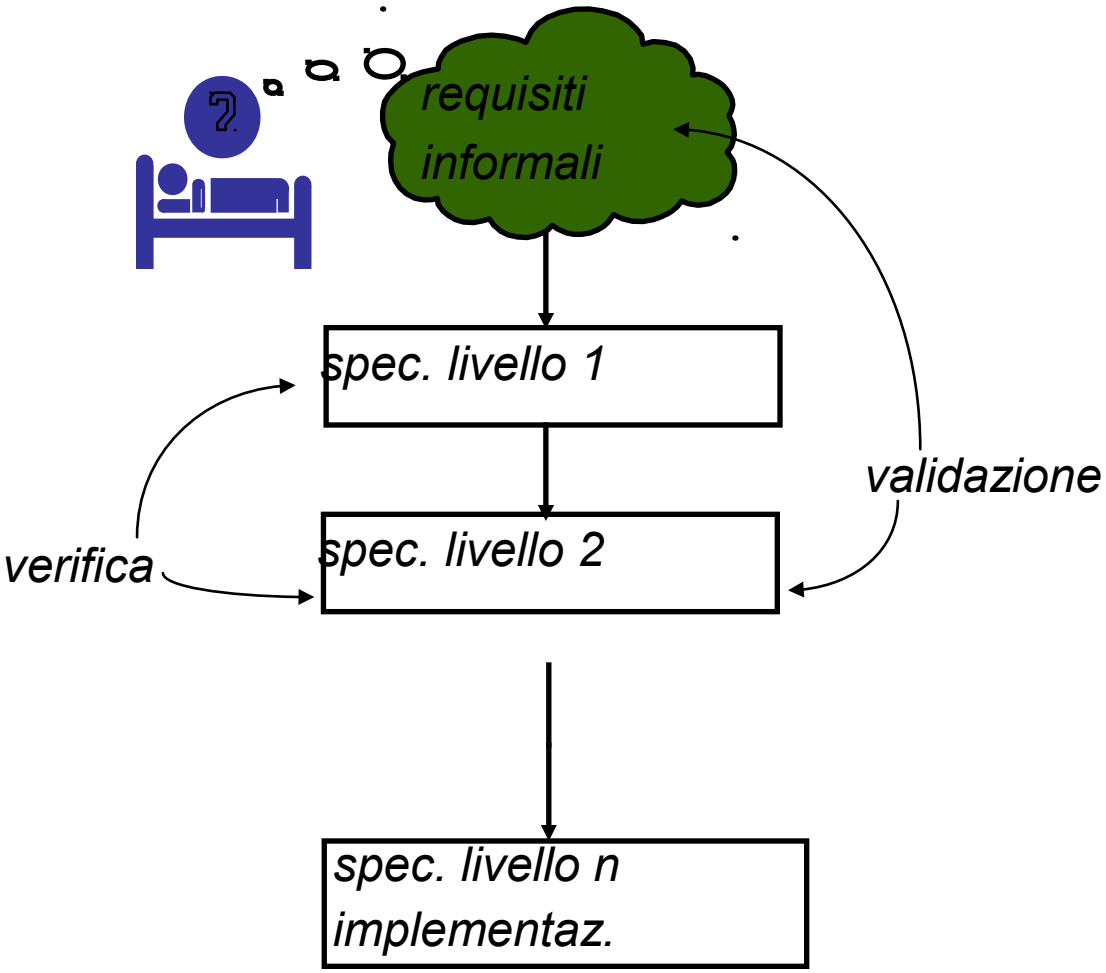
Università
Ca' Foscari
Venezia

Verifica e Validazione

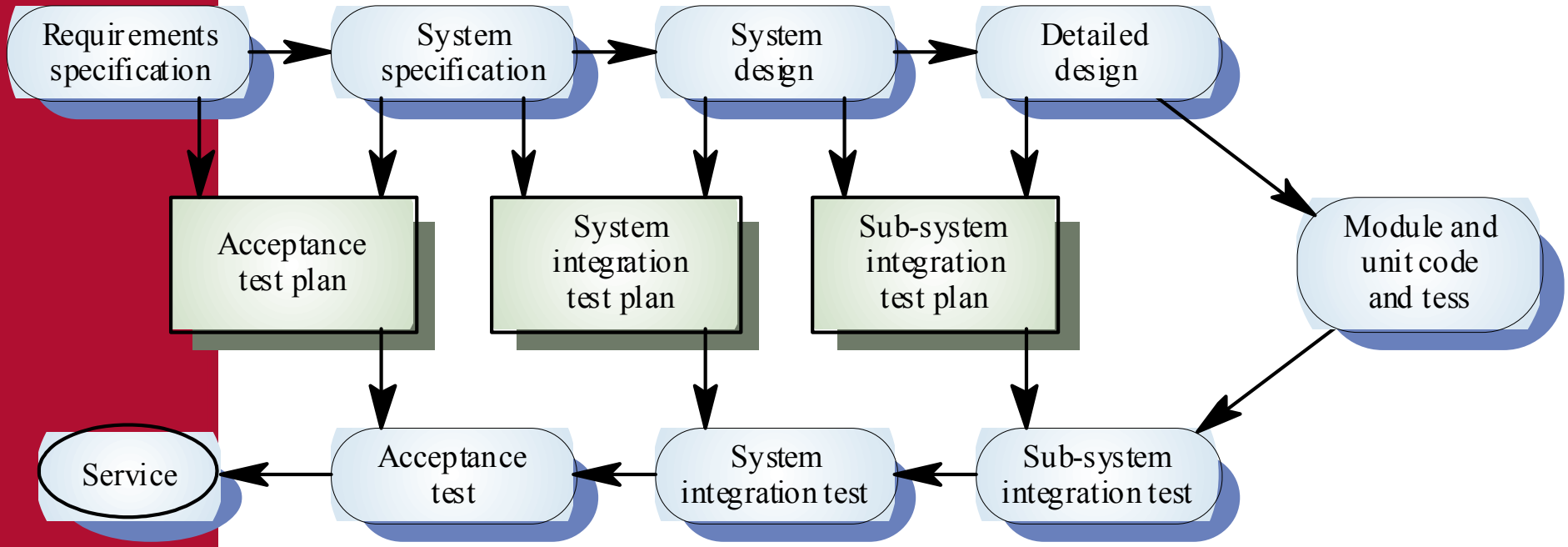
Università
Ca' Foscari
Venezia

Verifica e Validazione



Università Ca' Foscari Venezia

Modello a V del processo software



Testing

Il test può servire per scoprire la presenza di possibili malfunzionamenti, ma non a garantirne l'assenza (Dijkstra)

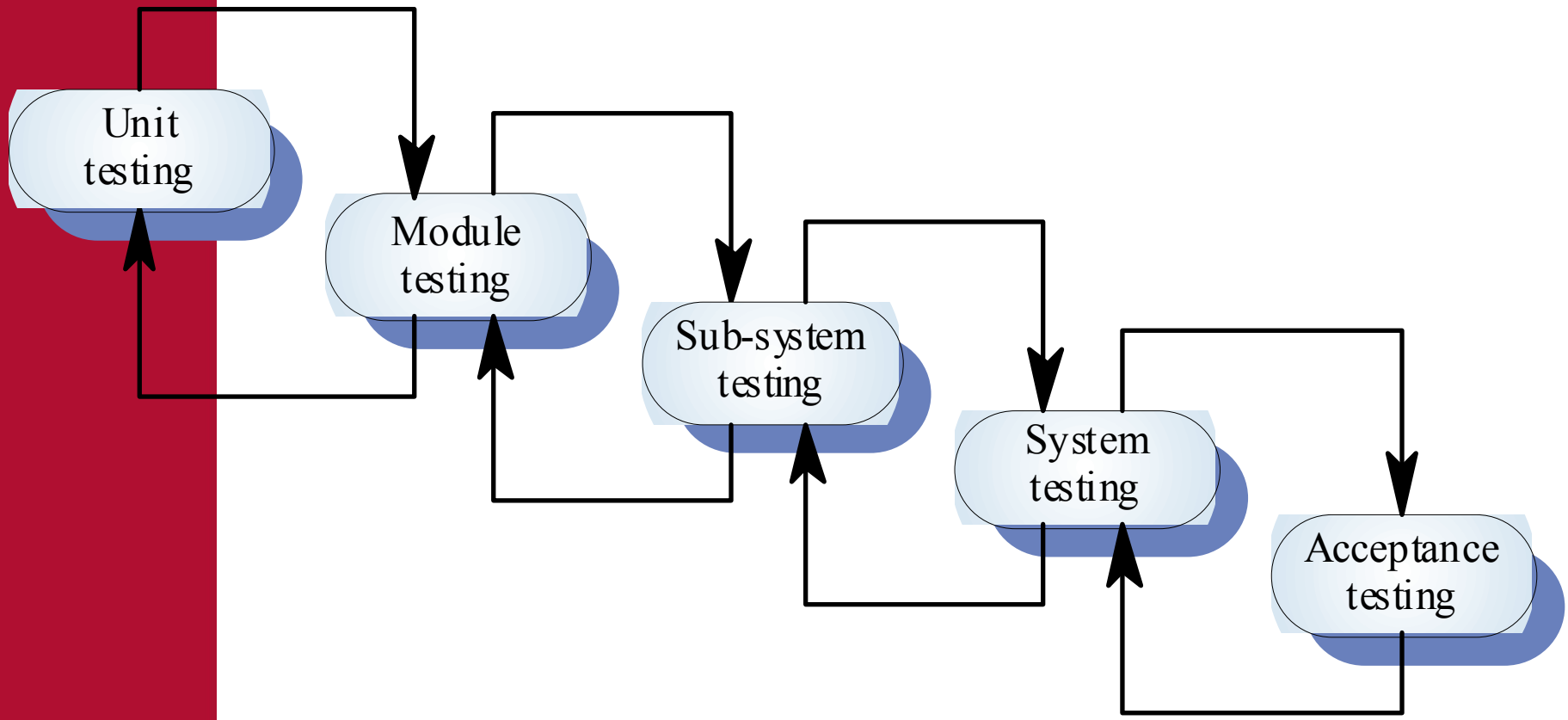
- Un test ha successo se permette di individuare uno o più errori
- Per i requisiti non funzionali possono solo essere utilizzate tecniche di validazione

Priorità nel testing

- I test devono sondare le caratteristiche globali del sistema nel suo insieme più che le singole componenti
- Se il sistema è una nuova versione di un sistema esistente, è più importante testare le vecchie caratteristiche che testare le nuove caratteristiche
- Testare le situazioni tipiche è più importante che testare i valori alla frontiera

Università
Ca' Foscari
Venezia

Fasi del testing



Il piano di testing

E' un documento che deve descrivere:

1. Il processo di testing adottato
2. Tracciabilità dei requisiti
3. Elementi testati
4. Schedule del testing: tempo e risorse allocate
5. Procedure di registrazione dei test
6. Requisiti hardware e software utilizzati
7. Vincoli che condizionano il testing

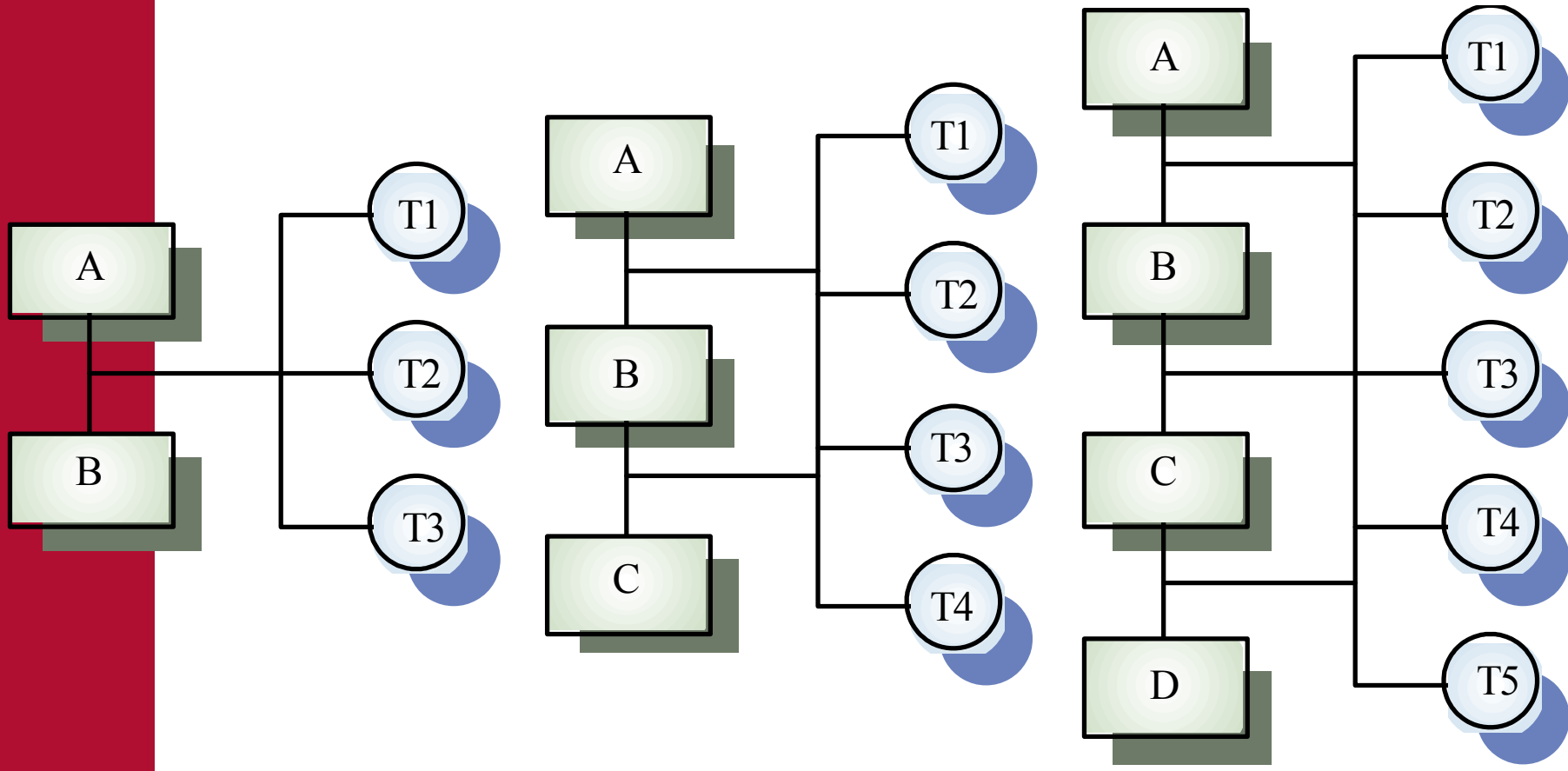
Strategie di testing

- Strategie diverse possono essere applicate nelle diverse fasi del processo di testing

1. Incremental testing
2. Top-down testing
3. Bottom-up testing
4. Thread testing
5. Stress testing
6. Back-to-back testing

Università Ca' Foscari Venezia

Incremental testing



Test sequence 1

Test sequence 2

Test sequence 3

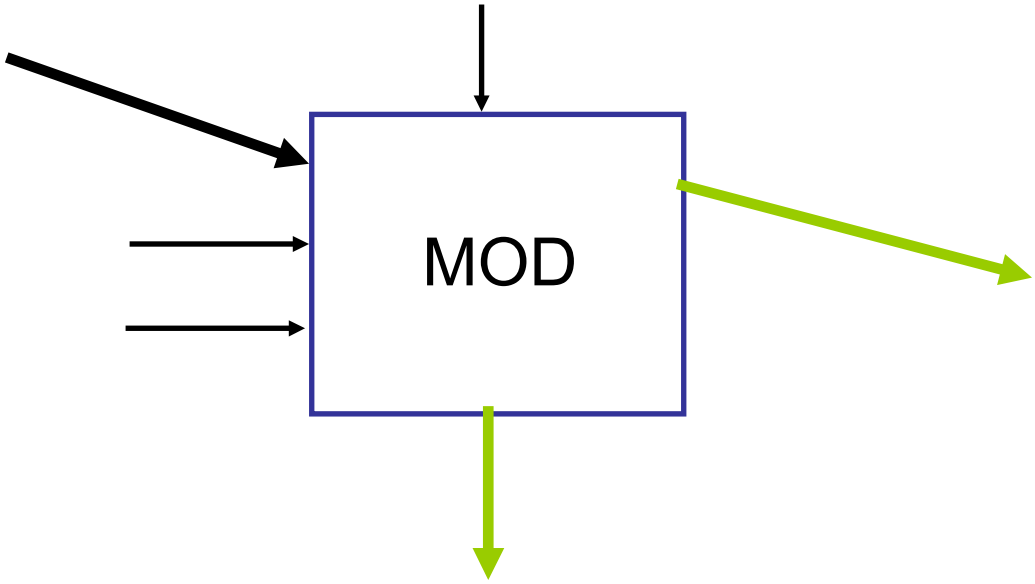
Esperienza

- Circa il 40% di malfunzionamenti si puo' attribuire a problemi di integrazione
- essenzialmente sono errori nell'interpretazione che un modulo fa delle specifiche dell'altro

Università Ca' Foscari Venezia

Test di modulo

- Un modulo fa parte di un sistema
  - è cliente di altri moduli
  - è usato da altri moduli

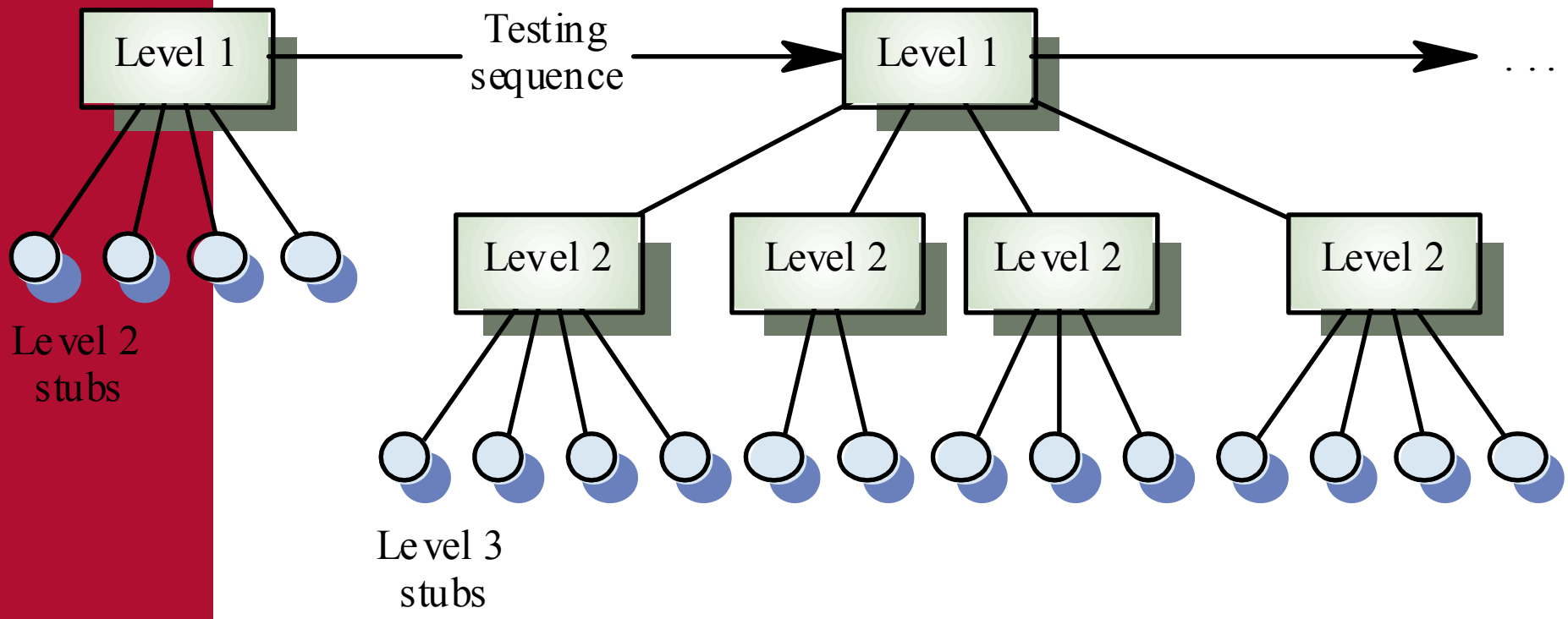


Test di modulo

- Occorre simulare i moduli usati
  - STUB
- Occorre simulare i moduli che lo usano
  - DRIVER
- Caso di MOD sottoprogramma
  - DRIVER
    - inizializza eventuali globali
    - chiama
  - STUB
    - uno per sottoprogramma usato

Università Ca' Foscari Venezia

Top-down testing



Top-down testing

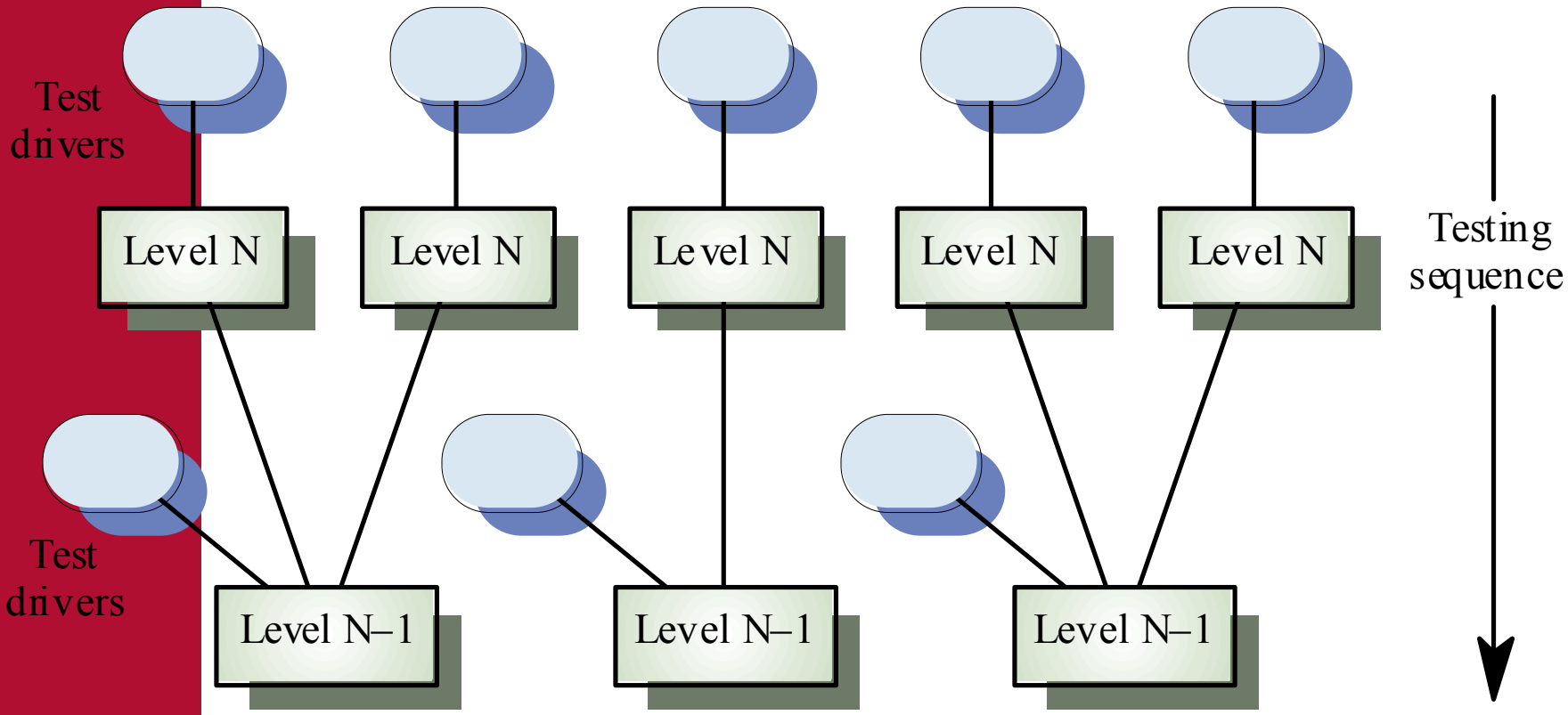
- Inizia con i livelli più alti del sistema e procede all'ingiù: le sottocomponenti sono rappresentate da "stub" (ceppi, monconi), che hanno la stessa interfaccia delle sottocomponenti ma funzionalità limitata
- E' una strategia di testing adatta quando si procede in uno sviluppo top-down
- Individua rapidamente errori architetturali
- Può richiedere di aver già completa la struttura del sistema, prima di poter iniziare qualsiasi test
- Può risultare difficile sviluppare gli "stub"

Casi possibili di "stub"

- Il chiamato non fa nulla (eventualmente stampa la traccia)
- Il chiamato colloquia con il programmatore per calcolare il risultato atteso
- Il chiamato è una versione semplificata (un prototipo) del modulo che verrà chiamato

Università Ca' Foscari Venezia

Bottom-up testing



Bottom-up testing

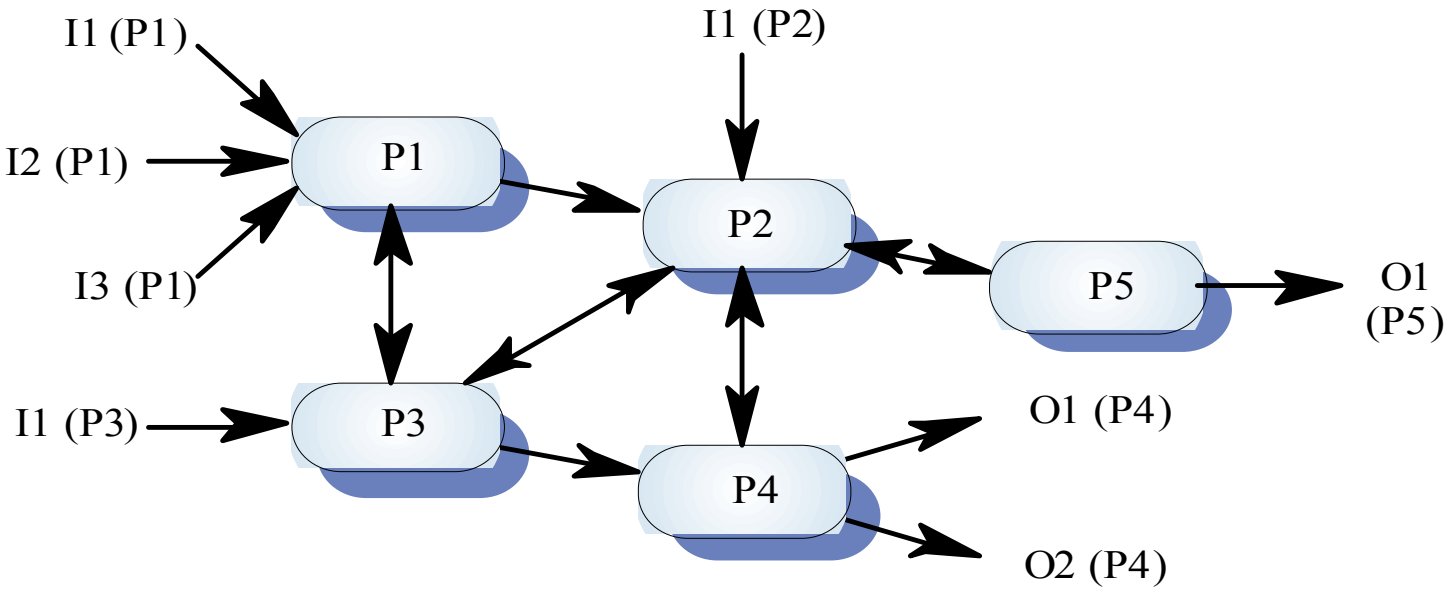
- I vantaggi del bottom-up sono gli svantaggi del top-down (e viceversa)
- Si inizia dalle componenti più a basso livello e si lavora all'insù, realizzando dei "drivers" che simulano l'ambiente nel quale le componenti sono valutabili
- Non individua errori di progettazione di alto livello se non molto avanti nel processo
- E' appropriato per sistemi object-oriented

Thread testing

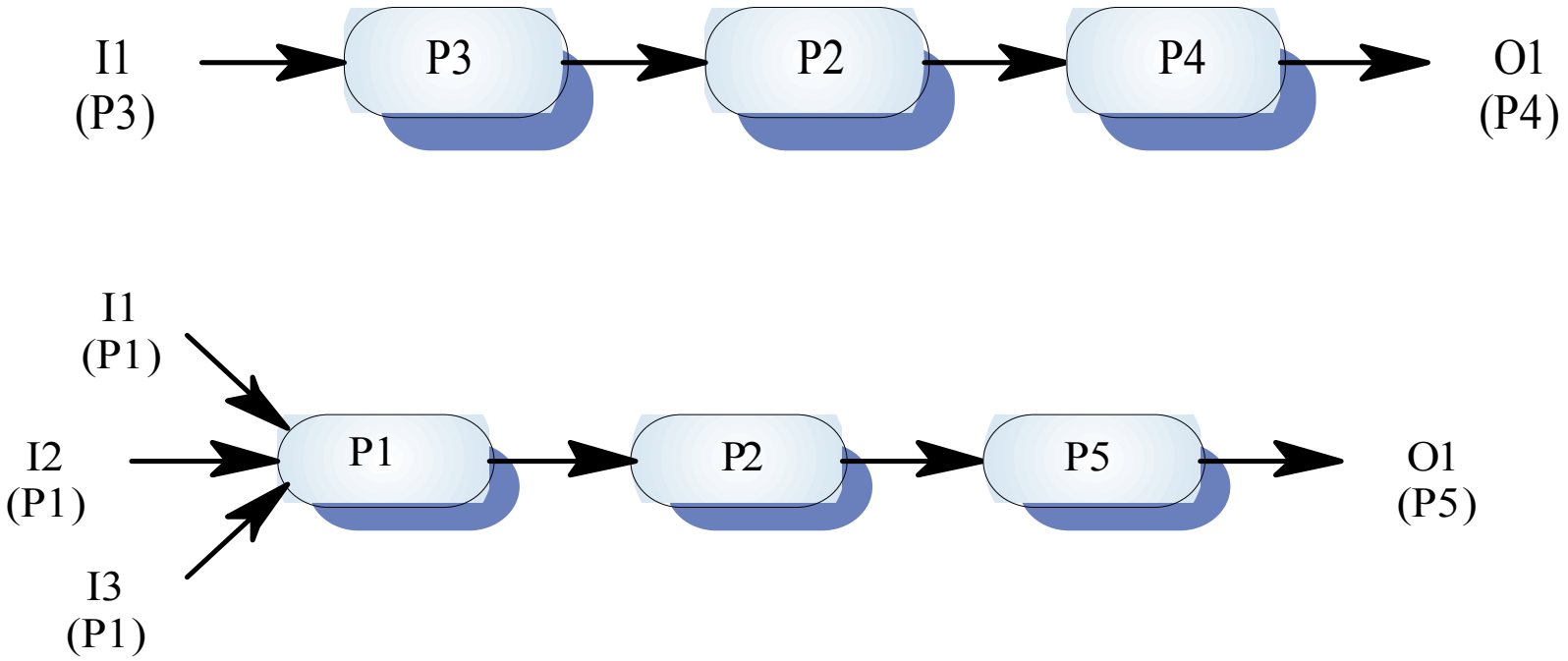
- Adatto a sistemi real-time e ad oggetti
- Si basa sul testare un'operazione che comporta una sequenza di passi di processo che sono legati da uno stesso thread (filo) nel sistema
- Inizia con thread legati a un singolo evento e poi viene reso più complesso testando threads a eventi multipli
- Può essere impossibile un "threading test" completo, a causa del numero elevato di combinazioni di eventi

Università
Ca' Foscari
Venezia

Esempio: interazione di processi

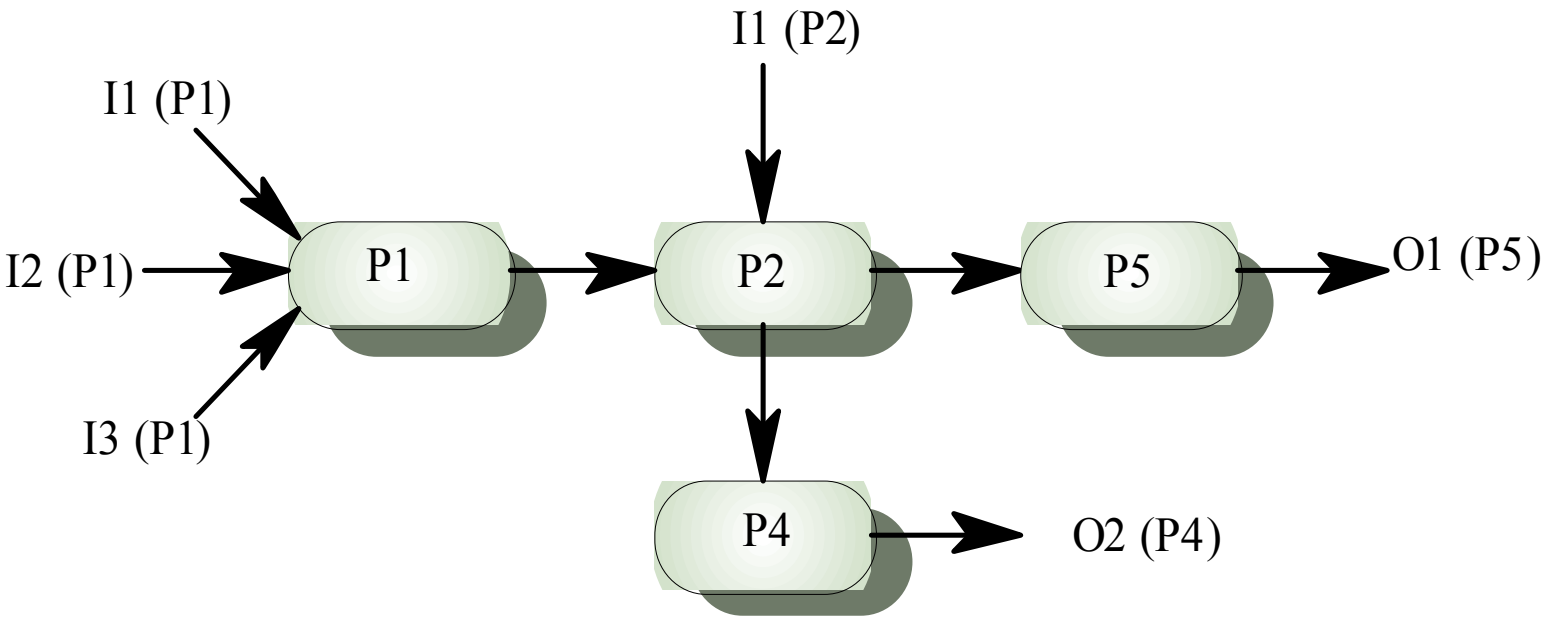


Thread testing



Università Ca' Foscari Venezia

Multiple-thread testing



Stress testing

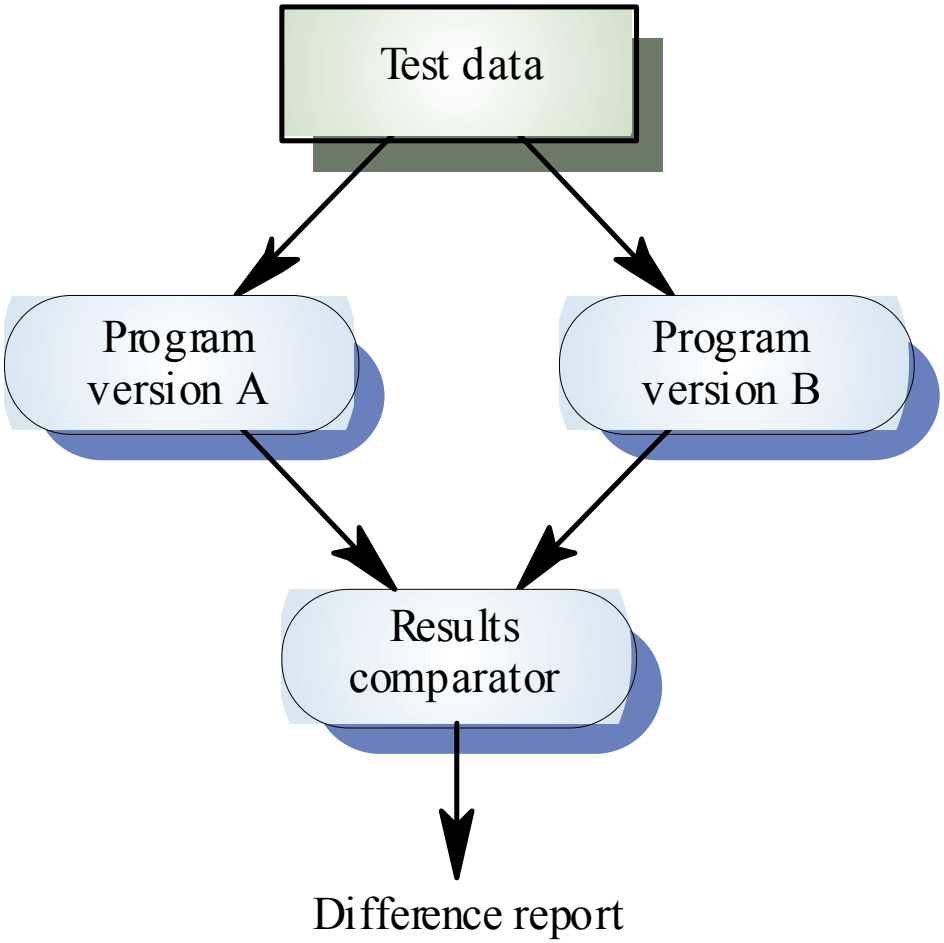
- Ha come obiettivo verificare che il sistema sopporti il carico massimo previsto in fase di progettazione. Il sistema viene testato oltre i limiti finché fallisce
- Testa il comportamento del sistema in caso di fallimento, e verifica le conseguenti perdite di dati o di servizi
- Particolarmente importante in sistemi distribuiti, che possono subire degrado quando la rete è troppo carica

Back-to-back testing

- Testa diverse versioni del programma con lo stesso input, e confronta gli output.
  Se l'output è diverso, ci sono errori potenziali
- Riduce il costo di esaminare il risultato dei test: il confronto degli output può essere automatizzato
- Si può usare quando è disponibile un prototipo, quando il sistema vine sviluppato in più versioni (su diverse piattaforme), o nel caso di upgrade di sistema

Università Ca' Foscari Venezia

Back-to-back testing



Dati di test e casi di test

- Dati di test
  L'insieme di input che devono essere costruiti per testare il sistema

- Casi di test
  L'insieme di input per testare il sistema e gli output previsti in corrispondenza di questi input se il sistema soddisfa la sua specifica

Università Ca' Foscari Venezia

Terminologia

- Per definire i Casi di Test devo adottare un criterio:

- Criterio C affidabile
  - i test selezionati da C producono lo stesso risultato (successo/ insuccesso)
- Criterio C valido
  - qualora P non sia corretto, ∃ un test T selezionato dal criterio C che ha successo (ovvero fa emergere un'anomalia del programma)

Università Ca' Foscari Venezia

Esempio

```
program RADDOPPIA ...
    ....
    read (x);
    y = x*x;
    write (y);
    ...
```

- Se C seleziona solo {0,2}
  - è affidabile, non valido
- Se C seleziona tutti i sottoinsiemi di {0,1,2,3,4}
  - è valido, non affidabile
- Se C seleziona insiemi di almeno 3 elementi distinti che contengono 0 o 2 e almeno un elemento diverso da 0 e 2
  - è valido e affidabile

Università Ca' Foscari Venezia

Elementi di teoria dei test

Teorema di Goodenough e Gerhart

Se

Esiste un C affidabile e valido per P

e

T è selezionato da C

e

T non ha successo su P

allora

P è corretto

Università
Ca' Foscari
Venezia

Elementi di teoria dei test

Teorema di Howden

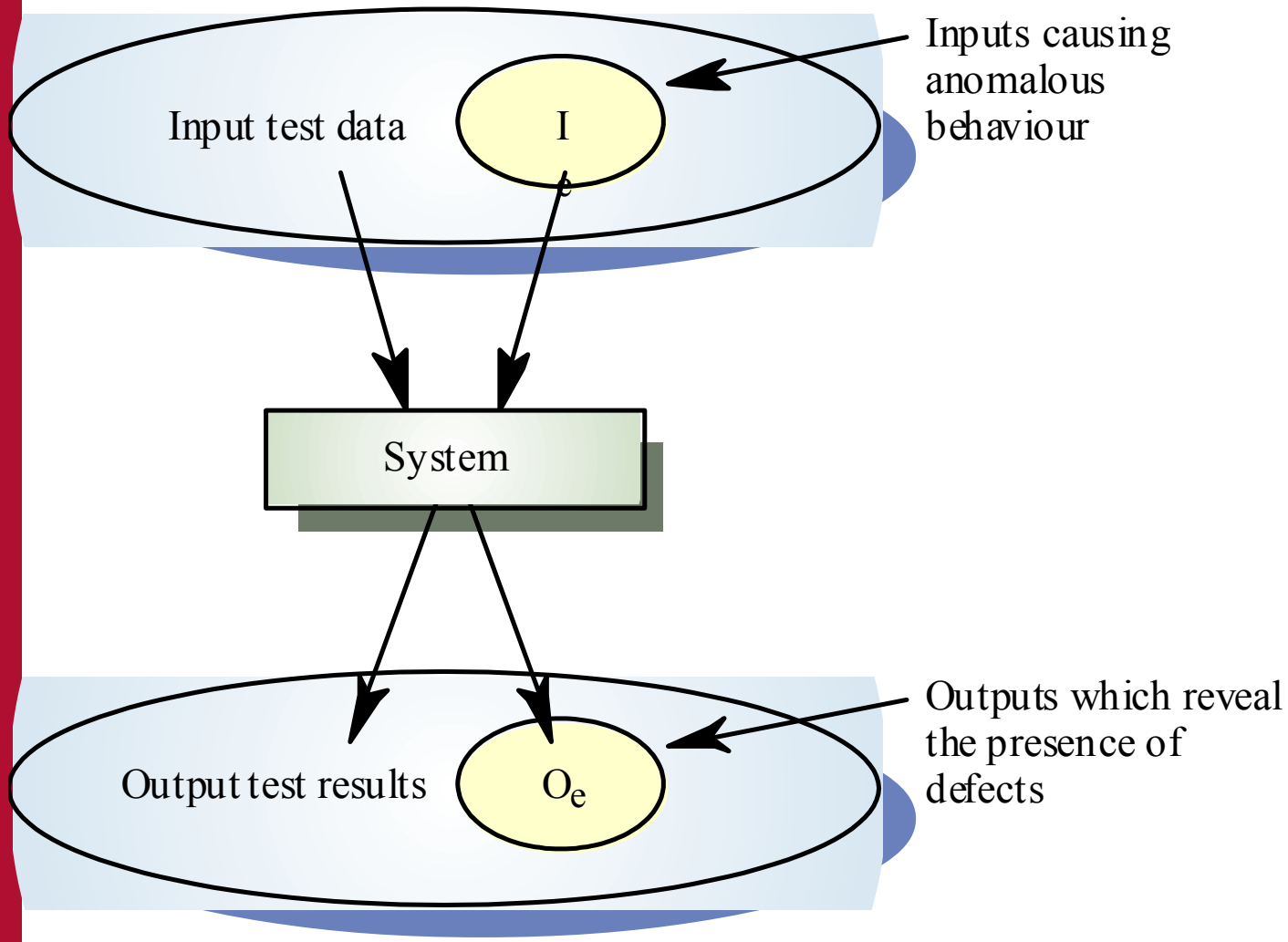
Non si può costruire meccanicamente (mediante un programma) un test finito che soddisfi un criterio affidabile e valido

Black-box testing

- Approccio al testing in cui il sistema è visto come una "scatola nera"
- I casi di test sono basati sulla specifica del sistema
- La pianificazione può iniziare molto presto nel processo software

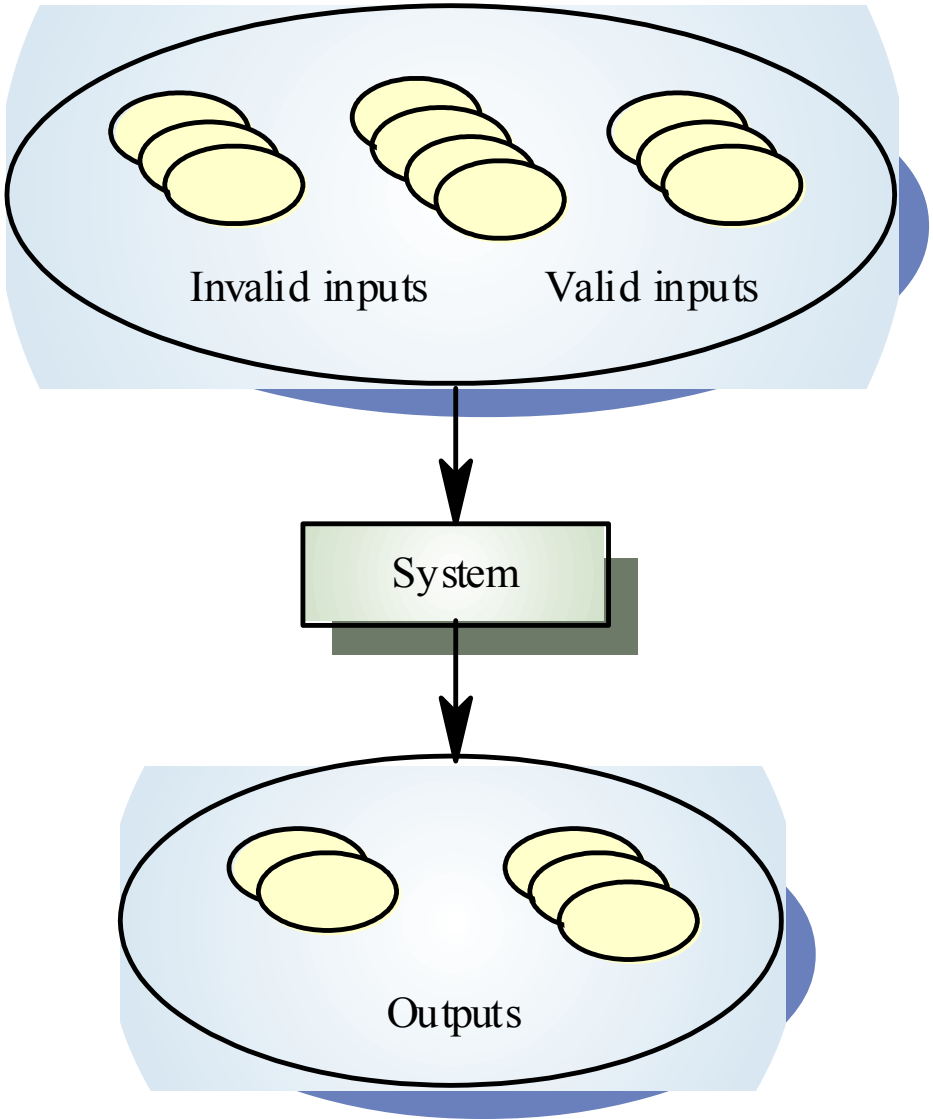
Università
Ca' Foscari
Venezia

Black-box testing



Università Ca' Foscari Venezia

Equivalence partitioning

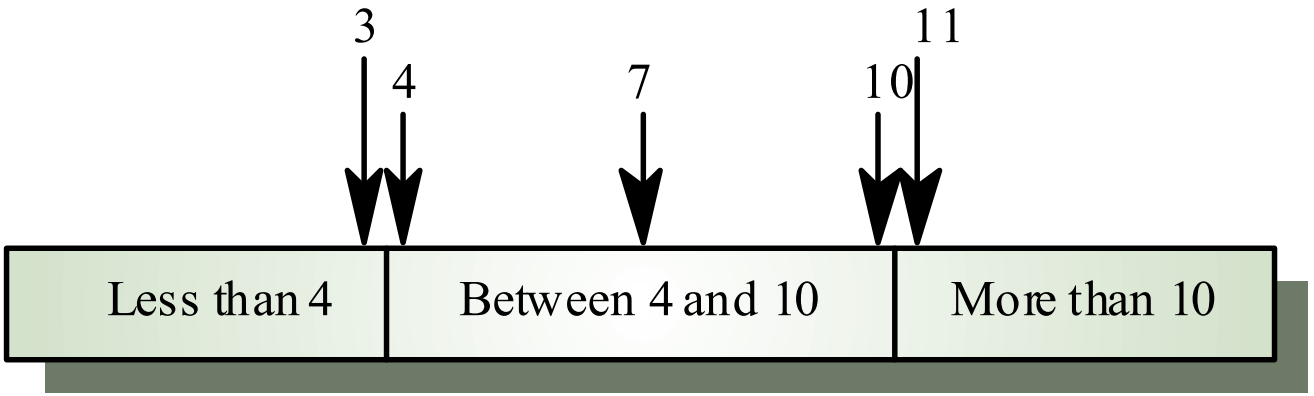


Università Ca' Foscari Venezia

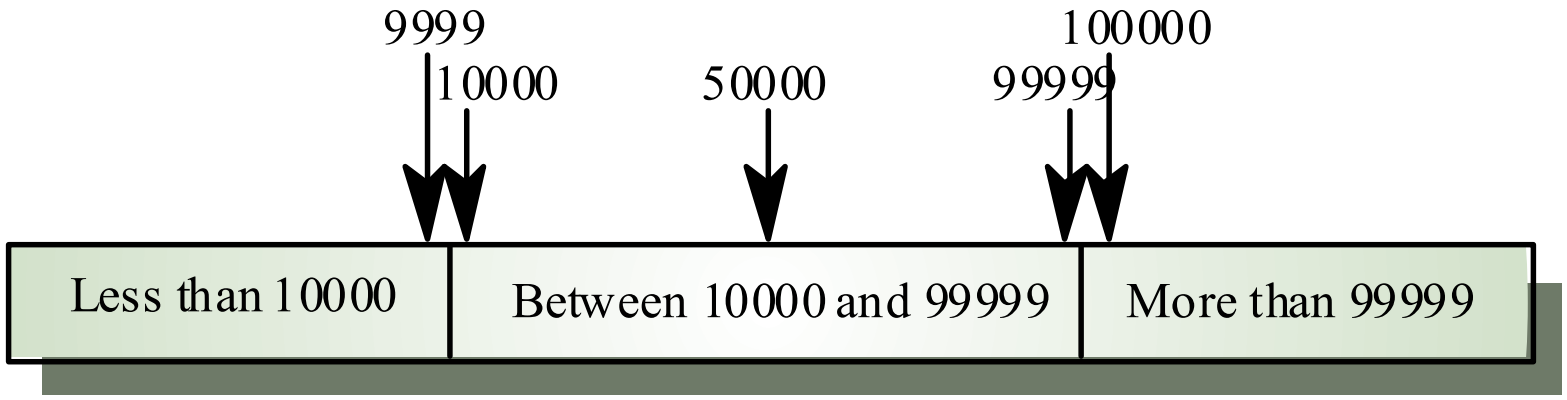
Equivalence partitioning

- Partiziona gli input e gli output del sistema in "insiemi equivalenti"
  - Se l'input è un insieme dai 4 ai 10 interi con valori tra 10,000 e 99,999, le classi di equivalenza sono:
    - < 10.000
    - da 10.000 a 99.999 (compresi)
    - >= 100.000
- Scegli un insieme di 3,4,10 e 11 casi di test
- Scegli i casi di test ai confini di questi insiemi
  - 0, 9.999, 10000, 99.999, 100.000, 150.000

Partizioni



Number of input values



Input values

Specifica di una routine di ricerca

procedure Search (Key : ELEM ; T: ELEM_ARRAY;
Found : **in out** BOOLEAN; L: **in out** ELEM_INDEX) ;

Pre-condition

-- the array has at least one element
T'FIRST <= T'LAST

Post-condition

-- the element is found and is referenced by L
(Found and T (L) = Key)

or

-- the element is not in the array
(**not** Found **and**
not (**exists** i, T'FIRST >= i <= T'LAST, T (i) = Key))

Partizioni di input

- Input che soddisfano le precondizioni
- Input dove una precondizione non vale
- Input dove l'elemento chiave è un membro dell'array
- Input dove l'elemento chiave non è un membro dell'array

Routine di ricerca: partizioni di input

Array	**Element**
Single value	In array
Single value	Not in array
More than 1 value	First element in array
More than 1 value	Last element in array
More than 1 value	Middle element in array
More than 1 value	Not in array

Routine di ricerca: casi di test

Input array (T)	**Key** (Key)	**Output** (Found, L)
17	17	true, 1
17	0	false, ??
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 7
17, 18, 21, 23, 29, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??

Testing strutturale

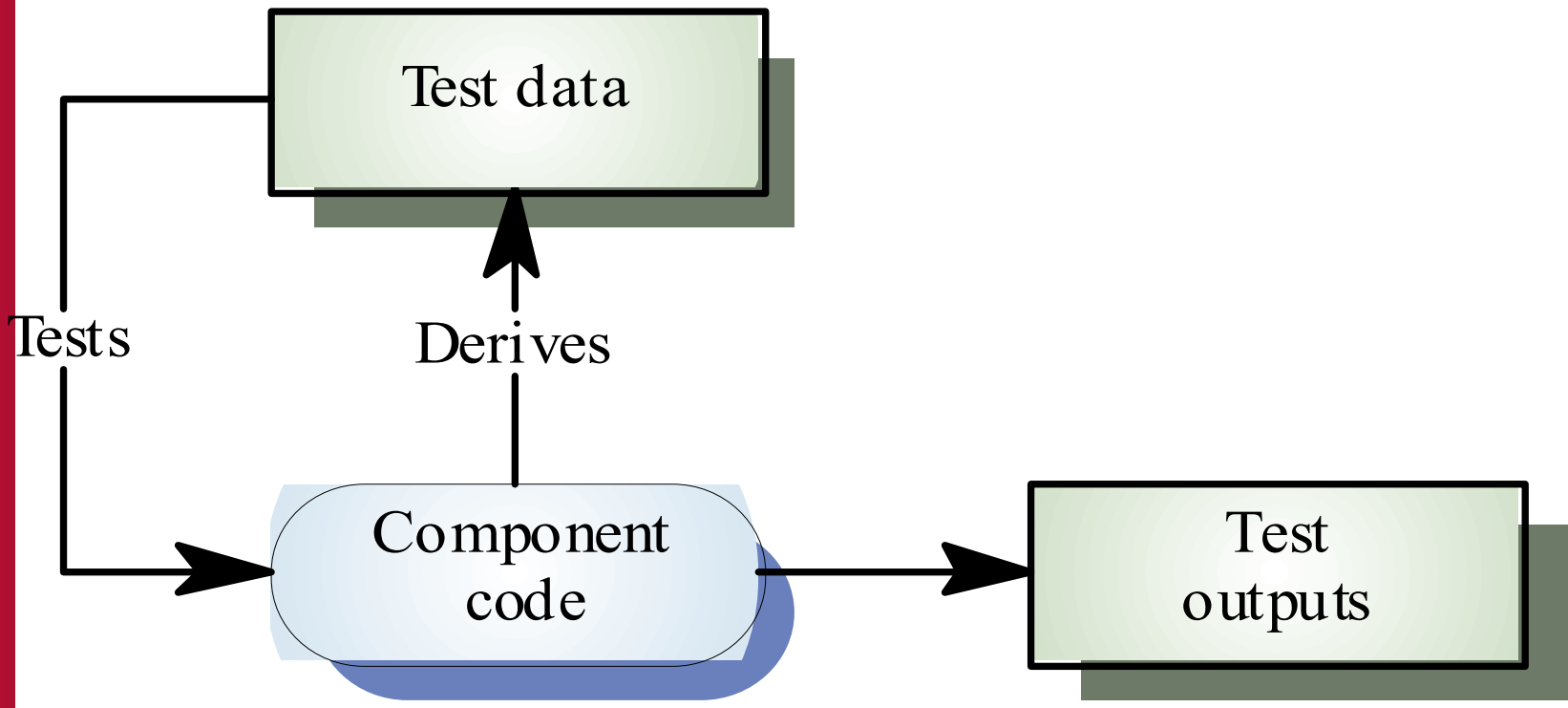
- Chiamato anche **white-box** testing
- I casi di test sono ottenuti a partire dalla struttura del programma. La conoscenza del programma viene utilzzata per identificare altri ulteriori casi di test
- Obiettivo: vagliare tutti i comandi del programma (non tutti i cammini di computazione)

Criteri White-box

- Sono criteri di selezione dei casi di test basati su concetti di "copertura" della struttura interna del programma
- Congettura: se un programma è stato poco sollecitato dai dati di test, potenzialmente contiene anomalie
- Definito il livello desiderato di "copertura" è possibile valutare quanto progressivamente ci si avvicina all'obiettivo

Università
Ca' Foscari
Venezia

White-box testing



Copertura

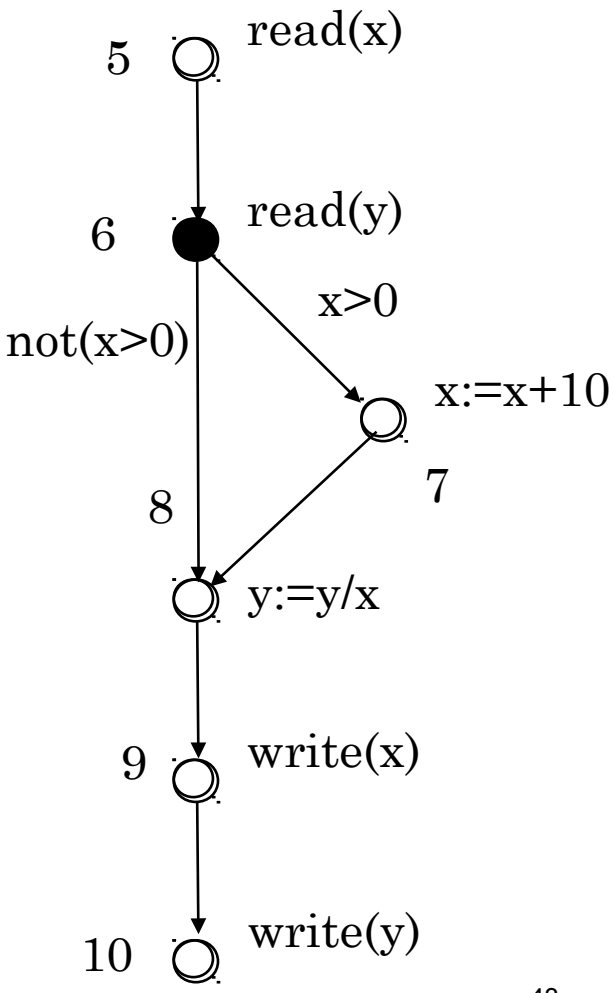
1. Copertura delle istruzioni

- Obiettivo: esercitare almeno una volta ogni istruzione durante il test
- Motivazione: altrimenti ci possono essere computazioni mai osservate

Università Ca' Foscari Venezia

Esempio

```
1   Program statement (input, output);
2           var
3                   x,y : real;
4           begin
5                   read(x);
6                   read(y);
7                   if x >  0 then x:=x+10;
8                   y:=y/x;
9                   write(x);
10                  write(y);
11          end.
```



S = {(x = 20, y = 30)} soddisfa il criterio

Copertura

2. Copertura delle decisioni

- Ogni arco del flusso di controllo deve venire percorso

S = {(x = 20, y = 30), (x = -3, y = 100} soddisfa il criterio

Ma continua a non far emergere il malfunzionamento!

Università Ca' Foscari Venezia

Arricchimento del criterio

- Utilizzare valori "ai limiti" dei campi di variabilità delle decisioni, oltre a valori "all'interno"
- Esempio

  se deve essere x >= 0,

  testare x = 0, oltre che x>0 e x<0
- Attitudine
  - "fare l'avvocato del diavolo"

Esempio

```
1       read (x,y);
2       if x = 10 then
3               x := x - 10
4       else    x := |x| +1;
5       if y <= 0 then
6               y := (y+1)/x
7       else    y := x/y;
8       ...
```

T = {(10, 5), (0, -3)} copre tutte le decisioni

Così facendo si coprono i cammini "then...else" e "else...then", non il cammino "then...then" che genera un malfunzionamento.

Copertura

3. Copertura dei cammini

- Test strutturale esaustivo: tutti i cammini… ma il numero di cammini è infinito!
- Occorre limitare il numero: quali?
- Criteri empirici per limitare il numero delle iterazioni
- Il numero dei dati di test tende comunque a crescere in maniera non controllabile.....

Ricerca binaria (Ada)

```
procedure Binary_search (Key: ELEM; T: ELEM_ARRAY;
      Found: in out BOOLEAN; L: in out ELEM_INDEX) is
   - Preconditions
   -- T'FIRST < =T'LAST and
   -- forall i: T'FIRST..T'LAST-1, T (i) <= T(i+1)
   Bott : ELEM_INDEX := T'FIRST;
   Top : ELEM_INDEX := T'LAST;
   Mid : ELEM_INDEX;
begin
   L := (T'FIRST + T'LAST ) / 2;
   Found := T( L ) = Key;
    while Bott <= Top and not Found loop
        Mid := (Top + Bott) mod 2;
        if T( Mid ) = Key then
            Found := true;
            L := Mid;
        elsif T( Mid ) < Key then
            Bott := Mid + 1;
        else
            Top := Mid - 1;
        end if;
   end loop;
end Binary_search;
```

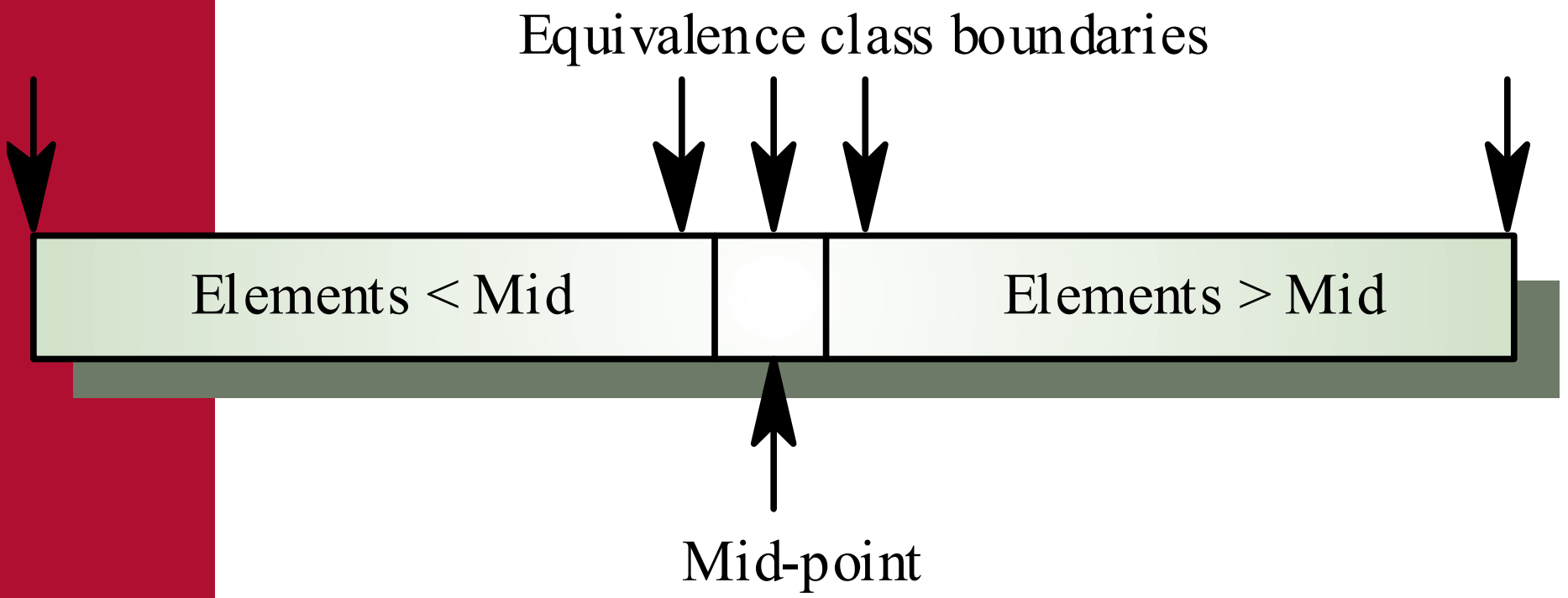
Ricerca binaria: partizioni di input

- Chiave nell'array
- Chiave non nell'array

- Array di input con un solo elemento
- Array di input con numero pari di elementi
- Array di input con numero dispari di elementi

Università
Ca' Foscari
Venezia

Ricerca binaria: partizioni di input



Ricerca binaria: casi di test

Input array (T)	**Key** (Key)	**Output** (Found, L)
17	17	true, 1
17	0	false, ??
17, 21, 23, 29	17	true, 1
9, 16, 18, 30, 31, 41, 45	45	true, 7
17, 18, 21, 23, 29, 38, 41	23	true, 4
17, 18, 21, 23, 29, 33, 38	21	true, 3
12, 18, 21, 23, 32	23	true, 4
21, 23, 29, 33, 38	25	false, ??

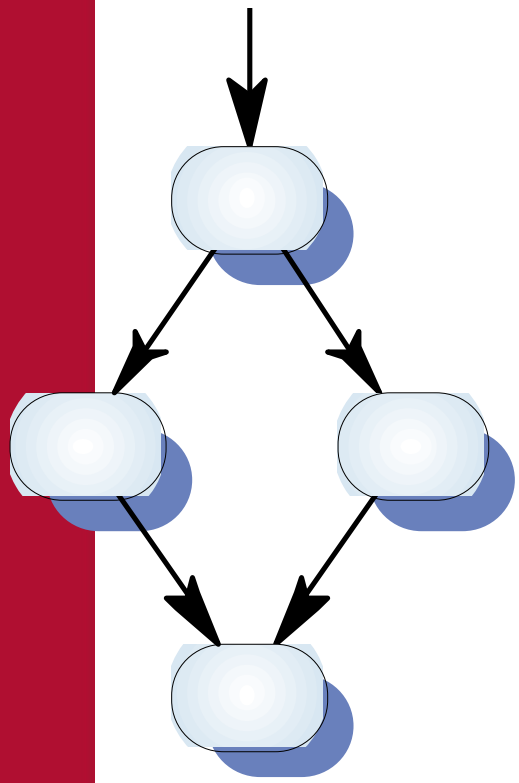
Grafi di flusso del programma

- Descrivono il flusso di controllo nel programma usati per calcolare la complessità ciclomatica
- Complessità =

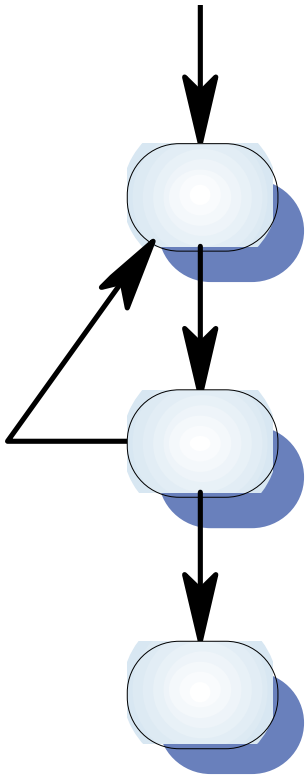
  Numero di archi - Numero di nodi + 1

Università Ca' Foscari Venezia

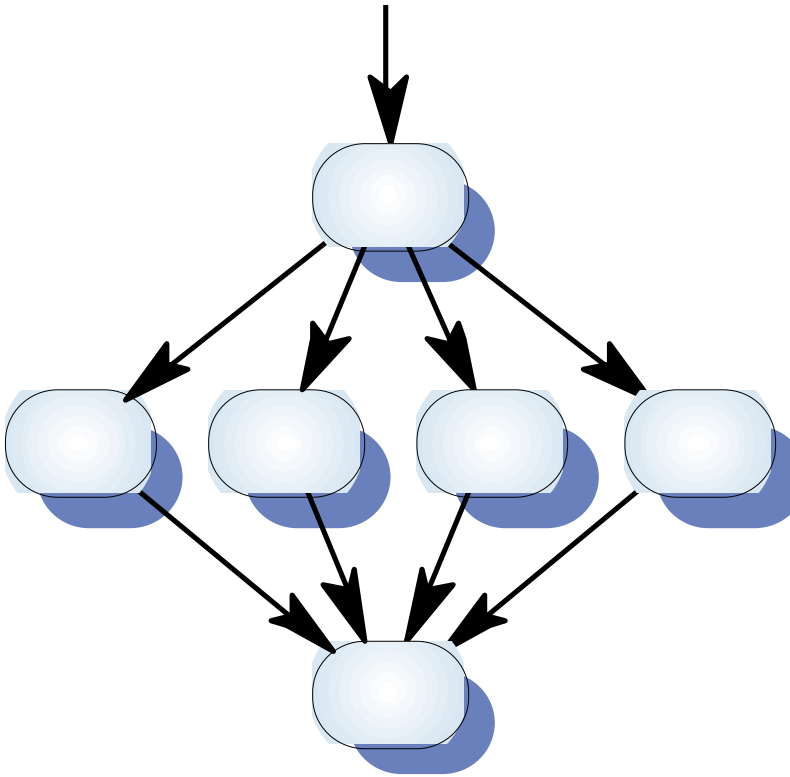
Rappresentazione di un grafo di flusso di un programma



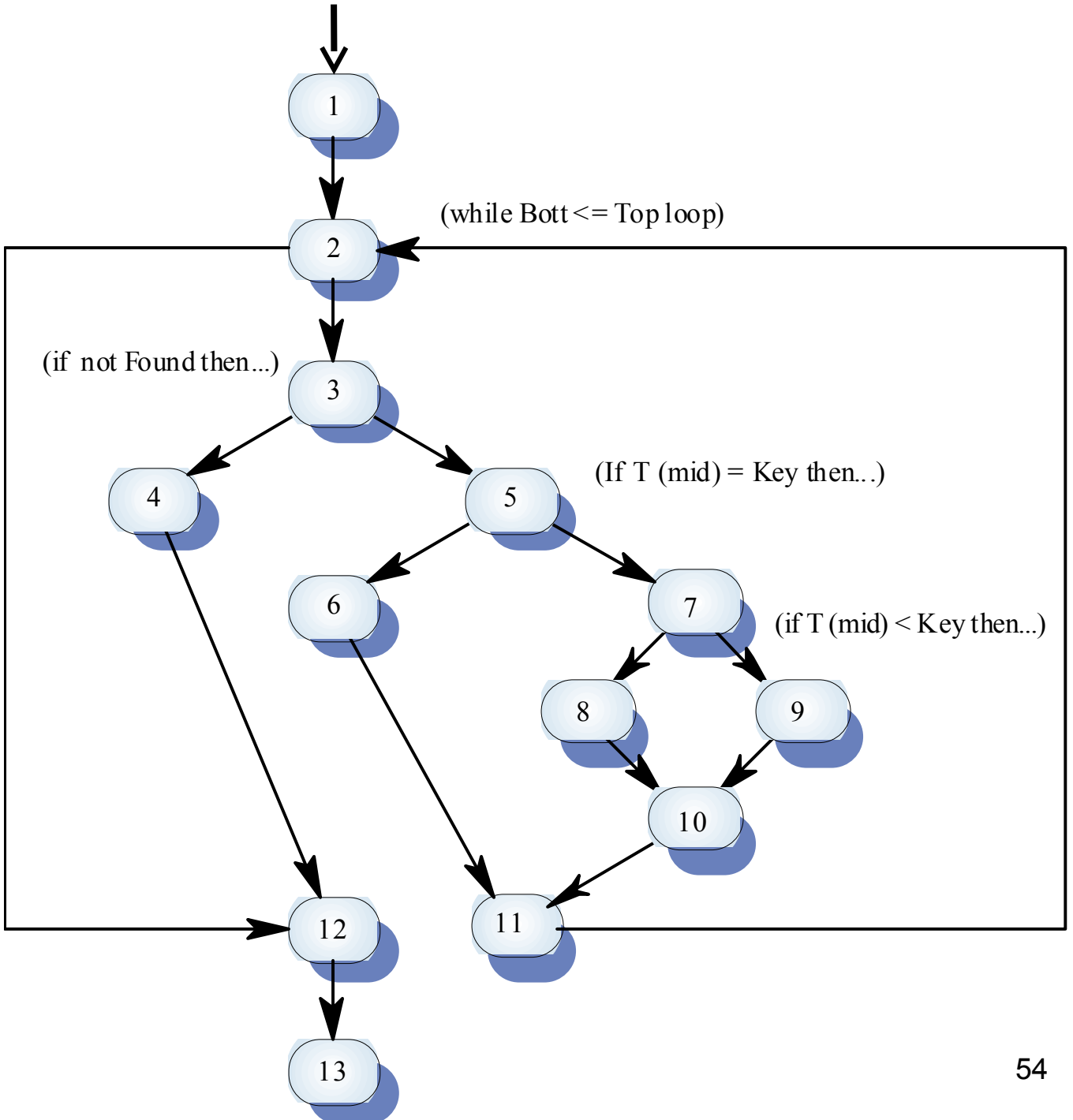
if-then-else



loop-while



case-of


1
2
(while Bott <= Top loop)
(if not Found then...)
3
4
5
(If T (mid) = Key then...)
6
7
(if T (mid) < Key then...)
8
9
10
11
12
13

Cammini indipendenti

- 1, 2, 3, 4, 12, 13
- 1, 2, 3, 5, 6, 11, 2, 12, 13
- 1, 2, 3, 5, 7, 8, 10, 11, 2, 12, 13
- 1, 2, 3, 5, 7, 9, 10, 11, 2, 12, 13
- 1, 2, 3, 5, 7, 9, 10, 11, 2, 3, 4, 12, 13
- Bisogna costruire dei casi di test in modo che tutti questi cammini siano percorsi
- Si può utilizzare un analizzatore dinamico per verificare che i cammini siano stati eseguiti

Complessità ciclomatica

- Il numero di test per controllare tutti i comandi di controllo è uguale alla complessità ciclomatica
- La complessità ciclomatica è uguale al numero di condizioni booleane nel programma (+1)
- E' utile se usata con attenzione. Non è sempre adeguata come test e non può essere usata per programmi data-driven

Programmi data-driven

```
case A is
        when "One" => i := 1 ;
        when "Two" => i := 2 ;
        when "Three" => i := 3 ;
        when "Four" => i := 4 ;
        when "Five" => i := 5 ;
end case ;
```

```
Strings: array (1..4) of STRING :=
   ("One", "Two", "Three", "Four", "Five");
 i := 1 ;
loop
        exit when Strings (i) = A ;
        i := i + 1 ;
end loop ;
```

Verifica Statica

- Verificare la corrispondenza tra un sistema software e la sua specifica senza eseguire il codice
- Consiste nell'analizzare in modo automatico o manuale i listati dei programmi
- Ha il vantaggio di scoprire errori presto, durante il processo di codifica, ed usualmente è più cost-effective del "detect testing" a livello di moduli
- Permette di combinare la ricerca di errori con altri tipi di valutazioni di qualità

Efficacia della verifica statica

- Più del 60% degli errori possono essere individuati con ispezione informale del codice. Più del 90% degli errori possono essere individuabili usando teniche formali di verifica
- Il processo di individuazione degli errori non è condizionato dalla presenza di errori precedenti
- Gli errori possono essere di tipo logico, o anomalie nel codice che possono indicare una condizione erronea (es. variabili non inizializzate) o inadeguatezza rispetto agli standards

Inspection

- Obiettivi
  - Rivelare la presenza di difetti
  - Eseguire una lettura mirata del codice
- Agenti
  - Persone diverse da chi ha sviluppato il software
- Strategia
  - Focalizzare la ricerca su aspetti ben definiti (error guessing)

Attività dell'Inspection

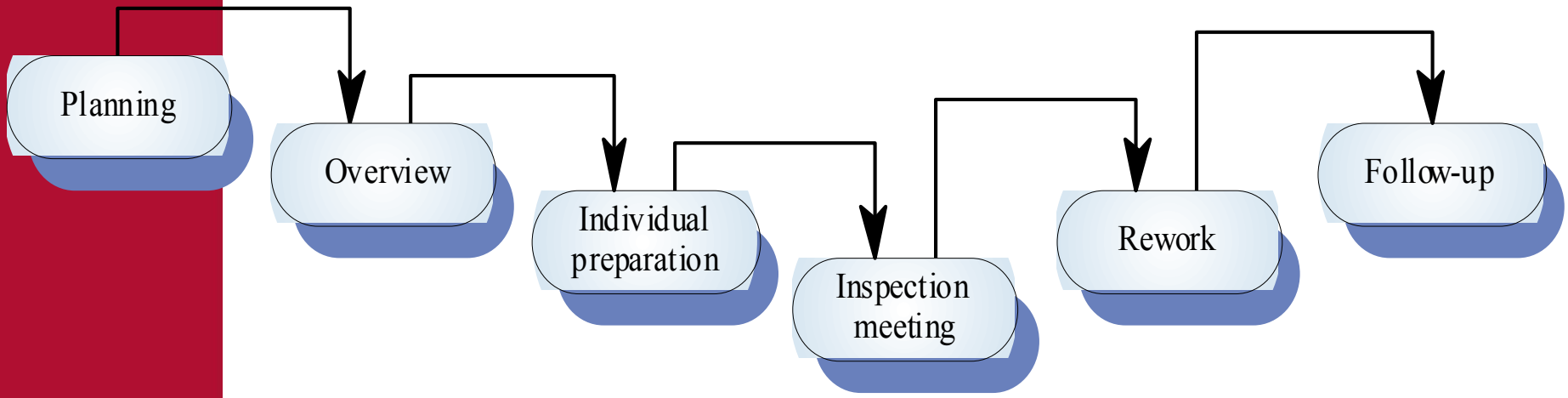
- Fase 1: pianificazione
- Fase 2: definizione della check-list
- Fase 3: lettura del codice
- Fase 4: correzione dei difetti
- Documentazione

Ispezione

- Un'overview del sistema deve essere presentata al team di ispezione
- Precedentemente, deve essere fornito al team di ispezione il codice e tutta la documentazione
- Il team di ispezione è costituito da almeno 4 persone
  - L'**autore** del codice che dev'essere ispezionato
  - Un **lettore** che legge il codice al team
  - Un **ispettor**e che trova errori, omissioni e inconsistenze
  - Un **moderatore**, che prende nota degli errori scoperti

Università Ca' Foscari Venezia

Il processo di ispezione



Precondizioni

- Deve essere disponibile una **specifica** precisa
- Deve essere disponibile **codice** sintatticamente corretto
- Bisogna preparare una **checklist** di possibili errori
- Obiettivo chiaro: individuare gli errori, non i colpevoli! (non bisogna usare l'ispezione per stroncare gli sviluppatori del codice).

Un po' di numeri...

- 500 comandi/ora durante l'overview
- 125 comandi/ora durante la preparazione individuale
- 90/125 comandi/ora durante l'ispezione
- Quindi l'ispezione di 500 linee di codice può costare circa 40 ore/uomo.

Checklist degli errori

- L'ispezione dev'essere guidata da una checklist di errori comuni, che dipende fortemente dal linguaggio di programmazione usato
- Più il linguaggio è a basso livello, più lunga è la lista degli errori
- Ad esempio: inizializzazione, nome delle costanti, terminazione dei cicli, limiti degli array, ecc.

Ispezione del codice

- Uso di elenco (checklist) di possibili situazioni erronee da ricercare (dipende dal linguaggio):
  - uso di variabili non inizializzate
  - salti all'interno di cicli
  - assegnamenti incompatibili
  - cicli che non terminano
  - indici di array fuori dai limiti
  - erronea allocazione/deallocazione
  - inconsistenza tra parametri formali e attuali
  - confronto di eguaglianza tra reali

Fault class	Inspection check
Data faults	Are all program variables initialised before their values are used? Have all constants been named? Should the lower bound of arrays be 0, 1, or something else? Should the upper bound of arrays be equal to the size of the array or Size -1? If character strings are used, is a delimiter explicitly assigned?
Control faults	For each conditional statement, is the condition correct? Is each loop certain to terminate? Are compound statements correctly bracketed? In case statements, are all possible cases accounted for?
Input/output faults	Are all input variables used? Are all output variables assigned a value before they are output?
Interface faults	Do all function and procedure calls have the correct number of parameters? Do formal and actual parameter types match? Are the parameters in the right order? If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	If a linked structure is modified, have all links been correctly reassigned? If dynamic storage is used, has space been allocated correctly? Is space explicitly de-allocated after it is no longer required?
Exception management faults	Have all possible error conditions been taken into account?

Walkthrough

- Obiettivo
  - Rivelare la presenza di difetti
  - Eseguire una lettura critica del codice
- Agenti
  - Gruppi misti ispettori/sviluppatori
- Strategia
  - Percorrere il codice simulandone l'esecuzione

Fasi del Walkthrough

- Fase 1: pianificazione
- Fase 2: lettura del codice
- Fase 3: discussione
- Fase 4: correzione dei difetti
- Documentazione

Inspection vs. Walkthrough

- Affinità
  - Controlli statici basati su desk-test
  - Contrapposizione fra programmatori e verificatori
  - Documentazione formale
- Differenze
  - Inspection è basata su errori presupposti (checklist)
  - Walkthrough è basato sull'esperienza dei verificatori
  - Walkthrough è più collaborativo
  - Inspection è più rapido

Dati sperimentali

- L'esperienza HP:
  - 50-75% degli errori scopribili con tecniche di revisione di progetto ("ispezioni")
  - alta complessità del flusso di controllo ("complessità ciclomatica" > 10) sintomo di possibile presenza di errori
  - se non si pone particolare cura, il test copre mediamente circa il 55% del codice

Dati sperimentali

- L'esperienza HP:
  - su 10 difetti scoperti durante il test, 1 si propaga nella manutenzione
  - eliminare difetti costa 4-10 volte in tempo nel caso di sistemi grossi e maturi rispetto a sistemi piccoli e in sviluppo
  - il costo di rimozione degli errori aumenta con il ritardo rispetto al quale gli errori sono introdotti