

G. Casadei A.G.B Teolis

LA GESTIONE DEI PROGETTI INFORMATICI

3. Le metriche

3.1 MarkII *function point*

3.2 *Object point*

3.3 *Full Function point (COSMIC)*

3.3.0 Introduzione

3.3.1 Fase di mappatura.

3.3.2 Fase di misurazione.

3.4 Valutazione preliminare della mole

3.5 Modello di previsione del costo di un progetto

REVISIONE DEL 10/11/2003

3.Metriche.

Si ricorda che il processo di pianificazione del progetto è composto (almeno) da cinque fasi successive:

1. determinazione del WBS;
2. determinazione per ogni attività del lavoro e del tipo di risorse necessarie;
3. determinazione dei costi delle singole attività;
4. in dipendenza della disponibilità delle risorse, determinazione di d e t (stesura del diagramma di Gantt);
5. valutazione dei rischi.

La teoria per stimare le “quantità” connesse con la realizzazione di *software* si studia in Ingegneria del *software*.

In relazione delle *attività* di produzione di *software*, occorre determinare:

- la *mole*: dipende dai requisiti del committente; nei progetti *software* dipende dal volume e dalla complessità delle funzioni da implementare (come percepita dall'utente e dal programmatore, non dalla macchina che esegue): entro ampi limiti, è indipendente dalla tecnologia; si esprime in unità di misura dette *Function Point*;
- il *lavoro* dipende dalla mole e dalla tecnologia usata (essenzialmente il linguaggio); le unità di misura del lavoro, in generale, dipendono dalle cose che devono essere prodotte; nel caso di progetti *software* (nei quali la tecnologia si riduce essenzialmente ai linguaggi di programmazione) questa unità è la linea di codice;
- l'*impegno*, è il tempo che una risorsa di personale con capacità standard impiega a produrre un certo *lavoro*; si misura in settimane (o mesi) uomo.

In generale per ogni attività, l'*impegno* necessario si ottiene determinando *la mole*, convertendola in *lavoro* e dividendolo per la *produttività standard*; la *durata* dell'attività si calcola come l'*impegno* diviso il numero di risorse (di personale) che si possono assegnare a quella attività; per tenere conto della diversa capacità delle persone reali coinvolte, a ciascuna di esse viene assegnato un coefficiente compreso fra 0.8 e 1.2 (la persona *standard* vale 1).

Una unità di misura del lavoro è il LOC (*Lines Of Code*, talvolta detto anche SLOC, la S sta per *Surce* o per *Standard*) e il suo multiplo KLOC (1000 LOC). Di norma la misura del lavoro non è primitiva, cioè non si ottiene contando le linee di codice prodotto o da produrre, ma viene ottenuta moltiplicando la mole per coefficienti che dipendono dal linguaggio usato.

La mole, come detto, si misura in unità convenzionali dette comunemente *Function Point*. Questa metodologia è stata introdotta da Allan Albrecht intorno alla metà degli anni settanta. A metà degli anni ottanta si sono affermati due diversi indirizzi: il primo, portato avanti da un organismo chiamato IFPUG (*Internatonal Function Points User Group*), ha continuato a mettere a punto e ad aggiornare la metodologia per calcolare la mole con unità che si chiamano ancora semplicemente *Function Points* ed è diffusa nel nord America; la seconda, proposta da Charles Symon, in cui le unità si chiamano *MarkII Function Points*, ed è principalmente diffusa nel nord Europa. Come mostrato in seguito, il primo indirizzo ha dato origine a una standardizzazione ISO che ragionevolmente sarà quella più usata nei prossimi anni.

Il processo per determinare la mole, detto comunemente di FPA (*Function Points Analysis*) è essenzialmente suddiviso in due passi. Partendo dalle specifiche del *software*, vengono prima individuate delle astrazioni significative anche per l'utente di riferimento; da queste si calcola un numero razionale che misura la mole.

3.1 MarkII function point

In questa metodologia i concetti chiave per calcolare la mole connessa con un applicazione sono *applicazione, utente, transazione, dati elementari*.

- **Applicazione:** è un insieme di funzionalità percepite come correlate dall'utente; è caratterizzata da un "confine" che la separa dall'esterno. In generale è di rilevanti dimensioni.
- **Utente:** è una persona o un'altra applicazione *software* sul medesimo sistema oppure una applicazione su un altro sistema; interagisce con le applicazioni standone all'esterno rispetto al confine. Quando l'utente è un'altra applicazione può a sua volta contenere dei data base con i quali l'applicazione in esame può scambiare informazioni.
- **Transazione:** è una interazione, tra un utente e l'applicazione, da un punto di vista logico o funzionale; sono le più piccole azioni a senso compiuto per l'utente (persona o altra applicazione) che lasciano l'applicazione stessa in uno stato consistente.
- **Dati elementari:** sono ciascuno degli insiemi più piccoli di dati che sono significativi per l'utilizzatore (per esempio un indirizzo, una data ecc., senza scendere nel dettaglio delle componenti).

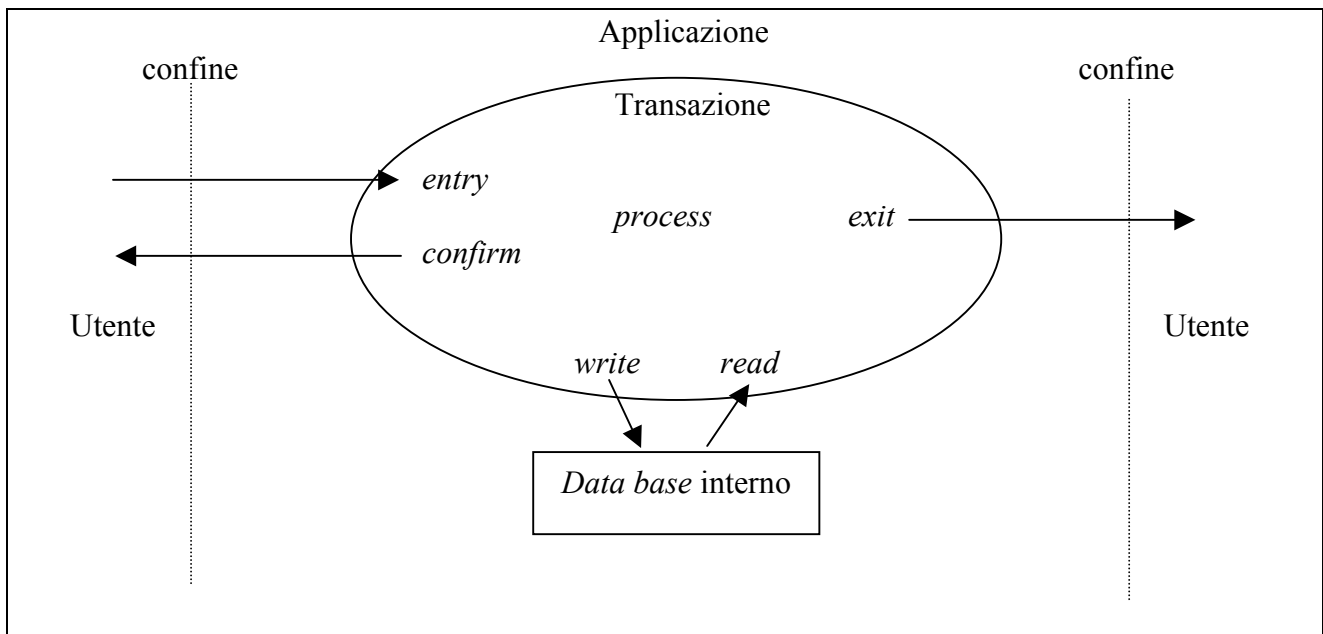


Figura 3.1

Uno schema concettuale di una transazione, all'interno di un'applicazione, è mostrato nella figura 3.1; schematicamente si può dire che una transazione compie sei tipi di azioni seguenti.

1. *Entry*: è l'azione per cui la transazione acquisisce dei dati dall'utente (persona o altra applicazione).
2. *Confirm*: è l'azione svolta dalla transazione che consiste nel dare all'utente informazioni sul proprio stato (per esempio "sto lavorando" oppure "ho completato il compito").
3. *Exit*: è l'azione con la quale la transazione fornisce dati all'utente (persona o altra applicazione).
4. *Read*: è l'acquisizione di dati direttamente da un data base di "proprietà" dell'applicazione.
5. *Write*: è la memorizzazione di dati direttamente su un data base di proprietà dell'applicazione.

6. *Process*: è l'insieme delle operazioni di calcolo che la transazione deve eseguire per portare a termine il compito.

Una applicazione è individuata da un insieme di transazioni e dall'insieme dei dati contenuti nel suo *data base* interno. Dal punto di vista formale, si può dire che una applicazione A è una coppia $A = (T, D)$, dove T è l'insieme delle transazioni e D è l'insieme dei dati elementari che compaiono nei data base di proprietà dell'applicazione; in particolare, posto $D = \{d_1, d_2, \dots, d_p\}$, $d_m = \{d_{m1}, d_{m2}, \dots, d_{m\mu}\}$ è l'insieme dei dati elementari di tipo m ; indicato con $T = \{t_1, t_2, \dots, t_\sigma\}$ insieme delle transazioni, ogni elemento t_i è formato da azioni e dai dati su cui esse agiscono, cioè $t_i = (a_i, b_i)$ con $a_i \subseteq \{\text{entry, confirm, exit, read, write, process}\}$ e $b_i: a_i \rightarrow \cup_p d_p$ è la funzione che associa ad ogni azione di t_i l'insieme di dati elementari su cui agisce.

Un conteggio di *function point* è una funzione che associa ad ogni applicazione di un certo contesto un numero razionale. Per essere "ragionevole" un conteggio deve essere

- *crescente*, con il numero e la complicazione delle transazioni,
- *monotono*, cioè se una applicazione ha più *function point* di un'altra, aggiungendo a entrambe una stessa componente, si mantiene la disuguaglianza.

Per precisare queste nozioni occorre introdurre due concetti: la *composizione* tra applicazioni e un *ordine parziale* tra esse.

Date due applicazioni $A = (T_A, D_A)$ e $B = (T_B, D_B)$, per la composizione la scelta naturale è porre

$$A \vee B = (T_A \cup T_B, D_A \cup D_B),$$

per l'ordine, invece, la scelta naturale è porre

$$A < B \Leftrightarrow T_A \subseteq T_B, D_A \subseteq D_B.$$

Il metodo MarkII consiste nel determinare un *Function Point index* o *Fpi* per una applicazione come somma dei *Function Point index* di ciascuna delle sue transazioni.

$$Fpi(A) = \sum_j Fpi(t_j)$$

L'indice *Fpi* per ogni transazione si calcola pesando opportunamente il numero dei dati elementari di *input*, il numero di entità accedute e il numero di dati elementari di *output*:

$$Fpi(t) = w_i \times (\text{numero di dati elementari di input}) + \\ w_e \times (\text{numero di entità accedute}) + \\ w_o \times (\text{numero di dati elementari di output}),$$

con $w_i = 0.58$, $w_e = 1.66$ e $w_o = 0.26$.

Il punto fondamentale, e anche il meno intuitivo, è che la complessità del *processing* di una transazione non contribuisce al calcolo del *function point index*. Altra caratteristica della formula è che per le azioni di *read* e *write* contano le entità accedute, mentre per le azioni di *entry* e *exit* contano i dati elementari. In particolare si fa osservare che quando si scambiano informazioni con un data base si contano le entità che compongono quelle informazioni pesate con un coefficiente "elevato"; quando per scambiare informazioni si attraversa un confine (per interagire con una persona o con altra applicazione), si contano i dati elementari (che sono più numerosi delle entità) pesati con un

coefficiente “basso”. È quindi molto rilevante per il calcolo di *Fpi* se un *data base* appartiene (cioè è interno) all’applicazione in esame oppure no.

I valori dei pesi w sono stati determinati in maniera statistica su parecchie centinaia di progetti per tener conto dell’apporto reciproco al lavoro di ciascuna delle tre componenti.

Esercizi

E3.1 Dimostrare che *Fpi* è crescente e discuterne la monotonicità.

Da osservazioni empiriche (statistiche) si dice che, in un progetto, per valutare l’indice *Fpi* delle applicazioni di cui esiste l’analisi si impiega da 0.2 a 0.5% del tempo impiegato per l’intero progetto.

Per procedere al calcolo dell’indice *Fpi* si eseguono i seguenti 5 passi.

Passo 1.

Occorre stabilire la finalità per cui viene calcolato l’indice.

Esempi sono:

1. stima del lavoro per pianificare in dettaglio un progetto che comprende lo sviluppo di *software*;
2. misura del “portafoglio applicativo” di un’azienda per calcolare il costo della manutenzione;
3. valutazione di una applicazione per determinarne (una base per) il prezzo di vendita;
4. misura di un “portafoglio applicativo” per valutarne il costo di rifacimento (per esempio perché di tecnologia obsoleta);
5. misura dell’*output* di un *team* di sviluppo per calcolarne la produttività.

Passo 2.

Dividere l’universo del discorso in applicazioni stabilendo i confini tra di esse.

Quando le applicazioni preesistono (e devono rimanere tali come al punto 2 degli esempi riportati nel passo1)), la loro individuazione non offre difficoltà. Completamente diverso è il caso di un progetto *software* nel quale si può scegliere in quante applicazioni distinte è opportuno suddividere il prodotto; infatti se il numero delle applicazioni distinte viene tenuto basso (al limite una sola) è bassa la sua valutazione in *function point* (perché sono minori le informazioni che attraversano i confini delle applicazioni), mentre può essere alto il lavoro organizzativo complessivo per gestire ciascuna di esse; se invece il numero delle applicazioni è elevato, è alta la valutazione complessiva in *function point* (perché sono maggiori le informazioni che attraversano i confini delle applicazioni), mentre ragionevolmente diminuisce lo sforzo organizzativo in quanto è stato applicato il principio del *divide et impera*.

La decisione finale sul numero di applicazioni da considerare deve tenere in considerazione anche quanto è stato deciso al passo 1.

Passo 3.

Elencare le transazioni di ogni applicazione.

Passo 4.

Identificare le entità (contenute nei *data base* dell’applicazione) e contare, per ogni transazione, quelle referenziate.

Elencate le entità, occorre distinguere tra entità *primarie* e entità *secondarie* o “di sistema”.

Di norma sono entità primarie quelle significative per il dominio dell’applicazione, come per esempio il “dipendente” in una applicazione per paghe e stipendi o il “conto” per una applicazione di sportello bancario; sono secondarie quelle che servono per amministrare il sistema, come per esempio “operatore” che serve per descrivere le autorizzazioni d’accesso. È possibile esplicitare formal-

mente alcuni criteri per operare questa distinzione: i principali sono contenuti nella tabella che segue.

CRITERI	ENTITÀ PRIMARIE	ENTITÀ SECONDARIE
numero di attributi	molti con valori che cambiano frequentemente	pochi con valori che cambiano raramente
numero di occorrenze	relativamente numerose con cambiamenti frequenti	relativamente poco numerose con cambiamenti poco frequenti
occasioni in cui cambiano i valori degli attributi	durante le “normali” operazioni	al di fuori delle “normali” operazioni
autorizzazioni per cambiare i valori	nessuna speciale autorizzazione	autorizzazioni molto “elevate” (per es. <i>system administrator</i>)

Ciascuna entità primaria deve essere disarticolata in tutte le sue sottoentità; con riferimento agli esempi sopra richiamati, l’entità “dipendente” deve ragionevolmente disarticolarsi nelle sottoentità “dipendente a tempo indeterminato” e “dipendente a tempo determinato” e, analogamente, l’entità “conto” si può articolare in “conto corrente”, “conto di risparmio”, “conto di deposito”, ecc.

Per fornire il risultato di questo passo, per ogni transazione si contano le sottoentità primarie referenziate; se la transazione fa riferimento anche a una (o più) entità secondarie, si aumenta il conteggio totale già fatto di 1.

Passo 5.

Identificare e contare per ogni transazione, i dati elementari di *input output*.

Per effettuare questa identificazione si procede con le seguenti regole.

- a) Se durante una transazione occorre fornire una lista di elementi, conta solo la complessità della struttura “elemento” e non il numero di elementi contenuti nella lista (questo numero può anche variare ad ogni interazione).
- b) I campi “composti” (come per esempio una data strutturata come gg/mm/aaaa) contano come 1, a meno che nella transazione non siano usati esplicitamente le singole componenti o sottocomponenti del campo.
- c) Ogni messaggio di errore conta come un solo dato elementare, a meno che non contenga dati applicativi, nel qual caso questi dati contano per quanti sono.
- d) Negli *array* (*record* ripetuti) contano le colonne e non le righe.
- e) I *menu* a scelta non contribuiscono al conteggio.
- f) Le funzioni di *scrolling* o *paging* non contribuiscono al conteggio.
- g) Gli *edit box*, i *dialog box* e i *radio button* contano per gli effettivi dati di *input output* scambiati.
- h) Le icone contano come un dato di *input output* se servono per scambiare informazioni, non contano se servono per la navigazione.
- i) I *sensitive touch screen* si trattano come le icone.

Esempio. Registrazione di pagamenti

Supponiamo che in una data azienda siano state emesse delle fatture; dalle banche arrivano, per via cartacea, le segnalazioni di pagamento e queste devono essere registrate nel sistema contabile dell’azienda. Per realizzare questa registrazione viene costruita la transazione REG-PAG. Si considerano due possibili implementazioni.

Nella prima si acquisiscono i dati della fattura dall’operatore, si aggiorna nel *data base* la data del pagamento e si emette una segnalazione di errore se la fattura non esiste nel *data base*. In questo caso, è facile verificare che le informazioni coinvolte sono quelle descritte nella tabella seguente.

Dati elementari di <i>input</i>	Entità coinvolte	Dati elementari di <i>output</i>
<ul style="list-style-type: none"> • numero fattura • data di pagamento (gg/mm/aaaa) • importo 	Fattura <ul style="list-style-type: none"> • data emissione • numero fattura • importo • data pagamento 	Segnalazione <ul style="list-style-type: none"> • OK • errore per mancanza di fattura

Si può concludere che sono coinvolti tre dati elementari di input, una entità e un dato elementare di output e quindi il *Function Point index* vale $0.58 \times 3 + 1.66 + 0.26 = 3.66$.

Nella seconda implementazione si decide di procedere come nella prima con l'unica variante che in *output* si vogliono due tipi di segnalazione: la prima per segnalare l'eventuale ritardo di pagamento e la seconda per segnalare la fattura mancante o di importo del pagamento minore.

In questo caso, è facile verificare che le informazioni coinvolte sono quelle descritte nella tabella seguente.

Dati elementari di <i>input</i>	Entità coinvolte	Dati elementari di <i>output</i>
<ul style="list-style-type: none"> • numero fattura • data di pagamento (gg/mm/aaaa) • importo 	Fattura <ul style="list-style-type: none"> • data emissione • numero fattura • importo • data pagamento Data di sistema <ul style="list-style-type: none"> • data (gg/mm/aaaa) 	Segnalazione 1 <ul style="list-style-type: none"> • OK • errore per mancanza di fattura • errore per importo minore Segnalazione 2 <ul style="list-style-type: none"> • giorni di ritardo

In questo caso, i dati elementari di *input* sono 5 perché la data di pagamento deve essere scomposta per calcolare i giorni di ritardo; le entità sono 2 perché esiste una entità secondaria (di sistema) e i dati elementari di *output* sono 3, perché esiste una segnalazione di ritardo (che contiene un solo campo variabile che conta per uno) e l'altra segnalazione vale 1 perché contiene tre campi fissi (che valgono 1). In questo secondo caso, il *Function Point index* vale 6.74.

Come sarà messo in luce successivamente, la produttività di un programmatore in *Function Point* di tipo Mark II dipende in modo essenziale dall'ambiente di sviluppo. Per fare un esempio, ad oggi ¹ è ancora molto usato l'ambiente di sviluppo COBOL/CICS/DB2. In tale ambiente, esempi di produttività media, relativi ad una tipica azienda del settore, sono mostrati nella seguente tabella; i dati si riferiscono allo sviluppo, allo *unit test*, alla quota parte di *system test* e messa in gestione (in progetti svolti con il modello incrementale o durante la manutenzione evolutiva).

Attività	Produttività in ambiente COBOL/CICS/DB2
Sviluppo	1.0 <i>Fpi</i> a persona al giorno
Manutenzione adattativa e correttiva	2.1 <i>Fpi</i> a persona al giorno
Manutenzione evolutiva e migliorativa	1.3 <i>Fpi</i> a persona al giorno

Questi valori medi sono significativi perché la tecnologia coinvolta è "matura" e quindi diffusa, il che comporta che la deviazione standard sia "piccola".

¹ Nel 2002 in alcune applicazioni bancarie e finanziarie e in quelle gestionali di grandi Enti Pubblici e grandi Aziende.

3.2 Object point

Questa metodologia, sviluppata a cavallo tra gli anni ottanta e novanta, si usa di solito con uno strumento di sviluppo integrato detto ICASE (per *Integrated Computer Aided Software Environment*) cioè con un sistema che contiene per esempio un editor, un compilatore/interprete del linguaggio, un ambiente che consente le esecuzioni di prova, che permetta di individuare e correggere facilmente errori, ecc; contrariamente a quanto il nome può fare immaginare, non ha niente a che fare con la programmazione ad oggetti; si usa tipicamente per valutare la mole delle applicazioni realizzate secondo lo schema *client/server*. I concetti chiave di questo metodo sono i tre tipi di oggetti seguenti: *schermate*, *report* e *moduli di logica applicativa*. Vengono prese in considerazione anche le tabelle come sono definite nei *data base* relazionali.

Gli oggetti vengono valutati come mostrato nella seguente tabella.

Tipo di oggetto	Punteggio per un oggetto semplice	Punteggio per un oggetto medio	Punteggio per un oggetto difficile
Schermata	1	2	3
Report	2	5	8
Modulo di logica	10		

I criteri per decidere se una schermata è semplice media o difficile dipendono dal numero di tabelle complessive, da come queste si dividono tra *client* e *server* e dal numero di viste. Questi criteri sono esplicitati nella seguente tabella.

Valutazione delle schermate			
Numero di viste	Meno di 4 tabelle di cui meno di 2 sul <i>server</i> e meno di 3 sul <i>client</i>	Da 4 a 7 tabelle di cui da 2 a 3 sul <i>server</i> da 2 a 5 sul <i>client</i>	Più di 7 tabelle di cui più di 3 sul <i>server</i> più di 5 sul <i>client</i>
Minore di 3	Semplice	Semplice	Medio
Da 3 a 7	Semplice	Medio	Difficile
Maggiore di 7	Medio	Difficile	Difficile

(Si noti la simmetria della tabella rispetto alla diagonale secondaria.)

I criteri per decidere se un *report* è semplice medio o difficile dipendono dal numero di tabelle complessive accedute, da come queste si dividono tra *client* e *server* e dal numero di sezioni del *report*. Questi criteri sono esplicitati nella seguente tabella.

Valutazione dei report			
Numero di sezioni	Meno di 4 tabelle di cui meno di 2 sul <i>server</i> e meno di 3 sul <i>client</i>	Da 4 a 7 tabelle di cui da 2 a 3 sul <i>server</i> da 2 a 5 sul <i>client</i>	Più di 7 tabelle di cui più di 3 sul <i>server</i> più di 5 sul <i>client</i>
Minore di 2	Semplice	Semplice	Medio
Da 2 a 3	Semplice	Medio	Difficile
Maggiore di 3	Medio	Difficile	Difficile

Gli *object point* si calcolano sommando su tutti gli oggetti dell'applicazione (schermate report e moduli) il risultato della valutazione di ciascun oggetto ottenuta applicando i criteri illustrati.

In questo caso (cioè le applicazioni in cui si usa questo metodo di valutazione), la produttività è molto differenziata, come illustrato dalla seguente tabella.

Esperienza con lo strumento di ICASE	Molto bassa	Bassa	Nominale	Alta	Molto alta
Produttività in <i>object point</i> a persona al mese	4	7	13	25	50

Questa tabella mette in evidenza la forte dipendenza della produttività dalla conoscenza dello strumento di sviluppo e quindi sottolinea come la riuscita del progetto (il rispetto dei tempi e dei costi pianificati, oltre alla qualità del prodotto realizzato) sia criticamente legata alla scelta degli strumenti, da farsi in funzione della competenza di chi li deve usare, piuttosto che della loro validità in assoluto.

3.3 Full Function point (COSMIC)

3.3.0 Introduzione

Come è stato precedentemente accennato, la misura della mole del *software* o FSM (*Functional Size Measurement*) ha una storia relativamente tormentata. Iniziato negli anni ottanta del secolo scorso, ha avuto un picco di diffusione intorno al 90/91 per avere una rapida caduta alla fine del millennio; la curva di diffusione/accettazione dei metodi di misura della mole è mostrata in figura 3.2

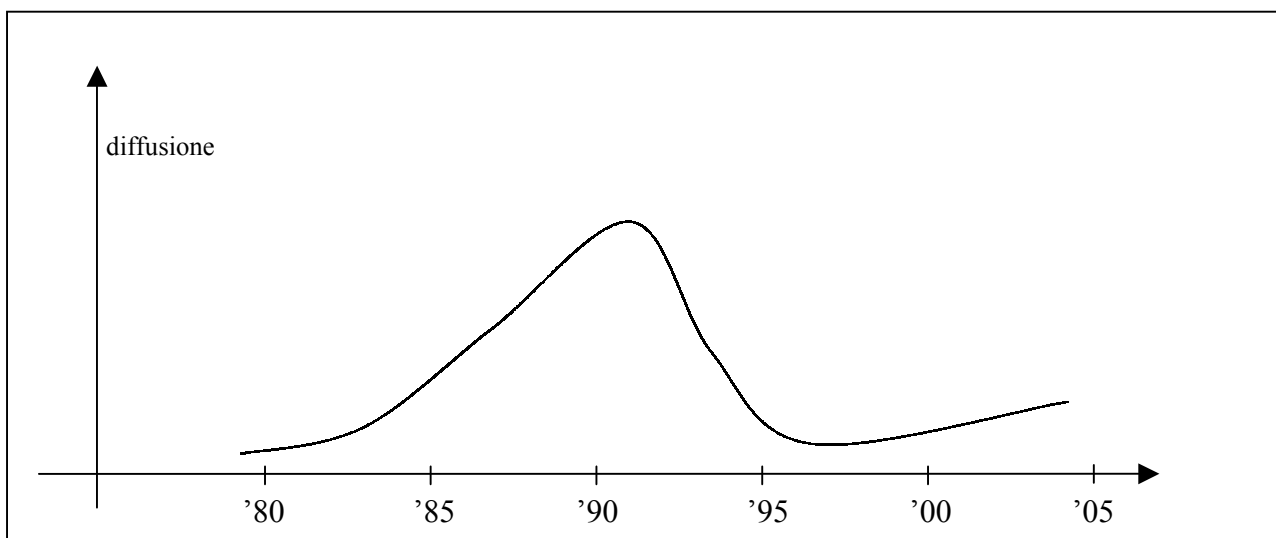


Figura 3.2

La ragione di caduta dell'interesse è stata duplice: da una parte la difficoltà della misura e l'inattendibilità dei risultati (cioè la non riproducibilità: persone diverse davano misure diverse per lo stesso *software*) e dall'altra il diffondersi di *software* per sistemi in tempo reale che non si prestava all'applicazione dei metodi che avevano come obiettivo la misura di *software* tradizionale (*transaction oriented*). Parallelamente si è venuto sviluppando, per le grandi applicazioni di telefonia mobile, il metodo FFP (*Full Functional Points*) adatto anche alle applicazioni in tempo reale.

Sempre verso la fine del millennio, il COSMIC (*Common Software measurement international Consortium*) affinava il metodo precedente che nel 2000 era presentato come proposta standard che veniva accettata nel dicembre 2002 con la sigla "ISO/IEC 19761- A Functional Size Measurement Method". Dall'inizio degli anni duemila la comparsa e la diffusione di questo metodo ha contribuito a un nuovo aumento della pratica di misurazione della mole funzionale del *software*.

Questo metodo, che d'ora in poi viene indicato con la sigla COSMIC FFP, ha i seguenti ambiti di applicabilità.

- Le applicazioni data intensive, come tipicamente sono i sistemi informativi aziendali.
- Le applicazioni *real time*, come il *message switching*, i sistemi *embedded* e quelli per il controllo di processi.
- Ibridi dei due tipi precedenti, come per esempio i sistemi operativi, i DBMS, i grandi sistemi di prenotazione.

Rimangono escluse dall'uso di questo metodo di misurazione ancora vaste aree di applicazioni come per esempio

- gli algoritmi complessi (metodi di simulazione, risoluzione di equazioni differenziali, ecc);
- i sistemi esperti;
- il trattamento di variabili continue connesse con fenomeni fisici (suoni, immagini, animazioni), come per esempio nel *software* multimediale di base.

Il metodo parte dai “requisiti funzionali dell'utente” o FUR (*Functional User Requirements*) e porta alla misura della mole funzionale escludendo le specifiche tecniche e le specifiche di qualità (in un certo senso viene misurato il contenuto semantico del *software*, indipendentemente dalla sua “forma”). Esistono ovviamente due momenti di applicazione del metodo:

- il caso *ex ante* (*pre-implementation*) nel quale il FUR viene dedotto dai documenti sui requisiti, dai documenti sul modello dei dati e dai documenti di analisi funzionale;
- il caso *ex post* (*post-implementation*) in cui il FUR viene dedotto dai programmi sviluppati, dai manuali di uso e gestione e dalla documentazione del *software* di base usato per l'esercizio.

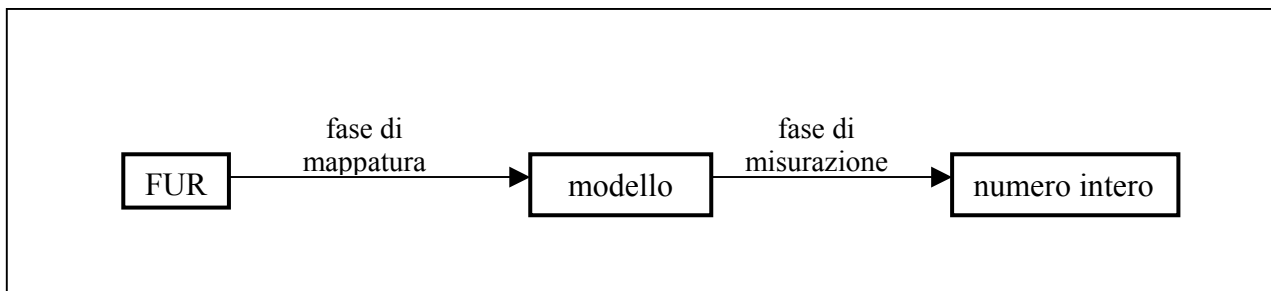


Figura 3.3

Come mostrato in figura 3.3, la valutazione avviene in due fasi: la prima, detta di mappatura, parte da un certo insieme di FUR e costruisce un modello; la seconda fase, detta di misurazione, costruisce un numero intero che rappresenta la valutazione.

La fase di mappatura, a sua volta, avviene in due tempi:

- l'individuazione del “modello di contesto”;
- l'individuazione del modello generico.

Per descrivere la sottofase in cui viene messo a punto il modello di contesto, è importante la definizione di utente: in questo contesto, si dice utente una persona, un modulo *software*, un dispositivo *hardware* che interagisce con il *software* da misurare. L'obiettivo del modello di contesto è quello di identificare:

- i confini (*boundary*),
- i moduli,
- gli utenti.

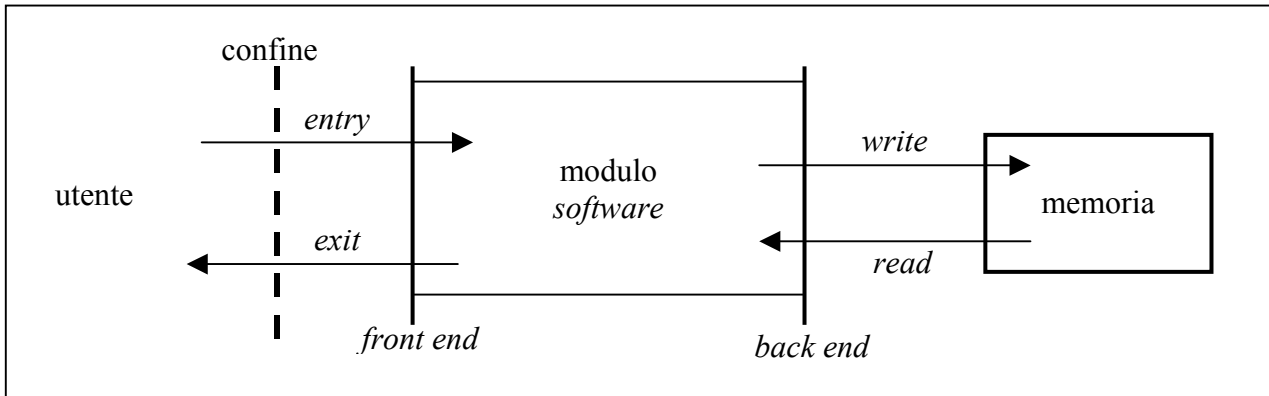


Figura 3.4

Lo schema elementare di questi tre componenti è mostrato in figura 3.4 che mostra come il confine separa l'utente dal modulo che ha per natura 4 interazioni, due per il *front end* verso l'utente e due per *back end* verso il sistema di memorizzazione. Le prime due interazioni varcano il confine e sono dette *entry* e *exit* a seconda del verso in cui transitano i dati. Analogamente le interazioni di *back end* si dicono *write* e *read*.

Naturalmente i confini, i moduli e gli utenti dipendono fortemente dal grado di dettaglio della descrizione; per esempio nella figura 3.5 il *software* da misurare è considerato visto dall'utente finale che si suppone essere una persona, mentre il sistema di memorizzazione viene ipotizzato essere un normale *hardisk*.

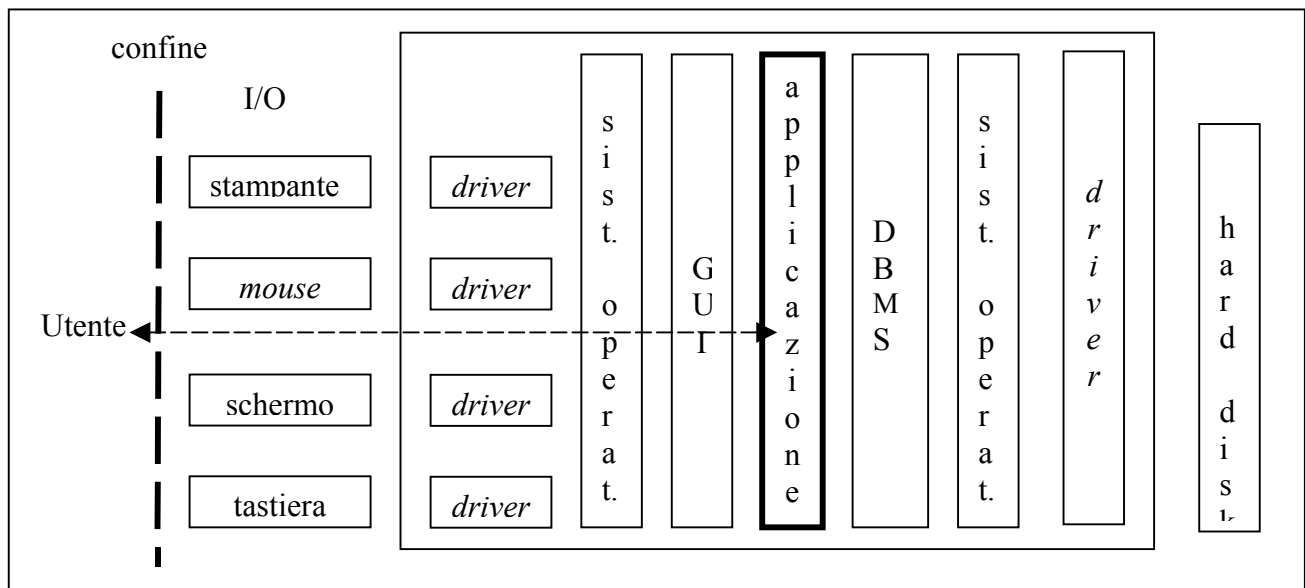


Figura 3.5

Nella figura 3.6 viene valutato lo stesso *software* come viene visto dallo sviluppatore; si vede immediatamente la differenza: tre moduli da sviluppare al posto di uno, non esiste utente persona e non esiste sistema di memorizzazione.

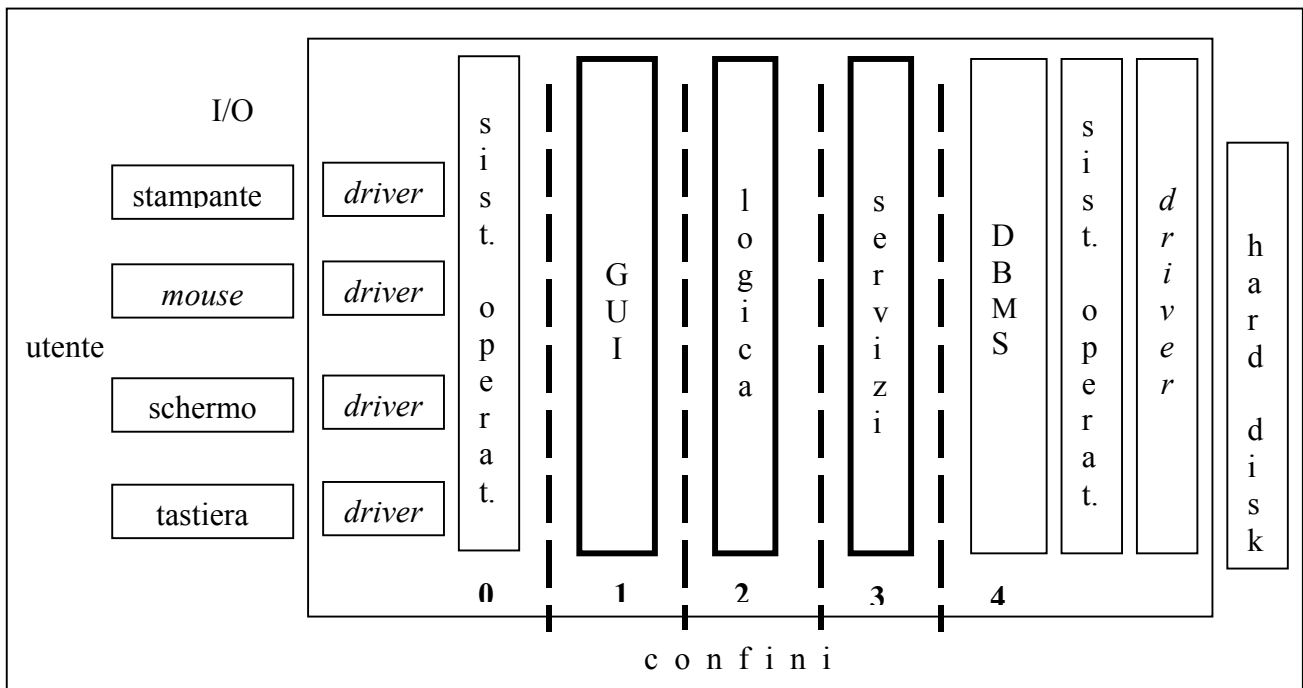


Figura 3.6

Il modulo 1 è utente di 0, il modulo 2 è utente di 1 e 3, il modulo 3 è utente di 4.

È immediato capire come nei due casi (diversi punti di vista) si possa calcolare, per il medesimo FUR, due differenti misure.

L'obiettivo del modello generico è quello di individuare i processi all'interno del modulo (che corrispondono grossolanamente alle transazione del metodo Mark II) e all'interno del processo individuare i sottoprocessi che movimentano o manipolano i dati. Come già osservato (si veda anche la figura 3.7) esistono 4 sottoprocessi di trasferimento dei dati (*entry*, *exit*, *read* e *wite*) e un solo processo generico di elaborazione.

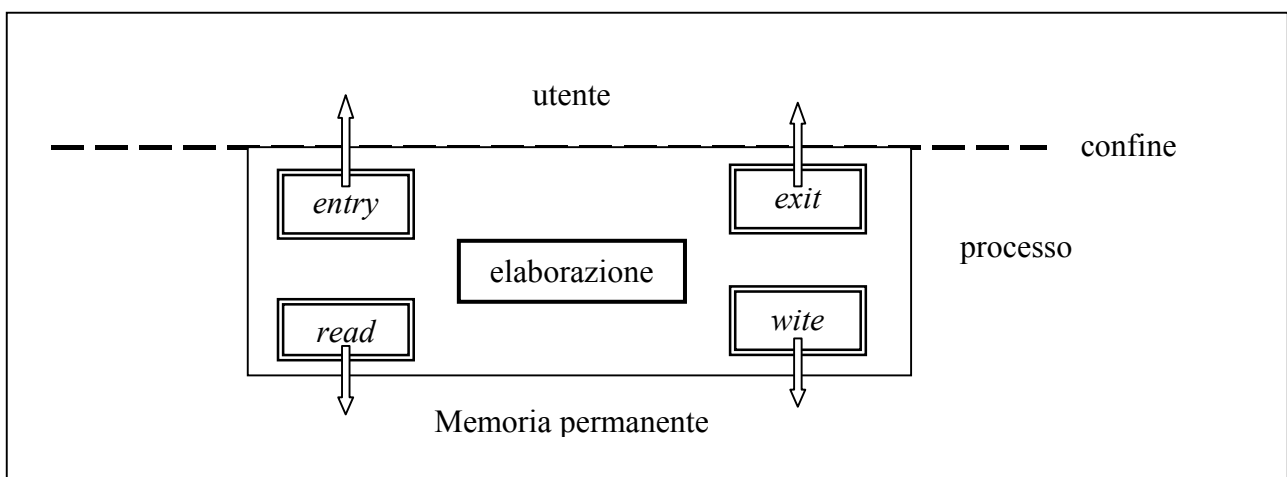


Figura 3.7

Nell'ambito di applicabilità del metodo, solo i 4 sottoprocessi di trasferimento dei dati contribuiscono alla mole funzionale; infatti, la fase di misura vera e propria è molto semplice: si conteggia un *cfsu* (COSMIC *Functional Size Unit*) per ogni istanza di processo di trasferimento di dati.

Si possono fare preliminarmente due osservazioni:

- due *cfsu* è il limite inferiore per ogni modulo *software*;
- per misurare, direttamente, un FUR, questo deve essere analizzato in termini di trasferimento di dati *entry, exit, read, write*; se questo non è stato fatto si può ancora valutarlo indirettamente da una sua scomposizione gerarchica (vedi nel seguito).

Prima di esaminare in dettaglio la procedura per attuare le due fasi (mappatura e misurazione), occorre stabilire tre caratteristiche che possono influenzare il risultato (cioè la misura):

- lo scopo (perché),
- l'estensione (cosa),
- il punto di vista (chi).

Non è paradossale che la misura vari con lo scopo: per esempio, la superficie abitabile di un appartamento è diversa a seconda che serva per determinare le tasse, per stipulare un contratto di pulizie, per progettare l'arredamento o per valutare il canone d'affitto. Esempi di scopo tipici sono (già visti per Mark II) i seguenti.

1. Prevedere il costo di un progetto di sviluppo.
2. Tenere sotto controllo i cambiamenti dei requisiti dell'utente durante l'attuarsi di un progetto.
3. Calcolare la produttività di un team di sviluppo (o di un metodo di sviluppo).
4. Confrontare la mole del *software* sviluppato con quella del *software* consegnato al committente.
5. Calcolare il costo di manutenzione.

La estensione delimita il FUR da misurare; esempi sono:

- quello descritto nel contratto,
- quello connesso con un oggetto riusabile,
- un "pacchetto *software*" da commercializzare,
- il patrimonio *software* di una azienda.

Il punto di vista è, allo stato attuale dello standard, solamente di due tipi (con il diffondersi del metodo, probabilmente ne compariranno altri):

- *end user* (tipicamente una persona o un gruppo di persone, ma anche sistemi *software* o *hardware*), prende in esame solo le funzioni del *software* (consegnato e nel suo complesso) necessarie a soddisfare un particolare FUR;
- sviluppatore, prende in esame tutte le funzioni di ciascuna parte del *software* che deve essere sviluppata per soddisfare un particolare FUR.

Si esaminano ora in dettaglio le fasi di mappatura e misurazione.

3.3.1 Fase di mappatura.

La rappresentazione schematica di questa fase è mostrata in figura 3.8.

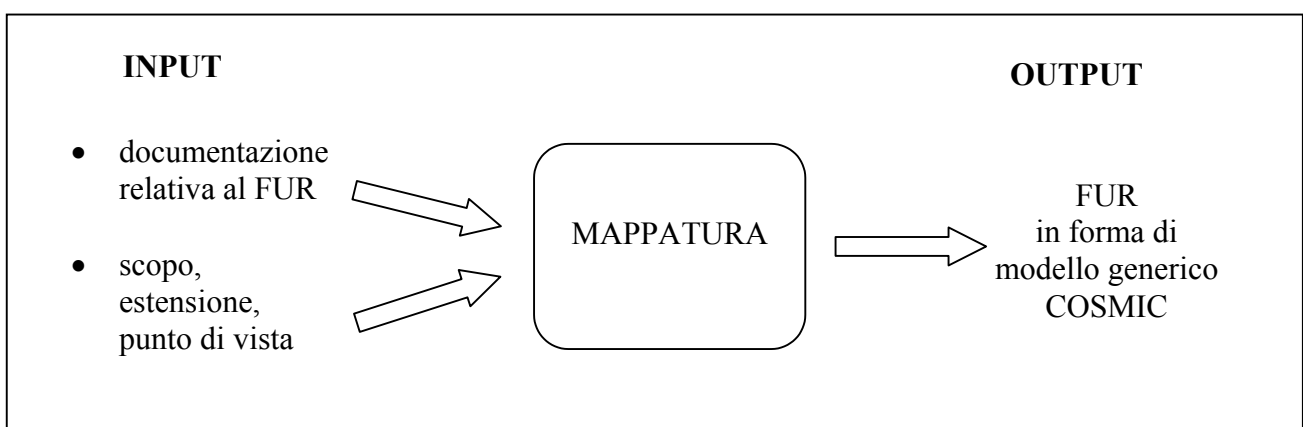


Figura 3.8

La fase si compone dei seguenti 5 passi (di cui l'ultimo è opzionale).

1. Identificare i livelli (*layer*).
2. Identificare i confini.
3. Identificare i processi funzionali.
4. Identificare i *data group*.
5. Identificare gli attributi.

Primo passo: i livelli. Un livello è il singolo elemento risultato di una partizione dell'ambiente software che include processi funzionali allo stesso livello di astrazione.

L'identificazione dei livelli è un processo iterativo che può continuare, se necessario, anche nei passi successivi. Per chiarire il concetto di livello occorre osservare che livelli distinti, naturalmente, possono condividere gli stessi dati; ma il *software* di un livello (di astrazione) interpreta gli attributi in maniera differente dal *software* di un altro livello.

Con riferimento alla figura 3.9, si considerino 4 livelli, ciascuno dei quali contiene una componente *software*: i *driver*, il sistema operativo, la GUI, l'applicazione. Gli stessi dati sono visti come una *bit stream* (eventualmente con caratteri di controllo) al primo livello; come *buffer* al secondo livello; come campi posizionati sullo schermo al terzo livello; come attributi al quarto. I livelli possono essere ordinati in maniera gerarchica, con quello subordinato che offre funzioni e servizi a quello superiore (e non viceversa).



Figura 3.9

Il *software* di un livello subordinato può funzionare (bene) anche se non funziona (bene) quello di un livello superiore, ma non viceversa. Usualmente DBMS, sistemi operativi, *driver* sono considerati livelli differenti, tuttavia la distinzione dipende dai contesti; il *software* applicativo occupa, in genere, il livello o i livelli più alti.

Dal punto di vista concettuale, si può dire che il principio fondamentale per partizionare il *software* è la *information hiding*, che permette di individuare i moduli; i moduli appartengono a diversi livelli se usano differenti astrazione sui dati che scambiano con l'esterno; se usano la medesima astrazione sono allo stesso livello (*peer-to-peer*).

Secondo passo: i confini. Un confine, come visto, è il limite concettuale tra un modulo *software* e i suoi utenti. La identificazione dei confini, come quella dei livelli, è un processo iterativo che può protrarsi per tutta la fase di mappatura.

Naturalmente può esistere un confine tra un livello e uno che gli è subordinato; ci può ovviamente essere un confine che separa una coppia di moduli dello stesso livello. Spesso è utile identificare gli eventi scatenanti (*triggering events*); in questo caso il confine separa (il processo che genera) l'evento e il *software* che incorpora il processo avviato da quell'evento.

Un primo esempio è riportato in figura 3.10 che descrive il punto di vista dello *end user*.

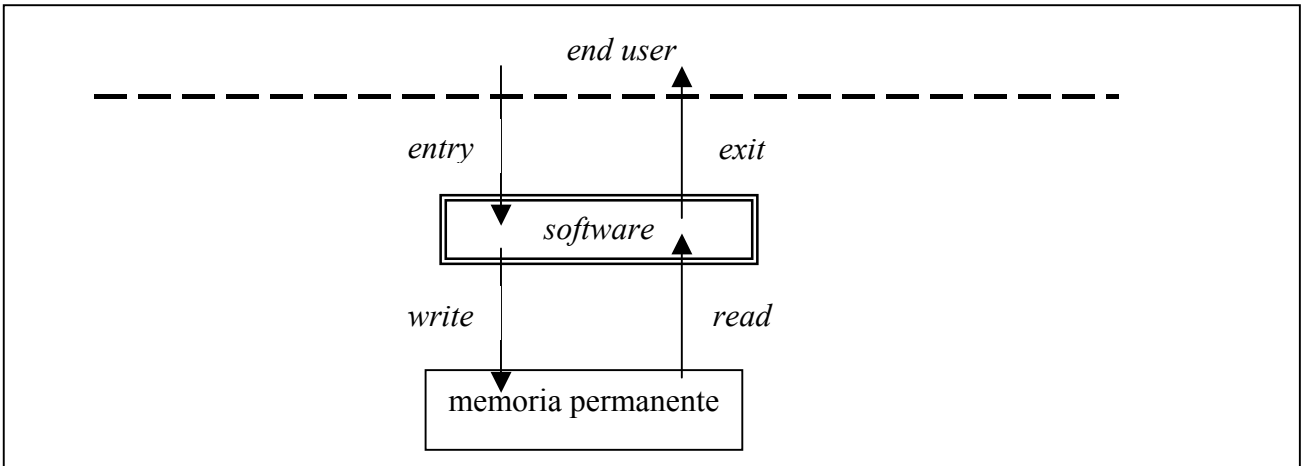


Figura 3.10

A questo livello di astrazione, le apparecchiature di *input/output*, il sistema operativo e i DBMS sono naturalmente invisibili. La figura 3.11 mostra un confine da un punto di vista dello sviluppatore nel caso di due moduli applicativi. Si noti come la coppia di sottoprocessi 1 equivale ad una *write* (vista dal modulo di logica applicativa) e la doppia coppia di sottoprocessi 2 equivale a una *read* (quella a sinistra a una *read request* e quella a destra alla *read* vera e propria).

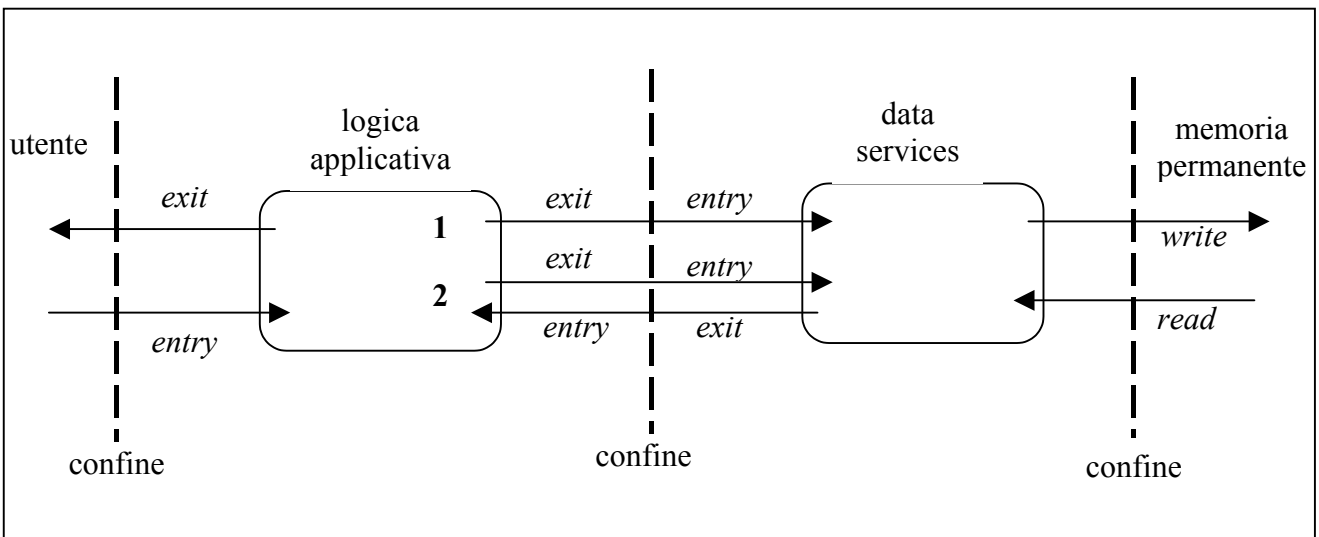


Figura 3.11

La figura 3.12 mostra confini dal punto di vista dello sviluppatore nel caso di una architettura *client/server*.

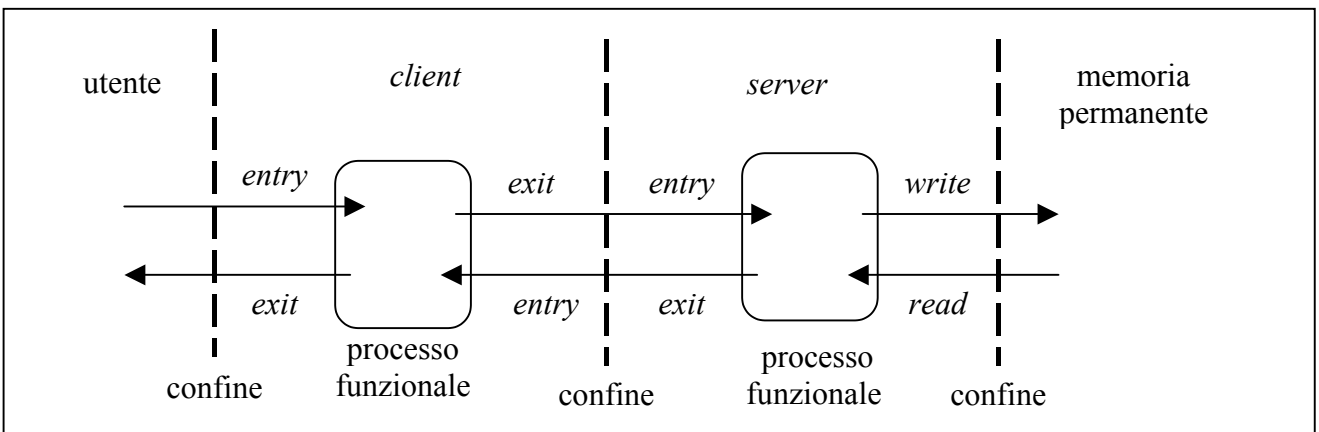


Figura 3.12

Terzo passo: i processi funzionali. Un processo funzionale è un componente elementare di un FUR che comprende un insieme di trasferimenti di dati che è eseguibile in modo indipendente da altri processi; come detto, corrisponde a una transazione nel metodo Mark II. Un processo funzionale deriva da almeno una componente ben identificata del FUR; viene in generale attivato da un evento e comprende almeno due trasferimenti dati. Un processo funzionale appartiene naturalmente a un solo livello e termina quando avviene l'ultimo trasferimento dati (che non sia sincronizzato con un altro trasferimento).

Quarto passo: i data group. Un *data group* è un insieme di attributi, cioè di valori che si riferiscono a oggetti di interesse. Un *data group* può essere transiente, quando non sopravvive al processo, oppure persistente; in quest'ultimo caso si distinguono quelli a durata indefinita, che risiedono nella memoria di massa, da quelli di corta durata che servono alla comunicazione tra processi.

Nelle applicazioni dei sistemi informativi, tipicamente un *data group* è una entità.

Come primo esempio si consideri una interrogazione verso un *data base* per conoscere gli *item* (valori di una certa entità) che soddisfano una data condizione. In questo caso, esiste un *data group* con i parametri di selezione associato al sottoprocesso (di trasferimento dati) di *entry*; un *data group* con la lista degli *item* associato a un sottoprocesso di *exit* e l'entità in questione associato al sottoprocesso di *read*. La figura 3.13 mostra il processo funzionale di risposta alla *query*: che misura 3 *cfsu*, perché ci sono tre attraversamenti di confine.

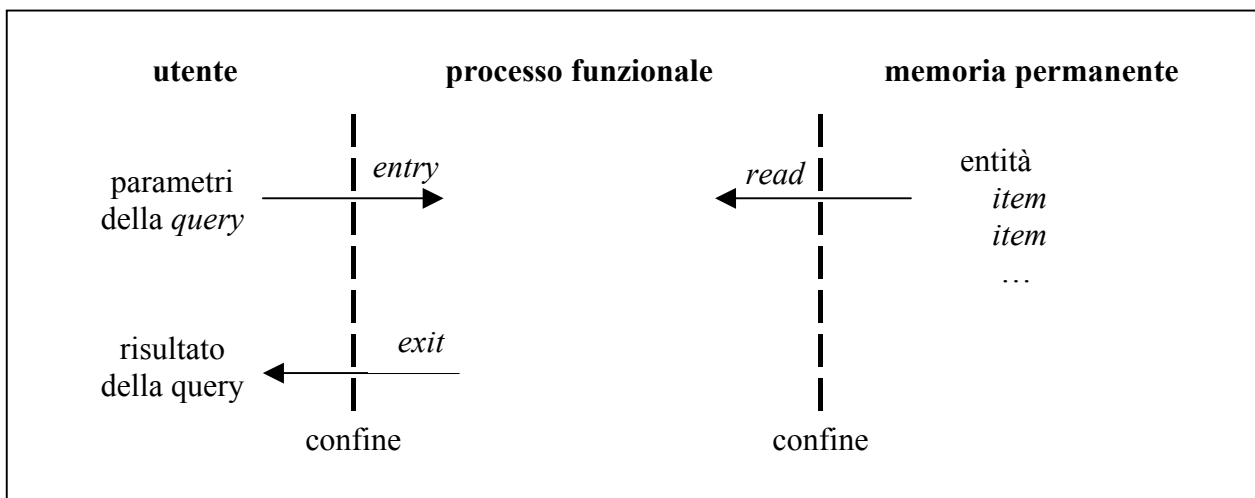


Figura 3.13

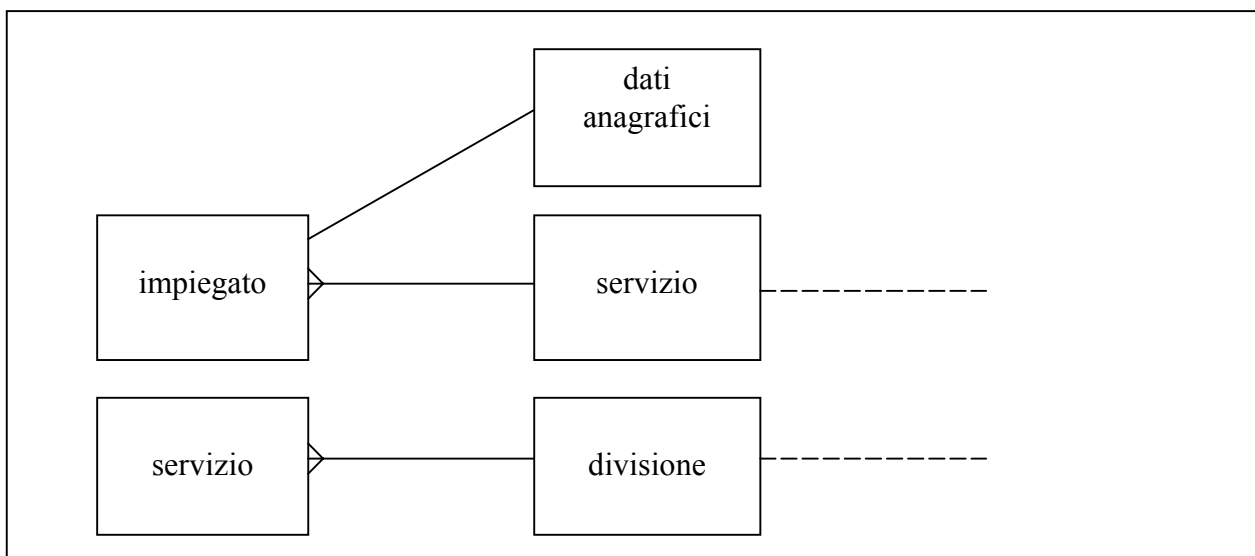


Figura 3.14

In figura 3.14 è mostrato un frammento di uno schema concettuale di dati che evidenzia due entità (impiegato e servizio). Si consideri il seguente FUR: costruire a richiesta un *report* con l'elenco degli impiegati ordinati per divisione e servizio interno alla divisione; dare i totali degli impiegati per tutta l'azienda, per divisione e per servizio. Il processo che soddisfa al FUR misura 7 *cfsu*; infatti i sottoprocessi di trasferimento dati sono:

- 1 *entry*: il *trigger* o evento scatenante,
- 2 *read*: le entità impiegato e servizio,
- 4 *exit*: l'entità impiegato (ovviamente) e i 3 totali.

Nei sistemi *realtime*, esempi di *data group* sono:

- un messaggio, associato a un sottoprocesso di *entry* in un sistema di *message switching* (di solito è anche associato, più o meno cambiato, a un sottoprocesso di *exit*);
- lo stato di un oggetto (valvola aperta o chiusa, livello di pressione di un recipiente, temperatura di un giunto, ecc) in un sistema di controllo di processo;
- struttura di dati, come grafi di percorsi, tabelle di indirizzamento, ecc, di cui si parla nei FUR e sono permanenti (per esempio in una ROM) e accessibili a tutti i processi.

Quinto passo: gli attributi. All'interno di un *data group*, un attributo è la più piccola parte di dati che viene nominata e ha significato per il FUR.

Non sono da considerare tutti gli attributi creati a causa dell'implementazione, come le maschere di stampa, le tabelle di conversione interna, ecc. Il passo per individuare gli attributi è "facoltativo" in quanto non serve per il calcolo della mole, ma è solo utile per controllare la compatibilità del modello generico con il FUR, controllando l'esistenza di tutti gli elementi dei dati significativi per l'utente.

3.3.2 Fase di misurazione.

La rappresentazione schematica di questa fase è mostrata in figura 3.15.

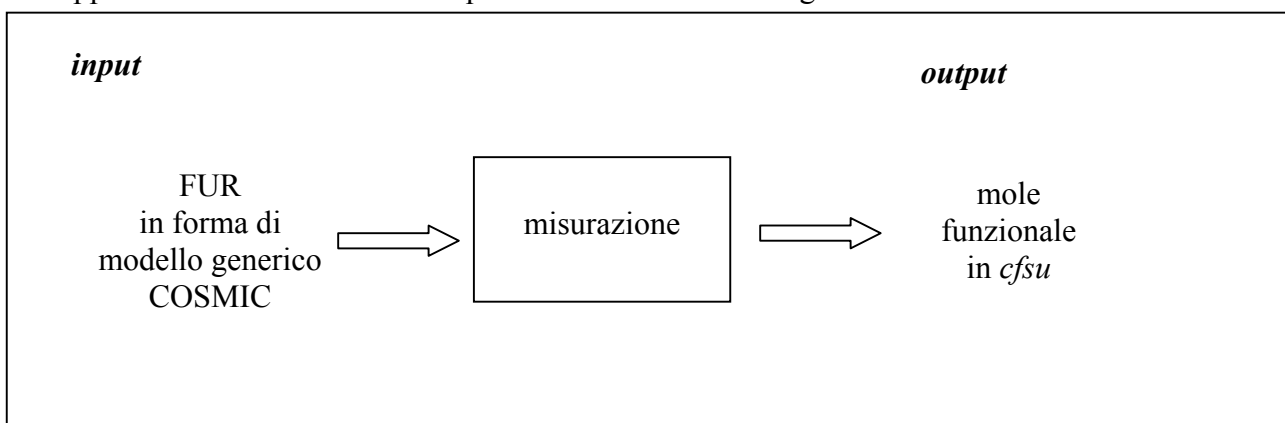


Figura 3.15

La fase si compone di due passi: identificare i movimenti dei dati e aggregare i risultati.

Primo passo: identificare i movimenti dei dati.

Un movimento di dati consiste in un processo di trasferimento di dati che riguarda un singolo *data group*; come per i sottoprocessi (di trasferimento dei dati) ne esistono 4 tipi:

- *entry*: sposta un singolo *data group* attraverso un confine dall'utente all'interno del processo; può contenere delle elaborazioni (per esempio la validazione), ma non cambia i valori dei dati;
- *exit*: sposta un singolo *data group* attraverso un confine dall'interno del processo verso l'utente; non legge (*read*) i dati che sposta, ma può eseguire elaborazioni come conversioni, editing, ecc.
- *read*: sposta un singolo *data group* dalla memoria persistente all'interno del processo; può includere eventuali elaborazioni necessarie per raggiungere e identificare i dati;
- *write*: sposta un singolo *data group* dall'interno del processo alla memoria persistente; include eventuali elaborazioni per cambiare i valori dei dati, necessarie per effettuare l'operazione (codifiche, segmentazioni, ecc.).

In questo passo valgono le seguenti regole generali:

1. Regola di non duplicazione; ogni movimentazione di dati riguarda un *data group* e non una sua istanza; per esempio, se occorre movimentare un certo *data group* e l'implementatore decide di farlo con due azioni in due punti diversi del processo, questa movimentazione viene contata una volta sola; come altro esempio, in un *software real time*, se un processo legge ciclicamente un dato (per vedere se è cambiato), allora esistono due movimentazioni (*entry* o *read* a seconda dei casi): una è associata all'istanza del sottoprocesso che legge il dato non cambiato e una al sottoprocesso che legge il dato cambiato. Inoltre, naturalmente, un sottoprocesso inserito in un *loop* o comunque in una ripetizione che legge istanze diverse dello stesso *data group* conta per uno.
2. Regola di aggiornamento (*update*); nel caso di un *read* seguito direttamente solo da elaborazione sui dati (senza altri sottoprocessi di trasferimento) e da un successivo *write* sui dati così elaborati, si conta un solo movimento (di aggiornamento).
3. Regola di cancellazione; cancellare l'istanza di un *data group* (dalla memoria permanente) equivale ad un *write*.
4. Se l'utente è, a sua volta, un modulo *software*, ad ogni *entry* del modulo in questione corrisponde un *exit* del modulo utente e viceversa.
5. Ad ogni sottoprocesso di trasferimento di dati è associata una sola movimentazione

Secondo passo: aggregare i risultati.

Per il *software* da sviluppare, si sommano, per tutti i moduli software da considerare e per tutti i processi che compongono ciascun modulo, i *cfsu* ottenuti contando le movimentazioni dei dati.

Per il *software* da mantenere, si sommano i *cfsu* ottenuti contando le movimentazioni dei dati aggiunte, modificate o tolte.

Naturalmente, tutti i conteggi, per poter essere sommati, devono essere fatti con lo stesso scopo, la stessa estensione e lo stesso punto di vista.

Per esempio, se si devono cambiare un certo numero di pagine di un sito aziendale, si pone il problema di valutare il costo di tale operazione.

I passi concettuali da fare per questa valutazione sono:

- valutare la mole associata a questo intervento di manutenzione (evolutiva), misurandola, per esempio, in *cfsu* (col metodo COSMIC) oppure in *Fpi* (col metodo Mark II) o in *object point*;
- calcolare l'impegno (direttamente o attraverso il lavoro) tipicamente in giorni uomo associato alla mole, al sistema di sviluppo, alle competenze necessarie e alle capacità delle persone a disposizione; in questo semplice caso, probabilmente, è necessario un solo tipo di competenze e corrispondentemente un solo valore dell'impegno; in casi più complessi possono rendersi necessari valutazioni più complesse (per analisti, sistemisti, altri esperti, ecc); tipicamente le varie aziende hanno dei coefficienti medi di produttività standard;
- calcolare il costo complessivo utilizzando i tipi di impegno previsti e le relative tariffe.

3.4 Valutazione preliminare della mole

Nella fase preliminare istruttoria di un progetto (per esempio per lo studio di fattibilità o durante la pianificazione di massima), si presenta il problema di stimare il costo del progetto, prima quindi di avere tutte le informazioni necessarie per valutare la mole del software da sviluppare, che, per esempio nel modello a cascata, sono disponibili alla fine dell'analisi. Occorre quindi una tecnica che consenta di fare questa stima. Questo procedimento, detto di preanalisi, consiste nella scomposizione gerarchica della applicazione da realizzare, in componenti, macrofunzioni e processi come mostrato nella figura 3.16.

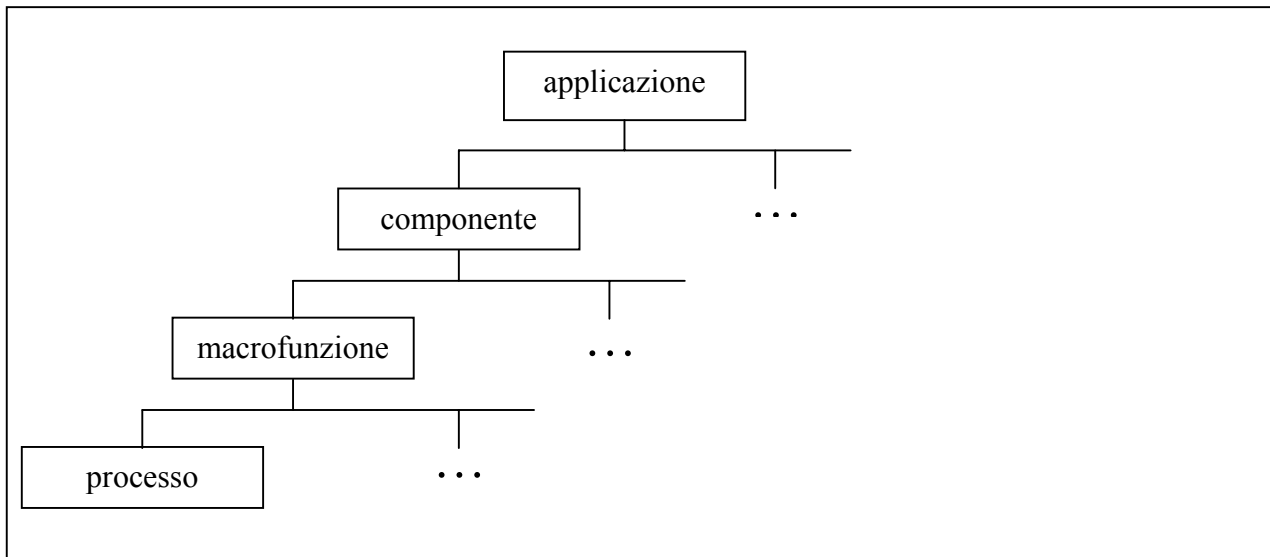


Figura 3.16

Naturalmente, la scomposizione in 4 livelli è una indicazione media che deve essere adattata di volta in volta a seconda della complessità del progetto da realizzare: l'albero che descrive la scomposizione viene espanso finché le foglie non hanno una dimensione analoga o al più differente per un ordine di grandezza. Il contributo informativo di questo processo dipende in modo essenziale dall'esperienza e dalla capacità di chi lo esegue. Alla fine della scomposizione si elencano i vari processi con una descrizione sommaria, ma sufficiente per ottenere una stima attendibile della mole complessiva; questa si può ottenere applicando uno dei due seguenti metodi.

Valutazione assoluta. Ciascun processo viene classificato in una delle 4 categorie (piccolo, medio, grande, molto grande); ad essi sono associati dei *cfsu* mediante la seguente tabella

Mole media di un processo in <i>cfsu</i>	
Piccolo	3 - 4
Medio	7 - 8
Grande	11 - 12
Molto grande	20 - 25

Valutazione relativa. Si supponga di avere n oggetti (numerati, quindi ordinati)

$$m_1, m_2, \dots, m_n,$$

per esempio gli n processi risultanti dalla scomposizione gerarchica (cioè le foglie sopra illustrate). Supponiamo che si conosca il valore, in generale di una misura, per esempio la mole, di ciascun oggetto:

$$m_j \rightarrow w_j;$$

fissando il primo oggetto, si può costruire il vettore colonna dei confronti (rapporti fra le misure) di ogni elemento con quello fissato:

$$\begin{array}{c} w_1/w_1 \\ w_2/w_1 \\ \dots \\ w_n/w_1 \end{array}$$

e quindi si può costruire la matrice

$$A = \begin{array}{c|cccc} & \frac{w_1}{w_1} & \frac{w_2}{w_1} & \dots & \frac{w_n}{w_1} \\ \hline \frac{w_1}{w_1} & \frac{w_1}{w_1} & \frac{w_2}{w_1} & \dots & \frac{w_n}{w_1} \\ \frac{w_2}{w_1} & \frac{w_1}{w_2} & \frac{w_2}{w_2} & \dots & \frac{w_n}{w_2} \\ \hline \frac{w_1}{w_n} & \frac{w_1}{w_n} & \frac{w_2}{w_n} & \dots & \frac{w_n}{w_n} \\ \vdots & \vdots & \vdots & & \vdots \\ \frac{w_n}{w_1} & \frac{w_1}{w_n} & \frac{w_2}{w_n} & \dots & \frac{w_n}{w_n} \\ \hline \frac{w_1}{w_1} & \frac{w_2}{w_1} & \dots & \dots & \frac{w_n}{w_1} \end{array},$$

la cui colonna j -esima è il confronto di tutti gli oggetti con l'oggetto j -esimo. Si vede immediatamente che sono vere le seguenti affermazioni:

- sulla diagonale ci sono tutti 1, quindi la traccia è uguale a n ;
- $a_{ij} = 1/a_{ji}$;
- $a_{ij} = a_{ik}/a_{jk}$ (proprietà transitiva);
- il rango della matrice è uguale a 1 perché ogni colonna (riga) è uguale alle altre a meno di un multiplo;
- poiché il rango è la dimensione di $im(A)$, esiste un solo autovettore (che è una qualunque delle colonne);
- l'autovalore corrispondente è la traccia, cioè n .

Per analogia, supponiamo di avere sempre gli n processi di cui però non si conosca la relativa misura, ma che si possano comunque confrontare l'uno rispetto all'altro, in maniera che per ogni coppia si possa dare una stima, anche se grossolana, del rapporto fra le relative moli. Quindi si può costruire una matrice

$$A = \| a_{ij} \|$$

Dove ancora per costruzione $a_{ii} = 1$ e $a_{ij} = 1/a_{ji}$, ma non è più $a_{ij} = a_{ik}/a_{jk}$, cioè la matrice non è "coerente" perché la valutazione dei rapporti fra le misure degli oggetti sono qualitative e quindi non è assicurata la proprietà transitiva. Se si suppone che tutte le misure varino al più di un ordine di grandezza, si può ancora concludere che l'autovettore corrispondente all'autovalore più alto ha le componenti che sono proporzionali a una stima attendibile delle moli dei rispettivi processi. Si noti naturalmente che per stabilire i rapporti fra n quantità sono sufficienti $n-1$ confronti; il metodo ne esegue $n(n-1)/2$ perché si suppone che i confronti non siano esatti e la ridondanza è usata, appunto, per diminuire l'inesattezza.

Invece di risolvere il problema agli autovalori, si può più semplicemente normalizzare le colonne e prenderne la media w ottenendo una buona valutazione dell'autovettore. Da

$$Aw \approx \lambda w$$

si può ottenere una stima dell'autovalore risolvendo ciascuna riga dell'equazione prendendone la media. La differenza tra il λ medio così ottenuto e n dice di quanto è "lontana" la matrice A dall'essere "transitiva", cioè coerente, e quindi stima di quanto il vettore w è lontano dalla valutazione reale dal rapporto tra le moli dei singoli processi. Alla fine, l'analisi approfondita di un singolo processo consente quindi di stimare con buona attendibilità la mole di tutti gli altri.

3.5 Modello di previsione del costo e della durata di un progetto

È ragionevole prevedere che nel prossimo lustro la distribuzione delle persone coinvolte nella produzione di *software* sia sulle cinque categorie seguenti.

1. Programmatori utenti finali (*end user programmers*): circa 95%. La competenza tipica di questa categoria riguarda principalmente prodotti come piccoli *data base management system*, strumenti statistici e di presentazione (come SAS e Business Objects) e fogli elettronici.
2. Costruttori di generatori e assemblatori di applicazioni (*Application generators e composition aid*: circa 1%). Esempi tipici dei prodotti messi a punto da questa fascia di specialisti sono gli ERP (*Enterprise Resources Processing*) e i RAD (*Rapid Application Developers*).
3. Costruttori di applicazioni (*Application composers*: poco più dell'uno per cento). Sono gli addetti che usano gli strumenti messi a punto dagli specialisti del punto 2) precedente.
4. Integratori di applicazioni (*System integrators*: circa 1%). Sono gli addetti che creano *software* per le "grandi" applicazioni come per esempio quelle di telecomunicazione, di finanza, della pubblica amministrazione e della difesa.
5. Costruttori di *software* di base (*Infrastructure programmers*: circa 1%). Sono gli specialisti che mettono a punto le nuove versioni dei sistemi operativi, dei DBMS, dei sistemi di rete, ecc.

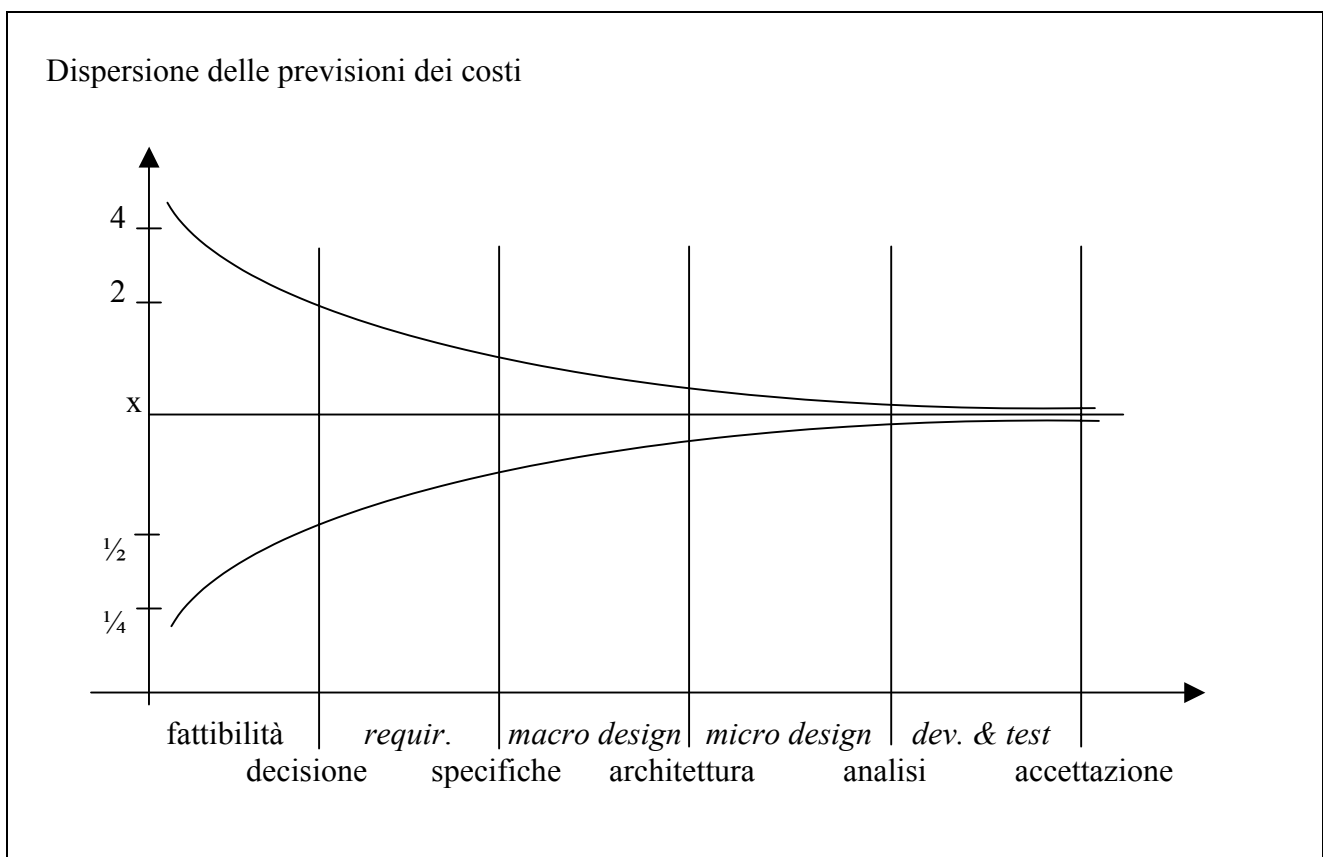


Figura 3.17

In particolare, il Bureau of Labor Statistics degli USA prevede, per il mercato interno, che nel 2005 ci siano oltre 50 milioni di unità del tipo 1 e circa 700000 di ciascuno degli altri tipi; previsioni analoghe, con cifre assolute più basse, sono ragionevoli anche per l'Unione Europea.

L'attività delle persone della prima fascia non richiede strumenti sofisticati per la gestione dei progetti. Gli addetti delle altre fasce, che secondo le stime sopra riportate sono oltre 2.5 milioni negli USA e poco meno di 2 milioni nella Unione Europea, sono coinvolti in progetti, spesso di elevatissima complessità, in cui il parametro di merito più importante è il costo (dell'intero progetto).

Un consolidato filone di analisi statistica mostra che le capacità previsionali di stimare il costo di un progetto varia in maniera abbastanza uniforme a mano a mano che il progetto si svolge; per un tipico progetto a cascata l'andamento della dispersione delle previsioni sui costi è quello mostrato nella figura 3.17. Si noti che la curva è simmetrica perché la scala usata nelle ordinate è logaritmica: cioè è di gran lunga più frequente sottostimare, piuttosto che sovrastimare, il costo dei progetti.

Per poter inquadrare nella giusta prospettiva il problema trattato in questo paragrafo, si ricorda che il costo di una *singola attività* di sviluppo di *software* è, in generale, proporzionale all'impegno necessario (in generale misurato in giorni \times uomo o mesi \times uomo):

$$\text{Costo} = (\text{costo unitario}) \times \text{impegno} = (\text{costo unitario}) \times \text{costante} \times \text{lavoro}.$$

Il problema attuale è ora quello di valutare il costo dell'intero progetto che, naturalmente, non consiste solo della somma dei costi delle attività di sviluppo dei singoli moduli (di cui si può valutare il lavoro). Un'indagine statistica svolta per molti anni e per un congruo numero di progetti ha evidenziato che il costo totale del progetto non è neppure proporzionale al lavoro complessivo del *software* da sviluppare.

Viene ora esposto il modello COCOMO II, abbastanza usato per prevedere il costo dei progetti di sviluppo *software*.

Le prime indagini statistiche alla base dello sviluppo di questo modello sono state svolte nella seconda metà degli anni settanta; una prima versione del modello è stata pubblicata nel 1981; nella seconda metà degli anni 90, con l'esperienza via via accumulata è stata pubblicata una seconda versione del modello che viene attualmente utilizzata.

COCOMO II si articola in tre (famiglie di) modelli che differiscono per l'ambito di applicazione. Tutti prevedono il costo totale del progetto in funzione di una stima del lavoro, misurato con metriche diverse a seconda del contesto.

I tre modelli sono descritti in funzione delle competenze dei programmatori articolate nelle categorie sopra definite; nella fase preliminare di un progetto può essere più adatto un modello e in fase avanza essere opportuno utilizzarne un altro. I tre modelli sono:

1. **application composition**: riguarda i progetti portati avanti esclusivamente da sviluppatori appartenenti alla categoria 3, usa come metrica gli *object point*;
2. **early design**: viene usato nella fase preliminare di un progetto in cui sono coinvolti sviluppatori delle categorie 2, 4 e 5; usa come metrica i *function point*;
3. **post architecture**: si usa nelle fasi avanzate (quando è stata fissata in maniera definitiva l'architettura *hw* e *sw*) dei progetti che richiedono le competenze delle categorie 2, 4 e 5.

L'idea di base che sta dietro ai modelli **early design** e **post architecture** è che l'impegno complessivo per il progetto, usualmente misurato in mesi \times uomo, non è proporzionale al lavoro necessario per sviluppare il *software*, ma viene espresso dalla formula:

$$PM = A \times \text{lavoro}^B \times \text{Correzioni}.$$

Il termine B è il così detto fattore di scala:

- $B < 1$, si ha economia di scala: se il lavoro che si deve fare raddoppia, l'impegno complessivo per ottenerlo è meno che raddoppiato, cioè la produttività aumenta con l'aumentare della

mole; un esempio è costituito dai progetti relativamente piccoli in cui il tempo di *set up* è rilevante rispetto allo sforzo complessivo del progetto come mostrato nella figura 3.18 .

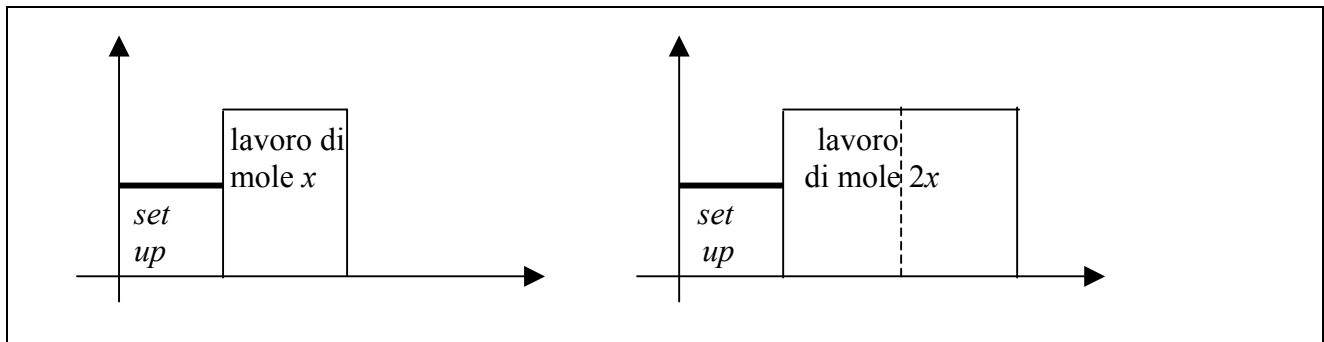


Figura 3.18

- $B = 1$, in questo caso le economie e le diseconomie di scala si bilanciano (questa è l'ipotesi per la prima famiglia di modelli).
- $B > 1$, si hanno diseconomie di scala; è il caso più frequente che è dato essenzialmente dai seguenti due fattori:
 - aumento non lineare della necessità di comunicazione tra le persone impegnate nel progetto (dovuto anche alla maggiore necessità di coordinamento),
 - aumento non lineare dello sforzo per l'integrazione in un unico sistema delle varie componenti.

In base all'analisi statistica fin qui prodotta, una stima per B viene espressa dalla formula:

$$B = 0.91 + 0.01 \times \sum_j SF_j,$$

dove gli SF_j sono detti *fattori di scala* e la loro somma può valere fino a 31; quindi nel modello COCOMO II si ha

$$0.91 \leq B \leq 1.22.$$

I valori numerici che compaiono in questo, come in altri modelli, sono dovuti a un processo di calibrazione statistico che usa un gran numero di casi in cui si presume valido il modello (progetti compiuti di cui si conosce dettagliatamente tutta la storia). È naturale quindi che, con il passare del tempo, questi valori possano cambiare sia perché si ha un campione più numeroso dal quale dedurre i parametri, sia perché il campione è più preciso, cioè contiene progetti selezionati in modo che ad essi si applica più puntualmente il modello.

I fattori di scala sono 5: la seguente tabella ne mostra il nome e i valori (arrotondati per semplicità).

	molto basso	basso	medio	alto	molto alto	estremamente alto
<i>PREC</i>	6	5	4	2	1	0
<i>FLEX</i>	5	4	3	2	1	0
<i>REAL</i>	7	6	4	3	1	0
<i>TEAM</i>	5	4	3	2	1	0
<i>PMAT</i>	8	6	4	3	1	0

Nel seguito viene illustrato il significato di ciascuno dei 5 parametri; i valori numerici riportati in tabella sono approssimati e fanno riferimento a una scala qualitativa asimmetrica a sei valori; rispetto alla normale scala qualitativa a 5 valori (vedi la scala per la valutazione dei rischi) è stato aggiunto il valore "estremamente alto".

- *PREC*: indica se il progetto ha precedenti noti e quindi se fa parte (valore alto) o meno (valore basso) della "cultura" di tutti gli attori;

- *FLEX*: indica che le specifiche del progetto di sviluppo sono poco (valore basso) o molto (valore alto) flessibili; come esempio, gli standard per lo sviluppo, le interfacce, le specifiche funzionali, etc.
- *REAL*: è relativo ai possibili rischi, cercando di valutare in che misura (bassa o alta) sono stati trattati e affrontati;
- *TEAM*: valuta la coesione (bassa o alta) tra tutti gli attori del progetto: utenti, committenti, sviluppatori, management, etc;
- *PMAT*: questo parametro misura la “maturità” (bassa o alta) del processo. L’accezione esatta in cui questo termine va inteso in questo contesto è ampiamente trattata nel capitolo degli Standard SEI (Software Engineering Institute).

A mo’ di esempio consideriamo due casi:

- 1) il rifacimento della informatizzazione dell’anagrafe della popolazione di un grande comune, utilizzando una tecnologia “web like”;
- 2) la messa a punto del programma di gestione di una lavatrice da parte di un’azienda elettromeccanica che per la prima volta costruisce elettrodomestici.

Esempio 1.

La gara bandita dal comune è stata fortemente voluta dagli amministratori, ma questa decisione è stata poco condivisa dagli utenti finali, già a loro agio col precedente sistema. La gara è stata aggiudicata a una ditta che ha vasta esperienza nello sviluppo di applicazioni per Enti Pubblici, non possiede la certificazione ISO 9000, ma ha dei buoni e riconosciuti standard aziendali per la produzione del *software*.

In questo caso, una ragionevole valutazione dei fattori di scala è la seguente

- *PREC* = 1, perché il progetto ha numerosi precedenti e quindi fa parte della cultura di tutti gli attori, anche se sviluppato con tecnologia innovativa;
- *FLEX*, = 3, perché da una parte le specifiche funzionali del problema sono rigorosamente definite (da norme e da leggi), dall’altra c’è ampia libertà di adattare la progettazione delle interfacce per migliorarne la produzione, sempre garantendo una elevata usabilità;
- *REAL* = 1, infatti i rischi di questi progetti sono ampiamente prevedibili e trattabili, anzi dire che una certa azienda ha esperienza su certi progetti significa esattamente che è capace di (contribuire a) prevedere ed evitare i rischi relativi;
- *TEAM* = 3, perché è prevedibile mancanza di coesione tra gli utenti, i committenti e il gruppo di sviluppo;
- *PMAT* = 5, perché, per le caratteristiche dell’azienda, si può prevedere che possa essere messo in campo un processo di sviluppo probabilmente a livello 2 del CMM (vedi capitolo degli standard SEI).

La somma dei fattori di scala è 13 e quindi $B = 1.04$, cioè il progetto si presenta con una piccola diseconomia di scala, sostanzialmente non avvertibile e comunque (ampiamente) all’interno degli errori di valutazione.

Esempio 2.

L’azienda elettromeccanica ha iniziato una nuova filiera di prodotti assumendo tecnici da aziende concorrenti per progettare un prodotto innovativo e ricco di funzioni automatizzate molto evidenti e sfruttabili efficacemente per la campagna di lancio. Oltre ai tecnici acquisiti dall’esterno, fanno parte del progetto altri esperti di queste problematiche e informatici dell’azienda.

In questo caso, una ragionevole valutazione dei fattori di scala è la seguente

- *PREC* = 5, perché il progetto non ha precedenti ai quali abbiano partecipato tutti gli attori e quindi non fa parte della cultura da loro condivisa;

- $FLEX = 4$, perché le interfacce (il pannello dell'elettrodomestico) e lo *hardware* da usare (il *chip* scelto per il controllo) sono già individuati, come pure il sistema di sviluppo, in modo che le specifiche sono flessibili solo per includere o meno funzionalità innovative;
- $REAL = 7$, infatti i rischi di questi progetti sono elevati e poco prevedibili dal momento che il prodotto è il primo di una serie e non fa quindi ancora parte della cultura aziendale;
- $TEAM = 3$, perché è prevedibile una moderata mancanza di coesione tra i partecipanti al gruppo di sviluppo che hanno estrazione e storia diverse;
- $PMAT = 6$, perché non sembra esistere alcun standard di processo adottato dal gruppo di lavoro.

La somma dei fattori di scala è 25 e quindi $B = 1.16$, cioè il progetto si presenta con una notevole diseconomia di scala (sostanzialmente avvertibile anche tenuto conto degli eventuali errori di valutazione).

Dal confronto dei fattori di scala emersi dai due esempi, si può dedurre che, nel primo caso, raddoppiando il lavoro, il costo (cioè l'impegno) sostanzialmente raddoppia ($2^B = 2.056$); nel secondo caso aumenta con velocità maggiore ($2^B = 2.235$).

Nella formula

$$PM = A \times (\text{lavoro})^B \times \text{Correzioni}$$

l'esponente B è adimensionale, e quindi non è connesso ad alcuna unità di misura; il fattore A fa usualmente riferimento al lavoro misurato in migliaia di SLOC (*Standard Lines Of Code*), o KLOC, e vale 2.94 mesi \times uomo/KLOC; per esempio, quando non ci sono correzioni, ci vogliono circa 3 mesi \times uomo per portare a compimento un progetto che al suo interno richiede la produzione di 1000 linee di codice (nei 3 mesi è compresa, naturalmente, la quota parte di tutte le fasi del progetto). Nella tabella che segue, sono mostrate, per alcuni linguaggi di programmazione, le linee standard di codice corrispondenti a un *function point*.

Linguaggio di programmazione	SLOC/FP
Ada 95	49
ABAP	16
Visual Basic	32
C	128
C++	55
Cobol (85)	91
Fortran 77	105
Pascal	91
Prolog	64
Lisp	64
Excel	6
Java	53

Nelle attività relative a un progetto, non si scrive solo il codice che sarà incluso nel prodotto finale, ma vengono scritti altri programmi che o vengono usati solo nella fase di *set up* oppure vengono scartati perché appartenenti a prototipi non usati o perché implementano funzionalità giudicate non più rilevanti. Per tener conto di questo, per determinare il valore del lavoro da usare per stimare il costo del progetto si usa la formula

$$\text{lavoro} = \text{lavoro}' \times (1 + Brak), \quad \text{con } 0 \leq Brak \leq 1.$$

Sia *lavoro'* sia *Brak* vengono dedotti dai *function point* del *software* effettivamente scritto; *lavoro'* si riferisce alla quantità di *software* realmente installata e *Brak* rende conto della percentuale di *sof-*

ware fatto, ma non installato, come per esempio i programmi di recupero dei dati da vecchi data base, prototipi scartati, programmi di correzione o “ripulitura” dei dati, ecc.

I due modelli visti (*early design* e *post architecture*) differiscono per un fattore che modifica l’impegno totale a seconda di quello che si conosce del progetto nei due diversi stadi. La forma generale è

$$PM = A \times (\text{lavoro})^B \times \prod_j EM_j,$$

dove sono state specificate le correzioni *EM* (*effort modifier*) che sono 7 per l’*early design* e 17 per la *post architecture* (il diverso numero esprime appunto la diversa conoscenza che si ha nei due stadi). Per semplicità, ci limitiamo a discutere con qualche esempio il primo modello. L’elenco dei *modifier* è il seguente.

- *RCPX*: misura la complessità (bassa o alta) del *software* costruito dal progetto e quindi la sua *reliability*: si noti che queste due caratteristiche sono fortemente (e negativamente) correlate;
- *RUSE*: indica la quantità (bassa o alta) di componenti fatte in modo da essere riusabili nel progetto corrente o in altri;
- *PDIFF*: rende conto delle difficoltà (basse o alte) connesse con la piattaforma (di sviluppo, ma soprattutto di esercizio): in particolare la volatilità (stabilità dello *hardware* e del *software*), vincoli sulle prestazioni (per esempio tempi di risposta) e vincoli sulle dimensioni (per esempio occupazione di memoria, fattore spesso critico nei sistemi *embedded*);
- *PERS*: misura la capacità professionale di base e la cultura (basse o alte) del personale (in tutte le varie figure coinvolte) e della sua stabilità durante il progetto (indice di *turnover*);
- *PREX*: rende conto della esperienza specifica (bassa o alta) del personale sul tipo di applicazione, piattaforma e *tool* di sviluppo adottati nel progetto;
- *FCIL*: rende conto delle *facilities* (cattive o buone), cioè stazione di lavoro, *software* di simulazione e *test* e, soprattutto, il numero di siti (molti o pochi) in cui viene sviluppato il progetto;
- *SCED*: rende conto del tempo a disposizione rispetto a quello *nominale* (più basso o più alto) per portare a termine il lavoro (vedi più avanti).

I valori numerici standard da usare per questi parametri sono riportati nella tabella seguente.

	molto basso	basso	medio	alto	molto alto	estremamente alto
<i>RCPX</i>	0.7	0.9	1.	1.1	1.3	1.6
<i>RUSE</i>		0.9	1.	1.1	1.3	1.5
<i>PDIFF</i>			1.	1.1	1.3	1.6
<i>PERS</i>	1.3	1.2	1.	0.9	0.7	
<i>PREX</i>	1.2	1.1	1.	0.9	0.8	
<i>FCIL</i>	1.2	1.1	1.	0.9	0.7	
<i>SCED</i>	1.4	1.2	1.	1.0	1.0	1.0

Il modello COCOMO II predice anche una *durata nominale* del progetto: si suppone che siano assegnate un “numero opportuno” di persone e, naturalmente, il valore ottenuto è il risultato della valutazione statistica della storia di un grande numero di progetti che si sono **conclusi con successo**.

La formula per valutare la durata nominale è la seguente

$$T_0 = E \times PM' (D + 0.002 \times \sum_j SF_j),$$

dove:

PM' è l’impegno calcolato con la formula precedente, **con SCED = 1**,

SF_j sono i fattori di scala già visti (quindi l’esponente varia fra 0.28 e 0.34),

e i parametri valgono

$$E = 3.67,$$

$$D = 0.28.$$

Dall'analisi della formula si deduce che la durata nominale T_0 dei progetti **conclusi con successo** è “poco” sensibile all'impegno PM che si deve porre in essere (perché l'esponente è minore di 1); questo apparente paradosso si spiega col fatto che per portare a conclusione con successo un progetto la quantità di risorse da dedicare deve crescere più velocemente del lavoro da fare. Sono fattori principalmente economici (come per esempio il *time to market*) che influenzano la durata ottimale del progetto e quindi determinano il numero di persone che devono essere coinvolte.

Ricapitolando, la formula complessiva per calcolare l'impegno (in mesi \times uomo) necessario per completare un progetto in funzione del lavoro (in linee di codice) è

$$PM = A \times (\text{lavoro})^{(0.91 + 0.01 \times \sum_j SF_j)} \times \prod_j EM_j.$$

Esempio.

Supponiamo che occorra fare un mini progetto che comporti la realizzazione di 150 *function point* (mole); se si prende la decisione di sviluppare in Java, questo corrisponde alla scrittura di circa 8000 linee di codice, cioè 8 KLOC (lavoro). Calcoliamo l'impegno, tralasciando per il momento il fattore di correzione (cioè supponiamo che il prodotto degli EM_j sia uguale a 1) e supponendo che la somma dei fattori di scala sia 18.97 (che è circa corrispondente al valore medio per tutti gli SF_j):

$$PM = 2.94 \times 8^{(0.91 + 0.001 \times 18.97)} = 2.94 \times 8^{1.0997} = 28.9 \text{ persone} \times \text{mese}.$$

Introduciamo ora la correzione dovuta agli *effort modifier* EM_j , supponendo che:

- si richieda un'alta riusabilità del codice prodotto, cioè $RUSE = 1.34$;
- sia presente una bassa dimestichezza con gli strumenti di sviluppo, $PREX = 1.09$;
- tutti gli altri parametri siano nella norma (tutti uguali a 1).

Si deduce quindi

$$PM = 28.9 \times 1.34 \times 1.09 = 42.3 \text{ persone} \times \text{mese}.$$

Calcoliamo ora la durata nominale:

$$T_0 = E \times PM^{(D + 0.002 \times \sum_j SF_j)} = 3.67 \times 42.3^{(0.28 + 0.002 \times 18.97)} = 12.1 \text{ mesi}.$$

Dall'esame dei due risultati si deduce che per concludere il progetto nel tempo nominale occorrono in media durante tutto il progetto circa 3.5 persone.

Se si vuole ridurre la durata T del progetto al 75% del tempo nominale T_0 , occorre introdurre per il parametro $SCED$ un valore appropriato che, per una riduzione del tempo del 75%, corrisponde a 1.43 (invece del valore 1 valido per il tempo nominale).

$$T = 0.75 \times 12.1 = 9.1 \text{ mesi}.$$

La correzione dovuta agli EM_j è $1.34 \times 1.09 \times 1.43$ e quindi:

$$PM = 28.9 \times 1.34 \times 1.09 \times 1.43 = 60.4 \text{ persone} \times \text{mese}.$$

Si noti come l'impegno aumenta *per il solo fatto* che il progetto deve essere terminato in un tempo più breve rispetto a quello standard.

Si deduce immediatamente che il numero medio di persone occorrenti per tutta la (nuova) durata sono $60.4/9.1 = 6.7$ persone.

L'andamento approssimato del parametro $SCED$ rispetto ai vincoli sulla terminazione del progetto è mostrato nella figura 3.18

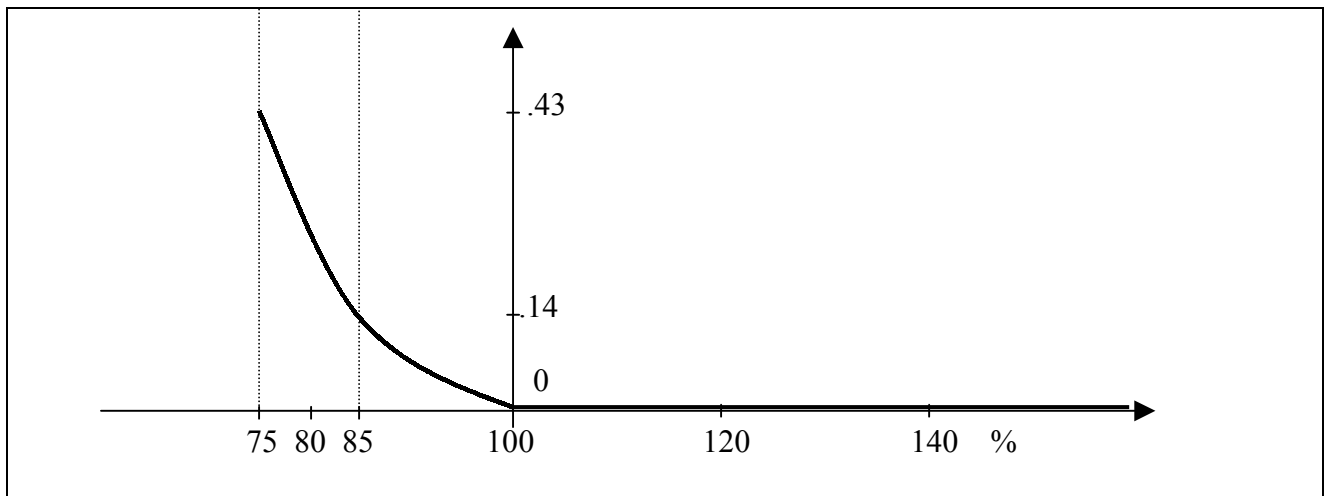


Figura 3.18

Una demo dell'uso di COCOMO II si trova (alla data di scrittura di questo testo) al sito:
www.softstarsystems.com.

In realtà, l'andamento dell'impegno in funzione del tempo di completamento è espresso dal diagramma PNR (Putnam, Norden, Rayleigh), si veda la figura 3.19.

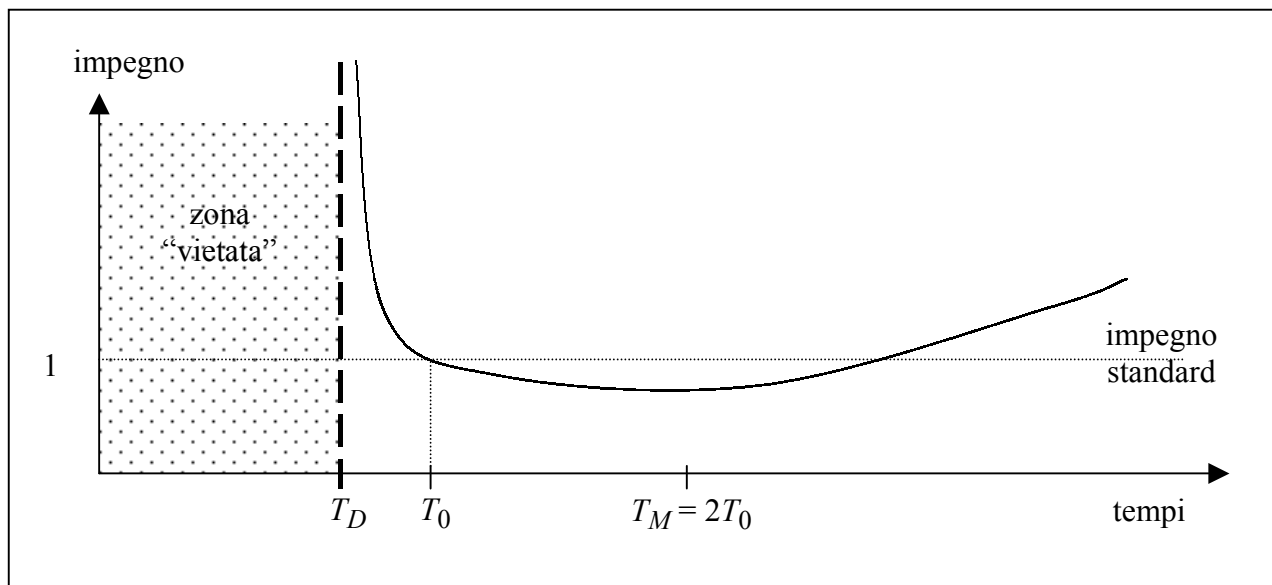


Figura 3.19

Se T_0 è il tempo standard di completamento del progetto (calcolato con COCOMO II) l'impegno minore si ottiene a circa $T_M = 2T_0$, con un *team* "ridotto", sostanzialmente perché si recupera in produttività, a causa di una migliore efficienza.

A destra di T_M l'impegno torna a crescere a causa del:

- *turnover* delle persone,
- cambiamento dei requisiti,
- cambiamento (dei *release*) degli ambienti di sviluppo o di esercizio (situazione che richiede un maggior impegno di risorse per la conversione).

Per spostarsi a sinistra di T_0 , cioè per diminuire il tempo totale del progetto, rispetto a quello standard, occorre aumentare le risorse di personale: in questo caso, in generale, diminuisce l'efficienza (cioè la produttività) per la necessità di maggiore comunicazione fra le persone. (Naturalmente queste considerazioni valgono per progetti in materie complesse come sono appunto quelli di sviluppo software.) Si è osservato infatti che aggiungendo ulteriori risorse allo *staff* che permetterebbe di completare il progetto nel tempo T_D , pari al 75% di T_0 , le persone aggiunte bilanciano la perdita di produttività e quindi non si ha miglioramento.

Ricapitolando, per la relazione fra impegno e lavoro, vale, come si è visto, la formula:

$$PM = A \times \text{lavoro}^B \times EM.$$

Dati, pubblicati alla fine del 2000 e raccolti nel biennio precedente, mostrano i seguenti valori medi:

Valori per B	
Media generale	1.052
Sistemi <i>embedded</i>	1.110
<i>e-commerce</i>	1.030
Applicazioni <i>web</i>	1.030
Industria aeronautica	1.070

Valori per $A \times EM$	
Media generale ($EM = 1$)	2.94
Sistemi <i>embedded</i> ($EM < 1$)	2.58
<i>e-commerce</i>	3.60
Applicazioni <i>web</i>	3.30
Industria aeronautica ($EM < 1$)	2.77

Riassumendo, si vede come i progetti di *e-commerce* e di applicazioni *web* sono relativamente semplici, ma sono condotti con "scarsa" efficienza. In contrasto, lo sviluppo di sistemi *embedded* e di applicazioni aeronautiche è caratterizzato da progetti complessi, ma realizzati con elevata efficienza.