Testing Object Oriented Software

Andrea Magni
Matr. 674863

andrea@ma5.it

ABSTRACT

In questo documento tratteremo alcuni aspetti dell'attività di testing di software realizzato secondo una metodologia object oriented. Le peculiarità di questo paradigma rendono necessarie alcune modifiche rispetto agli approcci tradizionali al testing e in questo documento passeremo in rassegna le principali caratteristiche dell'object orientation considerando per ognuna le problematiche correlate. Un aspetto sicuramente centrale del testing è la generazione dei casi di test; in questo documento verranno presentati sia un metodo di generazione automatica basato sull'analisi formale del componente da testare, sia un approccio operativo per uno sviluppatore che stia lavorando sul codice sorgente. Dopo aver accennato brevemente ai possibili metodi di integrazione si passerà ad affrontare alcune importanti considerazioni sul ruolo del testing nel contesto del processo di sviluppo e sulle sue diverse possibili interpretazioni e collocazioni all'interno della metodologia di progettazione e sviluppo.

Si assume che il lettore abbia buona familiarità con la metodologia object oriented e con i termini che la caratterizzano (incapsulazione, metodi, attributi, polimorfismo, ereditarietà etc.), inoltre si danno per noti almeno i rudimenti delle principali tecniche tradizionali di testing (testing funzionale e strutturale, coppie def-use, analisi data flow, test di copertura etc.).

1. INTRODUZIONE

Il testing è una pratica vecchia almeno quanto il software, da decenni ci si confronta con il problema fondamentale di ogni sistema: garantire il suo buon funzionamento in tutte le condizioni in cui viene utilizzato. Purtroppo spesso si è preferito spendere tempo, soldi e risorse su altre fasi del ciclo di sviluppo perdendo un po' di vista la materia prima di cui sono fatti i sistemi software: il codice.

La metodologia di progettazione e sviluppo object oriented ha avuto, dall'inizio degli anni '90, una fortissima diffusione. Oggi è sicuramente una delle più utilizzate e sembra avere un futuro roseo anche per gli anni a venire. Mentre molti sforzi sono stati fatti per avere dei formalismi e degli strumenti che fossero di supporto alla fase di analisi e design del ciclo di sviluppo, basti pensare ad UML e ai vari strumenti CASE attualmente disponibili, l'argomento del testing è stato in genere poco considerato. Fra le problematiche principali che l'object orientation, assieme alle pratiche e tecnologie ad essa collegate, ha saputo affrontare con successo vi sono sicuramente quelle legate alla realizzazione di architetture modulari, alla manutenzione del codice prodotto, al riuso del codice esistente e alla produzione di documentazione precisa e formale; in ognuno di questi ambiti si è avuto un miglioramento dovuto all'introduzione della metodologia OO ma vi sono anche problemi che non hanno ancora tratto particolari benefici dalle nuove tecnologie e metodologie. Uno di questi è il ben noto problema della correttezza: non siamo in grado di scrivere software error-free e da qui nascono (permangono) l'esigenza di fare testing e di farlo in maniera efficiente e efficace.

Se consideriamo certi livelli, come quello molto alto (di sistema) o molto basso (di singolo spezzone di codice), il testing dei sistemi object-oriented può avvenire in modi molto simili a quelli tradizionali: il test di sistema prescinde dalla metodologia utilizzata per progettare e realizzare il sistema e così pure questa non è un fattore discriminante a livello di test della singola procedura, anzi la tendenza della OOP di avere metodi brevi e con pochi, precisi, compiti semplifica di molto il testing a questo livello. La complessità invece è cresciuta nei livelli intermedi a causa delle peculiarità introdotte dall'object-orientation. Vediamole in dettaglio.

2. CARATTERISTICHE DEL MONDO OO

Segue una rapida carrellata delle peculiarità del mondo object oriented; l'intenzione non è quella di essere esaustivi ma solo di dare una panoramica quindi non si entrerà nei dettagli nè delle caratteristiche citate nè delle problematiche associate nè tantomeno delle possibili soluzioni. Per una trattazione approfondita di ognuna di queste tematiche si rimanda a testi più specifici [1] e [3].

2.1 Comportamento state-driven

Se si individuano i metodi come l'equivalente OO delle funzioni dei paradigmi tradizionali, la differenza fondamentale è quella che il loro comportamento non è più dipendente solo dai parametri di input. Gli oggetti sono dotati di uno stato interno e i metodi hanno comportamenti differenti a seconda dello stato dell'oggetto di cui fanno parte. Questa caratteristica, voluta e non occasionale, è una delle pratiche basilari del mondo OO.

2.2 Information hiding e incapsulazione

La presenza della distinzione fra parte pubblica e privata di una classe, presente in tutti i principali linguaggi OO e fondamentale per realizzare l'incapsulazione, porta a difficoltà nell'individuazione dei casi di test in quanto lo stato dell'oggetto potrebbe non essere più raggiungibile e rende difficile la creazione degli oracoli.

Dal punto di vista del testing, questa limitazione di visibilità è un problema e le soluzioni sono sostanzialmente due. La prima consiste nel rompere l'incapsulazione, operazione che però, oltre a non essere sempre praticabile perchè non sempre si è in condizione di modificare il sorgente, porta a sporcare il design delle classi. In alternativa si può usare la tecnica delle sequenze equivalenti, che si basa sull'ipotesi che sia possibile individuare sequenze di metodi che portino a delle precise transizioni di stato o ad un determinato stato finale.

2.3 Ereditarietà

L'ereditarietà in sè non aggiungerebbe problemi all'attività di testing: una classe che abbia anche molte classi progenitrici potrebbe comunque essere vista come una singola, grossa classe che contenga tutto il codice (suo e delle progenitrici). Per il testing si potrebbe procedere senza problemi.

Quello che si presenta problematico è individuare le strategie per riutilizzare i casi di test lungo la gerarchia o per individuare quali parti di una classe figlia vanno testate nuovamente e quali invece possono essere considerate già testate. Inoltre avere la definizione della classe sparsa in più punti può rendere difficoltoso, specie in caso di gerarchie molto frastagliate, avere una visione complessiva del codice della classe e questo può portare ad errori che il testing deve riuscire a scovare.

2.4 Binding dinamico e polimorfismo

Il binding dinamico e la possibilità di avere del codice polimorfico portano ad un'esplosione esponenziale del numero di casi di test necessari per coprire tutti i possibili casi.

In particolare supponiamo di avere una prima gerarchia di classi così composta:

X classe progenitrice, Y figlia di X, Z figlia di Y

Immaginiamo che X abbia un metodo Foo che accetti come parametro un oggetto di tipo A, dove A è una classe della seguente gerarchia: A classe progenitrice, B figlia di A, C figlia di B. Già in questa (semplice) situazione si pùò sperimentare come i le possibili combinazioni di chiamata siano numerose, se poi si considera che in sistemi reali le gerarchie possono essere anche molto più folte e che ogni classe potrebbe avere molti metodi con molti parametri ciascuno, è facile capire che un approccio esausitivo non risulti praticabile.

2.5 Astrazione

L'astrazione è un'altra caratteristica comune di molti linguaggi OO, il problema del testing qui è di sottile interpretazione: se da un lato ovviamente non è possibile testare codice che non è ancora stato scritto, dall'altro esistono situazioni (si pensi ad un qualunque middleware) dove è necessario poter testare almeno in parte il comportamento della classe, anche se non sono ancora a disposizione tutte le sue implementazioni.

2.6 Eccezioni

Nonostante non siano una peculiarità dei sistemi OO, le eccezioni vi hanno trovato un terreno fertile: sono uno dei meccanismi più potenti per quanto riguarda la gestione degli errori e il loro impiego è sempre più diffuso. Per loro natura, hanno un ruolo determinante nel testing e le problematiche principali volgono ad assicurare che queste vengano correttamente generate, intercettate e gestite anche a livelli differenti.

2.7 Altre caratteristiche

La presenza di meccanismi dinamici come i generics, le conversioni di tipo (type casting), le shadow invocations (che sono delle chiamate che avvengono in automatico e che possono sfuggire all'attenzione del programmatore) aggiungono ulteriore complessità ai moderni processi di sviluppo software e introducono un numero rilevante di casi da testare.

Alcune delle caratteristiche citate, come l'ereditarietà e il polimorfismo, sono peculiari dell'object orientation, altre invece (come le eccezioni) si inquadrano in contesti più generali. Assieme a queste ultime vogliamo citare anche il problema della concorrenza, tematica che si ripresenta invariata con i sistemi OO rispetto a come appare con i sistemi tradizionali: problemi di sincronizzazione, race condition per risorse condivise, indeterminismo etc. sono difficoltà che si incontrano facilmente nel mondo ad oggetti e questo è anche dovuto alla sempre più crescente propensione all'impiego di approcci OO per la realizzazione di sistemi distribuiti.

3. GENERAZIONE AUTOMATICA DEI CASI DI TEST

Quando si parla di attività di testing si intende un processo composto essenzialmente da due fasi: la prima consiste nell'individuazione di un insieme significativo di prove da far sostenere al componente (in senso lato) sotto test e successivamente ci dev'essere una fase di verifica vera e propria in cui fisicamente si mette alla prova il componente sotto test.

Individuare quali siano le situazioni da testare, specialmente per sistemi di una certa complessità, può non essere una attività semplice. Se abbiamo la possibilità di definire in modo formale il comportamento del sistema (o componente) da realizzare, si può ricorrere ad alcuni metodi per derivare in modo automatico dei casi di test significativi. Presenteremo brevemente uno di questi metodi rimandando a [4] per una trattazione esaustiva.

Il concetto fondamentale di questo procedimento è legato all'idea di stato di un oggetto. Abbiamo già detto che il comportamento dei metodi di un oggetto può essere legato allo stato interno dell'oggetto stesso. Questo porta a dire che lo stato di un oggetto in un determinato istante può essere considerato come il risultato di una serie di chiamate ai suoi metodi. Con il supporto di alcune informazioni derivanti dall'analisi del codice sotto test, è possibile generare delle sequenze di chiamate che siano rivelatrici di errori. Una volta ottenute queste sequenze, vanno eseguite e verificate. In sostanza l'obiettivo è generare un numero di casi di test non eccessivo (non è un testing esaustivo) ma significativo (che copra le coppie def-use, come vedremo) e capace di rilevare possibili faults.

3.1 Procedimento per lo unit testing

Dato un componente da testare, si procede in 3 passi alla generazione delle possibili sequenze da invocare.

1) Data-flow analysis, applicata ai metodi contenuti nel componente da testare generando il Class Control Flow Graph da questo si può procedere con le normali tecniche di analisi e derivare tutte le coppie def-use per gli attributi della classe
2) Symbolic execution, per derivare una specifica formale per ogni metodo, questa specifica include le precondizioni associate al metodo che portano ad un particolare cammino all'interno del metodo, le relazioni esistenti fra input e outputs del metodo e l'elenco delle attributi per cui si definisce un valore all'interno del metodo. Per ogni metodo quindi si possono avere differenti cammini possibili e le precondizioni di ognuno di questi cammini servono ad indicare quale verrà intrapreso di volta in volta
3) Automated deduction, per derivare sequenze di invocazioni di metodi sfruttando le informazioni ottenute ai passi 1 e 2. In particolare si usa la specifica ottenuta al punto 2 per generare sequenze che coprano le coppie def-use individuate al punto 1. La tecnica utilizzata è quella del backward chaining e per ogni coppia def-use si costruisce, partendo dal metodo dove si fa uso della variabile un albero di possibili invocazioni di metodi, avendo cura di non scegliere nodi le cui postcondizioni violino le precondizioni del nodo che si sta considerando

Una volta individuate queste sequenze (che stressano almeno una volta ogni coppia def-use individuata), queste vanno eseguite. Per ognuna di queste sequenze bisognerà approntare un opportuno test che verifichi l'effettivo corretto funzionamento del componente sotto test.

3.2 Procedimento per l'integration testing

Una volta che si è effettuato lo unit testing dei componenti del sistema, si può procedere con l'integrazione di questi ultimi accoppiandoli scegliendo un approccio fra quelli proposti. La scelta viene guidata generalmente dalla volontà di ridurre al minimo le necessità di scaffolding e viene quindi effettuata un'analisi delle dipendenze fra i vari componenti. I componenti foglia dell'albero dettato dalla relazione di "use" vengono integrati per primi in quanto non dipendenti da altri componenti e riutilizzabili per testare chi fa uso di loro.

Una volta individuati i due componenti (A e B) da testare accoppiati, posso scegliere un approccio che oscilla fra due estremi: quello che testa ogni sequenza di invocazione di metodi del componente A con ogni possibile stato di B e quello che propone invece di testare ogni sequenza di A con una sola sequenza di B. La scelta fra questi due approcci è in genere dettata dalla tipologia dei componenti A e B e dalla loro relazione.

Quello che praticamente si deve fare nel primo approccio è di considerare i due componenti come se fossero uno solo la cui interfaccia è la somma delle interfacce di A e B. Così facendo si vanno a testare numerose sequenze che stressano sia i metodi di A che quelli di B, portando entrambi i componenti in tutti i rispettivi stati possibili.

Nel secondo approccio invece si considerano i due componenti sempre raggruppati ma stavolta si considera l'interfaccia del raggruppamento come quella di uno dei due. In pratica si testano le sequenze di uno dei due componenti.

Come si può intuire il primo approccio genera molti casi di test mentre il secondo fornisce un livello di completezza nettamente inferiore. Occorre dunque valutare caso per caso il trade-off fra il numero di test case generati e la completezza del test per decidere che approccio utilizzare.

Il sistema è da considerarsi corretto se le invocazioni indirette dei metodi avvengono come individuato dallo unit testing dei singoli componenti e se i risultati ottenuti dalle sequenze applicate sono corretti.

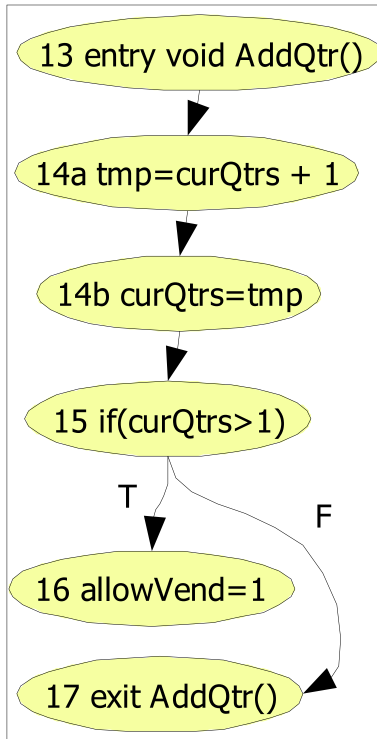
4. ESEMPIO: LA VENDING MACHINE

Riprendiamo brevemente l'esempio proposto in [4] della vending machine: abbiamo il codice della classe CoinBox e applicheremo i tre passi descritti nel paragrafo 3. Per brevità qui verranno riportati solo i dati significativi, si rimanda a [4] per la trattazione esaustiva.

```
class CoinBox {
 unsigned totalQtrs;
 unsigned curQtrs;
 unsigned allowVend;
public:
CoinBox() {
    totalQtrs = 0;
    allowVend = 0;
    curQtrs = 0;
}
void returnQtrs() {
    curQtrs = 0;
}
void addQtr() {
    curQtrs = curQtrs + 1;
    if (curQtrs > 1)
     allowVend = 1;
}
void vend() {
    if (allowedVend) {
     totalQtrs = totalQtrs + curQtrs;
     curQtrs = 0;
     allowVend = 0;
    }
}
};
```

Codice della classe CoinBox

Costruiamo ora il CCFG della classe CoinBox, per comodità riporteremo solo il pezzo di CCFG relativo ad uno dei metodi, a titolo esemplificativo.



CCFG del metodo AddQtr

Procedendo con i tradizionali metodi di analisi possiamo estrarre dal CCFG tutte le coppie def-use. Specifichiamo per ogni variabile anche il metodo dove viene definito il valore della variabile e il metodo dove invece viene utilizzato il valore assegnato. Fra parentesi sono indicati le etichette dei nodi del CCFG.

#	variable	m_d (node#)	m_u (node#)
01	curQtrs	CoinBox (4)	AddQtr (14a)
02	curQtrs	CoinBox (4)	AddQtr (15)
03	allowVend	CoinBox (3)	Vend (19)
04	totalQtrs	CoinBox (2)	Vend (20a)
05	curQtrs	ReturnQtrs (11)	AddQtr (14a)
06	curQtrs	ReturnQtrs (11)	AddQtr (15)
07	curQtrs	ReturnQtrs (11)	Vend (20a)
08	curQtrs	AddQtr (14b)	AddQtr (14a)
09	curQtrs	AddQtr (14b)	AddQtr (15)
10	curQtrs	AddQtr (14b)	Vend (20a)
11	allowVend	AddQtr (16)	Vend (19)
12	totalQtrs	Vend (20b)	Vend (20a)
13	curQtrs	Vend (21)	AddQtr (14a)
14	curQtrs	Vend (21)	AddQtr (15)
15	curQtrs	Vend (21)	Vend (20a)
16	allowVend	Vend (22)	Vend (19)
17	curQtrs	CoinBox (4)	Vend (20a)

Coppie def-use per la classe CoinBox

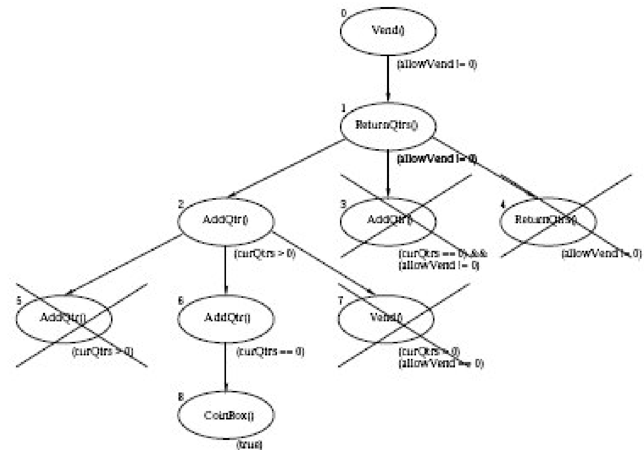
Alcune di queste coppie non sono realizzabili (per motivi legati strettamente al codice della classe), con riferimento alla tabella delle coppie def-use, le righe 2, 6, 14 e 15 sono infattibili. Passando al secondo punto del procedimento illustrato, generiamo, tramite la tecnica della symbolic execution, le precondizioni e le path condition per ogni metodo:

CoinBox		
$(true)$ def={totalQtrs, curQtrs, allowVend}	$\Rightarrow$	$totalQtrs' = 0$ $curQtrs' = 0$ $allowVend' = 0$
addQtr		
$(curQtrs > 0)$ def={curQtrs, allowVend}	$\Rightarrow$	$curQtrs' = curQtrs + 1$ $allowVend' = 1$
$(curQtrs == 0)$ def={curQtrs}	$\Rightarrow$	$curQtrs' = 1$
vend		
$(allowVend \neq 0)$ def={totalQtrs, curQtrs, allowVend}	$\Rightarrow$	$totalQtrs' = totalQtrs + curQtrs$ $curQtrs' = 0$ $allowVend' = 0$
$(allowVend == 0)$ def={}	$\Rightarrow$	$allowVend' = 0$
returnQtrs		
$(true)$ def={curQtrs}	$\Rightarrow$	$curQtrs' = 0$

Condizioni di esecuzione calcolate con la symbolic execution

Esaminiamo ora la coppia def-use numero 7, che riguarda la variabile curQtrs e l'assegnazione che essa subisce in ReturnQtrs e l'uso che ne avviene in Vend. La precondizione del metodo che definisce il valore della variabile è true, mentre quella del metodo che ne usa il valore è "allowVend diverso da zero".

Ora a partire dal metodo Vend si cerca di costruire una sequenza di chiamate che copra la coppia def-use scelta. Il procedimento dettagliato per la generazione dell'albero delle chiamate è descritto in [4], a noi basti sapere che, messo Vend come nodo radice, si aggiungono nodi le cui postcondizioni non contraddicono le precondizioni del nodo a cui vengono aggiunti.



L'albero delle chiamate generato.

Viene quindi individuata la sequenza:

CoinBox(), AddQtr(), AddQtr(), ReturnQtrs(), Vend()

che, oltre a coprire la coppia def-use associate, è anche la sequenza che rivela il comportamento scorretto della classe CoinBox. Un test case che fosse stato scritto con l'intento di verificare questa sequenza di invocazione avrebbe individuato l'errore.

5. APPROCCIO OPERATIVO PER LO SVILUPPATORE

Nel paragrafo 3 abbiamo visto come avvalerci di sistemi automatici per la generazione dei casi di test. Proponiamo ora anche un approccio operativo per lo sviluppatore che si trova ad affrontare la necessità di testare un sistema software OO preesistente. Riferimento [4].

Anche questo approccio si articola in più fasi, 3 per la precisione:

1) Intra-class testing, (unit testing): ha come scopo quello di verificare la correttezza di singole classi e ci si dovrebbe concentrare sostanzialmente sulle tematiche legate all'ereditarietà, all'influenza dello stato sul comportamento dei metodi e alle eccezioni

2) Inter-class testing, (integration testing): ha come scopo l'integrazione dei test per assicurare la qualità generale del sistema, in questa fase si dovrebbero considerare i problemi legati alla struttura della gerarchia delle classi e al polimorfismo

3) System and acceptance testing, che si possono effettuare con tecniche tradizionali e la cui trattazione quindi esula dall'intento di questo documento.

Vediamoli in dettaglio, ricordando che non sono uno schema rigido ma bensì una proposta di strutturazione volta anche a non tralasciare nulla.

5.1 Procedimento per lo unit testing

Il procedimento può essere sintetizzato nei seguenti passi:

1) Se la classe è astratta, comporre un insieme di istanze che coprano i casi significativi. Possono essere prese dall'applicazione (se disponibili) e/o create al solo scopo di essere testate.

2) Controllare la corretta invocazione dei metodi ereditati e di quelli ridefiniti (costruttori inclusi). Se vi sono ancestors già testati, determinare quali fra i metodi ereditati devono essere testati nuovamente e quali test case possono essere riutilizzati.

3) Progettare una serie di test case intra-class, basati sul modello a macchina a stati del comportamento della classe interessata.

4) Integrare i test case funzionali con quelli strutturali, aumentando il modello a macchina a stati con relazioni strutturali derivati dal codice della classe.

5) Progettare un insieme iniziale di test case per la gestione delle eccezioni, testando sistematicamente le eccezioni che devono essere sollevate dai metodi della classe e quelle che devono essere "trappate" e gestite.

6) Progettare un insieme iniziale di test case per le chiamate polimorfiche (considerando solo le chiamate locali).

5.2 Procedimento per l'integration testing

Lo schema per l'integration testing è il seguente, come potrete notare i due schemi seguono uno stesso percorso:

1) Identificare i "cluster" di classi da testare con un metodo incrementale: prima le classi foglie (che non includono/usano altre classi) fino a salire alle classi radice.

2) Progettare un insieme di inter-class test cases di tipo funzionale per il cluster sotto test.

3) Integrare i test cases funzionali con quelli strutturali, basati sulla copertura strutturale dei test funzionali.

4) Integrare l'insieme dei test per la gestione delle eccezioni con test che provino la corretta gestione di eccezioni non locali.

5) Integrare l'insieme dei test per il polimorfismo con test che provino l'interazione inter-class delle chiamate polimorfiche (e del dynamic binding).

L'approccio appena presentato (sia per lo unit testing che per l'integration testing) descrive genericamente il processo di generazione di una test suite seguendo un approccio diretto (partendo dal codice già scritto). Nel paragrafo 5 discuteremo anche di altri approcci, di provenienza legata al mondo delle metodologie agile ed XP, dove i test vengono scritti in parallelo con il codice di produzione.

5.3 Strategie per l'integration testing

Segue una panoramica sulle possibili strategie per la scelta dell'ordine dei moduli da testare durante l'integration testing. La scelta della strategia non è in relazione al tipo di approccio adottato per il testing, quella che segue è più che altro una classificazione delle tecniche più utilizzate nella pratica:

- Top-down: si parte da chi occupa il posto piu in alto della gerarchia dettata dalla relazione d'uso. Non necessita la realizzazione di drivers, richiede l'implementazione stub complessi.
- Bottom-up: è il duale della top-down; non richiede la realizzazione di stubs, comporta l'implementazione di drivers complessi. E' uno dei più usati.
- Big-bang: tutti i moduli integrati contemporaneamente; è molto problematica e sicuramente adatta solo a particolari progetti, uno dei pregi è che non richiede la scrittura nè di stubs nè di drivers e questo può venire in contro alle necessità contingenti di alcuni ambiti.

- Threads: per determinare la precedenza fra i componenti da testare, si seguono i thread di esecuzione che caratterizzano il sistema. Anche questo ha senso solo per ambiti particolari.
- Critical modules: si segue un criterio di criticità, è pensato per sistemi dove è individuabile uno o più componenti critici cui dare precedenza.

6. TESTING IN UN NUOVO CONTESTO METODOLOGICO

Nei paragrafi precedenti abbiamo cercato di fornire al lettore una panoramica dell'argomento testing con particolare riferimento alle caratteristiche distintive del mondo object oriented. Dopo aver presentato i due approcci operativi proposti è nostra intenzione affrontare brevemente le differenti posizioni sul collocamento e sul significato che va attribuito all'attività di testing all'interno del ciclo di sviluppo.

Negli ultimi anni, si è cercato di ripensare il concetto di testing seguendo due necessità molto sentite dalla comunità degli sviluppatori e degl ingegneri del software. La prima di queste necessità consiste nella riduzione, di quanto più possibile, il periodo di tempo che intercorre fra l'introduzione di un errore e la sua individuazione. Come è ben noto a chiunque si sia mai accostato al mondo dell'ingegneria del software, infatti più questo lasso di tempo aumenta, più aumentano i costi per la correzione dell'errore; inoltre è molto probabile che se a uno sviluppatore viene segnalato un errore introdotto pochi minuti prima, egli sia in grado di capire subito quale sia e dove sia situata la causa dell'errore. La seconda necessità in gioco è quella di non relegare la fase di testing ad un unico momento successivo alla produzione dell'intero sistema, non solo questo allungherebbe il lasso di tempo fra introduzione e scoperta dell'errore ma rischia anche di far degenerare l'attività di testing in una sorta di verifica di qualità da condurre fino a quando ci siano risorse disponibili (in genere alla fine del ciclo di sviluppo rimangono veramente poche risorse, soprattutto in termini di disponibilità temporale, e ciò influirebbe negativamente sulla completezza ed efficacia dell'attività stessa).

Da queste e altre considerazioni sono nate nuove filosofie di pensiero riguardo il testing e alcune di queste si sono spinte fino a soluzioni che possono sembrare estremiste (come la pratica di scrivere i test prima del codice di produzione proposta da E. Gamma) ma che sono guidate da una logica in genere precisa. Il più delle volte costa molta fatica cambiare la propria impostazione mentale per abbracciare queste nuove metodologie, ma se ci si sforza di capire quali sono gli intenti e, più pragmaticamente, a quali benefici si va incontro, si può giungere ad esprimere un giudizio sensato su queste tecniche e, spesso, essere propensi ad adottarle.

Illuminanti e fondamentali, per chi voglia approfondire queste nuove tecniche, sono alcuni articoli scritti da personaggi noti del panorama informatico mondiale quali Martin Fowler, Kent Beck (autore di SUnit, il primo framework per lo unit testing), Eric Gamma (membro della Gang of Four dei Design Patterns e autore del porting di SUnit da Smalltalk a Java, chiamato JUnit); una ricerca su un qualunque motore di ricerca vi condurrà a molti articoli di questi autori. Un altro punto di vista (non molto distante) riguardo questi argomenti è dato da Alistair Cockburn, che ha proposto le metodologie Crystal.

In particolare, vogliamo indicare al lettore l'articolo "Testing Methods: The Ugly Duckling" di Martin Fowler [2] del 1998; in questo articolo, forse a tratti un po' pittoresco ma decisamente all'avanguardia per il periodo in cui fu scritto, si possono riconoscere le motivazioni che portano alla necessità di un ripensamento del modo di fare testing e dei benefici che derivano dall'accoppiare l'attività di produzione del codice reale a quello di test. Un altro articolo, dello stesso autore, "Keeping software soft" [2] pone l'attenzione sul fattore di dinamicità che caratterizza il mondo dell'ingegneria del software, proponendo l'utilizzo simultaneo di una metodologia object-oriented, del testing e della pratica intensiva del refactoring come via per raggiungere una alta responsività ai cambiamenti nei requisiti, una migliore capacità di mantenere pulito il proprio design a fronte di nuove esigenze e l'abilità di rendere durevoli nel tempo le due proprietà appena elencate.

Spesso, purtroppo, quando si parla di testing e si accenna a queste (relativamente) nuove correnti di pensiero, si finisce con l'innescare scontri di carattere ideologico che sono all'ordine del giorno nell'ambiente informatico ma che spesso non hanno risvolti costruttivi. Noi ci limiteremo ad effettuare alcune considerazioni, senza prendere posizione (queste sono le nostre intenzioni).

Ci sono degli innegabili vantaggi nel cercare di scrivere i test prima del codice di produzione, anche se questa pratica richiede uno sforzo mentale che molti sviluppatori non sono abituati nè pronti a fare. Questa pratica aiuta a concentrarsi maggiormente sul codice, sull'interfaccia delle classi e portano a scrivere dei test che hanno in genere una maggiore significatività, un basso accoppiamento e una migliore adattabilità agli inevitabili cambiamenti futuri. Il motivo, secondo l'opinione di chi sta scrivendo, è lo stesso per cui è considerata un'ottima tecnica il preoccuparsi del design prima di passare all'implementazione: così come non si vuole rischiare di influenzare negativamente il design a causa di scelte implementative, così non si deve cercare di scrivere i test basandosi (unicamente) sul codice già scritto. Sforzandosi di realizzare prima il codice di test, si otterranno dei test che sono qualcosa di più di pura stimolazione "meccanica" di un codice predeterminato e anche il processo di individuazione dei test case può essere guidato dal comportamento voluto e non da quello indotto dall'implementazione corrente.

D'altra parte anche i metodi tradizionali, supportati da anni di studi e raffinamenti, non sono da abbandonare. Il test "meccanico" del codice può fornire ancora molte informazioni utili al progettista e allo sviluppatore. La soluzione ideale molto probabilmente non si trova nemmeno nel mezzo, nel senso che alcune tecniche convenzionali possono essere riutilizzate integrandole con alcune tecniche di recente introduzione che si sono rivelate produttive ma la scelta di dove utilizzare l'una e l'altra non può che dipendere dal contesto specifico. Di fatti è provato che, per molti tipi di progetti, l'approccio che prevede lo sviluppo incrementale (codice di produzione e codice di test insieme, Fowler) porta ad una riduzione significativa dei tempi di sviluppo e ad una maggiore visibilità sull'andamento del processo di produzione. In uguale misura è anche vero che non sempre è possibile per lo sviluppatore produrre tutti i casi di test e, anzi, alcuni fra i più subdoli possono sfuggire facilmente; in questo

scenario quindi possono trovare posto anche quelle tecniche che fanno testing a-posteriori, partendo dal codice già in produzione e generando i casi di test secondo criteri tradizionali (come quello di copertura delle coppie use-def trattato in questo documento).

7. CONCLUSIONI

Il testing è un'attività fondamentale del ciclo di sviluppo di un sistema software ed è importante che si continui a suscitare l'interesse della comunità informatica su questa tematica. In questo documento abbiamo voluto indicare quali sono le problematiche che caratterizzano il testing di sistemi sviluppati con metodologia object oriented, oggi una fra le più diffuse. Abbiamo affrontato il tema della generazione dei casi di test, proponendo due approcci diametralmente opposti e abbiamo voluto affrontare anche il discorso, più teorico, legato alle implicazioni metodologiche derivanti dal differente posizionamento ed impiego, all'interno della metodologia di sviluppo, dell'attività di testing.

Oggi che siamo sempre più portati ad un mondo object oriented, basti pensare alle tecnologie Java e .Net, è doveroso impegnare risorse per cercare di sviluppare tecniche di testing adeguate. Questo è un argomento che viene ancora troppo spesso sottovalutato e che invece può rappresentare un'arma formidabile nella gestione dei progetti software e non solo da un punto di vista teorico o accademico, ma anche prettamente pratico, con benefici consistenti e fruibili su progetti di ogni dimensione.

8. REFERENCES

[1] Mauro Pezzè, Michal Young, *Testing OO Software*, (materiale assegnato dal docente), 2004

[2] Martin Fowler, *Mocks objects aren't stubs*, *Testing Methods: The Ugly Duckling*, *Keeping Software Soft*, http://www.martinfowler.com

[3] Alessandro Orso, Sergio Silva, *Open Issues and Research Directions in Object-Oriented Testing*, Gennaio 1998.

[4] Ugo Buy, Carlo Ghezzi, Alessandro Orso, Mauro Pezzè, Matteo Valsasna, *A Framework for Testing Object Oriented Components*, May 1999.