UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

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

elaborato di laurea

Code Coverage, strumenti per la copertura del codice Java e JavaScript

Anno Accademico 2013/2014

relatore

Ch.mo Prof. Porfirio Tramontana

candidato
Pasquale Porricelli
matr. 534 003265

[Dedica]

Indice

Introduzione

Il testing è il processo di valutazione di un sistema software (o di un suo componente) basato sul suo comportamento in esecuzione sotto condizioni controllate.

Esso rappresenta la parte di analisi dinamica di un processo di revisione più ampio noto come "Verifica & Validazione". La parte complementare, ossia quella statica, è invece rappresentata dall'ispezione. Purtroppo un punto a sfavore dell'attività di testing, nei confronti di quest' ultima, risiede nel fatto che per poter essere effettuata necessita di una versione eseguibile del codice sorgente, quindi non ce ne si può servire per un codice incompleto.

Facendo una breve riflessione sugli intenti della fase di testing si evidenzia subito un'importante differenza, rispetto a tutte le altre attività che prendono parte al processo di sviluppo del software (analisi dei requisiti, progettazione, implementazione): esso è "distruttivo", a differenza delle altre fasi che invece sono tutte costruttive. Lo scopo specifico dell'attività di testing, citando l'informatico Wybe Dijkstra, è quello di trovare difetti nel programma prima della consegna all'utente, non di dimostrarne l'assenza.

L'importanza della fase di testing sta emergendo soprattutto negli ultimi anni, con la nascita di figure professionali operanti in questo settore: per fare un esempio si può pensare all'innovativo modello di sviluppo Test-Driven Development (TDD), che fa parte delle regole alla base dell'Extreme Programming (XP), secondo il quale lo sviluppo vero e

proprio dev'essere preceduto dalla stesura di test automatici.

Il concetto basilare da tener presente, infatti, è che il ciclo di vita di un programma non termina al momento della consegna, ma al contrario, ne rappresenta solo il punto di inizio. Se vogliamo che il nostro software continui ad essere usato esso si deve prestare a successive modifiche, evoluzioni e manutenzioni, e pertanto la predisposizione di un buon testing diventa un aspetto fondamentale.

Se per diversi motivi, come la ristrettezza dei tempi di consegna, si dovesse tralasciare di scrivere codice di test, il prodotto finale che otterremo rischierebbe di risultare poco flessibile e poco robusto rispetto a futuri cambiamenti. Per uno sviluppatore, questo, col passare del tempo, diventa motivo di scoraggiamento nell'apportare modifiche al codice, conducendo inevitabilmente il software alla sua "morte".

Considerare la fase di testing come attività di sviluppo quindi, comporta diversi benefici:

- Riduzione dei cicli di sviluppo, si evita che alcuni errori possano manifestarsi in uno stadio avanzato del processo di sviluppo
- Semplificazione di modifiche alla struttura interna atte a migliorarne la qualità (refactoring)
- Aumento della produttività, infatti una buona attività di testing costante consente di ottenere prodotti di qualità, risparmiando le risorse che sarebbero occorse per porvi rimedio

L'attività di test quindi è un operazione tutt'altro che secondaria ma è anzi un processo complesso nel quale è richiesta pratica, esperienza ed è necessaria una formazione del personale addetto se si vogliono ottenere risultati soddisfacenti.

Nel corso di questo elaborato si procederà all'analisi ed al confronto di strumenti che si occupano di un particolare aspetto della fase di testing, la copertura del codice, in particolare per due tra i linguaggi di programmazione più utilizzati e in crescita al giorno d'oggi Java e JavaScript.

Capitolo 1: Qualità del testing

L'obiettivo della fase di testing è quello di rilevare la presenza di malfunzionamenti per facilitare il successivo processo di debugging, che si occuperà della rimozione dei bugs dal codice. Il testing inoltre non deve alterare in alcun modo l'ambiente in cui opera il programma, in quanto potrebbe inficiare la proprietà di ripetibilità del test case.

Un test si dice *ideale* se non trova malfunzionamenti e al contempo riesce a provare la loro assenza, tuttavia questo è uno di quei problemi che vengono definiti come *indecidibili*, ovvero per i quali non vi è una soluzione. Quando non si trovano bug, infatti, non si comprende se il prodotto sia corretto o se l'attività di test non sia stata soddisfacente.

Per tale motivo la qualità di un processo di testing può anche essere definita attraverso due parametri:

Efficienza indica il numero di malfunzionamenti evidenziati in rapporto ai casi di test effettuati, in altre parole il testing deve essere in grado di trovare difetti utilizzando il minor numero possibile di casi di test. L'efficienza va privilegiata allo scopo di tenere bassi i costi del testing, non bisogna tuttavia esagerare, infatti test molto efficienti potrebbero avere l'inconveniente di coinvolgere più bug nello stesso caso di test, rendendo difficile il lavoro di debugging.

Efficacia indica la scoperta di quanti più malfunzionamenti è possibile in rapporto a quelli totali. Va invece privilegiata l'efficacia se si vuole ottenere un certo livello di

affidabilità (indispensabile per prodotti che andranno ad operare in ambiti critici). Non potendo conoscere l'insieme totale, l'efficacia è un parametro non quantificabile, se non in maniera statistica.

Dal momento che l'attività di testing richiede una buona parte delle risorse allocate per il progetto e che un testing esaustivo (ovvero un test ideale) è un problema indecidibile, si reputa conclusa l'attività di testing in base ad altri criteri, in termini di costo, di tempo o statistici.

I risultati ottenuti dai casi di test vanno poi confrontati con quelli attesi per verificare la bontà del software implementato.

Un *oracolo* genera previsioni dei risultati attesi, gli oracoli possono essere versioni precedenti del programma, o sistemi prototipo.

È da valutare, inoltre, anche il personale che pianifica e svolge l'attività di test, in quanto uno sviluppatore può agire con una mentalità più protettiva verso il proprio codice (si crea in effetti un conflitto di interessi) di quanto sarebbe quella di un tester che tende ad avere una visione complessiva del problema, imparziale e particolarmente vicina al modo di pensare dell'utente più che alla macchina.

1.1 Testing White e Black box

Le modalità che possono essere utilizzate nell'approccio al software testing sono essenzialmente riconducibili a due modalità: a "scatola nera" (black box testing) o testing funzionale, ed a "scatola bianca" (white box testing) o testing strutturale.

Nell'approccio black box, il testing è fondato sull'analisi degli output generati dal sistema o sai suoi componenti, in risposta a specifici input definiti sulla base della sola conoscenza dei requisiti specificati per il sistema o di suoi componenti. Il tester quindi non è a conoscenza dei dettagli implementativi ma ha una visione esterna delle funzionalità che il sistema dovrebbe offrire.

I vantaggi dell'utilizzo dell'approccio Black Box sono:

- È particolarmente adatto ed efficiente per grandi quantità di codice.
- Separa nettamente la prospettiva dell'utente (tester) dal punto di vista dello sviluppatore in quanto aventi ruoli chiaramente separati.
- Un buon numero di tester moderatamente esperti possono provare l'applicazione senza alcuna conoscenza dei sistemi di implementazione, linguaggio di programmazione o di sistemi operativi.

Tuttavia sono inevitabili gli svantaggi seguenti:

- La copertura è limitata, in quanto si possono eseguire solo un numero definito di possibili scenari.
- Il test può essere insufficiente a causa del fatto che il tester ha solo una limitata conoscenza dell'applicazione.
- Copertura "cieca" dato che il tester non può prosi come obiettivo specifici segmenti di codice o zone a rischio di errori.
- Casi di test difficili da progettare.

Nell'approccio White Box, è possibile avere un'analisi più approfondita, in quanto è fondato sull'individuazione di specifici input, definiti sulla base della conoscenza della struttura del software, ed in particolare del codice.

Il tester quindi deve avere una visione del codice sorgente e scoprire quale unità o blocco di codice potrebbe comportarsi in modo inappropriato.

Ovviamente, anche per questo approccio possiamo annoverare vantaggi e svantaggi.

Vantaggi:

- Dato che il tester è a conoscenza del codice sorgente, diventa semplice scoprire quale tipo di dati può aiutare a testare l'applicazione in modo efficace.
- Aiuta ad ottimizzare il codice.
- Righe di codice non utilizzate possono essere rimosse (laddove invece potrebbero portare a difetti imprevisti)

- Grazie alla conoscenza del codice, il tester può arrivare a una massima copertura nella scelta dello scenario.

Svantaggi:

- Essendo richiesti tester molto esperti per la creazione di test White Box, i costi sono più elevati.
- In alcuni casi è impossibile guardare in ogni punto per trovare errori nascosti che possono creare problemi, perciò diversi percorsi potrebbero non essere testati.

Analizziamo ora i criteri che rientrano nella strategia White-Box:

- *Copertura delle istruzioni (statement coverage)* : Bisogna trovare un insieme di input tale da garantire che ogni istruzione del codice sia eseguita almeno una volta.
- *Copertura delle decisioni (branch coverage)* : Bisogna esaminare ogni ramificazione del diagramma di flusso, sia per il valore di verità che per il valore di falsità, per garantire che il sistema non fallisca in situazioni in cui non è possibile non prendere decisioni: esistono almeno due eccezioni. Il primo caso degenere si ha se il codice non presenta alcuna decisione. Il secondo si ha quando una sezione viene eseguita opzionalmente, cosa che accade quando abbiamo diversi punti di ingresso nel codice. Deve essere affiancato dallo statement coverage.
- *Copertura delle condizioni (condition coverage)* : Bisogna scrivere un insieme di casi di test tali da far assumere a ciascuna condizione elementare tutti i possibili valori almeno una volta. Anche a questo va affiancato lo statement coverage.
- *Copertura delle condizioni e decisioni* : Bisogna ricercare un insieme di casi di test tali da garantire che ogni condizione assume il valore di verità e falsità almeno una volta e che ogni decisione sia percorsa almeno una volta sia per il ramo vero che per il ramo falso. Si deve tuttavia considerare che una macchina non è in grado di verificare condizioni multiple senza spezzarle in più condizioni singole su cui lavorare.

- *Copertura delle condizioni multiple* : Bisogna pianificare l'insieme di test case in modo che preveda tutti i possibili valori di ciascuna combinazione di condizioni presente in una decisione. Anche in questo caso si deve affiancare lo statement coverage per coprire i casi eccezionali prima elencati.

I criteri di selezione identificano le motivazioni che hanno spingono il capo progetto a scegliere una tecnica di test piuttosto che un'altra. Occorre fornire motivazioni corrette e coerenti con gli obiettivi da perseguire, che giustifichino la validità della scelta effettuata e ne illustrino brevemente i vantaggi. Gli obiettivi di adeguatezza del test rappresentano le percentuali di copertura del codice, ovvero quanto il nostro test è in grado di esaminare rispetto all'intero prodotto; maggiore sarà la copertura e maggiore sarà l'efficienza del test. Tuttavia non è possibile raggiungere una copertura totale del codice, a causa dei costi in termini economici, temporali e di fattibilità; come spesso accade in vari ambiti dell'ingegneria bisogna saper optare per un giusto compromesso.

1.2 Copertura del codice

Come anticipato, 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 persino una misura del livello di qualità del sistema (vedi qui [2] e lo studio sul **Software Testing** pubblicato su IEEE Explore da Rudolf Ramler, Theodoric Kopetzky, Wolfgang Platz, con il titolo "***Value-based coverage measurement in requirements-based testing: Lessons learned from an approach implemented in the TOSCA testsuite - Proceedings of the 38th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2012)***")[2]. Di sicuro c'è la necessità di aver ben chiaro il livello

di affidabilità dei test utilizzati. Non è raro infatti che i team di sviluppo abbiano la falsa sensazione di aver prodotto un codice di alta qualità giacché il sistema passa indenne per 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 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.

Alla luce delle precedenti considerazioni, si intuisce dunque che è molto importante utilizzare appositi tool di analisi della copertura del codice. Chiaramente si tratta di una condizione necessaria ma non sufficiente. È necessario eseguire frequentemente l'analisi della copertura per esaminare 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 questi casi può aver senso eseguire i test di copertura più frequentemente.

Il livello di copertura del codice da raggiungere è un valore che dipende da diversi fattori, per esempio il dominio dell'applicazione. È plausibile, facendo un esempio banale, che dovendo implementare il sistema di controllo di un reattore nucleare si voglia raggiungere

una copertura molto prossima al 100% (sebbene neppure questo 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 non risulti conforme a quello prestabilito, è opportuno implementare ulteriori casi di test. Tuttavia invece di procedere come spesso avviene implementando nuovi metodi in modo casuale (il che, con tutta probabilità, aumenta di 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 beneficerebbero maggiormente di ulteriori verifiche. Ovviamente è opportuno anche fare in modo che aree di particolare interesse/sensibilità presentino un livello molto elevato di copertura.

Capitolo 2: Strumenti per la copertura del codice Java

Fatte le dovute introduzioni possiamo passare adesso in rassegna, quali sono i migliori tools per eseguire dei test di copertura del codice rispetto al linguaggio di programmazione Java.
Fino a circa 10 anni fa, nell'ambito dello sviluppo in linguaggio Java, si poteva usufruire di ottimi ambienti integrati (IDE) e compilatori (Eclipse o Netbeans per citarne due dei più famosi) aderenti alla filosofia dell'open source, tuttavia mancavano alternative valide a prodotti commerciali per l'analisi di copertura, alternative che si sono invece sviluppate proprio a partire dalla seconda metà della scorsa decade, con l'aumentare dei progetti open source sul web. Restringendo il campo ai software che operano con il linguaggio java, l'osservazione è che ce ne sono davvero pochi che sono stati pensati e progettati per essere integrati. La maggior parte nasce per adattarsi ad un uso specifico, a linea di comando o come plug-in, per esempio, e non offrono una API approfonditamente documentata che consenta di incorporarli in contesti differenti. In java gli analizzatori di copertura si dividono sostanzialmente in tre categorie dipendenti dal livello su cui agiscono, quelli che aggiungono la strumentazione al codice sorgente, quelli che aggiungono la strumentazione al bytecode, quelli che eseguono il programma su una JVM modificata.

Per confrontare tra loro i diversi strumenti, essi verranno tutti utilizzati per verificare la copertura del codice di un semplice programmino Java.

I fattori di confronto saranno:

- la metodologia di istrumentazione
- tipo del toolkit (riga di comando / plugin per ambiente di sviluppo)
- formato del report di copertura (supporto all'estrazione di informazioni dal report)
- facilità di utilizzo
- interoperabilità (in particolare con JUnit, un framework che si occupa dell'automatizzazione di test, in particolare quelli di unità)
- convenzioni sul conteggio

2.1 Emma

Il primo toolkit che sarà analizzato è uno dei migliori e dei più utilizzati fra gli strumenti disponibili open source, Emma [3].

Emma è un toolkit per l'analisi e il report della copertura di codice java, distribuito secondo Common Public License (cpl). Questo strumento è uno dei primi open source nati per questo scopo e si differenzia dagli altri esistenti sul mercato poiché consente profili di copertura sia di piccoli progetti individuali, sia di progetti su larga scala aziendale. Un tale strumento è essenziale per la rilevazione di codici morti e verificare quali parti dell'applicazione sono esercitate dalla suite di test. Sono riportate di seguito le sue caratteristiche di rilievo:

- Rapidità: Emma è al 100% codice java (Bytecode) per cui viene eseguito contestualmente alla classe. Questo gli conferisce una considerevole velocità anche grazie ad un basso overhead runtime (dal 5 al 20%) mentre l'overhead della memoria si attesta su qualche centinaio di bytes per classe
- Non invasivo: essendo bytecode non necessita né della modifica di progetti, ne di

librerie esterne e quindi di elementi aggiuntivi. Questo fa si che possa essere eseguito su qualunque JVM 2

- Output flessibile: è possibile ottenere diverse tipologie di formati per i report, dal semplice testo a formati come HTML o XML, di cui l'utente controlla la profondità di dettaglio.

Emma è stato implementato per lavorare su ogni ambiente *Java 2 Platform Standard Edition (J2SE)*.

Il funzionamento del tool Emma si basa sulla marcatura di ogni linea di codice con un flag, che viene settato contestualmente all'esecuzione della relativa linea di codice. Tramite lo strumento *ctl*, è anche possibile avere un controllo remoto su Emma a runtime, e ripristinare Emma su una JVM remota.

Un esempio d'uso:

java –cp <classpath\emma.jar> emma ctl –connect localhost:47653

Emma permette la copertura di classi, metodi, singole istruzioni o blocchi di codice mentre non è affidabile invece in presenza di salti (jump). Supporta inoltre la copertura di *linee frazionarie di codice*, fornendo una percentuale di copertura anche per le singole linee in modo da aiutare nell'individuazione degli stessi salti (cfr. 2.1.2 Convenzioni sul conteggio).

Il tool a riga di comando viene distribuito in un archivio java, *emma.jar*.

Alcuni aspetti del comportamento di Emma possono essere modificati in vari modi, tramite le proprietà della JVM, tramite file esterni oppure tramite opzioni della riga di comando. I comandi che è possibile eseguire sono:

emmarun esegue un'applicazione java indipendente nel classloader di emma, senza una fase di utilizzazione separata. Fornisce in uscita uno o più report di copertura. E' una modalità *on-the-fly* (al volo).

instr <command> utilizza un set di classi e/o archivi in una lista di directories. E' una modalità offline.

report <command> riceve e combina un numero arbitrario di metadati della classe di dati di copertura a run time per produrre dei rapporti sulla copertura. E' una modalità offline.

merge <command> compatta i metadati diversi, copertura o file di dati di una sessione. Anche questa è una modalità offline e risulta utile per memorizzare dati in maniera compatta e mantenere insieme quelli relativi a una stessa sessione.

Per quanto riguarda il concetto di report bisogna considerare anche le modalità con cui è avvenuta la fase di compilazione, infatti il funzionamento è garantito solo nel caso in cui la classe venga compilata con le informazioni di debug complete. Quanto appena detto è importante perché in tal caso il package contenuto nella classe è comprensivo anche dei sorgenti del programma senza i quali non potremmo associare il livello gerarchico delle istruzioni alle linee di codice. Basandosi sulla sola analisi statica dei sorgenti tuttavia si espone il risultato a mancanze create da eventuale presenza di codice dinamico.

Le opzioni (flag) disponibili per i comandi sono invece:

-cp classpath

-r report_type

-merge [yes|no]

-sp sourcepath

-ix filter_patterns

Come comportamento predefinito emma non fa uso della funzione system.exit() a meno che non venga esplicitamente specificata l'opzione -exit. I valori ritornati saranno indicazione dell'esito della chiamata, essa ritornerà 0 in caso di esito corretto, 1 in caso di uso scorretto delle opzioni, 2 per tutti gli altri errori. Sebbene Emma sia un tool di per sé molto veloce, tale da far in modo che la velocità di elaborazione dipenda dal file di I/O, è comunque opportuno non appesantirlo, ma fargli gestire la giusta quantità di lavoro, definendo l'effettivo instruction set, ovvero l'insieme delle classi utilizzate.

La sequenza di passi da eseguire quando si usa emma dipende dalla tipologia di copertura scelta.

In modalità on-the-fly c'è ben poco da fare in aggiunta alla normale procedura, si deve utilizzare solo il comando *emmarun* all'inizio della nuova esecuzione ed eventualmente decidere dove collocare il report in output (con l'opzione –d <directory>). A runtime il

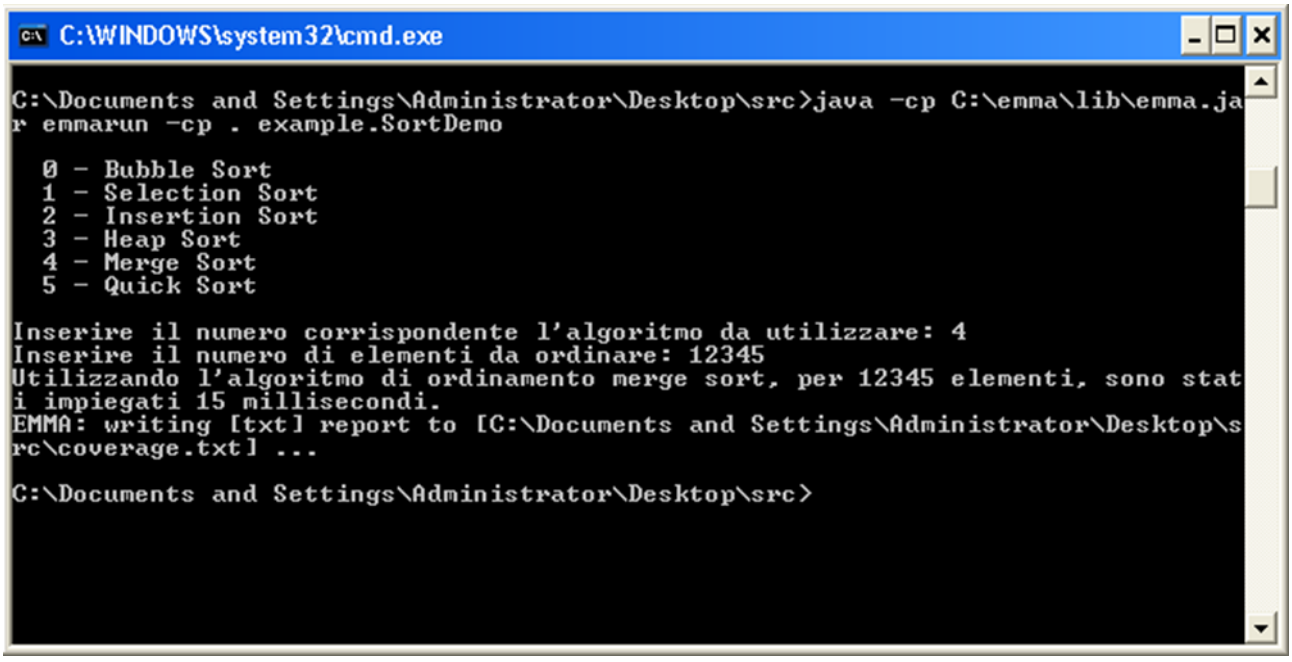
classloader di emma installa se stesso come classloader dell'applicazione, in modo da bypassare il classloader del sistema. In modalità offline invece la sequenza da eseguire è quella di utilizzare una o più volte il comando *instr* per utilizzare le directory e le classi, eseguire l'applicazione o la test suite usando le classi definite, opzionalmente utilizzare il merge dei metadati delle diverse sessioni di esecuzione, infine utilizzare il comando di report per ottenere il profilo di copertura richiesto.

Esempi d'uso

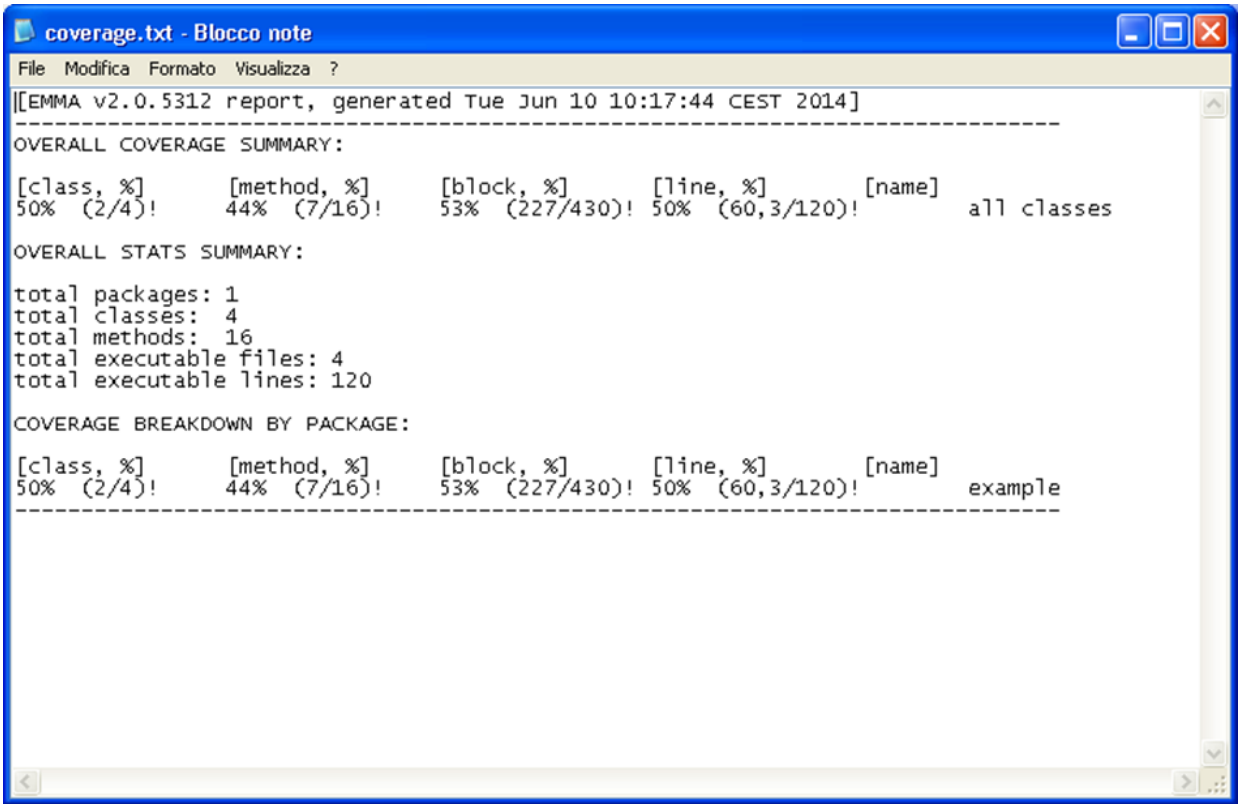
Il metodo consigliato dalla documentazione di emma per l'utilizzo tramite riga di comando è di aggiungere il file emma.jar direttamente alle librerie del jre, ma si può tranquillamente aggiungere il classpath della libreria emma.jar al momento dell'esecuzione.

Una volta scaricati ed estratti i files .jar (disponibili all'indirizzo http://emma.sourceforge.net/downloads.html), si apre una finestra di shell, ci si posiziona tramite il comando *cd* alla directory utilizzata e si lancia Emma eseguendo i comandi che ci interessano a seconda dell'output desiderato:

java –cp …/dir/emma.jar emmarun –cp . example.SortDemo

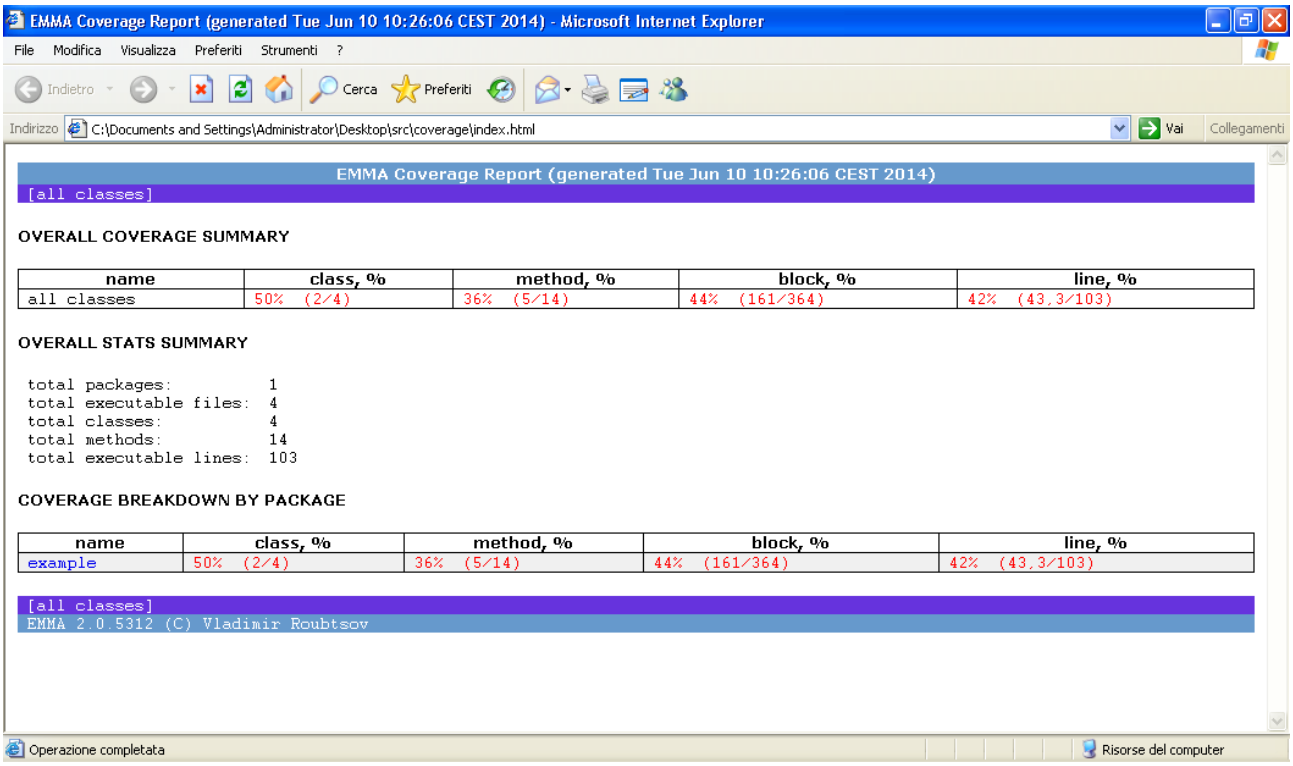


per ottenere informazioni nel file *coverage.txt* che di default si trova all'interno della directory del progetto utilizzato.



per ottenere invece, in output un file *.html* basta dare il comando

java –cp emma.jar emmarun -r html –cp . example.SortDemo



oppure

java –cp emma.jar emmarun –r html –sp [....] –cp . example.SortDemo

per ottenere in output un file .html specificando la directory dei sorgenti, qualora fossero in directories differenti.

Nella modalità on-the-fly, è possibile ottenere in maniera veloce un report per il test di programmi di piccole/medie dimensioni. Quando invece c'è bisogno di testare programmi più complessi si può utilizzare la modalità offline che è meno veloce ma molto più completa.

In questa modalità abbiamo una separazione tra fase di istrumentazione ed esecuzione; nello sviluppo di un programma di dimensioni importanti c'è la necessita di raccogliere ed unire dati di copertura da più esecuzioni nonché da più processi JVM.

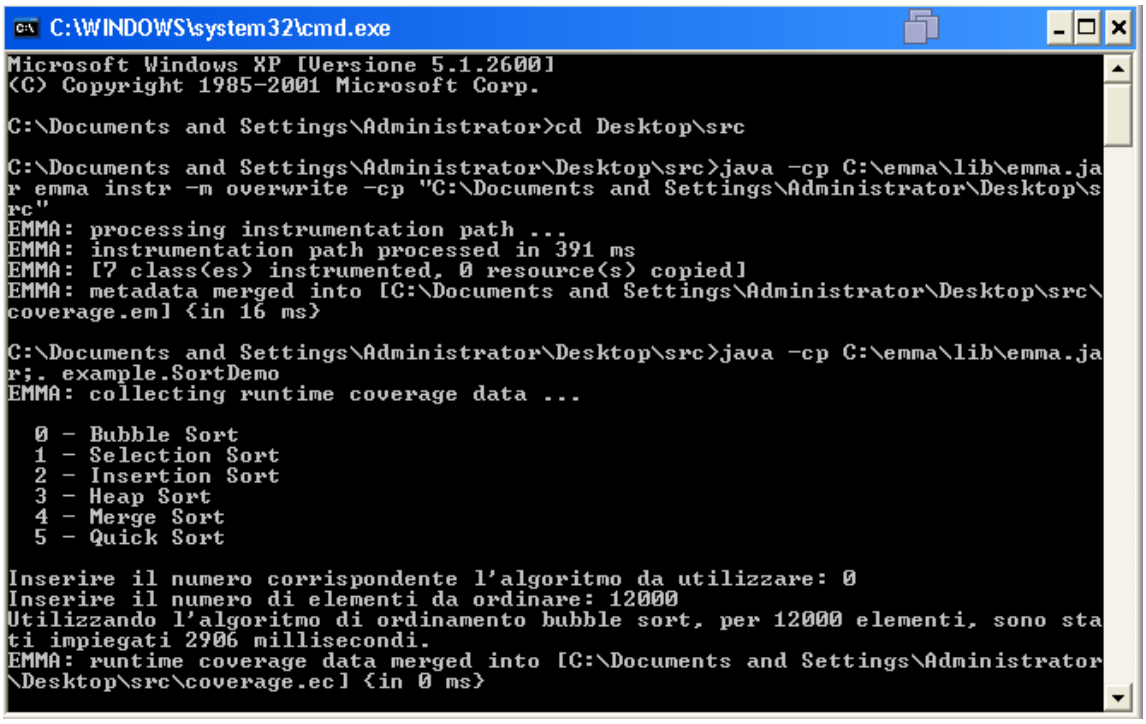
Analizziamo il funzionamento.

Dopo aver compilato il codice sorgente, le classi saranno istrumentate e quindi modificate (in particolare le nuove classi strumentate sostituiscono le precedenti) attraverso il comando ***instr***.

In questo modo viene generato anche un file *coverage.em* che contiene la classe metadata. Una delle features più utili sta nel fatto che EMMA possiede un "sistema di filtraggio" per il giusto insieme di classi da strumentare, in questo modo è possibile scalare il lavoro di testing a partire da progetti di grandi dimensioni garantendo prestazioni ottimali. Il filtro si attiva con il flag *–ix*.

A questo punto abbiamo le classi istrumentate e per ottenere dati di copertura basta eseguire l'applicazione più volte ed ottenere diversi gradi di copertura.

Come si può notare dallo screenshot, una volta eseguita l'applicazione EMMA crea un nuovo file coverage.ec dove carica il profilo di copertura runtime.



Infine per ottenere un report, Emma combina la classe metadata ed il profilo di copertura runtime producendo un report di testo, xml oppure html (dove sarà anche possibile vedere il codice sorgente evidenziato nei colori rosso verde e giallo a seconda della copertura di ciascuna linea).

2.1.1 Interoperabilità con JUnit

Per quanto riguarda l'interoperabilità con il framework JUnit, come descritto nelle FAQs della documentazione di Emma, siccome non c'è modo di prevedere come l'applicazione verrà eseguita e siccome non tutti i programmatori usano come test driver JUnit, non è supportata una vera e propria integrazione con tale framework ma si può, come descritto in precedenza, utilizzare emma on-the-fly sui casi di test con il comando emmarun oppure strumentare le classi dopo averle compilate e prima di partire con i vari tests.

2.1.2 Convenzioni sul conteggio

Le unità di copertura fondamentali di Emma sono i basic bloks (tutti gli altri tipi di copertura sono derivati dalla copertura del basic block in un modo o nell'altro). Un blocco di base o basic block è una sequenza di istruzioni bytecode, senza salti. In altre parole, un basic block viene sempre eseguito (in assenza di eccezioni) come un'unità atomica. Dato

che diverse linee di sorgente Java possono essere nello stesso basic block, per ragioni di efficienza in fase di esecuzione, secondo la filosofia di Emma ha più senso tenere traccia dei basic block, piuttosto che singole linee. Inoltre, un dato file sorgente Java contiene un gran numero di linee che non sono eseguibili (non sono presenti in nessun bytecode "reale"), ad esempio commenti, istruzioni import, ecc

Emma non tenta di "tracciare" il successo o il fallimento di ogni singola istruzione bytecode ma segna un basic block come coperto quando viene raggiunta la sua ultima istruzione. Secondo la filosofia degli sviluppatori di Emma, il numero di blocchi di base in un determinato metodo è una misura della complessità del metodo migliore rispetto, ad esempio, al puro conteggio delle righe.

Si noti che la copertura del basic block del 100% implica sempre il 100% di copertura delle linee. Per quanto riguarda la copertura delle linee in Emma, c'è un problema di linee "frazionarie" ossia linee eseguite solo parzialmente, questo perché un basic block appartiene sempre ad un unico metodo, ma una data linea sorgente può essere divisa tra metodi o classi multiple.

L'approccio di Emma all'analisi di copertura della linea è quello di proiettare la copertura del basic block sulla struttura delle righe del codice sorgente.

Emma verifica come la struttura basic block in un metodo e le informazioni della linea di sorgente si rappresentano l'un l'altra e per ogni linea, determina se tutti i basic blocks di cui questa linea è responsabile sono stati coperti. Se questa condizione è verificata, la linea (oppure le linee, dato che può verificarsi che più linee appartengano allo stesso basic block) è coperta al 100%. Altrimenti, la copertura per la linea è una percentuale (frazione) inferiore a 100%.

Anche se concettualmente semplice, questa analisi è complicata da alcuni fattori:

- Un dato file sorgente java può contenere più classi
- Una data linea sorgente può essere ripartita su più metodi o anche classi
- Il codice di una determinata linea sorgente può essere clonata dal compilatore in diversi metodi indipendenti (ad esempio il codice di inizializzazione)

Emma tenta di mantenere l'analisi della copertura delle linee coerente in questi casi, tuttavia questi sono i motivi del perché si possono vedere statistiche di copertura di linee di codice non riportate in maniera intuitiva (coperto/non coperto).

Per quanto riguarda la copertura delle classi, una classe, eseguibile, viene considerata coperta se è stata caricata e inizializzata dalla JVM. L'inizializzazione della classe implica che il costruttore statico della classe (se presente) venga eseguito. Una classe può essere coperta anche se nessuna delle sue altre funzioni membro è stata eseguita.

Emma infine considera un metodo coperto, quando il suo primo basic block è stato coperto.

Di seguito, il report di copertura di un semplice costrutto if:

CicloIf.java

```
package Prova;

public class CicloIf {
    public static void main(String[] args) {

        int x=0;

        if (x == 0) {
            System.out.println("X è zero");
        } else {
            System.out.println("X non è zero");
        }

    }

}
```

EMMA Coverage Report (generated Mon Sep 15 19:57:01 CEST 2014)
[all classes]

OVERALL COVERAGE SUMMARY

name	class, %	method, %	block, %	line, %
all classes	100% (1/1)	50% (1/2)	60% (9/15)	67% (4/6)

OVERALL STATS SUMMARY

```
total packages:          1
total executable files:  1
total classes:           1
total methods:           2
total executable lines:  6
```

COVERAGE BREAKDOWN BY PACKAGE

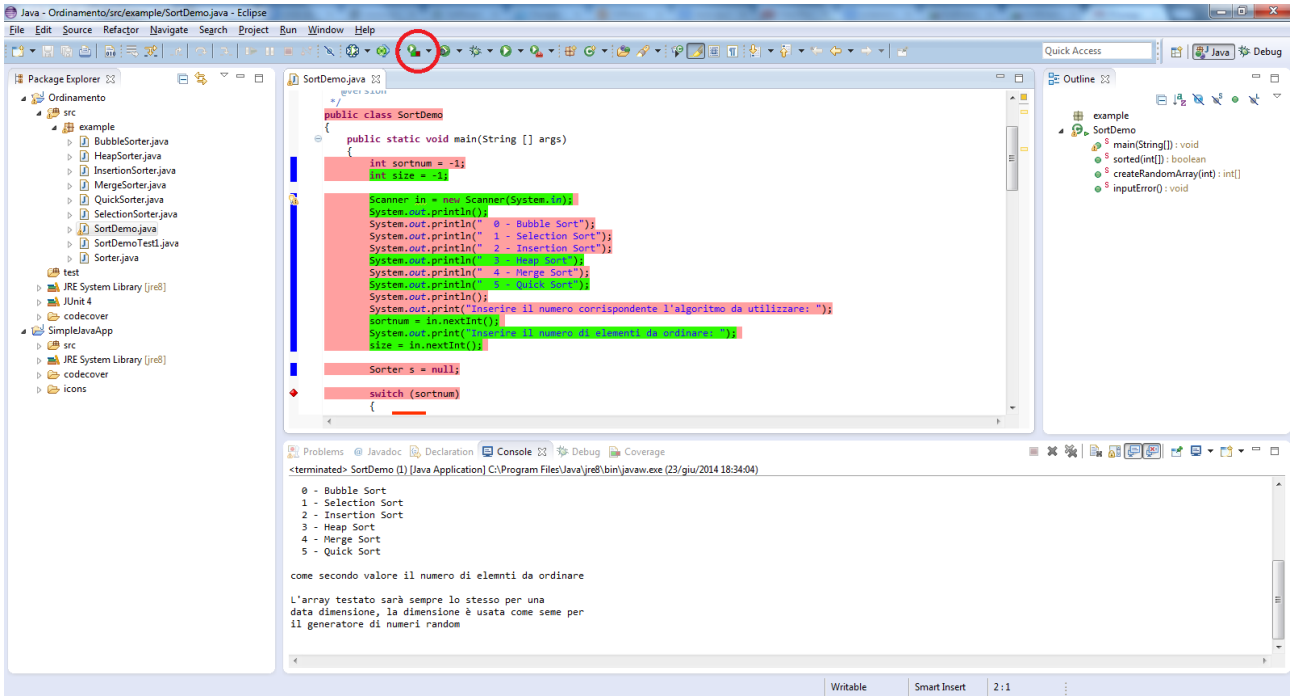
name	class, %	method, %	block, %	line, %
Prova	100% (1/1)	50% (1/2)	60% (9/15)	67% (4/6)

[all classes]
EMMA 2.0.5312 (C) Vladimir Roubtsov

2.2 EclEmma

EclEmma [4] nasce come plug-in basato sulle librerie di Emma, dalla versione 2.0 invece, si basa sulle librerie di code coverage del tool JaCoCo. Tramite questo plug-in 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 esportarle, fare il merge di diverse sessioni.

Le modalità di installazione sono analoghe a quelle di altri plug-in Eclipse, tramite la procedura di installazione di nuovo software. Una volta installato viene aggiunta una nuova funzionalità nel menù di esecuzione, ovvero l'esecuzione in modalità di copertura. A seguito di questa azione, viene aperta una nuova scheda nel pannello inferiore dell'interfaccia di eclipse che riporta le statistiche percentuali dettagliate di copertura. E' poi possibile, a richiesta, esportare (o anche importare) i dati di copertura. Viene di seguito riportato un esempio a riprova della immediatezza comunicativa di EclEmma.



Per esportare il report basta andare nel menu file −> export… −> scegliere Coverage Report dal menu Java

i report possono essere in formato XML, HTML oppure in formato CSV.

Con EclEmma è possibile quindi:

- Lanciare un programma e vedere la copertura durante il suo uso
- Lanciare casi di test e vedere la copertura
- Vedere in dettaglio i dati della copertura ed esportarli
- Fare il merge di sessioni

Annotazione del codice

• Verde: testato

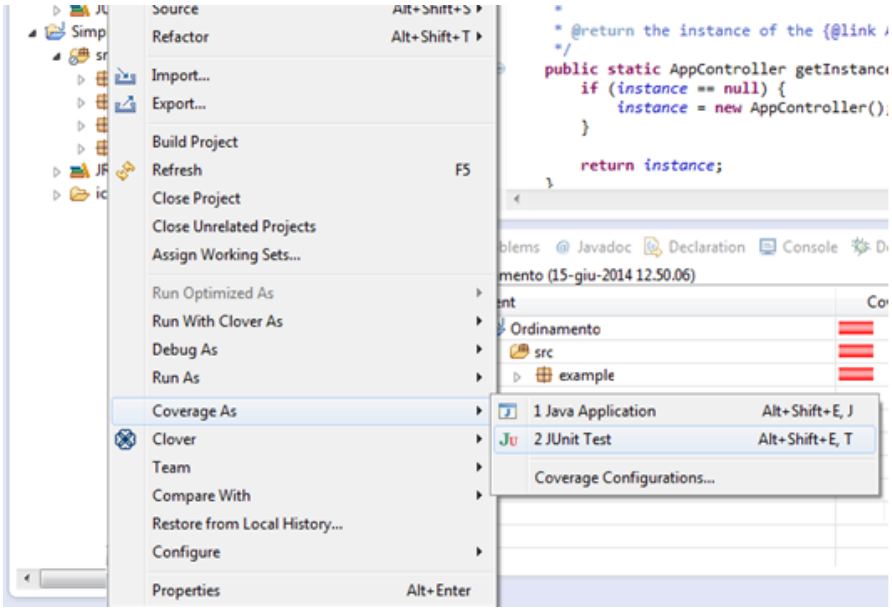
• Rosso: non testato

• Giallo: testato parzialmente

EclEmma istrumenta le classi "al volo". Questa operazione semplifica molto l'analisi della copertura del codice dato che non è necessaria nessuna pre-istrumentazione e modifica del classpath.

2.2.1 Interoperabilità con JUnit

EclEmma offre la copertura del codice dei casi di test JUnit direttamente all'interno dell'IDE Eclipse. Infatti, invece di eseguire i test utilizzando la procedura Run > Run As > JUnit Test, si può utilizzare il comando Coverage > Coverage As > JUnit Test.



Questo comando istrumenta il codice fornendo quindi una vista della copertura del codice ed evidenzia nell'editor qual è il codice che è stato eseguito. Dopo aver eseguito il codice si

può, come descritto in precedenza, effettuare un merge di diverse sessioni e fornire un report di copertura in formato html, xml o csv attraverso il comando export..

2.2.2 Convenzioni sul conteggio

Nel plugin EclEmma le metriche utilizzate sono differenti dal tool Emma.

La copertura delle istruzioni fornisce informazioni sulla quantità di codice che è stato eseguito. Questa metrica è completamente indipendente dalla formattazione del codice sorgente ed è sempre disponibile, anche in assenza di informazioni di debug nei class files.

La copertura dei rami in EclEmma è una metrica che conta il numero totale rami (come if e switch) all'interno di una funzione membro e determina il numero di rami eseguiti o meno. Le eccezioni non sono considerate come branches.

EclEmma calcola anche la complessità ciclomatica per ogni funzione membro non astratta e riassume la complessità per le classi, pacchetti e gruppi.

Per definizione, la complessità ciclomatica è il numero minimo di percorsi che possono, in combinazione (lineare), generare tutti i possibili percorsi attraverso una funzione membro. EclEmma calcola complessità ciclomatica di un metodo con la seguente equazione equivalente sulla base del numero di rami (B) e il numero di punti di decisione (D):

$v(G) = B - D + 1$

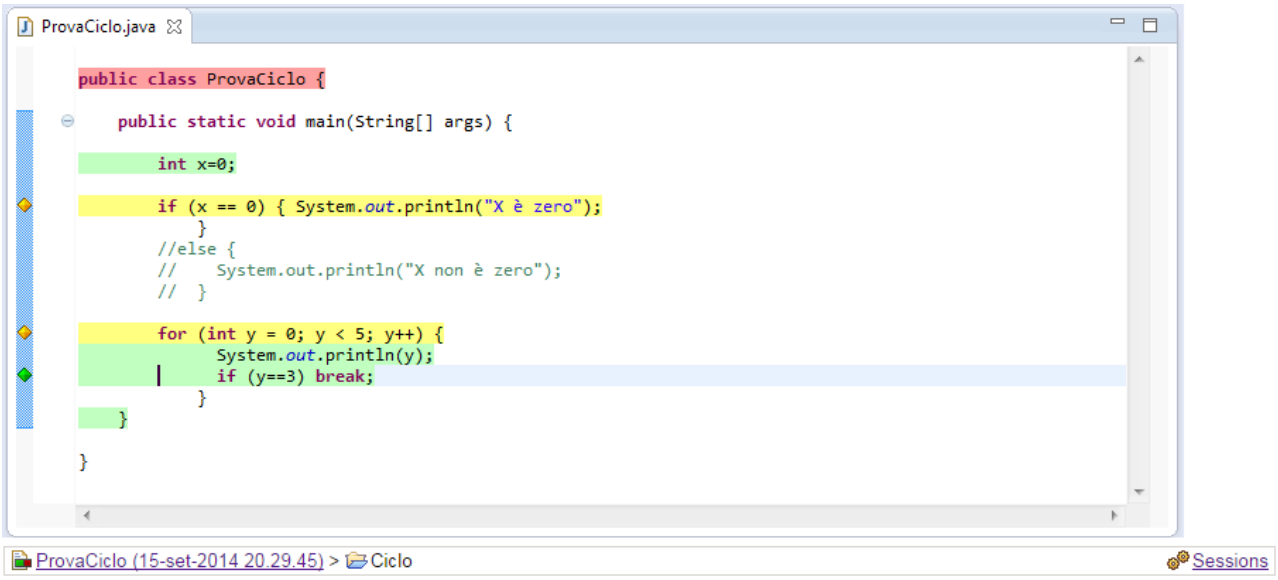
basandosi sullo stato di copertura di ciascun ramo, EclEmma calcola anche la complessità di ciascun metodo.

Per quanto riguarda la copertura delle linee, una linea sorgente si considera eseguita quando almeno una istruzione che viene assegnata a questa linea è stata eseguita.

Considerato il fatto che una singola linea compila tipicamente più istruzioni bytecode , il codice sorgente viene evidenziato in tre modi differenti per ciascuna linea contenente codice sorgente.

- Nessuna copertura: nessuna istruzione nella linea è stato eseguito (sfondo rosso)
- Copertura parziale: è stata eseguita solo una parte dell'istruzione nella linea (sfondo giallo)
- Copertura completa: tutte le istruzioni della linea sono state eseguite (sfondo verde)

Riguardo la copertura dei metodi, una funzione membro è considerata eseguita quando almeno una istruzione è stata eseguita mentre una classe è considerata eseguita quando almeno uno dei suoi metodi è stato eseguito. Di seguito, la verifica della copertura ed il report su un semplice costrutto if e un ciclo for interrotto dall'istruzione break:



Ciclo

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
src		88%		67%	3	5	1	7	1	2	0	1
Total	3 of 25	88%	2 of 6	67%	3	5	1	7	1	2	0	1

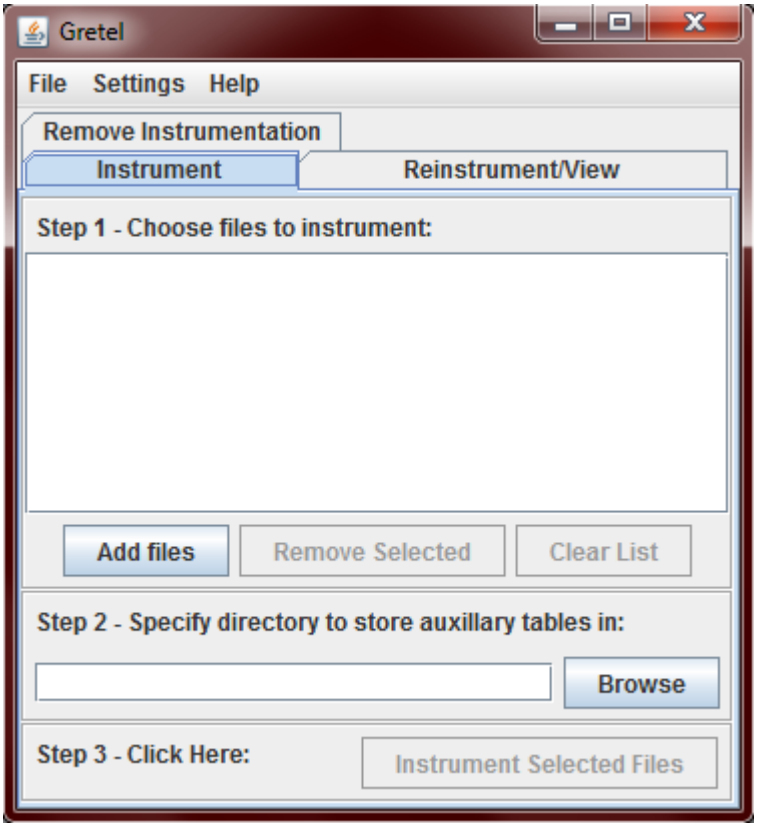
ProvaCiclo (15-set-2014 20.29.45) Created with JaCoCo 0.7.1.201405082137

2.3 Gretel

Sviluppato dall'università dell'Oregon, Gretel [5] si occupa della verifica della copertura di codice java. Esso predispone il codice per l'esecuzione (instrumentation), tramite delle sonde (le "briciole"), in pratica lavora con classi compilate alterandole (includendo cioè, gli strumenti per la verifica di quali linee di codice sono state eseguite). Si occupa inoltre del salvataggio e della visualizzazione dei dati. Fin qui Gretel è molto simile agli altri tool presenti sul mercato, la caratteristica per cui si distingue è quella della possibilità di "re-istrumentare" il codice dopo una prima parte dei test. In particolare Gretel rimuove le sonde per le parti di codice che sono già state testate e coperte. Questa caratteristica diventa

fondamentale nel momento in cui si deve lavorare con grosse moli di codice dove il calo di prestazioni dovuto al "peso" che la fase di istrumentazione comporta può diventare un problema in termini sia di tempo che di memoria occupata. È possibile ripetere questa operazione più volte, scalando in maniera incrementale le sonde.

Un'altra caratteristica che porta questo tool controcorrente è un'interfaccia molto essenziale che come mostrato tra poco, non offre molto in termini di user friendliness. Analizziamo il funzionamento. Per cominciare bisogna inizializzare le classi: ciò significa che ogni classe che sarà usata per cominciare l'esecuzione del programma deve essere "istrumentata". Per fare questo si apre il file gretel.jar avremo a disposizione un pannello che permette di scegliere quali classi utilizzare e la directory nella quale memorizzare la tabella generata.

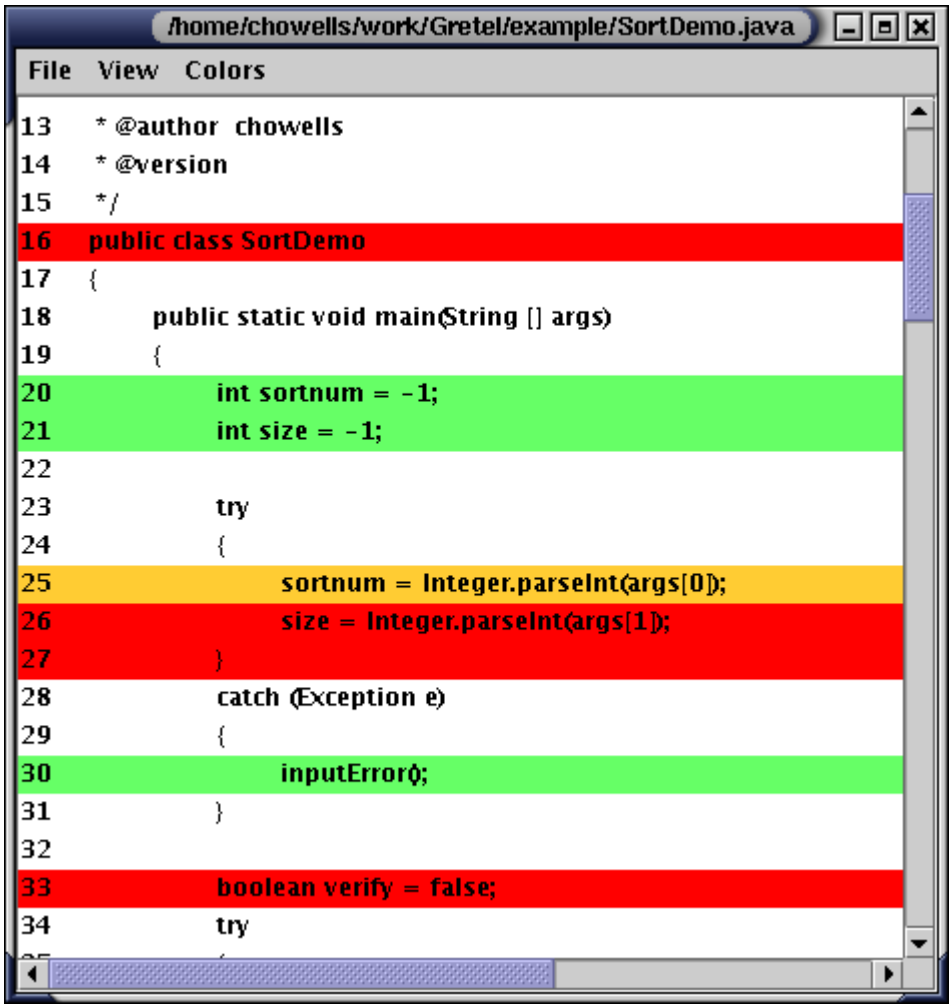


Durante l'Initial Instrumentation, Gretel aggiunge le sonde ai file classe selezionati dopodiché crea un file con estensione *.gretel* per i files classe che sono stati selezionati e

contiene un metodo main. Infine viene creato un file lineTable nella directory selezionata, che contiene le informazioni necessarie per associare il numero delle sonde, alle sezioni di codice. In sostanza serve ad interpretare i risultati.

Per verificare i risultati bisogna andare nella scheda Reinstrument/View si seleziona il file *.gretel* per avere una lista dei files strumentati. Si seleziona uno di questi files e si clicca sul bottone View Source.

Il file di origine verrà mostrato con molte linee evidenziate in colori differenti, rosso verde o arancio.



Come accennato prima in Gretel c'è la possibilità di re-istrumentare il codice dopo una

prima parte dei test scalando in maniera incrementale le sonde fino ad ottenere la copertura desiderata. Una volta raggiunto l'obbiettivo, per rimuovere tutte le sonde da un file, in modo che possa essere eseguito senza il file gretel-runtime.jar nel classpath, si fa partire Gretel e si va nella tab "Remove Instrumentation" tab, si seleziona il (oppure i) file(s) da cui si vuole rimuovere le sonde ed infine si schiaccia il bottone "remove Instrumentation".

Purtroppo con questo strumento non è possibile ottenere un report di copertura e non offre nessun tipo di possibilità di integrazione con il framework JUnit.

Non ci sono particolari convenzioni sul conteggio delle righe di codice dal momento che non è possibile ottenere un report di copertura, come precedentemente detto, le linee di codice sorgente saranno evidenziate in colori differenti, ogni colore indica il suo stato di esecuzione: di default il verde indica che il codice è stato eseguito arancio eseguito parzialmente, rosso il codice non è stato eseguito le linee non evidenziate, infine, indicano che non c'è codice istrumentato associato a quella linea.

2.4 Cobertura

Cobertura [6] è uno strumento open source che permette di calcolare la percentuale di codice coperto dai test, pertanto viene adoperato per identificare parti del programma che mancano della copertura. È in grado di calcolare la complessità ciclomatica di McCabe di ogni classe e la complessità ciclomatica per ogni package e per tutto il progetto.

E' in grado inoltre di mostrare la percentuale di copertura, di istruzioni e decisioni, per ogni classe, package o dell'intero progetto, aiuta quindi ad avere una visione gerarchica a diversi livelli di dettaglio. Una volta compilato cobertura utilizza il bytecode di java, quindi, agisce al secondo livello di profondità, di conseguenza ha un buon vantaggio in termini di velocità. Una caratteristica che invece si traduce in uno svantaggio è che eseguendo un check statico, viene falsato in presenza di elementi istanziati od utilizzati dinamicamente. Può generare report in XML e HTML.

Cobertura può essere eseguito da riga di comando e via Ant o Maven.

Per quanto riguarda l'esecuzione da riga di comando, prima di tutto c'è la fase di istrumentazione, con il comando

cobertura-instrument.bat [--basedir dir] [--datafile file] [--auxClasspath classPath] [--destination dir] [--ignore regex] classes [...]

dove i flag sono tutti opzionali e riguardano:

basedir specifica la directory nel caso si vuole strumentare classi che si trovano in diverse directory

datafile specificare nome del file che sarà utilizzato per memorizzare i metadati delle classi (nomi delle classi nomi dei metodi ecc)

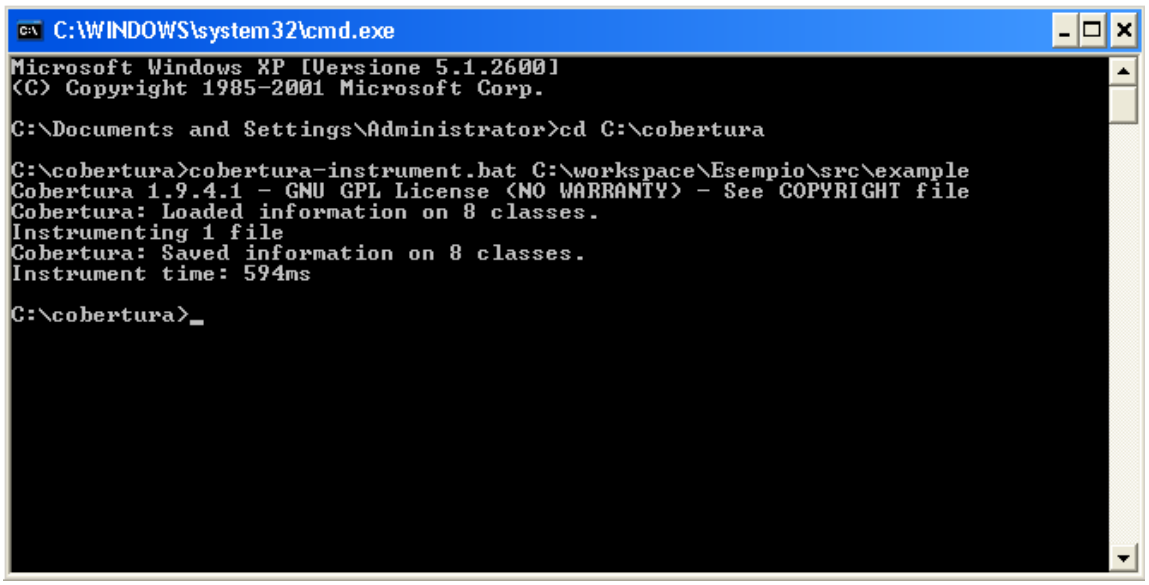
destination directory per l'output delle classi strumentate (se non specificato le classi originali saranno sostituite da quelle strumentate)

ignore specifica una espressione per filtrare certe linee di codice sorgente

auxClasspath aggiunge classi che Cobertura non riesce a trovare durante la fase di intrumentation

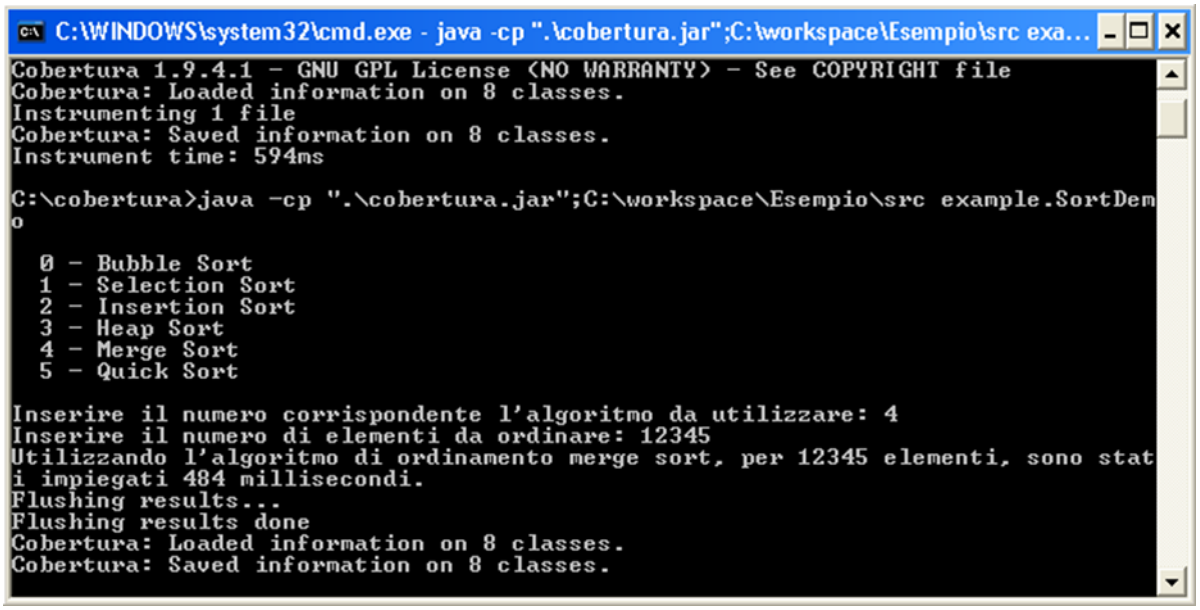
esempio:

cobertura-instrument.bat C:\MyProject\build\classes



dopodiché viene eseguito il programma normalmente, avendo cura di inserire nel classpath

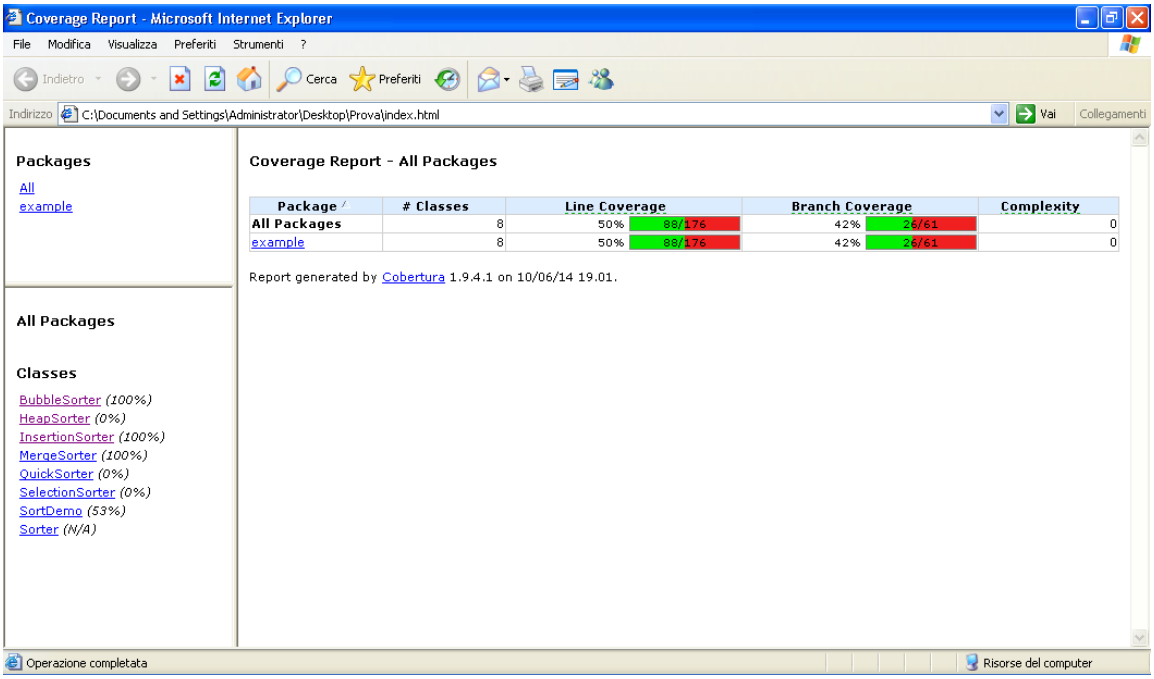
la libreria *cobertura.jar*



infine si può ottenere un file di report in formato html o xml attraverso il comando: *cobertura-report.bat* ***[--datafile file] [--destination dir] [--format (html|xml)]***

dove l'unico flag non opzionale è --destination ossia la directory dove memorizzare il report.

Di seguito un esempio di report in formato html.



È possibile inoltre sapere quale classe non ha soddisfatto i criteri di copertura previsti attraverso il comando

cobertura-check.bat [--datafile file] [--branch 0..100] [--line 0..100] [--totalbranch 0..100] [--totalline 0..100] [--regex regex:branchrate:linerate]

Dove:

branch	specifica il minimo tasso di copertura accettabile per ogni classe. Dev'essere un intero compreso tra 0 e 100
packagebranch	specifica il minimo tasso di copertura accettabile per ogni package e dev'essere un intero compreso tra 0 e 100
totalbranch	specifica il minimo tasso di copertura accettabile per l'intero progetto. Dev'essere un intero compreso tra 0 e 100

2.4.1 Interoperabilità con JUnit

La situazione per quanto riguarda il tool Cobertura, è del tutto simile a quella di Emma: il tool non offre una integrazione per la copertura del codice di casi di test JUnit, tuttavia nella roadmap è previsto un miglioramento da questo punto di vista in un prossimo futuro.

2.4.1 Convenzioni sul conteggio

Come spiegato in precedenza Cobertura offre nei suoi reports oltre alla misura della complessità ciclomatica, la copertura delle linee e la copertura dei rami.

Per quanto concerne la copertura delle linee, ogni riga che non viene saltata da un'eventuale istruzione è considerata coperta.

Riguardo invece la copertura dei rami, la documentazione di Cobertura offre degli esempi pratici per chiarire i casi in cui un branch risulta coperto in alcune istruzioni. Un branch è considerato completamente coperto nel caso in cui tutte le sue linee sono state eseguite, ad esempio nel caso di un if, l'unico modo di vedere l'intero ramo coperto è quello di eseguirlo più volte coprendo tutte le condizioni ed eccezioni.

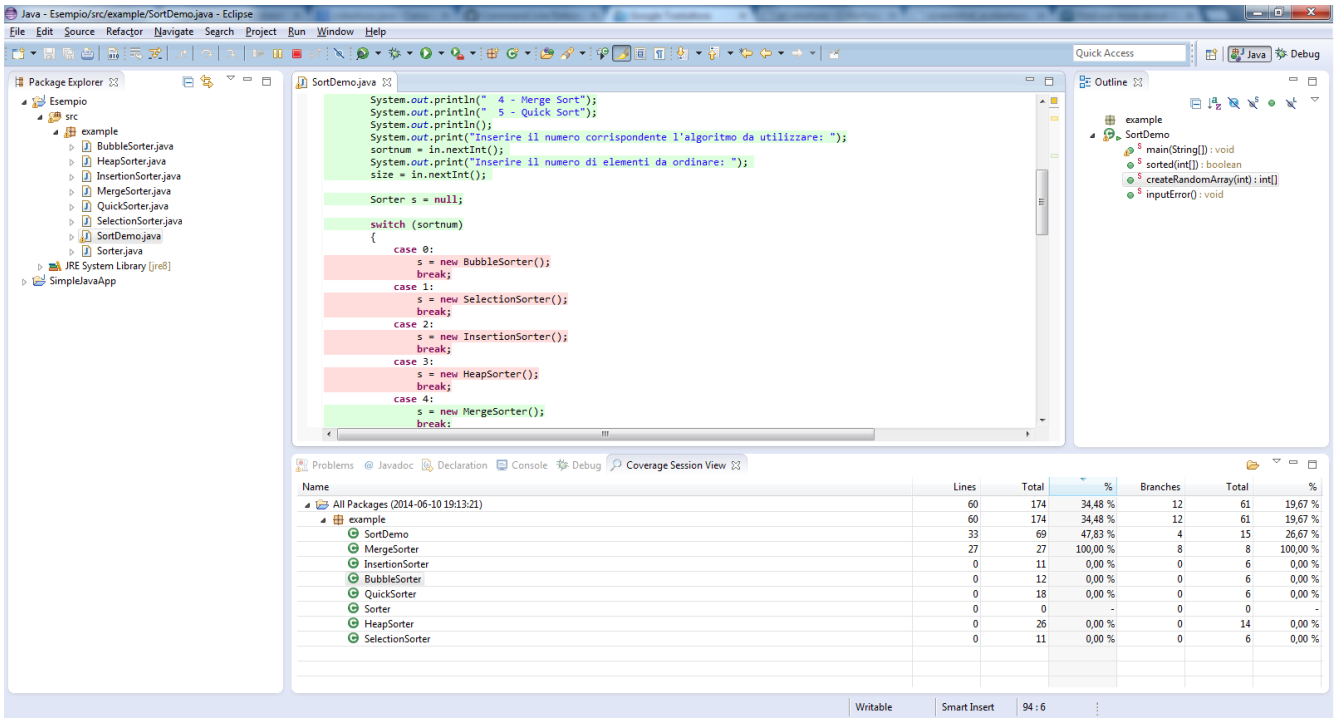
2.5 eCobertura

eCobertura [7] è un plugin opensource basato sul funzionamento di Cobertura per l'IDE Eclipse. Questo plugin consente di avviare le applicazioni, **o test** in modalità verifica di copertura (di Cobertura) direttamente tramite Eclipse. Grazie a eCobertura sarà possibile visualizzare i files sorgenti con le righe di codice colorate in maniera differente a seconda dello stato di copertura; è possibile applicare dei filtri in modo da verificare la copertura solo di alcune parti del programma alla volta.

Una volta lanciato il programma in modalità copertura, eCobertura copia tutte le classi dalla cartella di output nel suo workspace interno. Qui istrumenta le classi non filtrate dalle impostazioni di copertura. All'interno del classpath di esecuzione, la cartella di output sarà sostituita dal workspace interno del plugin in modo da far utilizzare le classi istrumentate invece di quelle originale al momento dell'esecuzione. In pratica eCobertura non effettua una istrumentazione in-place.

eCobertura salverà un file cobertura.ser al termine del processo lanciato. Questo file sarà interpretato dal plugin mostrando i risultati nella tab Coverage Session View. Sebbene il

funzionamento di eCobertura sia basato sul tool Cobertura, al contrario di quest'ultimo, NON è possibile effettuare report di nessun tipo.



2.5.1 Interoperabilità con JUnit

Il plugin eclipse eCobertura è in grado di eseguire la strumentazione delle classi di test e fornire a video il codice sorgente con evidenziate le linee di codice eseguite o non eseguite. Come visto precedentemente però, eCobertura soffre della pesante limitazione di non poter effettuare nessun tipo di report, cosa che in ultima analisi, lo rende decisamente inferiore al comparato eclEmma.

Non ci sono particolari convenzioni sul conteggio delle righe di codice dal momento che non è possibile ottenere un report di copertura, le linee di codice sorgente saranno evidenziate in colori differenti, verde codice eseguito, rosso non eseguito.

2.6 CodeCover

CodeCover [8] è un altro strumento utilissimo per verificare la copertura del codice e dispone di due possibili modalità di esecuzione, standalone e come plugin di eclipse.

Il funzionamento risulta diverso rispetto ai tools visti finora, infatti, mentre i precedenti

istrumentavano le classi già compilate, CodeCover nella fase di istrumentazione lavora direttamente sul codice sorgente, strumentandolo in una directory separata, che solo successivamente sarà compilato ed eseguito.

Vediamo il funzionamento per quanto riguarda l'utilizzo a riga di comando. Una volta scaricato il file codecover.bat quindi si dà il comando:

```
codecover instrument        --root-directory [source/directory]
                            --destination [destination/instrd/directory]
--container [dir/test-session-container.xml]
                            --language [java]
```

Dove

--*root-directory*	indica il percorso dove si trovano i files sorgenti del progetto che vogliamo istrumentare
--*destination*	indica il percorso dove saranno salvati i files sorgenti strumentati
--*container*	crea un file xml nel quale sono memorizzate le informazioni statiche del codice strumentato e quelle riguardanti i dati di copertura
--*language*	indica il linguaggio del codice che dev'essere strumentato

Con il comando sopra descritto si avvierà il processo di istrumentazione che genera una serie di nuovi file sorgenti, istrumentati, salvati nella directory specificata.

A questo punto bisognerà compilare i files istrumentati dopodiché si può procedere alla vera e propria verifica della copertura attraverso il comando

```
codecover analyze       --container dir/test-session-container.xml
                        --coverage-log  /instrd/directory/coverage_log.clf
                        --name TestSession1
                        --comment "The first test session"
```

Dove

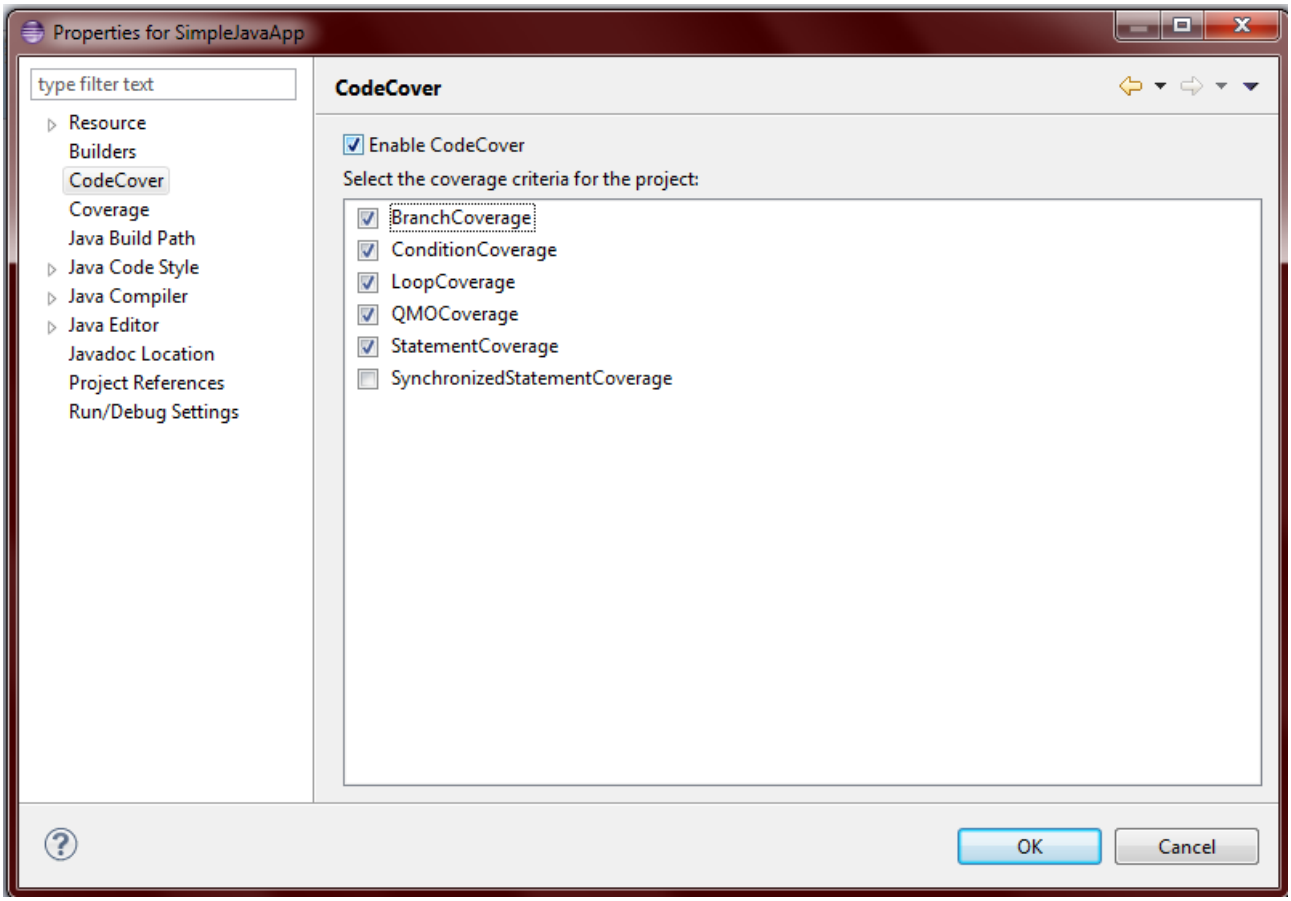
--container	identifica il "contenitore" definito durante la fase di instrumentation
--coverage-log	fa riferimento al file log di copertura dati
--name	fa riferimento al nome della sessione
--comment	si riferisce ad un commento opzionale di che può essere aggiunto a una sessione di test

Infine è possibile ottenere un report attraverso il comando

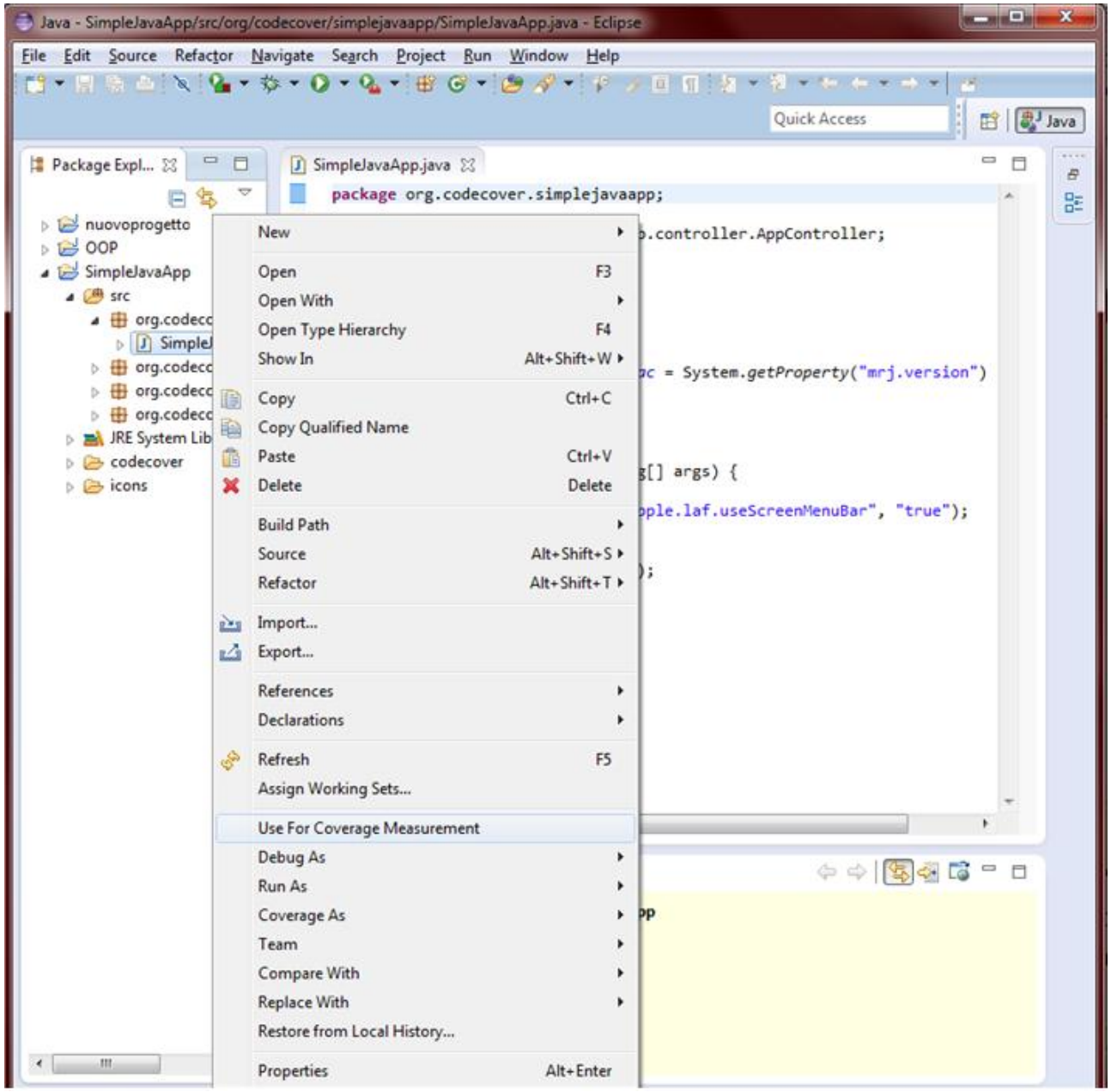
codecover report --container *SimpleJavaApp*/test-session-container.xml
--destination *SimpleJavaApp*/report/*SimpleJavaApp*Report.html
--session "TestSession1+2"
--template CODECOVER_HOME/report-templates/HTML_Report_hierarchic.xml

Codecover è disponibile in via ufficiale, anche come plug-in dell'ambiente Eclipse, molto più semplice da utilizzare e perfettamente in grado di compiere tutte le operazioni eseguibili dalla riga di comando al contrario ad esempio di Cobertura ed eCobertura. Per utilizzarlo è necessaria l'installazione utilizzando la voce nel menu "help" di eclipse e successivamente la voce install new software inserendo nell'apposito campo il link del sito **http://update.codecover.org/**

Una volta eseguita correttamente l'installazione per utilizzare il plugin bisogna abilitarlo nelle proprietà del progetto di nostro interesse selezionando inoltre i criteri di copertura da utilizzare nella fase di istrumentazione:

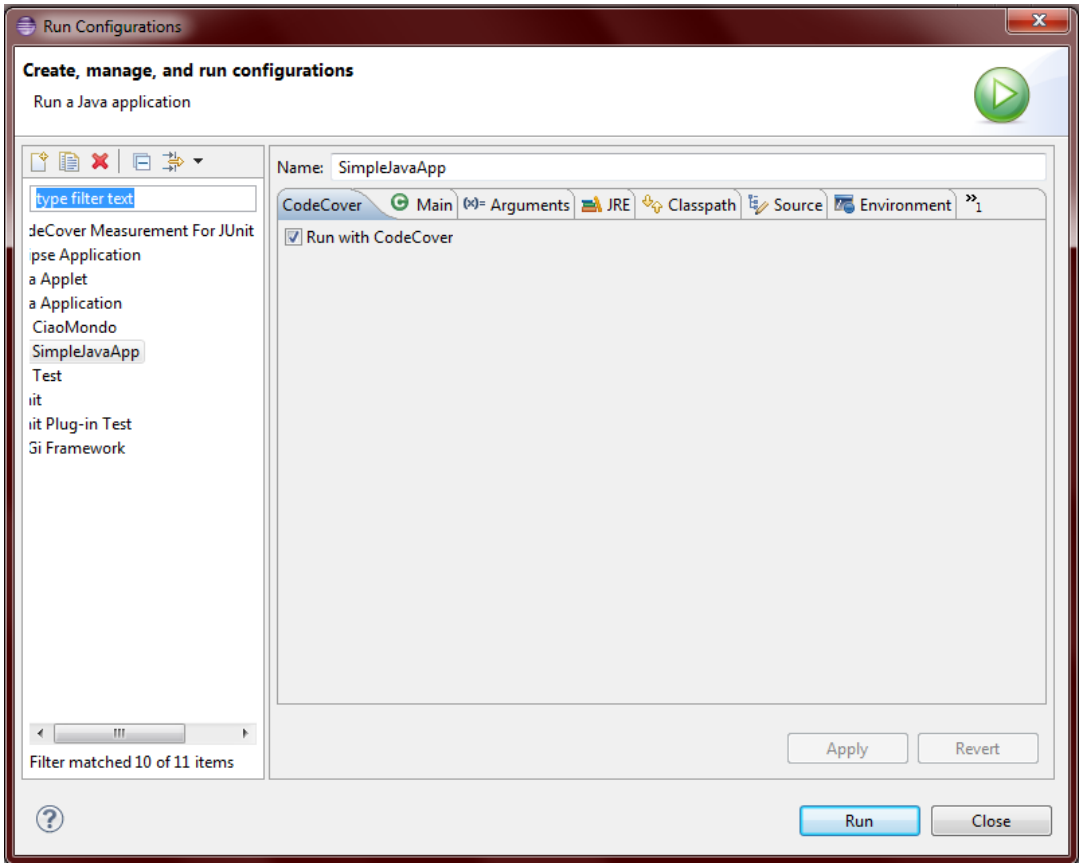


Successivamente si selezionano le classi che si vogliono strumentare per farlo basta un click col tasto destro del mouse sulla classe e si clicca sulla voce “Use For Coverage Measurement”

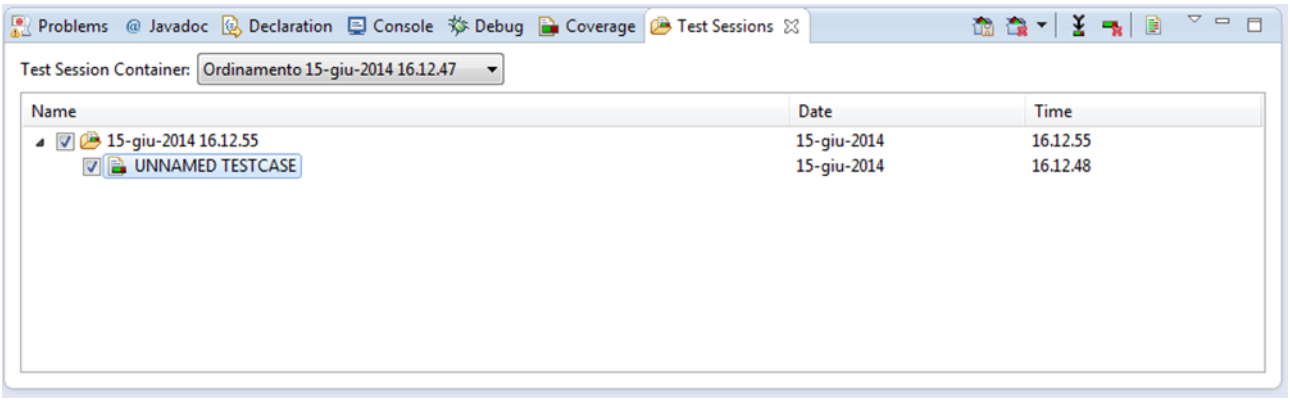


A questo punto bisogna forzare Eclipse ad usare il plugin CodeCover, per farlo basta cliccare “run as” – “Run configuration” e spuntare la casella “Run with CodeCover”:

quindi “Run”

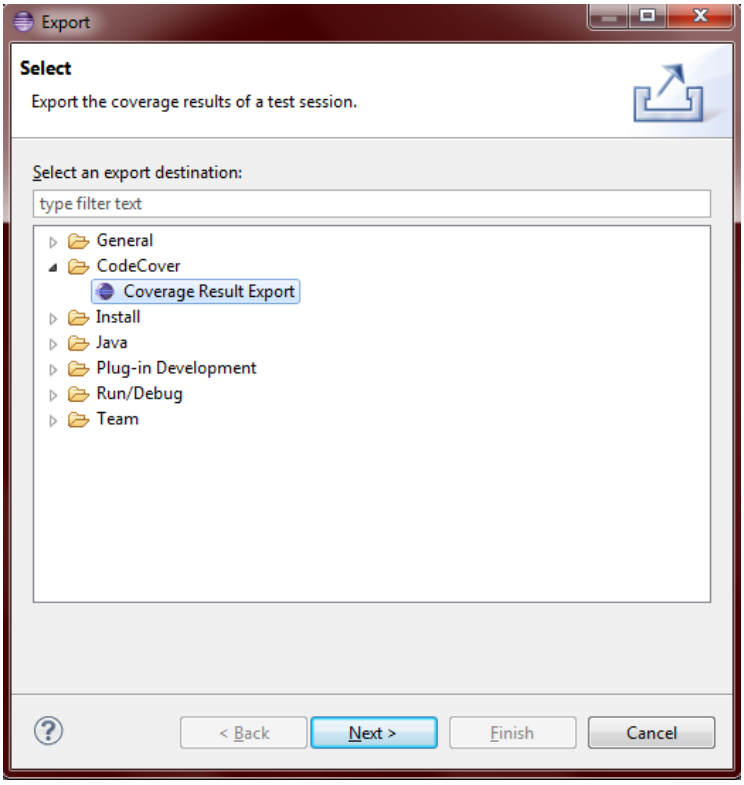


il progetto verrà eseguito regolarmente e alla fine sarà possibile visualizzare le sessioni di test direttamente in eclipse in un’apposita colonna in basso



dove rinominare, selezionare, deselezionare e fare il merge delle diverse sessioni.

È possibile infine ottenere un report in formato HTML, CSV o XML utilizzando il comando Export dal menu “File” di Eclipse, quindi selezionando “Coverage Result Export”



2.6.1 Interoperabilità con JUnit

CodeCover, supporta ufficialmente la misura della copertura del codice per i test JUnit e l'utilizzo di tale funzionalità attraverso il plugin di eclipse è davvero molto semplice.

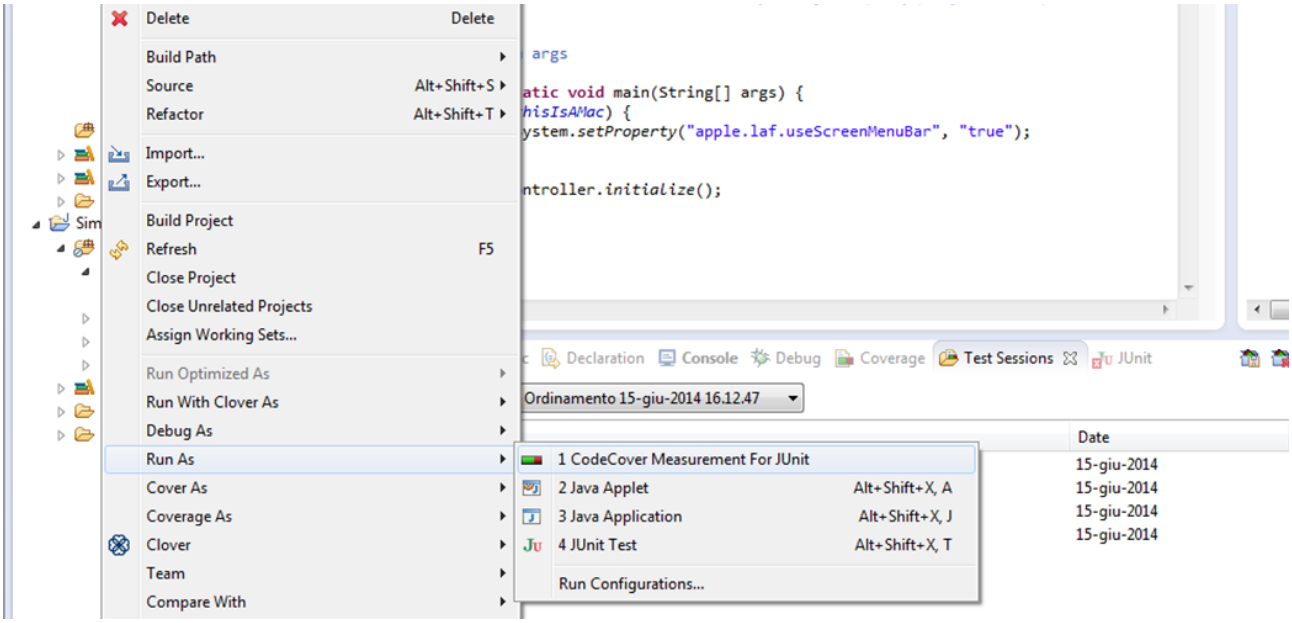
Basta infatti, dopo aver abilitato CodeCover per il progetto di interesse, andare su *Run As* > *CodeCover Measurement for JUnit*

Riguardo invece la verifica della copertura di test case JUnit tramite riga di comando, non c'è la necessità di strumentare i casi di test ma basta utilizzare i *TestRunners* forniti nel file jar *JUnit-TestRunner.jar*

Questo è possibile utilizzando il comando

java -cp junit.jar:JUnit-TestRunner.jar:bin org.codecover.junit.swing.TestRunner [-methodsAsTestCases] de.myprogram.AllTests

dove *AllTests* è una suite di test JUnit



2.6.2 Convenzioni sul conteggio

Per quanto riguarda la copertura delle istruzioni, statement coverage, questa è misurata soltanto per le istruzioni:

- Assegnazioni e chiamate di un metodo (a=b oppure system.out.println(person.getName)
- break
- continue
- return
- throw
- dichiarazioni vuote
- dichiarazioni di variabili con assegnazione (int x = 1)

Di conseguenza, tutte le istruzioni condizionali (per esempio *if*) e le dichiarazioni di ciclo (*while, for*…) non sono considerati per la copertura delle istruzioni.

Secondo il criterio di copertura di CodeCover un'istruzione è considerata coperta se è stata eseguita, e il program flow, procede all'istruzione successiva.

Copertura dei rami: i branches sono creati da

- if e else
- switch: case e default

- *try* crea un ramo per ciascun *catch*

Un branch è considerato coperto se è stato eseguito almeno una volta.

CodeCover, non considera il corpo di un ciclo come un ramo. Inoltre i rami, creati da possibili eccezioni, ad esempio un NullPointerException, vengono anch'essi ignorati.

Per quanto riguarda la copertura delle condizioni, sono considerati i valori booleani all'interno delle seguenti istruzioni

- if
- for
- while
- do .. while

Le espressioni booleane nelle assegnazioni, i parametri dei metodi e gli operatori ternari (a ? b : c) vengono ignorati.

Il criterio di copertura definisce un valore booleano di base come coperto, se è stato valutato sia vero che falso, e il cambiamento da vero a falso o viceversa cambia il risultato di tutta la condizione, quando tutti gli altri valori booleani di base rimangono costanti o non sono valutati.

La copertura dei cicli viene misurata per le seguenti istruzioni:

- for , anche se ha un numero fisso di esecuzioni
- while
- do .. while

Ci sono tre possibili scenari, il corpo del ciclo non è estato eseguito, eseguito una volta e non ripetuto e infine il corpo del ciclo è ripetuto più di una volta.

2.7 Clover

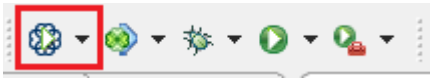
Clover [9] è l'unico tool di quelli analizzati, che non ha una licenza opensource tuttavia è possibile provare una versione di valutazione per 30 giorni.

Clover è degno di nota tra i vari strumenti utilizzati per il code coverage java per la sua estrema velocità e semplicità di utilizzo come plugin per Eclipse.

L'installazione può avvenire attraverso la ricerca dal menu "help" di eclipse quindi install new software inserendo nell'apposito campo il link del sito

http://update.atlassian.com/eclipse/clover

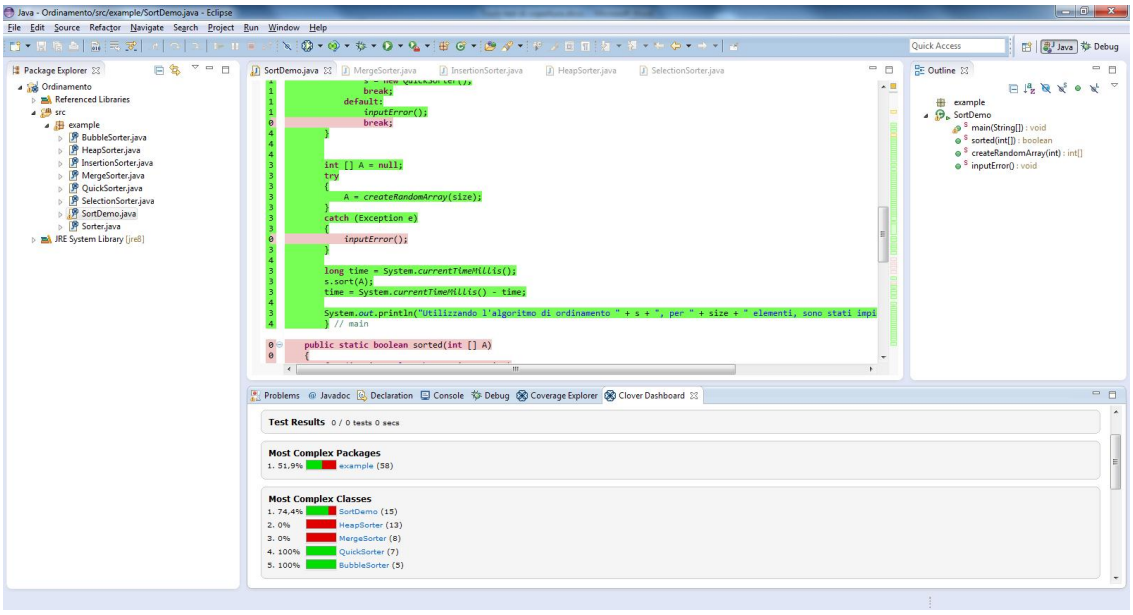
una volta installato, per verificare la copertura del codice del progetto bisogna compilarlo con l'apposito tasto che si aggiunge alla barra di eclipse



A questo punto basta inserire la vista di clover a quelle già presenti in eclipse per avere accesso alla miriade di opzioni disponibili, tra cui:

- **Clover Dashbord:** è una tab dove si può avere una panoramica del progetto (percentuali di copertura, metodi meno testati ecc)

- **Java editor :** le linee di codice dei files sorgenti vengono colorate a seconda se sono state eseguite o meno

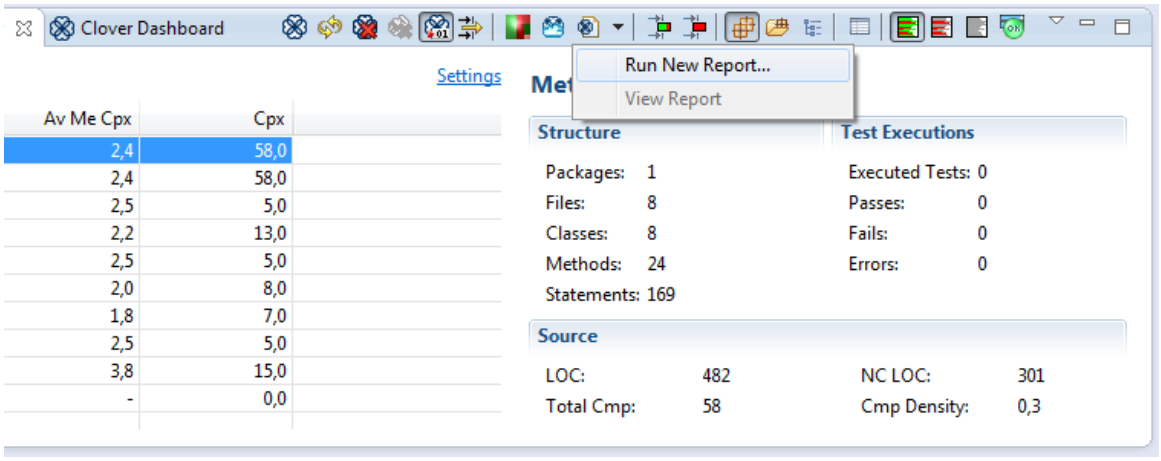


- **Coverage Treemap Report**: rapporto di copertura treemap che permette il confronto simultaneo di classi e package per complessità e per la copertura del codice. La treemap è divisa dal pacchetto (con etichetta), e poi ulteriormente suddivisa per classe (etichettate). La

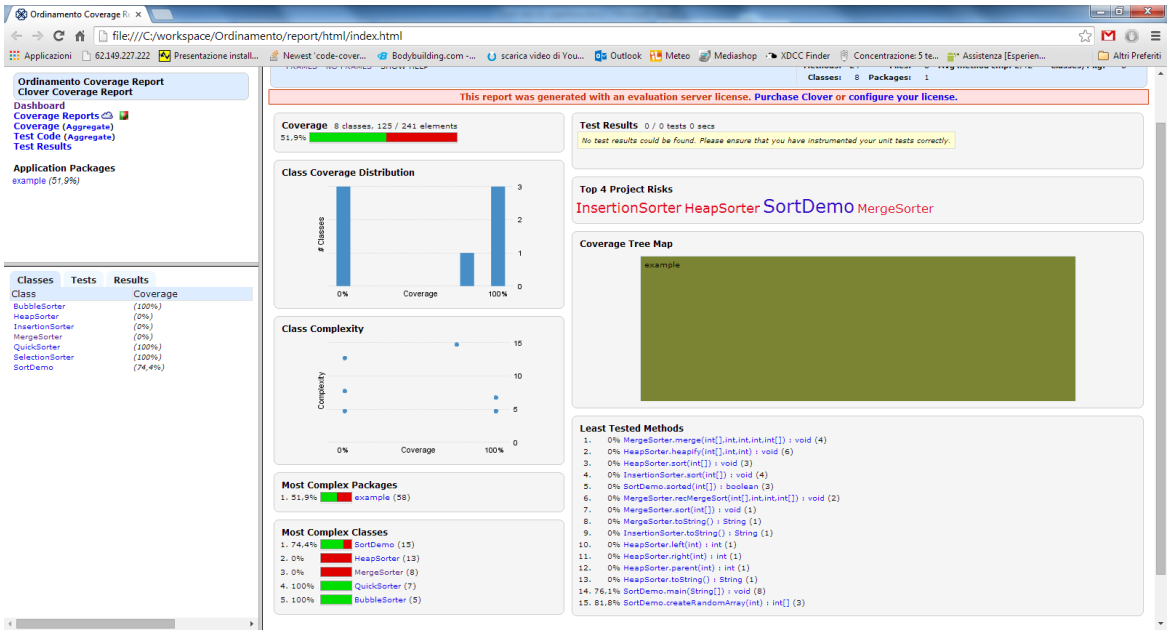
dimensione del pacchetto o una classe indica la sua complessità (quadri più grandi indicano una maggiore complessità). I colori indicano il livello di copertura (verde brillante - più coperto, rosso vivo - scoperto).

Con il tool Clover è possibile fare un merge di diverse sessioni di test ed ottenere reports in diversi formati, html, pdf ed xml.

Per ottenere un report di copertura basta cliccare sull'apposito bottone, selezionare il tipo di formato e seguire le varie impostazioni.



Di seguito un esempio di report in formato Html



2.7.1 Interoperabilità con JUnit

Clover infine, che ricordo è l'unico tool non disponibile in licenza open source, supporta una completa integrazione con JUnit.

Si nota che a partire dalla versione 4.0, JUnit ha introdotto una funzionalità che permette di eseguire lo stesso test più volte, utilizzando dati diversi come input.

Per fare questo è necessario

- contrassegnare la classe test con *@RunWith(Parameterized.class)*
- dichiarare un metodo *data()* che restituisce la collezione dei valori di input e contrassegnare questo metodo con l'annotazione *@Parameters*
- Infine dichiarare un metodo test contrassegnato da *@Test*

Per esempio:

```
@RunWith(Parameterized.class)
public class PersonTest {
    @Parameterized.Parameters(name = "{0} is a {1} [{index}]")
    public static Collection<Object[]> data() {
        return Arrays.asList(new Object[][]{
                {"Alice", "woman"}, {"Bob", "man"}, {"Rex", "unknown"}
        });
    }
    protected String input;
    protected String expected;
    public PersonTest(String input, String expected) {
        this.input = input;
        this.expected = expected;
    }
    @Test
    public void test() {
        assertEquals(expected, new Person(input).getSex());
    }
}
```

Dato che questi test sono eseguiti dal test runner di JUnit, Clover può registrare senza problemi i risultati dei test, tuttavia (almeno fino alla versione 3.3.0) non era possibile avere alcuna informazione riguardante quale iterazione del test era fallita ma tutte le iterazioni davano risultato negativo.

Dalla versione 3.3.0 Clover introduce un JUnitTestRunnerInterceptor che può essere collegato al Runner di JUnit. In questo modo oltre a registrare i risultati di coverage, "ascolta" quali test sono stati eseguiti con successo ed i loro "nomi" valutati da JUnit.

Grazie a questo è possibile ottenere nei reports un numero di iterazione,

Tests	Started	Status	Time (secs)	Message
SquareTest.test[0](SquareTest)	10 mar 12:06:32	PASS	0	
SquareTest.test[1](SquareTest)	10 mar 12:06:32	FAIL	0,001	expected:<1> but was:<2>
SquareTest.test[2](SquareTest)	10 mar 12:06:32	PASS	0	
SquareTest.test[3](SquareTest)	10 mar 12:06:32	FAIL	0,002	expected:<9> but was:<6>
SquareTest.test[4](SquareTest)	10 mar 12:06:32	FAIL	0,001	expected:<16> but was:<8>
SquareTest.test[5](SquareTest)	10 mar 12:06:32	FAIL	0	expected:<25> but was:<10>

così come anche i nomi stabiliti nella classe test (*@Parameters(name=..)*)

Class	Tests	Fail	Error	Time (secs)	% Tests Success
PersonTest	3	1	0	0,006	66,7%

Tests	Started	Status	Time (secs)	Message
PersonTest.test[Alice is a woman [0]](PersonTest)	10 mar 12:06:32	PASS	0,005	
PersonTest.test[Bob is a man [1]](PersonTest)	10 mar 12:06:32	PASS	0	
PersonTest.test[Rex is a unknown [2]](PersonTest)	10 mar 12:06:32	FAIL	0,001	expected:<[unknow]n> but was:<[ma]n>

2.7.2 Convenzioni sul conteggio

Clover misura tre metriche fondamentali nell'analisi della copertura

Copertura delle istruzioni, un istruzione è considerata coperta quando eseguita almeno una volta.

Copertura dei rami, o copertura delle decisioni, misura quali dei possibili rami sono eseguiti durante l'esecuzione. Per fare questo, Clover "registra" se le espressioni booleane nella struttura di controllo sono valutate sia vere che false.

Per copertura dei metodi, si intende una metrica che misura se un metodo è stato eseguito per intero durante l'esecuzione.

La percentuale "totale" di copertura di una classe o di un file o di un pacchetto, segue la formula

TPC = (CT + CF + SC + MC) / (2 * C + S + M)

Dove

CT condizioni che sono state valutate "true" almeno una volta

CF condizioni che sono state valutate "false" almeno una volta

ST istruzioni eseguite (coperte)

MC metodi inseriti

C numero totale delle condizioni

S numero totale delle istruzioni

M numero totale delle funzioni membro

Capitolo 3: Strumenti per la copertura del codice JavaScript

JavaScript è uno dei linguaggi di programmazione più usati al mondo, anche se con una storia di alti e bassi, la sua consolidata presenza nell'olimpo dei grandi linguaggi come C, C++ e Java è certa. Come si può constatare dalla tabella fornita dal TIOBE index [10] per Luglio 2014, l'uso di questo linguaggio è in ascesa.

L'enorme diffusione di JavaScript è dovuta principalmente al fiorire di numerose librerie nate allo scopo di semplificare la programmazione sul browser, ma anche alla nascita di framework lato server e nel mondo mobile che lo supportano come linguaggio principale.

Molti sviluppatori JavaScript hanno prima imparato a conoscere le varie librerie come jQuery, MooTools, Prototype ed altre e la loro effettiva conoscenza del linguaggio spesso è rimasta molto limitata.

Altri utenti JavaScript provengono da esperienze di sviluppo in altri linguaggi di programmazione e molto spesso questo genera confusione o sottovalutazione delle potenzialità del linguaggio di scripting. Ad esempio, diversi sviluppatori Java, C++ o C#, trasferiscono più o meno inconsciamente le caratteristiche del loro linguaggio a JavaScript, vista la somiglianza sintattica, ma questo talvolta è fonte di errori e di una non corretta valutazione delle sue effettive capacità.

Infine c'è da considerare che non tutti gli utenti di JavaScript sono veri e propri sviluppatori. Molti sono utenti che in un modo o nell'altro si sono trovati a dover integrare

delle pagine Web con JavaScript, spesso partendo con un copia e incolla di blocchi di codice trovati in giro per Internet.

La diffusione di JavaScript non è necessariamente indice della sua effettiva conoscenza per questo motivo diventa fondamentale una fase di testing utilizzando anche tools per la copertura del codice.

Purtroppo in questo ambito non ci sono grosse scelte nella selezione di un tool che verifichi il code coverage su progetti javascript, di seguito se ne descrivono due in particolare.

2.7 JSCover

JSCover [11] è uno strumento che consente di misurare la copertura del codice per i programmi JavaScript ed è l'evoluzione dell'ormai abbandonato JSCoverage. JSCover lavora aggiungendo la fase di istrumentazione al codice JavaScript prima di essere eseguito in un browser web.

Per fare ciò il tool fornisce diverse alternative:

- Il metodo più semplice è quello di utilizzare la modalità server, ossia un semplice server web che istrumenta il codice JavaScript appena è eseguito
- Un altro metodo è quello di usare la modalità file-system per generare dei files JavaScript strumentati
- Infine si può usare la modalità server con l'opzione --proxy per agire come un server proxy che strumenta tutto il codice JavaScript inoltrato attraverso lo stesso

La modalità server (sia web che proxy) ha due importanti vantaggi:

1. Può memorizzare i report di copertura nel filesystem
2. Può includere nei report quale parte del codice JavaScript non è stato caricato/testato

JSCover è racchiuso in due pacchetti JARs:

-JSCover-all.jar che include tutte le dipendenze

-JSCover.jar per il quale tutti i jar dipendenti devono essere inclusi nella stessa directory

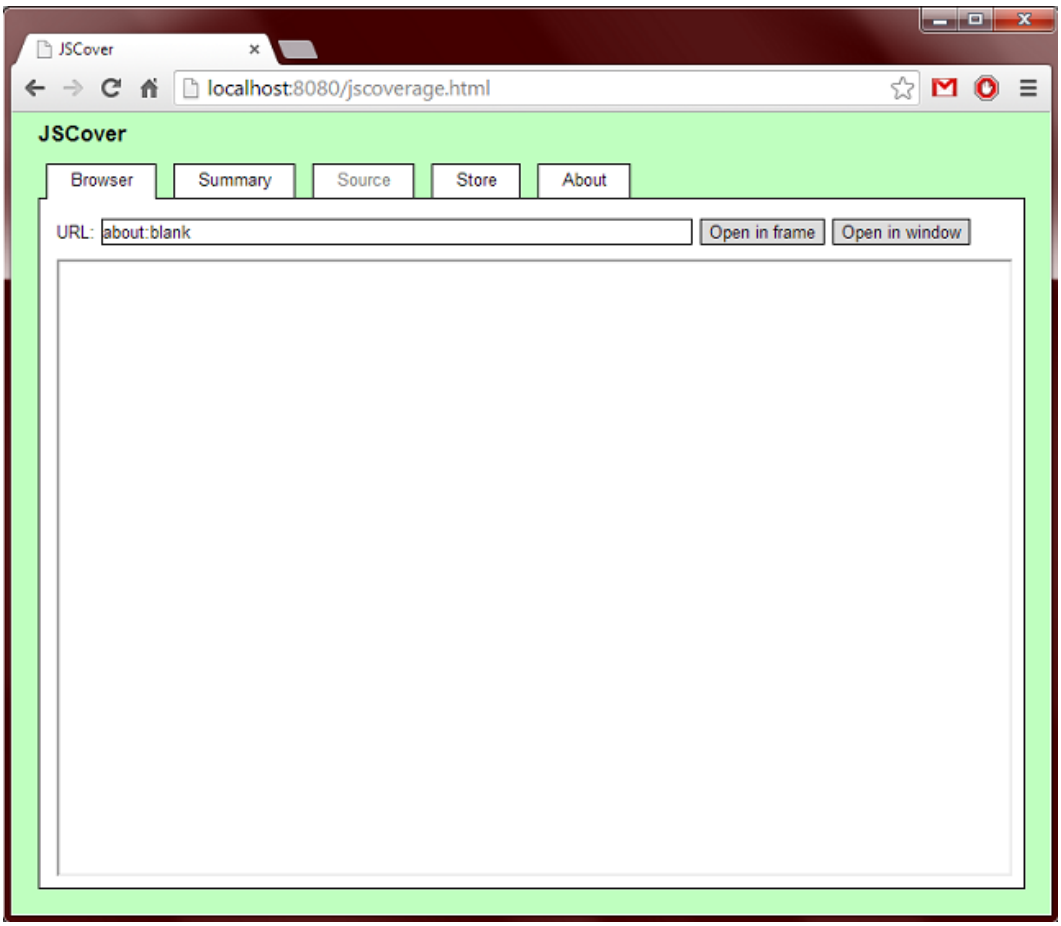
Per mostrare il funzionamento utilizziamo degli examples disponibili insieme al download dell'ultima release di JSCover.

La modalità che andremo ad utilizzare è quella server: per cominciare si avvia il file *jscover-server.bat*

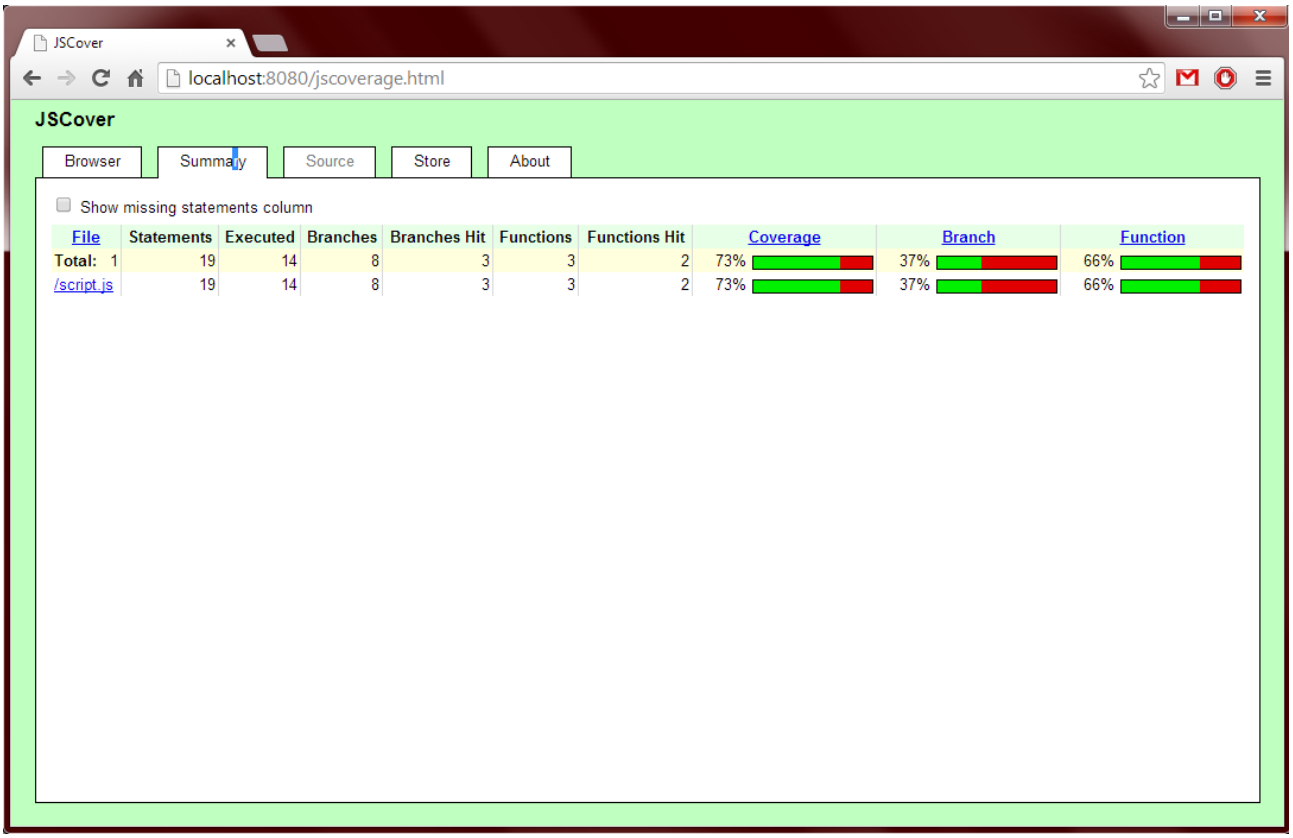
questo file batch include il comando

java -jar target/dist/JSCover-all.jar -ws --document-root=doc/example --report-dir=target

che avvierà un webserver, l'interfaccia di JSCover potrà essere caricata all'indirizzo http://localhost:8080/jscoverage.html



Inserendo l'indirizzo *http://localhost:8080/index.html* nella barra URL: e cliccando su "Open in frame" si avvierà il programma d'esempio che permette la scelta di un numero tra uno e quattro. Una volta effettuata la scelta lo script termina e per visualizzare i dati di copertura basterà cliccare sulla tab "Summary"



Nella quale sarà possibile apprezzare le percentuali e i numeri di copertura e precisamente, numero di dichiarazioni totali ed eseguite numero di branches totali e coperti numero di funzioni totali e coperte oltre alle tre percentuali.

Nella tab Source è possibile visualizzare il codice sorgente con le linee evidenziate verde o rosso a seconda che siano state o meno eseguite.

È possibile salvare il report appena ottenuto in un file html utilizzando il bottone "save report" nella tab Store (notare che ci sono di problemi di visualizzazione con il browser Google Chrome, il quale visualizza tutti i report come non testati; nessun problema con altri browsers).

Una volta salvato il report sarà possibile fermare il web server utilizzando il bottone Stop

Server presente sempre nella tab Store.

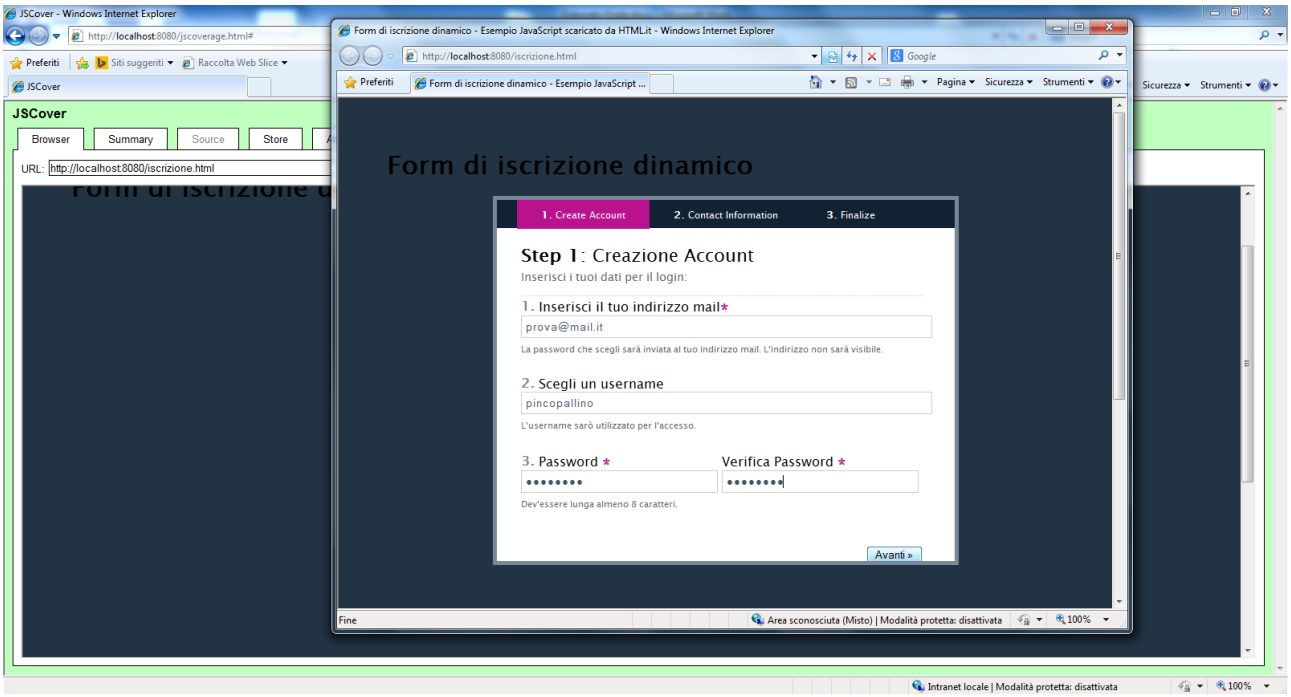
Riassumendo, grazie a JSCover è possibile avere la copertura di linee, funzioni e rami; fare un merge di diverse sessioni di test dato che i test di copertura sono raccolti in-browser; salvare i reports su filesystem; ottenere report in diversi formati; infine, è in costante aggiornamento.

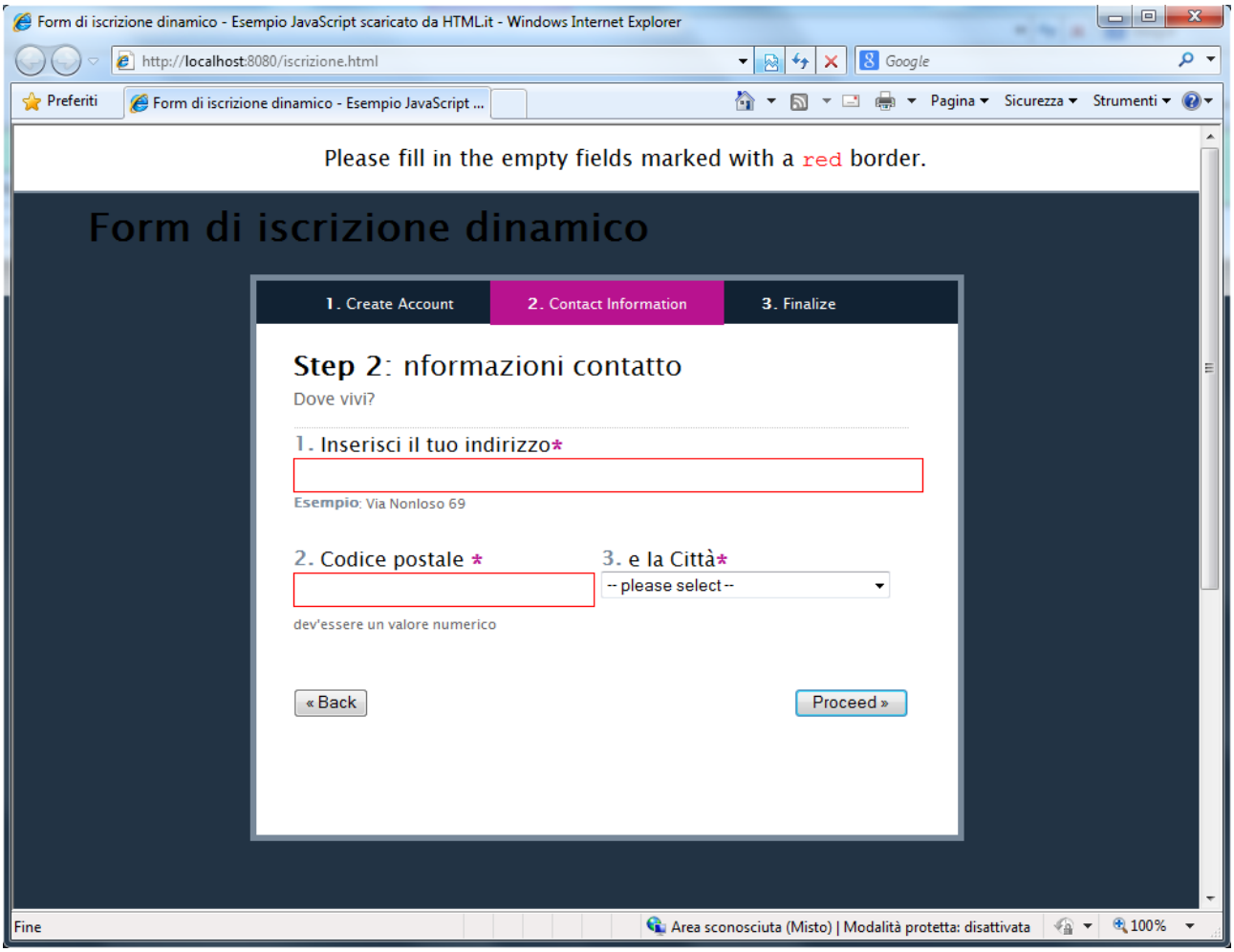
Un grosso problema nella misurazione della copertura di pagine web con javascript è il fatto che le pagine web sono create dinamicamente a tempo di esecuzione dalla parte server delle applicazioni web, per cui non esiste, in generale, il concetto di codice sorgente javascript (dato che può essere generato a tempo di esecuzione).

JSCover sopperisce in parte a questo problema, utilizzando una modalità alternativa di operazione rispetto alla "iFrame", ossia la windows mode. Per utilizzare questa modalità basta schiacciare il bottone "open in windows" piuttosto che "open in frame"; grazie a questa modalità sarà possibile aprire l'URL di test in una nuova finestra, in modo da eseguire e testare il codice in questa nuova finestra e tornare alla tab di JSCover visualizzando la tab Summary ed eventualmente ottenere un report di copertura.

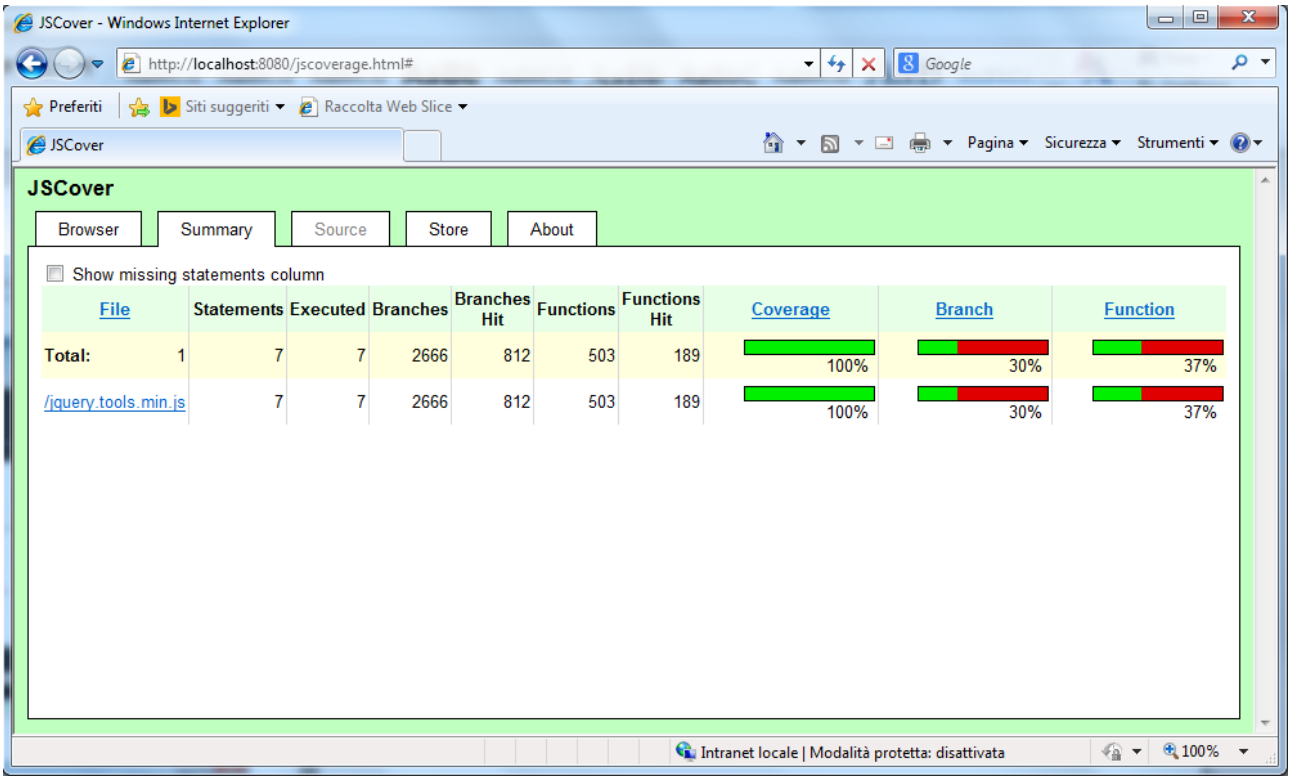
Per provare l'effettiva utilità di questa funzione, il tool è stato utilizzato per testare la copertura di codice javascript con contenuti dinamici. Il risultato è che lo strumento funziona solo in parte: come si può notare dagli screen di seguito, il test è stato fatto su un form di iscrizione dinamico.

Lo script di prova prevede l'inserimento di alcuni dati come email, scelta di nome utente e password nonché l'inserimento di alcuni dati anagrafici, inoltre è previsto un messaggio di errore nel caso delle caselle rimangano vuote


JSCover - Windows Internet Explorer
JSCover
Browser
Summary
Source
Store
URL: http://localhost:8080/iscrizione.html
Form di iscrizione dinamico - Esempio JavaScript scaricato da HTMLit - Windows Internet Explorer
http://localhost:8080/iscrizione.html
Form di iscrizione dinamico - Esempio JavaScript ...
Form di iscrizione dinamico
1. Create Account
2. Contact Information
3. Finalize
Step 1: Creazione Account
Inserisci i tuoi dati per il login:
1. Inserisci il tuo indirizzo mail*
prova@mail.it
La password che scegli sarà inviata al tuo indirizzo mail. L'indirizzo non sarà visibile.
2. Scegli un username
pincopallino
L'username sarò utilizzato per l'accesso.
3. Password *
Verifica Password *
Dev'essere lunga almeno 8 caratteri.
Avanti »
Fine
Area sconosciuta (Misto) | Modalità protetta: disattivata
100%
Intranet locale | Modalità protetta: disattivata


Form di iscrizione dinamico - Esempio JavaScript scaricato da HTMLit - Windows Internet Explorer
http://localhost:8080/iscrizione.html
Google
Preferiti
Form di iscrizione dinamico - Esempio JavaScript ...
Pagina
Sicurezza
Strumenti
Please fill in the empty fields marked with a red border.
Form di iscrizione dinamico
1. Create Account
2. Contact Information
3. Finalize
Step 2: nformazioni contatto
Dove vivi?
1. Inserisci il tuo indirizzo*
Esempio: Via Nonloso 69
2. Codice postale *
3. e la Città*
-- please select --
dev'essere un valore numerico
« Back
Proceed »
Fine
Area sconosciuta (Misto) | Modalità protetta: disattivata
100%

Come spiegato prima il tool funziona solo in parte in quanto il report risultante, qualsiasi siano le operazioni in fase di test, traccia una percentuale di copertura sempre del 100% anche nel caso in cui l'esecuzione dello scpript viene interrotta mentre le percentuali di branches e funzioni sono correttamente visualizzate:



Limite invalicabile invece è quello delle pagine che si automodificano, infatti essendo l'istrumentazione avvenuta prima dell'esecuzione via browser, è di fatto impossibile tenere traccia di codice che si genera proprio durante l'esecuzione.

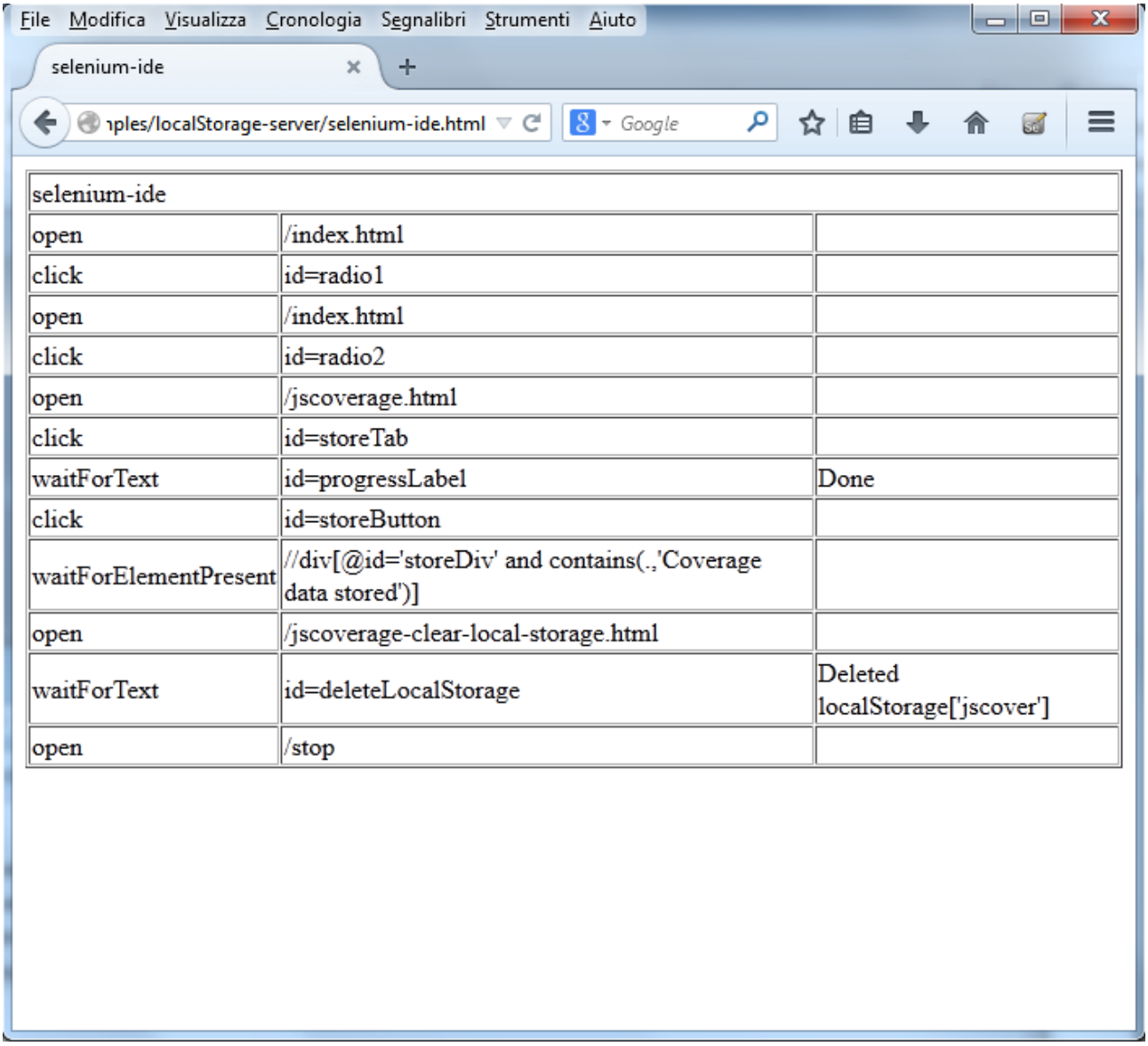
Per quanto riguarda l'automazione dei tests, JSCover offre supporto per alcuni strumenti più utilizzati per produrre test javascript, vediamo come funziona l'integrazione con Selenium. Quest'automazione avviene in maniera molto semplice, una volta creato un caso di test in formato Html attraverso Selenium IDE è possibile procedere col testing in questo modo:

Si avvia il webserver, si punta il browser (Firefox è l'unico browser che supporta il plugin Selenium IDE) si punta all'indirizzo http://localhost:8080/jscoverage.html , si carica il file

da testare e si preme il testo “play test case” del plugin selenium IDE per produrre il test.

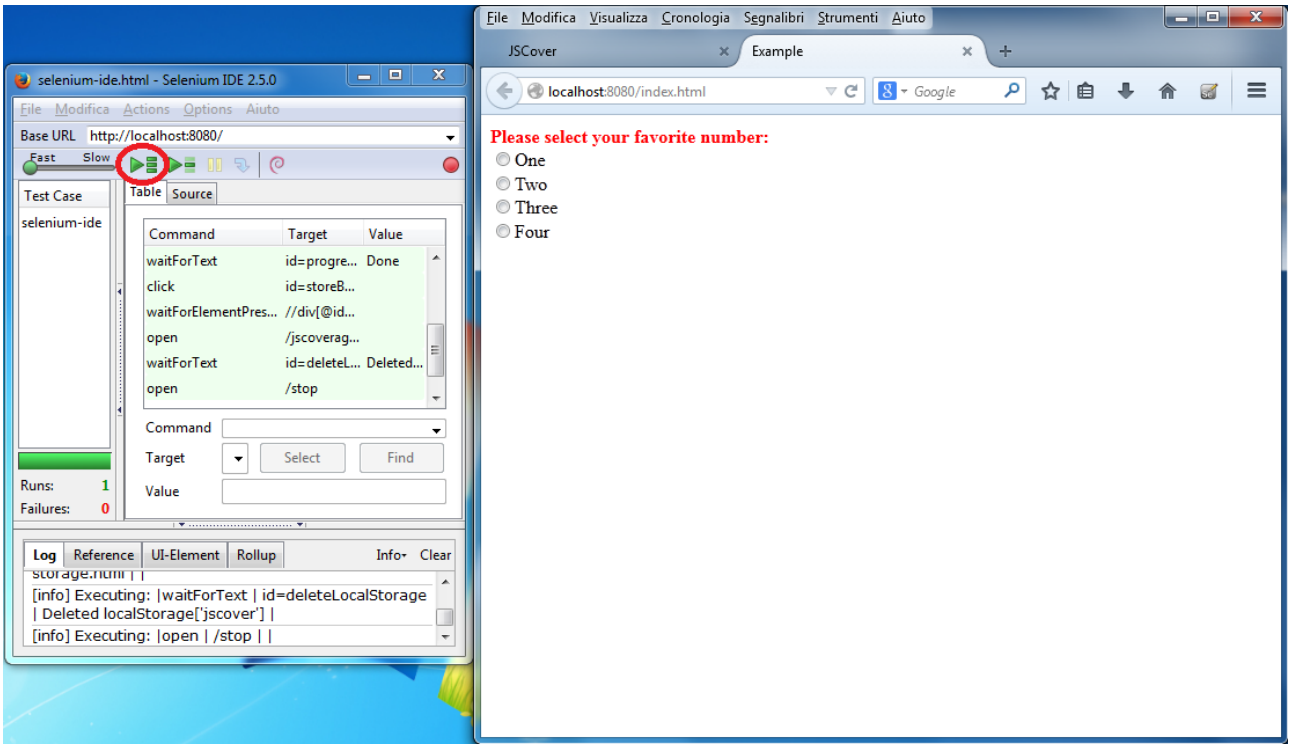
Esempio d’uso:

il caso di test è il seguente



selenium-ide		
open	/index.html	
click	id=radio1	
open	/index.html	
click	id=radio2	
open	/jscoverage.html	
click	id=storeTab	
waitForText	id=progressLabel	Done
click	id=storeButton	
waitForElementPresent	//div[@id='storeDiv' and contains(.,'Coverage data stored')]	
open	/jscoverage-clear-local-storage.html	
waitForText	id=deleteLocalStorage	Deleted localStorage['jscover']
open	/stop	

Funzionamento:



Infine per visualizzare il rapporto di copertura basta andare nella tab Summary

Oppure salvare il report nella tab Store.

3.2 Istanbul

Restando sempre sul linguaggio JavaScript un altro tool utile per il calcolo della percentuale di copertura del codice è Istanbul [12].

Al contrario di quanto avviene per JSCover, il funzionamento di questo strumento è legato alla piattaforma Node.js

Node.js (o, come è più brevemente chiamato, semplicemente "Node") è una soluzione server-side per JavaScript, in grado di ricevere e rispondere alle richieste http, in pratica, è possibile gestire mediante Node porte, socket e thread; in particolare è in grado di occuparsi di programmazione di rete e server-side e di effettuare richieste / risposte di elaborazione.

Focalizzando l'attenzione su quanto riguarda il code coverage una volta creata una test suite con Node, è possibile, grazie a Istanbul, ottenere la percentuale di copertura del codice e un

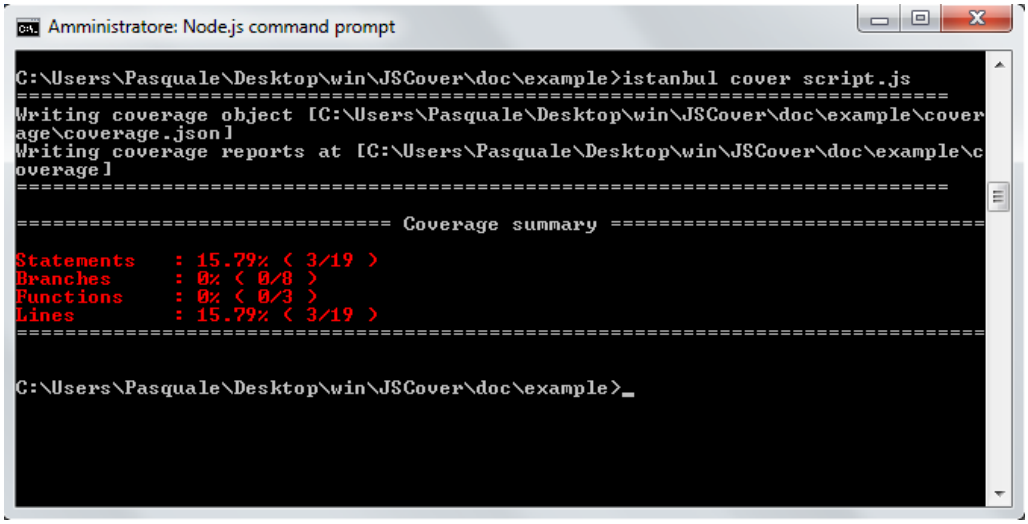
report in formato html in maniera veramente semplicissima.

La prima cosa da fare una volta scaricato e installato Node.js, è installare Istanbul, cosa che avviene con il comando (dal terminale di Node)

n*pm install -g istanbul*

Fatto questo, avendo a disposizione una test suite appositamente creata, per verificare la copertura del codice basta dare il comando

istanbul cover test.js



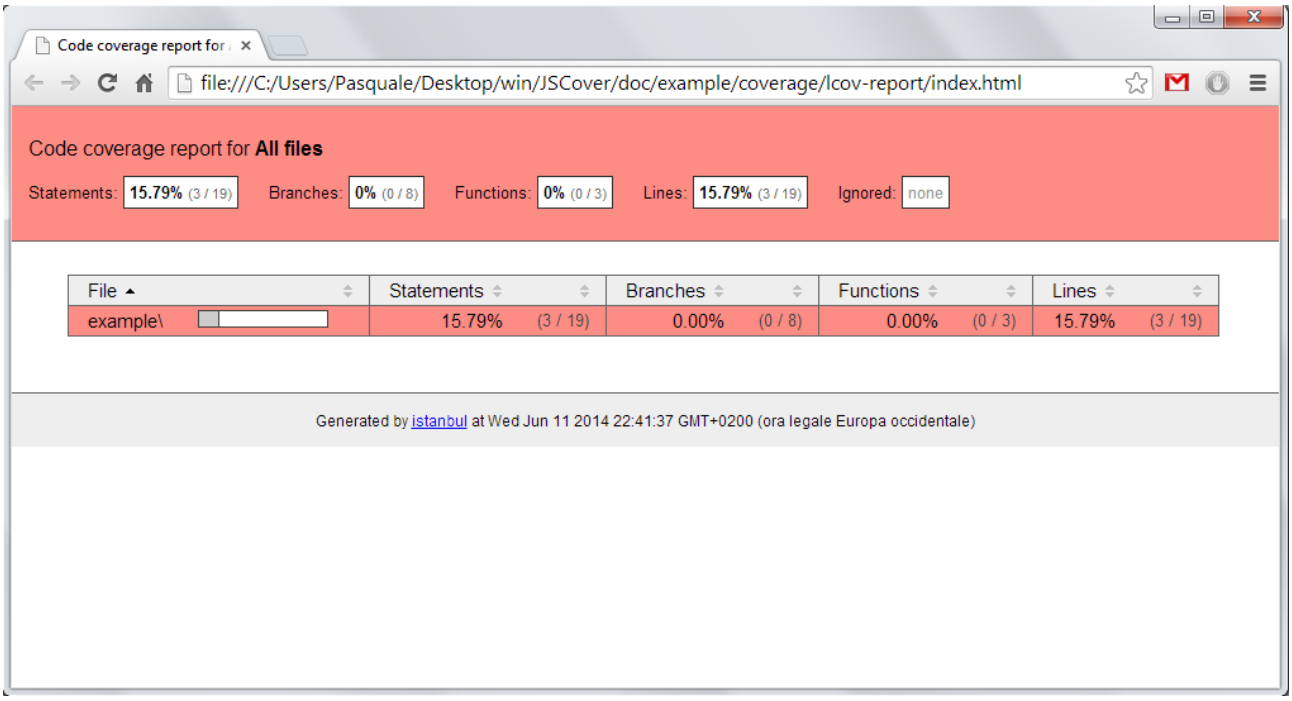
In questo modo Istanbul verifica la copertura del codice e genera un report nel file *coverage.json* memorizzandolo in una cartella chiamata coverage, creata nella directory dello script di test.

Il report di copertura può inoltre essere convertito in diversi formati tra cui html, xml (utilizzando come template l'output di cobertura o clover) oppure txt.

Di default il report viene salvato in html mentre il comando per ottenere il formato, ad esempio xml sul template cobertura è

Istanbul report cobertura

Di seguito un esempio di report di Istanbul in formato html utilizzando lo stesso file testato con JSCover:



Istanbul è in grado quindi di tenere traccia della copertura di istruzioni, rami e funzioni, è in grado di esportare reports in formato html o xml sulla base del template di Cobertura o anche di Clover.

Purtroppo Istanbul non è in grado di fornire una soluzione al problema del codice creato dinamicamente a tempo di esecuzione dalla parte server delle applicazioni web.

Capitolo 4: Confronto critico tra gli strumenti

Dopo aver analizzato in maniera dettagliata il funzionamento e le caratteristiche dei vari strumenti, procediamo ad un confronto diretto per poter valutare quale scegliere in diversi ambiti di utilizzo. Partiamo con una tabella riepilogativa

TOOL	IDE/RIGA DI COMANDO	TIPO DI ISTRUMENTAZIONE	FORMATO REPORT	SUPPORTO JUNIT
Emma	Riga di comando	Istrumentazione sulle classi	txt, html, xml	✗
EclEmma	IDE Eclipse	Istrumentazione sulle classi	html, xml, csv	✓
Gretel	Riga di comando/interfaccia grafica	Istrumentazione sulle classi	nessun report	✗
Cobertura	Riga di comando	Istrumentazione sulle classi	html, xml	✗
eCobertura	IDE Eclipse	Istrumentazione sulle classi	nessun report	✓
CodeCover	Riga di comando o IDE Eclipse	Istrumentazione sui files sorgenti	csv, html,xml	✓
Clover (trial)	Riga di comando o IDE Eclipse	Istrumentazione sui files sorgenti	html, xml, pdf	✓

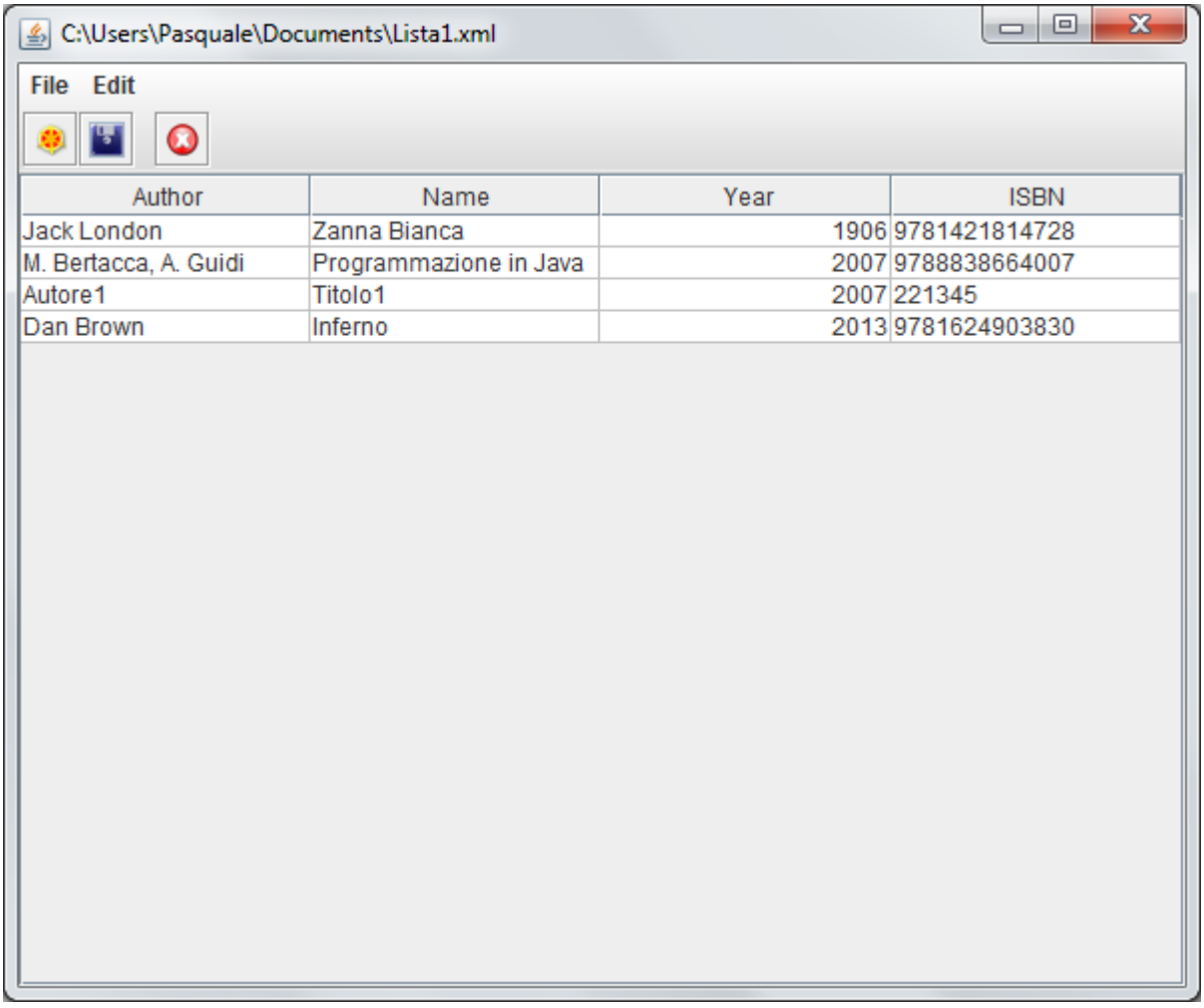
Per ottenere un confronto comparativo trai diversi tools, sono stati utilizzati tutti gli strumenti per valutare la copertura del codice su un'applicazione Java dotata di interfaccia grafica, SimpleJavaApp.

Questa semplice applicazione gestisce un elenco di libri memorizzando informazioni quali Titolo, Autore, Anno di pubblicazione e codice ISBN.

L'applicazione è in grado di creare una lista, cancellare delle righe, salvare la lista in formato xml, aprire e visualizzare liste create in precedenza.

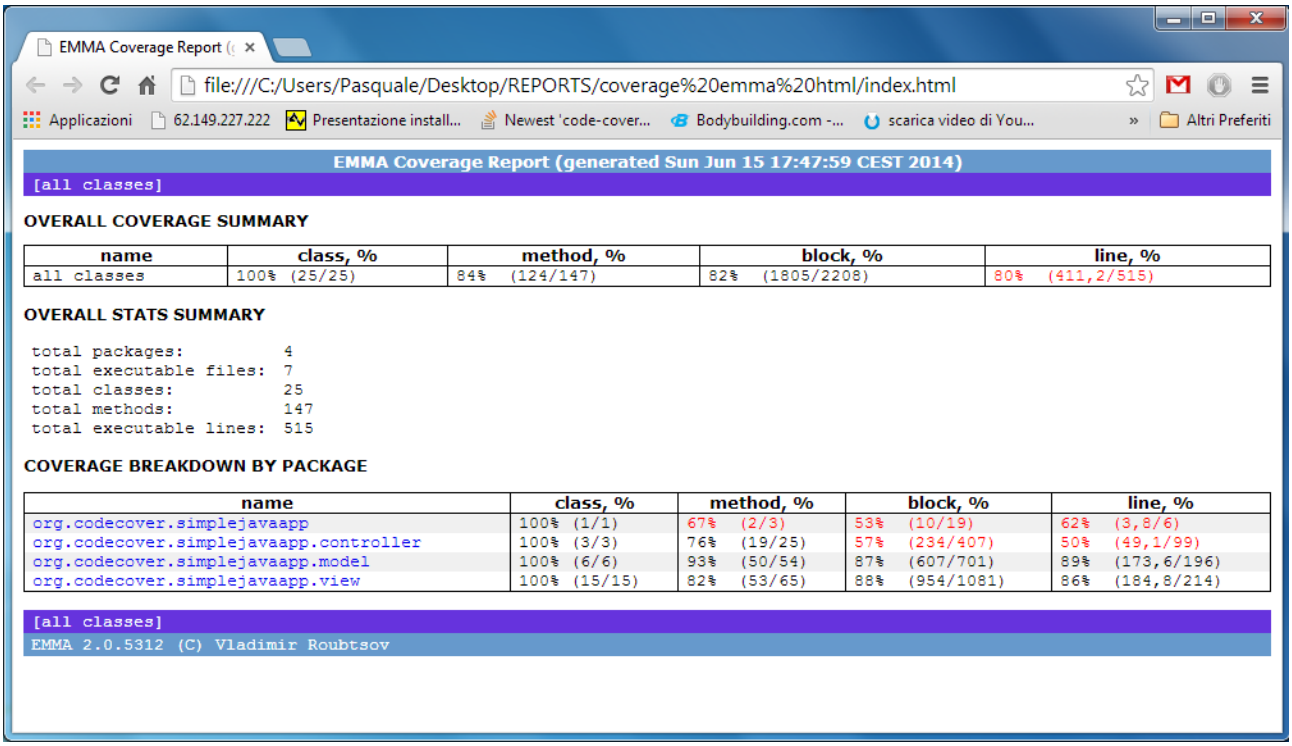
Le operazioni che saranno effettuate saranno:

- Creazione di una nuova lista con 3 libri
- Salvataggio della lista
- Apertura di una lista creata in precedenza

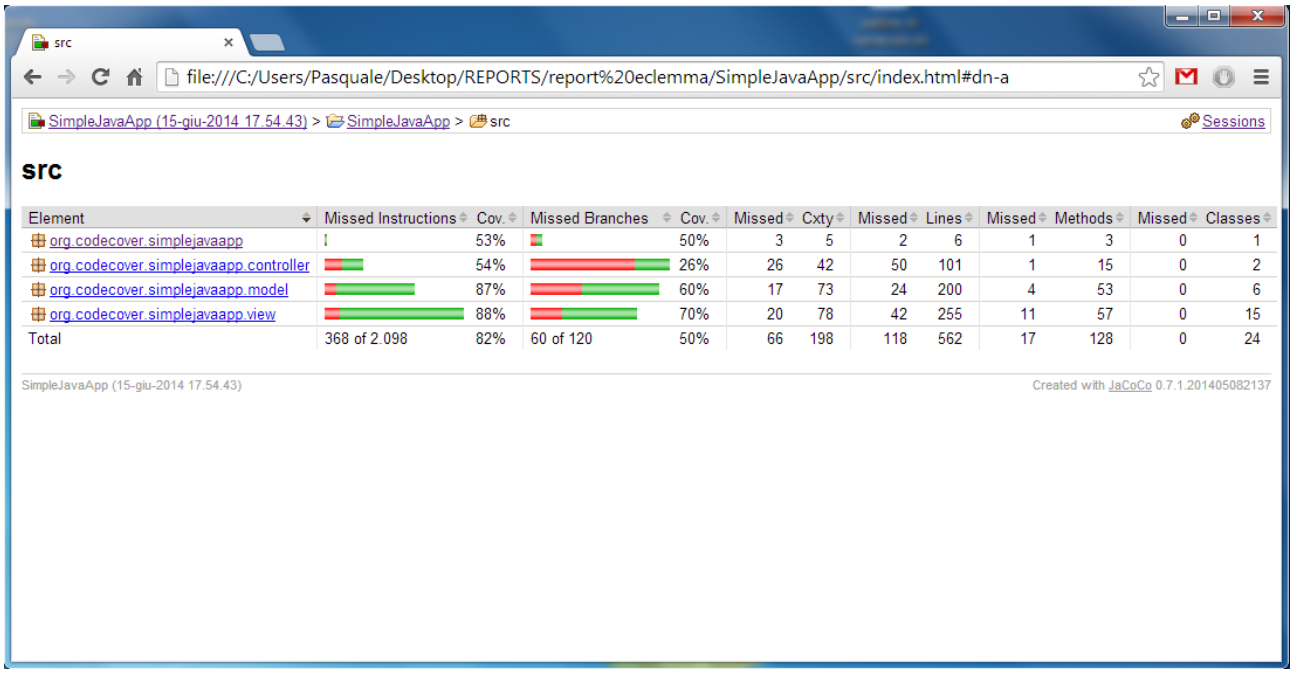


Come descritto in precedenza, lo strumento Gretel ed il plugin eCobertura non consentono la produzione di reports di copertura mentre di seguito sono presentati i reports di tutti gli altri strumenti nell'unico formato che hanno in comune, html.

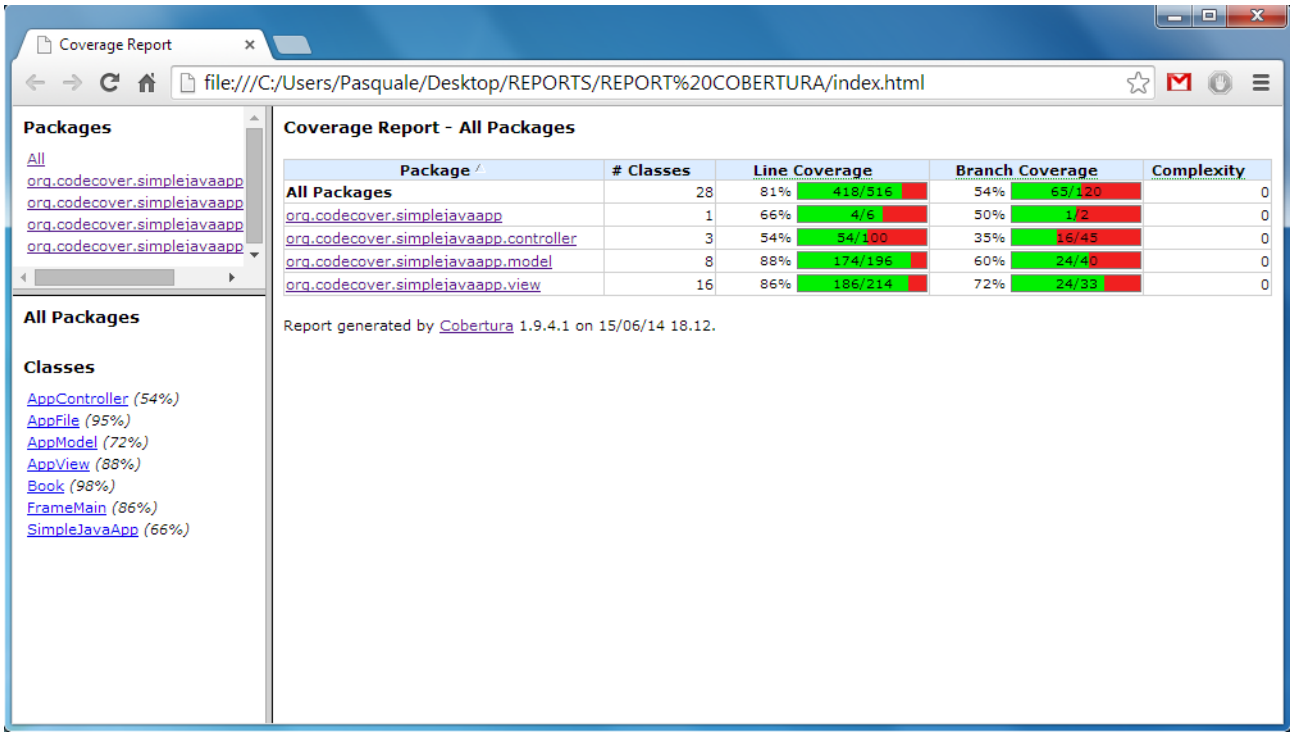
Emma:



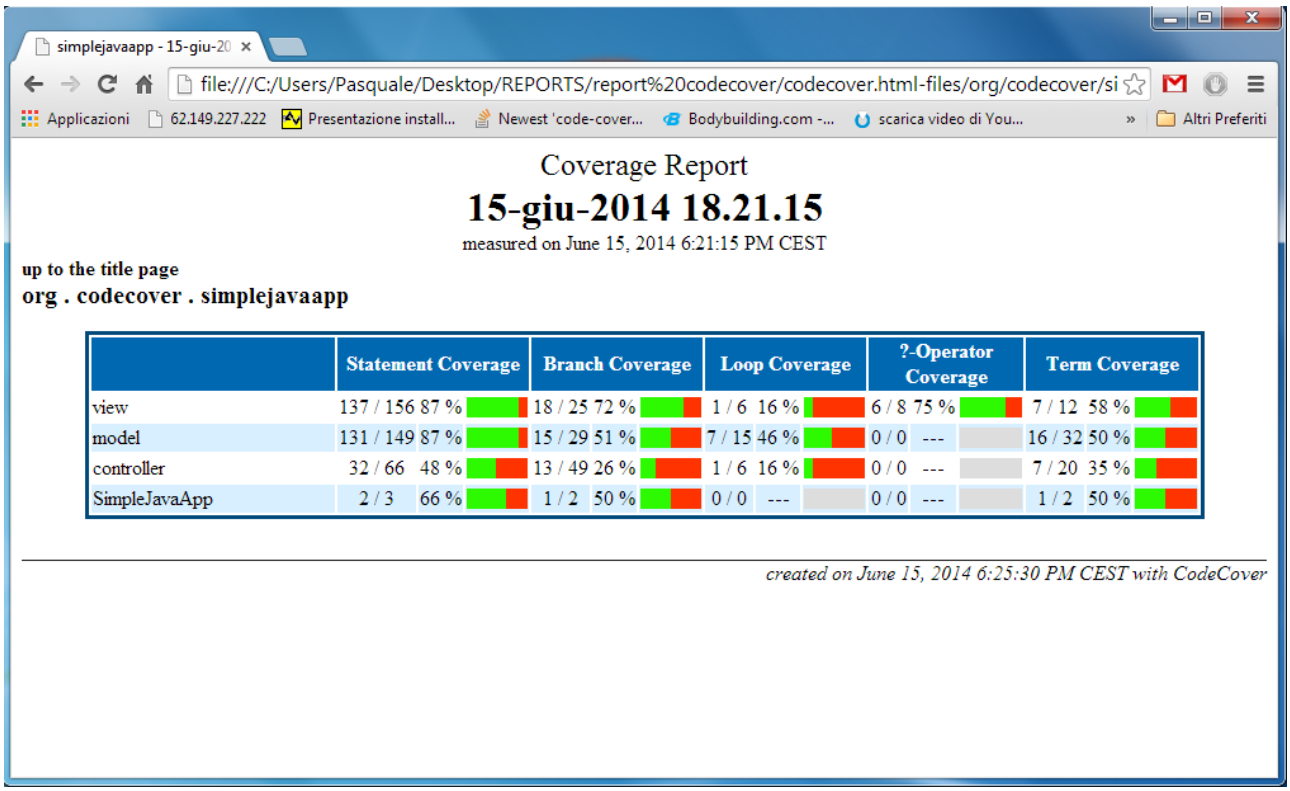
EclEmma:



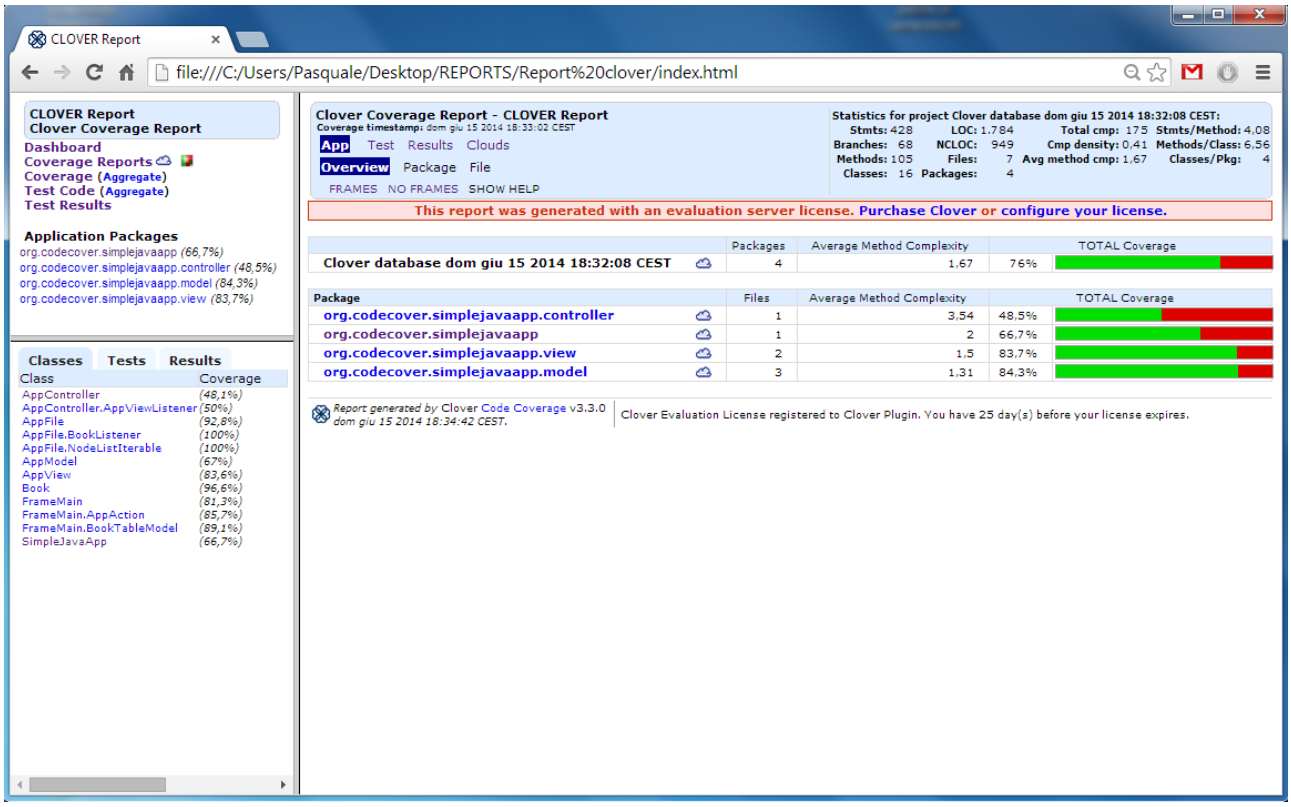
Cobertura:



CodeCover:



Clover:



Come si può notare dai vari reports, i risultati, anche se molto simili, sono quasi tutti diversi tra loro ed il motivo risiede nelle diverse convenzioni sul criterio di copertura utilizzati da ciascun tool.

Alla luce di quanto visto finora, possiamo dare un giudizio complessivo per stabilire quale degli strumenti analizzati può risultare più utile in diverse situazioni di testing.

La scelta dell'uno o dell'altro tool per la verifica della copertura del codice va fatta necessariamente in base al tipo di progetto che è stato realizzato, pertanto possiamo riassumere tre condizioni in base alle quali scegliere lo strumento più adatto alle proprie esigenze: progetto realizzato con l'IDE Eclipse, casi di test realizzati con JUnit, nessun caso di test.

Nel caso in cui si sia scelto di utilizzare Eclipse come ambiente di sviluppo, gli strumenti integrati come plugin visti fin ora sono EclEmma eCobertura CodeCover e Clover.

Partiamo da quello meno meno funzionale, che risulta essere eCobertura: sebbene basato su Cobertura, questo plugin non risulta molto utile in quanto è in grado di visualizzare le righe

di codice evidenziate con colori differenti in base alla loro copertura, e nient'altro: non è possibile fare un merge di diverse sessioni di test e soprattuto non è possibile produrre un qualsiasi report di copertura.

CodeCover è molto più completo, è possibile testare il proprio progetto in diversi momenti, unire le diverse sessioni di test, ottenere report di copertura in diversi formati tra cui XML (possibile quindi sfruttare il report da altri software) oltre che HTML e CSV.
Il plugin EclEmma tuttavia risulta essere anche più efficiente in quanto unisce le caratteristiche dei due precedenti in modo da visualizzare le linee di codice sorgente evidenziate secondo i criteri coperto/non coperto oltre ad avere diverse sessioni di test, fare un merge delle sessioni e ottenere report in diversi formati.

Clover infine risulta allo stesso livello di EclEmma anche se ancora più completo, più immediato (offrendo la visualizzazione della copertura con Coverage Treemap Report) e grazie ai suoi report estremamente completi e dettagliati (parlando di quello in formato html); è l'unico che tra i vari formati di report offre anche il pdf. A conti fatti però essendo Clover disponibile solo in versione trial, la miglior scelta risulta essere EclEmma.

Tra gli strumenti che si integrano con il framework JUnit per verificare la copertura dicasi di test invece ricordiamo che quelli che hanno un supporto ufficiale sono EclEmma, CodeCover, e Clover.

Essendo ognuno di questi, disponibili come plugin di Eclipse, scegliere l'uno o l'altro è quasi indifferente se non per il fatto che, come detto poco fa, EclEmma e Clover rispetto a CodeCover vantano la comodità di mostrare il codice sorgente evidenziato in colori differenti a seconda che la riga sia stata o meno coperta. Se per il progetto non è stato utilizzato Eclipse invece, EclEmma diventa inutilizzabile e quindi, dato sempre il fattore penalizzante della licenza trial di Clover, la scelta vincente e quella di utilizzare CodeCover da riga di comando.

L'ultimo scenario possibile, che è poi quello più utilizzato in caso di piccoli progetti, è quello di voler testare il proprio programma in java, senza casi di test e senza un ambiente di sviluppo.

In tale scenario i tools utilizzabili sono Emma, Gretel, Cobertura, CodeCover e Clover.
I tools Emma, Cobertura, CodeCover e Clover, dal punto di vista dell'utente funzionano esattamente allo stesso modo, anche se di fondo c'è una grossa differenza di funzionamento (Emma e Cobertura istrumentano le classi precompilate mentre CodeCover e Clover istrumentano direttamente i files sorgenti); gli unici due che si differenziano per una funzionalità in più sono Cobertura e Clover, i quali sono in grado anche di dare una misura della complessità di classi e package. Per quanto riguarda il tool Gretel, questo risulta macchinoso da utilizzare come primo approccio ma difatti abbastanza semplice dato che non fornisce particolari funzionalità limitandosi a visualizzare il codice sorgente evidenziato in maniera differente in base alla copertura. Può risultare a mio avviso utile solo per piccoli progetti e soltanto nel caso non si utilizzi un IDE come Eclipse, in tal caso la scelta può andare altrove.

Riguardo i due tools utilizzati per misurare la copertura del codice JavaScript, il più completo è senza dubbio JCCover il quale, oltre a poter fornire la copertura del codice di casi di test, integrandosi sia con Selenium che con HtmlUnit, fornisce una intuitiva e comoda interfaccia per poter testare ogni singolo file js in maniera completa rivelandosi quindi utilissimo sia per progetti di grossa mole che per quelli molto piccoli.
Istanbul dalla sua parte ha una facilità di utilizzo davvero eccezionale anche se implica la necessità di utilizzo della piattaforma Node.js e node unit test già pronti da eseguire.

Conclusioni

L'obiettivo prefissato dalla tesi era quello di valutare diversi software open source per la verifica della copertura del codice, al fine di individuarne uno che più soddisfi le esigenze degli sviluppatori in relazione al progetto a cui si lavora.

È un dato di fatto che molti progetti falliscono a causa della mancanza di qualità dei sistemi software. È per questa ragione che avere un occhio di riguardo al controllo di qualità del codice, durante le varie fasi di sviluppo, ormai non è più opzionale ma un fattore critico. Dal momento che è riprovato che una diagnosi precoce sui problemi di qualità permette una loro risoluzione più facilmente, negli ultimi anni si è osservato che sempre più aziende riconoscono l'importanza di produrre software di qualità, ed impiegano le giuste risorse nei processi di collaudo che hanno proprio questo come obiettivo. In fondo sebbene il software abbia caratteristiche alquanto differenti rispetto ad altri prodotti con cui siamo abituati a entrare in contatto, è pur sempre anch'esso il risultato di un lavoro e quindi un prodotto che, in quanto tale, necessita degli opportuni controlli per soddisfare chi ne fa uso. Per tale motivo, al fine di ottenere risultati quanto più possibili esenti da errori, ci si affida sistematicamente a strumenti automatici per supportare questo processo. Infatti al crescere ed evolversi di grossi sistemi, mantenere la complessità ad un livello gestibile è una sfida sempre aperta e ci si augura che nel prossimo futuro le potenzialità (nonché il numero) di questi strumenti si moltiplichino ancor più che negli ultimi anni.

Non è un caso l'aver utilizzato precedentemente la parola diagnosi precoce; infatti il processo di testing contestuale al processo di sviluppo può essere visto metaforicamente come un processo di prevenzione. Per evitare che il software si "ammali" ed il suo destino sia quello di essere abbandonato (che per un prodotto software significa la morte!), occorrono degli approcci di prevenzione di qualità, e quindi di testing. In fondo si sa che ... prevenire è meglio che curare.

Bibliografia

[1] [Ieexplore.ieee.org], [http://ieeexplore.ieee.org/xpl/login.jsp?reload=true&tp=&arnumber=6328176&url=http%3A%2F%2Fieeexplore.ieee.org%2Fiel5%2F6327839%2F6328119%2F06328176.pdf%3Farnumber%3D6328176]

[2] [itware.com], [http://www.itware.com/blog-itware/alm-qualita-del-software/item/855-test-del-software-incrementare-i-livelli-di-copertura-tagliandone-del-50-costi-ed-effort-con-un-approccio-pragmatico-ed-una-suite-specializzata]

[3] [Emma documentazione], [http://emma.sourceforge.net/]

[4] [EclEmma documentazione], [http://www.eclemma.org/]

[5] [Gretel documentazione], [http://www.cs.uoregon.edu/research/perpetual/dasada/Software/Gretel]

[6] [Cobertura documentazione], [http://cobertura.github.io/cobertura/]

[7] [cCobertura documentazione], [http://ecobertura.johoop.de/]

[8] [CodeCover], [http://codecover.org/]

[9] [Clover documentazione] [https://www.atlassian.com/software/clover/overview]

[10] [TIOBE tabella] [http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html]

[11] [JSCover, documentazione] [http://tntim96.github.io/JSCover/manual/manual.xml]

[12] [Istanbul, documentazione] [https://www.npmjs.org/package/istanbul]

[13] G. Romeo, Stato dell’arte e valutazione di strumenti open source per il testing white box di software object-orientedrivista, 2012