



Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Ingegneria Informatica

Elaborato finale in **Ingegneria del software**

Strumenti per la generazione automatica di casi di test per programmi Java

Anno Accademico 2014/2015

Candidato:

Raffaele Amalfitano

matr. N46000945

*Desidero ringraziare la mia famiglia
per avermi sempre sostenuto e per
avermi permesso di intraprendere
questo percorso di studi.*

*Un ringraziamento speciale anche ad
Alessandra per avermi supportato e
sopportato per il conseguimento di
questo primo traguardo.*

Indice

Indice	III
Introduzione	4
Capitolo 1: Test Automation.....	6
1.1 Manual Testing vs Automated Testing	6
1.2 Generazione dei casi di test.....	8
1.3 Preparazione ed esecuzione dei test	9
1.4 Valutazione dell'esito e dell'efficacia dei casi di test.....	10
Capitolo 2: Testing Tools.....	12
2.1 Strumenti a supporto per la generazione automatica	12
2.2 Tobias	13
2.3 JPet	19
2.4 EvoSuite	24
2.4.1 Test Suite Optimization	25
2.5 JPet vs EvoSuite.....	29
Conclusioni	33
Bibliografia	34

Introduzione

Il testing è considerato una disciplina fondamentale nell'ambito di sviluppo di qualsiasi processo software. Rappresenta la parte di analisi dinamica di un'attività più vasta denominata *Verifica & Validazione (V&V)*, la quale punta a mostrare che il software è conforme alle sue specifiche e soddisfa le aspettative del cliente. Il processo di test del software ha due obiettivi distinti: dimostrare al cliente e allo sviluppatore che il software soddisfa i suoi requisiti, in questo caso parliamo di *Test di Convalida* e tale test ha successo se il software si comporta come ci si aspetta; scoprire errori o difetti nel software, in questo caso parliamo di *Test dei Difetti* e tale test ha successo se rileva uno o più malfunzionamenti e porta il programma a comportarsi in maniera scorretta. Sfortunatamente il testing non può dimostrare che il software è privo di difetti o che si comporterà come specificato in ogni circostanza. E' sempre possibile che un test trascurato possa scoprire ulteriori difetti, citando E. Dijkstra: *"Il testing può solo mostrare la presenza degli errori, non la loro assenza"*.

E' possibile misurare la bontà di un processo di testing in termini di efficacia (numero di malfunzionamenti trovati / numero di malfunzionamenti da trovare) ed efficienza (numero di test in grado di scoprire malfunzionamenti / numero di test totali). Quello che si cerca di fare è trovare il maggior numero di malfunzionamenti possibile con il minor numero possibile di casi di test. A seconda se poi si vuole un software più affidabile, o meno costoso, bisogna privilegiare rispettivamente l'efficacia o l'efficienza.

Tra le attività di un processo software il testing si configura come la parte più impegnativa e costosa. Questo principalmente a causa della sua enorme difficoltà, che diventa sempre più rilevante man mano che la complessità del software cresce e i requisiti di sicurezza aumentano.

Dal momento che l'esigenza di produrre software di qualità cresce sempre più velocemente, è bene valutare la possibilità di automatizzare il processo di testing utilizzando appositi strumenti che consentano di ridurre lo sforzo umano al fine di ottenere risultati quanto più possibili esenti da errori.

Nel corso dell'elaborato saranno presentati alcuni strumenti in grado di generare automaticamente casi di test per programmi Java, con una panoramica sulle tecniche di generazione automatica sulle quali si basano tali strumenti e sui vantaggi che si ottengono automatizzando il processo di test. Saranno poi analizzati e confrontati i risultati ottenuti.

Capitolo 1: Test Automation

1.1 Manual Testing vs Automated Testing

L’obiettivo di qualsiasi progetto di successo è quello di ridurre i costi e i tempi necessari per completare quanto richiesto, mantenendo alta la qualità del prodotto che si sta realizzando.

Il Testing copre un dominio enorme ma può essere suddiviso in due aree principali:

- Testing manuale
- Testing automatizzato

Entrambi offrono vantaggi e svantaggi, ma vale la pena conoscere la differenza che sussiste tra queste due aree e capire quando usare l’uno o l’altro per ottenere i risultati migliori. Il tipo di testing da eseguire dipende da vari fattori, tra i quali i requisiti di progetto, budget, scadenze, competenze.

-Nel testing manuale i casi di test sono eseguiti manualmente senza l’ausilio di strumenti o script. Si tratta di un testing molto flessibile con costi a breve termine contenuti. Il problema è che non sempre è possibile svolgere tutti i compiti manualmente. Può essere ripetitivo e noioso in alcuni casi. Come risultato, molti tester hanno difficoltà a rimanere impegnati in questo processo e gli errori possono verificarsi con più probabilità.

-Nel testing automatico si ricorre all’uso di tool, script e software. I casi di test vengono eseguiti in maniera rapida ed efficace. E’ possibile il riuso di test già esistenti, il che è un vantaggio quando bisogna fare test di regressione in quanto sono necessarie modifiche frequenti nel codice e le operazioni di regressione devono essere eseguite in modo tempestivo.

Manual Testing	Automated Testing
<ul style="list-style-type: none"> • Manual testing is not accurate at all times due to human error, hence it is less reliable. • Manual testing is time-consuming, taking up human resources. • Investment is required for human resources. • Manual testing is only practical when the test cases are run once or twice, and frequent repetition is not required. • Manual testing allows for human observation, which may be more useful if the goal is user-friendliness or improved customer experience. 	<ul style="list-style-type: none"> • Automated testing is more reliable, as it is performed by tools and/or scripts. • Automated testing is executed by software tools, so it is significantly faster than a manual approach. • Investment is required for testing tools. • Automated testing is a practical option when the test cases are run repeatedly over a long time period. • Automated testing does not entail human observation and cannot guarantee user-friendliness or positive customer experience.

Figura 1.1: Manual Testing vs Automated Testing [1].

A causa dell'elevato numero di casi di test necessari per eseguire un testing efficace l'operazione di progettazione manuale dei casi di test può essere molto onerosa. Si ricorre quindi a delle tecniche di generazione automatica che possono ridurre drasticamente i costi e i tempi legati alla fase di test design. L'automazione offre la possibilità di svolgere efficacemente la fase di verifica e validazione del software, permettendo di eseguire i test in maniera più rapida e in modo facilmente ripetibile.

Il testing automatico può quindi favorire, se pianificato e progettato a dovere, diversi fattori come il riuso del progetto, affidabilità e riduzione degli errori a lungo termine.

Nel processo di automazione del testing si individuano tre aree di intervento:

- Generazione dei casi di test
- Preparazione ed esecuzione dei test
- Valutazione dell'esito e dell'efficacia dei casi di test

1.2 Generazione dei casi di test

Diversi sono i modi attraverso cui è possibile generare automaticamente casi di test. In particolar modo si può partire dalla specifica dei requisiti, dall'analisi della documentazione di progetto, dall'analisi statica del codice sorgente oppure dall'osservazione di esecuzioni reali dell'applicazione. Per ottenere però un test efficace è necessario eliminare anche le eventuali ridondanze che si potrebbero presentare in un processo di automazione del testing.

Un tipico esempio di automazione per il testing black-box potrebbe essere quello rappresentato da uno scenario in cui l'utente interagisce con il sistema mediante una *Graphical User Interface* (GUI). Vengono installati strumenti che sono in grado di mantenere un log delle interazioni che avvengono tra gli utenti dell'applicazione e l'applicazione stessa da testare. A partire dai dati catturati (*fase di Capture*) vengono poi formalizzati dei casi di test in grado di replicare le interazioni registrate nella fase precedente (*fase di Replay*). Grazie a questi strumenti gli utenti tester possono memorizzare le azioni in modo interattivo e riprodurle nuovamente sull'interfaccia un numero illimitato di volte, confrontando i risultati effettivi rispetto a quelli previsti. Questo tipo di approccio può essere applicato a qualsiasi applicazione che dispone di un'interfaccia grafica utente e richiede inoltre poco sviluppo software.

Il problema che però si verifica con questo tipo di testing, denominato **User Session Testing**, è legato al numero di sessioni che bisogna raccogliere per poter avere un insieme significativo di casi di test. Questo significa che potrebbe essere necessario catturare le interazioni tra l'utente e la GUI per un tempo molto lungo, con relativo aumento di casi di test. Per ridurre sensibilmente il tempo necessario per la generazione di nuovi casi di test si possono adottare delle tecniche di testing in grado di generare nuovi casi di test a partire dalla combinazione di test già esistenti.

Una di queste tecniche è il **Testing Mutazionale** che consiste nell'applicazione di alcuni operatori di mutazione che vanno a modificare/incrociare i dati dei test case esistenti, in modo da ottenerne dei nuovi. Con tale tecnica, applicata maggiormente per il testing di

interfacce o di protocolli, è possibile ottenere test suite più piccole con una maggiore copertura e con uno sforzo minore.

Un'altra tecnica che consente di automatizzare i casi di test va sotto il nome di ***Random Testing***. Essa consiste nella generazione di sequenze casuali di input allo scopo di testare l'applicazione. L'efficienza di questa tecnica è molto bassa ma può condurre alla scoperta di malfunzionamenti che non vengono trovati con strategie di testing più “intelligenti”. I random test inoltre non hanno oracolo e possono essere utilizzati solo per cercare possibili situazioni di crash/eccezioni. Di solito, si utilizza il *Random Testing* come benchmark per verificare l'efficacia di altre soluzioni.

Un ulteriore approccio è fornito dal ***Combinatorial Testing***. Si tratta di una tecnica black-box che, come tale, non richiede l'analisi del codice sorgente. Diversi studi hanno dimostrato che può ridurre drasticamente il numero di test da generare pur rimanendo efficace nel rilevare difetti all'interno del software, inoltre è relativamente facile da applicare. Consiste in una prima fase di strutturazione manuale in cui un insieme di parametri, così come i loro possibili valori, devono essere identificati, e di una seconda fase automatizzata per produrre combinazioni. Il *Combinatorial Testing* consente di ridurre significativamente i costi e di aumentare la qualità del software. La norma principale su cui si basa è quella dell'interazione. Tale norma dispone che la maggior parte dei guasti è indotta da singoli fattori o dall'effetto congiunto di due fattori. Man mano che il numero dei fattori interagenti aumenta vengono indotti progressivamente meno guasti. Oltre tutto è possibile trovare difetti che difficilmente possono essere rilevati mediante metodi di selezione manuale.

Un'altra tecnica degna di nota prende il nome di ***Model Based Testing***. Essa consente la generazione di casi di test a partire da un modello (e.g *Workflow Diagram*, *Statechart Diagram*) che descrive alcuni aspetti funzionali del sistema.

1.3 Preparazione ed esecuzione dei test

Si tratta della parte più meccanica della testing automation. Il completo automatismo si può ottenere scrivendo il codice di test sotto forma di codice eseguibile.

Per l'esecuzione dei casi di test vengono utilizzati degli appositi framework nati nell'ambito dell'*eXtreme Programming* per automatizzare il testing di unità, ma generalizzabili anche alle problematiche di testing black-box. Questa scelta rappresenta la soluzione più efficace in quanto è possibile monitorare eventualmente anche lo stato interno del software. Il vincolo imposto da questi framework riguarda il linguaggio con cui deve essere scritto il software, che deve supportare la reflection.

Tra questi, i più conosciuti sono quelli che appartengono alla famiglia *xUnit*:

- *JUnit* (Java)
- *CppUnit* (C++)
- *csUnit* (C#)
- *NUnit* (.NET framework)
- *HttpUnit* e *HtmlUnit* (Web Application)

L'esecuzione automatica di casi di test comporta numerosi vantaggi quantificabili in tempo risparmiato (nell'esecuzione dei test), affidabilità, riuso.

1.4 Valutazione dell'esito e dell'efficacia dei casi di test

Per poter valutare automaticamente l'esito di un caso di test, esso dovrebbe esser stato oggettivamente definito e dovrebbe esser disponibile un metodo per la sua valutazione (e.g gli *assert* nel caso di un test con *JUnit*). In alcuni casi, l'esito di un test non ha bisogno di esser definito, oppure può esser definito automaticamente ricorrendo al ***Crash Testing*** (che consiste nel testare un software andando alla ricerca di eccezioni o errori a run-time che ne interrompano l'esecuzione) o al ***Regression Testing*** (che viene effettuato in seguito ad un intervento di manutenzione per verificare se la modifica effettuata su modulo software ha impattato il resto del sistema).

Per valutare invece l'efficacia dei casi di test generati si ricorre all'***Analisi Mutazionale***. Difetti artificiali (mutanti) vengono iniettati nel software e i casi di test, della test suite da valutare, vengono eseguiti sulla versione modificata del programma. Un mutante non rilevato mostra una debolezza all'interno della test suite e indica, nella maggior parte dei casi, che un nuovo caso di test dovrebbe esser aggiunto o che un caso di test esistente

necessita di miglioramenti. Migliorare i casi di test dopo l’analisi mutazionale significa che il tester deve riprogettarli, prendendo come riscontro ciò che ha ottenuto in seguito all’analisi mutazionale. Questo processo però richiede una profonda comprensione del codice sorgente ed è un compito non banale.

In conclusione, una test suite che riesca a rivelare il maggior numero possibile di mutanti è da considerarsi più promettente nella rivelazione di potenziali difetti nell’applicazione.

Capitolo 2: Testing Tools

2.1 Strumenti a supporto per la generazione automatica

Tra i diversi tool disponibili per la generazione automatica di casi di test per programmi Java per il testing d'unità, sono stati presi in esame i seguenti:

- *Tobias*
- *JPet*
- *EvoSuite*

Come caso di studio verrà presa in considerazione una semplice applicazione che simula il comportamento di un distributore automatico di cioccolata. L'applicazione è costituita da una classe *VendingMachine* [4] e da tre metodi:

- *addChoc(String choc)*: ricarica il distributore con una cioccolata (capacità massima del distributore: 10 cioccolate).
- *coin(int coin)*: prende in ingresso una moneta. Monete accettate: 10, 25, 100; le monete vengono accettate fino a quando il credito è inferiore a 90.
- *getChoc(StringBuffer choc)*: ritorna una cioccolata se il credito è pari a 90 o più e se il distributore non è vuoto.

Il metodo preso in esame per la generazione dei casi di test è il metodo *coin()*, mentre per testare il funzionamento dei tool è stato utilizzato l'ambiente di sviluppo *Eclipse* ed, in base alla disponibilità offerta dagli strumenti, i test sono stati effettuati su *Mac Os* e su una distribuzione *GNU/Linux*.

2.2 Tobias

Tobias è un generatore di test combinatori che sviluppa un modello di prova fornito dal tester ed esegue in maniera esaustiva tutte le combinazioni dei parametri e dei metodi di ingresso. Lo strumento è disponibile attraverso un’interfaccia web e per poterlo utilizzare l’utente deve fornire il proprio indirizzo e-mail, selezionare il formato dei dati di input e di output e caricare il file. Il sito web elabora il file e gli output generati vengono raggruppati in un file zip, il cui url viene inviato all’e-mail dell’utente. Il file zip viene mantenuto sul sito web per un periodo limitato di tempo.

Il file di input può essere espresso in due formati: TSLT (Test Schema Language for Tobias) e INTOB. TSLT è un linguaggio testuale, compatto e comprende gran parte della sintassi Java. Si concentra sui costrutti più frequentemente utilizzati da *Tobias*. INTOB include costrutti aggiuntivi come i filtri ed è più complesso. In Figura 2.1 è riportata la schermata iniziale di *Tobias*.

The screenshot shows the Tobias online interface. At the top, a green header bar reads "Try Tobias !". Below it, a message states: "The following form allows you to upload a file and get it processed by Tobias." A bulleted list of requirements follows:

- Your file must conform to one of the Tobias input formats [dtd \(stored here\)](#).
- It must be less than 150K bytes.
- It will be kept on our server for research purposes.

The main area is titled "Upload your file". It includes an "Email:" field with a placeholder "", a "Select the input file on your computer : [Scegli file](#)" button with the text "nessuno selezionato", and a "Conditions of Use" checkbox labeled "I have read and I accept the [Conditions of Use](#)".

A table below these fields has three columns: "Input Format" (radio buttons for TSLT and INTOB, with TSLT selected), "Output Format" (radio buttons for JUnit, JUnit 4, JML, and OTB, with JUnit 4 selected), and "Captcha" (a reCAPTCHA image showing a red car on a road with the number 502, a text input field "Digita il testo", and two buttons: a blue arrow and a white arrow).

At the bottom of the form is an "Upload" button.

Figura 2.1: Schermata online Tobias.

Una volta scelto il formato, il file d’ingresso viene elaborato da *Tobias* che genera una o più suite di test memorizzati in file XML. La versione online di Tobias assicura agli utenti di utilizzare la versione più recente dello strumento, tuttavia vi sono delle limitazioni motivate da ragioni di sicurezza. Una di queste riguarda la dimensione dei file di ingresso e di uscita che è limitata per evitare che il disco sul server si saturi.

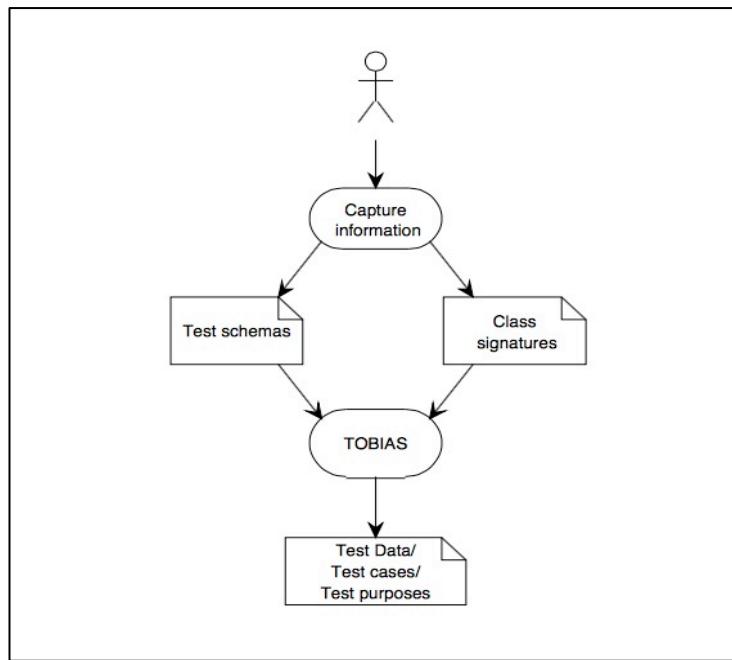


Figura 2.2: Funzionamento Tobias [2].

Il principio di funzionamento su cui si basa *Tobias* è illustrato in Figura 2.2. Come possiamo vedere *Tobias* riceve due input, la firma delle classi che si vuole testare e un test schema, a partire dai quali genera una sequenza di dati.

Lo strumento consente di produrre una grande quantità di casi di test a partire da poche linee di descrizione combinatoria. L'approccio sistematico che usa è tuttavia anche uno dei suoi principali punti deboli perché può condurre alla cosiddetta *combinatorial explosion*. E' infatti facile scrivere un test schema che si traduce in un milione di casi di test. Test suite di grandi dimensioni sono di solito incompatibili con i compilatori standard, inoltre la loro esecuzione può richiedere molto tempo e risorse. Pertanto è importante padroneggiare l'esplosione combinatoria. A tal proposito *Tobias* propone diverse tecniche come meccanismi di filtri e selezione che aiutano a ridurre la dimensione della test suite risultante. I filtri forniscono un modo semplice per padroneggiare la dimensione di una test suite e non sono altro che delle proprietà espresse dal tester sotto forma di funzioni booleane che devono essere soddisfatte dai casi di test della test suite.

I selettori rappresentano un'alternativa ai filtri. Mentre un filtro prende un singolo caso di test come argomento, un selettore prende l'intera test suite e ne restituisce un

sottoinsieme che soddisfi un certo criterio. Semplici selettori usano tecniche di campionamento casuale, mentre quelli complessi fanno uso del codice dei casi di test.

Esempio di applicazione:

Come primo file di input consideriamo quello in Figura 2.3. Esso è espresso nel linguaggio TSLT ed è memorizzato in un file di testo. Tale file descrive un gruppo nominato *Add2Coins* che è etichettato come *[us=true]*. Ciò significa che questo gruppo sarà “spiegato” da *Tobias* all’interno di una test suite. La prima cosa che viene fatta è una chiamata al costruttore della classe *VendingMachine*, seguita da una chiamata al metodo *coin()*. In particolar modo la chiamata viene effettuata due volte ({2}) e la lista dei possibili valori per il parametro di ingresso del metodo *coin()* è quella tra parentesi quadre.

```

1 header { package = #test#,
2           import = #vendingMachine.*# }
3
4
5 // First a group which adds two coins
6 group Add2Coins[us=true] {
7   VendingMachine vm = new VendingMachine();
8   vm.coin([10,20,25,50,100,200]) {2};
9 }
10

```

Figura 2.3: Add2Coins.txt.

Dopo aver eseguito la procedura di upload tramite il sito web, la schermata risultante è quella mostrata in Figura 2.4 nel caso di file corretto. Le informazioni fornite riguardano il numero dei casi di test generati (36 in questo caso), il tempo impiegato per processare il file di output, la dimensione del file generato etc.

You'll receive a mail with the link to the result file. Remember that the file will be in our sever only one day

```

TSLT V2 updated April 30th 2012
-----
File was processed and output generated in : 0.457597 seconds
-----
Tobias V2 updated October 24th 2012
Licensed to : Laboratoire d'Informatique de Grenoble, Equipe Vasco
Expires     : dimanche 17 ao?t 292278994
-----
Generation of the otb file for group Add2Coins
unfolding 36.0 testcases
->Writing 36 testcases
->Output file TS_Add2Coins.otb
->File size 26kbytes
->Execution time : 0.156747 sec
--
```

Figura 2.4: Schermata di output per file Add2Coins.txt.

Una volta estratto il contenuto del file inviato tramite email, si ottiene un file .java come quello mostrato in Figura 2.5, in cui sono rappresentati solo alcuni dei 36 casi di test generati. Come si può notare ogni caso di test include due chiamate al metodo *coin()* che ha come parametro d'ingresso uno dei sei valori tra quelli specificati in precedenza (36 possibili combinazioni). Successivamente bisogna creare un package all'interno dello stesso progetto in cui è contenuta la classe *VendingMachine* (nel nostro caso il nome del package sarà *test*) e copiare all'interno il file .java.

```

44@  @Test
45  public void testSequence_6()
46  { VendingMachine vm = new VendingMachine() ;
47    vm.coin(10) ;
48    vm.coin(200) ;
49  }
50
51@  @Test
52  public void testSequence_7()
53  { VendingMachine vm = new VendingMachine() ;
54    vm.coin(20) ;
55    vm.coin(10) ;
56  }
57
58@  @Test
59  public void testSequence_8()
60  { VendingMachine vm = new VendingMachine() ;
61    vm.coin(20) ;
62    vm.coin(20) ;
63  }
64
65@  @Test
66  public void testSequence_9()
67  { VendingMachine vm = new VendingMachine() ;
68    vm.coin(20) ;
69    vm.coin(25) ;
70  }

```

Figura 2.5: TS_Add2Coins.java.

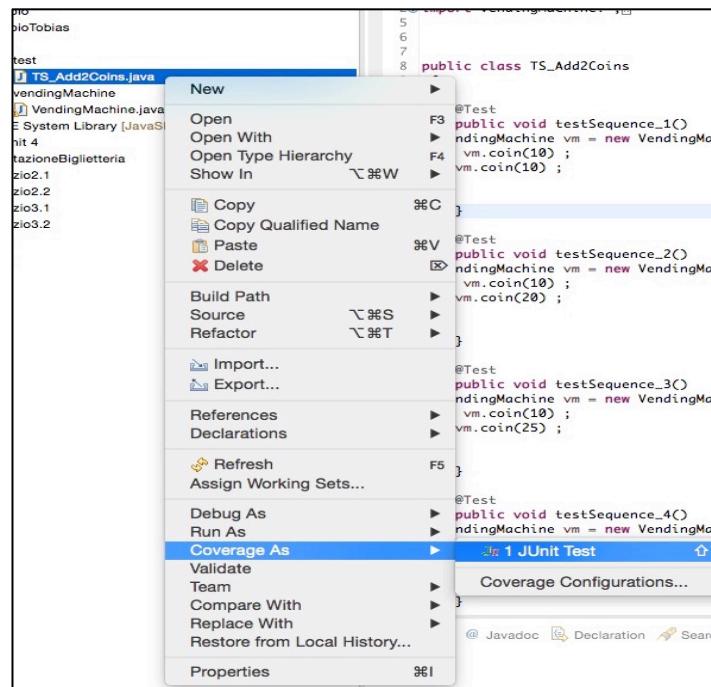


Figura 2.6: Copertura con EclEmma.

Per poter valutare la copertura fornita dai casi di test generati da *Tobias* utilizziamo *EclEmma*. Si tratta di un plug-in per *Eclipse* grazie al quale è possibile visualizzare in maniera grafica, tramite evidenziazione delle linee di codice in colori differenti, la copertura durante l'esecuzione del programma, vedere in dettaglio le statistiche ed esplorarle. Una volta installato viene aggiunta una nuova funzionalità nel menù di esecuzione di *Eclipse*, ovvero l'esecuzione in modalità copertura. Utilizzando questa funzione viene aperta una nuova scheda nel pannello inferiore dell'interfaccia di *Eclipse* che riporta le statistiche di copertura in termini percentuali. Per effettuare la copertura con *EclEmma* bisogna cliccare col tasto destro del mouse sul file d'interesse e selezionare la voce “*Coverage As*” e successivamente “*JUnit Test*”, come mostrato in Figura 2.6.

Quello che si ottiene con i casi di test generati in precedenza è la copertura totale del metodo *coin()*, come si vede dalla Figura 2.7, con 108 linee di codice generate a partire da 6 linee di codice sorgente.

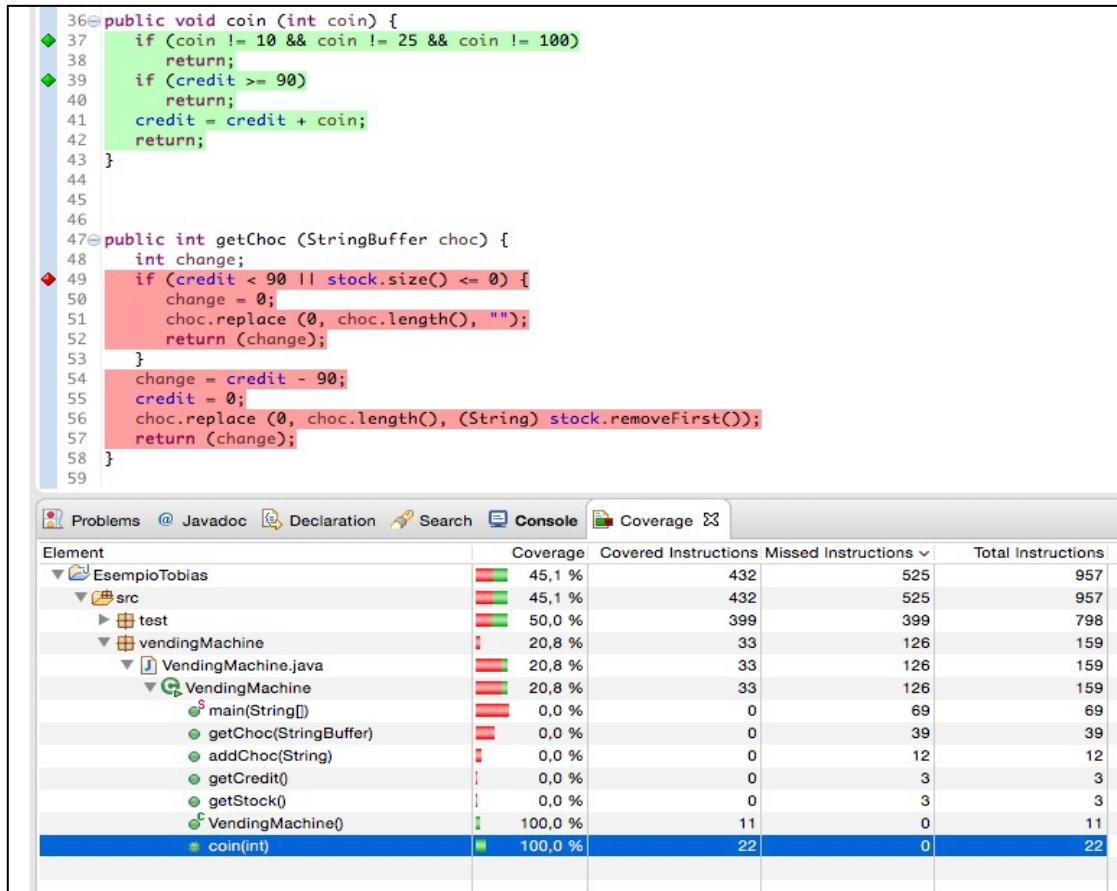


Figura 2.7: Analisi di copertura per file Add2Coins.txt.

Una versione strutturata del modello analizzato in precedenza, che fa uso anche di annotazioni (*NominalValues* e *RobustnessValues*), è quella riportata in Figura 2.8.

```

1 header { package = #test#,
2           import = #vendingMachine.*# }
3
4 // Definition of groups which add coins to the vending machine
5
6 // Then a structured version of the group which adds two coins.
7
8 groupheader {testCategory = #NominalValues#}
9 group validCoins { values = [10,25,100]; }
10
11 groupheader {testCategory = #RobustnessValues#}
12 group invalidCoins { values = [20,50,200]; }
13
14 group Coins { values = [@validCoins, @invalidCoins]; }
15
16 // First a group which adds two coins
17 group AddTwoCoins [us=true] {
18     VendingMachine vm = new VendingMachine();
19     vm.coin(@Coins){2};
20 }
```

Figura 2.8: Add2CoinsStrutturato.txt.

Anche in questo caso il file è scritto in linguaggio TSLT ed è memorizzato in un file di testo. A differenza del file presente in Figura 2.3 viene effettuata prima la definizione dei gruppi che aggiungono monete al distributore, e successivamente una versione strutturata del gruppo che aggiunge due monete. Il file .java prodotto da *Tobias* contiene 36 casi di test generati come in precedenza, ma la struttura è come quella rappresentata in Figura 2.9 (in cui viene mostrato solo uno dei casi di test generati).

```

49
50 @Test
51 public void testSequence_4()
52 {/*
53  * testCategory == NominalValues
54  */
55  /* testCategory == RobustnessValues
56  */
57  VendingMachine vm = new VendingMachine() ;
58  vm.coin(10) ;
59  vm.coin(20) ;
60
61
62 }
```

Figura 2.9: TS_AddTwoCoins.java.

Come si può notare ogni test case è contrassegnato con i tag dei gruppi che hanno contribuito a generarlo. In termini di copertura il risultato è analogo a quello visto in precedenza ma grazie all'uso delle annotazioni si riesce ad ottenere una tracciabilità tra le definizioni di gruppo e i casi di test.

E' inoltre possibile testare il funzionamento dell'intero programma facendo in modo da garantire la copertura di tutta la classe.

2.3 JPet

JPet è un generatore automatico di casi di test (TCG) che utilizza un approccio white-box e che può essere utilizzato durante lo sviluppo di applicazioni software. È integrato nell'ambiente di sviluppo *Eclipse* e basa il suo funzionamento su *PET*, un TCG che effettua l'esecuzione simbolica del bytecode associato ad un programma Java. Questo vuol dire che i valori di ingresso sono costituiti da simboli rappresentanti ognuno l'insieme dei valori che la variabile associata potrebbe assumere all'inizio di una normale esecuzione numerica. Oltre al bytecode, riceve in ingresso anche un determinato criterio di copertura per generare automaticamente casi di test. Tali criteri non sono altro che delle euristiche che cercano di stimare quanto bene un programma è esercitato da una test suite. Tra i possibili criteri rientrano la copertura delle istruzioni (*statement coverage*), che richiede che ciascuna istruzione del codice venga esercitata e la copertura dei cammini (*path coverage*), che richiede la copertura di ogni possibile cammino di esecuzione. Il numero di cammini eseguibili in un modulo può però essere potenzialmente infinito, non è quindi possibile esercitarli tutti. Tipicamente quando si parla di copertura dei cammini in presenza di cicli si è soliti porre un limite superiore al numero di iterazioni che vengono svolte. A tal proposito si parla di *copertura loop-k*, la quale richiede di attraversare tutti i cammini presenti nel programma tranne quelli con più di k iterazioni su ogni ciclo. *PET* può essere utilizzato sia da linea di comando sia da interfaccia web, ma in entrambi casi le funzionalità fornite non sono adeguate per testare applicazioni Java durante lo sviluppo del software.

JPet estende le funzionalità già presenti nel suo predecessore aggiungendone delle nuove. Effettua il *reverse engineering* dei casi di test prendendo le informazioni raccolte a livello di bytecode e mostrandole in maniera comprensibile a livello di codice sorgente Java.

Le principali estensioni introdotte da *JPet* sono le seguenti:

- Incorpora un visualizzatore di casi di test (*test case viewer*) che consente di visualizzare le informazioni calcolate nei casi di test. In particolar modo è possibile navigare graficamente attraverso gli input e gli output. Tale visualizzatore è

composto da due aree principali: la prima area è costituita da due tabelle all'interno delle quali sono rappresentati gli input/output e le informazioni relative all'oggetto selezionato (compresi tutti i suoi campi); nella seconda area vi è una rappresentazione ad albero dell'oggetto selezionato e di tutti gli oggetti raggiungibili da esso. Ogni nodo dell'albero può essere espanso e contratto per visualizzare e nascondere rispettivamente i suoi figli. Per accedere a tale funzionalità bisogna cliccare col tasto destro del mouse su uno dei casi di test e selezionare “*Show Test Case*”.

- Consente la visualizzazione della traccia dei casi di test, ovvero la sequenza di istruzioni che il test case esercita. Le informazioni possono essere visualizzate in due modi: la prima modalità consiste nell'evidenziare tutte le linee (e quindi tutte le istruzioni) che vengono eseguite. E' possibile inoltre distinguere tra linee normali di esecuzione e linee di eccezione evidenziandole con due colori diversi (rispettivamente verde e rosso); la seconda modalità permette di seguire la traccia del codice sorgente step-by-step tramite un debugger implementato utilizzando le opzioni di debug di *Eclipse*. E' possibile attivare queste funzionalità cliccando col tasto destro del mouse su uno dei casi di test e selezionando rispettivamente “*Show Trace*” oppure “*Show Trace Debug*”.
- Consente l'analisi delle precondizioni dei metodi scritte in JML (*Java Modelling Language*). Uno dei noti problemi che riguardano l'esecuzione simbolica del bytecode di un programma, è la grande quantità di cammini che devono essere considerati. Questo da un lato pone problemi di scalabilità dovuti ai requisiti che la memoria deve possedere per poter immagazzinare questa grande quantità di informazioni, dall' altro complica l'uso dei casi di test che devono essere considerati per testare il funzionamento di un programma software. Un modo per alleviare questa situazione è tramite l'uso di precondizioni che consentono allo sviluppatore di specificare condizioni sui parametri di ingresso in modo da prevenire la generazione di casi di test inutili. Le precondizioni vengono scritte usando JML e rappresentano sostanzialmente dei vincoli a cui sottostare. Nel caso

in cui sono presenti cammini che non rispettano tali vincoli, questi vengono ignorati e non vengono generati i casi di test ad essi associati.

Esempio di applicazione:

Una volta aperto un file sorgente Java (o il bytecode ad esso associato), l'utente può selezionare in *Eclipse* i metodi per i quali generare casi di test. Considerando la classe *VendingMachine* vista in precedenza, è possibile testare uno dei suoi metodi per osservare il funzionamento del tool.

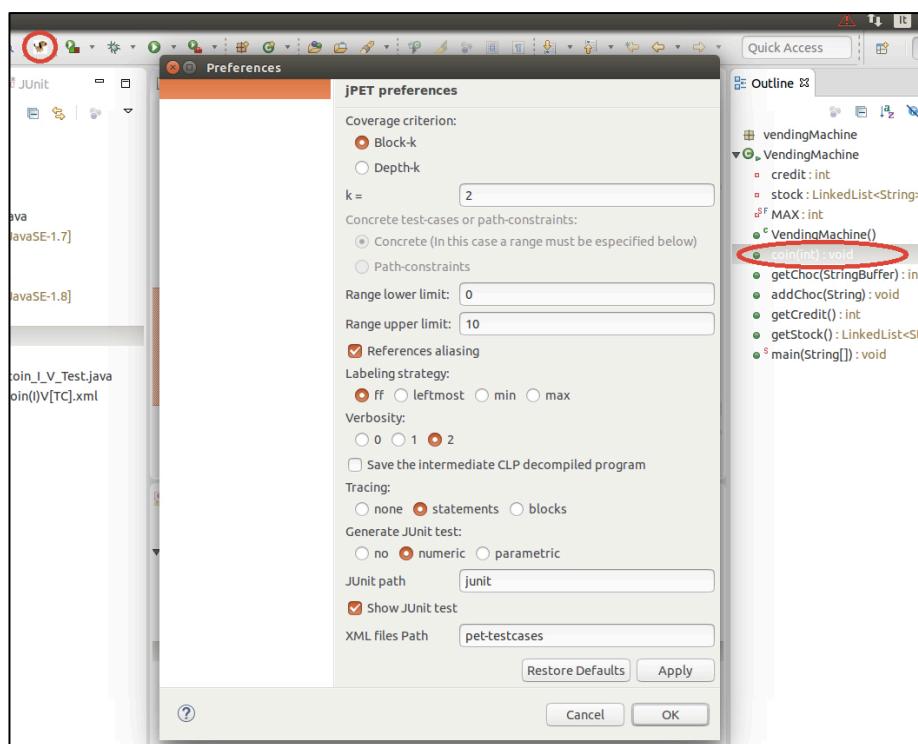


Figura 2.10: Schermata configurazione JPet.

In Figura 2.10 è mostrata la schermata che appare dopo aver selezionato il metodo *coin()* ed aver lanciato *JPet* premendo su “*Execute Pet*”. Successivamente è possibile settare una serie di preferenze, come la scelta del criterio di copertura. Attualmente *JPet* fornisce due criteri di copertura: *block-k* (un adattamento di *loop-k* nel caso del bytecode) e *depth-k* (che limita il numero di istruzioni da eseguire ad una certa soglia k). Viene aggiunta una cartella *pet-testcases* al progetto ed i casi di test ottenuti sono poi visualizzati nella *JPet view* secondo una struttura ad albero organizzati in nome_package, nome_classe, nome_mетодо come mostrato in Figura 2.11.

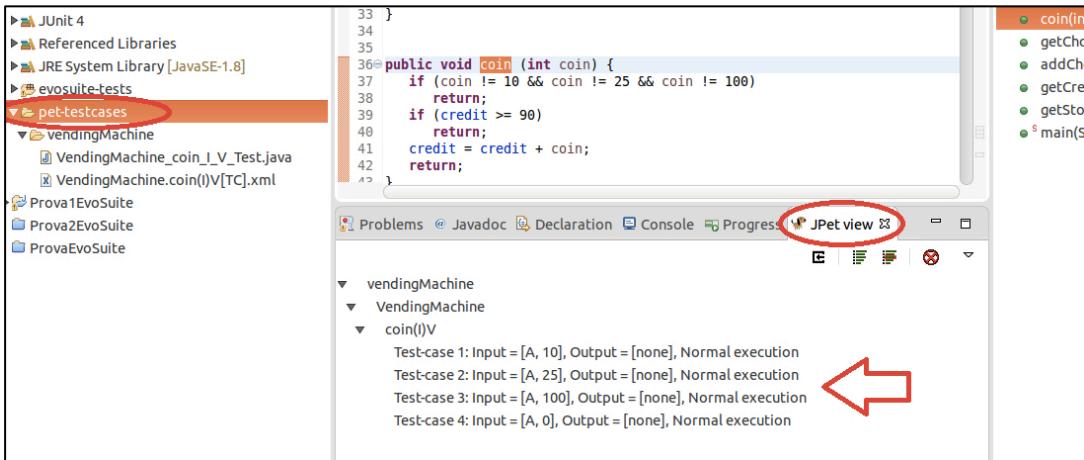


Figura 2.11: JPet view.

Come si può notare, i casi di test generati per il metodo *coin()* sono 4 e il comportamento rilevato da ognuno di essi non presenta anomalie. Oltre a poter accedere alle informazioni relative ad ogni singolo caso di test, è possibile anche visualizzare la sequenza di istruzioni che esso copre, come mostrato rispettivamente in Figura 2.12 e 2.13.

Come già detto in precedenza, *JPet* è in grado anche analizzare le eventuali precondizioni JML di cui un metodo è dotato, restringendo così il campo dei possibili casi di test generabili a quelli che rispettano i vincoli imposti. Un esempio di precondizione JML è mostrato in Figura 2.14 in cui è stato imposto che il parametro d'ingresso del metodo *coin()* non può assumere valore 10. Di conseguenza i casi di test generati non riguarderanno tale parametro e, da 4 test generati in precedenza si è passati a 3 con l'uso di precondizioni.

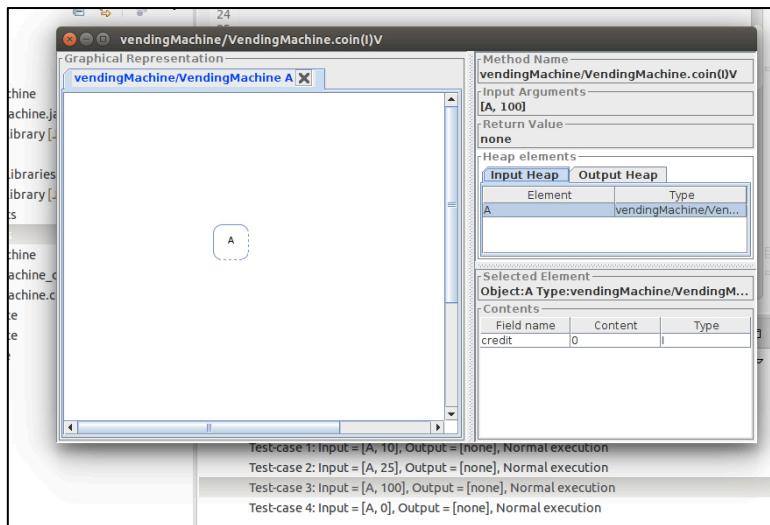


Figura 2.12: Test case viewer.

```

37 public void coin (int coin) {
38     if (coin != 10 && coin != 25 && coin != 100)
39         return;
40     if (credit >= 90)
41         return;
42     credit = credit + coin;
43     return;
44 }
45
46
47
48 public int getChoc (StringBuffer choc) {
49     int change;
50     if (credit < 90 || stock.size() <= 0) {
51         change = 0;
52         choc.replace (0, choc.length(), "");
53     }

```

Problems @ Javadoc Declaration Console JPet view

- vendingMachine
 - VendingMachine
 - coin()V

Test-case 1: Input = [A, 10], Output = [none], Normal execution
 Test-case 2: Input = [A, 25], Output = [none], Normal execution
 Test-case 3: Input = [A, 100], Output = [none], Normal execution
 Test-case 4: Input = [A, 0], Output = [none], Normal execution

Problems @ Javadoc Declaration Console JPet view

- vendingMachine
 - VendingMachine
 - coin()V

Test-case 1: Input = [A, 10], Output = [none], Normal execution
 Test-case 2: Input = [A, 25], Output = [none], Normal execution
 Test-case 3: Input = [A, 100], Output = [none], Normal execution
 Test-case 4: Input = [A, 0], Output = [none], Normal execution

Figura 2.13: Traccia del Test-case 1 (a sinistra) e del Test-case 2 (a destra).

```

35
36 /*@requires coin != 100 */
37
38 public void coin (int coin) {
39     if (coin != 10 && coin != 25 && coin != 100)
40         return;
41     if (credit >= 90)
42         return;
43     credit = credit + coin;
44     return;
45 }
46
47

```

Problems @ Javadoc Declaration Console JPet view

- vendingMachine
 - VendingMachine
 - coin()V

Test-case 1: Input = [A, 10], Output = [none], Normal execution
 Test-case 2: Input = [A, 25], Output = [none], Normal execution
 Test-case 3: Input = [A, 0], Output = [none], Normal execution

Figura 2.14: Uso di JPet con precondizioni.

Le informazioni circa la copertura fornita dai casi di test sono riportate in Figura 2.15 o, equivalentemente, in Figura 2.16 mostrate nella Console di Eclipse.

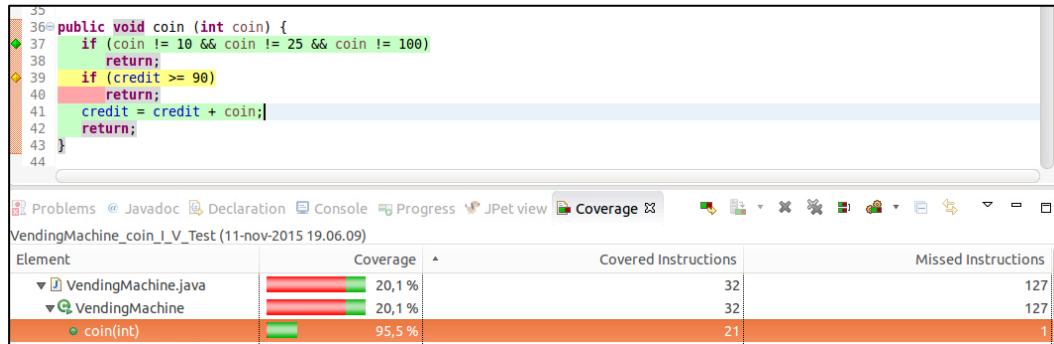


Figura 2.15: Copertura del metodo coin() tramite EclEmma.



Figura 2.16: Copertura del metodo coin() tramite JPet.

Il vantaggio principale di JPet consiste nell’interpretazione di programmi Java a livello di bytecode. In questo modo lo si può utilizzare per programmi Java per cui non è disponibile il codice sorgente, che nella pratica non è sempre recuperabile. Questo potrebbe essere molto utile dal punto di vista della *Reverse Engineering* in quanto è possibile la comprensione del programma osservando il comportamento di una serie di esecuzioni significative.

Attualmente il plug-in funziona solo per sistemi *Linux*, ma gli sviluppatori stanno lavorando per avere versioni disponibili per altri sistemi operativi.

2.4 EvoSuite

EvoSuite è uno strumento che produce automaticamente casi di test per classi Java, con l’obiettivo di massimizzare la copertura del codice. Come input richiede solo il bytecode della classe sotto test e le sue dipendenze. Questo viene fornito automaticamente quando lo si usa come plug-in in *Eclipse*, mentre se lo si usa da riga di comando bisogna impostare il percorso della classe e specificare il nome della classe di destinazione come parametro. Utilizza un approccio basato su ricerca in cui un algoritmo genetico (GA) ottimizza intere test suite sulla base di un criterio di copertura scelto (che rappresenta un insieme finito di obiettivi di copertura). Il vantaggio di utilizzare questo approccio, noto come *whole test suite generation*, è che la soluzione non viene negativamente influenzata dall’infattibilità di un dato obiettivo di copertura e non dipende dall’ordine con cui vengono considerati gli obiettivi stessi. *EvoSuite* inoltre suggerisce possibili oracoli aggiungendo una serie di asserzioni che sinteticamente riassumono il comportamento attuale. Siccome *EvoSuite* richiede solo il bytecode come input e tale bytecode spesso non comprende specifiche formali, le asserzioni prodotte rifletteranno il comportamento osservato piuttosto che quello previsto. Di conseguenza i casi di test sono intesi come test di regressione o servono

come punto di partenza per un tester umano, che può rivedere manualmente le asserzioni per determinare fallimenti. Prima di presentare il risultato all'utente e al fine di rendere più facili da capire e da utilizzare i casi di test prodotti, le test suite vengono minimizzate utilizzando un algoritmo di minimizzazione che tenta di rimuovere tutte le istruzioni che non contribuiscono alla copertura; in questo modo viene ridotto sia il numero dei casi di test che la loro lunghezza.

2.4.1 Test Suite Optimization

Per evolvere le test suite che ottimizzano il criterio di copertura scelto si utilizza un algoritmo di ricerca, chiamato algoritmo genetico che viene applicato su una popolazione di test suite. Gli algoritmi genetici si qualificano come tecniche di ricerca euristiche ed affondano le proprie radici nella teoria dell'evoluzione e si basano sulla manipolazione di un insieme di potenziali soluzioni per un problema di ottimizzazione o di ricerca. Il funzionamento di un GA è il seguente: una popolazione iniziale (tipicamente generata in maniera casuale) di soluzioni candidate viene fatta evolvere utilizzando operatori di ricerca che ricalcano i processi naturali. La riproduzione dei genitori è basata sulla loro idoneità e viene effettuata con una certa probabilità utilizzando operatori quali *crossover* e *mutation*. Ad ogni iterazione dell'algoritmo, l'idoneità della popolazione migliora fino a quando non viene raggiunta una soluzione ottimale oppure una data condizione di stop non si è verificata (tempo massimo imposto).

In un testing evolutivo, la popolazione è associata ai casi di test e l'idoneità, rappresentata mediante una *funzione di fitness*, stima quanto una soluzione candidata è vicina a soddisfare un certo obiettivo di copertura. Dire che la popolazione viene generata casualmente significa dire che un certo numero di valori di ingresso viene generato casualmente.

Il funzionamento degli operatori di *crossover* e *mutation*, che usano tecniche anch'esse mutuate dalla natura, è riportato in Figura 2.17.

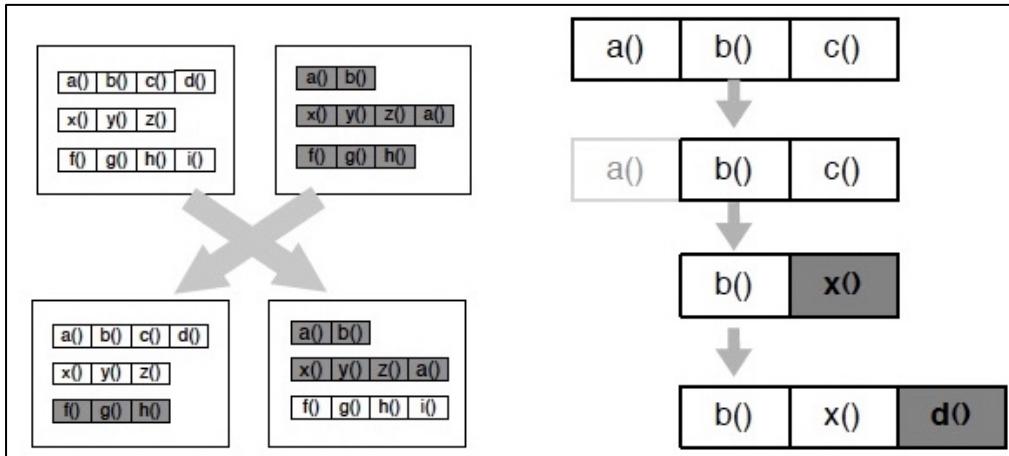


Figura 2.17: Crossover (a sinistra) e Mutation (a destra) [3].

- L’operatore di *crossover* opera a livello delle singole test suite. Tale operatore genera due figli O1 e O2 a partire da due genitori P1 e P2 (che rappresentano due test suite). Un valore casuale α viene scelto tra 0 ed 1. Il primo figlio O1 conterrà i primi $\alpha|P1|$ casi di test del primo genitore, seguiti dagli ultimi $(1-\alpha)|P2|$ casi di test del secondo genitore. Analogamente il secondo figlio O2 conterrà i primi $\alpha|P2|$ casi di test del secondo genitore, seguiti dagli ultimi $(1-\alpha)|P1|$ casi di test del primo genitore. Poiché i casi di test sono indipendenti tra loro, l’operatore di crossover produrrà sempre test suite figlie valide. Inoltre tale operatore fa decrescere la differenza del numero di casi di test tra le test suite, ovvero la differenza tra i casi di test che compongono le test suite figlie sarà minore della differenza tra i di test che compongono le test suite genitrici: $(|O1|-|O2|) \leq (|P1|-|P2|)$. Nessun figlio avrà più casi di test del più “grande” dei suoi genitori. Tuttavia è possibile che il numero dei casi di test in un figlio possa aumentare.

- L’operatore di *mutazione* è più complicato rispetto al precedente, questo perché esso opera sia a livello di test suite, sia a livello di casi test. Quando una test suite T viene mutata, ogni suo caso di test viene mutato con probabilità $1/|T|$. Quindi, in media, solo un caso di test viene mutato. Successivamente un numero casuale di casi di test viene aggiunto alla test suite T. Se un caso di test viene mutato, tre tipi di operazioni vengono eseguite in sequenza, come si può vedere dalla Figura 2.17: rimozione, modifica, inserimento. Se dopo l’applicazione di operatori di mutazione un caso di test rimane senza

statement, nel caso in cui questi sono stati eliminati, allora il caso di test viene rimosso dalla test suite T.

Per valutare la fitness di una test suite, è necessario eseguire tutti i suoi casi di test e raccogliere le informazioni di ramo (nel caso in cui il criterio scelto riguardi la copertura dei rami). Durante la ricerca solo un caso di test in media viene modificato in una test suite per ogni generazione. Ciò significa che rieseguire tutti i casi di test non è necessario in quanto le informazioni di copertura possono esser ricavate dall'esecuzione precedente.

Ottimizzare quindi intere test suite in relazione ad un dato criterio di copertura è meglio rispetto all'approccio tradizionale che tiene conto di un singolo obiettivo alla volta.

Esempio di applicazione:

Per generare casi di test con *EvoSuite* bisogna cliccare col tasto destro del mouse sulla classe da testare e selezionare “*Generate tests with EvoSuite*”, come mostrato in Figura 2.18. Prima di lanciare l'esecuzione del tool è possibile impostare una serie di preferenze. Per accedere alla schermata di configurazione di *EvoSuite* bisogna cliccare sulle proprietà del progetto che contiene la classe da testare e far riferimento alla sezione inerente al tool.

Le possibilità di configurazione sono mostrate in Figura 2.19.

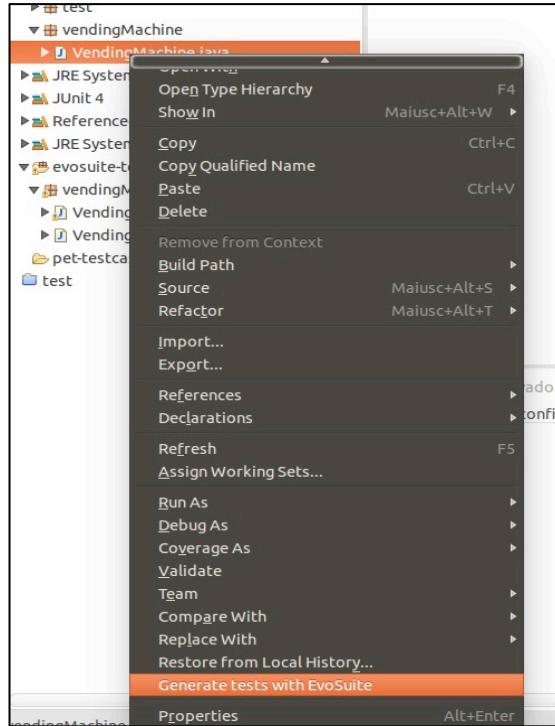


Figura 2.18: Avvio di *EvoSuite*.

Completata la configurazione si può lanciare l'esecuzione del tool. Terminata questa, si ottengono due file .java con suffisso pari a quello impostato durante la configurazione, che vengono direttamente aggiunti al progetto.

I casi di test generati sono in formato *JUnit* ed è quindi possibile rieseguirli automaticamente. Per quanto riguarda la copertura fornita dallo strumento, il risultato è mostrato in Figura 2.20 e, da come si può vedere, il metodo *coin()* risulta coperto completamente.

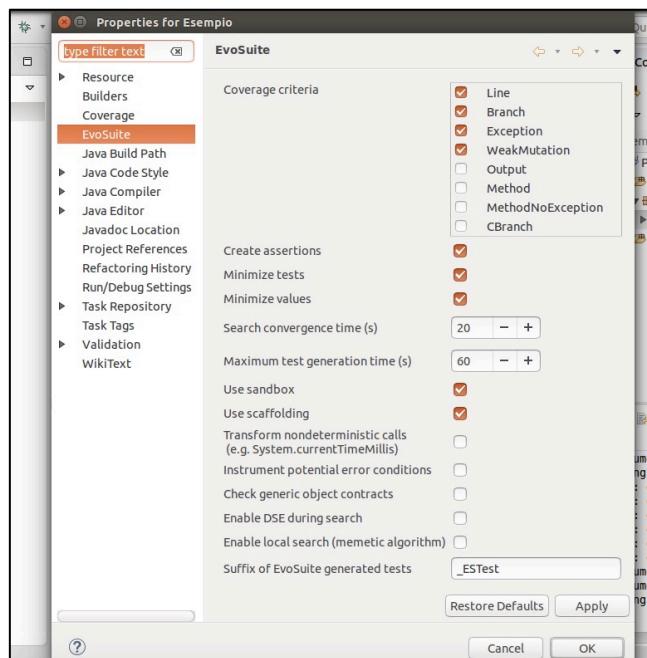


Figura 2.19: Finestra di configurazione di EvoSuite.

Nell'esempio riportato, è stata utilizzata la configurazione di default dello strumento, come quella riportata in Figura 2.19.

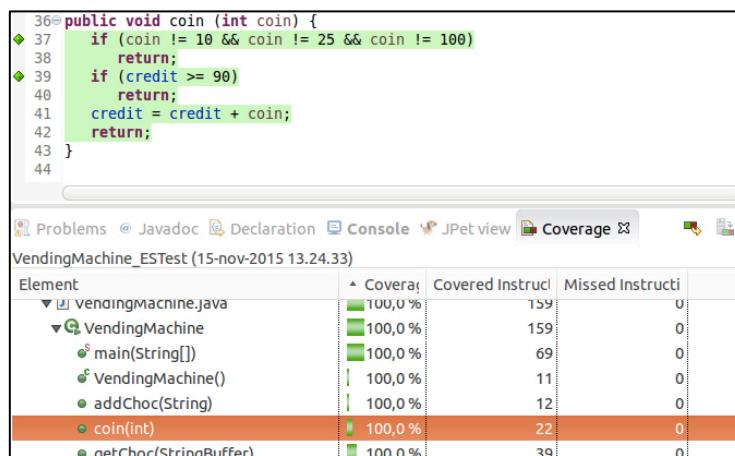


Figura 2.20: Analisi di copertura EvoSuite.

2.5 JPet vs EvoSuite

Per quanto riguarda l’analisi condotta in precedenza, tutti e tre i tool sono stati in grado di effettuare una copertura pari al 100% o quasi del metodo *coin()*. Bisogna però tenere in considerazione una serie di fattori.

Tobias, a differenza degli altri due, non genera casi di test automaticamente a partire da zero, ma a partire da uno schema assegnato. La riuscita dei casi di test generati riguarda le capacità del tester piuttosto che quelle dello strumento, il quale si limita ad effettuare le combinazioni dei valori forniti sollevando il tester dall’onere.

Inoltre, negli esempi riportati in precedenza è stato analizzato un programma molto semplice e con poche linee di codice, per cui il tempo impiegato per generare casi di test che ne massimizzassero la copertura è stato molto contenuto. Nel caso di programmi con un numero elevato di punti decisionali (e quindi di cammini indipendenti) ed una complessità sempre crescente, il tempo impiegato dai tool per generare casi di test efficaci ed efficienti potrebbe essere significativo. Vero è che *JPet* da parte sua consente di impostare un limite superiore al numero di iterazioni da eseguire, così come *EvoSuite* consente di impostare un limite temporale entro il quale lo strumento deve generare casi di test. Impostare però dei limiti relativamente bassi equivale a generare casi di test che potrebbero o meno garantire un’alta copertura del programma sotto test. Il tutto quindi dipende dalla complessità del programma che si intende testare e dal compromesso che si vuole raggiungere tra la copertura fornita dai casi di test risultanti e il tempo impiegato per generarli.

Restringendo l’attenzione ai due strumenti completamente automatici, *JPet* ed *EvoSuite*, è stato preso in esame un programma [5] più complesso del precedente e ne è stato analizzato il metodo *deleteEntry()*, riportato in Figura 2.21. Il programma analizzato è un Albero-RB, struttura dati utilizzata per implementare insiemi o array associativi.

Nelle Figure 2.22, 2.23 e 2.24 è riportata l’analisi di copertura dei casi di test generati con *EvoSuite*, assegnando rispettivamente un tempo pari a 60sec, 180sec e 360sec come “*Maximum Test Generation Time*”.

```

45 public void deleteEntry(EntryRBT p) {
46     int RED=0;
47     int BLACK=1;
48     modCount++;
49     size--;
50
51     // If strictly internal, copy successor's element to p
52     // and then make p
53     // point to successor.
54     if (p.left != null && p.right != null) {
55         EntryRBT s = successor (p);
56         p.key = s.key;
57         p.value = s.value;
58         p = s;
59     } // p has 2 children
60
61     // Start fixup at replacement node, if it exists.
62     EntryRBT replacement = (p.left != null ? p.left : p.right);
63
64     if (replacement != null) {
65         // Link replacement to parent
66         replacement.parent = p.parent;
67         if (p.parent == null)
68             root = replacement;
69         else if (p == p.parent.left)
70             p.parent.left = replacement;
71         else
72             p.parent.right = replacement;
73
74         // Null out links so they are OK
75         // to use by fixAfterDeletion.
76         p.left = p.right = p.parent = null;
77
78         // Fix replacement
79         if (p.color == BLACK)
80             fixAfterDeletion(replacement);
81     } else if (p.parent == null) {
82         // return if we are the only node.
83         root = null;
84     } else { // No children.
85         // Use self as phantom replacement and unlink.
86         if (p.color == BLACK)
87             fixAfterDeletion(p);
88         if (p.parent != null) {
89             if (p == p.parent.left)
90                 p.parent.left = null;
91             else if (p == p.parent.right)
92                 p.parent.right = null;
93             p.parent = null;
94         }
95     }
96 }

```

Figura 2.21: Metodo deleteEntry().

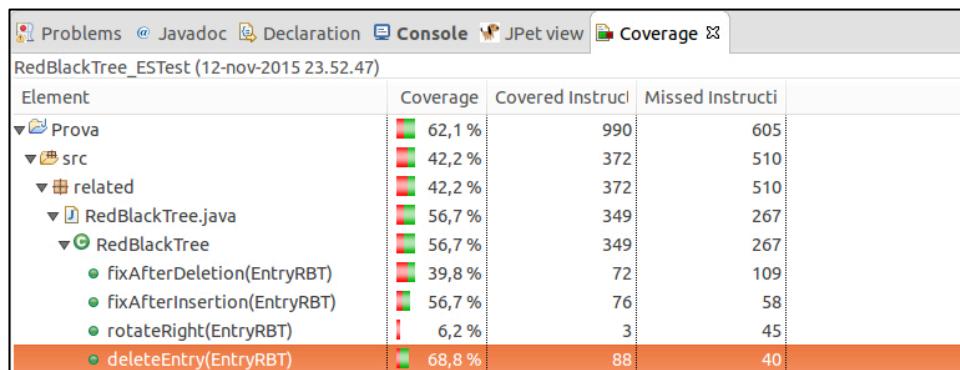


Figura 2.22: Copertura deleteEntry() con “Maximum test generation time” pari a 60sec.

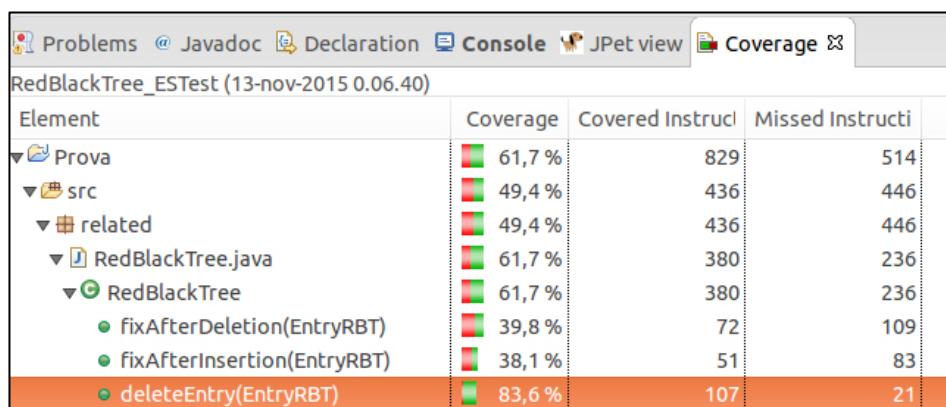


Figura 2.23: Copertura deleteEntry() con “Maximum test generation time” pari a 180sec.

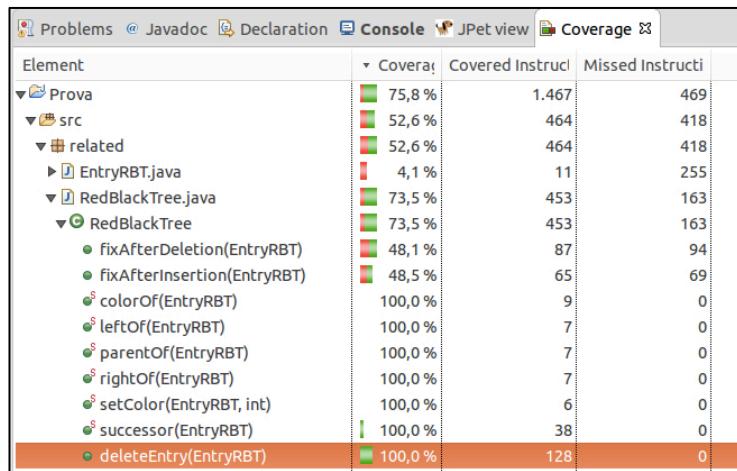


Figura 2.24: Copertura deleteEntry() con “Maximum test generation time” pari a 360sec.

Come si può osservare dalle figure precedenti, man mano che il tempo massimo assegnato allo strumento per la generazione dei casi di test aumenta, la copertura del metodo migliora fino a raggiungere il 100% quando vengono assegnati 360sec.

Per quanto *JPet*, i risultati ottenuti sono mostrati in Figura 2.25 e 2.26 dove si è posto k rispettivamente pari a 2 e 5. I risultati sono riportati nella Console di *Eclipse*, in accordo col fatto che la metrica di copertura utilizzata da *JPet* è la stessa di quella utilizzata da *EclEmma*.

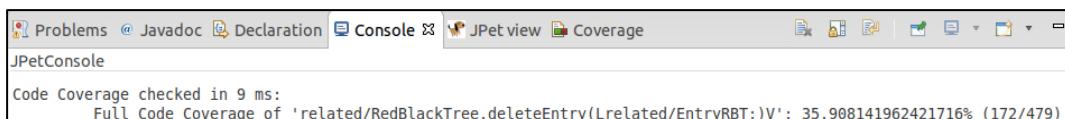


Figura 2.25: Copertura deleteEntry() con k = 2.



Figura 2.26: Copertura deleteEntry() con k = 5.

La copertura ottenuta con *JPet* risulta essere pari al 95%, nonostante k elevato. Inoltre, per raggiungere questo risultato, il numero di test generati da *JPet* risulta essere oltre 3000, mentre invece per raggiungere una copertura pari al 100% con *EvoSuite* ne sono stati sufficienti 22, in virtù delle minimizzazioni effettuate dallo strumento.

Per quanto riguarda invece il tempo impiegato per ottenere la massima copertura possibile, *JPet* risulta essere più rapido di *EvoSuite*, tenendo però presente che quest'ultimo, oltre ad impiegare un tempo pari al massimo di quello che gli viene assegnato (limite superiore costituito da 600sec), impiega un ulteriore tempo per la minimizzazione del risultato.

Conclusioni

E' ampiamente riconosciuto che il testing è un elemento essenziale di qualsiasi processo software di successo. Il problema fondamentale di tale disciplina riguarda l'oracolo, ovvero i risultati attesi da una serie di input che sollecitano il software sotto prova (SUT). Nonostante ci siano molti strumenti che generano automaticamente casi di test e che sono in grado di produrre test suite con un alto tasso di copertura del codice, non si può assumere la disponibilità di un oracolo automatizzato. Questo significa che un tester umano ha bisogno di specificare l'oracolo in termini di risultato atteso. Per rendere questo fattibile, bisognerebbe fare in modo di generare piccole test suite ad elevata copertura che rendano la generazione dell'oracolo il più semplice possibile. A tal proposito, si stanno cercando diverse soluzioni per sostenerlo lo sviluppatore. Tra queste, quella di produrre automaticamente asserzioni efficaci e di rendere i casi di test quanto più leggibili è possibile.

In conclusione, i vantaggi apportati dalla fase di testing, intesa come attività di sviluppo, sono notevoli e molteplici. Inoltre, affinchè la qualità del prodotto realizzato sia sempre maggiore è necessario velocizzare e rendere sempre più efficace tale attività. Per questo motivo ci si affida sempre più a strumenti automatici in grado di generare automaticamente casi di test, grazie ai quali è possibile sicuramente semplificare quello che sarebbe un lavoro lungo, complesso e costoso.

Bibliografia

- [1] Tratta da Automated Testing vs Manual Testing: Which Should You Use, and When?, <https://www.apicasystem.com/blog/automated-testing-vs-manual-testing/>.
- [2] Tratta da Yves Ledru, The TOBIAS test generator and its adaptation to some ASE challenges Position paper for the ASE Irvine Workshop.
- [3] Tratta da Gordon Fraser, Andrea Arcuri, Whole Test Suite Generation.
- [4] VendingMachine.java,
<http://tobias.liglab.fr/doc/Files/vendingMachine/VendingMachine.java>.
- [5] RedBlackTree.java,
http://costa.ls.fi.upm.es/pet/show_src_file.php?path=../../../../examples/related/RedBlackTree.
- [6] Ian Sommerville, Ingegneria del software, Pearson, 8 edizione, 1 Gennaio 2007.
- [7] Yves Ledru, German Vega, Taha Triki, Lydie du Bousquet, Test Suite Selection Based on Traceability Annotations Tool demonstration, Laboratoire d’Informatique de Grenoble UMR 5217, F-38041, Grenoble, France.
- [8] Federica Web Learing, <http://www.federica.unina.it/corsi/ingegneria-del-software-ingegneria/>.
- [9] Tobias On-Line documentation, <http://tobias.liglab.fr/doc/>.
- [10] Partial Evaluation-based Test Case Generator for Bytecode,

<http://costa.ls.fi.upm.es/pet/download.php>.

- [11] Elvira Albert, Israel Cabanas, Antonio Flores-Montoya, Miguel Gomez-Zamalloa, Sergio Gutierrez, jPET: an Automatic Test-Case Generator for Java, Complutense University of Madrid, Spain.
- [12] EvoSuite, <http://www.evosuite.org>.
- [13] Gordon Fraser, Andrea Arcuri, EvoSuite: Automatic Test Suite Generation for Object-Oriented Software.
- [14] Gordon Fraser, Andrea Arcuri, EvoSuite at the Second Unit Testing Tool Competition, University of Sheffield Dep. of Computer Science, Sheffield, UK.
- [15] Gordon Fraser, Andreas Zeller, Mutation-driven Generation of Unit Tests and Oracles.
- [16] Davide Balzarotti, Paolo Costa, Generazione Automatica e Valutazione di Casi di Test.