Laurea in Informatica e in Informatica applicata
Dipartimento di Informatica
Università di Pisa

Il controllo del software:

verifica e validazione

Giovanni A. Cignoni
Carlo Montangero
Laura Semini

Dispensa per il Corso di Ingegneria del Software

2010

Il contenuto di questa dispensa è una rielaborazione del materiale già usato per i capitoli 2 e 3 de:
Il test e la qualità del software, di Giovanni A. Cignoni e Piero De Risi,
Ed. Il Sole 24 Ore, Milano, 1998.
Il libro è attualmente fuori commercio.

Indice

1 Introduzione 1
1.1 Terminologia: malfunzionamento, difetto, errore 1
1.2 Controlli interni ed esterni 2
2 Verifica e Validazione 4
2.1 Controlli statici 5
2.2 Controlli dinamici 9
2.3 Livelli di test 10
2.4 Il V-Modell 14
3 Il test: metodi 15
3.1 Progettazione dei test 16
3.2 Criteri funzionali (black-box) 16
3.3 Criteri strutturali (white box) 19
3.4 Strategie di test 22
3.5 L'oracolo 25
3.6 Analisi e terminazione dei test 27
Glossario 29
Bibliografia 35

1 Introduzione

Queste note sono dedicate al controllo del software. Con controllo intendiamo le attività inserite alla fine del processo di sviluppo, volte a garantire un prefissato livello di qualità del prodotto. Nella prassi i termini verifica, validazione e test sono usati, con abuso di notazione, come sinonimi di controllo.
Queste note descrivono:

- i concetti di verifica e validazione;
- la differenza tra controlli statici e dinamici (test)
- le principali tecniche di controllo statico;
- i metodi di test, fornendo al lettore gli strumenti concettuali necessari per impostare un piano di test: la progettazione dei test è il principale tema affrontato;
- i livelli di test: dei moduli, integrazione, controllo di sistema e infine i controlli di accettazione;
- poiché il test ha il limite di mostrare l'esistenza di un difetto senza però indicare esattamente dove esso si trovala dispensa si conclude con il tema del debugging, affrontando cioè il problema dell'individuazione e dell'eliminazione dei difetti.

Il controllo come soluzione a posteriori per il conseguimento della qualità del prodotto è una tecnica semplice e intuitiva, che l'ingegneria del software ha ereditato dal patrimonio comune dei metodi industriali. Il *test* in particolare, ovvero il controllo sul banco di prova, ben si adatta alla natura del software, un prodotto destinato appunto a essere eseguito. Inoltre, la particolare caratteristica del software di poter sempre intervenire sul prodotto e, a costi molto contenuti, correggerne i difetti, fa del test un effettivo strumento per il miglioramento della qualità.

A completamento delle note un glossario dei termini usati nel testo, che contribuisce all'uso di queste note come strumento di riferimento oltre che come opera introduttiva al controllo del software. Nella bibliografia citiamo, tra gli altri, alcuni articoli e testi che trattano il tema del controllo del software [CD98, ZHM97, PM07].

1.1 Terminologia: malfunzionamento, difetto, errore

La qualità di un prodotto software non è limitata all'assenza di problemi, tuttavia la maggior parte delle attività connesse al controllo di qualità del software hanno come obiettivo la scoperta e l'eliminazione di ogni possibile problema. Il termine "problema" che abbiamo – volutamente – usato qui è assai generico, potremmo definirlo come qualsiasi caratteristica o comportamento che abbia come conseguenza un discostamento dai requisiti o dalla specifica del programma.
Per avere un maggior livello di dettaglio e per evitare che uno stesso termine sia usato in letteratura con significati diversi, il glossario IEEE [IEEE90] propone le seguenti definizioni (fra parentesi diamo il termine originale inglese):

- un ***malfunzionamento*** o *guasto* (*failure*) è un funzionamento di un programma diverso da quanto previsto dalla sua specifica;
- un ***difetto*** o *anomalia* (*fault*) è la causa di un malfunzionamento;

- un ***errore*** (*error*) è l'origine di un difetto.

Lo standard, proposto da IEEE e recepito concordemente nella quasi totalità dei testi di ingegneria del software, oltre a precisare il significato dei termini stabilisce fra essi un ordinamento basato sul rapporto di causa-effetto rappresentato schematicamente in fig. 2.2.
Un errore ha, nella maggior parte dei casi, radici umane: una distrazione in fase di codifica può introdurre un difetto nel programma che genera malfunzionamenti.

In alcuni casi – per fortuna rari – l'errore può risalire agli strumenti di sviluppo o nel modo con cui sono utilizzati. Ad esempio, un compilatore difettoso o usato maldestramente può generare un codice malfunzionante pur partendo da un sorgente a tutti gli effetti corretto.
Particolare è la caratterizzazione temporale dei tre concetti: un errore è un evento puntuale che lascia una traccia; un difetto, finché non viene corretto, è permanente ma può esibire un malfunzionamento solo in determinate e a volte particolarissime condizioni. A questo proposito, si parla di *difetti quiescenti*, che si annidano nei rami più raramente percorsi dei programmi o che si manifestano solo in presenza di particolari configurazioni dei dati di ingresso. I difetti quiescienti sono i più perniciosi: possono sfuggire ai controlli più severi e manifestarsi quando il software è in pieno regime d'uso, danneggiando l'utente e squalificando l'immagine del fornitore.

Si noti, infine, che lo scopo di un controllo è la ricerca di una non conformità, trovarla è un successo dell'attività di controllo.

1.2 Controlli interni ed esterni

Con l'attributo **interno** si identifica ogni attività di controllo che proviene dalla stessa organizzazione impegnata nel processo di sviluppo software. Il personale incaricato di eseguire i controlli appartiene cioè alla stessa azienda, o divisione, o squadra che lavora a un progetto software.
L'appartenenza al progetto di chi esegue i controlli rende i controlli interni estremamente mirati, con obiettivi precisi rispetto alle caratteristiche del prodotto in via di sviluppo, condotti a un livello di dettaglio molto fine. Caratteristica del tutto naturale, data la provenienza del personale che è incaricato dei controlli e delle condizioni privilegiate in cui si trova ad agire: ha sicuramente a disposizione tutte le informazioni possibili, è investito dell'autorità necessaria per chiedere delucidazioni al personale più direttamente coinvolto nello sviluppo, ha, poiché spesso coinvolto nel progetto fin dal suo inizio, la competenza per dare la migliore direzione alla strategia dei controlli e, nei casi più fortunati, per suggerire soluzioni ai problemi che l'attività di controllo ha messo in evidenza.
Un'attività di controllo è detta **esterna** quando è condotta da personale estraneo all'organizzazione che ha in carico lo sviluppo di un prodotto software. I controlli esterni sono tipici delle situazioni in cui il prodotto è realizzato su commissione ed è evidente la contrapposizione delle due parti contrattuali del *committente* e del *fornitore*.
In questi casi i controlli esterni si sovrappongono ai controlli interni effettuati per proprio conto dal fornitore e hanno l'obiettivo di costituire, a vantaggio del committente, un'ulteriore garanzia circa la qualità del prodotto. La natura del controllo è necessariamente esterna perché è nell'interesse del committente che sia lui stesso, o, come più spesso accade, una terza parte che agisce per suo conto, ad avere il vantaggio e la responsabilità dell'organizzazione e della conduzione dei controlli. Sul fronte opposto, il fornitore "subisce" l'attività di controllo ed è

tenuto, nei limiti stabiliti dal contratto, a fornire tutto il supporto necessario affinché il personale esterno incaricato dei controlli sia in grado di eseguirli materialmente.

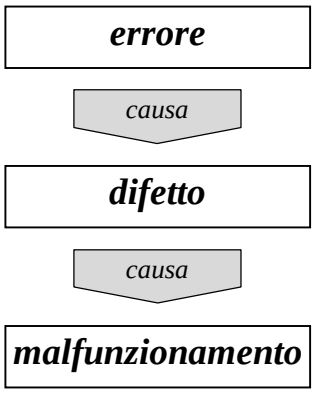

Fig. 1.1 Errori, difetti e malfunzionamenti.

Il limite dei controlli esterni sta appunto nelle ristrette possibilità di indagine del personale addetto alle operazioni di controllo: il fornitore, in sede di contratto, ha interesse a limitare il più possibile la visibilità del processo di sviluppo, anche per il suo innegabile diritto a mantenere riservate le tecnologie adottate e le proprie politiche organizzative. Per questo motivo, i controlli esterni assumono in generale il connotato di validazioni, essendo i requisiti il più comune elemento di riscontro in possesso del committente. Un tipico controllo esterno è il *collaudo*.

Una terminologia molto diffusa divide i controlli finali in *α-test* e *β-test*. Fatte salve le considerazioni circa l'uso del termine test, i due termini distinguono i controlli rispetto al soggetto che li esegue. Quando chi esegue i controlli appartiene alla stessa organizzazione – o divisione, o reparto, o squadra, la distinzione è relativa – che ha sviluppato il software si parla di α-test, altrimenti si deve parlare di β-test.
I controlli di sistema possono essere svolti sia come α-test che come β-test: nel secondo caso il prodotto, prima di essere rilasciato definitivamente, viene fatto circolare fra un insieme ristretto di gruppi di prova, "vicini" al fornitore, che si limitano a usare normalmente il software segnalando eventuali malfunzionamenti. I controlli di accettazione, poiché eseguiti dal committente, rientrano nella definizione di β-test, ma non è corretto considerare i termini "controlli di accettazione" e "β-test" sinonimi.
A conferma della moderna tendenza verso cicli di vita evolutivi è da evidenziare l'uso intensivo del β-test come metodo di valutazione dei requisiti di un prodotto. Nel caso di software commerciale, è molto comune la distribuzione di versioni incomplete del software, dette *β-release*, al preciso scopo di valutarne il gradimento rispetto alle esigenze di un'utenza che, altrimenti, non sarebbe direttamente raggiungibile. Per accrescere la base di valutazione, è normale distribuire le β-release gratuitamente a qualsiasi utente ne faccia richiesta. In questo caso, il mercato del prodotto finale è protetto introducendo nelle β-release dei meccanismi che ne consentono l'uso solo per un limitato periodo di tempo.

2 Verifica e Validazione

Con i termini di verifica e validazione si distinguono le attività di controllo in base alla loro portata rispetto ai diversi tempi del processo di sviluppo, individuando con il termine verifica le attività di controllo che sono caratterizzate dall'essere circoscritte a una sola fase. Per dirla con Bohem, verifica e validazione rispondono rispettivamente alle domande:

- stiamo realizzando correttamente il prodotto?
- stiamo realizzando il prodotto corretto?

Obiettivo della verifica è il controllo di qualità delle attività svolte durante una fase dello sviluppo; obiettivo della validazione è il controllo di qualità del prodotto rispetto ai requisiti del committente.

Come illustrato in fig. 2.1, è possibile interpretare il processo di sviluppo come una successione di lavorazioni di prodotti intermedi – la più stringente delle visioni del ciclo di vita a cascata. In questa prospettiva, un'attività di *verifica* è il controllo che il prodotto ottenuto al termine di una fase sia congruente con il semilavorato avuto come punto di partenza di quella fase. Per esempio, nella realizzazione di un modulo, è una tipica verifica il controllo che le specifiche del modulo siano state rispettate sia come interfaccia che come funzionalità.
La *validazione* è un'attività di controllo mirata a confrontare il risultato di una fase del processo di sviluppo con i requisiti del prodotto – tipicamente con quanto stabilito dal contratto o, meglio, dal documento di analisi dei requisiti. Un comune esempio di validazione è il controllo che il prodotto finito abbia funzionalità e prestazioni conformi con quelle stabilite all'inizio del processo di sviluppo.

La validazione è un'attività normalmente prevista sul prodotto finito. Tuttavia, limitandosi ad aspetti particolari, è comunque possibile effettuare delle operazioni di validazione anche durante il processo di sviluppo. Ad esempio, l'architettura può essere validata con i requisiti; in questo caso la validazione è una preventiva assicurazione contro possibili errori di interpretazione dei requisiti.

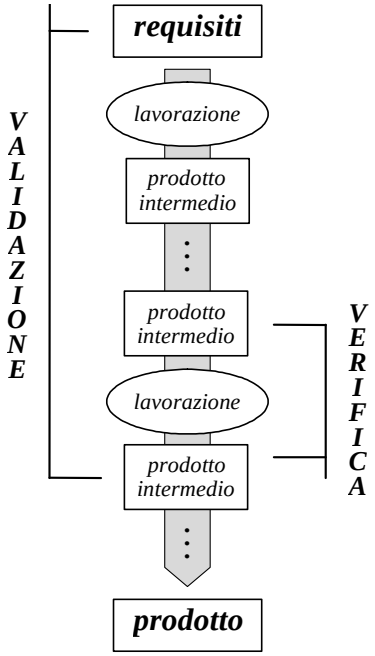

Fig. 2.1. Verifica e validazione rispetto al ciclo di vita.

Verifica e validazione sono attività che si sovrappongono a quelle tradizionali e concretamente produttive del processo di sviluppo e con le quali devono integrarsi nel più efficace possibile dei modi. La scoperta di un errore durante lo sviluppo è sicuramente un guadagno per la qualità del prodotto, ma è anche un evento la cui gestione va prevista.

L'organizzazione del processo di sviluppo deve prevedere e pianificare ogni attività di verifica e validazione, mediando fra la parallelizzazione e la serializzazione delle attività di sviluppo e di quelle di controllo: nel primo caso si tende a minimizzare i ritardi, nel secondo si cerca di risparmiare sulle risorse e di evitare gli sprechi.

2.1 Controlli statici

Il controllo statico di un programma, spesso chiamato anche *verifica statica* o *analisi*, è basato sulla non esecuzione del programma. Diversi sono i motivi per rinunciare alla naturale tendenza a lanciare un programma per sapere se questi funziona correttamente. Un primo motivo è che eseguire un programma, specialmente se questo è in realtà una procedura, cioè un pezzo di codice scollegato da ogni contesto eseguibile, ha un costo che, in molti casi può far preferire controlli basati sulla sola analisi del codice.

2.1.1 Desk check

Il modo più elementare di effettuare un controllo statico è basato sull'analisi informale del codice, condotta "a mano" – in inglese *desk check* – o mediante l'ausilio di strumenti automatici. Le tecniche più comuni per effettuare questo tipo di controlli statici sono **l'ispezione** e il **walkthrough**. Entrambi hanno lo scopo di trovare e denunciare dei difetti senza però che il revisore si sostituisca al progettista o al programmatore, a cui comunque rimane il compito di scoprire l'errore e correggere il codice. Sia ispezione che walkthrough

possono inoltre essere organizzati all'interno del processo software secondo i medesimi principî: il gruppo di revisione e quello di codifica dovrebbero essere diversi, l'attività di verifica si dovrebbe concludere con un rapporto da includere nella documentazione del processo di sviluppo.
La principale differenza fra ispezione e walkthrough sta invece nel tipo di errori che la tecnica si prefigge di scoprire.

- Il walkthrough è inteso come un'esecuzione simulata del codice e porta generalmente a scoprire difetti originati da errori algoritmici;
- L'ispezione ha invece, di norma, obiettivi ristretti e prestabiliti: poiché i difetti sono raggruppabili in categorie – ben note ai programmatori – è plausibile che un controllo ottenga migliori risultati se concentrato su un solo tipo di difetti.

Il controllo statico è tendenzialmente identificato come un'attività manuale. In realtà molti controlli statici possono essere eseguiti con l'ausilio di analizzatori di codice automatici molto più efficienti e rigorosi di qualsiasi professionista. Perciò, in base alle caratteristiche del linguaggio di programmazione usato, si dovrà ricorrere a controlli manuali – ad esempio per mezzo di ispezioni mirate – solo per individuare le categorie di difetti per cui non sono disponibili strumenti automatici.

2.1.2 Analisi statica supportata da strumenti

Oltre al desk check, è possibile eseguire una verifica statica usando opportuni strumenti. Da un punto di vista teorico, avere un **prova formale** della correttezza di un programma, ottenuta a partire dall'analisi del codice, corrisponde all'esecuzione su tutti i possibili insiemi di dati d'ingresso. Quindi, la prova formale rappresenta – in teoria – un risultato ottimo, specialmente se paragonato con le risorse necessarie a un reale controllo esaustivo. Purtroppo questa strada non è in generale percorribile, per cui sono state sviluppate teorie di analisi statica che approssimano la prova formale "ideale", rispondendo a diverse esigenze. Elenchiamo qui le più note.

2.1.3 Model checking (controllo sul modello)

Il model checking è nato nel 1980, per verificare i circuiti hardware. Il circuito hardware viene modellato con una macchina a stati finiti che viene verificata automaticamente rispetto ad un insieme di requisiti. Con questa tecnica si sono trovati errori in circuiti già progettati (uno dei più noti in circuiti della Texas Instrument già in commercio). Successivamente il model checking è stato applicato anche a sistemi software. Per il model checking devono essere definiti dei linguaggi sufficientemente espressivi per descrivere sia il comportamento del sistema (il "modello") sia le proprietà interessanti che devono essere verificate.

La **specifica del comportamento** del sistema è normalmente data come un sistema di transizioni, cioè un grafo orientato formato da nodi e archi (per esempio una macchina a stati finiti). I nodi rappresentano gli stati di un sistema, gli archi rappresentano le transizioni da uno stato ad un altro. Un insieme di proposizioni atomiche è associato ad ogni nodo per descrivere le proprietà fondamentali che caratterizzano un punto di esecuzione. Il diagramma di Figura 2.2 descrive una macchinetta distributrice di tè e caffè: inizialmente in stato di attesa (a), dopo l'inserimento di una moneta si passa a uno stato "moneta inserita" (m). Infine viene consegnato un caffè (c) o un tè (t).

Le **proprietà** esprimono i requisiti del sistema. Nel nostro esempio alcune proprietà sono:

1. è sempre vero che se è stata inserita una moneta, allora in tutti i cammini del modello si incontra prima o poi uno stato in cui viene consegnato un tè o uno stato in cui viene consegnato un caffè;
2. non vengono mai consegnati allo stesso tempo sia il tè che il caffè;
3. vediamo anche una proprietà falsa: è sempre vero che se è stata inserita una moneta, allora in tutti i cammini si incontra prima o poi uno stato in cui viene consegnato un tè.

Il linguaggio usato per esprimerle formalmente è la logica temporale, un'estensione della logica classica adatta a descrivere il modello dinamico di un sistema. Per completezza, forniamo le formule che esprimono le proprietà sopra elencate: 1. $AG\ [m \rightarrow AF\ (c\ OR\ t)]$, 2. $AG\ \sim(\ c\ AND\ t)$, 3. $AG\ [m \rightarrow AF\ t]$.

Il **model checker** esegue un algoritmo che controlla se le proprietà sono vere nel modello del comportamento del sistema. Formalmente, data una specifica M e una formula ϕ, si verifica se M soddisfa ϕ (M $\models \phi$), cioè se M è un modello per ϕ. Un algoritmo che risolve il problema del model checking (per la logica CTL) è stato definito da Clarke, Emerson e Sisla [CMS86].

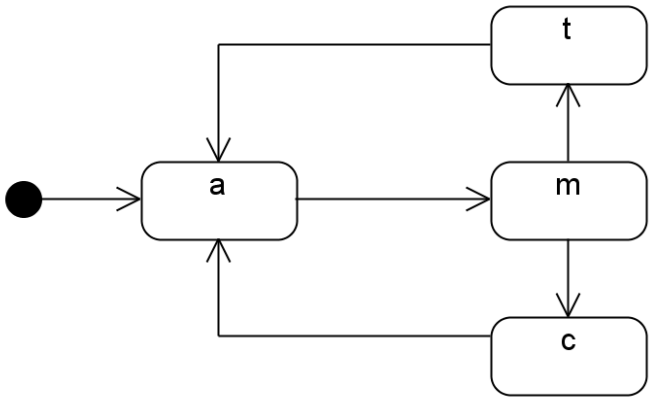

Fig. 2.2. Diagramma a stati di una macchinetta del caffè.

L'algoritmo permette inoltre, in caso di risultato negativo, di trovarne il "perché", visto che è possibile rintracciare gli stati che contribuiscono al fallimento della verifica; in generale, gli algoritmi di model-checking possono fornire un "controesempio", evidenziando ad esempio un cammino nel modello che non verifica la (sotto)formula. Nel nostro caso, il tentativo di verifica della proprietà 3 fallisce restituendo il controesempio *a.m.c*. L'algoritmo è lineare in termini della grandezza della formula (usualmente abbastanza piccola) e lineare in termini dello spazio degli stati. Le vere limitazioni riguardano:

1. Il vincolo ad avere un modello finito.
2. Il problema, noto come **esplosione dello spazio degli stati** legato al numero degli stati del modello, che in caso di sistemi descritti da processi paralleli, o in caso di specifiche parametriche, cresce in modo esponenziale col numero dei processi e sulle dimensioni del dominio dei dati di input. Ciò limita fortemente la capacità degli algoritmi di model-checking a lavorare su esempi di sistemi "reali". Recentemente sono state sviluppate, e applicate con successo a vari model checkers, alcune tecniche per ridurre il fenomeno dell'esplosione degli stati.

2.1.4 Esecuzione simbolica

Questa tecnica rappresenta una sintesi fra i metodi formali di controllo statico e il controllo dinamico. L'esecuzione di un programma viene simulata: si rappresentano insiemi di possibili dati d'ingresso con simboli algebrici, piuttosto che con valori numerici reali. L'esecuzione delle istruzioni del programma, simulata attraverso la manipolazione delle espressioni algebriche d'ingresso, produce rappresentazioni simboliche degli stati possibili per il programma in esame. Si generano così dei modelli matematici che esplicitano l'effettivo legame funzionale fra ingressi e uscite, da confrontare con quello specificato per il programma. Il vantaggio sta nel non dover considerare le combinazioni possibili dei dati d'ingresso. I limiti sono invece legati alla scarsezza di strumenti automatici e alla loro inadeguatezza quando il software da controllare raggiunge dimensioni industriali. Come esempio si consideri la seguente funzione, scritta per calcolare il valore assoluto del suo parametro:

```
public static int abs (int i){        //L1
int r;
if (i<0)
    r = -i;                           //L2
else
    r = i;                            //L3
                                      //L4
return r;
}
```

L'esecuzione simbolica simula l'esecuzione, associando a ogni stato un'espressione che ne descrive le proprietà, in funzione dei valori assunti dalle variabili. Nella tabella seguente mostriamo l'evoluzione di questa descrizione dello stato, *dopo* l'esecuzione dei comandi etichettati nel programma dato sopra. Il simbolo $\imath$ (iota) rappresenta il valore iniziale della variabile `i`, ossia l'argomento del metodo. I simboli & and | indicano rispettivamente congiunzione e disgiunzione logica.

passi	**stato**	**note**
L1	$i = \imath$	è un intero
L2	$i = \imath$ & ($\imath < 0$ & $r = -\imath$)	
L3	$i = \imath$ & ($0 \leq \imath$ & $r = \imath$)	
L4	$i = \imath$ & (($\imath < 0$ & $r = -\imath$) | ($0 \leq \imath$ & $r = \imath$))	I rami del condizionale si ricongiungono

Dal confronto dell'ultima espressione, che definisce il valore restituito dal metodo, con la specifica del metodo, si può decidere se il codice è corretto o meno. L'uso di dimostratori automatici (therem prover) per queste decisioni, che renderebbero la tecnica utilizzabile su larga scala, è tuttora oggetto di ricerca

Esercizio. Dire se il metodo `abs` è corretto, rispetto alla specifica

Pre: $-2^{-31} < \imath < 2^{31}$

Post: $(\imath < 0$ & abs($\imath$) = $-\imath) \mid (0 \leq \imath$ & abs($\imath$) = $\imath)$.

Perchè la precondizione non può essere $-2^{-31} \leq \iota < 2^{31}$?

2.1.5 Analisi statica in compilazione

I compilatori effettuano un'analisi statica del codice per verificare che un programma soddisfi particolari caratteristiche di correttezza statica, per poter generare il codice oggetto. Le informazioni e le anomalie che può rilevare un compilatore dipendono dalle caratteristiche del linguaggio. Tipiche anomalie identificabili sono: nomi di identificatori non dichiarati, incoerenza tra tipi di dati coinvolti in una istruzione, incoerenza tra parametri formali ed effettivi in chiamate a subroutine, codice non raggiungibile dal flusso di controllo.

2.2 Controlli dinamici

Il controllo dinamico, o *test*, richiede l'esecuzione di un programma in un ambiente e con dati di ingresso controllati, la registrazione di tutti i dati riguardanti l'esecuzione e interessanti ai fini dei fattori di qualità che si vogliono valutare e, infine, l'analisi dei dati e il loro confronto con i requisiti. Tecnicamente, il test consiste nell'esercitare un programma al fine di scoprire malfunzionamenti che denuncino l'esistenza di difetti.
Il termine test, assai usato anche in italiano, indica comunemente sia il controllo dinamico in generale, inteso come attività del processo di sviluppo e disciplina dell'ingegneria del software, che una singola sessione di prova di un programma identificata dall'insieme di condizioni – dati di ingresso e contesto di esecuzione – in cui essa si svolge.
Il test come tecnica di controllo ha come principale obiettivo la verifica della correttezza funzionale di un programma o di un sistema, ma sempre più spesso viene usato come tecnica di base per la realizzazione di controlli mirati alla valutazione di altri fattori di qualità come ad esempio affidabilità, usabilità ed efficienza.
Il test è il controllo di qualità del software di gran lunga più praticato, anche se spesso, proprio per la sua concettuale semplicità – funziona? Mah, proviamolo – risulta un'attività naïf, condotta senza la necessaria pianificazione e, di conseguenza, con scarsi risultati sul piano del raggiungimento di una migliore qualità del prodotto.
In questo capitolo introduciamo gli aspetti principali di questa tecnica di controllo (nel prossimo approfondiremo i metodi di applicazione):

- ***Progettazione dei test***. Un test non si improvvisa, va studiato e preparato. Obiettivo del test è esercitare un programma per provocare malfunzionamenti: per essere efficace un test deve essere realizzato in base alle caratteristiche del programma da esaminare. La progettazione è perciò mirata a definire il contesto di esecuzione, in particolar modo selezionando l'insieme dei dati di ingresso da fornire al programma durante il test.
- ***Ambiente di test***. Il test deve avvenire in un ambiente che permetta l'esecuzione controllata dei programmi in esame. Sono necessari strumenti per fornire nella giusta sequenza i dati di ingresso selezionati per il test e per registrare tutti i dati del comportamento del programma utili per la successiva analisi; l'ambiente di esecuzione deve essere configurabile e controllabile in tutti i suoi aspetti (memoria disponibile, potenza di calcolo, esecuzione concorrente di altri programmi, etc.). Se il software sottoposto a test non è un sistema compiuto, ma un suo componente o una sua parte – come vedremo una scelta molto frequente – il test comporta anche la realizzazione di tutti i programmi di contorno necessari a simulare il comportamento delle parti mancanti del sistema).

- ***Analisi dei risultati***. I dati ottenuti dall'esecuzione di un test devono essere analizzati alla ricerca di eventuali malfunzionamenti. Sono perciò necessari dei dati di riscontro, che identificano il *risultato atteso*, con cui confrontare i risultati di test.
- ***Debugging***. A differenza delle tecniche di controllo statico, che per definizione mirano direttamente ai difetti e che spesso nel trovarli danno anche buone indicazioni sugli errori che li hanno generati, il test ha come obiettivo la scoperta di malfunzionamenti. A un test positivo deve quindi seguire la necessaria attività per l'eliminazione dei difetti – detta *debugging* dall'inglese *bug*, termine gergale per indicare il difetto – vero fine di ogni controllo volto al miglioramento della qualità.

Il controllo dinamico si mostra quindi come un'attività particolarmente onerosa all'interno del processo di sviluppo, ma necessaria. I controlli statici, infatti, hanno grande importanza nel rivelare alcune categorie di difetti e sono un utile complemento al test nella ricerca dei difetti a fronte del manifestarsi di un funzionamento, ma non possono applicarsi in modo risolutivo a tutti i casi che si presentano nel corso della realizzazione di un prodotto software.
In particolare i controlli statici, per le limitata capacità delle persone e degli strumenti, si possono applicare solo a porzioni di codice di dimensione ridotta (tipicamente i *moduli*), lasciando così scoperta la prova di un sistema nel suo insieme, per la quale il test rimane la forma di controllo più indicata. Ci sono inoltre requisiti, come l'efficienza, che possono essere controllati solo a fronte della messa in esecuzione del software.

2.3 Livelli di test

I prodotti software, anche quando si presentano all'utente finale come un tutto unico, sono in realtà dei sistemi complessi costituiti da più componenti integrati e cooperanti. Le tecniche di testing si adeguano e, insieme, traggono vantaggio da questa organizzazione comune a tutti i sistemi software. Esistendo già una decomposizione in elementi più semplici, risulta naturale applicare il testing già a partire dai singoli componenti e quindi arrivare al test dell'intero sistema seguendo strategie di integrazione dettate dall'architettura.

2.3.1 Test di unità (o dei moduli)

A questo livello, il test prende in considerazione il più piccolo elemento software previsto dall'architettura del sistema: il *modulo*. Il controllo su un modulo ha come obiettivo principale la correttezza funzionale delle operazioni esportate dal modulo a fronte della specifica definita nel *progetto di dettaglio*. In questo senso il controllo sui moduli si configura come una tipica attività di verifica. Per il controllo sui moduli sono in realtà usati metodi sia statici (ispezione e walkthrough) sia dinamici.

Fare test sui moduli significa dover effettuare i test su sistemi che sono incompleti. È necessario, in questa evenienza, che l'ambiente di test preveda dei componenti fittizi che, come è mostrato in figura 2.3, simulino le parti mancanti del sistema.

La realizzazione di queste componenti – un costo di cui è necessario tener conto – deve essere semplice e diretta per poter un corretto comportamento ed evitare di inquinare i risultati dei controlli. A seconda del ruolo svolto nel test, le componenti si dividono in due categorie:

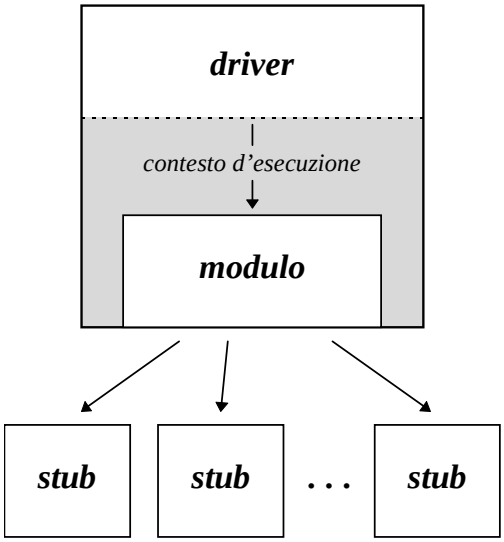

Fig. 2.3. Driver e stub nel test di un modulo.

- ***driver***, le componenti che invocano le funzionalità esportate dai moduli sottoposti a test letteralmente "pilotando" la loro esecuzione; un driver deve anche definire il contesto di esecuzione, eventualmente definendo i dati globali cui accedono i moduli sottoposti a test.
- ***stub*** (letteralmente "mozzicone"), le componenti usate per realizzare in modo *fittizio* le funzionalità invocate dai moduli sottoposti a test e necessarie per la loro esecuzione.

2.3.2 Metodi d'integrazione

È intuitivamente evidente la distanza che separa i controlli sui moduli dai controlli sul sistema completo. La pratica inoltre insegna che l'eliminazione di un difetto a partire da malfunzionamento rilevato da controlli sul sistema è più onerosa e in genere cade sempre in una fase del processo di sviluppo in cui le scadenze sono prossime e il rischio di ritardi elevato. A peggiorare questo quadro intervengono anche le statistiche: da esse scopriamo che circa il 40% dei malfunzionamenti riscontrati nei controlli di sistema sono riconducibili a difetti di mutuo interfacciamento tra i moduli. È quindi necessario colmare in modo efficace la lacuna che, dal punto di vista dei controlli, separa i moduli dal prodotto finito.
Le più recenti metodologie di sviluppo prevedono di condurre in modo progressivo l'integrazione di un sistema, intervallando i vari passi con sessioni di test. Obiettivi delle strategie d'integrazione sono la minimizzazione del lavoro e delle risorse necessarie all'integrazione e la massimizzazione del numero di difetti scoperti prima dei controlli sul sistema completo.
Alternare passi di integrazione a passi di controllo significa dover effettuare i test su sistemi che sono incompleti. È necessario, anche in questo caso, che l'ambiente di test preveda stub e driver.
Le strategie di integrazione descritte nel seguito nella loro formulazione fanno riferimento al grafo della *relazione di uso* dei moduli, una delle viste strutturali dell'architettura di un sistema software. In figura 2.4 ne è riportato un esempio: i nodi rappresentano i moduli del sistema, gli archi collegano i moduli che usano una funzionalità ai moduli che la realizzano.

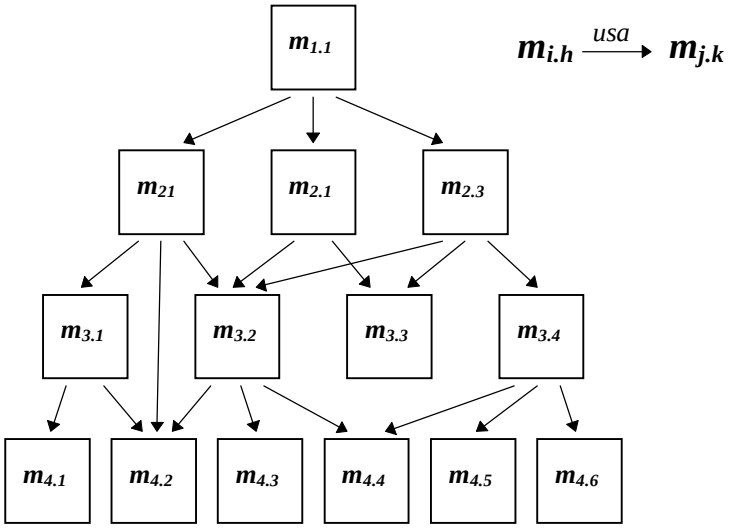

Fig 2.4. Un esempio di grafo della relazione d'uso.

Tutte le strategie d'integrazione si basano sull'assunzione – vera per la maggior parte dei sistemi software ben progettati – che il grafo, almeno come tendenza, sia privo di cicli. In altre parole si fa affidamento su una naturale strutturazione gerarchica dei sistemi in cui sia naturale identificare degli strati. Le strategie, di cui nel seguito è fornita una rassegna delle più note, costruiscono il sistema per livelli, eseguendo i controlli mano a mano che il prodotto prende forma.

- ***Big-bang***. È in realtà il nome con cui è identificata l'assenza di una strategia: tutti i moduli sono controllati singolarmente, l'integrazione avviene in un solo passo e si procede direttamente ai controlli di sistema. Oltre a soffrire di tutti i difetti discussi per evidenziare la necessità di una strategia d'integrazione, è anche un approccio costoso. Se infatti il controllo dei moduli è effettuato rigorosamente il costo è proporzionale al numero di moduli (per ogni modulo è necessario costruire una coppia driver-stub opportuna). Risulta quindi una via percorribile solo in caso di sistemi di ridotte dimensioni.
- ***Bottom-up***. Tutti i moduli che non dipendono da altri moduli sono controllati singolarmente. Quindi si integrano i moduli controllati e si sale nell'albero eseguendo un nuovo livello di controlli. Il procedimento si ripete fino a raggiungere il modulo radice. Una caratteristica di questa strategia è che per eseguire i test ai vari livelli sono necessari solamente driver.
- ***Top-down***. Il modulo radice dell'albero viene controllato singolarmente, quindi si integrano alla radice i moduli figli e si scende nell'albero eseguendo un nuovo livello di controlli. Il procedimento si ripete fino all'integrazione completa del sistema. È interessante notare che ad ogni livello si ripete sempre la stessa serie di controlli, dato che l'interfaccia del modulo radice è sempre la stessa. Per eseguire l'integrazione seguendo questa strategia sono necessari solamente stub, la cui realizzazione però, in particolare ai primi livelli, può risultare onerosa per la grande quantità e complessità delle funzionalità da simulare.
- ***Sandwich***. È, di fatto, la strategia più usata nella pratica dello sviluppo software. Consiste nell'adottare contemporaneamente entrambe le strategie bottom-up e top-down con l'obiettivo di minimizzare i costi per la realizzazione di stub e driver. La linea di demarcazione, corrispondente al livello di simulazione eliminato, non necessariamente

coincide con un livello dell'albero: in generale sarà una spezzata, che tocca più livelli, costruita per ottimizzare il rapporto costi/benefici dei controlli in dipendenza delle caratteristiche del sistema in esame.

2.3.3 Test sul sistema

Il test di sistema è la più canonica delle attività di validazione che valuta ogni caratteristica di qualità del prodotto software nella sua completezza, avendo come documento di riscontro i requisiti dell'utente.
Le tecniche più adottate per i test sul sistema sono basate su *criteri funzionali*. Gli obiettivi dei controlli di sistema sono normalmente mirati a esercitare il sistema sotto ben determinati aspetti:

- ***Facility test (test delle funzionalità)***. È il più intuitivo dei controlli, quello cioè che mira a controllare che ogni funzionalità del prodotto stabilita nei requisiti sia stata realizzata correttamente.
- ***Security test***. Cercando di accedere a dati o a funzionalità che dovrebbero essere riservate, si controlla l'efficacia dei meccanismi di sicurezza del sistema.
- ***Usability test***. Con questo controllo si vuole valutare la facilità d'uso del prodotto da parte dell'utente finale. È una valutazione su una delle caratteristiche di un prodotto software fra le più soggettive; il controllo deve prendere in esame oltre al prodotto anche tutta la documentazione che lo accompagna e deve tener conto del livello di competenza dell'utenza e delle caratteristiche operative dell'ambiente d'uso del prodotto.
- ***Performance test***. È un controllo mirato a valutare l'efficienza di un sistema soprattutto rispetto ai tempi di elaborazione e ai tempi di risposta. È un tipo di controllo critico per quelle categorie di prodotti, come ad esempio i sistemi in tempo reale, per le quali ai requisiti funzionali si aggiungono rigorosi vincoli temporali. Il sistema viene testato a diversi livelli di carico, per
- ***Storage use test***. È ancora un controllo legato all'efficienza di un sistema, ma mirato alla richiesta di risorse – la memoria in particolare – durante il funzionamento, e ha implicazioni sull'ambiente operativo richiesto per poter installare il sistema.
- ***Volume test (o load test, test di carico)***. Durante questo tipo di controllo il sistema è sottoposto al carico di lavoro massimo previsto dai requisiti e le sue funzionalità sono controllate in queste condizioni. Lo scopo è sia individuare malfunzionamenti che non si presentano in condizioni normali, quali difetti nella gestione della memoria, buffer overflows, etc., sia garantire un'efficienza base anche in condizioni di massimo carico.
  Le tecniche e gli strumenti del volume test sono di fatto usato anche per il performance test: vengono fissati alcuni livelli di carico, e su questi sono valutate le prestazioni del sistema. Sicuramente, però, i due tipi di test hanno scopi molto differenti, da un lato valutare le prestazioni a vari livelli di carico, non limite, dall'altro valutare il comportamento del sistema sui valori limite.
- ***Stress test***. Il sistema è sottoposto a carichi di lavoro superiori a quelli previsti dai requisiti o è portato in condizioni operative eccezionali – in genere sottraendogli risorse di memoria e di calcolo. Non è da confondere con il volume test da cui differisce per l'esplicito superamento dei limiti operativi previsti dai requisiti. Lo scopo è quello di controllare la capacità di "recovery" (recupero) del sistema dopo un fallimento.
- ***Configuration test***. Alcuni prodotti prevedono la possibilità di avere più configurazioni, per lo più in presenza di piattaforme di installazione diverse per sistema operativo o dispositivi hardware installati, in altri casi per soddisfare insiemi di requisiti funzionali

leggermente diversi. Questo tipo di controllo ha per obiettivo la prova del sistema in tutte le configurazioni previste.

- ***Compatibility test***. È un controllo che ha l'obiettivo di valutare la compatibilità del sistema con altri prodotti software. Gli oggetti del confronto possono essere versioni precedenti dello stesso prodotto, sistemi diversi, ma funzionalmente equivalenti che il prodotto deve rimpiazzare, oppure altri sistemi software con cui il prodotto deve interagire nel suo ambiente operativo finale.

È interessante notare come i test descritti mirano a esercitare gli aspetti di un sistema software che corrispondono a quelle caratteristiche che sono normalmente percepite come fattori di qualità. Troviamo ad esempio che il facility test e il security test sono direttamente legati con la *funzionalità*, volume test e stress test sono in rapporto con l'*affidabilità*, ovvia è la collocazione dell'usability test, performance test e storage use test rispecchiano i due aspetti dell'*efficienza*, infine, configuration test e compability test sono collegabili alla *portabilità* del sistema.

Il test di sistema è spesso replicato, condotto prima come controllo interno e poi come controllo esterno dal committente stesso o da una terza parte. Il ciclo di vita a cascata, anche nella sua semplicistica concezione, dopo i test sui moduli prevede due distinti livelli di test sul sistema: i *test di sistema* propriamente detti e i *test di accettazione*.

- Il **test di sistema** è svolto dall'organizzazione che ha condotto il processo di sviluppo. La decisione che ne segue è il rilascio del prodotto per la valutazione da parte del committente, con la quale, nella migliore delle ipotesi, termina la parte di sviluppo di un ciclo di vita.
- Il **test di accettazione** è il controllo, ultimo e definitivo, che il prodotto consegnato dal fornitore rispetti i requisiti del committente. E' di responsabilità del committente. Questi, in generale, si limita a svolgere un ruolo di revisore di una replica dei test di sistema svolti sotto la sua diretta osservazione. Quando il committente non possiede le competenze necessarie, è una soluzione comune eseguire i test di accettazione in presenza di una terza parte incaricata di curare gli interessi del committente.
  Il test di accettazione è spesso previsto nei contratti sotto la dizione **collaudo**. Da un punto di vista contrattuale, per il collaudo ha particolare importanza il documento di analisi dei requisiti. È su questo, sulla sua completezza e chiarezza, che deve essere basato il giudizio che confermerà o meno l'accettazione del prodotto: un documento lacunoso e impreciso difficilmente può giustificare il rifiuto di un prodotto. Da un punto di vista più pratico, è importante che il collaudo sia condotto dall'utenza a cui il prodotto è destinato e che si svolga in condizioni operative reali. Esistono infatti fattori di qualità che dipendono soggettivamente dall'utenza e che possono essere valutati correttamente solo a prodotto finito.

2.4 Il V-Modell

Per concludere, la figura 2.5 presenta un modello di ciclo di vita che integra i vari livelli di test e l'approccio top-down all'integrazione. Si tratta del V-Modell, uno standard tedesco. Come si può notare dalla figura, è una rivisitazione del modello a cascata: il flusso di lavoro della costruzione è lineare, come espresso dalle due frecce grandi. Inoltre, con le frecce tratteggiate orizzontali, mette in evidenza la relazione tra i vari livelli di test e le diverse

descrizioni del sistema. Così, il test d'accettazione viene progettato tenendo conto dei requisiti, quello di sistema del progetto a livello di sistema, l'integrazione è guidata dall'architettura software, e i test d'unità si basano sui risultati della progettazione di dettaglio.

La figura mette in evidenza come la progettazione dei test possa iniziare molto prima dell'effettiva esecuzione degli stessi. Le frecce diagonali stanno a rappresentare l'influenza che la progettazione dei test di ciascun livello può avere sull'attività di progettazione del livello successivo. Non sono mostrati in questo schema di principio i probabili cicli di ri-lavorazione conseguenti all'esecuzione dei test.

C'è da notare che il modello è stato recentemente aggiornato, tenendo conto dell'esperienza accumulata nel suo utilizzo, con l'introduzione di caratteristiche di tipo "extreme programming", come indicato anche dal nuovo nome, V-Modell-XP.

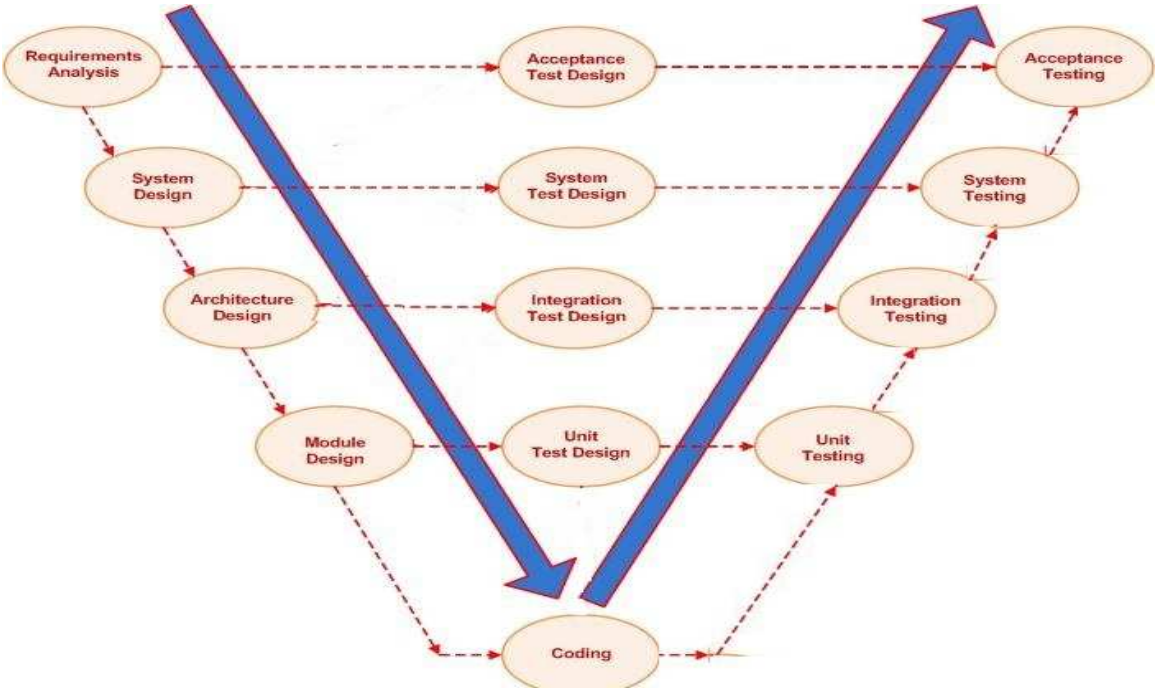

Fig. 2.5. Il V-Modell. (adattata da Wikipedia)

3 Il test: metodi

Il controllo dinamico, o test, è una delle tecniche più note e più usate nella valutazione della qualità dei prodotti software. L'intuitività dell'approccio non significa che possa essere eseguito in modo naïf. Come osservato da Dijkstra, il test *può* rilevare la presenza di malfunzionamenti in un programma, ma non dimostrarne l'assenza. I metodi di progettazione del test che tratteremo in questo capitolo definiscono tecniche e criteri per ottenere buone approssimazioni del test *ideale* che potrebbe garantre la correttezza del programma. Esamineremo inoltre le strategie da adottare nella conduzione dei test e le tecniche per

produrre i dati corretti da usare come pietra di paragone per valutare la correttezza dei risultati dei test.

L'importanza dei metodi per la progettazione e l'analisi dei risultati dei test deriva da questa considerazione: in un processo di produzione software è un obiettivo primario ottenere il miglior rapporto possibile fra la qualità del prodotto e il tempo e le risorse impiegate nelle attività di controllo.

3.1 Progettazione dei test

La progettazione dei test è l'attività di definizione di un insieme di casi di test. Si definisce come **caso di prova** (o *caso di test*, o *test case*) una tripla

<input, output, ambiente>

che specifica i dati di input, i risultati attesi, ed una descrizione dell'ambiente di esecuzione. Un insieme (o una sequenza) di casi di test è detta **batteria di test** (o *test suite*).

Non esistono test che siano validi in generale: ogni test deve essere costruito su misura, in dipendenza delle caratteristiche del software da controllare e dei fattori di qualità che si vogliono valutare. Questa realtà, insieme alle considerazioni economiche che sempre accompagnano ogni processo di sviluppo, è condensabile nelle tre seguenti affermazioni:

- la probabilità di rilevare il maggior numero di malfunzionamenti aumenta proporzionalmente al numero dei test eseguiti;
- i controlli di qualità, ivi compresi i test, hanno un costo che, ovviamente, è necessario tenere basso;
- i test non devono essere condotti artigianalmente: si perde la possibilità di valutare oggettivamente il valore dei controlli.

La progettazione dei test cerca di rispondere contemporaneamente a queste esigenze. I criteri che guidano la progettazione dei test possono essere divisi in due classi principali: *criteri funzionali* e *criteri strutturali*. I criteri descritti nei paragrafi seguenti sono formulati in generale, parleremo perciò di programmi più che di moduli o di sistemi software. Tuttavia, come vedremo, le caratteristiche proprie delle due classi rendono conveniente scegliere i criteri da applicare in dipendenza della fase del processo di sviluppo in cui il prodotto è sottoposto a controllo. Alle due classi principali di criteri si aggiungono strategie di test più articolate, ottenute combinando gli approcci funzionali e strutturali a partire da principî originali. Infine, come ultimo ma non meno importante aspetto della progettazione dei test, è necessario definire la pietra di paragone, l'*oracolo,* che genera dei valori (risultati attesi) con cui confrontare i risultati dei test, per evidenziare gli eventuali malfunzionamenti esibiti dal software.

3.2 Criteri funzionali (black-box)

Sotto la dizione funzionale si raccolgono i criteri di progettazione dei test che si basano solamente sui requisiti di un programma. Nella terminologia inglese, spesso più frequente nel linguaggio tecnico, sono definiti come criteri *black-box*, indicando appunto che il programma è trattato come una scatola chiusa (letteralmente "nera"): ai fini dei test non è necessaria la

conoscenza della sua realizzazione. Nella progettazione dei test interverranno solamente le caratteristiche esterne del programma: la sua interfaccia di ingresso/uscita e l'ambiente di esecuzione.
I criteri funzionali prevedono di selezionare l'insieme dei dati di ingresso che costituirà il test a partire dallo studio delle funzionalità di un programma. I vari criteri si distinguono in base alle regole con cui sono individuati i casi rilevanti che costituiranno la materia del test. Nel caso di specifiche espresse in linguaggio formale i criteri funzionali sono più facilmente applicabili e spesso è possibile ricorrere a strumenti automatici per generare direttamente i casi di test o, comunque, è possibile definire delle regole che rendono il procedimento automatizzabile. Quando invece le specifiche non sono espresse formalmente o peggio si hanno a disposizione solo i requisiti espressi in linguaggio naturale, i criteri funzionali risultano di più difficile applicazione e diventa fondamentale l'esperienza dei professionisti incaricati di progettare i test.
Nella migliore delle ipotesi, dalle specifiche si può ricavare una definizione esatta dell'insieme di tutti gli input di un programma: un test che eserciti il programma su questo insieme, cioè su tutti i possibili valori d'ingresso, è certamente esaustivo. Purtroppo nella pratica quest'evenienza è rara: nella maggior parte dei casi le specifiche non arrivano a questo livello di dettaglio. E quando anche si riesce ad avere una definizione dell'insieme dei possibili ingressi, normalmente risulta così grande che non è plausibile tentare di usarlo effettivamente come insieme di casi di test. Nel seguito presentiamo alcuni fra i più noti criteri funzionali descritti in letteratura: tutti hanno come comune obiettivo l'individuazione di insiemi di input di dimensioni limitate e comunque significativi per il test.

- ***Statistico***. I casi di test sono selezionati in base alla distribuzione di probabilità dei dati di ingresso del programma. Il test è quindi progettato per esercitare il programma sui valori di ingresso più probabili per il suo utilizzo a regime – tali valori costituiscono il *profilo operazionale* del programma. Il vantaggio principale di questo criterio è che, nota la distribuzione di probabilità, la generazione dei dati di test è facilmente automatizzabile. Non sempre però è possibile ricavare dalle specifiche una distribuzione di probabilità e non sempre è detto che quella dedotta dalle specifiche corrisponda alle effettive condizioni d'utilizzo del software. In quest'ultima evenienza la natura prettamente utilitaristica di questo test potrebbe rivelarsi pericolosa. Un altro lato debole del test statistico è che la generazione casuale dei dati d'ingresso può scegliere valori per i quali può essere particolarmente oneroso calcolare il risultato atteso.
- ***Partizione dei dati d'ingresso***. Il dominio dei dati di ingresso è ripartito in classi di equivalenza: due valori d'ingresso appartengono alla stessa classe di equivalenza se, in base ai requisiti, dovrebbero produrre lo stesso comportamento del programma. La progettazione dei casi di test prosegue in modo da esercitare il programma su valori rappresentanti di ognuna delle classi di equivalenza. Il criterio è economicamente valido solo per quei programmi per cui il numero dei possibili comportamenti è sensibilmente inferiore alle possibili configurazioni d'ingresso. D'altra parte, per come sono costruite le classi di equivalenza, i risultati attesi dal test sono noti e quindi non si pone il problema dell'oracolo. È infine necessario tener presente che il criterio è basato su un'affermazione generalmente plausibile, ma non vera in assoluto. Infatti, la deduzione che il corretto funzionamento sul valore rappresentante implichi la correttezza su tutta la classe di equivalenza dipende dalla realizzazione del programma e non è verificabile sulla base delle sole specifiche funzionali.
- ***Valori di frontiera***. Anche questo criterio è basato su una partizione dei dati di ingresso. Le classi di equivalenza possono essere realizzate o in base all'eguaglianza del

comportamento indotto sul programma o in base a considerazioni inerenti il tipo dei valori d'ingresso. Come dati di test sono considerati i valori estremi di ogni classe di equivalenza. In dipendenza di come sono definite le classi in cui l'insieme dei dati d'ingresso è ripartito, è possibile che non siano noti i risultati del programma in corrispondenza dei valori scelti, ciò significa che deve essere considerato il problema dell'oracolo. Questo criterio richiama i controlli sui valori limite tradizionali in altre discipline ingegneristiche per cui è vera la proprietà del comportamento continuo.
In meccanica, ad esempio, una parte provata per un certo carico resiste con certezza a tutti i carichi inferiori. Questa proprietà non è applicabile al software: quando anche sia possibile stabilire una nozione di continuità nell'insieme dei dati d'ingresso, non è dato dedurre il comportamento continuo del programma. La giustificazione del criterio sta invece nella considerazione – diametralmente opposta – per cui, nei programmi, i valori limite sono frequentemente trattati in modo particolare. In questa prospettiva, il criterio dei valori di frontiera risulta un utile complemento ad altri criteri basati sulla partizione dei dati di ingresso.

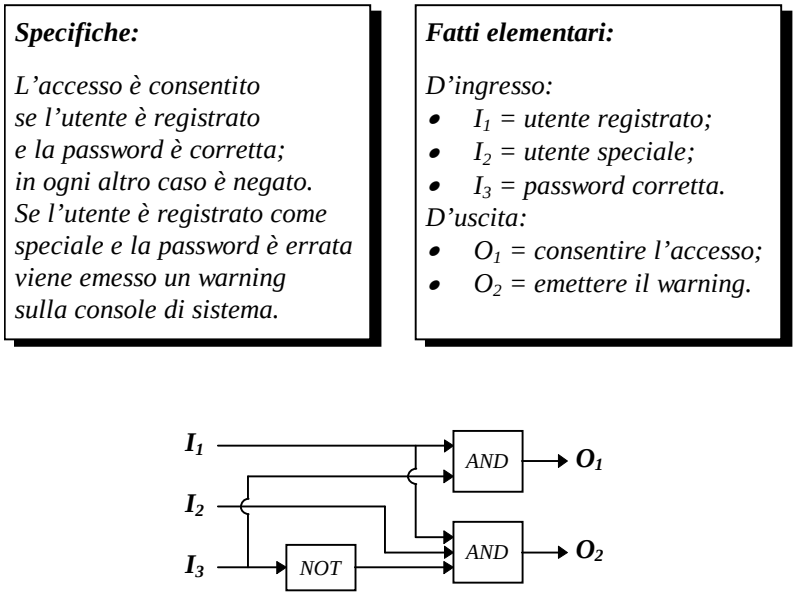

Fig. 3.1. Il grafo causa-effetto applicato a un sistema di autenticazione.

- ***Grafo causa-effetto***. L'applicazione di questo criterio si basa sulla costruzione, ottenuta a partire dai requisiti o dalle specifiche del programma, di un grafo che lega un insieme di fatti elementari di ingresso (cause) e di uscita (effetti) in una rete combinatoria che definisce delle relazioni di causa-effetto, come è possibile vedere nell'esempio di fig. 3.1. I casi di test si ricavano risalendo il grafo a partire dalle combinazioni di fatti elementari d'uscita e ottenendo le combinazioni di fatti elementari di ingresso che le generano. Per il modo con cui i casi di test sono ricavati, il criterio risolve direttamente il problema dell'oracolo. Inoltre, il grafo causa-effetto può essere costruito durante la fase di validazione dei requisiti, per mettere in luce contraddizioni ed evidenziare parti mancanti di logica. In tal caso è anche possibile riusare il grafo durante la progettazione dei test.

Il principale vantaggio dei criteri black-box consiste nella possibilità di anticipare la progettazione dei test. I criteri non dipendono dal codice del programma da controllare: la

progettazione dei test e la realizzazione dell'ambiente di test non sono perciò vincolate ad attendere la codifica del software.
Questa considerazione assume particolare importanza nel processo di sviluppo. La scelta dei criteri funzionali per la progettazione dei test di sistema – generalmente i più costosi da realizzare – permette di parallelizzare l'attività di realizzazione del prodotto e quella di progettazione e predisposizione dei test. Le informazioni contenute nel documento di analisi dei requisiti o, con maggior precisione formale, nelle specifiche, definiscono completamente le funzionalità del prodotto software e risultano quindi sufficienti per applicare i criteri funzionali.

3.3 Criteri strutturali (white box)

Sotto la dizione strutturali si raccolgono i criteri di progettazione dei test che tengono conto della realizzazione del programma da controllare, che cioè si basano sulla struttura del codice. Idea di partenza e obiettivo dei test ispirati ai criteri strutturali è esercitare il programma in ogni sua parte. In letteratura e nel linguaggio tecnico comune, i criteri strutturali sono indicati con l'attributo inglese *white-box*, scelto per contrapposizione all'opacità evocata dalla dizione black-box usata per i criteri funzionali.

I criteri white-box sono basati sull'analisi del *flusso di controllo* o del *flusso dei dati* di un programma. La progettazione dei test fa riferimento a un *grafo di flusso* ricavato dall'analisi del codice del programma. Ad ogni nodo del grafo è associato un comando (*statement*); gli archi orientati rappresentano il legame di sequenzialità fra i comandi: a essi possono essere associate le condizioni che determinano le direzioni (*branch*) prese dal flusso del controllo in seguito all'esecuzione di comandi condizionali. Per esempio, la figura 3.2 mostra, a destra, il grafo di flusso della semplice funzione data a sinistra.

I criteri basati sull'analisi del flusso di controllo sono attualmente i più adottati. Storicamente, sono stati i primi a essere proposti in letteratura: alla maggior notorietà è seguita anche la realizzazione di strumenti automatici e la proposta di metodi che ne facilitano l'applicazione. I più noti test ispirati a criteri appartenenti a questa categoria sono:

- ***test sui comandi*** (*statement test*); ha l'obiettivo di esercitare i comandi del programma: in termini di grafo di flusso, ogni nodo deve essere percorso almeno una volta;
- ***test sulle decisioni*** (*branch test*); obiettivo del test è far assumere alle espressioni che controllano i comandi condizionali i valori di vero e falso almeno una volta; in termini di grafo di flusso sono esercitati gli archi che corrispondono alle decisioni; è importante notare che il test sulle decisioni include propriamente il test sui comandi, infatti in base alle proprietà topologiche dei grafi di flusso – escluso il caso banale di un programma senza branch – esercitare tutti i branch significa esercitare anche tutti gli archi del grafo (i test ispirati a questo criterio in letteratura si trovano citati come *edge test*), e quindi tutti i nodi;

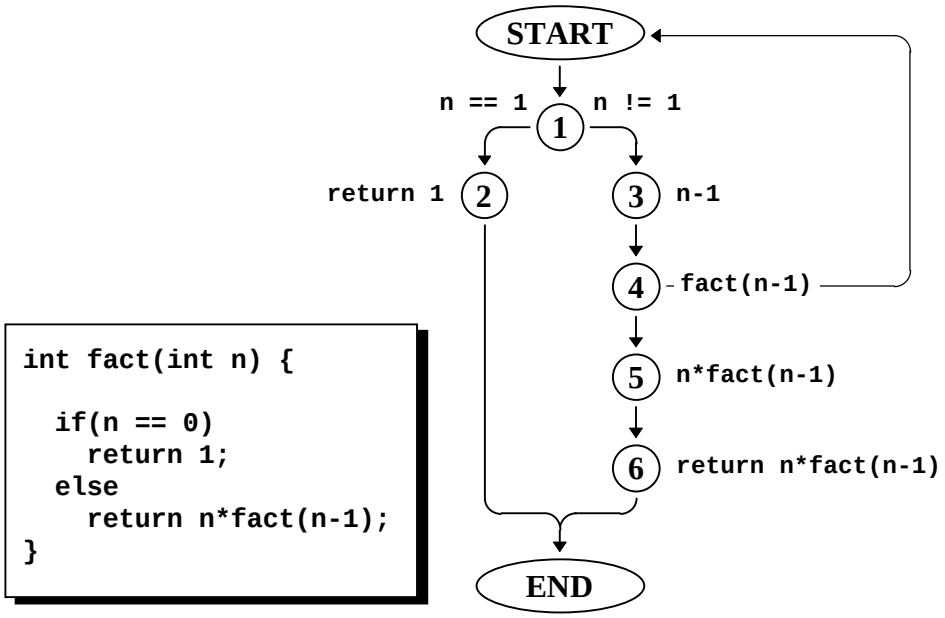

Fig. 3.2. Un esempio di grafo di flusso.

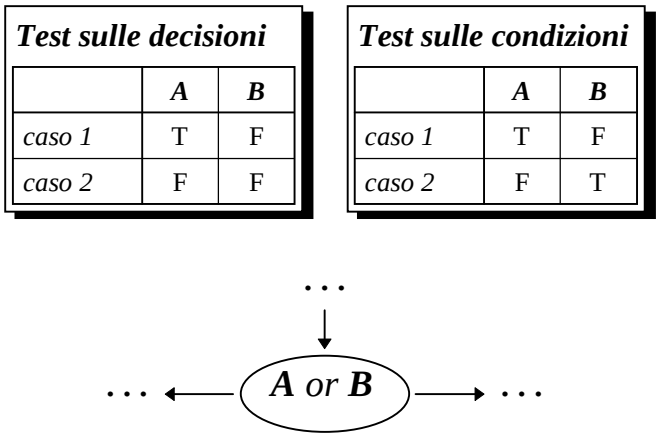

Fig. 3.3. Confronto fra test sulle decisioni e test sulle condizioni.

- ***test sulle condizioni*** (*condition test*); è una variazione del branch test che invece di concentrarsi sul risultato del comando condizionale (il branch o la direzione percorsa) pone attenzione sul modo con cui la direzione è decisa, cioè sul modo con cui viene calcolata l'espressione booleana: ogni espressione booleana elementare deve assumere almeno una volta entrambi i valori di vero e falso; la differenza con il test precedente si nota in modo evidente quando le espressioni booleane sono costituite da molte condizioni composte logicamente;
- ***test sulle decisioni e sulle condizioni*** (*decision/condition test*); è possibile dimostrare che esistono casi per cui branch test e condition test non sono inclusivi; è sufficiente ragionare sulla condizione di branch definita dall'*or* logico di fig. 3.3: i casi di test sufficienti a soddisfare un criterio non sono validi per l'altro e viceversa; è quindi ragionevole impostare una strategia di test basata sulla composizione dei due criteri: i dati di test sono definiti in modo che sia tutte le decisioni presenti nel programma che tutte le condizioni di cui sono composte assumano almeno una volta entrambi i valori di vero e falso.

I criteri a cui sono ispirati i test descritti finora non trattano cicli: è possibile ottenere dati di test che esercitano tutti i comandi, le decisioni e le condizioni percorrendo ogni ciclo una sola volta. È tuttavia comune che un difetto causi un malfunzionamento solo in seguito a un certo numero di ripetizioni di un ciclo, come ad esempio un banale *Obi-Wan*[1] (gli errori "di uno" nell'indicizzazione dei vettori o nelle condizioni di disuguaglianza) o un più subdolo *memory leak*. Per trattare i cicli è necessario considerare i cammini nel grafo di flusso del programma:

- ***test dei cammini*** (*path test*); il test è progettato in modo da percorrere tutti i cammini plausibili nel grafo di flusso; questo criterio, già in assenza di cicli vede una crescita potenzialmente esponenziale del numero di casi di test al crescere del numero di decisioni (2^3 cammini per le 3 decisioni nel grafo di figura 3.4) e in presenza di cicli ha valore puramente teorico: applicarlo direttamente significa provocare tutte le possibili esecuzioni del programma;

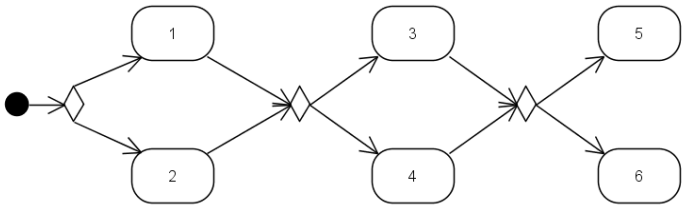

Fig. 3.4. Crescita dei cammini

- ***n-test dei cicli***; anche per programmi molto semplici, il numero dei cammini percorribili può essere assai alto o anche non limitabile a priori, implicando costi troppo elevati per l'esecuzione dei test. Questo criterio, rispetto al precedente, introduce un limite al numero di volte che un ciclo può essere percorso. Non esiste un metodo per determinare il numero ottimale di iterazioni per i cicli di un programma: è lasciato all'esperienza del progettista dei test individuare il miglior compromesso fra la profondità dei test e il loro costo.

Inizialmente l'analisi del flusso dei dati è stata utilizzata nei compilatori a supporto delle tecniche di ottimizzazione del codice. Più recentemente, studi statistici hanno confermato le ipotesi formulate dai teorici circa i promettenti legami fra il test basato sull'esercizio dei dati e la rilevazione dei malfunzionamenti di un programma.
I criteri fondati sul flusso dei dati hanno come obiettivo principale le *definizioni* e l'*uso* delle variabili nel programma. È da notare che, in questa terminologia, con "definizione" di una variabile si intende l'assegnamento, la scrittura cioè, mentre con "uso" si intende la lettura del suo valore. I più noti test basati su questi criteri sono:

- ***test delle definizioni***; i test progettati secondo questo criterio esercitano, per ogni definizione di variabile, almeno un cammino nel grafo di flusso che ne contenga un uso;
- ***test degli usi***; è una forma più raffinata del precedente test: per ogni definizione di variabile il test deve esercitare almeno un cammino nel grafo di flusso che contenga un uso prima di altre definizioni della stessa variabile; obiettivo di questo raffinamento – la cui applicabilità dipende anche dalla struttura del programma – è ridurre il numero di casi di test e circoscrivere il difetto rispetto al malfunzionamento.

[1] Gioco di parole per dire *off by one* usando il nome di Obi-Wan Kenobi, personaggio di Guerre Stellari.

- ***test di tutti gli usi nelle condizioni / di alcuni usi nei calcoli***; con questo test (in originale noto come *all-p-uses/some-c-uses*) si fa distinzione fra gli usi di una variabile in un calcolo (*c-use*) o nella valutazione di una condizione (*p-use*, da *predicate*); per ogni variabile, per ogni sua definizione si devono esercitare tutti i percorsi che contengono p-use; questo criterio può lasciare scoperte alcune definizioni, quelle per cui la variabile è usata solo in calcoli; al solito, il criterio che ispira questo test ha come obiettivo la scelta di un insieme di casi di test "rilevanti", qui la scelta è basata sull'assunzione che i p-use siano più critici ai fini della rilevazione di malfunzionamenti.

I test ispirati ai criteri white-box producono un insieme di dati d'ingresso che è determinato dalla struttura del programma e che non ha legami diretti con le sue funzionalità. Questo significa che i casi di test che ne derivano possono essere molto distanti dalla distribuzione di probabilità dei dati d'ingresso del programma. È infatti frequente che una parte sostanziale di un programma sia destinata al trattamento di casi particolari: le statistiche più pessimiste parlano di un 80% del codice usato solamente nel 20% dei casi. In questo senso, i casi di test ottenuti dai criteri strutturali, da un punto di vista strettamente utilitarista, possono essere considerati inefficienti.
Un ulteriore limite alla convenienza dei test basati su criteri strutturali è il problema dell'oracolo. Dai procedimenti con cui sono costruiti i casi di test non si ricavano indicazioni circa l'insieme dei dati di output che identifica il corretto funzionamento del programma; può quindi essere costoso generare un oracolo per il confronto dei risultati dei test.
I criteri strutturali non sono impiegati nella progettazione dei controlli di sistema. La grana fine del controllo esercitato sul software e l'onere dell'analisi del codice per ricavare i casi di test rendono i test ispirati a criteri white-box particolarmente indicati per controlli su oggetti di più ridotte dimensioni. La progettazione dei test a livello di modulo è il principale campo di applicazione. In particolare, quando la programmazione è molto strutturata, è conveniente ridurre ulteriormente la grana del controllo e considerare come oggetti del test le parti di codice (procedure, funzioni, subroutine, etc.) relative alla realizzazione delle singole funzionalità del modulo.

3.4 Strategie di test

Le famiglie di criteri black-box e white-box costituiscono lo "zoccolo duro" della teoria e della pratica della progettazione dei test. Tuttavia il test è una disciplina lungi dall'essere consolidata e non esistono regole o criteri che garantiscano la sicurezza dei risultati. Per questo motivo, da una parte vengono proposti sempre nuovi criteri di test, dall'altra – pur nel rispettabile intento di trovare il maggior numero di malfunzionamenti – non diminuisce la tendenza a usare tecniche artigianali basate soprattutto sul buon senso e sull'esperienza dei progettisti. Il risultato più concreto di questo fermento tecnologico è la formulazione di articolate strategie di test.
Una strategia comprende spesso sia criteri di tipo funzionale che criteri di tipo strutturale. Una prima famiglia di strategie deriva, abbastanza naturalmente, dall'idea di fondere gli approcci black-box e white-box; in letteratura troviamo questo approccio citato col nome di ***criteri gray-box***. Una strategia di tipo gray-box prevede di testare il programma conoscendo i requisiti ed avendo una limitata conoscenza della realizzazione, per esempio conoscendo solo l'architettura. Un'altra strategia gray-box propone di progettare il test usando criteri funzionali e quindi di usare le misure di copertura (si veda la sezione "Valutazione dei test") dei criteri strutturali per valutare l'adeguatezza del test: si può conoscere quanto del codice

del programma è stato esercitato. Il seguente approccio di tipo gray-box, che proponiamo a titolo di esempio, è stato suggerito da Myers [Mye79]:

- un primo test è progettato utilizzando il grafo causa-effetto, questo consente anche di eseguire un'analisi accurata degli ingressi e delle uscite per verificare la completezza funzionale del programma;
- il grafo causa-effetto, realizzato al passo precedente, fornisce informazioni utili per determinare una partizione del dominio dei dati d'ingresso che sarà usata per integrare il test precedente applicando il criterio dei valori di frontiera;
- i progettisti in base alla loro esperienza, all'analisi funzionale svolta fino a questo punto nella progettazione dei test precedenti, al comportamento e agli eventuali malfunzionamenti del programma evidenziati dai test, sono chiamati a formulare delle ipotesi di malfunzionamento (*error guessing*, vedi più avanti) e a integrare di conseguenza i casi di test;
- si usa, infine, la struttura del programma per stabilire, alla luce dei criteri white-box, se i test realizzati ai passi precedenti hanno esercitato a sufficienza il codice. In caso contrario si possono sviluppare ulteriori casi di test.

Una strategia diversa è rappresentata dal ***test mutazionale***. La tecnica si applica in congiunzione con altri criteri di test: nella sua formulazione è prevista infatti l'esistenza, oltre al programma da controllare, anche di un insieme di test già realizzati. La strategia prevede di introdurre modifiche controllate nel programma originale. Le modifiche riguardano in genere l'alterazione del valore delle variabili e la variazione delle condizioni booleane. I programmi così ottenuti, e scorretti – di regola – rispetto alle specifiche, sono definiti *mutanti*. L'insieme dei test realizzati precedentemente viene quindi applicato, senza modifiche, a tutti i mutanti e i risultati confrontati con quelli degli stessi test eseguiti sul programma originale.
Questa strategia è adottata con obiettivi diversi. Può essere usata per favorire la scoperta di malfunzionamenti ipotizzati: intervenire sul codice può essere più conveniente rispetto alla generazione di casi di test ad hoc. Il test mutazionale risulta inoltre utile per valutare l'efficacia dell'insieme di test, controllando se "si accorge" delle modifiche introdotte sul programma originale. La strategia può infine essere adottata per cercare indicazioni circa la localizzazione dei difetti la cui esistenza è stata denunciata dai test eseguiti sul programma originale.
L'uso della tecnica è limitato dal gran numero di mutanti che possono essere definiti, dal costo della loro realizzazione, e soprattutto dal tempo e dalle risorse necessarie a eseguire i test sui mutanti e a confrontare i risultati. Una buona conoscenza del codice aggiunta all'esperienza nell'ipotizzare i malfunzionamenti sono elementi necessari per introdurre un piccolo numero di modifiche mirate. Per questi motivi, il test mutazionale è usato spesso in congiunzione con criteri strutturali, adottati per la realizzazione dei test sul programma originale.

Una strategia applicabile al controllo dei programmi in evoluzione è quella nota in letteratura con il nome di ***test di regressione*** (o, a seconda dei testi, di *non regressione*). La strategia ha l'obiettivo di controllare se, dopo una modifica apportata per una necessità di sviluppo, il software è regredito, se cioè siano stati introdotti dei difetti non presenti nella versione precedente alla modifica. La strategia consiste nel riapplicare al software modificato i test progettati per la sua versione originale e confrontare i risultati. È giustificato parlare di strategia perché il test di regressione deve essere previsto e pianificato: è necessario mantenere i risultati e la documentazione di ogni esecuzione dei test ed è conveniente, visto il

riuso che se ne farà, investire maggiori risorse nella progettazione della prima batteria di test. In generale il riuso diretto dei test è un evento piuttosto raro, più comunemente si assiste a un'evoluzione parallela del software e dei suoi test.
Il test di regressione è una strategia che trova applicazione in diversi scenari della produzione di software. Un primo caso, probabilmente il più comune, è la manutenzione software. Il prodotto viene consegnato insieme a una batteria di test: in seguito ad ogni intervento di manutenzione i test sono applicati per verificare che il programma non sia regredito. Di fatto, però, con il tempo, il susseguirsi di interventi di manutenzione adattiva e soprattutto perfettiva (e non monotona) rendono la batteria di test obsoleta. Un caso diverso riguarda i processi di sviluppo software basati su un ciclo di vita evolutivo. Il prodotto è realizzato attraverso una serie di prototipi; i test, soprattutto mirati alle funzionalità del prodotto, sono sviluppati insieme al primo prototipo e accompagnano l'evoluzione del prodotto, controllando in particolare che ogni passo evolutivo non abbia introdotto nuovi difetti. Per certi aspetti simile è l'uso dei test di regressione durante l'integrazione di un sistema software condotta secondo la strategia top-down.

In molti casi il malfunzionamento di un sistema è dovuto a errori che coinvolgono le interfacce dei moduli. Esistono quindi delle strategie di test che cercano di esercitare un sistema con l'obiettivo di mettere in evidenza i malfunzionamenti dovuti a questo tipo di errori. In altre parole si tratta di una rivisitazione dei criteri strutturali in termini dell'architettura di un sistema invece che del codice di un programma. I ***test di interfaccia*** sono basati su una classificazione degli errori commessi nella definizione delle interazioni fra i moduli:

- ***errore di formato***; i parametri di invocazione o di ritorno di una funzionalità sono sbagliati per numero o per tipo; è un errore piuttosto frequente, ma fortunatamente è anche l'errore che i moderni linguaggi di programmazione e relativi mezzi di sviluppo – compilatori e linker – permettono di rilevare automaticamente con controlli statici;
- ***errore di contenuto***; i parametri di invocazione o di ritorno di una funzionalità sono sbagliati per valore; è il caso in cui i moduli si aspettano argomenti il cui valore deve rispettare ben precisi vincoli; si va da parametri non inizializzati (e.g. puntatori nulli) a strutture dati inutilizzabili (e.g. un vettore non ordinato passato a una procedura di ricerca binaria);
- ***errore di sequenza o di tempo***; in questo caso è sbagliata la sequenza con cui è invocata una serie di funzionalità – singolarmente corrette; nei sistemi dipendenti dal tempo possono anche risultare sbagliati gli intervalli temporali trascorsi fra un'invocazione e l'altra o fra un'invocazione e la corrispondente restituzione dei risultati.

Gli errori sono dovuti generalmente a una cattiva progettazione di dettaglio in conseguenza della quale si hanno specifiche di modulo lacunose o ambigue. In questo caso i moduli, presi singolarmente, rispettano le specifiche e i test di modulo non possono rilevare l'esistenza di un difetto. Più raro è il caso di specifiche di modulo corrette, cui però, per la scarsa importanza che viene riservata loro in un processo di sviluppo "rilassato", non si è fatta la dovuta attenzione durante la codifica dei moduli. Qui i moduli, anche singolarmente non rispettano le specifiche e i test di modulo avrebbero dovuto rilevare la presenza di difetti.
I test di interfaccia sono progettati in base agli errori che si suppone siano presenti nel sistema, cercando in particolare di esercitare i moduli affinché siano riprodotte le situazioni tipiche degli errori di contenuto e di sequenza o di tempo. I criteri di progettazione tipici del test di interfaccia sono adottati soprattutto a supporto delle strategie di integrazione, in

particolare per i test su tutto il sistema nelle strategie d'ispirazione top-down e come test ai vari livelli d'integrazione in quelle bottom-up (vedi § 2.3.2).

Non sempre è possibile ottenere un oracolo nei limiti di tempo e di risorse previsti da un processo di sviluppo. In questi casi è pratica comune operare una ***semplificazione dei casi di test*** a quelli per i quali è possibile determinare, con esattezza e semplicità, i risultati attesi. Questa semplificazione può essere applicata con successo quando, in base alla conoscenza algoritmica del programma, dalla correttezza del comportamento sui casi "semplici" possiamo dedurre, con un buon margine di probabilità, la sua correttezza su tutti i casi.

Per concludere la nostra rassegna, è infine da ricordare che in molti casi pratici l'asse portante di una strategia di test è del tutto empirico. I vari criteri, metodi e suggerimenti sono di fatto usati a supporto di una "tecnica" principale detta ***previsione degli errori*** e basata sul tentativo di indovinare (in inglese si parla proprio di *error guessing*) gli errori commessi dai professionisti che hanno partecipato alla realizzazione del software. Questa tecnica, che fa affidamento sulle incompetenze note di progettisti e programmatori, consiste nel formulare ipotesi sulla realizzazione di un programma e sui possibili malfunzionamenti che ne possono essere conseguenza: i casi di test sono progettati di conseguenza. La tecnica è efficace proporzionalmente all'esperienza e alle competenze dei professionisti che la adottano e, sebbene molto usata e spesso fruttuosa, non dovrebbe essere contemplata fra le tecniche ingegneristiche che, per definizione, devono essere documentabili e replicabili.

3.5 L'oracolo

In molti casi, in particolar modo quando i test sono progettati seguendo criteri strutturali, non si hanno indicazioni dirette circa i risultati che si devono attendere dall'esecuzione del test. Manca cioè un elemento di riscontro per valutare la correttezza del software sottoposto a test. Con il termine *oracolo* si identifica un metodo, spesso applicato da un agente automatico, per generare i risultati corretti da usare come pietra di paragone durante il test. In letteratura e nel linguaggio comune, con oracolo si intende anche l'insieme dei dati usati per il confronto. In fig. 3.6 è schematizzato il ruolo dell'oracolo nell'esecuzione di un test; in particolare si nota come l'oracolo e il programma sottoposto a controllo dovrebbero condividere condizioni di test identiche – dati di ingresso e ambiente di test.

Non esiste una soluzione generale al problema di procurarsi un oracolo. Nel seguito presentiamo alcune delle soluzioni più comunemente adottate in pratica.

- ***Analisi delle specifiche***. Fissati i dati di test in ingresso al programma si possono desumere i risultati attesi attraverso l'analisi delle sue specifiche. In teoria queste contengono una descrizione completa e formale della semantica del programma, è quindi possibile applicarle ai dati in ingresso per ricavare il risultato che si attende dalla sua esecuzione. Questa soluzione trova una naturale applicazione nei casi in cui il linguaggio di specifica sia eseguibile.
- ***Inversione della funzione***. Quando la funzionalità espressa dal programma da controllare è invertibile si realizza un programma che calcola l'inversa. I dati che si ottengono applicando l'inversa ai risultati del test sono confrontati con rispetivi dati d'ingresso. Questa

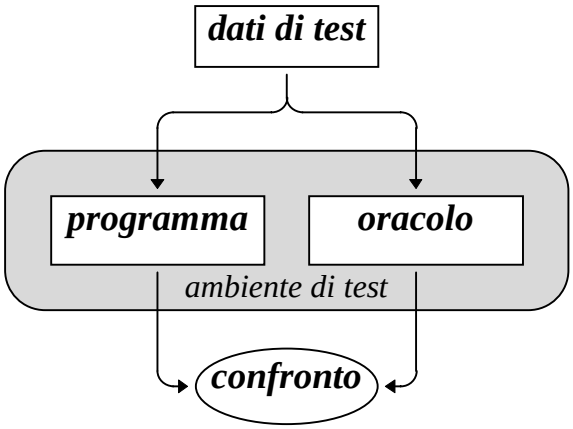

Fig. 3.6. L'oracolo nell'esecuzione dei test.

soluzione è applicabile raramente, in generale quando il software in esame è di tipo matematico: in molti di questi casi non solo le funzionalità sono invertibili, ma il programma che calcola l'inversa fa già parte del software in via di sviluppo, annullando il costo della realizzazione dell'oracolo. Una limitazione intrinseca di questo metodo è che, quando il confronto rileva un malfunzionamento, esso può essere tanto nel programma che calcola la funzionalità diretta che in quello che calcola l'inversa. Inoltre, l'approssimazione dei dati numerici nei calcoli può essere un ostacolo all'applicabilità del metodo. Il campo di applicazione di questo metodo può essere allargato evitando di calcolare la vera inversa – quando ciò risulta troppo costoso o impossibile – limitandosi a una pseudo-inversa i cui risultati sono confrontati con i dati d'ingresso del test mediante tecniche di *plausibilità* (vedi più avanti).

- ***Plausibilità dei risultati***. E' una tecnica di semplificazione sui dati d'uscita. A partire dalle specifiche si ricavano dei vincoli di plausibilità fra i dati d'ingresso e quelli di uscita. I risultati dei test sono quindi analizzati non a fronte del risultato corretto, ma in base a questi vincoli: più stretti sono e più bassa è la probabilità di non riconoscere un malfunzionamento accettando per buoni dei dati di uscita non corretti.
- ***Versioni multiple indipendenti***. Questa soluzione, nota anche con il termine inglese *back-to-back*, prevede lo sviluppo indipendente di due – o anche più – copie funzionalmente identiche del programma da usare vicendevolmente come oracolo. A meno di casi particolari, ragioni di costo sconsigliano l'applicazione diretta di questa soluzione, che soffre anche del limite di non fornire nessuna garanzia sull'oracolo: in caso di discordanza dei risultati sono necessari altri controlli – e un altro oracolo – per stabilire quale dei programmi usati per il test sia quello corretto.
- ***Versioni precedenti***. È un sottocaso del back-to-back in cui il vincolo di indipendenza è rilasciato e si considera come oracolo una precedente versione dello stesso programma.
- ***Programmi esistenti***. Ancora una particolarizzazione del back-to-back in cui, invece di prevedere esplicitamente lo sviluppo di copie funzionalmente identiche, si cerca di recuperare software già realizzato e provato, ad esempio funzionante su macchine o sistemi operativi diversi. È un tipo di oracolo conveniente quando il software prodotto è un porting o la reingegnerizzazione di un sistema già esistente.

La scelta dell'oracolo dipende in gran parte dal tipo di software oggetto del test e dalla situazione propria di un particolare processo di sviluppo. Nella pratica si assiste in generale all'uso di soluzioni basate sull'analisi delle specifiche, nella maggior parte dei casi senza il

supporto dell'esecuzione automatica, vista l'ancor rara applicazione di linguaggi di specifica formali. Le soluzioni back-to-back sono usate quando, essendo possibile adottare come oracolo versioni precedenti o software presistente, la loro applicazione non grava sull'economia del progetto.
Per ridurre i costi dei controlli sono spesso applicate le tecniche di semplificazione dell'oracolo. La loro applicazione diminuisce però, talvolta anche in modo sensibile, l'efficacia dei test. La scelta dell'oracolo, e in particolare il ricorso a soluzioni semplificative, devono essere valutate coerentemente con i criteri adottati nella realizzazione dei test: è inutile applicare criteri sofisticati per poi ridurre l'efficacia dei test con la scelta di oracoli poco affidabili.

3.6 Analisi e terminazione dei test

L'obiettivo immediato di un test è scoprire malfunzionamenti; l'obiettivo reale, quello per cui i test sono a diritto annoverati fra le attività legate alla qualità del software, è garantire che, fino a un certo livello di prova, non si manifestino malfunzionamenti. Il primo obiettivo corrisponde alla visione dello sviluppatore, che vuole eliminare i difetti di produzione, il secondo è nella prospettiva del controllo di qualità: il test deve garantire il raggiungimento di un prefissato livello di qualità.
Da un punto di vista pratico, possiamo tradurre la responsabilità del test nei confronti della qualità del prodotto in due domande; l'attività di test consiste semplicemente nella ricerca di due risposte affermative:

- i test sono rappresentativi del livello di qualità desiderato per il prodotto?
- il prodotto esegue i test senza mostrare malfunzionamenti?

La prima domanda apre la questione della valutazione dei test – in altre parole della qualità dei test rispetto al giudizio di qualità del prodotto che devono esprimere. La seconda è pertinente all'uso dei test per la valutazione del prodotto e in particolare al raggiungimento delle condizioni per cui è possibile terminare l'attività di test.

3.6.1 Valutazione dei test

Lo strumento più noto per valutare in modo assoluto l'efficacia di un test è il concetto di *copertura*. Assoluto perché tiene conto solo del test e del software da controllare, senza prendere in considerazione l'attività di test, le sue relazioni con il processo di sviluppo e, soprattutto i suoi costi. Come per i criteri alla base della progettazione dei test possiamo distinguere fra:

- ***copertura funzionale***, è la percentuale di funzionalità esercitate da un insieme di casi di test calcolata sul numero totale di funzionalità del software;
- ***copertura strutturale***, è la percentuale di elementi strutturali esercitati da un insieme di casi di test calcolata sul numero totale di elementi strutturali del codice del software.

L'uso del concetto di copertura nella valutazione di un test, e quindi nella valutazione della qualità del software, è basato sulla seguente catena di implicazioni: maggiore è la copertura raggiunta e maggiore è l'esercizio a cui il test ha sottoposto il software, quindi maggiore è la probabilità di aver scoperto un più alto numero di malfunzionamenti; di conseguenza è minore il numero dei probabili malfunzionamenti residui, cosa che, infine, implica una miglior qualità del software. In termini di metriche, la copertura, come percentuale, è la misura, la scala di

riferimento va dallo 0% al 100% che, indicando un test che esercita completamente il programma, indica la massima qualità raggiungibile in teoria.
I metodi basati sulla copertura strutturale si dividono, analogamente ai corrispondenti criteri per la progettazione dei test, in metodi basati sul flusso del controllo e metodi basati sul flusso dei dati. Fra i primi citiamo:

- ***copertura dei comandi***; la misura di copertura è data dal rapporto fra il numero di comandi esercitati dal test e il numero di comandi totale;
- ***copertura dei cammini***; la misura di copertura è data dal rapporto fra il numero di cammini percorsi dal test e il numero di cammini totale.

Per i criteri basati sul flusso del controllo è sempre più confermata l'ipotesi che il numero dei malfunzionamenti scoperti cresca in misura meno che proporzionale rispetto al crescere della copertura strutturale. Indagini statistiche hanno confermato che una copertura dell'85% costituisce una soglia ottimale: la quantità di lavoro e di risorse necessari per costruire ed eseguire test con copertura strutturale superiore a questo limite non sono, di solito, compensati dalla maggior quantità di malfunzionamenti scoperti. Fra i metodi basati sul flusso dei dati, citiamo:

- ***copertura delle definizioni***; la misura di copertura è data dal rapporto fra il numero di definizioni di variabili esercitati dal test e il numero di definizioni totale;
- ***copertura degli usi***; la misura di copertura è data dal rapporto fra il numero di usi di variabile esercitati dal test e il numero di usi totali.

Sulla scorta di studi statistici, è stata avanzata la congettura che le misure di copertura basate sul flusso dei dati siano le più indicative circa la qualità di un test. Per dare una giustificazione intuitiva a questa affermazione è sufficiente notare che, in molti casi, i malfunzionamenti in un programma si verificano per particolari valori delle variabili. Una progettazione dei test fondata sul flusso dei dati – e per la quale una misura di copertura simile è più indicata – è più naturalmente portata a individuare i valori critici e a esercitare il software su di essi.
Esiste tuttavia un limite di fondo alla validità delle misure di copertura, siano esse funzionali che strutturali: non tengono conto né della distribuzione né di altre caratteristiche dei valori d'ingresso dei test. Da questo punto di vista, un test, pur dimostrando un'ottima copertura, può risultare molto lontano dal caso medio di uso del software e, di conseguenza, non rivelare malfunzionamenti, che per la loro frequenza o criticità, potrebbero incidere pesantemente sulla qualità del software. È importante quindi ricordare che, come la maggior parte delle metriche, anche quelle applicabili ai test e ispirate alla copertura hanno valore soprattutto su base statistica.
Teoricamente, è possibile usare le misure di copertura, sia funzionali che strutturali, su qualsiasi tipo di test, in modo indipendente dai criteri a cui la sua progettazione è stata ispirata: ogni test infatti esercita sempre almeno un sottoinsieme delle funzionalità e una parte del codice del software. Una misura combinata è inoltre una valutazione sicuramente più completa delle caratteristiche del test. In pratica però l'uso di entrambi i tipi di misura risulta poco economico e in genere si tende ad applicare coperture affini ai criteri che hanno guidato la progettazione dei test. Ad esempio, per valutare la copertura dei cammini è necessaria la costruzione del grafo di flusso, costo che può essere accettabile se giustificato anche ai fini della realizzazione dei test.

Un altro metodo per valutare l'efficacia dei test prende spunto dai principî che ispirano *il test mutazionale*: in poche parole, un test è tanto più efficace quanto più è sensibile ai mutamenti avvenuti in un programma riconoscendo i mutanti dalla sua versione originale.

Alla valutazione dei test basata sui programmi mutanti è associata una misura detta *mutation score* e calcolata come segue. Dato un programma *P* e un test *T* e detto M_P l'insieme dei suoi mutanti, si può ottenere, in base all'esecuzione di *T*, una partizione di M_P in due insiemi; di questi indichiamo con R_{PT} l'insieme che contiene i mutanti di *P* che, sottoposti a *T*, hanno esibito un comportamento diverso da *P*, quelli cioè che *T* ha riconosciuto come diversi da *P*. Il mutation score associato a *T* rispetto a M_P è il rapporto fra la cardinalità di M_P e quella di R_{PT}. In termini di metriche, il mutation score assume valori nell'intervallo [0, 1], con i valori prossimi a 1 che indicano una maggior sensitività del test e quindi una migliore efficacia.
È da notare che il mutation score dipende dal programma e dall'insieme dei mutanti, in particolare sono rilevanti la cardinalità e il modo con cui i mutanti sono generati. Inoltre, l'assunzione che un test con un mutation score prossimo a 1 sia ottimo anche nello scoprire malfunzionamenti è vera, come sempre, solo su base statistica. Come controesempio infatti, è sufficiente pensare a un programma contenente almeno un difetto e a un insieme di mutanti, comunque grande, ma che non contiene il programma corretto, quello cioè senza difetto. Anche in presenza di un mutation score pieno non abbiamo nessuna reale garanzia sulla capacità del test di manifestare il malfunzionamento dovuto a quel difetto.

Glossario

accettazione: l'ultima fase del *ciclo di vita* in cui il prodotto è sottoposto alla valutazione del *committente* (cfr. *collaudo*). La valutazione è basata su un insieme di controlli – i **controlli di accettazione** appunto – che hanno come obiettivo la *validazione* delle *caratteristiche di qualità* del prodotto rispetto ai suoi *requisiti*.
affidabilità: la caratteristica di qualità di un prodotto software relativa alla sua capacità di funzionare correttamente nel tempo. Lo standard ISO 9126 la prevede come una caratteristica di qualità primaria, a sua volta de¬composta in maturità, tolleranza ai guasti, recuperabilità.
α-test: indica un *test* o più in generale una serie di controlli (e non necessariamente *controlli dinamici*) eseguiti su un prodotto software a cura della stessa organizzazione che lo ha sviluppato. Obiettivo dell'α-test è decidere se è possibile rilasciare, almeno in *β-test*, un prodotto. In questa prospettiva, i *test di sistema* propriamente detti fanno parte di un α-test, ma non è detto che un α-test si limiti ai test di sistema.
ambiente di test: è un ambiente controllato che permette il *controllo dinamico* di un *modulo* o di un *sistema software* indipendentemente da fattori esterni che potrebbero influenzarne il comportamento e quindi inquinare i risultati del *test* (cfr. *driver* e *stub*).
anomalia: vedi *difetto*.

back-to-back: soluzione per la realizzazione di un'*oracolo* da usare nell'analisi dei risultati dei *test*. Consiste nel sottoporre allo stesso test due copie funzionalmente identiche di un *programma*. Le copie possono essere indipendenti, una delle due realizzata in funzione del test o recuperata da librerie esistenti, oppure essere una versione precedente dello stesso programma (cfr. *regressione*).
β-test: indica un *test* o più in generale una serie di controlli (e non necessariamente *controlli dinamici*) eseguiti su un prodotto software da organizzazioni diverse da quella che lo ha sviluppato. Scopo di un β-test è far provare il prodotto da un insieme selezionato di utenti prima di rilasciarlo definitivamente. Il β-test non va confuso con i *controlli di accettazione*.
big bang: tecnica di *integrazione* dei *moduli* che consiste nel provare il *sistema software* completo immediatamente dopo aver controllato i singoli moduli. In presenza di *progettazione*, *realizzazione* e *controllo* dei moduli eseguiti a regola d'arte, il risultato atteso è il funzionamento corretto del sistema. In realtà questo è un evento spettacolare e insperato e questa tecnica è utilizzabile solamente per sistemi di limitate dimensioni.
black box: un modo per indicare un *programma* quando se ne vuole considerare la sola *specifica* funzionale, disinteressandosi alla sua realizzazione. A questa visione dei programmi sono ispirati i *criteri funzionali* per la progettazione dei *test*.
bottom-up: vedi *top-down*.
branch: vedi *decisione*.
bug: in inglese "cimice", in americano più generalmente "insetto", indica la causa di un *malfunzionamento* in un *sistema software*; in italiano è spesso tradotto per assonanza con **baco**. Esistono **hardware bug** e **software bug**: la leggenda vuole che il primo bug fosse un hardware bug e, più propriamente, una falena arrostita fra i relé del Mark II di Harvard (che si dice tuttora conservata allo Smithsonian). Con riferimento alla terminologia IEEE un bug è un *difetto*, ovvero ciò che deve essere rimosso per eliminare un malfunzionamento. Con **debugging** si intende propriamente la ricerca dei bug effettuata dinamicamente ispezionando lo stato del programma durante una sua esecuzione controllata; in senso più lato debugging è sinonimo di tutta l'attività di *controllo* del software. Con **debugger** si identifica uno strumento realizzato appositamente per supportare questo tipo di attività.
cammino: (o **path**) con riferimento al *grafo di flusso*, è una successione di nodi che rappresenta uno dei possibili comportamenti del programma in esecuzione. Con **path test** si indicano quei test progettati per coprire il maggior numero dei cammini possibili sul grafo di flusso.
collaudo: è il *controllo* definitivo che un prodotto finito rispetti i *requisiti* del *committente*. È un controllo effettuato da o per conto del committente e le cui modalità sono normalmente definite nel contratto che regola la commessa (cfr. *accettazione*).
comando: (o **statement**) con riferimento al *grafo di flusso*, è una singola istruzione del *programma* a cui corrisponde un nodo del grafo; se l'istruzione è di tipo condizionale darà luogo a una *decisione*. Con **statement test** si indicano quei test progettati per coprire il maggior numero di comandi sul grafo di flusso.
committente: l'organizzazione che commissiona lo sviluppo di un prodotto software. Non tutta l'industria del software è basata su prodotti commissionati e realizzati in versione unica, anzi: acquista sempre maggiore importanza l'industria del software di massa, realizzato per soddisfare un'utenza generica e venduto in milioni di copie. Tuttavia, anche in questi casi, è sempre possibile identificare il committente nel ruolo svolto dal reparto dell'azienda che studia il mercato e definisce i *requisiti* del prodotto (cfr. *fornitore*).
condizione: con riferimento al *grafo di flusso* di un *programma*, è una singola condizione booleana usata in un'istruzione condizionale, dalle condizioni dipendono perciò le *decisioni*.

Con **condition test** si indicano quei *test* progettati per far assumere a ogni condizione del grafo di flusso entrambi i valori di vero e falso.
controllo: la prova a posteriori della qualità di un prodotto, di un servizio o di un processo. Rispetto al rapporto *committente*/*fornitore*, si può distinguere fra **controllo interno**, quando è svolto dal fornitore stesso, e **controllo esterno** quando invece è svolto da o per conto di un'organizzazione esterna al fornitore. Nel caso di prodotti software è possibile distinguere fra **controllo dinamico** e **controllo statico** in base, rispettivamente, alla messa in esecuzione o no del prodotto.
copertura: una misura dell'efficacia di un *test*; è possibile distinguere fra **copertura funzionale**, in base alla percentuale di *funzionalità* del prodotto software esercitate dal test, e **copertura strutturale**, in base alla percentuale di componenti del *grafo di flusso* del programma esercitati dal test.
criterio: nella progettazione dei *test* si distinguono i **criteri funzionali** (detti anche **black-box**), basati solo sulla *specifica funzionale* del software e che quindi mirano a esercitarne soprattutto le funzionalità, e i **criteri strutturali** (detti anche **white-box**) che si basano sulla conoscenza della struttura del *codice* e che hanno l'obiettivo di esercitarlo in tutte le sue parti (cfr. *grafo di flusso*). Nell'ambito invece della decisione di terminare l'attività di test, possono essere adottati **criteri economici** basati sull'esaurimento delle risorse destinate ai controlli, **criteri qualitativi**, basati sul raggiungimento del prefissato grado di qualità del prodotto, e **criteri analitici**, basati sul confronto fra i costi dell'attività di test e i benefici che se ne ottengono in termini di aumento della qualità del prodotto.
debugger: vedi *bug*.
debugging: vedi *bug*.
decisione: (o **branch**) con riferimento al *grafo di flusso* di un *programma*, è una ramificazione nel flusso di controllo tipicamente controllata da un'istruzione condizionale. Con **branch test** si indicano quei test progettati per esercitare ogni decisione sia in un senso che nell'altro.
desk check: il controllo *statico* eseguito leggendo il *codice* e simulando "a mano" l'esecuzione di un *programma*, spesso mirato all'individuazione di un *difetto*. È una tecnica che risale ai primordi della *programmazione*. Sebbene esistano ancora estimatori di questa tecnica, specialmente per l'individuazione di errori algoritmici, essa tende a essere sostituita dall'uso di *debugger* sempre più sofisticati o da tecniche di controllo statico più formali (cfr. *inspection* e *walkthrough*).
difetto: un'imperfezione o una mancanza in un *sistema software* che può dar luogo a un *malfunzionamento*. Un **difetto quiescente** non si manifesta per un lungo periodo perché nascosto in parti del *programma* raramente utilizzate. Detto anche **anomalia** (*fault* nella terminologia dello standard *IEEE*), in *jargon* il difetto è il ben noto *bug*.
driver: in un *ambiente di test* o nella fase di *integrazione* di un *sistema software*, un componente usato per controllare il comportamento di uno o più *moduli*. Un *driver* invoca le funzionalità esportate dai moduli letteralmente "pilotando" la loro esecuzione (cfr. *stub*).
efficienza: la *caratteristica di qualità* legata alla capacità di un prodotto software di svolgere le proprie funzionalità con un consumo minimo di risorse di calcolo. In generale l'*efficienza* è considerata relativamente a una particolare risorsa: tempo macchina, occupazione di memoria, occupazione di disco. Lo standard *ISO 9126* distingue a questo proposito come sottocaratteristiche dell'efficienza l'**efficienza nel tempo** e l'**efficienza delle risorse**.
errore: in generale è una deviazione dal comportamento corretto di un *sistema software*. In una diversa e più precisa accezione (quella definita dalla terminologia *IEEE* nel corrispondente **error**), un errore è la causa di un *difetto*, ovvero una mancanza, di una persona

o di uno strumento, che introduce un difetto in un sistema software: ad esempio, la distrazione o la mal comprensione di un algoritmo.
error guessing: vedi *previsione degli errori*.
esecuzione simbolica: una tecnica a metà fra il *controllo dinamico* e il *controllo statico*, il *programma* viene cioè eseguito ma senza assegnare valori alle sue variabili che, appunto, sono trattate simbolicamente. In teoria, una singola esecuzione simbolica copre tutti i possibili casi di *test*, ma la tecnica è costosa e poco applicabile a casi industriali.
failure: vedi *malfunzionamento*.
fault: vedi *difetto*.
fornitore: l'azienda che sviluppa un prodotto software su commissione. Non sempre, nell'ambito di un *progetto* software, esiste una effettiva distinzione contrattuale fra *committente* e fornitore, in molti casi sono questi ruoli svolti da reparti diversi di una stessa azienda.
funzionalità: una particolare funzione o caratteristica di un prodotto software che lo rende abile a soddisfare una parte dei *requisiti*. In una diversa accezione del termine, la funzionalità è la *caratteristica di qualità* di un prodotto software relativa alla sua capacità di soddisfare i requisiti.
grafo causa-effetto: rappresentazione delle *funzionalità* di un *programma* che descrive i fatti elementari di uscita come risultati di espressioni booleane dei fatti elementari d'ingresso. Può essere usato come strumento di controllo della *specifica* o come *criterio funzionale* per la *progettazione* dei *test*.
grafo di flusso: rappresentazione schematica di un *programma*: a ogni nodo è associato un *comando* mentre gli archi, orientati, rappresentano il legame di sequenzialità fra i comandi. Altri elementi importanti nella struttura di un grafo di flusso sono le *decisioni* che rappresentano le ramificazioni del grafo e le *condizioni* che rappresentano le singole espressioni booleane. I grafi di flusso sono usati per la valutazione della *complessità ciclomatica* di un programma, per la progettazione dei *test* secondo i *criteri strutturali*, per la valutazione della *copertura strutturale* di un test.
gray box: strategia di *test* che mescola *criteri funzionali*, *criteri strutturali* ed *error guessing*; in pratica, è uno degli approcci più comuni alla progettazione dei test.
guasto: vedi *malfunzionamento*.
IEEE: (Institute of Electricaland Electronics Engineers, si legge I-triple-E) un'ente internazionale che si occupa di promuovere la ricerca e di redigere e diffondere standard nei campi dell'elettronica e dell'informatica. Riguardo ai temi di ingegneria del software ricordiamo gli standard **IEEE Std. 729**, un glossario di termini legati a questa disciplina, **IEEE Std. 1008**, che definisce le attività di un'unità di test, **IEEE Std. 829** che definisce lo standard per la documentazione dei test.
inspection: vedi ispezione.
integrazione: *attività* successiva alla *codifica* e al *controllo* dei singoli *moduli* e che consiste nel verificare se i moduli interagiscono correttamente nel rispetto delle loro interfacce e della *specifica funzionale* del *sistema software*. Alla strategia non incrementale di integrazione (cfr. *big bang*) sono preferibili strategie che integrano i moduli a poco a poco, costruendo il sistema secondo approcci *top-down* o *bottom-up* e che, facendo uso di *stub* e *driver* alternano passi d'integrazione a passi di *test*.
ISO: (per "International Organization for Standardization") è un'organizzazione internazionale che si occupa di definire e diffondere standard in tutti i settori produttivi. **ISO 9000** è una serie di standard per la gestione e la garanzia della qualità nei *processi* di produzione industriali. Fra essi lo standard **ISO 9000/3** riguarda in particolare la produzione

di software, mentre **ISO 9126** definisce un modello di *qualità* per i prodotti software. **ISO 8402** è il glossario dei termini inerenti la qualità e **ISO 10011** è lo standard che definisce l'attività di *verifica ispettiva*.

ispezione: analisi di un documento mirata alla scoperta di *difetti* ed *error*, generalmente effettuata sul *codice* o sui *requisiti*. L'ispezione ha sempre obiettivi precisi: ad esempio la ricerca di un *difetto* che provoca un *malfunzionamento* già noto o la *verifica* del rispetto degli *standard di codifica* (cfr. *walkthrough*).

malfunzionamento: un comportamento di un *sistema software* diverso da quello atteso e definito dalle *specifiche*. Per analogia con gli apparati fisici, un malfunzionamento è detto anche **guasto** o, nella terminologia inglese proposta da *IEEE*, **failure** (cfr. *errore* e *difetto*).

manutenzione: fase del *ciclo di vita* che ha l'obiettivo di modificare un prodotto software per eliminare *difetti*, per migliorarne le *caratteristiche di qualità*, per adattare il prodotto ai cambiamenti dell'ambiente operativo. Normalmente svolta quando se ne presenta l'occasione, si parla di **manutenzione adattiva** quando ha lo scopo di aggiornare il prodotto software, di **manutenzione perfettiva** quando vuole migliorarlo, di **manutenzione correttiva** quando nasce dalla necessità di eliminarne difetti.

modulo: un sottosistema di un *sistema software* (cfr. *architettura*). Per la sua genericità, il concetto di modulo è usato sia per definire le singole componenti di un sistema che interagiscono dinamicamente tra loro o, da un punto di vista statico, per descrivere una componente di un *programma* strutturato in procedure e funzioni. Secondo una visione funzionale, un modulo è esternamente visto come una *black box* accessibile solo tramite le *funzionalità* che fornisce al sistema. Un modulo deve essere abbastanza semplice da poter essere realizzato da un solo programmatore.

mutante: un *programma* in cui sono state introdotte intenzionalmente modifiche che lo rendono non conforme alle *specifiche*. Il **test mutazionale** è basato sul confronto dei risultati di test dei mutanti con quelli del programma originale, è usato per localizzare un *malfunzionamento* o per valutare l'efficacia dei test. In quest'ultimo caso, il **mutation score** è una misura di quanto un test sia sensibile alle modifiche introdotte nel programma.

oracolo: entità in grado di produrre il risultato corretto dell'esecuzione di un *programma* usata per produrre i dati con cui confrontare i risultati dei *test*.

path: vedi *cammino*.

portabilità: la *caratteristica di qualità* che indica la capacità di un prodotto software di essere trasportato da un ambiente a un altro. Si prende in considerazione il problema di convertire il software affinché funzioni su una piattaforma hardware, su un *sistema operativo* o, più in generale, in un ambiente operativo diverso da quello per cui è stato in origine progettato e realizzato.

previsione degli errori: una "non tecnica" di *test* che, basata sull'esperienza e sull'intuito di chi la applica, permette di trovare *difetti* ipotizzando gli errori che ne sono la causa. Molto usata quando i *debugger* erano una rarità o un lusso, è tuttora molto apprezzata. È sempre, almeno inconsciamente, alla base delle scelte circa il tipo di tecnica di test da adottare.

profilo operazionale: il contesto tipico di utilizzo di un'applicazione, dato dalle caratteristiche dell'ambiente operativo medio e dall'insieme di dati d'ingresso più probabili.

prova formale: una tecnica di controllo statico basata sulla dimostrazione dell'equivalenza fra un *programma* e la sua *specifica funzionale*.

qualità: l'insieme delle caratteristiche di un prodotto software che ne influenzano la capacità di soddisfare le specifiche esigenze dichiarate nei *requisiti*. Molte sono le proposte di razionalizzazione del concetto di qualità tramite la definizione di un modello di qualità che elenchi e definisca le caratteristiche di qualità di cui è importante tenere conto nella

valutazione di un prodotto. In inglese indicate genericamente come “ilities”, le caratteristiche di qualità previste, ad esempio, dal modello di qualità descritto dallo standard *ISO 9126* sono: *funzionalità*, *affidabilità*, *usabilità*, *efficienza*, *manutenibilità*, *portabilità*. In un contesto aziendale, il sistema qualità è l’insieme delle risorse e delle politiche che un’organizzazione dedica esplicitamente al conseguimento della qualità nei prodotti o nei servizi che essa offre.

regressione: la possibilità che un *programma*, in seguito a una modifica dovuta a un intervento di *manutenzione* o a un *processo* di sviluppo incrementale, peggiori le proprie funzionalità. Il confronto di *versioni* successive dello stesso programma prende il nome di **test di regressione**.
sandwich: la strategia per integrazione dei *moduli* di un *sistema software* che fonde le strategie *top-down* e *bottom-up*; di fatto, la più usata in pratica.
statement: vedi *comando*.
stub: in un *ambiente di test* o nella fase di *integrazione* di un *sistema software*, un componente che simula il comportamento di uno o più *moduli*. Uno stub (letteralmente “mozzicone”) è usato per esportare funzionalità (eventualmente semplificate) invocate dai moduli in analisi (cfr. *driver*).
test: una tecnica di *controllo* che ha l’obiettivo di rivelare eventuali *malfunzionamenti* analizzando il comportamento di un programma mentre è eseguito in un ambiente controllato (cfr. *ambiente di test*) e su insiemi predefiniti di dati di ingresso; il termine inglese test è molto più usato della dizione italiana *controllo dinamico*. In una differente accezione, con test si identifica il particolare insieme di dati e procedure usati per verificare un programma. Il test è usato come tecnica di controllo durante tutto il processo di sviluppo: è propria di questo contesto infatti la distinzione fra test di modulo, test d’integrazione e test di sistema.
top-down: un modo di affrontare la decomposizione di un problema che prevede di andare da una visione astratta della realtà alle sue entità fisiche, da una descrizione generale a una particolare, da un punto di vista ampio a uno dettagliato. Il termine **bottom-up** indica il tipo di procedimento inverso. In ingegneria del software esistono tecniche di *analisi*, di *progettazione* e di *integrazione* ispirate sia ai procedimenti top-down che bottom-up.
usabilità: la *caratteristica di qualità* di un prodotto software relativa alla sua possibilità di essere impiegato produttivamente da una particolare utenza. Lo standard *ISO 9126* prevede l’usabiltà come caratteristica di qualità primaria, a sua volta decomposta in *comprensibilità*, *apprendibilità*, *operabilità*.
validazione: l’attività di *controllo* che risponde alla domanda “stiamo realizzando il prodotto corretto?”, che confronta un risultato dello sviluppo, anche intermedio, con i requisiti iniziali del prodotto.
verifica: l’attività di *controllo* che risponde alla domanda “stiamo realizzando correttamente il prodotto?”, che mira a scoprire i *difetti* introdotti durante un passo di lavorazione analizzando due prodotti intermedi successivi o controllando il corretto svolgimento di un’attività di sviluppo (vedi in *ispezione*, *verifica ispettiva*).
walkthrough: una tecnica di analisi formale del *codice* che prevede due fasi: nella prima gli ispettori esaminano il codice simulandone l’esecuzione in assenza del gruppo che l’ha sviluppato, nella seconda fase i due gruppi discutono i problemi individuati. Come l’*inspection*, il walkthrough, nell’ambito del processo software, è un’attività prevista e documentata, ma ha obiettivi più generali mirando a favorire una collaborazione costruttiva fra i gruppi di sviluppo e di ispezione.
white box: vedi *criterio strutturale*.

Bibliografia

[CD98] G.A. Cignoni, P. De Risi, "Il test e la qualità del software", Ed. Il Sole 24 Ore, Milano, 1998.

[CMS86] E.M.Clarke, E.A. Emerson, A.P. Sistla, *Automatic Verification of Finite--State Concurrent Systems Using Temporal Logic Specification*, ACM Transaction on Programming Languages and Systems, 8(2):244-263, 1986.

[ZHM97] Hong Zhu, Patrick A. V. Hall, John H. R. May, Software unit test coverage and adequacy, ACM Computing Surveys Volume 29 , Issue 4, 1997.

[IEEE90] IEEE 610.12 Standard Glossary of Software Engineering Terminology.

[Mye79] G.J. Myers, *The art of software testing*, John Wiley & Sons, 1979.

[PM07] Mauro Pezzè and Michael Young, *Software Test and Analysis: Process, Principles, and Techniques* John Wiley & Sons, 2007.