Metriche del Software

Principali metriche del Software

© Luigi Lavazza, 2001

Indice

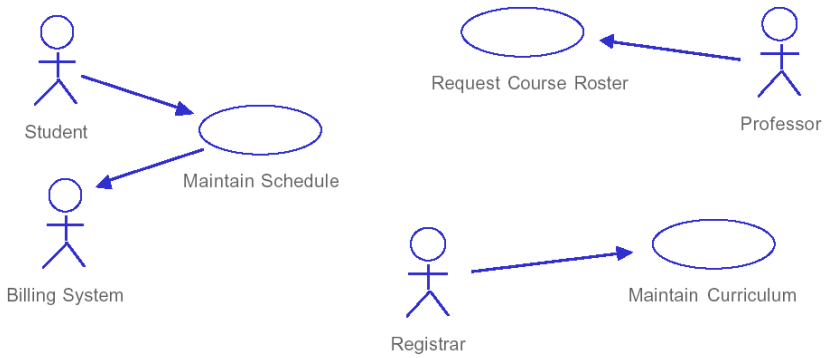
- Metriche dimensionali
  - Use case
  - Function Point
  - LOC

Metriche dimensionali

- La definizione delle metriche riguardanti le dimensioni del software dipende da diversi fattori, in particolare la fase dello sviluppo in cui si fa l'osservazione:
  - requisiti: numero di use case (in ambito object-oriented)
  - specifiche: function point
  - codifica: linee di codice (LOC), sono spesso indicate come "metriche dimensionali" per antonomasia

Metriche su use case

- Il numero di use case indica il numero di funzionalità percepibili dall'utente



Osservazioni

- Le funzionalità percepibili dall'utente sono rappresentabili a diversi livelli di granularità
  - *il numero cambia !*
- Catturano bene solo i requisiti funzionali
- In sostanza vanno bene nelle fasi alte, quando ci sono poche informazioni disponibili

I Punti Funzione

- I punti funzione vengono ideati nel 1975 da Allan Albrecht (IBM)
- L'idea fondamentale è di rappresentare le dimensioni di un prodotto software caratterizzato in base alle sue funzionalità, anziché rispetto alle sue dimensioni fisiche
- Definizioni seguenti:
  - IFPUG
  - Mark II Function Point
  - Feature point
  - Full Function Point
  - ...

La definizione originale

"una tecnica per validare le stime dello sforzo lavorativo necessario per progettare e sviluppare applicazioni software su misura"

Osservazioni iniziali

- Il "valore" di un prodotto software dipende dalle sue funzionalità
- La "funzionalità" è in relazione con gli input e gli output del sistema
- L'unità di misura deve essere comprensibile dall'utente e collegata ai requisiti da lui espressi
- Tale misura deve essere calcolabile nelle prime fasi dello sviluppo (quando si richiede la stima)
- Una misura di questo tipo è utilizzabile per valutare la "funzionalità rilasciata all'utente"

Gli obiettivi fondamentali

- Indipendenza dalla tecnologia
  - ci si basa sui documenti di specifica
  - realizzazioni diverse (per linguaggio di programmazione o per strumenti di sviluppo) danno la stessa misura
- Misurare tutte le funzionalità fornite all'utente
- Misurare solo le funzionalità fornite all'utente

Funzioni elementari (Function Types)

- Il metodo di Albrecht prevede di identificare e contare il numero di:
  - Input
  - Output
  - Archivi logici
  - Archivi di interfaccia
  - Inquiry esterne
- Questi elementi dovrebbero rappresentare nel loro complesso la "manifestazione esterna" di qualunque applicazione, cioè ne rappresentano le funzionalità.

Conteggio delle funzionalità di "gestione dati"

- Archivio logico interno
  - un insieme omogeneo di dati o di informazioni di controllo usati e gestiti dall'applicazione
- Archivio logico di interfaccia
  - un insieme omogeneo di dati o di informazioni di controllo usati dall'applicazione ma generati e mantenuti da altre applicazioni

Conteggio delle funzionalità di "gestione funzioni"

- Input esterno
  - operazione elementare di elaborazione di un dato o informazione di controllo proveniente dall'esterno dell'applicazione
- Output esterno
  - operazione elementare che genera un dato o un'informazione di controllo verso l'esterno dell'applicazione
- Inquiry esterna
  - operazione elementare che coinvolge input e output

Procedimento di definizione

- Albrecht ha considerato 24 applicazioni
  - 18 COBOL, 4 PL/1, 2 DMS
  - dalle 3000 alle 318000 LOC
  - da 500 a 105200 ore-persona
- Ha definito una formula per il calcolo di FP in base ai function types, e una per il calcolo dello sforzo in base ad FP, cercando di minimizzare gli errori per l'insieme delle 24 applicazioni
- Risultato: errori del 24% sulle stime dello sforzo

Applicazione

- Al termine della progettazione degli aspetti esteriori dell'applicazione
- All'installazione e quindi periodicamente (per valutare la produttività di sviluppo e manutenzione)
- Calcolo in base ai Function Types, opportunamente pesati
- Correzione finale (da -35% a +35%) in base a 14 parametri
- Il numero finale è utilizzabile come variabile indipendente per il calcolo di diverse grandezze (come lo sforzo di sviluppo).

Tabella dei pesi

- Si utilizza una tabella in cui appaiano alcuni pesi per i quali devono essere moltiplicati i diversi i valori dei Function Types (la complessità è valutata in base alle caratteristiche dell'applicazione).

Function Types	**Pesi**		
	Semplice	Medio	Complesso
N. Inputs	3	4	6
N. Outputs	4	5	7
N. Richieste	3	4	6
N. Files	7	10	15
N. Int. esterne	5	7	10

Formula per il calcolo di FP

- FP è dato dalla seguente formula:

$$FP = FP\ grezzi \times \left(0.65 + 0.01 \times \sum_{i=1}^{14} F_i \right)$$

Parametri correttivi F_i

0	1	2	3	4	5
Ininfluente	Incidenza scarsa	Incidenza moderata	Incidenza media	Incidenza significativa	Incidenza essenziale

Il sistema richiede procedure di recovery e backup affidabili?

È richiesta la trasmissione di dati?

Vi sono funzionalità che richiedono elaborazioni distribuite?

Le prestazioni sono critiche?

Il programma funzionerà in un ambiente operativo già pesantemente utilizzato?

Il sistema richiede funzionalità avanzate per l'immissione e la consultazione in linea dei dati?

Le funzionalità di immissione dei dati devono essere costruite tramite interfaccea finestre?

Gli archivi principali sono aggiornati in tempo reale?

Le informazioni scambiate tra utente e programma sono complesse?

Il codice del programma è complesso?

Il codice è scritto per essere riusabile?

Nel progetto sono include anche le attività di installazione e conversione?

Il programma è stato progettato per essere istallato presso diversi utenti?

Il programma è stato progettato per facilitare un facile uso e la possibilità di apportare modifiche da parte dell'utente?

L'International Function Point Users Group

- Produce un manuale con le istruzioni per il conteggio dei FP
- Missione: curare l'evoluzione della tecnica FP
- Scopo delle varie versioni:
  - applicabilità fuori dal contesto originale
  - consistenza delle indicazioni tra organizzazioni diverse
  - esempi, applicazioni con GUI
- http://www.ifpug.org/home/docs/ifpughome.html

Critiche ai function points

- Rispetto alla teoria rappresentazionale i FP non spiegano la correlazione tra la misura dei FP e la conoscenza empirica del concetto di funzionalità. Esempi:
  - 6 file logici di bassa complessità, 6 interfacce di media complessità e 7 input di alta complessità contribuiscono in egual misura alla "funzionalità" del sistema: perché?
- i FP ci dicono che una applicazione è 1,7 volte più complessa di un altra: come si spiega questo in termini concettuali? perché 1,7 e non 1,5 o 2?

Critiche ai Function Point

- La correzione dei FP grezzi appare irrazionale, essendo un tentativo di adattare la misura della funzionalità allo scopo di servire alla predizione dei costi di sviluppo.
  - Fenton: è come correggere la misura dell'altezza di una persona per poterla correlare alla misura dell'intelligenza.
  - La correzione non migliora la precisione delle stime, anzi appare che il semplice numero di input e output sia uno stimatore equivalente (se non migliore) dei Punti Funzione

Critiche ai Function Points

- Nel calcolo dei FP a partire dai Function Types si perde informazione. Si è verificato che utilizzando direttamente i Funtion Types come variabili indipendenti si ottiene un modello migliore
  - in pratica l'intervallo di variabilità delle stime diventa la metà rispetto al modello originale di Albrecht

Critiche ai Function Points

- Il conteggio dei punti funzione avviene nelle fasi iniziali del progetto, sulla base di documenti di specifica o di progetto, spesso del tutto informali, scritti in linguaggio naturale, con al più qualche DFD.
- Quindi il conteggio avviene su basi piuttosto fragili, con un ampio margine di discrezionalità
  - Le specifiche vengono "interpretate" dall'esperto di conteggio

Critiche ai Function Points

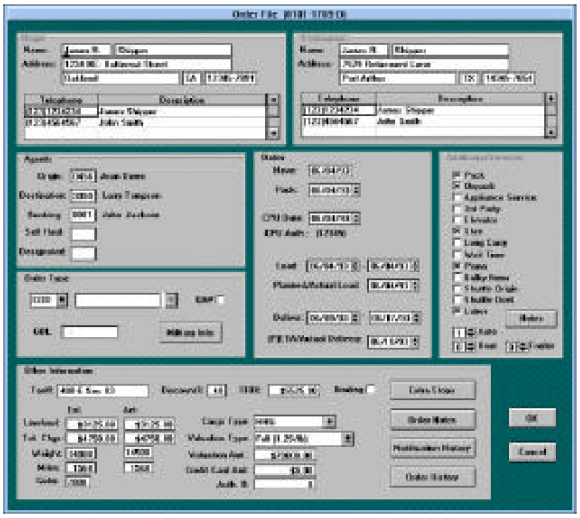
- Le modalità di conteggio dovrebbero essere invece espresse in modo preciso, basandosi su documenti formali
  - oggi il conteggio è "un'arte", di cui esistono esemplificazioni, ma nessuna definizione formale
  - i tool danno una mano, ma non automatizzano il processo
- Esperimenti hanno mostrato un ampia variabilità di risultati per conteggi effettuati da esperti sullo stesso documento
  - fino al 500% (anche per progetti piccoli)
  - mediamente del 30% per analisti di una stessa organizzazione

Critiche ai Function Points

- Il conteggio dipende dal contenuto delle specifiche
  - più sono dettagliate e più il numero di FP cresce
- Il conteggio è svolto con criteri ampiamente soggettivi.
- L'IFPUG ha anche lo scopo di rendere quanto più omogenei possibili i criteri di conteggio
  - cosa che appare comunque difficile, vista la mancanza di formalità nella procedura

Un esempio di calcolo di FP

- Consideriamo il problema di contare i FP di una tradizionale applicazione di gestione ordini, interattiva, scritta in Visual basic.



File interni ed esterni

- Identifichiamo i gruppi logici di informazioni (file) usati o mantenuti (modificati) dall'applicazione.
  - Ipotizziamo che non ci siano altri file che quelli rappresentati direttamente nella schermata vista.
- I dati sullo schermo derivano da due file logici
  - il file degli ordini - modificato dall'applicazione
  - il file degli agenti - solo consultato, per lettura e validazione
- Abbiamo un Internal Logical File (ILF): il file degli ordini e un External Interface File (EIF): il file degli agenti.

Complessità dei file: definizione

- La complessità di un file logico dipende da due fattori:
  - il numero di record element types (RETs)
  - il numero di data element types (DETs).
- Nel nostro caso ciascun file ha un solo RET
  - avremmo più RET se ad es. ci fossero diversi tipi di ordini caratterizzati da diverse informazioni.
- I DET sono essenzialmente i "campi" dati contenuti in ciascun "record"
  - le ripetizioni di istanze (es. più numeri di telefono) non contano
  - se l'applicazione utilizzasse un DB relazionale dovremmo contare gli attributi delle tabelle che logicamente compongono il "file"

Complessità dei file di input e output

- Il conteggio di RET e DET dà il seguente risultato:

File Name	File Type	RETs	DETs
Order File	ILF	1	70
Agent File	EIF	1	5

Tabella per il calcolo della complessità

- Il calcolo della complessità si basa sulla tabella seguente:

RETs	DETs		
	1-19	**20-50**	**50+**
1	Low	Low	Average
2-5	Low	Average	High
6+	Average	High	High

Complessità dei file di I/O

- L'applicazione della tabella ai nostri dati dà il seguente risultato:

File Name	File Type	RETs	DETs	Complexity
Order File	ILF	1	70	Average
Agent File	EIF	1	5	Low

Input, output e interrogazioni

- Il passo successivo riguarda il conteggio del numero di input, output e interrogazioni associate alla schermata.
  - all'atto della presentazione della schermata si ha una interrogazione dei file per estrarre i dati da mostrare/editare
  - quando si preme OK si ha una transazione che aggiorna i dati del file degli ordini
- La complessità di una interrogazione/transazione si determina in base a
  - numero di logical file types referenced (FTRs)
  - numero di data element types (DETs).

Input, output e interrogazioni

- Nel nostro caso:
  - per l'interrogazione iniziale 2 FTRs, 73 DETs.
  - Per la transazione di aggiornamento: 1 FTR (ordini), 68 DETs + l'evento di controllo (tasto return o click su OK) = 69 DETs.

Altre interrogazioni/transazioni

- La cancellazione di ordini (se possibile) sarebbe una ulteriore transazione.
- I messaggi di errore associati ad un input sono conteggiati come output separati.
- Nel nostro caso si ha un messaggio d'errore se il codice dell'agente non è valido o se gli indirizzi sono lasciati vuoti.
  - Si conta un FTR corrispondente al file degli agenti, consultato per validazione.
  - Si conta un DET per ogni messaggio: totale tre (codice non valido, indirizzo destinazione indefinito, indirizzo origine indefinito).

Tabelle per il calcolo della complessità

- La complessità degli input si calcola in base alla seguente tabella:

	DETs		
FTRs	**1-4**	**5-15**	**16+**
0-1	Low	Low	Average
2	Low	Average	High
3+	Average	High	High

Tabelle per il calcolo della complessità

- La complessità degli output si calcola in base alla seguente tabella:

	DETs		
FTRs	**1-5**	**6-19**	**20+**
0-1	Low	Low	Average
2-3	Low	Average	High
4+	Average	High	High

Complessità di interrogazioni, input e output

- La complessità delle transazioni si calcola usando la tabella del più complesso tra input ed output (di solito è l'output).
- Nel nostro caso la valutazione finale di input, output e transazioni è:

Transaction	Type	FTRs	DETs	Complexity
Screen Display	Inquiry	2	73	High
Edit	Input	1	69	Average
Error Messages	Output	1	3	Low

Calcolo dei FP

- Il calcolo finale dei FP avviene inserendo i dati trovati nella apposita tabella per la "pesatura":

Element	Low	Average	High	Total
Input	0 x 3	1 x 4	0 x 6	4
Output	1 x 4	0 x 5	0 x 7	4
Inquiry	0 x 3	0 x 4	1 x 6	6
ILF	0 x 7	1 x 10	0 x 15	10
EIF	1 x 5	0 x 7	0 x 10	5
FP				**29**

- La nostra applicazione consta di 29 function points.

Metriche dimensionali

- Si basano su una misura diretta delle linee di codice del programma.
- Si usano diverse sigle che tuttavia fanno sostanzialmente riferimento allo stesso concetto:
  - LOC (lines of code) o SLOC (source LOC)
  - DSLOC (delivered SLOC)
  - DSI (delivered source instructions) o KDSI (migliaia (Kilo) DSI)

Metriche dimensionali: calcolo

- A seconda dell'utilizzo che si intende fare di queste metriche può aver senso escludere dal conteggio delle parti:
  - se voglio misurare quanto sviluppo escludo le istruzioni riusate da altri programmi senza modifiche
  - se voglio correlare le dimensioni con gli errori escludo le linee di commento (che non generano errori)
  - viceversa se sto valutando la manutenibilità del prodotto le linee di commento sono rilevanti
  - ...

Metriche derivate

- Produttività = LOC/mesi-uomo
- Qualità = Numero totale di errori/LOC
- Costo unitario = Costo totale/LOC
- Livello di documentazione = Pagine di documentazione/LOC

Valutazione: Metriche Dimensionali

- Pro
  - Sono facilmente definibili.
  - È facile misurare il valore di LOC per un qualsiasi programma.
  - Sono basate su concetti molto diffusi tra gli addetti del settore.
  - Sono utilizzate in molti metodi esistenti per la stima dei costi e per la misura della produttività.
  - Quasi tutti i dati storici raccolti in passato sono basati su questo tipo di metriche.

Valutazione: Metriche Dimensionali

- Contro
  - La dimensione espressa in linee di codice sorgente risulta fortemente dipendente da attitudine e capacità del programmatore
  - La dimensione ha un "peso" diverso per linguaggi di programmazione diversi.
  - LOC_{Ada} non confrontabili con LOC_{C}, ecc.
  - Occorre definire in modo preciso cos'è una linea di codice.
    - » Si possono usare strumenti di conteggio (usare sempre lo stesso tool garantisce consistenza).
  - Poiché gli indici di qualità vengono a dipendere dal numero di linee rilasciate, un tentativo di migliorare la produttività potrebbe portare chi sviluppa a scrivere più codice, senza una particolare attenzione alla qualità di ciò che si produce.

Valutazione: Metriche Funzionali

- Pro
  - Sostanzialmente indipendente dalle capacità del programmatore (fa riferimento alle specifiche) e dal linguaggio di programmazione.
- Contro
  - Si basano su una definizione di complessità molto semplificata.
  - Il conteggio è parzialmente soggettivo. Non esiste un criterio univoco per i conteggi, le assegnazioni dei pesi e la valutazione dei parametri correttivi.
    - » Persone diverse danno generalmente valutazioni diverse.
    - » IFPUG cerca di fornire un criterio omogeneo.

Relazione tra metriche dimensionali e funzionali

- Secondo Albrecht e Gaffney esiste una correlazione tra metriche funzionali e dimensionali
  - determinata in base all'esame di 24 progetti reali.
- Valutazioni più recenti (Caper Jones):

Linguaggio	LOC/FP
Macro Assembler	213
C	150
Fortran 77	105
ANSI COBOL 85	91
Ada	71
Prolog, Lisp	64
Pseudo-language	53
C++, altri linguaggi OO	27
DB query language	13
Spreadsheet	6
Linguaggi 5a gen, VB	4

Complessità

- La dimensione del software è la metrica statica più evidentemente utile per caratterizzare un sistema SW.
- Tuttavia non riesce da sola a catturare tutti gli aspetti del sistema: come si comporta un programma grosso ma semplice rispetto a uno piccolo ma complesso?

- Un'altra metrica estremamente importante è la *complessità* del codice.
- Solo valutando sia dimensioni che complessità si ottiene un "identikit" del software sufficientemente completo

Valutazione della Complessità

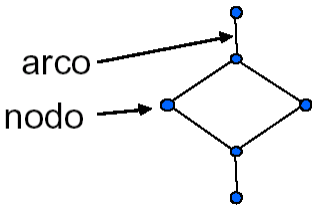
- Difficile. Opzioni:
  - valutazione soggettiva (ad esempio basata su interviste)
    - » importante avere un criterio uniforme di riferimento
  - valutazione oggettiva
    - » ci vuole un algoritmo di calcolo, basato sugli attributi oggettivi del programma
  - valutazione ibrida

Cyclomatic Branch

- Definita da Thomas McCabe nel 1976.
  - Rappresenta il numero di percorsi linearmenti indipendenti in un modulo.
- Terminologia:
  - Cyclomatic complexity,
  - complexity,
  - McCabe's complexity

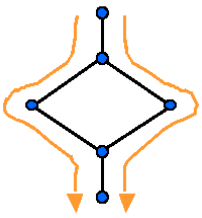
Cyclomatic Branch: calcolo

- Calcolato sul flowgraph del modulo
  - nodi (n) che rappresentano linee di codice
  - archi (e) rappresentano il flusso di controllo
- Esempio:
  - n = 6
  - e = 6
- La complessità ciclomatica è calcolata come v(G)=e-n+2.
- Per il grafo dell'esempio:
  - V(G) = 6-6+2 = 2



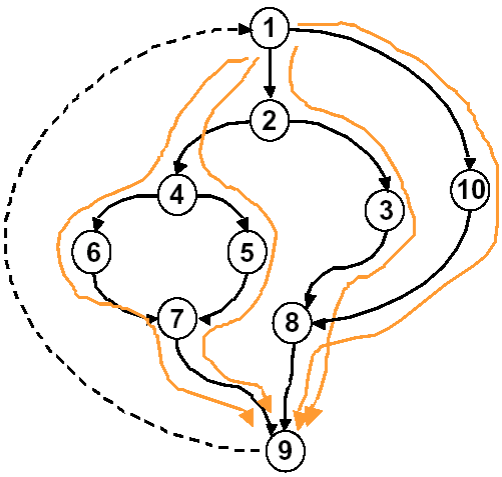
Cyclomatic Branch: significato

- Indica il numero di cammini indipendenti nel grafo (programma) dato.
- Quindi indica il limite superiore al numero di prove da effettuare per garantire che ogni arco sia eseguito almeno una volta.
- Un cammino è indipendente se introduce almeno un nuovo arco non incluso negli altri cammini.



Altro esempio di calcolo

- e = 12
- n = 10
- V(G) = 4



Utilizzo della complessità ciclomatica

- Un gran numero di programmi è stato misurato e analizzato.
- I risultati sono utilizzabili per stimare rischi, costi e stabilità dei programmi.
- Esiste correlazione tra cyclomatic complexity e frequenza degli errori.
- Complessità bassa indica anche buona manutenibilità e testabilità.

Cyclomatic complexity & test planning

- Le analisi hanno dimostrato che la cyclomatic complexity dà il numero esatto di test necessari per testare ogni punto di decisione di un programma per ogni risultato.
- Un modulo troppo complesso richiederà un numero di passi di test proibitivo.
  - In questo caso occorre suddividere il modulo in sotto-moduli meno complessi

Altri usi della complessità ciclomatica

- Code development risk analysis.
  - Valutazione del rischio durante la costruzione.
- Change risk analysis in manutenzione.
  - La complessità tende a crescere durante la manutenzione. Valutarla prima e dopo ogni intervento aiuta a tenere sotto controllo il rischio relativo.
- Reengineering.
  - Il rischio connesso con la reingegnerizzazione di un pezzo di codice è proporzionale alla complessità.

Valori di soglia

Cyclomatic Complexity	Valutazione del rischio
1-10	programma semplice, rischio basso
11-20	più complesso, rischio moderato
21-50	complesso, alto rischio
> 50	programma non testabile, rischio estremamente alto

Critiche e avvertenze

- Rappresenta il numero di decisioni, più che la complessità vera e propria.
  - Manca il fondamento empirico della metrica.
- Comunque è comunemente accettato come indicatore di complessità.
  - Perché si correla bene con molti indicatori interessanti.
- In generale una complessità alta non è di per sé un problema: bisogna valutare anche la funzione del codice.

Comparison Cyclomatic

- Misura della complessità comparativa: è uguale alla somma delle istanze di confronto e di operatori logici (and, or, !=, = =, <=, >=, <, >) presenti nelle condizioni di test per ogni istruzione di test (if) e ciclo.
- Può essere usata per determinare il grado di complessità di percorsi di decisione.

Halstead Complexity Measures

- La misura della complessità di Halstead (1977) misura la complessità di un modulo direttamente dal codice sorgente (operandi e operatori), con enfasi sulla complessità computazionale.
- Usata spesso a supporto della manutenzione.
- Esistono opinioni molto diverse sulla sua validità.
- Sicuramente utile per valutare la qualità del codice in applicazioni dove il calcolo è predominante.

Basi del calcolo

- Le misure di Halstead sono basate su quattro numeri ricavati direttamente dal codice sorgente:
  - n1 = il numero di operatori distinti
  - n2 = il numero di operandi distinti
  - N1 = il numero totale di operatori
  - N2 = il numero totale di operandi

Le misure di Halstead

- Program length (N): $N= N1 + N2$
- Program vocabulary (n): $n= n1 + n2$
- Volume (V): $V= N * (LOG_2 n)$
- Difficulty (D): $D= (n1/2) * (N2/n2)$
- Effort (E): $E= D * V$

Critiche

- Misurano la complessità lessicale e/o testuale piuttosto che strutturale (come fa invece la cyclomatic complexity).
  - Comunque danno buone indicazioni, soprattutto in manutenzione.
- La complessità del codice con un alto rapporto di calcolo rispetto a controllo è valutata più accuratamente dalle misure di Halstead che non da cyclomatic complexity.

Altre metriche di complessità

Complexity Measurement	Primary Measure of
Henry and Kafura metrics	Coupling between modules (parameters, global variables, calls)
Bowles metrics	Module and system complexity; coupling via parameters and global variables
Troy and Zweben metrics	Modularity or coupling; complexity of structure (maximum depth of structure chart); calls-to and called-by
Ligier metrics	Modularity of the structure chart

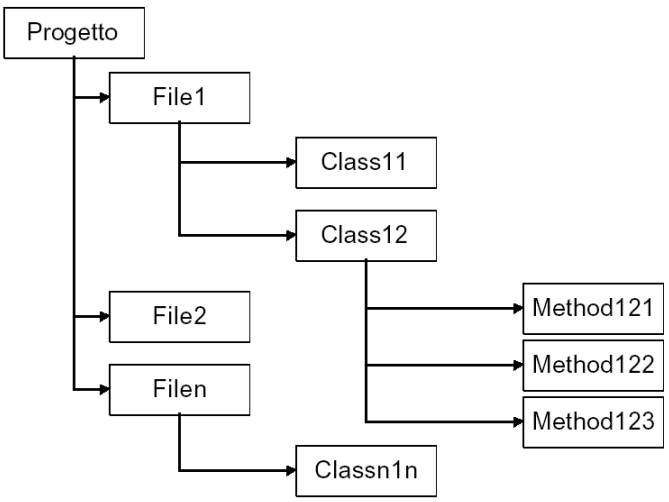
Altre metriche di complessità

- Per una descrizione e un confronto di tutte le metriche di complessità note in letteratura, si veda:
  - Marciniak, John J., ed. Encyclopedia of Software Engineering, 131-165. New York, NY: John Wiley & Sons, 1994.

Metriche per software object-oriented

- I sistemi software object-oriented sono strutturati in classi che contengono una parte pubblica e una parte privata.
- Ha quindi senso definire metriche relative alla strutturazione del sistema, e riportare le metriche classiche (LOC, complexity, ecc.) ai singoli elementi privati di ciascuna classe.

Struttura statica dei sistemi O.O.



Esempio di metriche statiche O.O.

- Metriche per Progetto:
  - ProjectTotalLines
  - ProjectLinesCode
  - FileCount
  - ProjectCommentLines
- Metriche per File:
  - FileTotalLines
  - FileLinesCode
  - ClassCount
  - FileCommentLines
- Metriche per Classe:
  - ClassTotalLines
  - MethodCount
- Metriche per Metodo:
  - MethodTotalLines
  - CyclomaticComplexity
  - InputParameter

Tipiche metriche object-oriented

- Depth of Inheritance (DIT)
- Number of Children (NOC)
- No. Attributes
- No. Operations:
  - Predefinite,
  - Ereditate,
  - Dichiarate localmente.

Tipiche metriche object-oriented

- Class Coupling – uno dei modi pi diffusi per misurarlo è il Data Abstraction Coupling (DAC), cioè il numero di tipi cui si fa riferimento da una classe (esclusi i tipi predefiniti)
- Class Cohesion – misura le connessioni entro una classe. LCOM (lack of cohesion of methods) [Chidamber & Kemerer] misura la relazione tra i metodi e gli attributi di una classe. Dà un valore tra 0 (massima coesione) e 1.

$$LCOM = \frac{\left(\frac{1}{a}\sum_{j=1}^{a}\mu(A_j)\right) - m}{1-m}$$

$\{\mu_i\}$ (i=1,…,m) è l'insieme di metodi che accedono ad un insieme di attributi $\{A_j\}$(j=1, …, a).

Strumenti di misurazione commerciali

- Cantata
- CMT++
- Codecheck
- Ensemble
- ENVY/QA
- Hindsight
- Krakatau
- LDRA's Testbed
- Logiscope
- Metamata Metrics
- Metrics ONE
- Metrics4[FORTRAN | Pascal | Project]
- Object Detail
- OOMetric/Developer
- Panorama
- PCA-RCM
- PC-Metric
- QA Family
- QualGen
- Resource Standard Metrics
- Software Metrics
- Testworks METRIC
- Total Metric for Java
- VISION:Assess
- Zinnote

Esempio (RSM per C)

Function: main Function Parameter 2 Function Returns 6 Interface Complexity 8 Cyclomatic Branch 7 Comparison Oper. 3 Logic Flow Complex. 10 LOC 48 eLOC 46 Comments 16 Blanks 14 Lines 63

Function: adddef Function Parameter 3 Function Returns 10 Interface Complexity 13 Cyclomatic Branch 14 Comparison Oper. 13 Logic Flow Complex. 27 LOC 100 eLOC 99 Comments 50 Blanks 28 Lines 176

Function: addscan Function Parameter 1 Function Returns 3 Interface Complexity 4 Cyclomatic Branch 17 Comparison Oper. 14 Logic Flow Complex. 31 LOC 103 eLOC 96 Comments 61 Blanks 47 Lines 207

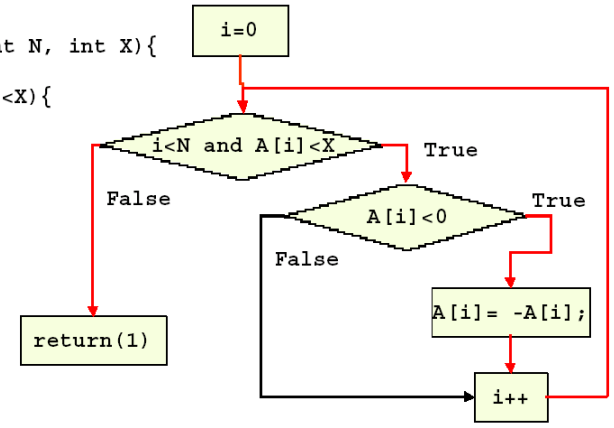
Function: adremdef Function Parameter 3 Function Returns 6 Interface Complexity 9 Cyclomatic Branch 7 Comparison Oper. 6 Logic Flow Complex. 13 LOC 35 eLOC 34 Comments 16 Blanks 11 Lines 60

Metriche dinamiche
Copertura strutturale

- criteri di (in)adeguatezza
  - se parti significative del programma non sono testate, il test è sicuramente inadeguato
- criteri di copertura del flusso di controllo
  - copertura delle istruzioni
  - copertura delle decisioni
  - copertura delle condizioni
  - copertura dei cammini
  - copertura del flusso dei dati
- tentativo di compromesso tra impossibile ed inadeguato

Copertura delle istruzioni

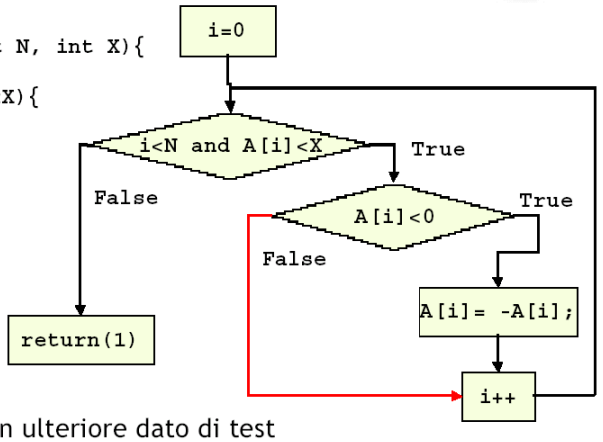
```
int select(int A[], int N, int X){
  int i=0;
  while (i<N and A[i] <X){
     if (A[i]<0)
        A[i] = - A[i];
     i++;
  }
  return(1);
}
```



Un solo dato di test (N=1, A[0]=-7, X=9) è sufficiente per garantire la copertura delle istruzioni ...
Ma errori nel trattare valori positivi di A[i] non sono rilevabili

Copertura delle decisioni

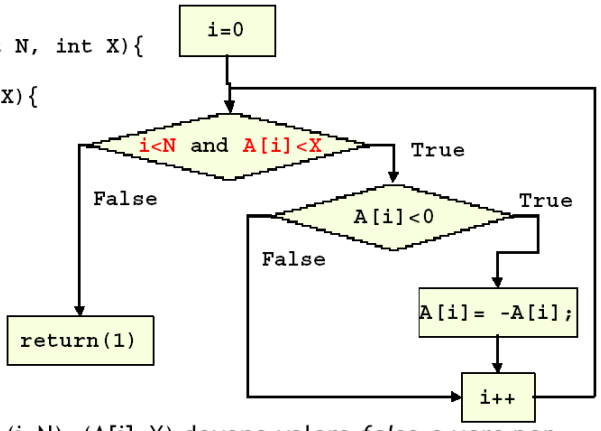
```
int select(int A[], int N, int X){
  int i=0;
  while (i<N and A[i] <X){
     if (A[i]<0)
        A[i] = - A[i];
     i++;
  }
  return(1);
}
```



Dobbiamo aggiungere un ulteriore dato di test (N=1, A[0]=7, X=9) per coprire il ramo *falso* del comando if.
Errori nel trattare valori positivi di A[i] sarebbero rilevabili. Errori nell'uscita dal ciclo con condizione A[i]<X non sarebbero rilevati

Copertura delle condizioni base

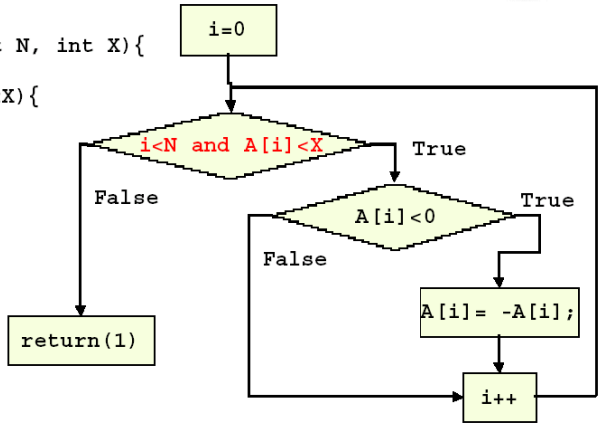
```
int select(int A[], int N, int X){
  int i=0;
  while (i<N and A[i] <X){
     if (A[i]<0)
        A[i] = - A[i];
     i++;
  }
  return(1);
}
```



Entrambe le condizioni (i<N), (A[i]<X) devono valere *falso* e *vero* per test differenti.
In questo casi dobbiamo aggiungere casi di test che fanno terminare il ciclo con la condizione not(A[i] < X).

Copertura delle condizioni composte

```
int select(int A[], int N, int X){
  int i=0;
  while (i<N and A[i] <X){
     if (A[i]<0)
        A[i] = - A[i];
     i++;
  }
  return(1);
}
```



Test di tutte le combinazioni di valori di verità delle condizioni.
Potrebbe non essere fattibile.

Copertura delle condizioni modificate

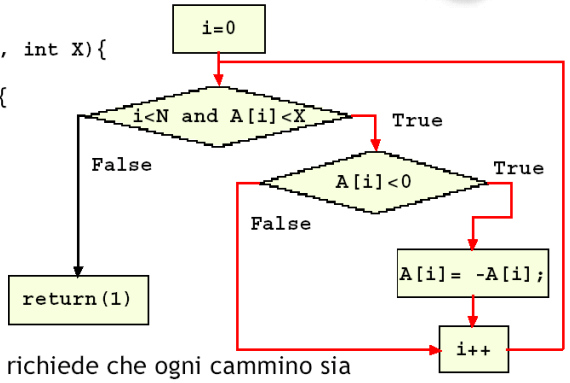
- Il valore di ogni condizione base deve determinare il valore di verità della decisione

```
if (A or (B and C))
 then do_something
 else do_someth_else
end if;
```

#	A	B	C	Out
1	false	false	true	false
2	true	false	true	true
3	false	true	true	true
4	false	true	false	false

Copertura dei cammini

```
int select(int A[], int N, int X){
  int i=0;
  while (i<N and A[i] <X){
     if (A[i]<0)
        A[i] = - A[i];
     i++;
  }
  return(1);
}
```

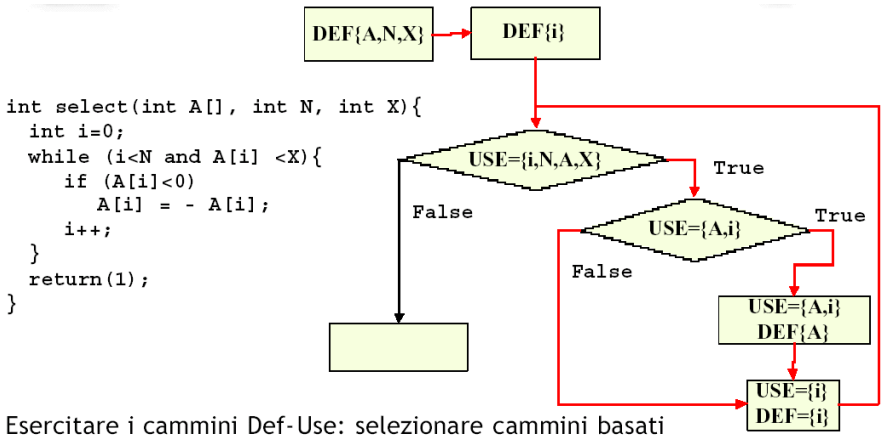


La copertura dei cammini richiede che ogni cammino sia eseguito almeno una volta, rivelando così i problemi che si verificano quando I cicli sono iterati diverse volte.
Problema: crescita incontrollata dei dati di test: il programma in figura ha infiniti cammini possibili. Dobbiamo selezionare un sottoinsieme ragionevole di casi, riducendo le iterazioni.

Copertura dei cammini interna e di confine (Boundary Interior Path Testing)

- cammini alternativi completi
- cammini alternativi attraverso cicli
- test di confine alternativi per i cicli:
  - confine - interno
  - ingresso da punti diversi
  - confine attraverso cammini differenti
  - interno attraverso cammini differenti

Copertura del flusso dei dati



Esercitare i cammini Def-Use: selezionare cammini basati sull'effetto delle variabili invece che sul numero di esecuzioni dei cicli. I criteri di copertura del flusso dei dati richiedono l'esecuzione di cammini che comprendano particolari configurazioni def/use.

Il problema della non eseguibilità

- Comportamenti sintatticamente validi (cammini, flussi di dati,...) possono essere non eseguibili
- I criteri di adeguatezza sono spesso impossibili da soddisfare.
- Approcci possibili:
  - giustificazione manuale dell'omissione di casi di test
  - accettazione di copertura parziale

Metriche e processo

- Come bilanciare budget, pianificazione, rischi e qualità
  - Necessità di
    - » ridurre budget
    - » garantire qualità
    - » minimizzare rischi
    - » anticipare costi (pianificare)
  - Non esistono regole generali
  - Si possono realizzare meccanismi per ambienti specifici

Prima fase: controllo della fase di test

- Raccolta dati storici
  - misure statiche su codice (LOC, numero ciclomatico, software science,…)
  - misure dinamiche su test effettuati (copertura codice e specifiche)
  - dati statistici su errori:
    - » localizzazione (modulo, blocco)
    - » tipo (violazione di memoria, errore passaggio parametri,…)
    - » gravità (catastrofico, …, cosmetico))
- Elaborazione statistica dei dati
  - costruzione modelli

Modelli statistici

- Modelli:
  - relazione statistica tra metriche statiche e
    - » presenza errori (per classi di errori)
    - » numero errori (per classi di errori)
  - relazione statistica tra metriche statiche, metriche dinamiche e
    - » presenza errori (per classi di errori)
    - » numero errori (per classi di errori)
- Possibile predire distribuzione errori per modulo

Esempio di Modello

- Insieme di metriche "sensibili" rispetto a presenza di errori
- Exits = numero di uscite
- Intervals = numero di blocchi
- C-head = numero di commenti in testa alle funzioni
- Lines = numero di linee di codice

Uso di modelli statistici per controllo budget

- Modelli statistici che fanno riferimento a metriche statiche permettono di identificare moduli particolarmente "pericolosi":
  - può essere più conveniente scartare il modulo e richiedere nuova produzione
  - può essere necessario ridisegnare il budget (o un diverso livello di qualità richiesto) per la fase di test prima dell'inizio del testing => possibilità di anticipare interventi e quindi ottenere risultati migliori

Uso di modelli statistici per pianificazione

- Piano di test standard per tutti i moduli
- Piano di test eccezionale per moduli "pericolosi"
- Pianificazione test in funzione di
  - rischio del modulo
  - qualità richiesta
  - potenziale difettosità del modulo

Uso di modelli di test per controllo di rischi e qualità

- Modelli statistici che fanno riferimento a metriche statiche e dinamiche permettono di identificare moduli particolarmente a rischio dopo la fase di test == alta probabilità di difetti presenti dedotti da complessità del modulo e copertura ottenuta:
  - possibile identificare moduli che
    - » richiedono ulteriore test
    - » possono causare problemi in fase di istallazione e/o uso

Introduzione nel processo di test

- Aspetti tecnici
  - necessari strumenti di raccolta di metriche statiche e dinamiche
  - necessaria automazione di raccolta ed elaborazione
- Aspetti psicologici
  - controllare impatto su progettisti
    - » chiarire scopi e metriche utilizzate
  - controllare impatto su addetti al test
    - » i risultati hanno solo valenza statistica => tutti i componenti vanno testati

Meccanismo di messa a punto come condizione essenziale

- I modelli hanno validità statistica
- i modelli sono fortemente dipendenti dal programma analizzato
- la messa a punto del modello su un numero maggiore di dati permette di avere valori statistici più stabili
- la revisione del modello su nuovi dati permette di
  - adattarsi a nuovi tipi di software
  - mantenere consistenza con evoluzione del team di sviluppo (feedback)

Implementazione di un meccanismo di messa a punto

- Raccolta di un ampio insieme di metriche statiche e dinamiche
- raccolta delle informazioni necessarie sugli errori rilevati
  - tipo
  - frequenza
  - gravità
  - localizzazioni
- Costruzione periodica di nuovi modelli statistici e confronto con i modelli precedenti

Complementarità con tecniche di ispezione

- Anomalie del codice possono influenzare in modo scorretto il codice
  - esempi: distribuzione linee bianche, numero linee di commento,...
- Alcune caratteristiche del codice non si riflettono nelle metriche
  - esempi: aritmetica dei puntatori,...
- Tecniche di ispezione di codice possono rilevare ed eliminare anomalie fastidiose e rendere più precisi i risultati