Verifica e validazione: analisi dinamica

IS

Anno accademico 2022/2023
Ingegneria del Software

Tullio Vardanega, tullio.vardanega@unipd.it

Catena causale – 1/2

*The fault tolerance discipline distinguishes between a 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

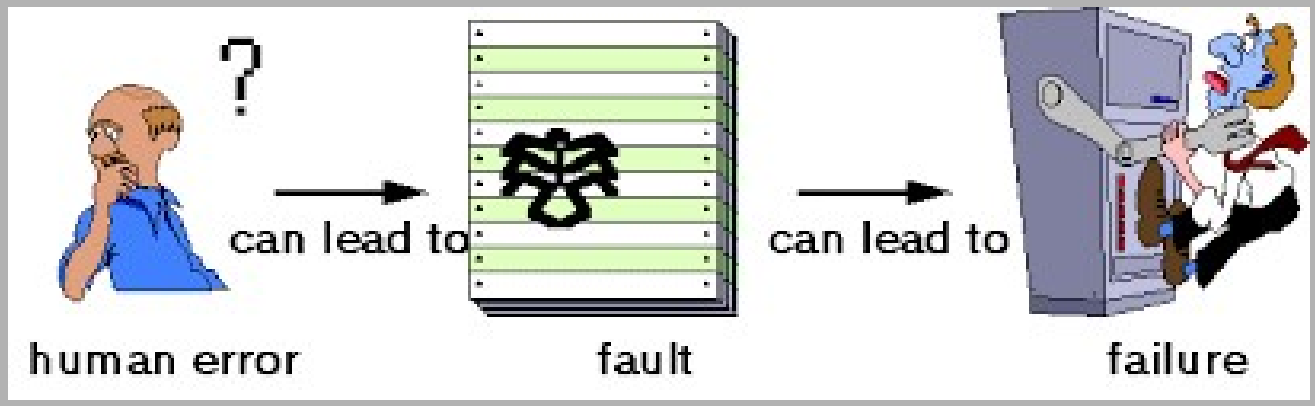
Catena causale – 2/2

Effetto non conforme esterno al sistema (*failure*)

Ampiezza della deviazione (*error*)

Origine nel sistema (*fault*)

Causa esterna al sistema (*mistake*)



Premesse generali – 1/2

- L'analisi dinamica consiste nell'esecuzione di vari oggetti di prova
  - Dunque 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 su un insieme finito di casi
  - Il dominio di tutte le esecuzioni possibili è spesso infinito
  - Bisogna ridurlo sensibilmente senza correre rischi di omissione!
- Ciascun caso di prova specifica
  - I valori di ingresso
  - Lo stato iniziale del sistema
  - L'effetto atteso (oracolo) che permette di decidere l'esito dell'esecuzione

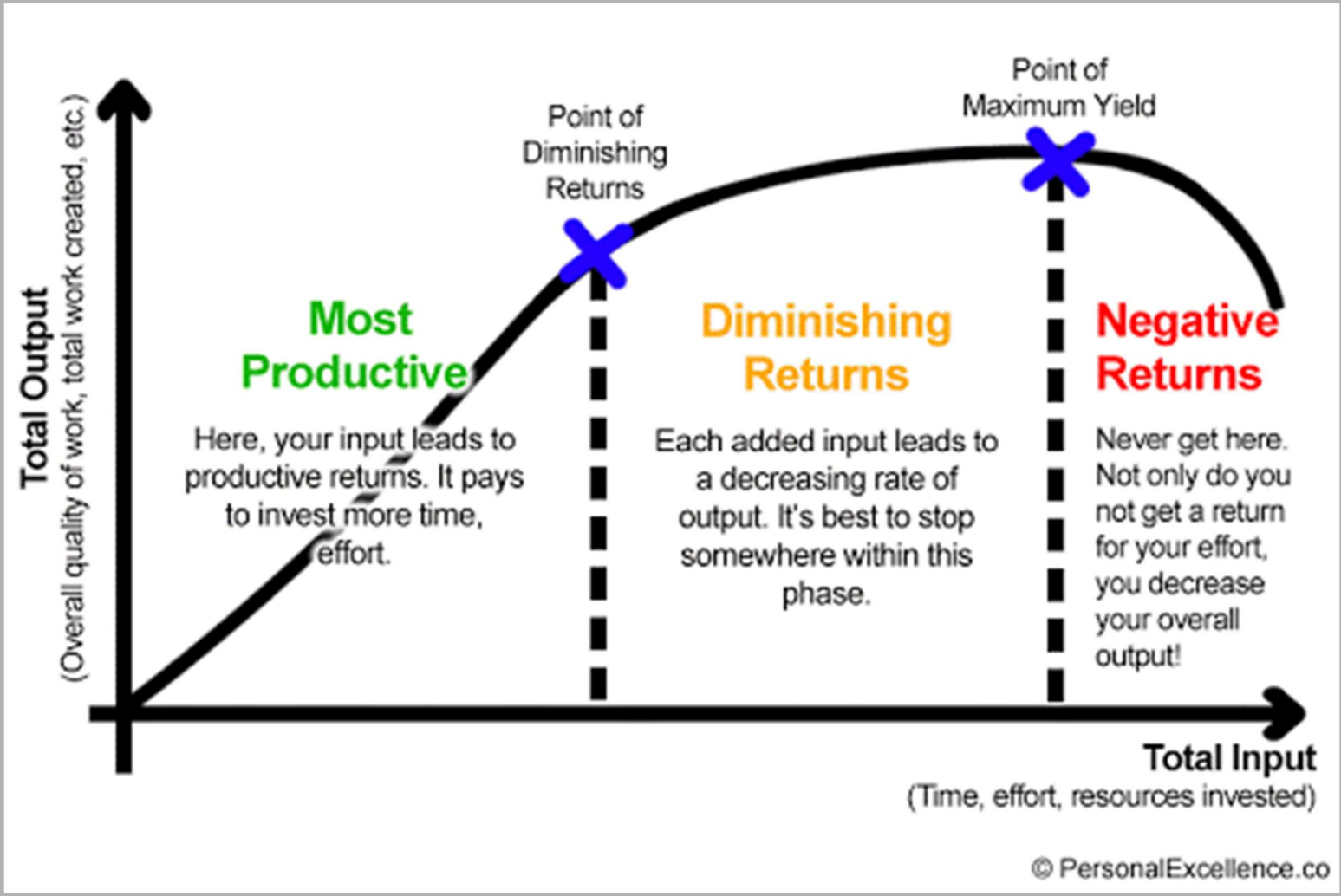
Premesse generali – 2/2

- **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**

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
  - Facendo attenzione alla legge del rendimento decrescente
- Il PdP determina la quantità massima di risorse assegnate alla verifica (quindi anche alle prove)
  - Disponibilità tardive o insufficienti danneggiano il progetto
- Il PdQ fissa gli obiettivi minimi di qualità da raggiungere nella verifica (quindi nelle prove)
  - Prima si fissa la strategia di prova (come, cosa, con quale intensità)
  - 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»
- Le esigenze di verifica devono essere assecondate dalla progettazione e dalla codifica
  - Progettare, realizzare, ed eseguire i *test* è 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

- La "provabilità" del SW va assicurata a monte dello sviluppo, non a valle della codifica
  - Progettazione architetturale e di dettaglio raffinate per assicurare provabilità
  - La complessità è nemica della provabilità

Criteri guida – 4/4

- Malfunzionamenti rilevati nelle prove indicano la presenza di guasti
  - Ma il buon esito delle prove non può provarne l'assenza!
- 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)
  - Vanno governate con efficienza ed efficacia
  - Richiedono cicli di analisi, progettazione, codifica, correzione

Limiti e problemi

- **Tesi di Dijkstra (1969)**
  - **Il *test* di un programma può rilevare la presenza di malfunzionamenti, ma non può dimostrarne l'assenza**
- **Teorema di Howden (1975)**
  - **Non esiste un algoritmo che, dato un programma P, generi per esso un *test* finito ideale (definito da criteri affidabili e validi)**
- **Teorema di Weyuker (1979)**
  - **Dato un programma P, sono indecidibili i seguenti problemi**
    - ∃ ingresso che causi l'esecuzione di un dato comando di P?
    - ∃ ingresso che causi l'esecuzione di una data condizione di P?
    - ∃ ingresso che causi l'esecuzione di ogni comando / condizione / cammino di P?

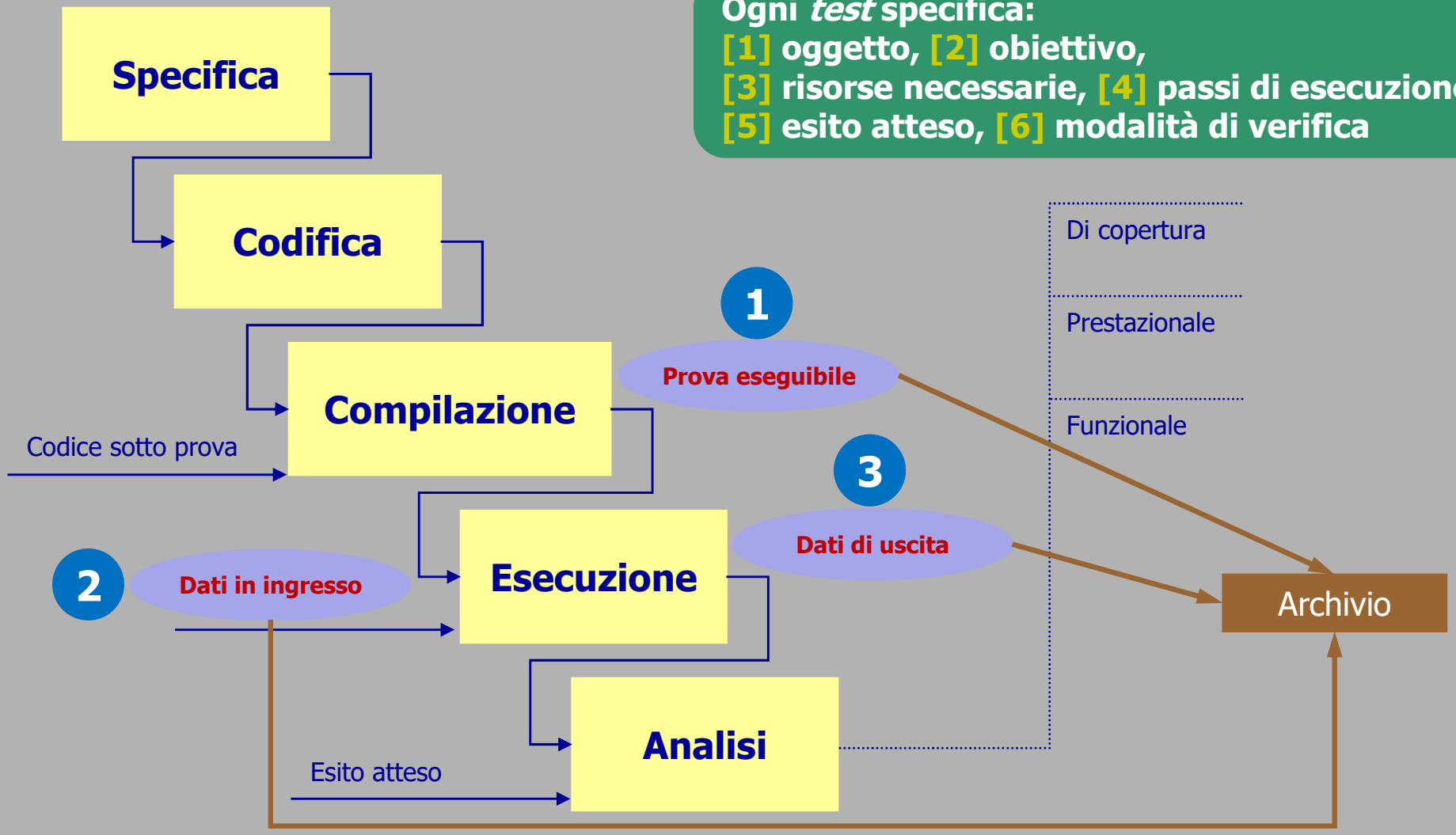
Principi del *testing software*

Per approfondire T11

- **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



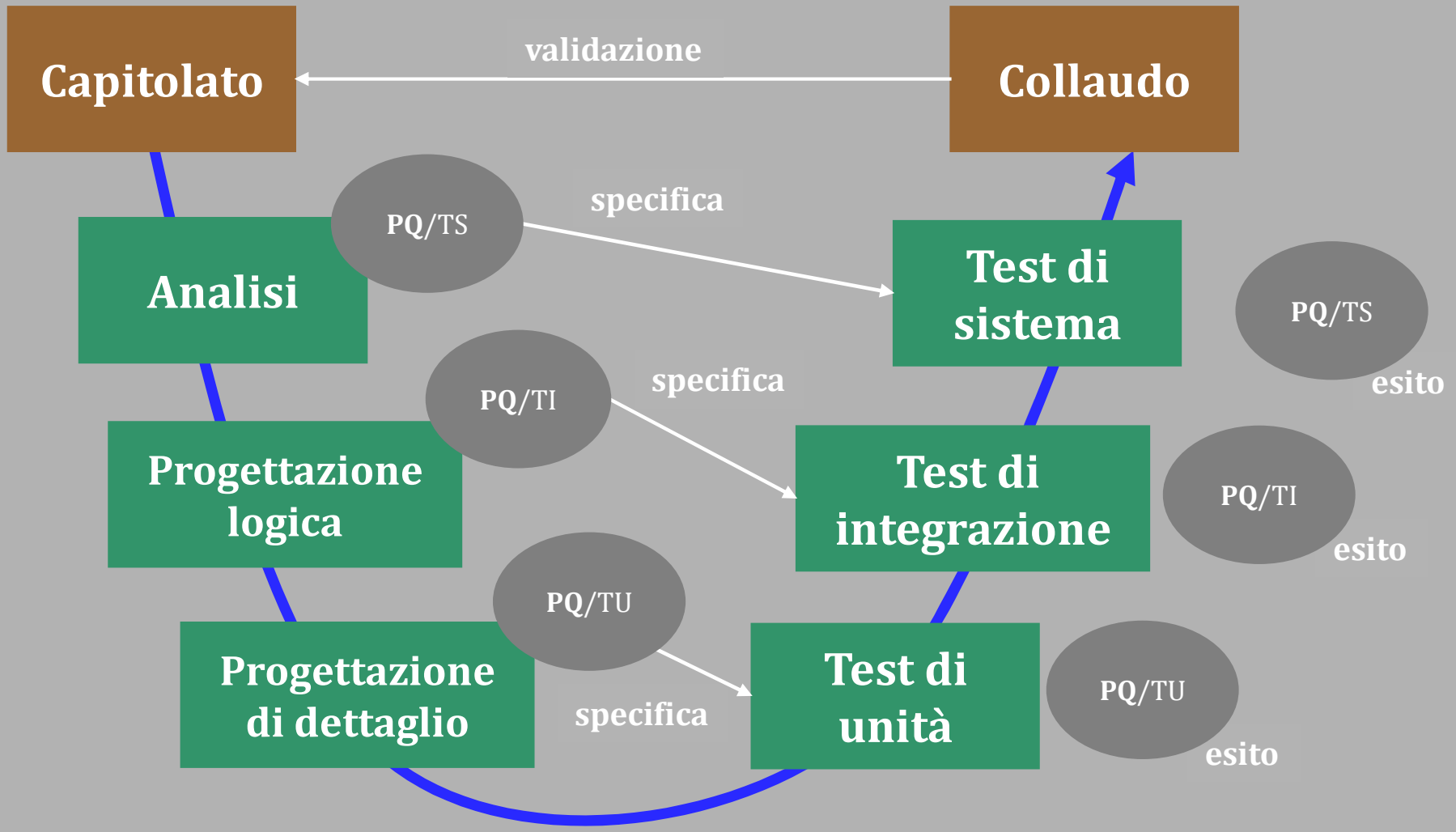
Gli elementi di una prova – 1/2

- **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



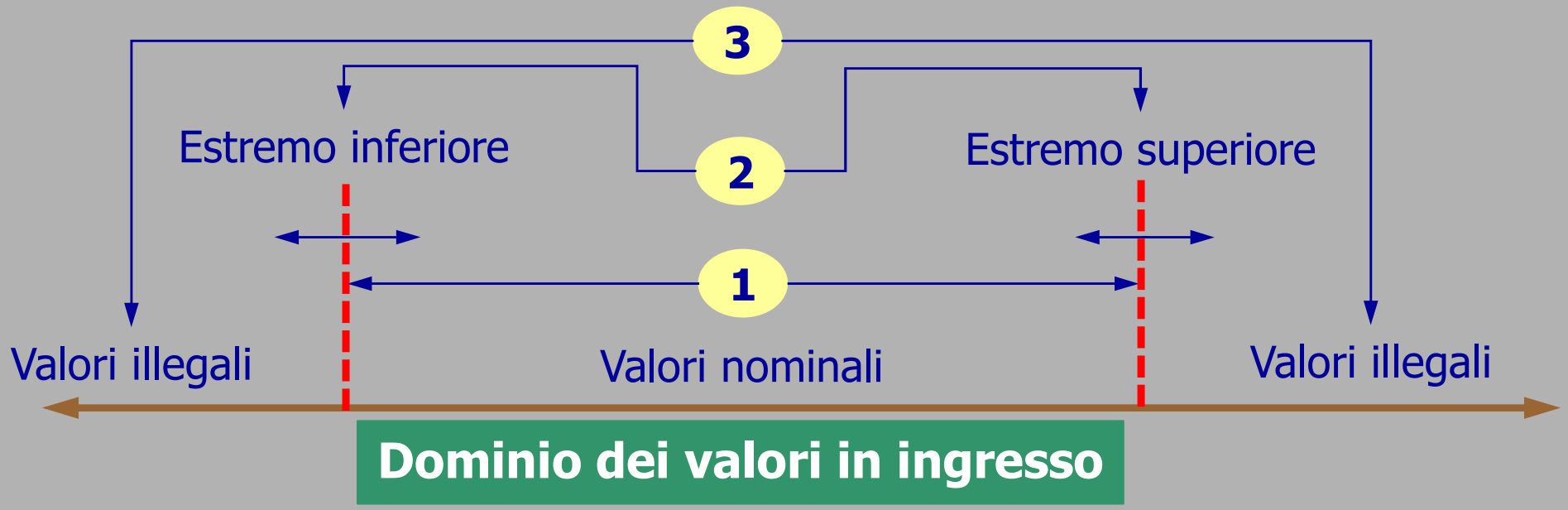
Test di unità – 1/2

- L'unità SW è composta da uno o più moduli
  - Modulo = componente elementare di architettura di dettaglio
- Unità e moduli sono decisi nella progettazione di dettaglio
  - È lì che nasce il piano di TU
- Il TU completa quando ha verificato tutte le unità
- Il TU è massimamente produttivo per capacità di rilevazione di difetti SW

Test di unità – 2/2

- ***Test* funzionale (*black-box*)**
  - **Fa riferimento solo alla specifica dell'unità**
  - **Utilizza dati di ingresso che provochino specifici esiti**
  - **Dati di ingresso che producano lo stesso comportamento funzionale formano un singolo caso di prova**
    - Tali dati formano **classi di equivalenza**, da cui si estraggono campioni
  - **Contribuisce cumulativamente al *requirements coverage***
    - Misura di quanti requisiti funzionali siano soddisfatti dal codice prodotto
  - **Non valuta la logica interna dell'unità**
    - Che certamente nasconde difetti che vanno scovati con altro tipo di *test*

Classi di equivalenza



3 classi di equivalenza

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

Test di unità – 3/3

- ***Test* strutturale (*white-box*)**
  - **Verifica la logica interna del codice dell'unità cercando massima *structural coverage***
    - Che ha più dimensioni ed è complementare alla *requirements coverage*
  - **Ogni singolo caso di prova deve attivare un singolo cammino di esecuzione all'interno dell'unità**
    - Creando 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*

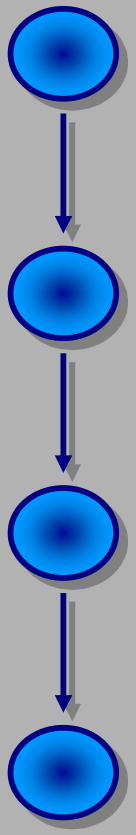
- Si ha ***Statement Coverage*** al 100%
  - Quando l'insieme di *test* effettuati sull'unità esegue almeno una volta tutti i comandi (*statement*) dell'unità, con esito corretto
- Si ha ***Branch Coverage*** al 100%
  - Quando ciascun ramo (`then/else`) del flusso di controllo dell'unità viene attraversato 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
  - Inizialmente 1, incrementata da *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)

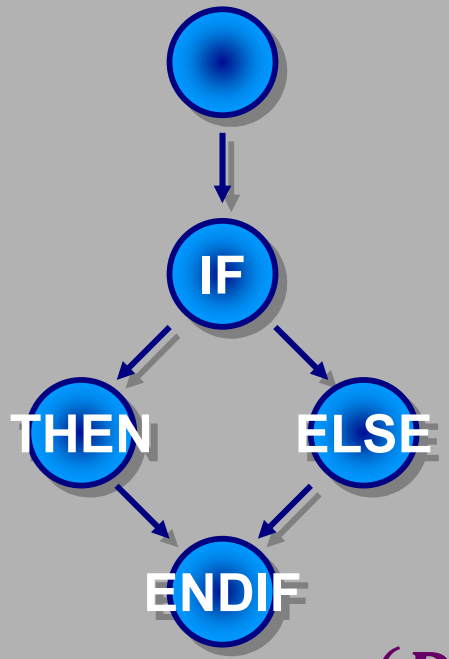
Complessità ciclomatica – 1/2

❑ Sequenza S



$$e = 3$$
$$n = 4$$
$$p = 2$$
$$v(S) = 3 - 4 + 2 = 1$$

❑ Decisione D



$$e = 5$$
$$n = 5$$
$$p = 2$$
$$v(D) = 5 - 5 + 2 = 2$$

La presenza di decisioni aumenta la CC

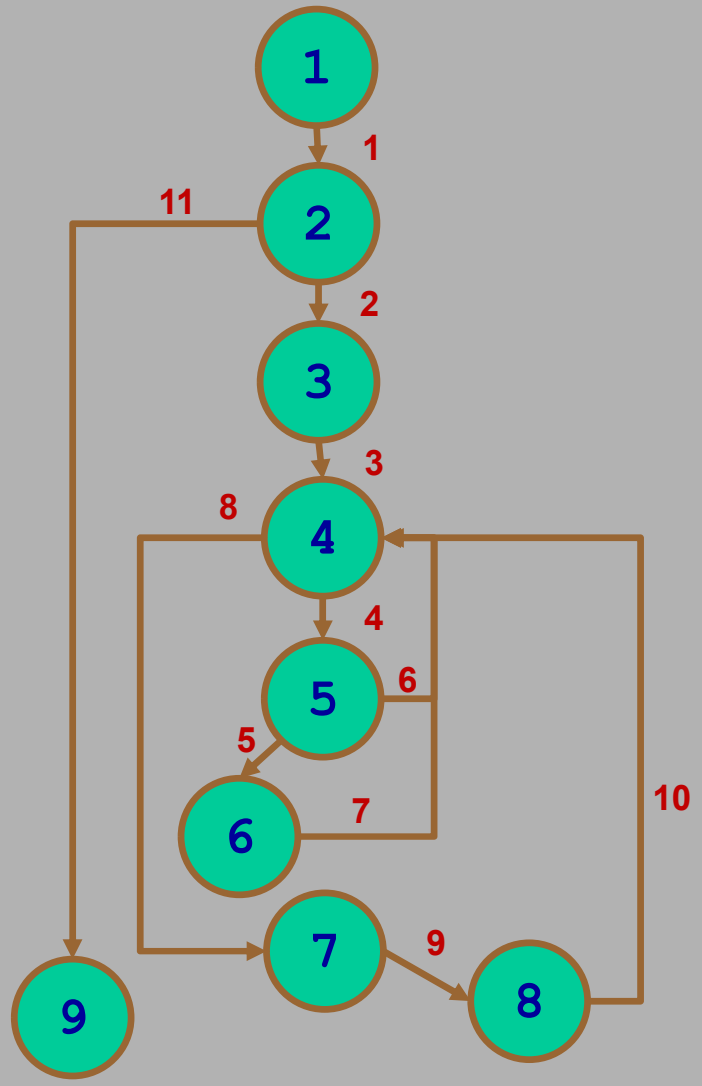
Complessità ciclomatica – 2/2

```
1: I := 0; N := 4;
2: while (I < N-1) do
3:    J := I+1;
4:    while (J < N) do
5:       if A[I] < A[J] then
6:          swap(A[I],A[J]);
7:    end do;
8:    I := I+1;
9: end do;
```

P

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

Archi 1,11 (F)
Archi 1,2 (T),3 (F),8,9,10
Archi 1,2 (T),3 (T),4,5 (T),7
Archi 1,2 (T),3 (T),4,6 (F)



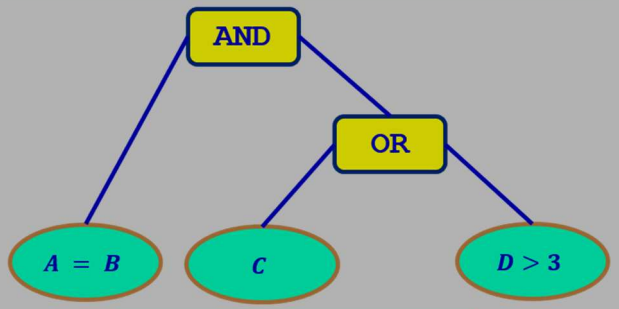
Decision/condition coverage

- La complessità delle espressioni di decisione influenza il grado di *branch coverage* effettivo
  - Più complessa l'espressione, più onerosa la sua copertura
- Condizione: espressione booleana semplice
  - Ciascuna condizione va soggetta a prove che producano T/F nella decisione almeno una volta ciascuna
- Decisione (*branch*): espressione composta da più condizioni
  - Ciascuna decisione va soggetta a prove che producano T/F almeno una volta ciascuna
- La tecnica *Modified Condition/Decision Coverage* (MCDC) ottiene massimo *condition-and-decision coverage* con il minor numero di prove

Approfondiamo MCDC …

- Per 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
- La tabella di verità ci mostra però che per raggiungere gli obiettivi di copertura MCDC servono 4 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 il comportamento dinamico del sistema completo rispetto ai requisiti SW**
  - **Si misura in *requirements coverage* conseguentemente alla copertura misurata dai TU funzionali**
  - **B. Meyer raccomanda che i TS includano tutti i casi di prova (TU, TI) che siano precedentemente falliti**
- **È inerentemente funzionale (*black-box*)**
  - **Non dovrebbe richiedere conoscenza della logica interna del SW**
  - **Così come i requisiti funzionali fissano l'aspettativa e non l'implementazione**
- **Ha inizio al completamento del TI**
- **È precursore del 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
    - Desiderabilmente, *test* già specificati e già eseguiti
  - **Coinvolge 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 alla presenza del committente**

Misure di copertura

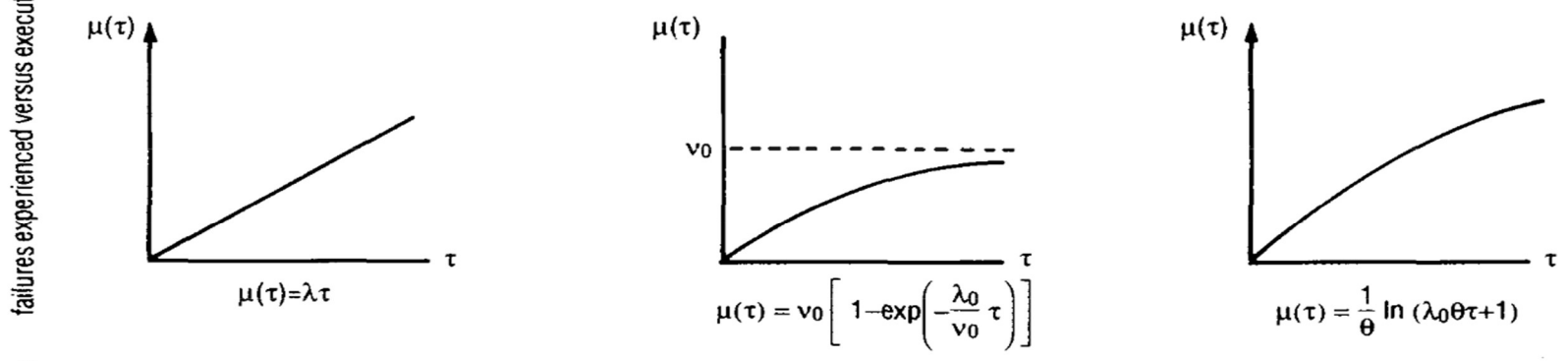
- **Dicono quanto le prove esercitano il prodotto**
  - **La copertura funzionale rispetto ai requisiti del prodotto**
  - **La copertura strutturale rispetto alla sua logica interna**
- **Quantificano la bontà della campagna di *test***
  - **La copertura del 100% complessivo non garantisce assenza di difetti**
  - **Raggiungere il 100% di copertura complessiva può non essere possibile**
    - Per ragioni di tempo/costo, di codifica, di strumenti
- **Gli obiettivi di copertura sono specificati nel PdQ**

Misure di maturità di prodotto

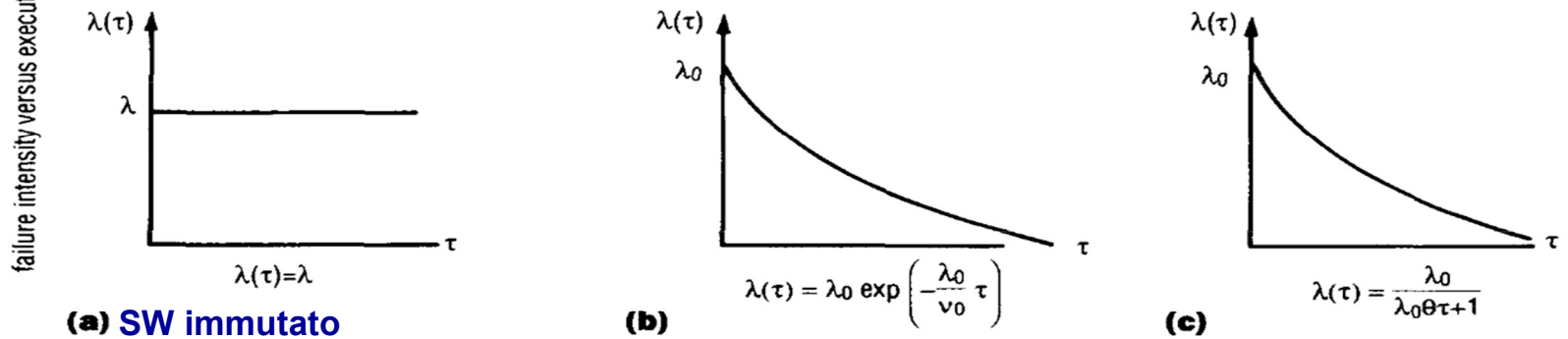
- Per valutare il grado di evoluzione del prodotto
  - Quanto il prodotto migliora in seguito alle prove
  - Quanto diminuisce la densità dei difetti
  - Quanto può costare la scoperta del prossimo difetto
- Le tecniche disponibili sono spesso empiriche
  - Ma sono stati fatti buoni passi avanti (vedi bibliografia)
- Serve definire un modello ideale di crescita della maturità
  - Modello base: il numero di difetti del SW è una costante iniziale
  - Modello logaritmico: le modifiche possono introdurre difetti

Quando conviene smettere i *test*?

Più *test* eseguo, più difetti rilevo?



Più *test* eseguo, meno difetti appaiono?



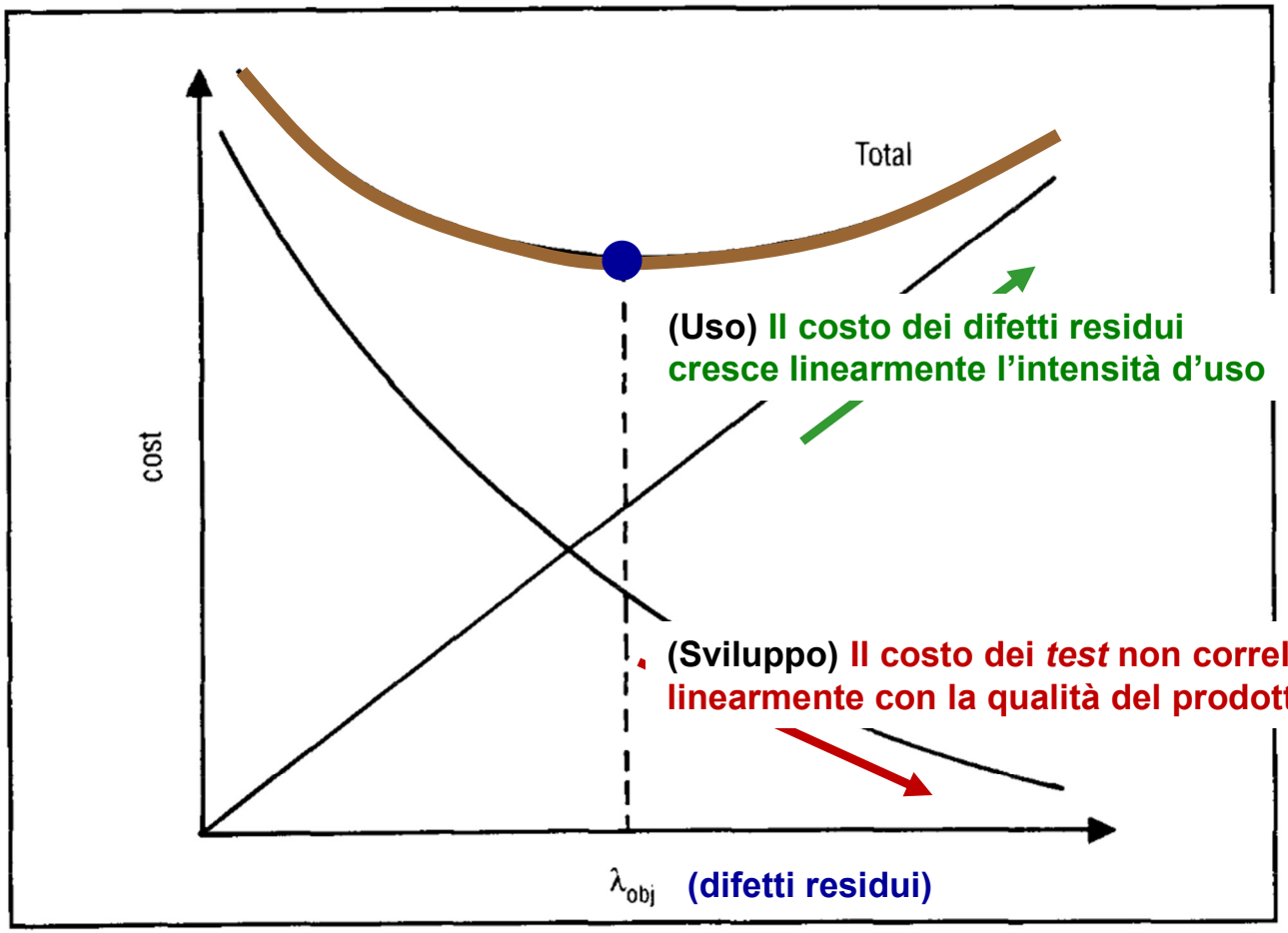
(a) SW immutato

(b) correzioni senza regressioni

(c) correzioni con regressioni

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 il costo di Sviluppo sostenibile

Un altro punto di vista [Ref. 3]

- Gli errori gravi spesso sono meno costosi di quelli più lievi
  - I primi sono trattati con urgenza, i secondi in modo più trascurato
- Correggere gli errori è molto costoso quando comporta modifiche architetturali
- Il costo degli errori non corretti 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*

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 T16)
3) J.C. Westland, *The cost of errors in software development: evidence from industry*, Journal of Systems and Software 62(1):1-9, 2002