# Criteri strutturali

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

# Perche’ criteri basati sul codice

- Abbiamo visto i criteri funzionali
- I criteri strutturali che vediamo oggi devono aiutare ad aggiungere altri test.
- Rispondono alla domanda:

“Quali altri casi devo  aggiungere per far emergere malfunzionamenti che non sono apparsi con il black-box testing?”

# Gli elementi di un flusso di controllo

- Banalmente potremmo dire che un programma non e' testato adeguatamente se alcuni suoi elementi non vengono mai esercitati dai test.

- I criteri di control flow testing sono definiti per classi particolari di elementi e richiedono che i tests esercitino tutti quegli elementi del programma

- Gli elementi possono essere: comandi, branches, condizioni o cammini.

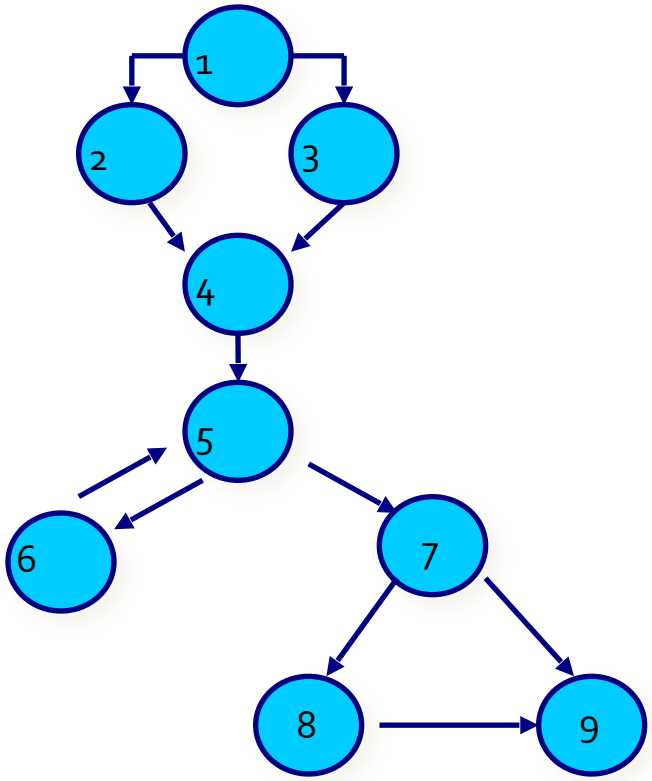
# Grafo di flusso

- Grafo di flusso
  - definisce la struttura del codice identificandone le parti e come sono collegate tra loro
  - 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)

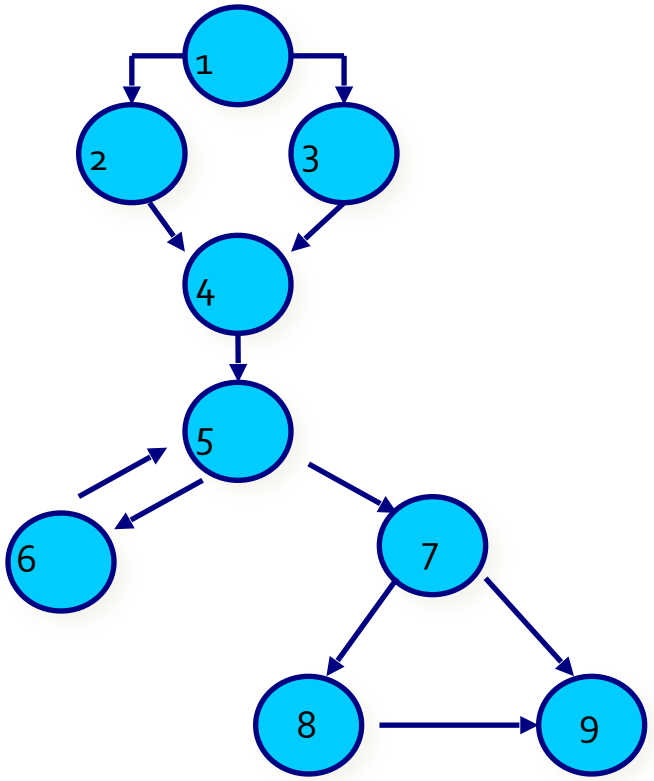
# Un esempio di grafo di flusso

```
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);
}
```




Insieme di valori per x e y che esercitino i comandi

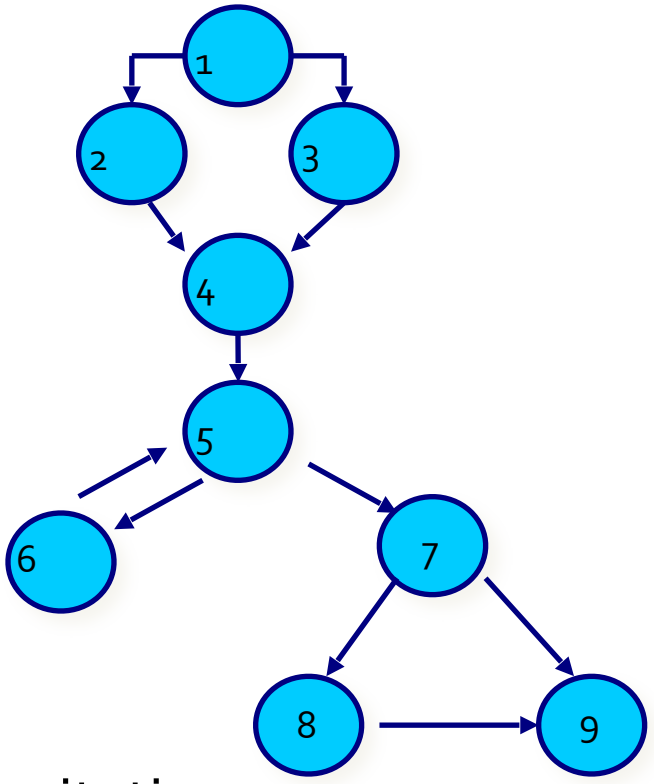
Esempio:

-{(x=2,y=2), (x=0,y=0)}

- {(x=-2,y=3), (x=0, y=-5)}

# Copertura dei comandi: come scegliamo tra insiemi di test

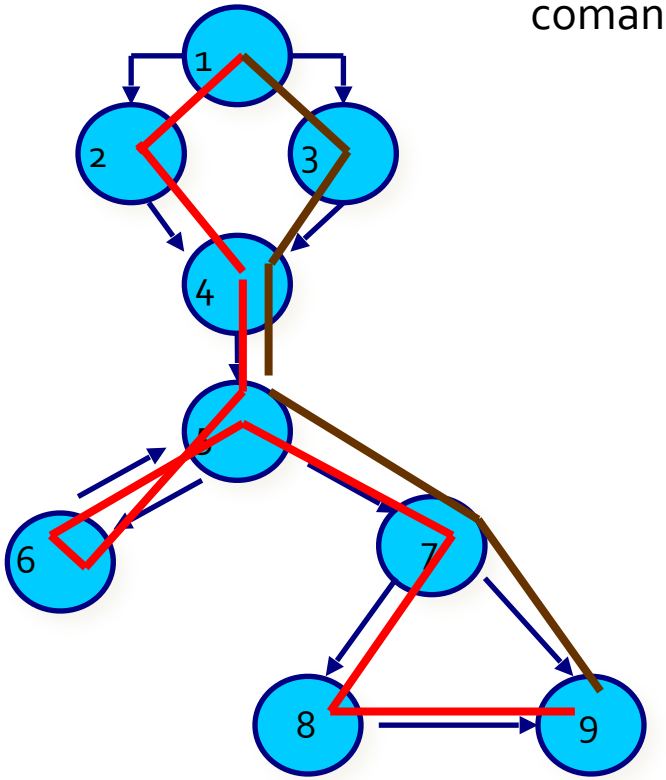
```
double eleva(int x, int y) {
   1. if (y<0)
   2.    pow = 0-y;
   3.    else pow = y;
   4. z = 1.0;
   5. while (pow!=0)
   6.    { z = z*x; pow = pow-1 }
   7. if (y<0)
   8.    z = 1.0 / z;
   9. return(z);
}
```




Misura di copertura = $\frac{\text{numero di comandi esercitati}}{\text{numero di comandi totali}}$

# 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);
}
```




Esiste una copertura totale che si ottiene con solo due casi di test:ur y<0 e uno y>=0.
(x=2,y=-2) esercita i comandi lungo il cammino rosso ed ha una copertura di 8/9=89%

(x=2,y=0) esercita i comandi lungo il cammino marrone ed ha una copertura di 6/9 =66%

{(x=2,y=-2), (x=2,y=0)} ha una copertura di 9/9=100%

# Copertura dei comandi

- La copertura non e' monotona rispetto alla dimensione dell'insieme di test:

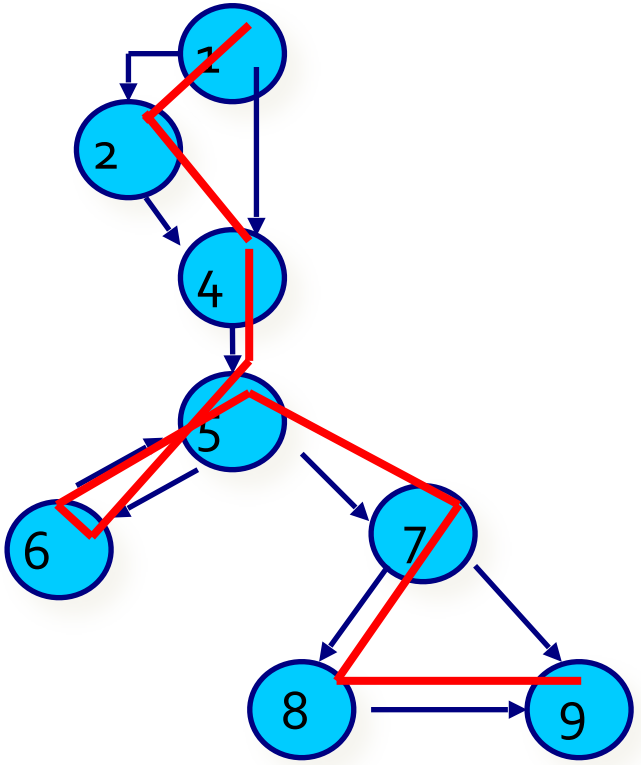
- {(x=4,y=-2)} ha una copertura piu' alta rispetto a {(x=2,y=0), (x=-2,y=2)} !

Ma non sempre vale la pena cercare a tutti i costi un copertura minima!

# 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);
      }
```

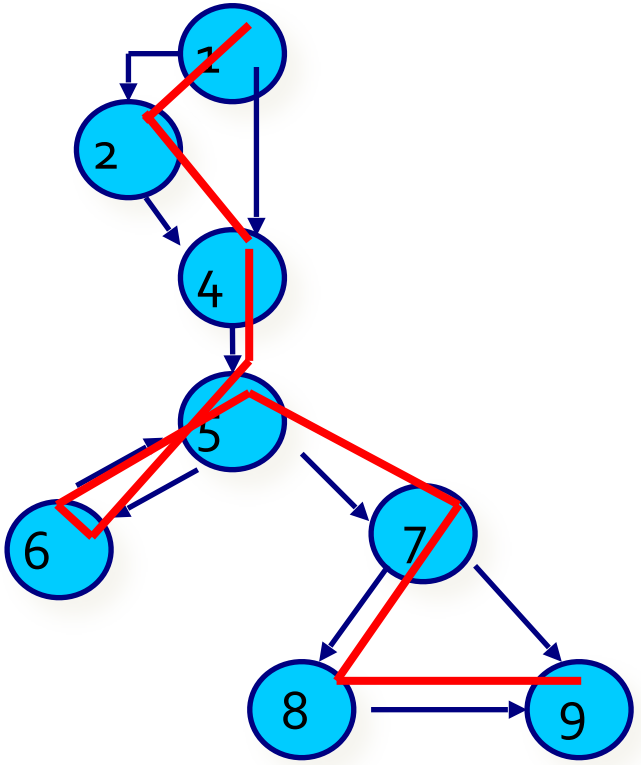



Posso esercitare tutti i comandi con (x=2,y=-1)
ma non mi accorgo del fatto che manca il ramo else!
Devo avere casi di test che esercitino entrambi i rami di una condizione.

# 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);
      }
```




Per avere una copertura delle decisioni devo avere almeno due casi di test uno y<o e uno y>=o. Devo coprire tutte le frecce!

Misura di copertura = $\frac{\text{numero di test esercitati}}{\text{numero di test totali}}$

# Condizioni composte

- si consideri il codice

  if (x>1 || y==0) {comando1}

  else {comando2}

- Il test {x=0, y=0} e {x=0, y=1} garantisce la piena copertura della decisione, ma non esercita tutti i valori di verità della prima condizione

- in particolare avrei potuto voler scrivere x<1 e aver invertito il verso del <

- Il test {x=2, y=2} e {x=0, y=0} esercita i valori di verità delle due condizioni (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)

# Copertura di condizioni semplici

- Un insieme di test T per un programma P copre tutte le condizioni semplici (basic condition) of P se ogni condizione semplice in P ha un test con output atteso true e un test con un test output atteso false

- Copertura delle basic condition=

$$\frac{\text{n. di valori di verita' assunti dalle basic conditions}}{\text{2*n. di basic conditions}}$$

# Multiple condition coverage

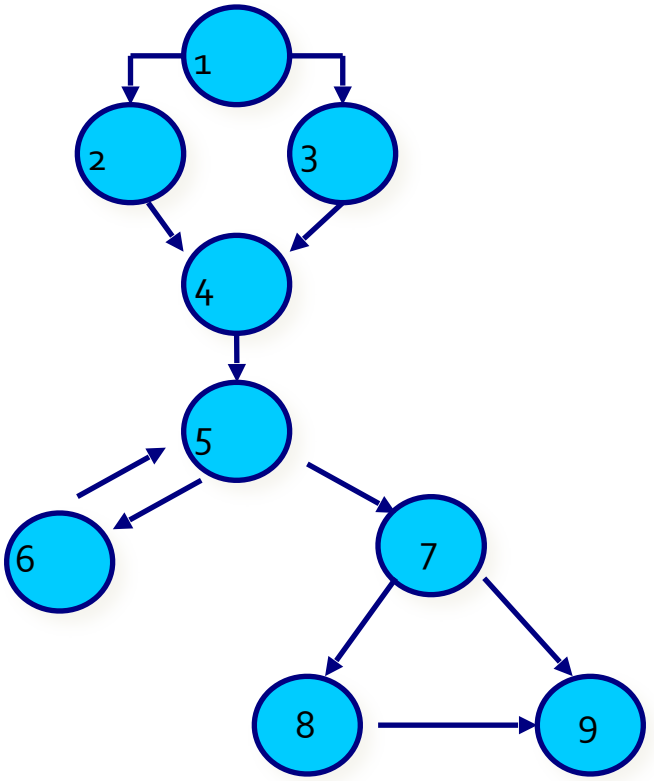
- si consideri il codice

  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 richiedano casi di test che esercitino il ciclo
  - o volte,
  - esattamente una volta
  - piu' di una volta
- Alcuni cammini 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

# Dobbiamo contare i pesci in un lago

# Contiamo i pesci nel lago

Mettiamo  M pesci meccanici nel lago con un numero imprecisato di pesci

Ne guardiamo N di questi
N1 sono quelli meccanici

# Contiamo…

Ne avevamo messi M meccanici

Ne guardiamo N.

Di questi  N1 sono quelli meccanici

$$N1 : N = M : Total$$

$$total = \frac{N^* M}{N1}$$

# contiamo i bug in un programma

```
void transduce() {
  #define BUFLEN 1000
  char buf[BUFLEN];
  int  pos = 0;
  char inChar;
  int atCR = 0;
  while ((inChar = getchar()) != EOF ) {
    switch (inChar) {
    case LF:
      if (atCR) {
        atCR = 0;
      } else {
        emit(buf, pos);
        pos = 0;
      }
      break;
    case CR:
      emit(buf, pos);
      pos = 0;
      atCR = 1;
      break;
    default:
      if (pos >= BUFLEN-2) fail("Buffer overflow");
      buf[pos++] = inChar;
    }
  }
  if (pos > 0) {
    emit(buf, pos);
  }
}
```

contiamo i bug  in un programma che converte il fine linea secondo le convenzioni Machintosh, DOS, Unix.

```
void transduce() {
  #define BUFLEN 1000
  char buf[BUFLEN];
  int  pos = 0;
  char inChar;
  int atCR = 0;
  while ((inChar = getchar()) != EOF ) {
    switch (inChar) {
    case LF:
      if (atCR) {
        atCR = 0;
      } else {
        emit(buf, pos);
        pos = 0;
      }
      break;
    case CR:
      emit(buf, pos);
      pos = 0;
      atCR = 1;
      break;
    default:
      if (pos >= BUFLEN-2) fail("Buffer overflow");
      buf[pos++] = inChar;
    }
  }
  if (pos > 0) {
    emit(buf, pos);
  }
}
```

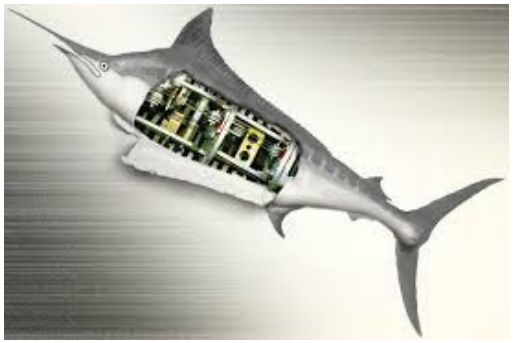
“seminiamo” un errore

(pos => 0)

```
void transduce() {
  #define BUFLEN 1000
  char buf[BUFLEN];
  int  pos = 0;
  char inChar;
  int atCR = 0;
  while ((inChar = getchar()) != EOF ) {
    switch (inChar) {
    case LF:
      if (atCR) {
        atCR = 0;
      } else {
        emit(buf, pos);
        pos = 0;
      }
      break;
    case CR:
      emit(buf, pos);
      pos = 0;
      atCR = 1;
      break;
    default:
      if (pos >= BUFLEN-2) fail("Buffer overflow");
      buf[pos++] = inChar;
    }
  }
  if (pos > 0) {
    emit(buf, pos);
  }
}
```

Troviamo l'errore con un caso di test
Una linea terminata dal ritorno linea DOS

# Assunzione



```
void transduce() {
  #define BUFLEN 1000
  char buf[BUFLEN];
  int  pos = 0;
  char inChar;
  int atCR = 0;
  while ((inChar = getchar()) != EOF ) {
    switch (inChar) {
    case LF:
      if (atCR) {
        atCR = 0;
      } else {
        emit(buf, pos);
        pos = 0;
      }
      break;
    case CR:
      emit(buf, pos);
      pos = 0;
      atCR = 1;
      break;
    default:
      if (pos >= BUFLEN-2) fail("Buffer overflow");
      buf[pos++] = inChar;
    }
  }
  if (pos > 0) {
    emit(buf, pos);
  }
}
```

# assunzione

gli errori che mettiamo sono rappresentativi di quelli che potrebbero esserci davvero

(pos => 0)

# 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, dette **mutazion**i) 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.
- Se li rivela, si suppone che ne riveli anche altri.
  - 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.

# Ipotesi del programmatore Competente

Gli errori reali sono **piccole variazioni sintattiche** del programma corretto

=>

Mutanti sono modelli ragionevoli dei programmi con difetti

# Mettendo insieme tutte le ipotesi

Tests che trovano semplici difetti allora trovano anche difetti piu' complessi

una test suite che uccide i mutanti e' capace anche di trovare difetti reali nel programma

# 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|

# Mutazioni per il C

| ID | Operator | Description | Constraint |
|---|---|---|---|
| *Operand Modifications* | | | |
| crp | constant for constant replacement | replace constant $C1$ with constant $C2$ | $C1 \neq C2$ |
| scr | scalar for constant replacement | replace constant $C$ with scalar variable $X$ | $C \neq X$ |
| acr | array for constant replacement | replace constant $C$ with array reference $A[I]$ | $C \neq A[I]$ |
| scr | struct for constant replacement | replace constant $C$ with struct field $S$ | $C \neq S$ |
| svr | scalar variable replacement | replace scalar variable $X$ with a scalar variable $Y$ | $X \neq Y$ |
| csr | constant for scalar variable replacement | replace scalar variable $X$ with a constant $C$ | $X \neq C$ |
| asr | array for scalar variable replacement | replace scalar variable $X$ with an array reference $A[I]$ | $X \neq A[I]$ |
| ssr | struct for scalar replacement | replace scalar variable $X$ with struct field $S$ | $X \neq S$ |
| vie | scalar variable initialization elimination | remove initialization of a scalar variable | |
| car | constant for array replacement | replace array reference $A[I]$ with constant $C$ | $A[I] \neq C$ |
| sar | scalar for array replacement | replace array reference $A[I]$ with scalar variable $X$ | $A[I] \neq X$ |
| cnr | comparable array replacement | replace array reference with a comparable array reference | |
| sar | struct for array reference replacement | replace array reference $A[I]$ with a struct field $S$ | $A[I] \neq S$ |
| *Expression Modifications* | | | |
| abs | absolute value insertion | replace $e$ by $\mathsf{abs}(e)$ | $e < 0$ |
| aor | arithmetic operator replacement | replace arithmetic operator $\psi$ with arithmetic operator $\phi$ | $e_1 \psi e_2 \neq e_1 \phi e_2$ |
| lcr | logical connector replacement | replace logical connector $\psi$ with logical connector $\phi$ | $e_1 \psi e_2 \neq e_1 \phi e_2$ |
| ror | relational operator replacement | replace relational operator $\psi$ with relational operator $\phi$ | $e_1 \psi e_2 \neq e_1 \phi e_2$ |
| uoi | unary operator insertion | insert unary operator | |
| cpr | constant for predicate replacement | replace predicate with a constant value | |
| *Statement Modifications* | | | |
| sdl | statement deletion | delete a statement | |
| sca | switch case replacement | replace the label of one case with another | |
| ses | end block shift | move } one statement earlier and later | |

# Mutanti Validi e Utili

- Un mutante e' invalido se non e' sintatticamente corretto, cioe' se non passa la compilazione, e' valido altrimenti
- Un mutante e' utile se e' valido e distinguerlo dal programma originale non e' facile, cioe' se esiste solo un piccolo sottoinsieme di test che permette di distinguerlo dal programma originale.

- Trovare mutazioni che producano mutanti validi e utili non e' facile e dipende dal linguaggio!

# Come sopravvive un mutante

- Un mutante può essere equivalente al programma originale
  - Cambiare (x <o) a (x <= o) 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

# Mutanti equivalenti

```
public void insertionSort(int [] array, int min, int max) {}
   for(int i = min+1; i < max; i++) {
      int x = i;
      int j = i-1;
      for(; j >= min; j--) {
          if(array[j]>array[x]) {
              int k = array[x];
              array[x] = array[j];
              array[j] = k;
              x = j;
          } else break;
      }
  }
}
```

Provide two sets of mutant operators M1 and M2 such that:
1. The mutant obtained by applying M1 is killed by TS
2. The mutant obtained by applying M2 is NOT killed by TS.

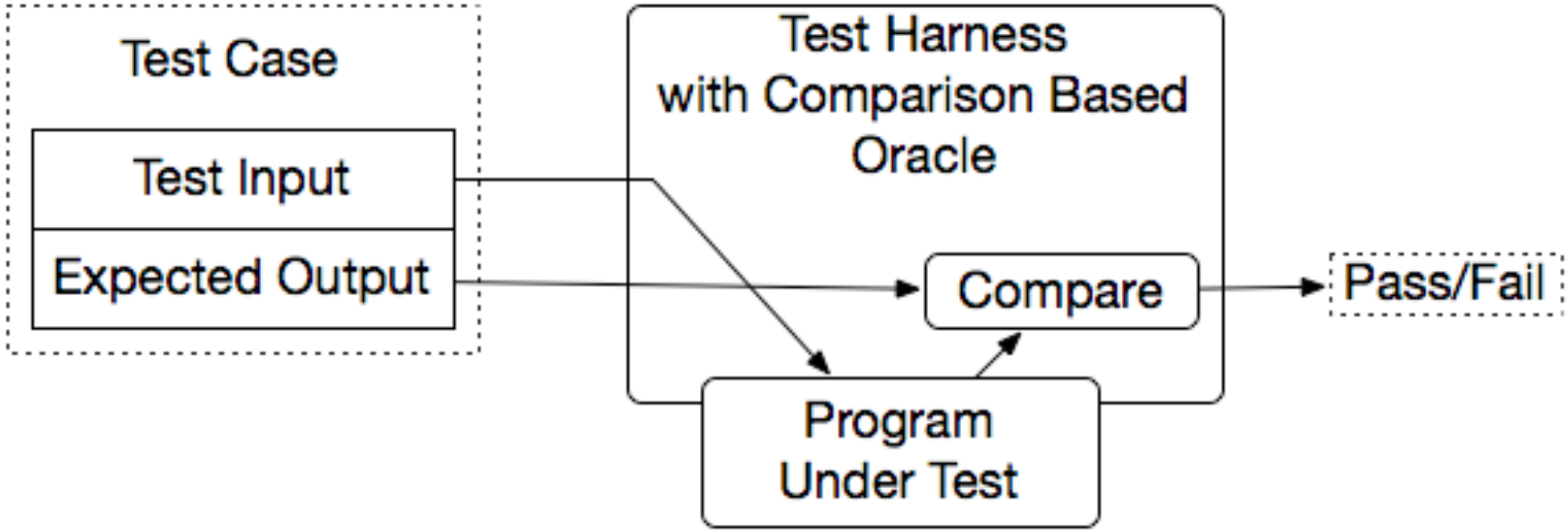
TS = {TC1, TC2}
TC1 = final int[] array = { 5, 9, 0, 2, 7, 3 }; insertionSort(array, 0, 6);
TC2 = final int[] array2 = { 3, 1, 0, 2, 7, 3 }; insertionSort(array2, 2, 4);

# 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
- 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