4 Codifica, Test e Collaudo.

Al termine della fase di progettazione, a volte anche in parallelo, si passa alla fase di codifica e successivamente alla fase di test e collaudo.
In questa parte viene approfondita la fase di codifica e vengono descritte le strategie di test e collaudo.

4.1 Codifica.

La fase di progettazione ha portato alla scomposizione del sistema in un insieme di moduli (progetto statico) ed all'effettuazione di scelte circa la loro realizzazione (progetto dettagliato).
Ogni modulo viene affidato ad una squadra di programmatori per la stesura del codice.
Durante la codifica si utilizza normalmente una metodologia (ad esempio programmazione strutturata).
Il linguaggio dovrebbe essere scelto alla fine della fase di progettazione, di solito in una tipica azienda di produzione software vengono utilizzati un numero limitato di linguaggi.

L'ambiente aziendale impone inoltre al programmatore di seguire una serie di standard:
- l'utilizzo di un linguaggio
- uno stile di programmazione, cioè convenzioni su nomi, identazione, dichiarazioni ecc.
- determinate modalità di documentazione
- un insieme di regole di conservazione del codice

Vi sono degli strumenti di ausilio al programmatore:
- interprete, per il debugging interattivo
- editor guidato della sintassi
- dizionario dei dati (ad esempio per l'identificazione di omonimie di variabili)
- cross referencing (generatore di riferimenti incrociati) spesso accoppiato al dizionario dei dati (ad esempio per trovare tutti i posti nei quali viene utilizzata una determinata variabile).
- Browser, per la visualizzazione grafica della struttura del programma con la possibilità di accedere direttamente ad ogni unità
- Generatore di rapporti per la documentazione

Generalmente si tratta di un ambiente di sviluppo che poggia su un database, che permette il riutilizzo del software.

4.2 Test e collaudo.

La fase di test e collaudo serve a verificare che il prodotto software sia corretto. Si tratta di una fase lunga e costosa, può incidere anche per il 50% sul costo del prodotto software.
Il test e collaudo è costituito a sua volta da due fasi.
- Verifica (verification) che viene svolta all'interno dell'azienda del produttore e che vuole scoprire se il prodotto realizzato funziona correttamente. Viene confrontato con il documento

delle specifiche e con il progetto di dettaglio. La domanda alla quale si cerca di rispondere è: "Are we building the product right?".

- Certificazione (Validation), si tratta in pratica dell'ultimo test. Viene svolta nell'ambiente finale, nell'azienda del cliente, davanti al cliente. Si verifica che sia stato ottenuto il prodotto richiesto. Viene fatto il confronto con il documento dei requisiti. La domanda alla quale i cerca di rispondere è "Are we building the right product?".

Vi sono diverse tecniche per verificare un prodotto software.

- Tecniche statiche. Viene verificata la correttezza del programma senza eseguirlo, attraverso:
  - Verifiche informali, in base all'esperienza
  - Verifiche formali, utilizzando tecniche matematiche
- Tecniche dinamiche e testing. Il codice viene eseguito su opportuni dati campione per individuare discrepanze tra il comportamento atteso e quello effettivo.
  Una verifica dinamica può rilevare la presenza di errori, ma non la loro assenza. Pertanto non si può avere la certezza assoluta della correttezza del prodotto software:
  - La prova di correttezza può a sua volta essere sbagliata
  - Può essere provata la correttezza relativamente alle specifiche funzionali, ma non si può stabilire se le specifiche non funzionali (ad esempio prestazioni) siano state soddisfatte.
- Ispezione del codice. Può apparire come una tecnica primitiva, ma invece è molto efficace nel rilevamento degli errori.

Queste tecniche non vanno utilizzate in modo esclusivo ma è vantaggioso integrarle: sono tecniche complementari.

Qual è l'obiettivo della fase di test?
L'obiettivo di questa fase non deve essere quello di far vedere che il software non contiene errori, bensì di trovarne il maggior numero possibile. Infatti per grandi programmi è utopistico aspettarsi che il codice venga prodotto senza errori.
Per raggiungere questo scopo non è opportuno far testare il codice allo stesso team che lo ha prodotto: si assegna di solito un team specifico per il test.

La fase si effettua a vari livelli e consiste in operazioni distinte.

- Test di unità. Un unità è il più piccolo blocco di software che ha senso collaudare, ad esempio una singola funzione di basso livello viene verificata come un'entità a se stante.
- Test di modulo. Un modulo è un insieme di unità interdipendenti.
- Test di sottosistema. Un sottosistema è un aggregato significativo di moduli spesso progettati da team diversi, e quindi vi possono essere problemi di interfaccia.
- Test di sistema o integrazione. Si tratta del test del prodotto completo.
- α-test. Se il sistema è sviluppato per un unico cliente, viene portato nel suo ambiente finale e collaudato con i dati sui quali dovrà normalmente operare.
- β-test. Se il sistema viene distribuito ad una comunità di utenti, viene dato in prova a più utenti, che lo utilizzano e forniscono al produttore le loro osservazioni ed il rilevamento degli errori riscontrati.
- Benchmark. Il sistema viene testato su dati standardizzati e di pubblico dominio per il confronto con altri prodotti equivalenti già presenti sul mercato. Può venire richiesto per contratto.

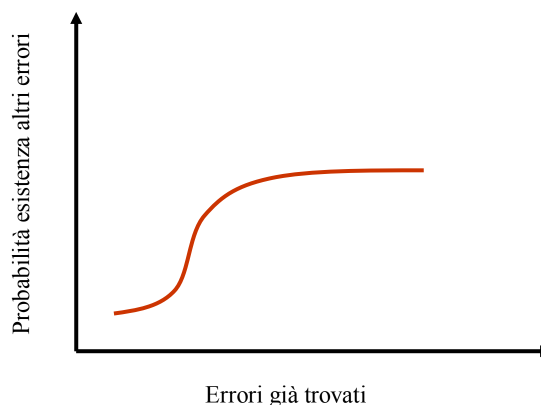
- Stress testing. Si verifica come si comporta il sistema quando è sovraccarico di lavoro, portandolo al limite. Questo test sotto sforzo permette di causare un errore (da stress) e verificare che il sistema fallisca in un modo accettabile (fail-soft).

Testing e debugging sono due concetti diversi.
Il Testing serve a rilevare la presenza di un errore.
Il Debugging consiste nella localizzazione dell'errore, della sua analisi e della sua correzione.
Terminato il debugging si fa seguire il test di regressione: si verifica il comportamento del modulo che è stato corretto nei confronti dei moduli con i quali coopera, assicurandosi così che la correzione dell'errore non ne abbia introdotti di nuovi.

4.2.1 Distribuzione degli errori

Esistono molti studi sulla distribuzione di errori (G. Myers: "The Art of Software Testing", John Wiley & Sons, 1979).
Gli errori tendono a concentrarsi in certi moduli, generalmente quelli più "complessi".
Un modulo che presenti molti errori va quindi verificato con attenzione, probabilmente ne conterrà altri: secondo alcuni studi statistici la probabilità che vi siano altri errori "latenti" cresce con il numero di errori già trovati secondo una curva di tipo "logistico".

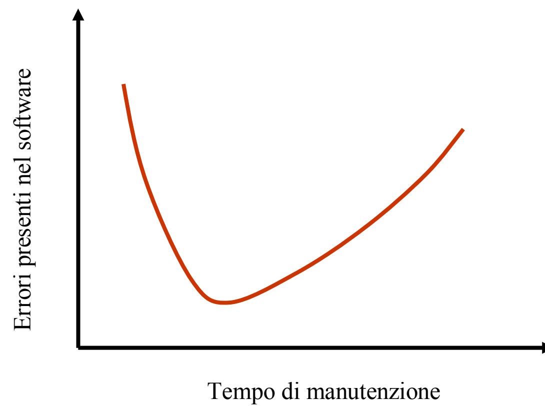
Distribuzione degli errori
Probabilità esistenza altri errori rispetto a quelli trovati



Nell'attività di manutenzione, sia rivolta a modifiche evolutive sia alla correzione di errori, si introducono nuovi errori.
Infatti molte delle decisioni prese inizialmente dal programmatore non vengono documentate, non sono evidenti dal codice, altre decisioni prese successivamente in fase di modifica tendono ad essere dimenticate da chi le ha fatte, oltre ad essere sconosciute agli altri.
Modificare una parte di codice scritto da altri, o semplicemente "datato", con lo scopo di eliminare errori, porta facilmente ad introdurre ulteriori errori.

La curva tipica degli errori in funzione del tempo di manutenzione tende spesso a seguire una curva di tipo quasi parabolico.

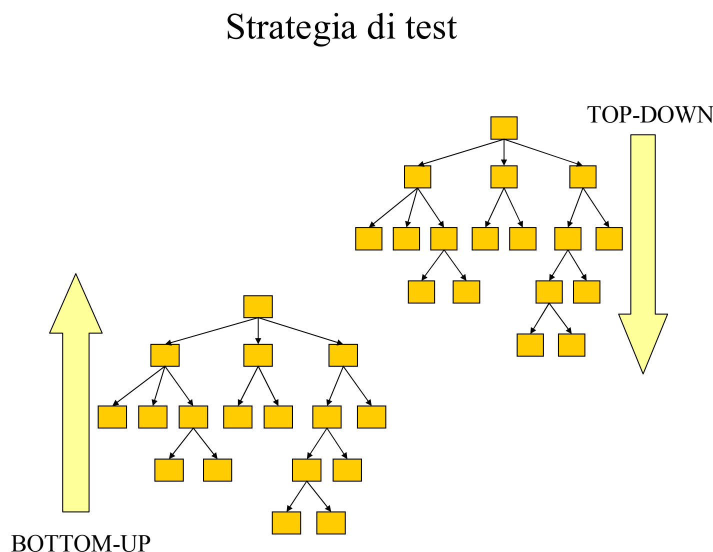
Distribuzione degli errori
Errori indotti dalla manutenzione



Inizialmente si ha una riduzione del numero di errori, al passare del tempo di manutenzione i continui interventi tendono ad introdurre nuovi errori.
Quando un modulo è soggetto a continua manutenzione va programmata una riprogettazione per ottimizzare manutenibilità ed estendibilità.

4.3 Strategia di test

Come la progettazione anche il test può essere effettuato top-down oppure bottom-up.



Con la strategia <u>bottom-up</u> vengono prima collaudati i moduli di più basso livello nella gerarchia prodotta dalla progettazione, si tratta cioè delle unità più piccole ed elementari che compongono il programma.
Quando sono valutati corretti si passa al livello superiore che viene testato utilizzando le funzionalità del precedente livello, si risale quindi così fino al sistema intero.
Si tratta di un approccio che richiede alcune complicazioni: generalmente sono necessari dei <u>driver</u>, cioè del software ausiliario che simula la chiamata ad un modulo generando dati da passare come parametri. I driver fanno le veci della parte di codice non ancora integrata nella porzione testata del programma: la parte alta della gerarchia dei moduli.
Vantaggi e svantaggi della strategia bottom-up.

- Il processo di test ed integrazione delle unità è intuitivamente più semplice da realizzare
- Più tardi viene scoperto un errore, più è costoso eliminarlo, in quanto richiede la sua correzione, solitamente con un modulo foglia e la ripetizione parziale del test fino ai moduli nei quali è stato trovato l’errore.

L’approccio <u>top-down</u> si comporta in modo diametralmente opposto. Inizialmente si testa il modulo corrispondente alla radice, senza utilizzare gli altri moduli del sistema. In loro vece vengono utilizzati dei <u>simulatori</u>, detti stub, si tratta di elementi che possono restituire risultati casuali, richiedere i dati al collaudatore, possono essere una versione semplificata del modulo.
Dopo aver testato il modulo radice della gerarchia si integrano i suoi figli diretti simulando i loro discendenti per mezzo di stub.

Si prosegue in questo modo fino a quando non vengono integrate tutte le foglie della gerarchia.
Vantaggi e svantaggi della strategia top-down.

- Vengono scoperti prima gli errori più costosi da eliminare. Ciò risulta particolarmente utile se la progettazione è anch'essa di tipo top-down e le fasi di progettazione, realizzazione e test sono parzialmente sovrapposte: un errore di progetto nella parte alta della gerarchia può essere scoperto e corretto prima che venga progettata la parte bassa
- In ogni istante è disponibile un sistema incompleto ma funzionante
- Gli stub possono essere costosi da realizzare

Qual è la migliore strategia di test?
Come già per la progettazione, viene solitamente adottata una soluzione di compromesso tra le due strategie, per attenuare gli inconvenienti che ognuna presenta.
Con entrambe le metodiche non è conveniente integrare ad ogni passo il maggior numero possibile di moduli.
Ad esempio in una strategia bottom-up testare i figli di un modulo e poi integrarli tutti assieme con il padre per testare questo, porta a codici non funzionanti senza una precisa idea di dove sia l'errore.
Risulta invece conveniente introdurre i moduli testati uno alla volta, si riesce in questo modo a localizzare meglio gli errori.

In generale si osserva che è più conveniente effettuare test frequenti e brevi.

4.4 Verifiche statiche.

Il punto critico del test è rappresentato dal costo: per contenerlo si cerca di migliorare il rapporto tra il costo e la potenza del test oppure affiancare il test con altre tecniche di verifica.
Una forma di verifica statica è l'ispezione del codice.
Permette di individuare tra il 60 % ed il 90 % degli errori che altrimenti si troverebbero solo durante il test con costi spesso più elevati.
L'ispezione può essere eseguita anche da un team specializzato, che disponga del documento delle specifiche, il cui compito è di segnalare ai programmatori gli errori, questi poi provvederanno a risolverli.

L'ambiente di codifica normalmente mette a disposizione degli strumenti automatici che analizzano il codice indicando gli errori (ad esempio la chiamata ad una funzione con un numero sbagliato di parametri).

4.5 Testing.

La sola ispezione del codice ovviamente non basta poiché non intercetta gli errori dovuti all'esecuzione del programma.
I moduli quindi vanno eseguiti su campioni di dati e si osserva il comportamento conseguente.
Il testing presenta due problemi:

- Non è mai esaustivo, non prova quindi la correttezza del codice
- È costoso, in termini di uso delle macchine e di tempo umano

Per effettuare il test è necessario mettere a punto un insieme di test-case ognuno comprendente:

- dati di input (test-data) per il modulo da testare

- descrizione della funzione interna al modulo
- output atteso dalla funzione

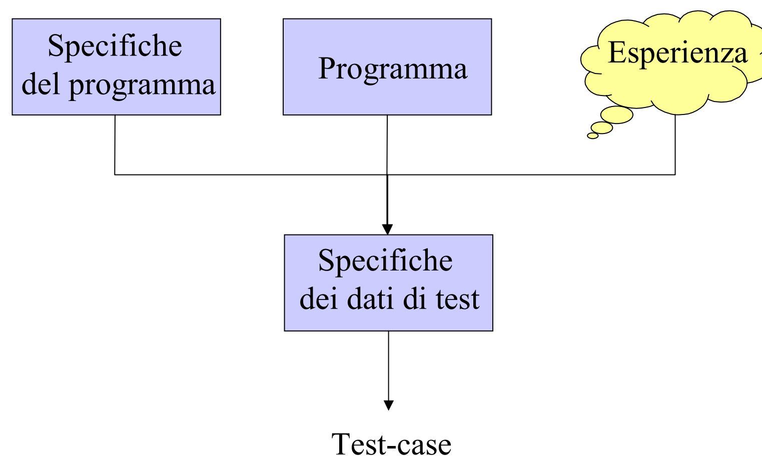
f
Un test-case rappresenta una caratteristica, un requisito, conformi alle specifiche ed è utilizzato come unità per costruire un test. Spesso è completato da un set di metadati che consentono di tracciare stato (approvato, rifiutato, modificato), contesto, collegamento alla specifica.

Come si costruisce un test-case?
Per costruire un test-case bisogna quindi conoscere l'output corretto per un certo input della funzione in esame: questo va ricavato dalle specifiche e dai documenti prodotti dalla progettazione.

Al termine dell'esecuzione dell'input, l'output ottenuto viene memorizzato assieme al test-case e viene confrontato con l'output atteso.

Il test deve essere ripetibile, per le fasi di test che si presentano in momenti successivi della vita del software.

Come si sceglie l'input?
La prima idea è di generare l'input casualmente.
Questa tecnica ha lo svantaggio di non essere molto uniforme per piccoli campioni: per avere una certa sicurezza bisogna generare molti test-data.
Sono necessari dei criteri migliori di scelta dell'input: il team dei collaudatori deve decidere le caratteristiche dei dati di test in base al programma, alle specifiche, all'esperienza.



Vi sono due modi principali di procedere:

- Testing a scatola nera (black-box testing), quando si ricavano le specifiche dei dati in input partendo solo dalle specifiche del programma, senza considerare il codice. Normalmente è utilizzato per i primi test ed ha il vantaggio di essere accessibile al cliente (che non vede il codice)
- Testing strutturale, quando si ricavano le specifiche dei dati di test guardando anche il codice. Generalmente è migliore del precedente.

4.5.1 Black-box testing.

La scelta dei dati di input si basa solamente sulle specifiche del programma, ad ogni funzione si fa corrispondere una descrizione più o meno formale dei suoi input corretti, dell'output che dovrebbe essere emesso e della funzione stessa.
Ad esempio una funzione che restituisce la data accetta in input il giorno, il mese nei rispettivi range di 1:31, 1:12.
L'idea del Black-box testing è di partizionare l'insieme degli input ammissibili in classi di equivalenza in modo che all'interno di ogni classe, ogni dato abbia lo stesso potere di testing.
L'esperienza determina i seguenti criteri di scelta:

- almeno un dato interno ad ogni classe
- tutti i dati di frontiera

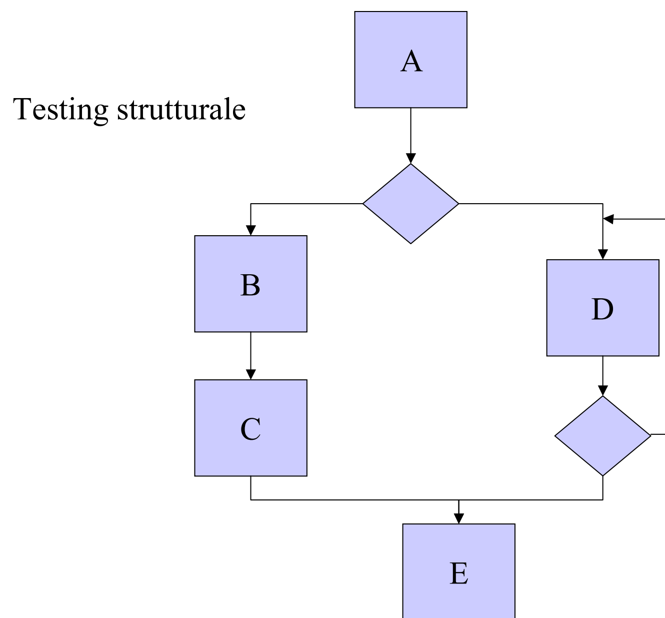
Alcuni criteri per determinare le classi si equivalenza:

- Per una variabile v definita in un intervallo a…..b avremo tre classi $v < a$; $a \leq v \leq b$; $v \geq b$
- Per le variabili intere vi è un'unica classe, l'esperienza suggerisce di utilizzare anche il valore 0 ed un valore negativo
- Per le stringhe alfanumeriche la determinazione delle classi e dei valori di frontiera dipendono dalle restrizioni sul formato della stringa. Ad esempio: la stringa vuota, quella contenente solo caratteri, quella che contiene dei numeri, ecc.

Un problema di questo metodo consiste nel fatto che la combinazione dei dati di test per tutte le componenti di input porta ad un numero enorme di test-case.

4.5.2 Testing strutturale.

Il testing strutturale viene condotto in base alla struttura del programma. Viene determinato il flusso di controllo del programma ed espresso sotto forma di flow-chart.



L'idea è di utilizzare i dati di test che portino a tutte le computazioni possibili.
Si cercano le combinazioni di dati per tutte le combinazioni di cammini nel flow-chart.
Nel caso di cicli si prendono almeno due test-data: uno che esegue il ciclo ed uno che non lo esegue.
Può essere convenientemente utilizzato nel test di Siti Web, applicato alle pagine statiche, erificando tutti i percorsi attraverso i link.

4.5.3 Strumenti automatici.

Il test può essere semplificato utilizzando strumenti di testing automatizzati.
Questi strumenti forniscono diversi supporti, fondamentali per la ripetibilità dei test:

- Gestione dei test-case
- Gestione dei test-data
- Memorizzazione dei test effettuati
- Reporting analitico

4.5.4 Simulatori.

Il software più difficile da testare riguarda situazioni di eccezione o che presentano pericolosità.

In questi casi vengono utilizzati dei simulatori dell'ambiente nel quale opererà il software.
Un simulatore è un programma ausiliario (che non verrà quindi utilizzato al termine del processo di sviluppo) che imita le azioni di un altro programma o di un comportamento hardware o di ambiente.
I simulatori vengono utilizzati per testare prodotti il cui malfunzionamento può provocare danni o catastrofi (ad esempio un programma per la gestione di un reattore nucleare).
I simulatori servono anche a provare il sistema in condizioni di carico particolare (stress-testing), difficilmente ottenibili se non in particolari condizioni dell'ambiente finale.

La costruzione di simulatori ovviamente incide notevolmente sul costo del prodotto finito.