



# Verifica e validazione: analisi dinamica

IS

Anno accademico 2023/2024

Ingegneria del Software

Tullio Vardanega, [tullio.vardanega@unipd.it](mailto:tullio.vardanega@unipd.it)



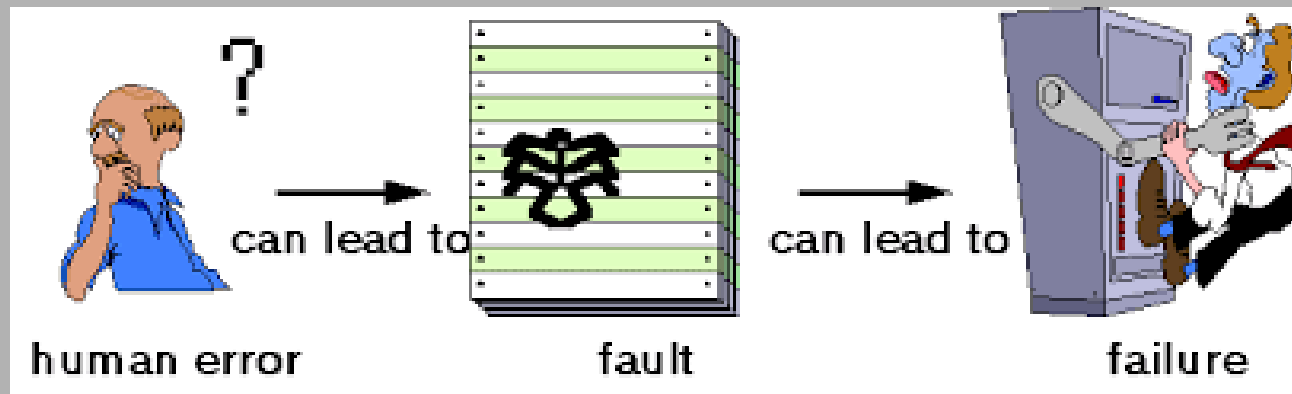
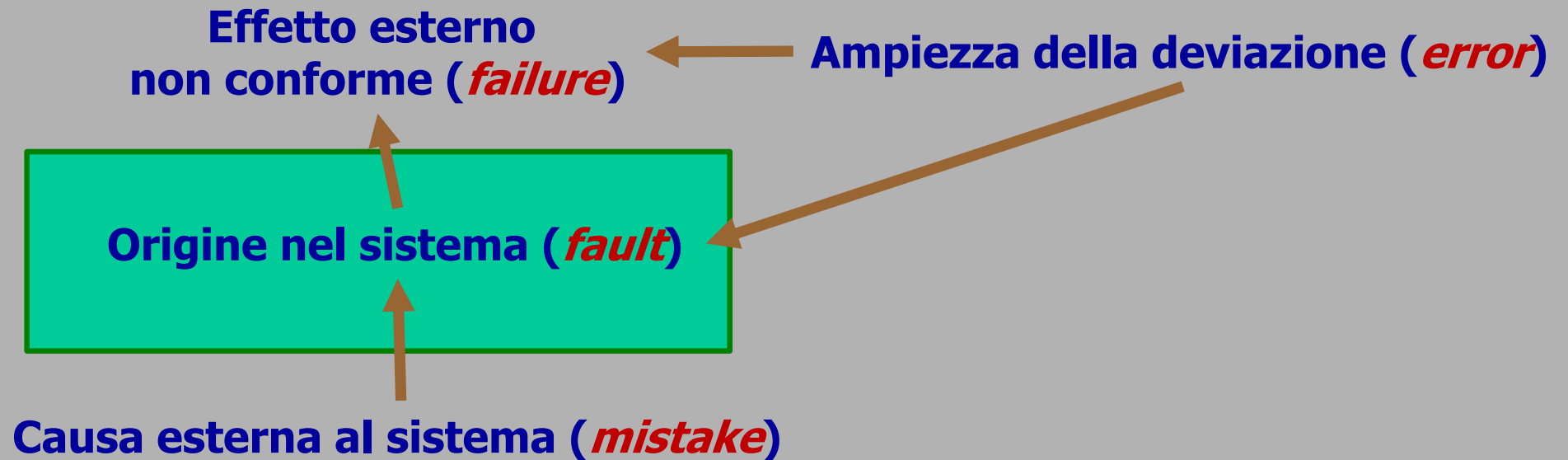
## Catena causale – 1/3

*The **fault tolerance** discipline distinguishes between an erroneous human action (a **mistake**), its manifestation (a hardware or software **fault**), the result of the fault (a **failure**), and the amount by which the result is incorrect (the **error**).*

IEEE Computer Society  
IEEE Standard Glossary of Software Engineering  
Terminology: IEEE Standard 610.12-1990. Number 610.12-  
1990 in IEEE Standard. 1990. ISBN 1-55937-067-X



## Catena causale – 2/3





## Catena causale – 3/3

- ❑ **Il processore è fisico: con l'uso può deperire fino a diventare *faulty***
- ❑ **Il SW non è materiale: ha natura immutabile**
  - Fa sempre e solo quello che il programma dice di fare
  - Non si guasta da sé e quindi non diventa *faulty*
- ❑ **Ma il SW può fare la cosa sbagliata**
  - Causando fallimenti (*failure*) di varia intensità (*error*)
- ❑ **Bisogna perciò assicurarsi che faccia la cosa giusta**
  - Questo è il compito della verifica del SW
  - L'analisi statica precede e integra l'analisi dinamica (i *test*)



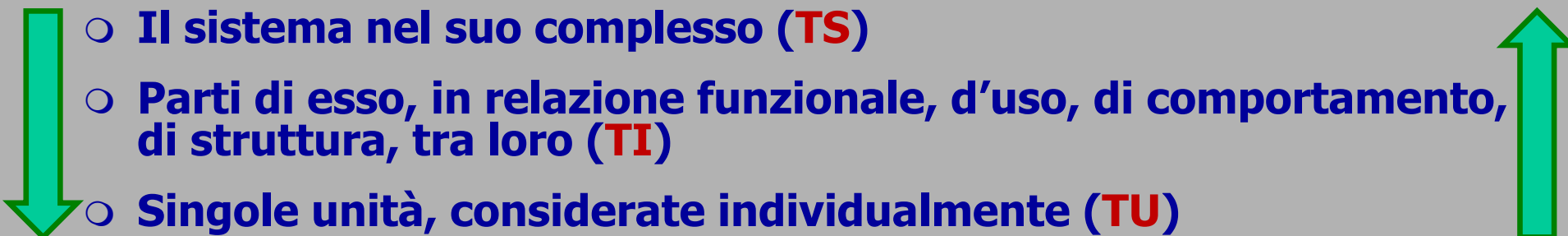
## Premesse generali – 1/2

- ❑ **L'analisi dinamica consiste nell'esecuzione di oggetti di prova**
  - Cioè di «programmi» che includono l'oggetto della prova
- ❑ **Ogni prova (*test*) è una esecuzione di un tale programma**
- ❑ **Le prove studiano il comportamento di singole parti di codice (oggetto di prova) su un insieme **finito** di «casi di prova»**
- ❑ **Il dominio di tutte le esecuzioni possibili è spesso **infinito**: per questo bisogna ridurlo senza rischiare omissioni significative**
- ❑ **Ciascun caso di prova specifica**
  - I valori di ingresso al programma
  - Lo stato iniziale atteso dell'esecuzione
  - L'effetto atteso (oracolo) che decide l'esito dell'esecuzione



## Premesse generali – 2/2

### □ L'oggetto della prova può essere

- 
- Il sistema nel suo complesso (**TS**)
  - Parti di esso, in relazione funzionale, d'uso, di comportamento, di struttura, tra loro (**TI**)
  - Singole unità, considerate individualmente (**TU**)

### □ L'obiettivo della prova deve essere

- Specificato in termini precisi e quantitativi
- Con esito decidibile in modo automatico

### □ Il PdQ specifica quali e quante prove effettuare

- Per raggiungere il massimo **grado di copertura** possibile

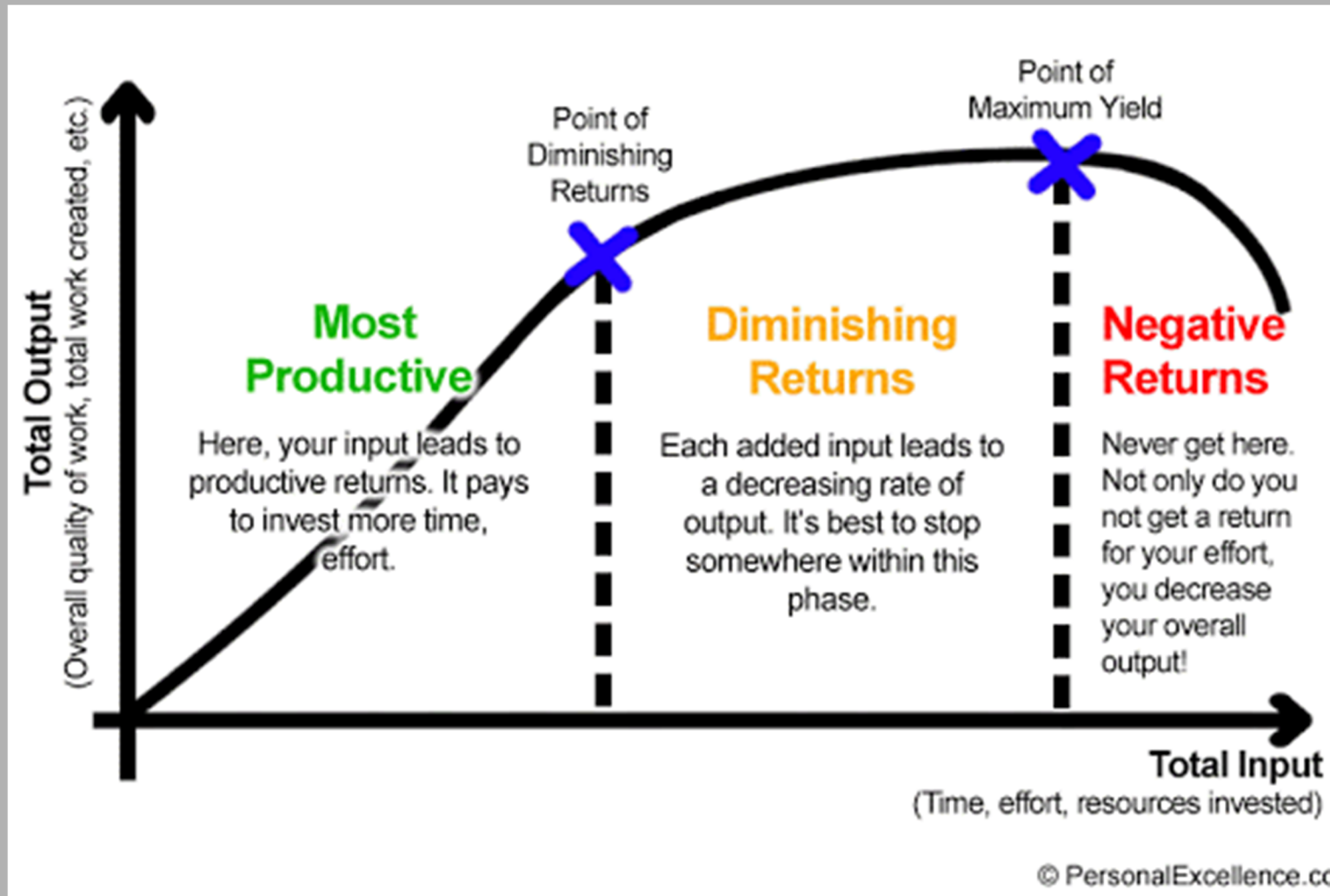


## Criteri guida – 1/4

- ❑ **La strategia di prova deve bilanciare costi e benefici**
  - Determinando la quantità minima di casi di prova sufficiente a garantire la qualità attesa
  - Attenzione alla **legge del rendimento decrescente**
- ❑ **Il PdP determina la quantità massima di risorse assegnate alla verifica (quindi anche alle prove)**
- ❑ **Il PdQ fissa gli obiettivi minimi di qualità da raggiungere nella verifica (quindi nelle prove)**
  - Prima si fissa la strategia di prova (cosa, come, quanto)
  - Poi la si correla con il piano delle attività



# Legge del rendimento decrescente







## Criteri guida – 2/4

- ❑ Il *test* è parte essenziale del processo di verifica
- ❑ Produce una misura della qualità del prodotto
  - La qualità aumenta (anche) con la rimozione di difetti
- ❑ Le attività di *test* devono iniziare il prima possibile
  - Al vertice basso della «V»
  - Assistite da analisi statica, che non richiede esecuzione
- ❑ Le esigenze di verifica devono essere assecondate dalla progettazione e dalla codifica
  - Progettare, realizzare, eseguire i *test* è molto costoso
  - Conviene renderlo più facile e produttivo possibile



## Criteri guida – 3/4

- ❑ Fare *test* significa eseguire programmi con l'intento di trovarvi difetti  
(G.J. Myers, *The Art of Software Testing*, Wiley, 2011)
- ❑ Fare prove (*test*) è costoso
  - Tanto SW → tante prove
  - SW organizzato male o «scritto male» ostacola lo sviluppo e l'esecuzione delle prova
- ❑ Progettiamo «bene», per dare ai moduli SW compiti chiari, specifici e circoscritti
- ❑ Scriviamo SW semplice: la complessità è nemica della provabilità



## Criteri guida – 4/4

- ❑ **Malfunzionamenti rilevati nei *test* rivelano la presenza di difetti**
- ❑ ***Test* eseguiti senza errori non provano l'assenza di difetti (Dijkstra, 1969)**
- ❑ **Le prove devono essere riproducibili per accertare**
  - Buon esito di correzione dei malfunzionamenti osservati
  - Funzionamento non perturbato dall'avanzare della codifica
- ❑ **Le prove sono costose**
  - Richiedono molte risorse (tempo, persone, infrastrutture)
  - Richiedono cicli di analisi, progettazione, codifica, correzione



# Principi del *testing software*

Per approfondire

## □ Secondo Bertrand Meyer

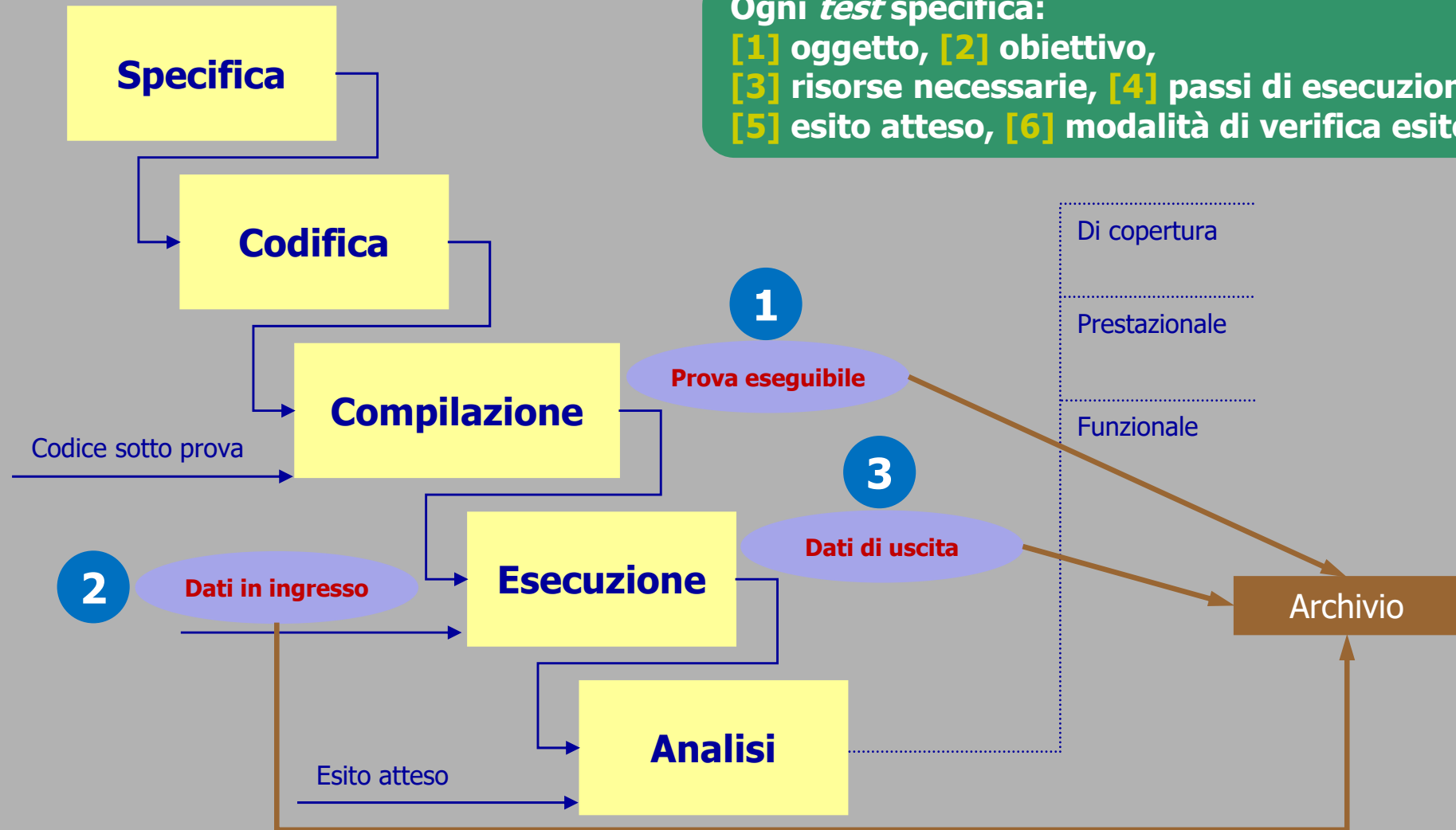
- *To test a program is to try to make it fail*
- *Tests are no substitutes for specifications*
- *Any failed execution must yield a test case, permanently included in the project's test suite*
- *Oracles should be part of the program text, as **contracts***
- *Any testing strategy should include a reproducible testing process and be evaluated objectively with explicit criteria*
- *A testing strategy's most important quality is the number of faults it uncovers as a function of time*



# Attività di prova

Ogni *test* specifica:

[1] oggetto, [2] obiettivo,  
[3] risorse necessarie, [4] passi di esecuzione,  
[5] esito atteso, [6] modalità di verifica esito





# Gli elementi di una prova – 1/2

## □ Caso di prova (*test case*)

- Tupla {oggetto di prova, ingresso richiesto, uscita attesa, ambiente di esecuzione e stato iniziale, passi di esecuzione}

## □ Batteria di prove (*test suite*)

- Insieme di casi di prova

## □ Procedura di prova

- Procedimento automatizzabile per eseguire prove e registrarne, analizzarne e valutarne i risultati

## □ Prova

- Esecuzione (automatica) di procedura di prova



## Gli elementi di una prova – 2/2

### □ **L'oracolo**

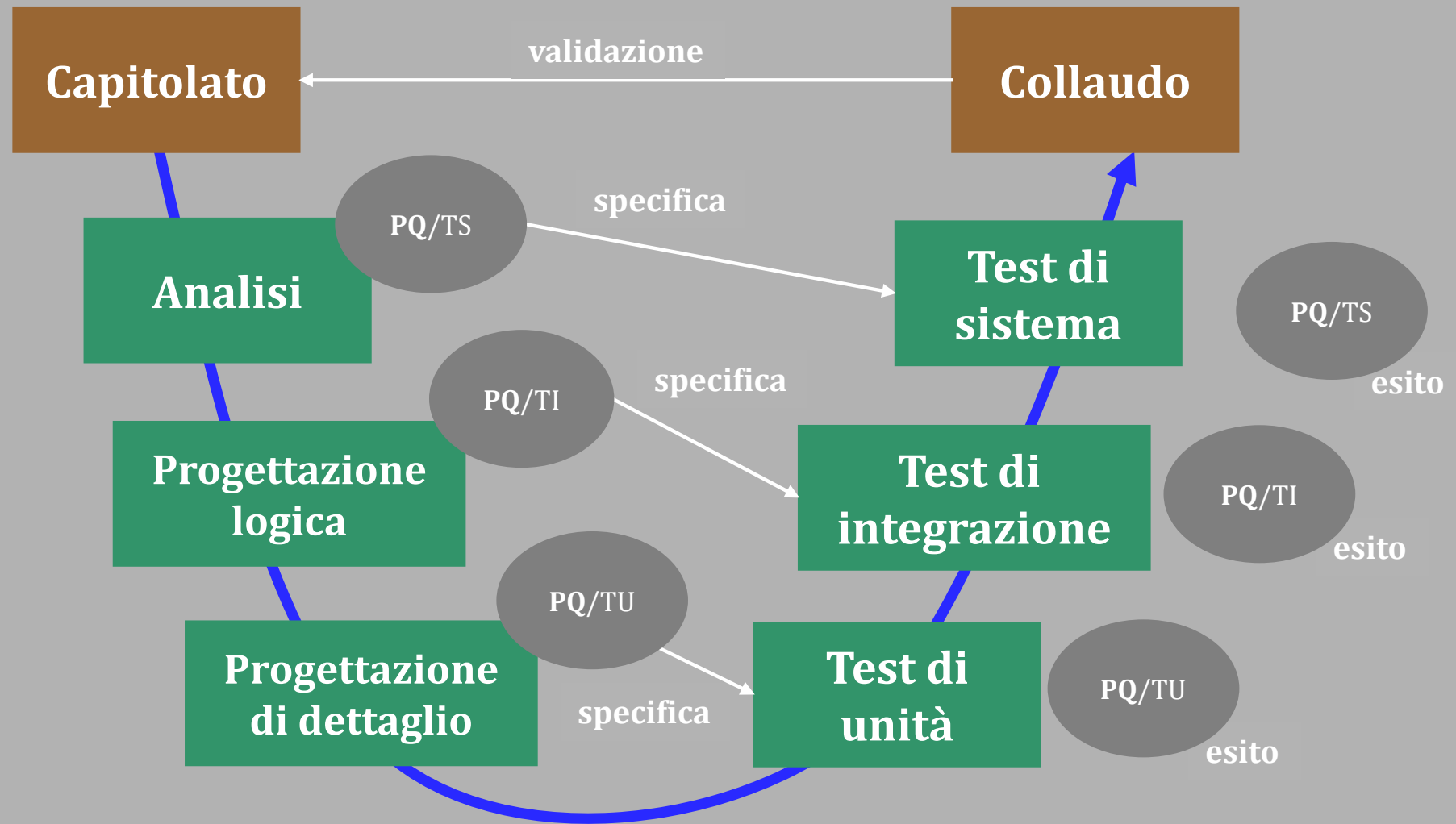
- Metodo per determinare a priori i risultati attesi e per convalidare i risultati ottenuti nella prova
- Applicato da agenti automatici, per velocizzare la convalida e renderla oggettiva

### □ **Come produrre oracoli?**

- Sulla base delle specifiche funzionali
- Entro prove semplici (facilmente decidibili)
- Tramite l'uso di componenti terze e fidate



# Esecuzione delle attività di prova







## *Test di unità – 1/2*

- ❑ **L'unità SW è composta da uno o più moduli**
  - **Modulo = componente elementare di architettura di dettaglio**
- ❑ **I moduli sono fissati nella progettazione di dettaglio**
- ❑ **Le unità possono dipendere dalla codifica**
- ❑ **Il piano di TU nasce nel vertice basso della «V»**
- ❑ **Il TU completa quando ha verificato tutte le unità**
- ❑ **La maggior parte dei difetti di un prodotto SW viene rilevata nei TU**
- ❑ **I TU sono di due tipi: funzionale o strutturale**

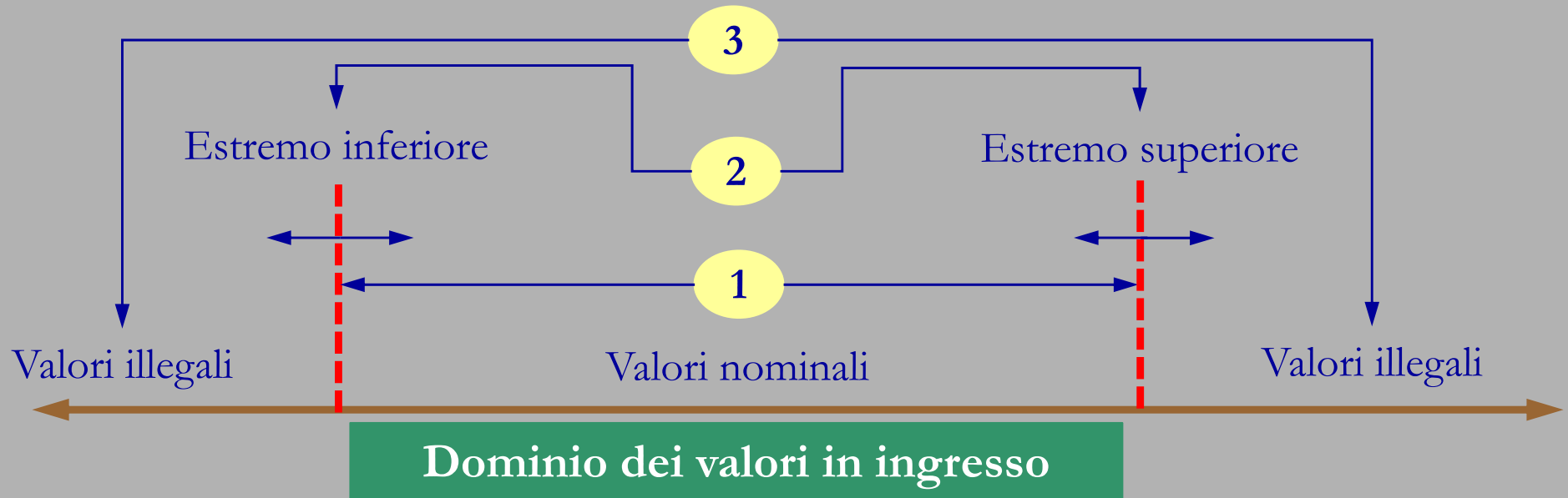


## Test funzionale (*black-box*)

- ❑ Fanno riferimento solo alla specifica dell'unità
- ❑ Non valutano la logica interna dell'unità
- ❑ Utilizzano dati di ingresso cui corrispondono specifici esiti
- ❑ Dati di ingresso che producano lo *stesso* comportamento funzionale (*classi di equivalenza*) formano un *singolo* caso di prova
- ❑ I TU-F contribuiscono al *requirements coverage*
  - % di requisiti funzionali soddisfatti dal prodotto



# Classi di equivalenza



## 3 classi di equivalenza

- Valori nominali interni al dominio **1**
- Valori legali di limite **2**
- Valori illegali **3**



## Test strutturale (*white-box*)

- ❑ Verificano la logica interna del codice dell'unità cercando massima *structural coverage*
- ❑ Ogni singolo caso di prova deve attivare un singolo *cammino di esecuzione* all'interno dell'unità
  - Generando le condizioni logiche che causano la scelta di quel cammino
- ❑ Ogni caso di prova è costituito dall'insieme di dati di ingresso e di configurazione di ambiente che produce uno specifico cammino d'esecuzione



## Dimensioni della *structural coverage*

- ❑ Si ha ***Statement Coverage*** al 100%
  - Quando l'insieme di *test* effettuati sull'unità esegue almeno una volta tutti i comandi (*statement*) dell'unità, con esito corretto
- ❑ Si ha ***Branch Coverage*** al 100%
  - Quando ciascun ramo (*then/else*) del flusso di controllo dell'unità viene attraversato almeno una volta da un *test*, con esito corretto
- ❑ Si ha ***Decision/Condition Coverage*** al 100%
  - Quando ogni condizione della decisione (*branch*) assume almeno una volta entrambi i valori di verità in un *test* dedicato
  - Metrica più precisa della *branch coverage*
  - Necessaria in presenza di espressioni di decisione complesse



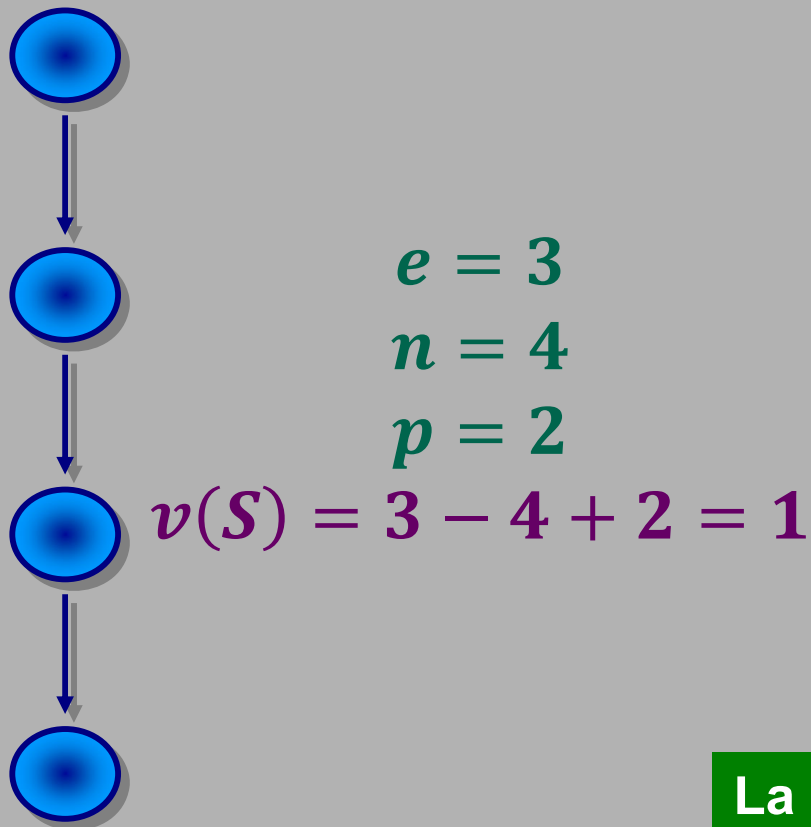
## *Branch coverage*

- Il numero di percorsi linearmente indipendenti in una esecuzione con singolo ingresso e singola uscita (unità) è detto **complessità ciclomatica**, CC
  - Cresce con *branch*, salti, e iterazioni
- La CC del grafo  $G$  che descrive i flussi d'esecuzione all'interno dell'unità, è  $v(G) = e - n + p$ 
  - $e$  numero degli archi in  $G$  (flusso tra comandi)
  - $n$  numero dei nodi in  $G$  (espressioni o comandi)
  - $p$  numero delle componenti connesse da ogni arco  
(l'esecuzione sequenziale ha  $p = 2$ , avendo 1 predecessore e 1 successore per ogni arco)

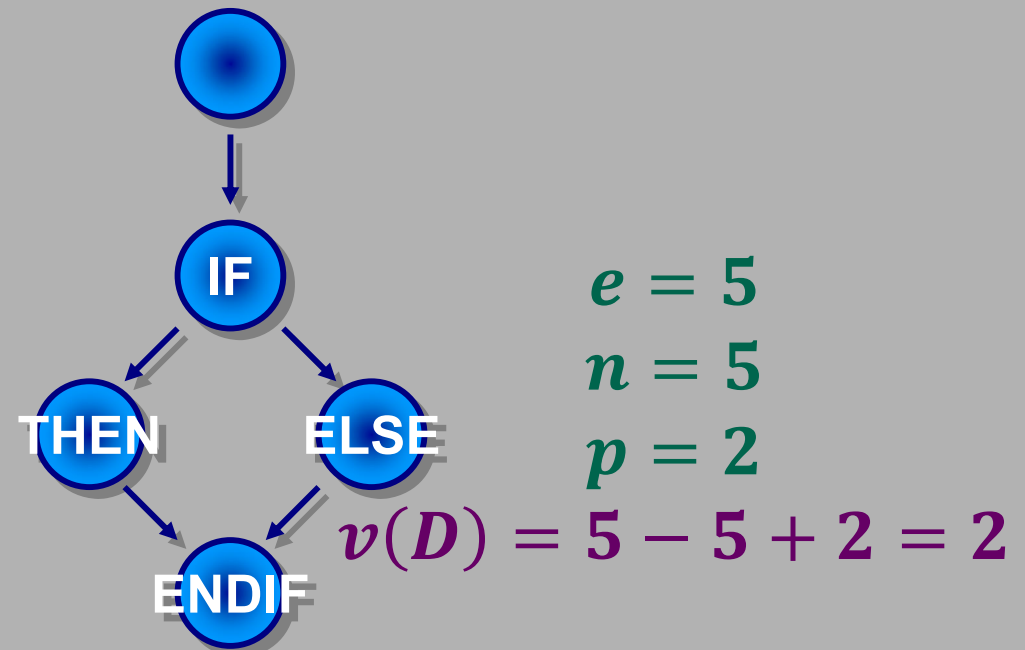


# Complessità ciclomatica – 1/2

## □ Sequenza S



## □ Decisione D



La presenza di decisioni aumenta la CC



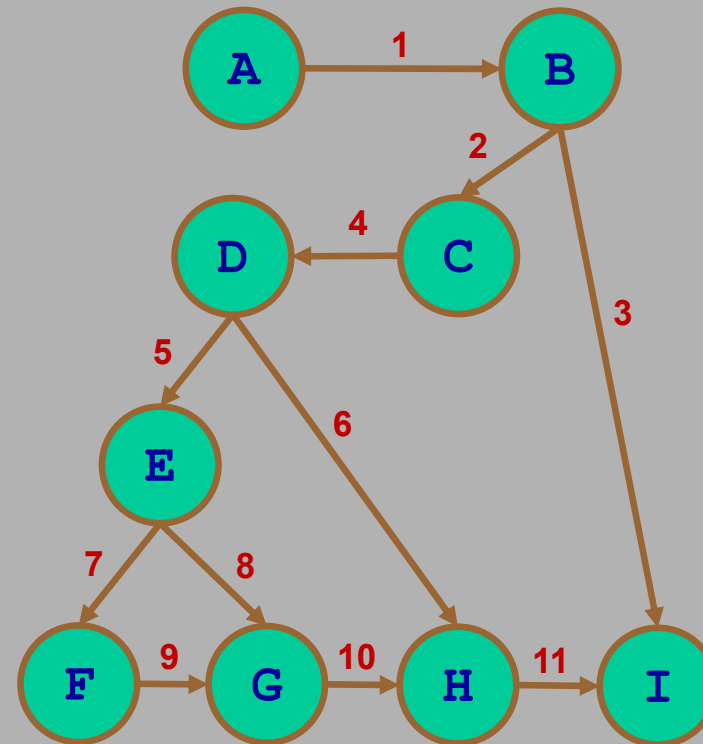
## Complessità ciclomatica – 2/2

```
A: I := 0; N := 4;  
B: if (I < N-1) then  
C:   J := I+1;  
D:   if (J < N) then  
E:     if A[I] < A[J] then  
F:       I := I+1;  
G:     end if;  
H:   end if;  
I: end if;
```

**P**

$$v(P) = 11 - 9 + 2 = 4$$

1. Archi A-B(F)-I
2. Archi A-B(T)-C-D(F)-H-I
3. Archi A-B(T)-C-D(T)-E(F)-G-H-I
4. Archi A-B(T)-C-D(T)-E(T)-F-G-H-I







# Condition-and-decision coverage

- ❑ **Decisione** (*branch*) è una espressione composta da più condizioni
  - Le prove di ogni singola decisione devono produrre almeno un T e un F
- ❑ **Condizione** è una espressione booleana semplice
  - Le prove di ogni singola condizione devono produrre almeno un T e un F
- ❑ Il *branch coverage* effettivo (***condition-and-decision coverage***) copre singolarmente tutte le condizioni della decisione
- ❑ Quanto più complessa la decisione, tanto più oneroso raggiungere un alto grado di *branch coverage* effettivo
- ❑ La tecnica ***Modified Condition/Decision Coverage*** (MCDC) massimizza il *branch coverage* effettivo con minor numero di prove



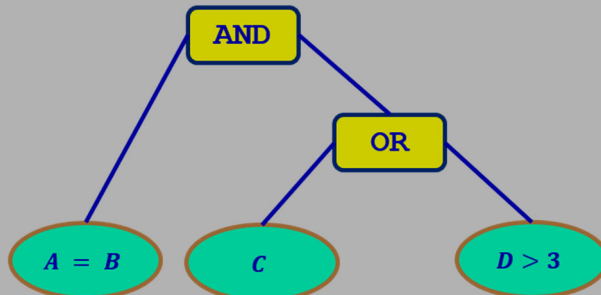
# Approfondiamo MCDC ...

- ❑ Con MCDC, questa decisione, richiede 4 prove

Decisione

```
if (A=B and (C or D>3)) then ...
```

Condizione



A, B, D non sono bool

Prova	Condizione			Decisione
	A=B	C	D>3	
1	•	F	F	F
2	T	T	•	T
3	T	•	T	T
4	F	•	•	F

- ❑ La complessità ciclomatica della decisione sarebbe 2 : due prove
- ❑ La tabella di verità ci mostra però che per raggiungere gli obiettivi di copertura MCDC servono 4 (quattro) prove
- ❑ La prova 1 copre il caso F per le condizioni 2 e 3, per entrambe producendo F per la decisione
- ❑ La prova 3 copre il caso T per le condizioni 1 e 3, per entrambe producendo T per la decisione



# Test di integrazione

- ❑ **Si applica alle componenti individuate nel *design* architetturale**
  - La loro integrazione totale costituisce il sistema completo
- ❑ **Rileva difetti di progettazione architetturale o bassa qualità di TU**
  - I dati scambiati attraverso ciascuna interfaccia concordano con la specifica?
  - Tutti i flussi di controllo specificati sono stati verificati corretti?
- ❑ **Assembla incrementalmente, a ogni passo aumentando il valore funzionale disponibile**
  - Integrando componenti nuove in insiemi già verificati, i difetti rilevati da TI su tale passo sono più probabilmente da attribuirsi all'ultima aggiunta
- ❑ **Assicura che ogni passo di integrazione sia reversibile**
  - Potendo sempre retrocedere a un precedente stato sicuro (*baseline*)



# Strategie di integrazione

## □ Integrazione incrementale di tipo *bottom-up*

- Si sviluppano e si integrano prima le componenti con minori dipendenze d'uso e maggiore utilità interna
  - Quelle che sono molto chiamate/attivate ma chiamano/attivano poco o nulla
  - Quelle più interne al sistema, meno visibili a livello utente
- Questa strategia richiede pochi *stub* ma ritarda la messa a disposizione di funzionalità visibile all'utente

## □ Integrazione incrementale di tipo *top-down*

- Si sviluppano e si integrano prima le componenti con maggiori dipendenze d'uso e quindi maggiore valore aggiunto esterno
  - Quelle che chiamano/attivano più di quanto siano chiamate/attivate
- Questa strategia comporta l'uso di molti *stub* ma integra prima le funzionalità di più alto livello, più visibili all'utente



## Test di sistema

- ❑ Verifica come l'esecuzione del sistema soddisfi i requisiti SW (quelli della AdR)
- ❑ Completa la misura di **requirements coverage** valutata a partire dai TU funzionali
  - B. Meyer raccomanda che i TS includano tutti i casi di prova (TU, TI) che siano falliti almeno una volta
- ❑ È funzionale (*black-box*)
  - Non deve richiedere conoscenza della logica interna del SW
  - I requisiti SW fissano l'aspettativa e non l'implementazione
- ❑ Inizia al completamento del TI e precede il collaudo



## Altri tipi di *test*

### □ **Test di regressione**

- Accerta che correzioni o estensioni effettuate su specifiche unità non danneggino il resto del sistema
- Consiste nella ripetizione selettiva di TU, TI e TS
  - Tutti i *test* necessari ad accertare che la modifica di una parte P di S non causi errori in P, in S, o in ogni altra parte del sistema in relazione con S
- Attua i processi ***Problem Resolution*** e ***Change Management***
  - Il primo valuta la necessità di modifiche (correttivo o adattative) e le approva
  - Il secondo gestisce la buona realizzazione delle modifiche approvate

### □ **Test di accettazione (collaudo)**

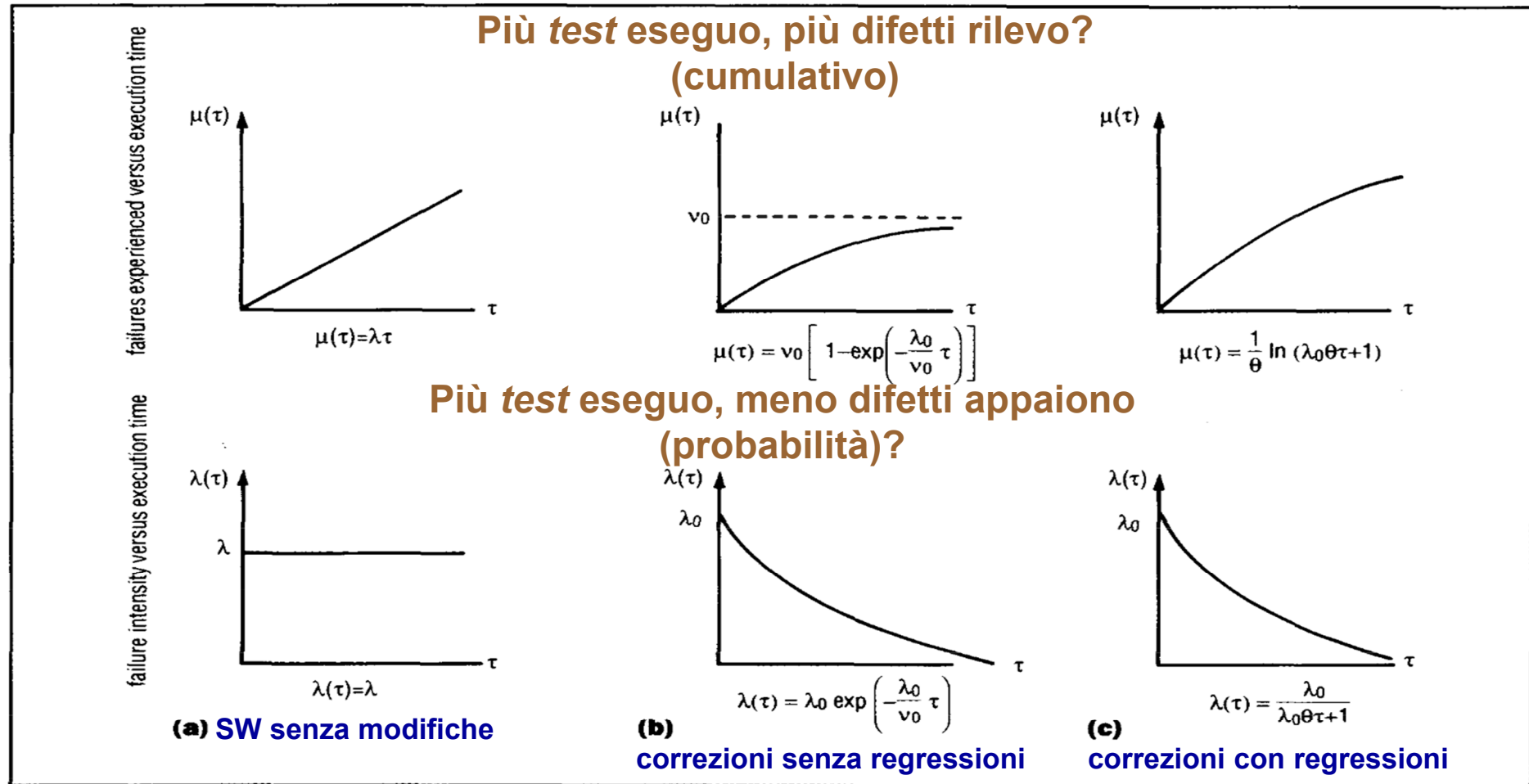
- Accerta il soddisfacimento dei requisiti utente (quelli del capitolato) alla presenza del committente



## Misure di copertura (*coverage*)

- ❑ **Dicono quanto le prove hanno «esplorato» il prodotto**
  - Copertura funzionale rispetto ai requisiti del prodotto
  - Copertura strutturale rispetto alla sua logica interna
- ❑ **Quantificano la bontà della campagna di *test***
  - Raggiungere il 100% di copertura complessiva può non essere possibile per ragioni di tempo/costo, complessità
  - Più alta la copertura, più basso il rischio di difetti residui
  - A ogni modo, raggiungere il 100% di copertura non garantisce assenza di difetti (ce lo dice Dijkstra)
- ❑ **Gli obiettivi di copertura sono specificati nel PdQ**

# Quando conviene smettere i test?

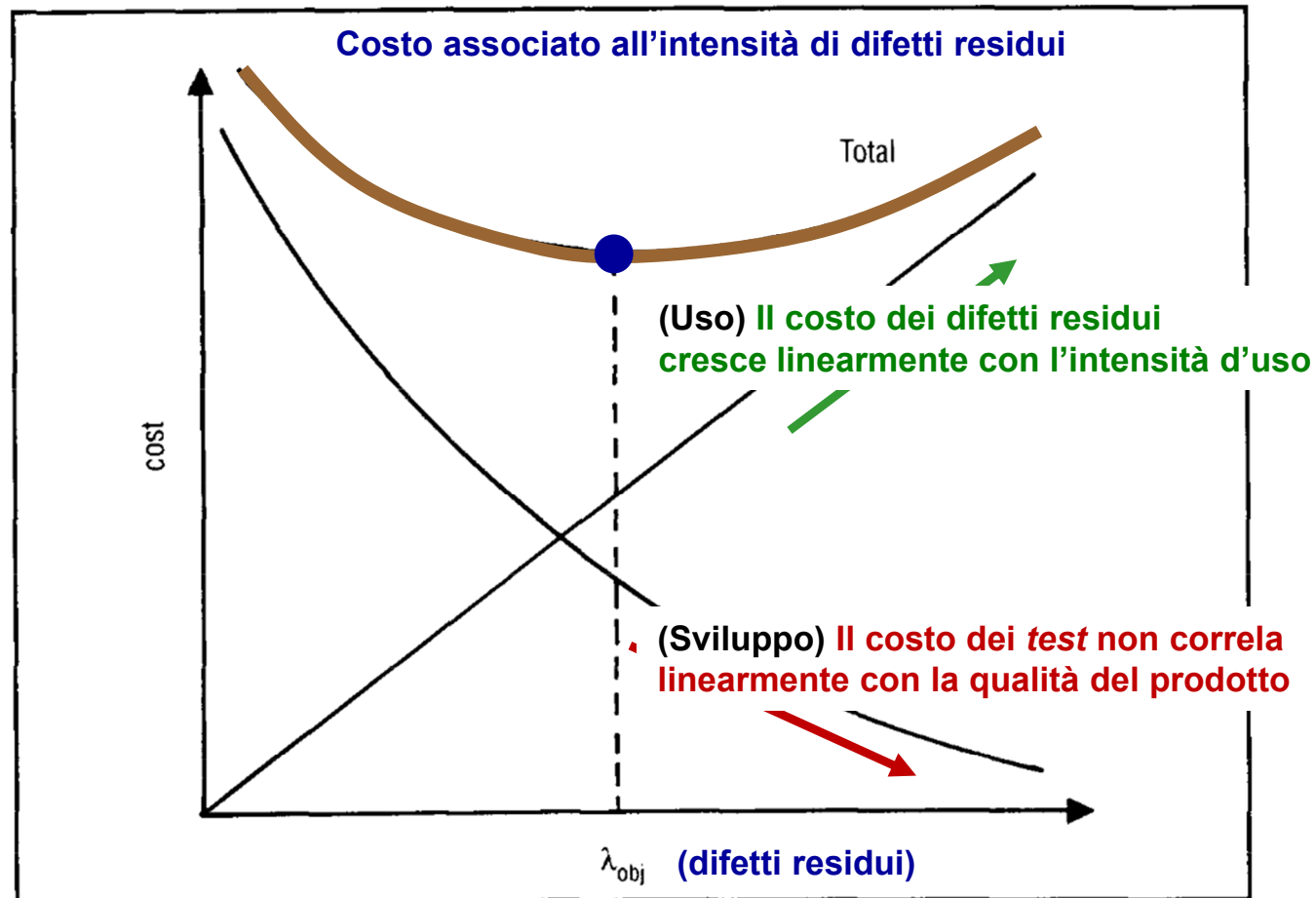


**Figure 1.** Three useful software-reliability models: **(a)** static, **(b)** basic, and **(c)** logarithmic Poisson. These are shown comparing both failures experienced versus execution time and failure intensity versus execution time.





# La risposta di Musa & Ackerman [1]



Bisogna decidere quali densità di difetti residui sia accettabile, che minimizzi il costo d'Uso entro un costo di Sviluppo sostenibile



## Un altro punto di vista [3]

- ❑ **Gli errori gravi spesso sono meno costosi di quelli più lievi**
  - Perché quelli gravi sono trattati con più urgenza
  - Quelli meno, spesso in modo più trascurato (poco attento o tardivo)
- ❑ **Correggere gli errori è molto costoso, quando farlo comporta modifiche architettureali**
- ❑ **Il costo degli errori residui cresce esponenzialmente con l'avanzare del progetto**
- ❑ **Il numero di errori rilevati cresce linearmente con la durata del progetto**
- ❑ **Usare bene *Continuous Integration* focalizza meglio le attività di sviluppo e amplia l'intensità di *test***



# Torna in scena il *technical debt* ...



□ Il TD può essere consapevole o meno

□ L'atteggiamento rispetto al TD ha quattro varianti

- PD: consapevolezza
- RD: superbia
- PI: umiltà
- RI: incompetenza



## Bibliografia

- 1) J.D. Musa, A.F. Ackerman, *Quantifying software validation: when to stop testing?*, IEEE Software, maggio 1989
  - <http://selab.netlab.uky.edu/homepage/musa-quantify-sw-test.pdf>
- 2) B. Meyer, *Seven Principles of Software Testing*, IEEE Computer, agosto 2008 (cf. per approfondire)
- 3) J.C. Westland, *The cost of errors in software development: evidence from industry*, Journal of Systems and Software 62(1):1-9, 2002