UNIVERSITÀ DEGLI STUDI
DI MILANO

Verifica e Convalida

Terminologia

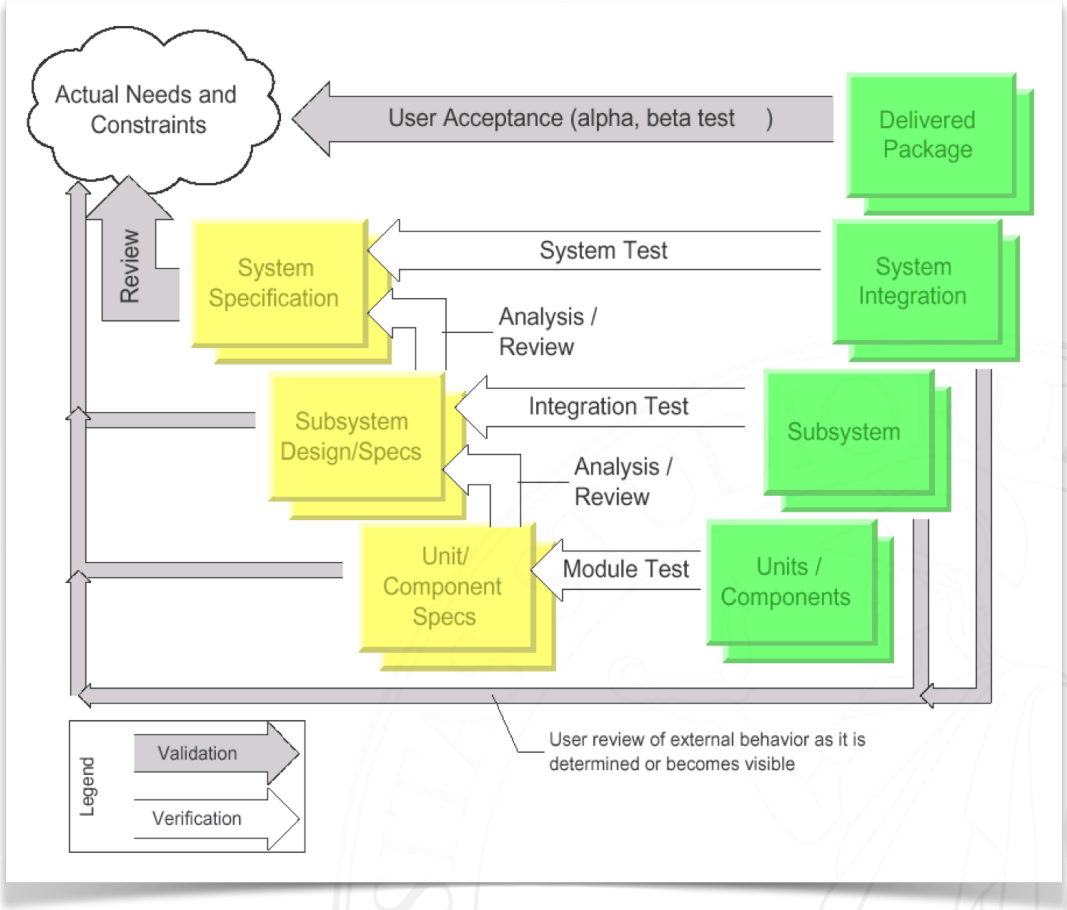
Convalida

- Confronto del software con i **requisiti** (informali) posti dal committente

Verifica

- Confronto del software con le **specifiche** (formali) prodotte dall'analista

Ricordiamo questo modello



Terminologia: *malfunzionamento*

```java
static int raddoppia (int par) {
  int risultato;
  risultato = (par * par);
  return risultato;
}
```

Malfunzionamento o Guasto (Failure)

- Funzionamento non *corretto* di un programma.
- È legato al **funzionamento** del programma e **non al suo codice**

Es. invocando la funzione con parametro 3, ottenere il valore 9 è un malfunzionamento del programma raddoppia

Terminologia: *difetto*

```
1  static int raddoppia (int par) {
2    int risultato;
3    risultato = (par * par);
4    return risultato;
5  }
```

Difetto o Anomalia (Fault)

- È **legato al codice** ed è condizione *necessaria* (ma non *sufficiente*) per il verificarsi di un malfunzionamento

Es. il difetto che causa il malfunzionamento segnalato nel lucido precedente si trova nella riga dell'assegnamento (3)

Terminologia: *sbaglio*

```
1  static int raddoppia (int par) {
2    int risultato;
3    risultato = (par * par);
4    return risultato;
5  }
```

Sbaglio (Mistake)

- È la causa di una anomalia. In genere si tratta di un errore umano (concettuale, battitura, scarsa conoscenza del linguaggio)

Es.

- *battitura:* ha scritto '*' invece di '+'
- *concettuale:* non sa cosa vuol dire raddoppiare
- *padronanza del linguaggio:* credeva che '*' fosse simbolo dell'addizione

https://youtu.be/PK_yguLapgA?t=67

Caso Ariane 5

MALFUNZIONAMENTO

- Il 4 giugno 1996, dopo 40 secondi dal lancio, il razzo Arianne 5 esce dalla sua traiettoria ad una altezza di 3700 metri e esplode

- Dopo una indagine accurata sulle cause, si trova che il problema era nel sistema di controllo di volo ed in particolare del sistema di riferimento inerziale che ha smesso di funzionare dopo circa 36 secondi.

https://web.archive.org/web/20000815230639/http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html

Caso Ariane 5

ANOMALIA

- Il malfunzionamento si è verificato per una eccezione di overflow sollevatasi durante una conversione di un 64-bit float a un 16-bit signed int del valore della velocità orizzontale. Questo ha bloccato sia l'unità principale che di backup.

SBAGLIO

- Tale eccezione non veniva gestita perché questa parte del software era stata ereditata da Ariane 4, la cui traiettoria faceva sì che non si raggiungessero mai velocità orizzontali non rappresentabili con int 16 bit
- La variabile incriminata non veniva protetta per gli "ampi" margini di sicurezza

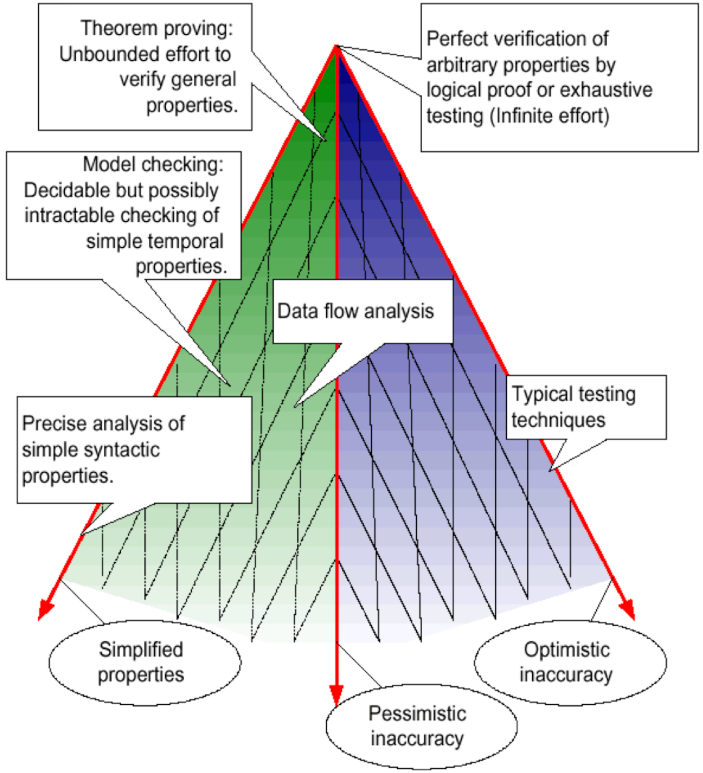
Considerazioni

- Il comportamento della variabile non era mai stato analizzato con dati relativi alla traiettoria da seguire
- I dati relativi alla traiettoria di Ariane V non sono stati considerati nell'analisi dei requisiti nè nelle specifiche
- Il meccanismo delle eccezioni era focalizzato su problemi hw e non sw (shutdown e utilizzo unità di back-up)

Terminologia: statico vs dinamico

- Tecniche **statiche**: sono basate sull'analisi del codice
  - Metodi Formali
  - Analisi Data Flow
  - Modelli statistici
- Tecniche **dinamiche**: sono basate sull'esecuzione del programma eseguibile
  - Testing
  - Debugging

Classificazione delle tecniche



Metodi formali

- Tecniche che si prefiggono di provare l'assenza di anomalie nel prodotto finale

Es.

- Analisi DataFlow
- Dimostrazione di correttezza delle specifiche logiche

Inaccuratezza pessimistica : se non riesce a dimostrare assenza problema dice che non va bene

Testing

- Tecniche che si prefiggono di rilevare **malfunzionamenti**…
- o fornire fiducia nel prodotto (*test di accettazione*): non ho trovato malfunzionamenti quindi…

Es.

- White Box
- Black Box
- Gray Box

Inaccuratezza ottimistica : se non riesce a dimostrare presenza di problemi dice che va bene

Debugging

- Tecniche che si prefiggono di localizzare le **anomalie** che causano malfunzionamenti rilevati in precedenza.

Es.

- Approccio incrementale: permette di limitare la parte in cui ricercare il difetto
- Produzione degli stati intermedi dell'esecuzione del programma

UNIVERSITÀ DEGLI STUDI
DI MILANO

Testing

Correttezza di un programma

- Consideriamo un generico programma P come una funzione da un insieme di dati D (dominio) a un insieme di dati R (codominio)
- $P(d)$ indica l'esecuzione di P sul dato in ingresso $d \in D$
- Il risultato $P(d)$ è corretto se soddisfa le specifiche, altrimenti scorretto
- $ok(P, d)$ indicherà la correttezza di P per il dato d

P è corretto se e solo se $\forall d \in D \; ok(P, d)$

Test e *caso di test*

- Un ***test*** T per un programma P è un sottoinsieme di D
- Un elemento t di un test T è detto ***caso di test***
- L'esecuzione di un test consiste nell'esecuzione del programma $\forall t \in T$
- Un programma **passa** o **supera** un test:

$$ok(P, T) \leftrightarrow \forall t \in T \; ok(P, t)$$

- Un test T ha **successo** se rileva uno o più malfunzionamenti presenti nel programma P

$$\textit{successo}(T, P) \leftrightarrow \exists t \in T \; \neg ok(P, t)$$

Test *ideale*

- T è ideale per P se e solo se $\neg successo(T, P) \rightarrow ok(P, D)$ cioè se il superamento del test implica la correttezza del programma

In generale è impossibile trovare un test ideale

Tesi di Dijkstra

- il test di un programma può rilevare la presenza di malfunzionamenti (non dimostrarne l'assenza)
- Non esiste un algoritmo che dato un programma arbitrario P, generi un test ideale finito (Il caso $T = D$ non va considerato)

"Piccola" prova esaustiva

```
class Trivial {
 static int sum(int a, int b)
  { return a + b; }
}
```

- In Java un `int` è espresso su 32 bit
- Il dominio è quindi di cardinalità

$$2^{32} * 2^{32} = 2^{64} \approx 2 * 10^{19}$$

Considerando un tempo di $1ns$ per ogni test

$$2 * 10^{10} \text{ sec...}$$

più di 600 anni

Criterio di selezione

Il ragionamento che facciamo (o le regole che ci diamo) nel selezionare un sottoinsieme di D (sperando che approssimi il test ideale)

affidabile

Un criterio C si dice *affidabile* se presi T1 e T2 in base al criterio C allora o hanno entrambi successo o nessuno dei due ha successo

$$affidabile(C, P) \leftrightarrow$$
$$\big(\forall T1 \in C, \forall T2 \in C \ successo(T1, P) \leftrightarrow successo(T2, P)\big)$$

valido

Un criterio C si dice *valido* se, qualora P non sia corretto, allora esiste almeno un T selezionato in base a C, che ha successo per il programma P

$$valido(C, P) \leftrightarrow \big(\neg ok(P, D) \rightarrow \exists T \in C \ successo(T, P)\big)$$

Esempio

```
1  static int raddoppia (int par) {
2    int risultato;
3    risultato = (par * par);
4    return risultato;
5  }
```

- Un criterio che seleziona "sottoinsiemi di $\{0, 2\}$" è…

  affidabile ma *non valido*

- Un criterio che seleziona "i sottoinsiemi di $\{0, 1, 2, 3, 4\}$" è…

  non affidabile ma *valido*

- Un criterio che seleziona "sottoinsiemi finiti di D con almeno un valore maggiore di 18" è…

  affidabile e *valido*

Attenzione

$$affidabile(C, P) \land valido(C, P) \land T \in C \land \neg successo(T, P)$$

$$\Longrightarrow$$

$$ok(P, D)$$

- Se prendiamo un test selezionato in base a un criterio che sia *affidabile* e *valido*, e il test non trova errori **allora** il programma è corretto.
- Cioè un criterio affidabile e valido selezionerebbe test ideali (che però sappiamo non esistere)

Come ragionare?

Dobbiamo identificare le caratteristiche che rendono *utile* un caso di test.

Possiamo lavorare sulle caratteristiche delle specifiche o del codice

Ad ogni criterio è possibile associare una metrica che ne misuri la *copertura* e che ci permetta di:

- decidere quando smettere
- decidere quale altro caso di test è opportuno aggiungere
- confrontare *bontà* di Test diversi

Quali caratteristiche?

Un caso di test, per poter portare a evidenziare un malfunzionamento causato da una anomalia, deve soddisfare tre requisiti:

1. eseguire il comando che contiene l'anomalia
2. l'esecuzione del comando contenente l'anomalia deve portare il sistema in uno stato scorretto
3. lo stato scorretto deve propagarsi fino all'uscita del codice in esame, in modo da produrre un output diverso da quello atteso

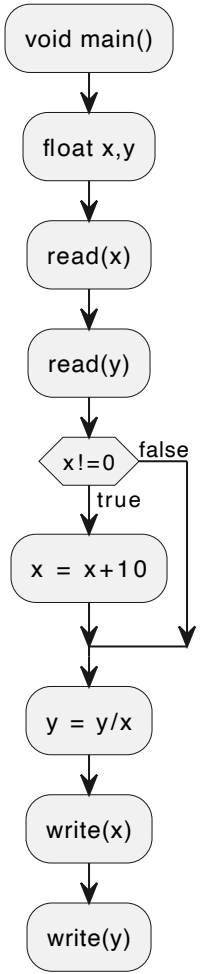
Copertura dei comandi

Un test T soddisfa il criterio di **copertura dei comandi** se e solo se ogni comando eseguibile del programma è eseguito in corrispondenza di almeno un caso di test t contenuto in T

La metrica è la frazione dei comandi eseguiti su quelli eseguibili

Esempio

```
1  void main(){
2    float x,y;
3    read(x);
4    read(y);
5    if (x!=0)
6      x = x+10;
7    y = y/x;
8    write(x);
9    write(y);
10 }
```



Graficamente corrisponde a passare per tutti i nodi (raggiungibili)

Il caso di test $< 3, 7 >$ è un esempio di caso che è già sufficiente

$T = \{< 3, 7 >\}$ è quindi un test che soddisfa il criterio di copertura dei comandi al 100%

Trova tutti i malfunzionamenti?

NO: ad esempio con il caso $< 0, 7 >$ eseguirebbe una divisione per zero

Copertura delle decisioni

Un test T soddisfa il criterio di **copertura delle decisioni** se e solo se ogni decisione effettiva viene resa sia vera che falsa in corrispondenza di almeno un caso di test t contenuto in T

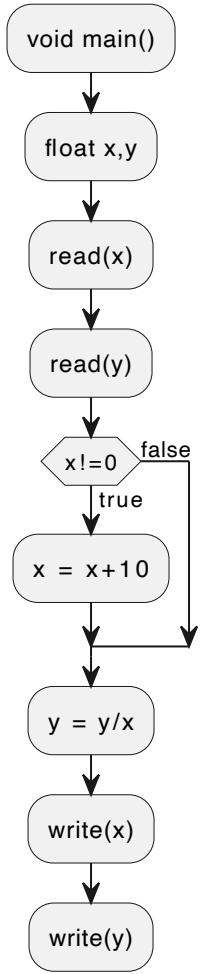
La metrica è la frazione delle decisioni che sono state rese sia vere che false su quelle per cui è possibile farlo

Implica copertura dei comandi

Se un test soddisfa la copertura delle decisioni, allora soddisfa anche la copertura dei comandi (ma non è garantito l'inverso)

Esempio

```
1  void main(){
2    float x,y;
3    read(x);
4    read(y);
5    if (x!=0)
6      x = x+10;
7    y = y/x;
8    write(x);
9    write(y);
10 }
```



Graficamente corrisponde a percorrere tutti gli archi (percorribili)

Ho bisogno di almeno due casi di test: ad esempio $< 3, 7 >$ e $< 0, 5 >$

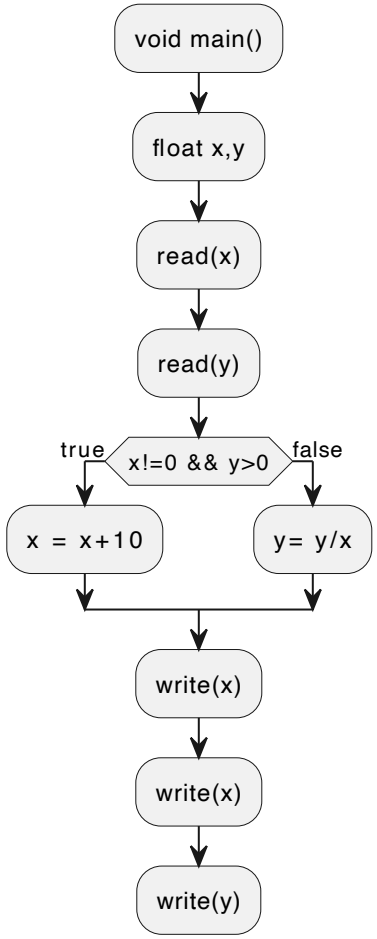
$T = \{< 3, 7 >, < 0, 5 >\}$ è quindi un test che soddisfa il criterio di copertura delle decisioni al 100%

Trova tutti i malfunzionamenti?

NO : ad esempio un possibile overflow alla somma di x alla riga 6

Esempio 2

```
1  void main(){
2    float x,y;
3    read(x);
4    read(y);
5    if (x!=0 && y>0)
6       x = x+10;
7    else
8       y= y/x;
9    write(x);
10   write(y);
11 }
```



Ho aggiunto anche il ramo `else` e ho messo due **condizioni** a comporre una **decisione**

Ho ancora bisogno di almeno due casi di test: ad esempio $< 3, 7 >$ e $< 3, -2 >$ vanno bene

$T = \{< 3, 7 >, < 3, -2 >\}$ è quindi un test che soddisfa il criterio di copertura delle decisioni al 100%

Trova tutti i malfunzionamenti?

NO: ad esempio il caso $< 0, 5 >$ percorrerebbe ramo *false* eseguendo divisione per zero

Copertura delle condizioni

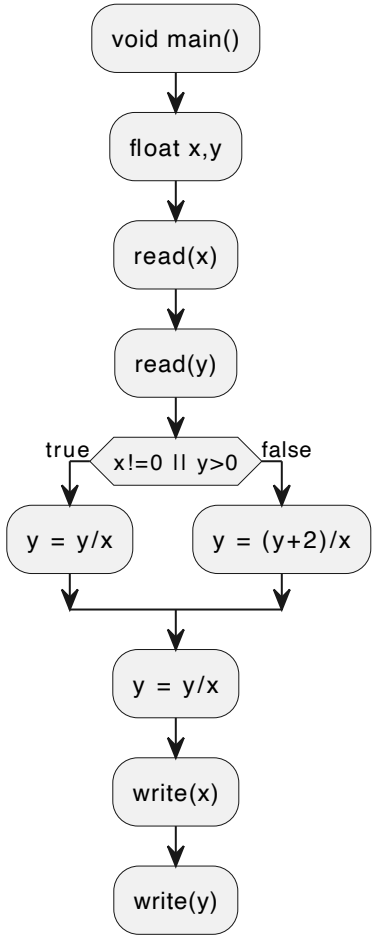
Un test T soddisfa il criterio di **copertura delle condizioni** se e solo se ogni singola condizione (effettiva) viene resa sia vera che falsa in corrispondenza di almeno un caso di test t contenuto in T

La metrica è la frazione delle condizioni che sono state rese sia vere che false su quelle per cui è possibile farlo

NON implica i criteri precedenti

Esempio

```
1  void main(){
2    float x,y;
3    read(x);
4    read(y);
5    if (x!=0 || y>0)
6      y = y/x;
7    else
8      y = (y+2)/x;
9    y = y/x;
10   write(x);
11   write(y);
12 }
```



$T = \{< 0, 5 >, < 5, -5 >\}$ è un test che soddisfa il criterio di copertura delle condizioni al 100%, ma la decisione è sempre vera

X	Y	decisione
0 (F)	5 (T)	T
5 (T)	-5 (F)	T

Ci sono anomalie sia alla riga 6 che 8, ma la seconda non verrebbe trovata non essendo coperta

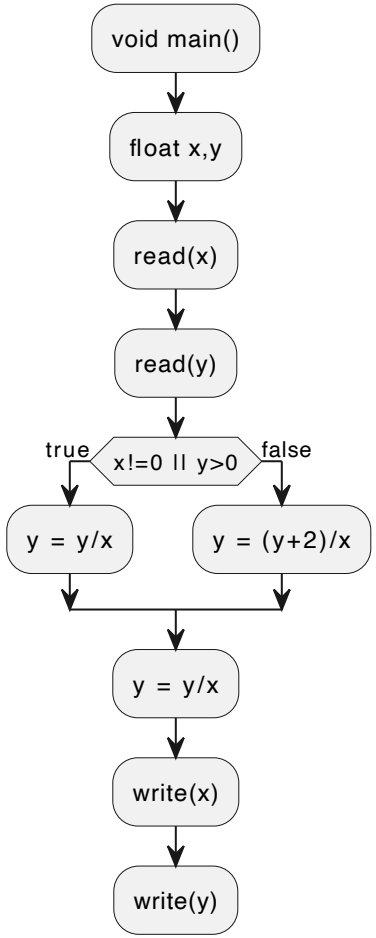
Copertura decisioni e condizioni

Un test T soddisfa il criterio di copertura delle decisioni e delle condizioni se e solo se ogni decisione vale sia vero che falso e ogni singola condizione che compare nelle decisioni del programma vale sia vero che falso per diversi casi di test in T

Questo criterio contiene i due precedenti

Esempio

```
1  void main(){
2    float x,y;
3    read(x);
4    read(y);
5    if (x!=0 || y>0)
6      y = y/x;
7    else
8      y = (y+2)/x;
9    y = y/x;
10   write(x);
11   write(y);
12 }
```



$T = \{< 0, -5 >, < 5, 5 >\}$ è un test che soddisfa il criterio di copertura delle decisioni e condizioni al 100%

X	Y	decisione
0 (F)	-5 (F)	F
5 (T)	5 (T)	T

scopre anomalia in 8 ma non quella in 6

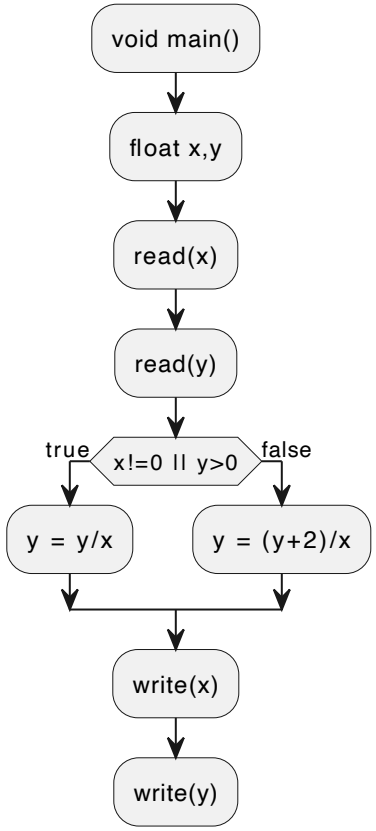
Copertura condizioni composte

Un test T soddisfa il criterio di copertura delle condizioni composte se e solo se ogni possibile composizione delle condizioni base vale sia vero che falso per diversi casi di test in T

Questo criterio contiene il precedente

Esempio

```
1  void main(){
2    float x,y;
3    read(x);
4    read(y);
5    if (x!=0 || y>0)
6      y = y/x;
7    else
8      y = (y+2)/x;
9    y = y/x;
10   write(x);
11   write(y);
12 }
```



X	Y	decisione
0 (F)	-5 (F)	F
0 (F)	5 (T)	T
5 (T)	-5 (F)	T
5 (T)	5 (T)	T

Crescita molto veloce del numero di casi al crescere del numero di condizioni base

Potrebbero esistere combinazioni non fattibili (condizioni base non indipendenti)

Copertura decisioni/condizioni modificate

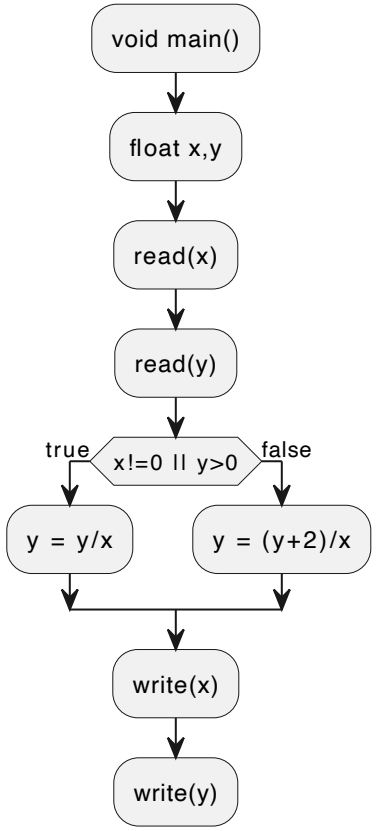
Si dà importanza, nella selezione delle combinazioni, al fatto che la modifica di una singola condizione base porti a modificare la decisione

Cioè devono esistere per ogni condizione base due casi di test che modificano il valore di una sola condizione base e che modificano il valore della decisione

Si può dimostrare che se ho N condizioni base, mi servono "solo" $N + 1$ casi di test.

Esempio

```
1  void main(){
2    float x,y;
3    read(x);
4    read(y);
5    if (x!=0 || y>0)
6      y = y/x;
7    else
8      y = (y+2)/x;
9    y = y/x;
10   write(x);
11   write(y);
12 }
```



X	Y	decisione
0 (F)	-5 (F)	F
0 (F)	5 (T)	T
5 (T)	-5 (F)	T

Tolto caso **vero - vero** perché con un singolo cambio di condizione non era possibile cambiare decisione

Implicazioni tra criteri di copertura