Riassunto Testing e Validazione

Luiggi

May 2020

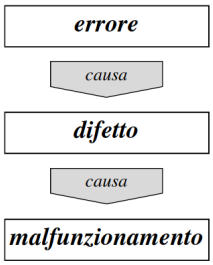
1 Introduzione

Il controllo come soluzione a posteriori, nello specifico il testing, è una tecnica semplice e intuitiva che si adatta molto bene alla natura di un software, poiché esso è fatto per essere eseguito; inoltre, i costi per poter testare un software sono molto contenuti perché nella sua natura, questi è facilmente modificabile e ci si può intervenire in qualsiasi momento. Dunque il testing può essere un ottimo strumento per migliorare la qualità del software.

1.1 Terminologia

La qualità non dipende esclusivamente dall'assenza di problemi ma molte attività inerenti al miglioramento della qualità trattano l'eliminazione dei problemi. Il termine è troppo generico, per cui ci affidiamo ai termini dettagliati dello standard IEEE:

- **Malfunzionamento (o guasto o failure):** funzionamento diverso da quello previsto dalla specifica.
- **Difetto (o anomalia o fault):** causa di un malfunzionamento.
- **Errore (o error):** origine di un difetto.



Un errore (tipicamente umano), dovuto a una distrazione può introdurre un difetto che genera poi un malfunzionamento. Raramente gli errori possono essere causati dagli strumenti. Un errore lascia una traccia, mentre un difetto può essere permanente e ci si può non accorgersi di esso fino a che non genera un malfunzionamento in particolari condizioni: *difetti quiescenti*, che si presentano solo con particolari input o eventi, sono i più difficili da trovare anche con controlli severi.

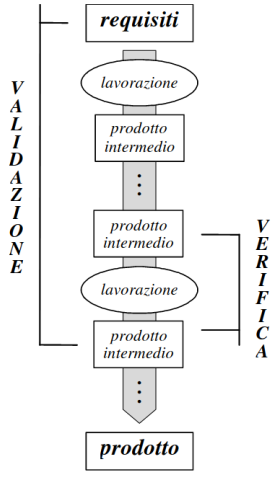
1.2 Controlli interni ed esterni

Controlli interni: a questa categoria appartengono tutte le attività di controllo effettuate dal personale della stessa azienda del processo software. I controlli interni sono mirati e precisi, poiché si hanno tutte le informazioni possibili

a portata di mano, nonché degli stessi sviluppatori, e dunque sono più dettagliati. Per questo e altri motivi dipendenti dalla natura dei controlli interni, se si è fortunati si riescono da subito a trovare soluzioni per i problemi messi in evidenza. **Controlli esterni:** a questa categoria invece appartengono tutte le attività di controllo messe a punto da organizzazioni esterne. Spesso sono effettuati dal committente o da terze parti, affinché sia lui stesso ad avere la responsabilità di condurre i controlli, dunque il fornitore subisce un'attività di controllo e deve fornire al committente i mezzi per farlo. Per motivi di privacy aziendale però, il fornitore non è costretto a dare informazioni troppo specifiche sulle tecnologie usate per lo sviluppo o altro, per cui i controlli esterni sono molto ristretti e prendono il nome di **validazioni**. Un tipico controllo esterno è il *collaudo*.
I controlli finali si dividono inoltre anche in α**-test** e β**-test**:
Alpha-test: Chi esegue i controlli è interno all'organizzazione o divisione o reparto che ha sviluppato il software.
Beta-test: Chi esegue i controlli è invece è sterno. Da non considerare sinonimi Beta-test e *controlli di accettazione*, poiché gli ultimi sono un sottoinsieme di Beta-test, che consistono in un metodo di valutazione dei requisiti. È possibile rilasciare delle Beta-release, ossia versioni incomplete del prodotto, gratuitamente, al fine di valutarne il gradimento oltre che ricevere feedback su eventuali malfunzionamenti (naturalmente hanno un periodo di utilizzo limitato).

2 Verifica e Validazione

Verifica: controllo di qualità delle attività svolte durante una fase dello sviluppo (stiamo realizzando correttamente il prodotto?)
Validazione: controllo di qualità del prodotto rispetto ai requisiti del committente (stiamo realizzando il prodotto corretto?)



Si può dire che la *verifica* è solitamente fatta sui singoli moduli di ciascuna fase per controllare che sia stata fatta bene, mentre la *validazione* è solitamente fatta sul prodotto finito per controllare che questo rispetti i requisiti effettivi. La validazione però può essere anche fatta durante la fase di sviluppo, per esempio può esser richiesto di validare l'architettura per comprendere se i requisiti sono rispettati, evitando preventivamente fraintendimenti. Ricapitolando, verifica e validazione si sovrappongono alle altre fasi del processo di sviluppo e devono ben integrarsi, vanno quindi pianificate a modo per minimizzare i ritardi e gli sprechi di risorse.

2.0.1 Come valutare se un prodotto è pronto per essere rilasciato

Si fa affidamento su alcune misure dette di **dependability (o affidabilità**, quali ad esempio la **disponibilità**, che misura la qualità di un sistema in termini di tempo di esecuzione rispetto a quando non è in esecuzione; il **tempo medio tra i guasti (MTBF)**, che misura la qualità di un sistema in termini di tempo tra un guasto e il successivo; l'**affidabilità** indica la percentuale di operazioni che terminano con successo.

2.1 Controlli statici

Chiamata anche **analisi o verifica statica**, è basata sulla non esecuzione del codice ma sull'analisi di esso.

2.1.1 Desk check

Consiste nel controllo "manuale" del codice, che solitamente viene affidato non ai programmatori ma ad altri addetti a commentare e a ridare il codice ai programmatori per correggerlo. Naturalmente si usano degli strumenti automatizzati per effettuare tali controlli salvo alcuni rari casi, come quelli di ispezione mirata, in cui i controlli si fanno a mano. Sostanzialmente esistono i **walkthrough**, che mirano a cercare errori algoritmici simulando l'esecuzione dei codici; e le **ispezioni**, che invece normalmente tendono a trovare errori tipici categorizzandoli.

2.1.2 Analisi statica supportata da strumenti

Una **prova formale** di correttezza di un programma ideale rappresenta un risultato ottimo valutando tutti i possibili dati in input, ma si usano delle approssimazioni perché nella realtà dei fatti non è una strada percorribile. I metodi sono il **Model Checking, esecuzione simbolica** e **interpretazione astratta**.
Tesi di Dijkstra: Il test di un programma può rilevare la presenza di difetti ma non dimostrarne l'assenza.

2.1.3 Model Checking (controllo sul modello) (FACOLTATIVO)

Per il model checking si ha bisogno di linguaggi sufficientemente espressivi per descrivere sia il comportamento che le proprietà interessanti che devono essere verificate. La **specifica del comportamento** è data da un grafo orientato i quali nodi specificano gli stati di un sistema e gli archi le transizioni da uno stato all'altro. Ogni nodo ha un insieme di proposizioni atomiche per descriverne le proprietà fondamentali. (nell'esempio, a->attesa, m->moneta inserita, c->consegnato caffè, t->consegnato tè). Le **proprietà** esprimono i requisiti del sistema (i.e. nel nostro esempio, è sempre vero che se inseriamo una moneta vengono consegnati o tè o caffè, oppure che non vengono consegnati entrambi assieme) e si esprimono con la logica temporale. Il **model checker** esegue un algoritmo che controlla se le proprietà sono vere nel modello del comportamento del sistema. Data una specifica M e una formula ϕ si verifica se M soddisfa ϕ (algoritmo CMS86).

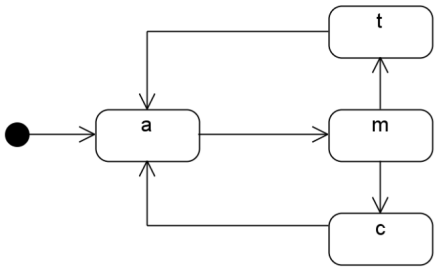


Fig. 2.2. Diagramma a stati di una macchinetta del caffè.

L'algoritmo fornisce anche un controesempio se la formula non è soddisfatta, ed è lineare in tempo e nello spazio (a seconda della formula e degli stati naturalmente). Tuttavia si hanno altri problemi legati al vincolo di avere un modello finito e l'**esplosione dello spazio degli stati**, ossia la crescita esponenziale del numero di processi e delle dimensioni del dominio dei dati in input dovuto alle specifiche parametriche, o ai sistemi descritti da processi paralleli.

3 Controllo dinamico

Il controllo dinamico o test è una tecnica di controllo che ha come obiettivo principale la verifica della correttezza funzionale di un programma o sistema, eseguito in un certo ambiente e con certi dati di input. È anche usato per valutare affidabilità, usabilità ed efficienza. Ci sono 4 aspetti principali di questa tecnica, ovvero: **Progettazione dei test:** un test va preparato e programmato ai fini di addestrare il programma a causare un malfunzionamento, mettendolo quindi in condizioni particolari (specifici dati di input); **Ambiente di test:** un test deve essere eseguito in un ambiente in cui è permessa la corretta esecuzione del programma o sistema da controllare, in particolar modo, se questo è una componente di un sistema più grande, occorre costruire tutto attorno ad esso l'insieme di programmi necessari a simulare il comportamento delle parti man-

canti; **Analisi dei risultati:** l'esecuzione di un test produce dei risultati i quali vanno confrontati con dei risultati di riscontro, di modo da individuare eventuali malfunzionamenti; **Debugging:** consiste nella procedura analitica attraverso la quale si identificano ed eliminano i difetti (bug) che causano i malfunzionamenti, e questo è il vero fine del miglioramento della qualità. Il controllo dinamico quindi sostanzialmente è una procedura ostica ma necessaria, soprattutto per testare l'intero sistema in quanto l'analisi statica sarebbe improponibile se non su alcuni moduli; inoltre la qualità in termini di efficienza può essere stimata solamente testando l'intero software in esecuzione.

3.1 Criteri pratici di (in)adeguatezza

Essi sono dei criteri che identificano inadeguatezza nei casi di test, ad esempio se avessimo un programma con n istruzioni e i casi di test ne testano k<n. Un criterio di adeguatezza consiste in un insieme di **test obligation**. Una test obligation è una specifica di casi di test ritenuti importanti per il testing, come ad esempio un input formato da due parole e uno formato da tre parole, sono due tra i tanti test che soddisfano la specifica.

3.1.1 Come definire le test obligation

Ci sono differenti strategie per definire delle test obligation funzionali, per esempio: dalla struttura del codice (**white-box**), in cui si mira a esercitare il codice indipendentemente dalle funzionalità poiché si conosce come è scritto (per esempio passare per un ciclo almeno una volta); dalla specifica software (**black-box**), in cui si mira a cercare malfunzionamenti relativi alle funzionalità poichè non si conosce la struttura del codice ma solo gli output; dal modello del programma (o sistema), quindi dalla specifica stessa del programma; da fault ipotetici, ovvero si prova a cercare ipotetici bug comuni.

3.2 Criteri di adeguatezza

Anche un criterio di adeguatezza consiste di un insieme di test obligation; un insieme di test soddisfa un criterio di adeguatezza solo se tutti i test hanno successo e se ogni test obligation è soddisfatta da almeno un caso di test.

3.3 Altre fasi del testing

Batteria di prova (o test suite): insieme di casi di prova.
Procedura di prova: le procedure per eseguire, analizzare, registrare e valutare i risultati di una batteria di prove.

3.3.1 Conduzione di una prova

Per poter condurre una prova, occorre anzitutto definire un obiettivo della prova; dopodiché la prova va progettata nella scelta e nella definizione dei casi di prova.

Poi si costruisce un ambiente di prova, ovvero driver e stub, ambienti di controllo e strumenti per la registrazione dei dati. Poi la prova va eseguita, e questa parte può richiedere del tempo, successivamente poi si procede con l'analisi dei risultati per controllare i riscontri di eventuali malfunzionamenti e si valuta infine la prova.
Test scaffolding: si tratta di codice aggiuntivo necessario per eseguire un test, e può includere: dei **driver** di test, che sostituiscono un programma principale; dei test harness, che sostituiscono parti dell'ambiente di distribuzione; dei stub, che sostituiscono invece funzionalità chiamate o usate dal sistema di prova (**mock**); strumenti per l'esecuzione del test e la registrazione dei risultati.

3.4 Metodi black-box per generare valori di input

La strategia di massima è quella di separare le varie funzionalità basandosi sui casi d'uso per costruire dei casi di test per ogni funzionalità. Quindi si individuano dei valori da testare per ogni tipo di parametro e si effettua del testing combinatorio per ridurre le possibili combinazioni. Con il **metodo random** si genera un insieme di valori a caso a costo zero, non ripetibile (random) e difficilmente considera i casi limite.

3.4.1 Metodo statistico

In questo metodo i casi di test sono selezionati in base alla distribuzione di probabilità dei dati in ingresso del programma, quindi si considerano i valori di input più probabili e quindi è un processo facilmente automatizzabile. Tuttavia è piuttosto oneroso calcolare il risultato atteso e non sempre corrisponde alle effettive condizioni di utilizzo del software.

3.4.2 Partizione dei dati in ingresso

Il dominio dei dati in ingresso è suddiviso in classi di equivalenza, ove due input appartengono alla stessa classe se producono uno stesso comportamento nel programma. Il criterio è basato su un'affermazione generalmente plausibile ma non sempre vera, e inoltre è economicamente conveniente solo per i programmi per cui il numero di possibili comportamenti è sensibilmente inferiore alle possibili configurazioni d'ingresso.

3.4.3 Valori di frontiera

Anche questo si basa su una partizione dei dati in ingresso, e i dati di input sono gli estremi di ogni classe di equivalenza. Questo criterio cerca di controllare il comportamento del software nei valori limite nell'ingegneria classica, anche se nel software ciò non è applicabile poiché i valori limite sono trattati in modi particolari.
Casi non validi: si tratta di dati in input non validi che devono generare errori.
Test basato su catalogo: un'azienda nel tempo magari si è costruita un

catalogo costituito da tipici test dovuti all'esperienza, e potrebbe essere utile 'riusare' gli stessi criteri usati su altri progetti in passato, magari simili.

3.5 Testing combinatorio

È una tecnica da applicare in genere quando il numero di parametri in input cresce.

3.5.1 Esplosione combinatoria

Se vi sono troppi dati di input, se si prende il prodotto cartesiano dei test individuati per tutti quegli input si rischia di avere un numero ingestibile, quindi occorre trovare una strategia per generare casi di test significativi in modo sistematico. Due tecniche comunemente usate per fare ciò sono i **vincoli** e il **pairwise testing**.

3.5.2 Vincoli (constraints)

Servono fondamentalmente per ridurre il numero di possibili combinazioni di errori, proprietà o singoli. Immaginiamo di avere 5 parametri in input
$< x_1, x_2, x_3, x_4, x_5 >$ e ipotizziamo che il dominio di x_1 e x_2 è ripartibile in 8 classi di cui una di valori non validi, che generano errore; x_3 e x_5 sono ripartibili in 4 classi di cui una di valori non validi e il dominio di x_4 è ripartibile in 7 classi di cui una che genera errore. Se prendessimo un rappresentante per ogni classe, avremmo $8 * 8 * 4 * 7 * 4 = 7168$ casi di test. Se invece, per quanto riguarda i valori che generano errori, per ciascuna classe ne prendessimo 1 solo, avremmo: $5 + 7 * 7 * 3 * 6 * 3 = 2651$ casi, che sono molti meno.
Imponendo altri vincoli, sulla proprietà basandosi ad esempio sugli 'if' positivi e negativi, oppure prendendo un singolo valore da testare per uno o più parametri, il numero dei casi di test si può ridurre ulteriormente.

3.5.3 Pairwise testing

Nel caso in cui il dominio non contenga in sé questi vincoli, è preferibile optare per la generazione di tutte le possibili combinazioni solo per i sottoinsiemi di k variabili con k<n (nel caso di pairwise k=2). L'idea è quindi quella di generare le combinazioni per tutte le *possibili* coppie di variabili. La generazione di combinazioni che in maniera efficiente coprano tutte le coppie è impossibile se il numero di parametri è alto ma esistono delle tecniche euristiche.

3.6 Criteri strutturali

Consistono in criteri per l'individuazione dei casi di input basati sulla struttura del codice. Permettono di individuare eventuali malfunzionamenti che non sono apparsi durante il test a scatola chiusa.

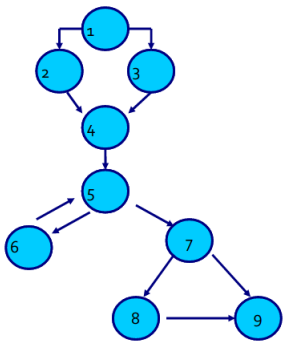
3.6.1 Elementi di un flusso di controllo

Un programma non è testato adeguatamente se alcuni suoi elementi non vengono mai esercitati dai test. I criteri strutturali di progettazione di casi di test (**control flow testing**) sono definiti per classi particolari di elementi e richiedono che i test esercitino *tutti* quegli elementi. Questi possono essere **comandi, branches (decisioni), condizioni** o **cammini**.

3.6.2 Grafi di flusso

Un grafo di flusso definisce la struttura del codice identificandone le parti e le loro correlazioni. Si ottiene a partire dal codice.

```
double eleva(int x, int y) {
   1. if (y<0)
   2.    pow = 0-y;
   3.    else pow = y;
   4. z = 1.0;
   5. while (pow!=0)
   6.    { z = z*x; pow = pow-1 }
   7. if (y<0)
   8.    z = 1.0 / z;
   9. return(z);
}
```



Per poter coprire tutti i comandi, occorre scegliere un opportuno insieme di valori, ad esempio:
$\{(x = 2, y = 2), (x = 0, y = 0)\}$ non esercita tutti i comandi.
$\{(x = -2, y = 3), (x = 4, y = 0), (x = 0, y = -5)\}$ li esercita tutti. Definiamo inoltre la **misura di copertura** come il rapporto tra il numero di comandi esercitati e il numero di comandi totali. In questo esempio, per avere una copertura totale occorrono almeno due casi di test: uno con $y < 0$ e uno con $y \geq 0$. Tuttavia, non sempre vale la pena cercare a tutti i costi una copertura totale con un insieme minimale.

3.6.3 Condizioni composte

Se si considera il codice

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

il test $\{x = 0, y = 0\}$ e $\{x = 0, y = 1\}$ garantisce la piena copertura delle decisioni ma non esercita tutti i valori di verità della condizione. Quindi il test ottimale sarebbe $\{x = 0, y = 0\}, \{x = 2, y = 1\}, \{x = 1, y = 1\}$ poiché esercita tutti i valori di verità e tutte le decisioni.

3.6.4 Copertura di condizioni semplici

Un insieme di test T per un programma P copre tutte le condizioni semplici di P se, per ogni condizione semplice CS in P, T contiene un test in cui CS vale *true* e un test in cui CS vale *false*. Nella copertura di condizioni multiple invece bisognerebbe testare tutte le possibili combinazioni, anche se le semantiche di alcuni linguaggi potrebbero ridurne il numero (vedi linguaggi lazy).

3.6.5 Copertura dei cammini

Questa copertura richiede di percorrere tutti i cammini possibili, anche se in presenza di cicli questi sono potenzialmente infiniti. Pertanto, un ciclo va esercitato 0 volte, 1 volta e più volte.

3.6.6 Riepilogo

I criteri funzionali sono indipendenti dal codice, mentre quelli strutturali si prestano bene alla valutazione della copertura.

3.7 Fault based testing

Questo tipo di test si prepone di ipotizzare dei difetti tipici, e la più nota tecnica per farlo è il **test mutazionale**, ovvero si iniettano difetti mutando il codice.

```
\\originale
int foo(int x, int y)
{ if(x <= y)
        return x+y;
  else return x*y;}
```

```
\\versione modificata (mutante)
int foo(int x, int y)
{ if(x < y)
        return x+y;
  else return x*y;}
```

Se prendiamo una batteria di test come $\{< (0,0), 0 >, < (2,3), 5 >, < (4,3), 12 >\}$ il test non fallisce in nessuna versione, quindi la batteria è poco efficace e va riprogettata (il mutante non viene ucciso).

```
\\originale con un difetto
int foo(int x, int y)
{ if(x < y)
        return x+y;
  else return x*y;}
```

```
\\mutante in cui inietto un difetto (M=1
int foo(int x, int y)
{ if(x < y)
        return 3;
  else return x*y;}
```

Se ora consideriamo come batteria di test $\{< (0,0), 0 >, < (2,3), 5 >, < (4,3), 12 >\}$, essa permette di individuare 0 difetti nell'originale mentre 2 difetti nel mutante. Uno dei 2 difetti del mutante lo abbiamo iniettato noi, dunque il totale è $tot = (N * M)/N_1 = 2$ dove N è il numero di difetti, M il numero dei difetti iniettati e N_1 pure. Invertendo troviamo che $tot - N_1 = 1$, quindi c'era un difetto nell'originale che non avevo trovato. Il **test mutazionale** in pratica quindi serve per verificare che una batteria di test sia effettivamente valida e se è il caso di cambiarla o di aggiungervi metodi più sofisticati. Una **mutazione** consiste in un mutamento sintattico, come per esempio cambiare $(i < 0)$ in $(i \leq 0)$. Un mutante viene **ucciso** se esso fallisce almeno in un caso di test, e

l'**efficacia di un test** si misura sulla quantità di mutanti uccisi. Questa tecnica si applica naturalmente anche insieme ad altri criteri di test.

3.7.1 Ipotesi del programmatore competente

I difetti reali sono **piccole variazioni sintattiche** del programma corretto, quindi i mutanti sono modelli ragionevoli di programmi con difetti. Se mettiamo infine insieme tutte le ipotesi otteniamo che i test che trovano semplici difetti trovano anche quelli più complessi e che una test suite che uccide tutti i mutanti è capace di trovare anche i difetti reali nel programma.

3.7.2 Esempi di mutanti

I tipici esempi di mutanti sono riassumibili in:

- **crp**: sostituzione di costante per costante (da $x < 5$ a $x < 12$).
- **ror**: sostituzione operatore relazionale (da $x \leq 5$ a $x < 5$).
- **vie**: eliminazione inizializzazione di una variabile (da $x = 5$ a x).
- **lrc**: sostituzione di un operatore logico (da && a ||).
- **abs**: inserimento di un valore assoluto (da x a $|x|$).

3.7.3 Mutanti validi e utili

Un mutante si dice **invalido** se è sintatticamente sbagliato, altrimenti è **valido**. Un mutante è inoltre **utile** se è valido e difficile da distinguere dal programma originale. Naturalmente non è facile trovare mutanti validi e utili, dipende dal linguaggio con cui si ha a che fare. La strategia dei mutanti ha diversi obiettivi solitamente, quali il favorire la scoperta di malfunzionamenti ipotizzati, verificare l'efficacia dei test e cercare indicazioni circa la locazione dei difetti. Ovviamente per motivi di costi e di tempi si tenta di ridurre al minimo il numero possibile di mutanti da realizzare, essendo un lavoro tutt'altro che semplice.

3.8 L'oracolo e l'individuazione degli output attesi

Questo esiste perché non avrebbe senso produrre tanti casi di test ma poi l'output va calcolato a mano. Quindi si tenta di trovare l'output atteso mediante i risultati ricavati dalle specifiche, formali o eseguibili, oppure mediante l'inversione delle funzioni, per esempio partire dall'output per trovare l'input. Si possono utilizzare, volendo, anche le vecchie versioni dello stesso codice per le funzionalità non modificate, oppure si possono semplificare ad esempio i dati in input per forzare dei risultati più semplici da calcolare per confrontarli con l'output vero e proprio. Infine, di solito ci si accontenta dei risultati plausibili, condizionati da vincoli su entrate ed uscite o con invarianti sulle uscite.