- ***Lezione 30*** - Esercizi

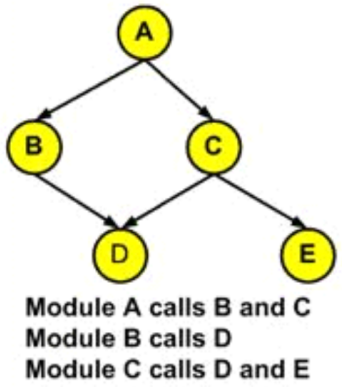
- ***Lezione 31***

  ➔ *Dodicesimo Blocco*

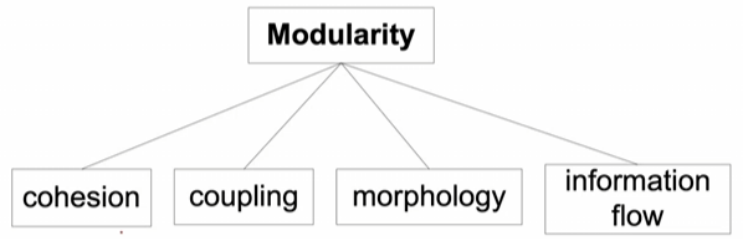
Le misure, vengono usate in fase di progettazione preliminare e dettagliata.
Affronteremo in particolare:

- le *misure intermodulari* → che permettono di quantificare la dipendenza tra moduli in base all'architettura software determinata in fase di progettazione preliminare.
- le *misure intramodulari* → che invece quantificano i singoli moduli (nella fase di progettazione dettagliata)

Un modulo come sappiamo è una sequenza contigua di istruzioni, delimitata da alcuni elementi e che ha un certo identificatore (quando si pensa a un modulo si pensa a una parte di software che può esser compilata indipendentemente).
È importante saper misurare la soluzione architetturale in quanto tutte le decisioni che prendiamo in questa fase hanno impatto significativo sul software risultante, in particolare su attributi di qualità come facilità di implementazione, affidabilità, manutenibilità e riusabilità (NB: affidabilità importante non solo per software critico).
Le misure ci permettono quindi di avere un feedback che ci faccia capire se le caratteristiche del software fanno al caso nostro (soddisfano requisiti etc…).
Il concetto di modulo introdotto in fase di progettazione preliminare poi collegato in modo diretto (relazione 1 a 1) con i moduli a livello di codice.
Le connessioni tra moduli identificate a livello di progettazione quindi tipicamente saranno riferimenti tra moduli a livello di codice, così come le interfacce di scambio dati identificate in fase di progettazione corrispondono alla condivisione dati a livello di codifica.
Torniamo ora al concetto di architettura software (anche chiamata structure chart).
Un'architettura software sappiamo essere un insieme di componenti che hanno tra loro relazioni di dipendenza.
Si può quindi concettualizzare l'architettura software usando un grafo, dove i nodi sono i moduli e gli archi le relazioni di dipendenza.
Le relazioni possono rappresentare diverse cose (es. chiamata di procedura, flusso di dati etc…).



Quando si definisce l'architettura dei moduli è importante definire il valore della modularità, che si ricorda essere il grado con cui un software è definito da componenti discrete tale che il cambiamento di una di esse comporta minimo impatto sulle altre componenti.
Si ricorda quindi che alta modularità è desiderabile, in quanto se si ha bassa modularità allora è più facile fare errori → difficile manutenzione, meno riusabili, meno affidabili etc…
Sappiamo che per misurare la modularità si fa riferimento alle due metriche di Coesione e Coupling.
Per misurare in modo più completo la modularità introduciamo altre metriche che sono maggiormente legate alla struttura del grafo con cui rappresentiamo l'architettura dei moduli. Si aggiungono a cohesion e coupling le due metriche Morfologia e Information Flow.



- Coesione → grado con cui un modulo individualmente realizza un task ben definito
- Coupling → grado di interdipendenza tra moduli
  Si voleva massimizzare coesione e minimizzare coupling per ottenere elevato livello di modularità.
- Morfologia → misura la forma della structure chart cercando di capire qual è la forma migliore in funzione della modularità
- Information Flow → considera il flusso di informazioni tra moduli → interconnessione tra moduli non solo dal punto di vista di scambio dati ma anche dal punto di vista del flusso di controllo (flusso di dati scambio di dati, flusso di controllo interdipendenza, misureremo ciò con tecniche che ci permetteranno di misurare fan-in e fan-out del modulo che sono rispettivamente quanto un modulo riceve in ingresso e invia in uscita)

Iniziamo dalla morfologia. Essa è caratterizzata da: Size (numero di nodi e archi), Depht (si vuole un grafo che sia quanto più simile a un albero possibile → depht come distanza radice-nodo), Width (massimo numero di nodi tra tutti i livelli), Edge-to-Node Ratio (misura di connettività nel grafo, ossia quanto è denso → rapporto archi nodi). Chiaramente per avere elevata modularità si vuole bassa connettività.
Vediamo un semplice esempio

Size:

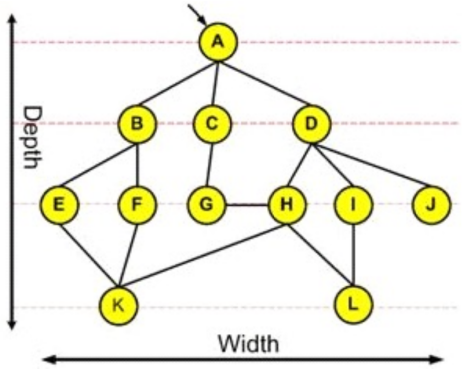
12 nodes

15 edges

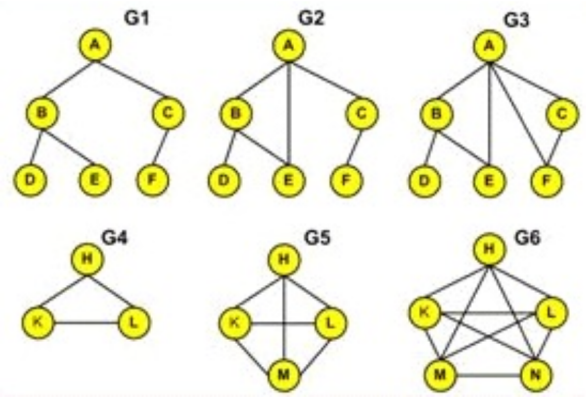
Depth: *4*

Width: *6*

e/n = 1.25



Maggiormente è densa l'architettura, maggiori sono i legami di interdipendenza tra moduli.
Un concetto importante per misurare la Morfologia è il concetto di Tree Impurity: esso misura quanto il grafo dell'architettura è lontano da un albero.
Sia m(G) quanto il grafo è diverso da un albero → minore m(G), migliore la morfologia. Più sono i cicli, peggiore è la morfologia, in particolare non si vogliono ovviamente grafi fortemente connessi o clique.
Si vogliono evitare situazioni come G5 e G6 e avvicinarsi quanto più possibile a G1.



Come misurare quantitativamente m(G)?

$$m(G) = \frac{\text{number of edges more than spanning tree}}{\text{maximum number of edges more than spanning tree}}$$

$$m(G) = \frac{2(e-n+1)}{(n-1)(n-2)}$$

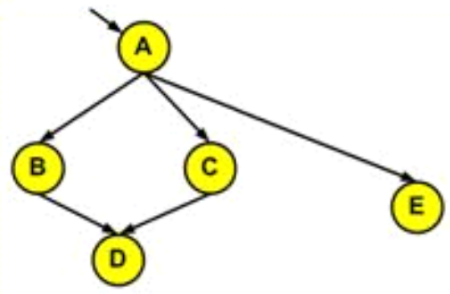
- Example

$$m(G_1) = 0 \quad m(G_2) = 0.1 \quad m(G_3) = 0.2$$
$$m(G_4) = 1 \quad m(G_5) = 1 \quad m(G_6) = 1$$

Spanning tree → albero che è sottografo di un grafo connesso e connette tutti i nodi.

Il numero massimo di archi per un grafo in più rispetto allo spanning tree è la differenza tra numero di archi di quel grafo con n nodi completo e quel grafo ma come spanning tree → n-1 archi spanning tree e (n(n-1))/2 per clique, la differenza porta a (n-1)(n-2) (da qui il denominatore).

Sopra invece numero di archi tra spanning tree e grafo effettivo.
M(G) varia tra 0 e 1.
Un altro per misurare sempre la Morfologia è l'Internal Reuse.
Essa indica il grado con cui i moduli sono riusati all'interno dello stesso prodotto.
Per misurarlo si usa r(G) che è e-n+1.
Meno r(G) significa meno riuso → migliore morfologia in quanto minor interdipendenza



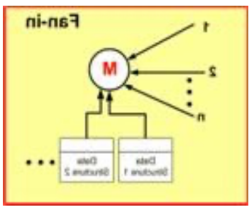
Module D is reused by modules B and C

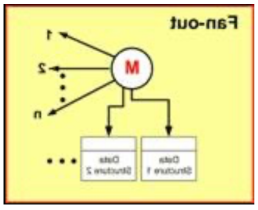
Example:

r(G1) = 0	***r(G2) = 1***
r(G3) = 2	***r(G4) = 1***
r(G5) = 3	***r(G6) = 6***

Questa tecnica di misurazione tuttavia non tiene conto del numero di chiamate fatte da un modulo ad un altro (non so D quante volte viene chiamato da B e C) e inoltre non si tiene

conto della dimensione dei moduli → bisogna utilizzare oltre che questa misura anche Tree Impurity per capire la morfologia.
Per approfondire come i moduli dipendano effettivamente uno dall'altro dobbiamo misurare l'Information Flow.
Questa misura assume che la complessità di un modulo dipenda da due fattori principali: la complessità intrinseca del modulo (il suo codice) e la complessità della sua interfaccia (ossia quanto è aperto all'ambiente circostante, quanto è dipendente da altri moduli o altri moduli dipendono da lui).
Il livello totale di information flow in un sistema è attributo intermodulare, tuttavia è possibile anche misurare l'information flow tra un singolo modulo e il resto del sistema come quindi attributo intramodulare.
Per misurare l'information flow si devono contare le connessioni tra un modulo con il resto dei moduli del sistema (fan-in e fan-out di un modulo), inoltre si assume che le misure di information flow sono basate su flussi di informazioni sia locali (ossia legati a un modulo che chiama un altro modulo se diretto o il valore di ritorno di un metodo invocato se indiretto) e globali (quando l'informazione tra moduli è condivisa e accessibile a tutti).
L'obiettivo dell'information flow è misurare quanto i moduli dipendano dagli altri, quindi può essere utilizzato per individuare le parti critiche del sistema e capire se la nostra progettazione può essere pericolosa in termini di affidabilità, manutenibilità, riusabilità etc…
- Fan-In di un modulo M: rappresenta il numero di flussi locali (diretti e indiretti) che terminano nel modulo M in aggiunta al numero di flussi globali (strutture dati globali utilizzate dal modulo M) → tutto ciò che entra nel modulo sia in termini di valori di ritorno che in termini di dati presi da strutture dati globali fa parte del Fan-in del modulo.
-Fan-Out: rappresenta viceversa tutto ciò che "esce del modulo": il numero di flussi locali (diretti o indiretti) che partono dal modulo M più il numero di aggiornamenti che M fa su strutture dati globali.





È importante misurarli in quanto se un modulo ha alto fan-out allora esso probabilmente influenza molti altri moduli, viceversa elevato fan-in indica che il modulo dipende da molti altri moduli.
Un modulo con elevato fan in e fan out è sicuramente un modulo che è al centro del sistema, mentre uno con basso fan in e fan out alla periferia del sistema.

L'obiettivo è ovviamente ridurre il fan in e fan out per ogni modulo in quanto se elevato allora indica un modulo complesso che potrebbe portare a errori in quanto il modulo può eseguire più di una funzione → 1) se devo modificarlo devo modificare molti altri moduli e inoltre 2) se voglio modificare una funzione devo guardare in più punti, non ho raffinato bene i moduli affinché ognuno svolga un unico compito ben preciso.
Si introduce quindi la misura dell'Information Flow.

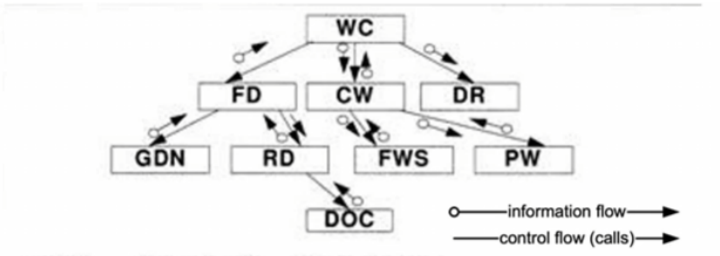
- Information flow (IF) for module M_i [Henry-Kafura, 1981]:

$$IF(M_i) = [\ fan\text{-}in(M_i) \times fan\text{-}out(M_i)\ ]^2$$

- IF for a system with *n* modules is:

$$IF = \sum_{i=1}^{n} IF(M_i)$$

Questa misura tiene conto dell'information flow sia come flusso dati che come flusso di controllo (quindi si contano anche le chiamate di un modulo a un altro senza effettivo scambio di dati).
Esiste una variante di information flow measure, la Shepperd, che si limita al flusso di dati.
Vediamo un esempio di questa misura con la variante Shepperd. Abbiamo una buona structure chart (in quanto ad albero). Ogni arco rappresenta chiamate fatte tra moduli (flusso di controllo) mentre le frecce con pallino flusso di dati.



Module	fan-in	fan-out	[(fan-in)(fan-out)]^2
WC	2	2	16
FD	2	2	16
CW	3	3	81
DR	1	0	0
GDN	0	1	0
RD	2	1	4
FWS	1	1	1
PW	1	1	1

IF = 119
(*Shepperd* variant)

(autoesplicativo, vedi frecce con pallino fuori e dentro per contare fan in e out per poi calcolare la shepperd measure dove si conta solo flusso di dati).
Quindi in generale non possiamo misurare direttamente la modularità, ma per capire il livello dobbiamo misurare i suoi 4 sottoattributi che invece sono direttamente misurabili.
Quando si passa alla parte di progettazione dettagliata ci si focalizza di più sulla complessità del singolo modulo, ora introduciamo delle misure strutturali (fanno riferimento alla struttura del modulo) che valgono sia in fase di progettazione dettagliata che in fase di implementazione (la soluzione è trovata in fase di progettazione dettagliata, in fase di implementazione solo traduzione della soluzione in linguaggio specifico).

Quando si parla di misure strutturali come detto si fa riferimento alla struttura del modulo, tre principali componenti:

- Struttura del flusso di controllo → sequenza di istruzini del programma
- Flusso di dati → si tiene traccia dei dati creati e gestiti dal modulo
- Struttura dei dati → organizzazione dei dati indipendenti dal modulo

È importante usare queste misure perché importanti in fase di reverse engineering (manutenibilità), testing (un passaggio importante è path coverage in questo senso, ossia cercare di percorrere tutti i possibili percorsi di esecuzione per rilevare eventuali errori), ristrutturazione codice, data flow analysis (si ricorda in questo senso l'importanza di modelli come il data flow diagram)…
Come rappresentare la struttura del singolo modulo? Attraverso un grafo del flusso di controllo. Una volta rappresentata la struttura potremo definirne la complessità usando la complessità ciclomatica, misura strutturale che vedremo si può fare sia a livello di codice che di modulo come grafo di flusso.

Come si esercita il flusso di controllo in un modulo?
Si definiscono delle strutture di controllo di base (BCS Basic Control Structure), che sono null'altro che i meccanismi essenziali di control flow usati per costruire la struttura logica del programma. Tre tipi di BCS:

- Sequenza (serie di istruzioni senza che vi siano altre BCS ad influire su di esse),
- Selezione (if, then, else…),
- Iterazione (while, repeat until …).

Esistono anche strutture di controllo avanzate ACS che permettono di introdurre nuovi concetti come chiamata di funzione/procedura, ricorsione, interrupt e concorrenza.
Vedremo come queste strutture di controllo di base vengono combinate per definire la struttura complessiva del modulo.

- ***Lezione 32***

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

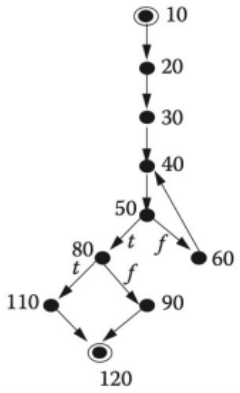
Tipicamente questi grafi di flusso hanno una struttura costituita da strutture di controllo di base e avanzate, noi ci focalizzeremo in questa fase di introduzione di metriche sulle strutture di controllo di base (sequenza, selezione, iterazione).

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

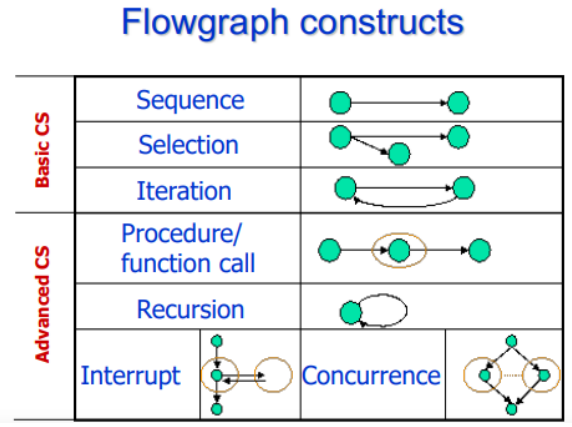
- Nodo Procedurale: nodo con un solo arco uscente (significa che quando il nodo termina la propria esecuzione esiste uno e un solo nodo successivo)
- Nodo Predicato: nodi con numero di archi uscenti != 1 (es. istruzioni decisionali, terminata l'esecuzione del nodo si "sceglie" uno e un solo nodo dei 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).

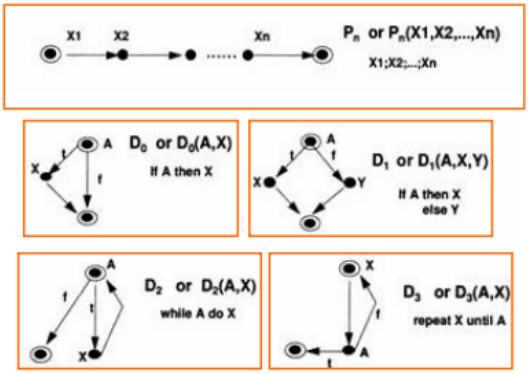
```
10  INPUT P
20  Div = 2
30  Lim = INT(SQR(P))
40  Flag = P/Div - INT(P/Div)
50  IF Flag = 0 OR Div = Lim THEN 80
60  Div = Div + 1
70  GO TO 40
80  IF Flag <>0 OR P>4 THEN 110
90  PRINT Div; "Smallest factor of"; P; "."
100 GO TO 120
110 PRINT P; " is prime"
120 END
```



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



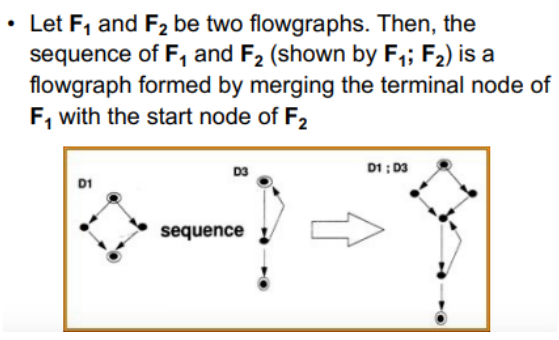
Limitandoci alle strutture base, è possibile individuare dei grafi di flusso comuni a tutti i programmi strutturati.



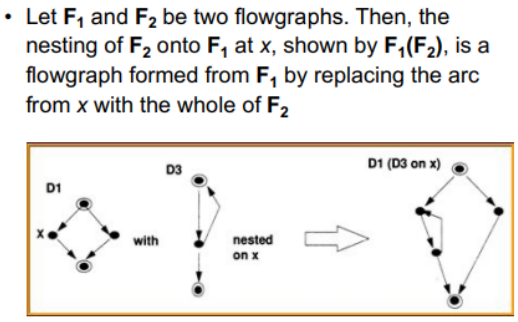
Pn come sequenza di n statement procedurali, D_0 o $D_0(A, X)$ l'if (dove A condizione e X codice da eseguire se la condizione è vera), poi D_1 o $D_1(A, X, Y)$ if then else, while do come D_2 e repeat until D_3. La differenza tra while do e repeat until è che con il repeat until l'istruzione che si 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:

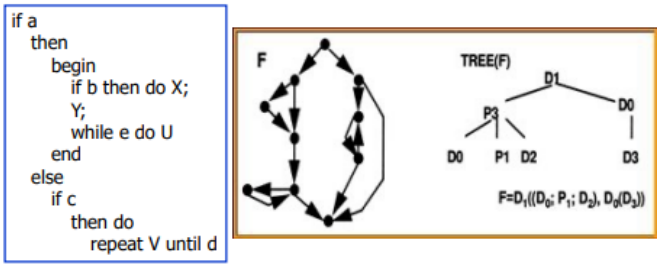
- *Sequenziamento:* dati due flow graph F_1 e F_2. Mettiamo i due flow graph in sequenza rendendo il nodo finale di F_1 il nodo iniziale di F_2, indichiamo il tutto con il simbolo; $(F_1; F_2)$.



- *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. $F_1(F_2)$

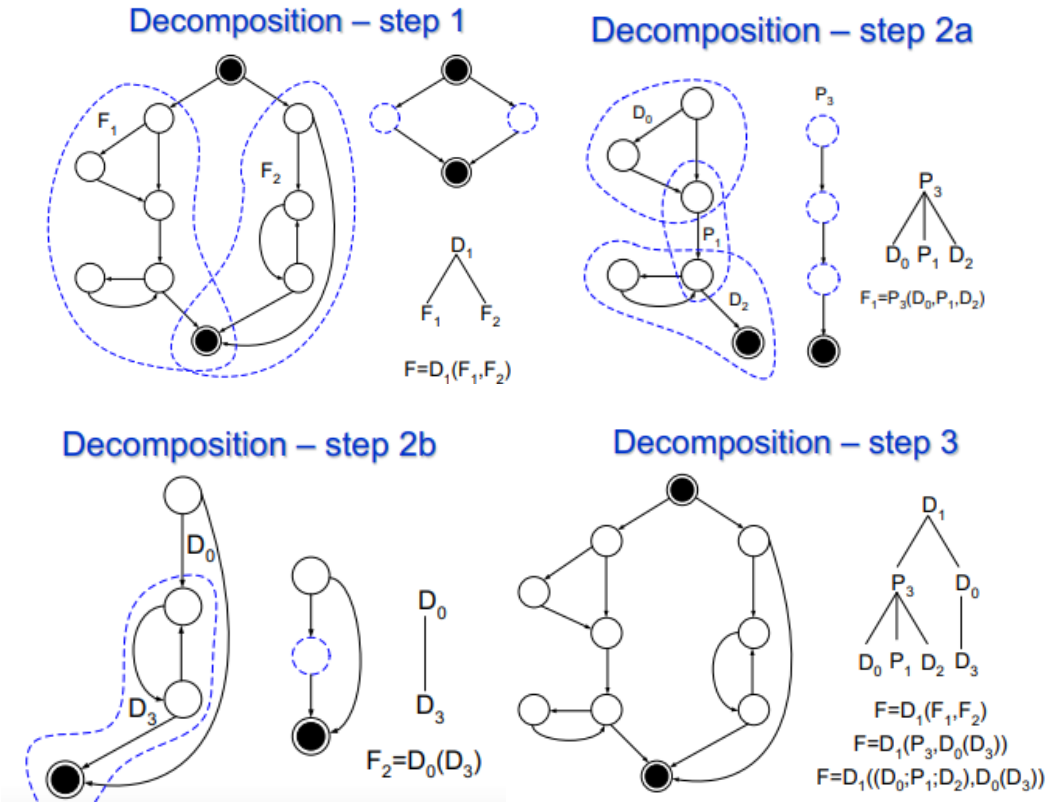


In generale, un Prime Flowgraph rappresenta un flowgraph che non può essere ulteriormente decomposto secondo il sequencing e il nesting.
Quindi P_1 (non P_n!), D_0, D_1, D_2 , D_3 sono tutti Prime Flowgraphs e sono dette D-structures.
Ciò che vogliamo fare a questo punto è verificare quanto è D-strutturato 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.

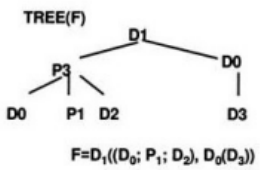
$$F = D_1((D_0; P_1; D_2), D_0(D_3))$$



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



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

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

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

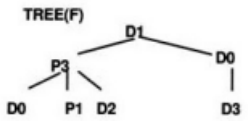
$= 1 + max\{ max\{ n(D_0), n(P_1), n(D_2) \}, 1+n(D_3) \} =$

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

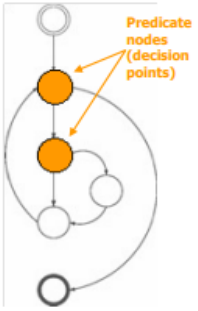
- La complessità delle primitive è quindi v(F) = 1 + d.
- Se invece si introduce il sequenziamento allora
$v(F_1;\ldots; F_n) = sum\ i = 1\ to\ n\ (v(F_1) - n + 1)$ (n è il numero di flow graph messi in sequenza).
- La complessità ciclomatica in caso di nesting è invece misurata come segue:
$v(F(F_1;\ldots;F_n))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

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

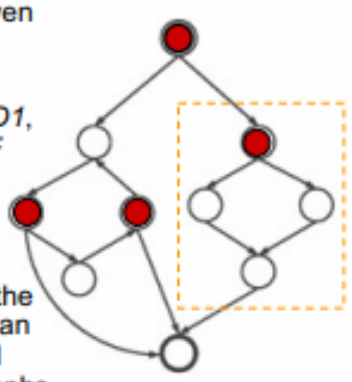
```
#include <stdio.h>
main()
{
int a ;
scanf ("%d", &a);
if ( a >= 10 )
    if ( a < 20 )      printf ("10 < a< 20 %d\n" , a);
    else               printf ("a >= 20    %d\n" , a);
else                   printf ("a <= 10    %d\n" , a);
}
```

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

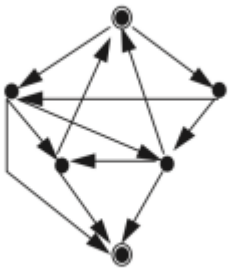
Si ricorda che d = nodi predicati = istruzioni decisionali

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.