UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

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

elaborato di laurea

Strumenti per la comparazione di misure di copertura

Anno Accademico 2014/2015

relatore

Ch.mo Prof. Porfirio Tramontana

candidato
Angelo Esposito
matr. 534001762

[Dedica]

Indice

Introduzione

Fino a pochi anni fa, quando si parlava di software in grado di soddisfare esigenze di aziende o singoli utenti, era piuttosto scontato riferirsi ad applicazioni sviluppate principalmente per server o personal computer, mentre, al giorno d'oggi, il numero dei tipi diversi di macchina per i quali un applicativo è generalmente destinato è molto più esteso. Difatti stiamo assistendo negli ultimi anni ad una crescente e sempre più capillare diffusione di diversi dispositivi elettronici quali ad esempio Smart-TV, console, tablet, smartphone e smartwatch, tutti in grado di supportare l'esecuzione di applicazioni più o meno complesse, esattamente come avviene per un personal computer.

Accade molto spesso che un qualsiasi programma, sia esso destinato ad aziende, che a dispositivi mobili oppure a sistemi critici, presenti diversi errori nel codice, detti anche *bug*. Essi sono in grado di compromettere la corretta esecuzione di un software causando *crash*, blocchi improvvisi, rallentamenti o comportamenti anomali, cioè quelli che in generale vengono chiamati *malfunzionamenti*. In alcuni contesti la conseguenza di tali malfunzionamenti può anche essere catastrofica, si pensi ad esempio a sistemi per il controllo del traffico aereo o ferroviario.

Possiamo ritenere che nessun software sia esente da errori e che la loro presenza sia da considerarsi una condizione normale. La probabilità di trovarne cresce tipicamente al

crescere della complessità del software stesso ma, un ulteriore fattore di causa, del quale non si può avere completo controllo, è rappresentato dall'ambiente stesso nel quale esso viene ad essere eseguito.

Per fare un esempio sulle possibili cause della presenza di errori nel codice, consideriamo il mercato delle applicazioni per smartphone. Il numero di applicazioni che si trovano sui *market* dedicati sono dell'ordine delle centinaia di migliaia [1] e molto spesso è facile trovarne più d'una che hanno funzionalità molto simili, se non identiche. La crescita esponenziale delle *app* rilasciate, la facilità di trovare alternative, l'accesa concorrenza tra sviluppatori o Software House possono essere alcuni dei fattori determinanti a spingere questi ultimi a rilasciare applicazioni in tempi stretti, senza adottare adeguati processi di sviluppo di qualità, oppure a rilasciare *update* senza adeguati collaudi. Quindi si rilascia il prodotto senza averlo opportunamente testato perché è semplicemente troppo lungo e costoso farlo. Ci si affida alla speranza che esso non presenti malfunzionamenti gravi oppure alle segnalazioni degli utenti sui comportamenti anomali e di conseguenza il prodotto potrebbe non essere percepito come qualitativamente valido. In questo scenario c'è il rischio concreto che un'app che presenti troppi malfunzionamenti abbia una bassissima probabilità di essere adottata, a maggior ragione se ne esiste una alternativa.

Si rende quindi necessario trovare il modo di scovare i *bug* software e risolverli, al fine di incrementare la qualità intrinseca di un sistema software e, di conseguenza, la qualità percepita dall'utente finale.

Un procedimento di verifica delle qualità di correttezza, completezza e affidabilità di un software è il *testing* o collaudo, un'attività fondamentale nel campo dello sviluppo software che, tuttavia, può essere a volte trascurata per motivi di budget o di tempo.

In una delle sue declinazioni, l'attività di testing viene impiegata per scovare bug e viene condotta utilizzando diverse tecniche e strategie tramite strumenti che consentono il testing automatico. Un esempio sono le cosiddette tecniche *Radom Testing* e *Ripper-Based Testing*, supportate da un software chiamato *GUI-Ripper* [2], uno strumento sviluppato

presso il Dipartimento di Informatica e Sistemistica della Federico II ed utilizzato a supporto delle attività di *testing* automatico di applicazioni Android.

Il processo di testing dovrebbe possedere alcune caratteristiche intrinseche di qualità al fine di poter riporre fiducia nei risultati che esso presenta. Una di esse è l'efficacia del testing, la quale rappresenta la capacità di rilevare errori nel prodotto testato ed è definibile come numero di difetti scoperti rispetto al numero di difetti esistenti nel software sotto analisi.

Per valutare la bontà del testing ci si può quindi affidare alla misura dell'efficacia dello stesso. Tuttavia una misura diretta è impossibile poiché il numero di difetti esistenti in un software non è conoscibile a priori ed è pertanto necessario ricorrere a misure indirette.

Una misura indiretta dell'efficacia di un test è la *copertura* (o *coverage*) delle linee codice, essa esprime in termini percentuali la quantità di linee di codice eseguite (coperte) durante l'esecuzione di un test rispetto alla quantità totale di linee di codice di cui è composto il programma. Quanto più alta è la percentuale di copertura data dall'esecuzione di un test, tanto più efficace possiamo considerare il test stesso.

Per conoscere la copertura data da un'esecuzione di una *test suite* su un programma, ci si può avvalere di strumenti come *EMMA* [3], il quale produce risultati in formato *HTML* [4] contenenti informazioni sia sintetiche che dettagliate sulla copertura ottenuta in un test.

Può essere utile analizzare e confrontare a posteriori i rapporti di copertura, prodotti su una medesima applicazione, dall'esecuzione di test che adottino diverse tecniche, con lo scopo, ad esempio, di valutarne l'efficacia relativa, per decidere quale tra quelle utilizzate sia la migliore; oppure per confrontarne l'efficacia aggregata, per stabilire la percentuale di copertura raggiunta da una combinazione di due o più tecniche di testing; o ancora, sapere se esistono porzioni di codice (e quali sono) che sono coperte da una sola tecnica ma non dalle altre.

Per fare un esempio pratico, si supponga di aver eseguito due test, indichiamoli con A e B, con tecniche diverse e di averne a disposizione i rispettivi rapporti di copertura.

Supponiamo il caso particolare in cui entrambi riportino la stessa percentuale di copertura. Per il confronto tra i due test, si può immaginare di doverli esaminare contemporaneamente ed essere in grado di trovare risposta ai seguenti quesiti:

- Essi coprono le stesse linee di codice?
- Quali e quante linee di codice sono state coperte da A ma non da B? E quali da B ma non da A?
- Quali e quante linee di codice sono state coperte da entrambi i test?
- Quali e quante linee di codice non sono state coperte da nessuno dei due test?
- Qual è la percentuale di copertura totale derivante dall'aggregazione delle linee di codice coperte da entrambi i test?

Gli strumenti attualmente a disposizione forniscono soltanto informazioni dettagliate di copertura su singole sessioni di test, non esistono strumenti in grado di confrontare diverse istanze/processi di testing sulla stessa applicazione in maniera dettagliata, per cui ci si dovrebbe affidare ad un ispezione visiva da parte di un operatore umano di rapporti di copertura di migliaia di linee di codice. Ci si può rendere conto di quanto un'operazione del genere possa risultare impraticabile per via del tempo necessario per completarla, senza contare che l'errore umano potrebbe inficiarne la correttezza.

Generalizzando, la quantità di tempo necessaria ad effettuare tali confronti per ogni applicazione testata, è funzione del numero di linee di codice di cui è composta, del numero di report di test da confrontare e dal numero di quesiti ai quali rispondere, questi a loro volta non stabiliti a priori.

Lo scopo del mio lavoro è stato quello di sviluppare uno strumento in grado di analizzare automaticamente più rapporti di copertura relativi ad una medesima applicazione, memorizzarne tutte le informazioni di interesse in un database e rendere possibile un confronto tra essi fornendo uno strumento predefinito di interrogazione del database.

Il tool sviluppato è scritto in *JAVA* [5] ed è diviso in due moduli. Un modulo è progettato

per essere in grado di analizzare qualsiasi rapporto di copertura in HTML prodotto tramite EMMA al fine di ricavarne tutte le informazioni di interesse su applicazione e copertura e memorizzarle in un database diverso per ogni applicazione. L'altro modulo fornisce uno strumento di interrogazione del database per la generazione automatica di un report dettagliato per ogni specifica interrogazione effettuata, le interrogazioni sono predeterminate ma il tool è aperto ad upgrade in grado di aggiungerne di nuove.

In definitiva il tool serve proprio a rispondere alle domande di cui sopra (tradotte in operazioni insiemistiche) in un tempo ragionevole ed in modo dettagliato o a supportare nuove esigenze di analisi e confronto.

Il dettaglio su come è stato progettato ed implementato il software è mostrato nel capitolo 3, mentre nel capitolo 4 viene riportato un manuale ed un esempio d'uso. Nei capitoli 1 e 2 invece. viene spiegato il background teorico sul quale poggia il lavoro descritto.

Capitolo 1

Testing – fondamenti teorici

1.1 Verifica e validazione

L'attività di *testing* va inquadrata nel più vasto processo di controllo ed analisi del software, la cosiddetta verifica e validazione (V & V). La verifica ha il compito di controllare che il software sia conforme alle sue specifiche, mentre la validazione (o convalida) ha un obiettivo più generale, quello di assicurarsi che il sistema software rispetti le attese del cliente (utente). Lo scopo di tale processo è quello di creare fiducia nel software cercando di convincere gli sviluppatori e i clienti del sistema che esso è pronto ed adeguato all'uso per il quale è destinato (cfr. Sommerville, 2007).

- "(Verifica) *Stiamo costruendo il prodotto nel modo giusto?* " (Boehm, 1979)
- "(Validazione) *Stiamo costruendo il prodotto giusto?* " (Boehm, 1979)

E' necessario che siano verificate tutte le qualità del software. Non basta verificare se il software implementato si comporta in maniera corretta rispetto ai documenti di specifica, ma è necessario certificare anche proprietà quali l'affidabilità, la funzionalità, l'usabilità, l'efficienza, la modificabilità, etc.

La verifica può essere effettuata in diversi momenti, da persone diverse, con diversi

obiettivi ed applicando tecniche differenti. Ad esempio, in alcuni casi può essere utile che un prodotto sia verificato da persone diverse rispetto a quelle che lo hanno implementato; in altri casi la verifica potrebbe essere effettuata attraverso l'applicazione di specifici algoritmi, e così via.

L'attività di verifica e validazione, come ogni altra attività di progettazione, deve procedere secondo principi rigorosi e tecniche adeguate; non può essere lasciata esclusivamente all'inventiva, all'esperienza né, tanto meno, alla buona sorte (Ghezzi, Jazayeri e Mandrioli, 2004)

Vi sono due metodi utilizzati a supporto della verifica e validazione, non usati in maniera esclusiva ma piuttosto in maniera complementare: l'*analisi statica* e l'*analisi dinamica* del sistema.

L'analisi statica non richiede che il software sia eseguito, consiste nel valutare il sistema o un suo componente analizzandone la forma, la struttura, il contenuto e la documentazione (revisioni, ispezioni, recensioni, analisi data-flow). L'analisi dinamica, al contrario, consiste nell'esecuzione del software con specifici dati, simili a quelli realmente elaborati, con il fine di esaminare i comportamenti e gli output prodotti per verificare se questi siano conformi alle attese.

Sono le tecniche di analisi dinamica ad essere identificate come *testing* (o *collaudo*).

1.2 Testing

"Il metodo più naturale e tradizionale per verificare un prodotto è quello di provarlo in un certo numero di situazioni rappresentative, accertando che si comporti come previsto" (Ghezzi, Jazayeri e Mandrioli, 2004, p.300)

Secondo la definizione dello Standard IEEE 610.12-1990, il testing è "*the process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component.*"

Le due attività fondamentali del test sono il test dei componenti ed il test del sistema. Nel primo, i test vengono eseguiti dagli *sviluppatori* del software e servono per scoprire i difetti dei singoli componenti del programma (funzioni, oggetti, componenti riutilizzabili); nel secondo, i test vengono eseguiti da *tester indipendenti* e servono per scoprire i difetti in gruppi di componenti integrati per formare il sistema o un sotto-sistema (cfr. Sommerville, 2007).

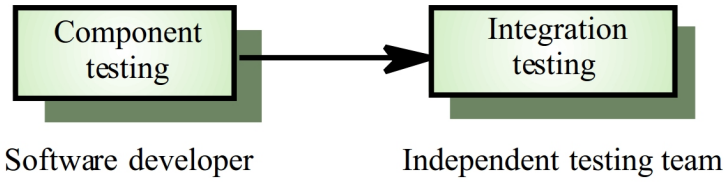


Figura 1.1 – Fasi di testing

Lo sviluppatore comprende bene il sistema ma è inconsciamente "gentile" perché tende ad avere la presunzione di aver realizzato software senza errori oppure perché lavora da tempo allo sviluppo, è annoiato dal sistema ed ha il timore inconscio di scoprire errori; il testing è guidato dalla consegna del software. Il *tester indipendente* deve imparare il sistema ma è più critico, esigente e maggiormente motivato perché non ha sviluppato lui il software, trovare un errore non sarebbe un suo "fallimento" in quanto il suo ruolo nel progetto è proprio quello di scovarli; il testing è guidato dalla qualità.

Il processo di test del software ha due obiettivi distinti:

1. **"Test di convalida**: serve per dimostrare che il software è quello che il cliente desidera, che soddisfa i suoi requisiti.[…]
2. **Test dei difetti**: serve per rilevare i difetti del sistema […] [con l'obiettivo di] trovare le inconsistenze tra il programma e le sue specifiche" (Sommerville, 2007, p.505).

1.2.1 Test dei difetti

Ghezzi, Jazayeri e Mandrioli (2004) descrivono in modo più formale le relazioni che intercorrono tra difetto, errore e malfunzionamenti nel software. Un difetto (o *bug*) è una manifestazione nel software di un errore umano dovuto all'incapacità di comprendere o risolvere un problema o di usare determinati strumenti. I difetti possono causare un fallimento del sistema nell'eseguire una funzione richiesta, cioè il sistema manifesta un malfunzionamento.

Un malfunzionamento (o *failure*) è l'incapacità del software di comportarsi secondo le aspettative o le specifiche, può essere osservato solo mediante esecuzione ed accade in un certo istante di tempo, per questo si dice che ha natura dinamica.

Un test dei difetti ha successo se porta il programma a comportarsi in modo errato, ovvero se lo porta ad esibire malfunzionamenti, indice della presenza di uno o più difetti nel software. Dunque, tramite il testing, si ricercano tutti i tipi di comportamento indesiderati del sistema, come crash, interazioni non volute con altri sistemi, calcoli errati, corruzione dei dati.

Purtroppo, come viene ben sintetizzato nella tesi di Dijkstra (1972), il test può solo dimostrare la presenza dei difetti non la loro assenza, ciò vuol dire che anche se un certo numero di test dei difetti fallisce (ovvero non vengono rilevati malfunzionamenti), non possiamo mai escludere che il software sotto esame abbia dei difetti nel codice. Anche nel caso abbia diversi bug, il software può continuare a comportarsi in maniera corretta ed alcuni malfunzionamenti possono manifestarsi dopo anni di messa in esercizio del programma, tutto dipende dagli input che gli vengono dati.

Solamente effettuando un **testing esaustivo**, un procedimento nel quale ogni possibile sequenza di esecuzione di un programma viene verificata, potremmo avere l'assoluta certezza di scovare tutti i bug. Esso è però in generale infattibile (Sommerville, 2007). Un

semplice esempio di programma con poche linee di codice e due cicli annidati può mostrare come un test esaustivo, supposto in grado di eseguire ogni sequenza di esecuzione in un millisecondo, possa impiegare più di tremila anni per essere portato a termine

1.3 Livelli di Testing

Pressman (2001) spiega come il testing possa essere applicato a vari livelli, da un singolo componente software ad un raggruppamento di unità, fino ad arrivare al sistema software nel suo complesso, con tutti i componenti integrati in moduli più grandi.

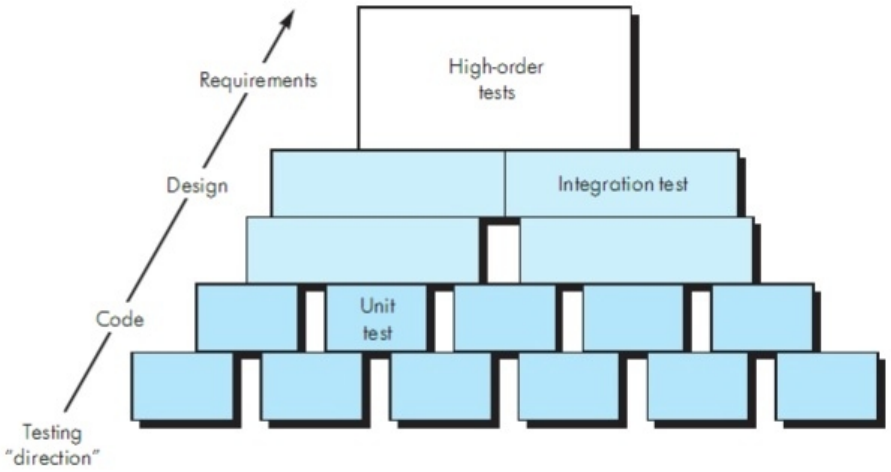


Figura 1.2 – Strategia di collaudo

Come mostrato in figura 1.2, una strategia di collaudo che viene scelta dalla maggior parte dei team di sviluppo adotta una visione incrementale del testing, iniziando con il collaudo delle singole unità del programma, procedendo con i test progettati per facilitare l'integrazione di queste unità e terminando con i test che mettono alla prova il sistema finale. A prescindere, comunque dall'ordine seguito, possiamo distinguere tra:

- **Testing di unità**: consiste nel testare la più piccola unità di progettazione software, il componente o modulo, allo scopo di evidenziarne i difetti. I test sono derivati in base all'esperienza d'uso, sono una responsabilità di chi sviluppa il software e possono essere eseguiti per singole funzioni, classi di oggetti (testing delle classi) o componenti composti che hanno una specifica interfaccia (testing delle interfacce).

- **Testing di integrazione:** è una tecnica sistematica per la costruzione della struttura di un programma a partire dai moduli che sono stati testati singolarmente. Questi vengono aggregati in gruppi che forniscono alcune funzionalità del sistema e fatti lavorare assieme, dopodiché, ai gruppi funzionanti vengono aggiunti nuovi componenti e ripetuti i test per ogni aggiunta (test di regressione).
  Applicata ad un'architettura di sistema organizzata gerarchicamente, le aggregazioni possono essere realizzate con approccio top-down, bottom-up o misto.

- **Testing di sistema** (verifica)**:** si compone di una serie di prove diverse il cui scopo primario è quello di esercitare completamente il sistema informatico nel quale il software deve operare. Vengono analizzate alcune proprietà globali che non hanno senso se viste nell'ambito del singolo modulo integrato.
  Può quindi includere: prove di affidabilità, in cui si forza il sistema a fallire per verificarne la capacità di recupero; prove di sicurezza, in cui si attenta alle caratteristiche di sicurezza del sistema per evidenziare le vulnerabilità; prove di performance, in cui si valutano le prestazioni in condizioni di utilizzo normale; prove di stress, in cui si sollecita il sistema con volumi di carico particolarmente elevati.

- **Testing di convalida** (validazione)**:** si verifica che il sistema funzioni secondo le aspettative del cliente così come sono state stabilite nel documento delle specifiche dei requisiti software (SRS). Può includere *alpha testing*, in cui i test vengono condotti dall'utente sotto il controllo dello sviluppatore, e *beta testing*, in cui gli utenti eseguono il software in un ambiente non controllato dallo sviluppatore, annotano i problemi riscontrati e li comunicano ad intervalli regolari.

1.4 Processi di testing

"Il testing è un'attività critica nell'ingegneria del software e dovrebbe essere eseguito nel modo più sistematico possibile, definendo in maniera chiara i risultati che si attendono e il modo in cui si vuole ottenerli." (Ghezzi, Jazayeri e Mandrioli, 2004, p. 301)

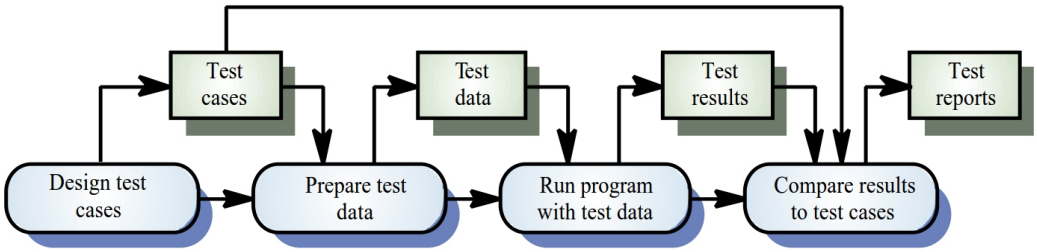


Figura 1.3 - Modello del processo di testing del software.

Come discusso da Sommerville (2007), il processo di testing si può modellare come mostrato in figura 1.3. In esso è possibile notare come il processo sia diviso in diversi passi, ognuno dei quali produce l'input per il passo successivo.

I *casi di test*, prodotti dell'attività di progettazione dei casi di test, sono specifiche degli input del test e degli output attesi dal sistema, oltre ad una dichiarazione di cosa si sta

testando. Per ognuno dei casi di test progettati vengono selezionati gli input effettivi con i quali bisogna esercitare il sistema (dati di test), dopodiché il programma viene effettivamente eseguito con i dati di input scelti. Dopo l'esecuzione, vengono prodotti degli output che devono essere confrontati con quelli attesi per poter capire se un test è fallito o ha avuto successo. I *risultati dei test* sono il prodotto della comparazione tra il comportamento atteso ed il comportamento osservato. Il comportamento atteso è rappresentato dal cosiddetto *oracolo* che può essere automatico o umano.

Una *test suite* si compone di un insieme di casi di test ed un programma è "esercitato" da un caso di test, l'esecuzione di una suite di test consiste nell'esecuzione del programma per tutti i casi di test scelti.

E' buona norma cercare di creare un insieme di test case efficace ed efficiente, cioè in grado di scoprire il maggior numero di difetti possibili, o validare il sistema, con il minor numero possibile di test case. Dato che il testing esaustivo non è fattibile, è necessario seguire delle politiche volte a selezionare alcuni casi di test che cerchino di approssimarlo e che forniscano una sufficiente evidenza che il prodotto avrà un comportamento accettabile anche nelle situazioni in cui non sia stato testato (Ghezzi, Jazayeri e Mandrioli, 2004). Oltre a ciò, bisogna stabilire dei criteri per valutare quando il testing debba essere terminato (criteri di tempo, criteri di costo, criteri di *copertura*, criterio statistico).

1.5 Modalità di testing

Ogni prodotto si può collaudare in due modi come mostrato in figura 1.4. Se si conoscono le funzioni specifiche che il prodotto deve svolgere, si possono effettuare prove tese a dimostrare che tali funzioni siano effettivamente svolte, cercando nel contempo eventuali errori in ciascuna funzione. Se si conosce la struttura interna del prodotto, si possono effettuare prove tese a garantire che le operazioni interne rispettino le specifiche, avendo

cura di mettere alla prova tutti i componenti. Il primo viene denominato collaudo **black-box**, il secondo, collaudo **white-box** (Pressman, 2001).

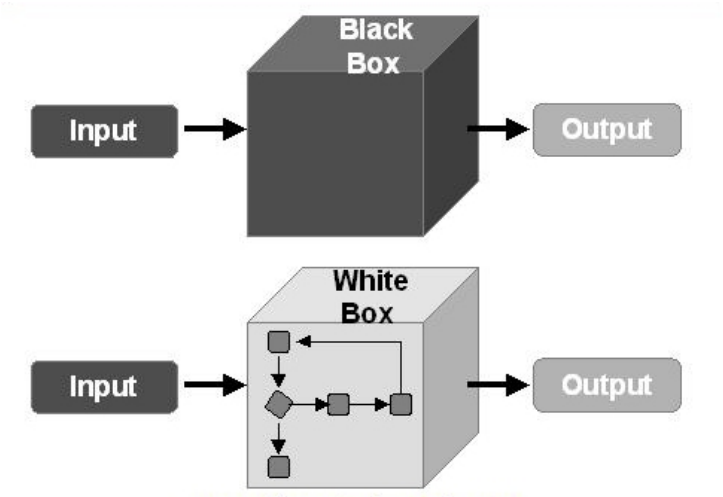


Figura 1.4 – Differenza tra testing White Box e Black Box

White Box e Black Box non individuano da soli alcuna tecnica di testing specifica, ma solo una famiglia di tecniche di testing. Ad esempio: "Testing Funzionale Black Box", "Testing di unità Black Box", "Testing di integrazione white box", "Test di unità White Box" sono tecniche di testing specifiche

1.5.1 Testing Black Box

Per il testing Black Box si esercita il sistema o un componente immettendo degli input e osservando i valori degli output mettendo in evidenza errori nei requisiti, funzionalità errate o mancanti, errori nei meccanismi di interfacciamento, limiti di performance, limiti di carico massimo. Si tiene conto unicamente dell'interfaccia e della documentazione nella progettazione dei test cases e si mette in evidenza il comportamento esterno del software o componente da testare, senza sapere cosa accade all'interno di esso o quale codice viene effettivamente eseguito.

Non esiste una sola tecnica Black Box, possiamo distinguere tra:

- Testing basato sui requisiti: vengono progettati vari test per ogni requisito.
- Testing basato sugli Use Case: noto lo Use Case Diagram e la descrizione di tutti gli scenari dei casi d'uso, per ogni scenario si progetta uno o più test case che lo eseguano (automaticamente o manualmente). La strategia di testing mira alla copertura dei casi d'uso e degli scenari.
- Testing delle partizioni o classi di equivalenza: come mostrato in figura 1.5 e descritto da Sommerville (2007), gli input e gli output del testing possono essere suddivisi in classi, nelle quali ogni membro è correlato con l'altro. Ognuna di queste classi costituisce una partizione di equivalenza, cioè il programma si comporta (verosimilmente) in maniera equivalente per ogni elemento appartenente ad essa. I casi di test dovrebbero essere scelti da ciascuna partizione e la scelta è effettuata applicando specifiche politiche. E' possibile che vengano scelti degli input tali che alcuni blocchi di codice vengano eseguiti più volte mentre altri non riescano mai ad essere eseguiti.
- Testing basato su tabelle di decisione: le colonne della tabella rappresentano le combinazioni degli input a cui corrispondono le diverse azioni, le righe della tabella riportano i valori delle variabili di input (nella Sezione Condizioni) e le sezioni eseguibili (nella Sezione Azioni).

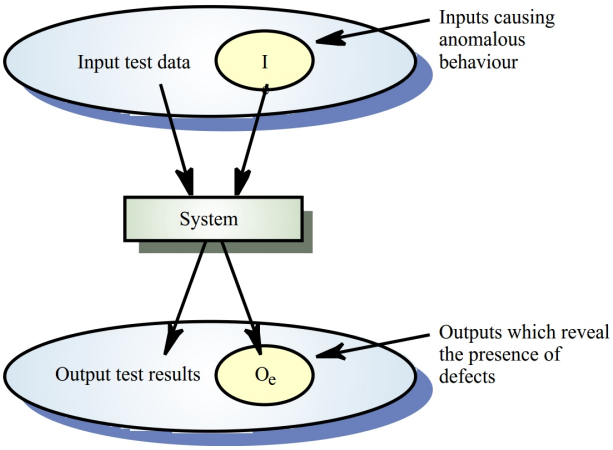


Figura 1.5 – Testing delle partizioni Black Box

1.5.2 Testing White Box

Nel testing White Box, come spiegato da Sommerville (2007), si utilizza la conoscenza del codice sorgente, degli algoritmi utilizzati e della struttura interna per ricavare i dati di test che saranno usati per esercitare il programma ed è per questo motivo che esso viene anche definito “testing strutturale” (vedi Figura 1.6).

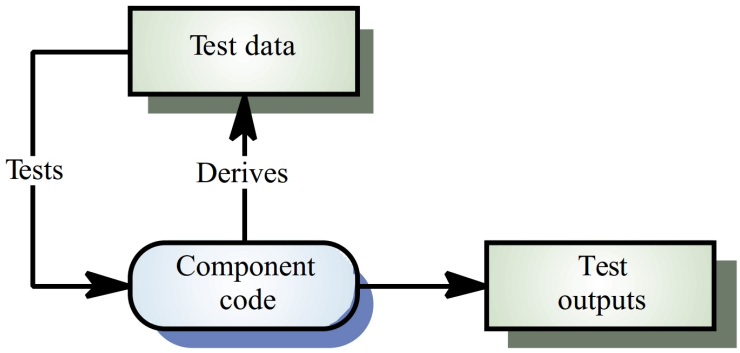


Figura 1.6 - Testing White Box

Tramite il testing White Box si possono formulare criteri di copertura più precisi di quelli formulabili con testing Black Box. I criteri di copertura sono fondati sull’adozione di metodi di copertura degli oggetti che compongono la struttura dei programmi come

istruzioni, strutture controllo, flusso di controllo (vedi figura 1.7). L'obiettivo è quello di assicurare che tutti i blocchi di istruzioni e le condizioni di cui è composto il programma siano eseguite.

I casi di test vengono definiti in modo tale che gli oggetti di una definita classe (istruzioni, archi del CFG, predicati, strutture di controllo, etc.) siano attivati (coperti) almeno una volta nell'esecuzione dei casi di test

1.5.2.1 Control-Flow Graph

Il Control Flow Graph (o CFG) è un modello di rappresentazione dei programmi definibile come:

$$CFG\ (P) = \langle N, AC, nI, nF \rangle$$

Dove:

- <N, AC> è un grafo diretto con archi etichettati.
- $\{nI, nF\} \subseteq N$, $N-\{nI, nF\} = Ns \cup Np$.
    - Ns e Np sono insiemi disgiunti di nodi istruzione e nodi predicato, nI ed nF sono rispettivamente il nodo iniziale e il nodo finale.
- $AC \subseteq N-\{nF\} \times N-\{nI\} \times \{vero, falso, incond\}$ rappresenta la relazione flusso di controllo;
- Un nodo $n \in Ns \cup \{nI\}$ ha un solo successore immediato e il suo arco uscente è etichettato con "incond".
- Un nodo $n \in Np$ ha due successori immediati e i suoi archi uscenti sono etichettati rispettivamente con vero e falso.

Un esempio di CFG per una procedura di nome "Quadrato" è riportato in figura 1.6

```
procedure Quadrato;
  var x, y, n: integer;
  begin
1.   read(x);
2.   if x > 0
     then begin
3.          n := 1;
4.          y := 1;
5.          while x > 1 do
             begin
6.             n := n + 2;
7.             y := y + n;
8.             x := x - 1;
             end;
9.          write(y);
          end;
  end;
```

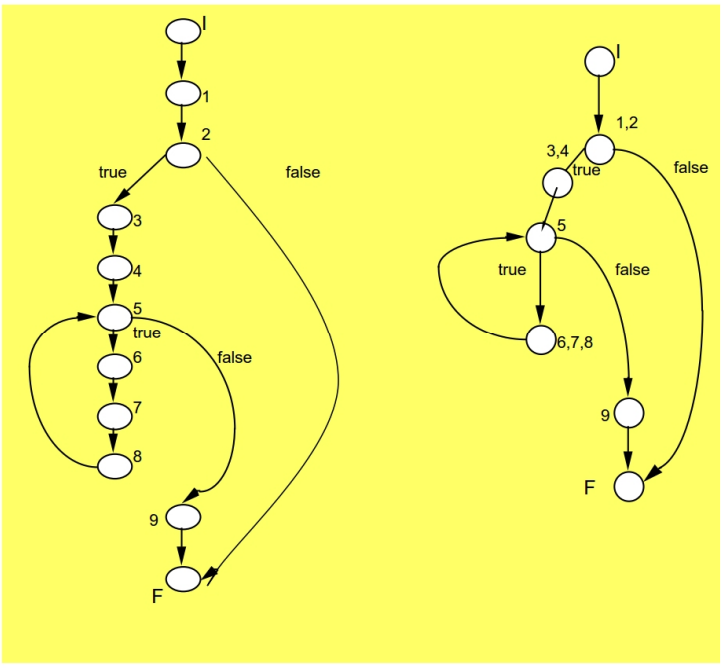


Figura 1.7 - Esempio di CFG

1.6 Automazione del testing

L'automazione del testing, o Test Automation, consiste nell'implementazione e l'uso di tecnologie software che consentono di automatizzare alcune attività del processo di testing (Sommerville, 2007).

Alcune aree di intervento sono:

- Generazione di casi di test.
- Preparazione ed esecuzione del test.
- Valutazione dell'esito dei casi di test.
- Valutazione dell'efficacia potenziale di Test Suite e diverse tecniche di testing.

Il supporto alla costruzione parziale o totale di casi di test ripetibili e consistenti consente di ridurre drasticamente i tempi ed i costi della fase di test design e contestualmente di incrementare l'efficienza del collaudo.

I test case possono essere generati automaticamente dall'analisi della documentazione di analisi (specifica dei requisiti), dall'analisi della documentazione di progetto (Model based testing), dall'analisi statica del codice sorgente, dall'osservazione di esecuzioni reali dell'applicazione (user session testing).

L'esecuzione dei casi di test è la parte più "meccanica" della Testing Automation, l'automatizzazione dell'esecuzione dei casi di test porta innumerevoli vantaggi in termini di tempo risparmiato nell'esecuzione dei test, di affidabilità poiché non c'è rischio di errore umano nell'esecuzione dei test e di riuso (parziale) dei test a seguito di modifiche nella classe di testing. Di solito si scrive codice con framework appartenenti alla famiglia *xUnit.*

La valutazione automatica dell'esito dei casi di test richiede che essi siano definiti e che venga creato un metodo per la loro effettiva valutazione sebbene in alcuni casi l'esito di un caso di test non ha bisogno di essere definito oppure esso può essere definito automaticamente come nel caso del *crash testing* oppure del *regression testing.*

Nel crash testing si testa un software in cerca di eccezioni o errori a run-time che interrompano l'esecuzione. Un esempio è lo *Smoke Testing*, una varietà del crash testing nella quale l'applicazione viene esplorata e navigata il più possibile cercando di causare un crash. Il regression testing si applica in seguito ad un intervento di manutenzione su di un software esistente, dopo il quale il sistema può regredire ("invecchiare") poiché è probabile che la modifica effettuata influisca sul resto del sistema generando nuovi difetti (è il cosiddetto *ripple effect*).

1.6.1 JUnit [6]

Nell'ambito del testing vi è la necessità di un approccio sistematico che separi il codice di test da quello della classe da testare, supporti la strutturazione dei casi di test in test suite, consenta l'esecuzione complessiva di un'intera test suite e fornisca un output separato dall'output dovuto all'esecuzione della classe.

La soluzione ai problemi precedenti è data dai framework della famiglia x-Unit:

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

JUnit è il capostipite della famiglia e fu sviluppato originariamente da Erich Gamma e Kent Beck nell'ambito degli strumenti a supporto dell' eXtreme Programming (XP).
Permette la scrittura di test in maniera ripetibile e di automatizzare l'esecuzione dei casi di test, consentendo di ottenere notevoli vantaggi in termini di tempo risparmiato, di affidabilità e di riuso (parziale) dei test.
JUnit può essere usato sia per il testing in modalità black box che in modalità white box, procedura utile nel caso si voglia valutare anche lo stato interno del software o per utilizzare strumenti di valutazione della copertura di un test come EMMA**.** Può altresì essere utilizzato per il Testing di integrazione, Testing della GUI e Testing di sistema / funzionale.
In alcuni ambienti di sviluppo, come ad esempio Eclipse [7], sono previsti *plug-in* che supportano il processo di scrittura ed esecuzione dei test JUnit su classi Java.

1.6.2 Random testing

Secondo l' *infinite monkey theorem* [8], una scimmia che scrive a macchina battendo tasti a caso in un tempo infinito sarà in grado di scrivere l'Amleto di Shakespeare . Il ragionamento dietro questa supposizione è che input random dati in un tempo infinito dovrebbero produrre tutti i possibili output. In altre parole ogni problema può essere risolto con l'utilizzo di risorse e tempo sufficienti.

Nelle tecniche di Random Testing, allo scopo di testare l'applicazione, vengono generate sequenze casuali di input. Il random testing viene eseguito in modo totalmente automatico può portare alla scoperta di malfunzionamenti che non vengono trovati con altre strategie.
Un esempio è il Monkey Testing [9], una specializzazione del Random Testing al caso di sistemi interattivi, nel quale gli input sono eventi. Con questa tecnica possono essere trovati soltanto crash oppure possono essere valutate condizioni invarianti di malfunzionamento, anche la lunghezza della sequenza può a sua volta essere scelta casualmente. Ce ne sono diverse tipologie come il Dumb Monkey Testing [9], nel quale eventi sono generati in maniera totalmente casuale, secondo una distribuzione uniforme di probabilità; il Brilliant Monkey Testing, nel quale gli eventi e le sequenze di eventi sono generati secondo una distribuzione di probabilità specifica (spesso dipendente da osservazioni precedenti della distribuzione degli eventi di utenti reali), lo Smart Monkey Testing [9], simile al Brilliant Monkey Testing, ma con ulteriori euristiche che possono essere introdotte per evitare ad esempio di ripetere eventi o sequenze di eventi già testate

Capitolo 2

2.1 Problemi del testing

Nel campo del software si ha a che fare con sistemi discreti, di conseguenza piccole variazioni dei valori dei dati in ingresso possono portare a risultati scorretti. Questo problema non si presenta in altri campi dell'ingegneria, dove il testing è semplificato dall'esistenza di proprietà di continuità, grazie alla quale piccole differenze nelle condizioni non producono comportamenti fortemente diversi. Ad esempio, se un ponte resiste ad un carico di 1000 tonnellate, sicuramente resisterà anche a carichi più leggeri (Ghezzi, Jazayeri e Mandrioli, 2004)

Inoltre il settore del testing è tormentato da problemi indecidibili per i quali è possibile dimostrare che non esistono algoritmi che li risolvono.

- Stabilire se l'esecuzione di un programma termina a fronte di un input arbitrario è un problema indecidibile
- Dati due programmi, il problema di stabilire se essi calcolano la stessa funzione è indecidibile (Brainerd Landerweber, 1974). Le conseguenze sono rilevanti: dato un programma e supposto noto e disponibile l'archetipo idealmente corretto di tale programma, non possiamo comunque dimostrare l'equivalenza dei due; non esiste un algoritmo in grado di stabilire se due generici cammini del grafo di flusso di

controllo di un programma (CFG) calcolino la stessa funzione o meno (teorema di equivalenza dei cammini).

- Teorema di Weyuker: Dato un generico programma P i seguenti problemi risultano indecidibili:
  - Esiste almeno un dato di ingresso che causa l'esecuzione di un particolare comando?
  - Esiste un particolare dato di ingresso che causa l'esecuzione di una particolare condizione (branch)?
  - Esiste un dato di ingresso che causa l'esecuzione di un particolare cammino?
  - E' possibile trovare almeno un dato di ingresso che causi l'esecuzione di ogni comando di P?
  - E' possibile trovare almeno un dato di ingresso che causi l'esecuzione di ogni condizione (branch) di P?
  - E' possibile trovare almeno un dato di ingresso che causi l'esecuzione di ogni cammino di P?
- Tesi di Dijkstra: "il testing non può dimostrare l'assenza di difetti, ma solo la loro presenza". Se alla n-esima prova un modulo od un sistema ha risposto correttamente (ovvero non sono stati più riscontrati difetti), non vi è garanzia alcuna che altrettanto possa fare alla (n+1)-esima. Ciò è dovuto all'impossibilità di produrre tutte le possibili configurazioni di valori di input (test case) in corrispondenza di tutti i possibili stati interni di un sistema software.
- La correttezza di un programma è un problema indecidibile.

2.2 Qualità del testing

Come abbiamo visto, dall'attività di testing non si possono trarre certezze assolute circa la correttezza del programma, tuttavia, integrato con altre tecniche di verifica, esso può contribuire a migliorare la fiducia dei progettisti circa le qualità del prodotto che hanno realizzato ed è per questo che il processo stesso di testing dovrebbe possedere alcune caratteristiche intrinseche di qualità.

Come affermato da Ghezzi, Jazayeri e Mandrioli (2004), il test dovrebbe aiutare a localizzare gli errori e non solo a rilevarne la presenza. Il risultato del test non può essere visto come una risposta puramente booleana alla correttezza dell'oggetto sotto test, talvolta alcuni difetti possono essere tollerati e, se si trovano difetti in un modulo, non per forza esso deve essere considerato non corretto.

Una volta rilevati gli errori, i test dovrebbero essere organizzati in modo da aiutare ad isolarli, compito non semplice in quanto un bug può essere "lontano" dal punto in cui il programma ha fallito.

Il test dovrebbe essere *ripetibile*, vale a dire che i test dovrebbero essere costruiti in maniera tale che la ripetizione dello stesso esperimento che fornisca gli stessi dati in ingresso allo stesso frammento di programma produca gli stessi risultati. Questa osservazione sembrerebbe quasi ovvia, tuttavia può succedere che alcuni esperimenti risultino difficilmente replicabili e ciò è causa di notevoli difficoltà nell'attività di debugging. Una delle ragioni della mancanza di ripetibilità nel test del software è dovuta all'ambiente di esecuzione il quale può avere decisiva influenza sulla semantica dei programmi.

Il test dovrebbe essere *accurato*, qualità che ne aumenta l'affidabilità. Il concetto di accuratezza dell'attività di test è strettamente dipendente dal livello di precisione, o addirittura di formalità, delle specifiche del software.

Il testing dovrebbe essere un'attività condotta secondo criteri di *efficacia* ed *efficienza*.

2.2.1 Efficacia ed efficienza

L'efficacia rappresenta la capacità di rilevare errori nel prodotto tramite le attività di test e le tecniche ed i metodi utilizzati per aumentare l'efficacia del test si concentrano su strategie che consentono la scoperta di quanti più difetti possibili.

L'efficienza si riferisce alla capacità di individuare il massimo numero possibile di malfunzionamenti con il minimo sforzo, inteso come impiego di risorse umane, tecnologiche e di tempo. Le tecniche ed i metodi utilizzati per aumentare l'efficienza si basano su strategie in grado di minimizzare l'utilizzo di casi di test per ridurre lo sforzo. A tal proposito si pensi che per il testing sono richiesti circa il 40% dei costi di produzione del software per il raggiungimento di ragionevoli livelli di qualità.

Le varie tecniche di testing abbinano livelli crescenti di efficacia a livelli decrescenti di efficienza. Non è facile trovare il giusto compromesso, l'efficacia va privilegiata quando si vuole un software affidabile (ad esempio per applicazioni critiche), l'efficienza va privilegiata se si vuole ridurre lo sforzo allocato alle attività di testing, in particolare se non può essere eseguito automaticamente.

2.2.1.1 Misura dell'efficacia e dell'efficienza

E' possibile esprimere numericamente l'efficacia e l'efficienza di un'attività di testing. Misurare questi due parametri può essere utile per confrontare direttamente la qualità di diverse tecniche di testing utilizzate per una stessa applicazione.

Efficacia:

$$\frac{\text{Numero di difetti scoperti}}{\text{Numero di difetti esistenti}}$$

Efficienza:

$$\frac{\text{Numero di casi di test rilevanti difetti}}{\text{Numero di casi di test eseguiti}}$$

Per aumentare l'efficacia spesso si aumenta il numero di casi di test eseguiti nella speranza di scoprire un numero elevato di difetti ma in questo modo diminuisce l'efficienza; per aumentare l'efficienza si eseguono soltanto i casi di test corrispondenti ad esecuzioni "critiche" del software diminuendo spesso l'efficacia.

Come si evince dalle formule, l'efficienza del test può essere misurata in termini di numero di casi di test eseguiti, tempo o sforzo necessario per eseguirli. Il numero di test è la grandezza più semplice da misurare ma se il testing è automatico, allora la misura più utile è il tempo di esecuzione del test.

La misura dell'efficacia dei casi di test è invece più problematica, in generale non può essere misurata poiché non sappiamo a priori quanti difetti sono presenti nel software, ciò è possibile solo in ambiente sperimentale nel caso in cui iniettassimo appositamente difetti nell'applicazione allo scopo di valutare l'efficacia della test suite nel rivelare malfunzionamenti. In mancanza di misura diretta dell'efficacia si può ricorrere a misure indirette.

2.2.1.2 Misure indirette: Copertura

Ci sono *metriche* che vengono sviluppate per misurare l'efficacia del collaudo. La principale è l'analisi della *copertura* del codice ottenuta tramite sonde nel codice dell'applicazione che ci permettano di monitorare i percorsi di esecuzione seguiti durante il test. Le istruzioni eseguite almeno una volta sono dette "coperte".

La copertura può essere interpretata come una "speranza" di verifica: se non si esegue la riga difettosa, sicuramente non si riscontrerà il fallimento; in caso contrario c'è la possibilità (non certezza) di riscontrarlo (Ghezzi, Jazayeri e Mandrioli, 2004).

Espressa in termini di copertura, l'efficacia può essere rappresentata attraverso le seguenti formule:

- $\text{CovLoc} = \frac{\text{LOC coperte da almeno un caso di test}}{\text{Totale LOC}}$
- $\text{CovMetodi} = \frac{\text{Metodi coperti da almeno un caso di test}}{\text{Totale Metodi}}$
- $\text{CovClassi} = \frac{\text{Classi coperte da almeno un caso di test}}{\text{Totale Classi}}$

Le tre espressioni di cui sopra esprimono una percentuale di copertura rispettivamente, delle linee di codice, dei metodi e delle classi coperte da un test.
LOC (in inglese "Lines of Code") è l'acronimo generico utilizzato per indicare il numero di linee di codice. Poiché esistono vari tipi di linea di codice come commenti, linee vuote, linee non eseguibili e linee eseguibili, esistono altrettante diverse definizioni. La più accettata è la seguente: una linea di codice è ogni linea di testo di un programma che non sia bianca o un commento, indipendentemente dal numero di istruzioni in essa inclusa. In particolare, tali linee di codice possono contenere intestazioni di unità di programma, dichiarazioni ed istruzioni esecutive.
La copertura si riferisce tecnicamente alle righe di codice eseguibile ma esistono delle difficoltà nel riportarlo al codice sorgente o al modello. Ad esempio la riga:

"if (a>b && x>y && f(&z)==5)"

è spesso tradotta con cicli innestati in assembler. Se in un caso di test il valore di *a* è tale che "*a<=b*", allora la verifica "*x>y*" non verrà effettuata ed "*f(&z)*" non verrà eseguita. Un' ulteriore attenzione bisogna porla per l'esecuzione di *f(&z)*, essa può modificare il

valore di z, per cui l'ordine con cui vengono eseguite le condizioni è anch'esso rilevante sull'esito del confronto. In questi casi, molti programmi convengono nel misurare la copertura delle LOC del codice sorgente in termini frazionari, indicando le linee come coperte in modo parziale.

Per valutare la copertura è richiesta la conoscenza e l'accesso al codice sorgente del programma per inserirvi (possibilmente in maniera automatica) delle sonde che ci permettano di monitorare i percorsi di esecuzione. Il testing è quindi effettuato in modalità white box.

2.3 Strumenti di misura della copertura

A supporto della misura della copertura derivante dall'esecuzione di un'applicazione è possibile utilizzare diversi strumenti, sia disponibili come plug-in di ambienti di sviluppo come Eclipse che stand-alone.

2.3.1 Code Cover [10]

Scritto per Java e Cobol e realizzato dall'università di Stoccarda consente di valutare metriche di copertura: "CodeCover measures statement, branch, loop, term coverage (subsumes MC/DC), question mark operator coverage, and synchronized coverage."

Può essere usato sia per misurare la copertura per esecuzioni di un'applicazione per la quale sia stato attivato preventivamente il monitoraggio, sia per esecuzioni di test case scritti con JUnit.

Mostra molte statistiche riguardanti l'esito dei casi di test e consente di generare report HTML con tali risultati.

Come mostrato in figura 2.1, la copertura delle righe del codice può essere vista anche direttamente sul codice:

- Rosso: non coperta
- Giallo: coperta parzialmente
- Verde: coperta

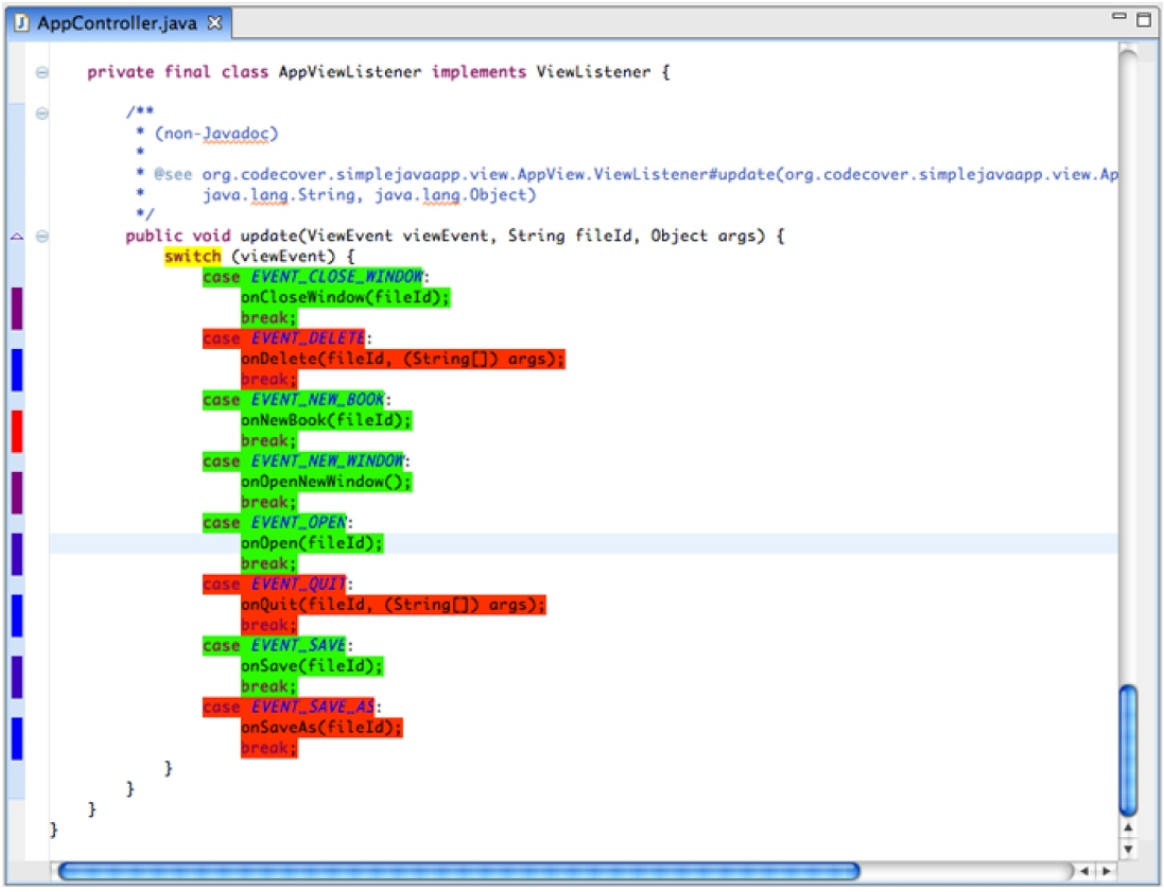


Figura 2.1 - Copertura del codice con CodeCover

2.3.2 EMMA [3]

Emma è uno strumento open-source di instrumentazione del codice sorgente che consente di misurare l'effettiva copertura del codice ottenuta a seguito dell'esecuzione di un insieme di casi di test. Genera report e metriche, anche in formato HTML e risulta molto utile per il testing White Box.

Può essere utilizzato per valutare a posteriori l'efficacia relativa di più test su una stessa applicazione analizzandone le percentuali di copertura.

2.3.2.1 Report HTML di EMMA

Organizzazione sul File System

Per ogni test, i report in HTML generati con EMMA sono memorizzati sul File System come una struttura semplice in cui è presente un file "index.html", contenente informazioni di sommario, ed una cartella di nome "_files" contenente altri file in formato html raggiungibili mediante navigazione interna, a partire dall' indice. In figura 2.2 è mostrato un esempio di report generato tramite EMMA dopo una sessione di testing su un'applicazione Android di nome Omnidroid.

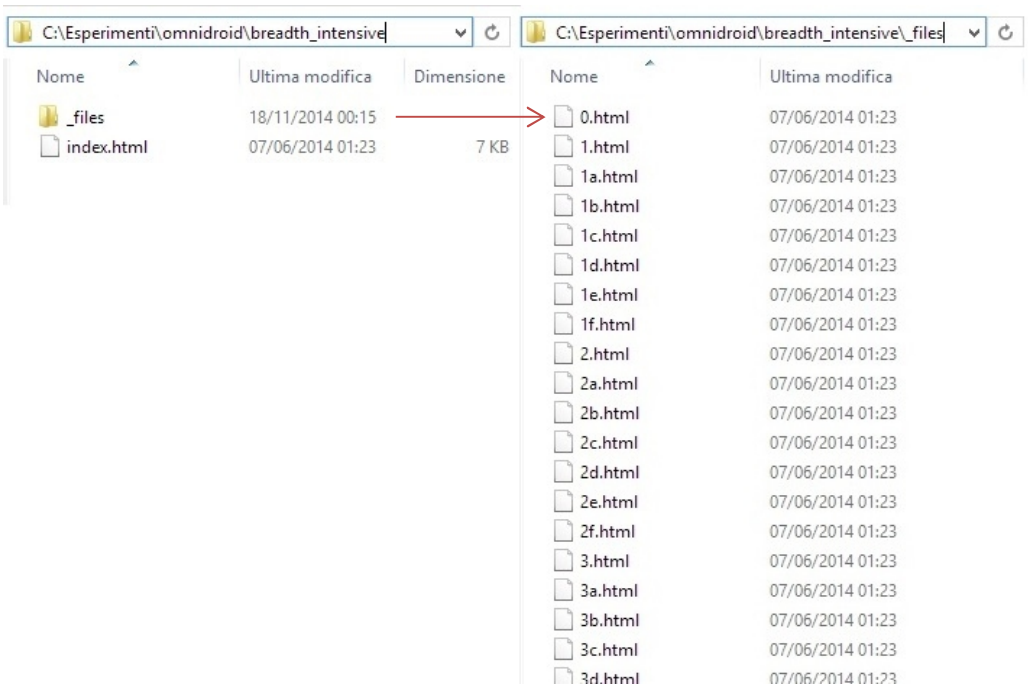


Figura 2.2 - Struttura sul File System del report HTML di EMMA

Navigazione nei report HTML

La navigazione nei report in HTML generati da EMMA segue l'organizzazione strutturale tipica di un programma JAVA.

Nell' "index.html" è elencata una lista di package di cui è composta l'applicazione, per ogni package c'è una pagina HTML nella quale viene elencata una lista di classi che lo compongono e, per ogni classe, c'è una pagina HTML nella quale vengono elencate una lista di classi interne, di metodi e di linee di codice che la compongono (vedi figure 2.4, 2.5, 2.6).

Index.html

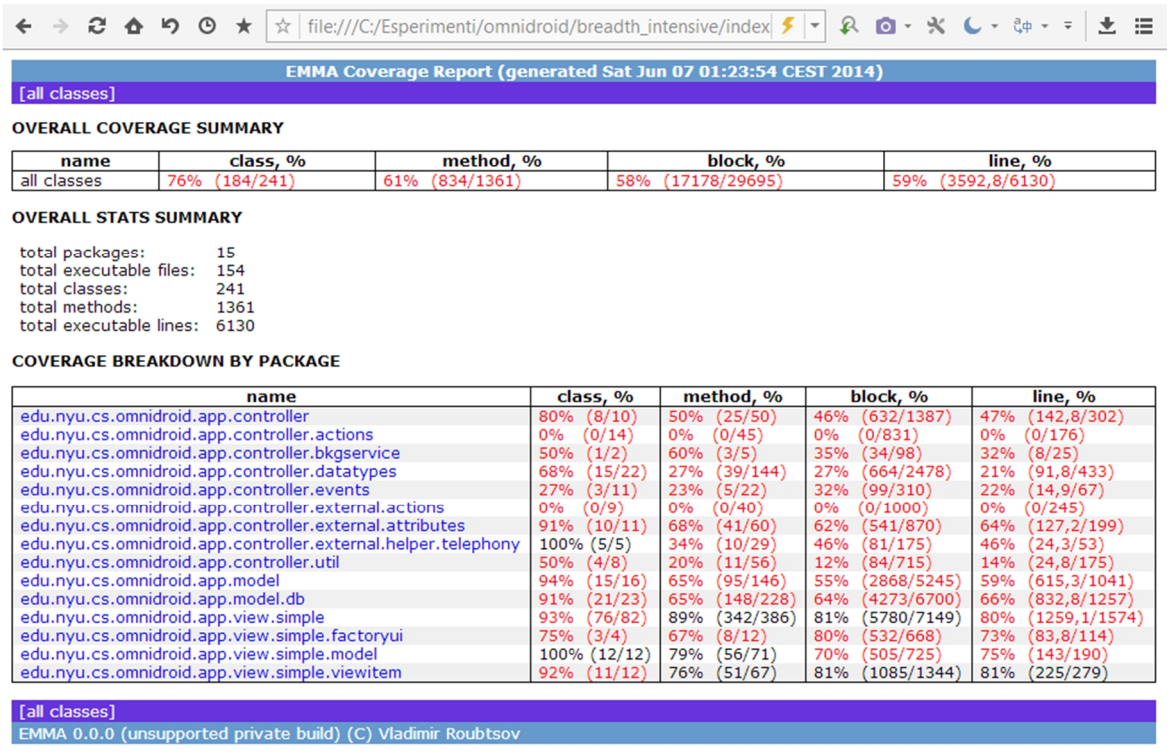


Figura 2.3 - Index.html di un generico output EMMA

Nel file index.html viene riportato (figura 2.3):

1. un sommario, sotto la voce “OVERALL COVERAGE SUMMARY”, sulla percentuale di copertura di tutte le classi, i metodi, i blocchi e le linee di codice componenti il programma.
2. un sommario, sotto la voce “OVERALL STATS SUMMARY”, che riporta il numero totale di package, file eseguibili, classi, metodi e linee di codice eseguibile componenti il programma.
3. la lista di tutti i package, sotto la voce “COVERAGE BREAKDOWN BY PACKAGE”, che compongono l’applicazione, ognuno con il suo sommario di copertura in termini di classi, metodi, blocchi e linee.

Ogni elemento della lista di package è un link che riporta ad una pagina HTML con informazioni più dettagliate relative a quel package.

Pagina html di un generico Package

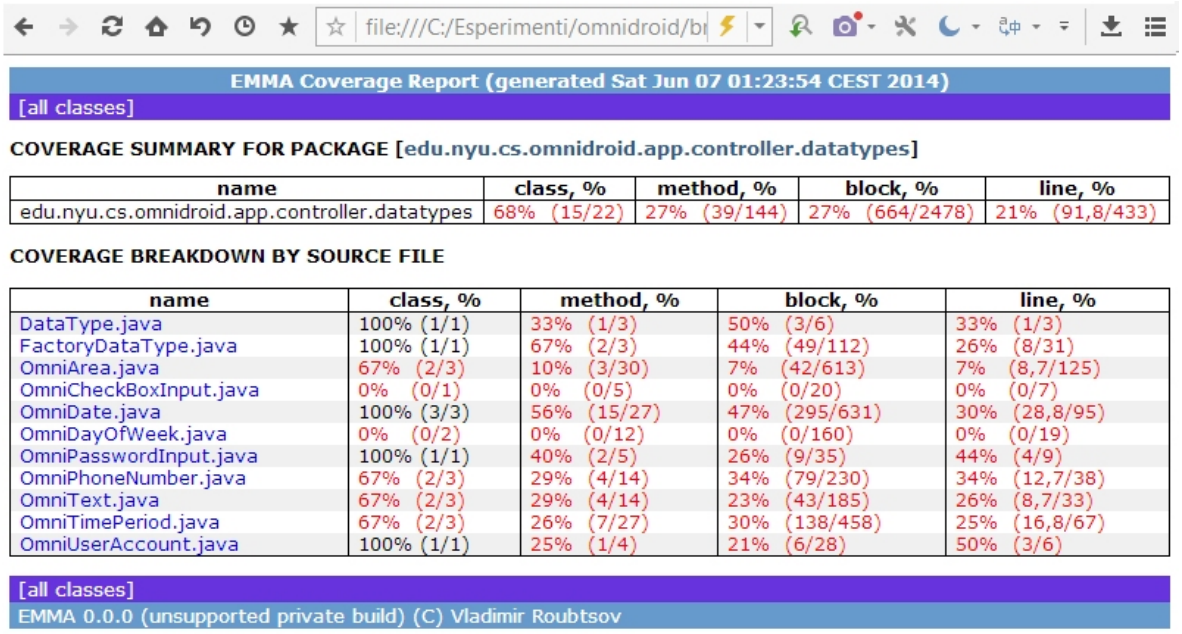


Figura 2.4 - Una pagina html relativa ad un package

Ogni pagina html relativa ai package presenta due sezioni (figura 2.4):

1. un sommario di copertura totale del package sotto la voce “COVERAGE SUMMARY FOR PACKAGE”.
2. una lista di classi che lo compongono, ognuna con relativo sommario di copertura, sotto la voce “COVERAGE BREAKDOWN BY SOURCE FILE”.

Ogni elemento della lista delle classi è link che riporta ad una pagina HTML con informazioni dettagliate relative a quella classe.

Pagina html di una generica Classe

Ogni pagina html relativa ad una classe presenta tre sezioni.

1. un sommario di copertura totale della classe in questione sotto la voce “COVERAGE SUMMARY FOR SOURCE FILE”.
2. una lista di classi (vengono elencate oltre alle classi principali anche le eventuali classi interne) ognuna di esse con la propria lista di metodi sotto la voce “COVERAGE BREAKDOWN BY CLASS AND METHOD”. Ogni elemento della lista ha nella parte destra il suo sommario di copertura. Un esempio per le prime due sezioni è mostrato nella figura 2.5.
3. La terza sezione viene mostrata in figura 2.6, essa è la lista dettagliata di linee di codice componente la classe.

COVERAGE SUMMARY FOR SOURCE FILE [OmniPhoneNumber.java]

name	class, %	method, %	block, %	line, %
OmniPhoneNumber.java	67% (2/3)	29% (4/14)	34% (79/230)	34% (12,7/38)

COVERAGE BREAKDOWN BY CLASS AND METHOD

name	class, %	method, %	block, %	line, %
class OmniPhoneNumber	100% (1/1)	22% (2/9)	27% (45/168)	24% (8/33)
OmniPhoneNumber (String): void		100% (1/1)	87% (27/31)	86% (6/7)
getFilterFromString (String): OmniPhoneNumber$Filter		0% (0/1)	0% (0/4)	0% (0/1)
getValue (): String		0% (0/1)	0% (0/3)	0% (0/1)
isAReferenceTag (String): boolean		100% (1/1)	90% (18/20)	67% (2/3)
isValidFilter (String): boolean		0% (0/1)	0% (0/9)	0% (0/5)
matchFilter (DataType$Filter, DataType): boolean		0% (0/1)	0% (0/44)	0% (0/5)
matchFilter (OmniPhoneNumber$Filter, OmniPhoneNumber): boolean		0% (0/1)	0% (0/23)	0% (0/4)
toString (): String		0% (0/1)	0% (0/3)	0% (0/1)
validateUserDefinedValue (DataType$Filter, String): void		0% (0/1)	0% (0/31)	0% (0/6)
class OmniPhoneNumber$1	0% (0/1)	0% (0/1)	0% (0/19)	0% (0/1)
<static initializer>		0% (0/1)	0% (0/19)	0% (0/1)
class OmniPhoneNumber$Filter	100% (1/1)	50% (2/4)	79% (34/43)	95% (4,7/5)
<static initializer>		100% (1/1)	100% (26/26)	100% (2/2)
OmniPhoneNumber$Filter (String, int, String): void		100% (1/1)	100% (8/8)	100% (3/3)
valueOf (String): OmniPhoneNumber$Filter		0% (0/1)	0% (0/5)	0% (0/1)
values (): OmniPhoneNumber$Filter []		0% (0/1)	0% (0/4)	0% (0/1)

Figura 2.5 – Prime due sezioni di una pagina html relativa ad una classe

```
34  public enum Filter implements DataType.Filter {
35    EQUALS("equals") , NOTEQUALS("not equals");
36
37    public final String displayName;
38
39    Filter(String displayName) {
40     this.displayName = displayName;
41    }
42  }
43
44  public OmniPhoneNumber(String phoneNumber) throws DataTypeValidationException {
45    if (!isAReferenceTag(phoneNumber)) {
46      if (!isWellFormedSmsAddress(phoneNumber)) {
47        throw new DataTypeValidationException("Invalid phone number " + phoneNumber + " provided.");
48      }
49      value = formatNumber(phoneNumber);
50    }
51    value = phoneNumber;
52  }
53
54  /**
55   * Check if the phone number is actually a reference to a data from the intent or external
56   * parameter.
57   */
58  private boolean isAReferenceTag(String str) {
59    if (str == null) {
60      return false;
61    }
62    return str.indexOf("<") == 0 && str.lastIndexOf(">") == (str.length() - 1);
63  }
```

Figura 2.6 – Terza sezione della pagina html relativa ad una classe

2.3.2.2 Percentuale di copertura

Ogni linea di codice può essere colorata in quattro modi: bianco, verde, rosso, giallo.

Le linee di codice bianche sono non eseguibili e pertanto non vengono conteggiate ai fini della misura di coverage. Gli altri tre tipi di linea sono considerate eseguibili e ad ognuna di esse viene assegnato un peso.

- Le linee verdi sono le linee di codice che sono state “coperte” durante l’esecuzione del test ed il loro peso è pari a uno.
- Le linee rosse sono le linee di codice che non sono state “coperte” durante l’esecuzione del test ed il loro peso è pari a zero.
- Le linee gialle sono linee di codice coperte solo parzialmente durante l’esecuzione del test, tipicamente corrispondono ad espressioni condizionali o cicli (non sempre). Il loro peso è pari ad un valore compreso strettamente tra zero e uno.

Di ogni singolo elemento del programma, come può essere un metodo, una classe, un package o l’intero programma, viene espressa una percentuale di copertura, essa viene calcolata come rapporto tra la somma dei pesi delle linee di codice eseguibili e numero totale di linee di codice eseguibili componenti l’elemento.

Capitolo 3

Progettazione della soluzione

Nei prossimi paragrafi viene presentato il modo in cui è stato prima progettato e poi implementato il sistema software realizzato, partendo da una panoramica generale degli obiettivi che il software si prefigge di raggiungere e da una specifica del problema che il software risolve per finire con una descrizione dettagliata della soluzione dal punto di vista concettuale e implementativo.

3.1 Obiettivo

Il software sviluppato si prefigge come obiettivo quello di riuscire a correlare tra loro i report HTML dei test prodotti tramite EMMA, secondo specifiche necessità di analisi prestabilite (differenze, unione, intersezione, complemento). Esso deve supportare la persistenza delle informazioni attraverso un database al fine di consentire di aggiungere al confronto altri esperimenti in tempi diversi. Dovrebbe inoltre supportare l'aggiunta di nuove modalità di confronto oltre a quelle predefinite nella prima versione del software, in modo da venire incontro a nuove necessità di analisi e confronto.

3.2 Problema

Sviluppare un sistema software che serva da strumento di elaborazione a supporto di misure comparative complesse e dettagliate di processi di testing di applicazioni scritte in linguaggio Java, mediante l'analisi del coverage delle linee di codice prodotti a seguito dei test condotti su di esse.

Dopo ogni singolo processo di testing, viene prodotto un file HTML tramite il tool EMMA, che descrive in modo sia sintetico che dettagliato la copertura ottenuta dall'esecuzione del test. I rapporti di copertura riproducono la struttura dell'applicazione in termini di package, classi, metodi e linee di codice e, navigando nelle pagine HTML, è possibile osservare un sommario di copertura di ognuno di questi componenti.

La copertura delle singole linee di codice è immediatamente visibile in quanto esse si presentano evidenziate di diversi colori: verde per linee coperte totalmente, giallo per linee coperte parzialmente, rosso per linee non coperte. Le linee bianche sono linee non eseguibili, ininfluenti ai fini del calcolo del coverage totale. Ad ogni colore è associato un peso che vale: uno per le linee verdi, un valore compreso tra zero e uno per le linee gialle e zero per le linee rosse.

La somma dei pesi delle singole linee di codice eseguibili rappresenta il coverage totale del test eseguiti sull'applicazione in esame. Esso è altresì riportato in termini percentuali, espressione del rapporto tra il valore totale della copertura delle linee di codice e il numero totale di linee di codice eseguibile.

E' richiesto un sistema in grado di analizzare i file HTML dei report EMMA ed estrarne tutte le informazioni riguardanti la copertura del test che esso rappresenta, al fine di memorizzarle in modo permanente in un database. Il sistema deve inoltre mettere a disposizione uno strumento di interrogazione del database che consenta di realizzare delle correlazioni predefinite di tipo insiemistico, di modo che sia possibile ottenere

informazioni dettagliate e sintetiche sulla copertura della differenza semplice tra due test e sulla copertura dell'unione, dell'intersezione e del complemento tra due o più test. Ogni singola funzione di correlazione deve rappresentare un "entry-point" del sistema, affinché siano richiedibili singolarmente ed in modo indipendente da linea di comando.

I risultati di una correlazione devono poter essere analizzati consultando un report testuale che deve essere generato per essa e salvato in una cartella avente il nome della correlazione effettuata. Nel report testuale, risultato della correlazione richiesta (tra più report HTML precedentemente salvati nel database), si dovranno presentare informazioni circa il tipo di correlazione che esso rappresenta, quali sono i test e l'applicazione di cui si richiede il confronto, una statistica sintetica di copertura totale, risultato della somma dei pesi delle linee di codice che risultano dalla correlazione dei test, ed un elenco dettagliato delle singole linee di codice risultanti.

3.3 Analisi dei Requisiti

Come si evince dal problema, il sistema richiesto deve offrire due funzioni fondamentali:

1. L'estrazione dei dati dei report HTML generati con EMMA e la loro archiviazione in un DBMS.
2. L'interrogazione del DBMS con specifiche query, in grado di correlare tra loro due o più test, secondo richiesta dell'utente, e conseguente generazione di un report che ne descrive il risultato.

Come mostrato in figura 3.1, per quanto attiene alla prima funzione, Il sistema deve accettare come input i percorsi dove sono memorizzati i report HTML, estrapolarne le informazioni e caricarle in un database.

Figura 3.1 – Prima funzionalità del sistema

Un descrizione della seconda funzione può essere invece osservata nella figura 3.2. Per valutare la correlazione tra due o più test memorizzati precedentemente nel Database, è necessario effettuare delle query specifiche a seconda del tipo di correlazione voluta. Il risultato delle correlazione sono quindi resi consultabili riportandoli su file testuali di tipo *txt*.

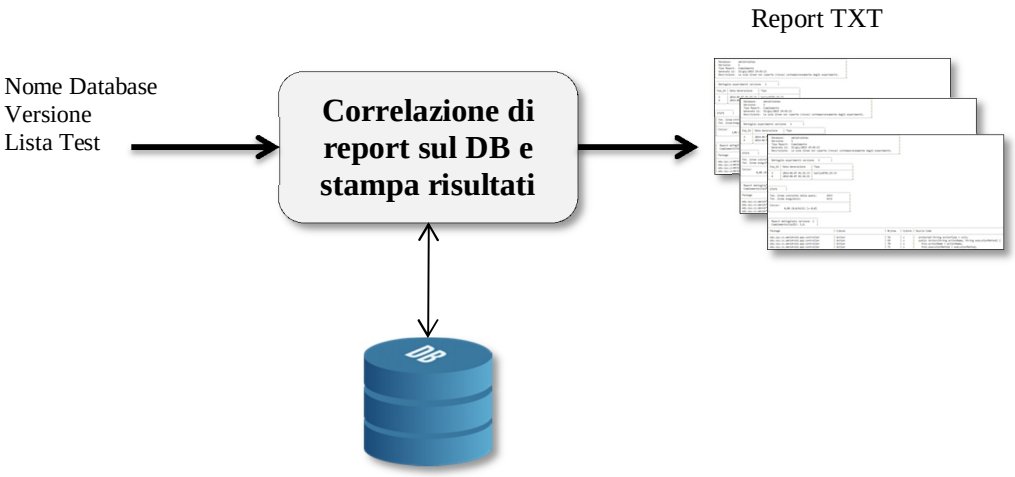


Figura 3.2 – Seconda funzionalità del sistema

Le due situazioni di cui sopra descrivono il flusso dei dati che dall'esterno del sistema sono trasmessi al database e poi di nuovo all'esterno sotto forma di report testuali.

Per quanto riguarda la descrizione del sistema e di come esso viene percepito da un utente esterno possiamo riferirci alla figura 3.3, nella quale viene mostrato un diagramma use-case in cui si evidenziano i confini del sistema e le funzioni che offre ad i suoi utilizzatori.

Le due funzionalità generali saranno discusse poi nei dettagli più avanti, quando verranno descritti i singoli casi d'uso.

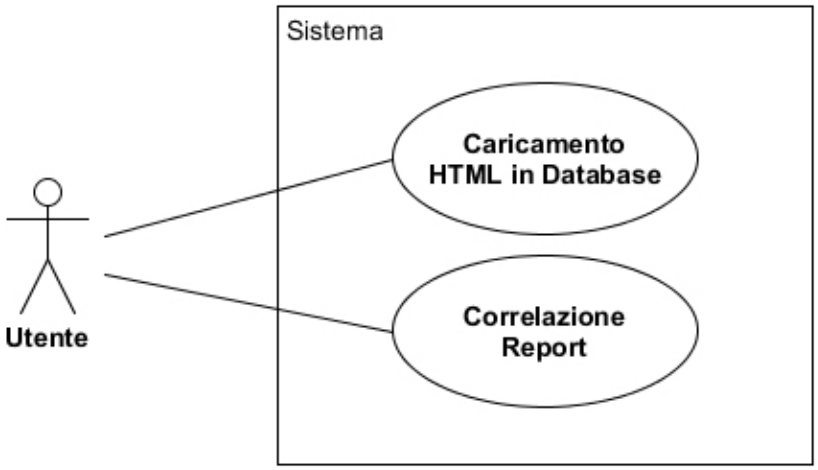


Figura 3.3 – Use Case del Sistema

3.4 Progettazione della base di dati

Dal paragrafo precedente si evince come la realizzazione del sistema non può prescindere dalla progettazione ed implementazione di un database, il quale deve essere gestito attraverso l'utilizzo di un DBMS.

Nel database dovranno essere trasferiti tutti gli elementi riguardanti il dominio del problema, in esso dovrà essere ricreata una rappresentazione completa dell' applicazione testata e dei report di copertura di ogni test effettuato su di essa. Dunque è necessario, in una prima fase, ricavare una serie dati elementari ma che siano rilevanti, al fine poi di farli confluire nel database rispettando un certo schema logico che li organizzi in entità complesse e ne descriva le connessioni.

3.4.1 Schema concettuale della base di dati

Iniziamo col vedere i concetti che devono essere memorizzati nel database e che vengono fuori da un'analisi del problema descritto nel paragrafo 3.2. Si possono evidenziare una serie di dati elementari fondamentali, i quali sono ricavabili da due fonti diverse: i report HTML e altri input dell'utente.

Tali dati sono:

- Nome dell'applicazione
- Versione dell'applicazione
- Nome dei package per ogni versione
- Nome delle classi per ogni package
- Nome delle classi interne per ogni classe
- Nome dei metodi per ogni classe
- Numero di linea e testo delle linee di codice per ogni classe
- Colore e peso delle linee di codice per ogni esperimento di test
- Data e tipo di un test

Questi devono essere poi correlati tra loro per formare gli schemi che saranno presenti nel database.

Per rappresentare adeguatamente i concetti in gioco, la base di dati deve mantenere in memoria da un lato le informazioni della struttura di un'applicazione JAVA e dall'altro le informazioni circa i test HTML che di volta in volta sono condotti su tale applicazione.

Nella figura 3.4 viene mostrato un modello concettuale semplificato, che tiene conto delle entità da rappresentare nel database.

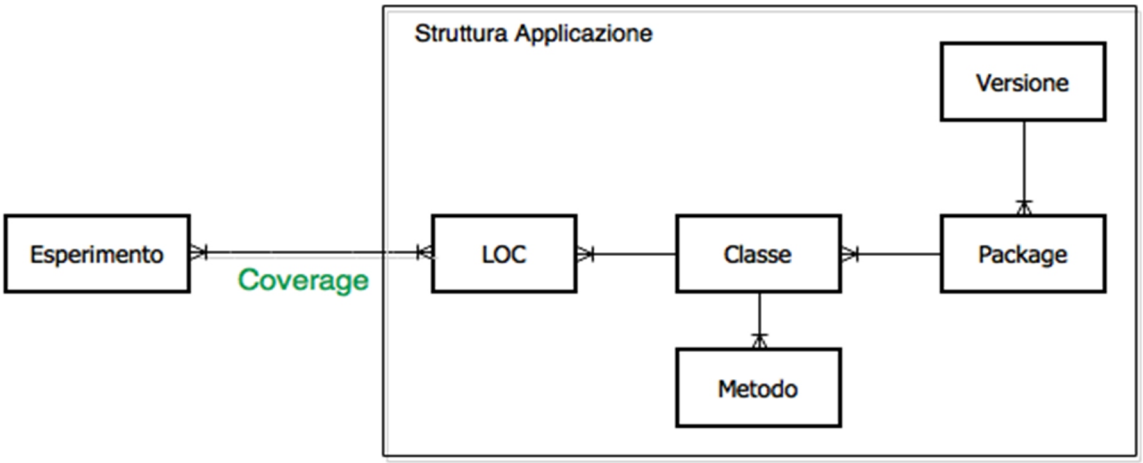


Figura 3.4 – Schema concettuale della base di dati

Le associazioni che possiamo vedere tra le entità che modellano la struttura dell'applicazione sono di tipo uno a molti e descrivono il modo in cui si compone una generica applicazione JAVA: partendo dall'entità "versione", un'applicazione può essere composta da più versioni, ogni versione può essere composta da più package, che possono essere composti da più classi, le quali, infine, possono essere composte da più metodi e da più linee di codice.

Più interessante è notare l'associazione che intercorre tra le entità "Esperimento", il quale modella un test effettuato su un'applicazione, e "LOC" (linee di codice). Essa è di tipo molti a molti ed è nominata "coverage" per il suo fondamentale significato: più test possono "coprire" una linea di codice e più linee di codice possono essere "coperte" da un solo test.

3.4.2 Schema logico MySQL [11]

Facciamo un salto in avanti per discutere dell'implementazione operata per gestire i database. In tal modo è possibile mostrare direttamente come sono stati organizzati i dati elementari dei concetti espressi nello schema concettuale di figura 3.4.

La scelta del DBMS con il quale vengono gestiti i database è ricaduta su MySQL e nella figura 3.5 viene mostrato come lo schema concettuale è stato tradotto in schema logico attraverso la modellazione del diagramma ER nel Workbench MySQL e come sono distribuiti i dati elementari nelle varie tabelle che traducono le entità dello schema concettuale.

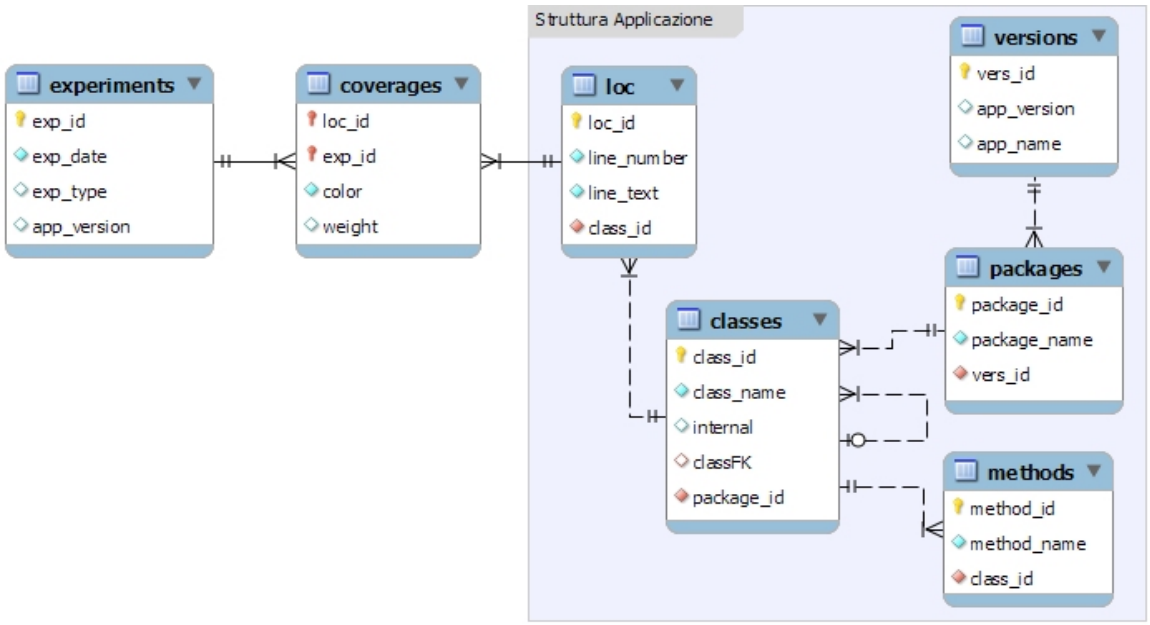


Figura 3.5 – Schema logico della base di dati in MySQL

La differenza più evidente rispetto allo schema concettuale è l'introduzione della tabella "coverages" utilizzata per tradurre l'associazione "molti a molti" tra l'entità Esperimento e l'entità LOC. Tale tabella è di fondamentale importanza per la realizzazione delle funzioni di correlazione e di relativo calcolo del coverage del sistema, in essa sono memorizzati tutti i valori di copertura, in termini di colore (attributo *color*) e peso (attributo *weight*), di ogni test per ogni linea di codice componente l'applicazione sotto test.

Ad ogni versione è associata l'intera parte strutturale della rappresentazione di una applicazione, quindi ad ogni nuova versione corrisponderanno nuovi record delle tabelle "packages", "classes", "methods", "loc".

3.5 Architettura di sistema

Partendo dalla specifica dei requisiti è possibile iniziare a tracciare una rappresentazione di massima dell'architettura del sistema realizzato. Come mostrato in figura 3.6, esso si compone sostanzialmente di due sottosistemi che hanno il compito di realizzare le rispettive funzionalità richieste.

Il primo sottosistema si occupa di gestire l'estrazione dei dati dei report HTML ed il caricamento in un Database dedicato, il secondo sottosistema invece si occupa di fornire degli strumenti predeterminati di interrogazione del Database e di fornire un report che mostra i risultati delle interrogazioni.

Lo scopo che il sistema si prefigge può essere raggiunto solo se vengono utilizzate entrambe le funzionalità, ciò significa che i due sottosistemi devono essere usati in sinergia se si vuole che il sistema produca report di correlazione tra più esperimenti in HTML.

Lo stile organizzativo utilizzato per il sistema è basato su un modello a repository (cfr. Sommerville, 2007), necessario per gestire la grande quantità di informazioni che i sottosistemi devono scambiarsi. In particolare, il secondo sottosistema è strettamente dipendente dal primo, infatti senza di esso non sarebbe disponibile alcun database tramite il quale produrre report di correlazione. Il primo sottosistema è indipendente dal secondo ma usarlo da solo non è utile ai fini della risoluzione del problema che il sistema deve risolvere.

Figura 3.6 – Architettura del Sistema

3.5.1 Scomposizione modulare dei sottosistemi e stili di controllo

Ogni sottosistema è stato suddiviso in moduli per gestire efficacemente le diverse fasi della gestione degli input, elaborazione dei dati e produzione dell'output che vengono attraversate nell'utilizzo delle rispettive funzioni.

I moduli di cui sono composti il primo e secondo sottosistema sono mostrati rispettivamente nelle figure 3.7 e 3.8, nelle quali si è utilizzato un modello ad oggetti in cui vengono descritte solamente le responsabilità generiche assegnate ad ogni modulo.

Per il primo sottosistema sono stati individuati tre moduli fondamentali:

1. un **gestore degli input** che consenta di controllare e filtrare i dati che provengono dall'ambiente esterno,
2. un **parser di file HTML** per estrapolare i dati dai report generati con EMMA,
3. un **gestore del database** per consentire la creazione, il controllo e l'inserimento dei dati ricavati dal parsing in un database.

Altrettanti moduli sono stati individuati per il secondo sottosistema:

1. un **gestore degli input** con le stesse funzioni di controllo e filtro dei dati immessi per il sottosistema,
2. un **gestore di query** che permetta l'esecuzione delle query specifiche per il calcolo delle correlazioni richieste in input,
3. un **printer dei risultati** che sia in grado di stampare su file il risultato delle query.

Per ogni sottosistema lo *stile di controllo* utilizzato per coordinare l'esecuzione dei moduli è quello *centralizzato* con modello *call-return* (cfr. Sommerville, 2007). Per tale motivo, per ogni sottosistema, è stato aggiunto un modulo che ha sostanzialmente il compito di gestire l'esecuzione degli altri moduli elencati in precedenza. Questi sono **StoreController** per il primo sottosistema e **ReportController** per il secondo.

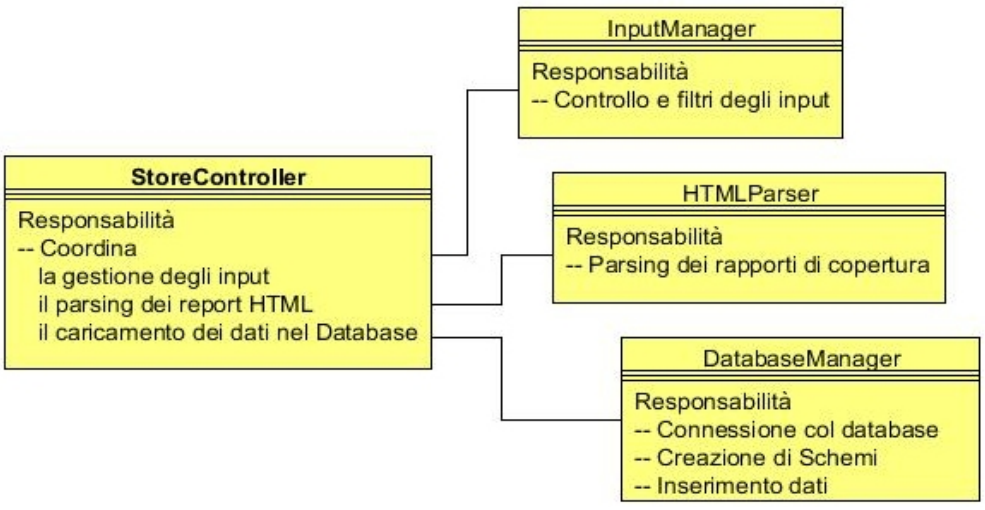


Figura 3.7 – Moduli del sottosistema 1

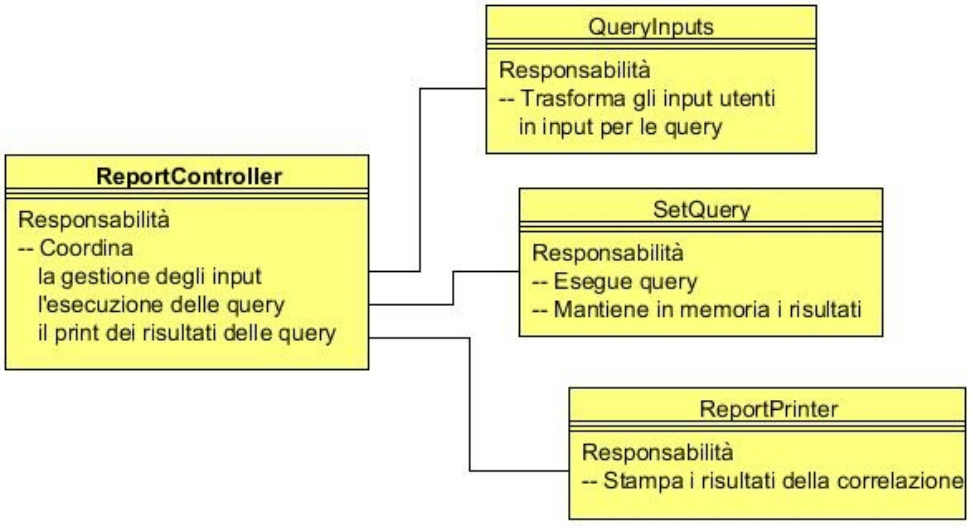


Figura 3.8 – Moduli del sottosistema 2

3.6 Diagrammi dei casi d'uso e di sequenza

Ad ognuna delle due funzioni del sistema vengono dedicati due sottoparagrafi, uno relativo ad un diagramma del caso d'uso, che descrive più dettagliatamente la modalità di utilizzo del sottosistema da parte di un utente, ed un altro relativo ad un sequence diagram, che mostra come i moduli, presentati nel paragrafo precedente, collaborino fra di loro per realizzare la funzionalità del sottosistema cui appartengono. I diagrammi dei casi d'uso esplicitano nei dettagli il significato di quello presentato in figura 3.3 del paragrafo 3.3.

Per entrambe le funzionalità, l'attore che inizia i casi d'uso viene descritto genericamente come "Utente". Esso rappresenta sia una classe di persone fisiche, le quali sono tipicamente dei *tester* (ma con un minimo tempo di apprendimento anche utenti comuni), che un altro sistema software, il quale si può interfacciare col sistema inviandogli i dati di input di cui esso ha bisogno per funzionare.

3.6.1 Caricamento dei report HTML in un Database. Caso d'uso.

Il caso d'uso mostrato in figura 3.9 si riferisce al requisito che il primo sottosistema deve soddisfare.

Per questa funzionalità sono previste un modo di utilizzo sia interattivo che automatico. La versione interattiva dà delle possibilità di gestire in modo più articolato l'inserimento degli input, consentendo maggiore libertà di azione all'utente se si verificano alcuni casi particolari.

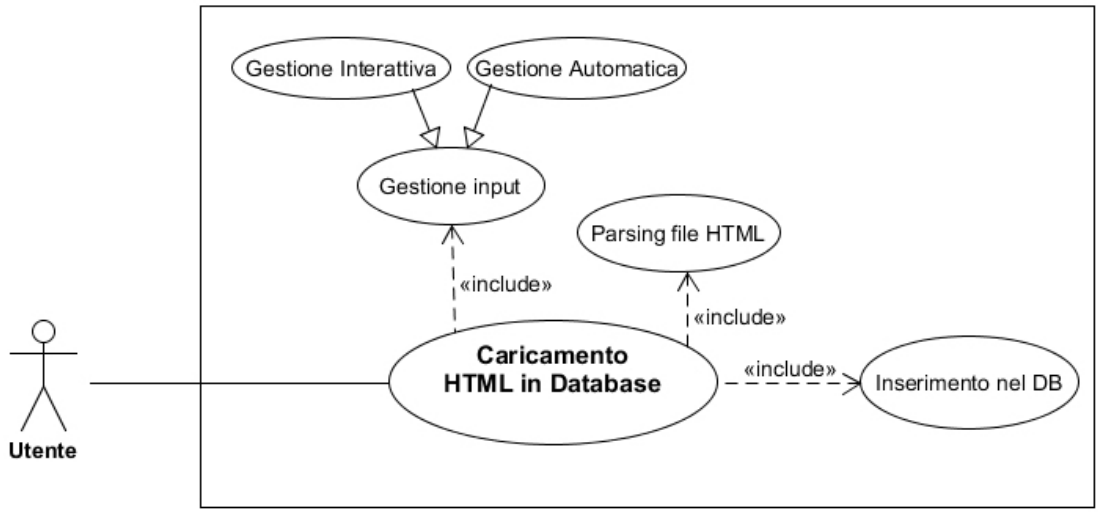


Figura 3.9 – Caricamento HTML in Database. Use case.

Il caso d'uso mostrato in figura viene descritto schematicamente di seguito.

Attori coinvolti

Utente generico.

Descrizione generale della funzione.

Caricamento dei report HTML generati con EMMA in un database.

Input

1. Nome dell'applicazione. Obbligatorio.
2. Versione dell'applicazione. Obbligatorio.
3. Lista dei path dove sono memorizzati i report HTML sul File System. Almeno uno obbligatorio.
4. Per ogni path, nome del tipo di test descritto nel report HTML che esso contiene. Facoltativo.

Pre-condizioni

Deve essere installato e avviato un server Sql sul pc dove viene eseguito il tool.

Descrizione del processo

Caso d'uso principale*: **Caricamento HTML in Database***

Sequenza di operazioni normale:

1. L'utente sceglie la modalità con la quale interagire col sistema, tra interattiva e automatica.
2. Il sistema si avvia nella modalità precedentemente stabilita dall'utente.
3. **include** (Gestione Input):

   Il sistema acquisisce dall'utente e verifica i seguenti input: il nome dell'applicazione, la versione dell'applicazione, i path dei report HTML ed il tipo di test per ogni path.
4. **include** (Parsing file HTML):

Per ogni path indicato in input, il sistema analizza ed estrapola le informazioni dai file HTML presenti al suo interno, circa la struttura dell'applicazione testata e i coverage delle linee di codice del test eseguito su di essa (data del test, nome dei package, delle classi e dei metodi; numero di linea, colore, peso e source code di ogni linea di codice).

5. **include** (Inserimento nel DB)

   Il sistema inserisce nel database sia le informazioni estrapolate dai report HTML sia alcune di quelle inserite in input (nome dell'applicazione, versione dell'applicazione, nome dei tipi di test)

Caso d'uso incluso: **Gestione input interattiva**

Percorso normale:

3.1 Il sistema richiede all'utente il nome dell'applicazione.

3.2 L'utente indica il nome dell'applicazione.

3.3 Il sistema controlla se un database con lo stesso nome esiste già nel database.

3.4 Il sistema richiede all'utente la versione dell'applicazione.

3.5 L'utente indica la versione.

3.6 Se un database dedicato all'applicazione esiste, il sistema controlla se la versione digitata è già memorizzata in esso.

3.7 Se il database non è già esistente, il sistema ne crea un nuovo con il nome dell'applicazione, secondo il modello descritto nel paragrafo 3.4.2..

3.8 Il sistema richiede di inserire un path dove sono presenti i report HTML generati tramite EMMA.

3.9 L'utente inserisce in input il percorso dove è localizzato un report HTML su File System.

3.10 Il sistema effettua i seguenti controlli sul dato inserito:

   i. Viene controllato se il percorso esiste sul File System.

ii. Viene controllato che il percorso contenga report HTML generati con EMMA

iii. Viene controllato che non si tenti di inserire due report identici nello stesso database

3.11 Il sistema richiede il tipo di test relativo al path appena digitato.

3.12 L'utente inserisce il tipo di test.

3.13 Vengono ripetuti i punti 3.8 – 3.12 finché l'utente non decide di terminare l'inserimento.

Percorsi alternativi:

3.3 Se un database con lo stesso nome dell'applicazione è già esistente, il sistema controlla che sia stato creato in una sessione precedente di esecuzione. In caso negativo si da la possibilità all'utente di cancellare il database per ricrearne uno nuovo oppure di cambiare il nome indicato in input.

3.6 Se una stessa versione è già memorizzata l'utente viene informato e viene data la possibilità di cambiare versione o confermare la scelta.

3.8 Se l'utente digita "fine" il programma termina la fase di inserimento ed inizia quella di elaborazione dei dati.

3.10 Se un path dato in input non rispetta i vincoli specificati nei punti i, ii, iii, il sistema lo scarta, informa l'utente di quale dei tre errori si tratta e richiede di inserire un altro path.

3.12 Se il tipo di test viene omesso, il sistema inserisce una versione abbreviata della data di generazione del test.

Caso d'uso: **Gestione input automatica**

Percorso normale:

3.1 L'utente, contestualmente all'avvio del sistema, invia i seguenti input nell'ordine: nome dell'applicazione, versione, lista di path e tipo di test per ogni path dove sono memorizzati i report di test in HTML generati con EMMA.

3.2 Il sistema effettua verifiche su tutti gli input indicati.

Percorsi alternativi:

3.2.1 Se un database con lo stesso nome è già esistente, viene controllato che sia stato creato con questo stesso tool. In caso negativo viene lanciato un messaggio di errore ed il programma termina.

3.2.2 Se il database non è già esistente, il sistema ne crea uno con il nome dell'applicazione, secondo il modello descritto nel paragrafo 3.4.2.

3.2.3 Un path inserito in input viene scartato se non esiste sul File System o non contiene report HTML generati da EMMA o è già stato inserito in input precedentemente.

3.2.4 Se il tipo di test per un path viene omesso, il sistema inserisce una versione abbreviata della data di generazione del test.

Caso d'uso: **Parsing file HTML**

Percorsi alternativi:

4.1 Se il sistema si accorge che un file HTML di un path indicato in input non è analizzabile, viene segnalato l'errore e viene interrotto il processo di analisi di altri eventuali file HTML presenti nello stesso path.

Dopodichè l'utente viene informato dell'errore e si prosegue con l'analisi del prossimo path nella lista.

Caso d'uso: **Inserimento nel DB**

Percorsi alternativi:

5.1 Se il database e la versione sono già esistenti, vengono inserite solo le informazioni di coverage e dei test (colore e peso di ogni linea di codice e tipo e data del test).

5.2 Per ogni report HTML analizzato da inserire nel database, se il sistema si accorge che un report inserito non si riferisce alla versione o applicazione indicati in input, viene lanciato un messaggio di errore per informare l'utente del problema occorso e si prosegue con l'inserimento dei dati del successivo report HTML nella lista.

Post-condizioni

Tutti i dati dei report HTML validi inseriti in input, sono memorizzati in un database creato ad-hoc per l'applicazione a cui i report HTML si riferiscono. Il nome del database è lo stesso del nome dell'applicazione indicato dall'utente in input ed esso è creato rispettando il modello mostrato nel paragrafo 3.4.2. La parte strutturale dell'applicazione ricostruita nel database, afferisce tutta alla versione indicata in input dall'utente.

3.6.1.1 Caricamento dei report HTML in un Database. Sequence Diagram.

In figura 3.10 viene descritto uno scenario d'uso nel quale si fornisce una descrizione del caso d'uso del paragrafo precedente dal punto di vista delle sequenza temporale di esecuzione. In particolare la figura mostra lo scenario in cui l'utente utilizza la versione interattiva del sistema. I nomi degli oggetti della figura rispecchiano quelli utilizzati in figura 3.7, quando si è parlato della scomposizione modulare del primo sottosistema.
L'oggetto di tipo StoreController si occupa di gestire tutto il sottosistema e si può notare come nella fase iniziale, dopo aver creato gli altri oggetti necessari, esso deleghi ad un oggetto di tipo InputManager la gestione dell'interazione con l'utente.

In questa fase l'InputManager si serve anche del DbManager per effettuare vari i controlli sui dati inseriti, come descritto anche nello use case del paragrafo precedente. In particolare si noti come, nel caso in cui un database con lo stesso nome dell'applicazione non esista, viene richiesto al DbManager di crearne uno nuovo.

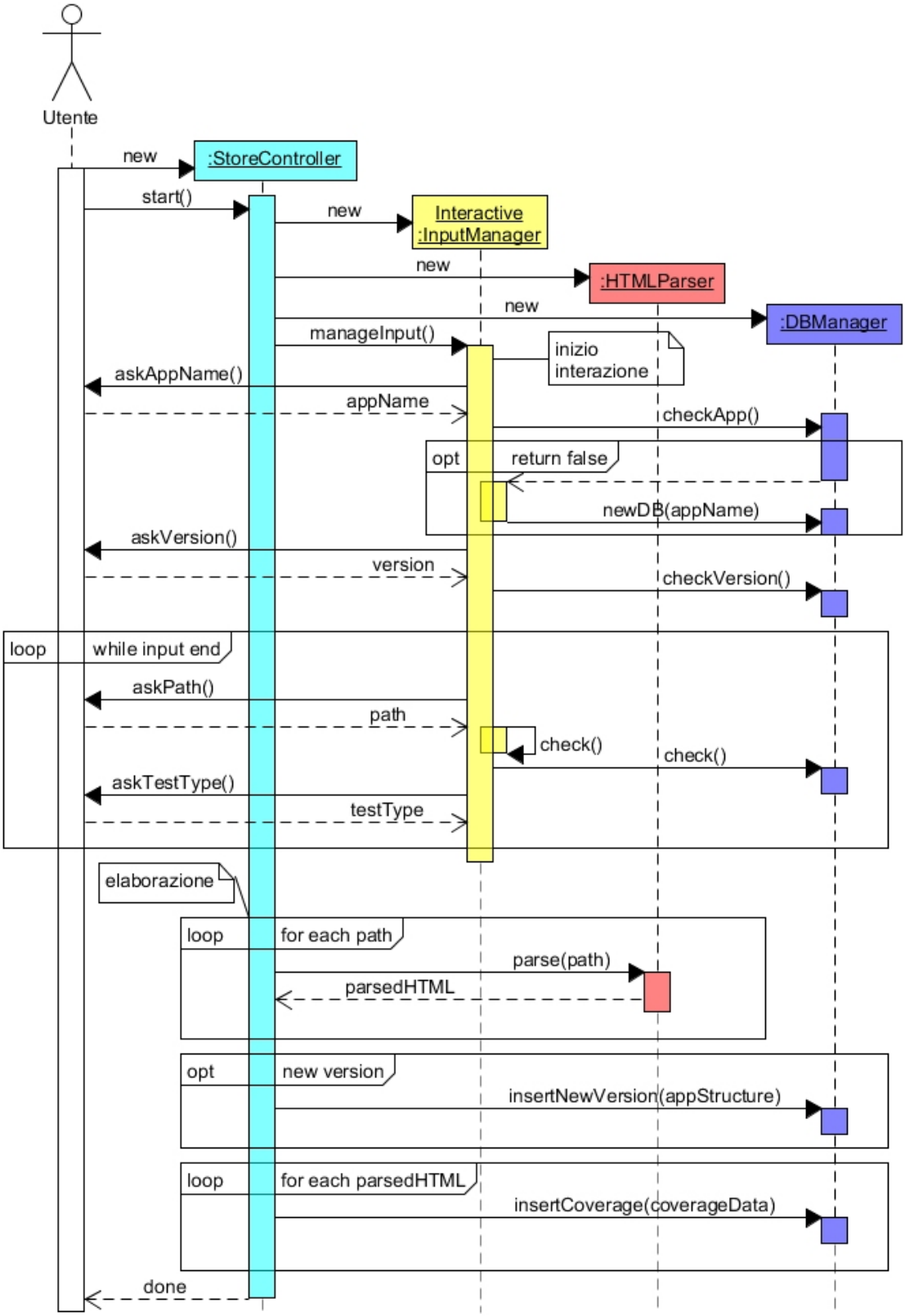


Figura 3.10 – Sequence diagram relativo al caricamento dei dati HTML in un DB

Dopo la richiesta del nome e della versione, l'InputManager gestisce l'inserimento dei path e dei tipi di test eseguendo vari tipi di controlli, fino a quando l'utente non decide di terminare l'inserimento degli input.

La fine dell'interazione coincide con l'inizio della fase di elaborazione in cui, con un oggetto di tipo HTMLParser, tramite parsing si ricavano i dati da ogni path HTML inserito in input nella fase precedente.

Una volta terminata la fase di analisi, il controller richiede al DbManager di inserire i dati della struttura dell'applicazione nel caso in cui la versione data in input dall'utente corrisponda ad una versione mai memorizzata nel database. Dopodiché, per ogni test di cui si sono ricavati i dati, il controller richiede al DbManager di inserirne i dati di coverage per la versione indicata dall'utente.

3.6.2 Correlazione dei report memorizzati. Caso d'uso.

Il caso d'uso mostrato in figura 3.11 si riferisce al requisito che il secondo sottosistema deve soddisfare.

Come si può notare le funzionalità di correlazione supportate sono 4: Unione, intersezione, Complemento e Differenza. Esse verranno descritte nel dettaglio più avanti, per ora ci interessa sapere che l'utente prima di avviare il sistema, sceglie quale delle correlazioni usare e, dal suo punto di vista, il modo di utilizzo del sistema è esattamente lo stesso per ogni funzione di correlazione eseguita. In tal senso il secondo sottosistema si può ritenere specializzato per offrire 4 sottofunzionalità dello stesso tipo all'utente

Parlando degli input, a completamento della descrizione che verrà proposta a breve, si specifica che, se viene indicato solo il nome del database, per ogni versione vengono effettuate le correlazioni tra tutti i test memorizzati (se ce ne sono almeno due per tale versione). Se invece si vuole indicare esattamente di quali test si vuole la correlazione, se

ne devono indicare ameno due. Il motivo è che non ha senso calcolare una correlazione di un test con se stesso, per cui il sistema controlla che ce ne siano almeno due da correlare.

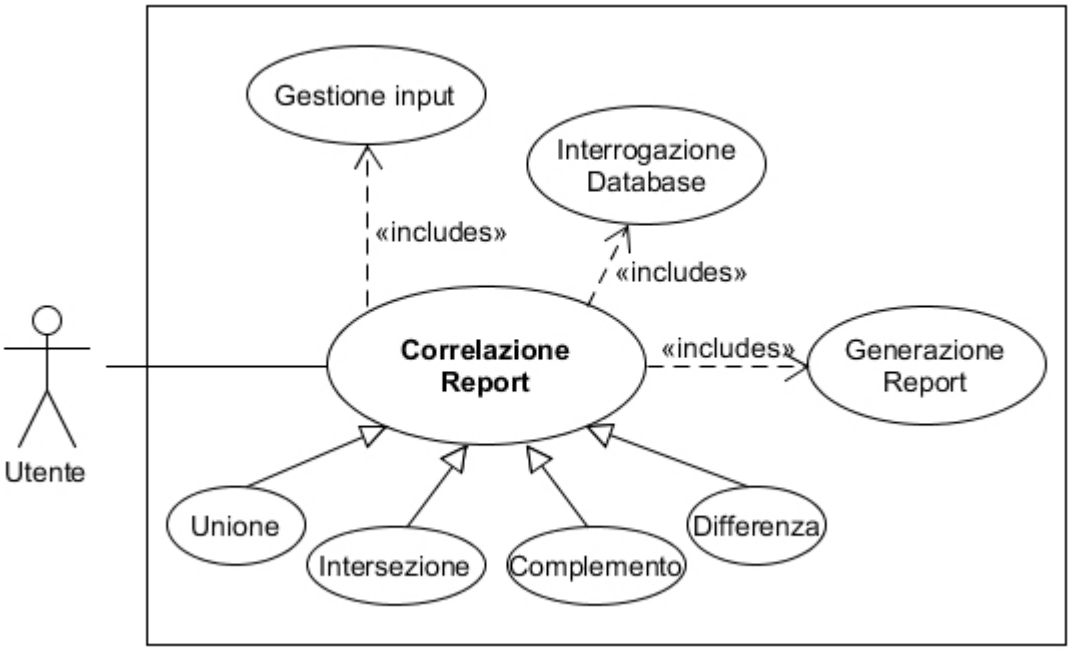


Figura 3.11 – Correlazione dei report. Use Case.

Di seguito viene fornita una descrizione schematica del caso d'uso mostrato in figura 3.11.

Attori coinvolti

Utente generico.

Descrizione generale della funzione.

Il sistema fornisce delle funzioni prestabilite di correlazione dei test memorizzati in un Database precedentemente creato con questo stesso tool. Per ogni funzione scelta, viene

generato un report testuale che descrive dettagliatamente i risultati ottenuti dalla correlazione realizzata.

Input

1. Nome del database. Obbligatorio
2. Versione dell'applicazione. Facoltativo
3. Lista di test. Obbligatoria se viene indicata la versione (in tal caso sono da indicarne almeno due), facoltativa altrimenti.

Pre-condizioni

Deve essere installato e avviato un server Sql sul pc dove viene eseguito il software.
Deve essere presente un Database sul server Sql creato precedentemente tramite l'utilizzo del primo sottosistema ed al suo interno devono essere presenti le informazioni di copertura di almeno due test appartenenti ad una stessa versione.

Descrizione del processo

Caso d'uso principale*:* **Correlazione Report**

Sequenza operazioni normale:

1. L'utente avvia il sistema attraverso una delle funzioni di correlazione supportate, indicando contestualmente gli input specifici richiesti per l'esecuzione.
2. **include** (Gestione input) – Il sistema acquisisce gli input, effettua vari controlli di validazione e li traduce in input utilizzabili per le query al database.

3. **include** (Interrogazione Database) – Il sistema costruisce ed effettua delle query che soddisfano la richiesta dell'utente in base agli input filtrati nella prima fase e alla modalità di esecuzione scelta.
4. **include** (Generazione Report) – Il sistema genera dei report i quali descrivono il risultato della correlazione voluta.

Caso d'uso incluso*:* **Gestione input**

Percorsi alternativi:

1.1 Se non esiste un Database col nome indicato in input oppure esiste ma non è stato creato con il primo sottosistema, il programma termina segnalando l'errore.

1.2 Se viene indicato solo il database ma nessuna delle versioni ha almeno due test memorizzati, il programma termina segnalando il problema.

1.3 Se la versione indicata non è memorizzata nel Database, il programma termina segnalando l'errore.

1.4 Se la versione indicata è memorizzata nel Database ma per essa non sono stati memorizzati almeno due report di test, il programma termina segnalando il problema.

1.5 Se i test indicati sono meno di due, il programma termina segnalando l'errore.

1.6 Per ogni test immesso in input, se esso non appartiene alla versione indicata viene scartato.

   a. Se dopo essere stati scartati, alla fine rimangono meno di due test, il programma termina segnalando l'errore

Post-condizioni

Viene generato un report testuale in formato *txt* per ogni correlazione effettuata il quale viene poi salvato in una specifica posizione del File System.

3.6.2.1 Correlazione dei report. Sequence Diagram.

In figura 3.12 è possibile vedere la sequenza temporale di esecuzione che descrive il caso d'uso di cui al paragrafo precedente. Il sequence diagram descrive uno scenario d'uso in cui l'utente richiede al sistema un report dell'*unione* tra i coverage dei test indicati in input e che sono stati precedentemente memorizzati nel database.

Analogamente al sequence diagram riguardante il primo sottosistema, il nome degli oggetti coinvolti nell'interazione rispecchia quello utilizzato per il diagramma in figura 3.8, riguardante la scomposizione modulare del secondo sottosistema, e anche qui si può vedere come la gestione del sottosistema sia affidata ad un oggetto di tipo controllore, identificato con ReportController, il quale si occupa di coordinare gli altri oggetti del sistema e gestire quindi l'intera fase di input-elaborazione-output.

Il controller crea un oggetto di tipo QueryInput e delega ad esso la gestione degli input. QueryInput dopo vari controlli nel database trasforma gli input utente in modo che gli stessi siano validi per le query effettuate nell'oggetto di tipo SetQuery.

Dopo questa prima fase, gli input trasformati vengono inviati dal controller ad un oggetto di tipo SetQuery opportunamente creato per la specifica correlazione che deve effettuare, in questo caso Union, insieme alla richiesta di correlazione dei test indicati input. Quindi Union esegue la query specifica che gli compete e rielabora i risultati restituiti dal database.

ReportController si occupa infine di inviare i risultati al ReportPrinter affinché siano stampati su un file di testo e salvati sul File System. Prima di terminare la sua esecuzione, il sistema informa l'utente sulla posizione nella quale è localizzato il report testuale.

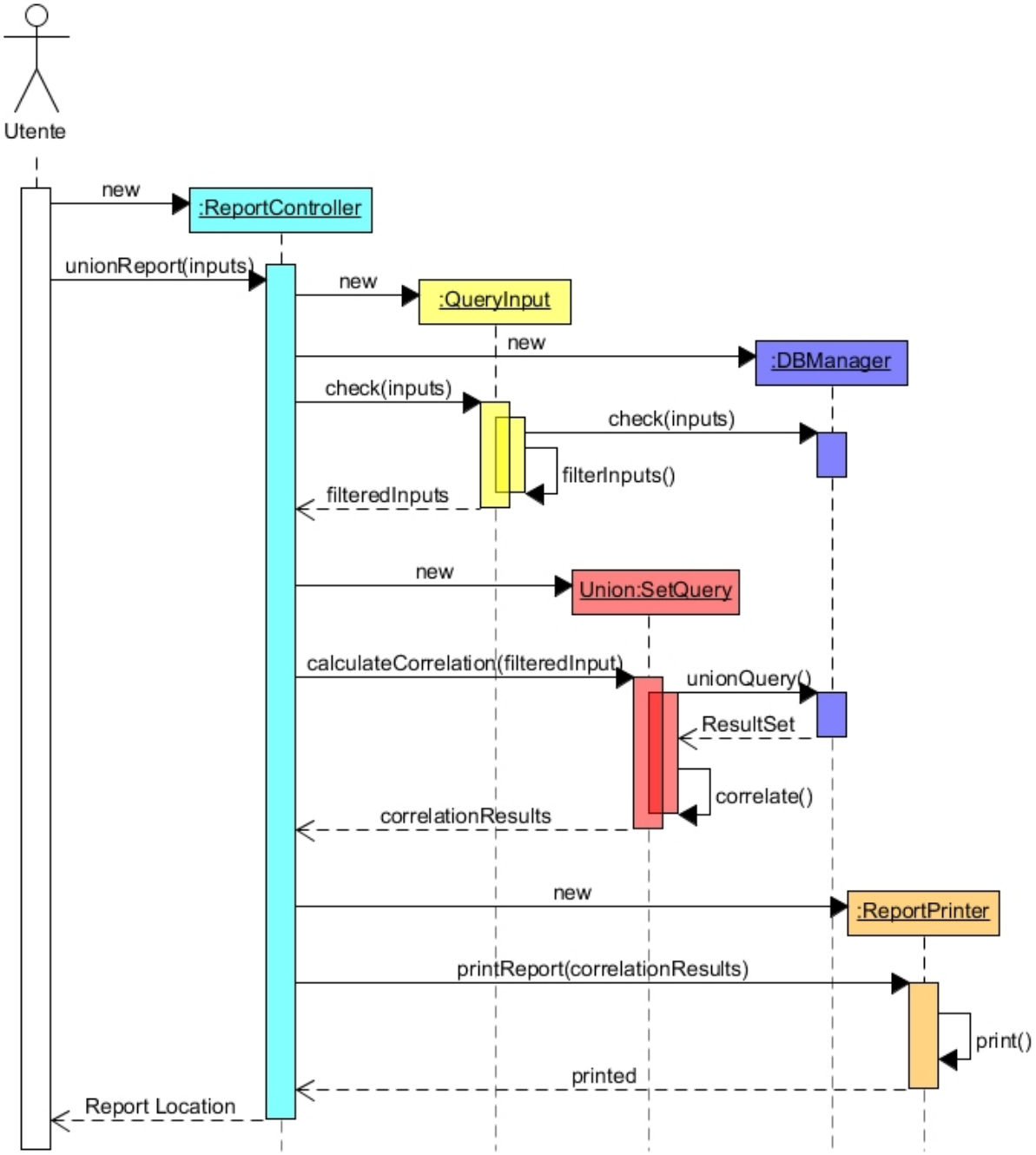


Figura 3.12 – Sequence diagram relativo alla funzione di unione tra due o più test

3.7 Implementazione del sistema

Il software realizzato per l'implementazione del sistema è stato scritto in JAVA, per cui la scomposizione modulare del sistema operata in fase di progetto si può tradurre facilmente in scomposizione in package.

Verrà dapprima presentato il package diagram relativo al sistema realizzato, poi due class diagram di medio dettaglio nel quale vengono mostrate le classi più importanti sviluppate per i due sottosistemi.

3.7.1 Package Diagram

In figura 3.13 viene mostrato una rappresentazione del software dal punto di vista dei package che lo compongono. Si può notare che essa rispecchia, eccetto che per la presenza di un package nuovo dedicato alla gestione del database, la scomposizione modulare discussa nel paragrafo 3.5.1, figure 3.7 e 3.8.

Dei due package maggiori, *store* rappresenta il primo sottosistema e *report* rappresenta il secondo sottosistema, entrambi dipendono dal package *database* per la realizzazione delle loro responsabilità principali.

Qui di seguito vengono descritti i package di cui il sistema è composto.

- *database*: fornisce le classi per il supporto alle funzionalità CRUD (create, read, update and delete) [12] sviluppate per la realizzazione degli specifici obiettivi del sistema.

Per il package *store* abbiamo:

- *controller*: fornisce le classi sviluppate per il controllo dell'esecuzione di una sessione del sottosistema.

- *input*: fornisce le classi sviluppate per la gestione degli input dati al sottosistema, anche attraverso una modalità interattiva oltre che automatica.
- *parser*: fornisce le classi sviluppate con lo scopo di effettuare il parsing dei rapporti di copertura del codice in formato HTML generati da EMMA.

Per il package *report*:

- *controller*: fornisce le classi per il controllo dell'esecuzione di una sessione del sottosistema.
- *query*: fornisce le classi che consentono l'esecuzione di specifiche query (verso un database creato con questo tool) al fine di realizzare le specifiche funzioni di correlazione (che si ricorda essere: unione, intersezione, complemento e differenza), le stesse classi elaborano ed incapsulano i risultati che dovranno poi essere stampati.
- *print*: fornisce le classi che consentono di stampare su file i risultati delle query effettuate.

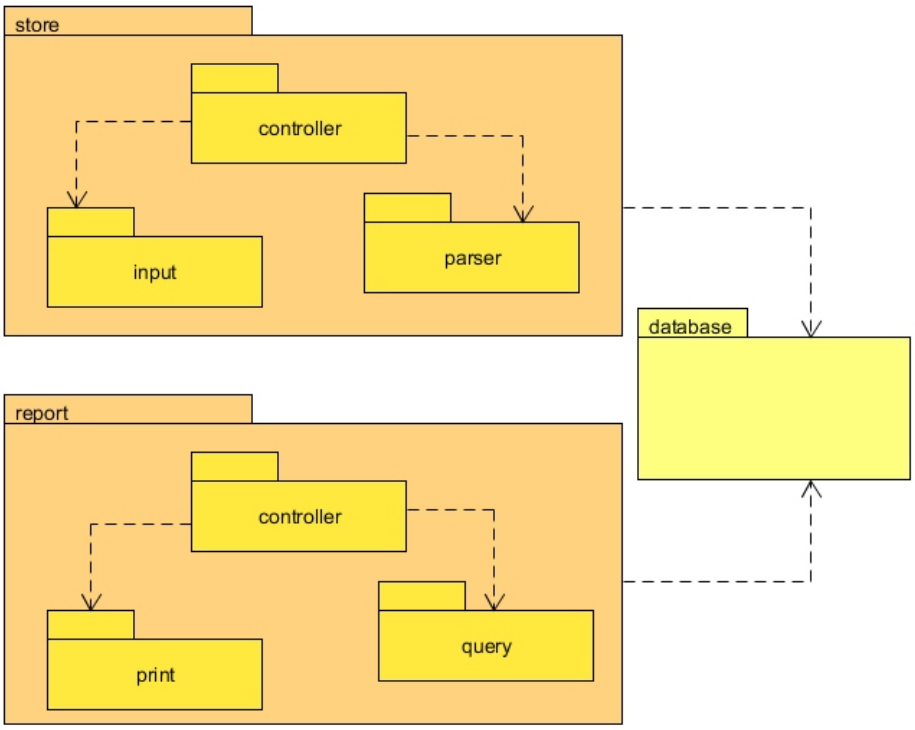


Figura 3.13 – Package Diagram del sistema

3.7.2 Class Diagram

Per ogni package visto nel paragrafo precedente, vengono presentate le classi fondamentali sviluppate attraverso un class diagram disegnato per ogni sottosistema.

Per ragioni di chiarezza e leggibilità si è evitato di scendere troppo in dettagli che potrebbero disorientare il lettore. Sono state omesse alcune classi e sono state scelte quelle che sono fondamentali per la realizzazione delle funzioni di business del sistema, evidenziandone le associazioni più importanti. Per lo stesso motivo anche alcuni attributi e metodi (presentati senza *signature*) stati omessi.

Per ogni sottoparagrafo verrà prima data una descrizione sintetica di ogni classe e poi una spiegazione di come esse collaborano fra di loro per realizzare insieme la funzione del sottosistema di cui fanno parte. L'andamento temporale di esecuzione segue quello descritto nei sequence diagram di figura 3.10 e 3.12, la differenza sta nel maggior livello di dettaglio che qui si adotta nella descrizione.

3.7.2.1 Class diagram del primo sottosistema. Package “store”.

Nella figura 3.14 viene presentato il class diagram semplificato del primo sottosistema. Nella parte bassa della figura si possono vedere le classi usate per la gestione del database (presenti nel package omonimo).

- **MyDbHelper**: rappresenta una connessione verso il database, incapsulata in un oggetto di tipo Connection. Offre metodi di utilità per l’avvio e la chiusura della connessione, l’esecuzione di query di tipo statement e la restituzione su richiesta di un oggetto di tipo Connection, Statement e PreparedStatement riguardanti la connessione incapsulata.
- **MyDbCreate**: è responsabile della creazione/eliminazione di database e creazione di tabelle, stored procedures e triggers.

  Questi ultimi sono usati per la gestione della cancellazione di un record da una tabella afferente la parte strutturale del database (cfr. par. 3.5.1) e la gestione degli indici auto_increment a supporto delle operazioni di inserimento dei dati nel database.
- **MyDbChecks**: offre metodi di utilità per effettuare vari tipi di verifiche sul database e i dati in esso memorizzati. Viene utilizzato in entrambi i sottosistemi per verificare l’esistenza di un database, una versione, un test e se questi ultimi appartengono alla versione indicata.
- **MyDbInsert**: responsabile dell'inserimento dei dati della struttura dell'app e di coverage, ricavati dai rapporti di copertura in Html, nel database. Delega tali compiti, rispettivamente, alle classi AppStructureInsert e CoverageInsert.
- **AppStructureInsert**: gestisce l'inserimento dei dati della struttura dell'app nel database. Mantiene in memoria le informazioni circa il numero di package, di classi, di metodi, di linee di codice totali e di linee di codice non eseguibili inserite ad ogni chiamata del metodo *insertNewVersion()*.

- **CoverageInsert**: Responsabile dell'inserimento dei dati di coverage nel database. Mantiene in memoria il numero di record totali e il numero di record di colore bianco (linee non eseguibili) inseriti nella tabella Coverages ad ogni chiamata del metodo *insertCoverage()*.

Per quanto le classi appartenenti al package *store.parser*, esse sono visibili nella parte centrale della figura.

- **AbstractEmmaParser**: E' una classe astratta che rappresenta un parser di un rapporto di copertura in HTML. Si serve delle classi DomGenerator ed XPathHelper per creare un albero DOM [13] di file HTML e le espressioni XPath [14] per ricavarne le informazioni.
- **DomGenerator**: offre un metodo per restituire un oggetto Document rappresentante un albero DOM.
- **XPathHelper**: offre un metodo per l'esecuzione di query XPath su un documento DOM e la restituzione di una NodeList [15] del risultato.
- **HtmlReportParser**: E' una classe derivata di AbstractEmmaParser. E' quindi un parser con la responsabilità di ricavare i dati di interesse da report di test HTML generati con EMMA tramite l'algoritmo di parsing implementato nel metodo *parse()* ereditato dalla classe padre.

  I Dati ricavati sono ordinati lessicograficamente in ordine ascendente negli array che rappresentano i packages, le classi (interne ed esterne), i metodi, il testo, il colore ed il peso di ogni linea di codice. Vengono in oltre memorizzate anche altre informazioni riguardo il path, la data e il tipo di test del report analizzati.

Le classi utilizzate per la gestione degli input sono visibili in alto a destra nella figura, esse appartengono al package *store.input*.

- **AbstractInputManager**: responsabile del controllo degli input (effettuati anche interrogando il server MySql) e della eventuale creazione di un database (delega a MyDbCreate).

  Memorizza le informazioni ricavate nella gestione degli input realizzata nelle classi derivate tra cui: il nome dell'applicazione, la versione, i path, il tipo di test per ogni path, la data dell'esperimento presente nei path, oltre ad informazioni circa l'esistenza o meno di un database con lo stesso nome dell'applicazione indicato in input e, in tal caso, se esso è popolato con tabelle proprie del modello di cui al paragrafo 3.5.1 (indicatore del fatto che il database sia stato creato o meno con questo tool).

  Il metodo *manageInput()* sarà implementato dalle sottoclassi per gestire l'input utente, esso assume un diverso significato a seconda della classe che lo implementa: per AutomaticInput significa che gli input dati devono solo essere analizzati così come sono, per InteractiveInput significa che gli input devono essere acquisiti e controllati di volta in volta tramite interazione con l'utente. Gli altri metodi vengono utilizzati per controllare e filtrare gli input
- **AutomaticInput**: E' una classe derivata da AbstractInputManager dalla quale eredita tutte le variabili di istanza che saranno valorizzate al termine del controllo e della gestione dell'input utente. I metodi implementati servono per filtri e controlli sugli input.
- **InteractiveInput**: E' una classe derivata da AbstractInputManager e valorizza tutte le variabili di istanza ereditate attraverso la gestione interattiva dell'inserimento degli input da parte degli utenti.

Le classi appartenenti al package *store.controller* sono visibili in alto a sinistra della figura.

- **StoreController**: ha la responsabilità della realizzazione della funzione del sottosistema cui appartiene e quindi delle fasi di acquisizione degli input, elaborazione dei dati e produzione dell'output. Si serve della classe ControllerHelper per assolvere alle ultime due fasi.
- **ControllerHelper**: ha la responsabilità del controllo delle operazioni di parsing dei file html ricavati dagli input e di inserimento dei dati risultanti dal parsing nel database gestire del parsing, come viene evidenziato dalle associazioni con le classi HtmlReportParser e MyDbInsert.

Infine notiamo la presenza della classe **LoadCoverages**, essa è la classe entry-point del sottosistema, nella quale è presente il metodo *main()* da dove il programma inizia la sua esecuzione. Seguiamo quindi una generica esecuzione partendo proprio da quest'ultima.

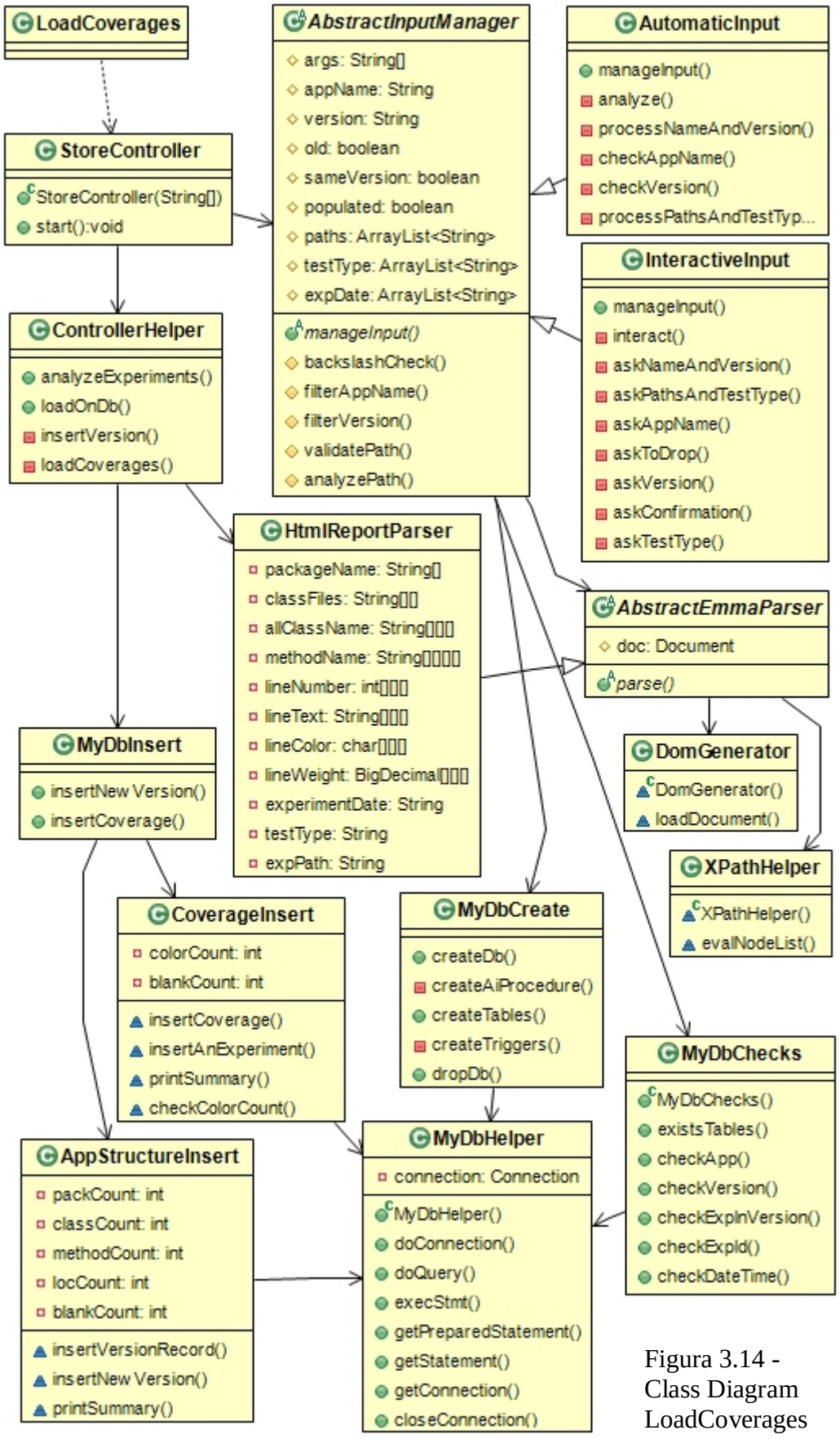


Figura 3.14 - Class Diagram LoadCoverages

3.7.2.2 Collaborazione delle classi del primo sottosistema. Package "store".

Inizializzazione

All'atto dell'avvio del sistema, l'utente specifica i parametri di input che verranno passati al metodo main(). LoadCoverages si occupa di creare un oggetto di tipo StoreController, inviandogli gli argomenti inseriti in input dall'utente come parametri di avvio e richiamare il suo metodo start(). Se non vengono passati parametri di input il sistema avvia la versione interattiva e l'utente deve specificare gli input di volta in volta, altrimenti l'utente inserisce tutti quelli necessari a produrre l'output e il sistema avvia la versione automatica. L'oggetto di tipo StoreController si occupa di creare a sua volta una connessione verso il server MySql tramite un oggetto di tipo MyDbHelper e, a tutti gli altri oggetti per la gestione del database, viene "iniettata" la stessa connessione. A seconda della modalità scelta dall'utente viene creato il giusto oggetto di tipo AbstractInputManager, tra AutomaticInput ed InteractiveInput servendosi del design pattern *Factory method* [16] (le cui classi non sono visibili nella figura) ed un oggetto di tipo ControllerHelper.

Input

L'invocazione del metodo *start*() innesca la fase di gestione degli input nel quale il controllo viene passato ad uno dei due oggetti di tipo AbstractInputManager. Entrambi usano i metodi comuni della loro classe base per il controllo della correttezza degli input.

- Col metodo *filterAppName()* si controlla che il nome dell'applicazione non sia di lunghezza superiore a 64 caratteri come richiesto nelle specifiche MySql per la lunghezza dei nomi di un database [17]; se la lunghezza supera i 64 caratteri viene restituita una sottostringa troncata a 64 caratteri.
- Tramite il metodo *filterVersion()* si filtra la stringa inserita affinché non sia di lunghezza superiore a 30 caratteri, poiché tale è la lunghezza massima scelta per l'attributo "app_version" della tabella "versions".
- Con i metodi *analyzePaths()* e *validatePaths()* si controlla un path inserito

dall'utente sotto forma di stringa, eventuali caratteri di backslash ("\") vengono convertiti in slash ("/"), viene controllato se la stringa corrisponde ad un path realmente esistente sul File System e se contiene un file index.html. Se i controlli precedenti hanno esito positivo, viene controllato se il path è stato già inserito in questa sessione o in un'altra precedente sessione (se il database esiste).

I metodi precedentemente elencati sono usati da entrambe le classi derivate.

Nel caso si scelga la versione automatica, un oggetto di tipo AutomaticInput filtra il nome dell'applicazione e la versione, quindi crea un database di nome "*appname*" se un'applicazione con tale nome non è mai stata analizzata prima. Infine ricava i path ed i tipi di test per ogni path dagli input inviati dopo averli analizzati e controllati.

Nel caso si scelga la versione interattiva, al pari di AutomaticInput, vengono fatti tutti i controlli del caso sia sul nome dell'applicazione, che della versione, che dei path e del tipo di test per ogni path inseriti.

A differenza della versione automatica, i controlli vengono fatti in tempo reale e, se qualcosa va male, l'utente viene informato e gli si da la possibilità di correggere l'input. Il nome dei metodi implementati tradiscono questo comportamento: infatti per ogni input necessario, viene richiesto l'intervento dell'utente per proseguire o anche solo per terminare l'esecuzione. La fine dell'inserimento dei dati in input oppure la terminazione anticipata del programma vengono gestiti a seguito dell'inserimento da parte dell'utente della parola "fine" (case insensitive), in qualsiasi momento dell'esecuzione.

Nella fase di gestione degli input vengono usati gli oggetti di tipo MyDbCheck per eseguire vari controlli sul server MySql o su un database e MyDbCreate per creare un database nel caso ce ne fosse bisogno.

Elaborazione

Dopo questa fase iniziale di gestione degli input, il controllo passa di nuovo al controller il quale invoca *analyzeExperiments*() di ControllerHelper per gestire il parsing degli esperimenti, passandogli l'oggetto di tipo AbstractInputManager che, a questo punto, contiene le informazioni anche sul dove sono localizzati gli esperimenti da analizzare sul File System.

ControllerHelper quindi, non fa nient'altro che creare un nuovo oggetto di tipo HtmlReportParser per ogni path ed invocare su di esso il suo metodo *parse()* al quale passa un path tra quelli in lista.

Dopo tale invocazione, tutti i dati ricavati dal parsing sono ordinati e memorizzati nelle variabili di istanza dell'oggetto.

Output

A questo punto per ogni path c'è un oggetto di tipo HtmlReportParser che ne conserva tutte le informazioni da stampare.

StoreController invoca il metodo *loadOnDb()* di ControllerHelper, il quale a sua volta invoca i metodi *insertVersion( )* e *loadCoverages()* per inserire i dati ricavati dal parsing nel database (se un database dedicato all'applicazione già esisteva prima della corrente sessione e non si deve memorizzare una nuova versione, allora ControllerHelper non invoca il primo metodo*).*

L'inserimento dei dati viene delegato ad un oggetto di tipo MyDbInsert, il quale a sua volta delega l'inserimento, di una nuova versione, ad un oggetto di tipo AppStructureInsert e, del coverage di ogni test, ad un oggetto di tipo CoverageInsert. Al termine dell'inserimento di un coverage tramite l'invocazione del metodo *checkColorCount()* di CoverageInsert viene verificato che il numero totale di record inseriti nella tabella Coverage sia pari al numero totale di record presenti nella tabella loc. In caso contrario viene impedito che i dati di coverage per quel path vengano inseriti

perché certamente essi non sono completi o corretti (possono appartenere ad un'altra versione o ad un'altra applicazione).

3.7.2.3 Class diagram del secondo sottosistema. Package "report".

Nella figura 3.15 viene presentato il class diagram semplificato del secondo sottosistema.
Le classi presenti in alto a sinistra della figura, cioè *Differences, Intersection, Complement, Union,* sono delle main-class tramite le quali un utente richiede la specifica correlazione che esse rappresentano. La classe *Total* consente all'utente di richiedere i report di tutti i tipi di correlazione supportate.

Le classi adibite al controllo del sottosistema sono ReportController e EmmaReportController, rispettivamente un'interfaccia e la sua classe derivata:

- **ReportController**: è la classe con cui si interfacciano le altre main-class le quali utilizzano il metodo *report()* per avviare il sottosistema.
- **EmmaReportController**: Responsabile del controllo di correttezza degli input immessi, dell'esecuzione di una query, a seconda della correlazione decisa dal client, e della stampa risultati.
  Implementa il metodo *report()* che sostanzialmente delega l'esecuzione della query e la stampa dei risultati ad oggetti di tipo EmmaQuery e ReportPrinter tramite i metodi *executeQuery()* e *printReport()*.

Le classi adibite al controllo degli input ed all'esecuzione delle query insiemistiche sono contenute tutte nel package *report.query*.
Quelle utilizzate per l'input sono:

- **QueryInputs**: La sua responsabilità è quella di controllare, filtrare e tradurre gli input utente in input che saranno utilizzati dalle classi di tipo EmmaQuery dello stesso package. Come si può intuire osservando le sue variabili d'istanza

private, gli input vengono organizzati secondo un criterio gerarchico: il nome del database in cima alla gerarchia, poi, per ogni database ci può essere una lista di versioni e per ogni versione una lista di test (expId, expDate, expType). Il numero di questi elementi dipende dagli input utente.

- **QueryInputsUtils**: come suggerisce il nome, è una classe di metodi di utilità pensati per essere utilizzati da QueryInputs. Vengono offerti metodi per controlli generici e per controlli sul database.

Le classi adibite all'esecuzione delle query sono visibili in basso a sinistra della figura. Esse formano una gerarchia in cui *EmmaQuery* è un'interfaccia ed è classe padre di *AbstractEmmaQuery*, da cui derivano le restanti classi descritte qui di seguito.

- **EmmaQuery**: rappresenta un oggetto in grado di effettuare una query specifica verso un database creato con il primo sottosistema e mantenerne in memoria i risultati.

  Il metodo *detailedQuery()* è un metodo che, implementato dalle sue sottoclassi, calcola la lista di linee di codice coinvolte nella query che realizza la correlazione e ne consente di memorizzare i risultati.

  Il metodo *coverageSummary()* serve per il calcolo di informazioni generali riguardo le query effettuate, come il numero di linee di codice coinvolte nella query, il numero di linee eseguibili della versione dell'app memorizzata nel database, una percentuale di copertura rispetto al numero di linee eseguibili e una media di copertura delle linee di codice con relativa incertezza di calcolo. La percentuale, la media e l'errore sono messi insieme in una stringa formattata e memorizzati nella variabile di istanza *totalCoverage* (della classe figlia).
- **AbstractEmmaQuery**: è una classe astratta che rappresenta una o più query verso un Database relativo alle coperture di un Applicazione. E' progettata per contenere le informazioni relative ai risultati di tali query, al nome del database

coinvolto, alla/e versione/i ed agli esperimenti, cioè tutte le informazioni utili per stampare un report.

- **UnionQuery**, **IntersectionQuery** e **ComplementQuery** Implementano i metodi dell'interfaccia per realizzare, rispettivamente, una correlazione di unione, intersezione e complemento tra due o più esperimenti di una o più versioni di un Applicazione memorizzata nel database.
- **DifferencesQuery**: Implementa i metodi dell'interfaccia per realizzare la differenza tra soli due esperimenti alla volta di una o più versioni di un Applicazione memorizzata nel database.

Le classi adibite alla produzione dei report dei risultati delle query sono visibili in basso a destra della figura e formano una gerarchia di generalizzazione/specializzazione: *ReportPrinter* è un'interfaccia, *AbstractEmmaReportPrinter* è la classe astratta da essa derivata ed *EmmaReportPrinter* deriva da quest'ultima. Esse appartengono al package *report.print.*

- **ReportPrinter**: rappresenta un printer di report generico.
- **AbstractEmmaReportPrinter**: rappresenta un printer su file txt di report riguardanti correlazioni tra test HTML generati con il tool EMMA.

  Ha la responsabilità di generare un path di base per tutti i report testuali prodotti, generare un nome di file per ogni report da stampare, generare un file sul File System per ogni report e di offrire alle sue sottoclassi metodi per il print di informazioni specifiche sui file di report, suddivise in riquadri.
- **EmmaReportPrinter**: Printer di query di tipo EmmaQuery che implementa il metodo *printReport()* dell'interfaccia ReportPrinter.

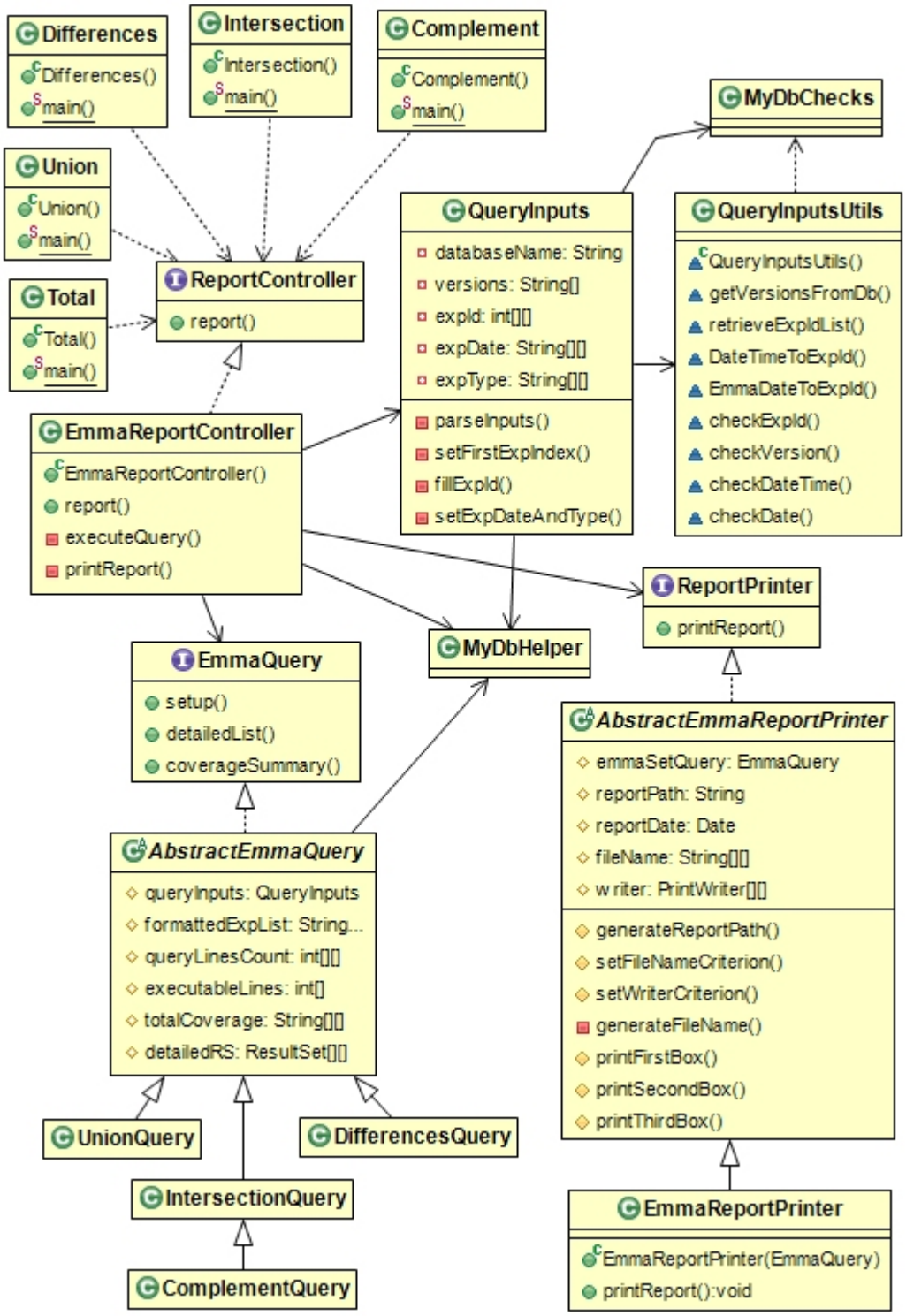


Figura 3.15 – Classi diagram del secondo sottosistema

3.7.2.4 Collaborazione tra le classi del secondo sottosistema. Package “report”.

Inizializzazione

L’utente specifica i parametri di input passati al metodo main() di uno degli entry-point del sottosistema scelto. La classe main crea un oggetto di tipo EmmaReport passandogli i parametri di input specificati dall’utente, l’oggetto concreto istanziato è di tipo EmmaReportController. Viene stabilita in questa classe la connessione (con un oggetto di tipo MyDbHelper) al database, essa viene poi “iniettata” nelle classi che ne hanno bisogno comprese le classi del package *report.query*.

Input

Contestualmente alla sua creazione, un oggetto di tipo EmmaReportController crea un oggetto di tipo QueryInput ai quali affida gli input utente che devono essere controllati. QueryInput esegue una procedura automatica per il controllo dei dati servendosi anche dei metodi contenuti in QueryInputsUtils. Le informazioni utili ad effettuare le query vengono ricavate, organizzate e correlate tra loro automaticamente e variano a seconda di quali input utente vengono inseriti come parametri di input:

1. Se l'utente inserisce solo il nome del Database, da tale db vengono ricavate e memorizzate tutte le versioni e, per ogni versione, tutti gli esperimenti.
2. Se l'utente inserisce il nome del Database e la versione, vengono memorizzati tutti gli esperimenti di tale versione ricavati dal database.
3. Se l'utente inserisce il nome del Database, la versione e almeno due esperimenti allora vengono memorizzate esattamente tali informazioni.

Le variabili d'istanza databaseName, versions, expId, expDate, expType vengono valorizzate tramite risultati di query specifiche per il reperimento di tali informazioni dal database e dopo una verifica che gli input utente abbiano corrispondenza nel database che si vuole interrogare.

Elaborazione

Dopo questa fase iniziale la Classe *main* richiama il metodo *report*() del *controller* ed è da questo punto in poi che inizia la fase di elaborazione.

L'oggetto di tipo EmmaReportController crea un oggetto di tipo EmmaQuery specifico per la correlazione richiesta, inviandogli un oggetto di tipo QueryInput, poi ne invoca i metodi *detailedList()* e *coverageSummary()* tramite i quali vengono ricavati la lista di linee di codice coinvolte nella query specifica per il calcolo della correlazione e un sommario di copertura totale risultante dalla somma dei pesi della lista di linee di codice risultante dalla query. Prima di effettuare le query, viene controllato che i test da correlare siano almeno due.

Output

Una volta che il controller riprende il controllo dell'esecuzione al termine delle operazioni di query di correlazione, crea un oggetto concreto di tipo ReportPrinter inviandogli un oggetto di tipo EmmaQuery che a questo punto dell'esecuzione contiene tutte le informazioni da stampare. Successivamente ne invoca il metodo *printReport()*

L'oggetto concreto di tipo EmmaReportPrinter stampa i report in base all'oggetto di tipo EmmaSetQuery ricevuto dal costruttore.

I report generati sono tanti quante sono le correlazioni effettuate: per le query commutative come l'intersezione, l'unione ed il complemento, viene generato un report per ogni versione. Per le query di tipo differenza vengono generati tanti report quante sono le differenze semplici effettuate. Ad esempio, se n è il numero di test di cui si deve calcolare la correlazione, vengono effettuate tante differenze semplici quante sono le disposizioni di due elementi su n posti (ovvero: D(n,k) = $\frac{n!}{(n-2)!} = n * (n-1)$).

3.8 Dettagli realizzativi

Il nome del software realizzato per l'implementazione del sistema è StoreAndReport.

Il software è stato scritto in JAVA usando il JDK 7 [18], funziona su un singolo pc ed è stato progettato per il sistema operativo Windows, anche se, data la portabilità del linguaggio di programmazione utilizzato, può essere portato su altri sistemi operativi con pochissimi sforzi.

Sia lo stile di scrittura del codice che la scelta dei nomi degli elementi componenti il programma sono stati scelti per prediligere il più possibile la chiarezza sacrificando talvolta la compattezza.

Il codice è stato documentato sia usando il tool JavaDoc [19] per i metodi, le classi ed i package che compongono il programma, sia usando commenti semplici all'interno dei metodi per cercare di spiegare meglio il significato degli algoritmi e delle scelte operate nel codice.

In figura 3.16 viene mostrato l'indice della documentazione generata tramite JavaDoc.

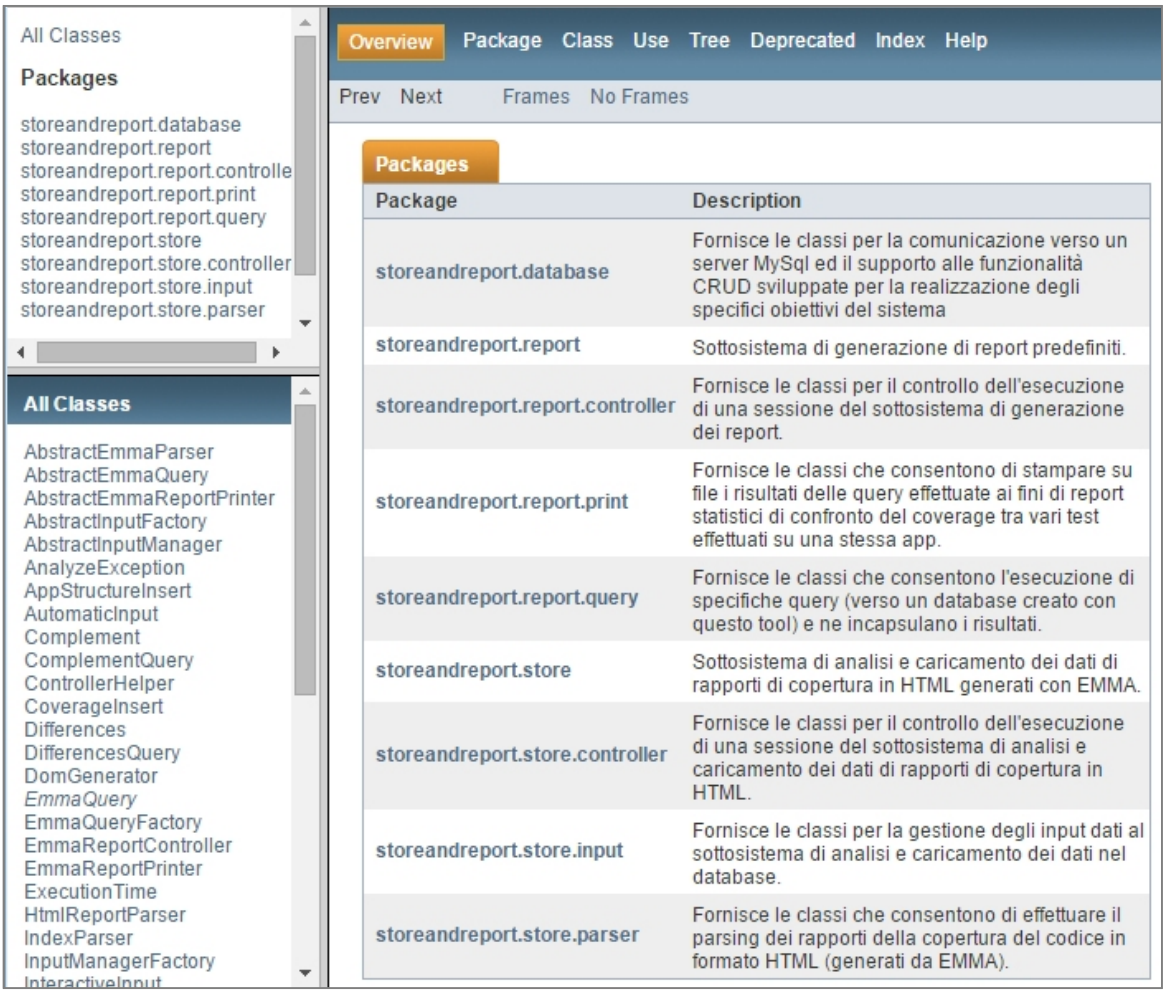


Figura 3.16 – Indice della documentazione HTML del programma

3.8.1 Descrizione dell'algoritmo di parsing HTML

Come già accennato nella descrizione del class diagram di figura 3.14, il parsing dei file HTML viene fatto nel metodo *parse()* ereditato dalla classe *AbstractEmmaParser*. Ad esso viene passato come parametro un path dove sono contenuti tutti i file HTML relativi ad un test.

Nell'algoritmo usato per il parsing, ogni file HTML viene trasformato in un documento DOM il quale successivamente viene esplorato servendosi della sintassi XPath, al fine di

ricavare dai nodi del documento le specifiche informazioni da memorizzare nelle variabili di istanza della classe.

Dal punto di vista logico ogni report viene visto come un albero al cui vertice, cioè al livello 0, c'è il file *index.html.* Al livello 1 ci sono i packages, al livello 2 ci sono le classi principali, al livello 3 ci sono le classi interne e le linee di codice, al livello 4 ci sono i metodi.

Le linee di codice in teoria dovrebbero essere al livello 5, per riferirle ai metodi, purtroppo nei file di report non si è trovato un modo efficace per legare le linee di codice né ai metodi né alle classi interne ma solo alle classi principali. Questo non inficia assolutamente sulla funzionalità che il sistema offre. I metodi sono al livello 4 poiché si è riusciti ad associarli alle classi interne. In realtà i metodi e le classi interne sono memorizzati nel database solo per completezza di informazioni e per supportare eventuali sviluppi futuri, essi non sono mai usati nel calcolo delle correlazioni.

Dal punto di vista "fisico", dal File System non è ricavabile una vera e propria gerarchia come quella descritta poc'anzi, per cui l'algoritmo di parsing in un certo senso ricostruisce l'albero logico per poi ricavarne i dati.

In figura 3.17 è possibile vedere la sintassi utilizzata per le espressioni XPath utili a ricavare i nodi fondamentali che consentono di ricavare le informazioni da memorizzare sia i nodi che consentono di proseguire nella navigazione dei file HTML.

```
//trova i packages nell'index.html
String xpathPackage     =
      "/HTML/BODY/TABLE/TR/TD/A[contains(@HREF, '_files')]";

//Trova le classi nei files html dei packages
String xpathHtmlClass   =
      "/HTML/BODY/TABLE[3]/TR/TD/A[@HREF]";

//trova tutti gli elementi contenenti il nome dei metodi
//(tag <a> con HREF di tipo ancora )
String xpathMethod   =
      "/HTML/BODY/TABLE[3]/TR[not(contains(@CLASS,'cis'))]"
      + "/TD[@CLASS='f']/A[starts-with(@HREF,'#')]";

//LOC (utili per numero di linea, codice e colore)
String xpathLOC     =
      "/HTML/BODY/TABLE[4]/TR";

//nodi delle sole classi all'interno di un html
String xpathClassInHtml =
      "/HTML/BODY/TABLE[3]/TR[contains(@CLASS,'cis')]"
      + "/TD[@CLASS='f']/A[starts-with(@HREF,'#')]";

//nodi delle classi e dei metodi (all'interno di un file html di una classe)
String xpathClassAndMethod =
      "/HTML/BODY/TABLE[3]/TR/TD[@CLASS='f']";
```

Figura 3.17 – Espressioni XPath per estrarre i nodi fondamentali

L'esplorazione dell'albero logico avviene in modalità DFS [20]. Sostanzialmente viene prima ricavata la lista dei package presenti nell'*index.html*, per ogni package viene ricavata la lista di classi principali e per ogni classe principale viene ricavata la lista di classi interne e linee di codice (numero di linea di codice, testo della linea di codice, colore), infine viene ricavata la lista di metodi per ogni classe.

Alla fine del parsing, dopo aver ricavato tutti i dati, questi vengono ordinati lessicograficamente in modo ascendente.

3.8.2 Descrizione delle query e del calcolo del coverage

Come si era anticipato nel paragrafo 3.4, la tabella *coverages* è quella cruciale ai fini della realizzazione del sistema, essa è il punto in cui convergono i dati di copertura dei diversi

test effettuati per una particolare versione, informazioni chiave per la realizzazione delle query insiemistiche. Le altre tabelle del database sono utilizzate in JOIN con la tabella *coverages* sostanzialmente per dare un senso comprensibile alle righe risultanti dalle query.

Le informazioni (attributi) contenute nella tabella *coverages* sono:

- *loc_id*: chiave primaria della tabella *loc* nella quale sono memorizzate le linee di codice dell'applicazione memorizzata nel database.
- *exp_id*: chiave primaria della tabella *experiments* dove sono memorizzati gli esperimenti per ogni versione dell'applicazione.
- *color:* colore di ogni linea di codice per ogni test memorizzato.
    - 'c' = linea coperta (*covered*)
    - 'p' = linea parzialmente coperta (*partial*)
    - 'z' = linea non coperta (*zero*)
    - 'b' = linea non eseguibile (*blank*)
- *weight*: per ogni esperimento, percentuale di copertura di una singola linea di codice (in termini di blocchi di codice) che vale:
    - Uno per le linee coperte.
    - Un valore compreso tra zero e uno per le linee coperte parzialmente.
    - Zero per le linee non coperte.
    - Zero per le linee non eseguibili.

Le query sono state sviluppate per assolvere a due obiettivi: calcolare dettagliatamente quali linee di codice sono il risultato della correlazione (query per la lista dettagliata) e calcolare il coverage totale che si ottiene sommando i pesi di delle linee di codice risultanti dalla correlazione (query per il coverage totale della correlazione)

.

3.8.2.1 Incertezza sul calcolo del coverage

Il calcolo del coverage totale contiene una certa quantità di incertezza per via della presenza delle linee di codice parziali: esse hanno un peso che varia tra zero e uno ma non ci sono informazioni su quali blocchi di codice (che le compongono) siano state realmente coperti.

Per capire meglio facciamo un esempio: supponiamo che una linea di codice sia stata coperta parzialmente da due test, A e B, e che il valore di copertura sia di 0,5 per entrambe. In tal caso non possiamo sapere se la loro unione porti ad un valore di copertura di 1 oppure di 0,7 o 0,5, come non possiamo sapere se la loro intersezione o la loro differenza sia vuota oppure porti ad un valore di peso di 0.3 o 0.4 o altro (sempre minore o uguale a 0.5 in questo caso).

Per tenere conto di questo fenomeno, per ogni query si calcola un valore pessimistico e uno ottimistico di copertura totale, relativi rispettivamente al caso in cui il peso risultante sia il minimo possibile e quello in cui sia il massimo. Dopodiché se ne fa la media ed infine, nel risultato, si riporta la media e l'incertezza pari rispettivamente a:

$$Media = \frac{\text{ottimistico} + \text{pessimistico}}{2}$$

$$Incertezza = \left|\frac{\text{ottimistico} - \text{pessimistico}}{2}\right|$$

Il coverage totale viene riportato nel report txt nella forma:

$$percentuale\ (\ mediaCov/totLOCEseguibili\ )\ [\pm incertezza]$$

Un esempio di come viene riportato il coverage totale nel report testuale prodotto dal tool realizzato è riportato in figura 3.16, sotto la voce *CovLoc* (preso da un file di report reale, che descrive i risultati dell'unione di due esperimenti su un'app Android chiamata

Omnidroid). I primi due esprimono lo stesso valore di coverage totale, uno in forma percentuale e l'altro in forma frazionaria; tra parentesi quadre è riportato l'errore massimo che si è potuto commettere nel calcolo.

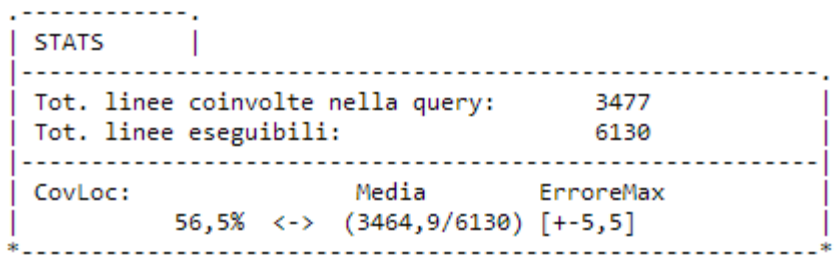


Figura 3.18 – Coverage totale di correlazione

3.8.2.2 Unione

La query di unione effettuata per la lista dettagliata, riporta tutte le linee di codice che hanno l'attributo *color* pari a 'c' oppure 'p' e che appartengono ad uno degli esperimenti dei quali si vuole la correlazione. Se per una linea di codice c'è qualche esperimento che l'ha coperta totalmente ('c') e altri che l'hanno coperta parzialmente ('p'), la query riporta che per quella linea la copertura è completa.

Nel caso ottimistico per ogni linea di codice si calcola il valore minimo tra la sommatoria dei pesi dei vari esperimenti e 1.

Nel caso pessimistico si calcola il valore massimo tra i pesi di copertura per ogni linea di codice.

Nell'esporre la lista delle linee di codice che sono correlate secondo l'unione, per le linee di codice che sono 'p' in almeno un test ma non sono mai 'c' in nessun altro test, viene mostrato nel report txt il colore 'p' ed il valore del peso massimo tra quelli ottenuti nei vari test.

3.8.2.3 Intersezione

La query riporta, per le linee dettagliate, tutte linee di codice che hanno l'attributo color pari a 'c' oppure 'p' e che appartengono a tutti e solo gli esperimenti per i quali si vuole la correlazione. Se per una linea di codice c'è qualche esperimento che l'ha coperta totalmente ('c') e altri che l'hanno coperta parzialmente ('p') la query riporta che per quella linea la copertura è parziale.

Nel caso ottimistico si prende il minimo valore tra i pesi per ogni linea di codice. Nel caso in cui per una linea di codice tutti i test l'hanno coperta solo parzialmente si considera che i blocchi coperti siano in comune cioè che si intersechino sempre.

Nel caso pessimistico, per le linee coperte solo parzialmente si calcola il minimo valore possibile dell'intersezione (che può essere anche nullo nel caso in cui le porzioni di linea di codice coperte di almeno un test non sono in comune con gli altri).

3.8.2.4 Differenza semplice (A-B)

La query riporta, per la lista di linee dettagliate, tutte le linee di codice che hanno attributo color pari a 'c' o 'p' in un test e 'p' o 'z' nell'altro.

Come è possibile notare in figura 3.19, riferendoci al report testuale generato dal sistema, si è scelto di riportare esattamente la differenza dei pesi tra i due test nella colonna *peso*, di conseguenza ci sono due casi particolari che devono essere discussi.

1. Per alcune linee di codice potrebbe accadere che nella colonna *colore* il valore sia 'c' ma il valore sotto la colonna *peso* sia minore di 1. Questo accade quando, per la linea di codice in questione, il test A la copre totalmente ('c') ed il test B parzialmente ('p').
2. Potrebbe accadere che nella colonna *colore* il valore sia 'p' ma il valore di weight sia minore o uguale a zero. Il secondo caso ricade nell'eventualità in cui entrambi i test coprano parzialmente una linea di codice ma il test 2 ne copre una porzione in più (maggiore o uguale) in termini di blocchi.

Report dettagliato versione 1
Differenza(ExpID): 2-1.

Package	Classe	#Linea	Colore	Peso	Source Code
org.tomdroid	Note	65	p	0.00	pri
org.tomdroid	Note	71	p	0.00	pub
org.tomdroid	Note	72	p	0.00	pub
org.tomdroid	Note	73	p	0.00	pub
org.tomdroid	Note	74	p	0.00	pub
org.tomdroid	Note	75	p	0.00	pub
org.tomdroid	Note	76	p	0.00	pub
org.tomdroid	Note	84	p	0.00	pub
org.tomdroid	NoteManager	455	p	0.00	
org.tomdroid.sync.web	OAuthConnection	75	p	0.00	
org.tomdroid.sync.web	SnowySyncService	90	p	0.00	
org.tomdroid.sync.web	WebConnection	64	p	0.00	
org.tomdroid.sync.web	WebConnection	67	p	0.00	
org.tomdroid.sync.web	WebConnection	108	p	0.00	
org.tomdroid.ui	EditNote	85	p	0.00	public clas
org.tomdroid.ui	EditNote	206	c	1.00	
org.tomdroid.ui	EditNote	213	p	0.00	
org.tomdroid.ui	EditNote	834	c	1.00	
org.tomdroid.ui	PreferencesActivity	66	p	-0.22	public clas
org.tomdroid.ui	Tomdroid	80	p	-0.09	public clas
org.tomdroid.ui	Tomdroid	592	p	0.00	
org.tomdroid.ui	Tomdroid	755	p	0.00	pub
org.tomdroid.ui.actionbar	ActionBarHelperBase	296	p	0.00	
org.tomdroid.ui.actionbar	ActionBarHelperBase	297	p	0.00	
org.tomdroid.ui.actionbar	SimpleMenuItem	80	p	0.00	ret
org.tomdroid.util	NoteXMLContentBuilder	91	p	0.00	
org.tomdroid.util	NoteXMLContentBuilder	185	p	0.00	
org.tomdroid.util	NoteXMLContentBuilder	199	p	0.00	
org.tomdroid.util	NoteXMLContentBuilder	209	p	0.00	
org.tomdroid.util	Preferences	33	p	0.00	pub
org.tomdroid.util	Send	53	p	0.00	
org.tomdroid.xml	NoteContentHandler	236	p	0.00	

Figura 3.18 – Lista di linee di codice dettagliata di una differenza

La lista di linee di codice dettagliata è utilizzata solo per indicare all'utente tutte le linee di codice coinvolte nella differenza e l'analisi combinata della colonna *colore* e della colonna *peso* può aiutare a capire quando si sono verificati i casi particolari descritti.

Nello scenario ottimistico, nel caso di linee di codice coperte parzialmente da entrambi gli esperimenti si considera che le porzioni di linee di codice si sovrappongano al minimo possibile.

Nello scenario pessimistico si considera esattamente il caso opposto e, nel caso in cui il peso di copertura del test B sia maggiore di quello del test A, la differenza vale zero.

3.8.2.5 Complemento

La query riporta tutte le linee di codice le quali non sono state coperte ('z') da nessuno dei test (selezionati in input). Per via della sua natura, ovviamente il coverage totale è

pari a zero. Gli unici dati significativi sono il numero di linee di codice coinvolte nella query e la lista dettagliata di linee di codice.

Capitolo 4

Manuale ed esempi d'uso

4.1 Manuale rapido

Nei seguenti paragrafi viene presentato un manuale rapido di utilizzo dei due sottosistemi componenti StoreAndReport. Al solito, il primo paragrafo si riferisce al caricamento dei dati dei report HTML nel database mentre il secondo riguarda il sistema di generazione di report di correlazione.

4.1.1 Caricamento dei dati dei report HTML nel database. Manuale d'uso.

- Nome dell' Eseguibile: *StoreAndReport.jar*.
- Scopo: *Caricare gli esperimenti in un database*.
- Main Class: *LoadCoverages*.

Precondizioni

Deve essere installato e avviato un server MySql sul pc dove viene eseguito il tool.

Il server deve essere impostato con i parametri:

- Porta: 3306
- Username: root

- Password: root

Comando per avviare il tool

Dal Windows Prompt:

```
java -cp {Tool-Path}\StoreAndReport.jar storeandreport.store.LoadCoverages
[parametri di input]
```

comando alternativo:

```
java –jar  {Tool-Path}\StoreAndReport.jar [parametri di input]
```

Spiegazione dei parametri

1. *Tool-Path*: Percorso del tool StoreAndReport.jar sul File System.
2. *Parametri di input*: Il sottosistema può avviarsi in due modalità: interattiva o automatica. Per avviare la versione interattiva l'utente non deve specificare nessun parametro di input, basta solo inserire il Tool-Path, dopodiché i parametri di input saranno richiesti dal sistema di volta in volta, per cui l'utente viene guidato nell'inserimento.

   Si devono indicare gli input secondo un criterio di ordine per avviare la versione automatica, cioè quella dove il sistema analizza automaticamente i parametri di input e produce il suo output.

Parametri di input e ordine da rispettare per l'inserimento

- Nome dell'applicazione.
- Versione.

- Path.
- Tipo di test (del path precedente) / oppure un altro path.
- Tipo di test (del path precedente) / oppure un altro path.
- …
- Tipo di test (del path precedente) / oppure un altro path..

Descrizione degli input e azioni

- *Nome dell'applicazione* - Diventerà il nome del database. Viene presa una sottostringa di 64 caratteri se il nome inserito ha una lunghezza maggiore. Evitare di inserire caratteri speciali se non si vuole causare la terminazione anticipata del programma.
- *Versione* - Versione alla quale appartengono gli esperimenti che si vogliono caricare nel database. Viene presa una sottostringa di 30 caratteri se viene inserita una stringa più lunga di questo valore. Evitare di inserire caratteri speciali se non si vuole causare la terminazione anticipata del programma
- *Path* - Percorso dove risiede il file index.html di un esperimento che si vuole caricare nel database.
- *Tipo di test* (Facoltativo) - Se inserito, indica il tipo di test del path che lo precede. Viene presa una sottostringa di 50 caratteri se viene inserita una stringa più lunga di questo valore. Nel caso in cui il tipo di test per un path è omesso, viene inserito automaticamente inserito per esso la data di generazione dell'esperimento come tipo di test.

Nota:

Input obbligatori: nome dell'applicazione, la sua versione, almeno un path.

Esempi di sequenze corrette di inserimento dei parametri di input:

1. applicazione - versione - path1
2. applicazione - versione - path1 - testType1 - path2 - testType2 - path3 - testType3
3. applicazione - versione - path1 - path2 - testType2 - path3 - testType3
4. applicazione - versione - path1 - path2 - path3
5. applicazione - versione - path1 - testType1 - path2 - testType2 - path3 - path4

Si noti che i primi tre input sono sempre uguali, quello che può variare a discrezione dell'utente è l'inserimento dei parametri dal quarto in poi.

Esempio di utilizzo dei parametri di avvio e dei comandi

L'eseguibile del tool è localizzato al path (*Tool-Path*) : "C:\tool"
Si vuole caricare in un database il report HTML di due test diversi, uno di *tipo* "depth intensive" e l'altro di *tipo* "breadth intensive", condotti sulla *versione* 2 dell'app Android di *nome* Tomdroid.
I due test sono localizzati ai *path*:

1. c:\tomdroid\depth_intensive
2. c:\tomdroid\breadth_intensive

Il comando per utilizzare la versione *automatica* è:

```
java –cp C:\tool\StoreAndReport.jar storeandreport.store.LoadCoverages
tomdroid 2 "c:\tomdroid\depth_intensive" "depth intensive"
"c:\tomdroid\breadth_intensive" "breadth intensive"
```

Se si vuole utilizzare la versione *interattiva* ed inserire in modo guidato gli input il comando da utilizzare è:

```
java –cp c:\Tool\StoreAndReport.jar storeandreport.store.LoadCoverages
```

oppure, in alternativa:

```
java –jar C:\Tool\StoreAndReport.jar
```

4.1.2 Generazione di report di correlazione. Manuale d'uso.

Nome dell' Eseguibile: *StoreAndReport.jar*.
Scopo: *Generazione di report di correlazione fra test memorizzati nel database*.
Main Classes (funzioni): *Total, Union, Differences, Intersection, Complement*.

Precondizioni

Deve essere installato e avviato un server MySql sul pc dove viene eseguito il tool.
Deve essere presente un Database sul server MySql creato precedentemente con questo tool stesso ed al suo interno devono essere presenti le informazioni di copertura di almeno due test.
Il server deve essere impostato con i seguenti parametri

- Porta: 3306
- Username: root
- Password: root

Comando per avviare il tool

Dal Windows Prompt:

```
java -cp {Tool-Path}\StoreAndReport.jar
storeandreport.report.[Main-Class] [parametri di input]
```

Spiegazione dei parametri

1. *Tool-Path*: Percorso del tool StoreAndReport.jar sul File System.
2. *Main-Class*: "Union" se si vuole un report dell'unione. "Intersection" se si vuole un report dell'intersezione. "Complement" se si vuole un report del complemento. "Differences" se si vuole un report delle differenze semplici
3. *Parametri di input*: Nome del database, versione, esperimenti di cui si vuole la correlazione. Deve essere rispettato un ordine preciso per inserire questi dati

Nota:

Nel caso di Differenze semplici, vengono generati n*(n-1) report se n è il numero di esperimenti indicati in input (numero di disposizioni di n elementi presi a due alla volta).

Parametri di input ed ordine da rispettare per l'inserimento

- Nome Database (obbligatorio)
- Versione.
- Esperimento 1.
- Esperimento 2.
- …
- Esperimento n.

Descrizione degli input

- *Nome Database:* è il nome del database che è stato utilizzato per memorizzare i test dell'applicazione.
- *Versione:* versione per la quale sono memorizzati i test di cui si vuole la correlazione.
- *Esperimento:* si intende un test precedentemente memorizzato nel database nella tabella "Experiments" ed afferente alla versione indicata nel secondo parametro.

  Per ogni esperimento, nel database sono memorizzate le seguenti informazioni con le quali si può identificare univocamente un test:
  - exp_id: Chiave primaria della tabella Experiments (*int*).
  - exp_date: (formato *datetime*) Data di generazione da parte di EMMA della struttura di file html del test.

Nota:

Il Nome del database è obbligatorio, la versione è facoltativa.

Se invece si vuole indicare esattamente di quali test si vuole la correlazione, la versione diventa un input obbligatorio e gli esperimenti indicati devono essere almeno due.

Per la data dell'esperimento da inserire in input essa si può indicare sia in formato *Datetime*, cioè come è memorizzato nel database, sia nel formato che è riportato in testa ad ogni pagina HTML navigando nei report generati con EMMA.

Combinazioni input-output

1. Se l'utente digita solo il nome del Database, per ogni versione vengono effettuate le correlazioni di tutti i test (se sono almeno due) memorizzati per essa.

2. Se l'utente digita il nome del Database e la versione, vengono effettuate le correlazioni di tutti i test memorizzati per essa.
3. Se l'utente digita il nome del Database, la versione e almeno due esperimenti, allora vengono effettuate le correlazioni degli esperimenti indicati.

Per ogni correlazione effettuata viene prodotto un report testuale che ne descrive i risultati.

Esempi di utilizzo dei parametri di avvio e dei comandi

L'eseguibile del tool è localizzato al path (*Tool-Path*) : "C:\tool"

Nel *database* Omnidroid sono stati caricati precedentemente 4 *esperimenti* della *versione* 1.1 e 5 *esperimenti* della *versione* 2 di test eseguiti sull'app omnidroid.

Intersezione:

Se ne vogliono i report dell'*intersezione* di tutti gli esperimenti della versione 1.1.

Parametri:

Main-Class: Intersection

Parametro 1: omnidroid

Parametro 2: 1.1

Comando di esecuzione:

```
java -cp c:\StoreAndReport.jar storeandreport.report.Intersection
omnidroid 1.1
```

Unione

Si vuole stampare un report di Unione relativi a due specifici esperimenti appartenenti alla versione 2 di omnidroid.

Esperimento 1: exp_id = "1" / Data = "Sat Jun 07 01:24:32 CEST 2014"

Esperimento 2: exp_id = 2" / Data = "Sat Jun 07 01:23:54 CEST 2014"

Degli esperimenti si può specificare sia la data che l'exp_id per cui abbiamo che i parametri possono essere immessi secondo diverse combinazioni.

Parametri:

Main-Class: Union

1 Parametro1(DB): omnidroid

2 Parametro2(Versione): 2

3 Parametro3(Exp1): 1

4 Parametro4(Exp2): 2

Oppure:

3 Parametro3(Exp1): "Sat Jun 07 01:24:32 CEST 2014"

4 Parametro4(Exp1): "Sat Jun 07 01:23:54 CEST 2014"

(Oppure altre combinazioni in cui c'è una data e un exp_id)

Comando:

```
java -cp c:\StoreAndReport.jar storeandreport.report.Union
omnidroid 2 Sat Jun 07 01:24:32 CEST 2014 Sat Jun 07 01:23:54 CEST
2014
```

oppure:

```
java -cp c:\StoreAndReport.jar storeandreport.report.Union
omnidroid 2 "Sat Jun 07 01:24:32 CEST 2014" 2
```

oppure:

```
java -cp c:\StoreAndReport.jar storeandreport.report.Union
omnidroid 2 1 2
```

Tutti i Report

Si vogliono generare, del *database* omnidroid, tutti i report (Differenze, intersezione, unione, complemento) relativi a tutti gli esperimenti memorizzati per ogni versione presente nel database.

Comando

```
java -cp c:\StoreAndReport.jar storeandreport.report.Total
omnidroid
```

4.2 Esempio di utilizzo del sistema e degli output generati

Nel seguito viene presentato un esempio concreto e completo di utilizzo del software. Viene mostrato come tre report in HTML vengono dapprima "trasferiti" in un database con l'ausilio del primo sottosistema e poi come, successivamente, vengono effettuate tutte le correlazioni supportate, tra gli stessi test caricati nel database, con relativa generazione dei report in formato testuale che ne riportano i risultati.
I tre report HTML di cui si vogliono conoscere le correlazioni sono riguardanti test eseguiti con tecniche diverse su un'applicazione Android di nome Omnidroid.

4.2.1 Caricamento dati di test su Omnidroid nel database

Nel seguito viene mostrato l'esempio di utilizzo della versione interattiva del sistema di caricamento dei dati nel database.
In figura 4.1 viene mostrata una fotografia della prima fase, in cui è stato richiesto di inserire il nome e la versione del software di cui si vuole caricare i test.
In questo esempio un database di nome *omnidroid* non esiste ancora e allora StoreAndReport ha informato di questo l'utente, quindi ha provveduto a crearne uno nuovo col nome "omnidroid" dopo aver richiesto e ricevuto in input la versione. Dopo

questa fase iniziale viene richiesto all'utente di inserire i path dove sono contenuti i file index.html (ed il tipo di test per ogni path).

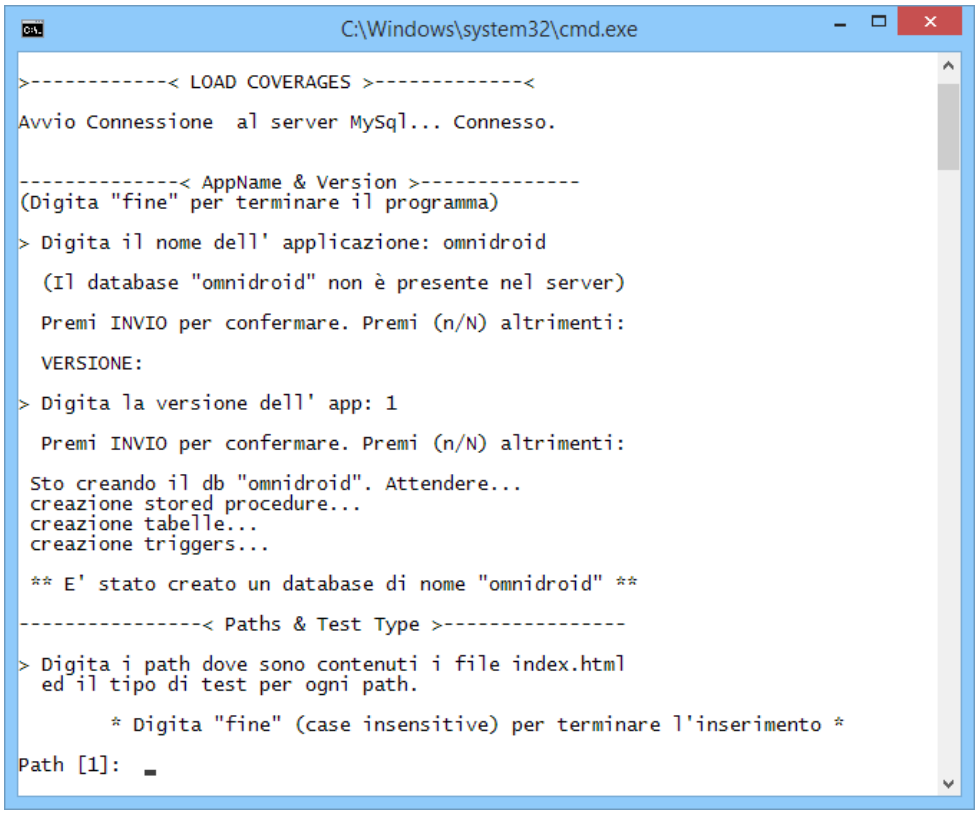


Figura 4.1 – Richiesta nome e versione del software testato.

In figura 4.2 è mostrata la seconda fase di richiesta degli input in cui l'utente deve inserire almeno un path e, facoltativamente, il tipo di test per ogni path. I primi tre path inseriti sono corretti, cioè contengono un *index.html* creato con EMMA, in tal caso il sistema ha informato l'utente che i path inseriti hanno passato i test di correttezza (con un: * ok *).
All'inserimento del quarto path sono stati inseriti un path che non conteneva un file index.html e lo stesso path inserito per terzo, in entrambi i casi il sistema si è accorto dell'errore ed ha informato l'utente che gli input inseriti erano errati.
L'inserimento della parola chiave "fine" corrisponde alla terminazione della fase di input e all'avvio di quella di elaborazione nella quale i file HTML contenuti nei path inseriti

vengono analizzati per estrapolarne le informazioni che poi saranno memorizzate nel database.

L'utente viene informato su quali path sono analizzati di volta in volta. Se c'è un errore, il programma prova ad indicare esattamente in quale punto si è verificato mostrandolo a schermo.

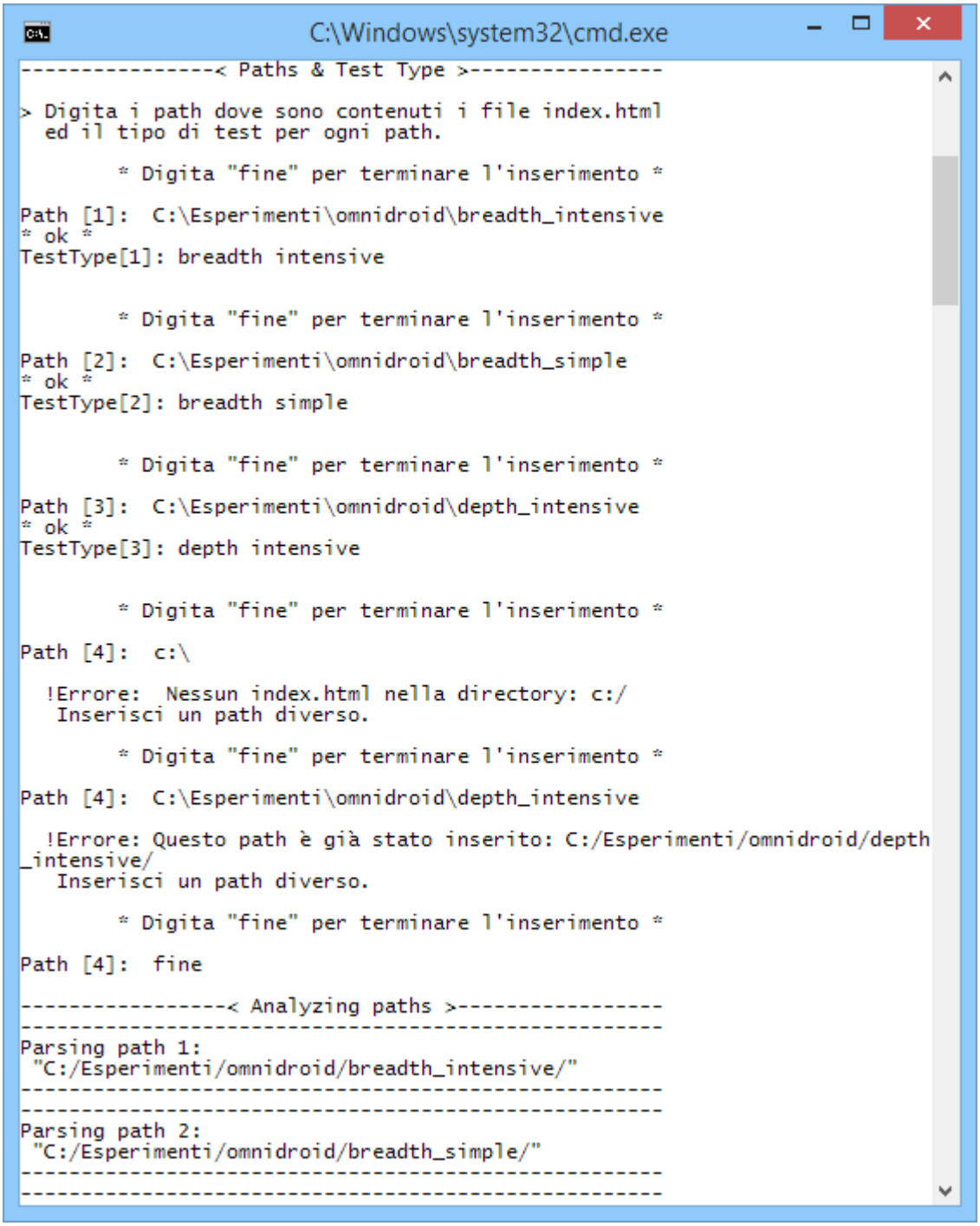


Figura 4.2 – Inserimento path e tipo di test. Inizio elaborazione

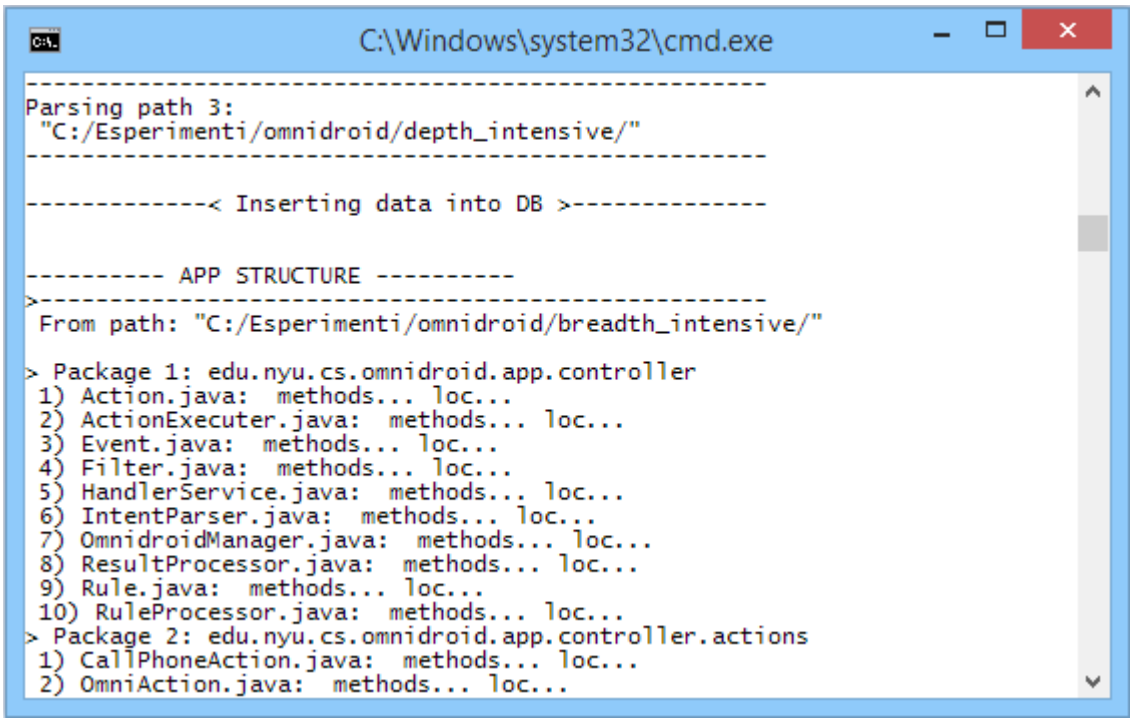


Figura 4.3 – Fine elaborazione. Inizio inserimento struttura applicazione nel DB

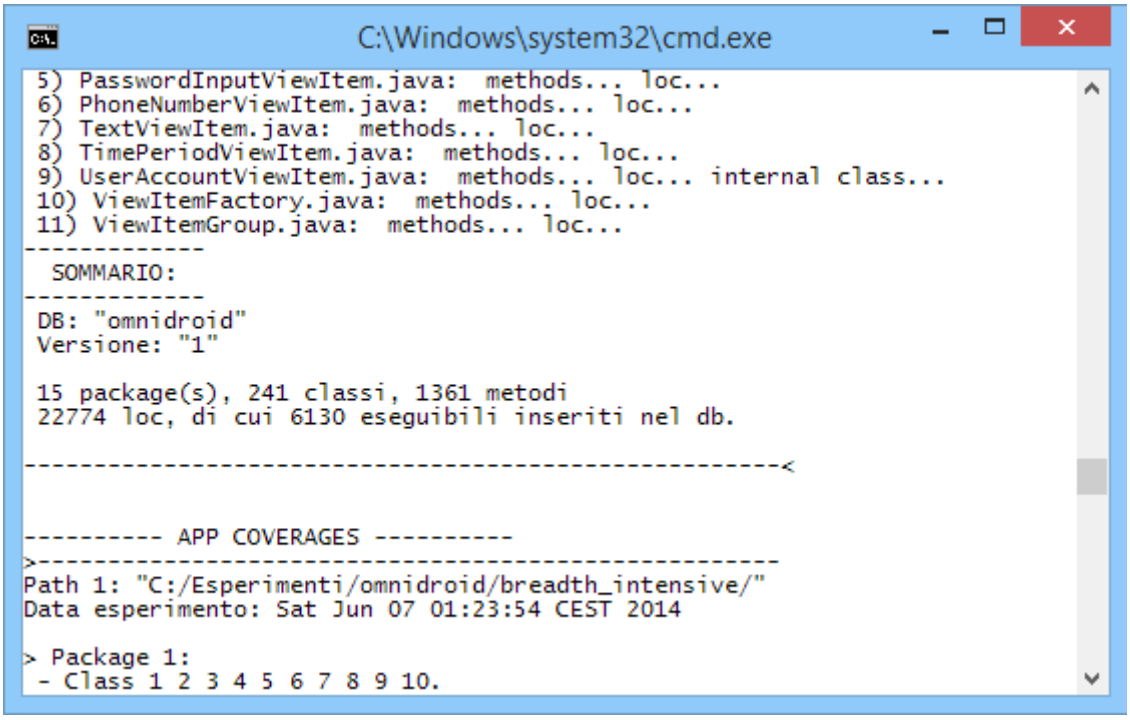


Figura 4.4 – Fine inserimento struttura. Inizio inserimento coverages nel DB

Terminata la fase di estrazione dei dati dai report HTML, inizia quella dell'inserimento della struttura dell'applicazione nel database come mostrato in figura 4.3. L'utente viene informato sui package e le classi che si stanno inserendo mano a mano. Se c'è un errore, il programma cerca di mostrare esattamente di che natura è e dove si è verificato, mostrandolo all'utente, in tal caso si tenta di inserire i dati della struttura dell'applicazione da altri path eventualmente inseriti in input, altrimenti il programma elimina i dati incompleti nel database e termina segnalando l'errore.

Come si vede nella figura 4.4, terminata la fase di inserimento della struttura dell'applicazione, il sistema mostra un piccolo sommario riassuntivo sui dati inseriti: il nome e la versione, il numero di package, di classi, di linee di codice in totale e di linee di codice eseguibili per l'applicazione indicata.

La fine della fase di inserimento della struttura dell'applicazione coincide con l'inizio di quella dell'inserimento dei dati di coverage di ogni test indicato in input, il quale viene meglio mostrato in figura 4.5

Nel nostro caso i test indicati all'inizio sono tre e per ognuno di essi il programma indica all'utente quale test sta inserendo, mostrandone il path e la data. Inoltre viene riportato per quale package e quale classe si stanno inserendo i dati di copertura delle linee di codice e alla fine dell'inserimento viene mostrato un piccolo sommario che indica il numero dei record inseriti e con quale valore di *exp_id* il test relativo è stato memorizzato nella tabella *experiments* del database.

A riguardo si veda la figura 4.6 che mostra la situazione nella tabella *experiments* al termine di tutti gli inserimenti. In ogni record viene inserita la data di generazione dei test HTML, il tipo di test e la versione dell'applicazione sul quale sono stati condotti.

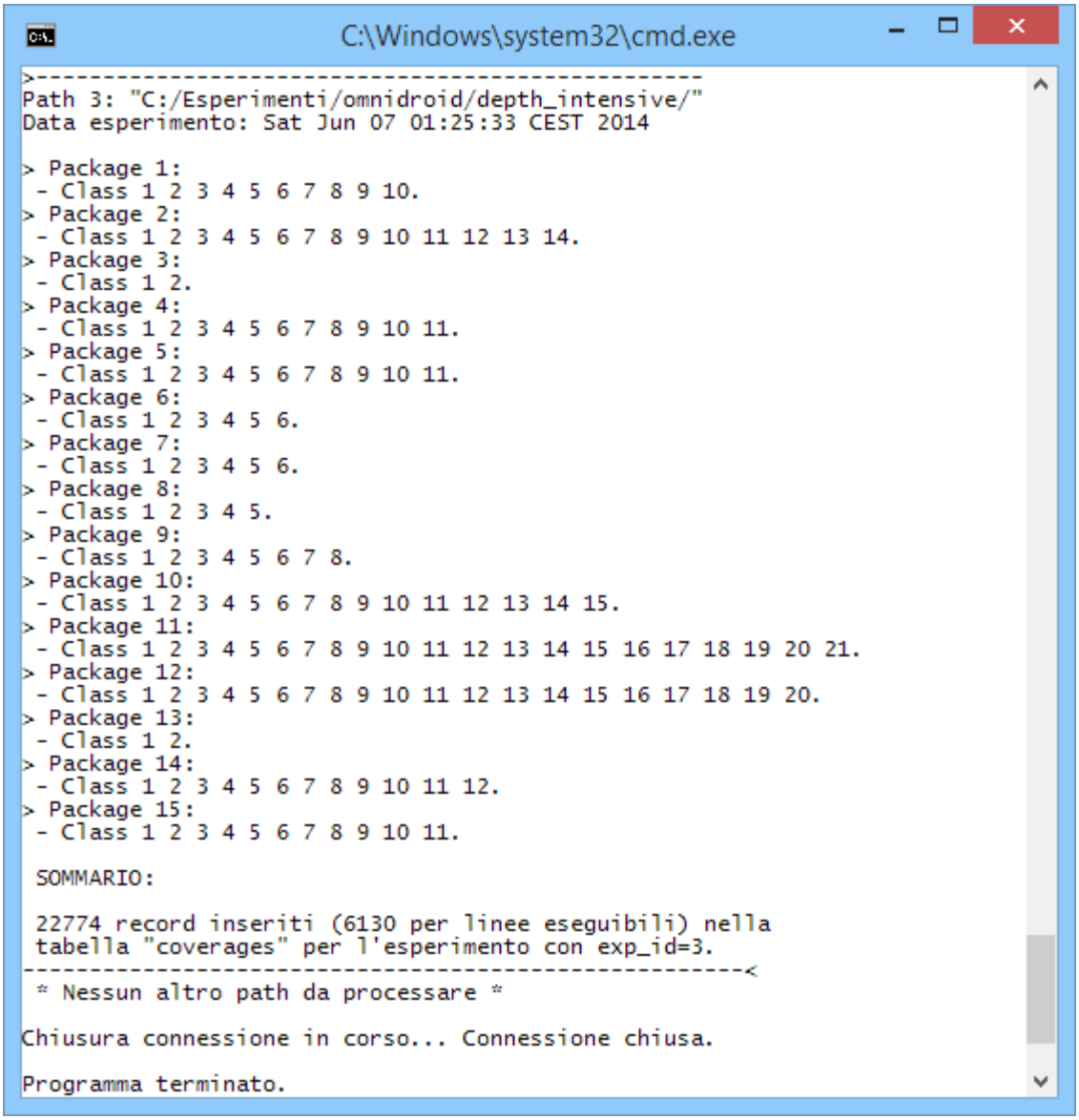


Figura 4.5 – Fine inserimento coverages nel DB. Terminazione del programma.

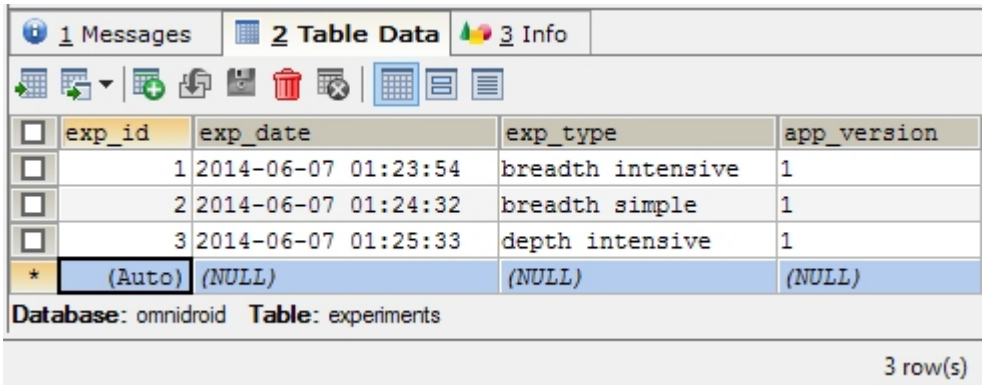
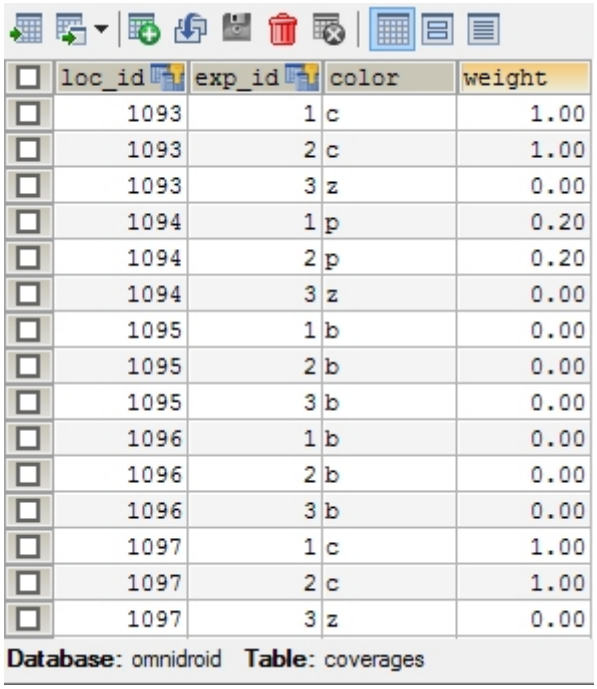


Figura 4.6 – Tabella *experiments* con i dati di tre test.

Nella figura 4.7 vengono mostrati come si presentano alcuni record della tabella coverages al termine dell'applicazione.

Si può notare che sono memorizzati i coverage relativi a cinque linee di codice in particolare. Ogni linea di codice è ripetuta tre volte, una per ogni test memorizzato, e nella colonna *color* e *weight* vengono mostrati la copertura ed il peso relativi ai test rappresentati dall'attributo *exp_id*.

Possiamo osservare che la linea *1093* è stata coperta dai test *1* e *2* ma non dal test *3*, la linea *1094* è stata coperta parzialmente per il 20% dai test *1* e *2* ma non è stata coperta dal test *3*, le linee *1095*, *1096* non sono eseguibili come è possibile notare dall'attributo *color* che ha valore *b*. Infine, la stessa situazione della linea 1093 si ripresenta per la linea *1097*



loc_id	exp_id	color	weight
1093	1	c	1.00
1093	2	c	1.00
1093	3	z	0.00
1094	1	p	0.20
1094	2	p	0.20
1094	3	z	0.00
1095	1	b	0.00
1095	2	b	0.00
1095	3	b	0.00
1096	1	b	0.00
1096	2	b	0.00
1096	3	b	0.00
1097	1	c	1.00
1097	2	c	1.00
1097	3	z	0.00

Database: omnidroid Table: coverages

Figura 4.7 – Tabella *coverages* con i dati di coverage 5 linee di codice

4.2.2 Generazione dei report

Nella figura 4.8 viene mostrato il procedimento di richiesta e generazione dei report testuali, relativi alla correlazione dei dati di coverage presenti nel database creato precedentemente, come mostrato nel paragrafo 4.2.1. In esso abbiamo visto che sono stati memorizzati i report di copertura relativi a tre diversi test eseguiti sull'applicazione Omnidroid.

Nell'esempio che segue viene richiesto di generare tutti i report supportati dal sistema, cioè Differenza, Intersezione, Unione, Complemento.

Come è possibile notare dalla prima linea della figura 4.8, è stata usata la sintassi discussa nel paragrafo 4.1.2 per il comando da inserire nel prompt dei comandi di Windows. In particolare come Main-Class è stata utilizzata *Total* (la quale permette di generare tutti i report supportati dal programma), come Tool-Path è stato inserito "*c:\Tool*" poiché in tale locazione si l'eseguibile del programma e, come parametro di input, è stato inserito *omnidroid*. Il solo nome del database è sufficiente perché vengono calcolate le correlazioni per ogni versione e per ogni test memorizzato per quella versione, nel nostro caso c'è una sola versione e, per essa, tre esperimenti.

A titolo di esempio si riportano qui di seguito quattro comandi alternativi che generano esattamente gli stessi report:

- `java -cp c:\Tool\StoreAndReport.jar storeandreport.report.Total omnidroid 1`
- `java -cp c:\Tool\StoreAndReport.jar storeandreport.report.Total omnidroid 1 1 2 3`
- `java -cp c:\Tool\StoreAndReport.jar storeandreport.report.Total omnidroid 1 1 2 "Sat Jun 07 01:25:33 CEST 2014"`

- `java -cp c:\Tool\StoreAndReport.jar storeandreport.report.Total omnidroid 1 1 “2014-06-07 01:24:32” “Sat Jun 07 01:25:33 CEST 2014”`

Nel primo è indicata il database e la versione, negli ultimi tre sono indicati il database, la versione e i test dei quali si può indicare sia il valore dell’attributo di *exp_id* che dell’attributo *exp_date*, così come sono riportati nella tabella *experiments* del database, che della data in formato *datetime*, così come è riportata nei file HTML relativi ai report di test.

Una volta avviato, il sistema controlla gli input e, se non ci sono stati errori, genera i report richiesti indicandone la locazione sul File System ed il nome di ogni singolo report.

Nelle figure 4.9 e 4.10 viene invece mostrato come sono organizzati gli output del sistema sul File System e come sono generati i nomi dei file di report.

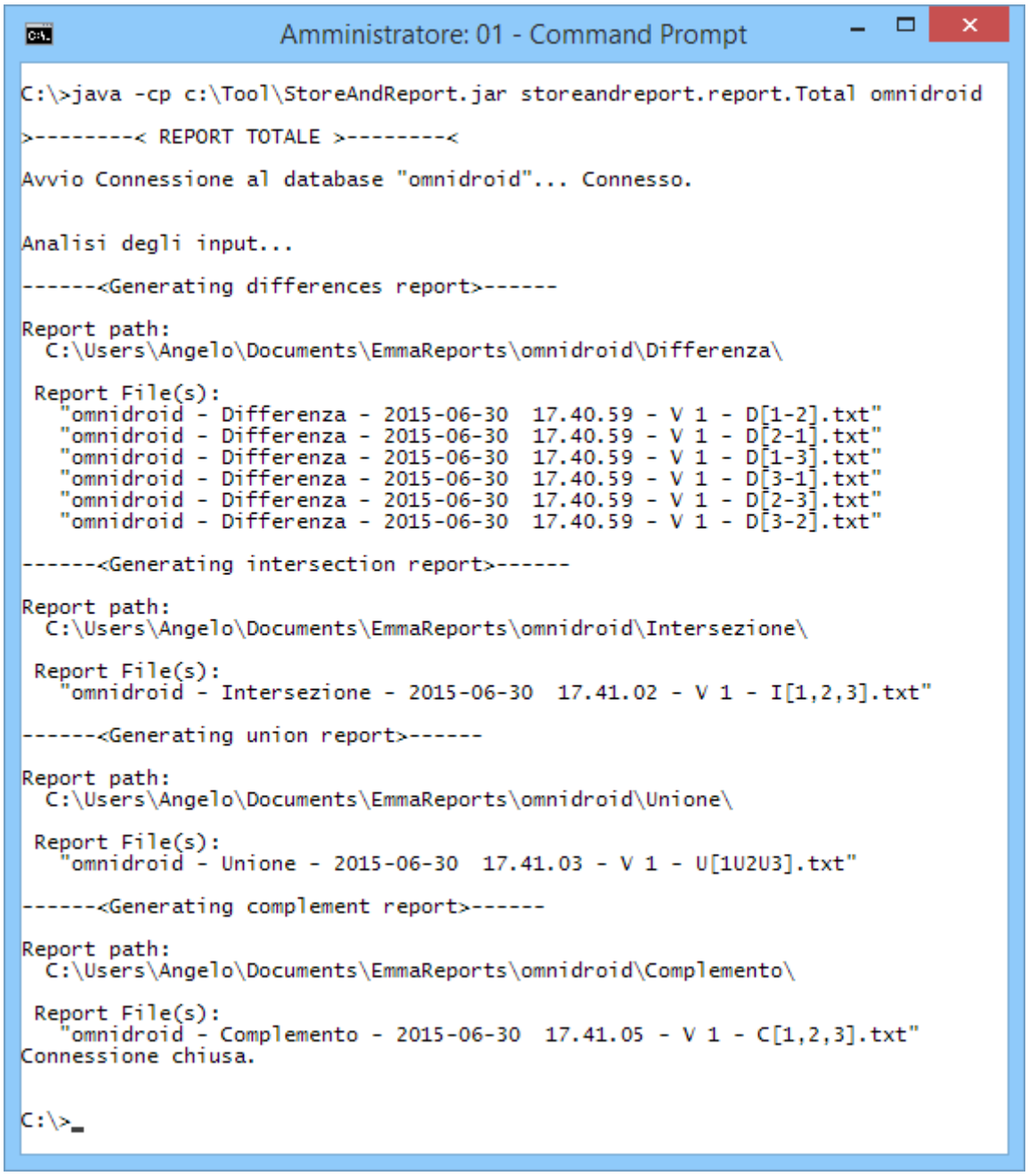


Figura 4.8 – Generazione report di correlazione.

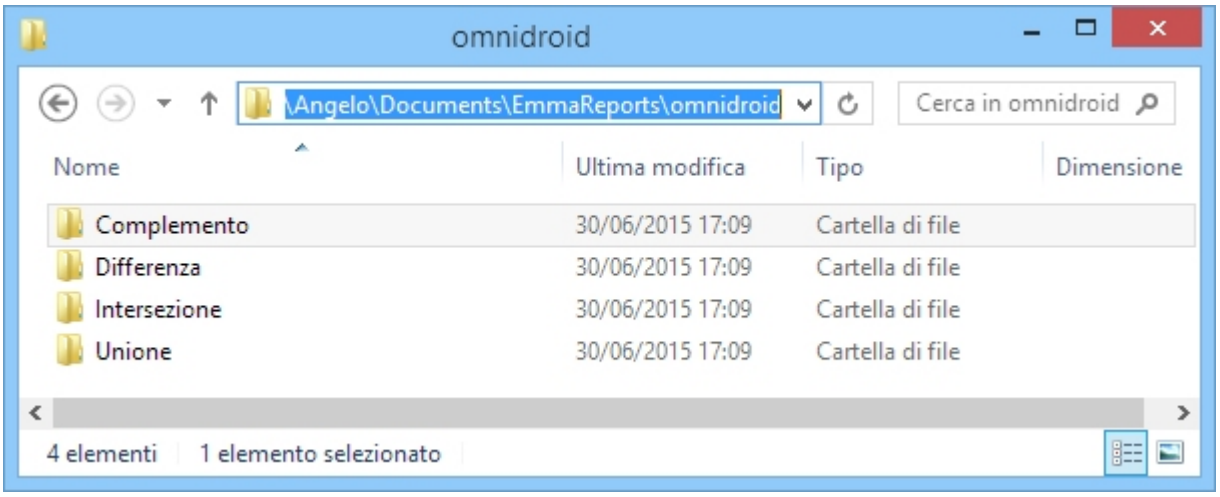


Figura 4.9 – Locazione sul File System dei report testuali.

Per ogni file nel nome vengono riportate le seguenti informazioni:

{dbname} - {queryname} - {reportDate} - {Version} - {ExpList}

Cioè nome del database, nome della correlazione, Data e ora di generazione del report, versione e lista di esperimenti (exp_id) che sono stati correlati tra loro.
In particolare, nel riportare la lista di esperimenti nel nome vengono seguite le seguenti regole:

- Per le differenze viene anteposto il simbolo “D” e tra i due esperimenti elencati tra parentesi quadre viene inserito il simbolo “-” .
- Per l’unione viene anteposto il simbolo “U” e tra gli esperimenti, elencati tra parentesi quadre, viene usato lo stesso simbolo.
- Per l’intersezione viene anteposto il simbolo “I” e tra gli esperimenti, elencati tra parentesi quadre, viene inserito il simbolo “,”.
- Per il complemento viene anteposto il simbolo “C” e tra gli esperimenti, elencati tra parentesi quadre, viene inserito il simbolo “,”.

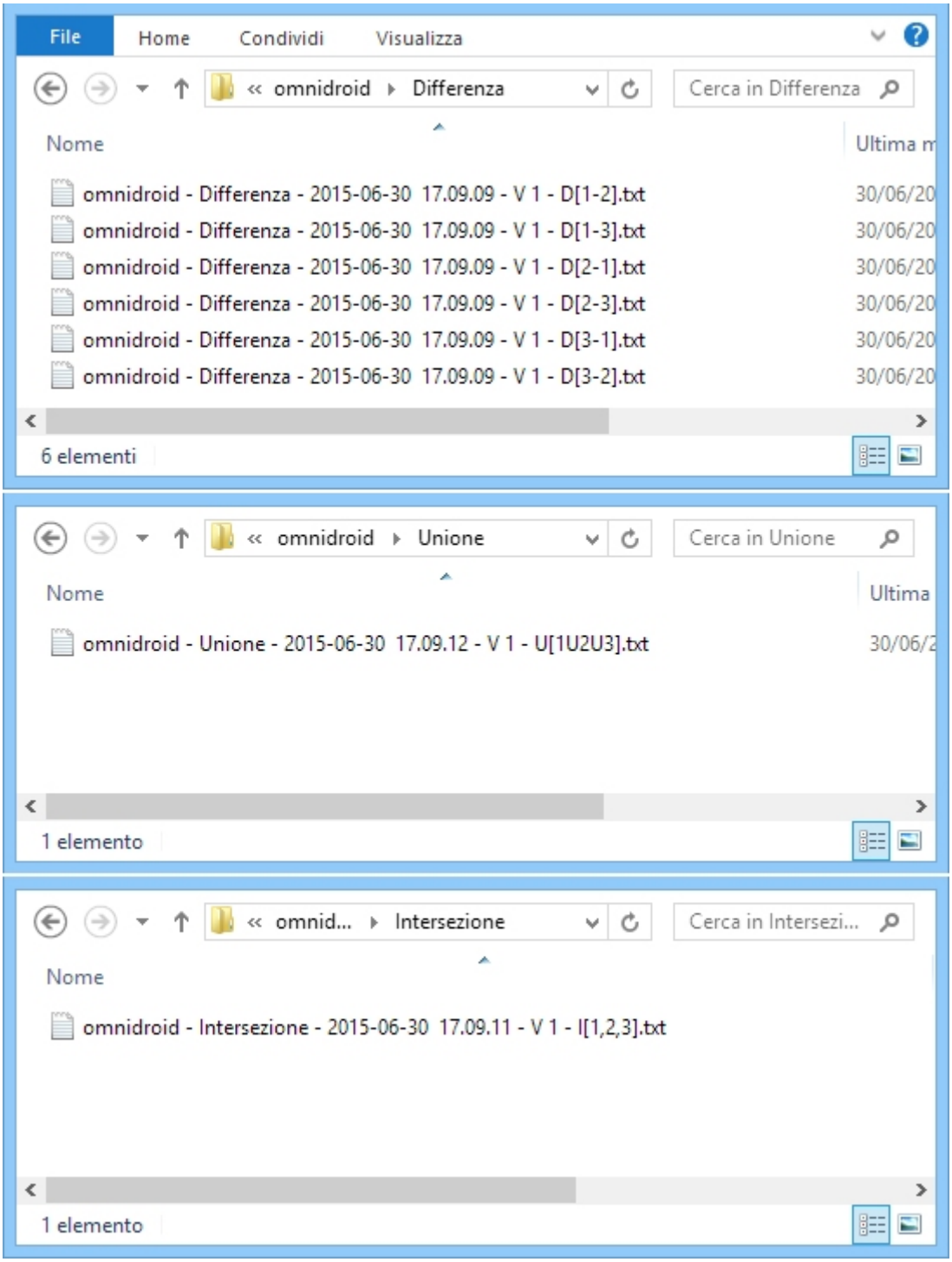


Figura 4.10 – File testuali di report di correlazione.

4.2.3 Descrizione di un file di report testuale generato per una correlazione di unione

In figura 4.10 viene mostrato il contenuto del report di unione dei tre testi inseriti nel database come visto al paragrafo 4.2.1 e richiesto al paragrafo 4.2.2. Esso si compone di quattro box.

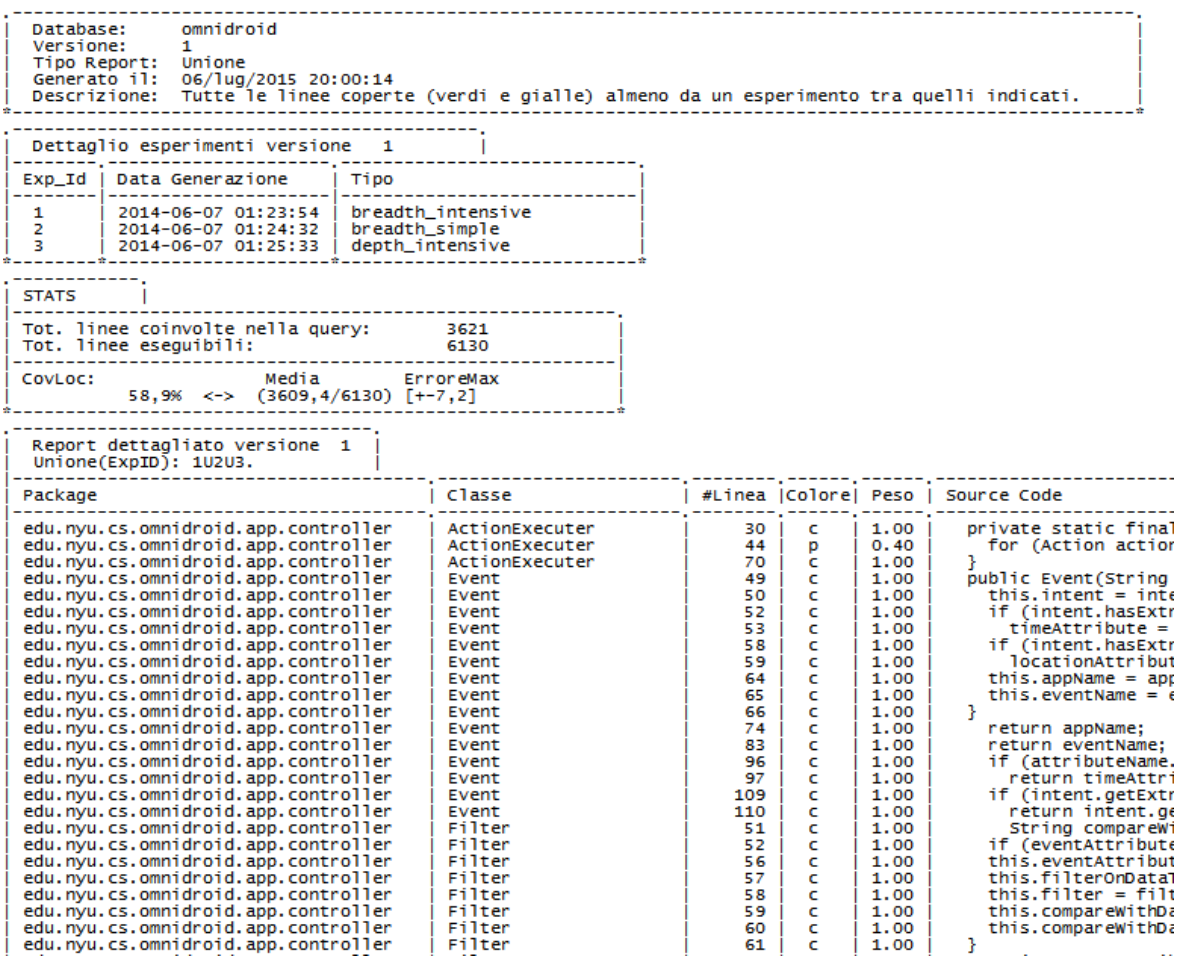
Nel primo vengono date informazioni generali riguardanti il nome del database interrogato, la versione degli esperimenti di cui si è chiesta la correlazione, il tipo di correlazione, la data di generazione del report testuale e una breve descrizione del significato della correlazione.

Nel secondo viene mostrata la lista di esperimenti selezionati per la correlazione, in pratica viene riprodotta la tabella experiments del database (eccetto che per la colonna *app_version*) per la lista di esperimenti di cui si è richiesta la correlazione.

Nel terzo ci sono statistiche generali sul numero di linee di codice e sul coverage totale risultante dopo la correlazione, espresso in termini percentuali e frazionari e con riporto dell'incertezza. Tali informazioni sono state discusse in particolare nel paragrafo 3.8.2.1.

Nel quarto sono riportate tutte le linee di codice risultanti dalla correlazione sotto forma di lista dettagliata, specificando per ogni linea di codice, il package e la classe a cui essa appartiene. il numero di linea di codice all'interno della classe, il colore ed il peso del coverage combinato dei tre test, e il testo del codice sorgente.

Lo stesso schema di presentazione è ripetuto per le altre tre correlazioni, le uniche differenze che intercorrono fra loro è nel significato delle colonne *colore* e *peso* della lista dettagliata delle linee di codice presentate nel quarto box. Per una spiegazione si rimanda al paragrafi 3.8.2.2, 3.8.2.3, 3.8.2.4, 3.8.2.5.



Database: omnidroid
Versione: 1
Tipo Report: Unione
Generato il: 06/lug/2015 20:00:14
Descrizione: Tutte le linee coperte (verdi e gialle) almeno da un esperimento tra quelli indicati.

Dettaglio esperimenti versione 1

Exp_Id	Data Generazione	Tipo
1	2014-06-07 01:23:54	breadth_intensive
2	2014-06-07 01:24:32	breadth_simple
3	2014-06-07 01:25:33	depth_intensive

STATS

Tot. linee coinvolte nella query: 3621
Tot. linee eseguibili: 6130

CovLoc:	Media	ErroreMax
58,9%	<-> (3609,4/6130)	[+-7,2]

Report dettagliato versione 1
Unione(ExpID): 1U2U3.

Package	Classe	#Linea	Colore	Peso	Source Code
edu.nyu.cs.omnidroid.app.controller	ActionExecuter	30	c	1.00	private static final
edu.nyu.cs.omnidroid.app.controller	ActionExecuter	44	p	0.40	for (Action actior
edu.nyu.cs.omnidroid.app.controller	ActionExecuter	70	c	1.00	}
edu.nyu.cs.omnidroid.app.controller	Event	49	c	1.00	public Event(String
edu.nyu.cs.omnidroid.app.controller	Event	50	c	1.00	this.intent = inte
edu.nyu.cs.omnidroid.app.controller	Event	52	c	1.00	if (intent.hasExtr
edu.nyu.cs.omnidroid.app.controller	Event	53	c	1.00	timeAttribute =
edu.nyu.cs.omnidroid.app.controller	Event	58	c	1.00	if (intent.hasExtr
edu.nyu.cs.omnidroid.app.controller	Event	59	c	1.00	locationAttribut
edu.nyu.cs.omnidroid.app.controller	Event	64	c	1.00	this.appName = app
edu.nyu.cs.omnidroid.app.controller	Event	65	c	1.00	this.eventName = e
edu.nyu.cs.omnidroid.app.controller	Event	66	c	1.00	}
edu.nyu.cs.omnidroid.app.controller	Event	74	c	1.00	return appName;
edu.nyu.cs.omnidroid.app.controller	Event	83	c	1.00	return eventName;
edu.nyu.cs.omnidroid.app.controller	Event	96	c	1.00	if (attributeName.
edu.nyu.cs.omnidroid.app.controller	Event	97	c	1.00	return timeAttri
edu.nyu.cs.omnidroid.app.controller	Event	109	c	1.00	if (intent.getExtr
edu.nyu.cs.omnidroid.app.controller	Event	110	c	1.00	return intent.ge
edu.nyu.cs.omnidroid.app.controller	Filter	51	c	1.00	String compareWi
edu.nyu.cs.omnidroid.app.controller	Filter	52	c	1.00	if (eventAttribute
edu.nyu.cs.omnidroid.app.controller	Filter	56	c	1.00	this.eventAttribut
edu.nyu.cs.omnidroid.app.controller	Filter	57	c	1.00	this.filterOnDataI
edu.nyu.cs.omnidroid.app.controller	Filter	58	c	1.00	this.filter = filt
edu.nyu.cs.omnidroid.app.controller	Filter	59	c	1.00	this.compareWithDa
edu.nyu.cs.omnidroid.app.controller	Filter	60	c	1.00	this.compareWithDa
edu.nyu.cs.omnidroid.app.controller	Filter	61	c	1.00	}

Figura 4.11 – Report testuale di unione di tre test.

Conclusioni

Si può a questo punto parlare finalmente dell'utilità dello strumento realizzato. Per farlo possiamo ricollegarci direttamente all'introduzione, precisamente alle pagine 3 e 4. In quel punto si era descritto uno scenario particolare in cui un *tester* può venire a trovarsi: una volta condotti due esperimenti di test su una applicazione ed averne prodotto dei rapporti dettagliati di copertura tramite EMMA, il tester ha necessità di un'analisi più approfondita delle differenze e dei punti in comune dei cammini di esecuzione percorsi nei singoli test:

1. se ci sono state differenze nei cammini di esecuzione, in quali punti del codice si sono verificate?
2. E quali linee di codice sono state eseguite in comune dai due test?
3. Qual è la reale percentuale di copertura che si ottiene sommando la copertura delle linee di codice coperte da un test A ma non dal test B e quelle coperte dal test B ma non dal test A?
4. Quali sono le linee di codice non coperte da nessuno dei due test?
5. Posso stabilire dei nuovi criteri di terminazione dei singoli test, usando diverse strategie, ponendo come obiettivo che la percentuale di copertura aggregata raggiunga una certa soglia prestabilita?

Alla fine di questa lunga esposizione si è mostrato come il programma sviluppato offre una risposta diretta a questi quesiti traducendoli in query insiemistiche, le quali considerano come insieme universo tutte le linee di codice eseguibili dell'applicazione (quelle che nei singoli report sono colorate di rosso, verde o giallo. Vedi par. 2.3.2.1).

Allora:

1. la differenza tra un test A ed un test B viene tradotta in differenza insiemistica mostrando come risultato le linee di codice che sono state coperte da A ma non da B.
2. Le linee di codice eseguite da entrambi i test viene tradotta in query di Intersezione tra i due test delle linee di codice che sono verdi e gialle.
3. La percentuale di copertura aggregata, con l'indicazione delle linee di codice che contribuiscono a formarla, può essere calcolata effettuando l'unione delle linee di codice gialle e verdi dei due esperimenti.
4. Le linee di codice che non sono state coperte da nessun test possono essere calcolate effettuando una query di intersezione delle linee di codice rosse di ambo i test (complemento).

La risposta alla quinta domanda è un po' più difficile da ottenere, sarebbe necessario integrare il sistema con gli strumenti di testing automatico e ciò potrebbe essere oggetto di sviluppi futuri del software. La base di partenza è comunque relativa ad una delle correlazioni di base implementate, cioè l'Unione.
Il software è stato realizzato con in mente sin dall'inizio la realizzazione delle quattro esigenze di correlazione mostrate e la presentazione dei risultati ottenuti su file di testo. Nel paragrafo 3.8.2 è possibile capire più precisamente cosa calcolano le varie query sviluppate.

Nel paragrafo 4.2.3 è riportato un esempio di un report prodotto dopo l'unione di tre test effettuati su un'applicazione. Si può consultare l'intero paragrafo 4.2 per seguire un esempio completo di utilizzo del sistema che mostra il processo di trasferimento dei dati dei report HTML in un database e successiva generazione di report.

Infine un ultimo appunto sui tempi di esecuzione. Il caricamento dei dati nel database può impiegare dai pochi secondi ai pochi minuti, a seconda di quante linee di codice è composta l'applicazione e di quanti testi alla volta si caricano. L'esecuzione delle query in un database già popolato e la produzione dei risultati viene fatta in pochi secondi. Parlando quindi del tempo necessario per ottenere un risultato di correlazione, esso si può ritenere ragionevole se rapportato al tempo necessario per eseguire i test che può arrivare ad essere misurato in ore o addirittura giorni.

Sviluppi Futuri

Parlando di output del software, sebbene la produzione di report testuali sia già un buon modo per analizzare le correlazioni che intercorrono tra due o più risultati di test, sarebbe ancora più chiaro e fruibile un report in HTML, meglio ancora se esso fosse strutturalmente identico ai report prodotti da EMMA. In questo modo, oltre ad essere familiari e più leggibili, potrebbero essere utilizzati come nuovo input del sistema stesso, realizzando correlazioni più complesse utilizzando semplicemente quelle di base sviluppate in questa prima e combinandole in modi diversi. Un esempio potrebbe essere realizzare la differenza tra un test e l'unione di altri due (o più) test o il viceversa, realizzando differenze più complesse di quelle di base introdotte

Gli altri sviluppi possibili sono riassunti per punti:

1. Miglioramento della formattazione dei report testuali, in particolare della lista di linee di codice dettagliata.
2. Sviluppo di un'interfaccia grafica per entrambi i sottosistemi.
3. Inserimento degli input in modalità automatica più flessibile.
4. Possibilità di configurare dall'esterno i parametri di connessione al server MySql, senza mettere mano al codice sorgente.
5. Refactoring del codice per un design architetturale migliore.

Bibliografia

[1] «Statista - The Statistics Portal,» Statista Inc., [Online]. Available: http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/. [Consultato il giorno 06 07 2015].

[2] «Android GUI Ripper Wiki,» [Online]. Available: http://wpage.unina.it/ptramont/GUIRipperWiki.htm. [Consultato il giorno 06 07 2015].

[3] «EMMA: a free Java code coverage tool,» [Online]. Available: http://emma.sourceforge.net/. [Consultato il giorno 06 07 2015].

[4] «HTML, The Web's Core Language,» [Online]. Available: http://www.w3.org/html/. [Consultato il giorno 06 07 2015].

[5] «Java - Essentials of the Java Programming Language,» [Online]. Available: http://www.oracle.com/technetwork/java/index-138747.html. [Consultato il giorno 06 07 2015].

[6] «JUnit Testing Framework,» [Online]. Available: http://www.junit.org/. [Consultato il giorno 06 07 2015].

[7] «Eclipse,» The Eclipse Foundation, [Online]. Available: http://www.eclipse.org/home/. [Consultato il giorno 06 07 2015].

[8] «Teorema della scimmia instancabile,» Wikipedia, [Online]. Available: https://it.wikipedia.org/wiki/Teorema_della_scimmia_instancabile. [Consultato il giorno 06 07 2015].

[9] K. Li e M. Wu, Effective GUI Testing Automation: Developing an Automated GUI Testing Tool, Sybex, 2006.

[10] «CodeCover,» [Online]. Available: http://codecover.org/index.html. [Consultato il giorno 06 07 2015].

[11] «MySQL,» Oracle Corporation, [Online]. Available: https://www.mysql.it/. [Consultato il giorno 06 07 2015].

[12] «CRUD: Create, read, update and delete,» Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Create,_read,_update_and_delete. [Consultato il giorno 06 07 2015].

[13] «Document Object Model (DOM),» [Online]. Available: http://www.w3.org/DOM/. [Consultato il giorno 06 07 2015].

[14] «XPath Tutorial,» [Online]. Available: http://www.w3schools.com/xpath/. [Consultato il giorno 06 07 2015].

[15] «XML DOM - The NodeList Object,» [Online]. Available: http://www.w3schools.com/dom/dom_nodelist.asp. [Consultato il giorno 06 07 2015].

[16] «Factory method pattern,» [Online]. Available: https://en.wikipedia.org/wiki/Factory_method_pattern. [Consultato il giorno 06 07 2015].

[17] «MySql - Schema Object Names,» Oracle Corporation, [Online]. Available: https://dev.mysql.com/doc/refman/5.0/en/identifiers.html. [Consultato il giorno 06 07 2015].

[18] «Java™ Platform, Standard Edition 7,» Oracle, [Online]. Available: http://download.oracle.com/javase/7/docs/api/. [Consultato il giorno 06 07 2015].

[19] «Javadoc Tool,» Oracle, [Online]. Available: http://www.oracle.com/technetwork/articles/java/index-jsp-135444.html. [Consultato il giorno 06 07 2015].

[20] «Ricerca in profondità, depth first search (DFS),» [Online]. Available: https://it.wikipedia.org/wiki/Ricerca_in_profondit%C3%A0. [Consultato il giorno 06 07 2015].

[22] I. Sommerville, Software Engineering, 8th edition a cura di, Addison-Wesley, 2007.

[23] Boehm, «Guidelines for verifying and validating software requirements design specifications,» 1979.

[24] R. S. Pressman, Software Engineering: A pratictioner’s approach, 5th edition a cura di, McGraw Hill, 2001.

[25] C. Gezzi, M. Jazayeri e D. Mandrioli, Ingegneria del software. Fondamenti e principi, seconda edizione a cura di, Pearson Education Italia, 2004.