Parte III
Test di unità

"I am not a great programmer; I am just a good programmer with great habits."

(Non sono un ottimo programmatore; sono semplicemente un buon programmatore con ottime abitudini.)

KENT BECK

Capitolo 16
Test automatico

Nei metodi di progettazione e sviluppo agile, il test automatico di unità ricopre un ruolo di primo piano, perché è il supporto indispensabile affinché i passaggi di refactoring possano essere concretizzati.

La presenza di test di unità è garanzia non solo di affidabilità del programma, ma anche di fiducia del programmatore nel codice. Solo uno sviluppatore con una consistente serie di test che gli garantiscono l'affidabilità del programma è in grado di apportare modifiche al codice esistente.

Necessità di test

Ogni sviluppatore è perfettamente consapevole del fatto che dovrebbe implementare test per permettere una verifica automatica e veloce del suo codice, ma raramente li scrive. Come mai? Quali sono i motivi? Proviamo ad analizzarne alcuni.

Fretta e termini di consegna

È risaputo che i termini di consegna sono sempre più stretti e questi mettono lo sviluppatore sotto continua pressione. Dallo sviluppatore ci si attende nuove funzionalità. Il test, in questo contesto, diventa l'anello più debole, perché non aggiunge elementi visibili al programma. Quando si ha fretta si tende quindi a tralasciarlo, ripromettendosi, invano, di aggiungerlo più tardi, quando si avrà maggiore tempo a disposizione.

Si entra in questo modo in un circolo vizioso. Più aumenta la fretta, meno tempo si investe per scrivere test. Meno test si scrivono, più errori contiene il codice, che diventa più instabile. Meno stabile è il codice, più si perde tempo a correggere errori, meno si è produttivi e più aumenta la fretta…

Eccessiva sicurezza

Chi sviluppa software deve essere sufficientemente modesto da rendersi conto che non è possibile produrre codice senza errori. Lo scopo deve essere quello di mantenere la percentuale d'errore più bassa possibile, con l'aiuto di test di unità.

L'eccessiva sicurezza è un atteggiamento sbagliato per chi sviluppa software. Ogni passaggio va verificato. Bisogna trovare il "piacere" di scoprire gli errori nel codice, perché ogni errore eliminato aumenta la stabilità del software. Scrivere test di unità, per uno sviluppatore, deve essere un'attività piacevole.

Formazione sbagliata

Sviluppare software è un'attività complessa. Trovare il giusto design e la codifica migliore non richiede un unico passaggio. Può essere il risultato di un lungo percorso iterativo. È il frutto di esperienza. Non è come risolvere un'equazione con un' unica soluzione e al massimo un paio di casi particolari da discutere. Le soluzioni sono parecchie e i casi da discutere non finiscono mai.

Usare esercizi puramente numerici per insegnare la programmazione può essere utile quando si tratta di imparare un linguaggio, ma è poco educativo quando si tratta di insegnare la progettazione, dove non conta solo il risultato finale, ma contano anche il modo in cui si è arrivati al risultato, e l'evoluzione che ha avuto il software per ottenere la forma ritenuta essere quella finale. Chi insegna la programmazione deve avere esperienza di sviluppo software, e questo non è sempre il caso.

Poca professionalità

La ricerca dell'errore fa parte dello sviluppo. L'attività di programmazione è inscindibile da quella di test. L'atteggiamento: "Io scrivo il programma, poi ci pensino pure gli altri a verificarlo…" è chiaro sintomo di scarsa professionalità.

Benefici

Utilizzando sistematicamente il test di unità, si ottengono i seguenti benefici:

- Accorciamento dei cicli di sviluppo, intesi come cicli in cui il codice cambia senza che alla fine il programma si trovi in uno stato di inconsistenza (massimo un giorno, generalmente minuti o ore). Si parla di cicli del tipo "code a little, test a little".

- Aumento dell'autostima e della fiducia nel proprio software, e questo non è sicuramente cosa da poco, se si considera che qualsiasi software, se vuole sopravvivere, deve trovarsi in uno stato di continua evoluzione.

- Aumento della produttività. Il test automatico ha un costo iniziale superiore, perché richiede una codifica. Ma questo costo iniziale viene velocemente ammortizzato ogni volta che si richiama il test e che questo verifica il codice del programma in modo automatico in pochi secondi.

- Facilitazione dei concetti di refactoring e "program to change". Il test è la struttura portante di ogni modifica, perché è in grado di confermare se quanto funzionava prima della modifica funziona anche dopo.

- Predisposizione al cambiamento. Mai più dire: "Funziona bene così, quindi non lo cambio…". Il test è una comoda impalcatura su cui lo sviluppatore può salire per sentirsi sicuro durante le modifiche.

Il test è un elemento importantissimo nei metodi di sviluppo agile. Tra questi ce ne sono alcuni (eXtreme Programming, Test-Driven Development) che considerano il test come l'elemento fondamentale della programmazione. In TDD il test è l'elemento di partenza per sviluppare il programma. Il concetto è questo: prima scrivo il test, poi il codice che lo porta a un corretto funzionamento. In questi casi, oltre all'atteggiamento, diventa importante l'ambiente di sviluppo che si ha a disposizione. Aiuta molto il fatto di poter generare dal test gli *stubs*, cioè le strutture surrogate della parte di codice mancante.

Come scrivere test

Solitamente il test viene scritto partendo dal test di dettaglio a quello globale (*bottom-up*). Questo perché è più naturale scrivere il test per la funzionalità di un singolo metodo appena il metodo è stato implementato: si sa esattamente a cosa serve e cosa deve calcolare, quindi scrivere il test è senza dubbio più semplice.

È comodo scrivere una classe di test per ogni classe del programma, in modo che ci sia un parallelismo che diventa utile sia quando si tratta di andare a eseguire modifiche, sia, più importante, quando si tratta di andare a capire a cosa serve un certo metodo e quali risultati si ottengono con un certo input. Basta infatti andare nella corrispondente funzione di test e vedere come questa ha richiamato il metodo in questione.

Altra cosa importante da considerare quando si intende scrivere test è la possibilità di accumulare quelli vecchi e poterli richiamare regolarmente. A volte alcuni test possono sembrare inutili, perché molto semplici e scontati ma, se li si guarda da questa prospettiva, diventano importanti perché garantiscono il controllo della stabilità del programma attraverso le varie modifiche.

Naturalmente il controllo della correttezza deve essere automatico. Dobbiamo evitare che il test visualizzi in output una lunga serie di dati che siamo poi noi a dover controllare ogni volta.

Se ci sono dati da controllare, meglio codificare il controllo nel test stesso, in modo che sia in grado di dire semplicemente sì o no, giusto o sbagliato.

I test dovrebbero essere sufficientemente semplici da scrivere. Questo sia per evitare di dover scrivere altro test che verifichi il test (in generale non è necessario, perché la verifica è implicita nel rapporto bidirezionale che esiste tra codice produttivo e codice di test; in altre parole: il codice produttivo stesso funge da verifica del codice di test), sia per servire da documentazione sul modo in cui utilizzare il codice. Possibilmente il codice di test dovrebbe essere implementato separatamente, senza intaccare il codice del programma da sviluppare.

Come facilitare il test

Sulla base dei criteri appena elencati, vediamo un paio di esempi di come si potrebbe inserire codice che favorisca il test.

Metodi di visualizzazione

Si tratta di inserire, nelle classi che gestiscono le strutture di dati più rilevanti, uno o più metodi in grado di visualizzare la struttura. Nell'esempio seguente inseriamo il metodo `debug()` nella classe `Arc` che gestisce gli archi in un sistema di automi a stati finiti. Chiamare `debug()` su un arco significa visualizzare l'intero automa (lista di stati e archi) che si sviluppa a partire da quello.

```
public class Arc
{
  ...
  public void debug(){
    System.err.println("Content: " + getInfo());
    if (nextStates != null){
      nextStates.debug();
    }
  }
}
```

Il metodo presuppone che esista un corrispondente `debug()` anche per l'oggetto `nextStates`, che qui rappresenta gli stati successivi dell'automa. Questo modo di inserire metodi di visualizzazione nella classe è abbastanza semplice, ma può essere facilmente evitato se si ha a disposizione un buon debugger. Inoltre, se consideriamo i criteri su come scrivere i test, vediamo che presenta un chiaro svantaggio, cioè il fatto di inserire codice di test in quello del programma.

Ogni classe eseguibile

Un linguaggio come Java permette di definire un metodo `main()` in ogni classe, quindi anche in quelle che non sono da considerare classi "principali" del programma. Questo fatto può

essere sfruttato per rendere eseguibile ogni classe in modo indipendente dal programma completo. In ogni classe posso inserire alcune sequenze di codice che eseguono alcuni test specifici, come nell'esempio che segue.

```
public class Traverse
{
  ...

  public static void main(String[] args){
    Traverse traverse = new Traverse();
    ...
    traverse.execute(structure);
    ...
    if (traverse.isOk()){
      System.err.println("Execution ok");
    }else{
      System.err.println("Error detected");
    }
  }
}
```

Ogni classe può poi essere chiamata singolarmente. Si tratta di un modo molto semplice, ma che presenta due svantaggi. Il primo è di nuovo il fatto di avere inserito codice di test nel programma, anche se isolato in un unico metodo; il secondo è invece quello di avere sì tanti test, ma distribuiti nelle singole classi. Se non esiste la possibilità di eseguire i test tutti assieme con un'unica chiamata (con l'aiuto dell'ambiente di sviluppo o altro) si tratta di uno svantaggio.

Classi di supporto

Per classi di supporto si intendono classi aggiuntive che permettono di interagire con il resto del programma mettendo in evidenza alcuni aspetti. Sono utili soprattutto quando si lavora con un framework, argomento trattato nella parte sui design pattern. Un framework è una collezione di classi che gestiscono un certo workflow e che permettono allo sviluppatore di scrivere un programma unicamente implementando sottoclassi e riscrivendo metodi chiamati nel framework stesso. La difficoltà maggiore che si ha quando si lavora con un framework è quella di capirne il funzionamento e le relazioni tra le classi. Per verificare il funzionamento è perciò utile avere a disposizione classi specifiche che permettano di evidenziare alcuni aspetti del workflow esistente.

Tornando al framework per la realizzazione di automi a stati finiti, una classe utile è quella che permette di lavorare in modo interattivo con l'automa, specificando manualmente l'arco attraverso il quale ci si vuole spostare da uno stato a un altro, ottenendo come risultato il nuovo stato (o i nuovi stati). Senza entrare nel dettaglio dell'implementazione, si può pensare di mettere a disposizione una classe Shell, con la quale sia possibile gestire questo meccanismo interattivo.

```
public class InteractiveRun
{
  public static void main(String[] args){
    FiniteStateTool fst = new FiniteStateTool(args[0]);
    Shell shell = new Shell(fst)
    shell.interactiveAnalyze();
  }
}
```

Utilizzo di un tool esterno

Con tool esterno si intende un programma di utilità che ci faciliti la codifica dei test e ci permetta di raggrupparli in vari modi per poi richiamarli a piacimento, singolarmente o in gruppo. La utility ci deve permettere di scrivere i test all'esterno del codice del programma, e visualizzare eventuali problemi in modo diretto e mirato, senza necessità di spulciare lunghe liste di output.

Negli ultimi anni si è fatto strada nella comunità Java un tool chiamato JUnit [JUnit], che serve esattamente a questi scopi. Sul modello e sull'architettura di JUnit sono poi nati utensili simili per altri linguaggi. Uno di questi è clos-unit, di cui avremo modo di parlare in una nota.

In questo capitolo…

Abbiamo superato il primo capitolo riguardante il test, in particolar modo il test automatico. Dopo aver analizzato i possibili motivi che portano a non scrivere test, abbiamo elencato i maggiori benefici derivanti da un test sistematico e mostrato, a livello teorico, come deve essere scritto il test.

Nei prossimi capitoli, con l'ausilio di JUnit, vedremo il test da un punto di vista pratico.