Verifica e validazione: analisi dinamica

IS

Anno accademico 2015/16
Ingegneria del Software mod. A

Tullio Vardanega, tullio.vardanega@math.unipd.it

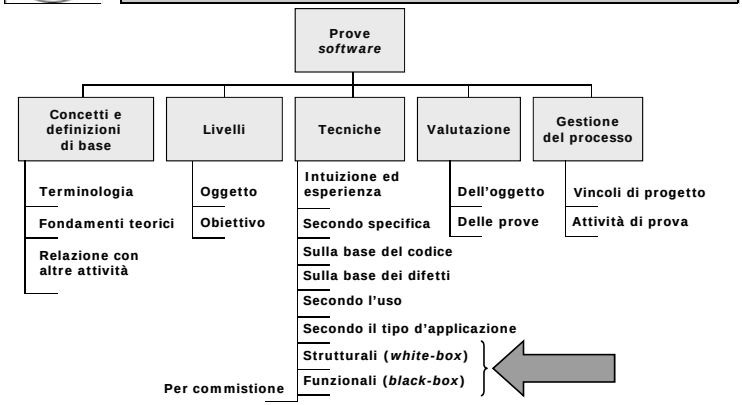
Definizione

- **Analisi dinamica = *test***
  - **Prova che comporta esecuzione**
- **La prova consiste nella verifica dinamica del comportamento del programma**
  - **Su un insieme finito di casi**
    - Selezionati nel dominio delle esecuzioni possibili che è in generale è infinito
    - Ciascun caso di prova specifica i valori di ingresso e lo stato iniziale del sistema
    - Ciascun caso di prova deve produrre un esito decidibile (**oracolo**)
  - **Verificati rispetto a un comportamento atteso**

Caratterizzazione

- **Parte essenziale del processo di verifica**
- **Produce una misura della qualità del sistema**
  - **Aumenta il valore di qualità del sistema identificandone e rimuovendone difetti**
- **Il suo inizio non deve essere differito al termine della fase di codifica**
- **Le sue esigenze devono essere tenute in conto nella progettazione del sistema**

Classificazione delle problematiche



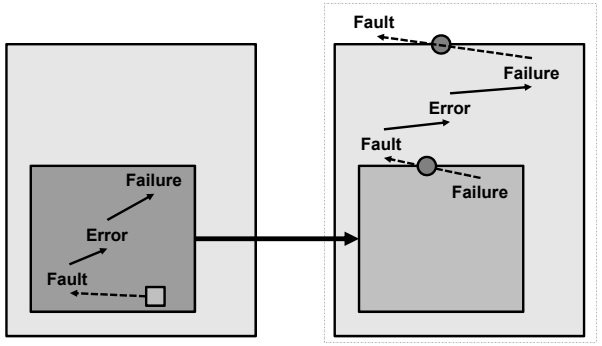
Dentro la classificazione – 1

- **Aree di studio e di conoscenza /1**
  - **Terminologia**
    - *Fault* → *Error* → *Failure*
    - Guasto → Errore (difetto) → Malfunzionamento
  - **Fondamenti teorici**
    - Decidibilità, testabilità, criteri
  - **Oggetto delle prove**
    - Unità, aggregati, sistema completo

Terminologia – 1

- ***A failure occurs when the behavior of a system deviates from what is specified for it***
  - ***Failures result from problems internal to the system which eventually manifest themselves in the system's external behavior***
- ***These problems are called errors and their mechanical or algorithmic or conceptual cause are termed faults***
  - ***Errors are states of the system***
  - ***Faults are what causes the error to exist***
- ***Systems are composed of components which are themselves systems: hierarchically therefore we have***
  - ***Failure} → {Fault → Error → Failure} → {Fault***

Terminologia – 2

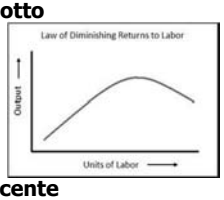


Dentro la classificazione – 2

- **Aree di studio e di conoscenza /2**
  - **Obiettivo delle prove**
    - Accettazione (=collaudo), qualifica, conformità, regresssione
    - Installazione nell'ambiente di prova, controllo delle prestazioni, …
  - **Vincoli di progetto**
    - Definizione del processo, dei prodotti, del personale addetto alle prove (interno o indipendente)
    - Stima e controllo dei costi, criteri di terminazione
  - **Attività di prova**
    - Pianificazione, specifica dei casi di prova, sviluppo dell'ambiente di prova
    - Esecuzione, valutazione, trattamento dei problemi (anomalie, discrepanze)

Fattori da bilanciare

- **La definizione della strategia di prova richiede bilanciamento tra**
  - **La quantità minima di casi di prova sufficienti a fornire certezza adeguate sulla qualità del prodotto**
    - Fattore governato da criteri tecnici
  - **La quantità massima di sforzo, tempo e risorse disponibile per il completamento della verifica**
    - Fattore governato da criteri gestionali
  - **Secondo la legge del rendimento decrescente (*diminishing returns*)**



Criteri guida – 1

- **Oggetto della prova**
  - **Il sistema nel suo complesso (TS)**
  - **Parti di esso, in relazione funzionale, d'uso, di comportamento, di struttura (TI)**
  - **Singole unità (TU)**
- **Obiettivo della prova**
  - **Specificato per ogni caso di prova**
  - **In termini precisi e quantitativi**
  - **Varia al variare dell'oggetto della prova**
  - **Il PdQ risponde alla domanda: quali e quante prove**

Criteri guida – 2

- **Una visione riduttiva ma efficace delle prove**
  - **"Il processo di eseguire un programma con l'intento di trovarvi difetti"**
    The Art of Software Testing, G.J. Myers, Wiley-Interscience, 1979
  - **Rappresenta lo spirito critico e l'atteggiamento scettico per metodo alla base di una strategia efficace di prova**
- **La "provabilità" del *software* 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: ne riparleremo!**

Criteri guida – 3

- **Una singola prova non basta**
  - **I risultati valgono solo per quella esecuzione**
    - Non possono essere generalizzati
  - **La prova deve essere ripetibile**
  - **Rileva malfunzionamenti indicando la presenza di guasti**
    - Ma 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

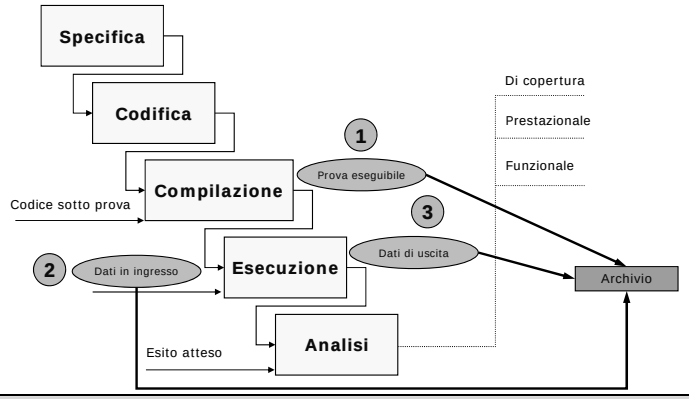
- **Teorema di Howden (1975)**
  - **Non esiste un algoritmo che, dato un programma P, generi per esso un *test* finito ideale (definito da criteri affidabili e validi)**
- **Tesi di Dijkstra (1969)**
  - **Il *test* di un programma può rilevare la presenza di malfunzionamenti, ma non dimostrarne l'assenza**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

Principi del *testing software*

Cf. per approfondire #25 di IS mod.A

- **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



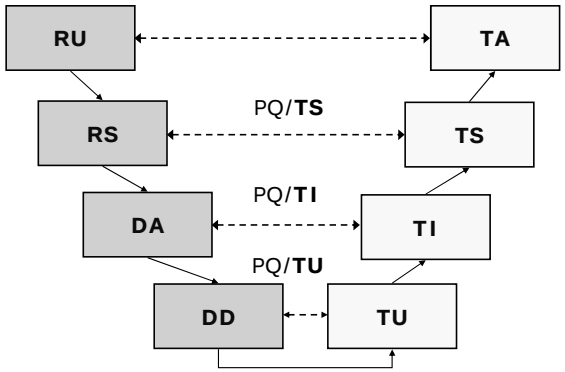
Gli elementi di una prova – 1

- **Caso di prova (*test case*)**
  - **Tripla <ingresso, uscita, ambiente>**
    - L'ambiente include l'oggetto della prova
- **Batteria di prove (*test suite*)**
  - **Insieme (sequenza) di casi di prova**
- **Procedura di prova**
  - **Il procedimento (automatico o manuale) per eseguire, registrare, analizzare e valutare i risultati di prove**
- **Prova = <Procedura, {caso di prova}>**

Gli elementi di una prova – 2

- **L'oracolo**
  - **Un metodo per generare a priori i risultati attesi**
  - **E per convalidare i risultati ottenuti**
  - **Generalmente applicato da agenti automatici**
    - Per velocizzare la convalida e renderla "oggettiva"
- **Come produrre oracoli**
  - **Sulla base delle specifiche funzionali**
  - **Sulla semplificazione delle prove**
  - **Sull'uso di componenti terze indipendenti**

Esecuzione delle attività di prova



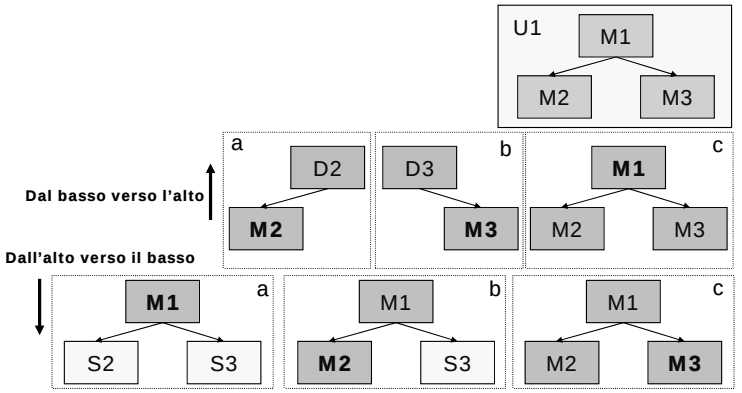
Strategie di integrazione

- **Assemblare parti in modo incrementale**
  - **I difetti rilevati in un *test* sono più probabilmente da attribuirsi alla parte ultima aggiunta**
- **Assemblare produttori prima dei consumatori**
  - **La verifica dei primi fornisce ai secondi flusso di controllo (chiamate) e flusso dei dati corretti**
- **Assemblare in modo che ogni passo di integrazione sia reversibile**
  - **Consente di retrocedere verso uno stato noto e sicuro**

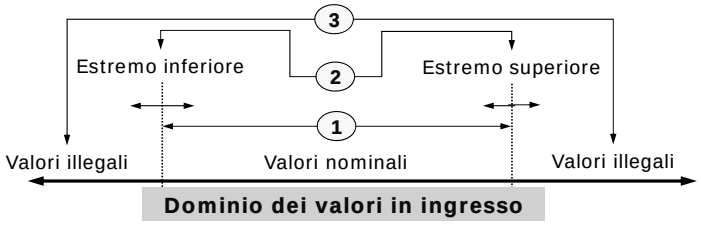
Integrazione incrementale – 1

- ***Bottom-up***
  - **Si sviluppano e si integrano prima le parti con minore dipendenza funzionale e maggiore utilità**
  - **Poi si risale l'albero delle dipendenze**
  - **Questa strategia riduce il numero di *stub* necessari al *test* ma porta più tardi alla disponibilità di funzionalità di alto livello**
- ***Top-down***
  - **Si sviluppano prima le parti più esterne, quelle poste sulle foglie dell'albero delle dipendenze e poi si scende**
  - **Questa strategia comporta l'uso di molti *stub* ma integra a partire dalla funzionalità di alto livello**

Integrazione incrementale – 2



Classi di equivalenza



- **3 classi di equivalenza**
  - Valori nominali interni al dominio (1)
  - Valori legali di limite (2)
  - Valori illegali (3)

Test di unità – 1

- **Unità *software* composta da uno o più moduli**
  - **Modulo = componente elementare di progetto di dettaglio (quindi non necessita di *test* di unità)**
- **Unità e moduli sono determinati in progettazione di dettaglio e quindi anche il piano di TU**
- **La TU completa quando ha verificato tutte le unità**
- **~2/3 dei difetti identificati tramite analisi dinamica vengono rilevati in TU**
  - **50% di essi viene identificato da prove strutturali (*white-box*)**

Copertura – 1

- **Per ogni test di unità si definiscono**
  - **Oggetto, strategia, risorse necessarie, piano di esecuzione**
- **Si ha *Statement Coverage* al 100%**
  - **Quando i *test* effettuati sull'unità sono sufficienti a eseguire – complessivamente – almeno una volta tutte le linee di comando di ciascuno dei moduli dell'unità**
- **Si ha *Branch Coverage* al 100%**
  - **Quando ciascun ramo del flusso di controllo viene attraversato – complessivamente – almeno una volta**

Copertura – 2

Lo *statement coverage* è meno potente del *branch coverage*

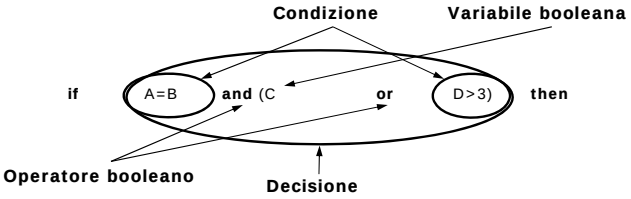

```
A1.1;
if (Cond)
{A1.2;
 A2};
A3;
```

Poniamo che A1.2 sia stato posto per errore nel ramo condizionale, quando invece dovrebbe essere sempre eseguito incondizionatamente insieme ad A.1.1

La strategia di ***statement coverage*** può non rilevare questo errore (che può produrre un *failure* funzionale) perché è solo interessata ad attraversare complessivamente quante più linee di comando possibile
La strategia di ***branch coverage*** (che è funzionale, per cammino) invece lo rileva, eseguendo la prova dove "Cond" vale Falso e A1.2 non viene eseguito

Copertura – 3

- **Definizione (DO-178B)**
  - **Condizione**
    - Espressione booleana semplice non contenente al suo interno ulteriori condizioni combinate da operatori booleani
  - **Decisione**
    - Espressione composta contenente condizioni combinate da operatori booleani



Copertura – 4

- **Per il frammento di codice a lato, massimizzare *statement coverage* e *branch coverage* non assicura il *test* di tutte le combinazioni T/F delle singole condizioni**

```
if (x>1 && y==0)
 {comando1}
else
{comando2}
```

- **Una batteria di *test* potrebbe per esempio esercitare soltanto i casi {x=2, y=0} e {x=2, y=1}**
- **Ma con un programma equivalente (vedi a lato), quella batteria di *test* raggiungerebbe *statement* e *branch coverage* solo dei 2/3**

```
if (x>1)
  if (y==0) {comando1}
  else {comando2}
else {comando2}
```

- **I casi di *test* {x=2, y=0}, {x=0, y=0}, {x=2, y=1} permettono invece di garantire *condition coverage* al 100%**

Il *branch coverage* può non comprendere appieno la logica del programma

Copertura – 5

- **Al crescere del numero di condizioni all'interno di una decisione il numero di *test* necessario a massimizzare il *condition coverage* diventa proibitivo**
- **DO-178B richiede allora di massimizzare il *modified decision condition coverage***
  - **MCDC implica *branch coverage***

Copertura – 6

- **Requisiti DO-178B (*software* di avionica)**
  - **Che tutte le decisioni siano soggette a *test* e tutti i loro possibili esiti siano effettivamente prodotti**
  - **Che ciascuna condizione all'interno di una decisione assuma entrambi gli esiti (vero/falso) almeno una volta**
- **Occorre allora verificare se e come ogni singola condizione possa da sola determinare tutti gli esiti possibili della decisione**
- **Non basta più provare l'intera decisione come vera o falsa!**

Copertura – 7

`if A=B and (C or D>3) then …`

Caso	Condizione / Variabile			Esito
	A=B	C	D>3	
1	*	F	F	F
2	T	T	*	T
3	T	*	T	T
4	F	*	*	F

Copertura – 8

```
if (pos < parseArray.length
   && (parseArray[pos] == '{'
       || parseArray[pos] == '}'
       || parseArray[pos] == '|'))
   {continue;}
```

C1, C2, C3, C4

Test case	C1	C2	C3	C4	*result*
1	false				false
2	true	true			true
3	true	false	true		true
4	true	false	false	true	true
5	true	false	false	false	false

Test di unità – 2

- ***Test* funzionale (*black-box*)**
  - **Da solo <u>non</u> può accertare correttezza e completezza della logica interna dell'unità**
    - Va necessariamente integrato con *test* strutturale
  - **Fa riferimento alla specifica dell'unità e utilizza dati di ingresso capaci di provocare l'esito atteso**
  - **Ciascun insieme di dati di ingresso che <u>producono un dato comportamento funzionale</u> costituisce un caso di prova**
  - **<u>Classi di equivalenza</u> invece che infiniti valori di ingresso**
    - Valori nella medesima classe producono lo stesso comportamento

Test di unità – 3

- ***Test* strutturale (*white-box*)**
  - Verifica la logica interna del codice dell'unità cercando massima copertura
  - Ciascuna prova deve essere progettato per attivare ogni cammino di esecuzione all'interno del modulo
  - Ciascun insieme di dati di ingresso che attivano un percorso costituiscono un caso di prova
  - L'uso di *debugger* può agevolarne l'esecuzione ma non esonera dalla progettazione dei casi di prova

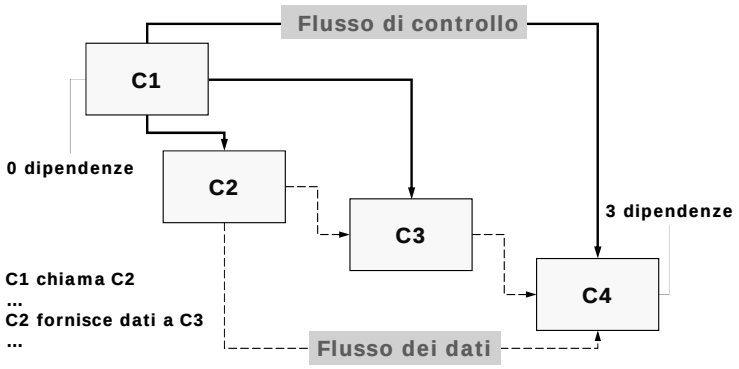
Test di integrazione – 1

- **Si applica alle componenti specificate nella progettazione architetturale**
  - La loro integrazione costituisce il sistema completo
- **Logica di integrazione funzionale**
  - Seleziona le funzionalità da integrare
  - Identifica le componenti che svolgono quelle funzionalità
  - Ordina le componenti per numero di dipendenze crescente
    - Dipendenze nel flusso di controllo (chiamata) e nel flusso di dati
  - Esegue l'integrazione in quell'ordine

Test di integrazione – 2

- **Problemi rilevati durante TI**
  - Manifestano difetti di progettazione o insufficiente qualità nei *test* di unità
- **TI ha tanti *test* quanto ne servono per**
  - Accertare che tutti i dati scambiati attraverso ciascun interfaccia siano conformi alla loro specifica
  - Accertare che tutti i flussi di controllo previsti in specifica siano stati effettivamente realizzati e provati

Esempio



Test di sistema

- Verifica il comportamento dinamico del sistema completo rispetto ai requisiti SW
- Ha inizio con il completamento del *test* di integrazione
- È inerentemente funzionale (*black-box*)
  - Non dovrebbe richiedere conoscenza della logica interna del *software*

Altri tipi di *test*

- *Test* di regressione
  - Ripetizione selettiva di TI (e poi di TS)
    - Integrando solo parti che abbiano precedentemente superato TU
    - Nel *repository* di progetto devono stare solo unità SW di questo tipo
  - Per accertare che modifiche intervenute per correzione o estensione di parti non comportino errori nel sistema
    - Può essere molto oneroso
  - I contenuti del *test* di regressione vanno decisi nel momento in cui si approvano modifiche al SW
- *Test* di accettazione (collaudo)
  - Accerta il soddisfacimento dei requisiti utente

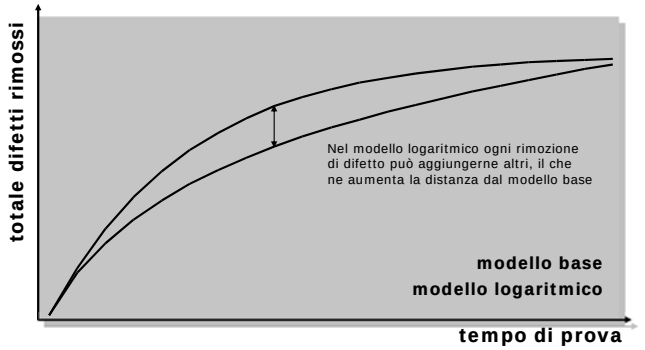
Fattore di copertura

- Quanto la prova esercita il prodotto
  - Copertura funzionale
    - Rispetto alla percentuale di funzionalità esercitate come viste dall'esterno
  - Copertura strutturale (*branch*, *condition*)
    - Rispetto alla percentuale di logica interna del codice esercitata
- Una misura della bontà di una prova
  - Copertura del 100% non prova assenza di difetti
  - Il 100% di copertura può essere irraggiungibile
    - Costi eccessivi, codice sorgente non disponibile, codice irraggiungile non eliminabile, copertura esaustiva dei cicli

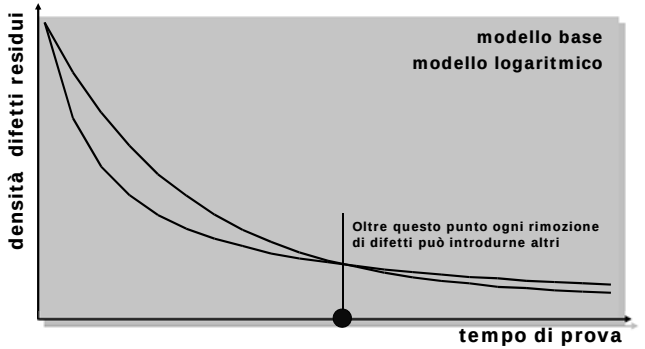
Maturità di prodotto

- 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
  - Sotto l'influenza del modello *code-and-fix*
- Definire un modello ideale
  - Modello base: il numero di difetti del *software* è una costante iniziale
  - Modello logaritmico: le modifiche introducono difetti!

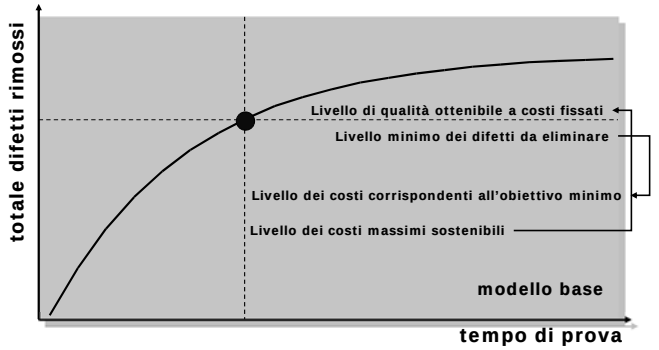
La funzione $\mu(t)$



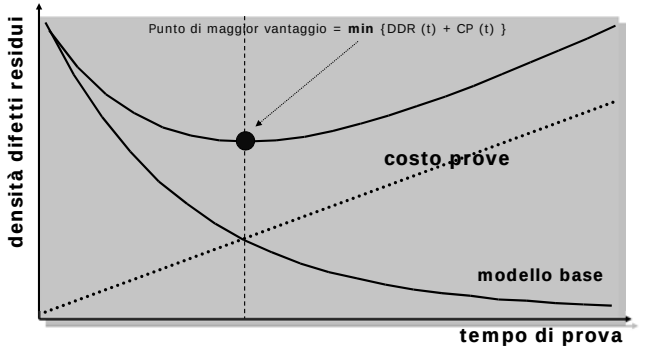
La funzione $\lambda(t)$



Criteri economici e qualitativi



Criteri analitici



Bibliografia

- J.D. Musa, A.F. Ackerman
  Quantifying software validation: when to stop testing?
  IEEE Software, maggio 1989

- B. Meyer
  Seven Principles of Software Testing
  IEEE Computer, agosto 2008
  (cf. per approfondire #25 di IS mod.A)