

UNIVERSITA' DEGLI STUDI DI  
NAPOLI FEDERICO II

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



Tesina Esame Software Testing

## **Automated Parallel Random Testing using Randoop and Emma**

Anno Accademico 2022/2023

Studenti:

**Riccardo Romano  
Giovanni Riccardi  
Francesco Panariello**

## Indice

---

Indice.....	2
Introduzione .....	3
Capitolo 1: Gli Strumenti .....	5
1.1 Randoop .....	5
1.2 Emma .....	6
1.3 GitHub Actions .....	8
Capitolo 2: La realizzazione .....	10
2.1 Come utilizzare la repository Parallel Random Testing .....	11
2.2 Esempio di esecuzione .....	14
2.2.1 Output generati.....	17
Conclusioni .....	19

## Introduzione

---

La richiesta iniziale per l'ideazione del Progetto è quella di implementare uno strumento in grado di eseguire casi di test in parallelo su macchine virtuali disponibili in cloud (come GitHub), capace di fornire e valutare differenti valori di copertura raggiunti.

L'idea di base è quella di andare ad utilizzare uno strumento per la generazione automatica di test casuali che possa, in poco tempo, fornire un numero molto elevato di casi di test. In questo modo è possibile ottenere test totalmente automatici che presentano però un'efficienza poco elevata, migliorabile mediante una buona strategia di testing.

Il problema dei test randomici è quello dell'efficienza, dunque, in quanto è probabile che alcuni test già creati vengano generati nuovamente; inoltre, la non presenza di un oracolo porta a valutare crash e la violazione di proprietà invarianti.

D'altro canto effettuare test in maniera randomica permette di ottenere un buon numero di test in poco tempo, differenti da quelli generati da un tester, scoprendo così nuovi difetti e guasti. Il progetto è stato implementato con l'utilizzo di due strumenti: Randoop ed Emma. Randoop è uno strumento per la generazione automatica di casi di test che parte da un'analisi del codice (limitata ai metodi ed ai parametri). I test sono generati come script JUnit dando la possibilità di rieseguirli e consistono in sequenze casuali di chiamate a metodi su oggetti. Emma invece è uno strumento utilizzato ottimamente con Randoop per la raccolta dei valori di copertura dei test. Infatti, il tool analizza i casi di test e li strumentalizza in modo da ottenere informazioni sui valori di copertura. Dopo che i test vengono eseguiti, Emma raccoglie tali informazioni all'interno di un report in formato vario, come HTML o XML.

Grazie ad Emma gli sviluppatori possono facilmente osservare dove il codice non è stato in grado di fornire una copertura soddisfacente in modo da concentrarsi sulla futura scrittura di test più corretti e completi.

Questi due tool vengono utilizzati sulla piattaforma GitHub mediante uno script YAML per la realizzazione di un workflow adoperabile attraverso GitHub Actions.

In questo modo la generazione dei casi di test avviene in maniera parallela su macchine virtuali in cloud.

# Capitolo 1: Gli Strumenti

---

Come descritto in precedenza, la realizzazione del progetto avviene mediante l'utilizzo di tre tool fondamentali: Randoop, Emma e GitHub Actions.

Il primo utilizzato per la realizzazione di casi di test randomici, il secondo per l'analisi e la memorizzazione dei valori di coverage raggiunti e l'ultimo per realizzare delle automazioni..

## 1.1 Randoop

Randoop è un generatore di test di unità per Java ideato da Microsoft Research.

Esso crea automaticamente test per le classi in formato JUnit. Il suo lavoro si basa sulla tecnica del feedback direct random testing (test casuale orientato a retroazione): genera sequenze casuali a partire dalle classi del programma in prova, che poi invia al programma stesso, formulando periodicamente delle asserzioni sul suo comportamento. In questo modo è possibile richiamare l'attenzione su eventuali carenze e/o eccezioni del relativo programma, nonché ottenere test di regressione per evitare di introdurre errori nelle versioni aggiornate. Pertanto, questa tecnica genera in modo pseudo-casuale ma intelligente, chiamate di metodo/costruttore per le classi in fase di test. Randoop esegue le sequenze generate, utilizzando i risultati dell'esecuzione per creare asserzioni che descrivono il comportamento del programma.

Randoop genera tipicamente due tipi di file:

- **Error Test:** Test che trovano errori/bug nel codice;
- **Regression Test:** Test di regressione per eventuali modifiche future;

Come descritto dal sito ufficiale di Randoop, la combinazione di generazione ed esecuzione di test di tale strumento, si traduce in una tecnica di generazione di test molto efficace. Randoop ha rivelato errori precedentemente sconosciuti anche in librerie molto utilizzate, tra cui i JDK di Sun e IBM e un componente core di .NET.

Ciò che risulta utile per i fini del progetto è la possibilità di inserire mediante linea di comando, alcuni parametri facoltativi, come:

*output-limit* : Utile per inserire il numero massimo di test da eseguire;

*time-limit*: Utile per inserire il tempo massimo durante il quale eseguire i test;

*randoom-seed*: Seme utilizzato per la generazione di test casuali;

*classlist*: Permette di inserire un file .txt, con la lista di classi da testare;



## 1.2 Emma

Emma è uno strumento utilizzato per misurare e riportare la copertura del codice Java di un dato progetto, ideato da Vlad Roubtsov per aiutare gli sviluppatori Java fornendo un tool gratuito per il controllo della copertura, anziché spendere cifre esorbitanti tra licenze varie anche per il controllo di un progetto di piccole dimensioni.

Emma tiene traccia del codice Java e riporta i valori di copertura in un dato report, che può essere in formato XML, HTML o semplicemente .txt, che contiene i valori di coverage in termini percentuali. Il report risulta chiaro e ben suddiviso, così da fornire allo sviluppatore la possibilità di comprendere velocemente dove ottimizzare i test per la realizzazione di un progetto più completo e stabile.

Inoltre, Emma può essere integrato in molti ambienti di sviluppo integrati (IDE) come Eclipse, IntelliJ IDEA e NetBeans, rendendo facile l'uso da parte degli sviluppatori durante il ciclo di sviluppo del software.

Ecco ora un esempio di report sulla coverage di un generico progetto in formato HTML:

**OVERALL COVERAGE SUMMARY**

name	class, %	method, %	block, %	line, %
all classes	100% (11/11)	74% (153/207)	51% (1549/3022)	49% (508.7/1035)

**OVERALL STATS SUMMARY**

total packages: 4  
total executable files: 10  
total classes: 11  
total methods: 207  
total executable lines: 1035

**COVERAGE BREAKDOWN BY PACKAGE**

name	class, %	method, %	block, %	line, %
com.werken.saxpath	100% (5/5)	62% (50/144)	39% (899/2297)	39% (326.9/830)
org.saxpath.helpers	100% (1/1)	100% (4/4)	55% (66/119)	65% (22.9/35)
org.saxpath	100% (4/4)	100% (11/11)	89% (184/206)	84% (56/67)
org.saxpath.conformance	100% (1/1)	100% (48/48)	100% (400/400)	100% (103/103)

Si noti che attraverso EMMA è possibile esplorare il codice sorgente secondo varie tipologie: per classi, per metodi, per linee e per blocchi, a seconda del dettaglio con cui si vuole misurare la coverage del testing.

Emma permette due tipi di esecuzione:

- *Offline*: La strumentazione è completamente indipendente dall'esecuzione dell'applicazione che si vuole testare; in questa modalità si eseguono i test prima di analizzare il codice e si ottiene un report di copertura dei test successivamente all'esecuzione.
- *“On the fly”*: Verificando gli eventi a tempo di esecuzione, l'analisi di copertura dunque viene fatta durante l'esecuzione dei test, in tempo reale.

Per lanciare Emma, contestualmente alla generazione dei test con Randoop, viene utilizzata la modalità offline, viene fatta quindi una strumentazione dell'applicazione per valutare la copertura dei test generati.

Emma, inoltre, è fondamentale poiché mette a disposizione la funzione “merge”, la quale permette di aggregare facilmente i risultati di sessioni differenti, ottenendo quindi la loro unione.

## 1.3 GitHub Actions

GitHub Actions è una piattaforma di continuous integration and continuous delivery (CI/CD), che attraverso un sistema a pipeline permette di effettuare operazioni di compilazione, testing e distribuzione.

Principalmente il suo funzionamento è correlato ad eventi o richieste effettuate su di una repository (ne sono un esempio operazioni di push, pull o update) ma non ci si limita solo a questo, in quanto oltre a numerosi filtri è possibile ad esempio creare un flusso di lavoro per aggiungere automaticamente le etichette appropriate ogni volta che qualcuno crea un nuovo problema nella repository.

Le actions possono essere eseguite su macchine virtuali messe a disposizione da GitHub, principalmente gli SO sono Linux, Windows e MacOS altrimenti è possibile eseguirle su strutture con hosting privato.

Scendendo più nel particolare, per realizzare un flusso di lavoro abbiamo bisogno di scrivere un file di tipo YAML ed inserirlo nella repository.

YAML è un formato che nasce come linguaggio di markup che diventa successivamente sempre più usato per la serializzazione e configurazione dati.

I suoi punti di forza sono:

- Maggior leggibilità
- Possibilità di concatenazione ottenendo YAML validi
- Possibilità di auto referenziarsi
- Supporto di tipi complessi
- Supporto di commenti e blocchi di testo

Il file YAML deve avere però un trigger di avvio che può essere: una push, un avvio manuale o una delle altre possibilità messe a disposizione da GitHub.

Nel nostro caso, risulta essere lo strumento adatto poiché ci permette non solo di avviare in autonomia qualunque programma .jar inserissimo sulla repository, ma anche perché





tramite dei comandi specifici è possibile parallelizzare le operazioni, che nel caso di testing aleatorio è essenziale per non avere un enorme dispendio di tempo.

Concetti fondamentali di GitHub Actions sono "jobs", "steps" e "uses" che vengono utilizzati per definire i workflow:

- *"jobs"* definisce una singola unità di lavoro che viene eseguita in parallelo ad altri job. Ogni job può avere uno o più *"steps"* associati, che vengono eseguiti in sequenza per completare il job.
- *"steps"* definisce un singolo passo all'interno di un job. Ogni step esegue un'azione specifica, come ad esempio il controllo del codice sorgente, la compilazione o l'esecuzione dei test, o l'esecuzione di codici bash.
- *"uses"* è una direttiva YAML utilizzata all'interno di uno step per richiamare un'azione di GitHub. Sono d'esempio azioni messe a disposizione dalla piattaforma, azioni personalizzate presente in un repository o nel marketplace di GitHub.

Per scambiare informazioni all'interno del workflow, fra job diversi, GitHub Actions mette a disposizione gli *artifacts*: file o pacchetti generati durante l'esecuzione di una workflow che vengono salvati in modo persistente. Una volta che un job di GitHub ha creato un artifact, esso viene salvato e reso disponibile per il workflow. Gli artifacts possono essere scaricati e utilizzati in fasi successive, consentendo di condividere i risultati tra diversi job e workflow. Inoltre, gli artifacts possono essere visualizzati nella pagina di esecuzione del workflow sulla piattaforma GitHub, consentendo di esaminare i risultati della build in modo più dettagliato.

Artifacts		
Produced during runtime		
Name		Size
	Coverage merge	163 KB
	Emma reports	1.42 MB

## Capitolo 2: La realizzazione

---

Per la realizzazione del progetto di parallelizzazione dei casi di test mediante l'utilizzo di strumenti quali Randoop ed Emma, è stato utilizzato un codice YAML su GitHub Actions che definisce un workflow eseguito su di un'istanza Ubuntu latest composta da due job fondamentali: “tests” e “merge”, ed un terzo, per la creazione dei badge di coverage e le operazioni finali dell'esecuzione.

Il job *tests* è definito come una matrice di otto nodi, ognuno dei quali esegue dei test mediante Randoop ed Emma in parallelo, in maniera totalmente indipendente fra loro. Si noti che prima di eseguire i test, avviene un controllo sulla repository ed una successiva configurazione di JDK versione 1.8 per poter eseguire con successo i tool utilizzati. Per ottenere con certezza casi di test completamente randomici tra di loro nelle varie istanze della matrice, è stata definita una variabile d'ambiente *randseed* mediante uno script bash per ottenere un seme random per la generazione dei test. Successivamente vengono avviati Randoop ed Emma e memorizzati i risultati delle coverage di ogni sessione, i quali, attraverso l'uso degli artifacts, vengono scaricati nell'esecuzione del job “merge”, in cui viene creato il file di coverage totale e “mergiato” a sua volta, con il file di coverage merge della precedente esecuzione del workflow, presente all'interno della repository.

Il job *badges* finale, effettua le operazioni per preparare la repository ad una nuova esecuzione, effettuando la push dei file di coverage merge generati e “triggerando” una nuova esecuzione del workflow.






Per analizzare nel dettaglio il ragionamento che c'è dietro il codice, è ottimale andare a studiare passo passo i vari step.

## 2.1 Come utilizzare la repository Parallel Random Testing

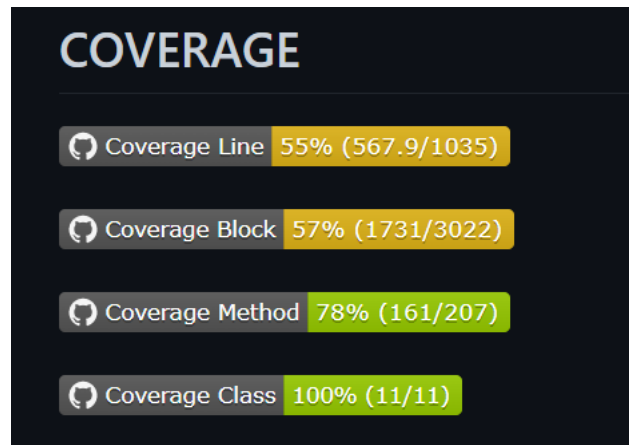
La repository ideata deve essere utilizzata seguendo uno schema ben preciso, in quanto avrà bisogno di un progetto in java (.jar) da testare, che dovrà essere inserito all'interno della directory "jars". Dopo aver caricato il progetto, è necessario modificare il file "lista.txt" inserendo il nome delle classi da testare.

Come ultimo passaggio, ma fondamentale, bisognerà andare a modificare autonomamente lo script YAML "main.yml" inserendo appositamente il nome del proprio progetto da testare all'interno della linea di codice di Randoop ed in quella di Emma (indicate nel file README del repository).

Infine, si potrà accedere alla sezione Action per avviare la generazione parallela e casuale dei casi di test, ottenendo in output un valore di coverage.

 .github/workflows	Update main.yml
 jars	Add files via upload
 merge/coverage	Changed files
 README.md	Update README.md
 lista.txt	Add files via upload

Il valore della coverage ad ogni esecuzione è memorizzato e visualizzato mediante dei badges all'interno del README per visualizzare velocemente gli incrementi ottenuti test dopo test. Ogni badge inoltre può assumere colori differenti: rosso, giallo e verde mediante l'utilizzo di una variabile d'ambiente che viene controllata. Infatti si otterrà il colore verde se la copertura è maggiore o uguale al 60%, gialla nel caso in cui sia compresa tra 51% e 59% e rosso quando è minore o uguale al 50%. Si noti che viene utilizzato il criterio di copertura del codice  $\geq 60\%$ , il quale si ottiene quando i test raggiungono il 60% di copertura del codice.



Il workflow valuta come criterio di terminazione, per l'esecuzione dei test, il raggiungimento di una coverage fissata, dunque, all'interno del codice è settato un valore di coverage, al raggiungimento del quale l'esecuzione ciclica del workflow si ferma, tramite un controllo che precede l'operazione "trigger" del workflow.

Tale valore di coverage è modificabile in "main.yml" inserendo un certo valore di copertura prima del quale l'esecuzione non si fermerà.

In questo modo l'action continuerà ad eseguire casi di test finché non verrà raggiunta la coverage desiderata, o in caso questa si stabilizzi prima, non raggiungendo mai l'obiettivo, è possibile fermare manualmente l'esecuzione dell'ultimo workflow avviato. Il codice che implementa questa funzione è il seguente:

```

- name: Percentuale
  run: |
    #!/bin/bash
    value=$(echo "${env.MY_VAR}" | awk '{match($0, /[0-9]+/); print substr($0, RSTART, RLENGTH)}')
    echo "value=$value" >> $GITHUB_ENV
    if (($value < 56)); then
      bool=1
      echo "bool=$bool" >> $GITHUB_ENV
    else
      bool=0
      echo "bool=$bool" >> $GITHUB_ENV
    fi

- if: ${env.bool == 1}
  uses: stefanzweifel/git-auto-commit-action@v4

  with:
    commit_message: Changed files
    file_pattern: merge/*

- if: ${env.bool == 1}
  uses: benc-uk/workflow-dispatch@v1
  with:
    workflow: main.yml

```

In primis avviene un controllo da parte dell'action sul file XML per estrapolare i valori di coverage di classi, metodi, blocchi e linee; successivamente mediante l'utilizzo di uno script bash salviamo solamente il numero che precede il carattere “%” all'interno di una variabile d'ambiente chiamata “value”. Tale scelta è dovuta al fatto che per rispettare le condizioni dell'if abbiamo bisogno di confrontare valori interi privi di caratteri speciali. (Si noti che il valore “56” all'interno dell'if è solamente a scopo esemplificativo e può essere modificato come si desidera.)

Dunque, nel momento in cui la coverage line assume un valore minore del 56%, l'action effettuerà una push automatica dei file “coveragemerge.es”, “coveragemerge.xml” e “coveragemerge.html” all'interno di una directory merge/coverage.

Il fine di questa operazione è quella di andare ad eseguire nuovamente l'action main.yml in maniera automatica mediante un comando GitHub; infatti l'action effettuerà un controllo nel quale verifica l'esistenza di un file “coverage.es” all'interno della repository, se il file è presente l'action effettua il merge tra il valore di coverage ottenuto in precedenza con il nuovo appena generato. Si analizza dunque nuovamente se il valore di coverage ha raggiunto il 56%, in caso contrario verrà eseguita nuovamente l'action dal bot di GitHub-Actions.

## 2.2 Esempio di esecuzione

Dopo aver analizzato il modo corretto per utilizzare efficientemente la repository, di seguito è proposto un esempio di esecuzione di un dato progetto “saxpath”.

Supponiamo di voler analizzare la copertura del progetto saxpath mediante l’ausilio di Randoop ed Emma per la generazione di casi di test randomici.

Per prima cosa si dovrà inserire il progetto in formato jar, all’interno della directory “jars” della repository, ed assicurarci che non sia presente la directory merge/coverage.

Successivamente vengono inserite anche tutte le classi che si desidera testare mediante la repository all’interno del file lista.txt :

```
1  org.saxpath.SAXPathException
2  org.saxpath.XPathReader
3  org.saxpath.SAXPathEventSource
4  org.saxpath.SAXPathParseException
5  com.werken.saxpath.DefaultXPathHandler$Singleton
6  org.saxpath.Operator
7  com.werken.saxpath.TokenTypes
8  org.saxpath.XPathSyntaxException
9  org.saxpath.Axis
10 com.werken.saxpath.Token
11 org.saxpath.XPathHandler
12 org.saxpath.helpers.XPathReaderFactory
13 com.werken.saxpath.DefaultXPathHandler
14 org.saxpath.conformance.ConformanceXPathHandler
15 com.werken.saxpath.XPathLexer
16 com.werken.saxpath.XPathReader
17
```

D’ora in poi bisognerà effettuare delle modifiche all’interno del workflow main.yml, andando ad inserire nelle linee di codice 47 e 50 il nome del nostro progetto.

## Emma:

```
java -noverify -classpath /home/runner/work/ParallelRandomTesting/ParallelRandomTesting/emma-2.0.5312/lib/emma.jar
emma instr -m fullcopy -d /home/runner/work/ParallelRandomTesting/ParallelRandomTesting/coverage_${timestamp}
-ip /home/runner/work/ParallelRandomTesting/ParallelRandomTesting/jars/saxpath.jar -out
coverage_${timestamp}/coverage_${timestamp}.em
```

## Randoop:

```
java -noverify -classpath
/home/runner/work/ParallelRandomTesting/ParallelRandomTesting/coverage_${timestamp}/lib/saxpath.jar :
/home/runner/work/ParallelRandomTesting/ParallelRandomTesting/emma-2.0.5312/lib/emma.jar :
/home/runner/work/ParallelRandomTesting/ParallelRandomTesting/randoop-all-3.0.6.jar
randoop.main.Main gentests -classlist=/home/runner/work/ParallelRandomTesting/ParallelRandomTesting/lista.txt
--timelimit=20 --randomseed=${{ env.randseed }} --no-regression-tests=true
--junit-output-dir = /home/runner/work/ParallelRandomTesting/ParallelRandomTesting/coverage_${timestamp}
```

Si noti che è possibile inserire il numero di test limite da eseguire per ogni sessione di test in parallelo. A scopo esemplificativo è stato inserito un valore di **timelimit=20** per non rendere l'esecuzione dei casi di test troppo prolissa. Dopo aver modificato tali linee, si prosegue con la modifica (se desiderata) del valore massimo di coverage da raggiungere, in questo caso, come mostrato precedentemente è stato scelto un valore limite di 56%. Fatto ciò, è possibile recarsi nella sezione **Actions** ed eseguire il workflow:



Come si può notare, vengono eseguiti parallelamente otto sessioni di test che effettuano testing randomico mediante Randoop.

Al termine dell'esecuzione di tutte ed otto le sessioni, otterremo all'interno dell'artifact un file zip "Emma reports", che contiene un report con i valori di coverage di ognuna delle otto sessioni di test. Ogni report è descritto da un file in formato HTML come questo:

#### OVERALL COVERAGE SUMMARY

name	class, %	method, %	block, %	line, %
all classes	100% (11/11)	73% (152/207)	50% (1525/3022)	48% (499.7/1035)

#### OVERALL STATS SUMMARY

```
total packages: 4
total executable files: 10
total classes: 11
total methods: 207
total executable lines: 1035
```

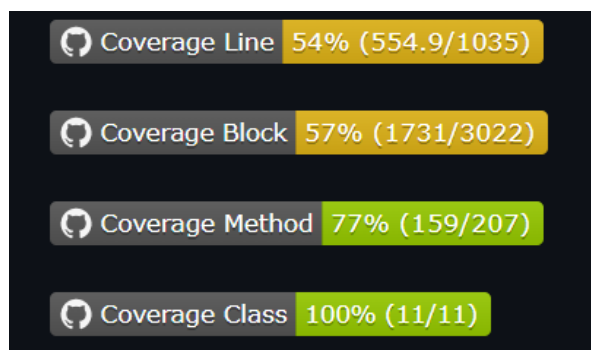
#### COVERAGE BREAKDOWN BY PACKAGE

name	class, %	method, %	block, %	line, %
com.werken.saxpath	100% (5/5)	62% (89/144)	38% (869/2297)	38% (314.9/830)
org.saxpath.helpers	100% (1/1)	100% (4/4)	55% (66/119)	65% (22.9/35)
org.saxpath	100% (4/4)	100% (11/11)	92% (190/206)	88% (59/67)
org.saxpath.conformance	100% (1/1)	100% (48/48)	100% (400/400)	100% (103/103)

Successivamente, solo dopo che le varie sessioni di test avranno terminato la propria esecuzione, verrà avviata la merge, la quale viene effettuata tra due test alla volta.

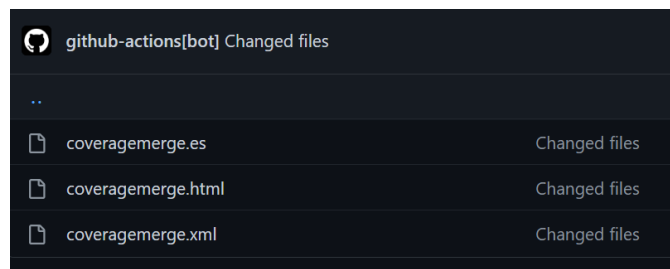
Conclusa anche quest'ultima si passerà alla generazione dei badges ed al controllo sul raggiungimento o meno del valore di copertura settato a priori.

Per controllare facilmente il valore di copertura è possibile spostarsi nel README e controllare i badge che in questo caso sono:



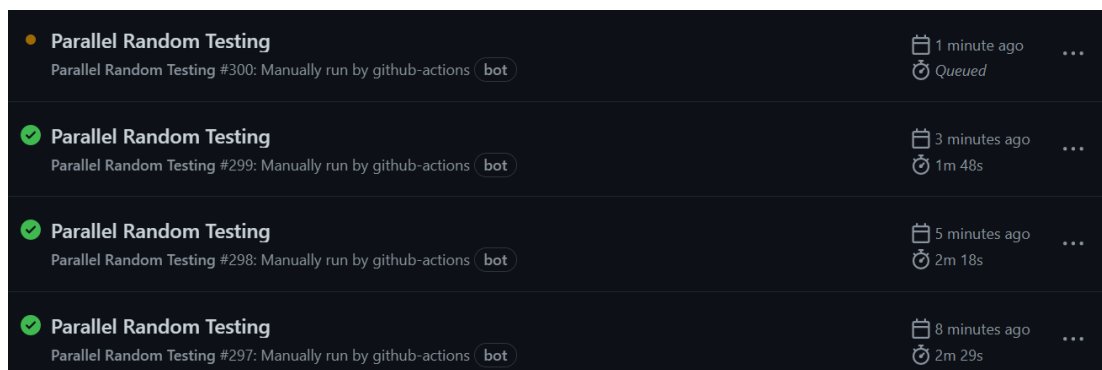
Essendo il valore di "Coverage Line" minore di quello prefissato (56%), sarà effettuato il commit automatico nella cartella merge/coverage:





github-actions[bot] Changed files	
..	
coveragemerge.es	Changed files
coveragemerge.html	Changed files
coveragemerge.xml	Changed files

Inoltre, si può notare come in maniera del tutto automatica, GitHub Actions effettuino nuovamente l'esecuzione del workflow finché non raggiunge il valore di copertura prefissato.



● <b>Parallel Random Testing</b> Parallel Random Testing #300: Manually run by github-actions [bot]	📅 1 minute ago 🕒 <i>Queued</i>	...
✅ <b>Parallel Random Testing</b> Parallel Random Testing #299: Manually run by github-actions [bot]	📅 3 minutes ago 🕒 1m 48s	...
✅ <b>Parallel Random Testing</b> Parallel Random Testing #298: Manually run by github-actions [bot]	📅 5 minutes ago 🕒 2m 18s	...
✅ <b>Parallel Random Testing</b> Parallel Random Testing #297: Manually run by github-actions [bot]	📅 8 minutes ago 🕒 2m 29s	...

### 2.2.1 Output generati

Dopo aver eseguito automaticamente una moltitudine di test case, potremo visualizzare inizialmente due file zippati all'interno della casella Artifacts: **Emma Reports** e **Coverage Merge**.

Come anticipato prima, all'interno di Emma Reports troveremo, dopo aver scaricato ed estratto la cartella compressa, otto cartelle numerate da 0 a 7 contenenti i risultati di ogni sessione di test degli otto nodi della matrice.

Ogni cartella presenterà all'interno un file in formato html riportante i valori di coverage per classi, metodi, blocchi e linee.

In maniera differente invece, all'interno di Coverage Merge, troveremo un file sempre in formato html riportante, questa volta, la coverage totale ottenuta dal merge di tutti i precedenti file di coverage della cartella Emma Reports.

In questo caso, andando a considerare il valore di coverage della prima sessione di test otterremo una cosa del genere:

line, %
53% (552.8/1035)

Dunque essendo il valore di coverage line minore di quello fissato (56%), il bot di GitHub Actions avvierà nuovamente il trigger per ottenere un valore di coverage di almeno 56%. Si può notare invece, che l'ultimo output ottenuto dopo una serie di actions sarà questo:

line, %
56% (569.7/1035)

Può capitare anche che sfortunatamente venga impostato un valore di copertura da raggiungere troppo alto o irraggiungibile mediante tali mezzi. Ciò che accade in questo caso è che l'Action di GitHub continuerà il procedimento dei test finché non saremo noi utilizzatori della repository, a fermarlo manualmente quando ci accorgiamo che i test abbiano un valore di coverage costante nel tempo.

In generale, se non volessimo andare a scaricare continuamente i valori dall'artifact di GitHub, potremo semplicemente visualizzare i cambiamenti del tasso di coverage attraverso di badges settati nel README, oppure recarci nella cartella "merge/coverage" creata mediante commit dal workflow, che conterrà i valori di coverage finali.

## Conclusioni

---

La tecnica del Random Testing è semplice da implementare e poco dispendiosa, in quanto le varie operazioni di testing vengono lasciate al sistema che li elabora, in questo caso GitHub Actions. L'utilità di questi test è quella di trovare facilmente e velocemente errori nel codice, in quanto abbiamo la possibilità, mediante i report in html di visualizzare i valori di copertura di classi, metodi, blocchi e linee, così da sapere dove andare a sistemare il codice o concentrarsi per migliorarlo. Inoltre il random testing permette il testing di un progetto di cui non si conoscano nel profondo tutti i dettagli.

D'altro canto, a contrapporsi a questi vantaggi c'è lo svantaggio del fatto che i test effettuati sono di natura arbitraria: potremmo effettuare test uguali tra di loro anche per lunghi lassi di tempo, non ottenendo un vero e proprio miglioramento sul valore di copertura del codice o di segnalazioni su bug presenti nel codice.

In conclusione, il random testing, soprattutto se effettuato in parallelo, presenta il vantaggio di ottenere numerosi test facilmente e velocemente, in quanto eseguiamo molteplici test in un tempo prefissato mediante "*timelimit*" che potranno tornarci utili per un primo controllo sui valori generali di copertura; lo svantaggio importante però, risulta essere il fatto di avere il rischio di ottenere casi di test uguali tra loro ed una coverage che non migliora anche dopo un lungo lasso di tempo.

Si noti che il metodo di terminazione della generazione dei test case utilizzato è quello t60%, il quale si ottiene quando i test raggiungono il 60% di copertura del codice.

Un possibile sviluppo futuro potrebbe essere quello di controllare il momento in cui i casi di test ottenuti si stabilizzano ad un certo valore senza aumentare più e fermarsi in maniera automatica anziché manualmente come facciamo in questo caso.

***Link Repository: [\[repository\]](#)***