F)* di un grafo di flusso F è:
- Per Primitive di Base (Primes): n (P_1) = 0; n (P_2) = n (P_3) = … = n (P_k) = 1:
n(D_0) = … = n(D_3) = 1
- Sequencing: quando si fa sequencing non si aggiunge alcun annidamento, quindi la depht of nesting di una sequenza corrisponde al max tra i singoli sequenziati
n($F_1, F_2, \ldots, F_k$) = max{ n (F_1, n (F_2), …, n (F_k)}
- Nesting: quando invece si fa nesting allora si introduce un ulteriore livello di annidamento quindi aggiungo 1:
n (F ($F_1, \ldots, F_k$)) = 1 + max{n (F_1), …, n(F_k)}

- **Example:**

F= $D_1((D_0;P_1;D_2), D_0(D_3))$



$F=D_1((D_0; P_1; D_2), D_0(D_3))$

$n(F) = n(D_1((D_0;P_1;D_2), D_0(D_3))) =$

$= 1 + max\{ n(D_0;P_1;D_2), n(D_0(D_3)) \} =$

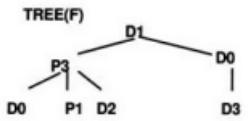
$= 1 + max\{ max\{ n(D_0), n(P_1), n(D_2) \}, 1+n(D_3) \} =$

$= 1 + max\{ max\{ 1,0,1 \}, 2 \} = 1 + max\{ 1,2 \} =$ ***3***

Più la depht of nesting è grande più il codice è complesso.
La D-Structurness d(F) permette di capire quanto è strutturato il codice.
Diciamo in particolare che un programma è strutturato se è D-strutturato.

Come prima:
- Primitive di base: $d(P_1) = 1; d(D_0) = \ldots = d(D_3) = 1$
- Sequencing: $d(F1;\ldots;Fk) = min\{ d(F_1), \ldots, d(F_k$
- Nesting: $d(F(F_1, F_2, \ldots, F_k) = min\{d(F), d(F_1), d(F_2), \ldots, d(F_k)\}$

- **Example:**
  $F = D_1((D_0;P_1;D_2), D_0(D_3))$



$d(F) = d(D_1((D_0;P_1;D_2), D_0(D_3))) =$ $F=D_1((D_0; P_1; D_2), D_0(D_3))$
$= min\{ d(D_1), d(D_0;P_1;D_2), d(D_0(D_3)) \} =$
$= min\{ d(D_1), min\{ d(D_0), d(P_1), d(D_2) \},$
$min\{ d(D0), d(D_3) \}\} =$
$= min\{ 1, min\{ 1,1,1 \}, min\{ 1,1 \}\} = min\{ 1,1,1 \} = 1$

➔ **F is D-structured** (F is built up of common primes, i.e. simple structures allowable in structured programming)

Se $d(F) = 1$ → il grafo di flusso è D-strutturato, ossia costruito partendo solo da primitive di base! Esistono casi di grafi non D-strutturati se si utilizzano primitive particolari.
Un'altra misura interessante che permette di valutare la complessità del codice è la Complessità Ciclomatica, essa può esser calcolata o sul flow graph ricavato come soluzione della fase progettuale o sul codice (quindi sia in fase di progettazione dettagliata che in fase di codifica).
Dato un flow graph F, la sua complessità ciclomatica $v(F)$ è pari al numero di archi meno il numero di nodi più 2 $v(F) = e - n + 2$
Questo valore misura quindi il numero di percorsi linearmente indipendenti di F (ossia tale che tale percorso non è insieme (combinazione lineare) di altri percorsi).
Riguardo il metodo basato non sul flow graph ma su codice, $v(F) = 1 + d$ ossia il numero di nodi predicati (decisionali) + 1.

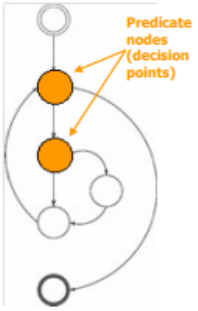
- La complessità delle primitive è quindi v(F) = 1 + d.
- Se invece si introduce il sequenziamento allora
$v(F_1;\ldots; F_n) = sum\ i = 1\ to\ n\ (v(F_1) - n + 1)$ (n è il numero di flow graph messi in sequenza).
- La complessità ciclomatica in caso di nesting è invece misurata come segue:
$v(F(F_1;\ldots;F_n)) = v(F) + \text{sum } i = 1 \text{ to } n\ (v(F_i) - n)$

Example: Flowgraph-based

$v(F) = e - n + 2$
$v(F) = 7 - 6 + 2$
$v(F) = 3$

or

$v(F) = 1 + d$
$v(F) = 1 + 2 = 3$



Example: Code-based

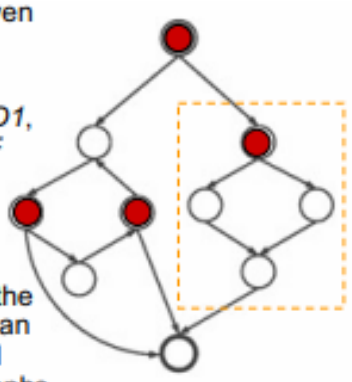
```
#include <stdio.h>
main()
{
int a ;
scanf ("%d", &a);
if ( a >= 10 )
    if ( a < 20 )      printf ("10 < a< 20 %d\n" , a);
    else               printf ("a >= 20    %d\n" , a);
else                   printf ("a <= 10    %d\n" , a);
}
```

v(F) = 1 + d = 1 + 2 = 3

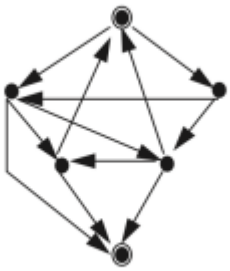
Si ricorda che d = nodi predicati = istruzioni decisionali

Ma cosa ci facciamo con la misurazione della complessità ciclomatica che restituisce un numero puro? Esso rappresenta una misura generale di complessità del nostro codice, più è alto più il codice risulta tipicamente difficile da mantenere e da testare.
McCabe suggerisce che quando questo numero inizia ad esser superiore a 10 allora il modulo inizia ad esser problematico.
Esiste un'altra misura introdotta proprio da McCabe: la Complessità Essenziale.
Essa è $ev(F) = v(F) - m$ dove m è il numero di sottografi di F di tipo D_0, D_1, D_2 *o* D_3. (si esclude P). Deve essere 1 se il grafo è d-strutturato (ha $d(F) = 1$).
La complessità essenziale rappresenta quindi il grado con cui il grafo di flusso può essere ridotto attraverso decomposizioni di sottografi D_0, D_1, D_2 *e* D_3.

- **Essential complexity** of a program with flowgraph *F* is given by:
  ev(F) = v(F) – m
  where *m* is the number of *D0*, *D1*, *D2* and *D3* sub-flowgraphs of *F*
- Example:
  v(F) = 5
  ev(F) = 5 – 4 = 1
- Essential complexity indicates the extent to which the flowgraph can be reduced by decomposing all *D0*, *D1*, *D2* and *D3* sub-flowgraphs (*ev(F)* = 1 for a D-structured program with flowgraph *F*)



"Spaghetti code", qui sotto un esempio di un grafo non d-strutturato.



Essential complexity = 6

La complessità ciclomatica ha il vantaggio di essere una misura oggettiva e generica di complessità del programma, tuttavia ha gli svantaggi di poter essere usata solo a livello di singola componente, due programmi con stessa complessità ciclomatica potrebbero essere diversi a livello di complessità.