

Ingegneria del software: Generalità

Introduzione

La soluzione di un generico problema mediante un sistema di elaborazione consiste nel passaggio dalla, descrizione del problema di partenza al programma finale, in cui l'utente può utilizzare il calcolatore per inserire i dati in ingresso e ottenere i risultati in modo automatico.

Essendo l'informatica una disciplina trasversale, il caso da risolvere può riguardare insegnamenti diversi quali l'economia e la matematica e, ad esempio, potrebbe essere il calcolo dello stipendio dei dipendenti di un'azienda o la valutazione delle orbite seguite dai pianeti nel nostro sistema solare.

Queste considerazioni ci suggeriscono che, se si decide di risolvere un problema mediante un sistema di elaborazione, si devono seguire alcune fasi intermedie che conducono sia alla scrittura del programma finale sia alla creazione della documentazione necessaria per la descrizione dell'intera soluzione del problema.

È prassi comune dividere l'attività di progettazione di software in due:

Programmazione in piccolo (programming in the small): Riguarda sistemi software di piccole dimensioni (di solito, un solo programmatore e meno di sei mesi di lavoro). Ad esempio: ordinare un vettore, trovare il massimo di una funzione matematica, trovare il cammino più corto su una carta geografica, cc. Servono: intuizione, originalità di ricerca delle soluzioni, profondità di analisi. È un'attività più scientifica che manageriale.

Programmazione in grande (programming in the large): Riguarda sistemi software di grandi dimensioni (di solito, più programmatori e più di sei mesi di lavoro). Ad esempio: scrivere il software di gestione di una biblioteca, o per il controllo di una centrale nucleare, o un sistema operativo, ecc. Servono: visione d'insieme, capacità organizzative. È un'attività più manageriale che scientifica.

Ovviamente il confine fra queste due tipologie di programmazione è sfumato.

L'ingegneria del software è la disciplina che si occupa di sviluppare modelli, metodi, tecniche e strumenti per lo sviluppo di sistemi software che rispettino certi standard di qualità. In particolare, si occupa delle problematiche che sorgono per sistemi software di grandi dimensioni, quindi della programmazione in grande.

Le caratteristiche del software

In seguito è mostrato un elenco di caratteristiche che un sistema di buona qualità deve possedere.

Correttezza

È ovviamente importante che il programma non contenga errori. In realtà, se è relativamente facile accertarsi della correttezza di programmi di piccole dimensioni, questo è ben più difficile, se non impossibile, per programmi di grosse dimensioni, in cui ci sono sempre errori.

Più che cercare un'ideale e irraggiungibile assenza di errori e parlare di correttezza, si cerca quindi di avere pochi errori ed errori non gravi, con conseguenze non catastrofiche, e si parla di:

1. **Affidabilità:** tempo medio fra due guasti (scoperte di errori). Se un programma funziona per parecchio tempo senza rivelare errori, non vuoi dire che ne sia privo, ma solo che, presumibilmente, ne ha pochi.
2. **Robustezza:** assenza di comportamenti catastrofici in seguito ad errori. Un programma non è sicuramente robusto se, ad esempio, a causa di una divisione per zero cancella tutto il disco rigido.

Efficienza

Un programma è efficiente se usa risorse (tempo, quantità di memoria, ecc.) in modo limitato. Un programma che ordina un array di un milione di elementi in un secondo senza richiedere altra memoria che quella per memorizzare gli elementi è più efficiente di uno che impiega 10 minuti e ha bisogno di 1 megabyte di memoria centrale.

Usabilità

Un buon programma deve essere usabile in modo semplice dagli utenti. Ad esempio, i telecomandi per videoregistratori con 42 tasti saranno potentissimi, ma ben pochi utenti se ne dicono soddisfatti. Ritornando ai sistemi software, un sistema operativo WIMP è più usabile di un sistema operativo a linea di comando, e lo stesso vale per un programma qualsiasi.

Modificabilità

Un buon programma deve essere modificabile in modo semplice, senza doverlo riscrivere da zero, per adattarlo ad un ambiente in evoluzione.

Comprensibilità (o leggibilità)

Un buon programma deve essere facilmente comprensibile e non deve usare, inutilmente, algoritmi astrusi. Questo sia perché capita che qualcuno metta le mani sul programma scritto da un altro, sia perché quando si riprende un programma scritto un mese prima, spesso, non si ricorda più nulla.

Esistono utili tecniche per avere una buona comprensibilità: usare i commenti, scegliere in modo accurato i nomi delle variabili, essere coerenti nella scelta delle convenzioni in tutto il programma, usare in modo corretto l'indentazione, ecc.

Manutenibilità

Deve essere semplice effettuare la manutenzione di un programma, perché un software di grandi dimensioni non è mai finito; servono sempre degli interventi, aggiustamenti, ritocchi per correggere errori e adattano a nuove esigenze.

Portabilità

Deve essere facile trasferire un software da un ambiente ad un altro (su un altro

hardware, su un altro sistema operativo, ecc.). Quest'attività è denominata [porting](#).

Per ottenere una buona portabilità è indispensabile cercare di usare le caratteristiche più standard del linguaggio, cercando di prevedere il comportamento del software in costruzione anche in ambienti differenti.

Riusabilità

Un software scritto per risolvere un problema è riusabile per risolvere un problema diverso. Più un software è leggibile e modificabile, più la sua riusabilità sarà alta; l'approccio orientato agli oggetti permette di ottenere una buona riusabilità del codice.

Il ciclo di vita del software

Un prodotto software ha una vita che può essere scomposta in varie fasi:

[Studio di fattibilità](#). È una fase preliminare in cui si valuta se è possibile costruire il sistema e se il rapporto costi/benefici sarà minore di 1. Per fare ciò si studia il problema, si prepara un progetto tecnico di massima e si analizzano più in dettaglio le fasi critiche, meno prevedibili.

[Analisi e specifica dei requisiti](#). Il problema e le esigenze degli utenti sono analizzate e rappresentate in un documento chiaro e univoco detto specifiche (dei requisiti). Le specifiche (che sono il prodotto di questa fase) contengono quindi:

- le funzionalità che il sistema dovrà avere,
- le prestazioni,
- l'ambiente di utilizzo,
- le interfacce esterne (con utenti, altro software, hardware),
- gli eventuali vincoli di progetto (tempi, soldi, risorse, ecc.),
- i requisiti di qualità (è diverso scrivere un software per una centrale nucleare o per una banca).

[Progetto dell'architettura](#). Vengono identificate le componenti (moduli) del sistema, e le relative connessioni. Per individuare i moduli, ci si basa sull'analisi del problema effettuata nella fase precedente (quindi sulle specifiche) e su una sua scomposizione in sottoproblemi.

Partendo dalle specifiche, la fase di progettazione ha come obiettivo quello di determinare la soluzione del problema indipendentemente dagli strumenti HW e SW che saranno impiegati in seguito per la realizzazione del progetto stesso; A tale proposito, al termine progetto può anche essere associato l'aggettivo astratto. Un progetto astratto è costituito dal [modello o struttura dei dati](#), ovvero l'insieme dei dati del problema e la loro organizzazione e dall'[algoritmo](#).

Poiché il modello dei dati deve essere generale, al suo interno devono comparire variabili ovvero "contenitori" destinati a memorizzare i dati da elaborare. Le variabili nel modello dei dati possono essere:

- di [ingresso](#), impiegate per memorizzare i dati di input;
- [intermedie](#), ausiliarie o di lavoro, che contengono tutti i dati parziali necessari durante l'elaborazione per passare dai dati in ingresso ai risultati;

- di **uscita**, per la presentazione dei risultati finali.

Il caso più semplice di modello di dati è quello costituito soltanto dalle variabili che contengono i dati in ingresso i risultati e i dati intermedi necessari per l'elaborazione. In molti casi i dati di un problema possono essere numerosi e disposti in modo confuso; lo scopo del modello è quello di organizzare i dati in modo ordinato, fornendo una struttura in modo da agevolare il processo di soluzione.

Lo sviluppo dell'algoritmo consiste nella definizione dell'insieme dei passi necessari per passare dai dati in ingresso ai risultati. Un algoritmo deve fornire la soluzione di un problema indipendentemente dal linguaggio di programmazione che si utilizzerà per scrivere il programma finale.

Progetto dei moduli. Viene progettato ogni singolo modulo (dedicato ad uno specifico sottoproblema) che compone il sistema; occorre comprendere come ogni modulo possa svolgere i compiti cui è destinato.

Si noti che finora non si è scritta neppure una riga di codice.

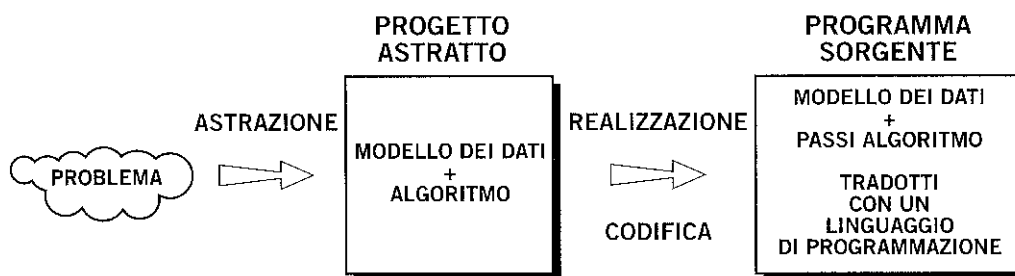
Codifica e verifica dei moduli. Qui finalmente si comincia a scrivere del codice; si lavora su ogni modulo isolatamente, senza considerare gli altri moduli, le interazioni fra moduli saranno considerate nelle fasi seguenti.

La codifica è la traduzione manuale del modello dei dati e dell'algoritmo risolutivo di un problema in un programma sorgente, tramite un linguaggio di programmazione.

A tal fine, ogni linguaggio di programmazione mette a disposizione del programmatore alcune istruzioni per tradurre:

- il modello dei dati in strutture di dati comprensibili alla macchina;
- i passi dell'algoritmo in operazioni eseguibili dal computer.

La traduzione del modello dei dati e dell'algoritmo nel programma finale eseguibile, scritto nel linguaggio naturale del calcolatore (linguaggio macchina binario), è di fatto impossibile da effettuare direttamente. Per agevolare l'attività del programmatore, la traduzione del programma sorgente in quello macchina è realizzata in modo automatico da opportuni programmi traduttori (compilatori e interpreti).



Frequentemente, nel passato, il termine codifica è stato impiegato come sinonimo di programmazione (in senso stretto), mentre in realtà indica soltanto una fase dell'intera attività della programmazione stessa

Integrazione e verifica del sistema. Ora i singoli moduli realizzati vengono messi insieme, effettuando la verifica della correttezza di tutto il sistema. A questo punto il sistema è quasi completo, ma in laboratorio: non è ancora stato provato nell'ambiente reale.

Installazione e verifica sul campo. Il sistema viene trasportato (magari via

Internet) nell'ambiente in cui dovrà operare e viene nuovamente verificato, controllando che rispetti le specifiche iniziali.

Accettazione e rilascio. Il funzionamento del sistema viene verificato dal committente, che lo accetta (firma e paga). Il sistema viene quindi rilasciato e comincia a funzionare.

Manutenzione. Come già osservato, sono sempre necessarie delle modifiche al sistema anche dopo la sua messa in opera, sia per correggere errori, sia per adattarlo alle modifiche dell'ambiente.

Si ricordi che tali fasi sono relative alla programmazione in grande; per la programmazione in piccolo le fasi che vengono sicuramente effettuate sono:

1. Analisi e specifica dei requisiti,
2. Progetto dei moduli,
3. Codifica e verifica dei moduli.

L'analisi

In questa fase si devono descrivere con precisione le esigenze dell'utente.

È una fase particolarmente critica perché gli errori commessi qui costano cari: spesso bisogna rifare tutto il lavoro. Nonostante ciò, ancora oggi l'analisi viene spesso effettuata in modo artigianale e affrettato; solo negli ultimi anni si stanno diffondendo metodologie e strumenti semi-automatici per effettuare l'analisi in modo più sistematico.

La costruzione di una specifica non è un'attività banale; spesso conviene procedere in modo incrementale, partendo da una prima versione da raffinare in passi successivi.

Il prodotto di questa fase è un documento, detto specifica. Una specifica è:

- Una definizione dei bisogni dell'utente
- Un'indicazione e guida per i realizzatori
- Un punto di riferimento per l'attività di verifica
- Un punto di riferimento durante la manutenzione.

Una specifica può essere espressa in linguaggio:

- **naturale** (italiano, inglese, ...). Questo porta a specifiche chiare ma spesso ambigue
- **formale** (formule matematiche, equazioni, ...). Questo porta a specifiche più precise ma spesso meno chiare.

Una buona specifica deve essere:

- **Precisa** (non ambigua)
- **Chiara** (facilmente comprensibile)
- **Coerente** (non contraddittoria). Soprattutto quando il sistema è complesso, capita sovente di dire da una parte "bianco" e dall'altra "nero", senza rendersene conto
- **Essenziale** (non ridondante). Bisogna limitarsi a dire che cosa il sistema deve fare, non come deve farlo (questo sarà compito del progettista, fasi successive).

Il progetto

In questa fase si deve definire l'architettura del sistema: individuare le componenti e le relative interazioni. Anziché affrontare un problema tutto insieme, lo si scompone in vari sottoproblemi, i quali a loro volta possono essere scomposti in sotto-sottoproblemi, così via.

Le singole componenti di un sistema software sono dette moduli.

Un modulo può essere definito come una componente dedicata a svolgere una specifica funzione; esso è costituito da due parti:

- **interfaccia** (la parte visibile dall'esterno),
- **implementazione** (l'insieme degli elementi che permettono di svolgerne le sue funzioni).

Un sistema è quindi composto da vari moduli che interagiscono fra di loro. I moduli e le loro interazioni costituiscono l'architettura di un sistema software.

I principali tipi di interazione tra moduli sono del tipo:

- **usa**, quando un modulo usa i servizi forniti da un altro modulo.
- **parte-di**, quando un modulo (sottomodulo) è una parte di un altro modulo.

L'architettura di un sistema software (i suoi moduli e le loro relazioni) viene solitamente rappresentata in forma grafica: un modulo è un rettangolo, che eventualmente ne contiene altri (relazione parte-di); fra i rettangoli vi possono essere frecce che rappresentano relazioni usa.

Nella progettazione dei moduli è opportuno privilegiare scelte che consentano il riuso dei moduli anche in situazioni diverse, sia all'interno dello stesso sistema, sia in un altro sistema. Ciò consente ovviamente un risparmio di lavoro e un aumento di produttività.

Per ottenere un progetto orientato al riuso vi sono due tecniche:

1. **Generalizzazione**. Se un modulo deve calcolare una funzione, estendere il dominio della funzione.
2. **Astrazione**. Consiste sia nell'individuare tipi di dati astratti che possono essere utilizzati in più situazioni sia nell'individuare compiti, azioni che si ripresentano spesso e possono essere utilizzati in varie situazioni.

Per avere una buona modularizzazione bisogna costruire moduli con:

- **Alta coesione**. La coesione è una misura di quanto le componenti di un modulo sono correlate fra di loro: quel che c'è dentro ad un modulo deve essere molto correlato, bisogna tenere insieme il più possibile le componenti che riguardano un certo aspetto.
- **Basso accoppiamento**. L'accoppiamento è una misura di quanto moduli diversi sono collegati fra di loro: quel che c'è in moduli diversi deve essere poco correlato, bisogna separare il più possibile componenti che riguardano aspetti differenti.

Top-down e bottom-up

In che ordine procedere nell'individuazione dei moduli? Vi sono due alternative:

Top-down (dall'alto in basso): si parte dai moduli, poi si individuano i sotto-moduli,

poi i sottosottomoduli e così via.

Bottom-up (dal basso in alto): si parte da moduli semplici (magari anche già costruiti), li si aggrega a formare altri moduli, poi si aggregano anche i moduli ottenuti, e così via.

Entrambe le alternative presentano vantaggi e svantaggi:

- la progettazione top-down porta sì a sistemi puliti ad alto livello, ma talvolta anche a una duplicazione di sforzi (perché, ad esempio, lo stesso sottomodulo, presente in moduli diversi, viene poi implementato due volte da gruppi diversi).
- la progettazione bottom-up, al contrario, evita la duplicazione di sforzi e facilita il riuso dei software (ci si accorge subito se qualcosa che ho a disposizione mi può servire), ma talvolta si tende a usare qualcosa che non è proprio adatto, o a costruire qualcosa che poi non serve, e porta spesso a sistemi concettualmente più confusi ad alto livello di aggregazione.

La progettazione top-down è stata privilegiata fino ad una decina di anni fa; oggi (grazie anche all'avvento della progettazione ad oggetti, illustrata nel prossimo paragrafo), la tendenza è forse di favorire di più l'approccio bottom-up. Come al solito, nella pratica si adotta un approccio misto, in parte top-down e in parte bottom-up: si parte top-down e poi si procede bottom-up.

È importante sottolineare l'indipendenza delle fasi di progettazione e di programmazione: il linguaggio di programmazione non deve influenzare, almeno in linea di principio, la fase di progetto.

La programmazione

Una volta terminata la fase di progetto bisogna implementare effettivamente il programma in un linguaggio di programmazione. Sono di seguito riassunti alcuni concetti che un buon programmatore deve conoscere.

Programma = algoritmi + strutture dati. Bisogna pensare parallelamente alle strutture dati e agli algoritmi che lavorano su di esse. L'esperienza aiuta molto, in quanto la conoscenza di algoritmi di base ricorrenti (ordinamento, ricerca binaria, ecc.) permette di ottenere in modo più veloce il prodotto finale.

Sviluppo incrementale. Un programma va sviluppato per raffinamenti successivi: si parte da una prima versione grezza (spesso parte in linguaggio naturale e parte in un linguaggio di programmazione) che si raffina più volte fino ad ottenere il programma completo.

Divide et impera. Se un problema è troppo complicato per essere affrontato in un colpo solo, va diviso in sottoproblemi, più semplici e più trattabili.

Programmazione strutturata. È oramai assodato da parecchi anni che il goto è un costrutto da usare il meno possibile.

Tipi di dati astratti. Il programmatore deve porsi come obiettivo la costruzione di tipi di dati astratti che avvicinino alla soluzione del problema.

Programmazione ad oggetti. Se usata in modo corretto, conduce in modo naturale a programmi che implementano tipi di dati astratti con occultamento delle informazioni.

Vi sono semplici regole di buona programmazione che aiutano a scrivere codice in

modo migliore:

- Privilegiare la leggibilità all'efficienza,
- Usare le caratteristiche standard, pure di un linguaggio di programmazione,
- Se dopo aver scritto il programma ci si accorge che non è chiaro, spesso ciò è dovuto al fatto che il programmatore non ha ben chiaro il problema, per cui, prima di ragionare a livello del linguaggio di programmazione, ragionare a livello del problema,
- Usare la programmazione difensiva: scrivere codice in più che fa controlli sui dati manipolati dal programma. Ad esempio, se il valore della variabile x deve essere sempre positivo, mettere in punti strategici del programma un'istruzione del tipo: $\text{if } (x < 0)$,
- Preferire gli annidamenti else nelle if,
- Usare le parentesi per evitare ambiguità,
- Scrivere (e aggiornare) i commenti insieme al codice,
- Non accontentarsi mai della prima versione funzionante,
- Scrivere bene i commenti, ma non inserirne troppi,
- Scegliere bene i nomi delle variabili (evitare abbreviazioni, usare convenzioni coerenti in tutto il programma).

La verifica

Un programma contiene sempre degli errori, per cui occorre verificare ogni linea di programma.

Nella fase di verifica (o controllo di qualità) si deve accertare che il prodotto sia privo di errori e che rispetti le specifiche. Spesso la risposta non è binaria (si/no), ma è un valore percentuale.

La verifica può inoltre riguardare una caratteristica:

- **oggettiva**: ad esempio, il tempo impiegato dal sistema per dare la risposta ad un certo input;
- **soggettiva**: ad esempio, quanto il programma è comprensibile, modificabile, usabile, ecc.

Le caratteristiche soggettive non sono meno importanti di quelle oggettive, anche se indubbiamente di più difficile misurazione. Da parecchi anni sono allo studio le cosiddette metriche del software, con cui si vuole cercare di misurare nel modo più preciso possibile queste caratteristiche soggettive. Possibili metriche potrebbero essere: il numero di commenti, il numero di errori rilevati durante un periodo prefissato di uso del sistema, ecc.

Un errore può essere più o meno grave: se si scrive un identificatore in modo scorretto, probabilmente verrà segnalato dal compilatore, ma se si scrive un software che non rispetta le specifiche (ad esempio, un software di gestione di un supermercato con cinque casse anziché cinquanta), ben difficilmente il cliente sarà soddisfatto. Si noti che la gravità di un errore dipende dal tempo: più tardi si scopre un errore e peggio è.

Una tipica classificazione degli errori è quella che distingue fra errori:

- durante la compilazione (compile time)
- durante l'esecuzione (run time).

I primi sono individuati dal compilatore, che di solito fornisce anche un'indicazione sul come correggerli. I secondi sono spesso più suddivisi: spesso si manifestano solo in corrispondenza di certi dati di input o, ancora peggio, di certe condizioni dell'ambiente.

Le tecniche di verifica

Vi sono due categorie di tecniche di verifica:

1. **Il collaudo** (o testing): si fanno delle prove di esecuzione, esperimenti, per vedere se le caratteristiche di qualità sono rispettate. Tali prove vanno documentate.
2. **L'analisi** (formale o meno): si analizza il prodotto per cercare di dedurre logicamente le sue caratteristiche. Se l'analisi è formale, si dimostra un teorema che asserisce, ad esempio, la correttezza del programma rispetto alle specifiche (che devono essere espresse anch'esse in modo formale).

Le tecniche del primo tipo sono dette anche dinamiche (si esegue il programma), quelle del secondo tipo sono dette statiche.

Il collaudo può dimostrare la presenza di errori, non l'assenza (a meno di fare tutte le prove possibili): si può dimostrare un teorema che asserisce che l'unico collaudo ideale (che se c'è un errore lo trova sicuramente) è quello che prova il sistema in tutte le condizioni.

Nonostante ciò, il collaudo può essere molto utile, se ben fatto. Per fare bene un collaudo bisogna massimizzare la probabilità di trovare errori. Vediamo tre possibili tecniche per fare ciò:

1. **Collaudo strutturale** (glass-box). Si scelgono i dati di test sulla base del programma, cercando di coprire tutte le istruzioni: se non si esegue mai un frammento di programma ovviamente non si trovano eventuali errori presenti.
2. **Collaudo funzionale** (black-box). Per scegliere i dati di test non ci si basa sul codice, ma sulle specifiche.
3. **Collaudo sulle condizioni limite**. Sia nel collaudo strutturale sia in quello funzionale si divide l'insieme dei dati di input in parti e si scelgono alcuni dati da ognuna di queste parti. Ma spesso gli errori si verificano sul confine fra due parti. Questa tecnica suggerisce di scegliere i dati di test sui confini. Questa tecnica è basata sul fatto che uno degli errori più comuni è l'uso di $<$ anziché \leq , o la ripetizione del corpo di un ciclo una volta di troppo (o una volta in meno).

Finora si è fatto riferimento solo al collaudo per la programmazione in piccolo.

Nel caso della programmazione in grande capita spesso di collaudare un modulo prima che siano realizzati i moduli che lo usano, o che sono usati da quel modulo. In questi casi bisogna costruire del software aggiuntivo, scrivendo dei moduli vuoti che simulino la presenza dei moduli mancanti, implementando le funzionalità minime necessarie per effettuare il test.

Nell'analisi formale, per verificare un programma, non lo si esegue, ma lo si analizza in modo formale cercando di dimostrare alcune proprietà.

Per dimostrare la correttezza di un programma bisogna innanzi tutto che le specifiche

siano espresse in modo formale; di solito si danno **precondizioni** e **postcondizioni**: condizioni che devono valere, rispettivamente, prima e dopo l'esecuzione di un programma. E poi possibile verificare che l'esecuzione di un frammento di codice, se effettuata in uno stato che rispetta le precondizioni, porta in uno stato che rispetta le postcondizioni.

Il debugging

L'attività di collaudo porta alla scoperta della presenza di malfunzionamenti (guasti) in un programma, ma il passaggio da sintomo (guasto) a causa (errore) è spesso tutt'altro che banale. Anche qui ci sono tecniche e strumenti che vengono in aiuto al verificatore, ma è importante l'intuito del programmatore.

L'attività di individuazione e correzione di errori è detta debugging. Questo termine indica l'eliminazione dei bug, che in inglese significa pulce, ma in gergo informatico sta per errore ed ha una sua origine storica: deriva dal fatto che nei primi calcolatori l'attività di debugging doveva essere eseguita quando qualche insetto (bug) si infilava fra valvole e diodi compromettendo il funzionamento del programma in esecuzione.

Si è detto che un buon insieme di dati di collaudo deve massimizzare la probabilità di trovare errori.

I primi programmatori avevano ben poco a disposizione per ricercare l'errore: un memory dump (la stampa del contenuto della memoria centrale, e quindi del valore delle variabili del programma) e tanta certosa pazienza per eseguire a mano il programma fino alla scoperta dell'errore. Ma questa tecnica è dispersiva e ormai non si usa più; oggi si usano i **watch point** (o spy point): si visualizzano i valori di alcune variabili in opportuni punti del programma (ad esempio, subito prima che si verifichi l'errore, alla chiamata di sottoprogrammi, all'ingresso e uscita di cicli, ecc.). Un watch point può essere realizzato in due modi:

1. Modificando il programma e aggiungendo opportune istruzioni nei punti chiave.
2. Usando strumenti forniti dall'ambiente di programmazione.

Per concludere, tre suggerimenti:

1. Cercare di restringere la ricerca ad aree sempre più piccole: non ridursi a leggere il codice per cercare un errore di cui l'unica cosa che si sa è che è nel programma, o nel modulo di interfaccia utente. Cercare prima di restringere il campo della ricerca con le tecniche appena viste.
2. Ha sempre ragione lui (il calcolatore): un calcolatore ben difficilmente commette errori, un essere umano lo fa spesso; se qualcosa non funziona, è sempre colpa del programmatore.
3. Avere tanta pazienza...