

# Elaborato di Ingegneria del Software 2



Studente: Felice Francesco Tufano- Mat. M63/982

## Random Testing

# Capitolo 1

## 1.1 Premessa

Il Random Testing è una tecnica di testing del software di tipo black-box, in cui i programmi vengono testati generando input casuali e indipendenti. Si può utilizzare in tutti casi in cui non sappiamo che test fare, oppure quando vogliamo spingerci oltre i test fatti. Ha un punto debole nella sua **efficienza**, in quanto:

- potrebbe ripetere più volte lo stesso test;
- potrebbe eseguire test molto lunghi;
- potrebbe generare test che non hanno senso.
- andare a caso può portare alla elevata difficoltà di raggiungere una condizione finale, ad esempio raggiungere la vittoria a campo minato.

Alcuni tipi di test casuali possono comportare l'uso di euristiche, che guidano l'uso di input casuali. In generale, e in particolare quando si ha a che fare con interi o altri tipi di variabili, il test casuale è casuale solo quanto l'insieme di input casuali utilizzati, in altre parole, i tester spesso scelgono di utilizzare un insieme limite di interi, piuttosto che un insieme infinito.

Ha un punto di forza nella sua **efficacia**, in quanto può trovare un test che un'altra tecnica non trova.

I Test Random non hanno oracolo. Possono essere utilizzati solo per cercare possibili situazioni di crash / eccezioni, o verificare la violazione di proprietà invarianti.

Il random testing viene condotto in maniera totalmente automatica. Una caratteristica del random testing è che esso può essere eseguito in maniera parallela.

Una delle domande più frequenti che ci si pone quando si esegue testing in maniera casuale è la seguente: quando terminare il Random Testing?

Secondo le pratiche dell'Ingegneria del Software la fase di testing dovrebbe terminare:

- al raggiungimento di un limite fissato, inteso sia come tempo speso che di risorse usate;
- al raggiungimento di un determinato valore di efficacia;
- quando dopo l'esecuzione di un determinato numero di test si nota che non si hanno miglioramenti dal punto di vista dell'efficacia raggiunta.

La condizione più soddisfacente per terminare il testing è il raggiungimento della copertura massima raggiungibile (saturazione), ma potremmo non essere in grado di riconoscerla.

La copertura a saturazione è l'insieme di tutte le righe di codice escluse quelle non raggiungibili, che non sono però note a priori.

Di solito si tende a trovare un compromesso tra l'accuratezza delle prove effettuate e il tempo ad esse dedicato a causa della limitatezza delle risorse disponibili; per tale motivo sono stati concepiti alcuni criteri di terminazione per interrompere i test nel minor tempo possibile, ma conservando un livello accettabile delle analisi compiute. Un primo criterio abbastanza semplicistico consiste nel fissare un limite massimo di tempo, di eventi o di chiamate di funzioni entro cui completare il testing casuale: se da un lato risulta la soluzione più diffusa, in quanto molto semplice da implementare, dall'altro non assicura per nulla che i risultati ottenuti siano affidabili, soprattutto se non si copre in maniera adeguata il codice sorgente a causa di fattori troppo stringenti. Di contro, l'uso di un criterio di terminazione basato sull'ottenimento di un determinato livello di copertura potrebbe comportare uno spreco di

tempo indefinito nell'attesa che ciò accada (soprattutto in presenza di strutture di controllo difficilmente percorribili), pertanto viene adottato solo in casi particolari. Un altro criterio abbastanza percorribile nella pratica prevede l'interruzione del test nel momento in cui viene superato un prestabilito numero di sessioni consecutive senza ulteriori miglioramenti nella copertura complessiva del codice. Esistono altri criteri di terminazione basati sul cosiddetto "effetto saturazione", ovvero sulla ridotta capacità di scoperta di nuovi bug col passare del tempo di esecuzione di un test. Si tratta di un fenomeno che influisce sull'incremento della percentuale di copertura del codice da testare, il quale, dopo un determinato numero di eventi eseguiti (generalmente nell'ordine delle migliaia) diminuisce e/o si arresta su un determinato limite, detto punto di saturazione, in cui si satura completamente, poichè a mano a mano che si procede con la scelta casuale degli eventi aumenteranno le probabilità di ripescare e ripetere quelli già verificati, mentre diminuiranno le possibilità di coprire parti di codice ancora inesplorate; per tale ragione, insistere sul proseguimento dei test oltre il punto di saturazione non risulta affatto conveniente (specie se è stata adottata la copertura come metrica di riferimento), dato che non si otterrebbe alcun progresso sulla percentuale di copertura (se non dopo innumerevoli tentativi a vuoto e senza alcuna certezza che la copertura aumenti ulteriormente).

I criteri di terminazione considerati in questo elaborato sono i seguenti:

- **criterio temporale:** dopo aver dedicato una certa quantità di tempo ad ogni classe testata, per ogni sessione di test;
- **Criterio di copertura T60%:** quando viene raggiunto il 60% di copertura del codice;
- **Criterio di reliability T50%:** quando si osserva che l'ultimo 50% di tempo di test non ha prodotto alcun miglioramento nella copertura raggiunta (cumulativamente da tutte le sessioni di test);
- **AIO (All-Include-One):** quando si osserva che c'è una sessione la cui copertura include tutte le altre (di conseguenza, tutte le altre sessioni non forniscono alcun contributo differente alla copertura);
- **AEQ (All-Equivalent):** quando si osserva che tutte le sessioni di test hanno raggiunto lo stesso insieme di copertura (di conseguenza corrispondente alla copertura cumulativa).

## 1.2 Randoop

Randoop è un generatore di test di unità per Java. Crea automaticamente unit test per le classi, in formato JUnit. Il suo funzionamento è basato su una tecnica di testing casuale a retroazione (feedback directed random testing): a partire dalle classi del programma da testare, esso crea delle sequenze casuali che successivamente sottopone al programma stesso, costruendosi di volta in volta delle asserzioni sul suo comportamento. In questo modo è possibile sia evidenziare eventuali falle e/o eccezioni dell'applicativo sotto esame, sia ottenere dei test di regressione per evitare l'immissione di errori nelle versioni revisionate. Quindi questa tecnica genera in modo pseudo-casuale, ma intelligente, sequenze di chiamate a metodo/costruttore per le classi in fase di test. Randoop esegue le sequenze create, utilizzando i risultati dell'esecuzione per creare asserzioni che acquisiscono il comportamento del programma.

Esso genera due tipi di test:

- **ErrorTest:** sono test che trovano errori o bug nel codice;

- **RegressionTest:** sono test di regressione per eventuali modifiche future del codice.

Quando Randoop include un test che rivela errori nel file , il test mostra che il codice viola la specifica o il contratto. Attualmente, Randoop verifica i seguenti contratti: Object.equals(), Object.hashCode(), Object.clone(), Object.toString(), Comparable.compareTo() , Comparable.compare().

La combinazione di Randoop di generazione di test ed esecuzione dei test si traduce in una tecnica di generazione di test altamente efficace. Randoop ha rivelato errori precedentemente sconosciuti anche in librerie ampiamente utilizzate, tra cui i JDK di Sun e IBM e un componente .NET di base.

Tra i vari parametri facoltativi che lo strumento permette di specificare notiamo un “*random seed*” (cioè un valore intero con cui estrarre la sequenza casuale di eventi; se non specificato, sarà pari a 0), e un limite temporale (denominato time limit) entro cui concludere il test sulla singola sequenza (di default esso è fissato a 60 secondi).

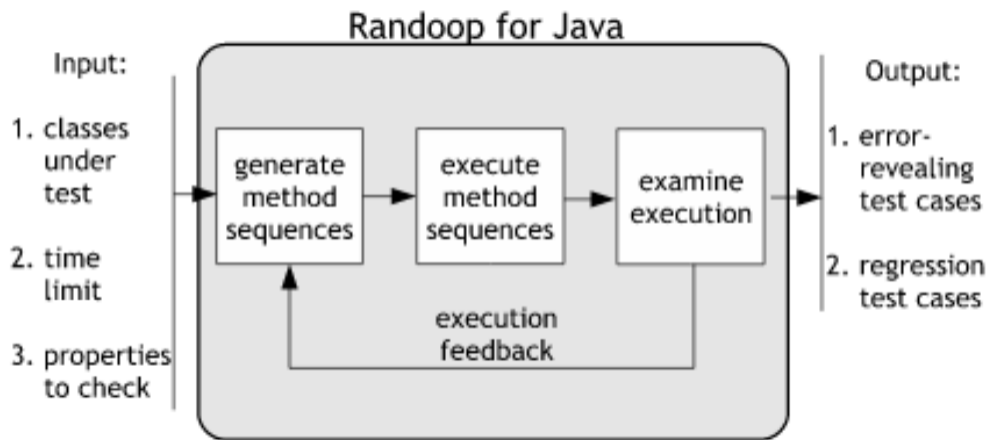


Figura 1 Funzionamento di Randoop

## 1.3 Emma

EMMA è uno strumento impiegato per le analisi di copertura del codice sorgente di applicazioni Java. Esso consente di effettuare le analisi sul programma da testare sia in modalità “offline” (ovvero scindendo l’strumentazione dell’applicazione sotto esame e la sua esecuzione), sia in modalità “on the fly” (cioè verificando gli eventi a tempo di esecuzione). La prima soluzione è quella adoperata in questo elaborato, in particolare EMMA si attiverà nel momento in cui verrà lanciato Randoop. Attraverso EMMA è possibile esplorare il codice sorgente secondo varie tipologie: per classi, per metodi, per linee e per blocchi. Inoltre, grazie alla funzione di “merge” possiamo aggregare facilmente i risultati di sessioni differenti appartenenti ad un medesimo ciclo, ottenendo quindi la loro unione, che fungerà da termine di paragone per la verifica dei criteri di terminazione.

## 1.4 Obiettivo

L'obiettivo dell'elaborato è creare un'applicazione scritta in linguaggio Java che sia in grado di condurre diverse sessioni di test in parallelo, riportando in tempo reale l'andamento al crescere dei cicli delle linee di codice coperte, e al tempo stesso informazioni sul numero di test eseguiti con relativo report degli errori ottenuti. Per ottenere la copertura del codice è stato utilizzato lo strumento **EMMA**. Verrà realizzato quindi un programma che una volta avviato, eseguirà quanto stabilito, servendosi di una serie di script (successivamente descritti nel dettaglio) che verranno lanciati dal programma in modo totalmente automatico. Quindi gli output prodotti serviranno a fornire due tipi di informazioni:

- Informazioni sull'andamento della copertura raggiunta dalle varie sessioni;
- Informazioni sull'andamento dei test eseguiti dal punto di vista del numero di test falliti, e dal punto di vista degli errori trovati.

L'idea è quella di fornire sia per quanto riguarda la coverage che per quanto riguarda i test eseguiti da un lato una visione quantitativa dei risultati ottenuti mediante degli opportuni grafici, e dall'altro una visione qualitativa mediante la produzione di vari file di output.

## 1.5 File utilizzati

Per l'esecuzione del programma verranno utilizzati i seguenti file:

- **emma-2.0.5312**: cartella all'interno della quale si trovano tutti i file necessari al funzionamento di Emma;
- **external\_jars**: cartella all'interno della quale si trovano i file jar necessari alla creazione dei grafici e all'esecuzione dei test. In particolare: i file jar "jcommon-1.0.23" e "jfreechart-1.0.19" sono quelli necessari all'applicativo per realizzare sia il grafico relativo alla copertura del codice che quello relativo ai test eseguiti, e che quindi dovranno essere inclusi all'interno del classpath. Invece i file "junit-4.13.2" e "hamcrest-core-1.3" sono quelli necessari all'esecuzione dei test JUnit. Questi ultimi due verranno copiati automaticamente dall'applicazione all'interno delle apposite cartelle in cui verranno creati ed eseguiti i test JUnit. La descrizione dettagliata su queste cartelle, e sul modo in cui verranno create dall'applicazione, verrà fornita nel paragrafo 2.1;
- **jars**: cartella in cui si trovano le applicazioni da testare. Per ogni applicazione, all'interno di questa cartella c'è anche un file di testo contenente l'elenco delle classi che devono essere prese in considerazione per la valutazione della copertura. La sottocartella lib include le librerie da cui dipendono le applicazioni testate;
- **Java-1.8**: all'interno di questa cartella è presente la versione di java 1.8. Questa versione verrà utilizzata esclusivamente per Emma, in quanto versioni successive a questa non sono supportate adeguatamente da questo strumento;
- **randoop-all-3.0.6.jar**: file jar necessario all'esecuzione di Randoop.

# Capitolo 2

## 2.1 Struttura dell'applicazione e script utilizzati

Il programma è composto da 6 classi. In particolare il punto di partenza è la classe “**Run**”, una volta che quest'ultima viene lanciata l'esecuzione del programma procede in totale autonomia. La descrizione di ciascuna classe è la seguente:

- **Run.** Come già detto questa è il punto di partenza del programma. Questa classe ha il compito di istanziare i vari thread rappresentativi delle varie sessioni di test, e gli oggetti necessari alla creazione dei grafici. All'avvio verrà chiesto di inserire i seguenti parametri: numero di sessioni di test in parallelo, nome dell'applicazione testata e il time limit per la generazione dei test (cioè la quantità di tempo da dedicare ad ogni classe dell'applicazione testata per la generazione dei test);
- **Produttore.** Un oggetto di questa classe rappresenta una sessione di testing. Ogni oggetto produce i propri risultati in modo indipendente. A tale scopo ognuno lancia gli script “starter6Randoop” e “esecuzione\_test” (che analizzeremo in seguito), e in base ai risultati prodotti da questi aggiorna di volta in volta sia il grafico della copertura che quello del numero di test eseguiti. In particolare lo script “starter6Randoop”, per ogni oggetto produttore, verrà lanciato un'unica volta all'inizio della propria esecuzione, mentre lo script “esecuzione\_test” verrà lanciato, per ogni oggetto produttore, una volta per ogni ciclo di test (e solo se in quel ciclo sono stati prodotti dei nuovi test da eseguire). Ogni thread ha anche il compito di verificare il raggiungimento, da parte della sessione di test che esso rappresenta, dei criteri di copertura T50%, T60% e AIO;
- **Consumatore.** L'oggetto di questa classe è colui che ha il compito di aggiornare i risultati relativi all'unione delle sessioni, per quanto riguarda la copertura del codice. In particolare esso lancia lo script “createCoverageUnion” (anche questo analizzato nel dettaglio successivamente), elabora i file che vengono prodotti, e aggiorna i relativi risultati sul grafico della copertura del codice. Compito di questo oggetto è anche quello di verificare che sia soddisfatto il criterio “AEQ” da parte delle varie sessioni di test;
- **ClasseCondivisa.** Questa classe contiene i metodi “synchronized” necessari alla valutazione in diretta dell'andamento della copertura da parte delle varie sessioni. Infatti per una valutazione più agevole e riassuntiva delle varie sessioni dal punto di vista della copertura del codice, ogni thread “produttore” scrive in maniera concorrente i risultati di copertura raggiunti all'interno di un file “LetturaCoverage.txt”, che verrà letto di volta in volta dal thread consumatore. Quindi i valori di coverage raggiunti di volta in volta dai vari thread, e che sono stati letti successivamente dal consumatore, verranno mostrati anche a video oltre che sul grafico;
- **DrawLOC.** Un'istanza di questa classe rappresenta un oggetto grafico visivo raffigurante l'andamento delle “n+1” sessioni di test (n sessioni più l'unione) dal punto di vista delle linee di codice coperte, che verrà mostrato all'utente;
- **DrawErrorTest.** Un'istanza di questa classe rappresenta un oggetto grafico visivo raffigurante l'andamento “Numero di test Eseguiti - Numero di test falliti (in totale nel corso dei vari cicli)” da parte delle n sessioni di test, che verrà mostrato all'utente.

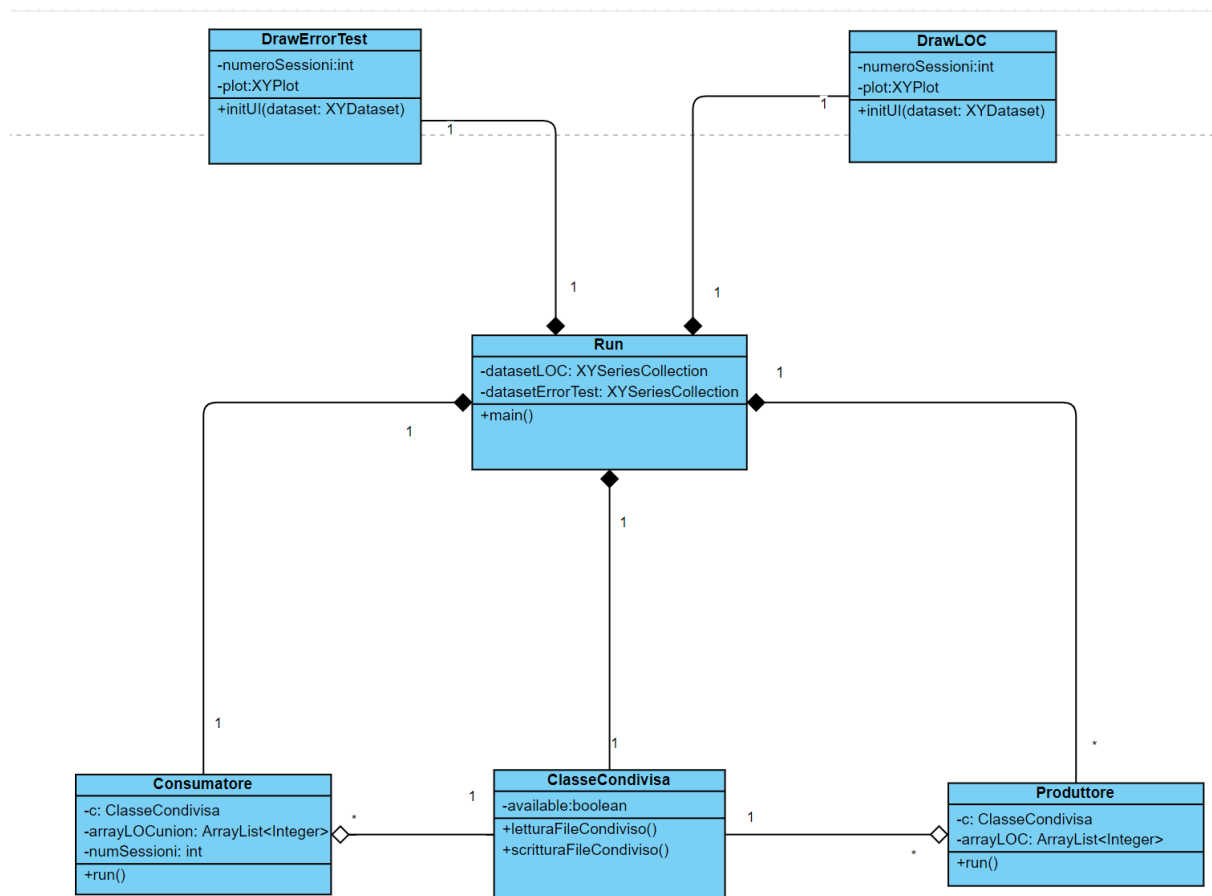


Figura 2 Class Diagramm dell'applicazione

Come anticipato in precedenza, sono stati realizzati degli script in linguaggio batch, che verranno usati dall'applicazione per produrre i risultati. Questi ultimi non dovranno essere eseguiti a mano, ma verranno lanciati dagli oggetti delle classi precedentemente descritte in modo automatico:

1. **starter6Randoop.bat**: per ogni sessione di test che lo lancia (si ricorda che ogni sessione di test è rappresentata da un oggetto della classe **Produttore**), crea in modo iterativo dei cicli di test lanciando Randoop. Per ogni ciclo vengono prodotti, oltre agli eventuali **ErrorTest**, tutti i file relativi alla copertura della sessione al crescere dei cicli, e i file relativi all'unione delle coverage dei cicli della sessione. Per quanto riguarda i test generati, questo script creerà una cartella che conterrà i test prodotti dalla sessione che lo ha lanciato e copierà all'interno di questa cartella i file necessari all'esecuzione dei test per l'applicazione che si sta testando, ossia lo script "esecuzione\_test.bat", i file "junit-4.13.2" e "hamcrest-core-1.3" della cartella `external_jars`, e le librerie da cui l'applicazione testata dipende. Per queste ultime bisogna assicurarsi che all'interno della folder generale ci sia una cartella avente lo stesso nome dell'applicazione testata, all'interno della quale dovranno trovarsi tutti i file e librerie necessarie per lanciare l'applicazione sotto test, in quanto lo script **starter6Randoop** cercherà le librerie da cui l'applicazione testata dipende all'interno della cartella suddetta. Per una maggiore chiarezza verrà mostrato nel paragrafo 2.3 un esempio di utilizzo.

Tra gli output che questo script produce, notiamo il file "SessioncoverageLOC.txt" che contiene le percentuali di LOC coperte dall'unione dei cicli della sessione al crescere dei cicli.

Quest'ultimo file è quello che verrà utilizzato per ottenere una lettura in tempo reale dell'andamento.

Questo script prende in ingresso i seguenti parametri: nome dell'applicazione da testare, quantità di tempo da dedicare ad ogni classe per la generazione dei test (ossia il time limit usato da Randoop per generare test) e l'id del thread rappresentativo della sessione (che verrà utilizzato per distinguere le varie sessioni tra di loro). Questi parametri verranno passati allo script automaticamente dall'oggetto della classe Produttore, che lo lancerà un'unica volta all'inizio della sua esecuzione;

2. ***esecuzione\_test.bat***: utilizzato per l'esecuzione dei test JUnit, una volta lanciato crea un file txt per ogni esecuzione riportandone l'esito. In particolare ogni thread produttore controlla ad ogni ciclo se è stato prodotto il relativo file "ErrorTest.java", in termini pratici ciò equivale ad effettuare un controllo sulla data di modifica di questo file (questo perché ad ogni nuova creazione Randoop sovrascriverà il file ErrorTest.java), se la data è cambiata vuol dire che al ciclo corrente sono stati prodotti dei nuovi test e si può procedere con la loro esecuzione. Anche questo script come il precedente, verrà lanciato in modo automatico dal thread produttore. In particolare esso verrà lanciato dai vari thread produttori una volta per ogni ciclo, solo se in quel ciclo sono stati prodotti dei nuovi test da eseguire;
3. ***createCoverageUnion.bat***: calcola l'unione delle coperture raggiunte dalle varie sessioni al crescere dei cicli, attendendo ad ogni ciclo la creazione degli n file "coverageUnion.es", da parte delle n sessioni, all'interno della cartella "coverageunion" (la quale verrà creata automaticamente all'inizio dell'esecuzione del programma grazie allo script starter6Randoop), e ne effettua il merge di volta in volta. Come già anticipato questo script verrà lanciato un'unica volta, all'avvio dell'esecuzione, dall'oggetto della classe Consumatore.

## 2.2 Output prodotti

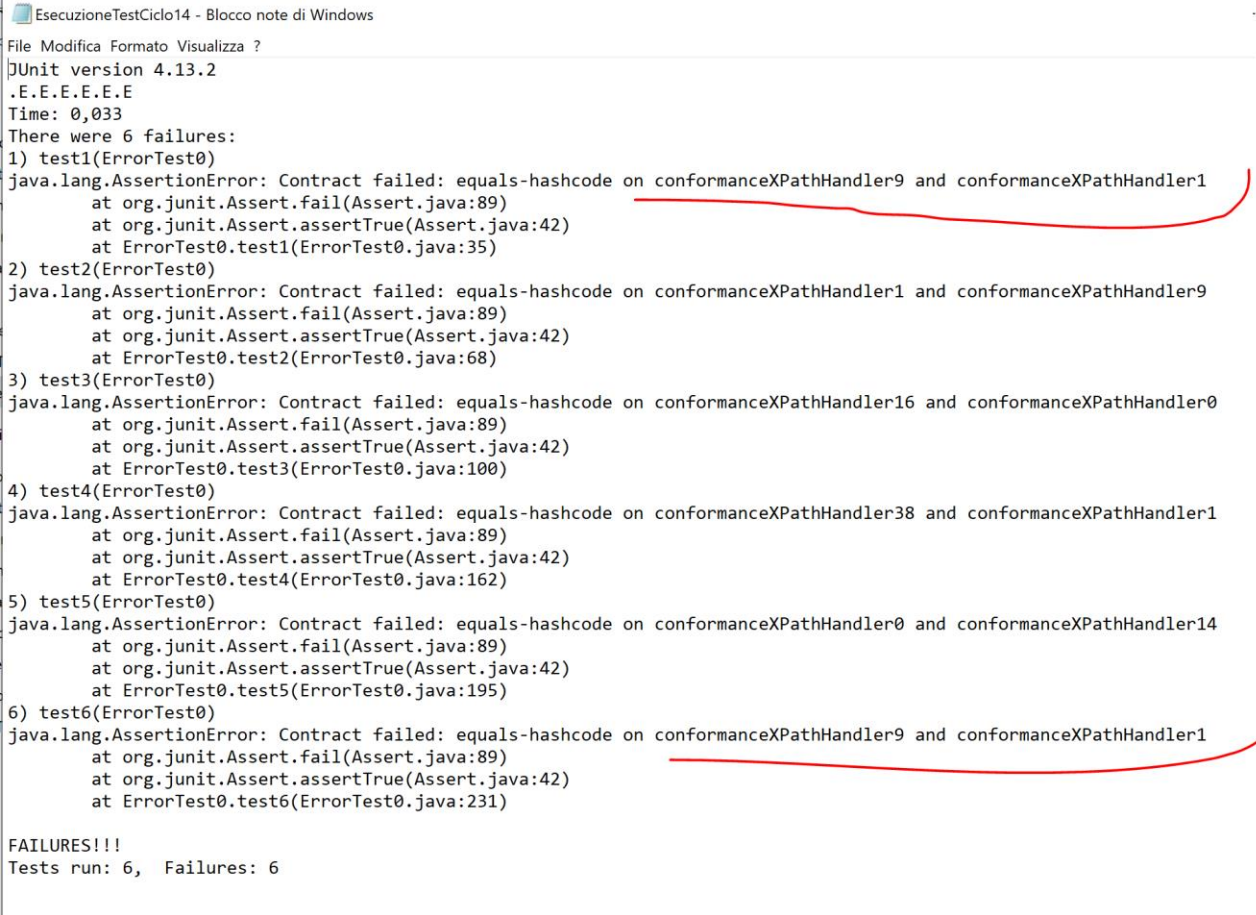
Quindi il programma permette l'esecuzione di n sessioni di testing in parallelo. Per quanto riguarda gli output prodotti:

- dal punto di vista quantitativo verranno generati **due grafici**: il primo rappresentante le linee di codice coperte dalle varie sessioni al crescere dei cicli, il secondo rappresentate il numero di test eseguiti da ogni sessione con relativo numero di test falliti (in totale nel corso dei vari cicli). In particolare per il primo grafico verrà notificato anche il raggiungimento dei criteri di copertura **T50%, T60%, AIO e AEQ**. Volendo conservare una sorta di "storico" dei criteri di terminazione raggiunti, mantenendo al contempo anche un buon grado di leggibilità del grafico, sono state fatte le seguenti scelte implementative: i criteri di copertura T50% e T60% per ogni sessione vengono segnalati una volta che sono stati raggiunti, il criterio AIO viene segnalato ogni qualvolta si verifica che una sessione lo raggiunge, mentre il criterio AEQ viene segnalato ogni volta che viene raggiunto dalle varie sessioni, ma un'unica volta per uno stesso valore di coverage. Quindi se ad esempio le sessioni soddisfano il criterio AEQ per tre cicli di test consecutivi, e se il valore di coverage raggiunto durante questi cicli non cambia, il raggiungimento verrà segnalato sul grafico solo al primo di questi cicli in cui il criterio è stato



soddisfatto. Eventualmente il raggiungimento del criterio AEQ verrà segnalato nuovamente sul grafico se le sessioni lo dovessero soddisfare per un nuovo valore di coverage.

- Per fornire una visione più chiara della copertura raggiunta dai vari thread nel corso dell'esecuzione, questi valori di copertura verranno riportati di volta in volta, come già detto nel paragrafo precedente, nel file "LetturaCoverage.txt" e **mostrati anche a video**.
- Per quanto riguarda gli ErrorTest prodotti da Randoop, abbiamo visto come si è fatto in modo che ogni sessione crei **una cartella contenente i propri test**, e i file necessari alla loro esecuzione. Ogni volta che un nuovo test viene eseguito verranno aggiornate le informazioni relative al numero totale di test eseguiti, al numero totale di test falliti, e conseguentemente verrà aggiornato anche il grafico relativo all'esecuzione dei test. Di ogni esecuzione dei test JUnit ad un determinato ciclo verrà prodotto il relativo file **"esecuzioneTestCicloN.txt"** contenente il rispettivo risultato, inoltre per ogni sessione verrà anche generato un file **"reportErrori.txt"** che conterrà al suo interno l'elenco di tutti gli errori trovati da quella sessione, e per ognuno degli errori trovati anche il numero di volte che quell'errore è apparso nel corso delle esecuzioni. Per stabilire se due errori sono uguali si procede nel seguente modo: viene controllata la prima riga della descrizione che si ottiene a seguito del fallimento di un test, se due test che falliscono hanno la prima riga della descrizione uguale, allora i due errori sono uguali. Quindi abbiamo gli **ErrorTest**, ossia quei test prodotti da Randoop che trovano errori o bug nel codice; e gli **errori**, ossia la causa del fallimento di un determinato test case.



```
File Modifica Formato Visualizza ?
JUnit version 4.13.2
.E.E.E.E.E.E
Time: 0,033
There were 6 failures:
1) test1(ErrorTest0)
java.lang.AssertionError: Contract failed: equals-hashcode on conformanceXPathHandler9 and conformanceXPathHandler1
    at org.junit.Assert.fail(Assert.java:89)
    at org.junit.Assert.assertTrue(Assert.java:42)
    at ErrorTest0.test1(ErrorTest0.java:35)
2) test2(ErrorTest0)
java.lang.AssertionError: Contract failed: equals-hashcode on conformanceXPathHandler1 and conformanceXPathHandler9
    at org.junit.Assert.fail(Assert.java:89)
    at org.junit.Assert.assertTrue(Assert.java:42)
    at ErrorTest0.test2(ErrorTest0.java:68)
3) test3(ErrorTest0)
java.lang.AssertionError: Contract failed: equals-hashcode on conformanceXPathHandler16 and conformanceXPathHandler0
    at org.junit.Assert.fail(Assert.java:89)
    at org.junit.Assert.assertTrue(Assert.java:42)
    at ErrorTest0.test3(ErrorTest0.java:100)
4) test4(ErrorTest0)
java.lang.AssertionError: Contract failed: equals-hashcode on conformanceXPathHandler38 and conformanceXPathHandler1
    at org.junit.Assert.fail(Assert.java:89)
    at org.junit.Assert.assertTrue(Assert.java:42)
    at ErrorTest0.test4(ErrorTest0.java:162)
5) test5(ErrorTest0)
java.lang.AssertionError: Contract failed: equals-hashcode on conformanceXPathHandler0 and conformanceXPathHandler14
    at org.junit.Assert.fail(Assert.java:89)
    at org.junit.Assert.assertTrue(Assert.java:42)
    at ErrorTest0.test5(ErrorTest0.java:195)
6) test6(ErrorTest0)
java.lang.AssertionError: Contract failed: equals-hashcode on conformanceXPathHandler9 and conformanceXPathHandler1
    at org.junit.Assert.fail(Assert.java:89)
    at org.junit.Assert.assertTrue(Assert.java:42)
    at ErrorTest0.test6(ErrorTest0.java:231)

FAILURES!!!
Tests run: 6, Failures: 6
```

Figura 2 Esempio di due errori uguali

## 2.3 Esempio di esecuzione

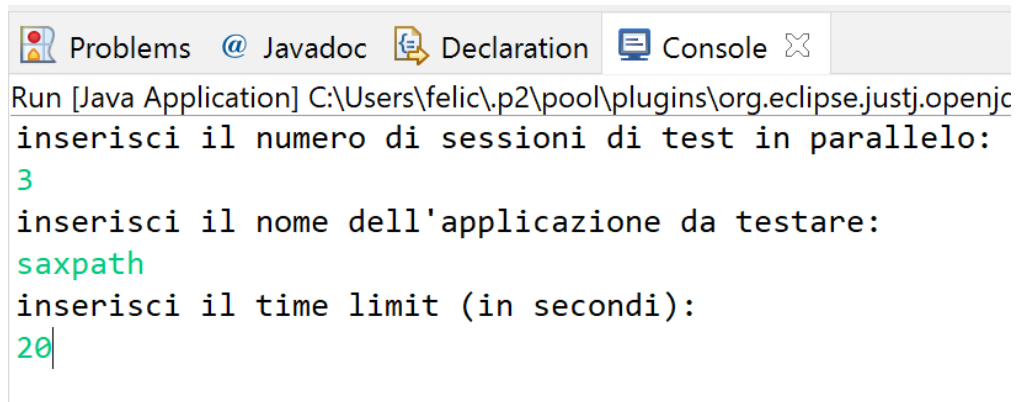
Prima di poter eseguire l'applicazione bisogna assicurarsi che ci sia, tra i vari file, una cartella con lo stesso nome dell'applicazione che si intende testare, all'interno della quale devono essere presenti tutti i file da cui l'applicazione testata dipende. Questi file possono essere ricavati dalla cartella "jars". Per gli esempi riportati è stata effettuata questa operazione per le applicazioni "saxpath e jipa".

Nome	Ultima modifica	Tipo
.settings	23/03/2021 20:34	Cartella di file
bin	23/03/2021 20:37	Cartella di file
emma-2.0.5312	23/03/2021 20:34	Cartella di file
external_jars	08/04/2021 18:26	Cartella di file
jars	27/03/2021 19:55	Cartella di file
java-1.8	07/04/2021 18:56	Cartella di file
jipa	07/04/2021 18:14	Cartella di file
saxpath	23/03/2021 20:35	Cartella di file
src	23/03/2021 20:35	Cartella di file
.classpath	23/03/2021 20:37	File CLASSPATH
.project	23/02/2021 17:16	File PROJECT
create_coverageUnion	07/04/2021 21:06	File batch Windows
DOCUMENTAZIONE	02/04/2021 12:06	Documento di testo
esecuzione_test	10/03/2021 19:08	File batch Windows
myfilters	31/10/2016 14:06	Documento di testo
randoop-all-3.0.6	14/10/2016 12:48	Executable Jar File
run	08/04/2021 11:05	Executable Jar File
starter6Randoop	07/04/2021 21:05	File batch Windows

All'interno della cartella "src" è presente il package runTests all'interno del quale si trovano i file .java dell'applicazione, mentre nel package runTests della cartella "bin" sono presenti i file .class.

Per eseguire l'applicazione ci sono tre possibilità:

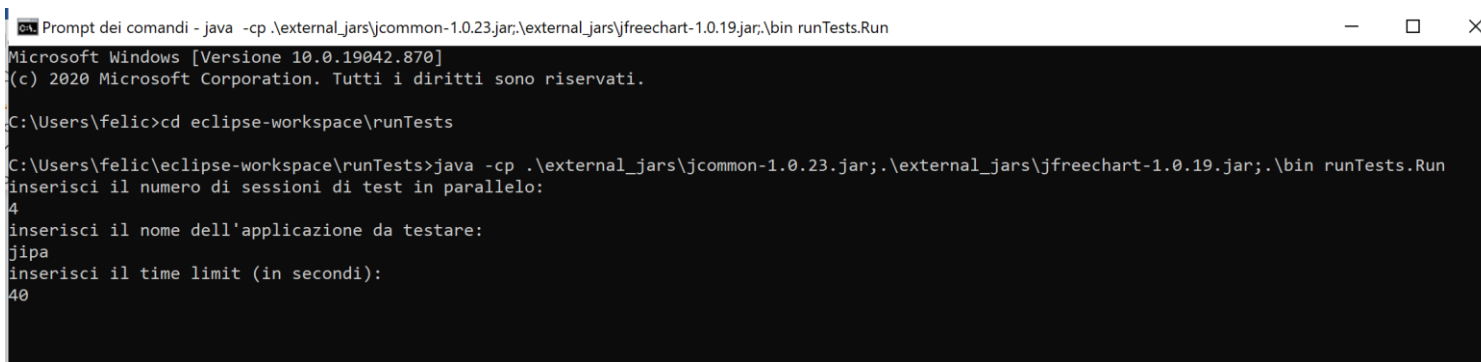
1. Importare l'intera cartella come progetto in Eclipse, aggiungendo i file jar "jcommon-1.0.23" e "jfreechart-1.0.19" che si trovano nella cartella "external\_jars" all'interno del Build Path di progetto. Dopodichè si può eseguire il tutto:



```
Run [Java Application] C:\Users\felic\p2\pool\plugins\org.eclipse.justj.openjc
inserisci il numero di sessioni di test in parallelo:
3
inserisci il nome dell'applicazione da testare:
saxpath
inserisci il time limit (in secondi):
20
```

2. Lanciare l'applicazione da linea di comando, dopo essersi posizionati all'interno della cartella generale che contiene tutti i file, mediante il comando:

`java -cp .\external_jars\jcommon-1.0.23.jar;.\external_jars\jfreechart-1.0.19.jar;.\bin runTests.Run`

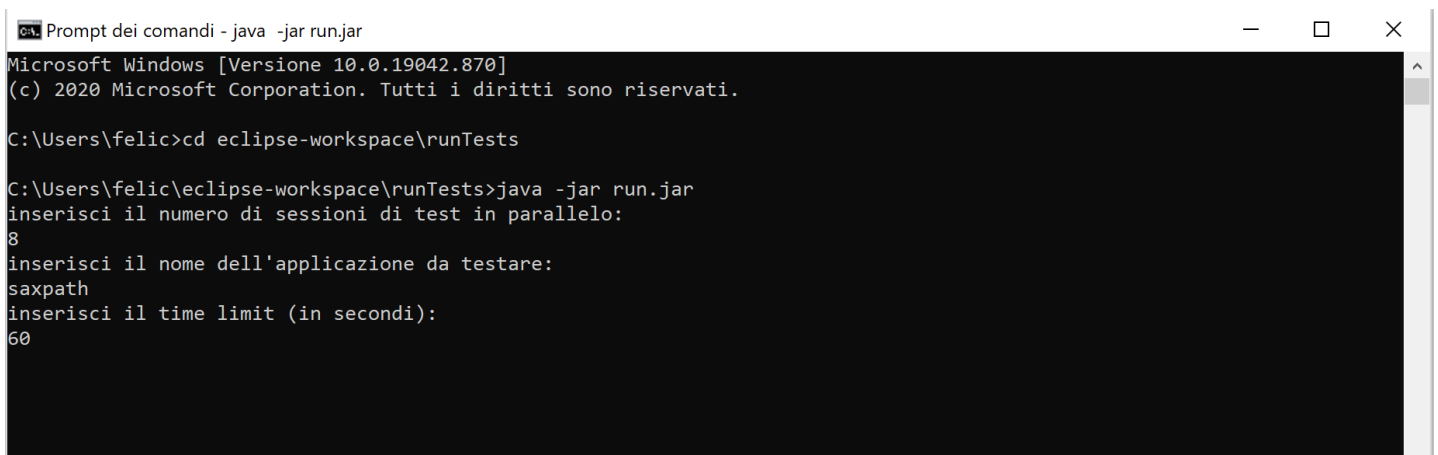


```
Prompt dei comandi - java -cp .\external_jars\jcommon-1.0.23.jar;.\external_jars\jfreechart-1.0.19.jar;.\bin runTests.Run
Microsoft Windows [Versione 10.0.19042.870]
(c) 2020 Microsoft Corporation. Tutti i diritti sono riservati.

C:\Users\felic>cd eclipse-workspace\runTests

C:\Users\felic\eclipse-workspace\runTests>java -cp .\external_jars\jcommon-1.0.23.jar;.\external_jars\jfreechart-1.0.19.jar;.\bin runTests.Run
inserisci il numero di sessioni di test in parallelo:
4
inserisci il nome dell'applicazione da testare:
jipa
inserisci il time limit (in secondi):
40
```

3. Lanciare l'applicazione mediante il file ***“run.jar”***, dopo essersi posizionati all'interno della cartella generale, mediante il comando: `java -jar run.jar`



```
Prompt dei comandi - java -jar run.jar
Microsoft Windows [Versione 10.0.19042.870]
(c) 2020 Microsoft Corporation. Tutti i diritti sono riservati.

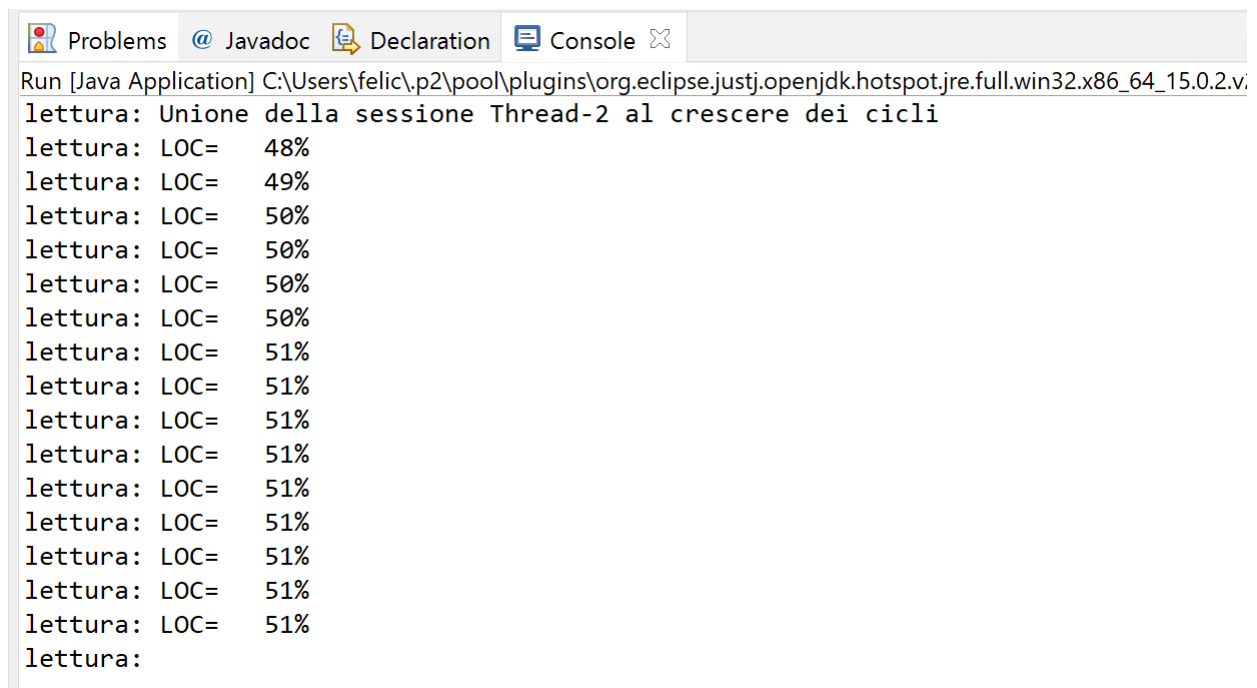
C:\Users\felic>cd eclipse-workspace\runTests

C:\Users\felic\eclipse-workspace\runTests>java -jar run.jar
inserisci il numero di sessioni di test in parallelo:
8
inserisci il nome dell'applicazione da testare:
saxpath
inserisci il time limit (in secondi):
60
```

Una volta lanciato il programma verrà chiesto di inserire il numero di sessioni di test, il nome dell'applicazione da testare e il time limit per la generazione dei test.

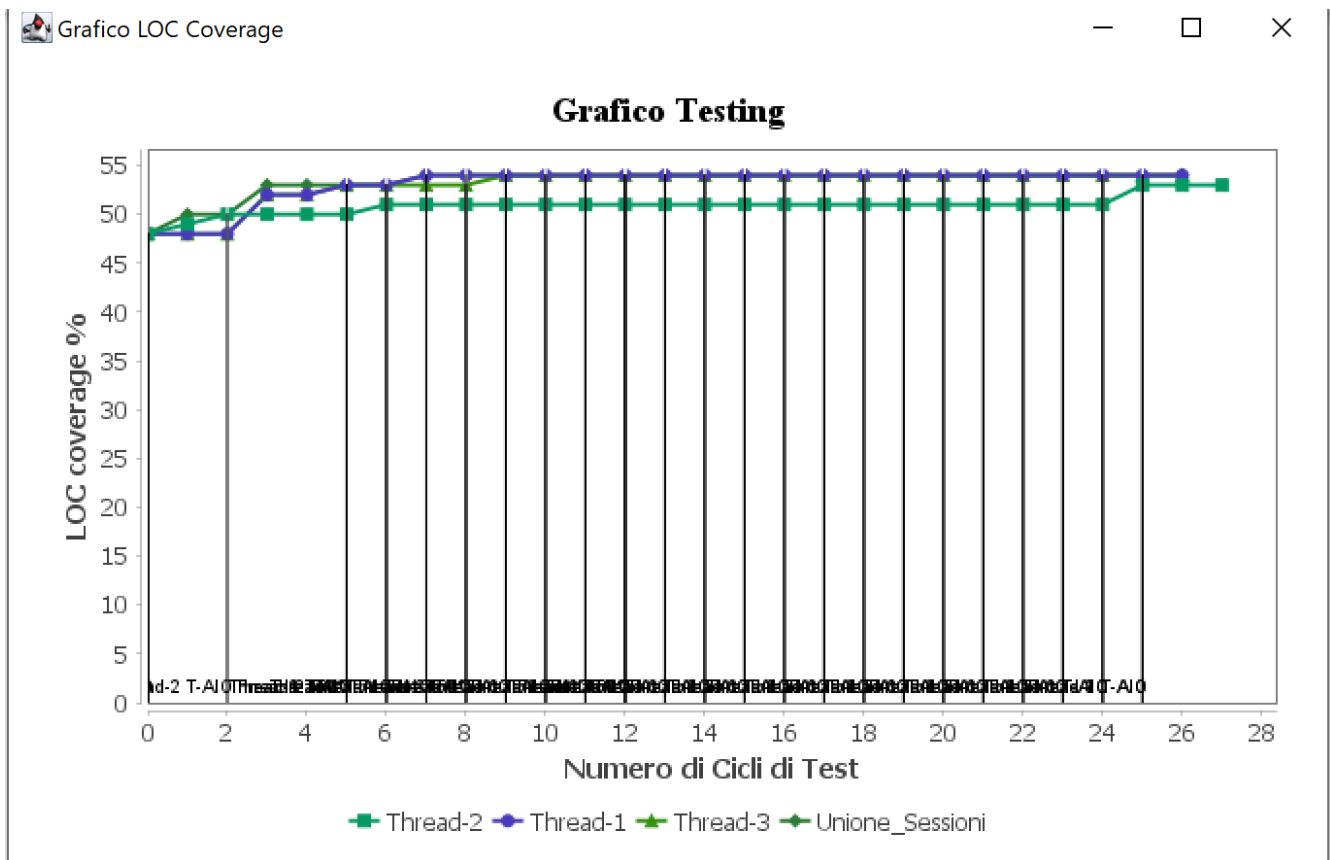
Una volta avviata l'applicazione vengono prodotte per prima cosa le cartelle che conterranno i dati di copertura delle sessioni, e le cartelle contenenti i test prodotti con i relativi file necessari alla loro esecuzione. Successivamente verrà creata la cartella "coverageunion" usata dallo script *createCoverageUnion*, e successivamente verranno creati i file SessionCoverageLOC degli n thread più quello relativo all'unione.

Il file "letturaCoverage" servirà per una lettura da console in diretta degli andamenti delle varie sessioni, di seguito ne è riportato un esempio:

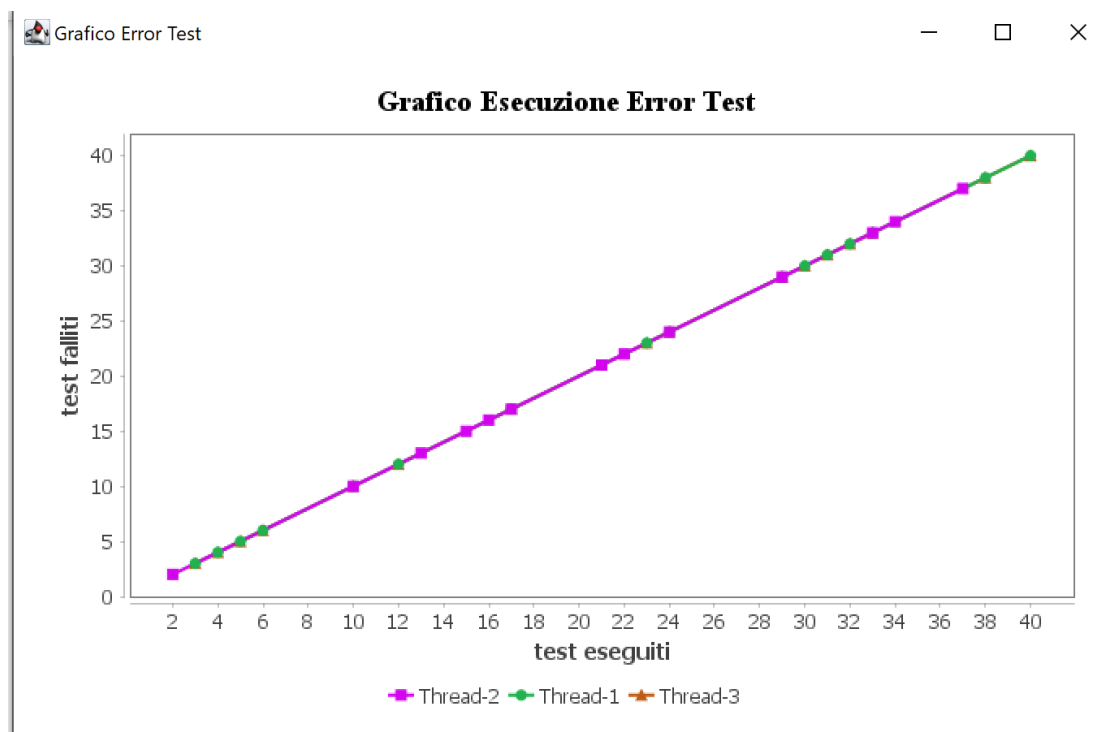


```
Run [Java Application] C:\Users\felic\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_15.0.2.v...
lettura: Unione della sessione Thread-2 al crescere dei cicli
lettura: LOC= 48%
lettura: LOC= 49%
lettura: LOC= 50%
lettura: LOC= 50%
lettura: LOC= 50%
lettura: LOC= 50%
lettura: LOC= 51%
lettura: LOC= 51%
lettura: LOC= 51%
lettura: LOC= 51%
lettura: LOC= 51%
lettura: LOC= 51%
lettura: LOC= 51%
lettura: LOC= 51%
lettura: LOC= 51%
lettura: LOC= 51%
lettura:
```

Di seguito verrà riportato un esempio di esecuzione del programma che lancia tre sessioni di test in parallelo. Notiamo il grafico relativo alla copertura del codice:



























Mentre per i test prodotti otteniamo un grafico che ha sull'asse delle ascisse il numero totale di test eseguiti nel corso del tempo, mentre sulle ordinate il numero totale di quelli falliti:

























Da notare che in questo caso, per l'applicazione testata, il numero dei test falliti è uguale a quello dei test eseguiti. Questa condizione si verifica per tutte e tre le sessioni. In particolare facendo proseguire il test fino al trentesimo ciclo, si ha che il thread-1 ha eseguito 40 test, il thread-2 ne ha eseguiti 37 mentre il thread-3 ne ha eseguiti 44. Come detto prima tutti i test prodotti dalle sessioni conducono ad un fallimento, motivo per cui come risultato abbiamo tre rette, di lunghezza diversa, sovrapposte. Volendo fare un'analisi qualitativa degli errori ottenuti dalle varie sessioni, possiamo consultare i report situati all'interno delle rispettive cartelle. Si noti come, a causa della natura casuale del test, ci saranno dei cicli in cui un thread non avrà prodotto errori, mentre qualcuno degli altri si:

























#### ErrorTest\_Thread-1:

 com	06/04/2021 21:20	Cartella di file	
 org	06/04/2021 21:20	Cartella di file	
 ErrorTest.class	06/04/2021 21:29	File CLASS	1 KB
 ErrorTest	06/04/2021 21:29	File JAVA	1 KB
 ErrorTest0.class	06/04/2021 21:29	File CLASS	4 KB
 ErrorTest0	06/04/2021 21:29	File JAVA	7 KB
 esecuzione_test	10/03/2021 19:08	File batch Windows	1 KB
 EsecuzioneTestCiclo5	06/04/2021 21:20	Documento di testo	1 KB
 EsecuzioneTestCiclo6	06/04/2021 21:21	Documento di testo	1 KB
 EsecuzioneTestCiclo7	06/04/2021 21:21	Documento di testo	1 KB
 EsecuzioneTestCiclo8	06/04/2021 21:22	Documento di testo	1 KB
 EsecuzioneTestCiclo9	06/04/2021 21:22	Documento di testo	2 KB
 EsecuzioneTestCiclo12	06/04/2021 21:23	Documento di testo	1 KB
 EsecuzioneTestCiclo13	06/04/2021 21:24	Documento di testo	2 KB
 EsecuzioneTestCiclo14	06/04/2021 21:24	Documento di testo	2 KB
 EsecuzioneTestCiclo15	06/04/2021 21:24	Documento di testo	1 KB
 EsecuzioneTestCiclo16	06/04/2021 21:25	Documento di testo	2 KB
 EsecuzioneTestCiclo19	06/04/2021 21:26	Documento di testo	1 KB
 EsecuzioneTestCiclo21	06/04/2021 21:27	Documento di testo	1 KB
 EsecuzioneTestCiclo22	06/04/2021 21:27	Documento di testo	1 KB
 EsecuzioneTestCiclo23	06/04/2021 21:28	Documento di testo	1 KB
 EsecuzioneTestCiclo24	06/04/2021 21:28	Documento di testo	2 KB
 EsecuzioneTestCiclo27	06/04/2021 21:29	Documento di testo	1 KB
 reportErrori	06/04/2021 21:30	Documento di testo	5 KB

## ErrorTest\_Thread-2:

 com	06/04/2021 21:19	Cartella di file	
 org	06/04/2021 21:19	Cartella di file	
 ErrorTest.class	06/04/2021 21:30	File CLASS	1 KB
 ErrorTest	06/04/2021 21:30	File JAVA	1 KB
 ErrorTest0.class	06/04/2021 21:30	File CLASS	6 KB
 ErrorTest0	06/04/2021 21:30	File JAVA	13 KB
 esecuzione_test	10/03/2021 19:08	File batch Windows	1 KB
 EsecuzioneTestCiclo2	06/04/2021 21:19	Documento di testo	1 KB
 EsecuzioneTestCiclo3	06/04/2021 21:20	Documento di testo	3 KB
 EsecuzioneTestCiclo5	06/04/2021 21:20	Documento di testo	1 KB
 EsecuzioneTestCiclo10	06/04/2021 21:22	Documento di testo	1 KB
 EsecuzioneTestCiclo11	06/04/2021 21:23	Documento di testo	1 KB
 EsecuzioneTestCiclo15	06/04/2021 21:24	Documento di testo	1 KB
 EsecuzioneTestCiclo17	06/04/2021 21:25	Documento di testo	2 KB
 EsecuzioneTestCiclo18	06/04/2021 21:26	Documento di testo	1 KB
 EsecuzioneTestCiclo20	06/04/2021 21:26	Documento di testo	1 KB
 EsecuzioneTestCiclo21	06/04/2021 21:27	Documento di testo	2 KB
 EsecuzioneTestCiclo23	06/04/2021 21:27	Documento di testo	2 KB
 EsecuzioneTestCiclo25	06/04/2021 21:28	Documento di testo	1 KB
 EsecuzioneTestCiclo28	06/04/2021 21:29	Documento di testo	1 KB
 EsecuzioneTestCiclo29	06/04/2021 21:30	Documento di testo	1 KB
 reportErrori	06/04/2021 21:30	Documento di testo	5 KB

## ErrorTest\_Thread-3:

Nome	Ultima modifica	Tipo	Dimensione
 com	06/04/2021 21:20	Cartella di file	
 org	06/04/2021 21:20	Cartella di file	
 ErrorTest.class	06/04/2021 21:29	File CLASS	1 KB
 ErrorTest	06/04/2021 21:29	File JAVA	1 KB
 ErrorTest0.class	06/04/2021 21:29	File CLASS	4 KB
 ErrorTest0	06/04/2021 21:29	File JAVA	7 KB
 esecuzione_test	10/03/2021 19:08	File batch Windows	1 KB
 EsecuzioneTestCiclo5	06/04/2021 21:20	Documento di testo	1 KB
 EsecuzioneTestCiclo6	06/04/2021 21:21	Documento di testo	1 KB
 EsecuzioneTestCiclo7	06/04/2021 21:21	Documento di testo	1 KB
 EsecuzioneTestCiclo8	06/04/2021 21:22	Documento di testo	1 KB
 EsecuzioneTestCiclo9	06/04/2021 21:22	Documento di testo	2 KB
 EsecuzioneTestCiclo12	06/04/2021 21:23	Documento di testo	1 KB
 EsecuzioneTestCiclo13	06/04/2021 21:24	Documento di testo	2 KB
 EsecuzioneTestCiclo14	06/04/2021 21:24	Documento di testo	2 KB
 EsecuzioneTestCiclo15	06/04/2021 21:24	Documento di testo	1 KB
 EsecuzioneTestCiclo16	06/04/2021 21:25	Documento di testo	2 KB
 EsecuzioneTestCiclo19	06/04/2021 21:26	Documento di testo	1 KB
 EsecuzioneTestCiclo21	06/04/2021 21:27	Documento di testo	1 KB
 EsecuzioneTestCiclo22	06/04/2021 21:27	Documento di testo	1 KB
 EsecuzioneTestCiclo23	06/04/2021 21:28	Documento di testo	1 KB
 EsecuzioneTestCiclo24	06/04/2021 21:28	Documento di testo	2 KB
 EsecuzioneTestCiclo27	06/04/2021 21:29	Documento di testo	1 KB
 reportErrori	06/04/2021 21:30	Documento di testo	5 KB



# Capitolo 3

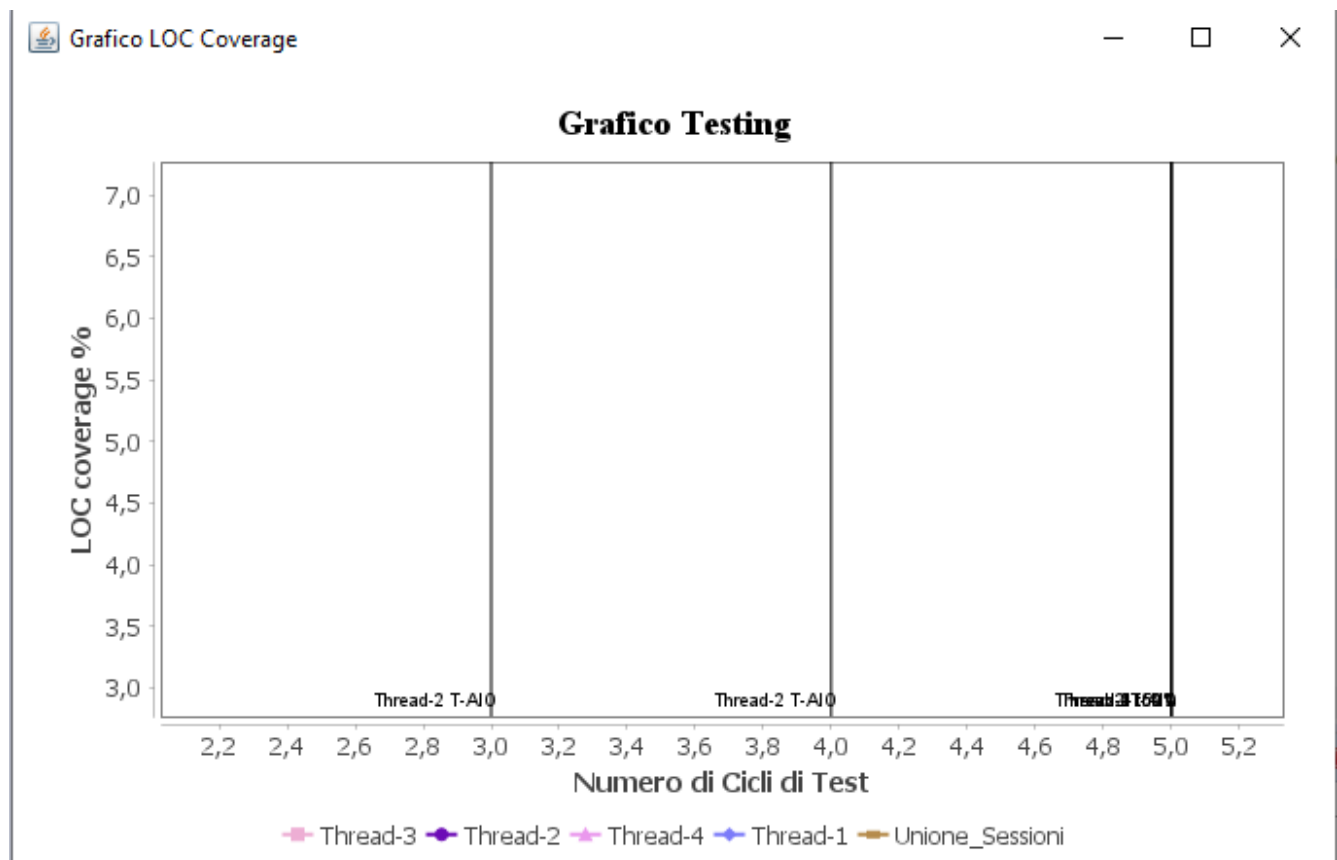
## 3.1 Valutazione delle prestazioni sull'applicazione "Saxpath"

Di seguito verranno riportati i risultati ottenuti testando l'applicazione Saxpath, essa è un'applicazione composta da 16 classi.

### 3.1.1 Esecuzione di 4 sessioni di test dedicando 20 secondi per ogni classe

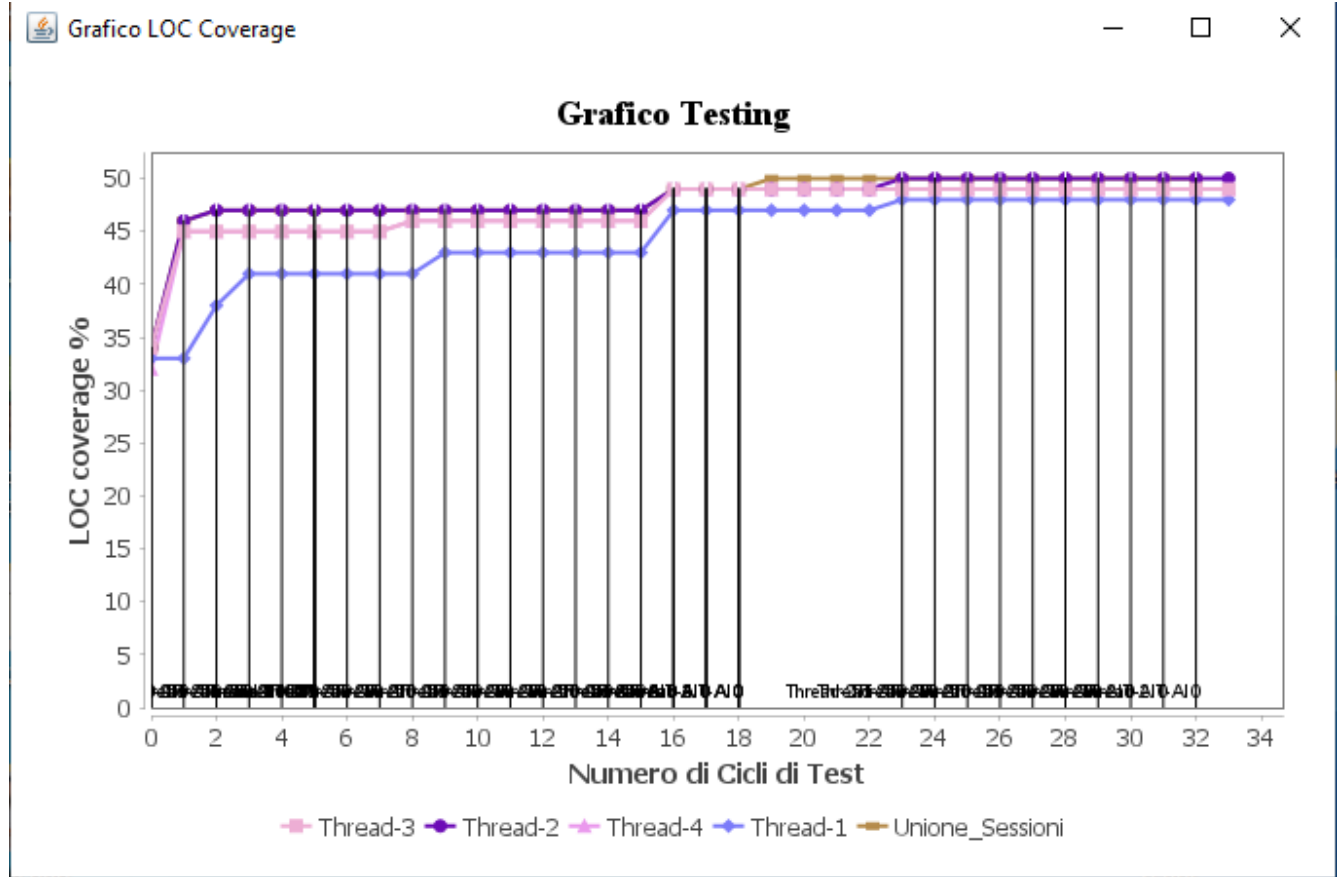
In questo primo esperimento sono state lanciate 4 sessioni di test in parallelo, ognuna delle quali, ad ogni ciclo, dedicherà 20 secondi per la generazione degli input per ogni classe.

La cosa che si è notata è stata il raggiungimento immediato da parte della sessione 2 del criterio "All Include One", il quale viene applicato nel momento in cui una sessione ha lo stesso andamento dell'unione delle sessioni. Pertanto se quest'ultimo fosse stato adottato come criterio di arresto, il thread 2 sarebbe stato interrotto dopo appena tre cicli. Per apprezzare meglio il raggiungimento di tale condizione, verrà mostrato di seguito un ingrandimento del grafico generato, con uno zoom sulla segnalazione ottenuta:



Un'altra considerazione interessante è, come si nota con qualche difficoltà dal grafico, che tutte le quattro sessioni raggiungono il criterio di terminazione T50% dopo 5 cicli di test.

Dopo 17 cicli di test il criterio AIO viene raggiunto anche dalla sessione 3 e dalla sessione 4, e dal ciclo 23 la copertura delle sessioni non ha nessun miglioramento:

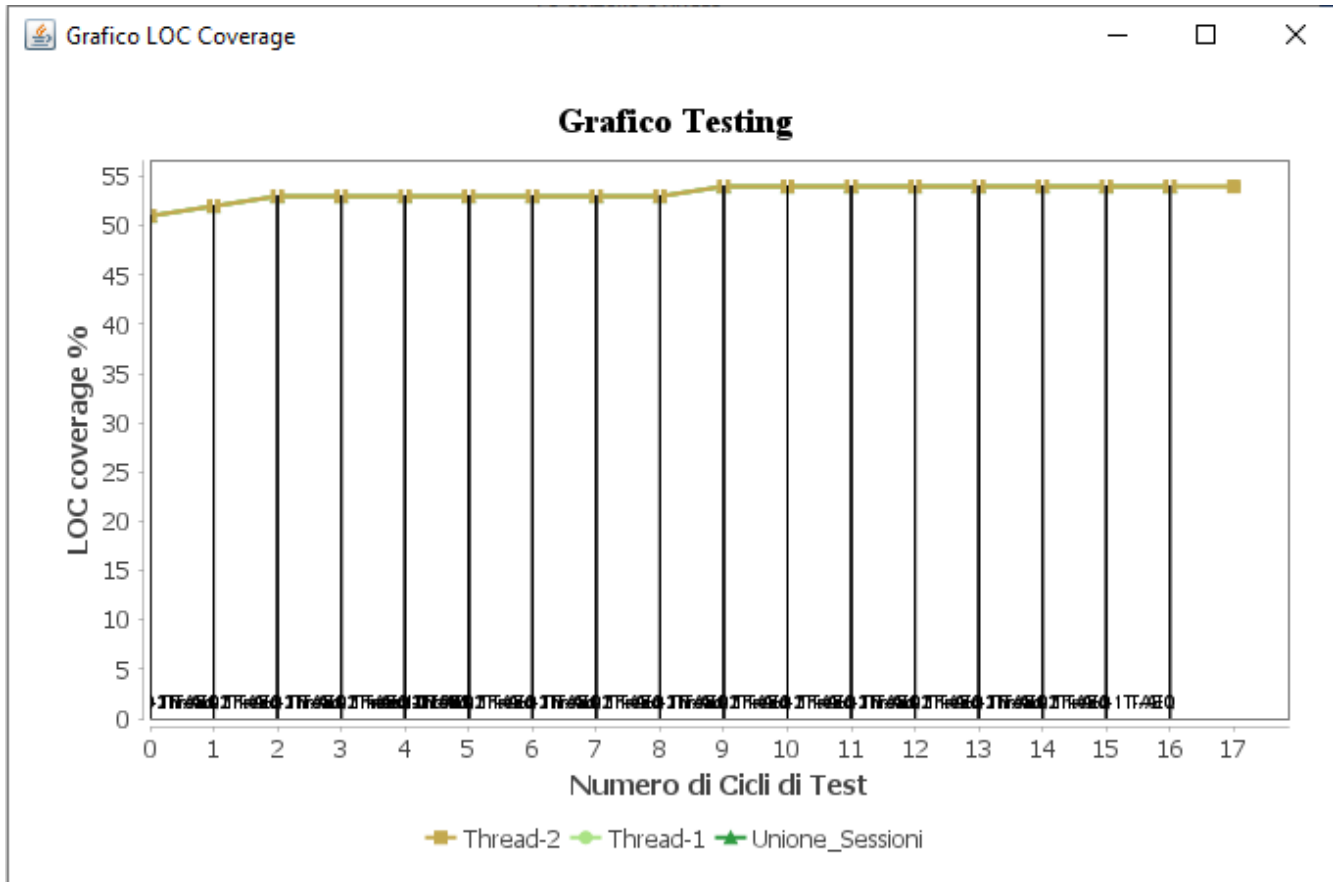


Si nota dal grafico che c'è un "buco" tra il ciclo 18 e il ciclo 23 in quanto in questi cicli non c'è nessuna sessione che ha la stessa coverage dell'unione, il criterio AEQ e T60% non sono soddisfatti e il criterio T50% è stato già raggiunto in precedenza.

Dal punto di vista degli errori generati, risulta che tutte e 4 le sessioni producono un unico ErrorTest contenente un solo test case ciascuno, i quali falliscono una volta eseguiti generando quindi quattro errori diversi, uno per ogni sessione di test.

### 3.1.2 Esecuzione di 2 sessioni di test dedicando 60 secondi per ogni classe

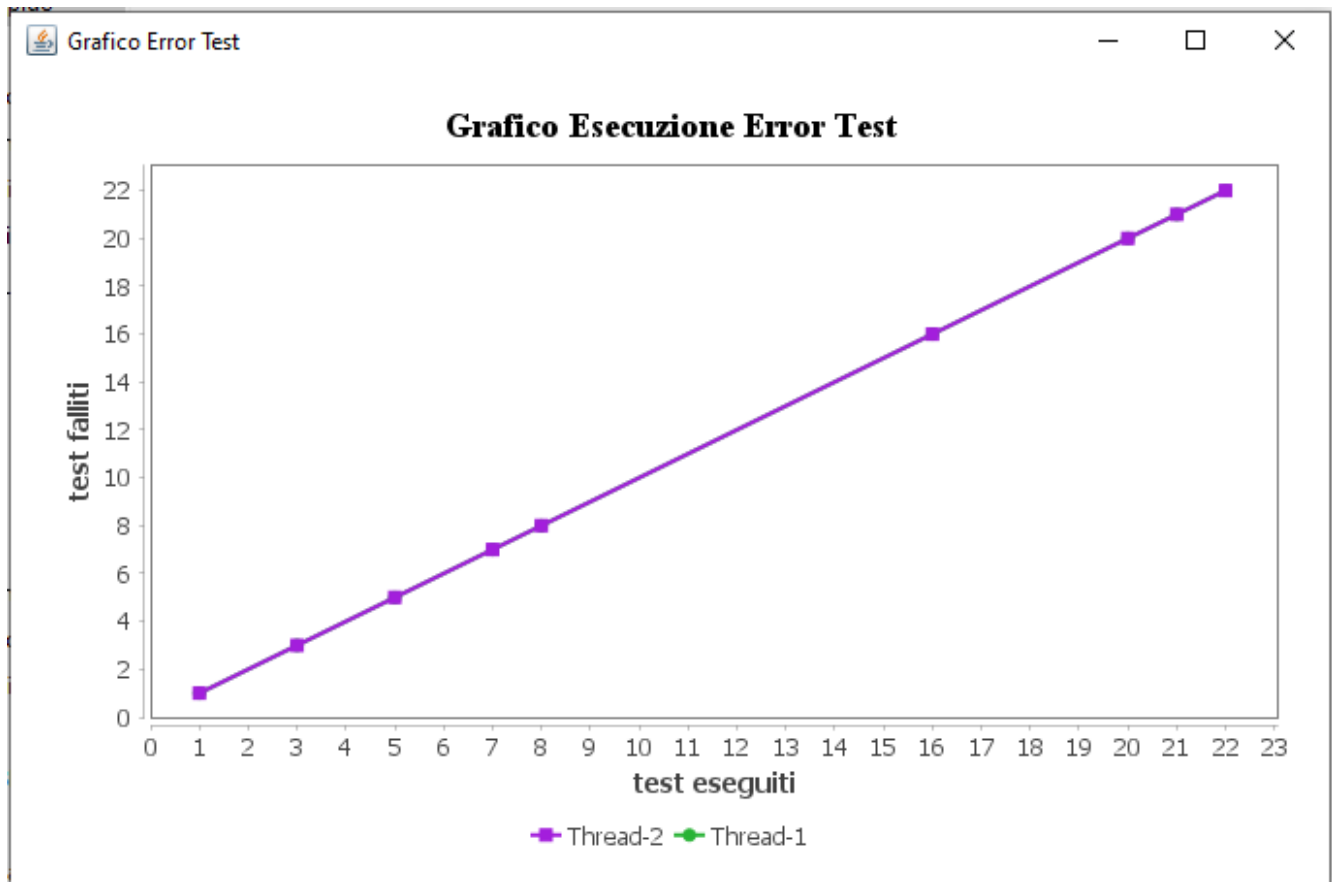
Si nota in questo caso che le sessioni hanno esattamente lo stesso andamento dal punto di vista della copertura. La differenza rispetto a prima è che le percentuali di coverage raggiunte sono più alte, ma non abbastanza da soddisfare il criterio T60%. Anche in questo caso dopo 5 cicli di test è stato raggiunto il criterio di arresto T50%.



Una differenza sostanziale rispetto al caso precedente è rappresentato dalla quantità di ErrorTest prodotti. Infatti in questo caso quasi ad ogni ciclo sono stati individuati degli input invalidi, di conseguenza quasi tutti i cicli hanno prodotto dei test da eseguire. In questo caso particolare i test prodotti sono gli stessi per le due sessioni:

Nome	Ultima modifica	Tipo	Dimensione
com	22/03/2021 17:27	Cartella di file	
org	22/03/2021 17:27	Cartella di file	
ErrorTest.class	22/03/2021 17:46	File CLASS	1 KB
ErrorTest.java	22/03/2021 17:45	File JAVA	1 KB
ErrorTest0.class	22/03/2021 17:46	File CLASS	3 KB
ErrorTest0.java	22/03/2021 17:45	File JAVA	4 KB
esecuzione_test.bat	10/03/2021 19:08	File batch Windows	1 KB
EsecuzioneTestCiclo2.txt	22/03/2021 17:27	Documento di testo	1 KB
EsecuzioneTestCiclo5.txt	22/03/2021 17:30	Documento di testo	1 KB
EsecuzioneTestCiclo6.txt	22/03/2021 17:31	Documento di testo	1 KB
EsecuzioneTestCiclo7.txt	22/03/2021 17:32	Documento di testo	1 KB
EsecuzioneTestCiclo10.txt	22/03/2021 17:36	Documento di testo	1 KB
EsecuzioneTestCiclo11.txt	22/03/2021 17:37	Documento di testo	3 KB
EsecuzioneTestCiclo12.txt	22/03/2021 17:38	Documento di testo	2 KB
EsecuzioneTestCiclo15.txt	22/03/2021 17:41	Documento di testo	1 KB
EsecuzioneTestCiclo16.txt	22/03/2021 17:42	Documento di testo	1 KB
EsecuzioneTestCiclo19.txt	22/03/2021 17:46	Documento di testo	1 KB
reportErrori.txt	22/03/2021 17:43	Documento di testo	3 KB

Il grafico riportante l'andamento dell'esecuzione dei test è il seguente:



Invece l'elenco degli errori trovati con relativa occorrenza è il seguente:

```

reportError.txt - Blocco note di Windows
File Modifica Formato Visualizza ?
i-hashcode on conformanceXPathHandler22 and XPathHandler21 OCCORRENZA -> 1
i-hashcode on conformanceXPathHandler8 and XPathHandler7 OCCORRENZA -> 1
i-hashcode on conformanceXPathHandler20 and conformanceXPathHandler6 OCCORRENZA -> 1
i-hashcode on conformanceXPathHandler22 and conformanceXPathHandler0 OCCORRENZA -> 2
i-hashcode on XPathHandler12 and conformanceXPathHandler14 OCCORRENZA -> 1
i-hashcode on conformanceXPathHandler54 and conformanceXPathHandler21 OCCORRENZA -> 1
i-hashcode on conformanceXPathHandler12 and conformanceXPathHandler1 OCCORRENZA -> 1
i-hashcode on conformanceXPathHandler7 and XPathHandler4 OCCORRENZA -> 1
i-hashcode on conformanceXPathHandler21 and XPathHandler4 OCCORRENZA -> 1
i-hashcode on XPathHandler6 and conformanceXPathHandler1 OCCORRENZA -> 1
i-hashcode on conformanceXPathHandler36 and XPathHandler8 OCCORRENZA -> 1
i-hashcode on conformanceXPathHandler8 and conformanceXPathHandler1 OCCORRENZA -> 1
i-hashcode on conformanceXPathHandler4 and conformanceXPathHandler28 OCCORRENZA -> 1
i-hashcode on conformanceXPathHandler35 and conformanceXPathHandler78 OCCORRENZA -> 1
i-hashcode on conformanceXPathHandler1 and XPathHandler52 OCCORRENZA -> 1
i-hashcode on conformanceXPathHandler0 and conformanceXPathHandler36 OCCORRENZA -> 1
i-hashcode on conformanceXPathHandler1 and conformanceXPathHandler14 OCCORRENZA -> 2
i-hashcode on conformanceXPathHandler1 and conformanceXPathHandler23 OCCORRENZA -> 1
i-hashcode on conformanceXPathHandler0 and conformanceXPathHandler19 OCCORRENZA -> 1
i-hashcode on conformanceXPathHandler14 and XPathHandler11 OCCORRENZA -> 1

```

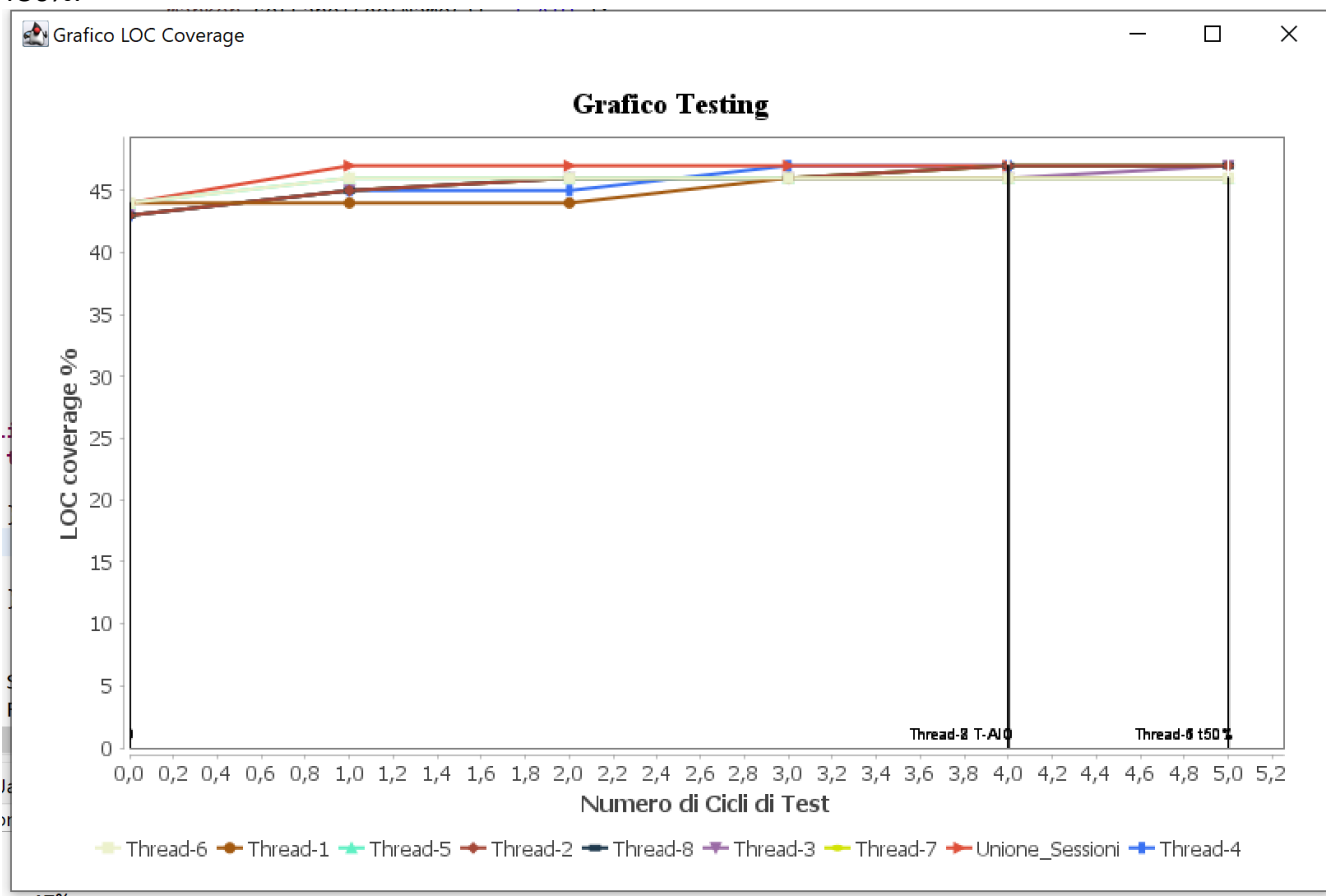
Linea 1, colonna 1 100% Unix (LF) UTF-8

## 3.2 Valutazione delle prestazioni sull'applicazione "Jipa"

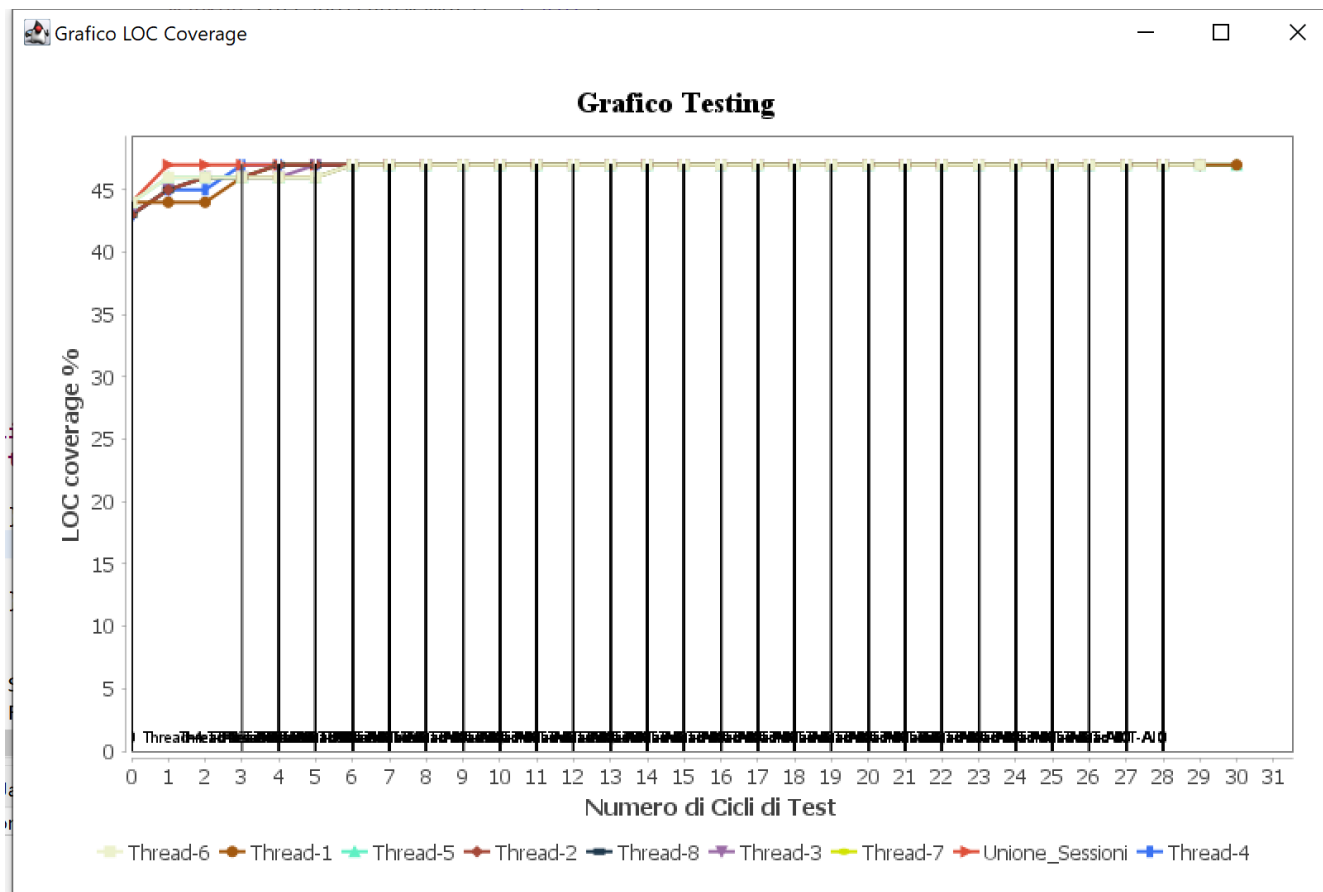
Di seguito verranno riportati i risultati ottenuti testando l'applicazione Jipa, essa a differenza della precedente è un'applicazione più piccola poiché composta da 5 classi.

### 3.2.1 Esecuzione di 8 sessioni di test con 60 secondi per ogni classe

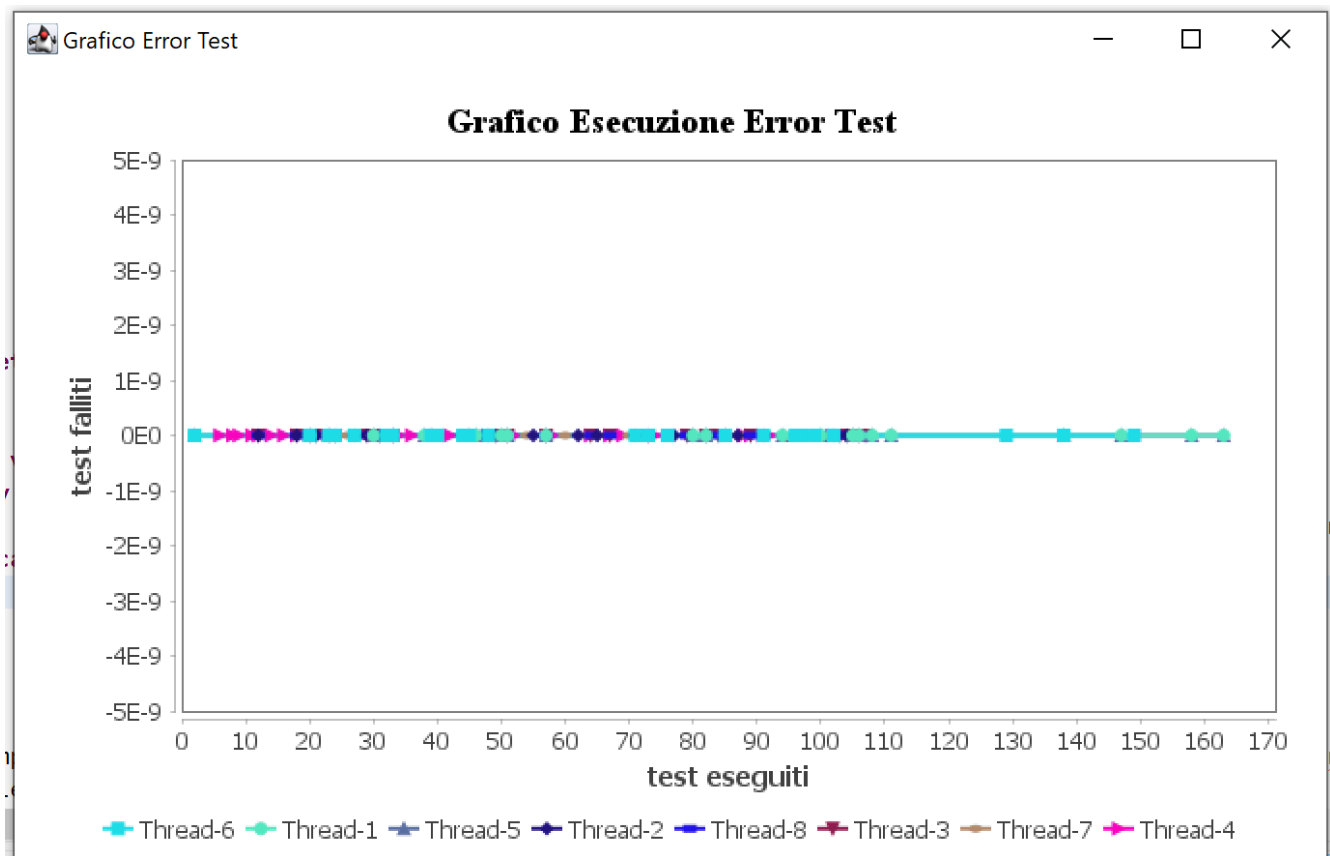
In questo esperimento sono state lanciate 8 sessioni in parallelo. Si nota che dopo quattro cicli di test diverse sessioni raggiungono la stessa copertura dell'unione, e altre raggiungono il criterio di arresto T50%.



Possiamo notare dal grafico seguente che dopo 5 cicli di test nessuna delle sessioni migliora, e tutte si stabilizzano sullo stesso valore:



La cosa interessante in questo esperimento è rappresentata dalla quantità di test eseguiti. Notiamo infatti come dopo 30 cicli il numero di test eseguiti dalle sessioni è superiore a 120 per ognuna. In particolare le sessioni 1 e 8 sono quelle che hanno prodotto più test, infatti la prima ne ha prodotti 161, mentre l'ottava ne ha prodotti 165. Nessuno di questi test ha prodotto un fallimento, così come emerge anche dal grafico.



## Capitolo 4

### Conclusioni

Il Random Testing è una tecnica poco dispendiosa e di facile utilizzo, dal momento che l'approccio con cui viene collaudato un applicativo è totalmente demandato al sistema di elaborazione. Inoltre esso offre la possibilità di effettuare test senza la necessità che si conoscano i dettagli dell'applicazione sotto test.

Più in generale l'automatizzazione rende più agevoli le operazioni di collaudo del software, ma richiede una progettazione attenta e precisa delle procedure da applicare affinché si ottengano buone prestazioni sulle metriche di copertura del codice, contenendo contemporaneamente le risorse temporali disponibili. Nonostante i vantaggi che offre, però, il testing casuale presenta anche qualche effetto collaterale: quello più rilevante è dato proprio dalla sua natura fortemente arbitraria, che non essendo predicibile potrebbe produrre sequenze simili tra loro per lunghi lassi di tempo, senza alcun miglioramento in termini di copertura del codice e/o di segnalazione di bug.