

Copyright © 2004 Moreno Marzolla

This work is licensed under the Creative Commons Attribution-Noncommercial-Share Alike 2.5 Italy License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/2.5/it/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Verifica e Validazione Collaudo del software

- Garantire che il sistema software sia privo di errori e difetti

Crediti: E. Leonardi, Corso di Ingegneria del Software, Università di Ferrara, AA 2000/2001

Principi fondamentali

- Nelle prime fasi del processo di sviluppo software si passa da una visione astratta ad una implementazione concreta (implementazione)
- A questo punto occorre iniziare una serie di casi di prova destinati a “demolire” il software realizzato
- Il collaudo è l'unico passo del processo software che si può considerare (dal punto di vista psicologico) *distruttivo* anziché *costruttivo*
- Il collaudo deve infondere un senso di colpa? I collaudi sono distruttivi?
 - Ovviamente no!

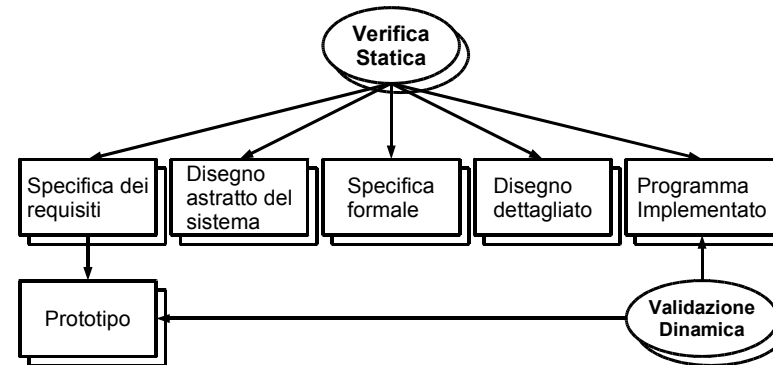
Verifica e Validazione

- Verifica:
 - “Stiamo costruendo il software in modo corretto?”
 - Il software deve essere conforme alle specifiche; cioè, deve comportarsi esattamente come era previsto
- Validazione
 - “Stiamo costruendo il giusto software?”
 - Il software deve essere implementato in modo da soddisfare le attese degli utenti e dei committenti
- La Verifica e Validazione ha due obiettivi principali
 - Scoprire i difetti presenti nel sistema (*verifica*)
 - Verificare se il sistema è o meno utilizzabile in condizioni operative (*validazione*)

Statica e dinamica

- *Verifica Statica* riguarda l'analisi di una rappresentazione statica del sistema per individuare difetti
 - Può essere utilizzata in ogni fase del processo di sviluppo, anche prima che il sistema sia implementato
- *Verifica e Validazione Dinamici* riguardano la fase di test con cui si esercita e osserva il comportamento dinamico del sistema
 - Richiede l'esistenza di un prototipo eseguibile del sistema, su cui effettuare i test
 - **La Validazione può essere solo dinamica:** come si può essere sicuri che il sistema soddisfi i requisiti del cliente semplicemente guardandone la struttura, e senza eseguirlo?

Verifica e Validazione



Test (collaudo) dei programmi

- Il collaudo consiste nell'eseguire un programma al fine di scoprire un errore
 - Rivela la presenza di errori, NON la loro assenza
- Un caso di prova è valido se ha un'alta probabilità di scoprire un errore ancora ignoto
- Un test (collaudo) è riuscito se ha scoperto un errore prima ignoto (non il contrario!)
- Requisiti non funzionali non possono essere verificati, solo validati
- Il test dei programmi deve essere usato assieme alla verifica statica

Principi / 1

- Ogni singola prova deve essere riconducibile ai requisiti del cliente
 - Dal punto di vista del cliente, i difetti più gravi sono quelli che impediscono al programma di soddisfare i requisiti
- I collaudi devono essere pianificati con largo anticipo
 - La pianificazione dei collaudi può iniziare non appena si è completata la definizione e specifica dei requisiti. La definizione dei casi di prova può cominciare appena il modello progettuale è stabile
- Il principio di Pareto si applica anche al collaudo del software
 - Principio di Pareto: l'80% degli errori scoperti è riconducibile al 20% dei componenti del programma. Il problema è identificare e isolare i componenti sospetti

Principi / 2

- I collaudi devono cominciare “in piccolo” e proseguire verso collaudi “in grande”
 - Le prime prove sono indirizzate alle singole componenti. Si passa poi a blocchi di componenti e all'intero sistema
- Un collaudo esauriente non è possibile
 - E' impossibile testare tutte le possibili esecuzioni del programma, perché anche sistemi di piccole dimensioni hanno un numero elevatissimo di percorsi d'esecuzione indipendenti
- Per essere efficace, il collaudo dovrebbe essere condotto da una parte indipendente
 - Non da chi ha scritto il codice

Collaudabilità / 1

- E' la facilità con cui un programma può essere provato. Collaudare un sistema è difficile. Cosa possiamo fare per agevolarlo?

Collaudabilità / 2

- **Operabilità** “Meglio funziona, meglio può essere collaudato”
 - il sistema contiene pochi errori
 - gli errori non impediscono l'esecuzione dei collaudi
 - il prodotto si evolve per stadi: sviluppo e collaudo assieme
- **Osservabilità** “Ciò che vedi è ciò che collaudi”
 - input e output sono chiaramente correlati
 - stati e variabili sono osservabili durante l'esecuzione
 - l'output dipende da fattori ben individuabili
 - l'output errato è di facile individuazione
 - errori interni vengono identificati automaticamente e riferiti

Collaudabilità / 3

- **Controllabilità** “Quanto più possiamo controllare il software, tanto più il collaudo può essere automatizzato e ottimizzato”
 - tutti i dati di output sono generabili mediante opportuno input
 - tutto il codice è eseguibile mediante opportuno input
- **Scomponibilità** “I moduli devono essere collaudabili separatamente”
 - il software è composto da moduli indipendenti
 - ogni modulo può essere collaudato da solo

Collaudabilità / 4

- **Semplicità** “Meno cose ci sono da collaudare, più velocemente si svolgono i collaudi”
 - semplicità funzionale: funzionalità ridotta al minimo necessario
 - semplicità strutturale: p.es. architettura modulare
 - semplicità del codice: standard unico di codifica
- **Stabilità** “Meno modifiche si apportano, meno situazioni devono essere collaudate”
 - le modifiche al software sono rare e controllate
 - le modifiche non inficiano i collaudi già svolti

Collaudabilità / 5

- **Comprensibilità** “Più informazioni abbiamo, più adeguati sono i collaudi che possiamo svolgere”
 - il progetto è chiaro
 - le dipendenze tra componenti sono chiare e ben descritte
 - le modifiche al progetto sono rese pubbliche
 - la documentazione tecnica è
 - facilmente accessibile
 - ben organizzata
 - specifica e dettagliata
 - accurata

Tipi di collaudo

- **Collaudo white-box**
 - Partendo dalla conoscenza della struttura interna del prodotto, il collaudo mira a verificare che le strutture interne funzionino secondo le specifiche progettuali testando al contempo tutti i componenti
- **Collaudo black-box**
 - Partendo dalle specifiche del progetto si va a verificare che tutte le funzionalità richieste nei requisiti siano presenti, cercando contemporaneamente eventuali errori

Collaudo white-box

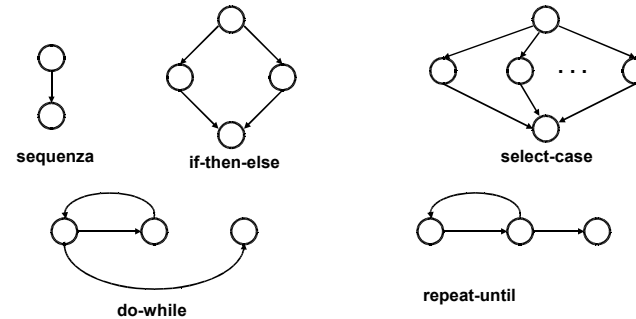
- I casi di prova vengono ricavati dalla struttura di controllo del progetto procedurale in modo da:
 - Garantire che tutti i cammini indipendenti dentro ciascun modulo siano eseguiti almeno una volta
 - Eseguire sia il ramo vero sia quello falso di ciascuna condizione
 - Eseguire tutti i cicli nei casi limite ed entro i confini operativi
 - Esaminare la validità di tutte le strutture dati interne

Collaudo black-box

- Il collaudo black-box complementa quello white-box occupandosi di rilevare errori dovuti a
 - funzionalità errate o mancanti
 - problemi di interfaccia
 - errori nelle strutture dati o nell'accesso a strutture dati esterne
 - comportamento erraneo o problemi prestazionali
 - errori nelle fasi d'inizializzazione o finalizzazione
- Contrariamente alle tecniche white-box, il collaudo black-box viene effettuato nelle fasi finali dell'implementazione
- Vanno individuati casi di prova che
 - siano significativi in termini di collaudo (riducano i test ulteriori)
 - individuino classi di errori piuttosto che errori singoli

I grafi di flusso

- I grafi di flusso sono una rappresentazione strutturale del programma che si concentra sui possibili flussi di esecuzione

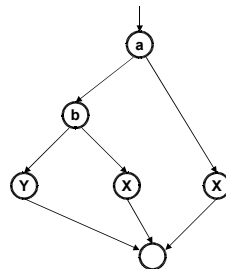


Condizioni logiche complesse nei grafi

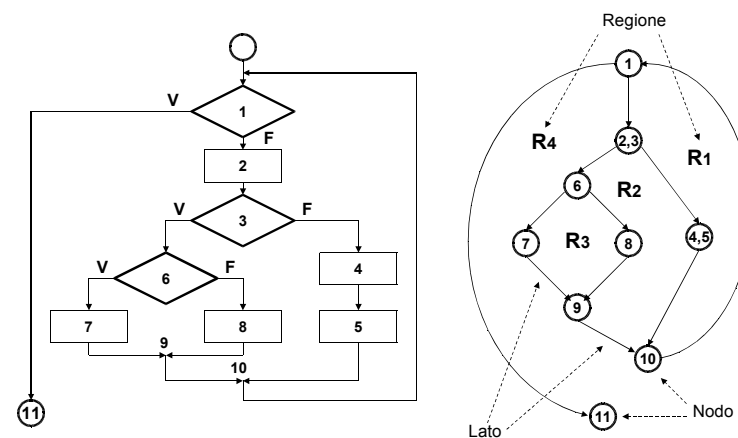
- Nel caso di condizioni logiche che includono uno o più operatori booleani la struttura dei grafi può essere più complessa

```

IF a OR b
  THEN Procedura X
  ELSE Procedura Y
ENDIF
    
```



Esempio di grafo di flusso



Complessità ciclomotica

- La complessità ciclomotica è una misura quantitativa della complessità logica di un programma
- Misura il numero di cammini indipendenti in un grafo
 - indipendente: che introduce almeno un lato non ancora esplorato
- Nell'esempio precedente si hanno 4 cammini indipendenti
 - 1) 1-11
 - 2) 1-2-3-4-5-10-1-11
 - 3) 1-2-3-6-8-9-10-1-11
 - 4) 1-2-3-6-7-9-10-1-11
- Collaudando i 4 cammini indipendenti abbiamo eseguito ogni istruzione e percorso ogni ramo logico del programma

Calcolo della complessità ciclomotica

- La complessità ciclomotica $V(G)$ si ricava in 3 modi

$$V(G) = R$$

dove R è il numero di regioni

$$V(G) = E - N + 2$$

dove E è il numero di lati e N il numero di nodi

$$V(G) = P + 1$$

dove P è il numero di nodi predicato (diramazioni)

N.B. in un select-case P è pari al numero di casi meno 1

- Nell'esempio avevo:
 - 4 regioni, quindi $V(G) = 4$
 - 9 nodi e 11 lati, quindi $V(G) = 11-9+2 = 4$
 - 3 nodi predicato, quindi $V(G) = 3+1 = 4$

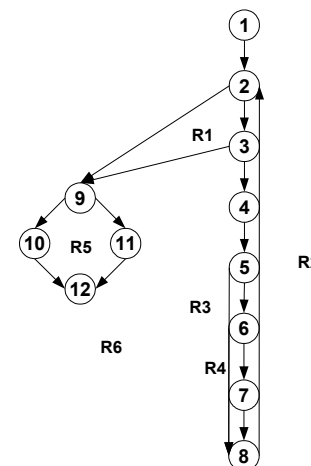
Esempio di derivazione dei casi di prova

```

INTERFACE ACCEPTS valore,minimo,massimo
INTERFACE RETURNS media,totale.valido
1 i = 1;
2 totale.input = totale.valido = 0;
3 somma = 0;
4 DO WHILE valore[i]<>-999 AND totale.input<100
5   incrementa totale.input di 1;
6   IF valore[i]>=minimo AND valore[i]<=massimo THEN
7     incrementa totale.valido di 1;
8     somma = somma+valore[i];
9   ENDDO
10  incrementa i di 1;
11 IF totale.valido>0 THEN
    media = somma/totale.valido;
ELSE
    media = -999;
ENDIF
    
```

Calcola la media di al più 100 numeri compresi tra i valori limite; calcola anche la somma e il numero totale valido

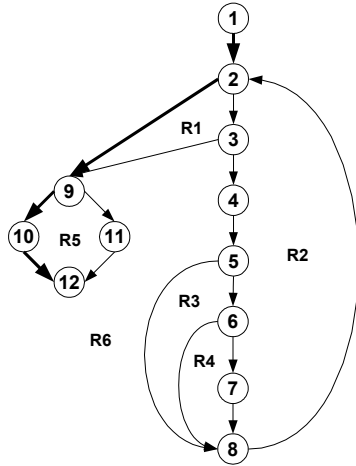
Grafo di flusso



- Cammini indipendenti

- 1) 1-2-9-10-12
- 2) 1-2-9-11-12
- 3) 1-2-3-9-10-12
- 4) 1-2-3-4-5-6-7-8-2-9-10-12
- 5) 1-2-3-4-5-8-2-9-10-12
- 6) 1-2-3-4-5-6-8-2-9-10-12

Grafo di flusso



- Cammini indipendenti

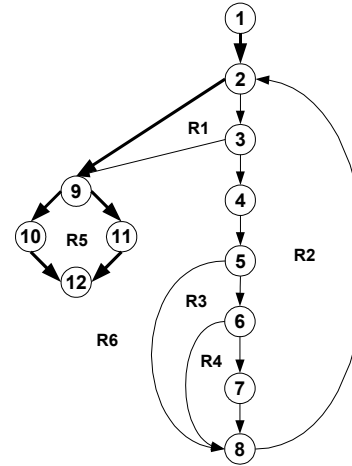
 - 1) 1-2-9-10-12
 - 2) 1-2-9-11-12
 - 3) 1-2-3-9-10-12
 - 4) 1-2-3-4-5-6-7-8-2-9-10-12
 - 5) 1-2-3-4-5-8-2-9-10-12
 - 6) 1-2-3-4-5-6-8-2-9-10-12

Moreno Marzolla

Ingegneria del Software

25

Grafo di flusso



- Cammini indipendenti

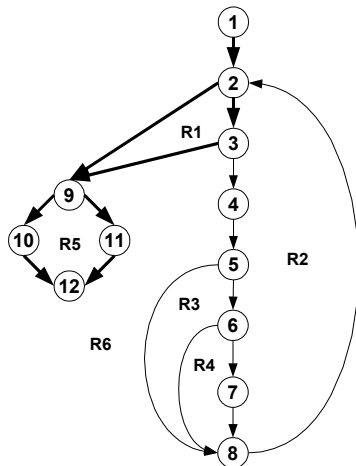
 - 1) 1-2-9-10-12
 - 2) 1-2-9-11-12
 - 3) 1-2-3-9-10-12
 - 4) 1-2-3-4-5-6-7-8-2-9-10-12
 - 5) 1-2-3-4-5-8-2-9-10-12
 - 6) 1-2-3-4-5-6-8-2-9-10-12

Moreno Marzolla

Ingegneria del Software

26

Grafo di flusso



- Cammini indipendenti

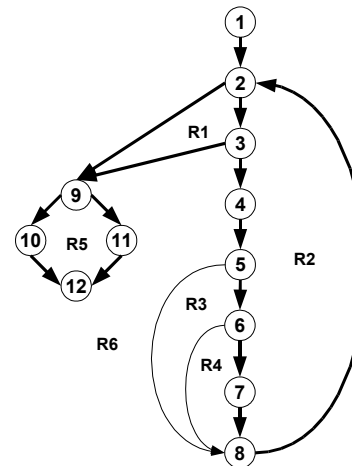
 - 1) 1-2-9-10-12
 - 2) 1-2-9-11-12
 - 3) 1-2-3-9-10-12
 - 4) 1-2-3-4-5-6-7-8-2-9-10-12
 - 5) 1-2-3-4-5-8-2-9-10-12
 - 6) 1-2-3-4-5-6-8-2-9-10-12

Moreno Marzolla

Ingegneria del Software

27

Grafo di flusso



- Cammini indipendenti

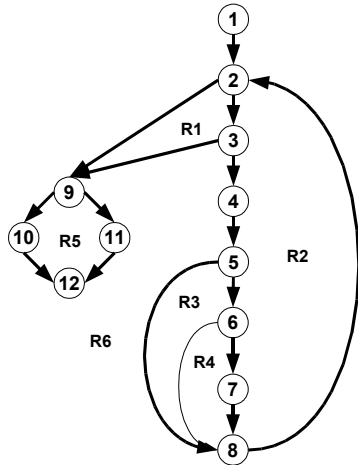
 - 1) 1-2-9-10-12
 - 2) 1-2-9-11-12
 - 3) 1-2-3-9-10-12
 - 4) 1-2-3-4-5-6-7-8-2-9-10-12
 - 5) 1-2-3-4-5-8-2-9-10-12
 - 6) 1-2-3-4-5-6-8-2-9-10-12

Moreno Marzolla

Ingegneria del Software

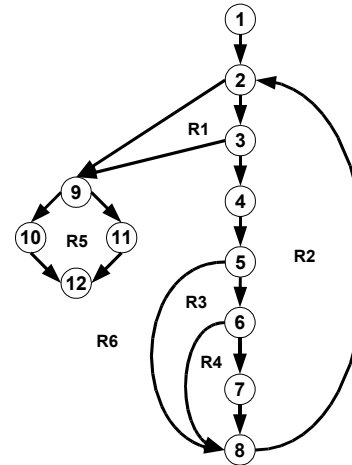
28

Grafo di flusso



- Cammini indipendenti
 - 1) 1-2-9-10-12
 - 2) 1-2-9-11-12
 - 3) 1-2-3-9-10-12
 - 4) 1-2-3-4-5-6-7-8-2-9-10-12
 - 5) 1-2-3-4-5-8-2-9-10-12
 - 6) 1-2-3-4-5-6-8-2-9-10-12

Grafo di flusso



- Cammini indipendenti
 - 1) 1-2-9-10-12
 - 2) 1-2-9-11-12
 - 3) 1-2-3-9-10-12
 - 4) 1-2-3-4-5-6-7-8-2-9-10-12
 - 5) 1-2-3-4-5-8-2-9-10-12
 - 6) 1-2-3-4-5-6-8-2-9-10-12

Scelta dei casi di prova

- A questo punto occorre scegliere i casi di prova in modo tale da verificare tutti i cammini individuati
 - Nota: alcuni cammini potrebbero esistere nel grafo di flusso, ma essere irrealizzabili come vere esecuzioni del programma
 - Es. il cammino 1
 - Questi cammini ovviamente si saltano (saranno inclusi in altri cammini testabili)

Collaudo per condizioni / 1

- Il collaudo per condizioni si concentra sui test di correttezza delle condizioni logiche del programma
- **Condizione logica semplice:** “ $E_1 <\text{operatore relazionale}> E_2$ ” dove $<\text{operatore relazionale}>$ può essere $<, <=, =, >=, >, !=$ ed E_1/E_2 sono espressioni aritmetiche
- **Condizione logica composta:** combinazione di condizioni logiche semplici collegate da operatori booleani OR (“|”), AND (“&”) e NOT (“-”) e organizzate con parentesi
- Se una condizione non comprende espressioni relazionali viene chiamata *espressione booleana*

Collaudo per condizioni / 2

- Errori in una condizione possono derivare da
 - errori negli operatori booleani (errati, mancanti o superflui)
 - errori in una variabile booleana
 - errori nella disposizione delle parentesi
 - errori in un operatore relazionale
 - errori in un'espressione aritmetica
- Questo tipo di metodi si propone di testare tutte le condizioni di un programma, assumendo che questo sia utile ad individuare errori anche nelle altre parti

Collaudo per cicli

- La tecnica del collaudo per cicli si concentra su prove che verifichino la validità dei costrutti di ciclo
- Cicli semplici (n=numero massimo di esecuzioni)
 - saltare il ciclo (1 prova)
 - percorrere il ciclo 0,1,2 volte (3 prove)
 - percorrere il ciclo m volte con $m < n$ (1 prova)
 - percorrere il ciclo n-1, n, n+1 volte (3 prove)
- Cicli annidati
 - fissare i cicli esterni ai valori minimi e testare quello più interno come un ciclo semplice
 - procedere verso i cicli esterni mantenendo quelli ancora più esterni al minimo e quelli più interni a un valore tipico

Testing e debugging

- Testing e debugging sono due attività distinte
- **Testing**: confermare la presenza di errori
- **Debugging**: localizzare e correggere tali errori

Strategia di collaudo

- La strategia di collaudo del software utilizza le tecniche di collaudo nell'ambito di una procedura di collaudo pianificata
- Si occupa di
 - definire il piano generale di collaudo
 - allocare le risorse necessarie al collaudo
 - progettare i casi di prova
 - definire le verifiche sui risultati dei casi di prova
 - raccogliere e valutare i risultati dei collaudi

Organizzazione dei collaudi

- Analisi e progettazione sono compiti intrinsecamente costruttivi mentre il collaudo è principalmente distruttivo
- Lo sviluppatore potrebbe essere psicologicamente condizionato nell'esecuzione dei collaudi
- Un gruppo indipendente può affiancare lo sviluppatore nei collaudi globali sul sistema
- Lo sviluppatore deve collaborare col gruppo di collaudo sia in fase di test, sia per correggere gli errori individuati nel corso dei collaudi

Quando finisce un collaudo?

- Risposta filosofica
 - *Il collaudo non termina mai, passa solo dalle mani dello sviluppatore a quelle del cliente*
- Risposta pragmatica
 - *Il collaudo finisce quando il budget economico e di tempo si è esaurito*
- Risposta quantitativa
 - *Il collaudo finisce quando la probabilità di verificarsi di un guasto software per unità di tempo d'uso scende sotto una soglia prefissata*
 - Es. richiedo che la probabilità di verificarsi di un guasto sia inferiore allo 0.5% per 1000 ore di funzionamento
 - Esistono metodi statistici per calcolare questa probabilità in funzione della misura del numero di guasti durante i collaudi

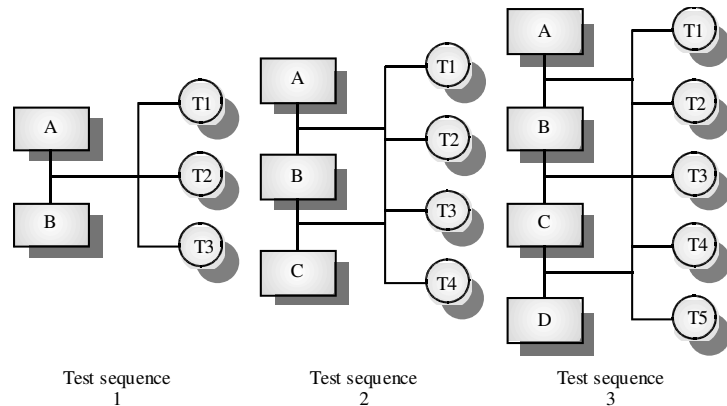
Criteri di collaudo

- Specificare quantitativamente i requisiti del prodotto prima di iniziare i collaudi
- Enunciare in maniera esplicita e quantitativa gli obiettivi del collaudo (copertura, MTBF finale, costi...)
- Sviluppare i profili di tutte le categorie di utenti
 - Porta alla limitazione dei casi d'uso da testare
- Sviluppare un piano basato su cicli rapidi
 - Iniziare dalle funzionalità principali
- Costruire software dotato di sistemi di autodiagnosi
- Definire metriche strategiche di collaudo

Strategie di testing

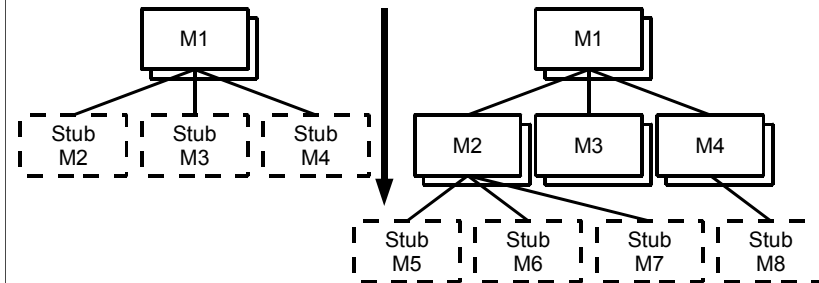
- Testing Top-down
 - I test cominciano dalle componenti più astratte, e ci si muove via via verso le componenti di basso livello
- Testing Bottom-up
 - I test cominciano dalle componenti più di basso livello, e ci si muove via via verso le componenti più astratte
- Collaudo per regressione
 - Usato per testare gli incrementi e le modifiche del software
- Stress testing
 - Utilizzato per testare come il sistema reagisce ai sovraccarichi
- Back-to-back testing
 - Utilizzato quando sono disponibili più versioni diverse dello stesso sistema

Test incrementali



Test top-down

- Primo passo: si testa il modulo radice, sostituendo quelli a livello inferiore con degli *stub*
- Secondo passo: si includono i moduli a livello 1, e si sostituiscono quelli a livello inferiore con degli *stub*

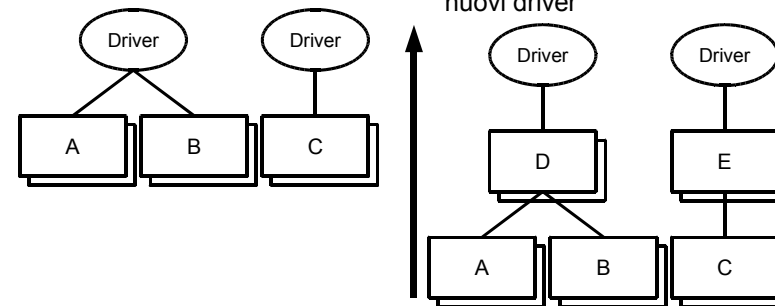


Test top-down

- Inizia dai livelli più elevati (radice) del sistema e procede verso il basso
- Questa strategia di test può essere facilmente usata in parallelo allo sviluppo top-down
- Può essere difficile individuare errore che dipendono dai moduli di più basso livello, perché verranno sviluppati per ultimi e gli stub passano dati non significativi ai livelli superiori
- Può individuare problemi architetturali sin dall'inizio
- Sviluppare stub può risultare difficoltoso

Test bottom-up

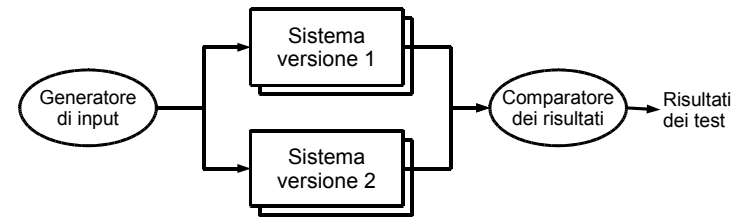
- Si parte dai moduli di più basso livello, sviluppando degli appositi *driver* che li usano e ne verificano i comportamenti
- Successivamente si continua a salire verso la radice della gerarchia, aggiungendo via via livelli e testandoli con nuovi driver



Test bottom-up

- Può essere utile per testare le componenti critiche di basso livello di un sistema
- Inizia dai livelli più bassi e si muove verso la radice delle componenti
- E' necessario implementare dei *test driver*
- Eventuali problemi di design sono scoperti relativamente tardi
- E' una tecnica appropriata per sistemi object-oriented (ma non solo)

Back-to-back testing



Back-to-back testing

- Utilizzare gli stessi casi di prova con versioni diverse del software, e confrontare i risultati. Output diversi segnalano possibili problemi
 - Gli output possono essere confrontati automaticamente, con conseguente riduzione dei costi di controllo dei risultati
 - Questa strategia richiede ovviamente due versioni diverse dello stesso sistema, che eventualmente possono anche essere due diversi prototipi

Collaudo per regressione

- Ad ogni introduzione di nuovi moduli il software cambia
 - Si creano nuovi cammini di flusso, vengono aggiunte nuove interfacce e vengono attivati nuovi blocchi logici
- Questo può introdurre problemi anche nelle parti collaudate prima dell'introduzione del nuovo modulo
- Il collaudo per regressione consiste nella ripetizione di prove già fatte per verificare che una modifica non abbia portato effetti collaterali
 - Questo tipo di prove va fatto ad ogni modifica del software (nuovo modulo, correzione di un bug, aggiornamento)
- L'uso di strumenti automatici consente di registrare alcune prove chiave per poi riprodurle in fase di collaudo di regressione

Approccio “smoke-testing”

- Se i tempi di consegna devono essere brevi è possibile inserire il collaudo nel corso dello sviluppo del codice (smoke testing)
- I moduli software via via prodotti vengono integrati in build che implementano una o più delle funzioni del programma
- Per ogni build vengono identificati dei test chiave che ne verificano il corretto funzionamento
- Tutti i build vengono integrati e il prodotto risultante viene testato quotidianamente
- Se ad un test si individuano degli errori prima inesistenti, il maggior indiziato è il codice aggiunto per ultimo
- Questo metodo consente di avere una misura continuamente aggiornata dei progressi nella realizzazione del prodotto software

Vantaggi dell'approccio “smoke test”

- Minimizza i rischi di integrazione
 - Uno dei rischi maggiori cui grossi progetti vanno incontro risiede nelle difficoltà inaspettate che si scoprono quando diverse componenti sono integrate. Qui l'integrazione viene fatta quotidianamente, e i problemi si scoprono subito
- Riduce i rischi di produrre codice di bassa qualità
 - Questo approccio fa in modo che il caos non prenda il sopravvento
- E' facile diagnosticare i problemi
 - Se il sistema viene compilato e testato ogni giorno si conosce subito quali sono i problemi in un dato giorno
- Migliora il morale degli sviluppatori
 - Vedere che il prodotto che si sta costruendo funziona effettivamente contribuisce a migliorare il morale (e quindi la produttività) degli sviluppatori

Come funziona

- “Build daily”
- Controlla se il build fallisce
- Effettua quotidianamente lo “smoke test”
- Aggiorna i pacchetti nel build solo quando serve
 - Lo sviluppatore mette nel build una versione aggiornata dei suoi pacchetti solo quando ci sono modifiche sufficienti
- Chi fa fallire il build deve essere penalizzato
 - L'idea è che un build fallito deve essere l'eccezione, non la norma
- Effettua i build e i test anche durante periodi critici
 - Quando le scadenze incombono, ecc.

Stress testing

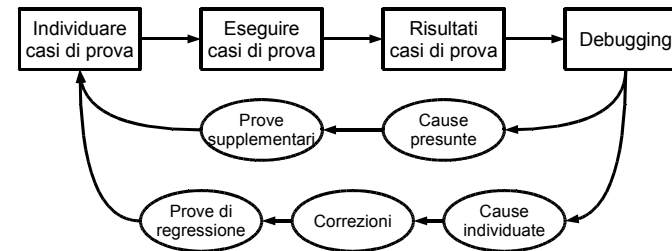
- Le prove di stress forzano il sistema a richiedere risorse in quantità, frequenza o volumi eccessivi
 - Generare 10 richieste al secondo a fronte di una media stimata di 2
 - Progettare casi di utilizzo che richiedono il massimo della memoria e il massimo di altre risorse possibili
 - ...
- In applicazioni matematiche si usano le “prove di sensibilità”
 - Si testa il programma utilizzando valori dei parametri molto vicini ai limiti di validità per l'algoritmo, per vedere se si verificano situazioni aberranti o risultati del tutto errati

Prove di validazione

- I criteri di validazione sono contenuti nelle specifiche dei requisiti e descrive quelle che sono le ragionevoli aspettative dell'utente
 - Un criterio importante di validazione è la verifica che la configurazione software sia opportunamente catalogata e documentata in modo da garantire il supporto futuro
- Il metodo migliore consiste nel far usare il prodotto ad un insieme ridotto di utenti per verificarne il comportamento "sul campo"
 - Collaudo alfa: gli utenti usano il prodotto sotto il diretto controllo degli sviluppatori
 - Collaudo beta: gli utenti usano il software per proprio conto e redigono un report con gli errori e i problemi individuati

Il debugging / 1

- Il debugging punta all'eliminazione degli errori individuati durante il collaudo del software



Il debugging / 2

- I motivi per cui trovare le cause di errore è difficile sono molti
 - distanza tra causa e punto in cui l'errore appare (accoppiamento)
 - non c'è una causa specifica (es. arrotondamento)
 - il sintomo dipende da un errore umano non individuato
 - il sintomo dipende da problemi di temporizzazione e non di elaborazione
 - è difficile riprodurre l'input che ha causato l'errore (real-time)
 - l'errore è intermittente (forte legame hardware-software)
 - il sintomo dipende da molte cause distribuite su vari processi
- La correzione di un errore può spesso provocarne di nuovi: applicare sempre il collaudo di regressione

Il debugging / 3

- La capacità di debugging varia molto con gli individui e dipende da fattori psicologici (ansia, rifiuto)
- I metodi principali di debugging sono
 - La forza bruta: dump della memoria, messaggi di debug, ... da tenersi come *ultima ratio*
 - Il cammino a ritroso: si parte dal punto in cui l'errore si è manifestato e si procede a ritroso, programmi piccoli
 - L'eliminazione delle cause: metodo scientifico in cui si formulano ipotesi e si inventano test per verificarle
- Diversi strumenti aiutano nelle fasi di debug: compilatori con controllo degli errori, debugger, ...
- Quando non sai più che fare, chiedi aiuto agli altri!
 - Legge di Linus: *Given enough eyeball, all bugs are shallow*