Capitolo 5
Test di unità

Introduzione

Il presente capitolo è dedicato all'illustrazione dei test di unità (*unit test*); in particolare, analogamente ai capitoli precedenti, sono presentate una serie di tecniche, framework di supporto, linee guida e *best practice* utili per la produzione efficace di validi test di unità.

Al momento in cui viene scritto questo libro, l'ingegneria del software annovera una serie di processi di sviluppo del software che spaziano da quelli strettamente centrati sul codice, come XP (*eXtreme Programming*) a quelli via via più formali, per esempio RUP (*Rational Unified Process*) e MDA (*Model Driven Architecture*). Nonostante le fondamentali differenze filosofiche e pratiche alla base dei vari processi, è comnque possibile individuare qualche sporadica area in cui vi è unanime consenso: una di queste è, appunto, l'importanza dei test. Verifiche formali, chiaramente, devono essere eseguite durante tutte le fasi del processo di sviluppo del software, non appena nuovi manufatti (per esempio il modello dei requisiti utente, il disegno del sistema, etc.) diventano disponibili. Tuttavia, il fine ultimo dei processi di sviluppo del software è realizzare sistemi software, pertanto è fondamentale che il codice prodotto sia verificato accuratamente. Ciò è possibile corredando il sistema con approfonditi test automatizzati.

Come si vedrà meglio nei prossimi capitoli, esistono diversi livelli di test. In questo capitolo l'attenzione è centrata su un primo stadio di verifica fornito dai test di unità. Con questo termine ci si riferisce alle procedure utilizzate per verificare che una particolare porzione di

codice (tipicamente una classe) presenti il comportamento aspettato: in una parola, si verifica che l'unità funzioni correttamente.

I test di unità, e più in generale l'intero insieme di test (unità, integrazione, sistema, etc.), sono fondamentali non solo per esaminare il funzionamento delle varie parti del sistema al momento della loro scrittura, ma anche per disporre di uno strumento formale che eviti un problema fondamentale: i processi di aggiornamento del codice non devono produrre malfunzionamenti in parti del sistema correttamente funzionanti prima dell'incorporazione degli aggiornamenti. Disponendo di una batteria di test automatizzati, è quindi possibile eseguirla dopo ogni insieme di modifiche (questo processo è tipicamente denominato test di regressione, *regression test*) al fine di individuare, istantaneamente, eventuali errori introdotti con la nuova versione. I test di regressione tendono a essere un formidabile supporto per contrastare la paura dei cambiamenti che spesso si annida nella mente degli sviluppatori. Pertanto, come logico attendersi, la presenza di questi test riduce i fattori di rischio intrinseci a ogni processo di modifica e aumenta il livello di sicurezza del team di sviluppo.

L'obiettivo di questo capitolo è fornire una serie di direttive relative alla produzione di efficaci test di unità. In particolare, la maggior parte delle direttive presentate fanno riferimento al framework JUnit che, al momento in cui viene redatto questo libro, è lo standard *de facto* per la produzione di test di unità in Java.

Produrre i test di unità è un'attività indispensabile del ciclo di vita del software. La loro esistenza offre molti vantaggi, come l'immediata individuazione di eventuali errori, l'aumento della qualità del codice prodotto, la semplificazione del processo di refactoring, la semplificazione del processo di integrazione e la produzione di documentazione supplementare. Tuttavia, test di bassa qualità o il cui livello di copertura sia eccessivamente limitato possono produrre disastrosi risultati — per esempio un falso livello di sicurezza circa la qualità del sistema — e portare alla consegna di un sistema dalla qualità piuttosto limitata, etc.

Quindi, per evitare questi effetti paradossali, è necessario porre attenzione a una serie di fattori, come il dominio di applicazione dei test di unità, la copertura dei test, l'impatto dei processi di sviluppo iterativi e incrementali sul processo di manutenzione dei test, etc. Questi argomenti, chiaramente, sono oggetto di studio del presente capitolo.

Un po' di teoria

Introduzione

Con il termine test di unità ci si riferisce alle procedure utilizzate per verificare che un particolare frammento di codice presenti il comportamento aspettato.

Al fine di comprendere chiaramente l'idea alla base di questo tipo di test, si consideri la produzione di schede elettroniche. Ciascuna scheda contiene una grande quantità di componenti: resistenze, condensatori, chip, etc. Ogni singolo componente, prima di essere inserito in circuiti complessi, viene sottoposto a un processo di verifica. Per esempio, si verifica che le resistenze presentino la resistenza dichiarata con un'approssimazione del 3% o 5% (a secon-

da della qualità della resistenza), che i condensatori presentino la capacità attesa, sempre entro certi limiti, e così via. Pertanto, ciascun componente è verificato singolarmente, indipendentemente dagli altri e dalla scheda in cui verrà applicato. Queste verifiche, eseguite nell'ambito dell'elettronica, sono l'equivalente dei test di unità eseguiti nel dominio della produzione del software. In questo dominio, l'obiettivo dei test di unità è analogo: verificare che ogni classe presenti il comportamento atteso, indipendentemente dal contesto di utilizzo. Chiaramente, come si vedrà di seguito, si tratta di test necessari ma non sufficienti: il fatto che un componente elettronico funzioni come previsto assolutamente non significa che automaticamente la scheda montata funzioni altrettanto correttamente.

La completa esecuzione senza errori dei test di unità rappresenta l'evento di avvio dei test di integrazione che, come si vedrà nel capitolo successivo, servono a verificare che componenti funzionanti singolarmente, una volta correttamente assemblati tra loro diano luogo a elementi più grandi ancora funzionanti correttamente.

I vantaggi dei test di unità

Produrre i test di unità è indubbiamente un lavoro dispendioso; tuttavia, si tratta di un'attività fondamentale e come tale, va considerata come un vero e proprio investimento e non come una dissipazione di risorse. Un'opportuna implementazione dei test di unità offre una indubbia serie di vantaggi.

- **Individuare rapidamente gli errori** . In particolare, questi test permettono di individuare eventuali malfunzionamenti, non appena redatto il codice (secondo alcuni estremisti anche prima!). Inoltre, l'indagine è spinta a un livello molto approfondito, tipicamente superiore a quello raggiunto con il classico test randomico. La logica conseguenza è un significativo risparmio di tempo e quindi di denaro. Frequentemente, errori individuati in fasi successive del processo di sviluppo del software, durante i test di accettazione o addirittura a sistema in produzione, risultano difficili da investigare, da risolvere e, in ultima analisi, decisamente più onerosi.

- **Aumentare la qualità del codice** . La verifica estensiva del codice permette di individuare e risolvere un'elevata quantità di errori, spesso anche quelli più nascosti, e quindi di garantire una maggiore qualità del sistema nel suo complesso. Ciò, inoltre, semplifica la produzione del codice basato su quanto già prodotto, aumentando, contestualmente, il grado di confidenza degli stessi programmatori.

- **Semplificare il processo di refactoring** . La presenza di estesi test di unità permette di minimizzare gli inevitabili fattori di rischio connessi a estesi processi di refactoring e, allo stesso tempo, aumenta il livello di sicurezza del team di sviluppo. In effetti, eventuali problemi introdotti dai processi di aggiornamento (chiamanti *regression bug*, errori regressivi) sono immediatamente individuati dalla batteria di test. La presenza di questi test, quindi, minimizza il rischio di degenerare la qualità del sistema a seguito di processi di modifica del codice e fornisce ai programmatori la tranquillità e la fiducia

necessaria per affrontare importanti aggiornamenti del codice. Non è infrequente il caso in cui l'assenza di estesi testi di unità, finisca per far desistere o comunque per scoraggiare i programmatori dall'eseguire importanti variazioni del codice.

- **Semplificare il processo di integrazione** . Sebbene la pertinenza dei test di unità, per loro stesso principio, non riguardi l'integrazione di parti del sistema, il fatto che ogni elemento sia stato verificato in dettaglio, prima di essere assemblato in elementi più grandi, riduce il livello di incertezza e quindi semplifica e velocizza il processo stesso di integrazione.

- **Migliorare la documentazione** . In alcuni casi, lo studio di determinate API o di parti di codice è notevolmente semplificato dall'analisi di esempi concreti come quelli implementati nei test di unità. Pertanto, tali test possono essere considerati a tutti gli effetti come parte integrante della documentazione del codice, oltretutto assolutamente in linea con l'ultima versione del codice, cosa che spesso non succede con la tradizionale documentazione: non è infrequente il caso in cui i programmatori tendano a dimenticare di mantenere la documentazione sincronizzata con il codice.

Processi iterativi e incrementali

Oltre all'importanza dei test, un'altra pratica che trova consenso tra la quasi totalità dei moderni processi di sviluppo del software, indipendentemente dalla loro diversa natura, è la necessità di integrare approcci iterativi e incrementali nel processo di sviluppo del software prescelto. L'idea alla base di questi consiste nel produrre il sistema finale attraverso una serie opportunamente pianificata di incrementi. L'intero processo di sviluppo del software si risolve in un insieme di mini progetti, ognuno dei quali produce una nuova versione (incremento) del sistema finale. Gli incrementi possono essere relativi sia allo sviluppo di nuovi servizi, sia alla variazione tecnologica di alcune parti del sistema (processi di refactoring).

Questo approccio è molto conveniente per una serie di motivi: permette di controllare più accuratamente l'evoluzione del sistema, di gestire attivamente i fattori di rischio concentrando rischi maggiori nelle prime versioni del sistema, fa sì che, ad ogni iterazione, il personale possa concentrarsi su un ben definito insieme di elementi, permette di ottenere il feedback continuo ed anticipato degli utenti, e così via.

Tuttavia, a dispetto di questi importantissimi vantaggi esistono degli inevitabili effetti collaterali. In particolare, è tipico degli approcci iterativi e incrementali che stesse porzioni di codice siano variate con una certa frequenza durante l'evoluzione del processo. Logica conseguenza di ciò è che anche i relativi test di unità debbano essere soggetti a continui processi di aggiornamento/riscrittura. Ciò comporta una serie di importanti conseguenze.

- I test di unità devono presentare un adeguato livello di qualità. Pertanto la tradizionale equazione *codice test = bassa qualità* perde di validità. In primo luogo, test mal scritti tendono a generare ripercussioni negative su diversi vantaggi propri relativi alla loro

presenza; poi, il processo di manutenzione risulta complicato e quindi, in ultima analisi, può addirittura causare il rallentamento dell'intero processo di sviluppo del sistema.

- I test devono essere disegnati ed implementati pensando all'estensibilità, ovviamente senza cadere nella facile tentazione dell'*over-engineering* . Ciò perché è tipico dei processi iterativi ed incrementali generare la variazione di determinate parti di codice. Pertanto, gli stessi codici di test devono essere scritti accuratamente soprattutto per quanto riguarda le parti che presentano un'elevata possibilità di dover essere variati.

- i test diventano parte del sistema e come tale sono soggetti a modifiche richieste dalle varie iterazioni. Pertanto, sebbene sia necessario verificare la maggior parte possibile del sistema, è altresì necessario evitare di scrivere test ridondanti che, come tali non aggiungono valore, ma comunque devono essere mantenuti.

La copertura dei test di unità

Uno degli argomenti ricorrenti nella produzione dei test di unità, oggetto soprattutto in passato di accesi dibattiti, riguarda il livello di copertura di tali test (rapporto tra le linee di codice eseguite dai test e quelle totali). In teoria, questi test dovrebbero interessare tutte le classi incluse nel sistema che si sta sviluppando (1 a 1), escludendo, ovviamente, le classi appartenenti al JDK e le librerie esterne, il cui rilascio è soggetto al superamento di uno specifico e approfondito processo di test e di QA. Anche se questo processo non sempre è infallibile, interviene un problema di ordine pratico: tentare di verificare tutte le classi del JDK o delle librerie fornite da terze parti causerebbe un dispendio di energie assolutamente improponibile. Pertanto, giocoforza, è necessario far affidamento sui test eseguiti dai fornitori delle librerie utilizzate, ed eventualmente scrivere specifici test solo qualora ci siano fondati sospetti che determinate classi/metodi non presentino il funzionamento atteso. Infine, in ultima analisi, è necessario considerare che le classi appartenenti a librerie esterne sono comunque verificate, sebbene indirettamente, attraverso i test relativi alle classi sviluppate che ne incapsulano il comportamento.

I test di unità, quindi, dovrebbero riguardare tutte le classi prodotte. Il quesito però che resta da sciogliere è relativo a quali metodi verificare e come.

Nella pratica, ci si può imbattere in professionisti un po' estremi che suggeriscono test in grado di verificare ogni possibile dettaglio, ogni possibile linea di codice del sistema. Questa strategia, contrariamente a quanto si sarebbe portati a pensare e a dispetto del relativo enorme consumo di tempo e budget, raramente risulta efficace, ed è inoltre percorribile solo in un limitato insieme di casi. La copertura esaustiva (100%) del sistema è inutile e spesso addirittura deleteria.

In primo luogo, i test di unità non verificano assolutamente i requisiti del sistema ma si occupano di verificare il funzionamento dei singoli componenti; qualora lo facessero poi, non è sempre possibile anticipare tutte le possibili combinazioni degli stimoli di input di un sistema, tanto meno è possibile ricreare tutti gli scenari di evoluzione dei *thread* presenti in un

sistema *multi-threaded* da verificare, e così via. Pertanto anche nel caso in cui si riuscisse, attraverso i test di unità, a esercitare tutte le linee di codice del sistema (copertura = 100%), ciò non assicurerebbe assolutamente il corretto funzionamento dello stesso. I test di unità, sono solo un primo livello delle procedure di test del sistema, le quali devono essere pianificate attentamente anche prima dell'inizio dell'implementazione del sistema.

Il matematico Kurt Gödel [GDLPRF ha ulteriorment, dimostrato come non sia possibile provare tutti gli aspetti di un sistema dall'interno del sistema stesso. In particolare, ha dimostrato che tutti i sistemi logici contengono necessariamente una serie di assiomi, ossia verità che devono essere accettate per buone e la cui verifica può essere effettuata solo al di fuori del sistema stesso.

Coperture esaustive sono difficilmente raggiungibili anche per motivi di ordine pratico; per esempio, tutti i sistemi includono delle porzioni di codice semplicemente non raggiungibili, come catch che possono anche non verificarsi mai. Alcuni tecnici, in queste condizioni, commettono l'errore di modificare il codice al fine di rendere possibile una copertura completa. Quindi, è importante prendere atto che realizzare test esaustivi del sistema è una strategia non sempre efficace e realizzabile e assolutamente dispendiosa. Si consideri il frammento di codice riportato di seguito. Si tratta dell'implementazione del metodo equals di un Data Transfer Object/Value Object (DTO/VO, ossia Java bean utilizzati semplicemente per trasportare dati). Questo oggetto possiede i seguenti attributi: id, login, name, middlename, surname, dob (*date of birth*), email (principale), addresses (oggetto composto che contiene ogni tipo di recapito aggiuntivo: e-mail, telefono, cellulare, pager, indirizzo, etc.), userStatus (stato dell'utente) e orgUnit (*organisational unit*, dipartimento di appartenenza).

```
public boolean equals(Object obj) {

  if (this == obj) {
    return true;
  }

  if (!super.equals(obj)) {
    return false;
  }

  if (getClass() != obj.getClass()) {
    return false;
  }

  final UserDTO other = (UserDTO) obj;

  if (id == null) {
    if (other.id != null) {
      return false;
    }
```

```
} else if (!id.equals(other.id)) {
   return false;
}

if (login == null) {
   if (other.login != null) {
      return false;
   }
} else if (!login.equals(other.login))
   return false;
}

if (name == null) {
   if (other.name != null) {
      return false;
   }
} else if (!name.equals(other.name)) {
   return false;
}

if (middlename == null) {
   if (other.middlename != null) {
      return false;
   }
} else if (!middlename.equals(other.middlename)) {
   return false;
}

if (surname == null) {
   if (other.surname != null) {
      return false;
   }
} else if (!surname.equals(other.surname)) {
   return false;
}

if (dob == null) {
   if (other.dob != null) {
      return false;
   }
} else if (!formatDate(dob).equals(formatDate(other.dob)))
   return false;
}
```

```
    if (email == null) {
      if (other.email != null) {
        return false;
      }
    } else if (!email.equals(other.email)) {
      return false;
    }

    if (userStatus == null) {
      if (other.userStatus != null) {
        return false;
      }
    } else if (!userStatus.equals(other.userStatus)) {
      return false;
    }

    if (addresses == null) {
      if (other.addresses != null) {
        return false;
      }
    } else if (!addresses.equals(other.addresses)) {
      return false;
    }

    if (orgUnit == null) {
      if (other.orgUnit != null) {
        return false;
      }
    } else if (!orgUnit.equals(other.orgUnit)) {
      return false;
    }

    return true;
  }
```

La verifica completa di questo semplice metodo richiederebbe di scrivere circa 25 test! La figura 5.1 fornisce una rappresentazione del grafo dei diversi percorsi che un test completo dovrebbe coprire. I test da produrre per verificare i diversi percorsi del metodo `equals` sono illustrati nella tabella 5.1.

È veramente necessario produrre test capaci di esercitare tutti i possibili percorsi del metodo `equals`? In definitiva, no… anche se ogni percorso non analizzato finisce con il ridurre la percentuale di copertura.

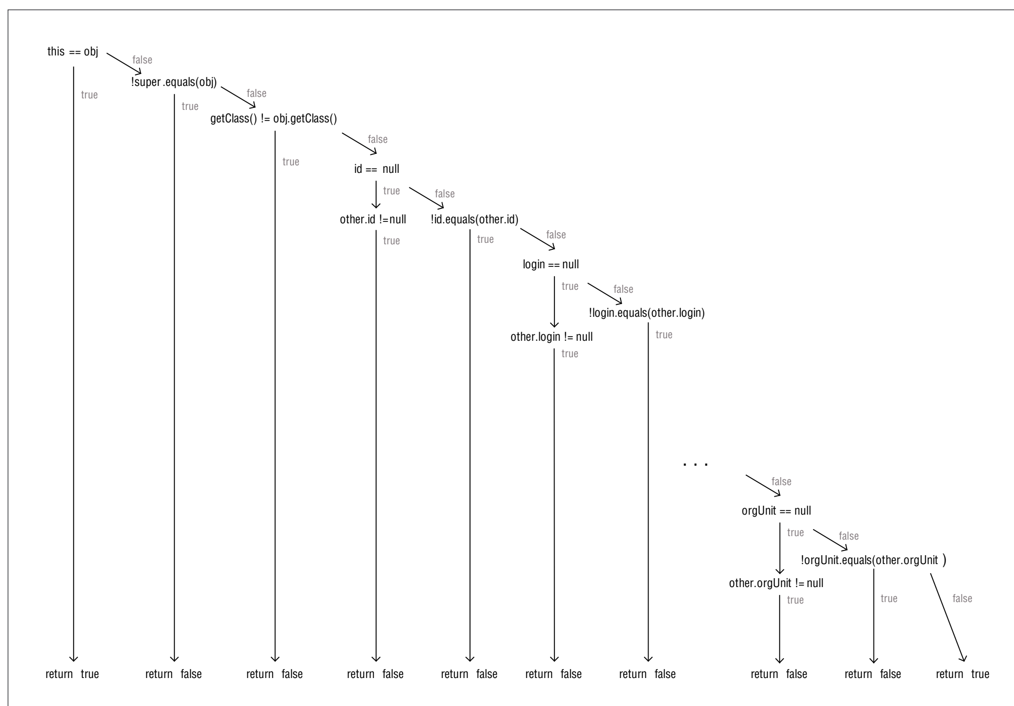

Figura 5.1 – Grafo dei diversi percorsi presenti nel metodo `equals`*.*

Il costo di produzione iniziale e di manutenzione di test esaustivi è, nella quasi totalità dei casi, insostenibile. Nel caso del metodo `equals`, si consideri, per esempio, l'impatto generato sul codice di test relativo all'introduzione di un nuovo attributo.

Quindi è importante discernere metodi che non è necessario verificare approfonditamente da quelli di cui invece è necessario verificare ogni singolo dettaglio.

Si consideri come ulteriore esempio il caso di semplici metodi `getXXX` e `setXXX`; a meno che questi non eseguano ulteriori operazioni, non è estremamente utile verificarli. Inoltre, poiché i metodi accessori (`getXXX`) dovrebbero essere l'unico modo per accedere ai dati di un oggetto, tali metodi sono comunque oggetto di investigazione da parte di altri test.

Qualora il metodo da verificare fosse l'implementazione di un algoritmo di un servizio business di rilevanza fondamentale, allora probabilmente la situazione sarebbe diversa. In tal caso infatti, a differenza del metodo di uguaglianza, sarebbe opportuno verificarne tutti i possibili percorsi.

Sebbene non sia possibile dar luogo a una copertura esaustiva, questa deve essere sicuramente estensiva. Coperture insufficienti possono generare una serie di dannosissimi effetti

Num	Descrizione	Esito
1	Tutti gli attributi dei due oggetti impostati allo stesso valore non nullo	true
2	Invocazione del metodo sullo stesso oggetto	true
3	Invocazione del metodo su due oggetti con un elemento della classe antenata diverso	false
4	Invocazione del metodo su due oggetti di tipo diverso ereditanti dallo stesso antenato con gli elementi in comune uguali	false
5	Invocazione del metodo su due oggetti uguali con tutti gli elementi impostati a null	true
6	Invocazione del metodo su due oggetti con tutti gli elementi uguali, eccetto l'id del primo impostato a null	false
7	Invocazione del metodo su due oggetti con tutti gli elementi uguali eccetto gli id (diversi ma non nulli)	false
8	Invocazione del metodo su due oggetti con tutti gli elementi uguali, eccetto l'attributo login del primo impostato a null	false
9	Invocazione del metodo su due oggetti con tutti gli elementi uguali, eccetto l'attributo login (diversi ma non nulli).	false
...	...	false
24	Invocazione del metodo su due oggetti con tutti gli elementi uguali, eccetto l'attributo orgUnit del primo impostato a null	false
25	Invocazione del metodo su due oggetti con tutti gli elementi uguali, eccetto l'attributo orgUnit (diversi ma non nulli)	false

Tabella 5.1 – Elenco dei test dei percorsi del metodo di equals*.*

collaterali, quali ad esempio una bassa qualità del codice, una falsa sensazione di robustezza, continui malfunzionamenti del sistema in produzione a seguito di processi di aggiornamento dovuti a test di regressione incapaci di eseguire approfondite analisi del sistema, etc.

Purtroppo, nella pratica lavorativa, accade anche spesso di imbattersi in progetti in cui i test di unità coprono una minima porzione del codice… Coperture inferiori al 60-70% del codice dovrebbero generare qualche preoccupazione al capo progetto di turno.

Anche se la produzione di questi test può sembrare un'attività eccessivamente dispendiosa e poco sostenibile, soprattutto in situazioni in cui il progetto risulti costretto da severi vincoli in termini di tempo e denaro, si tratta di una strategia che permette di realizzare più rapidamente sistemi di maggiore qualità; e la loro validità è evidenziata ad ogni iterazione/processo di *refactoring*. Pertanto, l'impegno profuso nella scrittura dei test di unità è ripagato, nel medio/lungo termine, in termini produzione di sistemi di qualità superiore, che, in ultima analisi, producono economie di tempo e budget.

Tutto il codice rilasciato, anche se non utilizzato, deve essere sottoposto a opportuni processi di test. Paradossalmente, anche le funzioni meno utilizzate devono essere verificate approfonditamente, proprio perché la normale pratica difficilmente potrebbe portare a individuarne i problemi.

Indipendentemente dalla strategia utilizzata per la scrittura dei test di unità, è importante misurarne il livello di copertura. A tal fine sono disponibili una serie di tool, molti dei quali sono illustrati all'indirizzo

http://java-source.net/open-source/code-coverage

La maggior parte di questi tool non si limita a indicare la percentuale di codice esercitata dai test, ma fornisce una serie utilissima di dati. Tra questi ci sono anche le classi/metodi non verificati, la traccia dei percorsi analizzati, etc. Pertanto, nella produzione dei test, invece di aggiungerne altri in modo randomico, è più conveniente introdurre test mirati che vadano a esercitare proprio le parti di codice non controllate da appositi test di unità. Ciò permette di evitare un'inutile ridondanza nella produzione di test, da evitare al fine di minimizzare il processo di manutenzione degli stessi test generato dalla variazione del corrispondente codice del sistema.

JUnit

Introduzione

Al momento in cui viene scritto questo libro non è possibile parlare di test di unità in Java senza menzionare JUnit (http://www.junit.org/index.htm): nella comunità dei programmatori Java i due concetti sono spesso utilizzati come sinonimi. JUnit, indubbiamente, appartiene di diritto alla cerchia dei progetti *open source* più popolari nella comunità dei programmatori Java e non solo. Tanta è la popolarità di questo strumento da avergli fruttato il titolo di strumento standard *de facto* per la produzione di test di unità in Java.

Sviluppato inizialmente da Kent Beck e Eric Gamma nel contesto del movimento XP, Martin Fowler, ha commentato JUnit in mondo trionfalistico scrivendo che "mai, nella disciplina dell'ingegneria del software, così tanto è stato dovuto a così poche linee di codice".

Quantunque l'obiettivo di questo capitolo non sia presentare in modo diffuso JUnit (esistono interi libri dedicati all'argomento, cfr. [JUNREC], [JUNACT], e [JUNPCK]), si è ritenuto che una minima conoscenza di questo framework sia il prerequisito irrinunciabile per la comprensione di quanto riportato di seguito. L'introduzione del JDK5, inoltre, ha avuto un grosso impatto non solo sui package interni di Java ma anche sulla maggior parte delle librerie fornite da terze parti. A questa regola non fa di certo eccezione JUnit che, come si vedrà a breve, ha subito significative variazioni.

Tuttavia, poiché la versione JUnit 4, al momento in cui viene scritto questo libro, non è ancora largamente utilizzata e poiché non sempre è possibile abbandonare il JDK4, si è deciso di mostrare, brevemente le due versioni.

JUnit prima della versione 4

Tradizionalmente una classe di test presentava una struttura simile a quella riportata di seguito.

```
0 package com.mb.mypackage;
1
2 // ——— third party imports ———
3 import junit.framework.TestCase;
4
5 public class MyTest extends TestCase {
6
7   public void testBasicTest() {
8
9     UserVO newUser = UserFactory.getUser(0);
10      assertNotNull(newUser);
11      . . .
12    }
```

Dall'analisi di tale listatino è possibile evidenziare gli elementi fondamentali delle classi di test. Questi erano:

- la classe di test doveva necessariamente ereditare dalla classe `TestCase` (linea 5);
- il nome dei metodi di test doveva necessariamente iniziare con il suffisso `test` (linea 7), al fine di poter essere riconosciuti come tali tramite i meccanismi della riflessione (package `java.lang.reflect`);
- all'interno dei vari metodi di test era necessario utilizzare le varie versioni dei metodi assert, come per esempio `assertNotNull` (linea 10).

Oltre questi elementi base, inoltre, era possibile definire del comportamento da invocare, rispettivamente, all'atto dell'avvio del test (`setUp()`) e alla conclusione (`tearDown()`). Metodi utili sia per allocare e rilasciare risorse (come per esempio connessioni al database), sia per inizializzare oggetti successivamente verificati. Pertanto, questi metodi permettono di evitare notevoli duplicazioni di codice. In effetti, se non ci fossero sarebbe necessario, per ogni test, riportare il codice necessario per impostare le condizioni necessarie per l'avvio dei test stessi e per la chiusura pulita, soprattutto in caso di errore.

Il listatino 3 mostra un esempio di utilizzo dei metodi di `setUp` and `tearDown`. In particolare, la procedura di `setUp` si occupa di inizializzare il "componente" del business service layer da verificare (`UserBS`). Questo, a sua volta, utilizza tre "componenti" del livello inferiore: business object. Poiché l'obiettivo del test è verificare il componente e non l'integrazione di diversi layer, le classi del livello BO sono sostituite da opportuni *mock up* (oggetti che implementano la stessa interfaccia ma che presentano semplici implementazioni). Il metodo di chiusura si

occupa di riportare metodi non invocati : se si definisce il mock up di un metodo è legittimo attendersi che questo debba essere invocato.

```
protected void setUp() {

   orgUnitBO = EasyMock.createMock(IOrgUnitBO.class);
   profileBO = EasyMock.createMock(IProfileBO.class);
   userBO  = EasyMock.createMock(IUserBO.class);

   UserBS userBS = new UserBS();

   userBS.setUserBO(userBO);
   userBS.setProfileBO(profileBO);
   userBS.setOrgUnitBO(orgUnitBO);

   this.userBS = userBS;
}


protected void tearDown() throws Exception {

   EasyMock.verify(new Object[] { userBO, profileBO, orgUnitBO });

}
```

Una buona pratica di programmazione consiste anche nello scrivere una classe (TestSuite) atta a invocare le varie classi di test. Lo scheletro di tale classe assume una forma del genere:

```
public class MyUnitSuite {

   // ———————————————— CONSTANTS SECTION ————————————————-

   // list of test classes to run
   private static Test[] testToRun = {
      XYZTest.suite(),
      MyTest.suite()
   };

   // ———————————————— ATTRIBUTES SECTION ————————————————-

   // ———————————————— METHODS SECTION ————————————————-

   /**
```

```
 * Return the test suite defined in this class
 * Other test classes has to be added in the testToRun attribute
 * @return TestSuite - test suite related to the VO classes
 */
public static TestSuite suite() {

  TestSuite suite = new TestSuite("My Test Suite");

  for (int ind = 0; ind < testToRun.length; ind++) {

    suite.addTest(testToRun[ind]);
  }

  return suite;
}
```

JUnit versione 4

La versione 4 del framework presenta alcune significative differenze dovute principalmente all'inclusione delle annotazioni introdotte con la versione Java 5.0.

> Brevemente, il meccanismo delle annotazioni permette di introdurre, direttamente nel codice sorgente, delle meta-informazioni. Queste rappresentano informazioni, più precisamente "dichiarazioni", circa il codice stesso. Il meccanismo è del tutto equivalente a quello utilizzato dal parser JavaDoc per generare la documentazione: vi è una parola chiave, preceduta dal carattere @. Tuttavia, in questo caso, l'utilizzo è decisamente più circoscritto per evitare ambiguità. Per esempio è possibile dichiarare che un metodo sia deprecato premettendo alla relativa dichiarazione la parola chiave @deprecated:
>
> ```
> @deprecated public void destroy()
> ```
>
> Si tratta del metodo destroy della classe Thread, izialmente ideato per sopprimere i thread in maniera brusca non consentendogli neanche di rilasciare eventuali monitor. Per questa ragione l'implementazione di questo metodo rappresenta un ottimo sistema per generare dead-lock.

Il listato seguente mostra l'adattamento dell'esempio riportato nel primo listato mostrato, modificato in base alle direttive della versione JUnit 4.

```
0 package com.mb.mypackage;
1
2// ——— third party imports ———
3 import static org.junit.assert.assertNotNull;
```

```
4 import org.junit.Test;
5 public class MyTest {
6
7   @Test public void BasicTest() {
8
9     UserVO newUser = UserFactory.getUser(0);
10      assertNotNull(newUser);
11      . . .
12    }
```

In particolare:

- non occorre importare il package: junit.framework.TestCase;
- bisogna importare le annotazioni di test (org.junit.Test);
- non è più necessario che le classi di test estendano dalla classe TestCase;
- non è obbligatorio aggiungere il suffisso test ai metodi di test: questi sono evidenziati attraverso l'utilizzo dell'annotazione @Test.

Qualora poi, si voglia far sì che la versione 4 del framework JUnit esegua correttamente alcune classi di test scritte per una versione precedente, è necessario includere il seguente metodo:

```
public static junit.framework.Test suite() {
  return new JUnit4TestAdapter(<nome della classe di test>.class);
```

L'uso delle annotazioni, come lecito attendersi, non è limitato ai soli metodi di test, ma è utilizzato in altri contesti. Per esempio, è possibile utilizzare le annotazioni @Before e @After, contenute nei package org.junit.Test.Before e org.junit.Test.After, per definire comportamento da eseguire, rispettivamente, prima e dopo l'esecuzione di ogni metodo di test. Queste annotazioni vanno premesse a opportuni metodi di inizializzazione e rilascio dei test, ed è possibile definirne quanti se ne vuole. Da notare che qualora si decida di implementare una classe antenata di test, tutte le classi figlie erediteranno i vari metodi @Before e @After.

Nella versione JUnit 4 è possibile dichiarare delle annotazioni, @BeforeClass e @AfterClass, che sono invocate una sola volta per tutti i test. Coerentemente con la relativa definizione è possibile specificarne una sola coppia per classe.

Per terminare questa sezione, è importante considerare che l'annotazione @Test permette di definire diversi parametri, come per esempio:

- eventuali eccezioni attese: @Test (expected=<nome eccezione>.class); test definiti in questo modo hanno successo solamente se l'eccezione attesa è effettivamente eseguita;

- un eventuale timeout: `@Test (timeout=6000)`; questo parametro permette di specificare il tempo massimo, espresso in millisecondi, a disposizione del test per essere eseguito: un'eventuale durata eccedente il timeout genera l'automatico fallimento del test.

Un'altra annotazione degna di nota è `@Ignore`. Questa, come lecito attendersi, permette di ignorare un test e di includere una stringa che riporti la motivazione dell'esclusione.

Direttive

5.1 Investire nei test di unità

Sebbene questa regola possa sembrare scontata, si continuano a trovare sistemi privi di test di unità. La mancanza di tali test, inevitabilmente, conferisce minore robustezza al codice, rende più rischiosi i processi di aggiornamento e quindi riduce la voglia/possibilità di eseguire processi di refactoring, aumenta il grado di difficoltà dell'attività di integrazione, e spesso ne riduce la comprensibilità. Per alcuni tecnici, un sistema non dotato di test automatici è un sistema non funzionante per definizione.

Indipendentemente dall'impegno profuso nell'attività di disegno del sistema e dalla meticolosità utilizzata nello scrivere il codice, è sempre possibile/frequente commettere errori. Pertanto la presenza di test automatizzati e quindi ripetibili fa spesso la differenza tra codice robusto e non.

5.1.1 Investire nella qualità dei test di unità

Dalla definizione dei test di unità consegue che il rapporto esistente tra questi test e il codice testato dovrebbe essere un'approssimazione della relazione 1 a 1. Giocoforza, gli stessi test finiscono per far parte del sistema stesso. Pertanto non dovrebbe sorprendere il fatto che processi di refactoring del codice generino importanti ripercussioni sui test di unità. Logica conseguenza quindi è che i processi di refactoring non sono limitati al codice sorgente ma si estendono, necessariamente, ai test. Al fine di semplificare il relativo processo di aggiornamento è quindi necessario che questi presentino un adeguato livello di qualità, altrimenti si potrebbe giungere alla fastidiosa situazione caratterizzata dal processo di sviluppo rallentato dalle attività di manutenzione dei test.

5.1.2 Scrivere i test di unità non è un'attività banale

Come si vedrà nel corso di questo capitolo, scrivere test di unità non è assolutamente un'attività banale. I test di unità sono parte integrante del sistema e come tali necessitano di essere rivisitati e aggiornati con l'evoluzione del sistema; la loro scrittura richiede la conoscenza di una serie di strumenti e tool quali i sistemi di controllo di copertura, di implementazione di oggetti mock up etc. La scrittura dei test di unità, inoltre, fornisce una prima valutazione sulla qualità del disegno delle parti oggetto di verifica, e così via. Pertanto, la scrittura dei test di

unità è un'attività che va pianificata accuratamente, assegnata a personale qualificato e soprattutto eseguita contemporaneamente alla scrittura del relativo codice.

5.1.3 Evitare l'over-engineering

Implementare i test ponendo attenzione alla loro qualità non equivale assolutamente a dissipare tempo nella scrittura di test ultra flessibili, super efficienti, etc. I test sono un'importante parte dello sviluppo del software, sono oggetto di continui refactoring, ma comunque non è opportuno renderli più flessibili e sofisticati di quello che dovrebbero effettivamente essere. Il tempo necessario per implementare test eccessivamente raffinati, *over-engineered*, per metterli a punto, per spiegarli ad altre persone, etc., potrebbe essere investito in maniera decisamente migliore.

5.1.4 Evitare scrivere metodi di test eccessivamente lunghi

Ogni classe di test, come visto in precedenza, dovrebbe far riferimento a un ben definito componente, tipicamente a una classe. Inoltre è opportuno che queste classi contengano una serie di semplici metodi di test, ciascuno demandato alla verifica di uno specifico e limitato aspetto.

Ciò sia per far sì che i test siano più facilmente comprensibili e mantenibili, sia per facilitare l'analisi dei report prodotti dall'esecuzione dei test: tool come JUnit, infatti, si limitano a riferire solo il primo test fallito di una batteria di test.

L'inconveniente relativo alla scrittura di molti semplici metodi di test potrebbe risiedere nella necessità di dove ripetere diverse volte una stessa porzione di codice. Questa eventualità in passato veniva risolta scrivendo opportuni metodi privati alla classe di test da invocare nel codice di ogni metodo di test. Ciò non è più necessario con la versione 4 di JUnit e in particolare grazie all'introduzione dell'annotazione `@Before`.

Nel codice seguente, viene mostrato come decomporre un unico test complesso suddividendolo in alcuni test semplici.

Test complesso

```
@Test
public void testABC() {
  // inizializzazione dei singoli
  // test. Diverso da setUP!
  assertTrue(condition1);
  assertNotNull(o);
  assertEquals(o1,o2);
}
```

Test semplici

```
@Before
public void testInit() {
  // inizializzazione dei singoli
```

```
}

@Test
public void testCondition1() {
  assertTrue(condition1);
}

@Test
public void testCondition2() {
  assertNotNull(o);
}

@Test
public void testCondition3() {
  assertEquals(o1,o2);
}
```

5.1.5 Scrivere i test di unità insieme alle relative classi

Questa regola potrebbe risultare abbastanza controversa. In effetti, vi è tutto il movimento basato sullo "sviluppo guidato dai test" (TDD, *Test Driven Development*), di cui XP è una delle istanze più popolari, che evoca la scrittura dei test addirittura prima della scrittura delle stesse classi da implementare. In sostanza si tratterebbe di un vero e proprio strumento per disegnare il sistema.

Questa regola non intende assolutamente fornire direttive circa il processo di sviluppo da adottare, tuttavia è importante che i test di unità siano prodotti con il codice stesso. Questo sia per evitare processi di scrittura dei test frettolosi e tipicamente superficiali, sia per far sì che questi assolvano al loro compito: permettere il rilascio di codice di qualità e quindi semplificarne il processo di integrazione, fornire informazioni circa l'utilizzo delle varie classi, etc.

5.1.6 Non assegnare la scrittura dei test al personale junior

Un altro problema che spesso interviene nella redazione dei test di unità è che la loro scrittura sia demandata, immediatamente prima della consegna, a programmatori junior. Questa pratica, spesso, si traduce nella scrittura veloce e superficiale dei vari test e quindi genera una riduzione o addirittura l'annullamento dei vantaggi propri della scrittura dei test di unità. I test, come visto in precedenza, vanno scritti non appena una classe si rende disponibile o, se si desidera abbracciare tecniche estreme, addirittura prima di redigere il codice stesso.

Una pratica spesso utilizzata consiste nel far scrivere le classi e i relativi test a diverse persone. Ciò però è ben diverso dallo scrivere i test di corsa poco prima della consegna del sistema.

5.2 Utilizzare JUnit

JUnit è uno dei framework open source di maggiore successo, tanto che ormai è diventato lo standard de facto per la produzione di test automatizzati in Java. Si tratta di un framework

ben progettato, efficace, collaudato, divenuto utilizzato come normale strumento di lavoro per tutti i programmatori Java, etc. Pertanto il suo utilizzo non dovrebbe essere oggetto di discussione.

5.2.1 Utilizzare gli Assert

Gli assertXXX (assertTrue, assertNotNull, AssertEquals, Fail, etc.) rappresentano uno dei meccanismi fondamentali del framework JUnit. In particolare questi metodi permettono a JUnit di verificare l'esito dei vari test. Pertanto, ogni metodo di test deve includere almeno uno statement di assert. Sebbene questa regola possa sembrare piuttosto ovvia, può capitare di imbattersi in codici di test in cui la verifica sia affidata a meccanismi diversi, tra cui eccezioni, stampe a video, etc. Un altro valido accorgimento consiste nell'utilizzare il campo etichetta degli assert, specialmente nei casi in cui uno stesso metodo di test ne contenga diversi. Sebbene l'etichetta sia un campo opzionale, in caso di fallimento può semplificare il processo di individuazione del problema, anche se i tool moderni permettono, attraverso un click del mouse, di posizionarsi direttamente sull'assert non riuscito.

5.2.2 Non utilizzare il System.out.println a fini di test

Può capitare di visionare metodi di test in cui la verifica del corretto funzionamento del sistema sia affidato a stringhe inviate a video (System.out.println). Si tratta chiaramente di una pratica fortemente sconsigliata per una serie di motivi: il più importante è che la verifica del corretto funzionamento del sistema è così demandata alla presenza di persone. Pertanto, ciò rende impossibile automatizzare le verifiche, eseguire continui controlli anche durante orari notturni, etc. Inoltre, rende gli stessi controlli poco efficaci e decisamente complessi. La verifica di algoritmi non banali, infatti, spesso richiede di controllare molteplici informazioni, difficilmente verificabili attraverso semplice ispezione di schermate video. Si considerino per esempio le tipiche schermate che riportano i prezzi di prodotti finanziari.

5.2.3 Implementare le classi TestSuite

Nel disegno e nell'implementazione di sistemi software è possibile individuare una serie di "macro-componenti" i cui elementi sono accomunati da importanti caratteristiche. Per esempio, una tipica architettura multistrato di un sistema prevede:

- strato di presentazione (*presentation layer*);
- strato di business service (*business service layer*);
- srato di business object (*business object layer*);
- strato di integrazione (*integration layer*).

Alcuni di questi strati, poi, sono ulteriormente composti da sottostrati; per esempio, lo strato di integrazione tipicamente è suddiviso per le varie sorgenti integrate: database, sistema di messaggistica, etc. Inoltre, sono presenti ulteriori componenti atti a ospitare elementi base,

come eccezioni comuni, value object utilizzati per scambiare informazioni tra i vari strati dell'architettura etc.

Pertanto è opportuno far sì che tutti i test degli elementi di macro-componenti (spesso anche di importanti package) siano invocabili da un'opportuna classe "suite test". Questa, a sua volta dovrebbe essere inclusa in una suite a livello più generale fino a giungere a quella di livello globale.

Da notare che le singole classi di test (elementi foglia) e le varie suite (elementi composti) sono state implementate secondo le direttiva del Composite Pattern e quindi è possibile avere un numero illimitato di classi "suite test".

5.2.4 Inserire le classi di test in una struttura equivalente a quella delle corrispondenti classi verificate

Nell'implementazione delle classi di test è molto utile far sì che il package (e non il path) delle classi di codice e di test coincidano, come riportato in figura 5.2.

Ciò è molto utile per fare in modo che le classi di test possano accedere agevolmente alle risorse *friendly* (elementi con visibilità di default) delle classi da verificare senza interferire nella struttura.

Non è infrequente che alcuni sviluppatori decidano di sistemare le classi di test insieme alle corrispondenti classi implementate. Chiaramente ciò non è assolutamente una buona pratica, in quanto rende meno chiari i package di codice, rende meno agevole il processo di implementazione (bisogna sempre porre molta attenzione alle classi da aprire), rende più difficile l'impacchettamento in JAR dei sorgenti, e così via.

5.2.5 Assegnare alle classi di test lo stesso nome delle classi testate

Come visto in precedenza, nel contesto dei test di unità, ogni classe di test dovrebbe verificare il comportamento di una classe del sistema. Pertanto, la convezione utilizzata per i nomi delle classi di test consiste nel riportare il nome della classe verificata con l'aggiunta del suffisso "test", come mostrato in figura 5.2.

5.2.6 Utilizzare la parola "test" per enfatizzare metodi di test

Qualora si utilizzi una versione JUnit precedente alla 4.1, è obbligatorio premettere al nome dei metodi la parola "test" per distinguere i metodi di test (che quindi devono essere invocati dal framework) dagli altri di utilità. Qualora invece si utilizzi una versione JUnit 4.1 (o superiore), i metodi di test sono individuati per mezzo dell'apposita annotazione: `@Test`. Tuttavia, rappresenta una buona strategia identificare i metodi di test riportando questo nome direttamente nel nome del metodo.

5.3 Test di unità e mock objects

I test di unità, per definizione, sono utilizzati per verificare che un particolare frammento di codice presenti il comportamento atteso. Pertanto, il campo d'azione dei test di unità è costituito da singole classi e, in casi particolari, da singoli package. Domini di maggiore estensione sono oggetto dei test di integrazione e di sistema.

Il problema di mantenere i test di unità nel proprio ambito di applicazione deriva dal fatto che i sistemi Object Oriented tendono naturalmente a essere costituiti da una miriade di classi interrconnesse. Quindi il corretto funzionamento di ciascuna di queste, quasi sempre, dipende dal corretto funzionamento di altre classi, e di componenti, package, sistemi esterni, e chi più ne ha più ne metta.

Per esempio, spesso l'erogazione di determinati servizi è subordinata all'autenticazione e autorizzazione dell'utente che sono generalmente responsabilità di un sistema esterno. Ancora, in architetture multi-tier, componenti appartenenti allo strato di presentazione dipendono da quelli del business service layer; questi, a loro volta, dipendono da quelle appartenenti al business object layer, e così via.

Come fare quindi ad implementare test di unità che non eccedano il proprio campo d'azione?

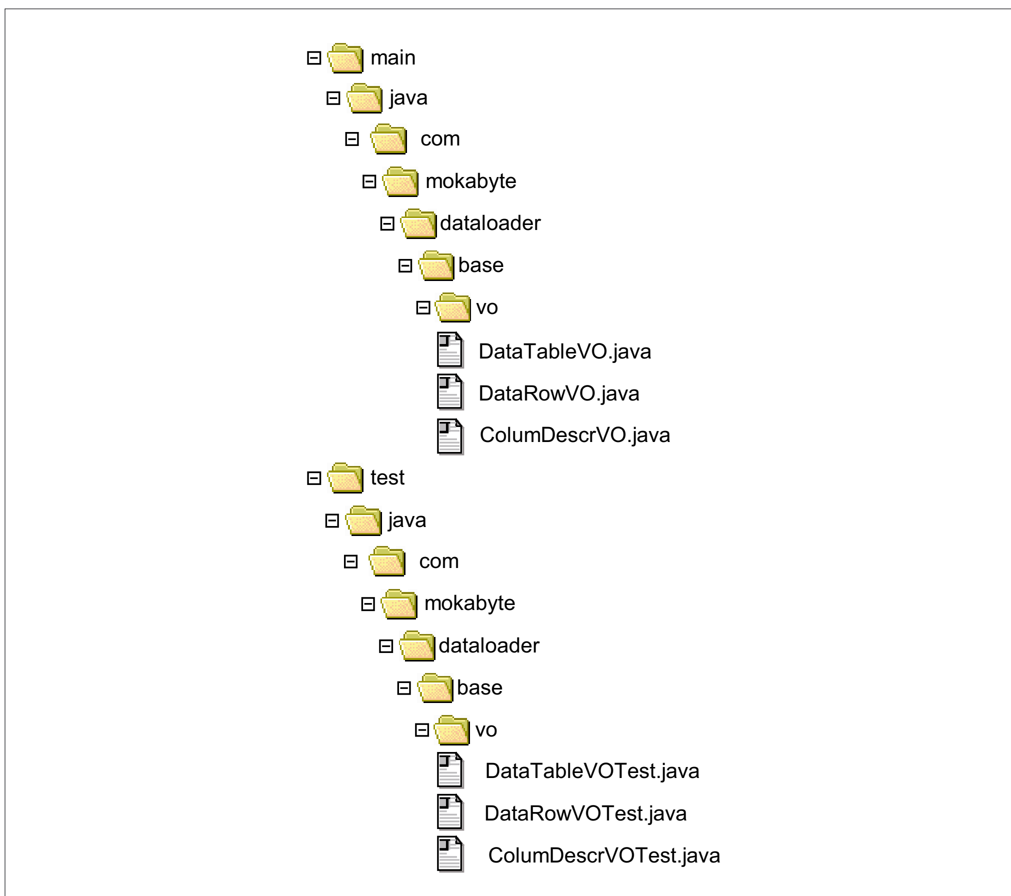

Figura 5.2 – Package delle classi di test e di quelle di codice.

La risposta viene dai *mock objects*. Si tratta di oggetti che fingono il comportamento di determinati oggetti reali di cui si ha bisogno. In particolare, questi oggetti, sviluppati rapidamente grazie ai relativi framework, implementano la medesima interfaccia degli oggetti reali simulati; però, invece di implementarne il comportamento reale, forniscono risposte pre-definite e/o comportamenti molto semplici. Si consideri per esempio un oggetto che si occupi di effettuare l'autenticazione degli utenti di un sistema, magari eseguendo un'invocazione remota. Il relativo mock up potrebbe limitarsi a fornire sempre una risposta affermativa oppure autenticare solo gli utenti il cui id è memorizzato in un'apposita lista eventualmente hard-coded, e così sia… Quindi i mock objects servono per sostituire, con semplici simulazioni, le entità che collaborano con quella oggetto di test.

Al termine di questa spiegazione viene riportato un listato. L'oggetto del test è una classe che fornisce il servizio di autenticazione (AuthenticationService). A tal fine, utilizza i servizi esposti dalle classi ProfileBO e UserBO, entrambe localizzate nel Business Object layer (da cui il suffisso BO). Poiché si tratta di un test di unità (e non di integrazione) non è necessario coinvolgere anche queste classi nel test in questione. Quindi, in questo contesto, al loro posto è possibile utilizzare appositi mock objects. La verifica del corretto funzionamento delle classi in questione (ProfileBO e UserBO), come lecito attendersi, è affidato alle relative classi di test.

Per la realizzazione delle classi di mock up, si è deciso di utilizzare il tool EasyMock (http://www.easymock.org/) la cui guida all'uso non è oggetto di questo libro. Il funzionamento base, però, è abbastanza semplice:

1. si genera l'implementazione di mock up relativa alle classi di cui è necessario simulare il comportamento (per esempio: IProfileBO profileBO = EasyMock.createMock(IProfileBO.class));

2. per ciascun metodo degli oggetti simulati coinvolti nel test:
   a. si dichiara la relativa invocazione corredata dai parametri attesi (per esempio: profileBO.findActiveProfileByLoginId("invalid"));
   b. si specifica la risposta attesa (per esempio EasyMock.expectLastCall().andReturn(null););

3. quando tutti i metodi richiesti dal test sono stati specificati, è necessario far sì che il framework ne prenda atto; ciò fa sì che l'oggetto simulante, dopo aver memorizzato invocazioni e risposte, si predisponga a ripetere le risposte (per esempio EasyMock.replay(new Object[] { userBO, profileBO})).

Il listatino di seguito riportato presenta due semplici test il primo negativo e il secondo positivo. In particolare, il primo serve a verificare che a fronte di una richiesta relativa a un utente non presente nel sistema (il login specificato non appartiene ad alcun utente), il sistema sia in grado di rilevare l'anomalia e quindi lanci un'opportuna eccezione. Il secondo test invece verifica che qualora tutti i dati siano corretti, il sistema sia in grado di autenticare l'utente specificato.

```
public class AuthenticationServiceTest extends TestCase {
  /** The profile BO. */
```

```
private IProfileBO profileBO = EasyMock.createMock(IProfileBO.class);
/** The profile BO. */
private IUserBO userBO    = EasyMock.createMock(IUserBO.class);
/** The authentication service. */
private AuthenticationService authenticationService = null;

/** Default Constructor. */
public AuthenticationServiceTest() {
   authenticationService = new AuthenticationService();
   authenticationService.setProfileBO(profileBO);
   authenticationService.setUserBO(userBO);
}

/**
 * This method verifies that the service is able to detect
 * requests related to not existent user.
 * This is a negative test
 * @throws Exception  a problem occurred during the process
 */
public  void testInvalidLoginId() throws Exception {

   profileBO.findActiveProfileByLoginId("invalid");
   EasyMock.expectLastCall().andReturn(null);

   userBO.findRefUserByLogin("invalid");
   EasyMock.expectLastCall().andReturn(null);
   replay();

   try {
      authenticationService.authenticate(createSubject("invalid"));
      fail(" MySystemSecurityException should be thrown");
   } catch ( MySystemSecurityException e) {
      // ignore this exception
      assertTrue(true);
   }
}

/**
 * This method verifies that the service, in case everything is correct
 * is able to correctly deliver the authentication service
 * @throws Exception  a problem occurred during the process
 */
public void testCorrectUserAuthentication() throws Exception {
```

```
        RefUserDTO correctUser = new RefUserDTO();
        correctUser.setLogin("correctUser");
        correctUser.setPassword(generateTmpPassword());
        correctUser.setStatus(UserStatusDTO.USER_ACTIVE);
        correctUser.setValidFrom(new Date(new Date().getTime() - 1000000));
        correctUser.setValidTo(new Date(new Date().getTime() + 1000000));

        RefProfileDTO correctProfile = new RefProfileDTO();
        correctProfile.setStatus(ProfileStatusDTO.PROFILE_ACTIVE);
        correctProfile.setValidFrom(new Date(new Date().getTime() - 1000000));
        correctProfile.setValidTo(new Date(new Date().getTime() + 1000000));
        correctProfile.setUser(correctUser);

        profileBO.findActiveProfileByLoginId(correctUser.getLogin());
        EasyMock.expectLastCall().andReturn(correctProfile);

        replay();

        try {
          authenticationService.authenticate(
            createSubject( correctUser.getLogin()));
        } catch { (MySystemSecurityException e) {
          fail("MySystemSecurityException should not be thrown");
        }
      }
    }
```

5.3.1 Utilizzare i test di unità per verificare singoli componenti

La prima regola di questa sezione non poteva che prescrivere di utilizzare i test di unità compatibilmente al relativo principio base: verificare che particolari frammenti di codice presentino il comportamento atteso. Ciascun test, quindi, dovrebbe verificare una determinata caratteristica considerata isolatamente. Ora, in sistemi non banali, la verifica di singole parti considerate isolatamente è un'attività spesso molto complessa. Pertanto è consigliabile utilizzare appositi framework progettati per consentire ai programmatori di generare rapidamente oggetti di mock up: semplici oggetti in grado di simulare specifici comportamenti dei corrispondenti oggetti del sistema reale.

Il ricorso a oggetti di mock up permette di implementare test di unità più approfonditi e più dettagliati e quindi, in ultima analisi, sistemi di maggiore qualità.

5.3.2 Implementare un sistema facilmente verificabile

Questa regola, probabilmente, potrebbe appartenere ai capitoli relativi l'implementazione del sistema e non dei relativi test. Tuttavia, alcuni problemi di disegno si scoprono proprio

durante la scrittura dei test. Inoltre, questa regola è di facile comprensione nel contesto dei mock objects.

Tutti i sistemi dovrebbero essere realizzati considerando, tra gli altri fattori, l'attività di test. Ciò non significa che si debba cambiare il disegno del sistema per semplificarne la verifica, come per esempio implementare tutti metodi pubblici in modo da semplificare l'implementazione dei relativi test; si tratta però di considerare, nella scelta delle diverse alternative implementative di uno stesso disegno, anche la facilità di verifica.

Per esempio, qualora una classe utilizzi i servizi esposti da altre classi (come nel listato precedente `AuthenticationService` utilizza i servizi esposti dalle classi `ProfileBO` e `UserBO`), una buona implementazione consiste nel fornire i riferimenti a tali istanze attraverso il costruttore di `AuthenticationService` o per mezzo di opportuni metodi di set (come nel caso in questione). In altre parole, si tratta di utilizzare il pattern Inversion Of Control (inversione del controllo). Ciò non solo risolve correttamente il disegno: al tempo stesso, ne semplifica il test.

Un'altra alternativa valida consiste nell'utilizzare il pattern di Factory. In questo caso però, nella scrittura dei test è necessario estendere la classe Factory con una necessaria ai fini di test. Questa classe estesa, in particolare, si deve occupare della generazione degli oggetti mock up utilizzati durante il test.

Uno scenario che invece rende complessa l'implementazione dei test si ha quando la generazione delle istanze delle classi utilizzate avviene all'interno del codice. In questo caso la sostituzione degli oggetti reali con appositi mock up è un'attività piuttosto complessa.

5.3.3 Utilizzare mock objects per simulare condizioni difficilmente riproducibili

La realizzazione di alcuni test di unità può risultare estremamente complessa. Questo è il caso in cui sia necessario verificare la risposta del sistema a situazione anomale, come per esempio connessioni perse durante la sessione, un sistema remoto che transita in uno stato di errore, e così via. Queste situazioni si prestano a essere simulate efficacemente attraverso i mock objects. In particolare, questi framework possono essere utilizzati o per implementare un proxy del server oppure un'opportuna simulazione del server stesso. E quindi è possibile decidere, programmaticamente, quando far simulare comportamenti corretti e quando far generare degli errori.

5.4 Utilizzare tool di analisi della copertura

Utilizzare un test di copertura è assolutamente importante per una serie di motivi.

- Disporre di una valutazione quantitativa del livello di copertura dei test. Per alcuni tecnici (estremisti) questi indici, indirettamente, costituiscono una misura del livello di qualità del sistema. Di sicuro c'è la necessità di aver ben chiaro il livello di affidabilità dei test di regressione e, in ultima analisi, il valore del superamento dei test di unità. Non è infrequente infatti che i team di sviluppo abbiamo la falsa sensazione di aver prodotto un codice di alta qualità giacché il sistema passa indenne i vari test anche quando questi coprono esclusivamente una piccola percentuale del codice.

- Individuare aree di codice non testate sufficientemente e che quindi beneficerebbero dall'aggiunta di nuove classi di test. Questo evita inutile dispendio di tempo e denaro derivante dall'aggiunta di test in modo randomico quando invece occorre aggiungere test solo per affrontare le aree effettivamente non coperte opportunamente.

- Individuare test ridondanti che risultano deleteri nel momento in cui sia necessario eseguire delle operazioni di refactoring che necessariamente si riflettono anche sulle classi di test. Quindi, maggiori sono le classi di test, maggiore è l'impegno richiesto dai processi di refactoring.

5.4.1 Eseguire periodicamente il test di copertura

Come appena visto, è molto importante utilizzare appositi tool di analisi della copertura dei test. Chiaramente si tratta di una condizione necessaria ma non sufficiente. È necessario eseguire frequentemente l'analisi della copertura per esaminarne i risultati prodotti. In particolare, bisognerebbe eseguire l'analisi ogni qualvolta si includano nuove classi, siano queste di codice o di test. In genere non è necessario eseguire una nuova analisi per ogni singola classe aggiunta (a meno che questa non sia una classe che include un numero significativo di test); è sufficiente eseguire una nuova scansione all'aggiunta di nuovi package, dopo sostanziali processi di aggiornamento/refactoring. Un discorso diverso ovviamente vale qualora si stia cercando di razionalizzare i test: aggiunta di test atti ad aumentare la copertura, rimozione di test ridondanti, e così via. In questo caso può aver senso eseguire i test di copertura più frequentemente.

5.4.2 Aggiungere nuovi test in maniera oculata

Il livello di copertura del codice da raggiungere è un valore che dipende da diversi fattori, per esempio il dominio dell'applicazione. Per esempio, è plausibile che dovendo implementare un sistema di controllo di un reattore nucleare si voglia cercare di raggiungere una copertura molto prossima al 100% (sebbene questo non ne garantisca automaticamente il corretto funzionamento…), mentre è altrettanto probabile che si accetti un fattore di copertura decisamente più rilassato quando si implementa un sistema di gestione degli ordini.

In ogni modo, qualora il livello di copertura ottenuto non risulti conforme a quello prestabilito, è opportuno implementare ulteriori classi di test. Tuttavia, invece di procedere come spesso avviene (implementare nuovi metodi in modo randomico, il che aumenta ben poco il livello di copertura), è consigliabile studiare i risultati dell'analisi di copertura al fine di identificare aree poco esercitate dai test e che quindi beneficierebbero maggiormente di ulteriori verifiche. Chiaramente è opportuno anche fare in modo che aree di particolare interesse/sensibilità presentino un livello molto elevato di copertura.

Nella produzione di test è facile notare come sia abbastanza semplice e veloce raggiungere coperture dell'ordine del 60-70% e come poi piccoli incrementi richiedano un elevato impegno. Pertanto è opportuno mirare attentamente le aree oggetto di test.

5.4.3 Valutare la possibilità di rimuovere test ridondanti

La presenza di test ridondanti non è un grosso problema, tuttavia finisce per accrescere l'impatto dei processi di refactoring. Questi, durante l'ultimo decennio, sono diventati una

prassi nella pratica lavorativa, principalmente come conseguenza dell'adozione sempre più frequente di processi di sviluppo del software che incorporano approcci iterativi ed incrementali.

5.5 Scrivere test chiari ed efficaci

Alcuni programmatori proclamano di considerare non funzionante, parzialmente o interamente, qualsiasi codice privo di una copertura estesa di test automatici. Sebbene tale affermazione sia abbastanza estrema, è necessario considerare che la presenza di test scritti rapidamente, in modo superficiale e senza un'attenta analisi può generare una serie di conseguenze negative, per esempio la falsa sensazione di aver prodotto un codice perfettamente funzionante. Sebbene questo problema possa essere minimizzato grazie software atti a misurare la copertura dei test di unità, non sempre è possibile valutare l'efficacia dei vari test prodotti. In particolare, non è immediato verificare se i test coprano correttamente scenari di errore, verifichino le dimensioni delle strutture dati e così via. Pertanto è importante produrre i test in modo che questi siano veramente utili ed efficaci.

5.5.1 Limitare l'utilizzo dei costruttori nelle classi di test

Nell'implementazione dei test di unità, è necessario limitare l'utilizzo dei metodi costruttori. In particolare, è opportuno non implementare costruttori:

- di test, per i quali esistono gli appositi metodi; il problema principale è che qualora si verifichi un problema nel metodo costruttore, questo verrebbe riportato in termini di un'eccezione invece che del fallimento di una assertion;

- di procedure di inizializziazione dei test; a tal fine esistono opportuni metodi di setup.

5.5.2 Implementare classi di help

Non è infrequente il caso in cui l'esecuzione di diversi test JUnit richieda di disporre di una stessa serie di oggetti prepopolati che costituiscono appunto la base del test. In questi casi, al fine di evitare lavoro tedioso e ridondanze del codice, è opportuno creare delle classi *helper* o Object builder atte proprio a fornire oggetti prepopolati necessari per l'esecuzione dei vari test. Queste classi, dovrebbero essere implementate a livello di package.

Per esempio, potrebbe risultare necessario disporre di un generatore di "oggetti" di prova di tipo utente, sia per verificare i metodi della classe stessa, sia per verificare il funzionamento di altri servizi come quelli di autenticazione, di rilascio di un nuovo profilo, di memorizzazione di informazioni di auditing, e così via. In effetti, è abbastanza normale che, all'interno di un sistema, una stessa classe dati (DTO/VO) sia utilizzata in diversi contesti.

Riportiamo di seguito il frammento di una classe di help utilizzata per produrre istanze della classe UserDTO

```
public class UserTestHelper {
  ...
  /** users data */
```

```
public static final Object USERS_DATA[] [] = {
  {  "1", UserStatusDTO.USER_ACTIVE, "123435467" , "VeraD" ,
     "Vera", "Grigorievna","Dianova", "22-02-1982" ,
     "VG@Dianova.com" ,"VeraGD","2342342", 10-01-2006", "10-01-2008" },

  { "2", UserStatusDTO.USER_ACTIVE, "454564564", "MarcoM" ,
     "Marco", null ,"Materazzi", "19-08-1973",
     "MarcoMaterazzi@WorldCupChampion2006.com", "MarcoB", "02033423",
     "12-02-2006", "15-05-2008" },

  { "3", UserStatusDTO.USER_DEACTIVED, "093445943", "AlbertE",
    "Albert", null,"Einstein", "14-03-1879",
    "Albert@Einstein.com","AlbertE", "34567", "06-10-1886", "10-10-1890"}
};
. . .
/**
 * Return a pre-set user
 * @param index index of the requested user
 * @return the requested user
 */
public static UserDTO getRequestedUser(int index) {

  UserDTO userDTO = null;

   if ( (index > -1) && (index < USERS_DATA.length) ) {
     userDTO =
       new UserDTO(
       new Long((String)USERS_DATA[index] [0]),              ________   // id
       (UserStatusDTO) USERS_DATA[index] [1], ________ // status
       (String)USERS_DATA[index] [2],                 // gpn
       (String)USERS_DATA[index] [3],                // login
       (String)USERS_DATA[index] [4],                 // name
       (String)USERS_DATA[index] [5],       // middlename
       (String)USERS_DATA[index] [6],          // surname
       getReqDate((String)USERS_DATA[index] [7]),              // dob
       (String)USERS_DATA[index] [8],               // e-mail
       (String)USERS_DATA[index] [9],        // nick name
       (String)USERS_DATA[index] [10],              // off tel
       getReqDate((String)USERS_DATA[index] [11]),         // start
       getReqDate((String)USERS_DATA[index] [12])             // end
       );

     userDTO.addAddress(
```

```
            new UserAltAddressDTO(
            null, UserAltAddTypeDTO.ADDRESS_HOME, "Home Address"));
            userDTO.addAddress(
            new  UserAltAddressDTO(
            null , UserAltAddTypeDTO.ADDRESS_MOBILE, "Mobile phone"));
            userDTO.addAddress(
            new UserAltAddressDTO(
            null, UserAltAddTypeDTO.ADDRESS_PAGE, "Pager"));
        }
        return  userDTO;
    }
    . . .
```

5.5.3 Non variare il disegno del sistema per facilitarne il test

Durante il processo di revisione dei sistemi, può capitare di imbattersi in classi il cui codice presenti delle stranezze; per esempio, possono esserci metodi che, sebbene siano assolutamente da dichiarare privati, sono invece pubblici. Spesso si tratta di variazioni di implementazione effettuate durante la stesura del codice di test per agevolare oltremisura la scrittura dei test di unità. Si tratta ovviamente di una pratica pessima.

Sebbene il codice debba essere scritto tenendo in mente anche la facilità di verifica, ciò non significa assolutamente che il disegno sia alla mercé dei test. L'obiettivo principale della scrittura dei test è aumentare la qualità del sistema prodotto e non di ridurla.

5.5.4 Inizializzare eventuali risorse esterne

Molto spesso il corretto funzionamento di determinati test dipende dalla corretta disponibilità di risorse esterne quali database, directory, file, etc. In questi casi è opportuno che il codice di test si occupi sia di inizializzare correttamente tali risorse prima dell'avvio dei test, sia di rilasciarle a test avvenuti o in caso di errore.

5.5.5 Verificare tutti gli scenari

Nella scrittura delle classi di test è necessario implementare sia i test positivi (*positive test*), sia test negativi (*negative test*). I primi servono a verificare che, in condizioni corrette (dati in ingresso validi, sistema funzionante, e così via), l'applicazione sia in grado di produrre i risultati previsti. Mentre i test negativi servono per verificare che il sistema sia robusto, ossia sia in grado di individuare condizioni anomale, come dati in ingresso non validi, e quindi di eseguire le corrispondenti procedure di gestione, come per esempio il lancio di un'opportuna eccezione, l'esecuzione di un percorso alternativo, la ripetizione controllata dell'istruzione che ha generato il problema, e così via.

Nella scrittura delle classi di test è quindi necessario verificare anche che, in caso di errore, il sistema comunichi le eventuali eccezioni previste come mostrato nei seguenti listati. In particolare, tra quelli seguenti, il primo listato mostra mostra la versione funzionante con JUnit 4, mentre quello successivo mostra l'implementazione necessaria con le versioni precedenti.

Il seguente test ha successo solo se l'eccezione attesa è effettivamente lanciata.

```
@Test(expected=IndexOutOfBoundsException.class)
public void testIndexOutOfBoundsException() {
   ArrayList emptyList = new ArrayList();
   Object o = emptyList.get(0);
}
```

Il seguente metodo di test ha successo solo in caso in cui il sistema lancia l'eccezione attesa (lo statement catch contiene un assertTrue(true)) mentre fallisce se il servizio non la lancia. Infatti, dopo l'istruzione successiva all'invio dell'eccezione c'è una fail.

```
/**
 * This test verifies that if an invalid login
 * is specified, then the system throws a proper
 * exception
 *
 * @throws Exception – a problem occurred during the test
 */
public void testInvalidLoginId() throws Exception {

   profileBO.findActiveProfileByLoginId("invalid");
   EasyMock.expectLastCall().andReturn(null);

   userBO.findUserByLogin("invalid");
   EasyMock.expectLastCall().andReturn(null);

   replay();

   try {

      authenticationService.authenticate(createSubject("invalid"));
      fail("SecurityException should be thrown");

   } catch (MySystemSecurityException secExp) {

      assertTrue(true);
   }
}
```

5.5.6 Implementare test in grado di verificare i costrutti decisionali

Un aspetto molto importante da verificare sono i costrutti decisionali. In particolare, è opportuno scrivere diversi test in grado di verificare le combinazioni dei vari percorsi. Questa

attività può risultare decisamente tediosa, in quanto, frequentemente, il numero delle combinazioni dei percorsi presenti all'interno di ciascun metodo sono elevati. Il lato positivo è che i tool di copertura sono in grado di rilevare eventuali percorsi intentati e che non è necessario verificare ogni dettaglio di tutte le classi implementate.

5.5.7 Verificare i cicli

Questa regola è molto simile alla precedente. Tuttavia è importante sottolineare il rilievo di verificare attentamente i cicli. Verificare che questi non diano luoghi a cicli infiniti, ma che terminino quando sia previsto.

5.5.8 Verificare i limiti delle strutture dati

Qualora l'oggetto dei test da implementare sia una classe che incapsula strutture dati (tipicamente si tratta di Value Object o di Data Transfer Object) è importante verificare la capacità della stessa classe di gestire correttamente tentativi di violare i limiti delle strutture dati. Le tipologie delle violazioni sono diverse e quelle applicabili dipendono dall'oggetto utilizzato per memorizzare la struttura: array, liste, tabelle hash, etc. Per esempio, è possibile eseguire tentativi di inserire più dati di quelli gestibili, di leggere oltre l'ultimo elemento e così via. In questo caso la mancanza di opportuni test non è rivelata da tool come quelli di copertura.

5.5.9 Non implementare test con dipendenze temporali

Nella scrittura dei test di unità è importante evitare che il corretto funzionamento di alcuni test dipenda dall'avvenuta esecuzione senza errore di altri. Pertanto i test devono essere scritti senza effettuare alcuna supposizione circa l'esecuzione degli altri. Ciò è necessario qualora si utilizzi JUnit, poiché questo non fornisce alcuna direttiva circa l'ordine di esecuzione dei test inseriti in una classe. Ciò è una diretta conseguenza del fatto che JUnit utilizza i meccanismi della riflessione Java e che il JDK non fornisce alcuna garanzia circa l'ordine con cui i metodi di una classe sono restituiti qualora siano riferiti attraverso i meccanismi della reflection.

Ecco di seguito un esempio errato di test ad accoppiamento temporale:

```
public static Test suite() {

   suite.addTest(new FirstTestCase("TestToPerformFirst"));
   suite.addTest(new SecondTestCase("TestToPerformSecond"));

   return suite;
}
```

5.5.10 Valutare attentamente eventuali test con effetti collaterali

È importante valutare attentamente l'implementazione di test in grado di generare effetti collaterali, che possano per esempio cambiare lo stato del sistema, lasciare dati nel database e così via, poiché:

- altri test potrebbero fallire in quanto lo stato del sistema è diverso da quello atteso;

- tipicamente è necessario intervenire manualmente per porre nuovamente il sistema in uno stato neutro.

5.5.11 Implementare test efficienti

Per quanto l'implementazione di test caratterizzati da buone performance non sia di certo un requisito prioritario nello scrivere i test, è comunque bene puntare a soluzioni con le migliori prestazioni, quando ciò sià possibile. Sistemi di dimensioni medie e grandi sono generalmente costituiti da diverse migliaia di classi; a questo numero corrisponde una quantità confrontabile di test, che solitamente vanno eseguiti molto di frequente durante il processo di sviluppo del software. Pertanto, ricorrere a soluzioni più efficienti evita che l'esecuzione delle procedure di test impieghi tempi eccessivi. Chiaramente, scegliere soluzioni efficienti non significa eseguirne l'over-engineering.