Progettazione delle prove

Vincenzo Gervasi, Laura Semini
Ingegneria del Software
Dipartimento di Informatica
Università di Pisa

prova (o collaudo o test)

- Verifiche (o validazioni) dinamiche
  - attività che prevedono l'esecuzione del software
  - in un ambiente controllato e con input e output definiti
  - sui moduli o sul sistema
- Metodo tanto intuitivo quanto complesso
  - pur essendo la forma di controllo più intuitiva (funzionerà? Beh, proviamo…) è la più complessa da realizzarsi (bene)
  - progettazione, esecuzione, analisi, debugging
  - validazione dei risultati, terminazione delle prove

Caratteristiche

- Una prova non è sempre definitiva
  - è definita e ripetibile
  - i suoi risultati non sono estendibili: In linea di principio, i risultati di una prova valgono solo per le condizioni di quella prova
  - evidenzia un malfunzionamento (presenza di difetti): in altre parole (quelle di Dijkstra) la prova non potrà mai dimostrare l'assenza di difetti
- Le prove sono costose
  - occorrono molte risorse (tempo, uomini, macchine)
  - è necessario un processo definito
  - richiedono ulteriori attività di ricerca del difetto e correzione
  - il controllo dinamico è legato alle dimensioni dell'input, dell'output, dello stato e dell'ambiente operativo: variabili che, oltre ad essere grandi, sono spesso difficili da controllare e modellare

Gli elementi di una prova

- Caso di prova (o test case)
  - una tripla <input, output, ambiente>
- Batteria di prove (o test suite)
  - un insieme (una sequenza) di casi di prova
  - una batteria può servire alla creazione di uno stato, alla copertura
- Procedura di prova
  - le procedure (automatiche e non) per eseguire, registrare analizzare e valutare i risultati di una batteria di prove

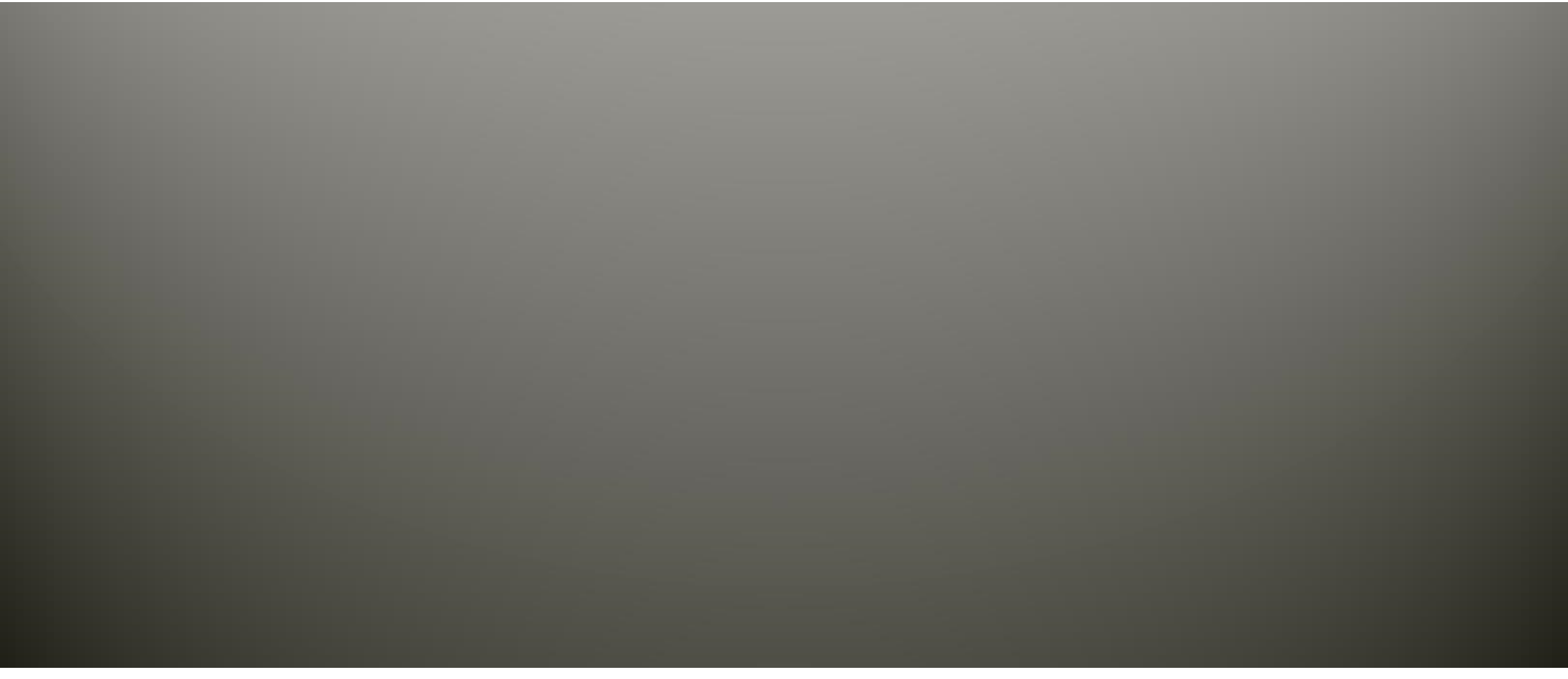
Conduzione di una prova

- Definizione dell'obiettivo della prova
  - è importante definire l'obiettivo
- Progettazione della prova
  - la progettazione consiste soprattutto nella scelta e nella definizione dei casi di prova (della batteria di prove)
- Realizzazione dell'ambiente di prova
  - ci sono driver e stub da realizzare, ambienti da controllare, strumenti per la registrazione dei dati da realizzare

Progettazione

- Criteri funzionali
  - a scatola chiusa (black box)
  - basati sulla conoscenza delle funzionalità
  - mirati a evidenziare malfunzionamenti sospettati o comunque relativi a funzionalità identificate
- Criteri strutturali
  - a scatola aperta (white box)
  - basati sulla conoscenza del codice
  - mirati a esercitare il codice indipendentemente dalle funzionalità
- Gray box, una strategia più che un criterio (come vedremo in seguito)

Criteri funzionali



Metodo statistico

- I casi di test sono selezionati in base alla distribuzione di probabilità dei dati di ingresso del programma
- Il test è quindi progettato per esercitare il programma sui valori di ingresso più probabili per il suo utilizzo a regime
- Il vantaggio è che, nota la distribuzione di probabilità, la generazione dei dati di test è facilmente automatizzabile
- Non sempre corrisponde alle effettive condizioni d'utilizzo del software
- È oneroso calcolare il risultato atteso

Partizione dei dati d'ingresso

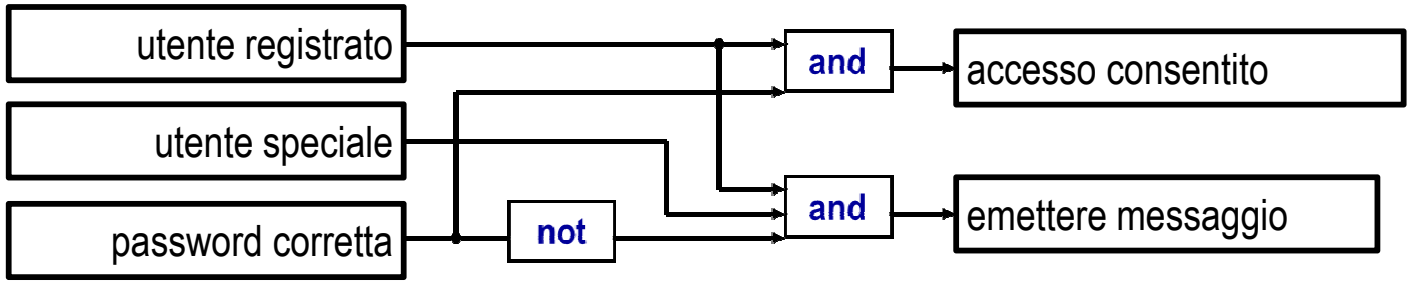
- Il dominio dei dati di ingresso è ripartito in classi di equivalenza
  - due valori d'ingresso appartengono alla stessa classe di equivalenza se, in base ai requisiti, dovrebbero produrre lo stesso comportamento del programma
- Il criterio è economicamente valido solo per quei programmi per cui il numero dei possibili comportamenti è sensibilmente inferiore alle possibili configurazioni d'ingresso
  - per come sono costruite le classi, i risultati attesi dal test sono noti e quindi non si pone il problema dell'oracolo
- Il criterio è basato su un'affermazione generalmente plausibile, ma non vera in assoluto
  - la deduzione che il corretto funzionamento sul valore rappresentante implichi la correttezza su tutta la classe di equivalenza dipende dalla realizzazione del programma e non è verificabile sulla base delle sole specifiche funzionali

Valori di frontiera

- Basato su una partizione dei dati di ingresso
  - le classi di equivalenza realizzate o in base all'eguaglianza del comportamento indotto sul programma o in base a considerazioni inerenti il tipo dei valori d'ingresso
- Dati di test: valori estremi di ogni classe di equivalenza
- È possibile che debba essere considerato il problema dell'oracolo
- Questo criterio richiama i controlli sui valori limite tradizionali in altre discipline ingegneristiche per le quali è vera la proprietà del comportamento continuo
  - in meccanica, ad esempio, una parte provata per un certo carico resiste con certezza a tutti i carichi inferiori
- Questa proprietà non è applicabile al software: i valori limite sono frequentemente trattati in modo particolare

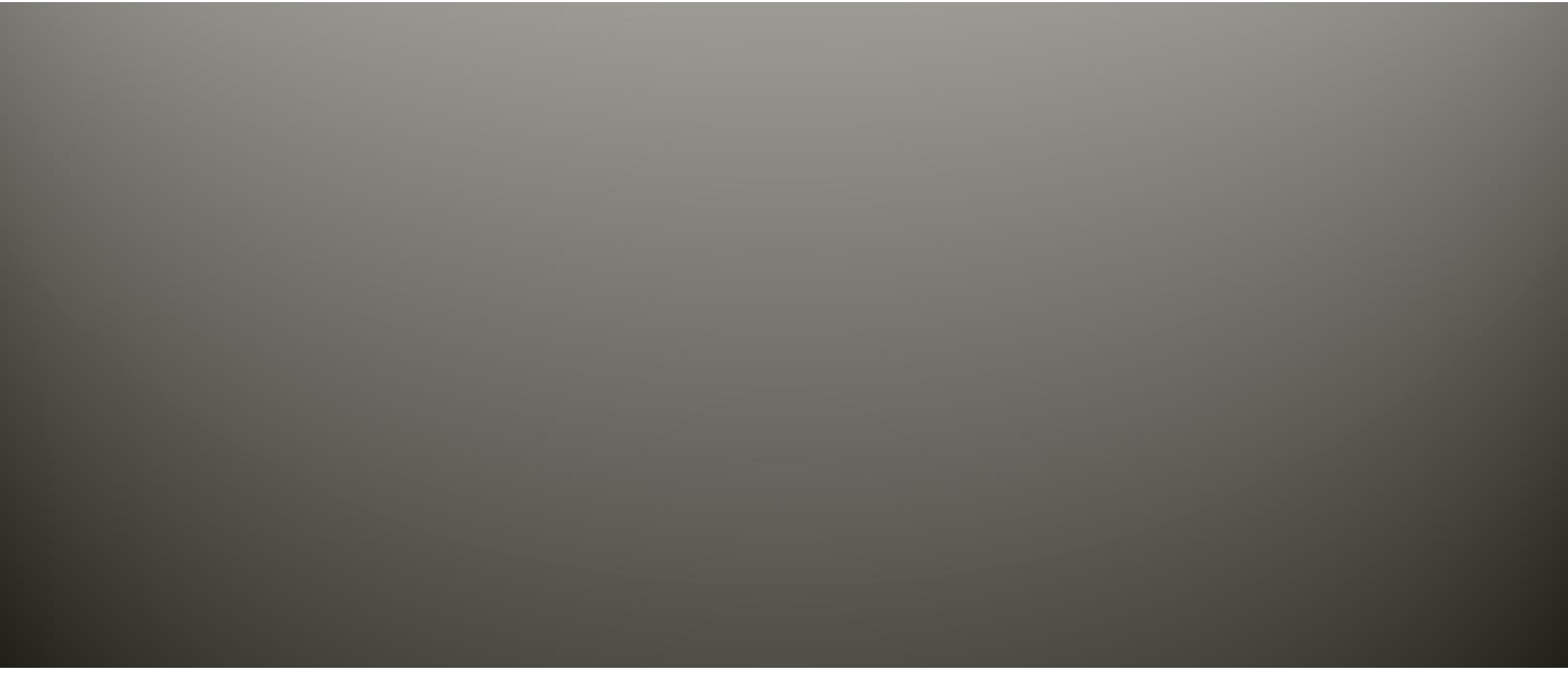
Grafi causa-effetto

- Requisiti
  - l'accesso è consentito se l'utente è registrato e la password è corretta, è negato in ogni altro caso è negato
  - se l'utente è speciale e la password è errata viene emesso un messaggio sulla console di sistema



- Grafo che lega un insieme di fatti elementari di ingresso (cause) e di uscita (effetti) in una rete combinatoria che definisce relazioni di causa-effetto

Criteri strutturali

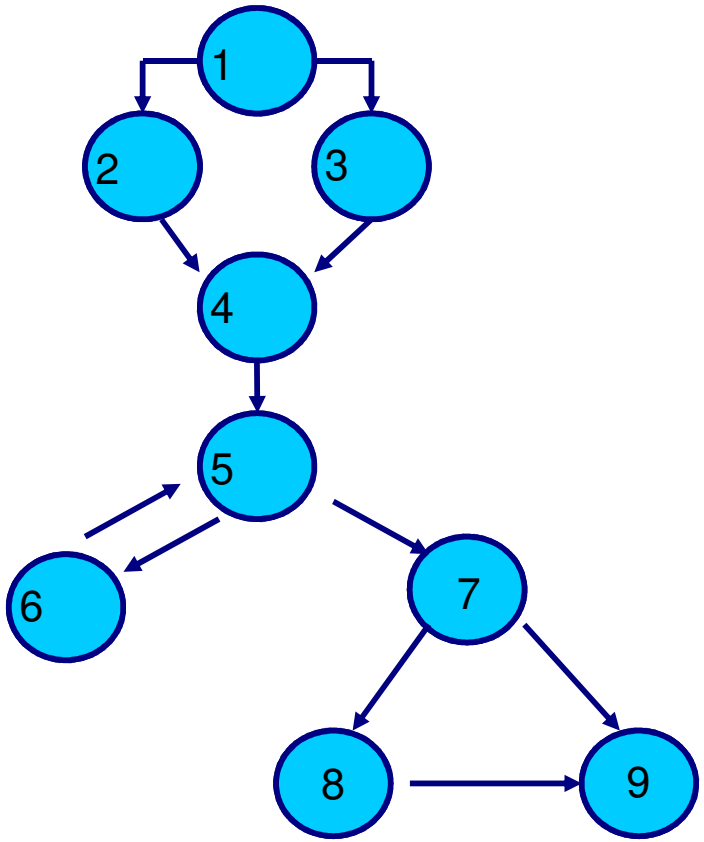


Grafo di flusso

- Grafo di flusso
  - definisce la struttura del codice identificandone le parti
  - è ottenuto a partire dal codice
- I diagrammi a blocchi (detti anche diagrammi di flusso, flow chart in inglese) sono un linguaggio di modellazione grafico per rappresentare algoritmi (in senso lato)

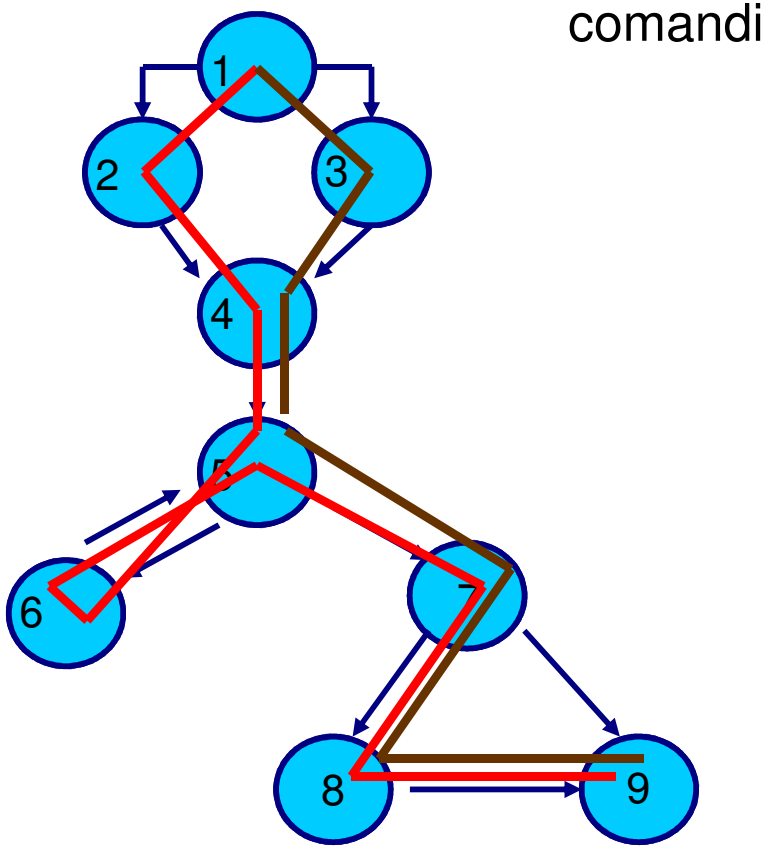
Un esempio di grafo di flusso

```
double eleva(int x, int y) {
    1. if (y<0)
    2.     pow = 0-y;
    3.     else pow = y;
    4. z = 1.0;
    5. while (pow!=0)
    6.     { z = z*x; pow = pow-1 }
    7. if (y<0)
    8.     z = 1.0 / z;
    9. return(z);
}
```



Copertura dei comandi

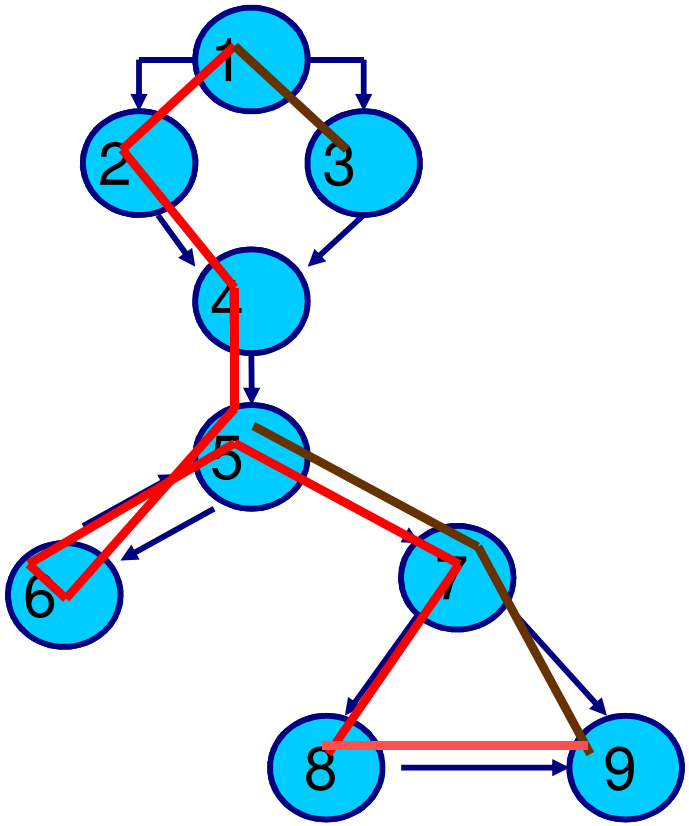
```
double eleva(int x, int y) {
    1. if (y<0)
    2.     pow = 0-y;
    3.     else pow = y;
    4. z = 1.0;
    5. while (pow!=0)
    6.     { z = z*x; pow = pow-1
}
    7. if (y<0)
    8.     z = 1.0 / z;
    9. return(z);
}
```



Copertura delle decisioni

```
double eleva(int x, int y){

if (y<0)
      pow = 0-y;
      else pow = y;
z = 1.0;
while (pow!=0)
      { z = z * x; pow = pow-1}
if (y<0)
      z = 1.0 / z;
return(z);
       }
```



decisioni

Criteri di copertura (da Binato et al.)

- Statement coverage = copertura comandi (meglio: copertura istruzioni)
- Edge coverage = copertura degli archi = copertura delle decisioni
- Condition coverage = copertura delle condizioni, (anzi di più), 4 casi di test per un OR / AND
  - si consideri il codice if (x>1 && y==0) {comando1} else {comando2}
  - e il test {x=2, y=0} e {x=2, y=1}
  - Il test garantisce la piena copertura delle decisioni, quindi dei rami e degli statement, ma non esercita tutte le combinazioni delle due condizioni in and
- Path coverage = copertura dei cammini: 4 casi per 2 if in sequenza
- Copertura cicli: si decide quanti

Criteri funzionali vs. strutturali

- Generalità degli approcci
  - rispetto alla validità dei risultati
  - rispetto alle caratteristiche da provare
  - rispetto ai costi da sostenere
- Dipendenze e implicazioni
  - l'applicazione dei criteri funzionali non dipende dal codice
  - i criteri strutturali si prestano alla valutazione della copertura

Criteri gray-box

- Una strategia di tipo gray-box prevede di testare il programma conoscendo i requisiti ed avendo una limitata conoscenza della realizzazione, per esempio conoscendo solo l'architettura
- Un'altra strategia gray-box propone di progettare il test usando criteri funzionali e quindi di usare le misure di copertura (si veda la sezione "Valutazione dei test") dei criteri strutturali per valutare l'adeguatezza del test

Un approccio (à la Myers)

- Un primo test è progettato utilizzando il grafo causa-effetto
- il grafo causa-effetto, per determinare una partizione del dominio dei dati d'ingresso che sarà usata per integrare il test precedente applicando il criterio dei valori di frontiera;
- I progettisti sono chiamati a formulare delle ipotesi di malfunzionamento (error guessing) e a integrare di conseguenza i casi di test
- Infine, la struttura del programma è usata per stabilire se i test realizzati ai passi precedenti hanno "esercitato" a sufficienza il codice

Test mutazionale

- La tecnica si applica in congiunzione con altri criteri di test
- Nella sua formulazione è prevista infatti l'esistenza, oltre al programma da controllare, anche di un insieme di test già realizzati. La strategia prevede di introdurre modifiche controllate nel programma originale
- Le modifiche riguardano in genere l'alterazione del valore delle variabili e la variazione delle condizioni booleane. I programmi così ottenuti, e incorretti – di regola – rispetto alle specifiche, sono definiti mutanti. L'insieme dei test realizzati precedentemente viene quindi applicato, senza modifiche, a tutti i mutanti e i risultati confrontati con quelli degli stessi test eseguiti sul programma originale
- Questa strategia è adottata con obiettivi diversi
  - favorire la scoperta di malfunzionamenti ipotizzati: intervenire sul codice può essere più conveniente rispetto alla generazione di casi di test ad hoc.
  - valutare l'efficacia dell'insieme di test, controllando se "si accorge" delle modifiche introdotte sul programma originale.
  - cercare indicazioni circa la localizzazione dei difetti la cui esistenza è stata denunciata dai test eseguiti sul programma originale
- uso limitato dal gran numero di mutanti che possono essere definiti, dal costo della loro realizzazione, e soprattutto dal tempo e dalle risorse necessarie a eseguire i test sui mutanti e a confrontare i risultati

Test di regressione

- Obiettivo: controllare se, dopo una modifica, il software è regredito, se cioè siano stati introdotti dei difetti non presenti nella versione precedente alla modifica
- Strategia: riapplicare al software modificato i test progettati per la sua versione originale e confrontare i risultati
- Uso in manutenzione. Di fatto, però, il susseguirsi di interventi di manutenzione adattiva e soprattutto perfettiva (e non monotona) rendono la batteria di test obsoleta
- Uso nei processi di sviluppo evolutivi
  - prototipi
  - i test, soprattutto mirati alle funzionalità del prodotto, sono sviluppati insieme al primo prototipo e accompagnano l'evoluzione
  - integrazione top-down

Test di interfaccia

- Rivisitazione dei criteri strutturali in termini dell'architettura di un sistema invece che del codice di un programma
- Basati su una classificazione degli errori commessi nella definizione delle interazioni fra i moduli
- Errore di formato: i parametri di invocazione o di ritorno di una funzionalità sono sbagliati per numero o per tipo
  - difetto frequente, ma fortunatamente compilatori e linker permettono di rilevare automaticamente con controlli statici
- Errore di contenuto: i parametri di invocazione o di ritorno di una funzionalità sono sbagliati per valore
  - è il caso in cui i moduli si aspettano argomenti il cui valore deve rispettare ben precisi vincoli; si va da parametri non inizializzati (e.g. puntatori nulli) a strutture dati inutilizzabili (e.g. un vettore non ordinato passato a una procedura di ricerca binaria)
- Errore di sequenza o di tempo
  - in questo caso è sbagliata la sequenza con cui è invocata una serie di funzionalità, singolarmente corrette; nei sistemi dipendenti dal tempo possono anche risultare sbagliati gli intervalli temporali trascorsi fra un'invocazione e l'altra o fra un'invocazione e la corrispondente restituzione dei risultati

L'oracolo

- Un metodo per generare risultati attesi…

Progettazione di un oracolo

- Risultati ricavati dalle specifiche
  - specifiche formali
  - specifiche eseguibili
  - Esempio: grafi causa-effetto
- Inversione delle funzioni
  - quando l'inversa è "più facile"
  - a volte disponibile fra le funzionalità
  - limitazioni per difetti di approssimazione

Progettazione di un oracolo (cont'd)

- Semplificazione dei dati d'ingresso
  - provare le funzionalità su dati semplici
  - risultati noti o calcolabili con altri mezzi
  - ipotesi di comportamento costante
- Semplificazione dei risultati
  - accontentarsi di risultati plausibili
  - tramite vincoli fra ingressi e uscite
  - tramite invarianti sulle uscite

Progettazione di un oracolo (cont'd)

- Versioni precedenti dello stesso codice
  - disponibili (per funzionalità non modificate)
  - prove di non regressione
- Versioni multiple indipendenti
  - programmi preesistenti (back-to-back)
  - sviluppate ad hoc
  - semplificazione degli algoritmi

Valutazione delle prove

- Costi vs. confidenza
  - le prove sono eseguite per sviluppo o per accettazione
  - devono fornire adeguata confidenza
  - la confidenza di una prova costa
- Valutazione di una prova
  - qualità di una prova in sé
  - raggiungimento della confidenza desiderata

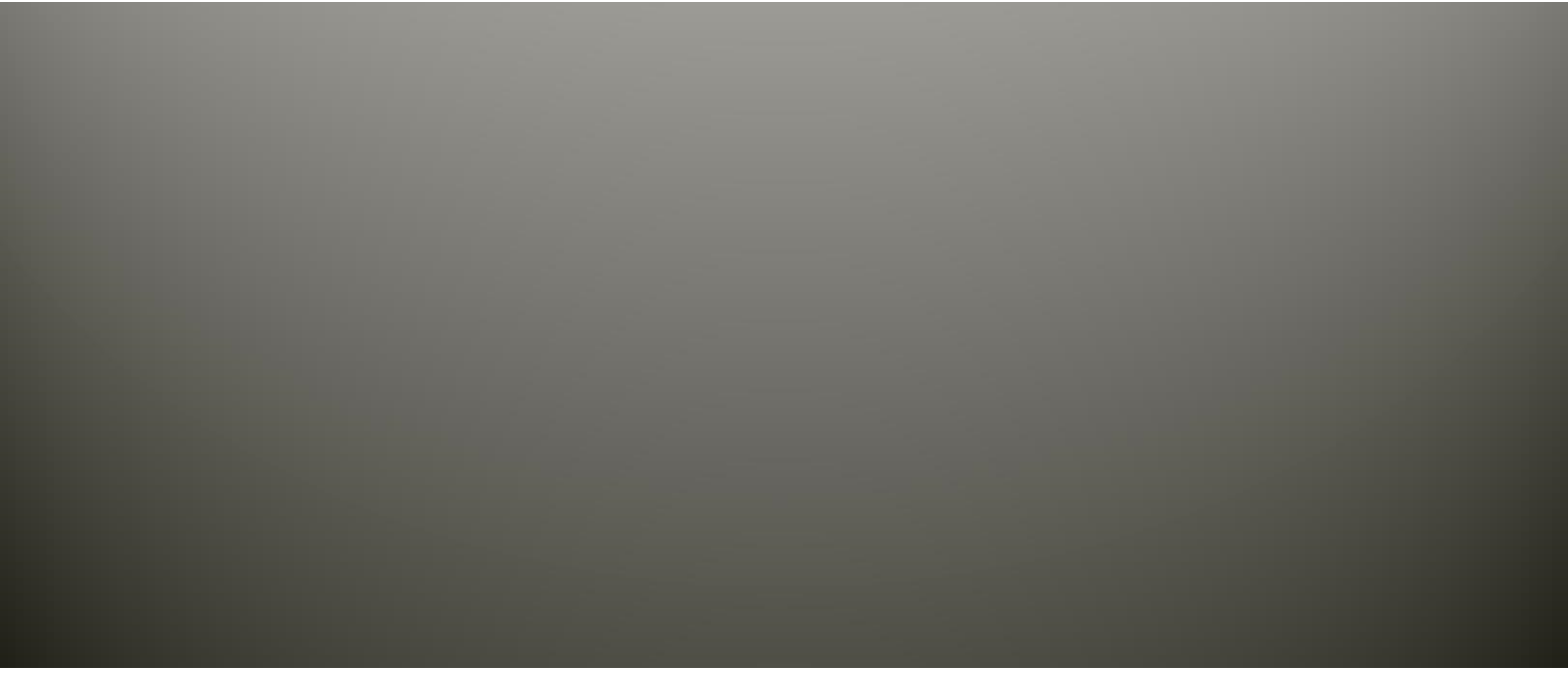
La percentuale di copertura

- Quanto la prova "esercita" il prodotto
  - copertura funzionale: rispetto alla percentuale di funzionalità esercitate
  - copertura strutturale: rispetto alla percentuale di codice esercitata
- Una misura della bontà di una prova
  - la copertura del 100% non significa assenza di difetti
  - non è detto che il 100% di copertura sia raggiungibile

La maturità

- Valutare l'evoluzione del prodotto
- quanto, in seguito alle prove, il prodotto migliora
- quanto i malfunzionamenti tendono a "sparire"
- quanto costa la scoperta del prossimo malfunzionamento
- Definire un modello ideale
  - modello base: il numero di difetti del software è costante
  - modello logaritmico: le modifiche introducono difetti

Verifica di requisiti non funzionali



Verifica, validazione e qualità

- Strumento di evidenza
  - A fronte di una metrica e di livelli definiti
  - Verificare (validare) per dare evidenza
  - Controllo (interno) e assicurazione (esterna) della qualità
- ISO/IEC 9126 come riferimento
  - Quali strumenti per quali caratteristiche?
  - La qualità in uso è esclusa

Funzionalità

- Tendenzialmente prove
- Verifica statica come attività preliminare
- Liste di controllo (rispetto ai requisiti)
  - Appropriatezza (tutte e sole le funzionalità)
  - Interoperabilità (soluzioni adottate)
  - Sicurezza (soluzioni adottate)
  - Aderenza alle prescrizioni
- Prove per accuratezza

Affidabilità

- Tendenzialmente prove
- Verifica statica come attività preliminare
- Liste di controllo (rispetto ai requisiti)
  - Tolleranza ai guasti (guasti tollerati)
  - Recuperabilità (soluzioni adottate)
  - Aderenza alle prescrizioni
- Prove per maturità

Usabilità

- Impossibile fare a meno delle prove
- Verifica statica come attività complementare
- Liste di controllo (rispetto ai manuali)
  - Comprensibilità
  - Apprendibilità
  - Aderenza alle prescrizioni
- Questionari all'utenza (a seguito di prove)
  - Operabilità
  - Attraenza

Efficienza

- Impossibile fare a meno delle prove
- Verifica statica come attività preliminare
- Liste di controllo (risp. a criteri realizzativi)
  - Efficienza algoritmica
  - Allocazione/deallocazione delle risorse
- Miglioramento vs confidenza
  - L'efficienza deve essere provata
  - La verifica statica non dà confidenza, ma migliora il codice

Manutenibilità

- Verifica statica come strumento ideale
- Liste di controllo (norme di codifica)
  - Analizzabilità
  - Modificabilità
  - Aderenza alle prescrizioni
- Liste di controllo (batterie di prove)
  - Verificabilità
- Prove per la stabilità

Portabilità

- Verifica statica come strumento ideale
- Liste di controllo (norme di codifica)
  - Adattabilità
  - Aderenza alle prescrizioni
- Prove come strumento complementare
  - Installabilità
  - Coesistenza
  - Rimpiazzabilità

Riepilogo

- Caratteristiche delle prove del software
- Criteri funzionali
- Criteri strutturali
- Oracolo
- Verifica di requisiti non funzionali