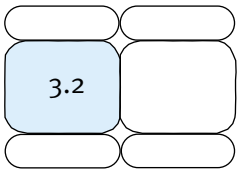
A CACCIA DEGLI ERRORI DEL SOFTWARE

Alberto Sillitti

In tutti i campi dell'ingegneria la fase di test o di collaudo di un sistema è basata su pratiche assodate ed esistono regole ben precise per collaudare correttamente un sistema fisico. Nell'area dell'ingegneria del software, però, collaudare un sistema presenta problemi totalmente diversi da quelli tradizionali, rendendo inapplicabili tutte le pratiche utilizzate negli altri settori dell'ingegneria. Il presente articolo si propone di presentare al lettore i problemi connessi con il testing del software nonché approcci e strumenti che si possono usare per effettuarlo.



1. INTRODUZIONE

Collaudare un prodotto software significa verificare che il suo comportamento sia aderente alle specifiche di progetto. In altre parole, significa analizzare il prodotto per individuare eventuali difetti e, successivamente, eliminarli. Con la parola *difetto*, si intende genericamente un errore nel sistema, ma esiste una classificazione più accurata definita dall'IEEE [14] e riassunta nel riquadro sui **tipi di errori nel software**. Il collaudo comprende due aspetti ben distinti:

1. verifica che tutti i requisiti siano implementati correttamente secondo le specifiche;
2. identificazione e conseguente eliminazione dei difetti presenti nel codice.

Il primo aspetto consiste nel verificare che il software implementato contenga tutte le funzionalità espresse nelle specifiche e che siano implementate correttamente. In questo caso si verifica il sistema nel suo insieme analizzandone il comportamento (per esempio interazioni tra moduli o sotto-sistemi) ma non scendendo nei dettagli (ad esempio non si testano singole classi o funzioni). Questo tipo di verifica, però, non riguarda solo i requisiti funzionali di un sistema ma anche quelli non funzionali come prestazioni, affidabilità, sicurezza, usabilità ecc..

Il secondo aspetto si focalizza nella verifica che ogni porzione di codice sia corretta, quindi si analizza in dettaglio ogni singola classe o funzione del sistema.

Come si vedrà nel paragrafo 3, le tecniche utilizzate per testare questi due aspetti sono diverse, anche se alcune possono essere applicate in entrambi i casi.

Quando si parla di *testing*, alcune volte ci si riferisce indifferentemente al *testing* o al *de-*

Tipi di errori nel software

Secondo la terminologia definita dall'IEEE [14] esistono diversi tipi di errori nel software:

- **Error**: è un errore umano nel processo di interpretazione delle specifiche, nel risolvere un problema o nell'usare un metodo.
- **Failure**: è un comportamento del software non previsto dalle specifiche.
- **Fault**: è un difetto del codice sorgente (detto anche *bug*).

bugging, ma questi termini non hanno lo stesso significato e identificano due attività molto diverse tra loro. Il *testing* consiste nell'individuare eventuali difetti che producono un comportamento non corretto del sistema, mentre il *debugging* consiste nella loro rimozione dal codice. Di conseguenza, il *debugging* utilizza i risultati del *testing* per individuare i difetti da eliminare.

Il *testing* è la fase dello sviluppo del software più complessa. Infatti, collaudare un prodotto software moderno non significa solamente verificare che gli algoritmi implementati siano corretti, ma anche che il sistema in esame interagisca correttamente con altri sistemi (i quali possono a loro volta essere difettosi).

Collaudare un sistema software è completamente diverso dal collaudare un sistema fisico. Quando si effettua il collaudo di un ponte, per esempio, non ci sono dubbi che, se questo è in grado di sostenere un camion da 20 tonnellate, sarà anche in grado di sostenere il peso di un uomo di 70 chili. In questo tipo di sistemi, il collaudo viene fatto verificando i limiti fisici con un numero di prove molto limitato: se un ponte regge 20 t, non è necessario verificare la sua resistenza con carichi da 10, 5 o 1 tonnellate.

Nel software, purtroppo, non è possibile fare un ragionamento analogo e non esistono limiti fisici da poter verificare. In realtà, quest'ultima affermazione non è del tutto corretta, in quanto anche nel software esistono dei limiti ma questi non sono altro che tutti i possibili input del programma (sia input validi che non). Inoltre, se si verifica che un sistema funziona correttamente per un valore di input "grande", questo non implica che il sistema si comporti bene anche per valori più "piccoli" e, se sono state eseguite 100 prove diverse, non è detto che la 101-esima abbia successo. In generale, nel software, ogni valore di ingresso si comporta come un caso a se. Quindi, per avere la certezza che un sistema software sia corretto sarebbe necessario verificare il suo comportamento con tutti i valori possibili di input. Sfortunatamente, un approccio del genere è impraticabile per qualunque sistema software, anche se semplice. Si pensi solamente che per verificare in modo esaustivo l'addizione tra due numeri interi a 16 bit, bisognerebbe considerare tutte le $2^{16} \times 2^{16} = 2^{32}$ coppie di valori di input.

A complicare ulteriormente la situazione si tenga presente che un sistema software interagisce sempre con altri sistemi software (almeno con il sistema operativo) e hardware (almeno la macchina che lo esegue) ed è estremamente difficile collaudare un prodotto con tutte le combinazioni di sistemi e periferiche che l'utente finale potrebbe utilizzare. Quindi collaudare in modo esaustivo un sistema software è teoricamente possibile, ma richiede di provare tutte le possibili configurazioni. Ovviamente, un approccio del genere è impraticabile a causa del tempo (e quindi costi) che richiederebbe.

2. TESTING E MERCATO DEL SOFTWARE

Negli ultimi anni ci sono stati cambiamenti rilevanti nel software, sia dal punto di vista del produttore che da quello del consumatore (l'utente) e, come in molti sistemi economici, la domanda ha guidato la produzione. In particolare, il profilo dell'utente medio è cambiato notevolmente, modificando, di conseguenza, la domanda: si è passati gradualmente da un utente/programmatore con conoscenze informatiche avanzate ad un utente con un livello di scolarizzazione informatica piuttosto bassa. Questa evoluzione ha modificato la domanda di prodotti software: si è passati da prodotti difficili da usare per i quali era necessario seguire un corso sul loro utilizzo, a prodotti semplici da usare che chiunque può maneggiare, per lo meno per le funzionalità di base [20].

Allo stesso tempo l'utente è diventato sempre più esigente ed il numero delle funzionalità richieste è cresciuto, così come le richieste di integrazione e di interoperabilità con altri prodotti. Questi requisiti, insieme alla facilità d'uso, hanno contribuito ad aumentare notevolmente il numero di righe di codice che devono essere scritte per presentare un prodotto sul mercato e la sua complessità [5], con ripercussioni su tutto il ciclo di sviluppo e, in particolare, sul *testing*.

L'intero ciclo di produzione del software è cambiato e si è adattato alle nuove esigenze, sia pure con qualche difficoltà [8, 10]. Un contributo fondamentale è stato dato dai linguaggi di programmazione: si è passati

dallo scrivere applicazioni con linguaggi a basso livello (linguaggio macchina ed assembly), alla utilizzazione di linguaggi evoluti (C/C++, Java, C# ecc.) e, da sola, questa evoluzione ha aumentato la produttività dei programmatori [23] e ha ridotto il numero di errori nel codice [17]. Questo fattore, però, non è sufficiente a garantire la produzione di sistemi di qualità. Infatti, collaudare un sistema software composto da centinaia di migliaia o milioni di righe di codice è un'attività complessa e costosa e, per questo motivo, si cerca di disciplinare ed automatizzare il più possibile il collaudo in modo da individuare il maggior numero di errori al costo più basso.

A riprova della complessità del testing, ogni giorno è possibile leggere sul web articoli che evidenziano nuovi problemi in diversi prodotti software. Questo non significa (generalmente) che i produttori di software mettano sul mercato sistemi non collaudati ma che, come già rilevato, collaudare un sistema è un'attività estremamente complessa e alcuni difetti possono sfuggire ai test effettuati. Inoltre, problemi di questo tipo non si verificano solamente in sistemi commerciali ma anche in altri tipi di sistemi, come quelli dedicati ad applicazioni spaziali in cui il costo del software per linea di codice è altissimo (anche 100 volte il costo di un software commerciale) ed il collaudo incide sul costo in modo considerevole [6, 18]. Nonostante tutto, anche in sistemi altamente collaudati come questi, è possibile che si trovino errori una volta in produzione (riquadro su **esempi di errori nel software**).

3. TECNICHE DI TESTING

Esistono diverse tecniche che permettono di scegliere un sottoinsieme di valori di input con i quali collaudare un sistema, tenendo sempre presente che, in ogni caso, tali tecniche non garantiscono che il sistema si comporti correttamente per qualunque insieme di parametri di ingresso. Per esempio, se un sistema si comporta correttamente per un certo insieme di test, questo potrebbe significare sia che il sistema è corretto, sia che i test sono inadeguati ad evidenziare gli errori.

Esempi di errori nel software

Sito web di riferimento: http://www5.in.tum.de/~huckle/bugse.html

Ariane 5

Nel 1996 il vettore europeo per trasporto Ariane 5 è esploso al suo primo lancio ufficiale, pochi secondi dopo aver lasciato la rampa di lancio. Il razzo è esploso a causa di un difetto nel software di navigazione, originariamente progettato per l'Ariane 4, il cui software era stato riutilizzato a causa dell'affidabilità dimostrata nelle precedenti missioni. In particolare, l'errore deriva dal fatto che la precisione di alcune misure effettuate nell'Arianne 4 era di 16 bit, mentre quelle effettuate nell'Arianne 5 di 64 bit. L'errore consisteva nel fatto che non si è provveduto ad aumentare la memoria allocata per alcune di queste variabili. Il danno è stato di 1 miliardo di euro. (http://www.around.com/ariane.html e http://www.rvs.uni-bielefeld.de/publications/Incidents/DOCS/ComAndRep/Ariane/Esa/ariane5/COPY/ariane5rep.html)

Sonda Mariner 1

Nel 1962 la sonda Mariner 1 della NASA sbaglia traiettoria subito dopo il lancio e viene distrutta per evitare che ricada al suolo. L'incidente è stato causato da un errore nel software, un singolo carattere sbagliato all'interno del codice sorgente. In particolare, l'errore consisteva nell'aver inserito un punto invece di una virgola all'interno di un costrutto *do-loop* scritto in Fortran, causando un'esecuzione errata di un algoritmo. (http://www5.in.tum.de/~huckle/bugse.html#mariner)

Radioterapia

Tra il 1975 ed il 1987, una macchina per radioterapia ha provocato la morte di 4 persone a causa di diversi errori nel software di controllo. Il sistema non dosava correttamente le radiazioni sottoponendo i pazienti a dosaggi letali. (http://courses.cs.vt.edu/~cs3604/lib/Therac_25/Therac_1.html)

Missili Patriot

Nel 1991, durante la prima Guerra del Golfo, il sistema antimissile Patriot intercetta molti missili iracheni ma i fallimenti sono frequenti. Le mancate intercettazioni sono dovute ad un errore nel software che introduce un errore che si accumula durante il funzionamento della batteria di missili. In particolare, l'errore consiste nel fatto che il sistema utilizzava operazioni aritmetiche a virgola fissa a 24 bit e frequenti moltiplicazioni per 1/10, valore che in binario è periodico. Il troncamento a 24 bit di questo valore causava un errore che si accumulava nel tempo. Nonostante il sistema fosse stato progettato per funzionare per un limitato numero di ore, spesso veniva tenuto acceso per molto più tempo, quindi l'errore accumulato a causa del mancato reset del sistema poteva diventare troppo grande. Questo malfunzionamento ha causato decine di morti tra i soldati americani. (http://www.fas.org/spp/starwars/gao/im92026.htm)

In genere, possiamo raggruppare le tecniche in due categorie:

1. *black-box testing*;
2. *white-box testing*.

1. Il *black-box testing* consiste nel considerare il prodotto o il sottosistema software da collaudare come una scatola nera: si inviano dei dati di input e si verifica che l'output sia coerente con le specifiche. Questo tipo di verifica è effettuata senza conoscere la struttura interna del sistema da collaudare e, per questo motivo, i valori che vengono utilizzati come input sono scelti a caso o in base a caratteristiche delle specifiche [19].

Per esempio, si consideri un algoritmo che calcola la radice quadrata di un numero intero. Per effettuare un *black-box testing* si può procedere inviando al sistema numeri a caso, oppure si osservano le specifiche e si considerano i valori critici per la funzione radice quadrata. In quest'ultimo caso, si potrebbe verificare il corretto comportamento investigando i limiti del dominio della funzione, quindi verificando il comportamento del sistema con valori come 0, 1, 2, -1. Questo approccio non garantisce che l'implementazione sia corretta, ma si può supporre che, se il sistema si comporta correttamente per valori critici, il comportamento sarà analogo per gli altri valori.

Nel *black-box testing*, si conoscono in dettaglio solo le specifiche e non i dettagli dell'implementazione, quindi non è possibile verificare il comportamento per valori critici per lo specifico algoritmo adottato o per la specifica implementazione, ma solo valori critici deducibili dalle specifiche.

2. Il *white-box testing* consiste nel verificare il corretto comportamento di un sistema software avendo la possibilità di osservarne l'implementazione e, quindi, è possibile effettuare una serie di verifiche più accurate rispetto al *black-box testing*. In generale, è possibile dividere questi test in due categorie:

1. verifiche formali;
2. testing informale.

Verificare formalmente un sistema software significa dimostrare matematicamente la correttezza di un determinato segmento di codice [19]. Per fare questo è necessario descrivere in un qualche linguaggio formale sia le specifiche che il codice. Se non vengono fatti errori nella traduzione delle specifiche e del codice nel linguaggio formale scelto, questo tipo di test può essere eseguito automaticamente e garantisce la correttezza del segmento di codice analizzato (a meno di errori nel sistema automatico di verifica). Purtroppo l'applicazione di questo tipo di verifiche presenta alcuni svantaggi, tra cui:

- ❑ sono richieste abilità che vanno al di là di quelle possedute da un normale programmatore;
- ❑ richiede molto tempo per le traduzioni in linguaggio formale, che devono essere fatte manualmente.

Tutto ciò fa si che il *testing formale* sia estremamente costoso e, quindi, applicabile solo su una piccola parte del codice. Di conseguenza nei normali sistemi commerciali, test di questo tipo sono fatti raramente e solo per sezioni di codice molto limitate e critiche, come il sistema di schedulazione dei processi di un sistema operativo.

Il *testing informale* è di gran lunga quello più utilizzato e consiste nel collaudare il sistema con un limitato numero di valori di ingresso, scelti opportunamente osservando la struttura del codice sorgente. La scelta dei valori di input da utilizzare per il collaudo è quindi estremamente importante e sono state elaborate numerose tecniche per selezionare questi valori. Alcune delle tecniche più note sono le seguenti:

- ❑ *Structured Basis Testing*;
- ❑ *Data-Flow Testing*;
- ❑ *Equivalence Partitioning*;
- ❑ *Boundary Analysis*;
- ❑ *Classes of Data*.

Lo *Structured Basis Testing* consiste nel predisporre dei test con valori di input che garantiscano che ogni istruzione del codice sorgente venga eseguite almeno una volta. Questo può essere fatto in diversi modi: uno dei più efficaci consiste nel considerare tutti i possibili percorsi logici all'interno del programma costruendo dei test che li verifichino. In questo caso, il numero minimo di test necessari è dato dal numero di percorsi logici di esecuzione presenti all'interno del programma.

Per chiarire i diversi approcci alla definizione dei test, si fa uso di piccole porzioni di codice scritte in C. La scelta di questo linguaggio è dovuta alla sua diffusione sia in ambiti accademici che industriali.

Si consideri il seguente segmento di codice:

```
Istruzione1;
Istruzione2;

if (condizione) {Istruzione3;}
  else {Istruzione4;}

Istruzione5;
```

In esecuzione, i percorsi possibili sono due:
1. Istruzione1, Istruzione2, Istruzione3, Istruzione5;
2. Istruzione1, Istruzione2, Istruzione4, Istruzione5.
Per verificare entrambi i percorsi è necessario costruire almeno due test: uno per il quale la condizione all'interno dell'if sia vera ed uno per il quale sia falsa.
Il *Data-Flow Testing* consiste nel collaudare ogni coppia assegnazione-uso di variabile. Questo può essere fatto considerando ogni combinazione di assegnazione ed uso di variabile nel flusso logico del programma.
Si consideri il seguente segmento di codice:

```
if (condizione) {a = 1;}
  else {a = 2;}

b = a;
```

In questo caso le coppie assegnazione-uso di variabile sono due:
1. a = 1, b = a
2. a = 2, b = a
Per verificare entrambe le coppie è necessario costruire almeno due test: uno per il quale la condizione all'interno dell'if sia vera ed uno per il quale sia falsa.
L'*Equivalence Partitioning* è una tecnica per ridurre il numero di test necessari. Il metodo si basa sulla seguente considerazione: se per due valori di input si ottiene lo stesso errore, allora è sufficiente considerare nei test solo uno dei valori. In questo modo è possibile ridurre il numero di test, scartando quelli ridondanti, eseguendo e mantenendo solo quelli utili. Questo approccio, dopo la prima esecuzione dei test, l'individuazione e rimozione di quelli ridondanti, presenta due vantaggi:
1. si risparmia tempo nelle successive esecuzioni dei test. Spesso i test vengono eseguiti migliaia di volte durante lo sviluppo, quindi il risparmio di tempo è evidente;
2. si risparmia tempo per la manutenzione dei test. Accade spesso che, a causa di una modifica nel sistema, anche i test debbano essere aggiornati, quindi meno test da modificare significa meno tempo speso in questa attività.
Questa tecnica è particolarmente utile nel caso di sistemi complessi in cui individuare i test ridondanti durante la loro scrittura non è un compito semplice.
La *Boundary Analysis* consiste nell'analisi delle condizioni limite dei costrutti condizionali, delle funzioni, degli array, ecc. e queste informazioni sono utilizzate per progettare test che verifichino il corretto comportamento del sistema in queste condizioni limite.
Si consideri il seguente segmento di codice:

```
if (i > 0) {...}
  else {...}
```

In questo caso è opportuno collaudare il codice per i = –1, 0, 1 esplorando, quindi, il comportamento del sistema per valori che potrebbero essere critici per la correttezza del sistema.
In questo esempio è presente una sola variabile e, quindi, l'analisi risulta estremamente semplice. Le cose si complicano nel caso in cui il limite non sia espresso su una singola variabile ma su una composizione. Per esempio, si consideri il seguente segmento di codice:

```
if (a*b > 0) {...}
  else {...}
```

In questo caso, il numero di test necessari per esplorare il comportamento del sistema è maggiore e bisogna anche tenere in considerazione gli errori che si possono verificare prima della valutazione dell'espressione booleana. Per esempio, se gli operandi a e b sono sufficientemente grandi la moltiplicazione può causare un overflow che non viene gestito correttamente nel segmento di codice presentato.
Per creare test efficaci può essere utile considerare le *Classes of Data*: un insieme di input

corretti e uno di input non corretti la cui selezione è altamente dipendente dal tipo di sistema considerato. Per quanto riguarda l'insieme degli input corretti, si possono considerare input validi ma che potrebbero presentare problemi. Per esempio, si potrebbe verificare il comportamento di un sistema anagrafico inserendo in input un nome contenente uno spazio (esempio, "De Rossi"). Questo è sicuramente un input valido, ma la presenza di uno spazio potrebbe creare problemi ad alcune implementazioni. Per quanto riguarda l'insieme degli input non corretti, invece, si possono considerare diversi tipi di errori: un tipo molto frequente è l'inserimento di un dato di un certo tipo al posto di un altro (per esempio, l'inserimento di una stringa di caratteri al posto di un valore numerico o di una data) o l'inserimento di valori che potrebbero generare qualche overflow durante l'esecuzione.

Tutte queste tecniche sono utili per individuare un numero ristretto di test utili ma, ad ogni modo, il bravo collaudatore usa anche la propria esperienza per progettare test che potrebbero portare all'individuazione di errori. La maggior parte di questi test sono basati su una profonda conoscenza dei sistemi di sviluppo utilizzati e degli errori di programmazione comuni, come l'uso dei puntatori in C/C^{++} o l'overflow dei contatori nei cicli.

In molti casi, la scelta di opportuni valori di input non è sufficiente ed accade spesso che la sequenza delle operazioni svolte assuma un ruolo fondamentale nell'individuazione di potenziali errori nel codice. In questi casi si analizzano gli algoritmi e si cercano di individuare sequenze di input che potrebbero evidenziare problemi.

Infine, sia per la generazione di valori input sia per la generazione di sequenze, sono usati frequentemente generatori automatici che creano sequenze semi-casuali in base ad una serie di parametri impostati dall'utente. L'utilizzo di questi generatori è estremamente utile nel caso in cui il sistema da testare sia particolarmente complesso ed un'analisi manuale sia eccessivamente complessa.

Le tecniche di testing illustrate in questa sezione e nel riquadro su **tecniche di testing** sono utilizzate sia per verificare che i requisiti siamo implementati secondo le specifiche, sia per identificare errori in ogni porzione di codice. In particolare, per verificare i requisiti si usa spesso il *black-box testing* e le tecniche maggiormente utilizzate sono le seguenti: *integration testing*, *system testing*, *regression testing*. Per verificare che tutto il codice sia corretto si usa spesso il *white-box testing* e le tecniche più utilizzate sono: *unit testing*, *integration testing*, *regression testing*. Inoltre sono utilizzate tutte le tecniche di test formale ed informale.

Accanto a queste tecniche generali utilizzabili in quasi tutti i contesti, esistono tecniche adatte a domini applicativi specifici. Per esempio, testare sistemi *real-time* [9] e sistemi *embedded* [7] richiedono l'applicazione di tecniche specifiche. Nel caso dei sistemi real-time, alcune delle tecniche utilizzate sono:

- *load testing*;
- *non-intrusive testing*.

Il *Load Testing* consiste nel testare un sistema software simulando accessi multipli e verificarne il corretto comportamento. Inoltre, nel caso dei sistemi *real-time*, si verifica anche che le operazioni vengano eseguite entro un tempo massimo.

Esistono diversi metodi per simulare gli accessi ad un sistema, come la generazione ca-

Tecniche di testing

- **Unit testing**: il test unitario è la verifica di una singola funzionalità del sistema in esame. Lo sviluppatore stesso scrive questo tipo di test per verificare la correttezza del codice prima di rilasciarlo agli altri componenti del team di sviluppo. Normalmente, questo tipo di test è automatizzato e viene eseguito ogni volta che si modifica una singola funzionalità del sistema.
- **Integration testing**: il test di integrazione verifica che tutte le parti di un sistema funzionino correttamente. Questo tipo di test viene effettuato per verificare che il codice nuovo non interferisca con le funzionalità già presenti e funzionanti di un sistema. Normalmente, questo tipo di test è automatizzato e viene eseguito periodicamente (ad ogni integrazione, ogni notte o ogni settimana, a seconda dei casi).
- **System testing**: il test di sistema consiste nel verificare l'intero prodotto per il rilascio di una versione al cliente. Il test comprende la verifica di tutte le componenti del prodotto che può comprendere più sistemi.
- **Regression testing**: il test di regressione è una ripetizione dei test per verificare che il sistema in esame non regredisca. In altre parole, si verifica che ogni cambiamento introdotto nel codice non introduca nuovi errori o reintroduca errori che erano stati corretti in precedenza. Normalmente, questo tipo di test è automatizzato e viene eseguito ogni volta che viene apportata una modifica al codice.

suale di sequenze di operazioni valide o invalide eseguite da più utenti simulati, l'utilizzo di sequenze di operazioni comuni o ricavate monitorando l'utilizzo di sistemi simili ecc..
Nel caso il sistema venga testato utilizzando un numero di accessi contemporanei superiore a quello normale si parla di *Stress Testing* e lo si utilizza per verificare il comportamento del sistema in condizioni inusuali o di emergenza.
Il *Non-intrusive Testing* non è una tecnica di testing vera e propria ma un approccio al testing dei sistemi *real-time*. Infatti, visto che in questi sistemi le prestazioni hanno una importanza fondamentale, tutti i test devono tenere conto delle prestazioni del sistema e non alterare le prestazioni del sistema durante le fasi di test. Questo implica che i test (ed anche il sistema da testare) devono essere progettati in modo da permettere la verifica senza compromettere le prestazioni.
Nel caso dei sistemi *embedded*, alcune delle tecniche utilizzate sono:

- *Hardware/Software Integration Testing*;
- *Resource-constrained Testing*.

L'*Hardware/Software Integration Testing* è un test di integrazione che non coinvolge solamente il software ma anche l'hardware su cui l'applicazione dovrà essere eseguita. In questo caso è possibile fare due tipi di test di integrazione: utilizzando un simulatore software dell'hardware oppure utilizzando l'hardware reale. Questo tipo di test è estremamente importante perché una delle difficoltà maggiori nello sviluppo di sistemi *embedded* è verificare che l'intero sistema (composto sia da hardware sia software) funzioni in modo corretto.
Anche il *Resource-constrained Testing* non è una tecnica di testing vera e propria ma un approccio al testing, in questo caso dei sistemi *embedded*. I sistemi ed i test devono essere progettati in modo tale da permettere la verifica delle caratteristiche di un sistema in un ambiente che è spesso caratterizzato da risorse (memoria, capacità computazionale ecc.) limitate.
Testare il software non significa solamente verificare la correttezza del codice. Esistono diverse tecniche applicabili per verificare la correttezza di ciò che viene fatto durante le diverse fasi del ciclo di vita. Alcune delle tecniche di *testing* più usate sono: *User Acceptance Testing, Walkthrough* ed *Inspections*.
Lo *User Acceptance Testing* consiste nel verificare, insieme all'utente, che il prodotto sviluppato soddisfi i requisiti e sia utilizzabile nell'ambiente del cliente. Questo tipo di test permette di verificare la correttezza dell'analisi.
Il *Walkthrough* consiste nell'effettuare un meeting informale per condividere informazioni con gli altri membri del team di sviluppo e/o valutare il progetto o il codice prodotto. Questo tipo di verifica viene usata spesso per analizzare ciò che è stato fatto durante le fasi di progetto o di codifica.
Le *Inspections* consistono in una formalizzazione del *Walkthrough*. Tipicamente viene effettuato da un team di 3-8 persone e lo scopo è quello di scovare qualunque tipo di errore. Questo tipo di verifica richiede un'attenta fase di preparazione prima del meeting vero e proprio e la scrittura di un report finale sul risultato dell'operazione. Anche in questo caso le fasi a cui viene applicato sono, tipicamente, progetto e codifica.
La maggior parti delle tecniche fin qui esposte riguardano le proprietà funzionali di un sistema software. Per testare le proprietà non funzionali esistono tecniche specifiche tra le quali le seguenti: *Performance Testing, Usability Testing, Reliability Testing, Security Testing, Interoperability Testing, Compatibility Testing* ecc.. Se si escludono i test relativi alle prestazioni, gli altri test sono difficili da progettare ed eseguire perché non è facile definire alcune delle proprietà coinvolte come, ad esempio, l'usabilità o la sicurezza di un sistema. Inoltre, solo alcuni di questi test possono essere automatizzati ed eseguiti ripetutamente durante tutto lo sviluppo. Per esempio, testare la sicurezza di un sistema potrebbe richiedere sia la scrittura di alcuni test automatici che verificano la resistenza ad alcuni tipi di attacchi, ma anche l'esecuzione manuale di altri tipi di attacchi eseguiti da esperti di sicurezza tramite strumenti automatici e non.

4. PROBLEMI DI TESTING

Se, in generale, il *testing* non è in grado di dimostrare l'assenza di errori, allora come

si fa a sapere se un codice è collaudato a sufficienza? La risposta a questa domanda non è univoca e dipende da innumerevoli fattori. Quello che accade nella pratica è che circa il 50% del tempo impiegato per sviluppare un sistema software commerciale viene speso per i test [18], comprendendo in tale attività sia il *testing* sia il *debugging* del codice. Inoltre, questo dato si riferisce a quello che accade nella realtà, non al tempo che sarebbe opportuno dedicare al collaudo. In genere, le considerazioni che si dovrebbero fare per sapere se un sistema è sufficientemente corretto da essere messo sul mercato sono le seguenti:

- Si potrebbe supporre che la distribuzione degli errori nel codice sia pressoché uniforme, ma ciò non è vero ed esistono numerosi studi che evidenziano che circa l'80% degli errori si trovano nel 20% del codice [6, 11, 13, 25] e addirittura il 50% degli errori sono concentrati nel 5% del codice [15]. Da questi dati ne segue che, se si è in grado di individuare le parti di codice più soggette ad errori, un collaudo accurato e mirato è in grado di ridurre notevolmente la difettosità di un prodotto, contenendo sia i tempi che i costi. Normalmente, un programmatore esperto è in grado di individuare le parti più critiche di un sistema a cui ha lavorato, quindi questa informazione può essere utilizzata per migliorare notevolmente un prodotto.
- Il numero di errori che ci si può aspettare di trovare in un prodotto è fortemente dipendente dal processo di produzione utilizzato [5, 13, 15] e, in media, questo numero è compreso tra 1 e 25 errori per migliaio di righe di codice.

Utilizzando queste informazioni è possibile ipotizzare il numero totale di errori presenti in un sistema software, localizzarne la maggior parte e, quindi, decidere quando il sistema è sufficientemente corretto da poter essere rilasciato.
In ogni caso, nonostante la buona volontà, uno sviluppatore non è in grado di trovare tutti gli errori nel proprio codice a causa di numerosi fattori tra i quali vi sono i seguenti [19]:

- tendenza a verificare il funzionamento del sistema tramite input validi (*clean test*) invece di utilizzare input non validi (*dirty test*);
- errata percezione della quantità di codice collaudato. Se non si usano strumenti automatici che verificano quanto codice sorgente è effettivamente eseguito nei test, lo sviluppatore tende a sovrastimare la quantità di codice verificato;
- tendenza ad evitare i test più complicati.

Per questi motivi i test fatti dagli sviluppatori non sono sufficienti ed ulteriori test devono essere fatti da personale specializzato che non deve aver contribuito alla realizzazione del prodotto.
Inoltre, non bisogna dimenticare che anche i test sono codice e, come tale, sono soggetti ad errore. In questo caso si hanno due tipi di conseguenze:

1. **falsi positivi**: il test evidenzia un errore che in realtà non è presente nel codice. In questo caso, dopo aver verificato che l'errore non è presente nel sistema, si verifica il codice del test. Questo tipo di errore ha un costo dovuto al tempo speso per cercare un errore che, in effetti, non esiste;
2. **falsi negativi**: il test non evidenzia un errore che in realtà è presente. In questo caso, il sistema è difettoso, ma il test non è in grado di rilevarlo. Questo tipo di errore vanifica parzialmente la presenza del test ed ha un costo più alto del precedente in quanto di deve sommare al costo della scrittura del test inutile quello dell'errore nel sistema in produzione.

5. IL TESTING ED IL PROCESSO DI PRODUZIONE DEL SOFTWARE

Il *testing* è una delle fasi del processo di produzione del software [24]. Se si considera il modello a cascata (*waterfall)*, le fasi sono le seguenti: analisi, progetto, codifica, *testing*, manutenzione. In questo ed in altri modelli di sviluppo il *testing* si trova alla fine del ciclo di produzione o, in ogni caso, si collauda il prodotto dopo aver scritto una parte consistente del codice. Questo approccio allo sviluppo è adeguato per alcune categorie di prodotti, ma non per tutte [8]. Nel caso di molti prodotti commerciali, un approccio migliore potrebbe essere diverso: non lasciare il collaudo alla fine, ma verificare il corretto funzionamento del sistema fin dalla scrittura delle prime righe di codice

e, in alcuni casi, scrivere i test addirittura prima del codice stesso (approccio definito come *Test First*) [1, 2].
Proprio per affrontare alcuni dei problemi associati allo sviluppo della maggior parte dei prodotti commerciali, negli ultimi anni sono nate e si stanno diffondendo delle nuove metodologie per lo sviluppo del software che vengono chiamate *Metodologie Agili* (MA) [3, 16]. L'approccio allo sviluppo software in questi metodi è centrato sul *testing* e, in particolare, sul *testing* del prodotto durante tutto il ciclo di produzione. Questo approccio è stato introdotto recentemente nel campo del software [22], ma le sue radici si possono trovare nell'industria manifatturiera a partire dagli anni '60: in particolare nei principi della *lean production* [27] e del *just-in-time* [21] che sono nati con lo scopo di migliorare il processo di produzione delle automobili alla Toyota. L'obiettivo consisteva nel produrre auto velocemente, con un numero ridottissimo di difetti e riducendo al minimo le scorte di componenti e di prodotti finiti. Queste tecniche di gestione del sistema manifatturiero si focalizzano sulla produzione di manufatti di alta qualità senza alcuno spreco, evitando di produrre oggetti difettosi e riducendo a zero i relativi costi. Infatti, la produzione di oggetti difettosi genera diversi tipi di costi che possono essere raggruppati in due categorie:

- **costi diretti**: materiale sprecato, tempo impiegato per la produzione ecc.;
- **costi generati**: tempo impiegato per la gestione del prodotto difettoso, spazio di magazzino per lo stoccaggio temporaneo ecc..

Per questo motivo, lo spreco (*muda* in giapponese) è visto come una sorgente di costi poiché la sua gestione richiede tempo, riducendo quello dedicato alla produzione dei manufatti che saranno immessi sul mercato e da cui proviene il profitto.
La *lean production*, in particolare, si focalizza sulla riduzione dello spreco andando a modificare il processo di produzione in modo tale da inserirvi degli strumenti di controllo della qualità. Inoltre, per ridurre possibili errori, si utilizzano sistemi automatici per monitorare la produzione. Questo approccio fa si che la verifica della qualità di un prodotto o di un semi-lavorato non sia più una attività separata dalla produzione, ma sia integrata completamente nella produzione. Quindi, non viene più valutata solo la qualità del prodotto finito o dei singoli componenti ma viene valutata la qualità ad ogni passaggio intermedio, in modo tale da identificare immediatamente qualunque tipo di difetto ed agire di conseguenza.
Le Metodologie Agili sono l'applicazione dei principi della *lean production* alla produzione del software [22] e come tali sono orientate alla produzione di manufatti di alta qualità, tramite la riduzione dello spreco. Anche nel caso del software esistono costi direttamente imputabili allo spreco e costi da esso generati:

- **costi diretti**: tempo impiegato per la produzione;
- **costi generati**: tempo impiegato per la correzione dei difetti, danni provocati dal sistema difettoso ecc..

Per aumentare la qualità dei prodotti e ridurre lo spreco, le MA non considerano il *testing* come una fase separata dello sviluppo del software, ma la integrano in tutte le fasi dello sviluppo utilizzando diverse tecniche di test (riquadro *Tecniche di testing*) e facendo uso di strumenti automatici per la loro continua esecuzione.
Tra tutte le Metodologie Agili, l'*Extreme Programming* [1] è quella più conosciuta e più utilizzata. Questa metodologia di sviluppo si basa su 12 pratiche fondamentali (in realtà, nell'ultima edizione del libro di Back [1] se ne sono aggiunte molte altre, ma in questo articolo ci limitiamo a considerare quelle della prima edizione che sono ormai consolidate). Molte di queste pratiche hanno, direttamente o indirettamente, una qualche relazione con il *testing*. Tra queste, quelle che hanno una relazione diretta con il testing sono: il *Test Driven Development* (Sviluppo Guidato dai Test), la *Continuous Integration* (Integrazione Continua) e le *Small Releases* (Piccoli Rilasci).
Il *Test Driven Development* (TDD) è un approccio allo sviluppo del software che pone al centro dell'attenzione il test, sia per quanto riguarda lo sviluppatore, sia per il cliente. Dal punto di vista del programmatore, mettere al centro dell'attenzione il test significa:

❑ Applicare la tecnica del *Test First*, vale a dire scrivere i test di verifica del codice prima di scrivere il codice stesso.
❑ Utlizzare i test unitari (*Unit Testing*) per verificare che ogni micro-funzionalità sia corretta
❑ Sviluppare in modo incrementale, implementando pochissimo codice alla volta e verificandone continuamente la correttezza.

Con la diffusione dell'*Extreme Programming* è nata una discussione attorno ai benefici di questo approccio e alle difficoltà che comporta una sua corretta implementazione. Tra i benefici troviamo i seguenti [2]:
❑ per scrivere un test si è costretti a pensare alla struttura del sistema prima di scrivere anche una singola riga di codice;
❑ i test non vengono scritti di fretta alla fine e questo contribuisce ad aumentarne l'efficacia (sia riducendo la probabilità di scrivere test errati che di scrivere test troppo semplici);
❑ i test possono essere eseguiti non appena il codice è stato scritto, evidenziando immediatamente i problemi.

Scrivere i test prima del codice non è semplice e solo un programmatore esperto è in grado di scrivere test rilevanti prima del prodotto: infatti, è necessario avere una visione ben chiara della struttura del sistema e della specifica funzionalità che si sta per implementare. Il rischio è che programmatori inesperti scrivano dei test che non sono adeguati a verificare in modo efficace il codice che scriveranno.

Dopo aver scritto il codice della funzionalità ed aver verificato che tutti i test siano eseguiti correttamente, lo sviluppatore deve riorganizzare il codice in modo da eliminare eventuali duplicati, migliorare l'architettura e la leggibilità del codice ecc.. Questo insieme di modifiche e miglioramenti del codice prende il nome di *Refactoring* [12].

Quindi, lo sviluppatore implementa il TDD ripetendo continuamente i seguenti passi:

1. scrittura del test unitario e dei **Mock Object** per una funzionalità minimale;
2. implementazione della funzionalità;
3. verifica di correttezza tramite l'esecuzione di tutti i test (questo fa si che non si verifichi solo se la funzionalità appena implementata è corretta ma anche che il codice scritto non provochi effetti indesiderati sul codice esistente);
4. *Refactoring*;
5. verifica di correttezza tramite l'esecuzione di tutti i test (questo serve per verificare che il *Refactoring* non abbia introdotto inavvertitamente degli errori).

Applicare il TDD senza degli strumenti adeguati è pressoché impossibile a causa della quantità di tempo che si dovrebbe dedicare al test. Per questo motivo sono nati diversi strumenti di supporto al *testing* che permettono di automatizzare totalmente la loro esecuzione e rendere questo approccio applicabile (paragrafo 6).

Dal punto di vista del cliente, mettere al centro dell'attenzione il test significa descrivere in dettaglio i criteri che saranno utilizzati per valutare la corretta implementazione di un prodotto e, quindi, accettare il software sviluppato (Test di Accettazione o *Acceptance Test*). I test di accettazione vengono scritti dal cliente (o dai programmatori in stretta collaborazione col cliente), in un linguaggio a lui comprensibile (normalmente si usa il linguaggio naturale) e prima dell'implementazione di ogni singola funzionalità. Gli sviluppatori hanno il compito di tradurre questi test nel linguaggio usato per l'implementazione del prodotto e di eseguirli prima di presentare il sistema al cliente.

Il TDD, quindi, consente di implementare nella produzione del software i meccanismi di controllo che sono tipici della *lean production*, permettendo di individuare tempestivamente i problemi all'interno del prodotto.

La *Continuous Integration* consiste nell'integrare continuamente il codice sviluppato

Mock Objects

In alcuni casi, scrivere i test prima del codice sembrerebbe impossibile perché, per esempio, la funzionalità che si vuole sviluppare utilizza un sottosistema che sarà sviluppato in futuro. Per aiutare gli sviluppatori in situazioni come questa si utilizzano i così detti *Mock Object* che non sono altro che classi che si sostituiscono temporaneamente ai sottosistemi ancora da sviluppare, semplificando la creazione dei test unitari. Un *Mock Object* presenta le seguenti caratteristiche:

- espone la stessa interfaccia del sistema che sostituisce (questo fa si che non sia necessario modificare i test quando si utilizzerà il sistema reale invece del *Mock Object*);
- per alcuni valori di input predefiniti (quelli utilizzati nel test), si comporta come il sistema reale.

In questo modo, è possibile scrivere i test prima del codice in qualunque situazione. Insieme ai test, quindi, lo sviluppatore deve scrivere anche tutti i *Mock Object* necessari a sostituire i sottosistemi non ancora disponibili ma necessari per collaudare efficacemente il codice che intende sviluppare.

dai singoli programmatori per evidenziare immediatamente i potenziali problemi di integrazione. Questo significa che ogni sviluppatore, non appena completata una funzionalità, deve integrarla nel prodotto ed eseguire i test di integrazione (riquadro su **tipi di testing**) per verificare che il proprio codice si comporti correttamente quando interagisce con quello scritto dagli altri e, in caso vi siano problemi, individuare e correggere immediatamente gli errori.
L'integrazione continua si basa sull'utilizzo di diversi strumenti tra cui un sistema di gestione delle configurazioni (CVS, V*isual Source Safe*, *ClearCase* ecc.) ed un sistema per l'automazione dei test di integrazione (paragrafo 6). Questa pratica permette di tenere sotto controllo i problemi legati all'interoperabilità del codice scritto da persone diverse per evitare che si individuino problemi di questo tipo quando è troppo

Tipi di testing

- **Black-box o functional testing**: significa verificare il comportamento di un sistema software o di una parte di codice senza vederne la struttura interna. Il software è come una scatola nera a cui si inviano dei dati in input e si verifica che i dati di output siano coerenti con le specifiche. In questo caso, esistono diverse strategie per scegliere i valori di input in base al tipo di sistema sotto esame.
- **White-box o structural testing**: significa verificare il comportamento di un sistema software o di una parte di codice avendo la possibilità di osservarne la struttura interna. In questo caso, i valori di input tramite i quali si collauda il sistema sono ricavati dall'osservazione del codice sorgente. Si individuano i valori che potrebbero dare origine ad errori e si verifica il comportamento del sistema in questi casi.

Strumenti di supporto al testing	Descrizione
Bugzilla (http://www.bugzilla.org/) *Scarab* (http://scarab.tigris.org/)	Sono due sistemi di tracciamento dei *bug*. Questi sistemi aiutano gli sviluppatori a raccogliere le segnalazioni di *bug*, a prioritarizzarle, ad assegnare la correzione ad un programmatore e, infine, a tenere traccia dell'evoluzione di ogni singolo difetto trovato nel codice.
Cactus (http://jakarta.apache.org/cactus/)	Questo strumento permette di collaudare applicazioni server-side scritte in Java. Tra le tecnologie supportate ci sono: servlet, JSP, EJB. Il sistema permette di scrivere vari tipi di test (unitari, di integrazione ecc.) e di automatizzarne l'esecuzione.
CruiseControl (http://cruisecontrol.sourceforge.net/)	Strumento per l'integrazione continua che si integra con i sistemi di gestione delle configurazioni (CVS, *Visual Source Safe* ecc.), scarica il codice del prodotto in esame, lo compila ed esegue una batteria di test. *CruiseControl* è in grado di generare rapporti sullo stato del sistema analizzato e pubblicarli via web o inviarli automaticamente per e-mail alle persone interessate.
JBlanket (http://csdl.ics.hawaii.edu/Tools/JBlanket/)	Strumento che permette di verificare quanto e quale codice sorgente è coperto dai test. Le informazioni ricavate sono molto utili per verificare che tutto il codice sia considerato almeno in un test. Anche in questo caso, il sistema può generare rapporti relativi allo stato del sistema sotto analisi e inviarli agli sviluppatori interessati.
JUnit (http://www.junit.org/)	Fornisce supporto alla creazione e all'esecuzione di Unit Test. Lo strumento aiuta il programmatore ad automatizzare questi test e a verificare che il codice sviluppato sia corretto. Esistono versioni di questo prodotto per moltissimi linguaggi di programmazione tra cui Java, C/C^{++}, C#, Pascal, Pyton ecc..
Mock Objects (http://www.mockobjects.com/)	Sistema per la creazione di *mock object*, strutture che semplificano la scrittura dei test unitari sostituendosi temporaneamente alle parti del sistema che devono ancora essere sviluppate.
Reflector (http://www.aisto.com/roeder/dotnet/)	Strumento per la visualizzazione delle dipendenze tra componenti NET.

TABELLA 1
Strumenti OpenSource di supporto al testing

tardi per evitare ritardi nella consegna del prodotto o di ridurre la qualità finale.
La pratica delle *Small Releases* consiste nel rilasciare il prodotto al cliente molto frequentemente (ogni 2-4 settimane) e con poche funzionalità aggiunte ogni volta. Questo approccio permette al cliente di valutare il prodotto sviluppato ed identificare tempestivamente eventuali incomprensioni nei requisiti. In questo modo, se il cliente individua qualche problema, gli sviluppatori possono intervenire prima che si sviluppino altre parti di codice riducendo al minimo lo spreco associato.

6. STRUMENTI DI SUPPORTO PER IL TESTING

Esistono molti strumenti che forniscono supporto alla fase di test. Il supporto è fornito in modi molto diversi tra cui il tracciamento dei *bug*, l'esecuzione automatica dei test, la verifica di quanto codice è coperto dai test ecc..
Tra gli strumenti dedicati al collaudo, quelli *OpenSourc*e sono sicuramente quelli più noti e diffusi, anche in ambiti commerciali. Nella tabella 1 sono elencati alcuni dei più noti strumenti *OpenSource* di supporto al testing.

7. CONCLUSIONI

Collaudare un sistema software presenta delle problematiche totalmente diverse dal collaudare un qualunque altro prodotto dell'ingegneria. Inoltre, data l'impossibilità di verificare esaustivamente un sistema, la probabilità che vi siano errori è sempre maggiore di zero, indipendentemente da quanto si testi il prodotto. I sistemi automatici di test, la disciplina nello scrivere i test e nell'eseguirli, le nuove tecniche di sviluppo del software sono tutti strumenti che hanno, tra gli obiettivi, la riduzione del numero di difetti nei prodotti software. A riprova di questa situazione, la ricerca in questo campo è molto attiva, come dimostrato dalla quantità di articoli su questo tema che vengono regolarmente presentati nelle maggiori conferenze internazionali e sulle riviste più prestigiose.

Bibliografia

[1] Beck K.: *eXtreme Programming Explained*. Addison-Wesley, 2000.

[2] Beck K.: *Test Driven Development*. Addison-Wesley, 2003.

[3] Beck K., Beedle M., Bennekum A., Cockburn A., Cunningham W., Fowler M., Grenning J., Highsmith J., Hunt A., Jeffries R., Kern J., Marick B., Martin R., Mellor S., Schwaber K., Sutherland J., Thomas D.: *Manifesto for Agile Software Development, 2001*. Disponibile online: http://www.agilemanifesto.org/

[4] Beizer B.: Design for Testability in Object-Oriented Systems. *Communications of the ACM*, Vol. 37, n. 9, Settembre 1990.

[5] Boehm B.W.: *Software Engineering Economics*. Prentice Hall, 1981.

[6] Boehm B.W.: *Improving Software Productivity*. IEEE Computer, Agosto 1987.

[7] Broekman B., Notenboom E.: *Testing Embedded Software*. Pearson, 2002.

[8] Brooks F.P.: *The Mythical Man-Month: Essays on Software Engineering*. Anniversary Edition, Addison-Wesley, 1995.

[9] Cooling J.: *Software Engineering for Real-Time Systems*. Addison-Wesley, 2002.

[10] DeMarco T.: *Controlling Software Projects – Management, Measurement & Estimation*. Yourdan Press, 1982.

[11] Endres A.: An analysis of errors and their causes in system programs. *ACM SIGPLAN Notices*, Vol. 10, n. 6, Giugno 1975.

[12] Fowler M., Beck K., Brant J., Opdyke W., Roberts D.: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[13] Gremillion L.L.: Determinants of Program Repair Maintenance Requirements. *Communications of the ACM*, Vol. 27, n. 8, Agosto1984.

[14] IEEE: I*EEE Standard for Glossary of Software Engineering Terminology*. IEEE Std. 828, 1983.

[15] Jones C.: *Software assessments, benchmarks, and best practices*. Addison-Wesley, 2000.

[16] Marchesi M., Succi G.: *Extreme Programming and Agile Processes* in *Software Engineering*. Springer, 2003.

[17] McConnell S.C.: *Rapid Development*. Microsoft Press, 1996.

[18] McConnell S.C.: *Code Complete*. 2nd edition, Microsoft Press, 2003.

[19] Myers G.J.: *The Art of Software Testing*. John Wiley & Sons, 1976.

[20] Nielsen J.: *Usability Engineering*. Morgan Kaufmann, 1994.

[21] Ohno T.: *Toyota Production System: Beyond Large-Scale Production*. Productivity Press, 1988.

[22] Poppendieck M., Poppendieck T.: *Lean Software Development: An Agile Toolkit*. Addison-Wesley, 2003.

[23] Port D., McArthur M.: *A Study of Productivity and Efficiency for Object-Oriented Methods and Languages*. Sixth Asia Pacific Software Engineering Conference, Takamatsu, Giappone, 7 - 10 Dicembre 1999.

[24] Pressman R.S.: S*oftware engineering: a practitioner's approach*. McGraw-Hill, 1992.

[25] Shull F., Basili V., Boehm B., Brown A.W., Costa P., Lindvall M., Port D., Rus I., Tesoriero R., Zelkowitz M.: *What We Have Learned About Fighting Defects*. Eighth IEEE Symposium on Software Metrics (METRICS 2002), Ottawa, Canada, 04 - 07 Giugno 2002.

[26] Whittaker J.A.: *What Is Software Testing? And Why Is It So Hard?*. IEEE Software, Gennaio/Febbraio 2000.

[27] Womack J.P., Jones D.T.: *Lean Thinking: banish waste and create wealth in your corporation*. Simon & Schuster, 1996.

ALBERTO SILLITTI è ricercatore presso il Centro per l'Ingegneria del Software Applicata della Facoltà di Informatica della Libera Università di Bolzano e docente del corso Internet Technologies 2. La sua attività di ricerca si concentra nell'area dell'ingegneria del software e, in particolare, su metodi di misura del processo di produzione del software, metodi agili di sviluppo del software, tecnologie per il Web e Open Source Software. Inoltre, partecipa a numerosi progetti di ricerca nell'area dell'ingegneria del software.
alberto.sillitti@unibz.it