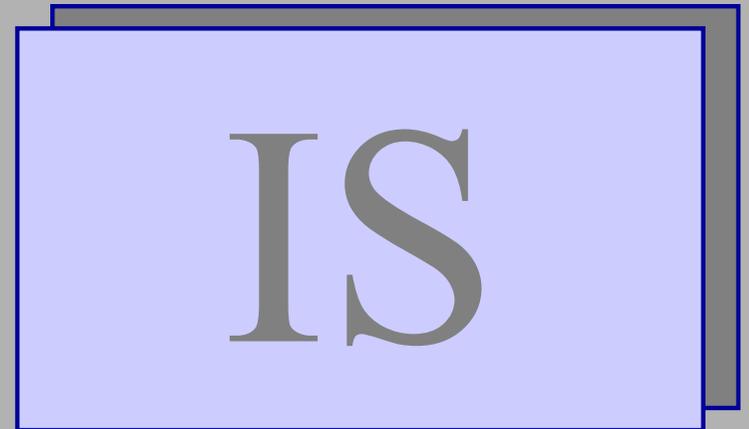




# Verifica e validazione: analisi dinamica



Anno accademico 2020/2021

Ingegneria del Software

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



# Terminologia

*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



## Definizioni – 1

- L'analisi dinamica richiede esecuzione dell'oggetto di verifica
  - Per farlo, utilizza un insieme di prove (*test*)
- Per studiare il comportamento del programma su un insieme finito di casi
  - In generale, il dominio di tutte le esecuzioni possibili è infinito: bisogna quindi ridurlo opportunamente
  - Ciascun caso di prova specifica i valori di ingresso e lo stato iniziale del sistema
  - Ciascun caso di prova deve produrre un esito decidibile, da confrontare con un comportamento atteso (**oracolo**)



## Definizioni – 2

### □ Oggetto della prova

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

### □ Obiettivo della prova

- Specificato per ogni caso di prova
- In termini precisi e quantitativi
- Varia al variare dell'oggetto della prova
- Il PdQ specifica quali e quante prove effettuare

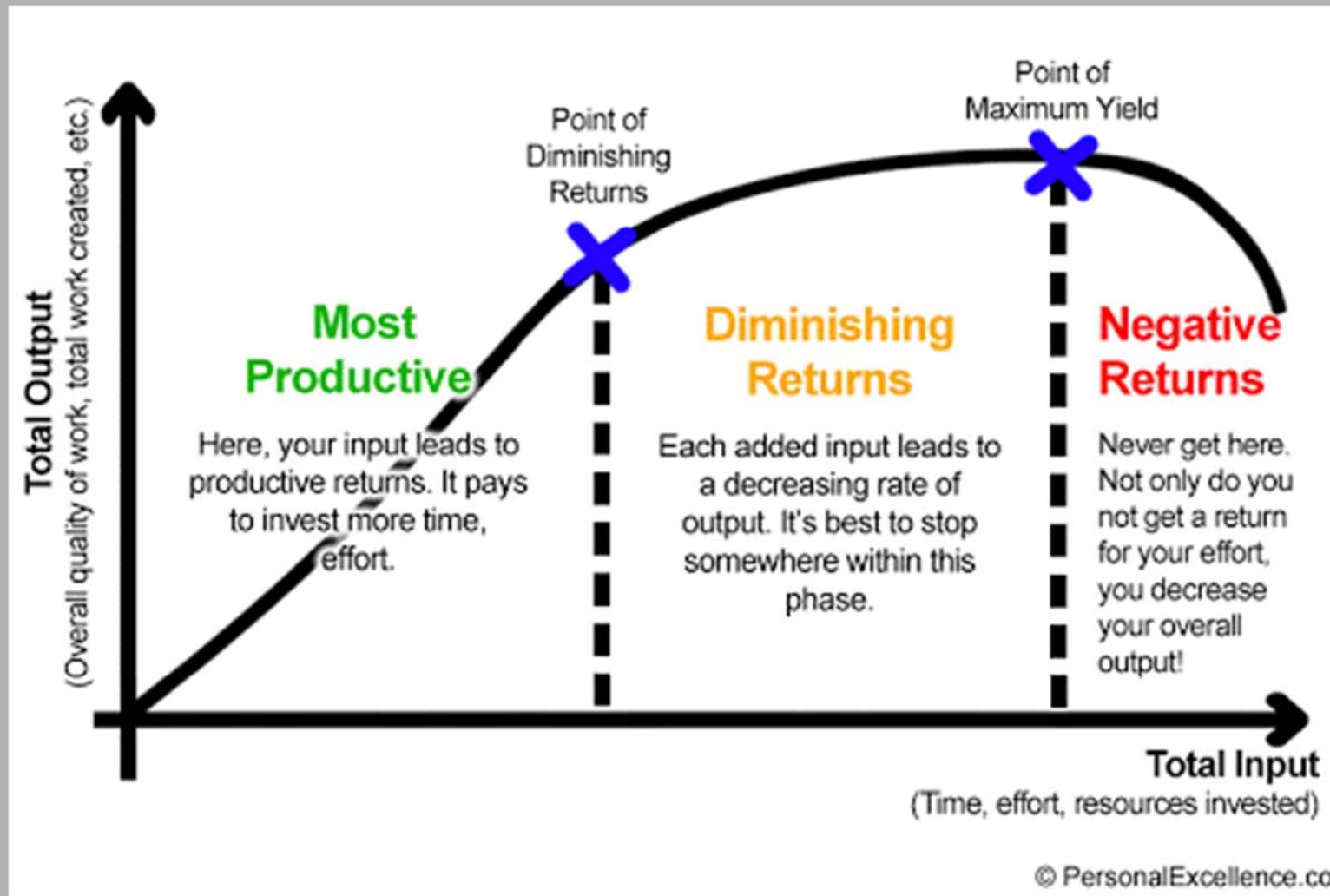


## Criteri guida – 1

- **La strategia di prova deve bilanciare il costo con i 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 alle prove, ma il PdQ fissa le necessità**
  - Disponibilità tardive o insufficienti danneggiano il progetto
  - Prima si fissa la strategia di prova (come, cosa, e con quale intensità), poi la si correla con il piano delle attività



# Legge del rendimento decrescente





## Criteri guida – 2

- ❑ Il *test* è parte essenziale del processo di verifica
- ❑ Produce una misura della qualità del prodotto
  - Identificando/rimuovendo difetti, ne aumenta la qualità
- ❑ L'inizio delle attività di *test* non va differito al termine delle attività di codifica
  - Farlo, blocca lo sviluppo
- ❑ Le sue esigenze devono essere assecondate dalla progettazione e la codifica del sistema
  - Progettare, realizzare, ed eseguire i *test* è costoso: conviene renderlo facile e produttivo



## Criteri guida – 3

- Il *test* è il processo di eseguire un programma con l'intento di trovarvi difetti

G.J. Myers, *The Art of Software Testing*, Wiley, 2011  
(3° edizione)

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





## Criteri guida – 4

- ❑ Una singola prova non basta
  - I suoi risultati valgono solo per quella specifica esecuzione
  - Non valgono più dopo una qualunque modifica
- ❑ La prova deve essere ripetibile
- ❑ Rileva malfunzionamenti indicando la presenza di guasti
  - In generale, non può provarne l'assenza!
- ❑ Le prove sono costose
  - Richiedono molte risorse (tempo, persone, infrastrutture)
  - Necessitano di un processo definito
  - Richiedono attività di ricerca, analisi, correzione



# Limiti e problemi

## □ 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)

## □ Tesi di Dijkstra (1969)

- Il *test* di un programma può rilevare la presenza di malfunzionamenti, ma non può dimostrarne l'assenza

## □ Teorema di Weyuker (1979)

- Dato un programma  $P$ , i seguenti problemi sono indecidibili
  - $\exists$  ingresso che causi l'esecuzione di un particolare comando di  $P$ ?
  - $\exists$  ingresso che causi l'esecuzione di una particolare condizione di  $P$ ?
  - $\exists$  ingresso che causi l'esecuzione di ogni comando/condizione/cammino di  $P$ ?



# Principi del *testing software*

Cf. per approfondire #23

## □ Secondo Bertrand Meyer

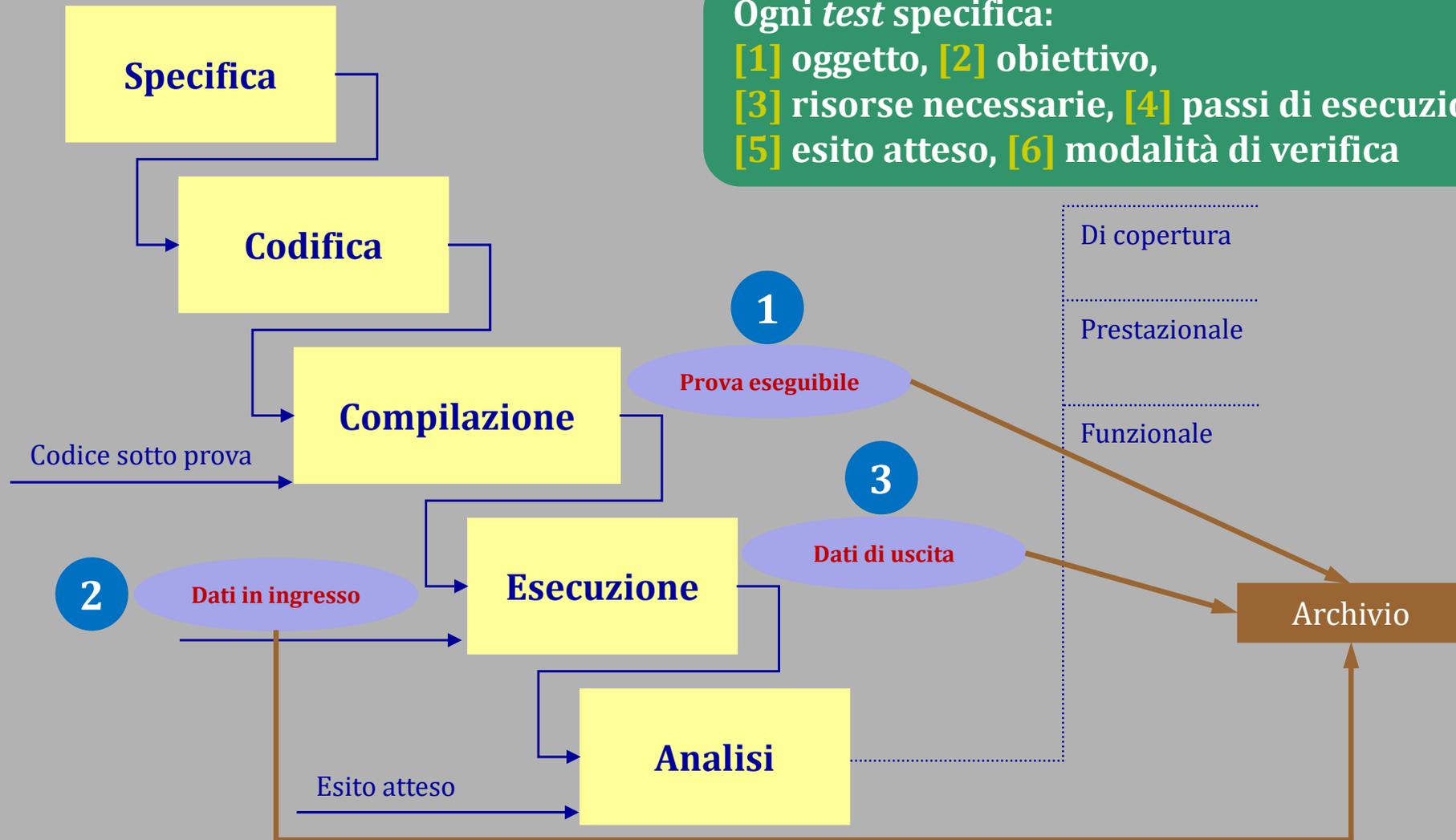
- *To test a program is to try to make it fail*
- *Tests are no substitutes for specifications*
- *Any failed execution must yield a test case, to be permanently included in the project's test suite*
- *Oracles should be part of the program text, as **contracts***
- *Any testing strategy should include 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

- **Caso di prova (*test case*)**
  - <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, registrare, analizzare e valutare i risultati di prove
- **Prova**
  - Procedura applicata a una batteria di prove



## Gli elementi di una prova – 2

### □ L'oracolo

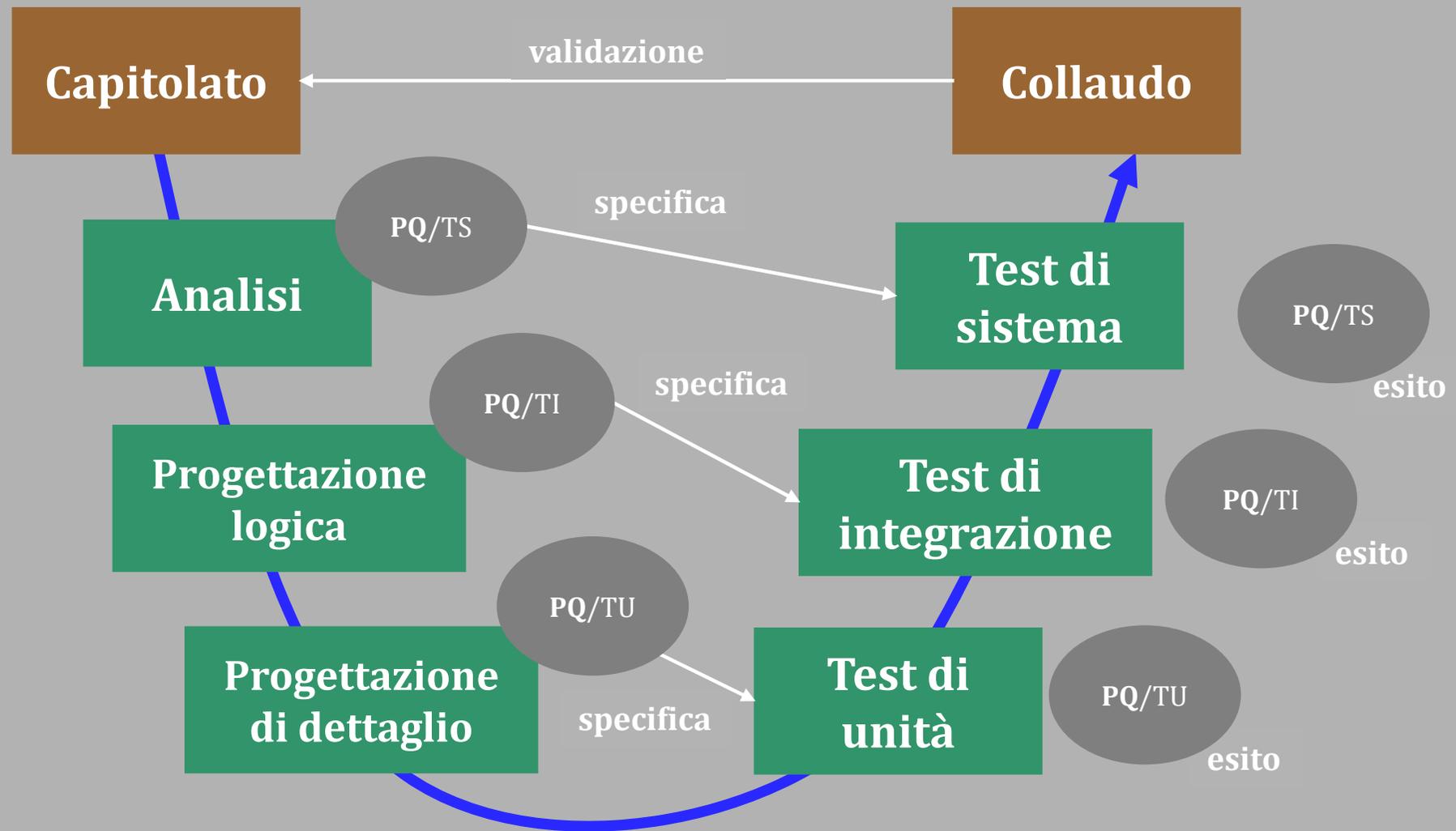
- Metodo per generare a priori i risultati attesi e per convalidare i risultati ottenuti nella prova
- Applicato da agenti automatici, per velocizzare la convalida e renderla oggettiva

### □ Come produrre oracoli

- Sulla base delle specifiche funzionali
- Con prove semplici (facilmente decidibili)
- Tramite l'uso di componenti terze indipendenti



# Esecuzione delle attività di prova





## Test di unità – 1

- ❑ **Unità SW composta da uno o più moduli**
  - Modulo = componente elementare di architettura di dettaglio
- ❑ **Unità/moduli specificati nella progettazione di dettaglio**
  - Il piano di TU viene definito con essa
- ❑ **La TU completa quando ha verificato tutte le unità**
- ❑ **~2/3 dei difetti rilevati tramite analisi dinamica viene segnalato in attività di TU**
  - 50% di essi viene identificato da prove strutturali (*white-box*)



## Test di unità – 2

### □ Test funzionale (*black-box*)

○ Fa riferimento alla specifica dell'unità e utilizza dati di ingresso capaci di provocare l'esito atteso

- Contribuisce, cumulativamente, al *requirements coverage*, misura di quanti requisiti funzionali siano verificati soddisfatti dal prodotto SW

○ Non può valutare correttezza e completezza della logica interna dell'unità

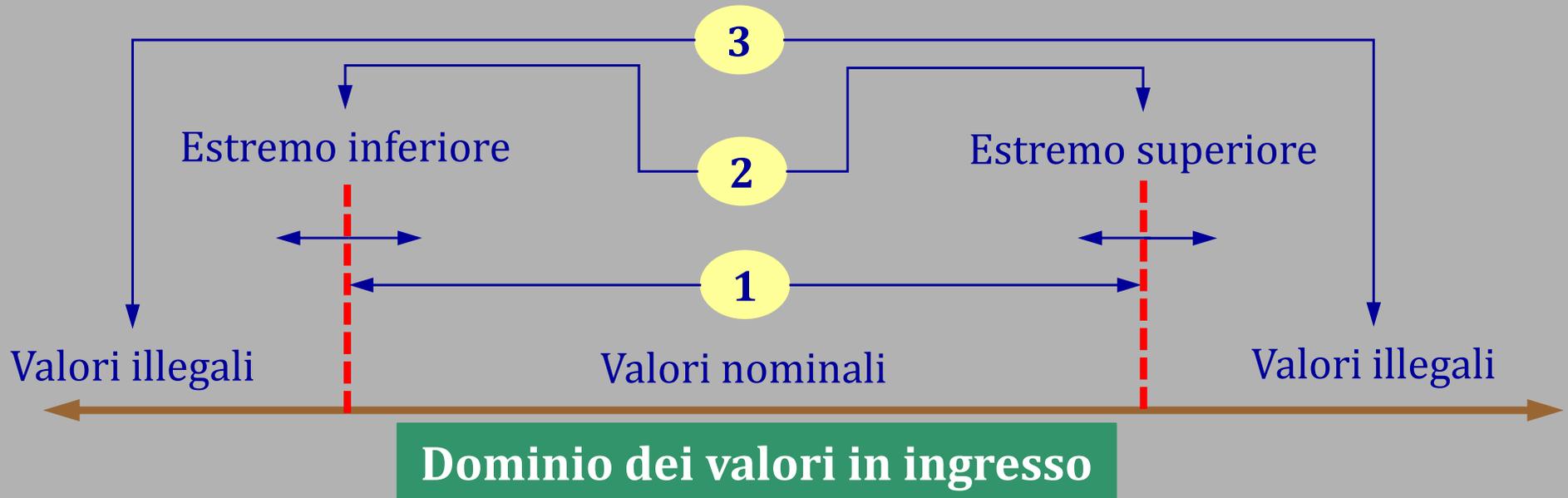
- Per questo va necessariamente integrato con *test* strutturale

○ I dati di ingresso che producono un dato comportamento funzionale formano un singolo caso di prova

- Utilizzando campioni delle *classi di equivalenza* dei valori di ingresso
- Valori nella medesima classe producono lo stesso comportamento



# 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

### □ Test strutturale (*white-box*)

- Verifica la logica interna del codice dell'unità cercando massima **copertura**
  - Aggiuntiva rispetto alla copertura dei requisiti
- Ogni singola prova deve attivare un singolo cammino di esecuzione all'interno dell'unità
  - E deve esplorare le condizioni logiche che determinano la scelta di ogni specifico cammino
- L'insieme di dati di ingresso e di configurazione di ambiente che produce un singolo cammino d'esecuzione costituisce un caso di prova



## Copertura *white-box*

- ❑ 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 decisioni prodotte da espressioni complesse



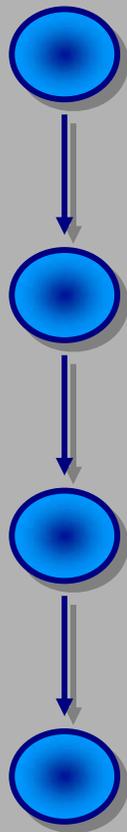
## *Branch coverage*

- Il numero di cammini (percorsi lineari) in una unità è detto **complessità ciclomatica**, CC
  - Tali cammini sono prodotti da *branch* e iterazioni
- La CC del grafo  $G$  che descrive il flusso d'esecuzione dell'unità, è  $v(G) = e - n + p$ , con
  - $e$  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 (per esecuzione sequenziale:  $p = 2$ , essendovi 1 predecessore e 1 successore per ogni arco)



# Complessità ciclomatica: esempi

## □ Sequenza S



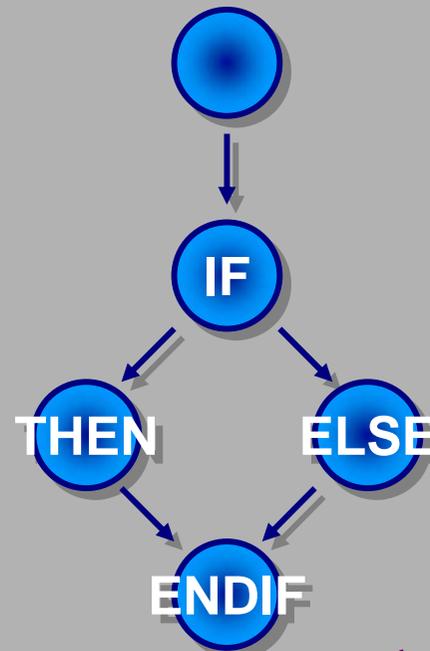
$$e = 3$$

$$n = 4$$

$$p = 2$$

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

## □ Decisione D



$$e = 5$$

$$n = 5$$

$$p = 2$$

$$v(D) = 5 - 5 + 2 = 2$$

La presenza di decisioni aumenta la CC



## ***Decision/condition coverage***

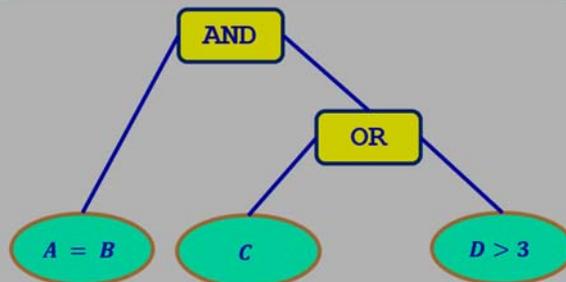
- La complessità delle **espressioni di decisione** (i.e., *branch*) influenza il grado di *branch coverage* effettivo
  - Più complessa l'espressione, più onerosa la sua copertura
- Il dominio avionico fornisce le seguenti direttive
  - **Condizione**: espressione booleana semplice
    - Ciascuna condizione va individualmente esplorata da almeno un *test*
  - **Decisione (*branch*)**: espressione composta da più condizioni
    - Come nel *branch coverage*, ciascuna decisione va individualmente esplorata da almeno un *test*
- Serve una tecnica per massimizzare il ***condition-and-decision coverage*** con il minor numero di prove possibile
  - Nota come ***Modified Condition/Decision Coverage*** (MCDC)



# Approfondiamo MCDC ...

- Sia data la seguente decisione, composta da tre decisioni

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



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

- **Obiettivo 1:** ciascuna condizione va sottoposta a prove che producano True o False nella decisione almeno una volta
- **Obiettivo 2:** ciascuna decisione va sottoposta a prove che producano True o False almeno una volta
- La tabella di verità qui a lato ci mostra che 4 prove bastano per soddisfare gli obiettivi MCDC
  - 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* architettuale**
  - La loro integrazione totale costituisce il sistema completo
- ❑ **Rileva difetti di progettazione architettuale o bassa qualità di TU**
  - Accerta che i dati scambiati attraverso ciascuna interfaccia concordino con la specifica
  - Accerta che tutti i flussi di controllo specificati siano 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 uno stato (baseline) precedente noto e sicuro



# Strategie di integrazione

- **Integrazione incrementale di tipo *bottom-up***
  - **Si sviluppano e si integrano prima le componenti con minori dipendenze d'uso e quindi maggiore utilità interna**
    - Quelle che sono molto chiamate/attivate ma chiamano/attivano poco o nulla
    - Quindi prima quelle più interne al sistema, meno visibili a livello utente
  - **Questa strategia richiede pochi *stub* ma ritarda la messa a disposizione di funzionalità di livello 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 a partire dalle 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
  - Esattamente come i requisiti funzionali fissano l'aspettativa e non il dettaglio della soluzione
- ❑ **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 che sia 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**
  - Copertura funzionale, rispetto ai requisiti del prodotto
  - Copertura strutturale, rispetto alla logica interna del SW
- ❑ **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**

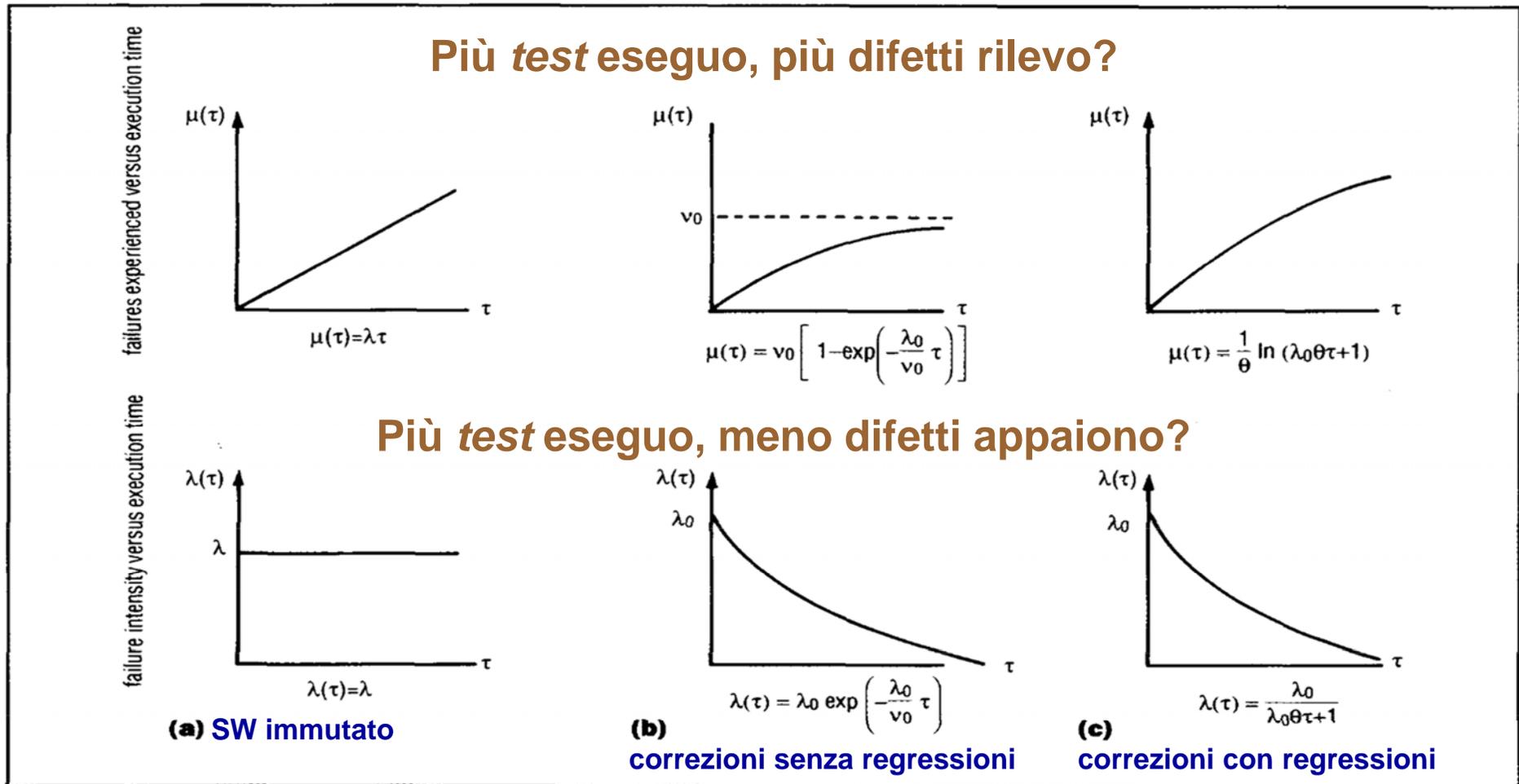


# Misure di maturità di prodotto

- 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 correnti 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 introducono difetti



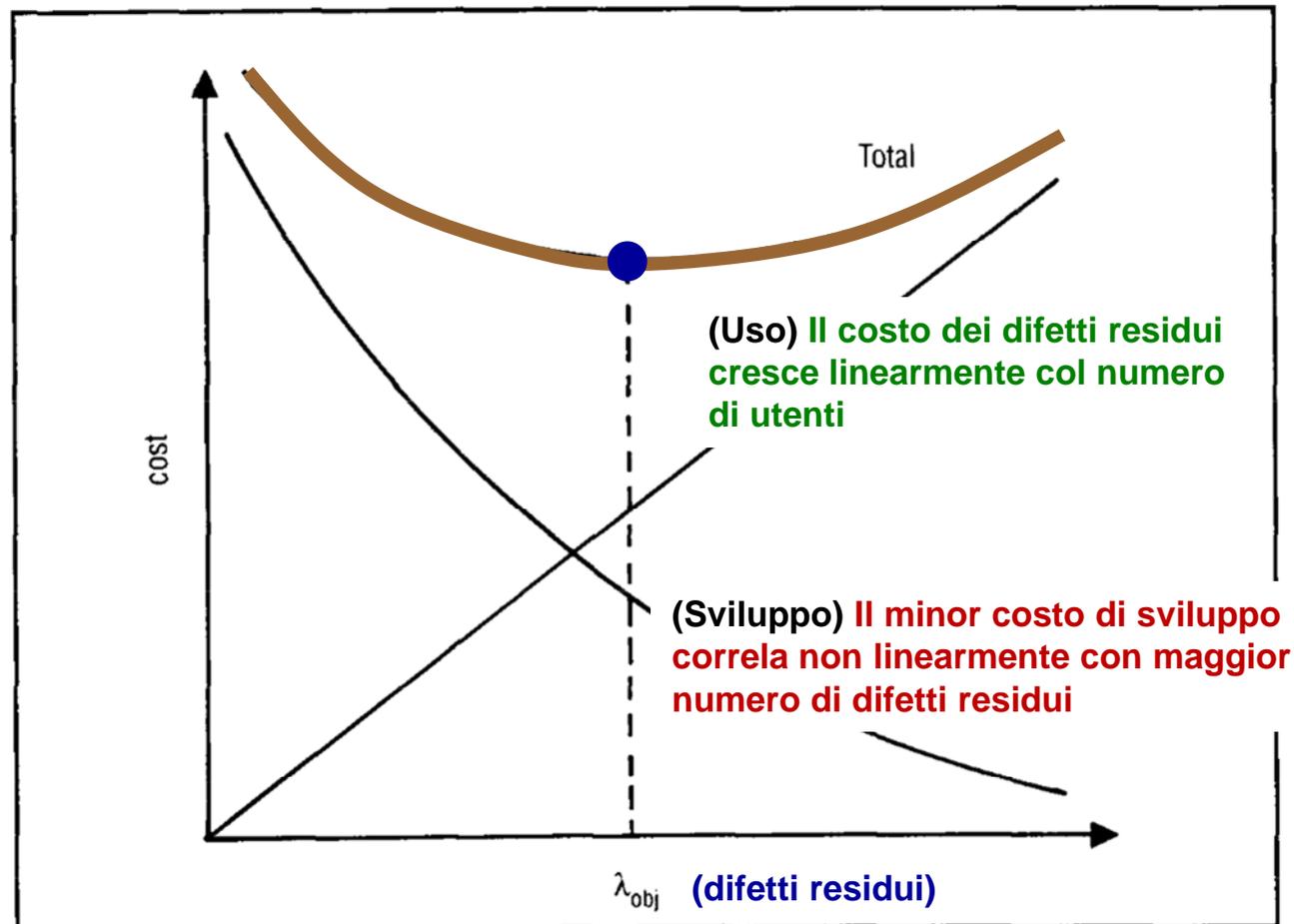
# Quando conviene smettere i *test*?



**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 il costo di Sviluppo sostenibile



## Un altro punto di vista [3]

- ❑ **Gli errori gravi spesso sono meno costosi di quelli più lievi**
  - Perché i primi sono trattati con urgenza, mentre i secondi in modo più discrezionale
- ❑ **Correggere gli errori è molto costoso quando comporta modifiche architeturali**
- ❑ **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 #23)
- 3) J.C. Westland, *The cost of errors in software development: evidence from industry*, Journal of Systems and Software 62(1):1-9, 2002