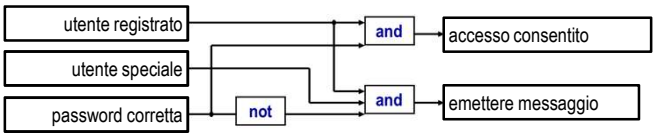
Progettazione delle prove (2)

Roberta Gori, Laura Semini
Ingegneria del Software
Dipartimento di Informatica
Università di Pisa

Grafi causa-effetto

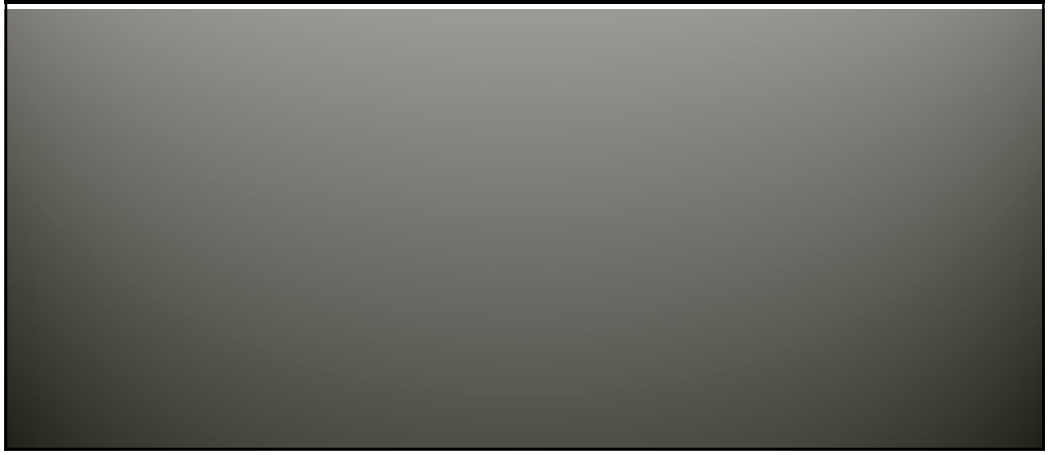
- Requisiti
  - l'accesso è consentito se l'utente è registrato e la password è corretta, è negato in ogni altro caso è negato
  - se l'utente è speciale e la password è errata viene emesso un messaggio sulla console di sistema



- 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

Sono criteri per l'individuazione dei casi di input che si basano sulla struttura del codice

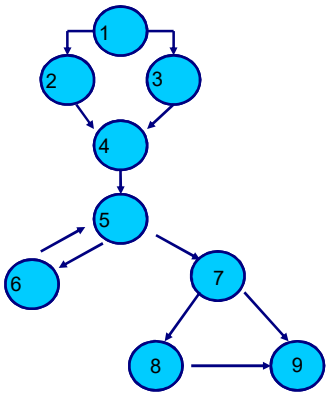


Grafo di flusso

- 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

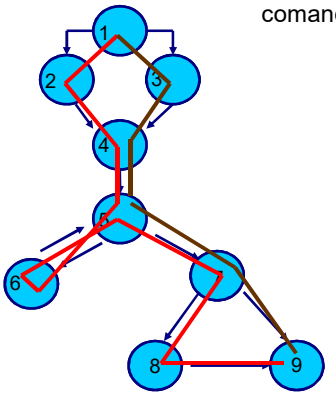
```
double eleva(int x, int y) {
   1. if (y<0)
   2.    pow = 0-y;
   3.    else pow = y;
   4. z = 1.0;
   5. while (pow!=0)
   6.    { z = z*x; pow = pow-1 }
   7. if (y<0)
   8.    z = 1.0 / z;
   9. return(z);
}
```



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);
}
```

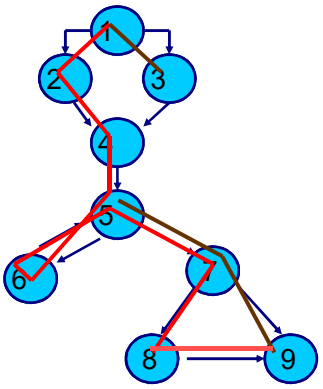
Con due coppie di valori di input si può avere il 100% di copertura dei comandi:
basta una coppia con y<0 e una con y>=0
L'input <3,5> copre il 66,6% dei comandi
L'input <5,-3> copre l'88,8% dei comandi



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

Copertura delle condizioni

- si consideri il codice
  if (x>1 && y==0) {comando1}
  else {comando2}

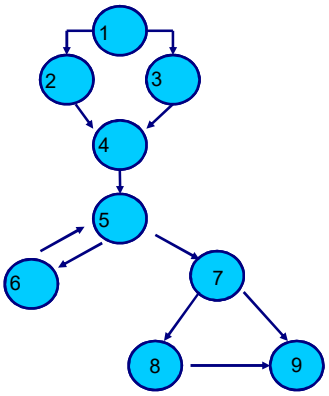
- Il test {x=2, y=0} e {x=2, y=1} garantisce la piena copertura delle decisioni, ma non esercita tutti i valori di verità delle due condizioni in &&
- Il test {x=2, y=2} e {x=0, y=0} esercita tutti i valori di verità delle due condizioni in && (ma non tutte le decisioni)
- Il test {x=2, y=0} e {x=0, y=2} esercita tutti i valori di verità delle due condizioni in && (e tutte le decisioni)

Multiple condition coverage

- si consideri il codice
  if (x>1 && y==0 && z>3) {comando1}
  else {comando2}
- La multiple condition coverage richiede di testare tutte le possibili combinazioni (2^3)
- In presenza di && ci si può ridurre a 4 casi:
  - vero, vero, vero
  - vero, vero, falso
  - vero, falso, -
  - falso, - , -

Copertura dei cammini

- Richiede di percorrere tutti i cammini
- In presenza di cicli il numero di cammini è potenzialmente infinito
- Per limitare il numero di cammini da attraversare:
  - Si fissa un limite al numero di cicli
- Ma comunque può essere esponenziale (sulle scelte)
- Alcuni camini impossibili (1245679)



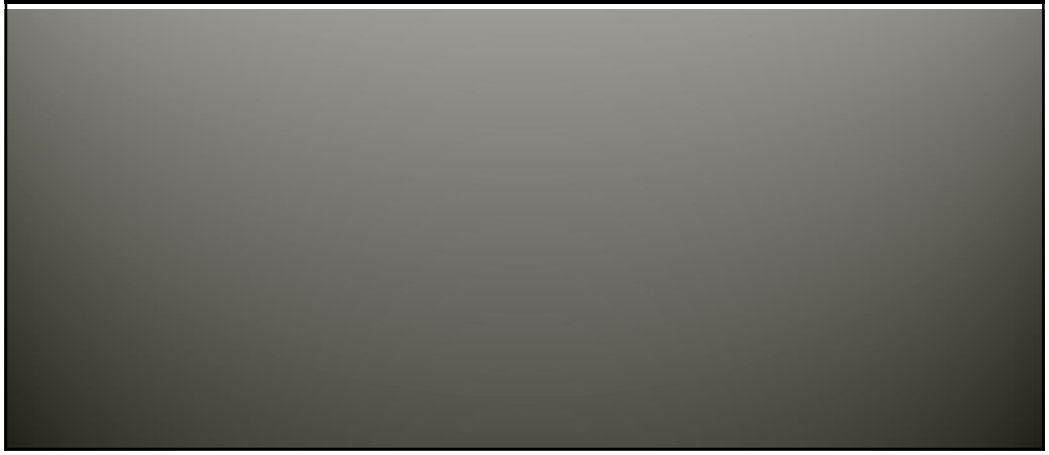
Criteri funzionali vs. strutturali

- 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

Fault based testing



Fault based testing

- Ipotizza dei difetti potenziali nel codice sotto test
- Crea o valuta una test suite sulla base della sua capacità di rilevare i difetti ipotizzati
- Si iniettano difetti modificando il codice

Test mutazionale

- Dopo aver esercitato *P* su T, si verifica *P* corretto rispetto a T.
- Si vuole fare una verifica più profonda sulla correttezza di *P*: introduco dei difetti (piccoli) su *P* e chiamo il programma modificato *P'*. Questo *P'* viene detto **mutante**.
- Si eseguono su *P'* gli stessi test di T. Il test dovrebbe rilevare gli errori. Se il test non rileva questi errori, allora significa che il test non era valido. Questo è un metodo per valutare la capacità di un test, e vedere se è il caso di introdurre test più sofisticati.

Test mutazionale

- *mutazione*: cambiamento sintattico (un bug inserito nel codice)
- Esempio: modifica (i <o) in (i <= o)
- Un mutante viene *ucciso* se fallisce almeno in un caso di test
- *efficacia di un test* = quantità di mutanti uccisi
- 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.

Mutazioni, esempi

- crp: sostituzione (replacement) di costante per costante
  - ad esempio: da (x <5) a (x <12)
- ror: sostituzione dell'operatore relazionale
  - ad esempio: da (x <= 5) a (x <5)
- vie: eliminazione dell'inizializzazione di una variabile
  - cambia int x = 5; a int x;
- lrc: sostituzione di un operatore logico
  - Ad esempio da & a |
- abs: inserimento di un valore assoluto
  - Da x a |x|

Come sopravvive un mutante

- Un mutante può essere equivalente al programma originale
  - Cambiare (x <0) a (x <= 0) non ha cambiato affatto l'output: La mutazione non è un vero difetto
  - Determinare se un mutante è equivalente al programma originale può essere facile o difficile; nel peggiore dei casi è indecidibile
- Oppure la suite di test potrebbe essere inadeguata
  - Se il mutante poteva essere stato ucciso, ma non lo era, indica una debolezza nella suite di test

Test mutazionale

- 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

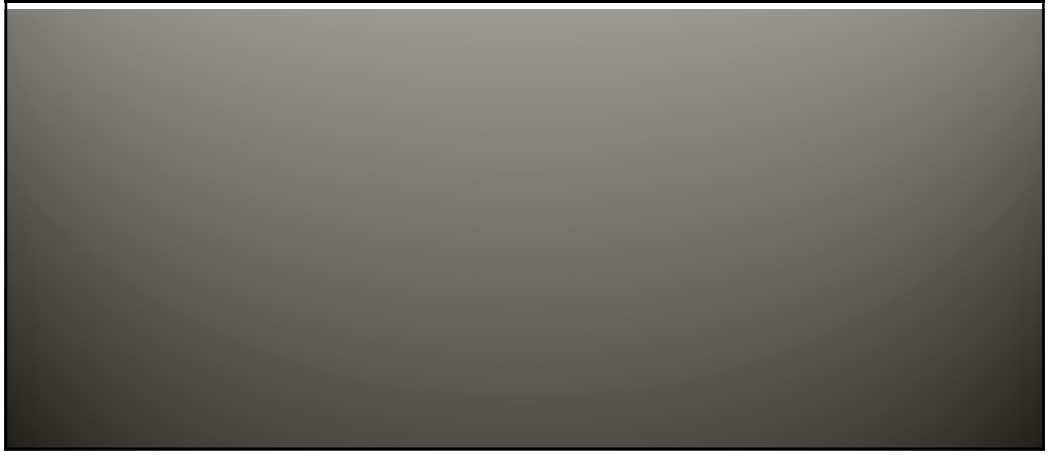
Test di regressione

- 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

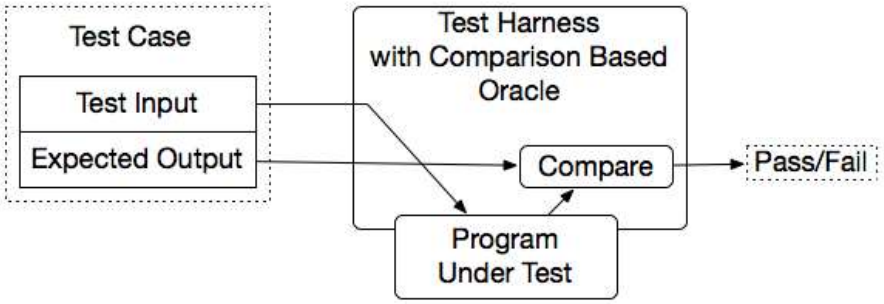
L'oracolo e l'individuazione degli output attesi



Motivazione

- Questo test case ha avuto successo o non ha funzionato?
- Inutile testare automaticamente 10.000 casi di test se i risultati devono essere controllati a mano!
  - ex. JUnit: un oracolo specifico ("assert") codificato a mano in ciascun caso di test
- Approccio tipico: oracolo basato sul confronto con valore di output previsto
- Non l'unico approccio!

Oracolo basato sul confronto



Come trovare l'output atteso

- 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

Come trovare l'output atteso

- 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
  - magari poco efficienti ma corrette

Come trovare l'output atteso

- Semplificazione dei dati d'ingresso
  - provare le funzionalità su dati semplici
  - risultati noti o calcolabili con altri mezzi
  - ipotesi di comportamento costante
- Partire dall'output e trovare l'input
  - Per esempio per testare un algoritmo di ordinamento prendere un array ordinato (output atteso) e rimescolarlo per ottenere un input

Come trovare l'output atteso

- Semplificazione dei risultati
  - accontentarsi di risultati plausibili
  - tramite vincoli fra ingressi e uscite
  - tramite invarianti sulle uscite

Syllabus

- Cap 12-16-17
  - Software Testing and Analysis: Process, Principles and Techniques-
- In particolare:
  - Cap 12: tutto tranne 12.6
  - Cap 16: tutto tranne 16.5
  - Cap 17: in dettaglio solo 17.5
  - Mauro Pezzè e Michal Young