Verifica e validazione: analisi dinamica

Anno accademico 2018/19
Ingegneria del Software

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

Definizione

- Analisi dinamica = *test* (prova)
  - Comporta esecuzione dell'oggetto di verifica
- Verifica dinamica del 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, verificato rispetto a un comportamento atteso (oracolo)

Caratterizzazione

- Il *test* è parte essenziale del processo di verifica
- Produce una misura della qualità del sistema
  - Identificando e rimuovendo difetti nel sistema, ne aumenta il valore di qualità
- L'inizio delle attività di *test* non va differito al termine delle attività di codifica
- Le sue esigenze devono essere tenute in conto nella progettazione del sistema

Terminologia – 1

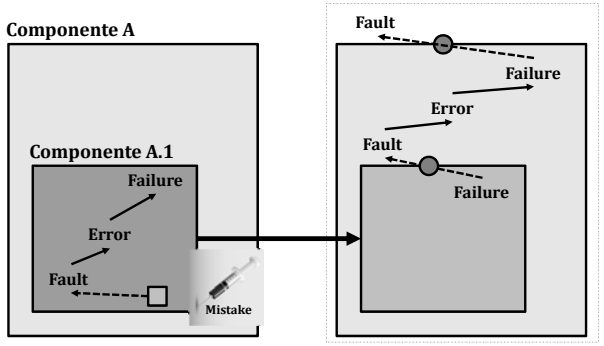
The fault tolerance discipline distinguishes between a human action (a mistake), its manifestation (a hardware or software fault), the result of the fault (a failure), and the amount by which the result is incorrect (the error).

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

Terminologia – 2

- ***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 in the system's external behavior***
- ***Such problems are called errors and their mechanical, algorithmic or conceptual cause are termed faults***
  - ***Errors are states of the system***
  - ***Faults are what causes the error to exist***
- ***Systems are hierarchical compositions of components which are themselves systems***

Visione gerarchica



Fattori da bilanciare

- **La strategia di prova richiede bilanciamento**
  - **Determinare la quantità minima di casi di prova sufficienti a garantire la qualità del prodotto**
    - Fattore governato da criteri tecnici
  - **Determinare la quantità massima di sforzo, tempo e risorse disponibile per la verifica**
    - Fattore governato da criteri gestionali
- **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, 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 – 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
- **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: ne riparleremo!** 

Criteri guida – 3

- **Una singola prova non basta**
  - **I suoi risultati valgono solo per quella 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**
    - ∃ ingresso che causi l'esecuzione di un particolare comando di P?
    - ∃ ingresso che causi l'esecuzione di una particolare condizione di P?
    - È possibile trovare un ingresso che causi l'esecuzione di ogni comando / condizione / cammino di P?

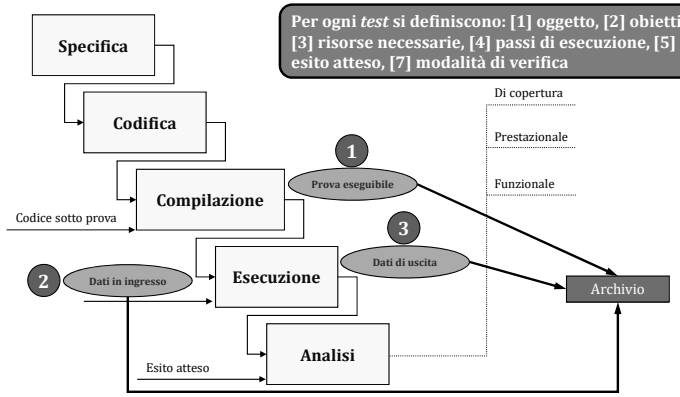
Principi del *testing software*

Cf. per approfondire #24

- **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

Per ogni *test* si definiscono: [1] oggetto, [2] obiettivo, [3] risorse necessarie, [4] passi di esecuzione, [5] esito atteso, [7] modalità di verifica



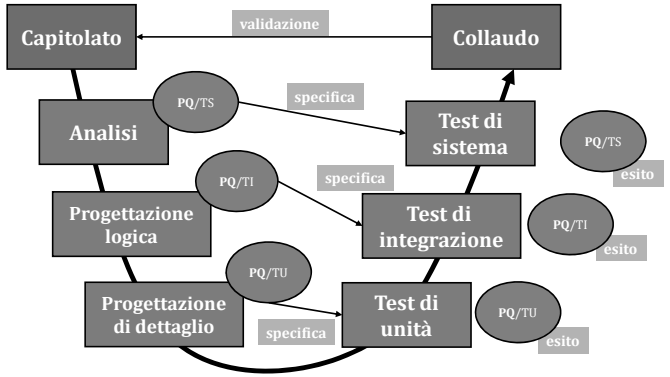
Gli elementi di una prova – 1

- **Caso di prova (*test case*)**
  - **<Ingresso, Uscita, Oggetto di prova, Ambiente di esecuzione>**
- **Batteria di prove (*test suite*)**
  - **Insieme di casi di prova**
- **Procedura di prova**
  - **Il procedimento (automatico) per eseguire, registrare, analizzare e valutare i risultati di prove**
- **Prova**
  - **Una procedura applicata a una batteria di prove**

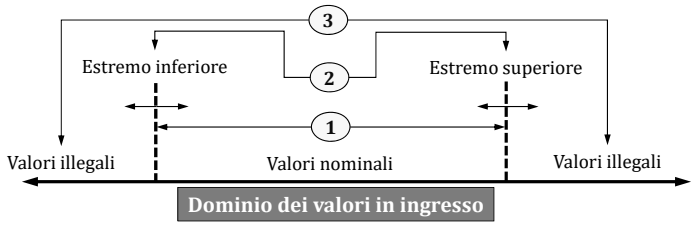
Gli elementi di una prova – 2

- **L'oracolo**
  - **Un metodo per generare a priori i risultati attesi e per convalidare i risultati ottenuti nella prova**
  - **Generalmente 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



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 SW composta da uno o più moduli**
  - **Modulo = componente elementare di architettura di dettaglio**
- **Unità e moduli sono 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*)**
  - **Da solo, non 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 produca un dato comportamento funzionale costituisce un singolo caso di prova**
    - Utilizzando campioni delle classi di equivalenza dei 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**
  - **Ogni singola prova deve attivare un singolo cammino di esecuzione all'interno dell'unità**
  - **L'insieme di dati di ingresso (e di configurazione di ambiente) che ottiene quell'effetto costituisce un caso di prova**

Copertura – 1

- Due aspetti complementari
- Si ha *Statement Coverage* al 100%
  - Quando i *test* effettuati sull'unità sono sufficienti a eseguire – cumulativamente – almeno una volta tutti i comandi dell'unità, ciascuno con esito corretto
- Si ha *Branch Coverage* al 100%
  - Quando ciascun ramo del flusso di controllo dell'unità viene attraversato – complessivamente – almeno una volta, ciascuna con esito corretto

Copertura – 2

- Il valore di copertura del *branch coverage* è determinato dalla complessità delle espressioni di decisione
- Secondo DO-178B (per SW di avionica)
  - Condizione: espressione booleana semplice, non contenente operatori booleani
  - Decisione: espressione composta, contenente condizioni combinate da operatori booleani
  - MCDC per massimizzare il BC al minor costo

Modified condition/decision coverage

Esempio: una decisione, composta da tre condizioni

`if (A=B) and ((C) or (D>3)) then …`

Caso	Condizioni			Esito
	A=B	C	D>3	
1	•	F	F	F
2	T	T	•	T
3	T	•	T	T
4	F	•	•	F

- Tutte le decisioni vanno sottoposte a test; tutti i possibili esiti di ogni decisione devono essere singolarmente prodotti da un *test*
- Ciascuna condizione all'interno di una decisione deve assumere entrambi i valori di verità (T/F) almeno una volta

Test di integrazione – 1

- Si applica alle componenti specificate nella progettazione architetturale
  - La loro integrazione totale 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

Strategie di integrazione

- Assemblare parti in modo <u>incrementale</u>
  - Aggiungendo solo a insiemi ben verificati, i difetti rilevati in un *test* di integrazione sono più probabilmente da attribuirsi alla parte ultima aggiunta
- Assemblare produttori <u>prima</u> 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 <u>reversibile</u>
  - Consente di retrocedere verso uno stato noto e sicuro

Integrazione incrementale

- *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 ritarda la 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 dalle funzionalità di più alto livello

Test di integrazione – 2

- Problemi rilevati durante TI
  - Manifestano difetti di progettazione o bassa qualità di TU
- TI ha tanti *test* quanto ne servono per
  - Accertare che <u>tutti i dati</u> scambiati attraverso ciascuna interfaccia siano conformi alla loro specifica
  - Accertare che <u>tutti i flussi di controllo</u> previsti in specifica siano stati effettivamente provati

Test di sistema

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

Altri tipi di *test*

- ***Test* di regressione**
  - **Ripetizione selettiva di TU, TI e TS**
    - Integrando solo parti che abbiano precedentemente superato TU
    - Nel *repository* di progetto devono stare solo unità SW con questa caratteristica
  - **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**
    - Parte del processo *Change Management*
- ***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% <u>non</u> prova assenza di difetti**
  - **Il 100% di copertura può essere <u>irraggiungibile</u>**
    - Costi eccessivi, codice sorgente non disponibile, codice irraggiungibile 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 *SW* è una costante iniziale**
  - **Modello logaritmico: le modifiche introducono difetti**

Quando conviene smettere i *test*?

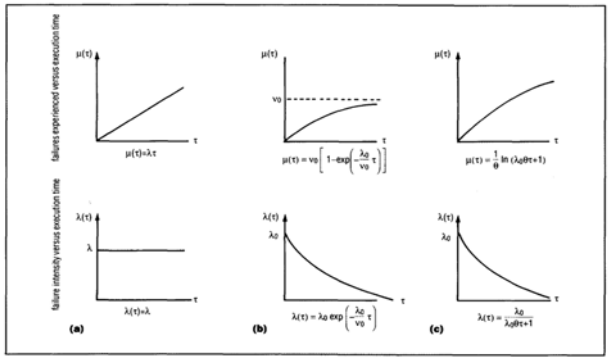


Figure 1. Three useful software-reliability models: **(a)** static, **(b)** basic, and **(c)** logarithmic Poisson. These are shown comparing both failures experienced versus execution time and failure intensity versus execution time.

La risposta di Musa & Ackerman (1989)

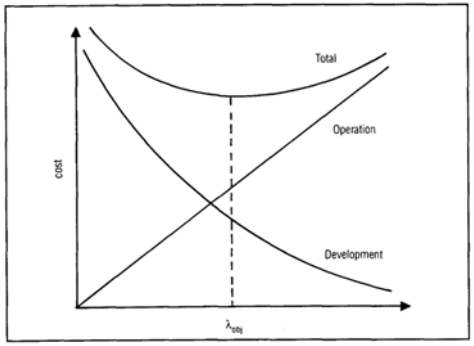


Figure 5. Selecting a failure-intensity objective that will minimize the cost of failure over the life cycle.

Bibliografia

- **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**
- **B. Meyer**
  Seven Principles of Software Testing
  IEEE Computer, agosto 2008
  (cf. per approfondire #24)