Verifica e validazione: analisi dinamica

IS

Anno accademico 2019/2020
Ingegneria del Software

Tullio Vardanega, tullio.vardanega@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, da confrontare con un comportamento atteso (oracolo)

Caratterizzazione

- Il *test* è parte essenziale del processo di verifica
- Produce una misura della qualità del prodotto
  - Identificando/rimuovendo difetti, 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

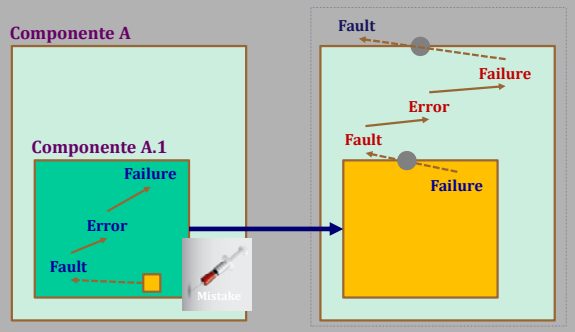
The fault tolerance discipline distinguishes between a human action (a mistake), its manifestation (a hardware or software fault), the result of the fault (a failure), and the amount by which the result is incorrect (the error).

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 deve bilanciare**
  - **La quantità minima di casi di prova sufficienti a garantire la qualità di prodotto desiderata**
    - Fattore governato da criteri tecnici
  - **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

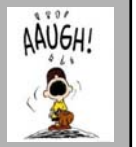
- **Oggetto della prova**
  - **Il sistema nel suo complesso (TS)**
  - **Parti di esso, in relazione funzionale, d'uso, di comportamento, di struttura, tra loro (TI)**
  - **Singole unità, considerate indipendentemente (TU)**
- **Obiettivo della prova**
  - **Specificato per ogni caso di prova**
  -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

Criteri guida – 2

- Il *test* è il processo di eseguire un programma con l'intento di trovarvi difetti
  - G.J. Myers, *The Art of Software Testing*, 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

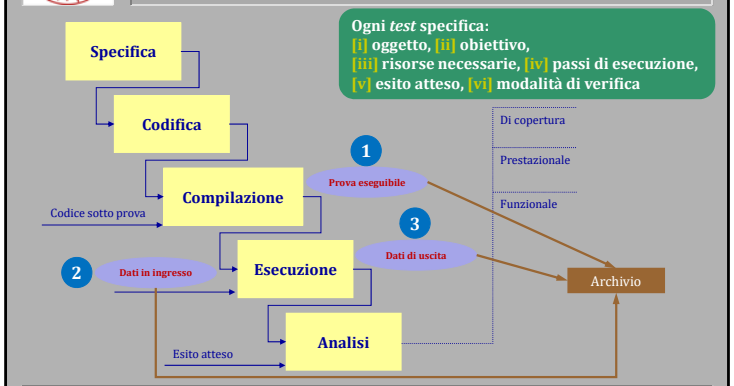
- Teorema di Howden (1975)
  - Non esiste un algoritmo che, dato un programma P, generi per esso un *test* finito ideale (definito da criteri affidabili e validi)
- Tesi di Dijkstra (1969)
  - Il *test* di un programma può rilevare la presenza di malfunzionamenti, ma non può dimostrarne l'assenza
- Teorema 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?
    - ∃ 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



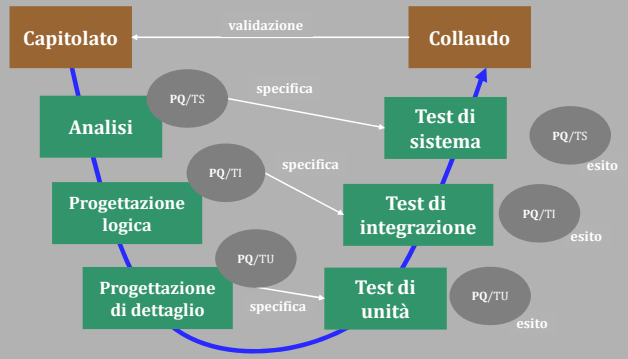
Gli elementi di una prova – 1

- 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



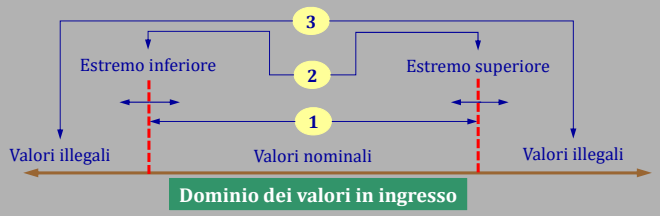
Test di unità – 1

- 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
  - 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 *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 del flusso di controllo dell'unità viene attraversato almeno una volta da un *test* dedicato, con esito corretto
- Si ha ***Decision/Condition Coverage*** al 100 %
  - Quando ogni condizione della decisione assume almeno uno volta entrambi i valori di verità in un *test* dedicato

Branch coverage

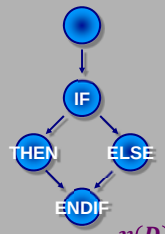
- Il numero di cammini (percorsi lineari) in una unità è detto **complessità ciclomatica**, CC
- La CC del grafo G che descrive il flusso d'esecuzione dell'unità, è $v(G) = e - n + p$, con
  - e numero degli archi in G (flusso tra comandi)
  - n numero dei nodi in G (espressioni o comandi)
  - p numero delle componenti connesse da ogni arco (per esecuzione sequenziale: $p = 2$, essendovi 1 predecessore e 1 successore)

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** influenza il grado di *branch coverage*
  - Più complessa l'espressione, più onerosa la sua copertura
- Secondo DO-178B (dominio avionico)
  - **Condizione**: espressione booleana semplice, non contenente operatori booleani
  - **Decisione**: espressione composta, contenente condizioni combinate da operatori booleani
- Serve una tecnica per massimizzare il DC con il minor numero di *test* possibile
  - *Modified condition/decision coverage* (MCDC)

MCDC

- Ciascuna condizione va sottoposta a *test*
  - Producendo ciascuno dei suoi esiti in almeno un *test* dedicato
- Ciascuna decisione va sottoposta a *test*
  - Producendo ciascuno dei suoi esiti in almeno un *test* di condizione

Esempio: una decisione composta da tre condizioni

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

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

Con MCDC, per questa decisione "bastano" 4 *test*

Test di integrazione – 1

- Si applica alle componenti specificate nella progettazione architetturale
  - La loro integrazione totale costituisce il sistema completo
- La logica di integrazione funzionale
  - Seleziona le funzionalità da integrare
  - Identifica le componenti che svolgono quelle funzionalità
  - Ordina le componenti per numero crescente di dipendenze
    - Nel flusso di controllo (chiamata) e nel flusso di dati
  - Esegue l'integrazione in quell'ordine
    - Da chi chiama/attiva verso chi è chiamato/attivato, come in una *pipeline* …

Strategie di integrazione – 1

- Assemblare parti in modo <u>incrementale</u>
  - Aggiungendo solo a insiemi già verificati, i difetti rilevati in un TI sono più probabilmente da attribuirsi alla parte ultima aggiunta
- Assemblare produttori <u>prima</u> di consumatori
  - La verifica dei primi abilita la corretta attivazione dei secondi
- Assemblare in modo che ogni passo di integrazione sia <u>reversibile</u>
  - Potendo retrocedere a uno stato noto e sicuro (una *baseline* precedente)

Strategie di integrazione – 2

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

Test di integrazione – 2

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

Test di sistema

- Verifica il comportamento dinamico del sistema completo rispetto ai requisiti SW
  - Si misura in *requirements coverage*
- Ha inizio con il completamento del TI
- 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
- È precursore del collaudo

Altri tipi di *test*

- *Test* di regressione
  - Per accertare che modifiche intervenute per correzione o estensione di parti non introducano errori nel sistema
  - Consiste nella ripetizione selettiva di TU, TI e TS
    - Integrando solo parti che abbiano precedentemente superato TU
  - I contenuti dei TR vanno decisi contestualmente all'approvazione di modifiche al prodotto
    - Come parte del processo *Change Management*

- *Test* di accettazione (collaudo)
  - Accerta il soddisfacimento dei requisiti utente, alla presenza del committente

Fattore di copertura

- Misura di quanto le prove esercitano il prodotto
  - Copertura funzionale, rispetto ai requisiti del prodotto
  - Copertura strutturale, rispetto alla logica interna del SW
- Quantifica la bontà della campagna di *test*
  - La copertura del 100% su ciascuno dei fronti non assicura assenza di difetti
- Raggiungere il 100% di copertura complessivo può non essere possibile
  - Per ragioni di tempo/costo, di codifica, di strumenti

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*?

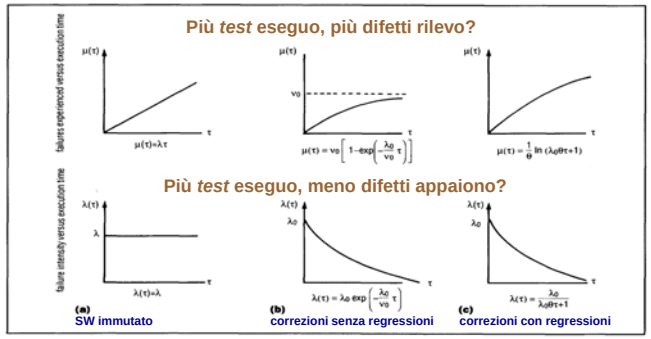


Figure 1. Three useful software-reliability models: **(a)** static, **(b)** basic, and **(c)** logarithmic Poisson. These are shown comparing both failures experienced versus execution time and failure intensity versus execution time.

La risposta di Musa & Ackerman (1989)

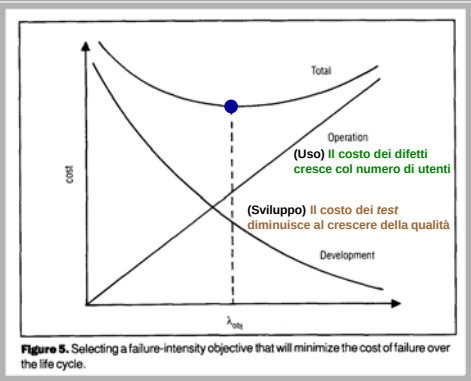


Figure 5. Selecting a failure-intensity objective that will minimize the cost of failure over the life cycle.

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)