Software testing

Angelo Gargantini

angelo.gargantiniatunibg.it

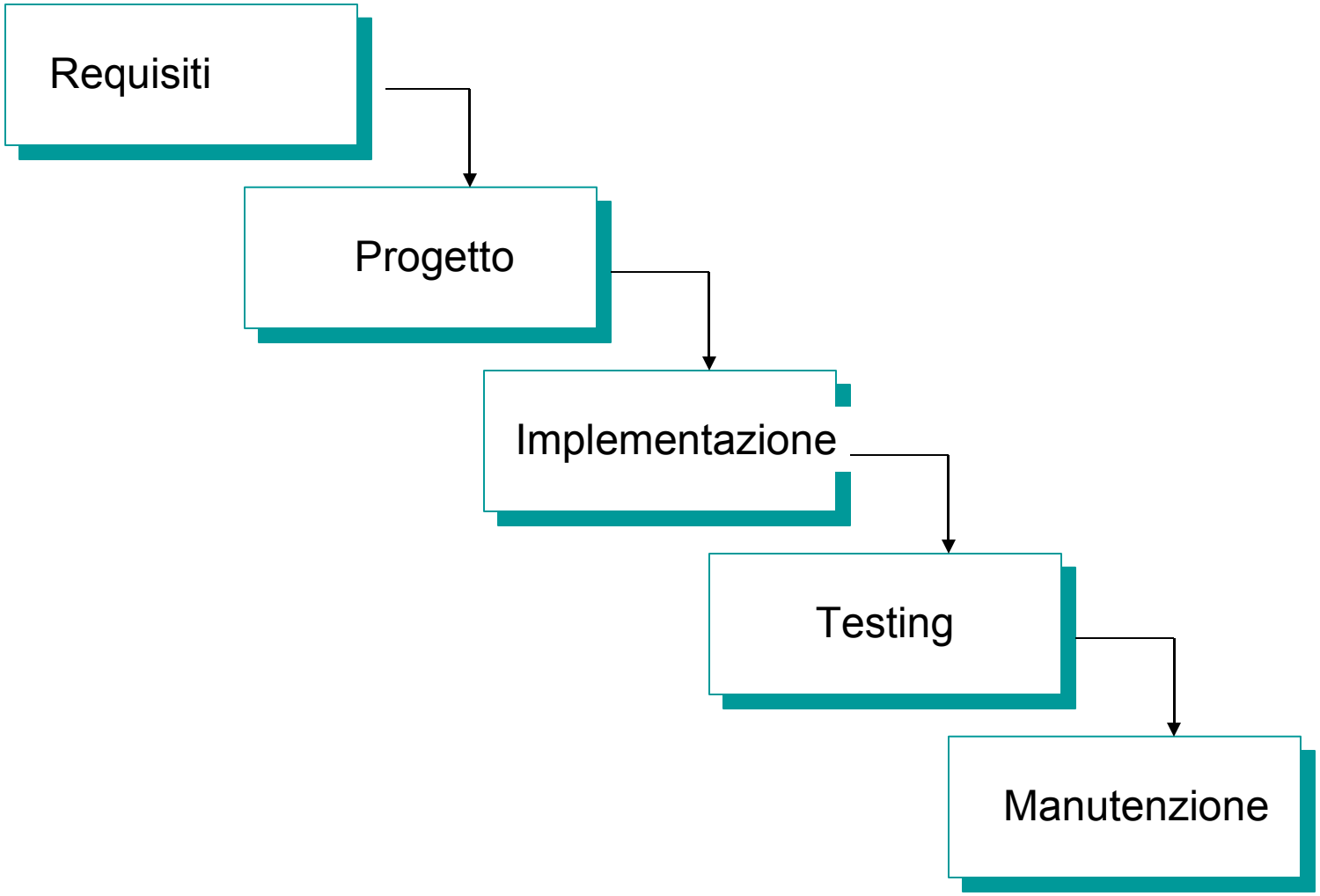
0. Introduzione - Concetti base

- TESTING: cosa serve? Perché?
  - non solo scrivere programmi, non solo progettare, …
- In che contesto si inserisce il testing?
- All’interno del ciclo di vita del sw
- Cosa vuol dire verifica/convalida ?
- Cosa e’ un guasto, errore, un bug ?

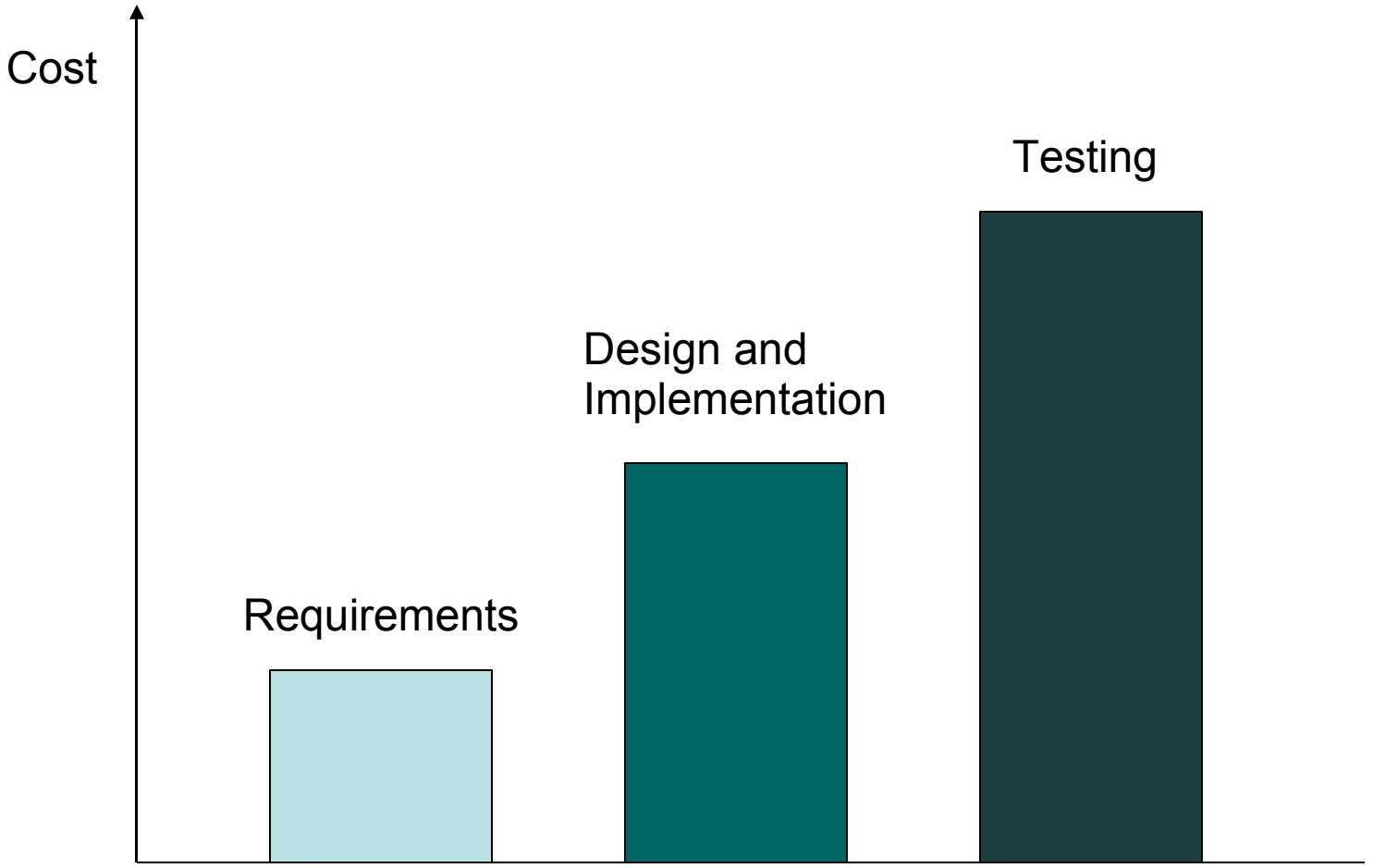
Materiale

- libro di SE o di testing
- http://www.cs.uoregon.edu/~michal/book/

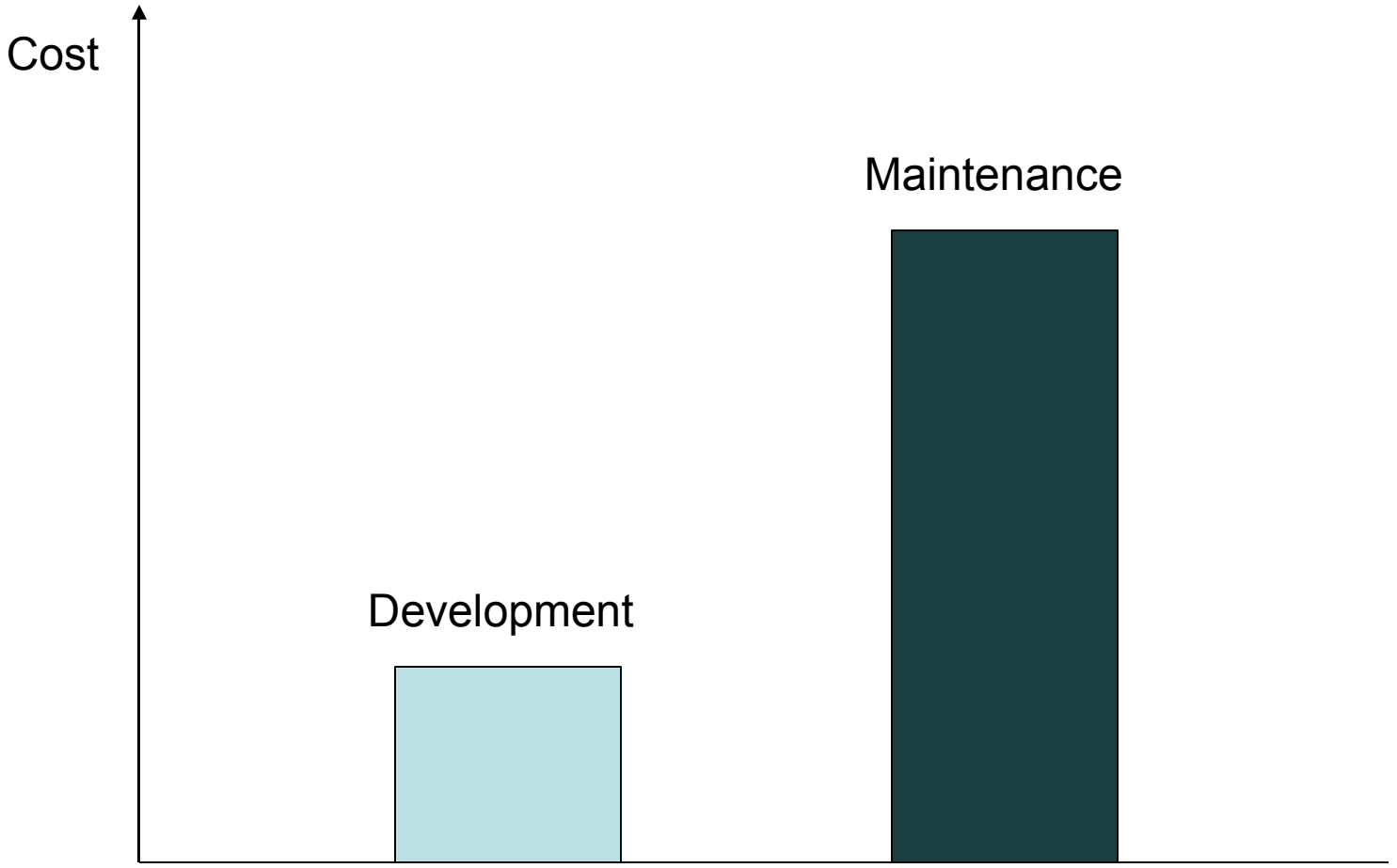
Ciclo di vita del Software



Costi dello sviluppo del Software



Costi nella vita del Software



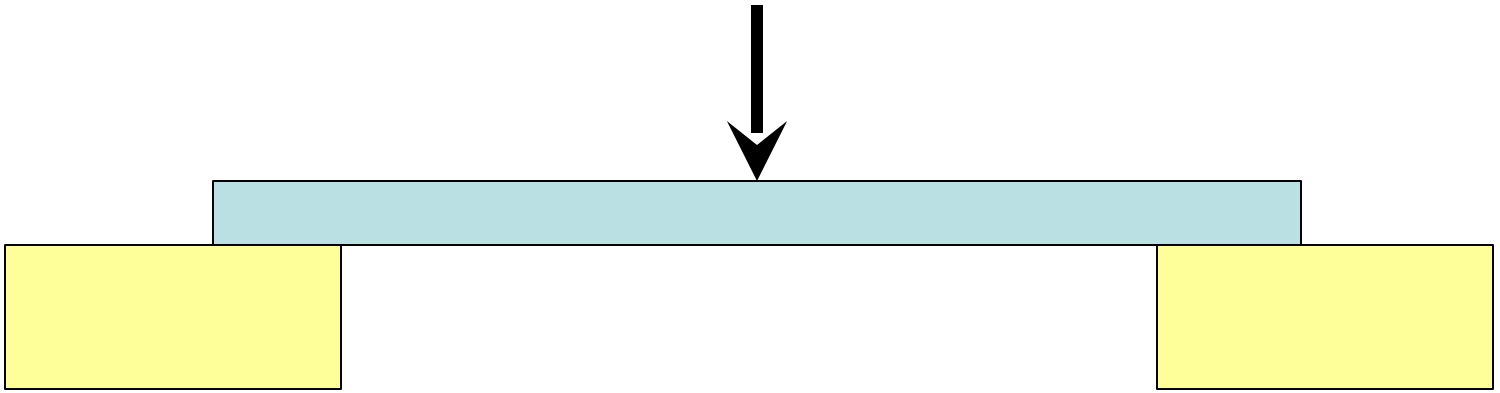
Importanza del testing

- Software senza difetti?
- Ogni parte va verificata (anche la verifica)
- Con profondità diverse
  - sistemi critici
  - sistemi per la produttività individuale

"software industry, the: unique industry where selling substandard goods is legal and you can charge extra for fixing the problems."

Test in Ingegneria

- In ingegneria gli oggetti hanno un comportamento “continuo”:
- Un ponte si testa in un punto particolare e se funziona lì funziona tutt’attorno (o addirittura tutto)



Test del software

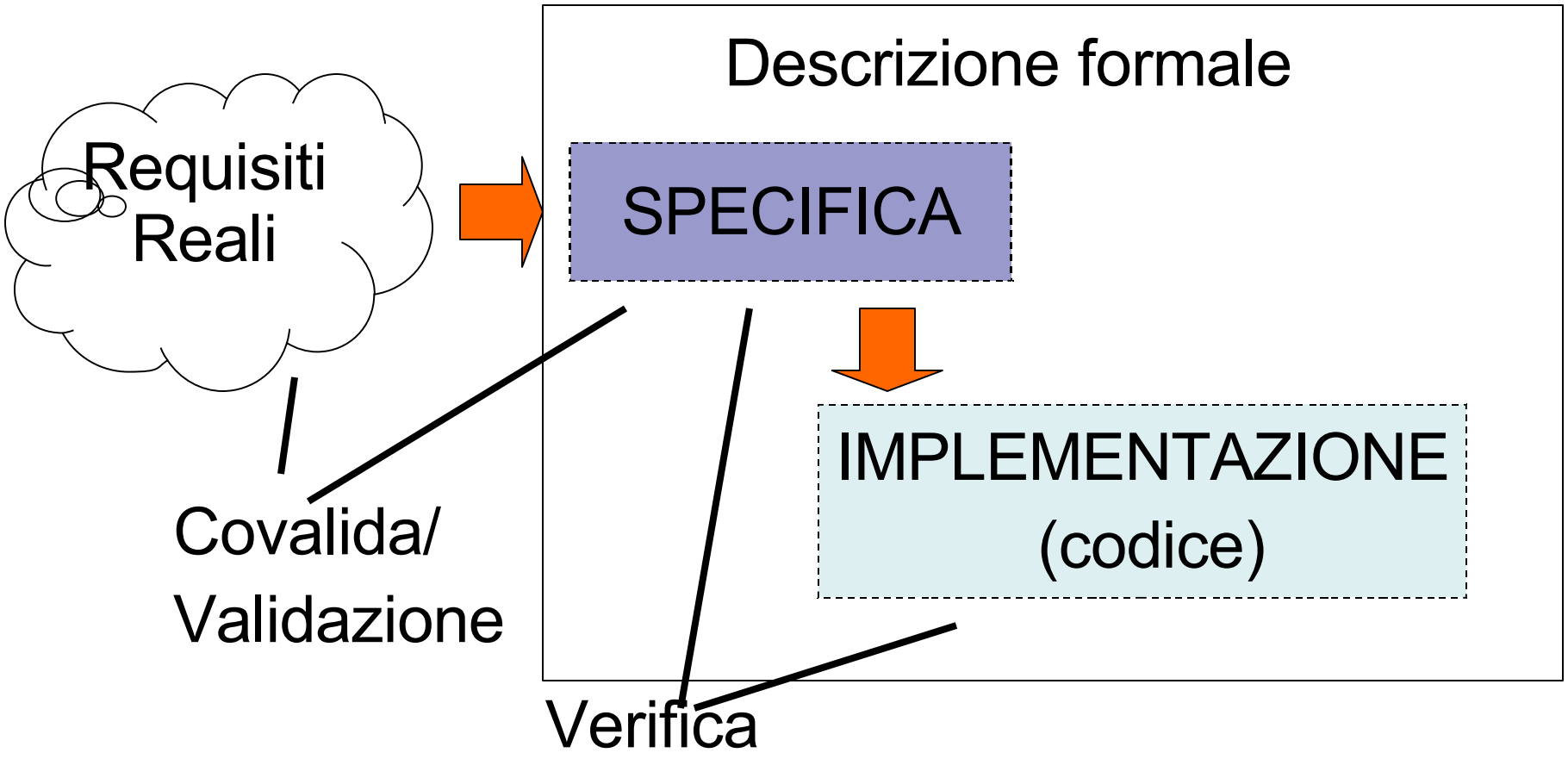
- Il software ha un comportamento molto discontinuo quindi la selezione dei “punti” e’ molto critica
  - Esempio
    a := x / (x+20)

  Quale valore prendo per vedere se funziona?
- Problema dei requisiti – un ponte basta che resista, un software?

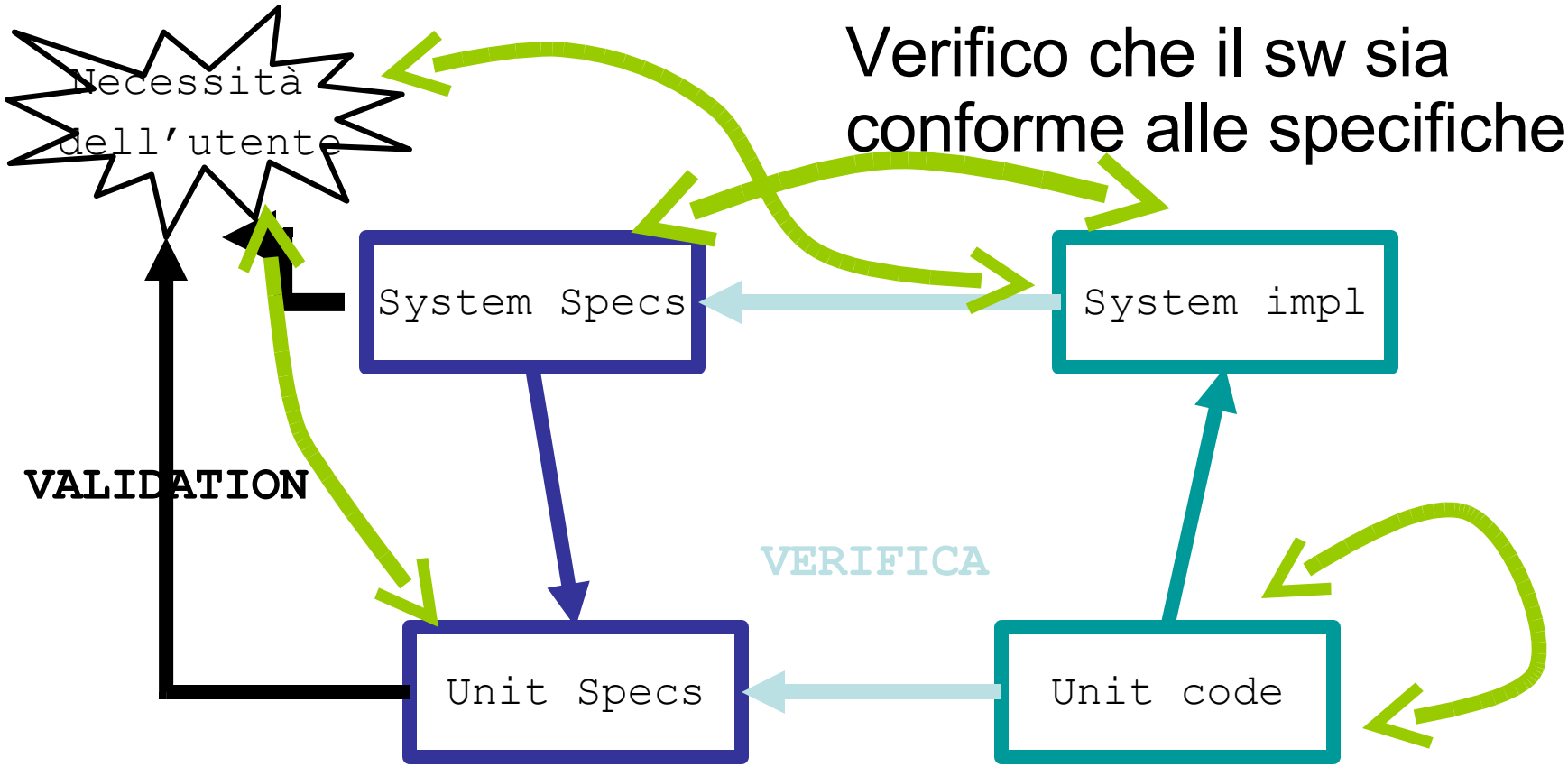
Validazione (Convalida) e Verifica

- validazione: che il sw sia corretto.
  accertamento che un software soddisfi i requisiti dell'utente intesi come i suoi bisogni
  - Si confronta con i requisiti informali espressi dall'utente
  - *building the right system*
- verifica: implementazione corrisponda alla sua specifica
  - *building the system right*

Validazione / verifica



Validazione vs verifica



Testing: tecnica di validazione/verifica mediante l'esecuzione

salta

Tecniche per V&V

- Static
  - Collects information about a software without executing it
    - Reviews, walkthroughs, and inspections
    - Static analysis
    - Formal verification
- Dynamic
  - Collects information about a software with executing it
    - Testing: finding errors
    - Debugging: removing errors

Tecniche per la validazione e la verifica: testing e analisi

- Tecniche di analisi statiche:
  - Il programma viene valutato ma non eseguito (verifica formale, controllo nella compilazione, …)
- testing: analisi dinamica osservazione del comportamento del sistema
- necessità di definire dei criteri di selezione e di valutarli
  - random testing
  - Oppure uso di test criteria

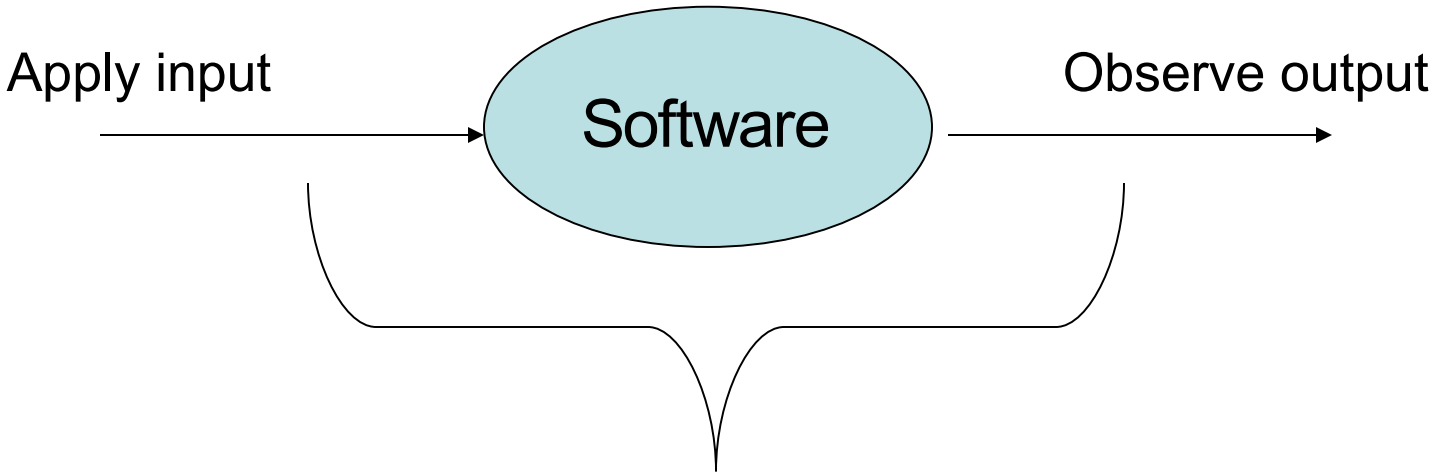
Static Analysis

- Control flow analysis and data flow analysis
  - Extensively used for compiler optimization and software engineering
- Examples
  - Unreachable statements
  - Variables used before initialization
  - Variables declared but never used
  - Variables assigned twice but never used between assignments
  - Variables used twice with no intervening assignment
  - Possible array bound violations

Formal Verification

- Given a model of a program and a property, determine whether the model satisfies the property based on mathematics
- Examples
  - Safety
    - If the light for east-west is green, then the light for south-north should be red
  - Liveness
    - If a request occurs, there should be a response eventually in the future

Testing



Validate the observed output

Is the observed output the same as the expected output?

Failure o guasto o malfunzionamento

- è il funzionamento non corretto del programma (legato al comportamento)
- Esempio:

```
int raddoppia( int x ) {
        return x*x ;
}
```

se chiamo raddoppia(3) noto un malfunzionamento
se chiamo raddoppia(2) non vi e' malfunzionamento

Anomalia o difetto (fault)

- Difetto (fault): elemento del programma non corrispondente alle aspettative (uno o più difetti possono causare malfunzionamenti del software)
  - Riguarda quindi la parte statica.
  - Nell'esempio e' *x invece che *2.
    Nota un programma può avere molti difetti e non presentare alcun malfunzionamento.
- I difetti si chiamano anche “bug”
- Scopo del testing e' quello di evidenziare difetti mediante malfunzionamenti.

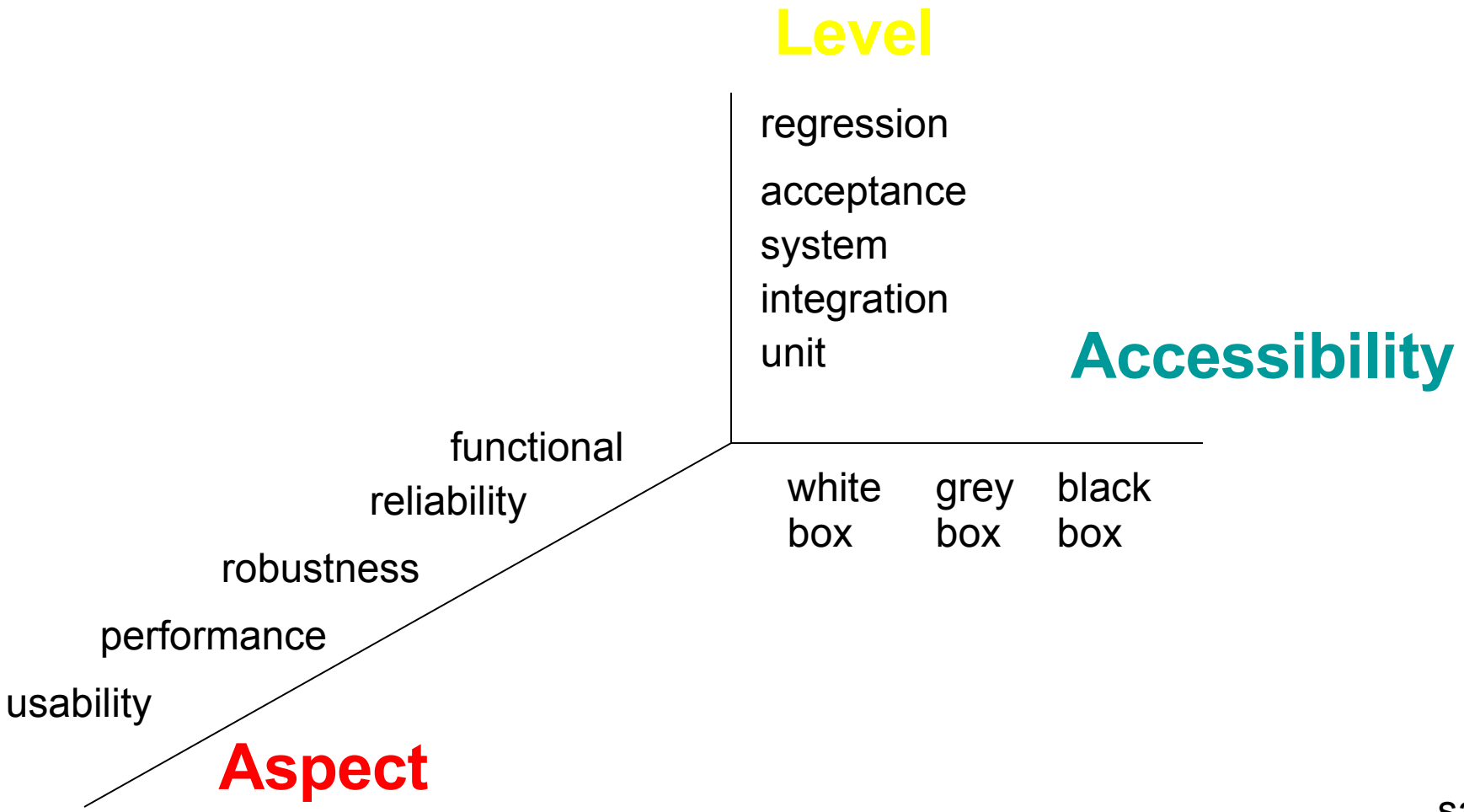
Errore

- Errore: fattore (umano) che causa una deviazione tra il software prodotto e il programma ideale (uno o più errori possono produrre uno o più difetti nel codice)
  - Esempio: errore di analisi dei requisiti, progetto, battitura,...
  - Quando il programmatore commette un errore, il programma ha un difetto che può generare un malfunzionamento

Testing e debugging

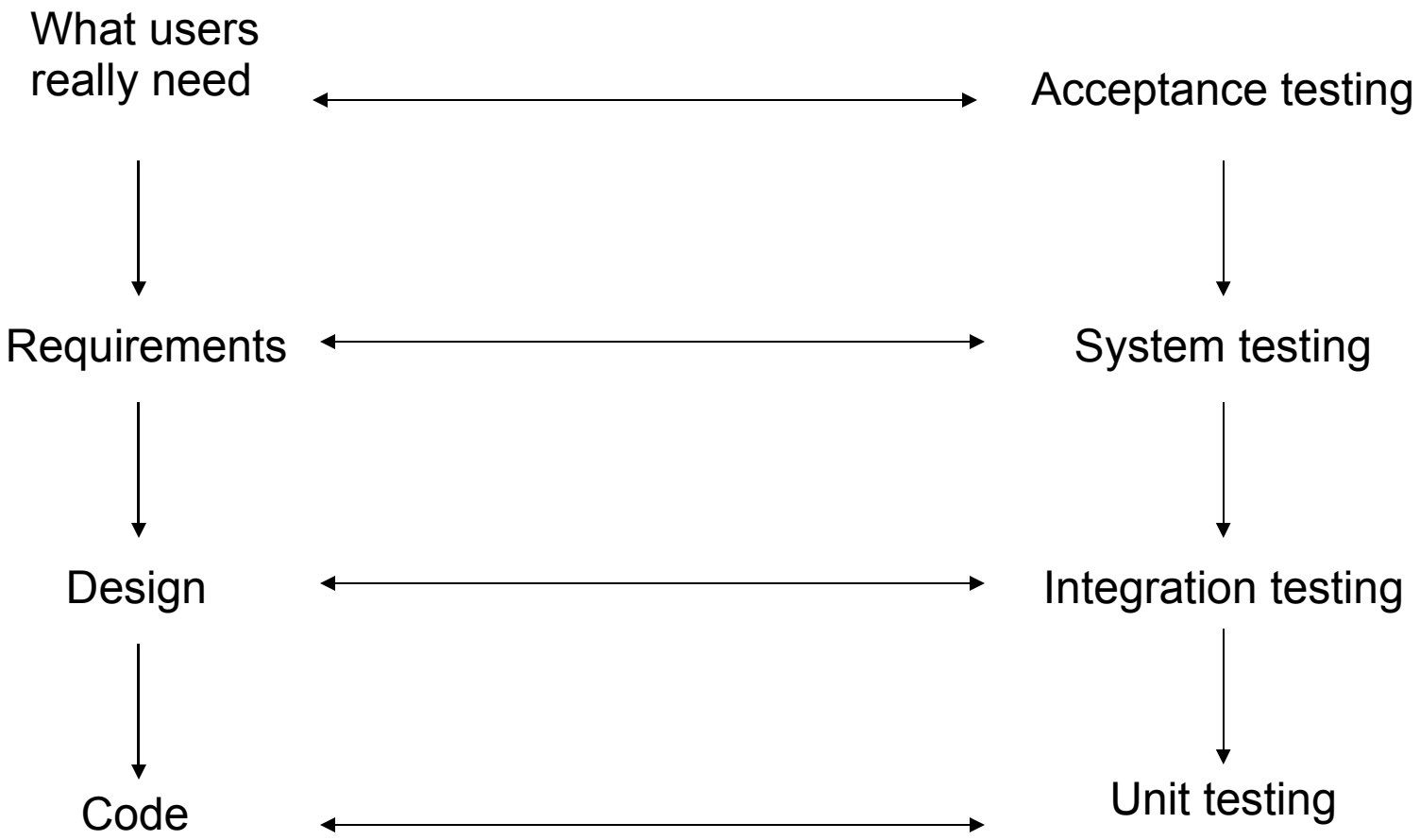
- Testing: scopi del testing
  - mettere in evidenza i difetti (bug) mediante i malfunzionamenti, per scoprire eventuali errori.
    - Cercare quei comportamenti che mettono in evidenza difetti
  - poter valutare l'affidabilità di un sw (reliability) e fornire confidenza (test di accettazione)
    - dell'affidabilità del prodotto e della (probabile) correttezza
    - del aver rilevato (quindi dell'assenza) di particolari tipi di errore
    - Cercare quei comportamenti più critici o più frequenti per controllare che causino errori
- Debugging:

Tipi di testing



salta

Levels of Testing



Livelli di granularità / tipi di testing

- **Test di accettazione**: il comportamento del software è confrontato con i requisiti dell'utente finale
- **Test di conformità**: il comportamento del software (tutto) è confrontato con le specifiche dei requisiti
- **Test di sistema**: controlla il comportamento dell'intero sistema come se fosse monolitico.
- **Test di integrazione**: controllo sul modo di cooperazione delle unità
- **Test di unità**: controllo del comportamento delle singole unità
- **Test di regressione**: controllo del

Unit Testing

- A unit of testing
  - Functions in procedural programming languages such as C, Fortran, …

Test driver

```
F1(int x1, y1) {
        ……
        F2(x1+1, y1-1);
}
```

Test unit

```
F2(int x2, y2) {
        ……
        F3(x2+2, y2-1);
}
```

Test stub

```
F3(int x3, y3) {
        ……
}
```

the test driver simulates a calling unit

the test stub simulates a called unit. (se servono)

Limite del testing

- Testing is **not** about showing that there are no errors in the program.
- Testing **cannot** show that the program performs its intended goal correctly.

So, what *is* software testing?

Testing is the process of executing the program in order to find errors.

A successful test is one that finds an error.

Limite del Testing (I)

- To test all possible inputs is impractical or impossible

```
int foo(int x) {
        y = very-complex-computation(x);
        write(y);
}
```

- To test all possible paths is impractical or impossible

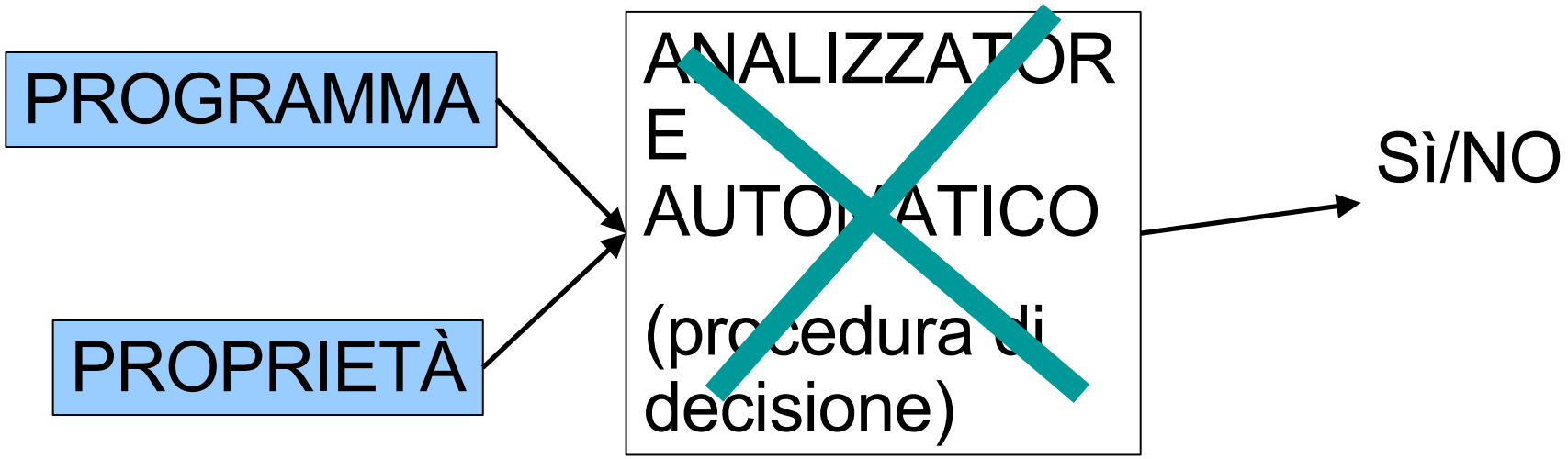
```
int foo(int x) {
        for (index = 1; index < 10000; index++)
                write(x);
}
```

Limitations of Testing (II)

- Dijkstra, 1972
  - Testing can be used to show the presence of bugs, but never their absence
- Goodenough and Gerhart, 1975
  - Testing is successful if the program fails
- The (modest) goal of testing
  - Testing cannot guarantee the correctness of software but can be effectively used to find errors (of certain types)

Un problema di fondo

- Sapere se esiste un input per cui il programma eseguirà una certa istruzione è indecidibile: non esiste un algoritmo che lo risolva per tutti i programmi
- La correttezza di un programma è un problema indecidibile

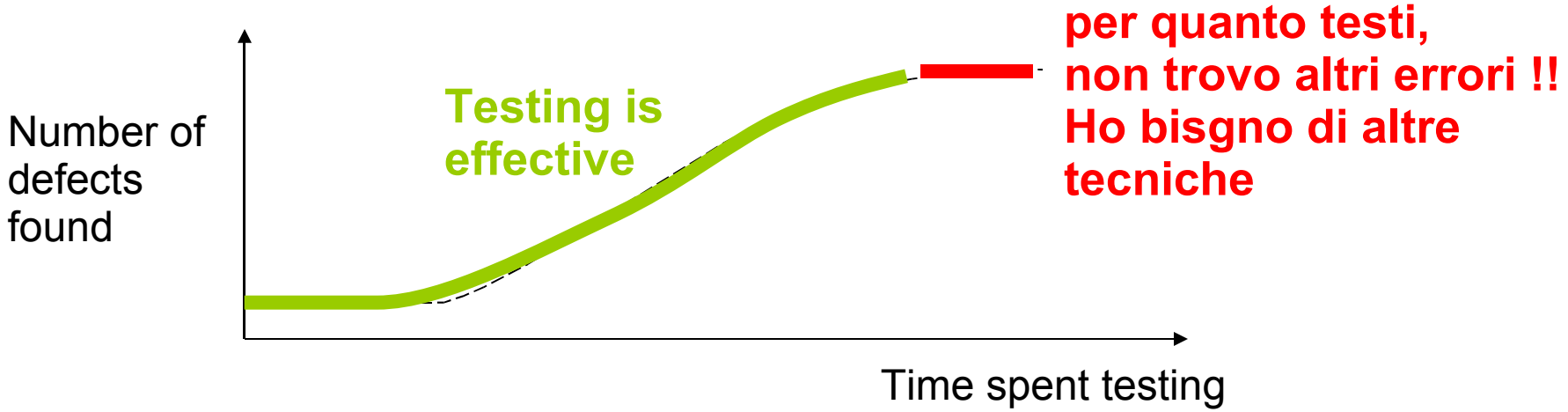


Alcuni problemi del testing

- **I casi di test non sono mai abbastanza**
  - **quando fermarmi?**
- **Testing non è banale e richiede uno sforzo considerevole e tempo**
- **Testing è ancora molto informale**
  - **vedremo però tecniche e tool a supporto del porcesso di testing**

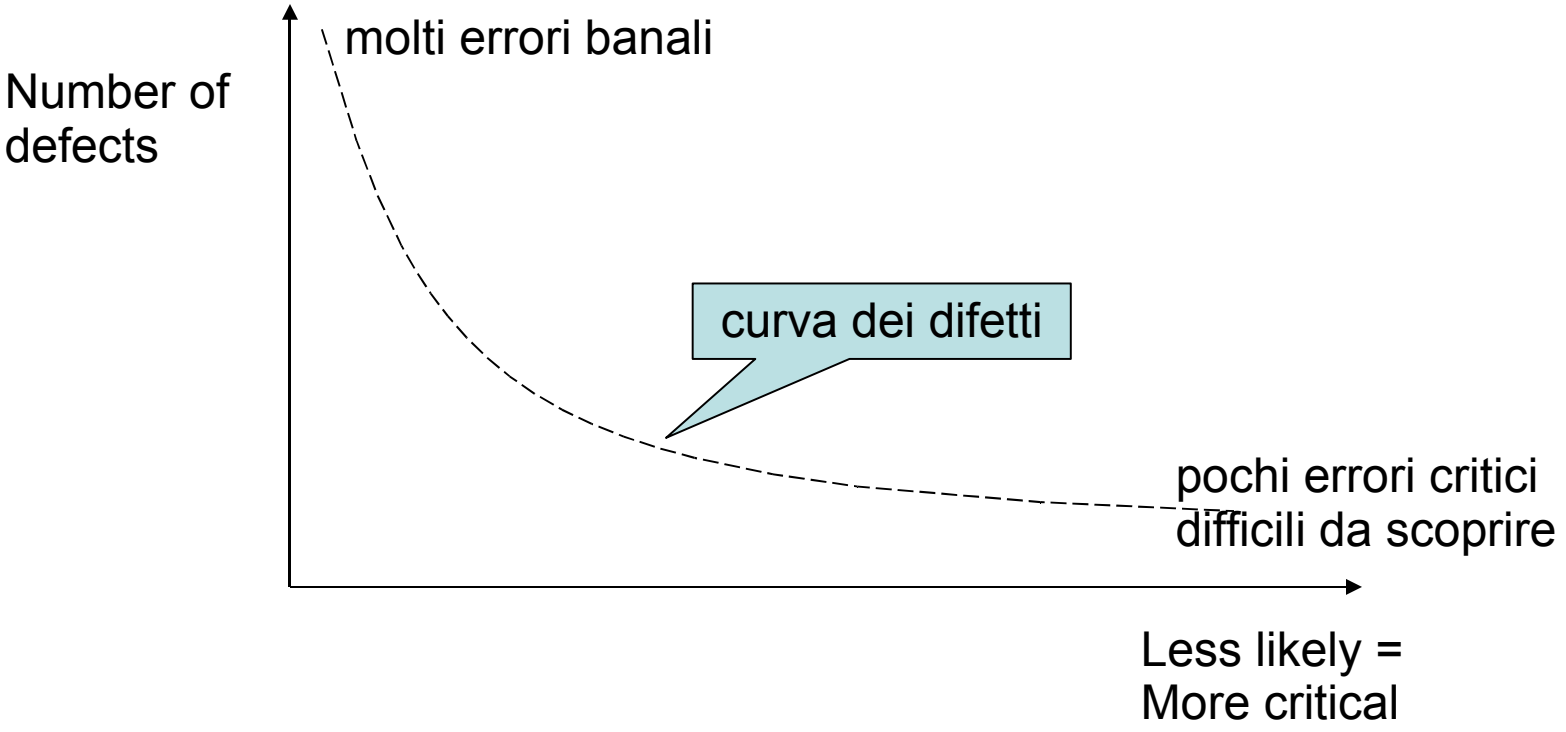
Economia del Testing (I)

- curva di rimozione degli errori



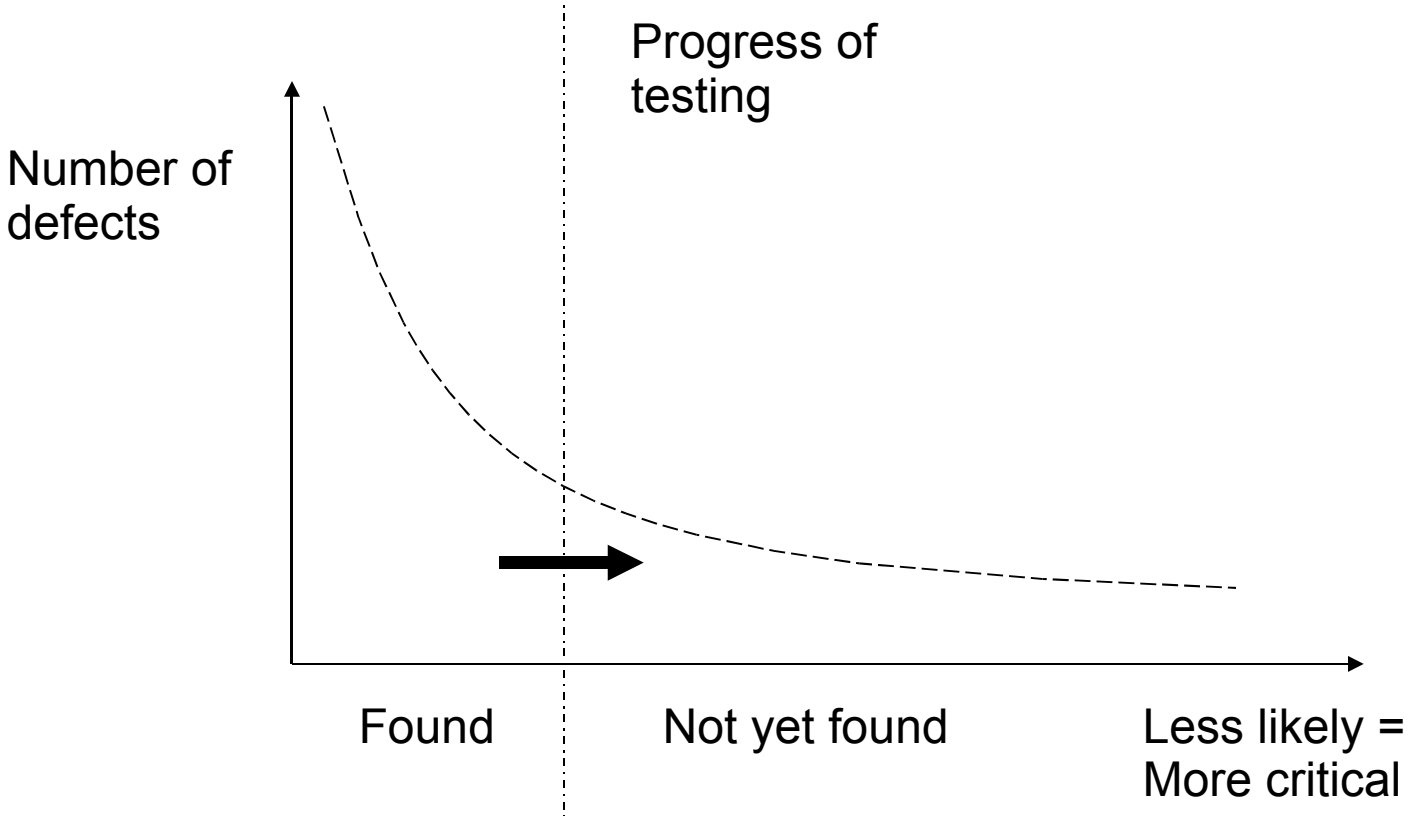
Economics of Testing (II)

- ipotesi: i difetti più critici sono di meno e più difficili da scoprire



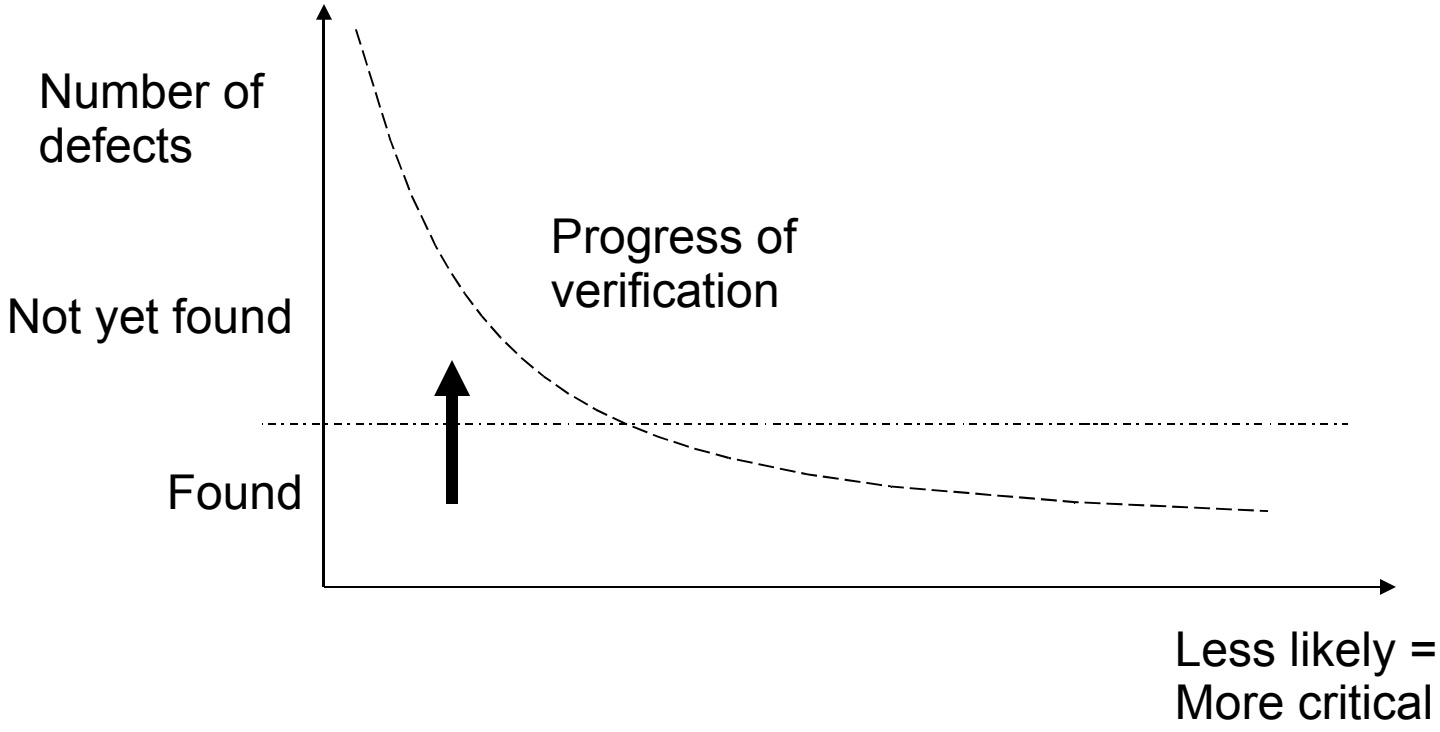
Economics of Testing (II)

- Testing tends to intercept errors in order of their probability of occurrence



Economics of Testing (III)

- Verification is insensitive to the probability of occurrence of errors



Economia del testing

- Attenti però:
- non sempre si ha questa distribuzione
- Quanto costa avere un sw con un difetto?
  - Costo del danno che provoca x
    probabilità che accada
  - quindi è “inutile” spendere troppo tempo a cercare difetti che non provocano danno

Fundamental Questions in Testing

- When can we stop testing?
  - Test coverage
- What should we test?
  - Test generation
- Is the observed output correct?
  - Test oracle
- How well did we do?
  - Test efficiency
- Who should test your program?
  - Independent V&V

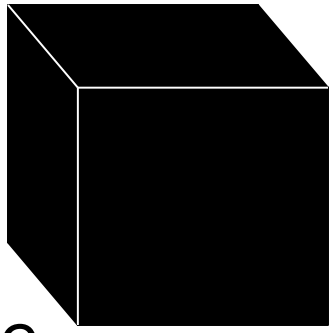
Test criteria (informale)

- test criteria svolgono diversi ruoli:
- Adequacy criteria:
  - Un insieme di casi di test è adeguato a testare un dato programma? Devo fermarmi?
- Coverage criteria:
  - In genere sono definiti in base alla copertura che richiedono: “Ho coperto glia spetti che mi interessano?”
- Selection criteria:
  - Come selezionare i casi di test?

Accessibility of Testing

- White box testing (structural testing, program-based testing)
  - Assumes that the program is available
    - Derives test cases from the program
    - Controls and observes the internal structure of the program
- Black box testing (functional testing, specification-based testing)
  - Assumes that the program is unavailable or testers do not want to look at the details of the program
    - Derives test cases from the requirements of the program
    - Controls and observes the program only through external interfaces

Black-Box (data-driven, inp output) testing

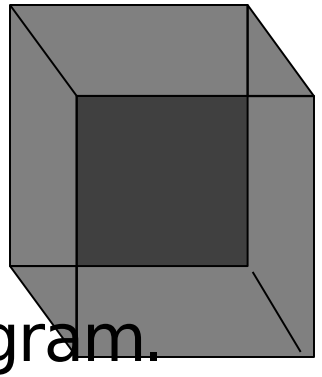


The testing is not based on the structure of the program (which is unknown).

In order to **ensure** correctness, every possible input needs to be tested - this is impossible!

The goal: to maximize the number of errors found.

White Box testing



Is based on the internal structure of the program.

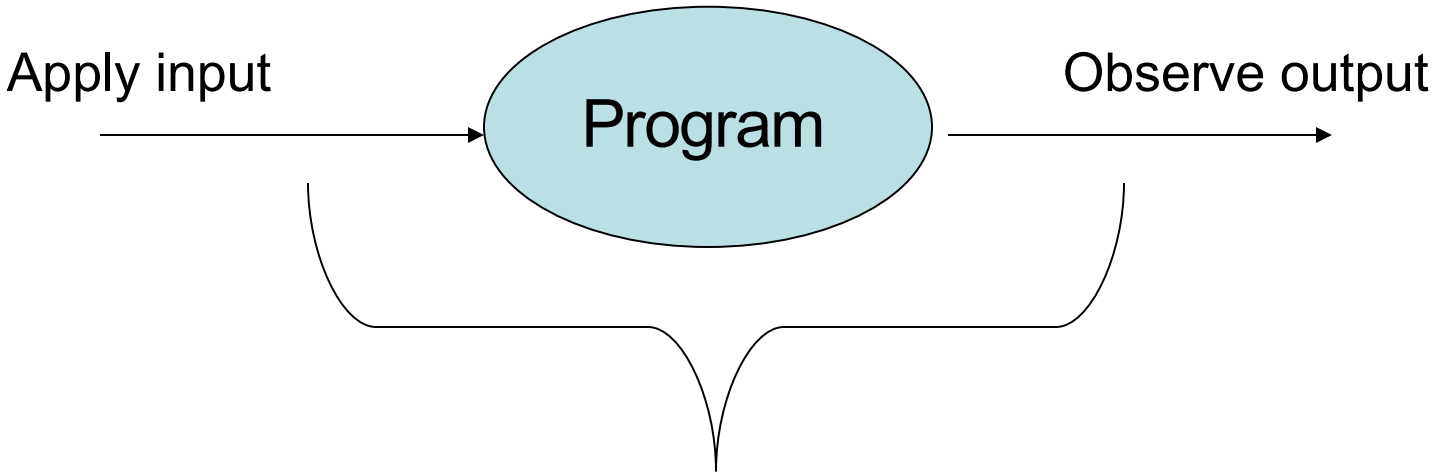
There are several alternative criterions for checking “enough” paths in the program.

Even checking all paths (highly impractical) does not guarantee finding all errors (e.g., missing paths!)

Program-Based Testing (I)

- Main steps
  - Examine the internal structure of a program
  - Design a set of inputs satisfying a coverage criterion
  - Apply the inputs to the program and collect the actual outputs
  - Compare the actual outputs with the expected outputs
- Limitazioni
  - Non riesce a trovare errori di omissione
    - What requirements are missing in the program?
  - Cannot provide test oracles

Program-Based Testing (II)



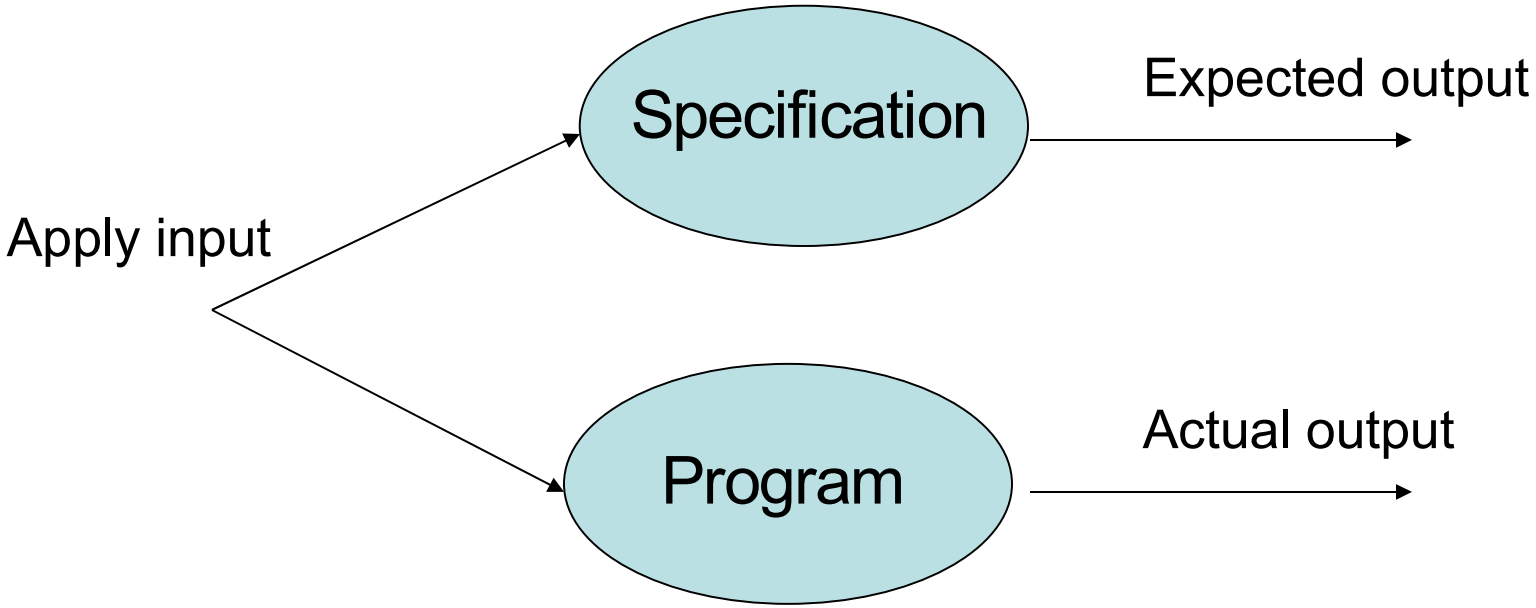
Validate the observed output against the expected output

Who will take care of test oracles?

Specification-Based Testing (I)

- Main steps
  - Examine the structure of the program's specification
  - Design a set of inputs from the specification satisfying a coverage criterion
  - Apply the inputs to the specification and collect the expected outputs
  - Apply the inputs to the program and collect the actual outputs
  - Compare the actual outputs with the expected outputs
- Limitations
  - Specifications are not usually available
    - Many companies still have only code, there is no other document.

Specification-Based Testing (II)



Validate the observed output against the expected output

Object-Oriented Program Testing

- Unit testing for OO Programs
  - A class is a set of variables and member functions
  - 50% of member functions are just 10 lines of code
  - A class is often a unit of testing in C++ or Java
- Integration testing for OO Programs
  - Rule of thumb in OO development
    - Make a large number of small classes in a bottom-up fashion
  - There are several relationships between classes

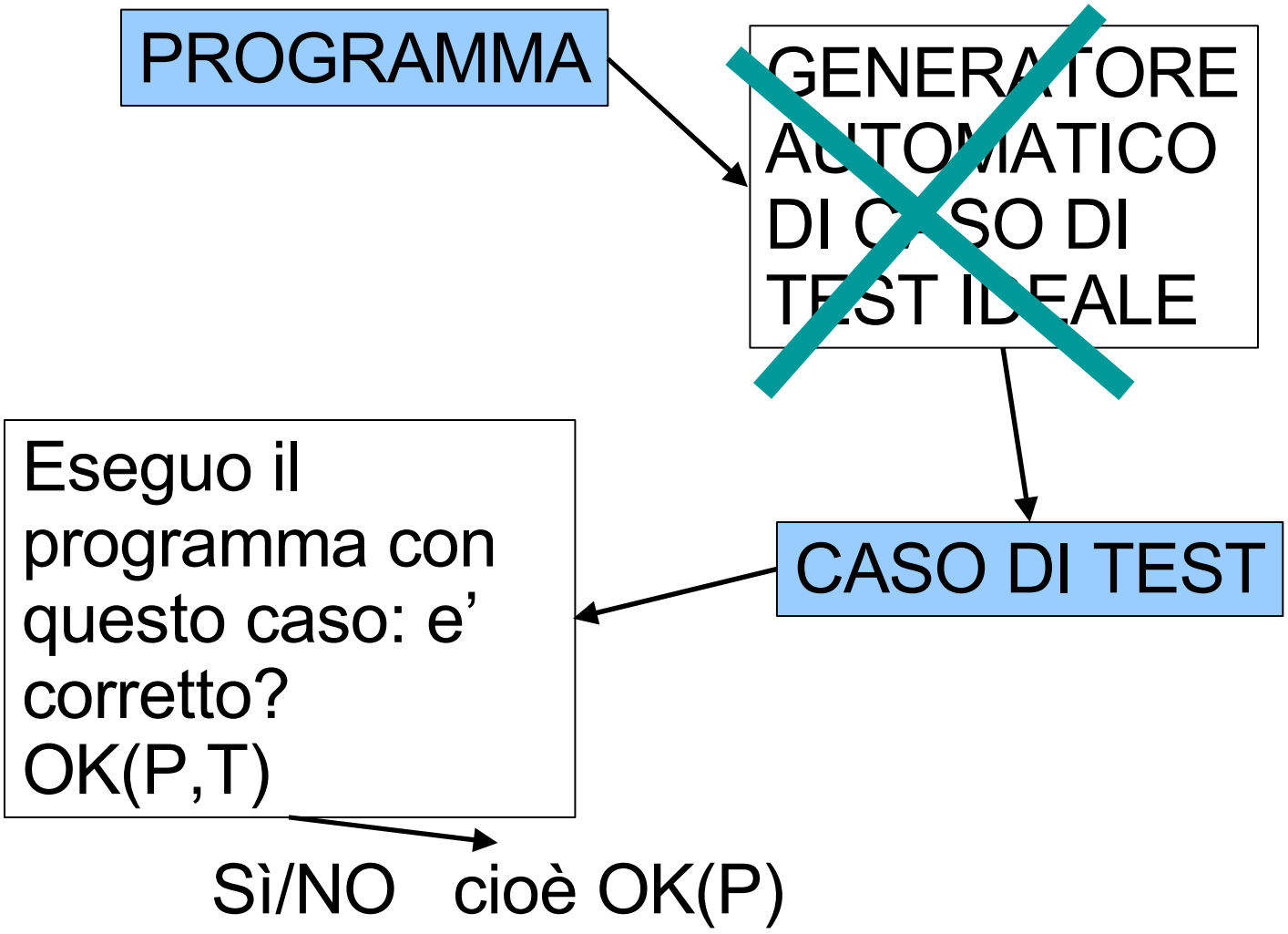
1.Teoria del testing: Bibliografia

- Ghezzi et alt, Ingegneria del SW, molti libri di SE
  - Teoria del testing sui libri (consigliato)
  - Articoli di Goodenough e Gerhart, Howden, Weyuker
- Test adequacy criteria (su web)

ZHU, HALL e MAY,

Software Unit Test Coverage and Adequacy
ACM Computing Surveys, Vol. 29, No. 4, December 1997

Correttezza mediante testing?



Conclusione Parte 1

- Definizione di “empirical” test criteria
  - criteri che non sono ne’ validi ne’ affidabili ma si sono dimostrati utili
- Test criteria come “stopping rule”:
  - ho testato abbastanza (posso essere confidente che P sia corretto?)
- Test adequacy e’ una misura di copertura:
  P x S x T -> [0, 1]

2. Da criteri ideali a criteri pratici – categorie e usi di test criteria

- Adequacy criteria:
  - Quando devo fermarmi?
- Coverage criteria:
  - “Ho coperto gli aspetti che mi interessano?”
- Selection criteria:
  - Come selezionare i casi di test?
- In generale ogni definizione copre i tre aspetti.

Categorie di test criteria

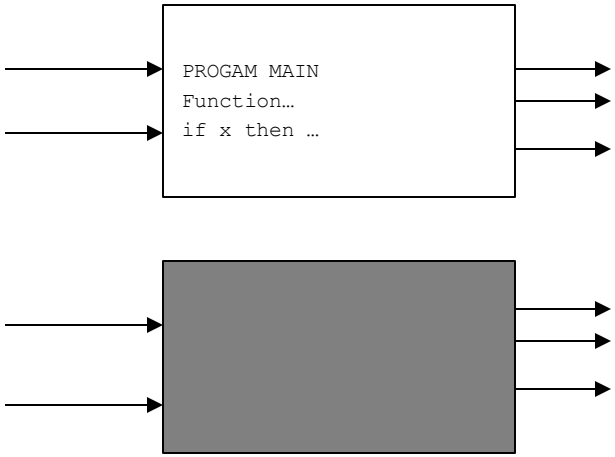
Adeguatezza, copertura e selezione definite in base a:

- Specification-based: specifiche o requisiti.
- Program-based: programma sotto test
- Error-oriented: gli errori che voglio scoprire
- Combined specification-program based criteria

- Solo l'interfaccia: Random testing o statistico

Black box /white box

- White-box: si conosce l'interno dell'implementazione
- Black-box: si utilizza il programma come black box
  - Program-based: white box
  - Spec based e random : black box



Black or White? (1)

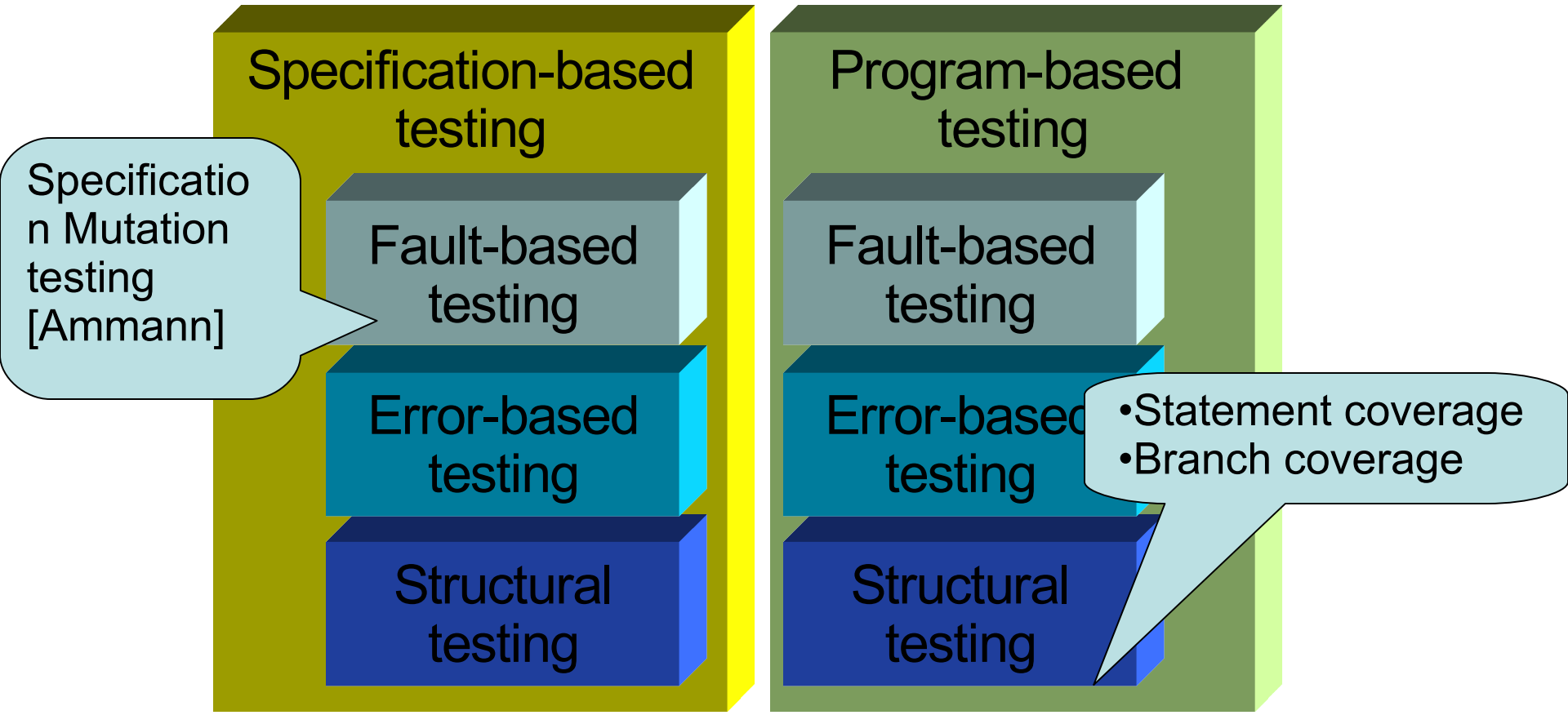
- Black box
  - Dipende dalla notazione di specifica
  - È scalabile (può essere utilizzato a tutti i livelli di granularità)
  - Non può rilevare difetti che dipendono dal codice (la stessa
- White box
  - È basato su coperture dei flussi di controllo o dei dati
  - Non è scalabile (è utilizzato soprattutto a livello di unità o sottosistema)
  - Non può rilevare difetti che dipendono dalla mancata implementazione di

Ulteriore classificazione

- **structural**: in termini dei elementi della struttura del programma o della specifica
- **fault based**: si focalizza sulla ricerca di particolari difetti
  - Ad esempio tipici errori del programmatore
- **failure based**: si focalizza su certi malfunzionamenti (tipici o quelli più critici)

Categorie dei criteri di test

[Zhu, Hall & May]



Importanza di una specifica

- il testing e’ una forma di verifica: non può essere fatto senza dei requisiti di riferimento!!
- Anche nel caso di program-based testing i requisiti sono indispensabili: cosa vuol dire che un programma ha passato i test?

Importanza degli oracoli

- La correttezza del comportamento di un programma non dovrebbe essere lasciata al giudizio umano (non è affidabile ed è costoso)
- **Oracle**: una procedura meccanizza che decide se un certo output dato un certo input è accettabile oppure no
- Esempi immediati
  - Oracoli nel codice: `if (x ==0) printl("ERRORE");`
  - `assert(x>0)` (in Java 1.4)

Software testing
white box testing

Angelo Gargantini

Bibliografia

- Ghezzi et alt, Ingegneria del SW, qualsiasi libro di SE
- Capitolo 14 di “**Software Test and Analysis: Process, Principles, and Techniques”** Pezze e Young: http://www.cs.uoregon.edu/~michal/book/
- http://www.testing.com/writings/coverage.pdf

Testing strutturale basato sui programmi

- Basati sul flusso di controllo:
  - Statement coverage – copertura delle istruzioni
  - Edge (branch) coverage – copertura dei spigoli
  - Condition coverage
  - Path coverage
- Data flow (syntactic dependency) coverage

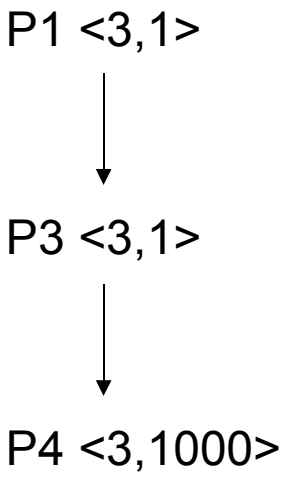
Semantica dei Programmi (1)

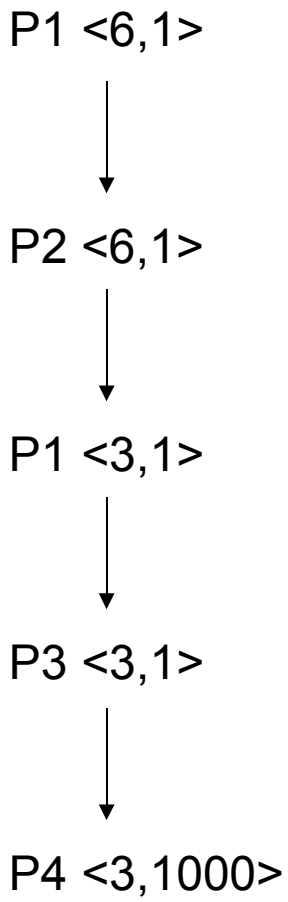
- We will mainly focus on a program module, e.g., a function in C
- The semantics (behavior) of a program
  - What is the meaning of a program?
    - The set of computations executed by the program

Program Semantics (2)

- Example: Program1

```
     int x, y;
P1:  while (even x)
P2:          x = x div 2;
P3:  y = 1000;
P4:  ……
```





Program Semantics (3)

- We cannot analyze many interesting properties of a program (so we need abstractions)
  - P3 is reachable from P1?
    - Is there a value of x that leads to the infinite execution of the loop?
    - The (undecidable) halting problem

```
          int x, y;
P1:       while (even x)
P2:             x = x div 2;
P3:       y = 1000;
P4:       ……
```

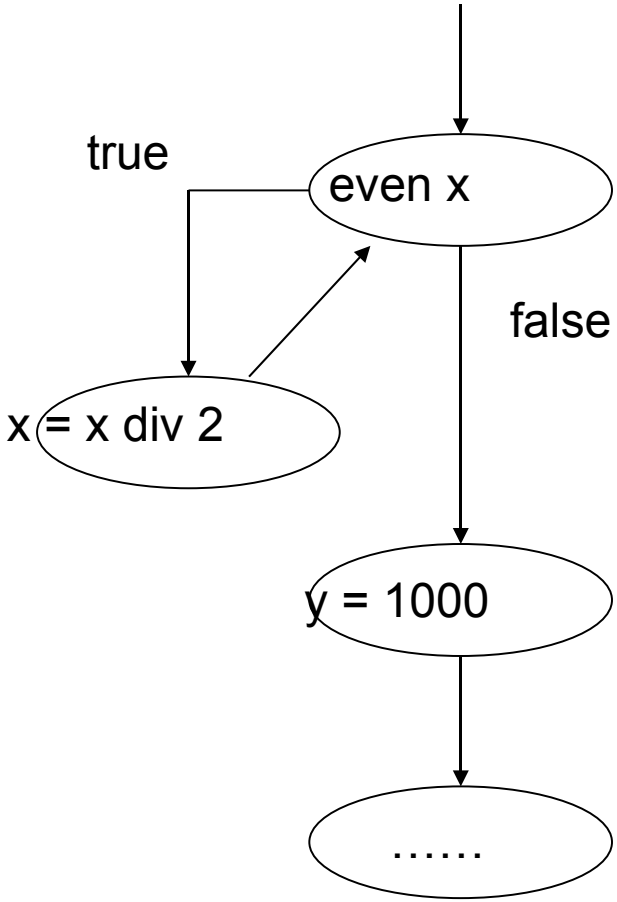
```
          int x, y;
P1:       while (very-complex-expression)
P2:             x = x div 2;
P3:       y = 1000;
P4:       ……
```

Control Flow Graph (1)

- The control flow graph of a program
  - Reflects the flow of control
  - Ignore the values of variables

```
x and y are integers

P1:         while (even x)
P2:                 x = x div 2;
P3:         y = 1000;
P4:         ……
```

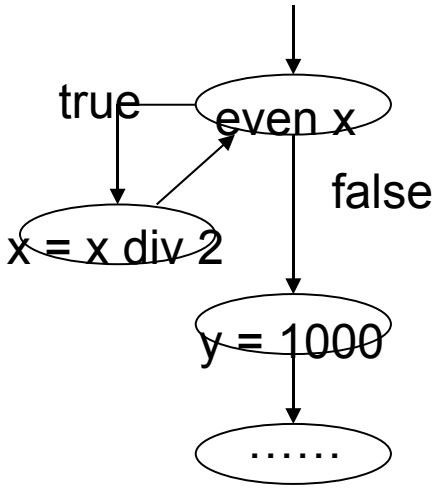


Control Flow Graph (2)

- What is the relationship between a program and its control flow graph?
  - The CFG is a sound abstraction of a program
  - For every computation of a program, there is a corresponding computation of the flow graph
    - What about the other direction?

Control Flow Graph (3)

```
        int x, y;
P1:     while (even x)
P2:         x = x div 2;
P3:     y = 1000;
P4:     ……
```



P1 <3,1>
P3 <3,1>
P4 <3,1000>

P1 <6,1>
P2 <6,1>
P1 <3,1>
P3 <3,1>
P4 <3,1000>

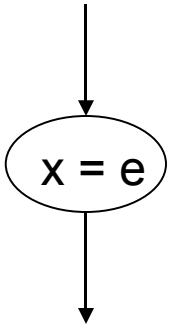
P1
P3
P4

P1
P2
P1
P3
P4

Control Flow Graph (4)

- Assignment, read, write, and return statements

Assignment-statement
x = e;

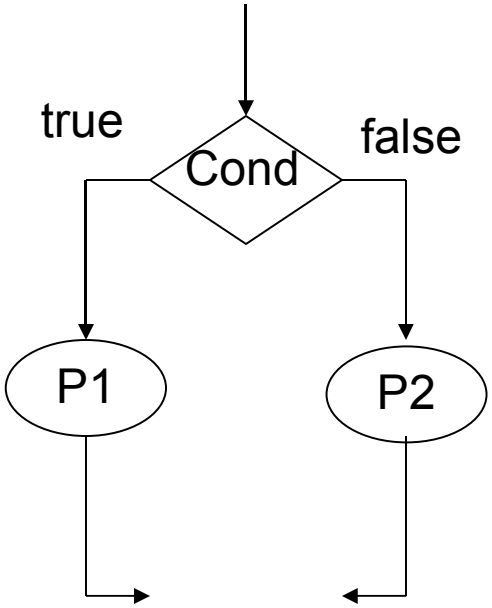


Also, Read-statement,
Write-statement, and
Return-statement

Control Flow Graph (5)

- If statements

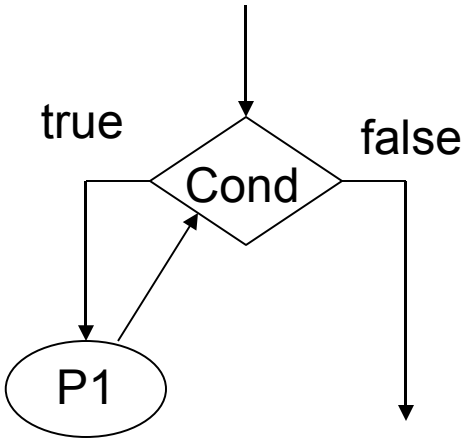
if (cond) then P1 else P2



Control Flow Graph (6)

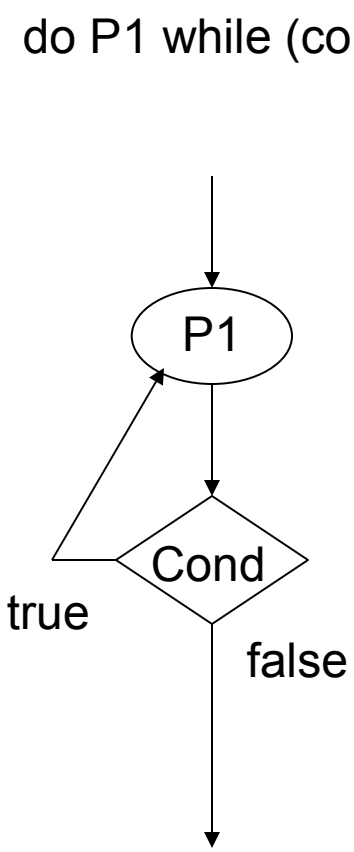
- While statements

while (cond) P1



Control Flow Graph (7)

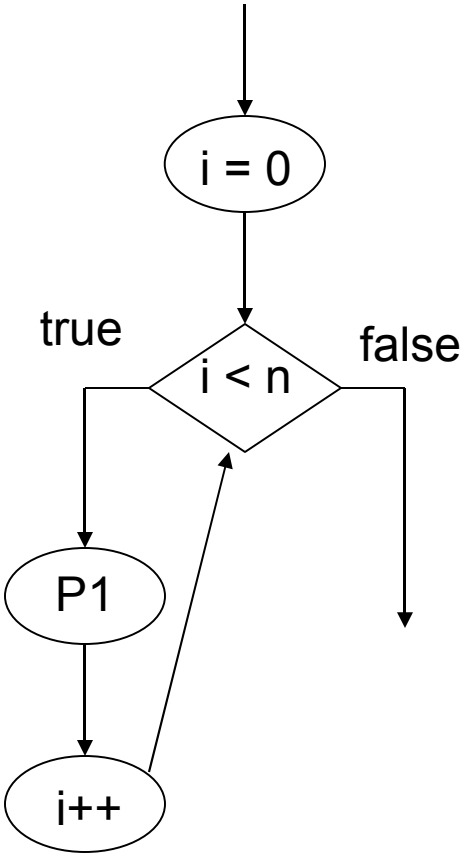
- Do-while statements



Control Flow Graph (8)

- For statements

for (i = 0; i++; i < n) P1

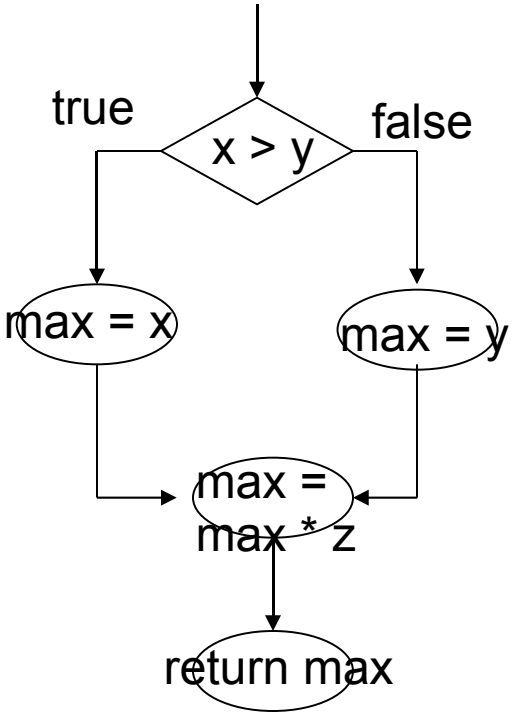


Control Flow Graph (9)

- Example

```
int main(int x, y, z) {
        int max;

        if (x > y)
                max = x;
        else
                max = y;
        max = max * z;
        return max;
}
```

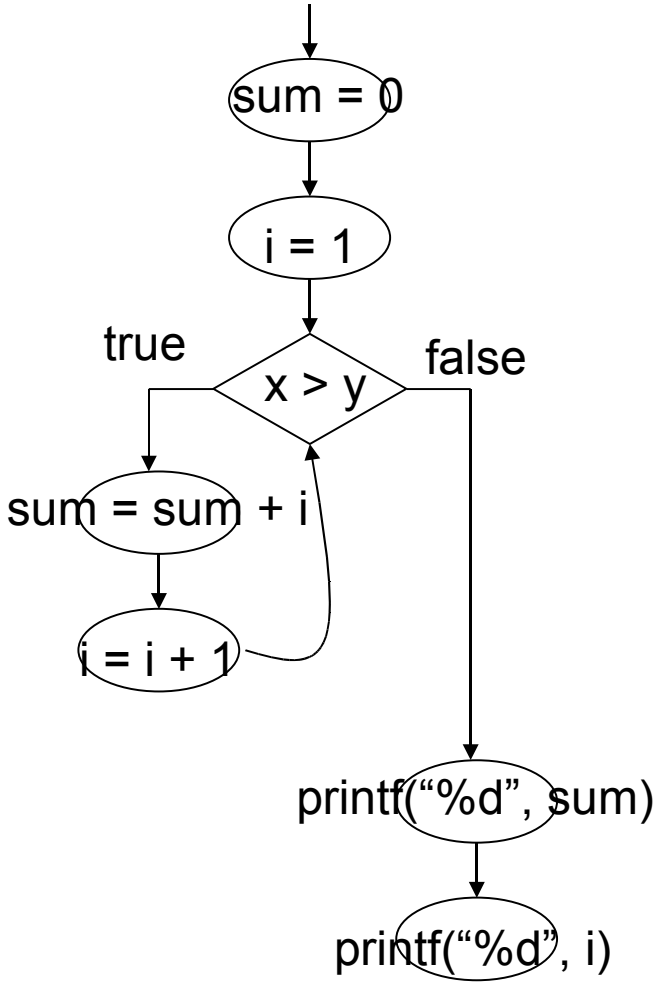


Control Flow Graph (10)

- Example

```
int main() {
        int sum, I;

        sum = 0;
        i = 1;
        while (i < 11) {
                sum = sum + i;
                 i = i + 1;
        }
        printf(“%d”,sum);
        printf(“%d”,i);
}
```



Esercizio

- Costruisci il control flow graph di:

```
int main(int x, y) {
        int i;

        for (i = 0; i < 10; i++) {
                if (x > y)
                        printf(“Hi”);
                else if (x == y)
                        printf(“Hello”);
                else
                        printf(“Wazup”);
        }
        printf(“ByeBye”);
}
```

Statement coverage (comandi)

- Un test set T e' adeguato per testare un programma P se per ogni istruzione *s* di P esiste un caso di test in T che esegue *s*
  - Ogni istruzione viene eseguita almeno una volta
  - Esempio:
    - if x /= 0 then x:=x+10 else x:=x-10

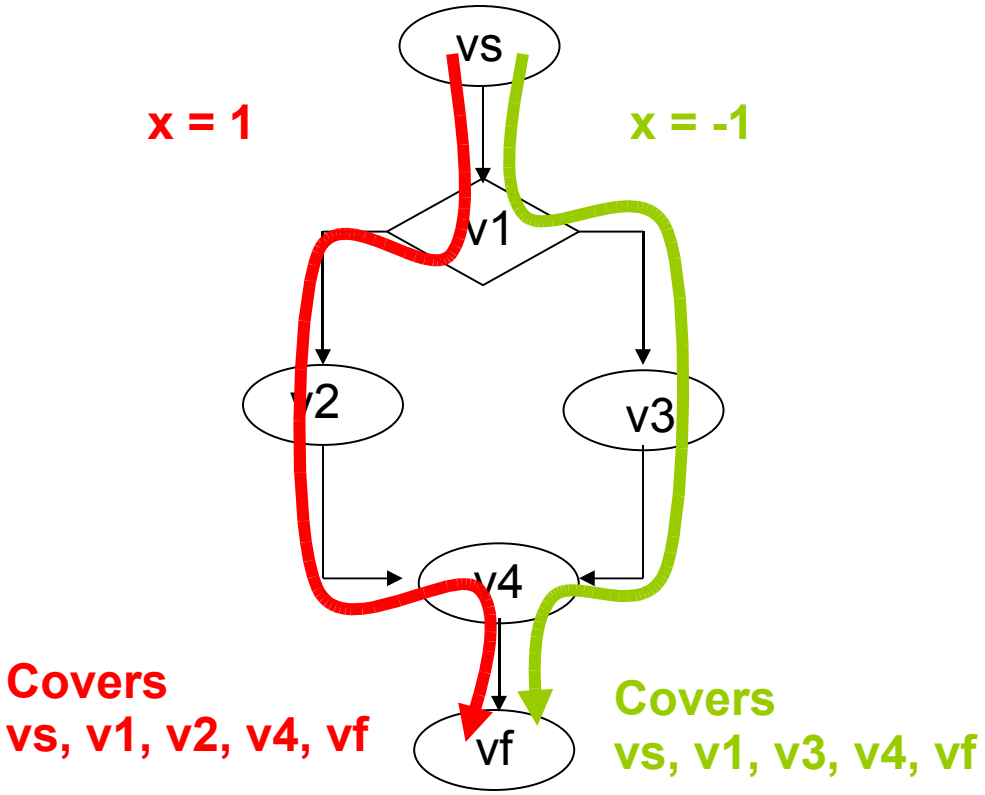
Devo prendere un caso con X=0 e uno con X/=0

Statement Coverage

- Statements
  - vs, v1, v2, v3, v4, vf

```
vs:   int main(int x) {
                int result;

v1:             if (x > 0)
v2:                   result = x;
                else
v3:                   result = 1 / x;
v4:             printf("%d", result);
vf:   }
```

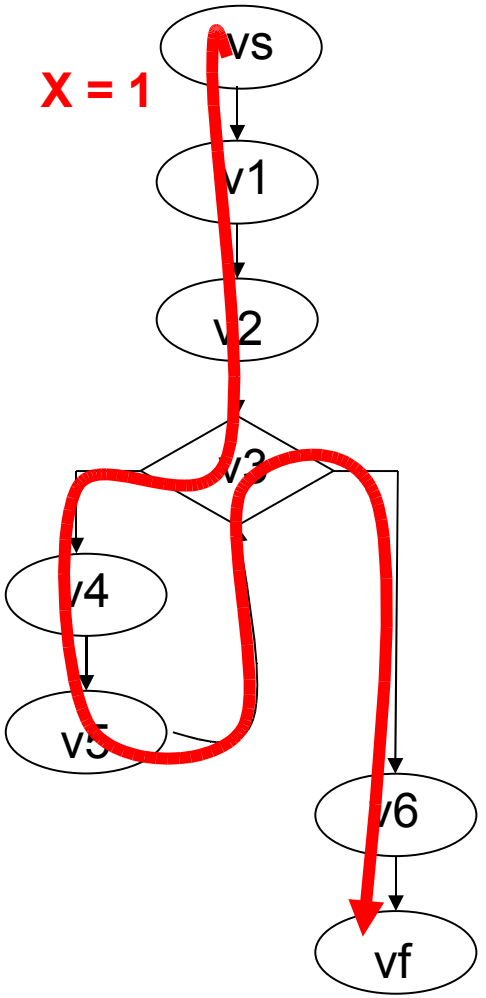


Statement coverage

- Statements
  - vs, v1, v2, v3, v4, v5, v6, vf

```
vs:     int main(int x) {
            int sum, I;

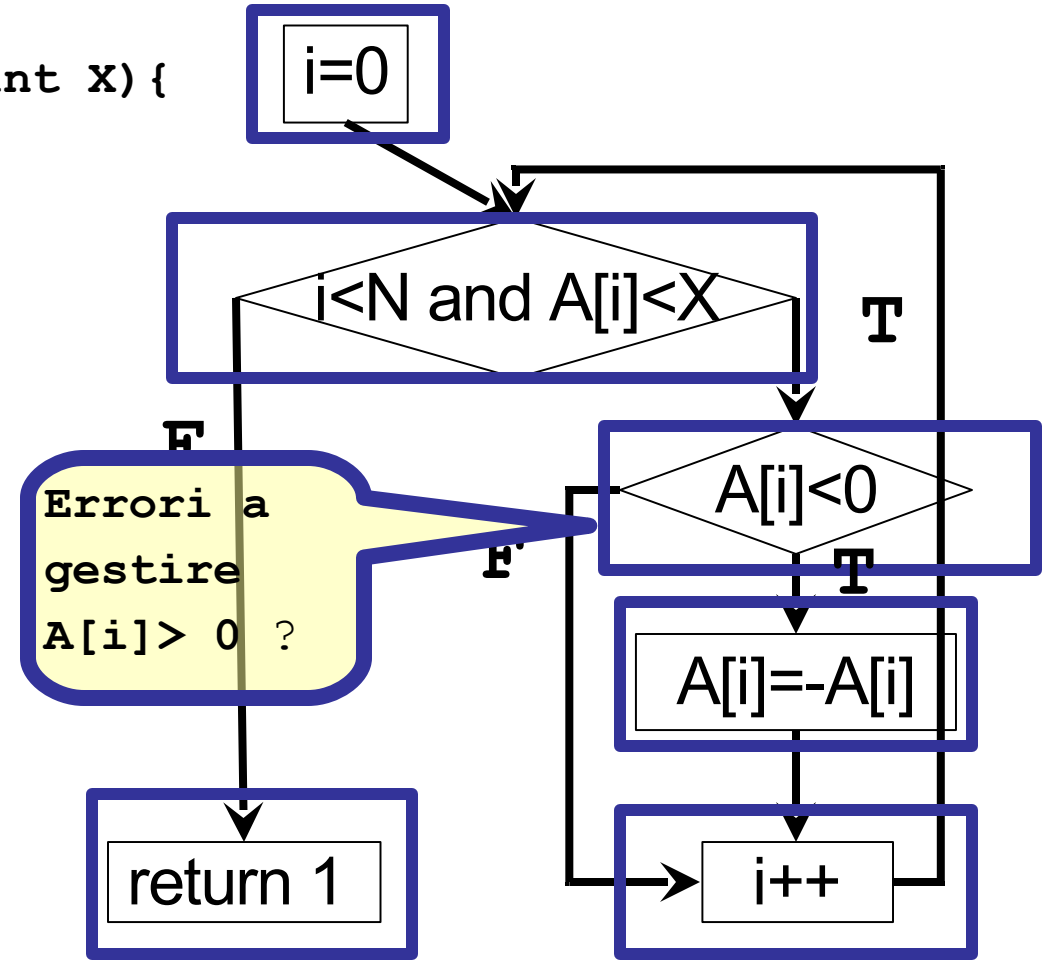
v1:         sum = 0;
v2:         i = 1;
v3:         while (i <= x) {
v4:             sum = sum + i;
v5:             i = i + 1;
            }
v6:         printf("%d",sum);
vf:     }
```



Statement coverage (esempio)

```
int select(int A[], int N, int X){
  int i=0;
  while (i<N and A[i] <X) {
     if (A[i]<0)
          A[i] = - A[i];
     i++;
  }
  return(1);
}
```

- Il caso di test: (N=1, A[0]=-7, X=9) è suff.



Fault detection capability

- Come valutare un criterio di copertura ?
  - Fault detection ability
- Statement coverage:
  - Errori di passaggio di controllo in generale non vengono evidenziati
  - Questo criterio e' debole
  - Però potrebbe anche bastare: idea di MINIMAL CRITERION FOR COMMERCIAL SOFTWARE

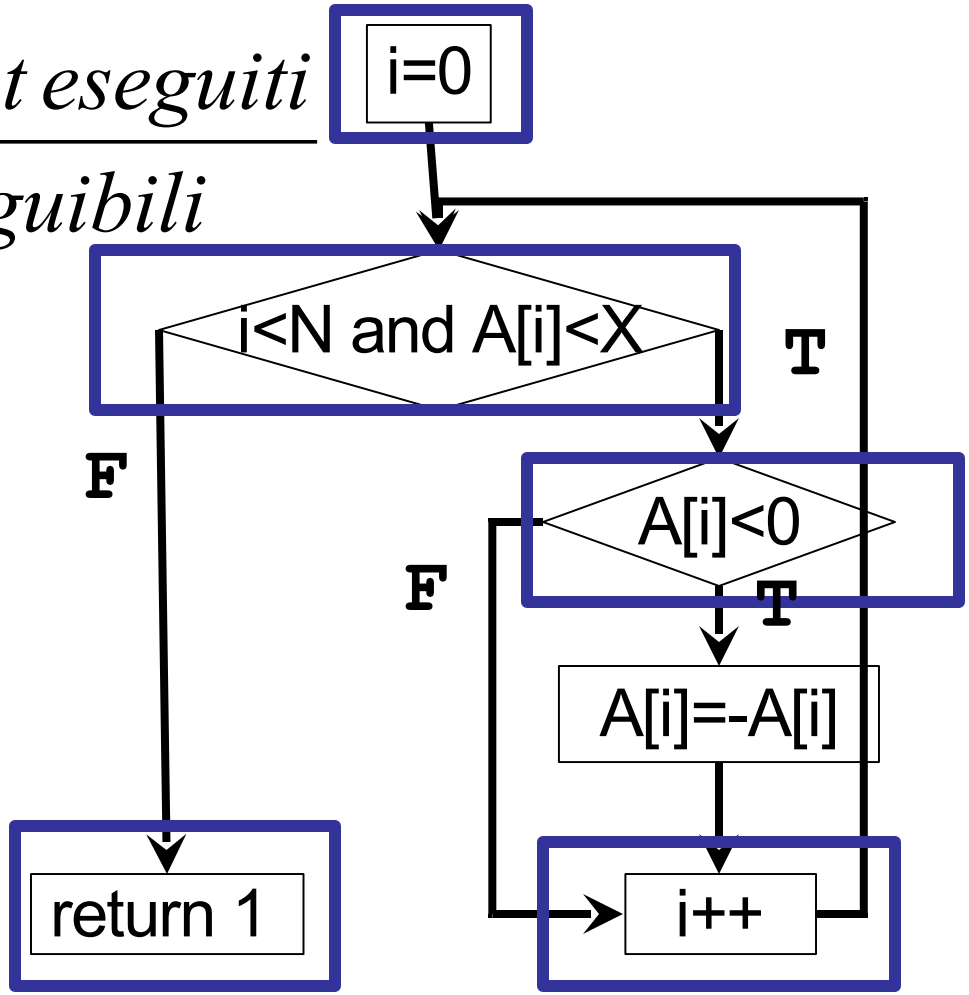
Secondo voi?

Misura di copertura dei comandi

$$C_0 = \frac{numero\ \ di\ statement\ eseguiti}{numero\ di\ st.eseguibili}$$

- Il caso di test:
  (N=1, A[0]=7, X=9)

  copre 5/6



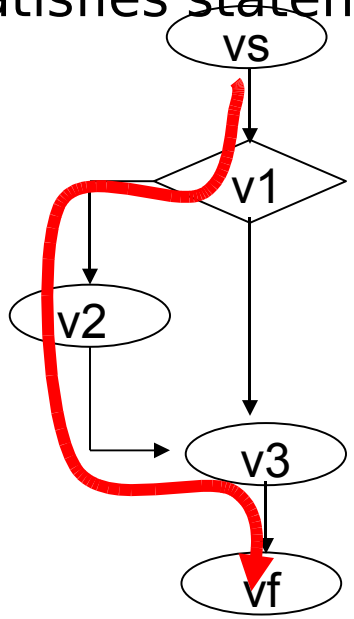
Branch coverage (decisioni)

- Un test T soddisfa il criterio di copertura delle decisioni se e solo se ogni arco (branch) del grafo e' percorso almeno una volta

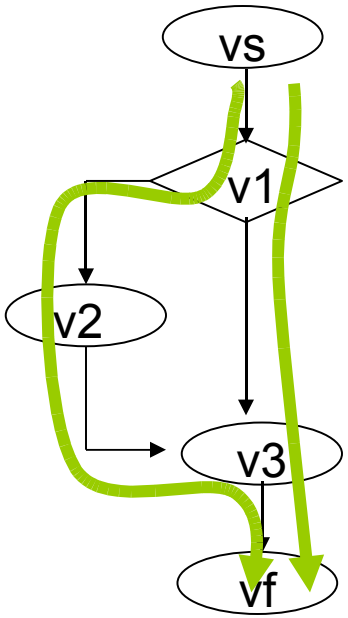
$$C_{path} = \frac{numero\ \ di\ branch\ eseguiti}{numero\ di\ branch\ eseguibili}$$

Diiferenza tra Statements e Branch coverage

- Statement coverage è più debole
  - Branch coverage **subsumes** statement coverage
    - If a test suite satisfies branch coverage, then it also satisfies statement coverage



Statement coverage

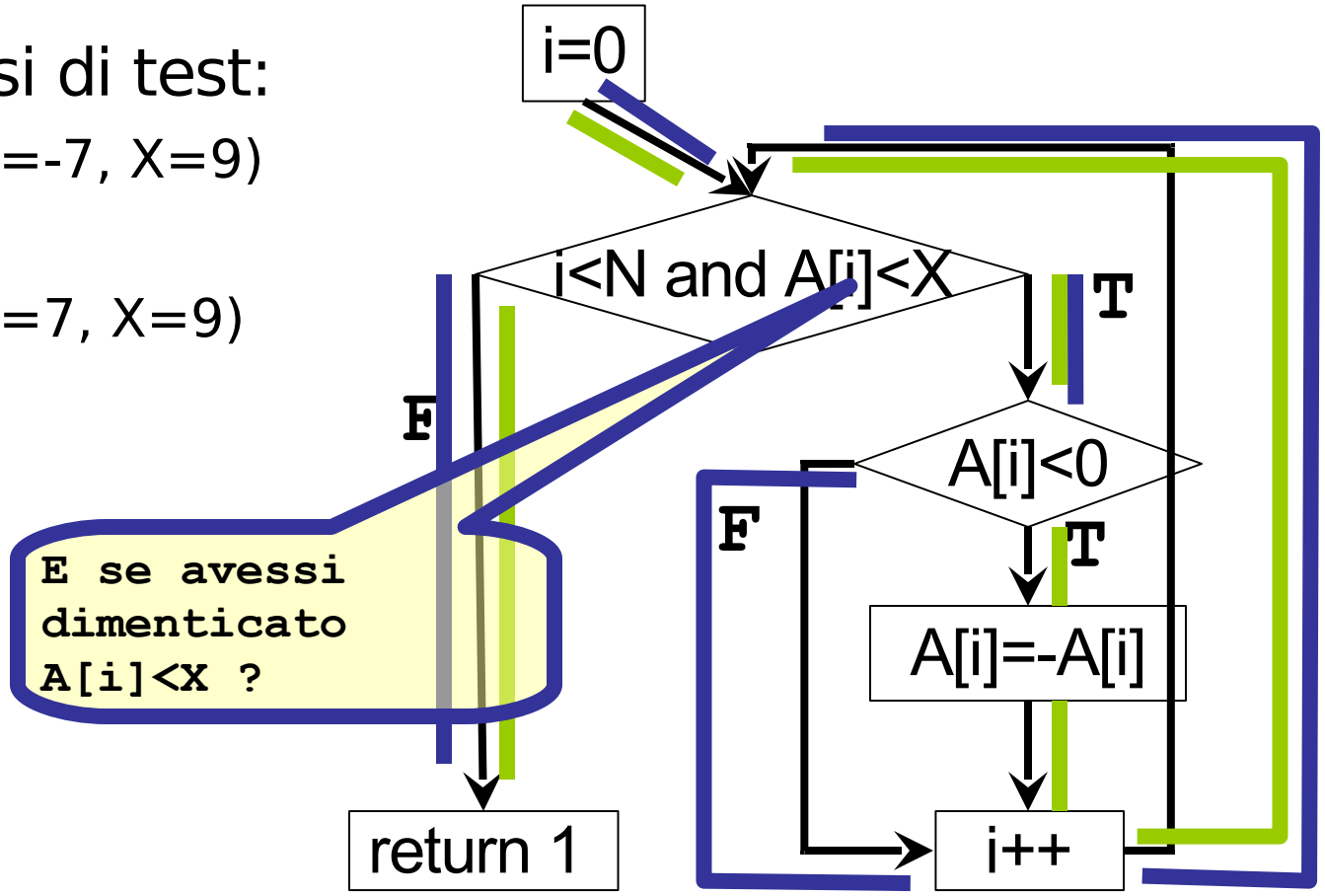


Branch coverage

Branch coverage (esempio)

Abbiamo 2 casi di test:

- (N=1, A[0]=-7, X=9)
- (N=1, A[0]=7, X=9)



Esercizi

```
…
if x != 0 then y := y/2
…
----------------------
…
if x = 0 or y>0 then y := y/x
                else x := y+2/x
…
```

Importanza di valutare le condizioni atomiche (che non posso essere divise in altre condizioni) separatamente

Esempio gcd

```
read (x,y)
a:= x; b:= y;
while a <> b do
   if a > b then a:= a-b;
            else b:= b-a;
   end if
end while
```

Decisions and Conditions (1)

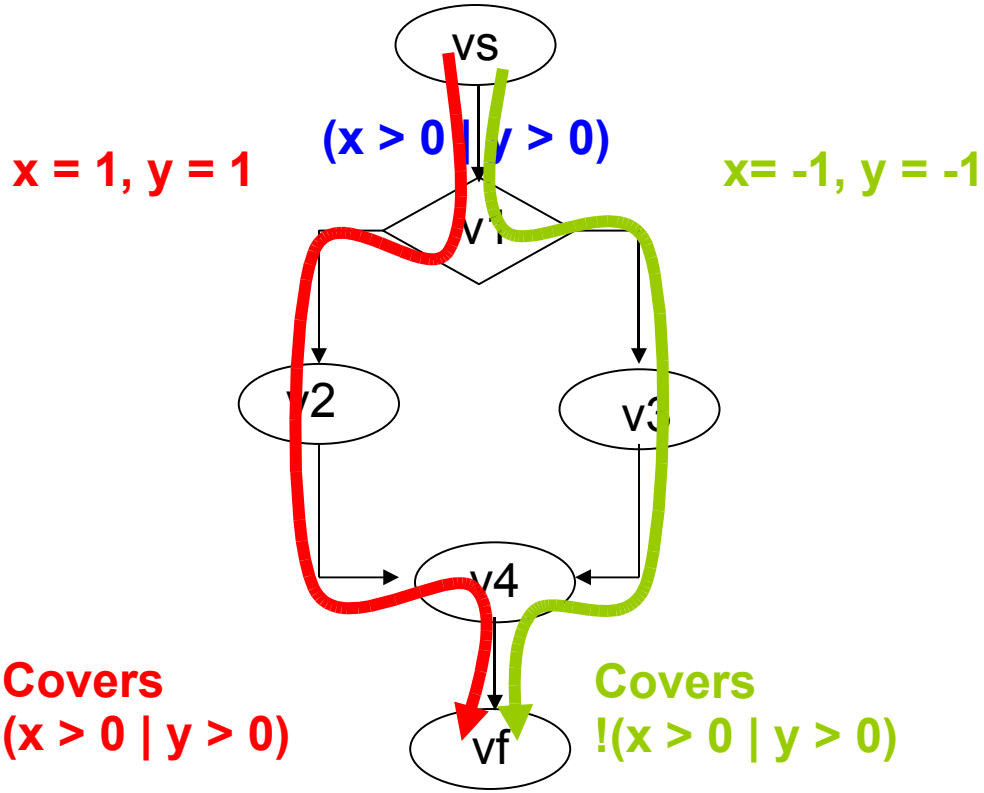
- Decision
  - predicato (espressione) di una istruzione condizionale (if) o di una iterativa (while, for, …)
  - esempio (x > 0 | y > 0)
- Condition
  - An atomic boolean expression appearing in a decision
  - E.g., (x > 0), (y > 0)

```
vs: int main(int x, y) {
          int result;

v1:       if (x > 0 | y > 0)
v2:            result = x;
          else
v3:            result = y;
v4:       printf("%d", result);
vf:     }
```

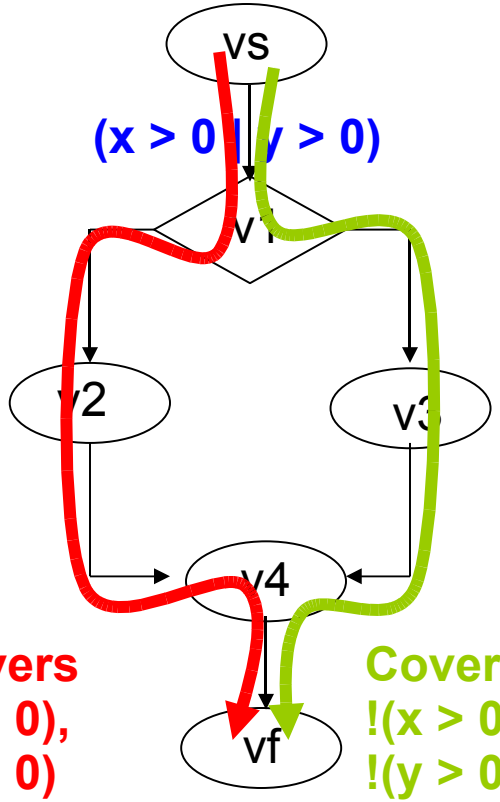
Decisions and Conditions (2)

- Decision coverage
  - Covers every decision and its negation
  - Is equivalent to branches coverage criterion



Decisions and Conditions (3)

- Condition coverage
  - Covers every condition and its negation

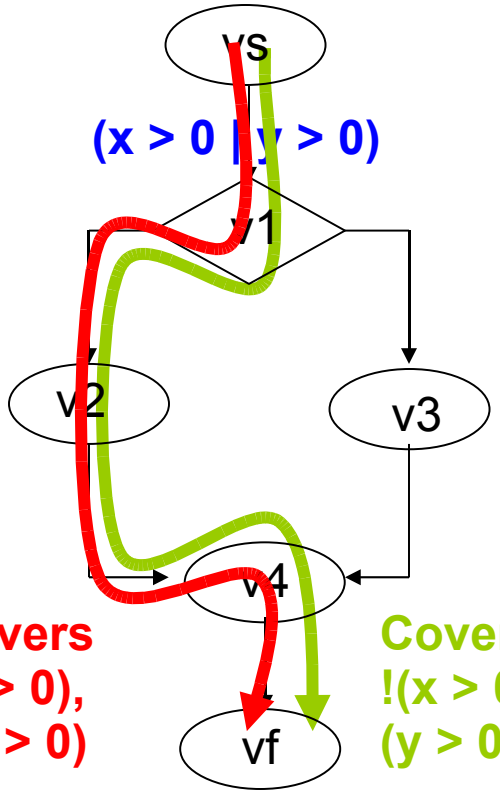


Covers (x > 0), (y > 0)

x = 1, y = 1

Covers !(x > 0), !(y > 0)

x= -1, y = -1



Covers (x > 0), !(y > 0)

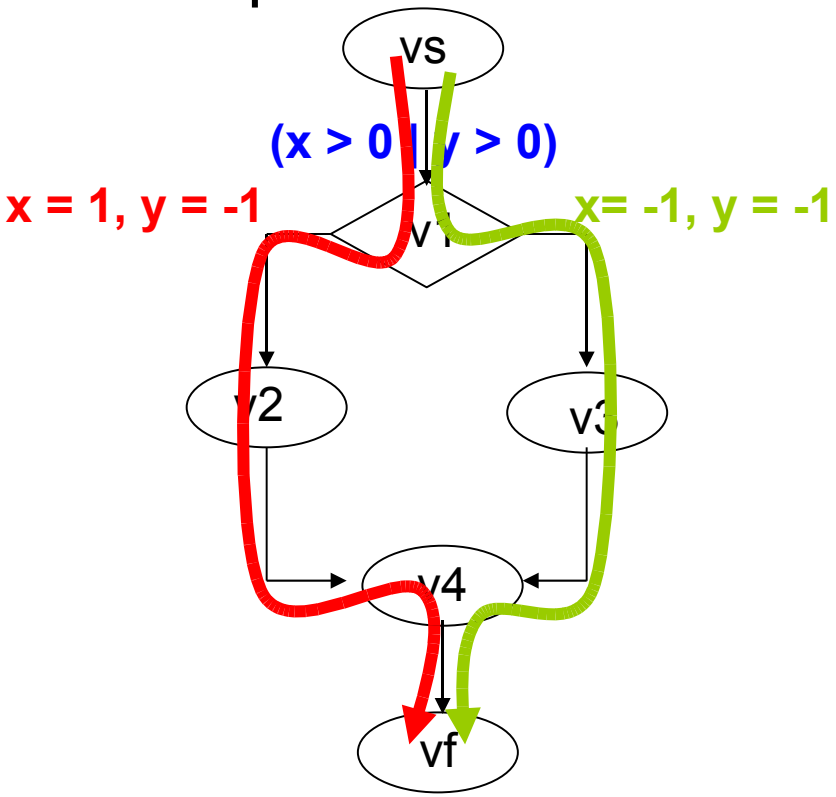
x = 1, y = -1

Covers !(x > 0), (y > 0)

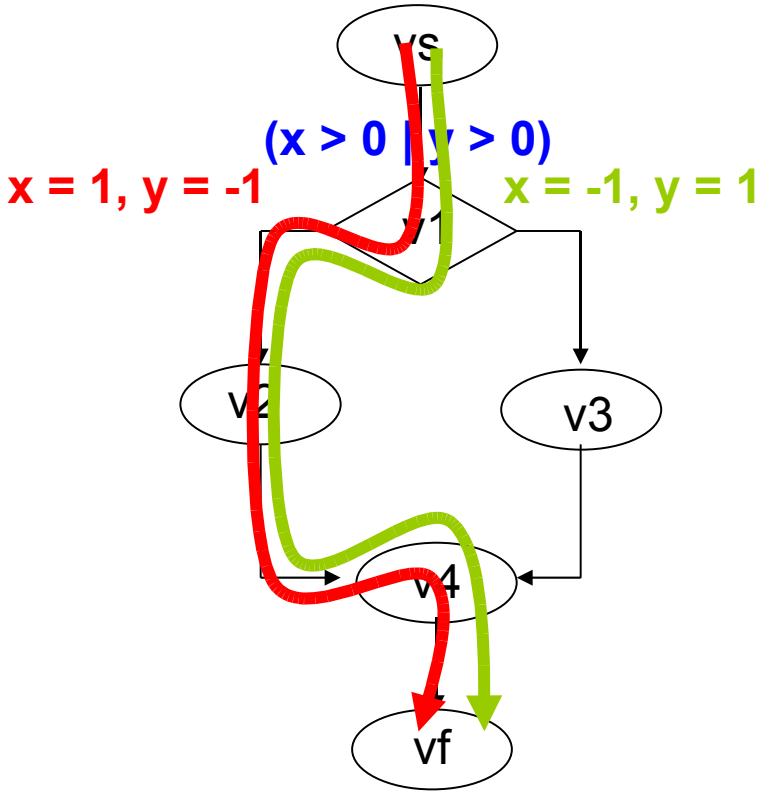
x = -1, y = 1

Decisions and Conditions (4)

- Decision coverage and condition coverage are incomparable



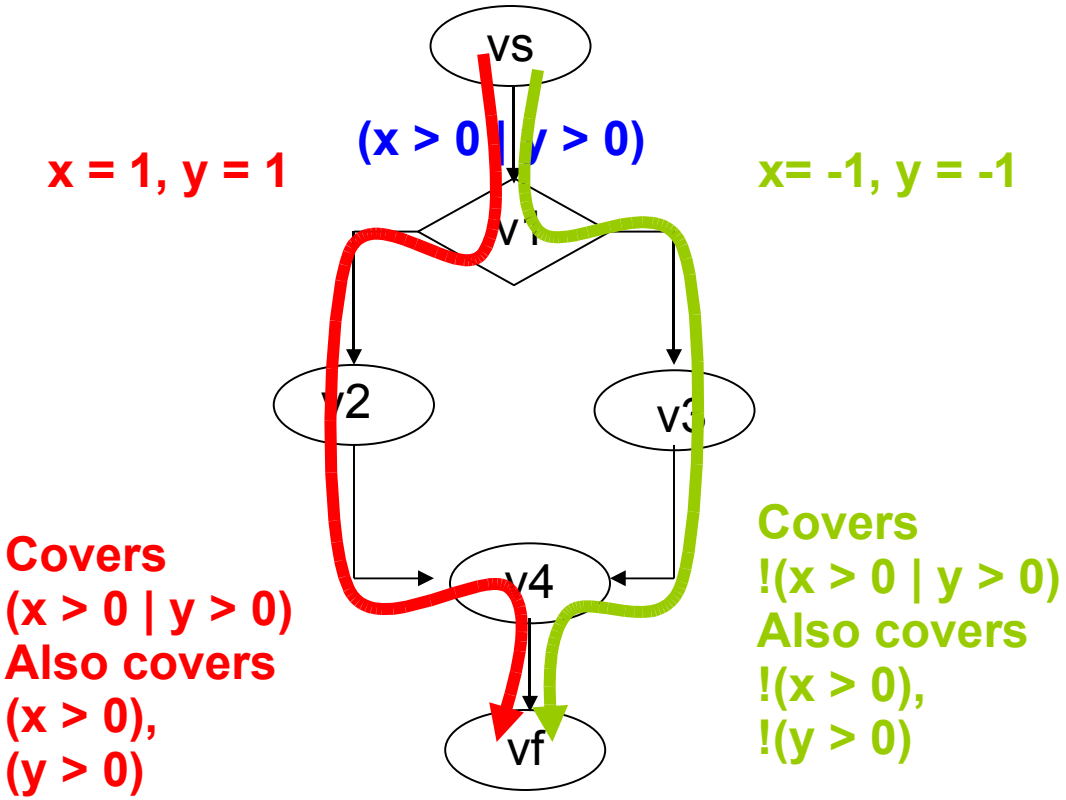
Satisfies decision coverage
But not condition coverage



Satisfies condition coverage
But not decision coverage

Decisions and Conditions (5)

- Condition/decision coverage
  - Combines condition coverage and decision coverage



(short circuit evaluation)

- I compilatori spesso usano per efficienza la valutazione a corto circuito per le espressioni booleane:
- Esempi:
  - a && b: Se a e' falso non valuto b
  - a || b: Se a e' vero non valuto b
- Nota che spesso esistono operatori che evitano ciò (esempio & e | al posto di && e ||)

(Basic) Condition coverage

- per ogni condizione atomica *c* esiste un caso di test T per cui *c* e' valutata true, e un T per cui e' valutata false
- **`Esempio: (i<N and A[i] <X)`**

		i<N and	A[i]<X
N=1, A[0]=-7, X=9	i=0 i=1	T (i=0) F (i=1)	T (T)
N=1, A[0]=7, X=9		T,F	T,(T)
N=1, A[0]=7, X=5		T	F

Condition/decision coverage

- Condition coverage non implica il branch coverage
  - Esempio:

    if (A and B) then …

    Prendo [A true, B false] e [A false, B true]

    Ho condition coverage ma non branch
- C/D cov: soddisfa sia branch coverage (ogni decisione è presa) e condition coverage (ogni condizione e’ testata)

Esercizio C/D

- if (A and B) then
- If x > i and A[x] < i
- ```
  int foo(int a, int b){
  ```
  ```
  while (a<>b and a>1){
  ```
  ```
      a = a - b
  ```
  ```
  }
  ```
- gcd
  ```

Esercizio

- Generate test suites for each of the coverage criteria
  - Statements coverage
  - Branches coverage
  - Decision coverage
  - Condition coverage
  - Condition/decision coverage
  - MC/DC coverage

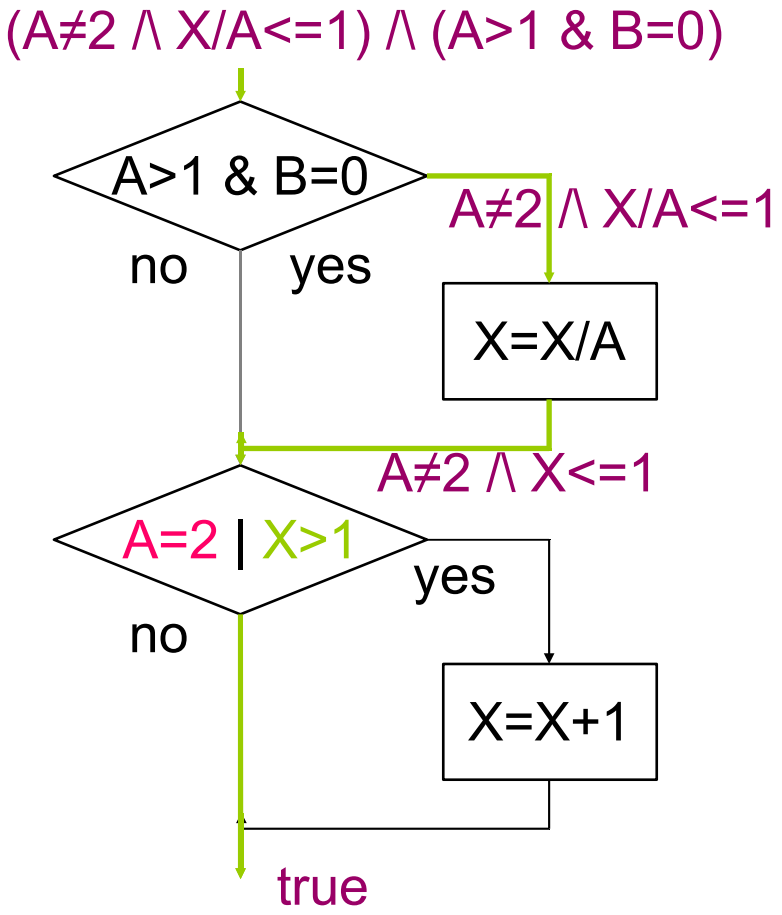
```
vs:   int main(int x, y) {
v1:         while (x > 0 | y > 0) {
v2:               if (x >= y)
v3:                     x = x - 1;
                  else
v4:                     y = y -1;
            }
vf:   }
```

Generazione dei casi di test

- Per un certo programma e un certo criterio
  - Costruisci il grafo di flusso del programma
  - Cerca i percorsi nel grafo che devono essere eseguiti per soddisfare quel criterio
  - per ogni percorso cerca i valori di input che inducono quel percorso
  - ottimizza il test suite se possibile

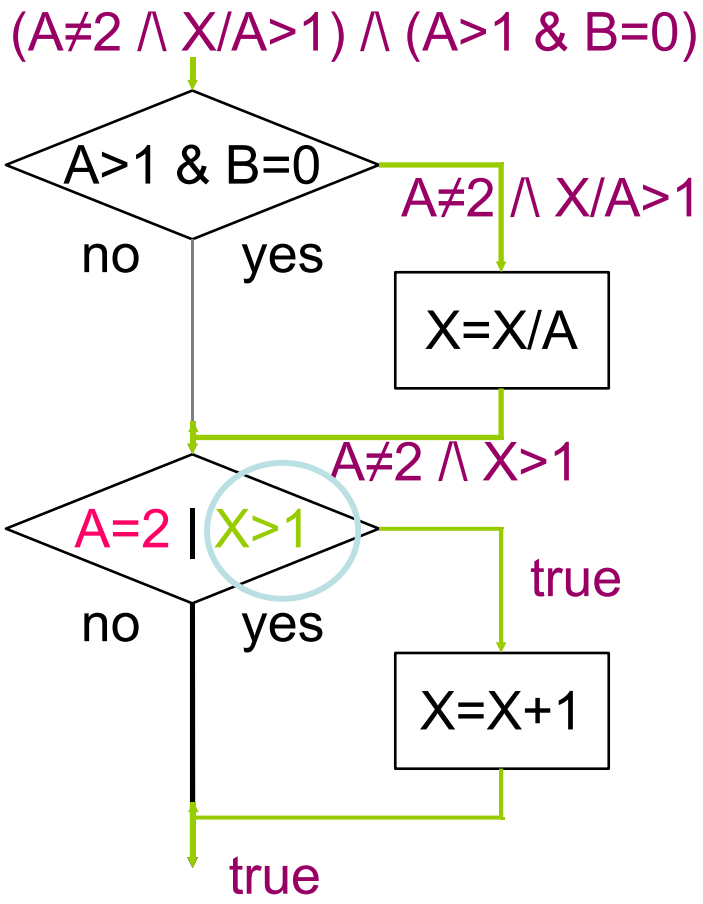
Come trovo i valori per una certa copertura?

- metti true alla fine del cammino che vuoi coprire
- procedi all'indietro sul cammino
- se c'è una ssegnamento, sostituisci la variabile assegnata con il valore che è stato assegnato
- se c'è una ramo "yes" in una decisione aggiungi la decisione alla condizione
- se c'è un no, aggiungi la negazione



Esempio 2

• Se voglio coprire una certa condizione in una decisione , la includo



Debolezza del program-based testing

- Possono non rilevare alcuni errori:
  - Esempio classico “missing path”: un path (o una condizione o un caso) che e’ stato dimenticato nel programma non potrà essere testato
- Non tanto “come scegliere i casi di test” ma “ho testato abbastanza?”

Importanza di una specifica

- il testing e’ una forma di verifica: non può essere fatto senza dei requisiti di riferimento!!
- Anche nel caso di program-based testing i requisiti sono indispensabili: cosa vuol dire che un programma ha passato i test?

Importanza degli oracoli

- La correttezza del comportamento di un programma non dovrebbe essere lasciata al giudizio umano (non è affidabile ed è costoso)
- **Oracle**: una procedura meccanizza che decide se un certo output dato un certo input è accettabile oppure no
- Esempi immediati
  - Oracoli nel codice:
  - `if (x ==0)  printl("ERRORE");`
  - `assert(x>0)` (in Java 1.4)

Infattibilità

- Problema dell'infattibilita':
  - Comportamenti sintatticamente validi (cammini, flussi di dati,...) possono essere non eseguibili
  - come faccio a sapere se un path o un'istruzione (o...) e' fattibile oppure no ? (problema indecidibile)
- i criteri di adeguatezza sono spesso impossibili da soddisfare.
  - Definizione di versioni finite dei criteri introdotti
- Di fronte ad un caso di non copertura:
  - giustificazione manuale dell'omissione di casi di test