Verifica e Validazione

Alberto Gianoli

Univ. Ferrara - Corso di Laurea in Informatica

Dove?

- Pressman cap. 15 e 16
- Sommerville qualcosa nei cap. 22 e 23

Di cosa si tratta?

- Vogliamo garantire che il sistema software sia privo di errori e difetti
- Nelle prime fasi del processo di sviluppo software si passa da una visione astratta ad una implementazione concreta (implementazione)
- A questo punto bisogna iniziare una serie di casi di prova destinati a "demolire" il software realizzato
- Il collaudo è l'unico passo del processo di produzione che si può considerare (dal punto di vista psicologico) distruttivo invece che costruttivo
- Non bisogna farsi dei sensi di colpa...

Di cosa si tratta?

Esistono varie strategie di testing, in generale hanno tutte queste caratteristiche

- Fate delle technical review: in questo modo molti errori sono eliminati prima ancora di arrivare a fare i test
- I test cominciano a “component level” e si vanno ingrandendo fino a riguardare l’integrazione dell’intero sistema
- Tecniche di testing diverse sono appropriate a modi diversi di fare il testing e a tempi diversi nel ciclo del progetto
- Il testing e’ fatto dagli sviluppatori e (per progetti grandi) da un gruppo indipendente di testing
- Testing ≠ debugging: ma una strategia di testing prevede il debugging

Verifica & Validazione

Controllo di qualità delle attività svolte durante una fase dello sviluppo

- **Verifica: stiamo costruendo il prodotto nel modo giusto?**
  - il sw deve essere conforme alle specifiche, cioè deve comportarsi esattamente come era previsto (e voluto)
- **Validazione: stiamo costruendo il prodotto giusto?**
  - il sw deve essere implementato in modo da soddisfare quello che l'utente realmente richiede

Controllo di qualità del prodotto rispetto ai desiderata del committente

- Sono domande che devono percorrere tutto il ciclo di vita del software, per correggere errori e per valutare se il prodotto è usabile dal punto di vista operativo

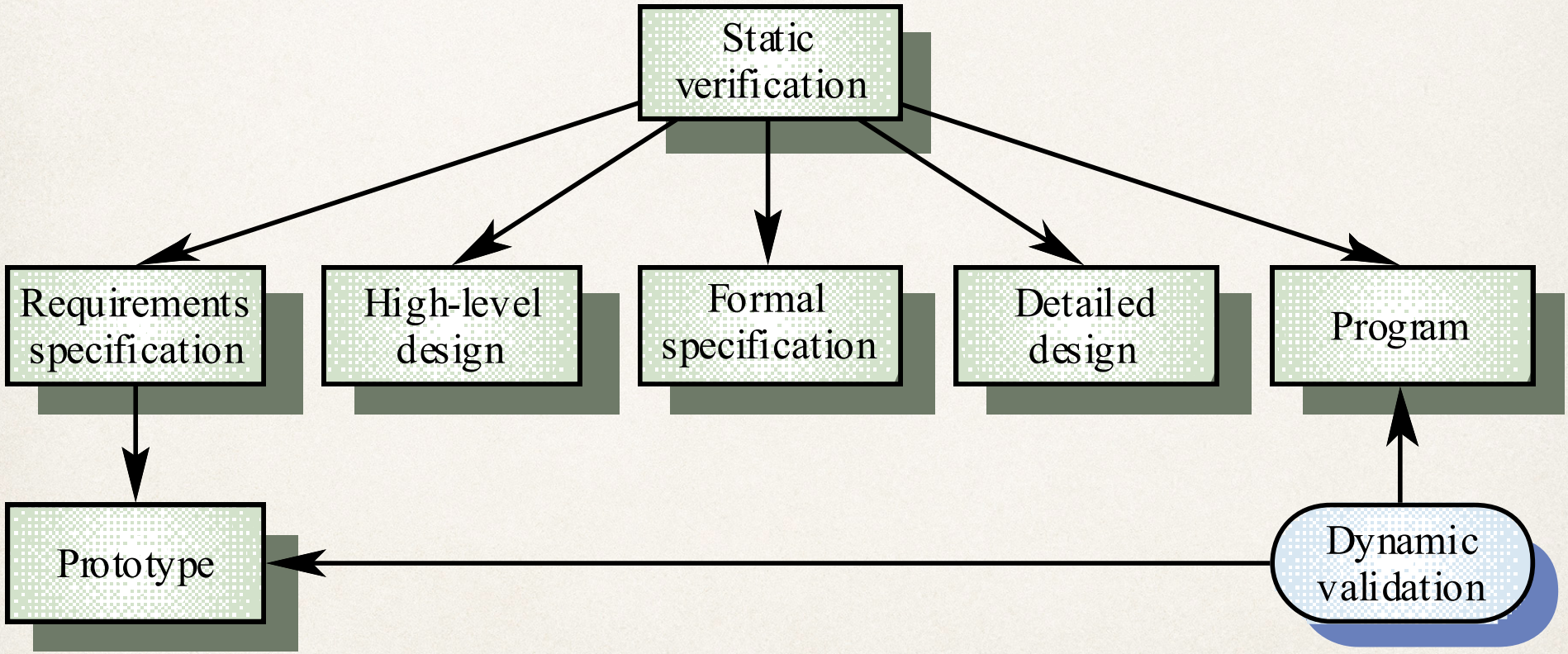
Verifica & Validazione

- V & V deve stabilire con un buon grado di confidenza che il software è adeguato allo scopo del progetto
- Questo NON significa completamente privo di difetti
- Piuttosto, che è sufficientemente buono per lo scopo previsto, e il tipo confidenza richiesto dipende dalla funzione, dalle aspettative degli utenti, dall'attuale ambiente di mercato del sistema

Verifica statica/dinamica

- ***Verifica statica*** riguarda l'analisi di una rappresentazione statica del sistema per individuare i problemi
  - si può fare in ogni fase del processo di sviluppo, anche prima che il sistema sia implementato
- ***Verifica e validazione dinamici*** consiste nel testare il prodotto facendo prove e osservandone il comportamento
  - richiede l'esistenza di un prototipo eseguibile del sistema su cui effettuare i test
  - **la validazione può essere solo dinamica**: non si può essere sicuri che un sistema soddisfi i requisiti del cliente semplicemente guardando la struttura, senza eseguirlo

V & V statica e dinamica



Testing

- Il collaudo consiste nell'eseguire un programma al fine di scoprire un errore
  - N.B.: riveliamo la presenza di errori, non la loro assenza! (non possiamo avere la certezza di non avere errori mediante il collaudo)
  - questo non vuol dire che sia una tecnica di verifica inutile
- 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
- Requisiti non funzionali non possono essere verificati, solo validati
- Il test dei programmi va usato assieme alla verifica statica

Principi

- Ogni singola prova deve essere riconducibile ai requisiti del cliente
  - nell'ottica del cliente, i difetti più gravi sono quelle che impediscono al programma di soddisfare i requisiti
- I collaudi vanno pianificati con largo anticipo
  - la pianificazione dei collaudi può iniziare appena è completata la definizione e la specifica dei requisiti; la definizione dei casi di prova può cominciare non 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

- I collaudi devono cominciare in piccolo, e proseguire verso collaudi in grande
  - inizialmente si testano le singole componenti, poi si passa a blocchi di componenti e si risale fino all'intero sistema
- Un collaudo esauriente spesso è impossibile
  - impossibile provare tutte le possibili esecuzioni del programma: già sistemi piccoli hanno un numero elevato di percorsi di esecuzione
- Per essere efficace il collaudo andrebbe fatto da terze parti, non da chi ha scritto il codice

Collaudabilità

- Con questo termine intendiamo la facilità con cui un sistema può essere provato. Collaudare un sistema è difficile: cosa possiamo fare per facilitare il compito?

Collaudabilità

- **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

- **Controllabilità**: quanto più possiamo controllare il sw, 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 collaudati separatamente
  - il software è composto da moduli indipendenti
  - ogni modulo può essere collaudato da solo

Collaudabilità

- **Semplicità**: meno cose ci sono da collaudare, più velocemente si svolgono i collaudi
  - semplicità funzionale: funzionalità ridotta al minimo necessario
  - semplicità strutturale: 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 svolte

Collaudabilità

- **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**
  - si parte dalla conoscenza della struttura interna del prodotto: il collaudo vuole verificare che le strutture interne funzionino secondo le specifiche progettuali testando al contempo tutti i componenti
    - controllo meticoloso degli aspetti procedurali
- **Collaudo black-box**
  - si parte dalle specifiche del progetto e si va a verificare che tutte le funzionalità richieste nei requisiti siano presenti, cercando allo stesso tempo eventuali errori
    - non mi interessa la logica interna, svolge le sue funzioni?

Tipi di collaudo

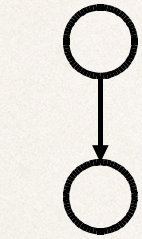
- A prima vista white box approfondito dovrebbe coprire tutto, ma è sempre possibile?
  - programma C di 100 righe
  - due cicli annidati, eseguiti da 1 a 20 volte (dipende da input)
  - 4 costrutti if-then-else nel ciclo interno
  - ➡ totale cammini possibili ~10^{14}

Collaudo white-box

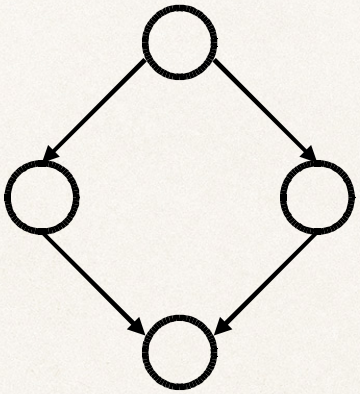
- I casi di prova si ricavano dalla struttura di controllo del progetto procedurale in modo da
  - garantire che tutti i cammini indipendenti dentro ciascun modulo siano eseguiti almeno una volta
  - eseguire sia il ramo vero che quello falso di ciascuna condizione
  - eseguire tutti i cicli nei casi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

I grafi di flusso

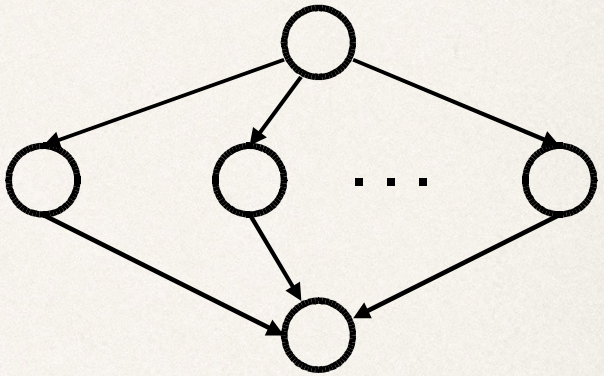
- Sono una rappresentazione strutturale del programma che si focalizza sui possibili flussi di esecuzione



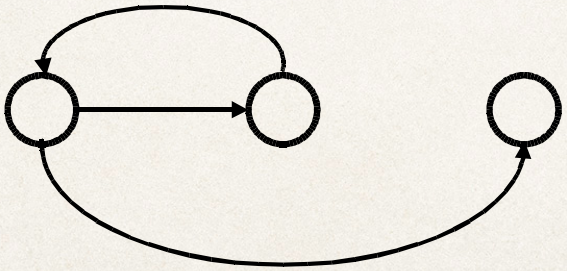
sequenza



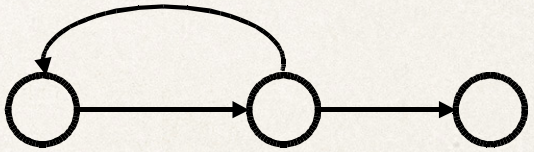
if-then-else



select-case



do-while

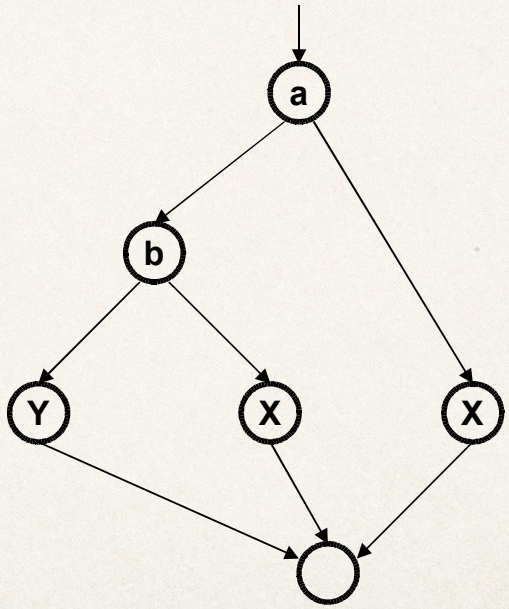


repeat-until

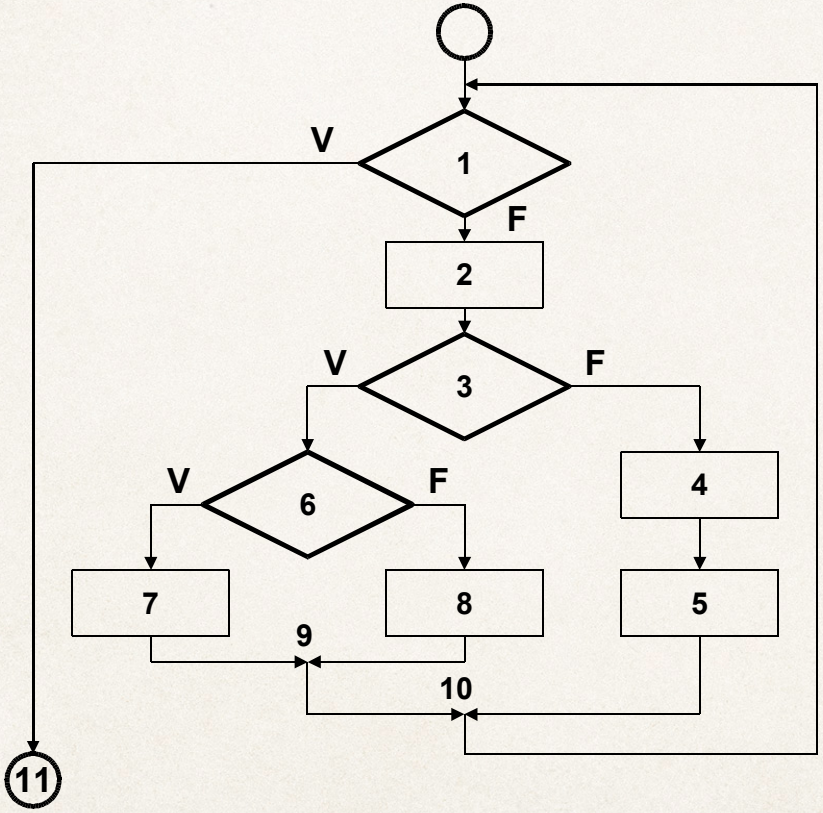
Condizioni logiche complesse nei grafi

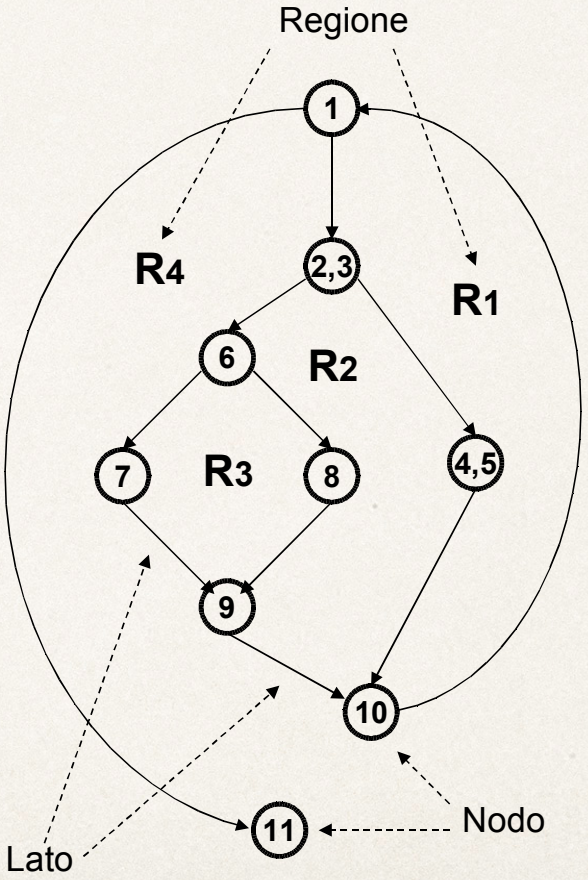
- Una condizione logica può includere più di un operatore booleano: questo complica un po' la situazione ma è comunque rappresentabile

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



Esempio


V
1
F
2
V
3
F
V
6
F
4
7
8
5
9
10
11


Regione
1
R4
2,3
R1
6
R2
7
R3
8
4,5
9
10
11
Nodo
Lato

Complessità ciclomatica

- **Complessità ciclomatica**: misura quantitativa della complessità logica di un programma
- Misura il numero di cammini *indipendenti* in un grafo
  - *indipendente*: che introduce almeno un lato (cioè una parte di percorso) non ancora esplorato o una nuova condizione
- Prendiamo l'esempio precedente: abbiamo 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
- Collaudiamo i 4 cammini indipendenti: abbiamo eseguito ogni istruzione e percorso ogni ramo logico del programma

Calcolo della complessità ciclomatica

- Esistono 3 modi equivalenti per calcolare la complessità ciclomatica di un grafo V(G)
  - V(G) = R dove R è il numero delle 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 diramazioni
    - ➡ **N.B.**: in un select-case P è uguale al numero di casi meno 1
- Nell'esempio avevamo
  - 4 regioni V(G)=4
  - 9 nodi e 11 lati V(G) = 11 - 9 + 2 = 4
  - 3 diramazioni V(G) = 3 + 1 = 4

Esempio di derivazione dal codice

- Il metodo visto si applica sia al design procedurale sia al codice sorgente.
- Supponiamo di partire da una procedura linguaggio procedurale

1. Dobbiamo tracciare il flow graph corrispondente
2. Calcoliamo la complessità ciclomatica
3. Determiniamo una base di cammini linearmente indipendenti
4. Prepariamo dei test case che portino all'esecuzione di tutti i cammini della base

Esempio di derivazione dal codice

```
INTERFACE ACCEPTS valore, minimo, massimo
INTERFACE RETURNS media, totale.valido
i = 1;
totale.input = totale.valido = 0;
somma = 0;
WHILE ((valore[i]!= -999) && (totale.input<100)) {
  totale.input++;
  IF((valore[i]>= minimo) && (valore[i]<= massimo)) {
    totale.valido++;
    somma += valore[i];
  }
  i++;
};
IF(totale.valido>0)
  media = somma/totale.valido;
ELSE
  media = -999;
```

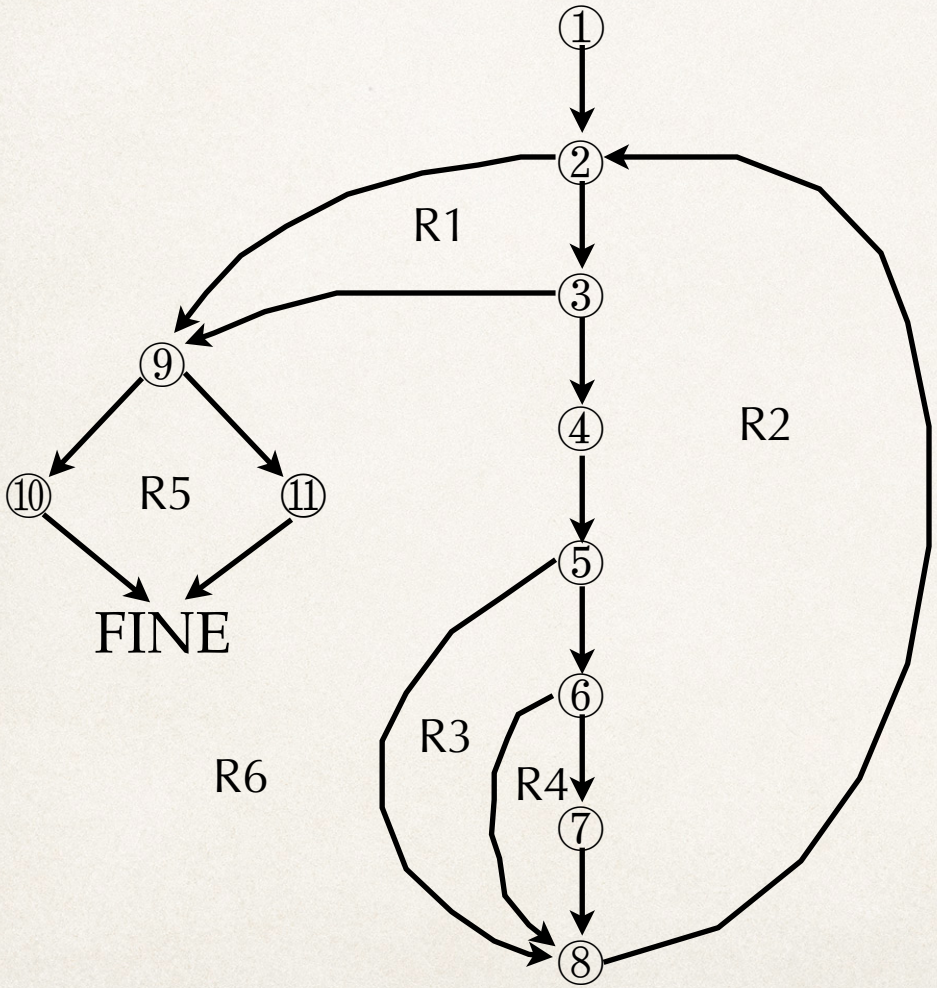
calcola la media di al più 100 numeri compresi tra i valori "minimo" e "massimo", restituendo la media e il numero di valori nell'intervallo

Esempio di derivazione dal codice

```
INTERFACE ACCEPTS valore, minimo, massimo
INTERFACE RETURNS media, totale.valido
i = 1;                                                    ①
totale.input = totale.valido = 0;                         ②
somma = 0;
WHILE ((valore[i]!= -999) && (totale.input<100)) {        ③
  totale.input++;                                         ④
  IF((valore[i]>= minimo) && (valore[i]<= massimo)) {     ⑤ ⑥
    totale.valido++;                                      ⑦
    somma += valore[i];
  }
  i++;                                                    ⑧
};
IF(totale.valido>0)                                       ⑨
  media = somma/totale.valido;                            ⑩
ELSE
  media = -999;                                           ⑪
```

calcola la media di al più 100 numeri compresi tra i valori "minimo" e "massimo", restituendo la media e il numero di valori nell'intervallo

Esempio di derivazione dal codice



Cammini indipendenti

(1) 1-2-9-10-FINE

(2) 1-2-9-11-FINE

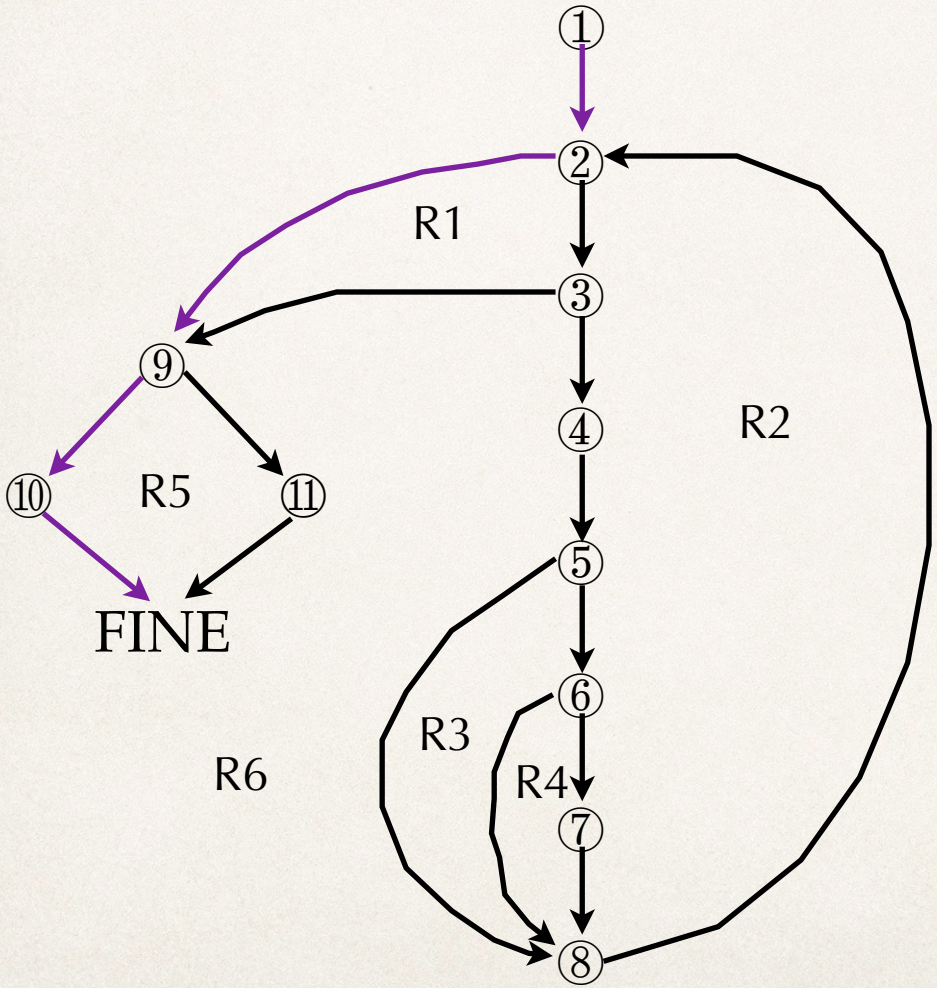
(3) 1-2-3-9-10-FINE

(4) 1-2-3-4-5-6-7-8-2-...-FINE

(5) 1-2-3-4-5-8-2-...-FINE

(6) 1-2-3-4-5-6-8-2-...-FINE

Esempio di derivazione dal codice



Cammini indipendenti

(1) 1-2-9-10-FINE

(2) 1-2-9-11-FINE

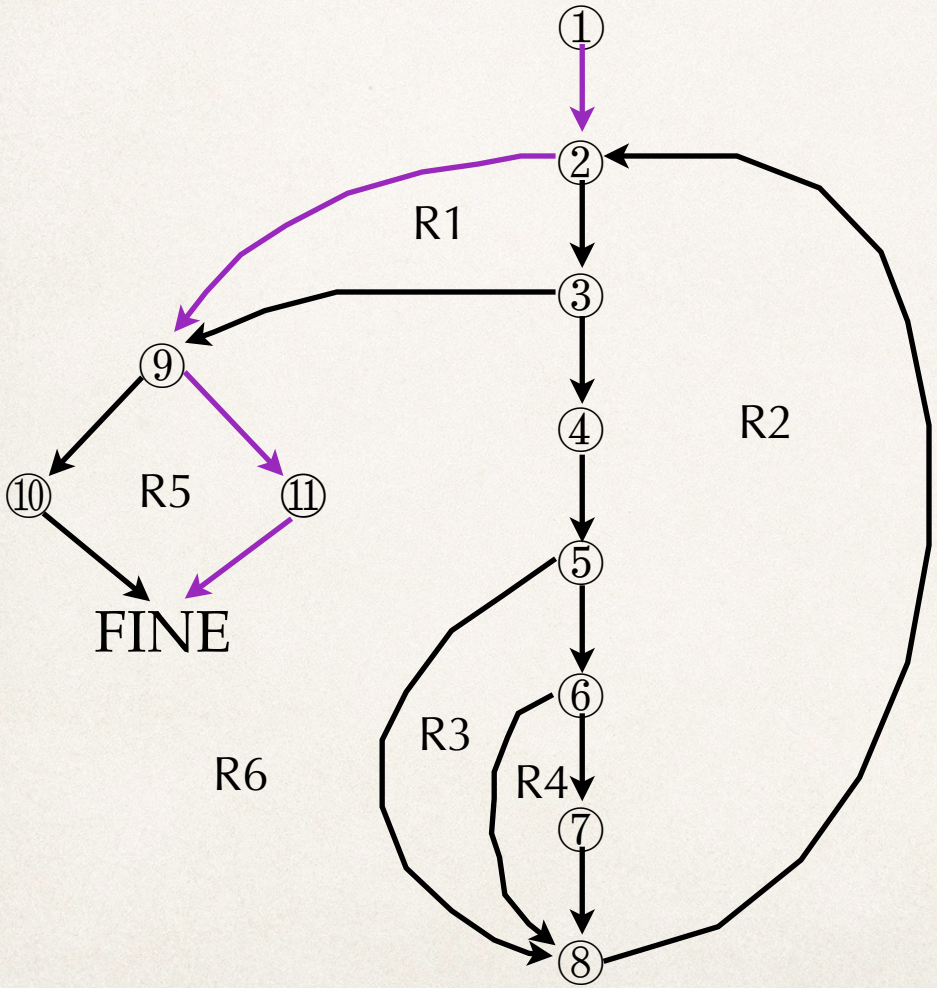
(3) 1-2-3-9-10-FINE

(4) 1-2-3-4-5-6-7-8-2-...-FINE

(5) 1-2-3-4-5-8-2-...-FINE

(6) 1-2-3-4-5-6-8-2...-FINE

Esempio di derivazione dal codice



Cammini indipendenti

(1) 1-2-9-10-FINE

(2) 1-2-9-11-FINE

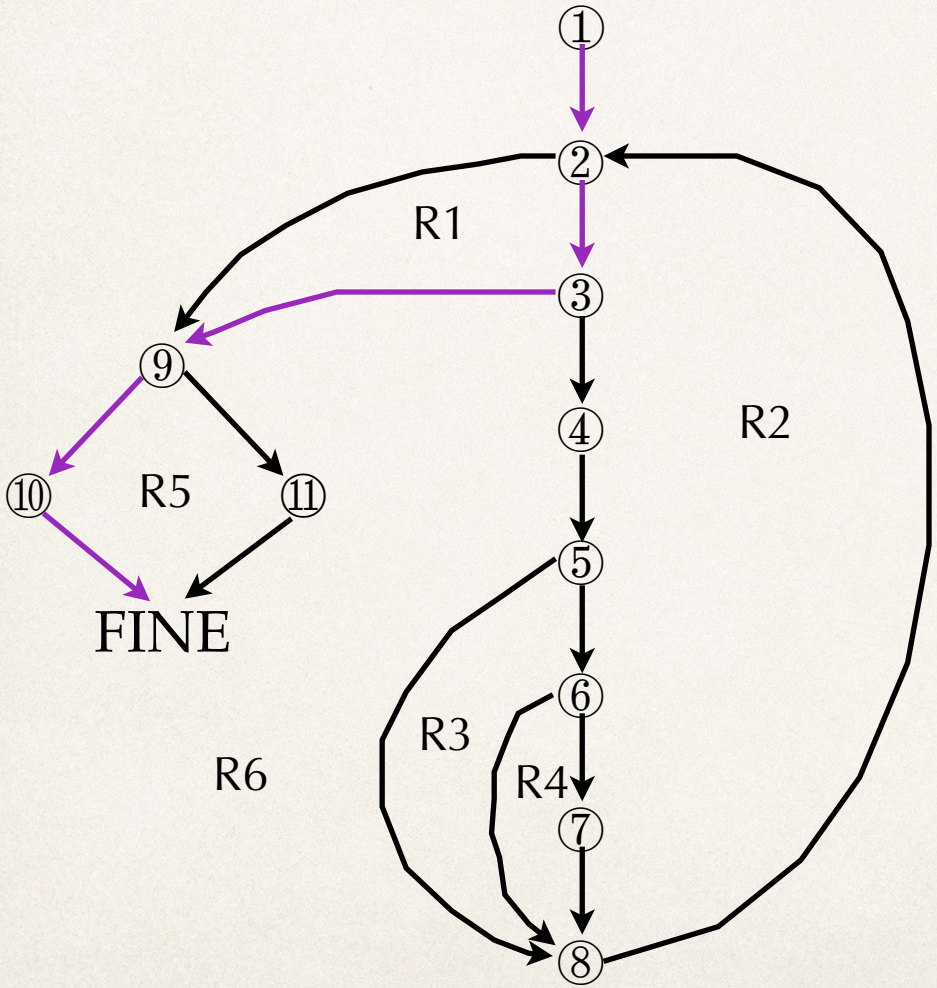
(3) 1-2-3-9-10-FINE

(4) 1-2-3-4-5-6-7-8-2-...-FINE

(5) 1-2-3-4-5-8-2-...-FINE

(6) 1-2-3-4-5-6-8-2-...-FINE

Esempio di derivazione dal codice



Cammini indipendenti

(1) 1-2-9-10-FINE

(2) 1-2-9-11-FINE

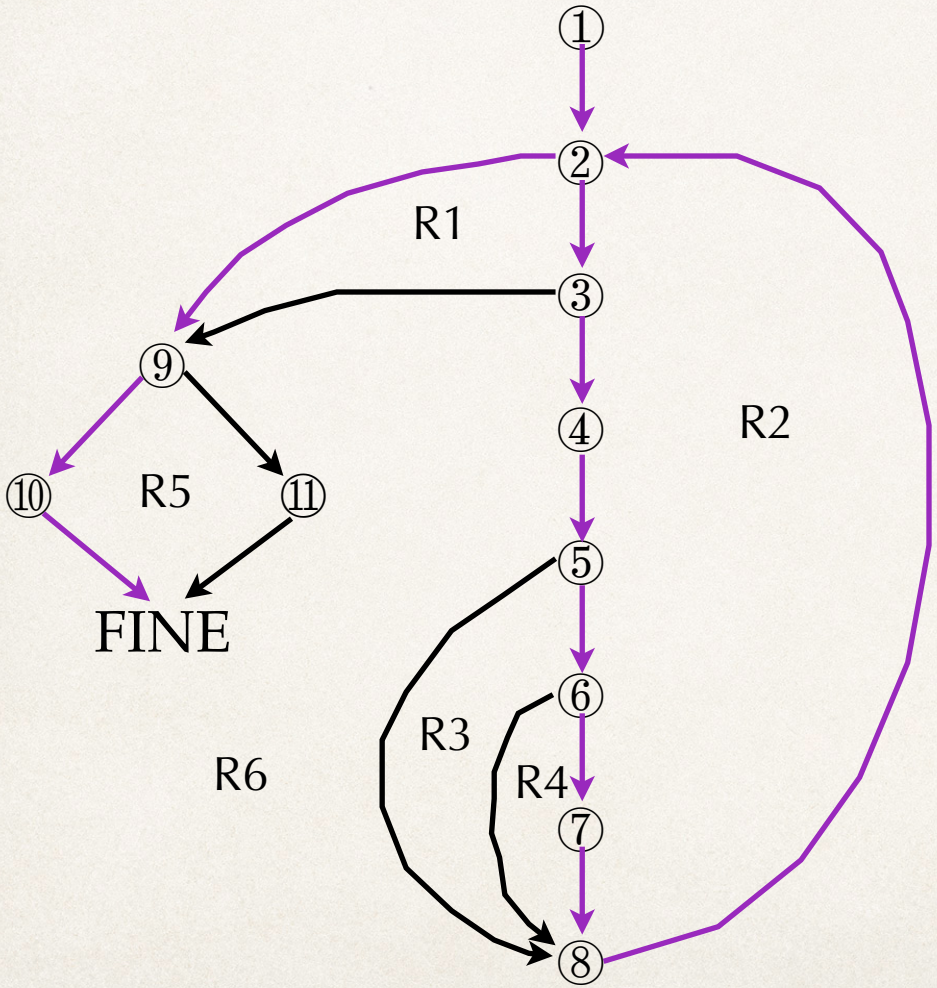
(3) 1-2-3-9-10-FINE

(4) 1-2-3-4-5-6-7-8-2-...-FINE

(5) 1-2-3-4-5-8-2-...-FINE

(6) 1-2-3-4-5-6-8-2-...-FINE

Esempio di derivazione dal codice



Cammini indipendenti

(1) 1-2-9-10-FINE

(2) 1-2-9-11-FINE

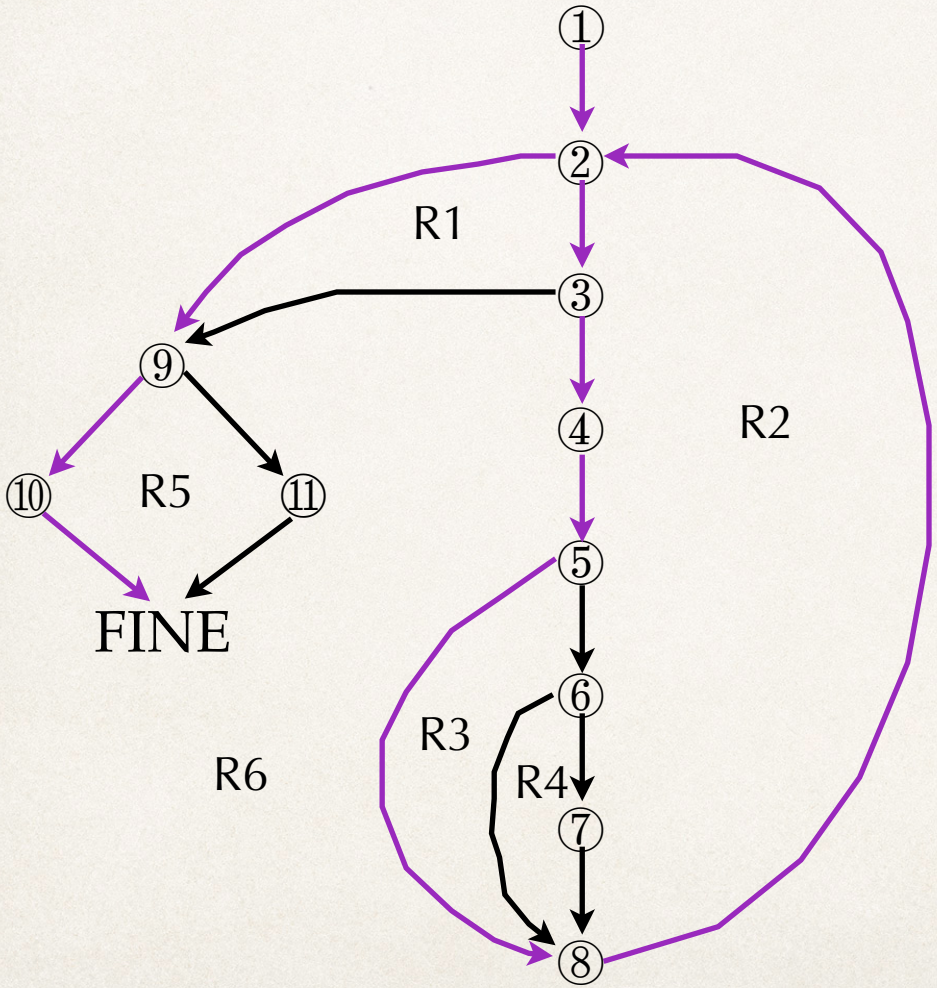
(3) 1-2-3-9-10-FINE

(4) 1-2-3-4-5-6-7-8-2-...-FINE

(5) 1-2-3-4-5-8-2-...-FINE

(6) 1-2-3-4-5-6-8-2-...-FINE

Esempio di derivazione dal codice



Cammini indipendenti

(1) 1-2-9-10-FINE

(2) 1-2-9-11-FINE

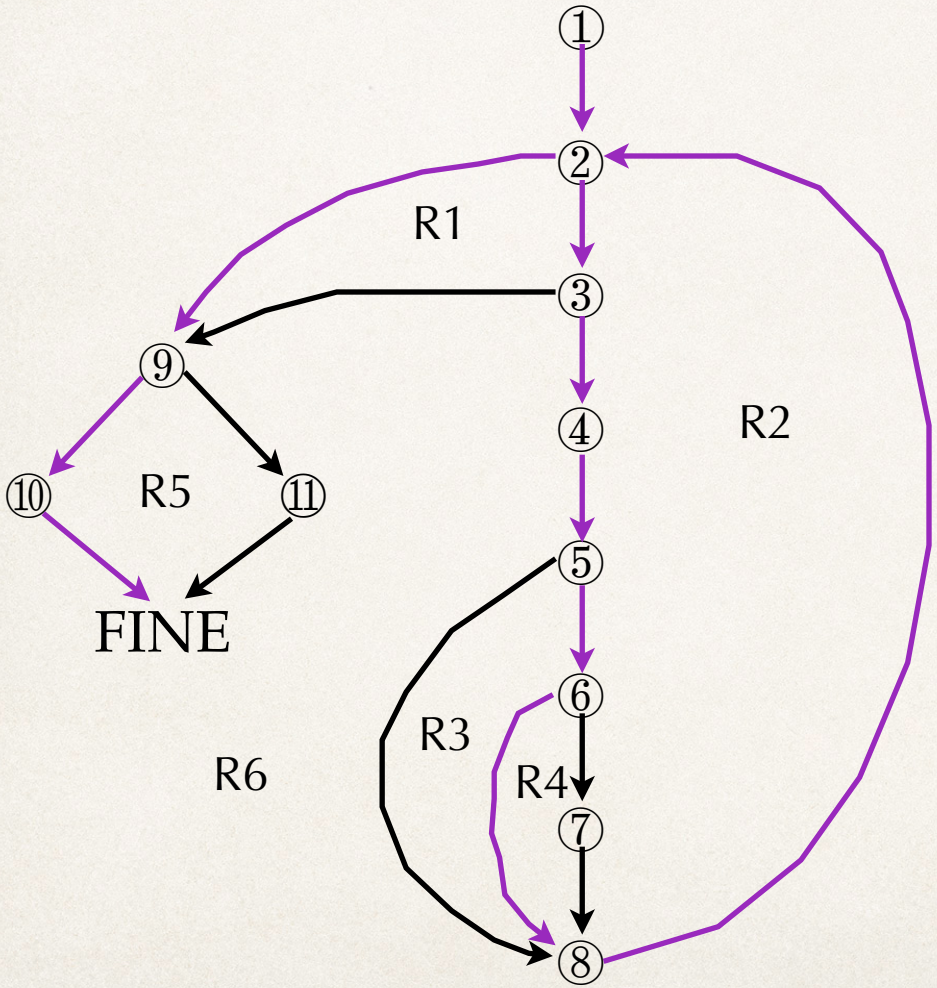
(3) 1-2-3-9-10-FINE

(4) 1-2-3-4-5-6-7-8-2-...FINE

(5) 1-2-3-4-5-8-2-...-FINE

(6) 1-2-3-4-5-6-8-2-...-FINE

Esempio di derivazione dal codice



Cammini indipendenti

(1) 1-2-9-10-FINE

(2) 1-2-9-11-FINE

(3) 1-2-3-9-10-FINE

(4) 1-2-3-4-5-6-7-8-2-...-FINE

(5) 1-2-3-4-5-8-2-...-FINE

(6) 1-2-3-4-5-6-8-2-...-FINE

Scelta dei casi di prova

- A questo punto bisogna scegliere i casi di prova in modo da verificare tutti i cammini individuati
- **N.B.**: alcuni dei cammini potrebbero esistere nel grafo di flusso ma in realtà non si possono eseguire isolatamente
  - esempio: il cammino (1)
- Questi cammini ovviamente si omettono (saranno inclusi in altri cammini testabili)

Testing della struttura di controllo

- Lo studio dei cammini di base e della complessità ciclomatica fa parte di una famiglia di tecniche per il testing della struttura di controllo
- Vi sono altre tecniche che aumentano la portata del test
- Fanno sempre parte dell'approccio white box

Testing della struttura di controllo Collaudo per condizioni

- Il collaudo per condizioni si focalizza sui test di correttezza delle condizioni logiche del programma
- ***Condizione logica semplice:*** X_1`<operatore relazionale>`X_2 dove l'operatore relazionale può essere <, <=, =, >=, >, != e X_1, X_2 sono espressioni aritmetiche
- ***Condizione logica composta***: combinazione di condizioni logiche semplici usando operatori booleani OR, AND, NOT e strutturate mediante parentesi
- Se una condizione non comprende espressioni relazionali viene chiamata *espressione booleana*

Collaudo per condizioni

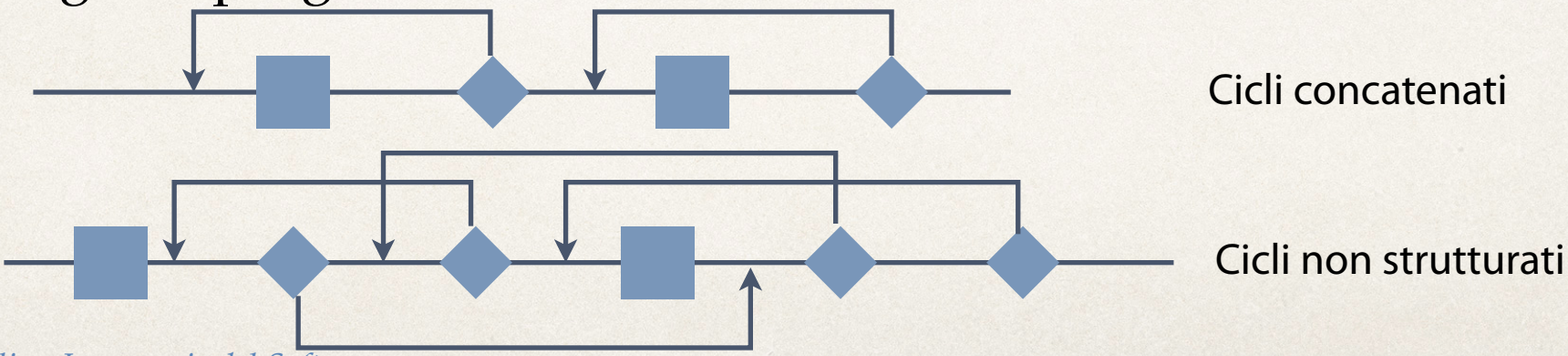
- Errori in una condizione possono essere causati da
  - errori negli operatori booleani (sbagliati, mancanti o superflui)
  - errori in una variabile booleana
  - errori nella costruzione della condizione (disposizione parentesi)
  - errori in un operatore relazionale
  - errori in una espressione aritmetica
- Il collaudo per condizioni si propone di testare tutte le condizioni di un programma: si assume che questo sia utile per individuare errori anche nelle altre parti

Testing della struttura di controllo Collaudo per cicli

- Con il collaudo per cicli ci concentriamo su test che verifichino la validità dei loop del programma
- Cicli semplici (n=numero 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 (3 prove)
- Cicli annidati
  - fissare i cicli esterni ai valori minimi e testare il ciclo interno come se fosse un ciclo semplice
  - procedere verso l'esterno mantenendo quelli più esterni al minimo e quelli più interni a un valore tipico prefissato

Collaudo per cicli

- Cicli concatenati
  - in generale si possono considerare come i semplici
  - se però il contatore di ciclo del primo viene usato come valore iniziale del secondo, non sono indipendenti: vedi cicli annidati
- Cicli non strutturati
  - è impossibile testare come si deve dei cicli non strutturati: è meglio riprogettarli



Collaudo black-box

- Questo tipo di collaudo complementa il white-box, occupandosi di errori dovuti a
  - funzioni errate o mancanti
  - errori di interfaccia
  - errori nelle strutture dati o nell'accesso a DB esterne
  - comportamento erroneo o problemi prestazionali
  - errori nelle fasi di inizializzazione o terminazione
- Contrariamente alle tecniche white-box questo collaudo viene effettuato nelle fasi finali dell'implementazione
- Bisogna individuare casi di prova che
  - siano significativi in termini di collaudo (riducano test ulteriori)
  - individuino classi di errori piuttosto che singoli errori

Collaudo black-box

- Prevede di selezionare l'insieme dei dati di input che costituirà il test a partire dallo studio delle funzionalità di un programma
- Vari criteri, a seconda delle regole con cui sono individuati i casi rilevanti che costituiranno la materia del test
- **Statistico**: si provano gli input più probabili (profilo operazionale)
- **Partizione dati di input**: dominio dati input è ripartito in classi di equivalenza (due valori della stessa classe dovrebbero produrre lo stesso comportamento). Valido se il numero di comportamenti è "ragionevole"
- **Valori di frontiera**: classi equivalenza o causa uguaglianza comportamento o in base a considerazioni inerenti il tipo di valori. Prendiamo gli estremi di queste classi.

Collaudo black-box

- **Grafo causa-effetto**: se possiamo costruire dai requisiti o dalle specifiche un grafo che lega input (cause) e output(effetto) in una rete combinatoria che definisce delle relazioni di cause-effetto.

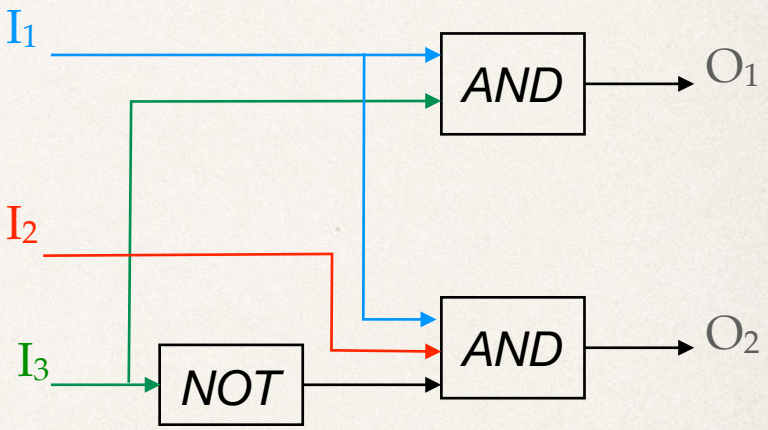
Specifiche:

L'accesso è consentito se l'utente è registrato e la password è corretta; in ogni altro caso è negato. Se l'utente è registrato come speciale e la password è errata viene emesso un warning

Fatti elementari:

Input:
- I_1 = utente registrato
- I_2 = utente speciale
- I_3 = password corretta

Output:
- O_1 = consenti accesso
- O_2 = emetti warning



Esempio classi di equivalenza

- Dividiamo il dominio dei dati di input in classi di dati, da cui derivare i test case.
- L'ideazione dei test case si basa su una valutazione delle classi di equivalenza per una condizione di input
  - es. se l'input è un intero di 5 digit tra 10000 e 99999, le classi di equivalenza sono
    - < 10000
    - tra 10000 e 99999
    - > 99999
  - si scelgono dei casi di test vicino ai confini di questi insiemi: 9999, 10000, 50000, 99999, 100000

Black box vs White box

Black box	White box
Considera solo il comportamento esterno del sistema; non si considera come è fatto internamente	Si considera il funzionamento interno del software nel fare il test.
E' svolto da testers	E' svolto dagli sviluppatori stessi
Si usa nel System Testing o nel Acceptance Testing: si usa nelle fasi finali di testing.	Si usa molto nel Unit Testing e nel Integration Testing: si comincia a fare testing con questo.
E' più rapido del white box	Richiede più tempo del black box
E' il test del comportamento	E' il test della logica di funzionamento
Non va bene per testare un algoritmo	E' adatto per testare algoritmi

Testing & Debugging

Chiariamo subito: sono due attività distinte

- Testing: conferma la presenza di errori
  - giochiamo d'anticipo, andando a caccia di possibili errori
- Debugging: localizzazione e correzione degli errori
  - giochiamo di rimessa, siamo sicuri che c'è almeno un errore da trovare

Strategia di collaudo

- La strategia di collaudo del software utilizza le stesse tecniche di collaudo sviluppate per la procedura di collaudo pianificata di un generico processo di produzione
- Bisogna
  - 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 viene visto come distruttivo
- Lo sviluppatore potrebbe essere psicologicamente condizionato nell'esecuzione del collaudo
- Affiancare un gruppo indipendente allo sviluppatore nei collaudi globali sul sistema
- Lo sviluppatore deve collaborare col team di collaudo sia in fase di test, sia per correggere gli errori individuati durante i collaudi

Valutazione dei test

- Per valutare quanto efficace è il testing che stiamo facendo usiamo il concetto di “copertura”, e distinguiamo tra:
- copertura funzionale: percentuale di funzionalità testate rispetto al numero totale di funzionalità del software
- copertura strutturale: percentuale di elementi testati rispetto al numero totale di elementi che compongono il codice. Possiamo pensare in termini di flusso di controllo o di flusso dei dati
  - Flusso di controllo:
    - copertura dei comandi
    - copertura dei cammini
  - Flusso dei dati
    - copertura delle definizioni
    - copertura degli usi

Test basati sul flusso di controllo: statisticamente la soglia ottimale da coprire è l’85%

Quando finisce un collaudo?

- Risposta filosofica
  - Mai: semplicemente da un certo punto in poi sarà il cliente a fare il collaudatore.....
- Risposta pragmatica
  - Il collaudo finisce quando finiscono le risorse allocate per farlo (tempo, persone, soldi..)
- Risposta quantitativa
  - Il collaudo finisce quando la probabilità di verificarsi di un bug per unità di tempo d'uso scende sotto una soglia prefissata
  - p.e.: stabilisco che la probabilità di verificarsi di un guasto deve essere inferiore allo 0.5% per 100 ore di funzionamento
- Esistono metodi statistici per calcolare questa probabilità in funzione dei numeri di guasti durante i collaudi

Criteri di collaudo

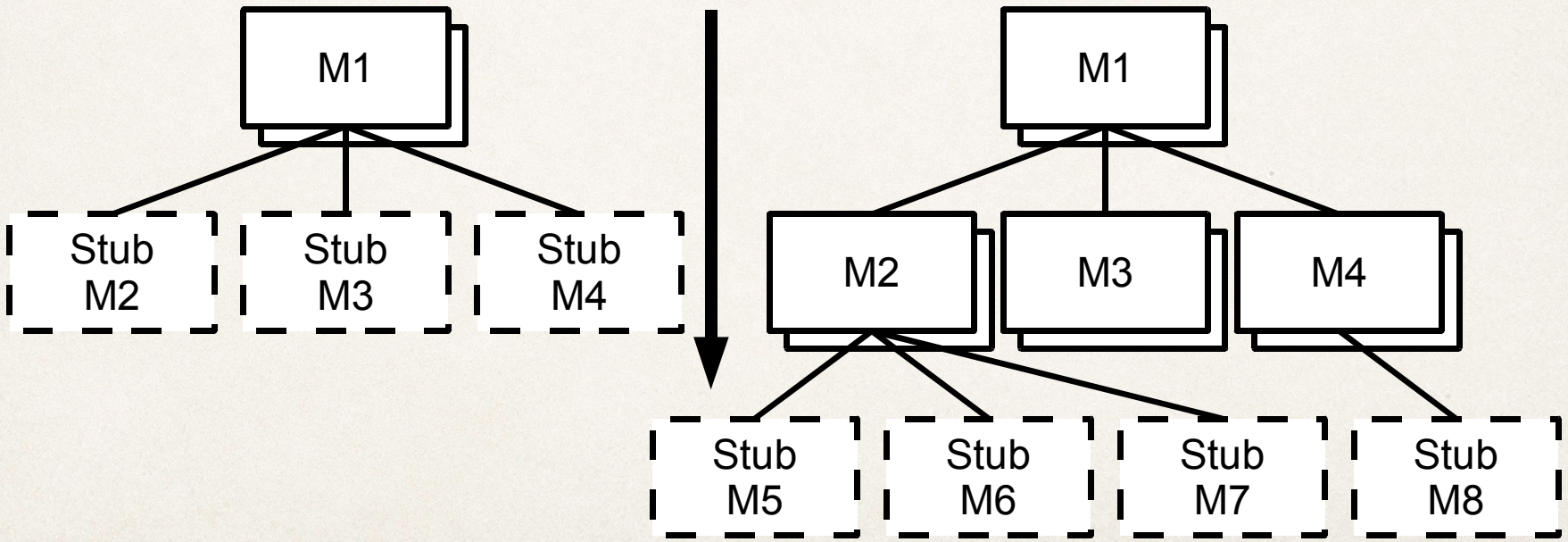
- Specificare quantitativamente i requisiti del prodotto prima di iniziare i collaudi
- Enunciare in maniere esplicita e quantitativa gli obiettivi del collaudo (MTBF finale, costi, ...)
- Sviluppare i profili di tutte le categorie di utenti
  - porta alla limitazione dei casi d'uso da controllare
-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

Strategie di testing

- Testing Top-Down
  - i test cominciano dalle componenti più astratte e si spostano via via verso le componenti di basso livello
- Testing Bottom-Up
  - i test cominciano dalle componenti di basso livello e si spostano via via verso le componenti di alto livello
- Collaudo per regressione
  - si usa per testare gli incrementi e le modifiche
- 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 Top-Down

- Primo passo: si testa il modulo radice sostituendo quelli di livello inferiore con degli stub
- Secondo passo: si includono i moduli del primo livello e si sostituiscono a quelli di livello inferiore degli stub



Test Top-Down

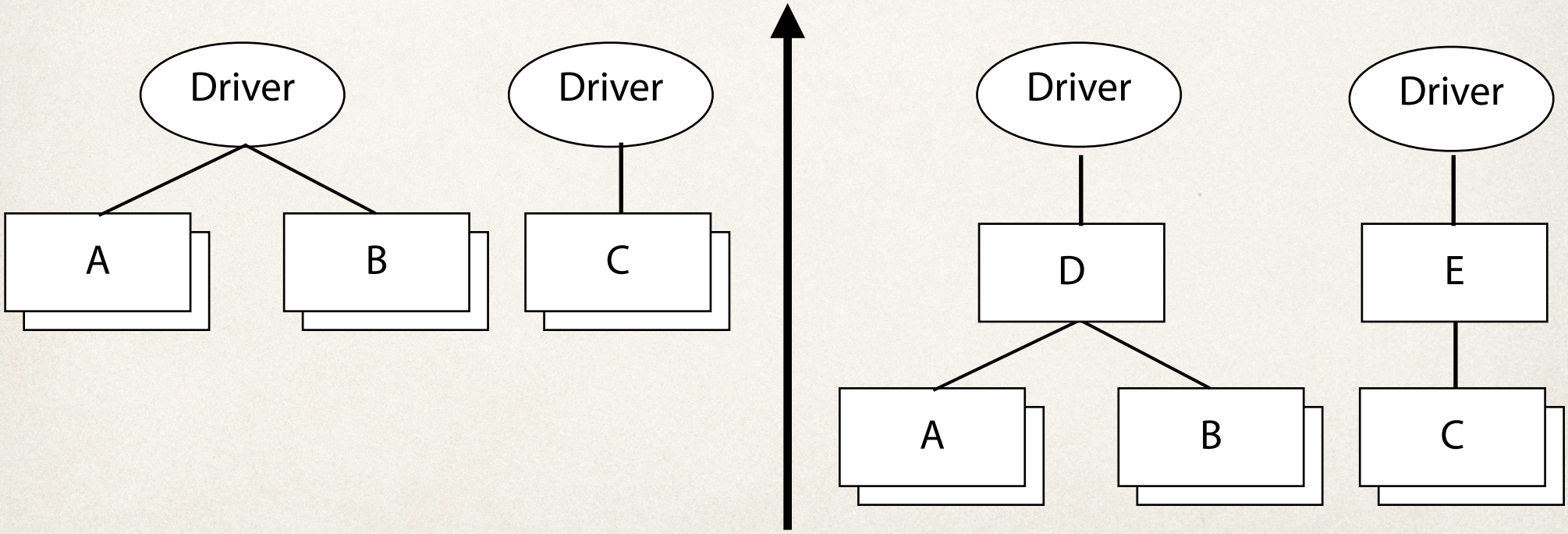
- Inizia dai livelli più elevati (radice) del sistema e procede verso il basso
- Questa strategia di test può essere facilmente usata in parallelo allo sviluppo top-down
- Può essere difficile individuare errori che dipendono dal livello più basso perché sono sviluppati per ultimi e gli stub passano dati non significativi al livello superiore
- Può individuare problemi architetturali fin dall'inizio
- Sviluppare stub può essere complicato

Casi possibili di “stub”

- Lo stub non fa nulla
  - eventualmente stampa una traccia della chiamata e dei parametri
- Lo stub colloquia con il programmatore per restituire un valore particolare
- Lo stub è una versione semplificata (un prototipo) del modulo che verrà chiamato

Test Bottom-Up

- Si parte dai moduli di livello più basso sviluppando dei driver che li usano e ne verificano il comportamento
- Successivamente si continua a salire verso la radice della gerarchia, aggiungendo livelli e testandoli con nuovi driver



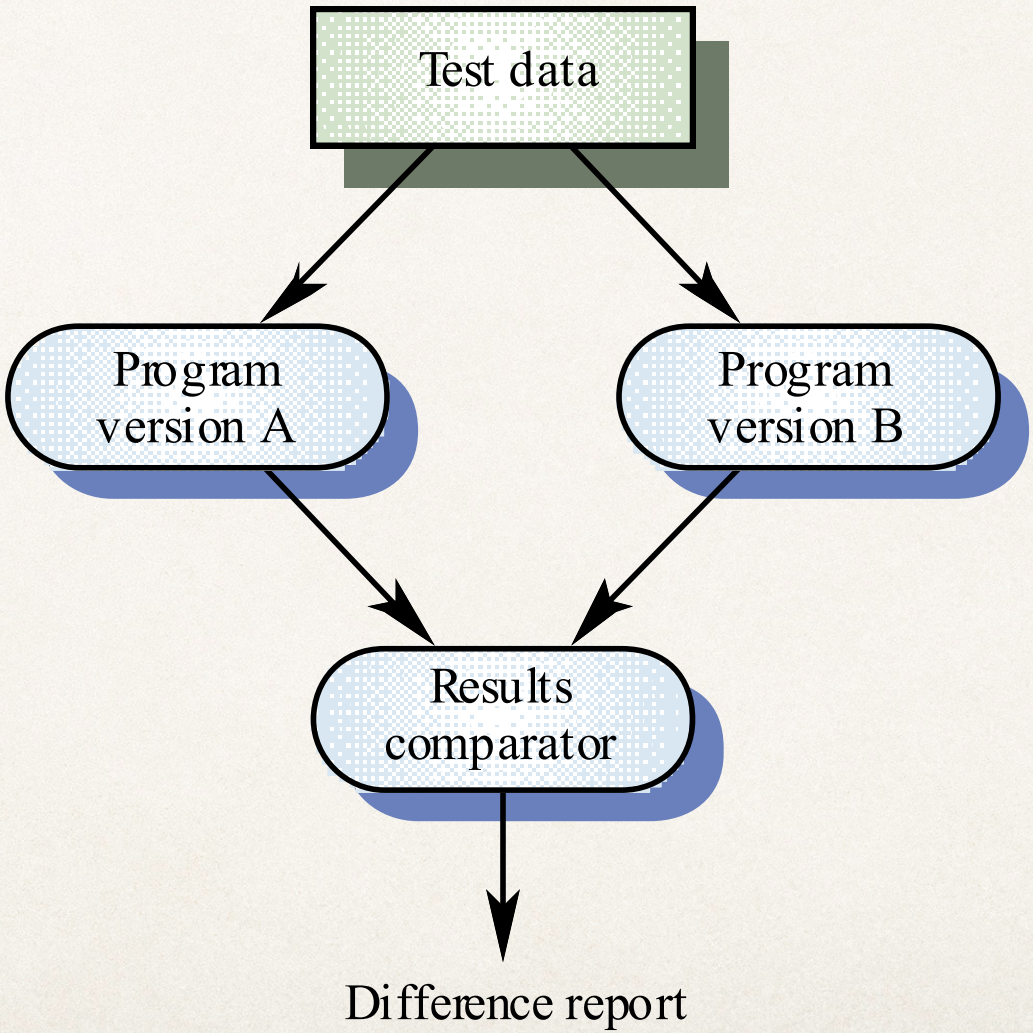
Test Bottom-Up

- Può essere utile per testare le componenti critiche di basso livello di un sistema
  - i vantaggi del bottom-up sono gli svantaggi del top-down e viceversa
- Si inizia dai livelli più bassi e ci si muove verso la radice dei componenti
- E' necessario implementare dei test driver
- Eventuali problemi di design sono scoperti relativamente tardi
- E' una tecnica che risulta appropriata per sistemi object-oriented (non solo quelli in realtà)

Back-to-back testing

- Testa diverse versioni del programma con lo stesso input e confronta gli output
- se l'output è diverso ci sono errori potenziali
- Riduce il costo di esaminare il risultato dei test: il confronto degli output può essere automatizzato
- Si può usare quando è disponibile un prototipo, quando un sistema viene sviluppato in più versioni (magari su diverse piattaforme), oppure nel caso di upgrade o nuove release del sistema

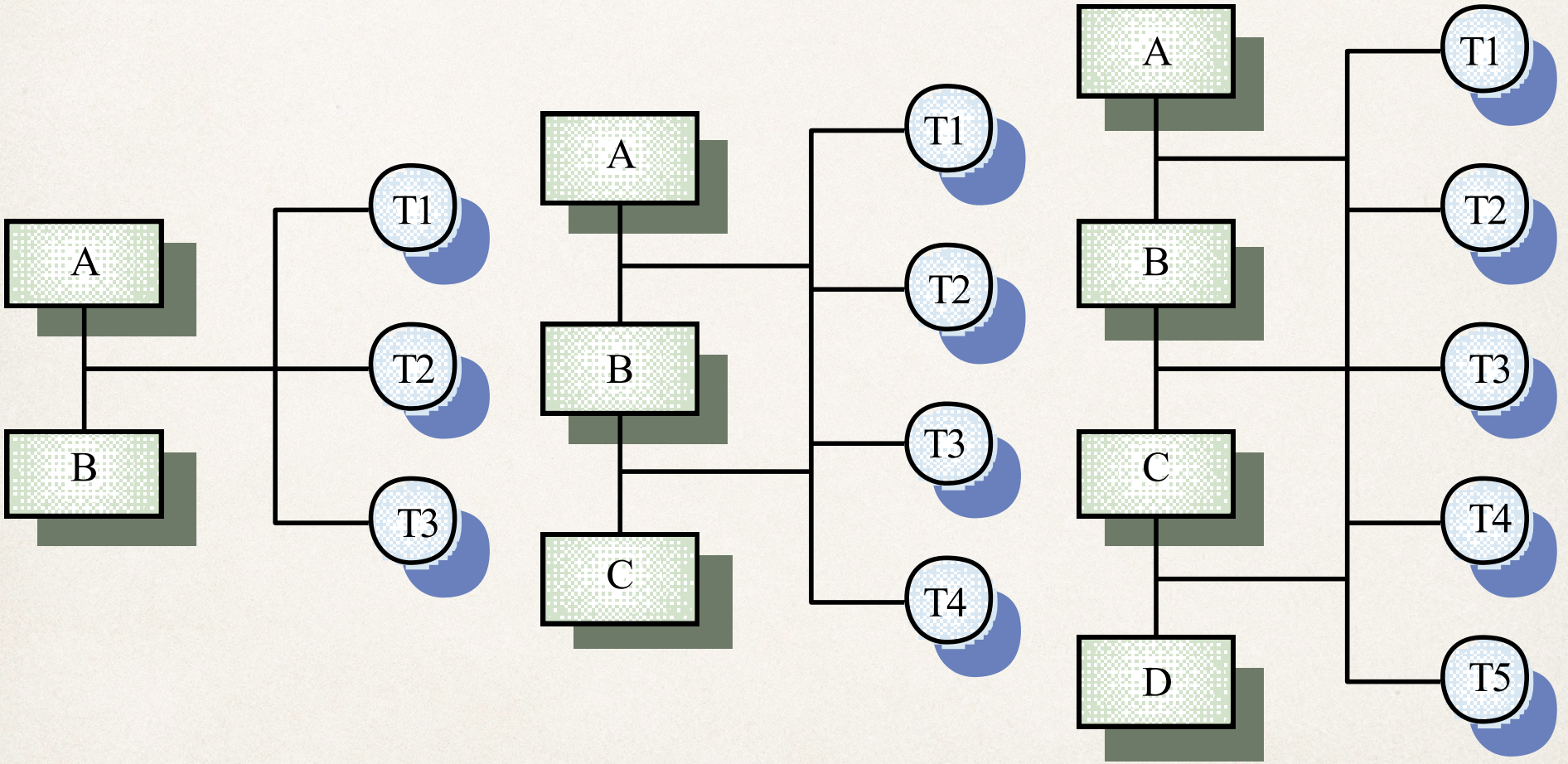
Back-to-back testing



Collaudo incrementale / Continuous Integration

- Ad ogni introduzione di un nuovo modulo il software cambia
- sono possibili nuovi cammini di flusso, nuove interfacce, ...
- Questo può introdurre problemi anche nelle parti già collaudate prima dell'aggiunta del nuovo modulo
- Occorre ripetere le prove già fatte per verificare che una modifica non abbia portato effetti collaterali
  - va fatto ad ogni modifica: nuovo modulo, correzione di un bug, aggiornamento
- Si possono usare strumenti automatici per registrare alcune prove chiave e poi riprodurle in automatico in fase di collaudo

Collaudo incrementale



Test sequence 1

Test sequence 2

Test sequence 3

“Smoke-testing”

- Quando i tempi di un nuovo incremento sono brevi, si può inserire il collaudo nel corso dello sviluppo del codice
- I moduli 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 funzionamento
- Tutti i build vengono integrati e il programma risultante viene testato quotidianamente
- Se si identificano degli errori prima inesistenti, il codice aggiunto per ultimo è il maggiore indiziato come causa
- Con questo metodo si ha una misura continuamente aggiornata dei progressi nello sviluppo del programma

Vantaggi del smoke test

- Minimizza i rischi di integrazione
  - un rischio classico nello sviluppo di grossi progetti consiste nel trovare difficoltà inaspettate all'atto dell'integrazione di varie componenti: con questo metodo l'integrazione viene fatta quotidianamente e monitorata
- Riduce i rischi di produrre codice di bassa qualità
  - la build viene controllata ogni giorno
- E' facile diagnosticare i problemi
  - di solito un controllo su quello che è stato cambiato o aggiunto quel giorno è sufficiente a trovare l'errore
- Migliora il morale del team di sviluppo
  - si vede che il prodotto che si sta costruendo funziona effettivamente

Smoke test all'atto pratico

- Build daily
- Controlla se il build fallisce
- Effettua quotidianamente lo smoke test
- Aggiorna i pacchetti nel build solo dopo il test
  - lo sviluppatore mette nel build una versione aggiornata dei suoi pacchetti solo se passano lo smoke test
- Chi fa fallire il build deve essere penalizzato
  - l'idea è che un build fallito deve essere l'eccezione
  - se il build fallisce bisognerebbe fermare tutto e risolvere prima il problema
- Effettua i build e i test anche durante i periodi critici

Recovery testing

- Molti sistemi informatici devono recuperare la situazione dopo un malfunzionamento e riprendere l'elaborazione in un tempo determinato
- In alcuni casi il sistema deve essere tollerante ai guasti; in altri deve correggere il malfunzionamento in un tempo predefinito
- Si forza il software a incorrere in errore in molti modi diversi e si verifica che il recupero sia eseguito correttamente
  - se il recupero richiede intervento umano, si valuta il tempo medio necessario

Security testing

- Molti sistemi devono essere protetti contro un uso non autorizzato oppure contro un accesso non autorizzato alle risorse
- Si cerca di verificare i meccanismi di protezione costruiti all'interno del sistema
- Avendo tempo e risorse, un buon security testing riuscirà a violare il sistema: l'obiettivo è avere un sistema che richieda un tempo o risorse tali per cui il costo per violare il sistema è superiore al guadagno

Stress testing

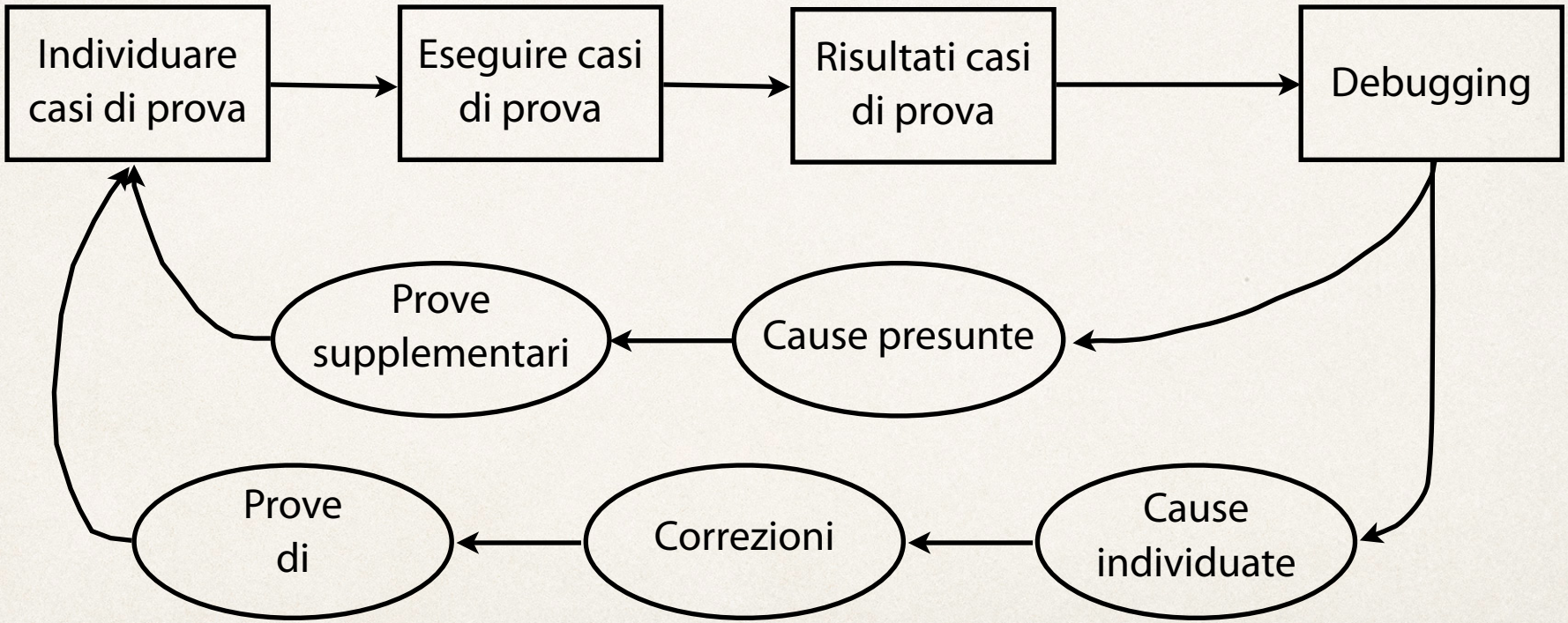
- Gli stress test sono progettati per provare i programmi in situazioni di carico eccessivo
  - generare 10 richiesta al secondo a fronte di una media stimata di 2
  - progettare casi di utilizzo che richiedono il massimo della memoria e il massimo di altre risorse possibili
- Sostanzialmente si cerca di far crollare il programma
- Se si sta testando degli algoritmi matematici, si usa il sensitivity test: valori dei parametri molto vicini ai limiti di validità dell'algoritmo
  - lo scopo è cercare combinazioni di dati che, anche se validi come input, possono causare instabilità o elaborazioni non corrette

Prove di validazione

- I criteri di validazione sono contenuti nelle specifiche dei requisiti: descrivono le ragionevoli aspettative del cliente
  - Un criterio di validazione spesso trascurato: verificare che la configurazione sw sia opportunamente catalogata e documentata per garantire il supporto futuro
- Il modo migliore di effettuare la validazione consiste nel far usare il sistema ad un insieme ridotto di utenti per verificarne il comportamento in situazioni reali
  - alpha version: gli utenti usano il sistema sotto diretto controllo degli sviluppatori
  - beta version: gli utenti usano il sistema da soli e redigono un report con errori e problemi individuati

Il debugging

- Lo scopo è eliminare gli errori individuati durante il collaudo del sistema



Il debugging

- Perché è difficile trovare le cause di un errore?
  - distanza tra causa e punto in cui l'errore appare (accoppiamento)
  - non c'è una causa specifica (p.e. errori 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 (p.e. in sistemi real-time)
  - l'errore è intermittente (forte legame hw-sw)
  - il sintomo dipende da varie cause distribuite su vari processi
- Correggere un errore può spesso provocarne di nuovi: applicare sempre il collaudo di regressione

Il debugging

- La capacità di debuggare varia molto da individuo a individuo e dipende anche da fattori psicologici (ansia, rifiuto)
- Metodi principali di debugging
  - forza bruta: dump della memoria, messaggi di debug, ... di solito si tengono come ultima ratio
  - cammino a ritroso: si parte dal punto in cui si è manifestato l'errore e si procede a ritroso; per programmi piccoli
  - eliminazione delle cause: metodo scientifico in cui si formulano delle ipotesi e si inventano test per verificarle
- Strumenti che aiutano nelle fasi di debugging: debugger, compilatori con controllo degli errori, ...
- Quando tutto fallisce, chiedi aiuto agli altri!
  - un alto paio di occhi può notare quello che non vedi

La revisione di progetto

- Si basa su riunioni di revisione
- Perché funzionino vanno pianificate e gestite
  - numero ridotto di partecipanti
  - documentazione scritta dal progettista disponibile prima della riunione
  - durata prestabilita e limitata del meeting
  - discussione focalizzata sulla scoperta di problemi, non sulla correzione
  - persone coinvolte: progettista che presenta e spiega il lavoro, moderatore, verbalizzatore
  - atmosfera cooperativa => no managers

La checklist

- L'ispezione deve essere guidata da una checklist di errori comuni, che dipende fortemente dal linguaggio di programmazione usato
- Più il linguaggio è di basso livello, più la lista degli errori è lunga

Esempio: ispezione del codice

- Uso di elenco (checklist) di possibili situazioni erronee da ricercare (dipende dal linguaggio)
  - uso di variabili non inizializzate
  - salti all'interno di cicli
  - assegnamenti incompatibili
  - cicli che non terminano
  - indici di array fuori dai limiti
  - erronea allocazione/deallocazione
  - inconsistenza tra parametri formali e attuali
  - confronto di eguaglianza tra reali

Fault class	Inspection check
Data faults	Are all program variables initialised before their values are used? Have all constants been named? Should the lower bound of an array be 0, 1, or something else? Should the upper bound of an array be equal to the size of the array or Size-1? If character strings are used, is a delimiter explicitly assigned?
Control faults	For each conditional statement, is the condition correct? Is each loop certain to terminate? Are compound statement correctly bracketed? In case statements, are all possible cases accounted for?
Input/Output faults	Are all input variables used? Are all output variables assigned a value before they are output?
Interface faults	Do all functions and procedure calls have the correct number of parameters? Do formal and actual parameter types match? Are the parameters in the right order? If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	If a linked structure is modified, have all links been correctly reassigned? If dynamic storage is used, have space been allocated correctly? Is space explicitly de-allocated after if is no longer required?
Exception management faults	Have all possible conditions been taken into account?

Walkthrough

- Obiettivo
  - rilevare la presenza di difetti
  - eseguire una lettura critica del codice
- Agenti
  - gruppi misti ispettori/sviluppatori
- Strategia
  - percorrere il codice simulandone l'esecuzione

Le fasi del walkthrough

- Fase 1: pianificazione
- Fase 2: lettura del codice
- Fase 3: discussione
- Fase 4: correzione dei difetti
- Documentazione

Revisione vs. walkthrough

- Similitudini
  - controllo statici basati su desk-test
  - contrapposizione fra programmatori e verificatori
  - documentazione formale
- Differenze
  - revisione si basa su errori presupposti (checklist)
  - walkthrough si basa sull'esperienza dei verificatori
  - walkthrough è più collaborativo
  - revisione è più rapido