T9-G13: Requisiti sull'esecuzione del robot Randoop

Software Architecture Design

Amodio Anna - M63001455
Bramante Salvatore - M63001529
Cataldo Cristian - M63001462
Guarino Stefano - M63001447

AA: 2022/2023

Indice

Capitolo 1

Documento di Analisi

1.1 Descrizione del Task

L'applicazione deve offrire la funzionalità di generazione dei test su una data classe Java usando il Robot Randoop. Tale funzionalità riceverà in input un file di testo (classe da testare), dovrà lanciare il generatore ed esecutore di Test Randoop, restituendo in output il codice di casi di test generati ed i risultati dell'esecuzione. L'esito dell'esecuzione dovrà essere elaborato in maniera da estrarre da essi le informazioni rilevanti ai fini del gioco (ad esempio, la copertura del codice, etc.).

1.2 Analisi e specifica dei requisiti

1.2.1 Revisione dei Requisiti

I seguenti requisiti sono stati definiti e raffinati durante le tre iterazioni svolte:

1. La funzionalità deve prendere in ingresso il nome del file di testo che corrisponde al codice della classe da testare e il numero massimo di livelli che si desidera generare
2. La funzionalità deve poter chiamare il generatore automatico Randoop su una classe
3. La funzionalità deve poter valutare la saturazione della copertura
4. La funzionalità deve interrompere la generazione di test al raggiungimento della saturazione della copertura
5. La funzionalità deve poter verificare che una classe non generi errori di compilazione
6. La funzionalità deve fornire dei meccanismi per l'identificazione degli errori qualora la classe non esista o generi errori di compilazione
7. La generazione dei test deve essere eseguita offline rispetto al gioco
8. La funzionalità deve garantire che tutte le richieste inoltrate siano servite
9. L'inoltro di una richiesta deve essere asincrono rispetto alla notifica di completamento
10. Le classi da testare devono essere presenti in un percorso locale concordato a priori
11. I test devono essere salvati in un percorso locale concordato a priori
12. La funzionalità deve selezionare i test ed organizzarli opportunamente in cartelle per livelli ottenendo al massimo un certo numero di livelli
13. La funzionalità deve notificare il chiamante al termine della generazione con il numero di livelli generati

14. Il nome della classe deve iniziare con la lettera maiuscola
15. Il nome della classe non deve contenere caratteri speciali
16. Il nome della classe deve essere una stringa non vuota
17. Il numero dei test deve essere un intero positivo diverso da zero
18. Il meccanismo di richiesta e salvataggio dei test avviene in locale

1.2.2 Glossario

- **Copertura**: Metrica di valutazione che considera il numero di linee di codice della classe testata coperte dal test
- **Livelli**: Gruppo di test che vengono presentati al giocatore come sfida da affrontare, associati ad una certa difficoltà. Sono previsti più livelli con difficoltà potenzialmente crescente.

1.2.3 Classificazione dei Requisiti

Requisiti funzionali

ID	Requisito	Origine (n. frase dei requisiti revisionati)
RF01	La funzionalità deve prendere in ingresso il nome del file di testo che corrisponde al codice della classe da testare e il numero massimo di livelli che si desidera generare	1
RF02	La funzionalità deve poter chiamare il generatore automatico Randoop su una classe	2
RF03	La funzionalità deve poter valutare la saturazione della copertura	3
RF04	La funzionalità deve interrompere la generazione di test al raggiungimento della saturazione della copertura	4
RF05	La funzionalità deve poter verificare che una classe non generi errori di compilazione	5
RF06	La funzionalità deve fornire dei meccanismi per l'identificazione degli errori qualora la classe non esista o generi errori di compilazione	6
RF07	La funzionalità deve selezionare i test ed organizzarli opportunamente in cartelle per livelli ottenendo al massimo un certo numero di livelli	12
RF08	La funzionalità deve notificare il chiamante al termine della generazione con il numero di livelli generati	13

Requisiti non funzionali e vincoli

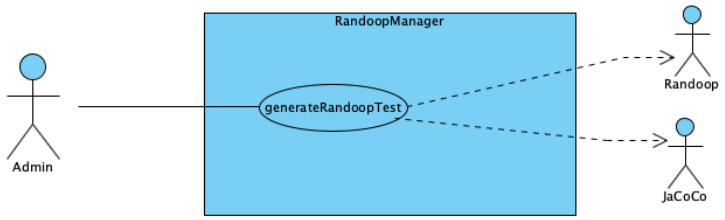
ID	Requisito	Origine (n. frase dei requisiti revisionati)
RNF01	La funzionalità deve garantire che tutte le richieste inoltrate siano servite	8
RNF02	La generazione dei test deve essere eseguita offline rispetto al gioco	7
RNF03	L'inoltro di una richiesta deve essere asincrono rispetto alla notifica di completamento	9
V01	Le classi da testare devono essere presenti in un percorso locale concordato a priori	10
V02	I test devono essere salvati in un percorso locale concordato a priori	11
V03	Il meccanismo di richiesta e salvataggio dei test avviene in locale	18

Requisiti sui Dati

ID	Requisito	Origine (n. frase dei requisiti revisionati)
RD01	Il nome della classe deve iniziare con la lettera maiuscola	14
RD02	Il nome della classe non deve contenere caratteri speciali	15
RD03	Il nome della classe deve essere una stringa non vuota	16
RD04	Il numero dei test deve essere un intero positivo diverso da zero	17

1.2.4 Diagramma dei Casi d'Uso

Il sistema da noi sviluppato presenta un'unica responsabilità, generaTestRandoop, la quale ingloberà tutti i requisiti individuati nella fase di analisi.

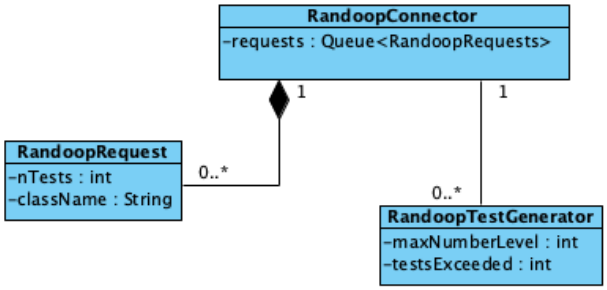


Scenario per il caso d'uso generaTestRandoop

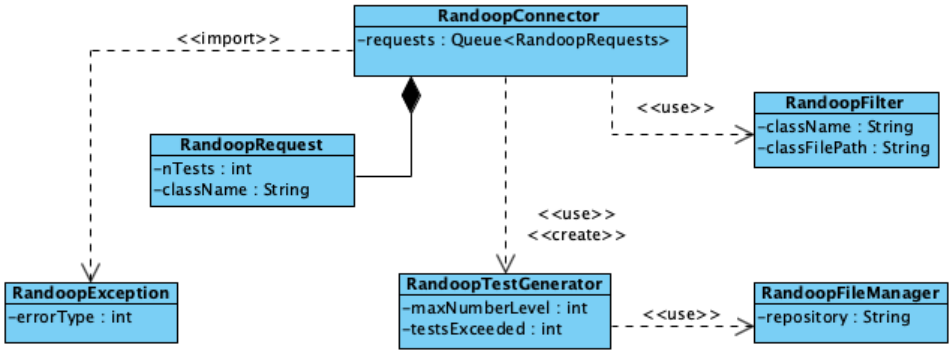
Attore Primario	Admin
Attore Secondario	Randoop, JaCoCo
Descrizione	Vengono generati ed eseguiti dei casi di test sfruttando Randoop a partire da una classe java
Pre-Condizioni	Il sistema conosce il percorso del repository condiviso in cui si trovano le classi di test
Sequenza di eventi principale	1. Il caso d'uso inizia quando Admin chiama la funzionalità con il nome del file di testo in ingresso e il numero massimo di livelli 1.1 Il sistema controlla che esista il file .java con il nome indicato nell'opportuna cartella del repository 1.2 Il sistema controlla che la classe corrispondente al file non generi errori di compilazione 2. il sistema chiama il robot Randoop *2.1 do* a) Il sistema genera test fino a saturazione della copertura b) Il sistema chiama il tool Jacoco per valutare la copertura *while* non è raggiunta saturazione 3. Il sistema organizza in n directory i test generati dove n è minore o uguale al numero massimo di livelli 4. Il sistema salva le directory generate in un percorso predefinito 5. Il caso d'uso termina
Post-Condizioni	Il sistema memorizza i file relativi ai test generati nel repository condiviso
Casi d'uso correlati	-
Sequenza di eventi alternativi	1.1.a Se il file .java non esiste, il caso d'uso termina 1.2.a Se il file genera errori di compilazione, il caso d'uso termina

1.2.5 Diagramma delle Classi

Class diagramm di Analisi

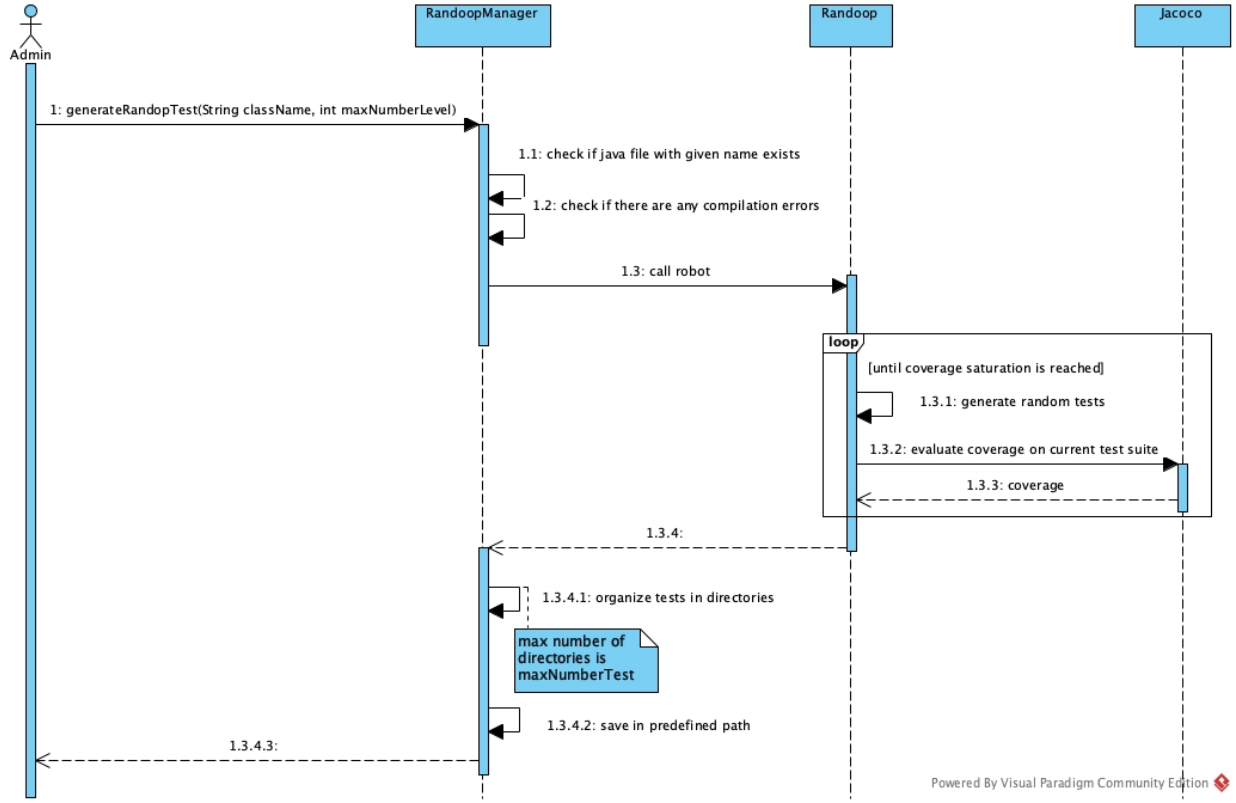


System Domain Model



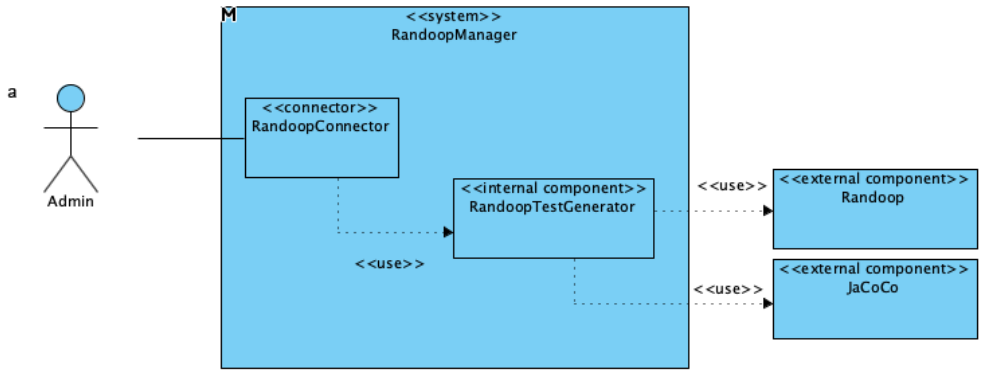
1.2.6 Sequence diagram di Analisi

Il sequence diagram mostra un modello di analisi dell'interazione tra attori e sistema.



1.2.7 Context Diagram

Descriviamo il contesto di esecuzione con il seguente diagramma di contesto:



1.3 Tecnologie utilizzate

1.3.1 Randoop

Randoop è uno strumento automatico che permette di generare test di unità casuali a partire da una classe java. I test restituiti sono in formato JUnit; essi possono essere ripetuti utilizzando l'esito della prima esecuzione come oracolo. Si può eseguire Randoop da linea di comando. Per generare casualmente test è necessario passare allo strumento il file .class della classe da testare; vengono restituiti in output dei file che riportano i test, raggruppati in base all'esito. È possibile specificare come parametro il tempo da dedicare alla generazione automatica dei test. Randoop non offre uno strumento dedicato alla valutazione della copertura, pertanto per valutarla è necessario eseguire i test con un tool specifico.
Il comando è il seguente:

```
1 java -classpath <path_to_randoop_jar:path_to_class_file>
     randoop.main.Main gentests --testclass=<Classname>
2 --time-limit=<time>
3 --regression-test-basename=<nomeRegressionTest>
4 --error-test-basename=<nomeErrorTest>
5 --randomseed=<seed>
6 --junit-output-dir=<destination_directory>
```

1.3.2 JaCoCo

JaCoCo è uno strumento per valutare in maniera automatica la copertura delle linee di codice per test Java. Esso viene incluso come plugin e genera automaticamente i report riguardanti la copertura ogni volta che i test vengono eseguiti. Per eseguire il plugin è sufficiente utilizzare il comando `mvn test`.

1.3.3 Java

Linguaggio di programmazione utilizzato per l'implementazione della funzionalità. Utilizziamo la versione Java 11.

1.3.4 Maven

Strumento di gestione di progetti software basati su Java e di build automation.

1.3.5 Linguaggio Bash

Linguaggio shell di Linux utilizzato per la realizzazione degli script di configurazione.

Capitolo 2

Documento di Progettazione

In questo capitolo analizzeremo nel dettaglio progettazione e implementazione di due versioni alternative della funzionalità. Queste due versioni differiscono per l'implementazione dell'algoritmo di generazione dei test con Randoop, come discusso in fase di installazione del progetto. Analizzeremo le differenze tra le due versione nel diagramma delle classi e nei diagrammi comportamentali. Le due versioni risultano uguali in termini di interfaccia e quindi il metodo da chiamare è lo stesso, ma differiscono per le seguenti scelte implementative:

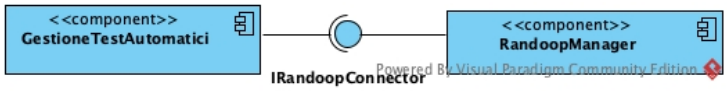
- Versione 1 (branch main del repository): l'algoritmo prevede che i livelli siano generati iterativamente (fino alla saturazione della copertura) e che ad ogni iterazione il livello generato presenti un numero di test suite fissate. Se si raggiunge il massimo numero di livelli viene creato un livello extra in cui vengono salvate tutte le test-suite in eccesso. In tal caso al termine dell'algoritmo verrà campionato casualmente un sottoinsieme delle test-suite generate, e queste saranno distribuite nei livelli previsti. Può accadere che vengono generati meno livelli di quelli richiesti.

- Versione 2 (branch main-versione-2): le test-suite vengono generate iterativamente fino alla saturazione e al termine vengono organizzate in livelli. Vengono quindi generati un numero di livelli pari al numero di livelli richiesti.

2.1 Strutture C&C

2.1.1 Component Diagram

Component Diagram di Progetto

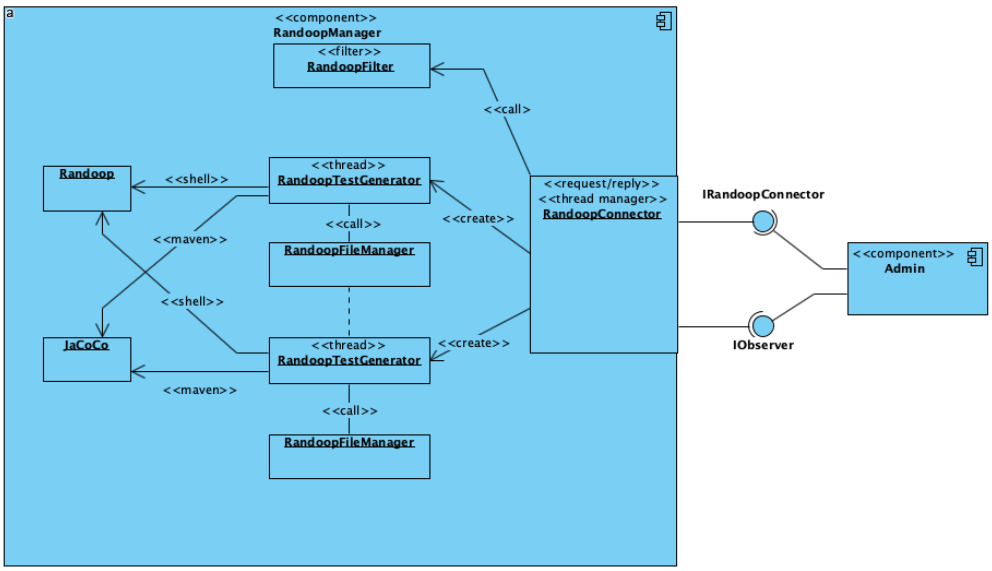


Dal diagramma in figura possiamo vedere che abbiamo due componenti e un'interfaccia; in particolare:

- `GestioneTestAutomatici`: utilizza l'interfaccia `IRandoopConnector`
- `RandoopManager`: implementa l'interfaccia `IRandoopConnector`

Component Diagram di Implementazione

Il seguente component diagram mostra in dettaglio i componenti interni dell'architettura.



Legenda

- <<`shell`>>: per l'interfacciamento con il componente Randoop è prevista un API richiamabile da shell; pertanto il connettore è la shell stessa.
- <<`maven`>>: per l'interfacciamento con il componente JaCoCo è previsto l'utilizzo di un plugin maven che funge da connettore.
- <<`thread manager`>>: il componente che possiede tale stereotipo è un particolare connettore che gestisce la concorrenza.

A partire da tale vista possiamo descrivere e motivare due principali scelte architetturali.
In primo luogo, abbiamo deciso di consentire la gestione di richieste multiple; osserviamo infatti in figura che è possibile generare più thread (`RandoopTestGenerator`) per servire contemporaneamente richieste differenti. Nella versione realizzata i thread sono 4: è possibile però estendere se si desidera un maggior grado di multithreading.
Inoltre, osserviamo che il componente implementato utilizza un'interfaccia `IObserver`: come dettagliato in seguito, è stato deciso di adottare il DP Observer in versione sincrona per la gestione della notifica che viene inviata al completamento delle operazioni.

2.1.2 Specifica Interfaccia

- **Identità dell'interfaccia**
  - L'interfaccia si presenta come un'interfaccia Java: `IRandoopConnector`
- **Risorse Presentate**
  - Metodo `generateRandoopTest()`
    - Sintassi: `void generateRandoopTest (String className, int maxNumberLevel, IObserver o) throws RandoopException`
    - Semantica:
      - `String className` è una che stringa che rappresenta il nome della classe su cui eseguire i test
      - `int maxNumberLevel` è il numero massimo di livelli da generare per quella classe
      - `IObserver o` è il riferimento all'oggetto che implementa l'interfaccia `IObserver` il quale vuole essere notificato del completamento delle operazioni
    - Restrizioni di utilizzo:
      - Il file `AUTClass.java` relativo alla classe deve essere presente nella cartella `AUTName/AUTSourceCode/` all'interno del repository condiviso
      - La classe non deve generare errori di compilazione
  - Eccezione `RandoopException()`
    - Sintassi: `RandoopException (String message, int errorType)`
    - Semantica:
      - `String message` rappresenta il messaggio relativo all'eccezione
      - `int errorType` rappresenta la tipologia di errore verificatosi, e può essere prelevato tramite il metodo di `get()`
- **Tipi di dato e costanti**
  - Le costanti seguenti sono utilizzate per la gestione degli errori
    - `RandoopException.INVALID_NTEST: int`
    - `RandoopException.INVALID_CLASSNAME: int`

- `RandoopException.DIR_ERROR : int`
- `RandoopException.CLASS_NOT_FOUND: int`
- `RandoopException.CLASS_NO_COMPILE : int`

- **Gestione degli errori**
  - Le possibili condizioni di errore da gestire sono:
    - Se il numero massimo di test richiesti è negativo o nullo
    - Se il nome della classe non è valido, ovvero comprende caratteri alfanumerici, è vuoto o non inizia per lettera maiuscola
    - Se la cartella con il nome della classe non è presente all'interno delrepository condiviso
    - Se la classe non è presente all'interno della cartella `AUTName/AUTSourceCode/`
    - Se la classe è stata trovata ma contiene degli errori che non permettono la compilazione

    Questi vengono gestiti tramite il lancio di un'eccezione di tipo `RandoopException`. L'eccezione è caratterizzata da uno specifico messaggio di errore e un codice che identifica il tipo di errore verificatosi. Esso può esser ottenuto tramite il metodo `int getERROR_TYPE()`. L'intero restituito corrisponderà ad una delle tre costanti indicate.
  - Nel caso in cui sia impossibile generare test per un motivo diverso da quelli previsti, sarà notificato 0 come numero di test generati.

- **Variabilità**
  - L'interfaccia proposta non presenta punti di variabilità o proprietà configurabili

- **Attributi di qualità e caratteristiche**
  - Performance: al fine di migliorare la performance a runtime si è previsto un sistema di generazione parallelo (fino ad un certo grado di parallelismo) dei test su richieste differenti. Inoltre, si è previsto che il chiamante venga notificato al termine della generazione, vista la latenza causata dall'algoritmo di generazione dei test con Randoop

- Reliability: l'esecuzione della richiesta è garantita dalla gestione tramite una coda delle richieste che non possono essere servite immediatamente per mancata disponibilità di risorse

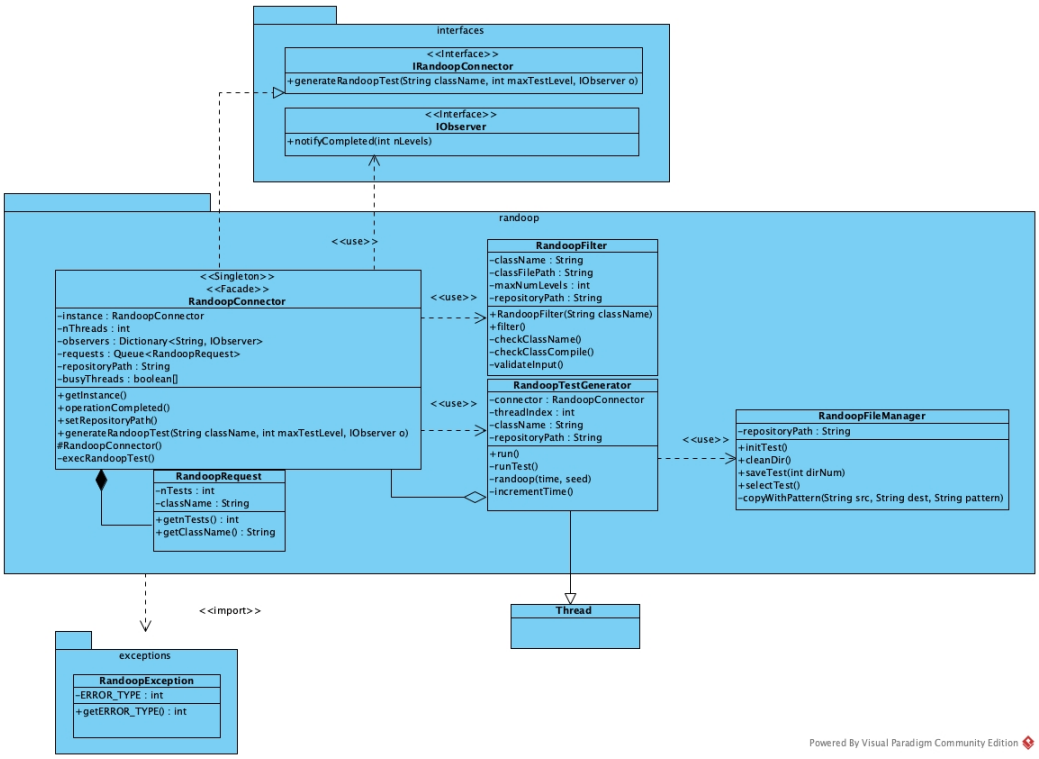
- **Risorse richieste**
  - È necessario che chi desidera essere notificato del completamento della generazione, tipicamente chi chiama il metodo, implementi l'interfaccia `IObserver`.
    In particolare, il metodo `notifyCompleted()`:
    - Sintassi: `void notifyCompleted (int nLevels)`
    - Semantica:
      - `int nLevels` identifica il numero di livelli di difficoltà generati per la classe

- **Logica e problemi di progettazione**
  - È stato deciso di realizzare un'interfaccia Java per diminuire l'accoppiamento con altri elementi del sistema e per facilitare l'integrazione

- **Guida di utilizzo**
  - Per ottenere un riferimento alla classe che implementa l'interfaccia è necessario chiamare il metodo statico `RandoopConnector.getInstance()`. La funzionalità deve essere chiamata sul riferimento ottenuto con il nome della classe per la quale si vogliono generare test, con il numero massimo di livelli richiesti e con il riferimento all'oggetto che desidera essere notificato del completamento della generazione di test. Il chiamante del metodo deve gestire i casi di eccezione mediante i diversi codici di errore dell'eccezione `RandoopException`
    Se si desidera cambiare il percorso di salvataggio dei test è necessario chiamare il metodo `RandoopConnector.setRepositoryPath()`

2.2 Strutture dei Moduli

2.2.1 Package Diagram: versione 1

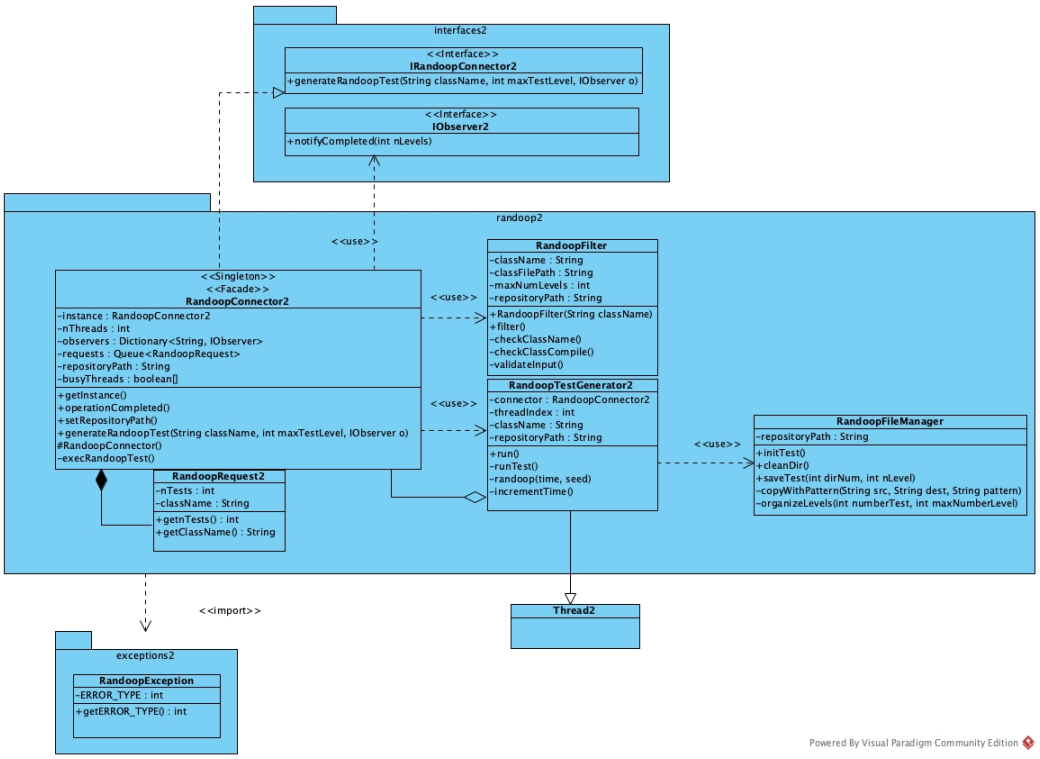


Per tale struttura abbiamo deciso ti utilizzare un'unica vista realizzata come combinazione di più viste, quali vista di decomposizione, vista d'uso e vista delle classi. In particolare tale vista composita ci fornisce i dettagli sull'implementazione del componente `RandoopManager`, che risulta essere suddiviso in tre package:

1. `interfaces`: package relativo alle interfacce esposte dal componente.
2. `randoop`: package relativo alla logica del componente
3. `exception`: package relativo alle eccezioni per la gestione degli errori

Osserviamo che il package `randoop`, che realizza la logica del componente, presenta una classe *Façade* implementata come *Singleton* che ha responsabilità di gestione e coordinamento delle richieste. Tale classe realizza l'interfaccia esposta all'esterno.

2.2.2 Package Diagram: versione 2

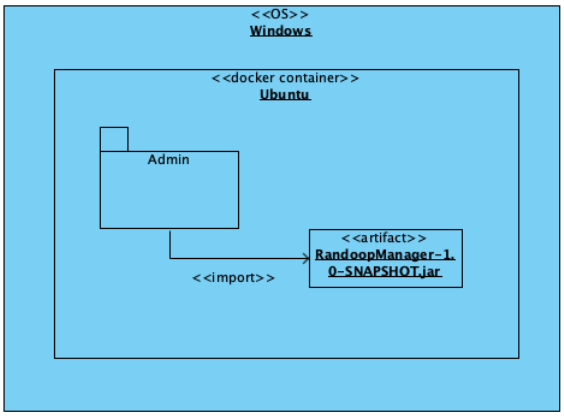


Possiamo osservare che il package diagram per la versione 2 differisce dalla versione 1 per due aspetti:

- viene aggiunto `organizeLevel()` al posto di `selectTest()`
- cambiano i parametri del metodo `saveTests()`.

2.3 Strutture di Allocazione

2.3.1 Deployment Diagram

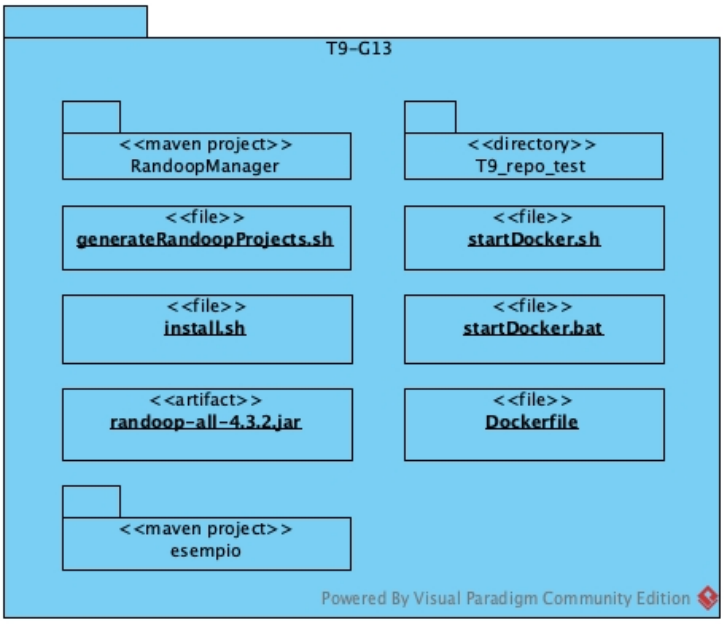


Il diagramma descrive il contesto software di esecuzione. Al fine di rendere la soluzione os-indipendent, abbiamo previsto l'utilizzo di un container docker Linux con Java 11 su cui sarà eseguito l'intero applicativo. La funzionalità è esportata come archivio jar che dovrà essere importato nel progetto che realizza l'effettivo servizio relativo all'admin. Come già descritto, la soluzione presentata prevede un metodo di interfaccia che dovrà essere chiamato in locale.

2.3.2 Installation View

Struttura Repository

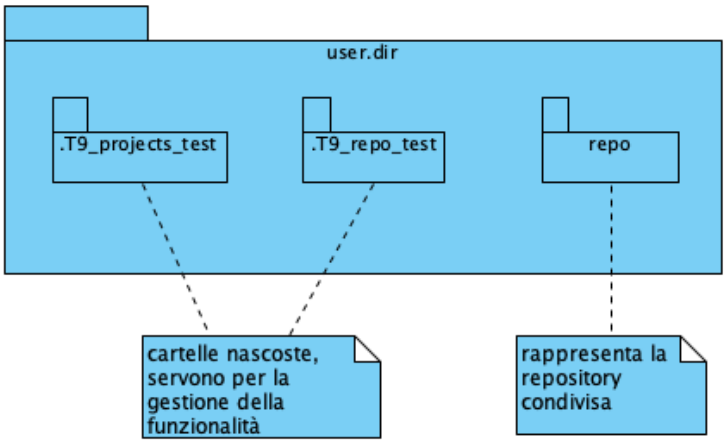
La struttura della repository è identica per entrambi i branch ed è rappresentata in figura:



Quando si effettua il clone della repository la directory si presenterà come in figura. Se la macchina target è Linux per l'installazione è sufficiente da shell far partire lo script `install.sh`; altrimenti si esegue lo script `startDocker` come amministratore, che include la creazione del docker ed una chiamata allo script. Entrambi gli script saranno descritti in seguito.

Struttura Directory User

In seguito all'installazione la user directory (interna al container) sarà modificata come in figura.



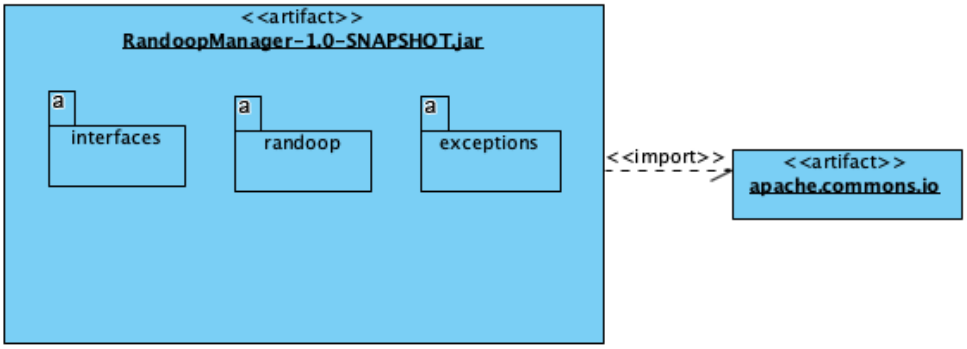
Verranno aggiunte le directory:

- `T9_projects_test`: directory che contiene i progetti necessari per l'esecuzione dell'algoritmo di generazione dei test Randoop.
- `T9_repo_test`: directory che contiene la repository fittizia per la realizzazione del piano di test.

Le directory specificate risultano essere nascoste all'utente, in quanto sono esclusivamente parte della logica di gestione.
Si prevede inoltre la presenza della directory condivisa repo in cui andranno inserite le directory per le classi sui cui verranno generati i test.

RandoopManager jar

Come già analizzato nel deployment diagram verrà generato un archivio jar che potrà essere importato nel servizio che implementa l'utilizzatore. Tale archivio si presenta come nella seguente figura:



La dipendenza dal package `apache.commons.io` viene automaticamente importata.

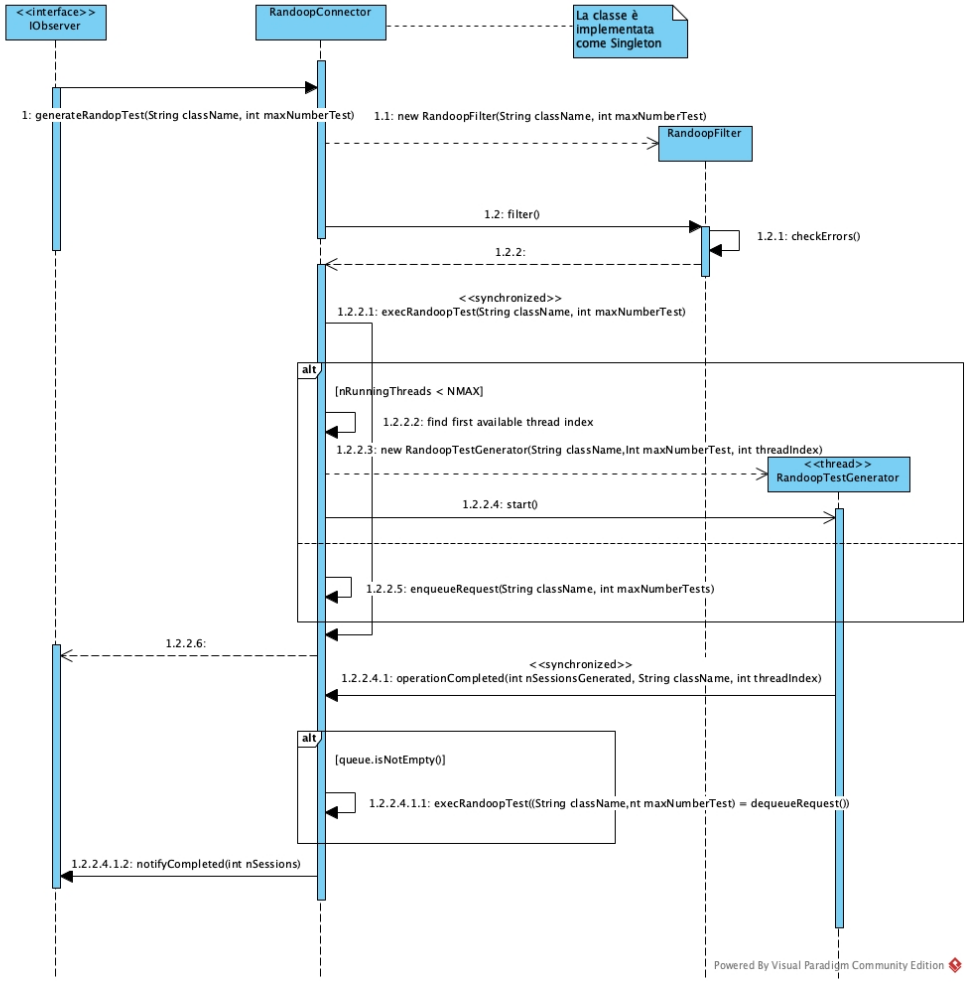
2.4 Vista Comportamentale

In tale vista andremo ad analizzare in primo luogo l'implementazione dei metodi fondamentali per la realizzazione della funzionalità. In seconda battuta analizzeremo l'implementazione degli script di installazione.

2.4.1 Funzionalità: Sequence Diagrams e Activity Diagrams

I seguenti diagrammi ci consentiranno di comprendere nel dettaglio la progettazione e l'implementazione della responsabilità `generateRandoopTest` offerta dall'interfaccia `IRandoopConnector`. Per facilitare la comprensione della logica della funzionalità abbiamo scelto di selezionare l'ordine dei diagrammi seguendo una logica di refinment, rispettivamente per le due versioni.

Sequence diagram: generateRandoopTest



Legenda

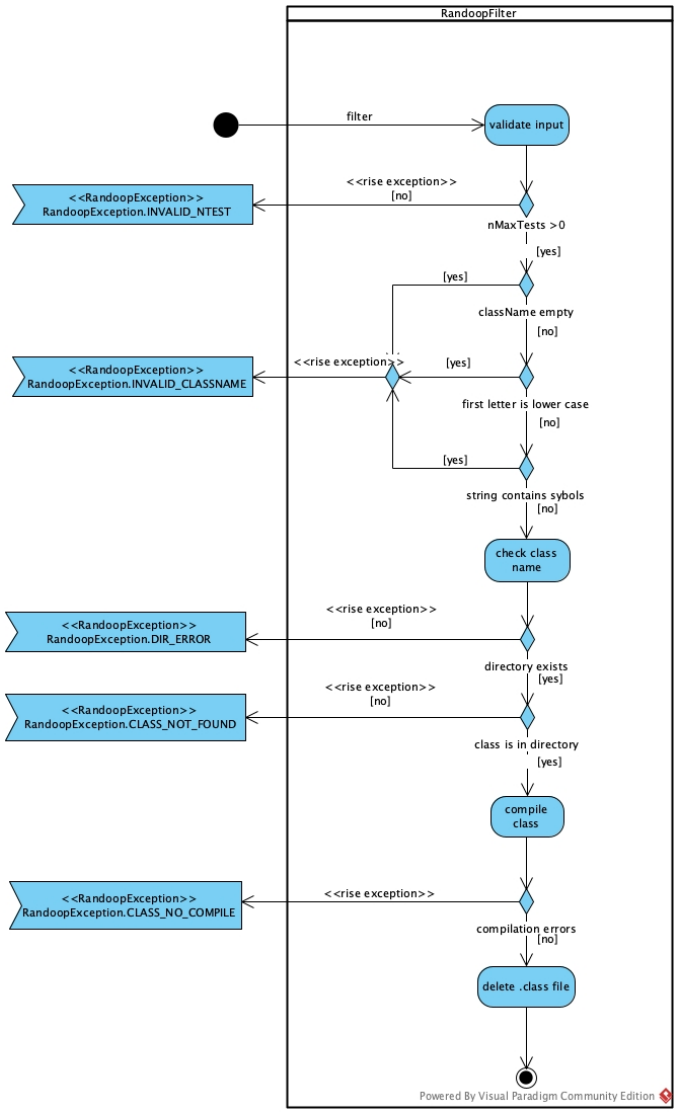
- <<`synchronized`>>: tale stereotipo indica che il metodo è invocato in mutua esclusione.

Tale digramma rappresenta un caso d'uso favorevole in cui non vengono individuati errori dal filtro e pertanto non vengono generate eccezioni. In caso di rilevazione di errori viene subito passato il flusso di controllo al chiamante

(`IObserver`).
Questo diagramma fornisce dettagli sulla coordinazione degli oggetti chiamati per realizzare del servizio di generazione automatica dei test.
Il diagramma mostra, come descritto in precedenza, la presenza della logica multithread e del sistema di notifica: osserviamo che all'arrivo di una nuova richiesta si prevede di controllare il numero di thread già attivi: se questi superano il numero massimo, si accoda una richiesta che verrà servita al completamento di una di quelle in corso.
Inoltre si osserva nel diagramma l'implementazione del pattern Observer. La logica di registrazione è implementata nella stessa funzione `generateRandoopTest`; all'atto di una richiesta, viene passato un riferimento all'oggetto `IObserver` che desidera essere notificato. Al termine dell'esecuzione si invoca il metodo `notifyCompleted(int nLevels)` che corrisponde alla notifica sincrona.

Activity diagram: filter

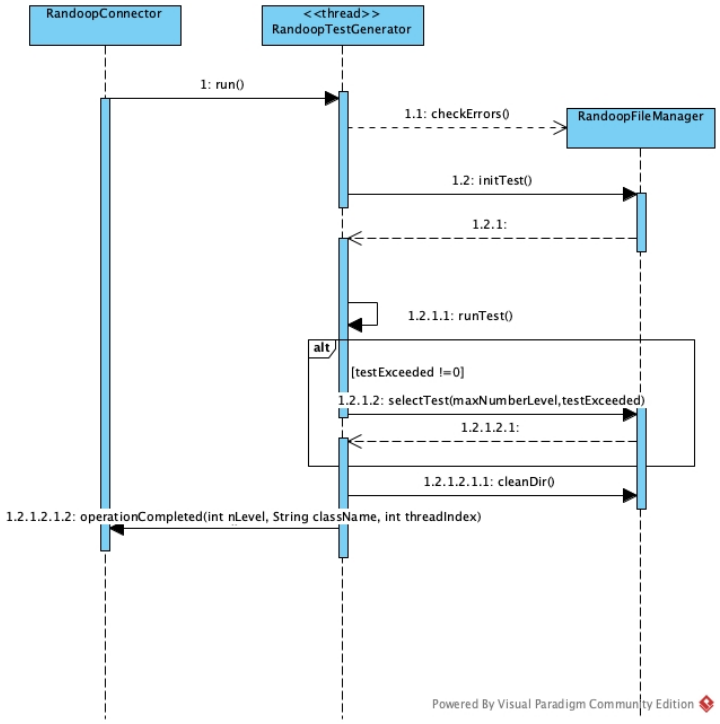
L'activity diagram seguente mostra il flusso di controllo del metodo `filter()` della classe `RandoopFilter` che gestisce il lancio delle eccezioni di tipo `RandoopException`. Come descritto esse possono essere di differenti tipologie e sono identificate da un codice di errore.



Mostriamo ora di seguito i diagrammi che descrivono l'implementazione della prima versione dell'algoritmo.

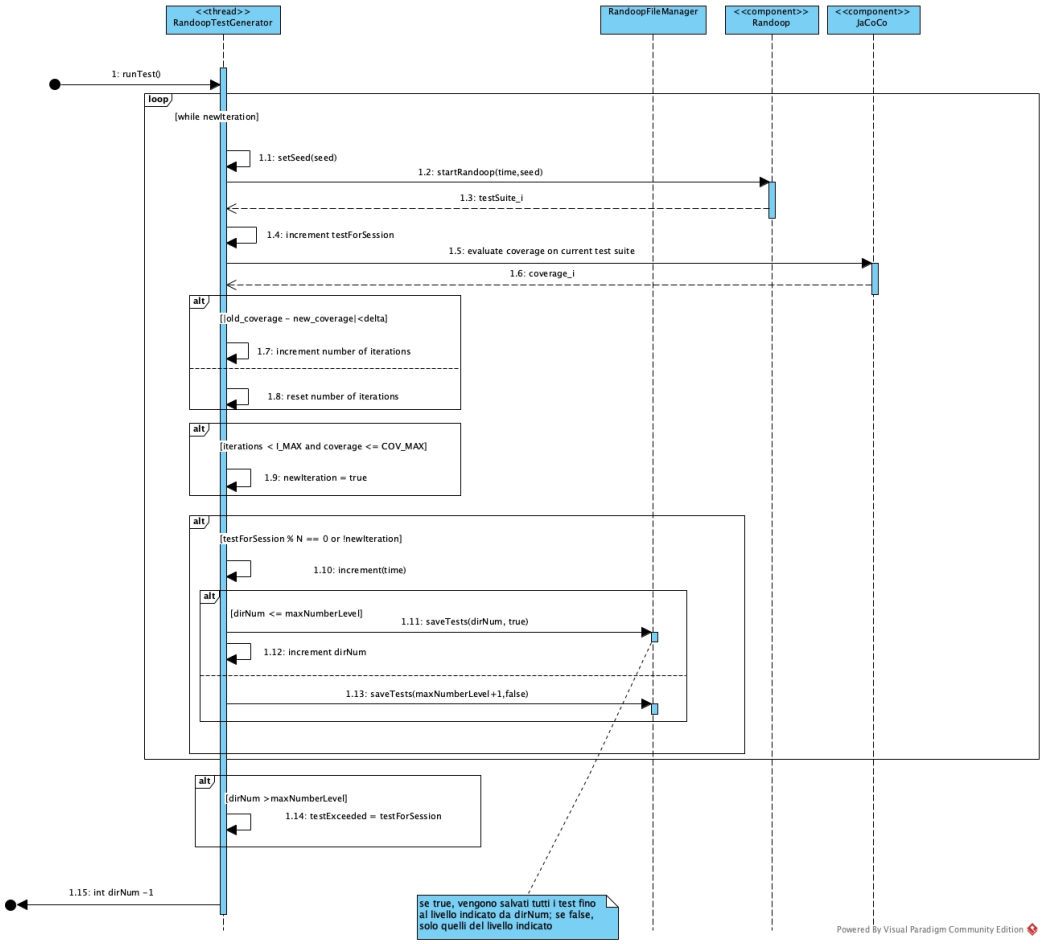
Sequence diagram: run - versione 1

Il seguente diagramma mostra il meccanismo che si innesca in seguito alla chiamata del metodo `run()` della classe `RandoopTestGenerator`.



Sequence diagram: generazioneTest - versione 1

Il diagramma mostra l'algoritmo di generazione dei test.



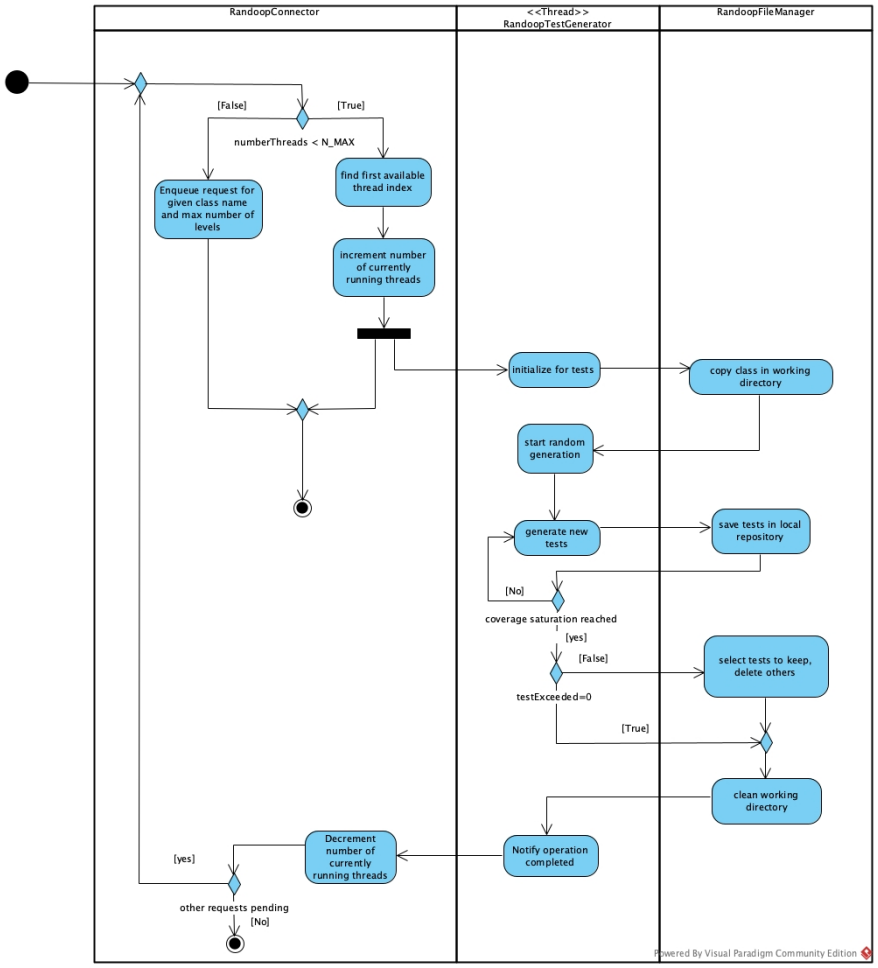
Sono stati scelti una serie di parametri di configurazione:

- `I_MAX`
- `maxTestForSession`
- `DELTA`

I livelli sono generati iterativamente (fino alla saturazione della copertura) e ad ogni iterazione il livello generato presenterà un numero di test suite fissate pari a `maxTestForSession`. Se si raggiunge il massimo numero di livelli viene creato un livello extra in cui vengono salvati tutti le test-suite in eccesso; tali test suite saranno poi distribuite opportunamente (la logica è spiegata successivamente)

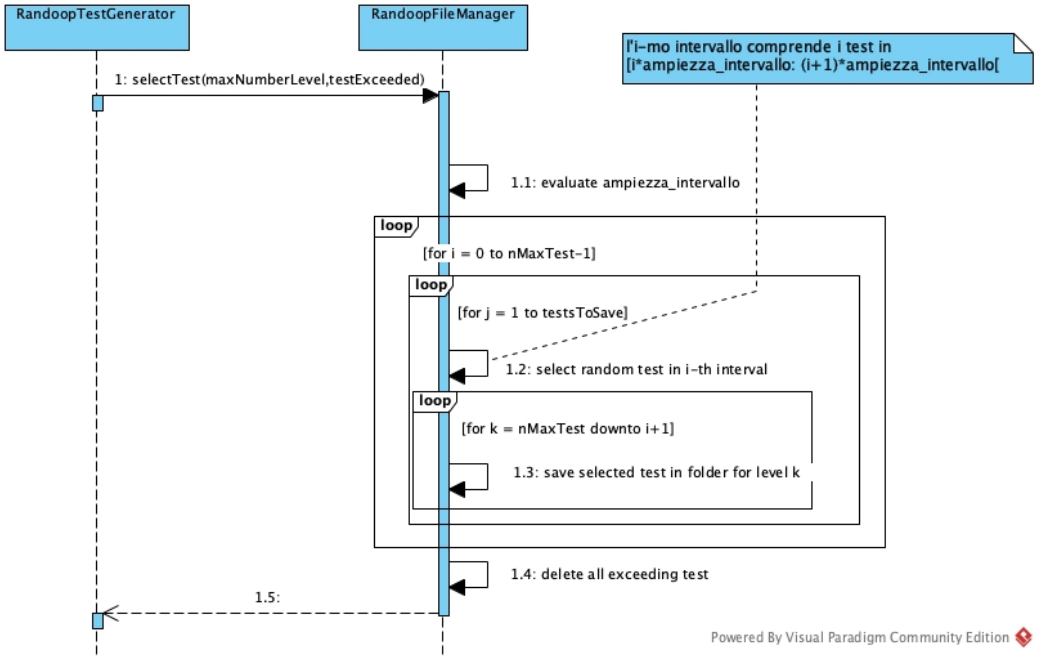
Activity diagram: execRandoopTest - versione 1

Il seguente activity diagram mostra il flusso di controllo nella funzione `execRandoopTest()` nella prima versione dell'algortimo.



Sequence diagram: selectTest - versione 1

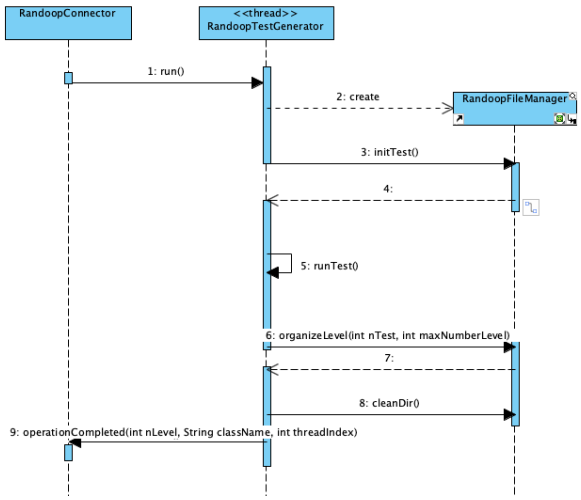
Il diagramma descrive la logica di selezione dei test. Il metodo `selectTest()` consente di campionare casualmente tra i test in eccesso delle test suite che poi verranno distribuiti nelle cartelle dei livelli generati in precedenza.



Mostriamo ora di seguito i diagrammi che descrivono l'implementazione della seconda versione dell'algoritmo, presentando le differenze.

Sequence diagram: run - versione 2

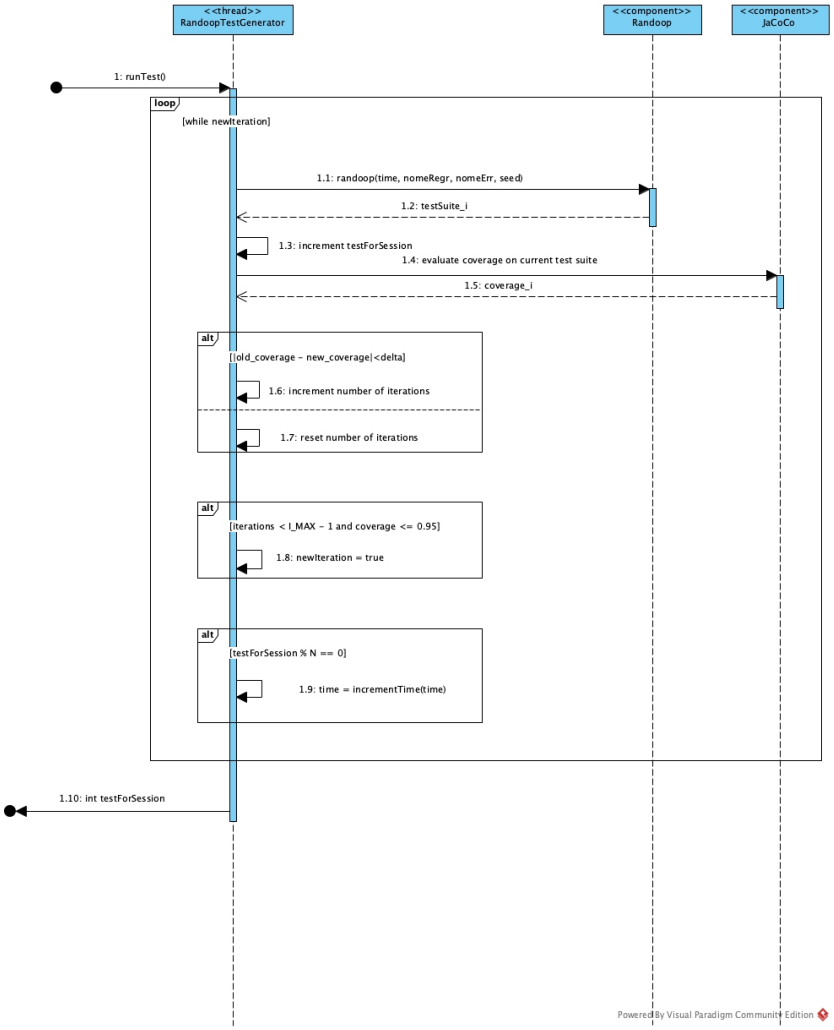
Il seguente diagramma mostra il meccanismo che si innesca in seguito alla chiamata del metodo `run()` della classe `RandoopTestGenerator`.



I principali cambiamenti rispetto alla versione 1 sono:

- il metodo privato `runTest`() restituisce il numero di test-suite generate e non il numero di livelli generate, in modo da poter effettuare la ridistribuzione;
- non viene gestita la logica dei test in eccesso, in quanto non prevista in questa logica.

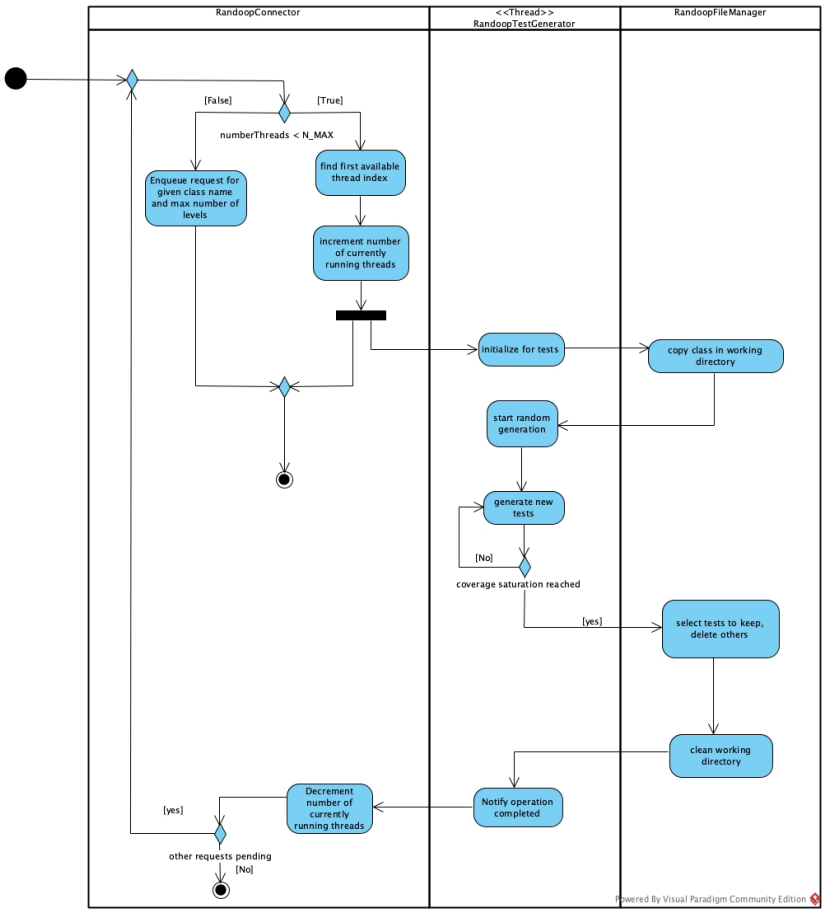
Sequence diagram: generazioneTest - versione 2



Il diagramma mostra la soluzione 2 per l'algoritmo di generazione dei test.
In tale soluzione le test-suite vengono generate iterativamente fino al raggiungimento della saturazione.

Activity diagram: execRandoopTest - versione 2

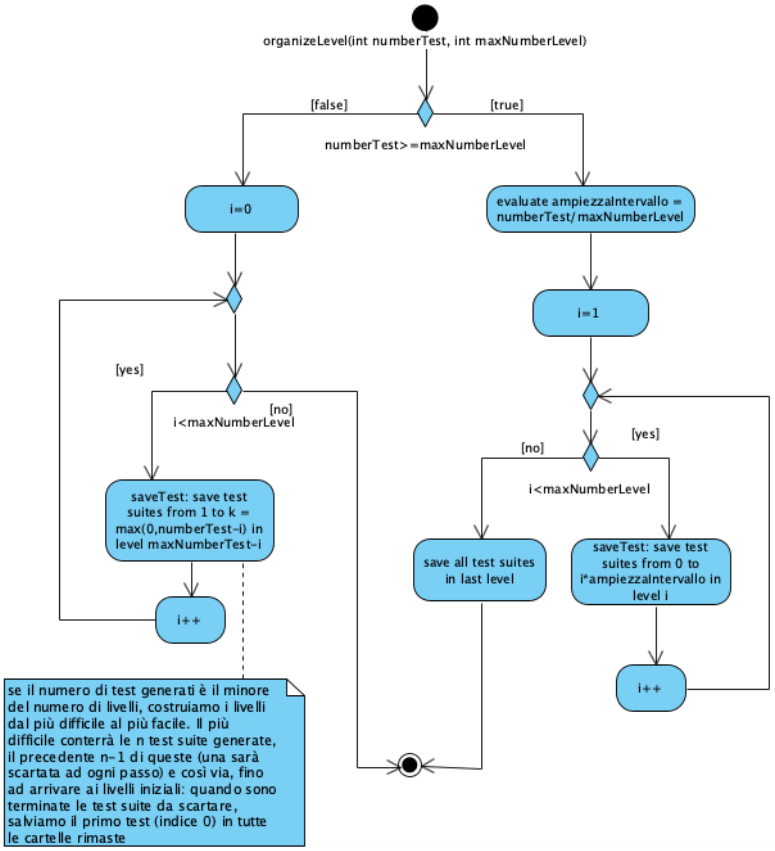
Il metodo execRandoopTest è descritto nel seguente activity diagram:



Rispetto alla versione precedente, in questo caso non vengono salvati i test ad ogni sessione, ma solo alla fine; inoltre ancora una volta non c'è la gestione dei test in eccesso.

Sequence diagram: organizeLevel - versione 2

Tale metodo consente di distribuire i test generati dall'algoritmo nelle directory relative ai livelli da generare. Esso non è presente nella prima versione poichè non necessario; si sostiuisce al metodo `selectTest()`, che in questa versione non è utilizzato.

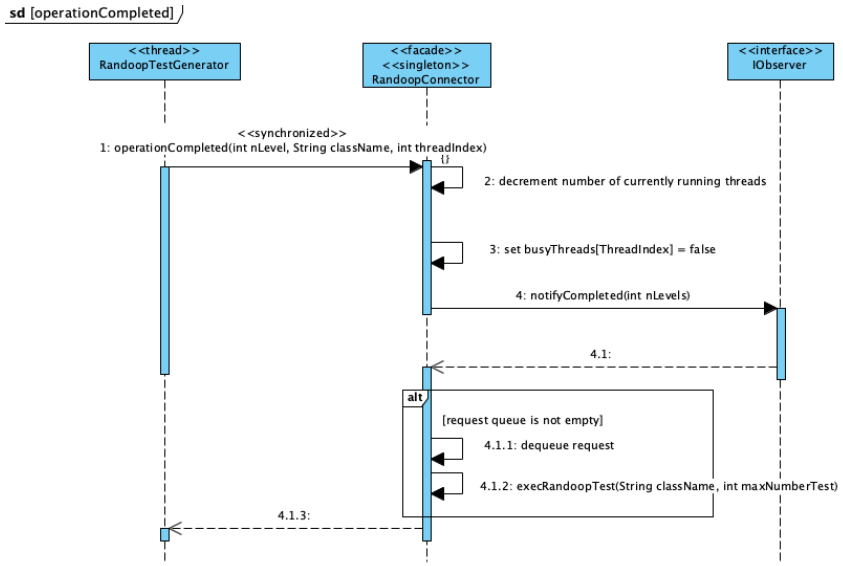


Il metodo campiona e salva le test suite in modo da costruire il numero di livelli richiesto.

Di seguito, mostriamo la descrizione di alcuni aspetti comuni ad entrambe le versioni.

Sequence diagram: operationCompleted

Nel seguente diagramma si descrive in dettaglio il meccanismo di notifica asincrono tipico del DP Observer.

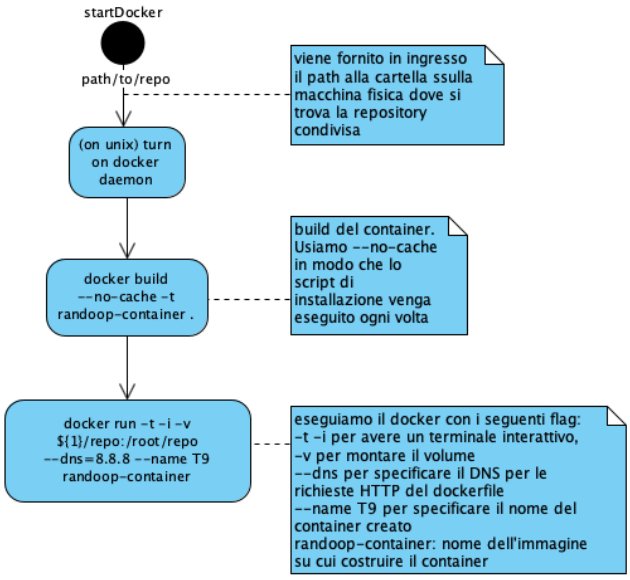


La classe `RandoopConnector` notificherà l'Observer registrato al completamento delle operazioni.

2.4.2 Script di Installazione: Activity Diagram

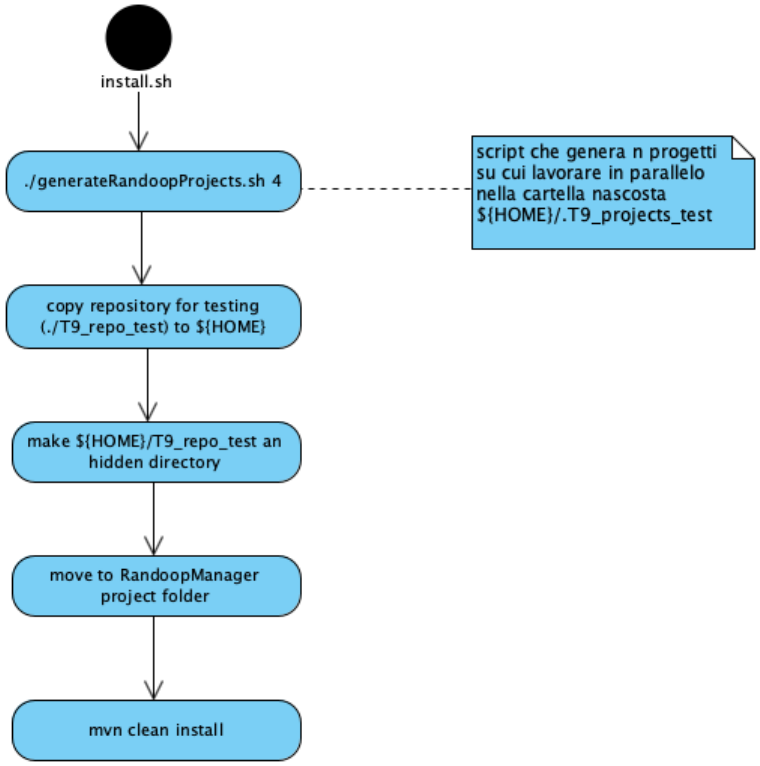
startDocker

Questo script è stato realizzato in due versioni, sia per essere utilizzato in windows (.bat) che in sistemi operativi Unix-based (.sh).



install.sh

Tale script ci consente di configurare l'ambiente per il corretto funzionamento della funzionalità.



Osserviamo che lo script utilizza un ulteriore script per la creazione dei progetti come supporto all'algortimo di generazione dei test.

Capitolo 3

Testing

Di seguito descriviamo il piano di test realizzato per la funzionalità offerta (`generateRandoopTest()`).
Il piano di test di seguito riguarda i test che possono essere condotti con logica *assertion-based*; sono stati effettuati altri test oltre a quelli sistematicamente descritti, in particolare per l'algoritmo e il gestore del filesystem, ma questi necessitavano di un oracolo umano, pertanto sono stati condotti direttamente e non sistematizzati in test suite JUnit. Inoltre abbiamo deciso di seguire un approccio *bottom-up*: in tal modo non è stato necessario l'utilizzo di oggetti Mock, tranne nell'ultimo caso in cui è stato necessario realizzare un oggetto mock della classe `IObserver`.

3.1 Test di Unità

Test suite: FilterTest

Abbiamo innanzitutto testato in unità la classe `RandoopFilter`, con una test suite realizzata come segue:

ID	DESCRIZIONE	PRE-CONDIZIONI	INPUT	OUTPUT ATTESO	POST-CONDIZIONI
11	Test non valido: className stringa vuota	PC01	className="" maxNumberLevel=10	RandoopException: INVALID_CLASSNAME	NULL
12	Test non valido: className stringa con caratteri speciali	PC01	className="Calcolat.rice" maxNumberLevel=10	RandoopException: INVALID_CLASSNAME	NULL
13	Test non valido: className stringa con inziale minuscola	PC01	className="calcolatrice" maxNumberLevel=10	RandoopException: INVALID_CLASSNAME	NULL
14	Test non valido: maxNumberLevel numero negativo	PC01	className="Calcolatrice" maxNumberLevel=-1	RandoopException: INVALID_NTEST	NULL
15	Test non valido: maxNumberLevel uguale a zero	PC01	className="Calcolatrice" maxNumberLevel=0	RandoopException: INVALID_NTEST	NULL
16	Test non valido: la classe non compila	PC01	className="Esempio" maxNumberLevel=5	RandoopException: CLASS_NO_COMPILE	NULL
17	Test non valido: la cartella per la classe non esiste	- PC01 - Nella repository dei test non esiste la cartella Calcolatore	className="Calcolatore" maxNumberLevel=2	RandoopException: DIR_ERROR	NULL
18	Test non valido: la cartella per la classe esiste ma non c'è la classe	- PC01 - Nella repository dei test esiste la cartella	className="Calendario" maxNumberLevel=2	RandoopException: CLASS_NOT_FOUND	NULL

3.2 Testing black box

Abbiamo proceduto quindi con il testing black-box, in modo da testare il raggiungimento dei requisiti.

Classi di Equivalenza

Test suite: EquivalenceClassTesting

In primo luogo sono state individuate le seguenti classi di equivalenza per i due valori di input:

CLASSNAME	MAXNUMBERLEVEL
- Stringa alfanumerica non vuota con inziale maiuscola - Stringa con inziale minuscola **[ERRORE]** - Stringa vuota **[ERRORE]** - Stringa con caratteri speciali **[ERRORE]**	- Intero positivo diverso da zero - Intero = 0 **[ERRORE]** - Intero < 0 **[ERRORE]**

Gli input tra di loro sono fortemente disaccoppiati: per tale motivo abbiamo scelto di realizzare una copertura BCC (Base Choice Coverage).
Abbiamo quindi individuato una test-suite di 6 test in cui ogni classe di equivalenza è coperta da almeno un caso di test.
Da notare come sono state individuate delle precondizioni condivise che vengono raggruppate sotto la sigla **PC01** per una migliore leggibilità:

- Deve esistere il riferimento `RandoopConnector randoopConnector`;
- La repository dei test deve esistere ed avere la struttura convenzionale;
- Il path relativo alla repository deve essere settato opportunamente come repository di test;

La descrizione della test suite è la seguente:

ID	DESCRIZIONE	CLASSI DI EQUIVALENZA COPERTE	PRE-CONDIZIONI	INPUT	OUTPUT ATTESO	POST-CONDIZIONI
1	Test non valido: className stringa vuota	**className**: Stringa vuota **[ERRORE] maxNumberLevel**: valido	PC01	className="" maxNumberLevel=10	RandoopException: INVALID_CLASSNAME	Non vengono generati test
2	Test non valido: className stringa con caratteri speciali	**className**: Stringa con caratteri speciali **[ERRORE] maxNumberLevel**: valido	PC01	className=" Calcolat.rice" maxNumberLevel=10	RandoopException: INVALID_CLASSNAME	Non vengono generati test
3	Test non valido: className stringa con inziale minuscola	**className**: Stringa con inziale minuscola **[ERRORE] maxNumberLevel**: valido	PC01	className=" calcolatrice" maxNumberLevel=10	RandoopException: INVALID_CLASSNAME	Non vengono generati test
4	Test non valido: maxNumberLevel numero negativo	**className**: valida **maxNumberLevel**: intero < 0**[ERRORE]**	PC01	className=" Calcolatrice" maxNumberLevel=-1	RandoopException: INVALID_NTEST	Non vengono generati test
5	Test non valido: maxNumberLevel uguale a zero	**className**: valida **maxNumberLevel**: intero = 0 0**[ERRORE]**	PC01	className=" Calcolatrice" maxNumberLevel=0	RandoopException: INVALID_NTEST	Non vengono generati test
7	Test valido	**className**: valida **maxNumberLevel**: valido	- PC01 - mock di IObserver - Deve esistere la subdirectory convenzionale per Calcolatrice	className=" Calcolatrice" maxNumberLevel=1	Viene invocato il metodo notify prendendo in input 1	Eliminati dal repository dei test i test generati

Test suite: RequestTest

In questa test suite andiamo a verificare il requisito `RNF01` per il quale tutte le richieste devono essere servite. Anche qui possiamo individuare delle precondizioni condivise da tutti i casi di test, indicate con la sigla **PC01** per motivi di leggibilità.

- `RandoopConnector: randoopConnector` esiste
- La repository dei test deve esistere e deve avere la struttura convenzionale.
- Va settato il path relativo alla repository dei test.

Le test suite sono state organizzate come di seguito:

ID	DESCRIZIONE	PRE-CONDIZIONI	INPUT	OUTPUT ATTESO	POST-CONDIZIONI
6	Test valido: vengono effettuate 5 richieste successive	- PC01 - mock IObserver - Deve esistere la subdirectory convenzionale per le classi specificate	**Input1:** className="Calcolatrice" maxNumberLevel=1 **Input2:** className="Calcolatrice1" maxNumberLevel=1 **Input3:** className="Calcolatrice2" maxNumberLevel=1 **Input4:** className="Calcolatrice3" maxNumberLevel=1 **Input5:** className="Calcolatrice4" maxNumberLevel=1	Tutte le richieste sono servite: il metodo notify viene invocato per 5 volte prendendo in input 1	Eliminati dal repository dei test i test generati

Test suite: SpecificationBasedTest

In questa test suite andiamo a testare i requisiti funzionali `RF05` e `RF06`. Allo stesso modo di quella precedente, abbiamo individuato come precondizioni condivise (**PC01**) le seguenti:

- `RandoopConnector: randoopConnector` esiste
- È stato settato il path relativo alla repository dei test tramite l'opportuno metodo di RandoopConnector

ID	DESCRIZIONE	PRE-CONDIZIONI	INPUT	OUTPUT ATTESO	POST-CONDIZIONI
7	Test non valido: la classe non compila	PC01 mock IObserver	className="Esempio" maxNumberLevel=5	RandoopException: CLASS_NO_COMPILE	Non vengono generati test
8	Test non valido: la cartella per la classe non esiste	- PC01 - mock IObserver - Nella repository dei test non esiste la cartella Calcolatore	className="Calcolatore" maxNumberLevel=2	RandoopException: DIR_ERROR	Non vengono generati test
9	Test non valido: la cartella per la classe esiste ma non c'è la classe	- PC01 - mock IObserver - Nella repository dei test esiste la cartella Calendario ma all'interno non c'è la classe Calendario	className="Calendario" maxNumberLevel=2	RandoopException: CLASS_NOT_FOUND	Non vengono generati test
10	Test valido	- PC01 - mock IObserver - Nella repository dei test esiste la subdirectory per la classe VCardBean	className="VCardBean" maxNumberLevel=3	Viene invocato il metodo notify prendendo in input un valore intero minore di 3	Eliminati dal repository dei test i test generati

Guida di Installazione

La seguente guida di utilizzo è relativa all'installazione su un container. Per installare e integrare su un container docker il servizio è necessario:

1. Effettuare il clone di questa repostitory;
2. Eseguire il file ./startDocker.sh su Mac o ./startDocker.bat su Windows passando come argomento il percorso della repository condivisa dove si memorizzano le classi di test.

Se si esegue su linux, è sufficiente seguire i seguenti passi per installare e integrare il servizio:

- Effettuare il clone del questa repostitory;
- Eseguire lo script ./install.sh

A titolo di esempio, abbiamo incluso anche un progetto di esempio che importa il jar creato e inoltra una richiesta di prova per le classi che inizialmente sono presenti nella directory condivisa.
Al fine di eseguire il main di tale progetto è necessario spostarsi all'interno del container nella directory `esempio` ed eseguire i seguenti comandi:

```
mvn compile
mvn exec:java -Dexec.mainClass="org.example.Main"
```