

## Lez 32 (17/04)

Si entra ora nell'argomento relativo alle **misure** che si fanno 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 che invece quantificano i singoli moduli sono dette **misure intramodulari**, e ne parleremo quando parleremo della 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.

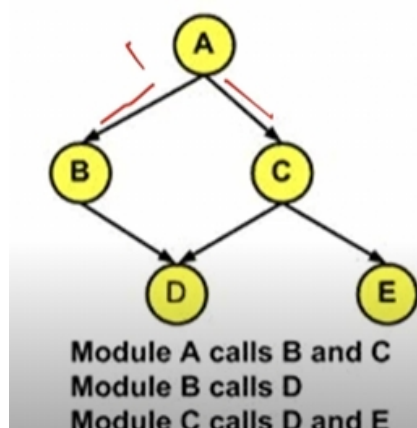
Le connessioni tra moduli identificate a livello di progettazione quindi tipicamente saranno riferimenti tra moduli a livello di codice, così come le interfacce di scambio dati identificate in fase di progettazione corrispondono alla condivisione dati a livello di codifica.

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...).

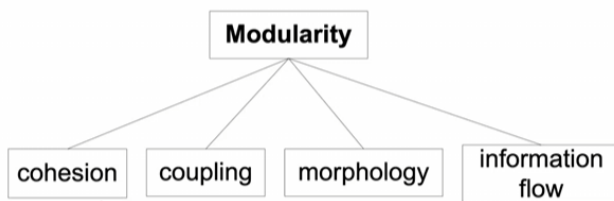


Quando si definisce l'architettura dei moduli è importante definire il valore della **modularità**, che si ricorda essere il grado con cui un software è definito da componenti discrete tale che il cambiamento di una di esse comporta minimo impatto sulle altre componenti.

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:

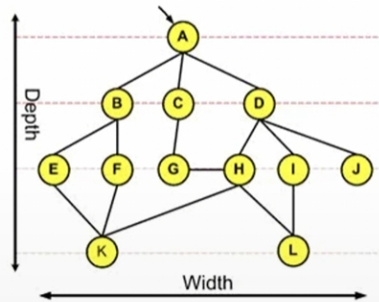
12 nodes

15 edges

Depth: 4

Width: 6

$e/n = 1.25$

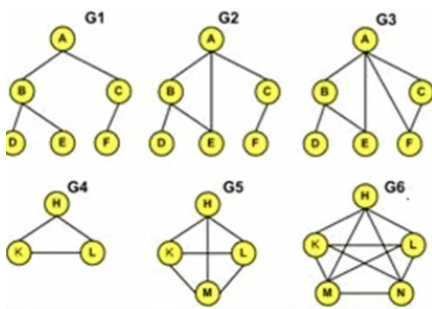


Maggiormente è densa l'architettura, maggiori sono i legami di interdipendenza tra moduli.

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).

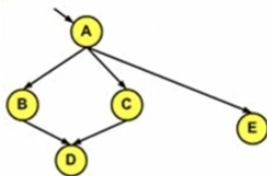
Sopra invece numero di archi tra spanning tree e grafo effettivo.

**$M(G)$  varia tra 0 e 1.**

Un altro modo 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** (quanto è dipendente da altri moduli o altri moduli dipendono da lui).

Il livello totale di information flow in un sistema è attribuito **intermodulare**, tuttavia è possibile anche misurare l'information flow tra un singolo modulo e il resto del sistema come quindi attributo **intramodulare**.

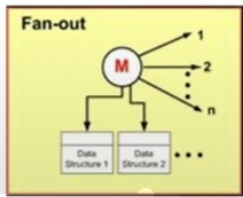
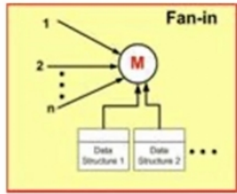
Per misurare l'information flow si devono contare le connessioni tra un modulo con il resto dei moduli del sistema (**fan-in** e **fan-out** di un modulo), *inoltre si assume che le misure di information flow sono basate su flussi di informazioni sia **locali** (ossia legati a un modulo che chiama un 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]:

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

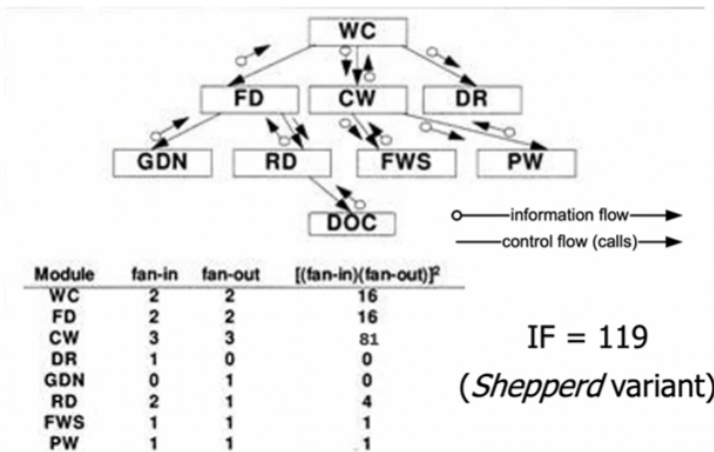
- IF for a system with  $n$  modules is:

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

**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.



(autoesplicativo, vedi frecce con pallino fuori e dentro per contare fan in e out per poi calcolare la shepherd 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:

- **Flusso di controllo** == sequenza di istruzioni 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 NB se si fanno tutti non vuol dire che il software sia privo di difetti in quanto magari non restituisce errore ma non si comporta comunque adeguatamente), **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.

### Lez 33 (22/04)

Abbiamo iniziato a vedere nella scorsa lezione le metriche tipicamente usate in fase di progettazione preliminare: oltre a coesione e coupling si valuta la morfologia dell'architettura che si vuol sviluppare rispetto al criterio di modularità. Dopodiché abbiamo iniziato a vedere quali metriche si utilizzano in fase di progettazione dettagliata, analizzando separatamente i singoli moduli.

*Mentre nella fase preliminare per utilizzare le metriche abbiamo visto l'architettura come un grafo dove ogni nodo è un modulo, similmente in fase di progettazione dettagliata il flow chart è rappresentato da un grafo che stavolta fa riferimento a un singolo modulo.*

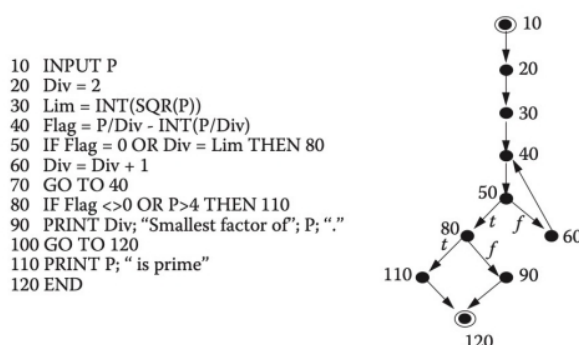
Tipicamente questi grafi di flusso hanno una struttura costituita da strutture di controllo di base e avanzate, noi ci focalizzeremo in questa fase di introduzione di metriche sulle strutture di controllo di base (**sequenza, selezione, iterazione**).

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  $\neq 1$  (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 ingresso 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, 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).**

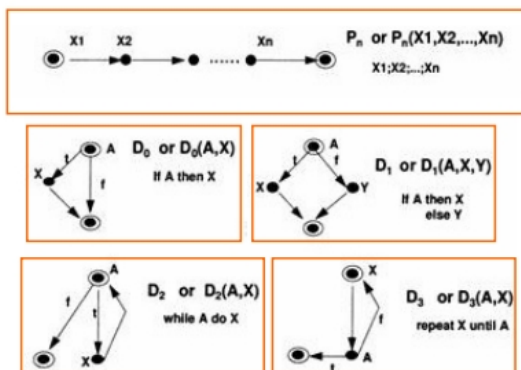
Per quanto riguarda le strutture avanzate che non vedremo l'invocazione a funzione rappresenta il passaggio da un nodo a un nuovo sottografo che poi viene eseguito a parte, la ricorsione una chiamata del modulo a se stesso, l'interrupt interruzione e concorrenza rappresentata da pallini gialli

### Flowgraph constructs

Basic CS	Sequence	
	Selection	
	Iteration	
Advanced CS	Procedure/function call	
	Recursion	
	Interrupt	
	Concurrency	

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

### Common Flowgraph Program Models



**$P_n$  o  $P_n(X_1, \dots, X_n)$**  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  **$D_2(A, X)$**  (while  **$A$**  do  **$X$** ) e **repeat  $X$  until  $A$**   **$D_3(A, X)$** . 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.

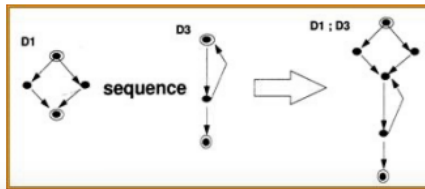
Un grafo di flusso ben strutturato a livello di sottofase di progettazione dettagliata è costituito solo ed esclusivamente da questi sottografi di base, opportunamente combinati. Due possibili operazioni per combinarle:

- **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 ; ( **$F_1; F_2$** ).



## Sequencing

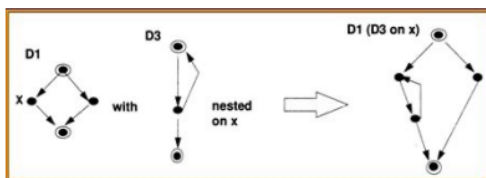
- Let  $F_1$  and  $F_2$  be two flowgraphs. Then, the sequence of  $F_1$  and  $F_2$  (shown by  $F_1; F_2$ ) is a flowgraph formed by merging the terminal node of  $F_1$  with the start node of  $F_2$



- **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 il codice  $X$  di  $F_1$  -> si sostituisce  $X$  con  $F_2$ .  **$F_1(F_2)$** .

## Nesting

- Let  $F_1$  and  $F_2$  be two flowgraphs. Then, the nesting of  $F_2$  onto  $F_1$  at  $x$ , shown by  $F_1(F_2)$ , is a flowgraph formed from  $F_1$  by replacing the arc from  $x$  with the whole of  $F_2$



In generale, un **Prime Flowgraph** rappresenta un flowgraph che non può essere ulteriormente decomposto secondo il sequencing e il nesting.

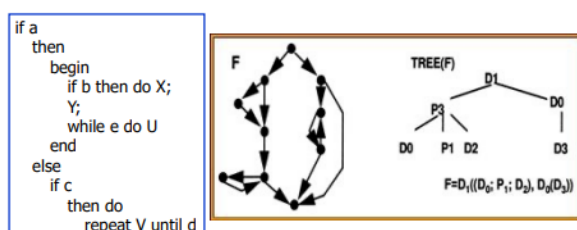
Quindi  $P_1$  (non  $P_n!$ ),  $D_0$ ,  $D_1$ ,  $D_2$ ,  $D_3$  sono tutti Prime Flowgraphs e sono dette **D-structures**.

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. (si determina **l'albero di decomposizione**)

Vediamo nell'esempio come si passa da codice a flowgraph e poi ad albero di decomposizione.

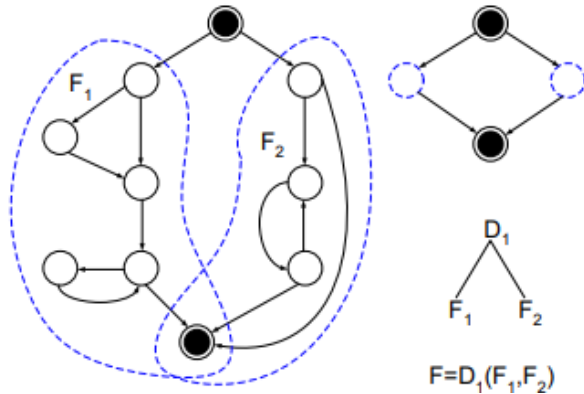


La prima istruzione è if then else D1 -> radice D1. If true tre istruzioni -> P3 di cui un if then D0, un singolo nodo procedurale P1 e infine una while D2.

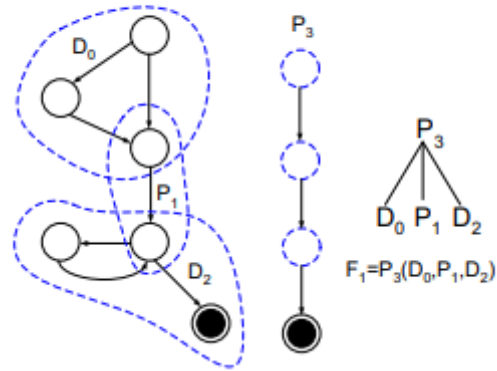
D'altra parte se la radice è falsa abbiamo un if then D0 a cui è annidata una repeat until D3.

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

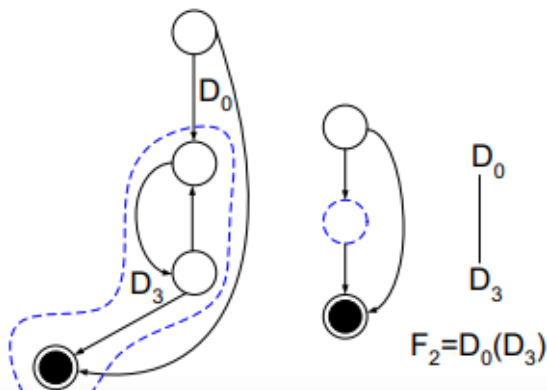
Decomposition – step 1



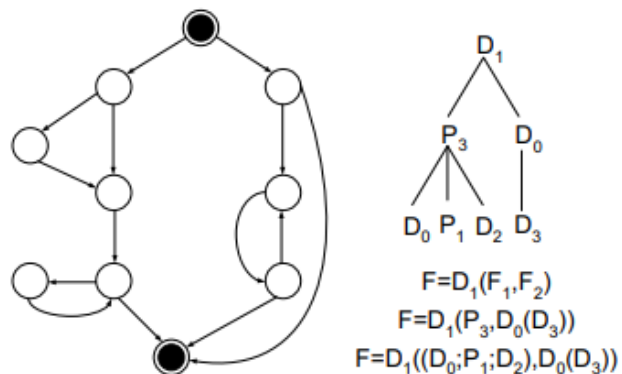
Decomposition – step 2a



Decomposition – step 2b



Decomposition – step 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 **Depth 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):  $n(P1) = 0$ ;  $n(P2) = n(P3) = \dots = n(Pk) = 1$ :

$n(D0) = \dots = n(D3) = 1$

- **Sequencing**: quando si fa sequencing non si aggiunge alcun annidamento, quindi la depth of nesting di una sequenza corrisponde al max tra i singoli sequenziati

$n(F1;F2;\dots;Fk) = \max\{n(F1), n(F2), \dots, n(Fk)\}$

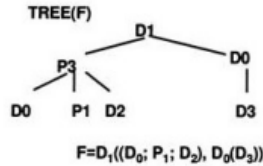
- **Nesting**: quando invece si fa nesting allora si introduce un ulteriore livello di annidamento quindi aggiungo 1:

$n(F(F1,\dots,Fk)) = 1 + \max\{n(F1), \dots, n(Fk)\}$

### Depth of Nesting: example

• Example:

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



$$\begin{aligned} n(F) &= n(D_1((D_0; P_1; D_2), D_0(D_3))) = \\ &= 1 + \max\{n(D_0; P_1; D_2), n(D_0(D_3))\} = \\ &= 1 + \max\{\max\{n(D_0), n(P_1), n(D_2)\}, 1 + n(D_3)\} = \\ &= 1 + \max\{\max\{1, 0, 1\}, 2\} = 1 + \max\{1, 2\} = 3 \end{aligned}$$

Più la depth 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.

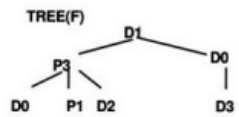
Come prima:

- **Primitive di base:**  $d(P1) = 1$ ;  $d(D0) = \dots = d(D3) = 1$
- **Sequencing:**  $d(F1; \dots; Fk) = \min\{d(F1), \dots, d(Fk)\}$
- **Nesting:**  $d(F(F1, F2, \dots, Fk)) = \min\{d(F), d(F1), d(F2), \dots, d(Fk)\}$

### D-Structuredness: example

• Example:

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



$$\begin{aligned} d(F) &= d(D_1((D_0; P_1; D_2), D_0(D_3))) = \\ &= \min\{d(D_1), d(D_0; P_1; D_2), d(D_0(D_3))\} = \\ &= \min\{d(D_1), \min\{d(D_0), d(P_1), d(D_2)\}, \\ &\quad \min\{d(D_0), d(D_3)\}\} = \\ &= \min\{1, \min\{1, 1, 1\}, \min\{1, 1\}\} = \min\{1, 1, 1\} = 1 \end{aligned}$$

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

Se  $d(F) = 1 \rightarrow$  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  $v(F) = e - n + 2$

Misura il numero di cammini linearmente indipendenti nel grafo. In termini semplici: quanti percorsi minimi diversi devono essere testati per garantire che ogni ramo logico venga eseguito almeno una volta. (importante per il Path Testing!!!)

(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

$v(F_1; \dots; F_n) = \sum_{i=1}^n (v(F_i) - n + 1)$  (n è il numero di flow graph messi in sequenza).

- La complessità ciclomatica in caso di nesting è invece misurata come segue:

$v(F(F_1; \dots; F_n)) = v(F) + \sum_{i=1}^n (v(F_i) - n)$

#### Example: Flowgraph-based

$$v(F) = e - n + 2$$

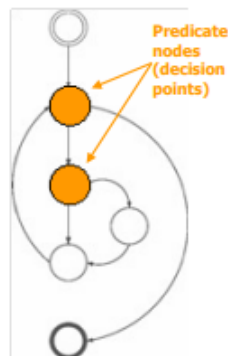
$$v(F) = 7 - 6 + 2$$

$$v(F) = 3$$

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/concorrenti**

**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 D0, D1, D2 o D3 (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 D0, D1, D2 e D3.**

- **Essential complexity** of a program with flowgraph  $F$  is given by:

$$ev(F) = v(F) - m$$

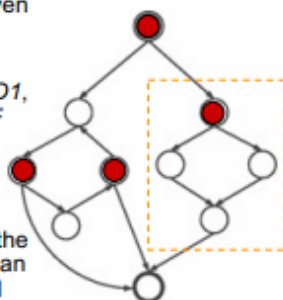
where  $m$  is the number of D0, D1, D2 and D3 sub-flowgraphs of  $F$

- Example:

$$v(F) = 5$$

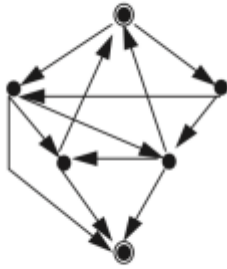
$$ev(F) = 5 - 4 = 1$$

- 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.

## Example Unstructured Prime



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.

### Lez 34 (29/04)

Si vedono alcuni esercizi sulle metriche di struttura studiate fino ad ora.

**Metriche necessarie per valutare il lavoro svolto in fase di progettazione preliminare e dettagliata, a livello architetturale per la parte preliminare e a livello di singolo modulo per la parte dettagliata.**

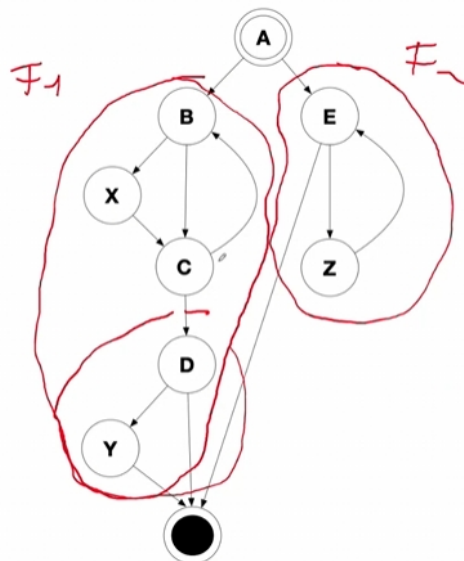
### Esercizio 1

- Dare il flow-graph del modulo descritto dalla formula

$$F = D1((D3(D0); D0), D2)$$

ed esprimerne

- pseudo-codice
- depth of nesting
- D-structuredness
- complessità ciclomatica



Per costruire il flow graph si procede per passi seguendo il nesting () e la sequenza ;. Abbiamo anzitutto D1 esterno a tutto dove D1 == if then else dove if true (D3(D0);D0) == F1 else D2 == F2.

Ma F1 è sequenza di due sottografi, uno F11 rappresenta un repeat until con annidato una D0 == if then e F12 che è una if then. F2 invece è una D2 == while do.

Ad ogni primitiva corrisponde poi uno pseudocodice scritto usando i costrutti di base della programmazione strutturata, ed è proprio quello che ci viene chiesto dall'esercizio.

	$F = D1(F1, F2)$ dove
	$F1 = P2(D3(D0); D0)$
	$F2 = D2$
<b>BEGIN</b>	
if A then	
<b>BEGIN</b>	$n(F) = 1 + \max\{F1, F2\}$
repeat	$= 1 + \max\{d((D3(D0); D0), d(D2))\} =$
if B then X;	$= 1 + \max\{\max\{d(D3(D0)), d(D0)\}, 1\} =$
until C;	$= 1 + \max\{\max\{1 + \max\{d(D0), 1\}, 1\} =$
if D then Y;	$= 1 + \max\{\max\{1+1, 1\}, 1\} =$
<b>END</b>	$= 1 + \max\{\max\{1+1, 1\}, 1\} =$
else	$= 1 + \max\{2, 1\} =$
while E do Z;	$= 1 + 2 = 3$
<b>END</b>	

Ricordiamo che per la depth of nesting quando abbiamo sequenziamento allora prendiamo il max dei singoli blocchi sequenziali, se nesting invece aggiungo 1 al max. Ricordiamo poi che la depth of nesting delle primitive  $P2, \dots, Pn$  e  $D0, \dots, D3$  è 1 (qui dovrebbe essere in figura  $n(D0)$  teoricamente e non  $d(D0)$ ).

$F = D1(F1, F2)$  dove  
 $F1 = P2(D3(D0); D0)$   
 $F2 = D2$

$d(F) = \min\{d(F1), d(F2)\} =$   
 $= \min\{\min\{d(D3), d(D0)\}, 1\} =$   
 $= \min\{\min\{1, 1\}, 1\} =$   
 $= \min\{1, 1\} = 1$

Qui sopra la d-structurness  $d(F)$ . Essendo 1 possiamo affermare che il grafo di flusso è d-strutturato (come ci aspettavamo, dato che usiamo strutture primitive di base “ben poste” su annidamento e sequenziamento). Esistono grafi di flusso particolarmente complessi, dove possono esistere cicli non corrispondenti a strutture primitive come while do o do while che possono nascere ad esempio in linguaggio assembly dove si usano istruzioni di salto simil go to, per cui si creano cicli particolari che portano una d-structurness maggiore di 1 e quindi grafi di flusso non d-strutturati. Ci dobbiamo aspettare quindi nel nostro caso di ottenere d-structurness 1, come dovremmo aspettarci di trovare complessità ciclomatica  $\leq 10$  e complessità essenziale 1 dal momento che la d-structurness = 1.

## Esercizio 2

- Dare il flow-graph del modulo descritto dalla formula

$F = D1((D0(D1); P1), D3(D0))$

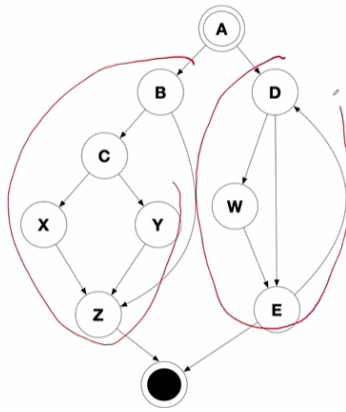
ed esprimerne

- pseudo-codice
- depth of nesting
- D-structuredness
- complessità ciclomatica

$v(F) = e - n + 2 = 13 - 9 + 2 = 6$  se calcolata con grafo

$v(F) = 1 + d = 1 + \text{if} + \text{repeat} + \text{if} + \text{if} + \text{while} = 6$  con lo pseudocodice contando le primitive  
 $ev(F) = v(F) - m$  dove  $m = \text{primitive base } D0, \dots, D3 \rightarrow ev(F) = 6 - 5 = 1$





Stesso ragionamento di prima, solo F2 annidato.

$F = D1(F1, F2)$  dove

$F1 = P2(D0(D1); P1)$

$F2 = D3(D0)$

$n(F) = 1 + \max\{F1, F2\}$

$= 1 + \max\{n(D0(D1); n(P1)), n(D3(D0))\} =$

$= 1 + \max\{\max\{1 + n(D1), 1\}, 1 + \max\{1\}\} =$

$= 1 + \max\{\max\{2, 1\}, 1 + 1\} =$

$= 1 + \max\{2, 2\} =$

$= 1 + 2 = 3$

$F = D1(F1, F2)$  dove

$F1 = P2(D0(D1); P1)$

$F2 = D3(D0)$

$d(F) = \min\{d(F1), d(F2)\} =$

$= \min\{d(D0(D1)), d(P1), d(D3), d(D0)\} =$

$= \min\{d(D0), d(D1), d(P1), d(D3), d(D0)\} =$

$= \min\{1, 1, 1, 1, 1\} = 1$

Complessità ciclomatica e essenziale

$v(F) = e - n + 2 = 14 - 10 + 2 = 6 = 1 + d = 1 + 5$

$ev(F) = v(F) - m = 6 - 5 = 1$

Passiamo ora all'argomento successivo direttamente correlato a quanto appena visto:  
la **Qualità del Software**.

Essa rappresenta per definizione il grado per cui il software possiede una serie di attributi “desiderabili”.

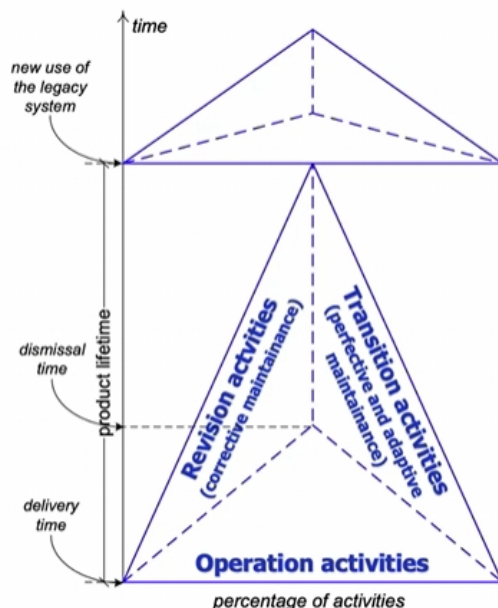
La qualità del software può essere analizzata da vari punti di vista:

- **Trascendentale**: *prodotto eccellente. Questo punto di vista l’abbiamo escluso fin dall’inizio, dal momento che l’obiettivo nello sviluppo software non deve essere tanto sviluppare un prodotto che sia eccellente quanto un prodotto che faccia correttamente quanto deve fare*
- **Utente**: **quanto il prodotto software contribuisca a raggiungere gli obiettivi dell’utente**
- **Prodotto**: **si valutano le caratteristiche del prodotto software** (correttezza, affidabilità etc..) **nonché la conformità con i requisiti**, qui si fa riferimento alla qualità del software con la definizione introdotta poco fa dove la si vede come combinazione di queste caratteristiche “desiderabili”, di qualità.
- **Organizzazione**: **ci si focalizza sui benefici aziendali -> costi e profitti** (se il prodotto è facile da mettere sul mercato, quanto il prodotto permetta di incrementare i profitti/ridurre i costi etc...).

Vediamo ora come definire in modo quantitativo questa combinazione di attributi, nel tempo infatti per cercare di dare una valutazione oggettiva a questi attributi sono stati introdotti dei modelli di qualità che definiscono in modo preciso cosa si intende per qualità del software.

*Importante introdurre questi modelli perché il concetto di qualità è un insieme di caratteristiche facilmente inquadrabili in modo soggettivo, ma si devono valutare in modo oggettivo per poter essere misurate e dare valutazione quantitativa.*

In particolare ci soffermiamo sul **Quality Model** introdotto da McCall: il **Quality Triangle**.



*Sulle ascisse la percentuale di attività che vengono svolte sul prodotto a partire dal momento in cui questo è immesso sul mercato. Sulle ordinate il tempo, che parte*

**dal momento di delivery (rilascio, momento in cui il software passa dallo stadio di sviluppo a quello di uso/manutenzione).**

Dal momento del rilascio il software sarà usato da diversi attori in modi diversi: vi saranno gli utenti che svolgeranno le così dette **Operation Activities** (ossia utilizzeranno il prodotto software e ba). Mentre gli utenti usano il software gli sviluppatori avvieranno delle attività di monitoraggio e manutenzione pressoché continue per molti motivi (non solo correttivi!).

Infatti tra le attività di manutenzione le **Revision Activities** (manutenzione correttiva) e **Transition Activities** (manutenzione perfetta e adattiva).

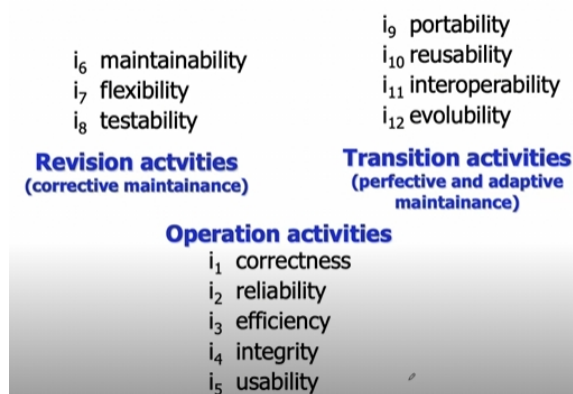
Ad un certo punto, quando gli utenti terminano di usare il software (Operation Activities), **il software entra nello stadio di dismissione.**

Come sappiamo questo stadio non è istantaneo e non rappresenta necessariamente la totale cestinazione del software, ma magari una sua **rielaborazione** (quindi ulteriori attività di sviluppo e manutenzione) **per arrivare eventualmente a un nuovo utilizzo del sistema legacy (triangolo in alto).**

Ciò che fa McCall, **per ogni possibile attività di Revision, Transition o Operation,** è definire degli **attributi di qualità del software** (detti nel caso di McCall **Indici di Qualità**).

Quindi qualità come combinazione di attributi desiderabili, nel caso di McCall come combinazione di 12 indici differenziati in base al tipo di attività.

### Quality Indices



### **Operation:**

- i1 **Correttezza**: rappresenta il grado con cui il prodotto soddisfa le specifiche e gli obiettivi degli utenti (quanto il software fa quello che l'utente vuole?)
- i2 **Affidabilità**: grado con cui il software esegue le sue funzioni con la precisione richiesta (diverso dall'affidabilità classica che avevamo definito come la probabilità che il software funzioni correttamente in un certo intervallo di tempo)
- i3 **Efficienza**: fa riferimento alle risorse di calcolo, quanto me serve potente il pc per eseguire il software
- i4 **Integrità**: il software è sicuro? Si fa riferimento alla sicurezza del software in termini di accesso esterno indesiderato

- i5 **Usabilità**: impegno richiesto per riuscire ad utilizzare il software da parte dell'utente

#### **Revision:**

- i6 **Manutenibilità**: effort necessario per individuare e correggere un difetto del programma

- i7 **Testabilità**: effort necessario per testare il prodotto e verificare che funzioni come desidero

- i8 **Flessibilità**: effort richiesto per modificare il prodotto (legato alla complessità del software stesso).

#### **Transition:**

- i9 **Portabilità**: effort richiesto per trasportare il prodotto da un ambiente operativo a un altro

- i10 **Riusabilità**: il grado con cui il prodotto o sue parti possono essere riusate in applicazioni differenti

- i11 **Interoperabilità**: quanto il software può essere accoppiato con altri prodotti

- i12 **Evolubilità**: effort richiesto per aggiornare il prodotto al fine di aggiornare eventuali nuovi requisiti (manutenzione perfetta anche detta evolutiva)

**Si ricorda che questi indici sono di alto livello, ciò che poi farà il modello di qualità è scomporli ulteriormente per arrivare a un livello tale da poter quantificare il valore dell'attributo.** Per ora li stiamo definendo solo a livello astratto.

Il modello di qualità funziona come segue. Misuro questi 12 indici, che rappresentano le possibili caratteristiche che permetterebbero di definire oggettivamente la qualità del mio software. **Per misurarli è necessario introdurre dei sottoindici, che mi permettano di arrivare ad un livello di dettaglio tale da poter quantificare nella pratica queste caratteristiche.**

Secondo McCall in particolare la qualità del software è definita da un vettore contenente i 12 indici di qualità appena visti  **$q = (i1, ..., i12)$** .

Ciascun indice può essere misurato facendo riferimento ad un insieme di **Attributi di Qualità** (attento alla terminologia e quindi alla differenza tra Indice di Qualità e Attributo di Qualità per McCall).

->  **$ij = (a1, a2, ..., an)$** .

**Un attributo può impattare anche più indici di qualità.**

Gli attributi sono 10 e tocca studia pure questi si godeeeee

**Complessità** come livello di comprensibilità, **Accuratezza** come precisione nel calcolo, **Completezza** come quanto il software ha implementato in modo completo le funzionalità richieste, **Consistenza** come utilizzo di approcci di design uniforme, **Error Tolerance** come quanto il software riesce a “sopravvivere” ad eventuali malfunzionamenti e continuare a funzionare, **Tracciabilità** grado di relazione tra due o più prodotti nel processo di sviluppo (es. tra codice e documento requisiti etc.), **Espandibilità** come quanto il software può essere esteso in termini di storage e funzioni, **Generalità** come quanto il software può essere applicato e adattato per

scenari diversi da quello per cui è stato progettato (*breadth of potential applications*), **Modularità** come grado di indipendenza tra moduli e **Auto-documentation** come quanto il software è in grado di aiutare l'utente attraverso un *help in linea*.

### Quality Attributes

<b>a<sub>1</sub> Complexity</b> level of understandability and verifiability of elements of the software and their interactions.	<b>a<sub>6</sub> Traceability</b> degree to which a relationship can be established between two or more products of the development process
<b>a<sub>2</sub> Accuracy</b> precision of computations and output	<b>a<sub>7</sub> Expandability</b> storage or functions can be expanded
<b>a<sub>3</sub> Completeness</b> full implementation of the required functionalities	<b>a<sub>8</sub> Generality</b> breadth of potential applications
<b>a<sub>4</sub> Consistency</b> use of uniform design and implementation techniques and notations	<b>a<sub>9</sub> Modularity</b> provisions of highly independent modules
<b>a<sub>5</sub> Error tolerance</b> continuity of operation ensured under adverse conditions	<b>a<sub>10</sub> Auto-documentation</b> in-line docs

Quindi McCall definisce alcuni di questi attributi per ogni indice e misura tutti questi attributi secondo specifiche metriche. Vediamo un esempio per 4 indici

### Software Quality Indices and Attributes

(+/- denotes positive/negative impact)

<b>i<sub>1</sub> Correctness</b> + Completeness + Consistency + Traceability	<b>i<sub>2</sub> Reliability</b> + Error Tolerance + Consistency + Accuracy - Complexity
<b>i<sub>7</sub> Flexibility</b> + Traceability + Consistency - Complexity + Modularity + Generality + Auto-documentation	<b>i<sub>12</sub> Evolvability</b> + Consistency - Complexity + Modularity + Expandability + Generality + Auto-documentation

$$i_1 = (a_3, a_4, a_6)$$

$$i_2 = (a_1, a_2, a_4, a_5)$$

$$i_7 = (a_1, a_4, a_6, a_8, a_9, a_{10}) \quad i_{12} = (a_1, a_4, a_7, a_8, a_9, a_{10})$$

Si noti come gli attributi possono avere impatto positivo o negativo per gli indici.

Ad es. per la flessibilità maggiore è la complessità, peggiore è l'indice di qualità!

In generale + se l'attributo influisce positivamente sull'indice, - altrimenti.

Per misurare gli attributi a loro volta sono necessari tipicamente dei sottoattributi per arrivare ad un'effettiva misura: es. per la modularità a<sub>9</sub> per misurarla si utilizzano tipicamente i sottoattributi come abbiamo visto nelle lezioni precedenti di coesione e coupling, morfologia e information flow (poi esistono anche altri modi, ogni modello usa il proprio e può far quindi riferimento a diverse metriche).

Per questi calcoli, che quindi possono avvenire usando diverse metriche, si introduce il **Checklist Method**, che mette a disposizione a chi deve valutare la qualità del software delle checklist che tipicamente pongono delle domande alle quali semplicemente l'utilizzatore deve porre una risposta.

**Sarà a carico del metodo checklist, una volta fornite le risposte, calcolare il valore complessivo dell'attributo.**

*Una volta calcolati i valori dei vari attributi questi vengono raggruppati per ottenere il livello del corrispondente indice (tenendo ovviamente conto dell'impatto, se positivo o negativo).*

### **Lez 35 (06/05)**

**Abbiamo detto come gli attributi possono essere definiti in termini a loro volta di sottoattributi fino ad arrivare ad un lvl di dettaglio tale per cui è possibile misurarlo direttamente nel prodotto software.**

Uno degli approcci per calcolare i valori di questi attributi come abbiamo introdotto l'altra volta è il **Checklist Method**.

*Come detto questo metodo si basa sul porre domande all'utilizzatore che, fornendo risposte, permetterà al checklist di calcolare il valore degli attributi.*

Vediamo ora come avviene la valutazione delle checklist.

**Una checklist rappresenta un insieme di domande, e ad ogni domanda sono associate quattro possibili risposte con un valore numerico.** Alcune domande potranno essere identificate come **Non Applicabili** se non si vuole prendere in considerazione la domanda per il calcolo del valore di un attributo, **Non Valutabili** se invece si vuole scegliere lo score più basso possibile tra quelli elencati nelle quattro risposte alla domanda.

Colui che si prende carico di rispondere a queste domande è il **Checklist Evaluation Team**, composto da almeno quattro persone che svolgono ruoli differenti e sono competenti in termini di qualità del software.

Si raccomanda che il team sia costituito da: un **Quality Assurance Specialist**, un **Project Leader**, un **System Engineer**, un **Software Analyst**.

Nella prima parte del corso, quando abbiamo parlato di ingegneria dei requisiti e in particolare delle modalità di valutazione della documentazione, abbiamo detto che esistono due tecniche principali: **Walkthrough** (meno formale, il documento viene letto dai valutatori per rispondere in modo accurato alle domande della relativa checklist) e **Ispezione** (più formale).

***In generale per ognuno dei 10 attributi definiti da McCall esiste una checklist corrispondente che permette di ottenerne il valore. Dopo averli valutati, secondo metodi specifici, è possibile risalire al valore dell'indice desiderato.***

In particolare il metodo per calcolare il valore numerico di ciascun attributo (**ECCETTO CHE PER LA MODULARITÀ**) è basato sulla seguente **formula**:



$$V_{attribute} = \frac{\sum_{i=1}^{\# questions} V_{answer_i}}{\sum_{i=1}^{\# questions} \max(V_{answer_i})}$$

(si ricorda infatti che a ciascuna risposta è associato un valore numerico).  
Calcolati i valori per ogni attributo, si procede al calcolo del valore dell'indice.

$$V_{index} = \frac{\sum V_{attribute(+)} + \sum (1 - V_{attribute(-)})}{\text{number of attr. associated to index}}$$

*Si noti come si prende in considerazione l'impatto positivo o negativo degli attributi sull'indice in questione, in particolare se impatto negativo allora metto 1-valore attr.*

**Chiaramente, essendo valori normalizzati, i valori degli attributi sono compresi tra 0 e 1 e lo stesso vale per i valori associati agli indici.**

Una volta ottenuti i valori degli indici, vengono definiti i così detti “livelli di accettazione” degli indici di qualità (**Acceptance Level Scale**)

### Acceptance Level Scale

- The output of the formula yielding  $V_{index}$  is a value between 0 and 1
- The following scheme can be used in order to identify threshold values for quality acceptance:

$V_{index}$	Quality Level
$0.66 < V \leq 1$	High
$0.33 < V \leq 0.66$	Medium
$0 < V \leq 0.33$	Low

Vediamo ora degli esempi di checklist per capire meglio il funzionamento di questa valutazione.

Immaginiamo di trovarci nella fase di progettazione architettuale (preliminare), dopo la quale abbiamo prodotto l'architettura del nostro sistema software.

Tra i 10 attributi consideriamo da calcolare a1 Complessità, a8 Generalità e a9 Modularità (per la quale vedremo un metodo di calcolo diverso da quello visto).

Iniziamo dalla Complessità, ecco la checklist:

### General aspects

- 1) Are modules, procedures and structure names significant or conform to a standard, if any.
  - 0 : Yes, almost always
  - 1 : Quite often
  - 2 : Rarely
  - 3 : No
- 2) Is the formalism utilised to describe the system architecture only one, or multiple ones are present.
  - 0 : Unique formalism, standard and properly chosen
  - 1 : Few formalisms, standard and properly chosen
  - 2 : Few formalisms and someone out of standard
  - 3 : A lot of formalisms, someone out of standard
- 3) Is the global system design properly structured and easy understandable by people without specific knowledge about the system.
  - 0 : Yes
  - 1 : Almost positive
  - 2 : Not properly structured
  - 3 : Bad structured, and hardly understandable

Chiaro come complessità abbia impatto negativo sugli indici -> caso migliore quello per cui si scelgono tutte risposte con 0 (si vuole ridurre la complessità, infatti nella successiva sommatoria per calcolare l'indice si dovrà fare il calcolo con 1-valore!). Queste erano domande generali, la checklist prosegue con domande più specifiche direttamente misurabili sull'architettura software nel nostro caso (metrics application). ***Chiaramente le checklist cambiano in base anche al tipo di documentazione che si utilizza*** (ogni documentazione fornisce informazioni diverse, un conto è lavorare con il documento di specifica requisiti un conto con l'architettura software).

- 4) Is it possible to deduce the class of information of each single data present into the logical model. Are all the utilisation of this data declared in the documentation and realised with the purpose to read or write that class of information.
  - 0 : No
  - 1 : Rarely
  - 2 : Quite often
  - 3 : Approximately always.

### Metrics applications

- 5) If is possible to perform measurements about the program complexity, based on the program calling graph, then use the metrics below defined.
  - 5.1) *Hierarchical complexity*: is the average number of modules per calling-tree level, that is the total number of modules divided by the number of tree levels
    - 0 : 0 up to 4
    - 1 : 4 up to 8
    - 2 : 8 up to 12
    - 3 : Less than 2 or greater than 12
  - 5.2) *Structural complexity*: is the average number of call for each module, that is the number of inter-module calls divided by the number modules.
    - 0 : Less than 2
    - 1 : 2 up to 4
    - 2 : 4 up to 8
    - 3 : Greater than 8

Per quanto riguarda invece l'attributo generalità questo impatta positivamente -> si vogliono valori più alti

### Generality checklist

- 1) Are all functions offered by modules (and their interfaces) properly planned so that is possible to reuse them in some implementations that wasn't planned at the start time of the project. (i.e. : a specific percentage [18%], or any percentage of any value)
  - 0 : No
  - 1 : Rarely
  - 2 : Quite often
  - 3 : Always
- 2) Concerning modules that haven't enough generality, is it possible to make them more generic with low effort (near 10% of global effort to obtain each of them).
  - 0 : No
  - 1 : Rarely
  - 2 : Quite often
  - 3 : Always
- 3) Are all the data structure manipulations combined into modules specifically dedicated to this.
  - 0 : No
  - 1 : Rarely
  - 2 : Quite often
  - 3 : Always

Vediamo ora come si calcola invece l'attributo **Modularità**.

*Sappiamo che un modo per misurarla riguarda l'utilizzo dei concetti e le misurazioni di Cohesion e Coupling, ma anche di Morfologia e Information Flow.*

In questo caso si fa riferimento solo alla coesione e al coupling.

**Alle risposte non sono associati valori numerici, ma delle etichette A, B C, o D.**

*Viene richiesto di fornire la percentuale di moduli in base al tipo (coesione coincidentale, logica o temporale, procedurale o comunicativa, informational o functional). Invece di separare i 7 li si raggruppa quindi in gruppi per un totale di 4 risposte.*

Lo stesso vale per il coupling, dove viene richiesto di definire la distribuzione percentuale di coppie di moduli in base al tipo di coupling (content, common, control, stamp o data).

#### **Cohesion**

- 1) Which is the percentage distribution of system modules according the following four types:
  - 1A modules with coincidental cohesion
  - 1B modules with logical or temporal cohesion
  - 1C modules with procedural or communicational cohesion
  - 1D modules with informational or functional cohesion

#### **Coupling**

- 2) Which is the percentage distribution of communication modes (coupling) between modules according to the following four types:
  - 2A pairs of modules with content coupling
  - 2B pairs of modules with common coupling
  - 2C pairs of modules with control coupling
  - 2D pairs of modules with stamp or data coupling

Successivamente si passa sugli attributi generali di modularità e si torna quindi alle classiche checklist. Anche in questo caso, essendo modularità un attributo con impatto positivo sull'indice, le risposte migliori sono quelle tali per cui il valore è più alto:

#### **General**

- 3) Is the system decomposition organised in a hierarchical way
  - 0 : No
  - 1 : Rarely
  - 2 : Yes, enough
  - 3 : Yes
- 4) Which is the percentage of hierarchical structures (each program contains a comparable number of modules)
  - 0 : less then 70%
  - 1 : within 70% and 80%
  - 2 : within 80% and 90%
  - 3 : more than 90%

Vediamo ora come calcolare il valore complessivo di modularità.

$$V_{answer_1} = \frac{(0 \times \%1A) + (1 \times \%1B) + (2 \times \%1C) + (3 \times \%1D)}{50}$$

$$V_{answer_2} = \frac{(0 \times \%2A) + (1 \times \%2B) + (2 \times \%2C) + (3 \times \%2D)}{50}$$

**Le prime due domande sono calcolate a parte come si vede sopra: in particolare si pesa zero la coesione e il coupling peggiore (coincidentale e content) mentre si pesa a 3 quelle migliori. Dividendo tutto per 50 si normalizza ottenendo quindi ancora un valore tra 0 e 1.**

$$V_{Modularity} = \frac{V_{answer_1} + V_{answer_2} + V_{answer_3} + V_{answer_4}}{\max(V_{answer_1}) + \max(V_{answer_2}) + \max(V_{answer_3}) + \max(V_{answer_4})}$$

Torniamo a vedere come si effettua la valutazione di questi attributi.

**La documentazione come anticipato deve essere analizzata in dettaglio dal Checklist Evaluation Team per rispondere a ciascuna domanda, e per farlo come anticipato e già spiegato nel primo modulo esistono tecniche come Walkthrough o Ispezioni più formali.**


Ogni membro del team deve rispondere alle domande in modo indipendente senza confrontarsi subito con gli altri. Talvolta vi possono essere domande della checklist non applicabili al caso specifico che si sta valutando, in tal caso quelle domande si escludono dal calcolo definendole **Not Applicable**.

Quando invece la domanda risulta Applicabile ma **Not Valutable** (non valutabile magari in quanto la documentazione non risulta conforme per riuscire a rispondere alla domanda) -> si assegna il punteggio più basso tra quelli disponibili.

Durante le riunioni di Walkthrough o i meeting di ispezione i membri del team finalmente si confrontano per verificare di aver dato le stesse risposte alle domande e il perché. Se sono diverse allora bisogna discuterne per arrivare a un'unica risposta comune, per poter finalmente calcolare il valore dell'attributo.

Si dà ad ogni attributo un template contenente la lista di domande della checklist, se la domanda non è valutabile o non applicabile, e il punteggio. Si fa lo stesso per gli indici con la lista degli attributi, se hanno peso negativo o positivo, il punteggio di ogni attributo, il calcolo dell'index e il quality level (low, high o avg).

Template for attribute scoring

Attribute: 			
Question #	N/A	Not Valuable	Score
-			0-1
-			
-			
Total Score (see formula on slide 12)			

Template for index evaluation

Index:		
Attribute	Weight (+/-)	Score
Evaluation (see formula on slide 13)		
Quality level (see acceptance scale on slide 14)		

L'utilizzo delle checklist (ognuna diversa per ogni attributo e in base al documento di specifica) e la valutazione di qualità fa parte di un'attività più generale molto importante nel progetto software: la **Software Quality Assurance**.

*Questa fase del progetto rappresenta l'approccio sistematico per assicurare che sia il processo software (sviluppo) che il prodotto software in sé siano conformi agli standard, processi e procedure stabilite.*

Gli **obiettivi della fase SQA** sono in particolare *migliorare la qualità del software* monitorando sia il software che il processo di sviluppo assicurandosi di star seguendo gli standard prestabiliti.

**Fare SQA è un'attività molto costosa che richiede personale esperto, tempo effort... quindi l'attività incide sui costi del progetto.**

*Per introdurre un programma SQA è quindi necessario anzitutto che i top manager che gestiscono il progetto siano d'accordo e supportino SQA con budget adeguato. A quel punto si dovrà identificare un piano di SQA stabilendo quali sono gli standard di riferimento, per poi realizzarlo al fine di valutare la qualità del software prodotto.*

Il ruolo del team di SQA è garantire ai manager responsabili che quanto si sta facendo appunto è conforme agli standard e alle procedure.

Si garantisce l'uso di un'appropriata metodologia per lo sviluppo, che si proceda secondo standard e procedure, che vengano realizzate opportune review per valutare quanto viene fatto e se viene fatto correttamente, che si producano documenti per supportare la manutenzione e il miglioramento del software, che si utilizzi un sistema di software configuration management per garantire la tracciabilità di quanto prodotto, che venga effettuato il testing e che questo venga superato correttamente, che si identifichino deviazioni ed eventuali problemi (se rilevati in modo tempestivo si possono affrontare in modo più semplice e meno costoso).

*Gli obiettivi principali di SQA sono quindi ridurre rischi monitorando in modo appropriato il software e il processo di sviluppo, assicurandosi la conformità con standard e procedure e assicurandosi anche che le inadeguatezze e gli eventuali problemi del prodotto software, processo software o standard siano portati immediatamente all'attenzione del management affinché si risolvano velocemente.*

**È importante dire che SQA non è responsabile della produzione di software di qualità, ma di verificare che chi lavora al prodotto lo faccia correttamente seguendo gli standard e le procedure cercando di capire se vi sono problemi nel mentre** (auditing, monitoraggio su chi produce il software per accorgersi se c'è qualcosa che non va).

Parlando di SQA abbiamo detto spesso i termini standard e procedure. Ma cosa rappresentano nel concreto?

Gli **Standard** in questo contesto rappresentano delle **linee guida alle quali il progetto software dovrebbe esser comparato per assicurarsi di procedere bene** (gli standard definiscono QUELLO che in generale dovrebbe essere fatto).

Il minimo standard indispensabile da seguire include:

- **Documentation Standard** (es. che si utilizzi un template per il documento di specifica dei requisiti piuttosto che inventarsene la struttura di sana pianta)
- **Design Standard** (standard di progettazione, si vuole cioè trasformare i requisiti software in progettazione software utilizzando un'adeguata documentazione progettuale come linguaggi di modellazione, specifiche di definizione degli algoritmi...)
- **Code Standard** (standard di codifica per ottenere codice che sia quanto più possibile uniforme)

Quando si utilizza il termine "**Procedure**" invece si fa riferimento alle **linee guida che mi suggeriscono COME procedere effettivamente nello sviluppo** (nella specifica, progettazione etc...) anche suggerendo chi deve farla, quando e cosa fare con i risultati.

Tutta l'attività svolta dal team di SQA deve essere definita e pianificata con precisione affinché il management sappia quello che il team farà.

Tutto ciò fa parte dell'SQA Planning, che è esso stesso un documento che segue uno standard ben preciso. Uno di questi standard per l'SQA Plan è l'IEEE che specifica gli obiettivi, le attività da svolgere, gli standard e procedure da seguire etc...

Abbiamo quindi capito come il tema della qualità software sia molto importante e preveda costi dedicati anche talvolta significativi. Ciò permette un lavoro più agevole a chi realizza il prodotto essendo guidato da standard e procedure riconosciute.

Dalla prossima lezione si approfondisce il Testing.