**Report Software Testing**

**Progetto “1+”: FastJSON**

L’obiettivo del progetto *1+* è quello di sperimentare le tecniche di Software Testing e gli strumenti a supporto di esse, rendendo parametrici dei test cases di due classi di test preesistenti nel progetto *FastJSON*. Inoltre, si vuole valutare ed analizzare l’andamento della copertura strutturale al variare dei casi di test eseguiti e dei parametri passati in input agli stessi.

Le due classi di test in esame sono *JSONPatchTest\_0.java* e *MapTest2.java*.

Per quanto riguarda la configurazione del progetto, è stato inserito in una pipeline usando il framework di CI delle GitHub Actions. Per stimare la coverage ottenuta con l’esecuzione dei test e generarne dei report, viene utilizzato JaCoCo, che è stato agganciato al processo di build della pipeline. Tuttavia, non essendo possibile leggerne i risultati su SonarCloud, in quanto i sorgenti testati sono instrumentati on-the-fly a partire da un jar, i report sulla coverage vengono analizzati esclusivamente in seguito ad una build in locale.

Come possibile dedurre dal Codice originale della classe JSONPatchTest\_0.java, lo scopo di questa classe è quello di testare il metodo statico *apply* della classe *JSONPatch*. Tale metodo prende in input due rappresentazioni di JSONObject in formato Stringa, di cui la seconda descrive un pool di operazioni da effettuare sul primo JSONObject. Ognuna di queste operazioni può essere una add, remove, replace, move, copy o test.

La classe di test originale prevedeva un metodo di test separato per ogni passaggio di parametri differente al metodo da testare; inoltre, il risultato atteso è passato in maniera hardcoded. Dunque, in sostanza il test case risulta essere soltanto uno, ma invocato con parametri differenti. Quindi, la parametrizzazione dei test cases è avvenuta separando la logica di configurazione del test dall’effettiva esecuzione, operando nel seguente modo: il costruttore della classe di test chiama internamente un metodo privato *configureTestClass*, passandogli in input i parametri da utilizzare nel metodo di test; inoltre, viene passato anche il risultato atteso come output dell’invocazione del metodo sotto test.

Tale scelta è stata fatta in quanto FastJSON è basato su un concetto di oggetto JSON differente dallo standard ed implementare un oracolo per il calcolo dell’output atteso avrebbe richiesto una completa re-implementazione del metodo under test e della logica sottostante a livello di dominio. Tutto ciò è stato ritenuto andare oltre gli obiettivi del progetto *1+*.

Il metodo *configureTestClass* non fa altro che configurare l’ambiente di test, inizializzando gli attributi privati della classe di test con i due parametri di input da usare nell’invocazione del metodo sotto test e con il valore dell’output atteso. Nel costruttore è lasciata de-commentata solo una invocazione per volta al metodo di configure; in questo modo, si seleziona ad ogni build una chiamata al configure con parametri diversi, così da valutare il cambiamento ottenuto in termini di copertura.

In Codice 2 è riportata una visione complessiva del codice della classe *JSONPatchTest\_0* re-implementata per avere dei test parametrici.

Come già anticipato, la coverage viene stimata utilizzando il framework JaCoCo, il quale produce principalmente due metriche di copertura: una statement coverage, basata sul numero di istruzioni in bytecode coperte dall’esecuzione dei casi di test, ed una branch coverage, la quale prende in considerazione i costrutti di switch e gli if-statement.

La prima analisi viene effettuata invocando il metodo under test con una patch comprendente un pool di operazioni; in particolare, si tratta di una operazione di *replace*, una di *add* ed una di *remove*.

Com’è possibile vedere in Figura 1, si ottiene una statement coverage complessiva dei sorgenti del 14% e, in particolare, del package *com.alibaba.fastjson* (in cui è presente la classe *JSONPatch*) del 7%. La singola classe *JSONPatch* risulta avere una copertura strutturale del 51% ed una branch coverage del 38 % (Figura 2). La copertura dei singoli metodi della classe in questione è visibile in Figura 3.

Una seconda analisi è effettuata lanciando il test con una patch composta da una singola operazione di *move*.

Per quanto riguarda la copertura strutturale del codice sorgente, la Figura 4 ci dimostra che si hanno davvero pochi cambiamenti rispetto al caso precedente; in particolare, la percentuale di statement coverage del package *com.alibaba.fastjson* è esattamente uguale. Invece, la copertura della classe *JSONPatch* risulta essere aumentata in statement coverage (dal 51% al 58%), ma non in branch coverage (Figura 5). Infatti, guardando la copertura strutturale dei metodi della classe (Figura 6), essa risulta essere aumentata sia per il metodo *apply(Object, String)* che per *isObject(String)*. Per comprendere tali risultati, è stato analizzato il decompilato ottenuto a partire dal file *JSONPatch.class*. Il blocco di codice che si occupa di gestire l’operazione di *move* richiede molteplici istruzioni, più di quante ne richiedano un’operazione di *add*, di *replace* e di *remove* insieme.

L’altra classe di test è *MapTest2.java*. Essa include originariamente un solo metodo di test in cui viene testato il metodo statico *parseObject(String text, TypeReference<T> type)* della classe *JSON*. Tale metodo ha il compito di fare il parsing della stringa in input (rappresentante un JSON Object) in un oggetto il cui tipo è indicato dalla classe *TypeReference*. In particolare, per un JSON Object ci si aspetta che venga trasformato in una *Map<Object, Object>*. Il test passa se e solo se ad ogni chiave è associato il valore atteso.

Il codice originale della classe è riportato in Codice 3. Anche in questo caso, per la parametrizzazione si è fatto uso di un metodo *configureTestClass* invocato dal costruttore della classe di test. L’unico parametro passato al metodo di configure è la stringa rappresentante il *JSONObject*. La configurazione dell’ambiente di test consiste nel settare un attributo privato della classe di test con la stringa di cui sopra e nell’allocare un’istanza di *TypeReference<Map<Object,Object>>*, associando anch’essa ad un attributo privato.

Questa volta, è stato possibile fare un semplice parsing della stringa rappresentante il *JSONObject* per ottenerne un array di chiavi di tipo *Object* da utilizzare nel metodo di test. Inoltre, è stata utilizzata la libreria **JSON-JAVA (org.json)** per ottenere un oggetto JSON da cui estrarre gli output attesi per il test. Tali responsabilità sono state attribuite ad una classe *Oracle* interna alla classe di test, la quale agisce appunto da oracolo di test.

La nuova implementazione parametrica di *MapTest2* è consultabile in Codice 4.

Essendo il test originario eseguito con una sola rappresentazione in stringa di un *JSONObject*, non è possibile valutare la variazione della copertura al cambio dei valori dei parametri di test (a meno di progettare ed inserire nuovi parametri appropriati). La coverage che si ottiene a seguito dell’esecuzione del solo test in esame è consultabile in Figura 7 e Figura 8: la statement coverage su tutti i package ammonta al 5%, quella del package *org.alibaba.fastjson* (contenente la classe *JSON*) appena all’1%, mentre la statement coverage della classe *JSON* è del 9%. Sono valori molto bassi, ma giustificati dall’esecuzione di un solo test case su un metodo di una singola classe, la quale ha una size molto elevata.

Sono state in seguito valutate le metriche di copertura dell’intera test-suite, composta dai due metodi di test delle classi parametrizzate in precedenza. Ci si aspetta naturalmente un incremento dei valori di coverage rispetto a quelli analizzati precedentemente.

Relativamente ai risultati ottenuti dal primo run del test di *JSONPatchTest\_0.java*, la branch coverage è aumentata dal 6% al 7% su tutti i package (Figura 9), mentre, dal punto di vista percentuale, la statement coverage è rimasta assestata sul 14%. Tuttavia, è possibile notare che il numero di missed instructions è leggermente diminuito. Quindi, come ci si aspettava, si ha un miglioramento generale della coverage.

In particolare, è molto interessante notare come la statement coverage della classe *JSON* sia passata dal 9%, con l’esecuzione del test di *MapTest2,* al 14%, con l’esecuzione dell’intera test-suite (Figura 10). Questo perché ovviamente la classe viene coinvolta anche nel flusso d’esecuzione del test di *JSONPatchTest\_0*, molto probabilmente per eseguire il parsing delle stringhe in oggetti JSON.

Repository GitHub: <https://github.com/AndreaPepe/fastjsonTesting>.

**Progetto “2”: Apache Syncope**

**Dominio**

Apache *Syncope* è un sistema open-source per la gestione di identità digitali in ambienti enterprise. Il suo obiettivo primario è quello di gestire i dati e le informazioni degli utenti e dei loro account su sistemi e applicazioni. Inoltre, svolge il ruolo di access management system, dando la possibilità di definire delle politiche di accesso a risorse, applicabili a singoli utenti, a gruppi o persino ad altri oggetti definibili in modo personalizzato (e.g. stampanti, dispositivi IoT, etc.). Per ulteriori informazioni, consultare la documentazione ufficiale del progetto al seguente [link](https://syncope.apache.org/docs/2.1/getting-started.html).

**Prima classe: DefaultPasswordGenerator**

Una delle due classi scelte per l’attività di Software Testing è stata *DefaultPasswordGenerator*. Tale scelta è stata dettata sia dall’importanza della classe, in quanto la gestione e la generazione delle passwords è un punto focale dell’identity management, sia dalla discreta documentazione trovata online. Infatti, una delle grandi mancanze del progetto in questione è di certo una documentazione dal livello di dettaglio assolutamente insoddisfacente, sia ad alto livello che a basso livello (Javadoc nel codice).

Come desumibile, dal nome della classe e dal seguente commento di documentazione, la classe fornisce la funzionalità di generare automaticamente delle password conformi a determinate policies:

*Generate random passwords according to given policies.*

*When no minimum and / or maximum length are specified, default values are set.*

*WARNING:This class only takes DefaultPasswordRuleConf into account.*

Al seguente [link](https://syncope.apache.org/docs/2.1/reference-guide.html#policies-password) è possibile trovare ciò che la documentazione ufficiale riporta in merito alle password e alle possibili policies cui devono aderire. In particolare, viene specificato che:

When defining a password policy, the following information must be provided:

* allow null password - whether a password is mandatory for Users or not
* history length - how many values shall be considered in the history
* rules - set of password rules to evaluate with the current policy

Inoltre, per quanto riguarda le *rules* possibili, abbiamo quanto segue:

The default password rule (enforced by *DefaultPasswordRule* and configurable via *DefaultPasswordRuleConf*) contains the following controls:

* maximum length - the maximum length to allow; 0 means no limit set;
* minimum length - the minimum length to allow; 0 means no limit set;
* non-alphanumeric required
* alphanumeric required
* digit required
* lowercase required
* uppercase required
* must start with digit
* must not start with digit
* must end with digit
* must not end with digit
* must start with alphanumeric
* must start with non-alphanumeric
* must not start with alphanumeric
* must not start with non-alphanumeric
* must end with alphanumeric
* must end with non-alphanumeric
* must not end with alphanumeric
* must not end with non-alphanumeric
* username allowed - whether a username value can be used
* words not permitted - list of words that cannot be present, even as a substring;
* schemas not permitted - list of schemas whose values cannot be present, even as a substring;
* prefixes not permitted - list of strings that cannot be present as a prefix;
* suffixes not permitted - list of strings that cannot be present as a suffix.

Come prima cosa, si decide di testare il metodo principale della classe, che genera una password in accordo alla lista di policies indicata. Il codice di tale metodo è consultabile nella tabella Codice 5.

L’unico parametro di input del metodo è una lista di *PasswordPolicy*, la quale è un’interfaccia che è implementata dalla classe *DefaultPasswordPolicy*. Il tipo di ritorno è una stringa, ma può essere lanciata un’eccezione di tipo *InvalidPasswordRuleConf* se la configurazione delle policies è non valida.

Alla luce di queste informazioni, si procede nell’effettuare **domain partitioning** dei parametri di input e dei possibili output dell’operazione. Per quanto riguarda l’unico parametro di input, si tratta di una lista, quindi si considerano i seguenti sottoinsiemi:

Tuttavia, si tratta di una lista di *PasswordPolicy*, dunque, un tipo di dato complesso che, per quanto descritto in precedenza può avere una configurazione più o meno valida all’interno del dominio di riferimento. Si nota che lo stato della classe *DefaultPasswordRule* è determinato dai parametri *allowNullPassword*, *historyLength* e *rules*; è ovvio che il fatto che la password sia o meno obbligatoria per l’utente o la history non sono rilevanti dal punto di vista della generazione automatica della password stessa. L’unico fattore che realmente impatta il risultato ottenuto è l’insieme di regole che la password dovrà rispettare.

Dunque, si distingue subito che tra le *non empty list* è lecito distinguere tra liste con policies valide e non valide, quindi le classi di equivalenza individuate diventano:

Per quanto riguarda i possibili output, si hanno tre possibilità rappresentate dal verificarsi di un’eccezione per regole non valide, da password conforme alle policies e da password non conforme alle policies.

Effettuando la **boundary analysis**, si ottengono i seguenti boundary values:

Essendo il parametro di input unico, è possibile soltanto avere un approccio unidimensionale. Si procede quindi alla definizione ad alto livello dei casi di test da implementare e del risultato atteso, come riportato in Tabella 1. In occasione di una lista di policies vuota, ci si aspetta che vengano rispettate le regole di default, come confermato anche da una successiva analisi del codice sorgente. Tali regole hanno unicamente constraints sulla size della password: lunghezza minima di 8 caratteri e massima di 64.

La policy non valida è stata ottenuta imponendo due regole in contrasto: la password deve terminare con una lettera e la password deve terminare con un numero.

Per l’esecuzione dei test, è stato necessario effettuare un mock statico della classe *ImplementationManager*.

Tutti i test eseguiti in questa prima iterazione vanno a buon fine e si ottengono risultati per la statement coverage e la branch coverage visibili in Figura 11 e in Figura 12. Le percentuali di coverage sono già molto elevate, questo perché il metodo testato invoca internamente quasi tutti gli altri metodi, perlopiù protected, della classe.

Si vuole cercare di migliorare la coverage ottenuta, sia di classe che di metodi, andando in particolare a stimolare con inputs differenti il metodo *check(DefaultPasswordRuleConf)*, il quale controlla che le policies siano valide. Infatti, è questo il punto che con più probabilità può essere buggato, ad esempio ritenendo valida una policy non valida o viceversa.

Per fare ciò, è necessario operare sullo stato interno della classe *DefaultPasswordRuleConf*, la quale contiene i settaggi di tutte le *rules* elencate nella documentazione, che viene utilizzata per caratterizzare le varie policies.

Tale stato è caratterizzato da ben 24 attributi (uno per ognuna delle *rules* indicate sulla documentazione), di cui due di tipo intero (massima e minima size della password), 16 di tipo booleano e 4 di tipo List<String> per le blacklist di parole da non contenere come prefisso, suffisso o all’interno della password.

Si procede con l’individuazione delle classi di equivalenza:

* Per i parametri booleani si individuano le classi {true} e {false}; per completezza si elencano i nomi dei parametri: *nonAlphanumericRequired, alphanumericRequired, digitRequired, lowercaseRequired, uppercaseRequired, mustStartWithDigit, mustntStartWithDigit, mustEndWithDigit, mustntEndWithDigit, mustStartWithNonAlpha, mustntStartWithNonAlpha, mustStartWithAlpha, mustntStartWithAlpha, mustEndWithNonAlpha, mustntEndWithNonAlpha, mustEndWithAlpha, mustntEndWithAlpha, usernameAllowed;*
* *wordsNotPermitted, schemasNotPermitted, prefixesNotPermitted, suffixesNotPermitted:*

Per questi ultimi parametri, non è stata considerata la classe di equivalenza {null} perché ritenuta non significativa all’interno dello scenario.

Effettuando la boundary-value analysis, si identificano i seguenti valori:

* Parametri booleani:
* wordsNotPermitted, schemasNotPermitted, prefixesNotPermitted, suffixesNotPermitted:

Come valore >0 per *maxLength* si è scelto un valore sufficientemente grande così che la stringa possa contenere abbastanza caratteri per poter verificare la validità dei requisiti espressi dai parametri booleani.

Per i primi due parametri, si userà un approccio multidimensionale, essendo tra loro strettamente correlati.

Per i parametri di tipo *List* e per i booleani *nonAlphanumericRequired, alphanumericRequired, digitRequired, lowercaseRequired, uppercaseRequired e usernameAllowed* si userà un approccio unidimensionale.

Per tutti gli altri parametri booleani, si userà un approccio multidimensionale sulle coppie e triple di parametri che possono potenzialmente essere in contrasto tra loro: ad esempio tra *mustStartWithDigit* e *mustntStartWithDigit* oppure tra *mustStartWithAlpha* e *mustStartWithNonAlpha*.

In generale, per , si avrà un approccio multidimensionale per tutte le coppie di parametri del tipo *mustStartWithX* e *mustntStartWithX*, così come per *mustEndWithX* e *mustntEndWithX.* Inoltre, si avrà un approccio multidimensionale per le triple (*mustntStartWithDigit, mustntStartWithAlpha, mustntStartWithNonAlpha*) e (*mustntEndWithDigit, mustntEndWithAlpha, mustntEndWithNonAlpha*).

È importante notare che sono state desunte le seguenti informazioni dal codice sorgente:

* *isAlphanumericRequired* è ritenuto soddisfatto sia dalla presenza di una lettera che di una cifra;
* in tutti gli altri casi, la dicitura *nonAlpha* è soddisfatta da qualsiasi carattere diverso da una lettera (quindi cifra o carattere speciale); invece *Alpha* è soddisfatto solo da lettere e *Digit* solo da cifre.

È chiaro che i requisiti espressi dai parametri booleani hanno un effetto solo nel caso in cui tali parametri assumano valore *true*. In tutti gli altri casi, è come se quei parametri non esistessero. Alla luce di ciò, tra tutte le possibili combinazioni di parametri del test, ne sono state selezionate alcune ritenute più significative delle altre. Sono stati così definiti i casi di test da aggiungere a quelli già implementati nella prima iterazione; i valori dei parametri e i risultati attesi sono consultabili nella Tabella 2.

L’implementazione di tali test-cases si è concretizzata nell’evoluzione della classe di test precedentemente implementata. I test sono strutturati in maniera parametrica e la classe ha dei metodi privati che hanno il compito di computare il risultato atteso o un eventuale verificarsi di un’eccezione, in accordo a quanto specificato nella documentazione. L’esecuzione dei test rivela la presenza di alcune incongruenze a causa di difetti nel codice:

* con ci si aspetta di ottenere una password lunga minimo un carattere, in quanto impostando non si imposta un limite per la lunghezza massima; invece, si ottiene un’eccezione di tipo *ArrayIndexOutOfBoundsException*;
* se si indica che la password deve terminare con una cifra e non deve terminare con una cifra, ci si aspetta un’eccezione, che invece non viene lanciata, bensì si ottiene una password che termina con una cifra;
* accade lo stesso indicando che la password deve iniziare con una un *Alpha* e anche con un *NonAlpha*; l’eccezione attesa non viene lanciata e si ottiene una password che inizia con un *NonAlpha*;
* analogamente se si indica che la password debba terminare sia con un *Alpha* che con un *NonAlpha*;
* se si specifica che la password NON deve iniziare né con un *Alpha,* né con un *NonAlpha*, né con un *Digit*, ci si aspetta un’eccezione perché non c’è alcun carattere con cui la password possa iniziare, ma invece viene restituita una password che ha come primo carattere un *Alpha*;
* analogamente se si indica che la password NON deve terminare con nessuna delle tre tipologie di caratteri.

La maggior parte dei bugs sono causati dalla mancanza di controlli di consistenza, in particolare nel metodo protected *check()* della classe *DefaultPasswordGenerator*. Inoltre, come è ben visibile dai report di coverage generati da PITest (per la mutation coverage) e da JaCoCo, risulta esserci una linea di codice non coperta inaspettatamente: ciò è dovuto alla mancanza di un operatore logico di NOT nella condizione di un’if-statement, come visibile in Figura 14 e in Figura 15.

Con questa seconda iterazione, sono stati ottenuti miglioramenti significativi sia per la statement coverage che in particolare per la branch coverage della classe testata e dei suoi metodi, come visibile in Figura 16 e in Figura 18. Attraverso l’uso del framework PITest, si è inoltre valutata la mutation coverage, risultata pari al 92% (Figura 17). In particolare, la branch coverage del metodo protected *check()* è giunta ad un valore ritenuto soddisfacente.

Tuttavia, per incrementare ulteriormente la copertura e la mutation coverage, si decide di testare anche l’altro metodo pubblico della classe, ovvero *generate(ExternalResource),* per il quale risultano esserci mutanti non uccisi (Figura 19). *ExternalResource* è un’interfaccia che, in particolare, definisce delle operazioni per settare ed ottenere una singola *PasswordPolicy*. Il metodo che si vuole testare non fa altro che ottenere tale policy dall’istanza della classe che implementa *ExternalResource* ed usarla per invocare l’altro metodo pubblico precedentemente testato.

Essendo l’esito del test determinato sostanzialmente dagli stessi parametri di configurazione dei test-cases precedenti, si decide di sfruttare le stesse combinazioni di valori per i parametri in questione per implementari i casi di test per il nuovo metodo. Dal punto di vista implementativo, ciò è stato realizzato aggiungendo un ulteriore metodo di test all’interno della classe di test originaria. Ovviamente, anche questi test-cases evidenziano gli stessi fallimenti rilevati in precedenza.

Tuttavia, la coverage calcolata con JaCoCo è ulteriormente migliorata (Figura 20), così come la mutation coverage, salita al 94% (Figura 21).

Assumendo un profilo operazionale con probabilità di utilizzo uniforme rispetto al solo insieme di casi di test progettati ed implementati, visibili ancora una volta in Tabella 2, e ai quali aggiungere i due casi di test con una lista di policy *null* ed una vuota implementati nella prima iterazione (prima e quarta riga di Tabella 1), si hanno in totale 24 combinazioni testate e, quindi, ognuna di esse ha probabilità di utilizzo pari a . Essendo che 6 di esse evidenziano fallimenti, la reliability stimata ammonta al

**Seconda classe: Encryptor**

La successiva classe che si è deciso di testare è la classe *Encryptor*, per la quale non sono presenti tantissime informazioni sulla documentazione ufficiale, ma abbastanza per poterne progettare e implementare dei test, inferendo, ove necessario, dal contesto e dal codice sorgente, adottando una strategia di Software Testing di tipo gray-box.

La classe non è documentata in maniera specifica, ma è stato possibile reperire alcune informazioni, disposte in modo sparso nella documentazione ufficiale, utili ad intuire le funzionalità e il comportamento previsto; ad esempio, da *password (Users only) - hashed or encrypted value, depending on the selected password.cipher.algorithm which can be used for authentication* si intuisce che l’uso di un encryptor sia necessario per la cifratura o l’hashing delle password e che possono essere selezionati diversi algoritmi per effettuare tali operazioni. Sugli algoritmi disponibili, la documentazione è molto chiara ed esaustiva, elencandoli ([documentazione ufficiale](https://syncope.apache.org/docs/2.0/reference-guide.html#configuration-parameters)):

* SHA-1
* SHA-256
* SHA-512
* AES
* S-MD5
* S-SHA-1
* S-SHA-256
* S-SHA-512
* BCRYPT

Sono tutti algoritmi di hashing, fatta eccezione per AES (algoritmo di cifratura a blocchi, a chiave simmetrica).

La documentazione fornisce anche altre due importanti informazioni:

* *The value of the secretKey property in the security.properties file is used for AES-based encryption / decryption. Besides password values, this is also used whenever reversible encryption is needed, throughout the whole system*.
* *When the secretKey value has length less than 16, it is right-padded by random characters during startup, to reach such mininum value.*

*It is strongly recommended to provide a value long at least 16 characters, in order to avoid unexpected behaviors at runtime, expecially with high-availability*.

Come prima funzionalità da testare, si sceglie quella ritenuta più importante, ovvero la cifratura. La classe *Encryptor* espone un metodo pubblico *encode()*, il quale prende in input due parametri: il primo parametro *value* è di tipo String e rappresenta la stringa da cifrare, mentre il secondo parametro *cipherAlgorithm* è di tipo *CipherAlgorithm* (una Enum) e rappresenta ovviamente l’algoritmo da utilizzare. Tuttavia, la documentazione fa riferimento anche ad una *secretKey* che è necessaria per l’encryption con AES, ma se non viene specificata alcuna *secretKey*, ne viene impostata una di default.

Per una prima iterazione di test, si decide di utilizzare esclusivamente la secret key di default, focalizzandosi sulla riuscita o meno dell’encoding. Si procede effettuando **domain partitioning** per i parametri presi in input dal metodo sotto test:

Effettuando **boundary-value analysis**, si decide che la stringa di dimensione inferiore ad un blocco di cifratura, deve essere inferiore al blocco più piccolo usato tra tutti gli algoritmi (64 bit per Blowfish, usato da BCRYPT), mentre per la stringa più grande del blocco di cifratura, dovrà essere più grande del blocco di dimensioni maggiori tra tutti gli algoritmi (1024 bit per SHA-512). Dunque, si ha:

* “” (stringa vuota),

Si è deciso di avere un approccio multidimensionale, considerando l’insieme di casi di test da category partition riportati in Tabella 3. Il controllo che la stringa codificata ritornata dal metodo sia valida è stato effettuato invocando il metodo *verify()* della stessa classe *Encryptor*, assumendolo essere corretto.

Come visibile in Figura 22 e in Figura 23, si ottiene una class coverage del 77% per quanto riguarda la statement coverage e del 70 % per quanto riguarda la branch coverage; inoltre, per il metodo testato *encode()*, si ha una statement coverage del 100% ed una branch coverage dell’ 87%.

La mutation coverage calcolata usando PITest è pari al 50% (Figura 24). Inoltre, si nota che la branch coverage del metodo *encode()* è ancora migliorabile in quanto non è stato esplorato un ramo di un if-statement che viene considerato quando *cipherAlgorithm* è *null* (Figura 25), valore inizialmente non previsto nei casi di test progettati ed implementati fin’ora. Si decide dunque di aggiungere altri 4 test-cases in cui tale valore è *null*, combinandolo con gli altri 4 valori del parametro *value* originariamente previsti.

Inoltre, per migliorare la coverage, si decidono di testare anche i metodi *verify()* e *decode().* In aggiunta, si otterà un’istanza di *Encryptor* passando una *secretKey* non di default di lunghezza minore di 16 caratteri, per verificare che, come specificato dalla documentazione, ne venga effettuato il padding a 16 caratteri non sollevando errori.

Per il metodo *verify()*, le classi di equivalenza e i boundary-values individuati per i due parametri *value* e *cipherAlgorithm* sono le stesse del metodo *encode()*, mentre per il terzo parametro *encoded* abbiamo:

E i boundary-values sono risultati essere e . Si è perseguito con un approccio multidimensionale tra i tre parametri, con i test-cases elencati in Tabella 4.

Per il metodo *decode()*, in base a quanto specificato nella documentazione, è utilizzabile in maniera efficace solo con AES, che è l’unico algoritmo di cifratura a chiave simmetrica tra quelli elencati. Dunque, per i due parametri *encoded* e *cipherAlgorithm* si hanno le seguenti classi di equivalenza:

Risultanti nei seguenti boundary-values:

* “”,

A causa della mancanza di documentazione, è stato necessario un approccio white-box per comprendere gli output attesi nelle diverse situazioni. Quando viene passato un *CipherAlgorithm* non consono al decoding, era naturale attendersi il lancio di un’eccezione, ma dal codice sorgente si è appreso che il metodo ritorna *null.* Si entra in logica di errore solo se si verificano errori durante il processo di encoding vero e proprio. I casi di test ottenuti da category partition, applicando un approccio multidimensionale, sono elencati in Tabella 5.

Grazie alla seconda iterazione di test, si è ottenuta una migliore coverage di classe (sia statement coverage che branch coverage) per Encryptor, così come per i singoli metodi, come testimoniato in Figura 26 e Figura 27. Inoltre, anche la mutation coverage risulta essere migliorata, arrivando al 64%, rispetto al 50% ottenuto nella prima iterazione (Figura 28).

Un profilo operazionale per l’operazione *encode()* è definito in base ai valori assumibili dai due parametri per cui è stato utilizzato un approccio multidimensionale, testando tutte le possibili combinazioni:

* value: [null, “”, “ab”, stringa lunga 129 caratteri]
* cipherAlgorithm: [null, SHA-1, SHA-256, SHA-512, AES, S-MD5, S-SHA-1, S-SHA-256, S-SHA-512, BCRYPT].

In totale si hanno 40 combinazioni, ognuna con probabilità di utilizzo uniforme pari a . Non si ha nessun fallimento nell’esecuzione dei test cases, quindi la reliability per tale metodo risulta essere del .

Analogamente, per i metodi *verify()* e *decode()*, si considerano profili operazionali desumibili dalla Tabella 4 e dalla Tabella 5, in cui ogni combinazione ha rispettivamente probabilità di utilizzo pari a e ad . Anche in questo caso, non essendoci fallimenti nei test, la reliability ammonta al

Lo sviluppo dei test è stato incluso nel ciclo di build del progetto utilizzando le GitHub Actions ([link repository GitHub](https://github.com/AndreaPepe/syncope)) ed è stato inoltre integrato con [SonarCloud](https://sonarcloud.io/summary/overall?id=AndreaPepe_syncope), rendendo possibile l’analisi della coverage ottenuta tramite JaCoCo. Purtroppo, non è stato possibile stimare l’adeguatezza dei test secondo dei criteri data-flow oriented con il tool *ba-dua*, a causa di incompatibilità tra le versioni dei JDK richieste dal tool e dalle altre dipendeze del progetto.

**Progetto “2”: Apache BookKeeper**

**Dominio**

*Apache BookKeeper* è un servizio di storage ottimizzato per carichi di lavoro real-time ed è in grado di garantire scalabilità, resistenza ai guasti, consistenza, replicazione e durabilità.

Dal punto di vista della terminologia, in *BookKeeper*, ogni unità di un log da salvare è chiamata **entry**. Flussi di log entries sono chiamati **ledgers** e vengono salvati su servers chiamati **bookies**.

Per maggiori informazioni, consultare la [documentazione ufficiale](https://bookkeeper.apache.org/docs/overview/).

**Prima classe: BookKeeper**

Come prima classe da testare, è stata scelta *BookKeeper*, in quanto è la classe che rappresenta il client utilizzabile dall’utente per connettersi ad un server BookKeeper, e dunque è di fondamentale importanza.

Come desumibile dalla [Javadoc](https://bookkeeper.apache.org/docs/latest/api/javadoc/org/apache/bookkeeper/client/BookKeeper.html), ci sono 4 operazioni possibili: start di un nuovo ledger, scrittura su un ledger, lettura da un ledger, rimozione di un ledger.

Alla luce di ciò, si decide in una prima iterazione di testare un paio delle funzionalità descritte, in particolare la creazione e la rimozione di un ledger.

Per quanto riguarda la creazione, la classe espone diverse operazioni che la realizzano e ci sono sia creazioni sincrone che asincrone. Si decide di iniziare dal testing del metodo *createLedger()* sincrono e che prende in input il maggior numero di parametri tra tutti i metodi sincroni che offrono la stessa funzionalità.

I parametri risultano essere i seguenti:

* (int) *ensSize* - number of bookies over which to stripe entries
* (int) *writeQuorumSize* - number of bookies each entry will be written to
* (int) *ackQuorumSize* - number of bookies which must acknowledge an entry before the call is completed
* (BookKeeper.DigestType) *digestType* - digest type, either MAC or CRC32
* (byte[]) *passwd* - password
* (Map<String, byte[]>) *customMetadata* - optional customMetadata that holds user specified metadata

Il tipo di ritorno è un tipo complesso di tipo *LedgerHandle*, rappresentante un handle al ledger creato; inoltre, l’operazione dichiara di poter lanciare delle eccezioni di tipo *InterruptedException* e *BKException.*

Per quanto riguarda il parametro *digestType*, vengono indicati solo i valori MAC e CRC32. Tuttavia, dalla [documentazione](https://bookkeeper.apache.org/docs/latest/api/javadoc/org/apache/bookkeeper/client/BookKeeper.DigestType.html), si desume che la enum presenta anche i valori DUMMY e CRC32C, pertanto, anche tali valori verranno presi in considerazione per i test.

Inoltre, sempre dalla documentazione, si apprende la seguente regola:

*When creating a ledger, the following invariant must hold:*

***E >= Qw >= Qa***

*Thus, the ensemble size (****E****) must be larger than the write quorum size (****Qw****), which must in turn be larger than the ack quorum size (****Qa****). If that condition does not hold, then the ledger creation operation will fail.*

Si procede con l’individuazione delle classi di equivalenza e quindi dei valori derivanti da boundary-value analysis, ottenendo i risultati in Tabella 6. Per i parametri riguardanti le size, è stato scelto un approccio multidimensionale, dovendo valere la relazione prima descritta. Per gli altri parametri, è stato adottato un approccio unidimensionale. Ciò ha portato alla definizione dei casi di test elencati in Tabella 7.

I test sono stati realizzati lanciando un’istanza di server di BookKeeper (e anche una necessaria di ZookKeeper) in locale, utilizzando la classe apposita *LocalBookKeeper, già* presente nel progetto, come consigliato nella seguente [pagina web](https://zookeeper.apache.org/doc/r3.4.13/bookkeeperStarted.html#bk_localBK). L’istanziazione, sia del server, che della classe client da testare, è stata effettuata prima dell’esecuzione di ogni metodo di test (@Before) e opportunamente chiusa una volta terminato il metodo (@After). Ciò ha garantito che ogni metodo di test usufruisse di un’istanza pulita di server. Essendo questa una procedura comune a tutte le classi di test per la classe *BookKeeper*, si è deciso di creare una classe astratta che implementasse tali operazioni, estesa dalle classi di test concrete.

Per quanto riguarda il metodo *createLedger(),* si è assunto che l’output atteso dovesse essere un’eccezione se la relazione tra le size non fosse stata rispettata oppure se l’*ensSize* fosse negativo.

Per il metodo *deleteLedger()*, invece, l’unico parametro di input è *lId,* di tipo *long,* rappresentante un identificativo del ledger che si vuole eliminare. Non viene ritornato alcun valore, ma ci si aspetta un’eccezione in caso di errore. Per il parametro di input, si individuano le seguenti classi di equivalenza:

Ciò si tramuta nei boundary-values:



, assumendo che i valori negativi non vengono mai associati ad un ledger al momento della sua creazione. Ci si aspetta che venga lanciata un’eccezione quando la *deleteLedger()* viene invocata con un ID che non corrisponde ad alcun ledger. Di conseguenza, vengono individuati, per tale metodo, i casi di test descritti in Tabella 8.

La copertura della classe e dei suoi metodi ottenuta con la test-suite costruita nella prima iterazione è consultabile in Figura 29, Figura 30 e Figura 31. Per il metodo *deleteLedger()*, la statement coverage è del 100%, mentre la branch coverage non è calcolabile in quanto non sono identificabili dei branch. La mutation coverage della classe è pari al 33%, come visibile in Figura 32.

Tuttavia, ci sono delle run di test che falliscono, non ottenendo il risultato atteso:

* Metodo *createLedger()*: i casi di test per i quali *ensSize < writeQuorumSize*, in cui il risultato atteso è un’eccezione (non viene rispettata la relazione ), provocano invece una situazione di stallo, causando un’esecuzione infinita della run di test. Tali situazioni sono classificate come difettose. Inoltre, nei casi in cui la relazione tra le size viene rispettata, ma *ensSize < 0*, viene lanciata un’eccezione non di tipo *BKException*, bensì di tipo *IllegalArgumentException*. Quest’ultimo caso non è stato considerato come un bug, non essendo specificato nella documentazione quale eccezione attendersi in tale situazione.
* Metodo *deleteLedger():* il tentativo di eliminare un ledger passando un ID non esistente, non provoca il lancio di un’eccezione verso il client (classe *BookKeeper*), al contrario di quanto ci si aspetterebbe da una situazione simile. Infatti, ciò è confermato dal log dell’esecuzione del caso di test, riportato in Figura 33. Ciò è considerato come un bug.

Nella seconda iterazione, con lo scopo di aumentare la copertura della classe, vengono testati gli altri metodi *createLedger()* esposti dalla classe ed il metodo *openLedger()*.

Per quanto riguarda le varianti di *createLedger()*, sono le seguenti:

* *createLedger(ensSize, writeQuorumSize, ackQuorumSize, digestType, passwd)*
* *createLedger(ensSize, qSize, digestType, passwd)*
* *createLedger(digestType, passwd)*

Per il primo dei tre, i parametri sono gli stessi del metodo *createLedger()* testato nella prima iterazione, semplicemente non viene passato il parametro *customMetadata*. Dunque, non considerando tale parametro, le classi di equivalenza, i boundary-values e i casi di test con i risultati attesi, rimangono gli stessi (Tabella 7).

Per il secondo dei tre, viene introdotto il parametro *qSize*, ovvero quorum size, il quale rappresenta la taglia sia del *writeQuorum* che dell’*ackQuorum*. Di conseguenza, la relazione tra le size viene semplificata in Per le classi di equivalenza e i boundary-values dei parametri individuati per tale metodo, consultare la Tabella 9, mentre per i casi di test la Tabella 10, generata considerando un approccio multidimensionale solo per i parametri *ensSize* e *qSize*, unidimensionale per gli altri.

Per il terzo e ultimo metodo, gli unici parametri sono i già discussi *digestType* e *passwd*. Dalla [documentazione](https://bookkeeper.apache.org/docs/latest/api/javadoc/org/apache/bookkeeper/client/BookKeeper.html#createLedger(org.apache.bookkeeper.client.BookKeeper.DigestType,byte%5B%5D):~:text=Creates%20a%20new%20ledger.%20Default%20of%203%20servers%2C%20and%20quorum%20of%202%20servers.), si evince che passando solo tali parametri, verrà restituito un handle ad un ledger che abbia *ensSize* pari a 3 e *writeQuorumSize* ed *ackQuorumSize* pari a 2. Tali informazioni sono state considerate per valutare il successo o meno delle run di test. I casi di test ottenuti applicando un approccio unidimensionale ai due parametri in questione sono elencati in Tabella 11.

Il metodo *openLedger()* svolge la funzionalità di aprire un ledger in lettura. Prende come parametri in input:

* ; deve coincidere con quella indicata al momento della creazione del ledger.

Dunque, sono state individuate le seguenti classi di equivalenza per gli attributi:

L’analisi dei valori di boundary è risultata in:

* [] *corretta, “passwd”.getBytes() corretta, “wrongPassword”.getBytes()*

Quando è indicato che la password deve essere corretta, significa che deve essere creato un ledger con la password specificata per il test, prima di invocare *openLedger()* usando il *lId* del ledger creato. Quando invece la password deve essere errata, il ledger deve essere creato (solo se si è nel caso di ledger esistente) con una password diversa da quella usata per il test.

Quindi, sono stati selezionati i casi di test in Tabella 12, applicando un approccio multidimensionale per i parametri *lId* e *passwd*, unidimensionale per *digestType*. In caso di ledger inesistente o di password errata ci si aspetta un’eccezione; in tutti gli altri casi un *LedgerHandle* valido.

La coverage della classe ottenuta con questa seconda iterazione di test è leggermente migliorata, arrivando al 42% di statement coverage e al 27% di branch coverage (Figura 34). Inoltre, i metodi testati nella seconda iterazione risultano essere completamente coperti (Figura 35). Anche la mutation coverage calcolata con PITest è aumentata fino al raggiungimento del 42% (Figura 36).

Le run di test per il metodo *openLedger()* non hanno evidenziato fallimenti, mentre, per quanto riguarda i metodi *createLedger(ensSize, writeQuorumSize, ackQuorumSize, digestType, passwd)* e *createLedger(ensSize, qSize, digestType, passwd),* è stato riscontrato lo stesso bug discusso in precedenza, per cui il caso in cui *ensSize < writeQuorumSize* oppure *ensSize < qSize* porta l’esecuzione a non terminare mai, anziché lanciare un’eccezione.

TODO: op profiles and reliabilty (per method and per class)

Al termine della seconda iterazione, si è sufficientemente soddisfatti dei risultati ottenuti e si decide di continuare l’attività di testing spostando il focus su un’altra classe del progetto.

**Nota**: i test riguardanti la classe *BookKeeper***,** come già specificato, prevedono l’istanziazione e il teardown di un nuovo server prima e dopo ogni run di test. Ciò comporta la necessità di un ingente quantitativo di tempo per l’esecuzione dei test

**Seconda classe: NetworkTopologyImpl**

Come descritto dalla [Javadoc](https://bookkeeper.apache.org/docs/latest/api/javadoc/org/apache/bookkeeper/net/NetworkTopologyImpl.html#:~:text=The%20class%20represents,centers%20or%20racks.), la classe *NetworkTopolgyImpl* implementa l’interfaccia *NetworkTopolgy* e rappresenta un cluster di computer organizzati in una topologia di rete gerarchica ad albero. Ad esempio, un cluster può consistere in diversi data centers in cui sono presenti racks di computer. Le foglie rappresentano **data nodes** (computer), mentre gli **inner nodes** rappresentano switch o router che gestiscono il traffico in ingresso e in uscita dai data centers o dai rack.

Per quanto riguarda un generico nodo, sempre dalla [documentazione](https://bookkeeper.apache.org/docs/latest/api/javadoc/org/apache/bookkeeper/net/Node.html#:~:text=A%20node%20may,is%20/dog/orange), abbiamo la seguente descrizione:

*A node may be a leave representing a data node or an inner node representing a datacenter or rack. Each data has a name and its location in the network is decided by a string with syntax similar to a file name. For example, a data node's* ***name*** *is* ***hostname:port#*** *and if it's located at rack "orange" in datacenter "dog", the string representation of its network* ***location*** *is* ***/dog/orange*.**

Il primo metodo che si decide di testare è il metodo *add()*, in quanto ritenuto essere il metodo fondamentale, per importanza e funzionalità che rappresenta, esposto dalla classe. Esso prendo in input un unico parametro, di tipo *Node* (interfaccia) e, come [documentato](https://bookkeeper.apache.org/docs/latest/api/javadoc/org/apache/bookkeeper/net/NetworkTopologyImpl.html#add(org.apache.bookkeeper.net.Node):~:text=Node%C2%A0node)-,Add%20a%20leaf%20node,-.%20Update%20node%20counter), implementa la funzionalità di aggiungere un nodo foglia all’albero della topologia di rete. Viene specificato che il nodo passato come parametro può essere *null* e che ci si aspetta una *IllegalArgumentException* in due specifici casi:

* Se si tenta di aggiungere un nodo ad un rack che è un nodo foglia (si può aggiungere solo ad inner nodes)
* Se il nodo che si tenta di aggiungere non è una foglia (è esso stesso un inner node)

Dunque, si individuano per il parametro *node* le seguenti classi di equivalenza:

Assumendo che le classi che implementano *Node* abbiano un costruttore del tipo *Node(name, networkLoacation)*, si usa tale notazione per indicare i valori di boundary individuati:

* *node: null, DataNode(“127.0.0.1:4000”, “/root”), InnerNode(“127.0.0.1:4001”, “/inner”), DataNode(“127.0.0.1:4002”, “/rack/127.0.0.1:3999”)*

Per eseguire il caso di test con l’ultimo boundary value indicato, è necessario prima inserire nella topologia della rete il DataNode(“127.0.0.1:3999”, “/rack”).

Dall’analisi effettuata, si procede individuando i casi di test da implementare, descritti in Tabella 13.

Affinché il test abbia successo, è necessario controllare che il nodo sia stato effettivamente inserito nella network topology nei casi in cui non è attesa un’eccezione.

A tale proposito, parallelamente al test di unità per il metodo *add()*, si è deciso di progettare e sviluppare un semplice test d’integrazione per accertarsi che il SUT (la classe *NetworkTopologyImpl*) agisse realmente sull’istanza di nodo passata come parametro al metodo. Infatti, l’interfaccia *Node* dichiara l’operazione *setParent(Node)*. Ci si aspetta che, durante l’esecuzione della *add()*, il metodo implementante l’operazione dichiarata dall’interfaccia venga invocato esattamente una volta sull’istanza di *Node* passata come parametro al metodo add(). Ovviamente, ciò è da considerare solo in caso di aggiunta di un nodo che vada a buon fine.

Per la realizzazione del test, è stato realizzato un mock spia sull’istanza di *Node* passata al metodo *add()*, tramite l’utilizzo di *Mockito*, verificando che il metodo *setParent()* venisse invocato esattamente 1 volta (Codice 6).

Inoltre, nella prima iterazione, si decide di testare anche il metodo *remove(Node)*. Ovviamente, prende come parametro in input il nodo da rimuovere dalla topologia di rete. Come specificato dalla [documentazione](https://bookkeeper.apache.org/docs/latest/api/javadoc/org/apache/bookkeeper/net/NetworkTopologyImpl.html#remove(org.apache.bookkeeper.net.Node):~:text=node%20to%20be%20removed%3B%20can%20be%20null), il nodo può essere *null*. Anche se non viene specificato, ci si aspetta che non sia possibile la rimozione di un *InnerNode*, in quanto non ne è possibile nemmeno l’aggiunta con il metodo *add()*. In tal caso, l’output atteso è un’eccezione, molto probabilmente di tipo *IllegalArgumentException*, per analogia col metodo *add()*.

Le classi di equivalenza per *node* sono:

I boundary-values diventano: *null, DataNode(“127.0.0.1:4000”, “/root”), InnerNode(“127.0.0.1:4001”, “/inner”)*.

I casi di test progettati sono elencati in Tabella 14.

La test-suite implementata ha permesso di ottenere una statement coverage di classe pari al 35% ed una branch coverage pari al 25% (Figura 37). La copertura ottenuta per i singoli metodi con JaCoCo è consultabile invece in Figura 38, mentre per la mutation coverage si ha un valore del 15% (Figura 39). Il test di integrazione risulta essere passato, come visibile dal report generato dal plugin *Failsafe* (Figura 40).

Tuttavia, per il metodo *add()*, come descritto da documentazione, ci si aspettava un’eccezione di tipo *IllegalArgumentException* quando si cerca di aggiungere un nodo foglia ad un nodo che non sia un *InnerNode*. In realtà, tale operazione risulta comunque impossibile, ma l’eccezione lanciata è di tipo *InvalidTopologyException*. In effetti, come visibile dalla coverage generata da JaCoCo (Figura 41), l’if-statement che dovrebbe triggerare l’eccezione attesa non viene coperto, in quanto si entra nell’if precedente (riga 416 in figura; risulta comunque non coperta perché il test che portava al fallimento è stato disabilitato).

Si decide di aggiungere un ulteriore caso di test nella seconda iterazione, così da coprire anche il lancio dell’eccezione di tipo *InvalidTopologyException*. Infatti, valutando la copertura del metodo *add()* anche seguendo un approccio data-flow oriented (Figura 42), con il supporto del tool *ba-dua*, risultano esserci delle coppie def-use non coperte che è possibile coprire.

**Allegati**

|  |
| --- |
| public class JSONPatchTest\_0 **extends** TestCase **{**  public void test\_for\_multi\_0**()** **throws** Exception **{**  String original **=** "{\n" **+**  " \"baz\": \"qux\",\n" **+**  " \"foo\": \"bar\"\n" **+**  "}"**;**  String patch **=** "[\n" **+**  " { \"op\": \"replace\", \"path\": \"/baz\", \"value\": \"boo\" },\n" **+**  " { \"op\": \"add\", \"path\": \"/hello\", \"value\": [\"world\"] },\n" **+**  " { \"op\": \"remove\", \"path\": \"/foo\" }\n" **+**  "]"**;**  String result **=** JSONPatch**.**apply**(**original**,** patch**);**  assertEquals**(**"{\"baz\":\"boo\",\"hello\":[\"world\"]}"**,** result**);**  **}**  public void test\_for\_add\_1**()** **throws** Exception **{**  String original **=** "{}"**;**  String patch **=** "{ \"op\": \"add\", \"path\": \"/a/b/c\", \"value\": [ \"foo\", \"bar\" ] }"**;**  String result **=** JSONPatch**.**apply**(**original**,** patch**);**  assertEquals**(**"{\"a\":{\"b\":{\"c\":[\"foo\",\"bar\"]}}}"**,** result**);**  **}**  public void test\_for\_remove\_0**()** **throws** Exception **{**  String original **=** "{}"**;**  String patch **=** "{ \"op\": \"remove\", \"path\": \"/a/b/c\" }\n"**;**  String result **=** JSONPatch**.**apply**(**original**,** patch**);**  assertEquals**(**"{}"**,** result**);**  **}**  public void test\_for\_remove\_1**()** **throws** Exception **{**  String original **=** "{\"a\":{\"b\":{\"c\":[\"foo\",\"bar\"]}}}"**;**  String patch **=** "{ \"op\": \"remove\", \"path\": \"/a/b/c\" }\n"**;**  String result **=** JSONPatch**.**apply**(**original**,** patch**);**  assertEquals**(**"{\"a\":{\"b\":{}}}"**,** result**);**  **}**  public void test\_for\_replace\_1**()** **throws** Exception **{**  String original **=** "{\"a\":{\"b\":{\"c\":[\"foo\",\"bar\"]}}}"**;**  String patch **=** "{ \"op\": \"replace\", \"path\": \"/a/b/c\", \"value\": 42 }"**;**  String result **=** JSONPatch**.**apply**(**original**,** patch**);**  assertEquals**(**"{\"a\":{\"b\":{\"c\":42}}}"**,** result**);**  **}**  public void test\_for\_move\_0**()** **throws** Exception **{**  String original **=** "{\"a\":{\"b\":{\"c\":[\"foo\",\"bar\"]}}}"**;**  String patch **=** "{ \"op\": \"move\", \"from\": \"/a/b/c\", \"path\": \"/a/b/d\" }"**;**  String result **=** JSONPatch**.**apply**(**original**,** patch**);**  assertEquals**(**"{\"a\":{\"b\":{\"d\":[\"foo\",\"bar\"]}}}"**,** result**);**  **}**  public void test\_for\_copy\_0**()** **throws** Exception **{**  String original **=** "{\"a\":{\"b\":{\"c\":[\"foo\",\"bar\"]}}}"**;**  String patch **=** "{ \"op\": \"copy\", \"from\": \"/a/b/c\", \"path\": \"/a/b/e\" }"**;**  String result **=** JSONPatch**.**apply**(**original**,** patch**);**  assertEquals**(**"{\"a\":{\"b\":{\"c\":[\"foo\",\"bar\"],\"e\":[\"foo\",\"bar\"]}}}"**,** result**);**  **}**  public void test\_for\_test\_0**()** **throws** Exception **{**  String original **=** "{\"a\":{\"b\":{\"c\":[\"foo\",\"bar\"]}}}"**;**  String patch **=** "{ \"op\": \"test\", \"path\": \"/a/b/c\", \"value\": \"foo\" }"**;**  String result **=** JSONPatch**.**apply**(**original**,** patch**);**  assertEquals**(**"false"**,** result**);**  **}**  **}** |

Codice Codice originale della classe JSONPatchTest\_0.java

|  |
| --- |
| public class JSONPatchTest\_0 **{**  /\* Instance under test is not needed, because the tested method is static\*/  private String original**;**  private String patch**;**  private String expected**;**  public JSONPatchTest\_0**()** **{**  configureTestClass**(**"{\n" **+** " \"baz\": \"qux\",\n" **+** " \"foo\": \"bar\"\n" **+** "}"**,**  "[\n" **+**  " { \"op\": \"replace\", \"path\": \"/baz\", \"value\": \"boo\" },\n" **+**  " { \"op\": \"add\", \"path\": \"/hello\", \"value\": [\"world\"] },\n" **+**  " { \"op\": \"remove\", \"path\": \"/foo\" }\n" **+** "]"**,**  "{\"baz\":\"boo\",\"hello\":[\"world\"]}"**);**  // configureTestClass("{}", "{ \"op\": \"add\", \"path\": \"/a/b/c\", \"value\": [ \"foo\", \"bar\" ] }",  // "{\"a\":{\"b\":{\"c\":[\"foo\",\"bar\"]}}}");  //  // configureTestClass("{}", "{ \"op\": \"remove\", \"path\": \"/a/b/c\" }\n",  // "{}");  // configureTestClass("{\"a\":{\"b\":{\"c\":[\"foo\",\"bar\"]}}}", "{ \"op\": \"remove\", \"path\": \"/a/b/c\" }\n",  // "{\"a\":{\"b\":{}}}");  // configureTestClass("{\"a\":{\"b\":{\"c\":[\"foo\",\"bar\"]}}}", "{ \"op\": \"replace\", \"path\": \"/a/b/c\", \"value\": 42 }",  // "{\"a\":{\"b\":{\"c\":42}}}");  // configureTestClass("{\"a\":{\"b\":{\"c\":[\"foo\",\"bar\"]}}}", "{ \"op\": \"move\", \"from\": \"/a/b/c\", \"path\": \"/a/b/d\" }",  // "{\"a\":{\"b\":{\"d\":[\"foo\",\"bar\"]}}}");  // configureTestClass("{\"a\":{\"b\":{\"c\":[\"foo\",\"bar\"]}}}", "{ \"op\": \"copy\", \"from\": \"/a/b/c\", \"path\": \"/a/b/e\" }",  // "{\"a\":{\"b\":{\"c\":[\"foo\",\"bar\"],\"e\":[\"foo\",\"bar\"]}}}");  // configureTestClass("{\"a\":{\"b\":{\"c\":[\"foo\",\"bar\"]}}}", "{ \"op\": \"test\", \"path\": \"/a/b/c\", \"value\": \"foo\" }",  // "false");  **}**  private void configureTestClass**(**String original**,** String patch**,** String expected**)** **{**  **this.**original **=** original**;**  **this.**patch **=** patch**;**  **this.**expected **=** expected**;**  **}**  @Test  public void testJSONPatch**()** **{**  String result **=** JSONPatch**.**apply**(**original**,** patch**);**  Assert**.**assertEquals**(**expected**,** result**);**  **}**  **}** |

Codice Classe JSONPatchTest\_0.java con test parametrico

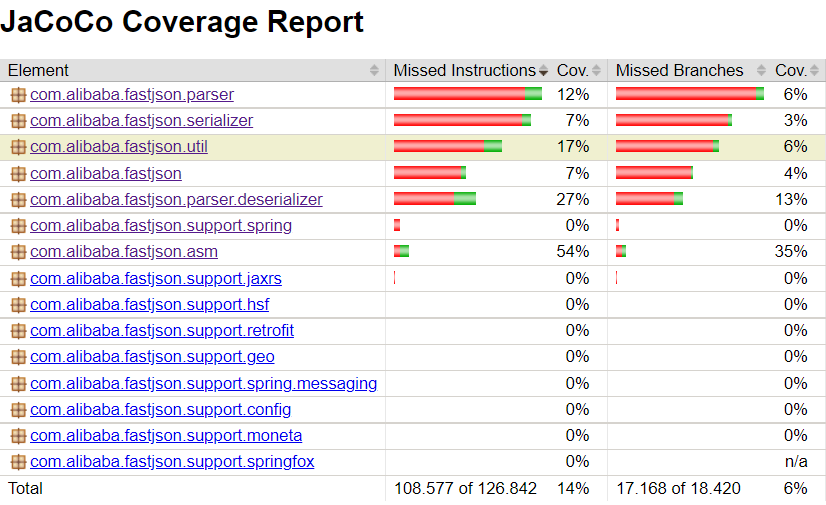


Figura Coverage eseguendo solo JSONPatchTest\_0 con una patch da 3 operazioni



Figura Coverage della classe JSONPatch con una patch da 3 operazioni

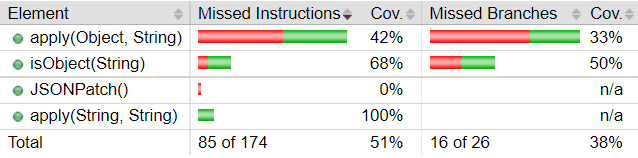


Figura Coverage dei metodi della classe JSONPatch con una patch da 3 operazioni

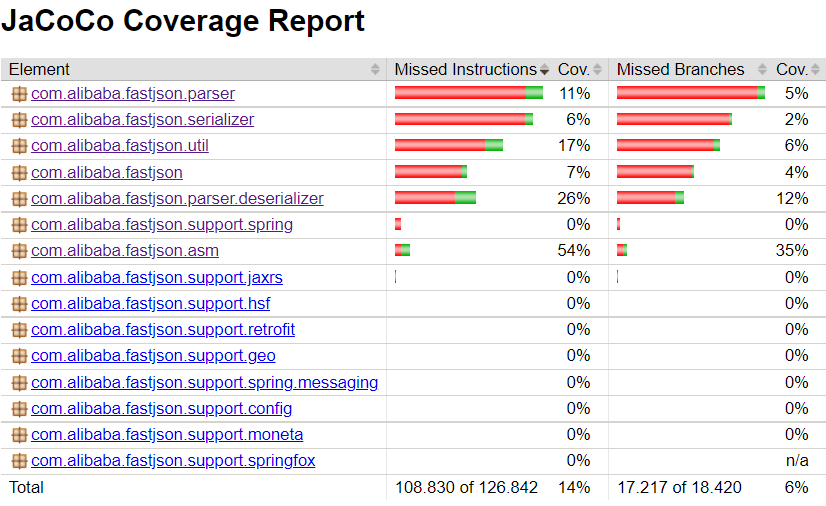


Figura Coverage ottenuta eseguendo solo JSONPatchTest\_0 con una patch da una singola operazione



Figura Copertura della classe JSONPatch a seguito dell'esecuzione di un test con una patch da una singola operazione

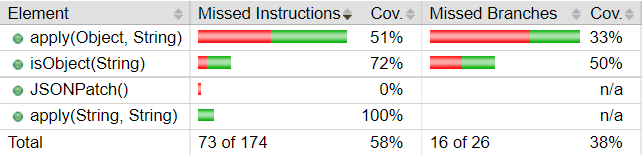


Figura Coverage dei metodi della classe JSONPatch in seguito al test con patch con singola operazione

|  |
| --- |
| public class MapTest2 **extends** TestCase **{**  public void test\_map **()** **throws** Exception **{**  Map**<**Object**,** Object**>** map **=** JSON**.**parseObject**(**"{1:\"2\",\"3\":4,'5':6}"**,** **new** TypeReference**<**Map**<**Object**,** Object**>>()** **{});**  Assert**.**assertEquals**(**"2"**,** map**.**get**(**1**));**  Assert**.**assertEquals**(**4**,** map**.**get**(**"3"**));**  Assert**.**assertEquals**(**6**,** map**.**get**(**"5"**));**  **}**  **}** |

Codice Codice originale della classe MapTest2

|  |
| --- |
| public class MapTest2 **{**  /\* The instance to be tested is not needed because the tested method is static\*/  private String jsonObject**;**  private TypeReference**<**Map**<**Object**,**Object**>>** typeReference**;**  public MapTest2 **(){**  configureTestClass**(**"{1:\"2\",\"3\":4,'5':6}"**);**  **}**  private void configureTestClass**(**String jsonObject**){**  **this.**jsonObject **=** jsonObject**;**  **this.**typeReference **=** **new** TypeReference**<>()** **{};**  **}**  /\*\*  \* This test method invokes the <i>parseObject</i> static method of the JSON class.  \* In particular, it checks that the conversion from the string representation  \* of the jsonObject to a Map between keys and values is well done.  \* The test is passed only if each value is associated to the correct key.  \*/  @Test  public void test\_map **()** **{**  Map**<**Object**,** Object**>** map **=** JSON**.**parseObject**(this.**jsonObject**,** **this.**typeReference**);**  // Use an oracle to compute the keys and the expected values bounded with those keys  Oracle oracle **=** **new** Oracle**();**  Object**[]** keys **=** oracle**.**getKeys**(**jsonObject**);**  Object**[]** expected **=** oracle**.**getExpectedResults**(**jsonObject**,** keys**);**  boolean passed **=** **true;**  **for** **(**int i **=** 0**;** i **<** keys**.**length**;** i**++){**  **if** **(**map**.**get**(**keys**[**i**])** **==** **null){**  **if** **(!** **(**map**.**get**(**keys**[**i**])** **==** expected**[**i**])){**  passed **=** **false;**  **break;**  **}**  **}**  **else** **if** **(!**map**.**get**(**keys**[**i**]).**equals**(**expected**[**i**])){**  passed **=** **false;**  **break;**  **}**  **}**  Assert**.**assertTrue**(**passed**);**  **}**  private static class Oracle**{**  /\*\*  \* This private method is used as a Software Testing oracle to know the  \* expected values of the JSON Object referred by the key in the input array of keys.  \* It's an oracle because it uses the library <b>org.json</b> as a trusted entity.  \*  \* **@param** jsonObject The string representing the JSONObject of the FastJSON project  \* **@param** keys The array of the objects representing the keys of the JSON Object  \* **@return** an array of Object, containing the values expected to be contained in the jsonObject,  \* in the same order of their keys  \*/  private Object**[]** getExpectedResults**(**String jsonObject**,** Object**[]** keys**){**  JSONObject obj **=** **new** JSONObject**(**jsonObject**);**  List**<**Object**>** expected **=** **new** LinkedList**<>();**  **for** **(**int i **=** 0**;** i **<** keys**.**length**;** i**++){**  expected**.**add**(**i**,** obj**.**get**(**String**.**valueOf**(**keys**[**i**])));**  **}**  **return** expected**.**toArray**();**  **}**  /\*\*  \* This private method is used to retrieve the array of keys  \* of the json object string representation in input.  \* **@param** jsonObject The string representing the JSONObject of the FastJSON project  \* **@return** an array of Objects, containing the keys of the JSONObject.  \*/  private Object**[]** getKeys**(**String jsonObject**){**  String noBrackets **=** jsonObject**.**replace**(**"{"**,** ""**);**  noBrackets **=** noBrackets**.**replace**(**"}"**,**""**);**  noBrackets **=** noBrackets**.**replace**(**" "**,** ""**);**  noBrackets **=** noBrackets**.**replace**(**"\n"**,** ""**);**  noBrackets **=** noBrackets**.**replace**(**"\r"**,** ""**);**  noBrackets **=** noBrackets**.**replace**(**"\t"**,** ""**);**  String**[]** pairs **=** noBrackets**.**split**(**","**);**  List**<**Object**>** ret **=** **new** LinkedList**<>();**  **for** **(**int i **=** 0**;** i **<** pairs**.**length**;** i**++){**  String key **=** pairs**[**i**].**split**(**":"**)[**0**];**  **if** **(**key**.**contains**(**"\""**)** **||** key**.**contains**(**"'"**)){**  ret**.**add**(**i**,** key**.**replace**(**"\""**,** ""**).**replace**(**"'"**,** ""**));**  **}**  **else** **if** **(**key**.**contains**(**"."**))**  ret**.**add**(**i**,** Float**.**valueOf**(**key**));**  **else**  ret**.**add**(**i**,** Integer**.**valueOf**(**key**));**  **}**  **return** ret**.**toArray**();**  **}**  **}**  **}** |

Codice Classe MapTest2.java con test parametrico e inner class Oracle

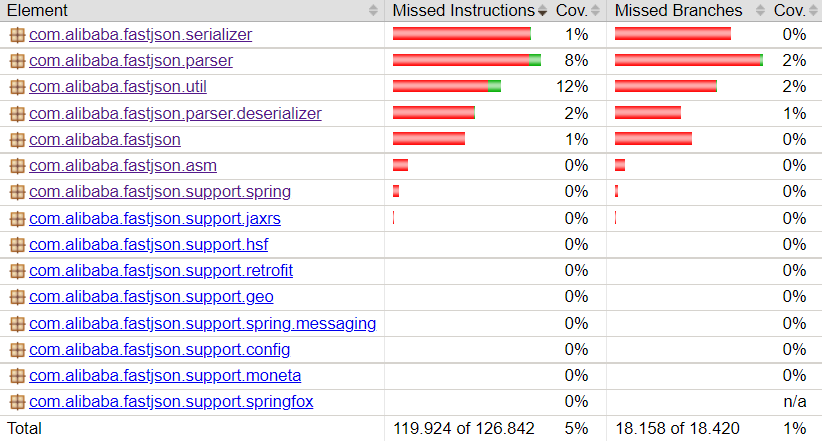


Figura Coverage ottenuta eseguendo il solo test della classe MapTest2.java



Figura Class coverage della classe JSON in seguito al test di MapTest2.java

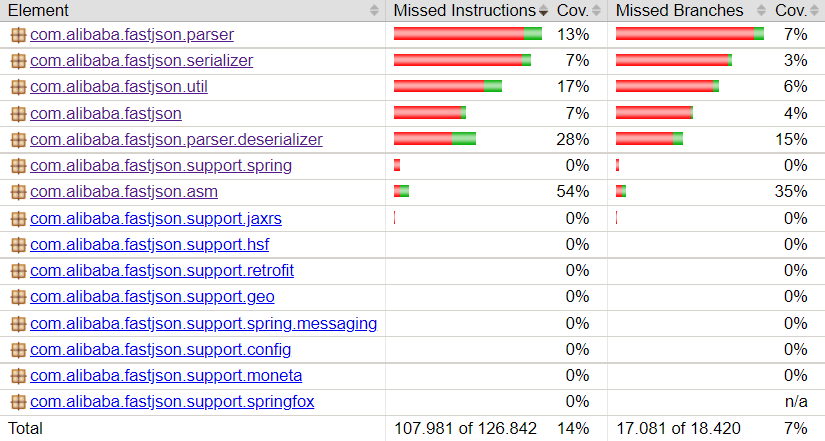


Figura Coverage in seguito all'esecuzione dell'intera test-suite

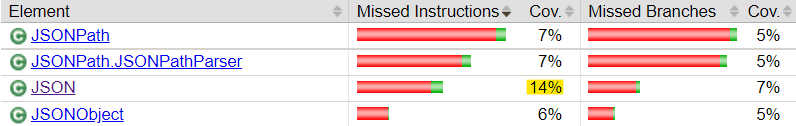


Figura Coverage della classe JSON aumentata con l'esecuzione di tutti i test della test-suite

|  |
| --- |
| public String generate**(**final List**<**PasswordPolicy**>** policies**)** **throws** InvalidPasswordRuleConf **{**  List**<**DefaultPasswordRuleConf**>** defaultRuleConfs **=** **new** ArrayList**<>();**  policies**.**stream**().**forEach**(**policy **->** policy**.**getRules**().**forEach**(**impl **->** **{**  **try** **{**  ImplementationManager**.**buildPasswordRule**(**impl**).**ifPresent**(**rule **->** **{**  **if** **(**rule**.**getConf**()** **instanceof** DefaultPasswordRuleConf**)** **{**  defaultRuleConfs**.**add**((**DefaultPasswordRuleConf**)** rule**.**getConf**());**  **}**  **});**  **}** **catch** **(**Exception e**)** **{**  LOG**.**error**(**"Invalid {}, ignoring..."**,** impl**,** e**);**  **}**  **}));**  DefaultPasswordRuleConf ruleConf **=** merge**(**defaultRuleConfs**);**  check**(**ruleConf**);**  **return** generate**(**ruleConf**);**  **}** |

Codice metodo generate(List<PasswordPolicy>) di DefaultPasswordGenerator

|  |  |
| --- | --- |
| **policies** | **Expected output** |
| Empty list | Password conforme alle policies di default |
| List with 1 valid policy | Password conforme alla policy indicata |
| List with 1 invalid policy | Eccezione |
| null | Eccezione |

Tabella Test cases iterazione 1 metodo generate(List<PasswordPolicy) di DefaultPasswordGenerator



Figura Class coverage DefaultPasswordGenerator (iterazione 1)

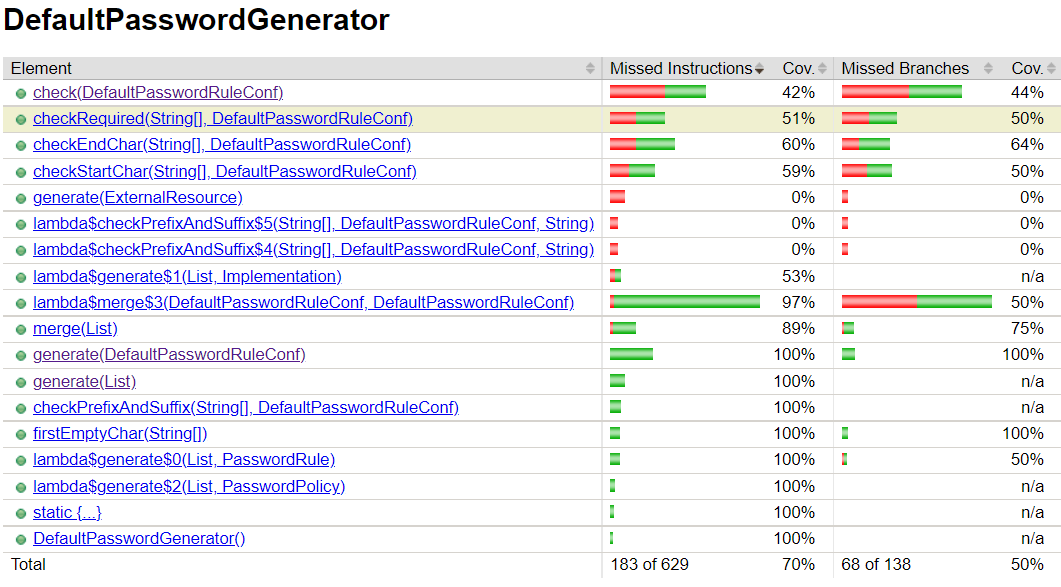


Figura Method coverage DefaultPasswordGenerator (iterazione 1)

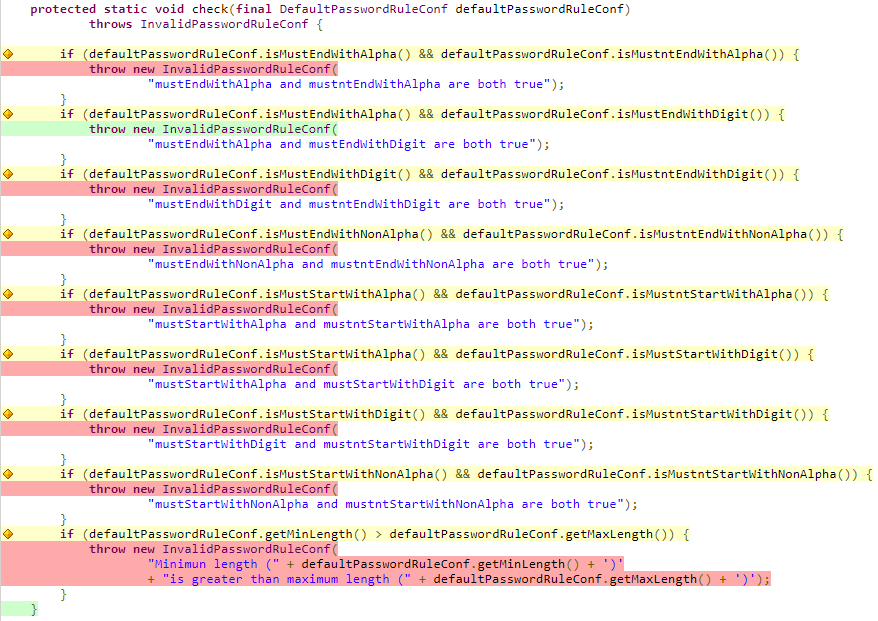


Figura Coverage del metodo check(DefaultPasswordRuleConf) di DefaultPasswordGenerator



Tabella Test cases per iterazione 2 di DefaultPasswordGenerator (fare doppio click sulla tabella per consultare il file Excel)

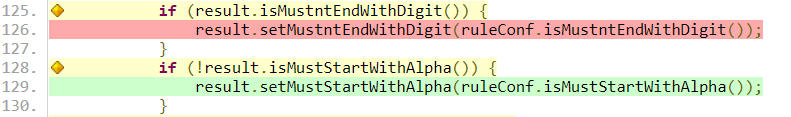


Figura Bug nella funzione merge(); manca un operatore NOT (!). JaCoCo report

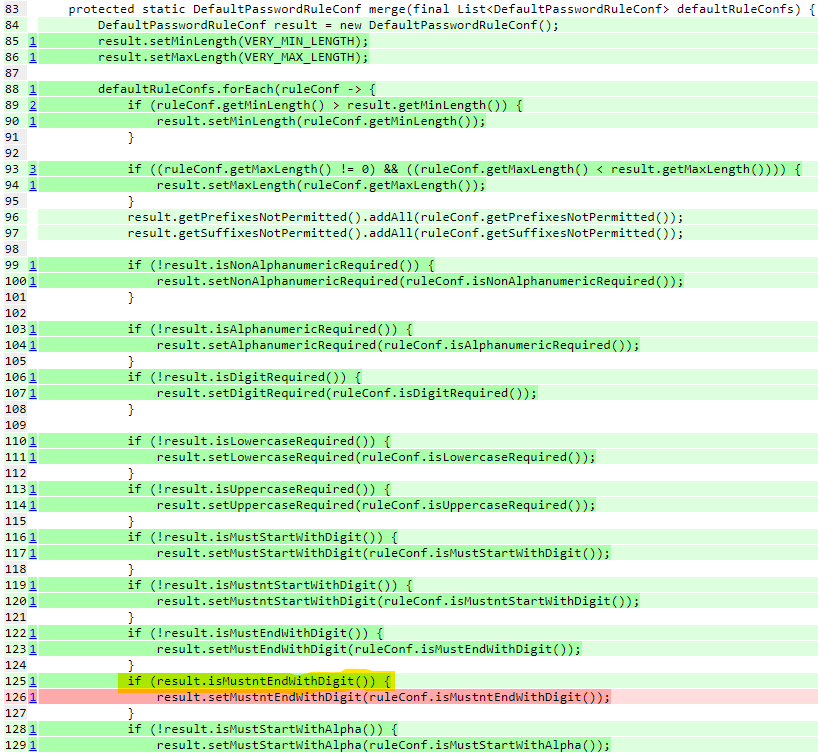


Figura Bug nella funzione merge() visto dal report di PIT. Evidenziato l'if-statement incriminato



Figura JaCoCo class coverage DefaultPasswordGenerator (iterazione 2)

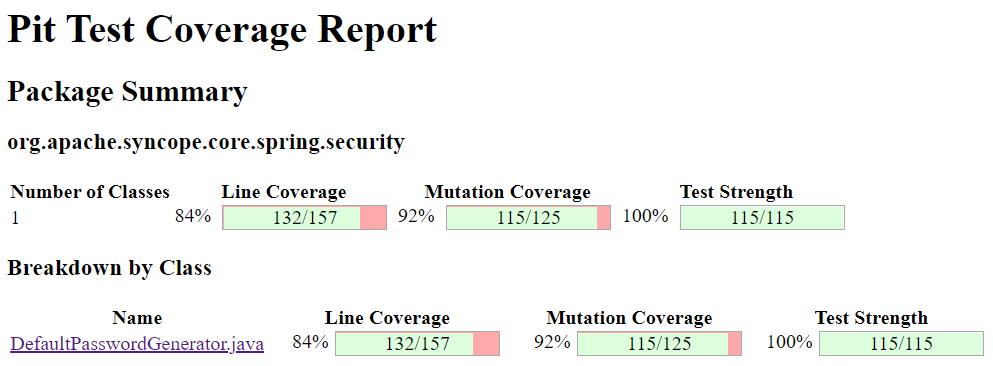


Figura PIT mutation coverage su DefaultPasswordGenerator (iterazione 2)

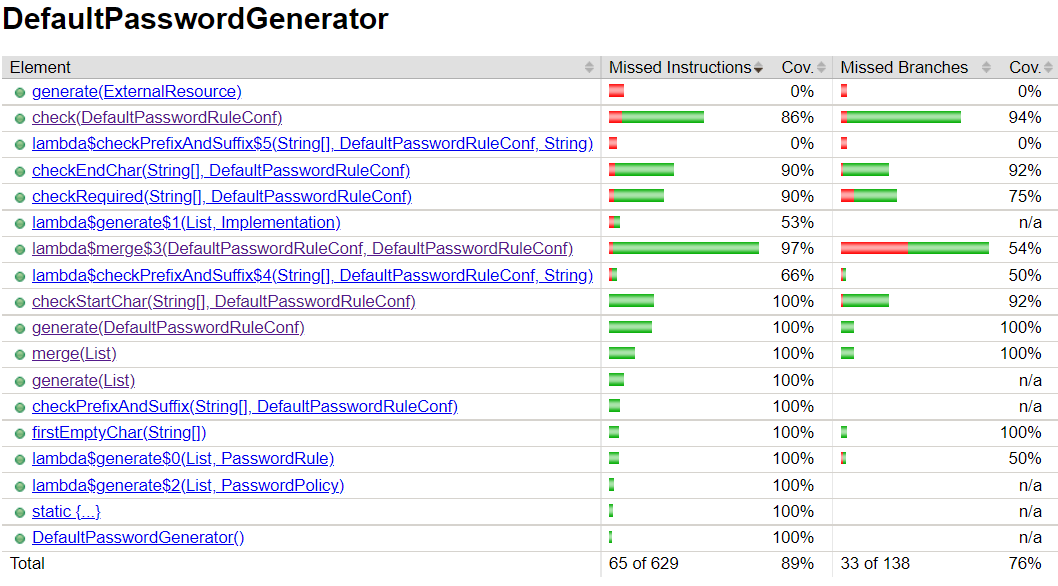


Figura JaCoCo method coverage su DefaultPasswordGenerator (iterazione 2)

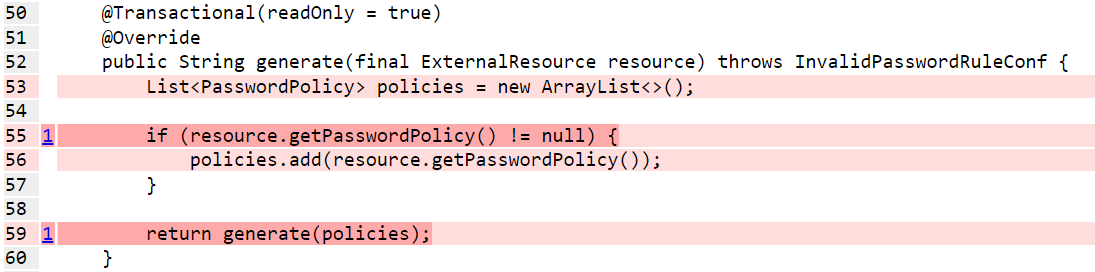


Figura PIT: mutanti non uccisi nel metodo generate(ExternalResource)



Figura JaCoCo class coverage DefaultPasswordGenerator (iterazione 3)

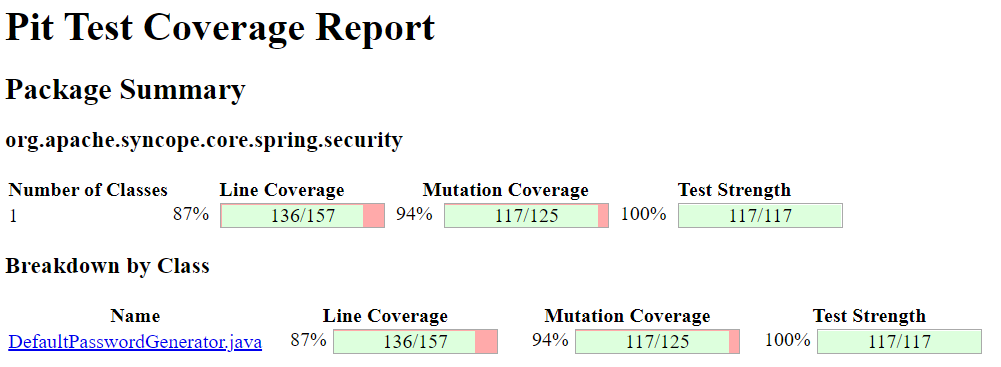


Figura PIT: mutation coverage incrementata sulla classe DefaultPasswordGenerator (iterazione 3)

|  |  |  |
| --- | --- | --- |
| **value** | **cipherAlgorithm** | **expected output** |
| null | SHA-1 | Stringa valida |
| “” | SHA-1 | Stringa valida |
| “ab” | SHA-1 | Stringa valida |
| Stringa da 129 char (8 bit per char) | SHA-1 | Stringa valida |
| null | SHA-256 | Stringa valida |
| “” | SHA-256 | Stringa valida |
| “ab” | SHA-256 | Stringa valida |
| Stringa da 129 char (8 bit per char) | SHA-256 | Stringa valida |
| null | SHA-512 | Stringa valida |
| “” | SHA-512 | Stringa valida |
| “ab” | SHA-512 | Stringa valida |
| Stringa da 129 char (8 bit per char) | SHA-512 | Stringa valida |
| null | AES | Stringa valida |
| “” | AES | Stringa valida |
| “ab” | AES | Stringa valida |
| Stringa da 129 char (8 bit per char) | AES | Stringa valida |
| null | S-MD5 | Stringa valida |
| “” | S-MD5 | Stringa valida |
| “ab” | S-MD5 | Stringa valida |
| Stringa da 129 char (8 bit per char) | S-MD5 | Stringa valida |
| null | S-SHA-1 | Stringa valida |
| “” | S-SHA-1 | Stringa valida |
| “ab” | S-SHA-1 | Stringa valida |
| Stringa da 129 char (8 bit per char) | S-SHA-1 | Stringa valida |
| null | S-SHA-256 | Stringa valida |
| “” | S-SHA-256 | Stringa valida |
| “ab” | S-SHA-256 | Stringa valida |
| Stringa da 129 char (8 bit per char) | S-SHA-256 | Stringa valida |
| null | S-SHA-512 | Stringa valida |
| “” | S-SHA-512 | Stringa valida |
| “ab” | S-SHA-512 | Stringa valida |
| Stringa da 129 char (8 bit per char) | S-SHA-512 | Stringa valida |
| null | BCRYPT | Stringa valida |
| “” | BCRYPT | Stringa valida |
| “ab” | BCRYPT | Stringa valida |
| Stringa da 129 char (8 bit per char) | BCRYPT | Stringa valida |

Tabella Test cases per il metodo encode() della classe Encryptor



Figura JaCoCo class coverage della classe Encryptor (iterazione 1)

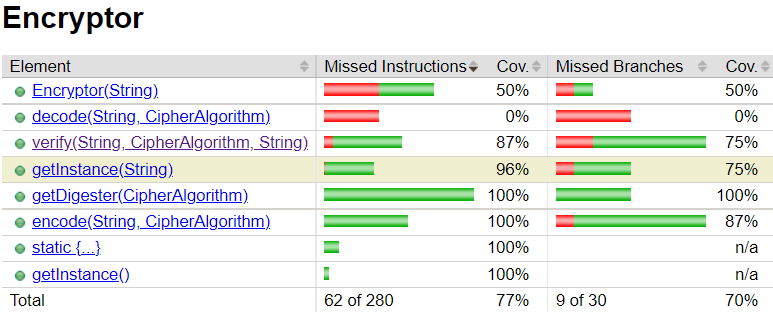


Figura 23 JaCoCo method coverage, classe Encryptor (iterazione 1)

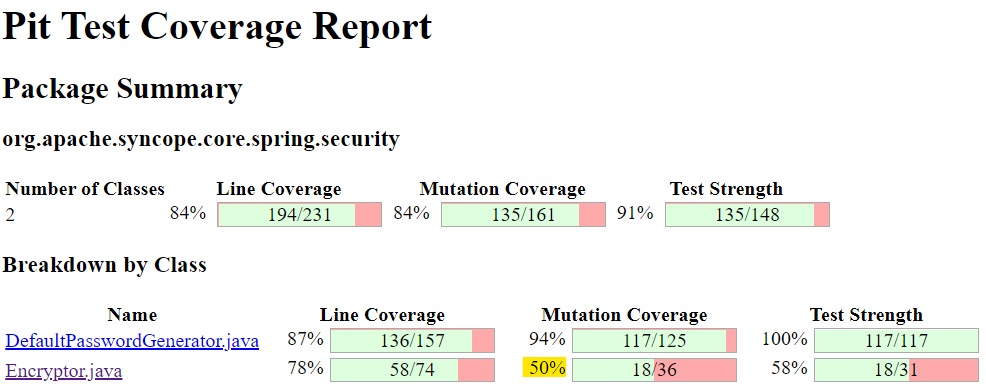


Figura PIT: mutation coverage classe Encryptor (iterazione 1)

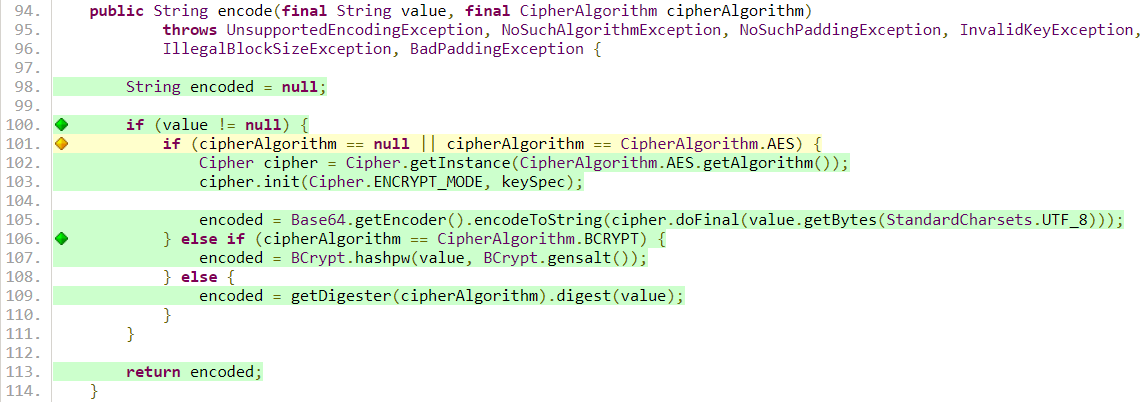


Figura Missing branch per cipherAlgoritm == null

|  |  |  |  |
| --- | --- | --- | --- |
| **value** | **cipherAlgorithm** | **encoded** | **expected output** |
| null | SHA-1 | encode(value, cipherAlgorithm) | false |
| “” | SHA-1 | encode(value, cipherAlgorithm) | true |
| “ab” | SHA-1 | encode(value, cipherAlgorithm) | true |
| Stringa da 129 char (8 bit per char) | SHA-1 | encode(value, cipherAlgorithm) | true |
| null | SHA-256 | encode(value, cipherAlgorithm) | false |
| “” | SHA-256 | encode(value, cipherAlgorithm) | true |
| “ab” | SHA-256 | encode(value, cipherAlgorithm) | true |
| Stringa da 129 char (8 bit per char) | SHA-256 | encode(value, cipherAlgorithm) | true |
| null | SHA-512 | encode(value, cipherAlgorithm) | false |
| “” | SHA-512 | encode(value, cipherAlgorithm) | true |
| “ab” | SHA-512 | encode(value, cipherAlgorithm) | true |
| Stringa da 129 char (8 bit per char) | SHA-512 | encode(value, cipherAlgorithm) | true |
| null | AES | encode(value, cipherAlgorithm) | false |
| “” | AES | encode(value, cipherAlgorithm) | true |
| “ab” | AES | encode(value, cipherAlgorithm) | true |
| Stringa da 129 char (8 bit per char) | AES | encode(value, cipherAlgorithm) | true |
| null | S-MD5 | encode(value, cipherAlgorithm) | false |
| “” | S-MD5 | encode(value, cipherAlgorithm) | true |
| “ab” | S-MD5 | encode(value, cipherAlgorithm) | true |
| Stringa da 129 char (8 bit per char) | S-MD5 | encode(value, cipherAlgorithm) | true |
| null | S-SHA-1 | encode(value, cipherAlgorithm) | false |
| “” | S-SHA-1 | encode(value, cipherAlgorithm) | true |
| “ab” | S-SHA-1 | encode(value, cipherAlgorithm) | true |
| Stringa da 129 char (8 bit per char) | S-SHA-1 | encode(value, cipherAlgorithm) | true |
| null | S-SHA-256 | encode(value, cipherAlgorithm) | false |
| “” | S-SHA-256 | encode(value, cipherAlgorithm) | true |
| “ab” | S-SHA-256 | encode(value, cipherAlgorithm) | true |
| Stringa da 129 char (8 bit per char) | S-SHA-256 | encode(value, cipherAlgorithm) | true |
| null | S-SHA-512 | encode(value, cipherAlgorithm) | false |
| “” | S-SHA-512 | encode(value, cipherAlgorithm) | true |
| “ab” | S-SHA-512 | encode(value, cipherAlgorithm) | true |
| Stringa da 129 char (8 bit per char) | S-SHA-512 | encode(value, cipherAlgorithm) | true |
| null | BCRYPT | encode(value, cipherAlgorithm) | false |
| “” | BCRYPT | encode(value, cipherAlgorithm) | true |
| “ab” | BCRYPT | encode(value, cipherAlgorithm) | true |
| Stringa da 129 char (8 bit per char) | BCRYPT | encode(value, cipherAlgorithm) | true |
| null | null | encode(value, cipherAlgorithm) | false |
| “” | null | encode(value, cipherAlgorithm) | false |
| “ab” | null | encode(value, cipherAlgorithm) | false |
| Stringa da 129 char (8 bit per char) | null | encode(value, cipherAlgorithm) | false |
| Stringa da 129 char (8 bit per char) | SHA-1 | “randomCrap” | false |
| null | SHA-256 | “randomCrap” | false |
| “” | SHA-256 | “randomCrap” | false |
| “ab” | SHA-256 | “randomCrap” | false |
| Stringa da 129 char (8 bit per char) | SHA-256 | “randomCrap” | false |
| null | SHA-512 | “randomCrap” | false |
| “” | SHA-512 | “randomCrap” | false |
| “ab” | SHA-512 | “randomCrap” | false |
| Stringa da 129 char (8 bit per char) | SHA-512 | “randomCrap” | false |
| null | AES | “randomCrap” | false |
| “” | AES | “randomCrap” | false |
| “ab” | AES | “randomCrap” | false |
| Stringa da 129 char (8 bit per char) | AES | “randomCrap” | false |
| null | S-MD5 | “randomCrap” | false |
| “” | S-MD5 | “randomCrap” | false |
| “ab” | S-MD5 | “randomCrap” | false |
| Stringa da 129 char (8 bit per char) | S-MD5 | “randomCrap” | false |
| null | S-SHA-1 | “randomCrap” | false |
| “” | S-SHA-1 | “randomCrap” | false |
| “ab” | S-SHA-1 | “randomCrap” | false |
| Stringa da 129 char (8 bit per char) | S-SHA-1 | “randomCrap” | false |
| null | S-SHA-256 | “randomCrap” | false |
| “” | S-SHA-256 | “randomCrap” | false |
| “ab” | S-SHA-256 | “randomCrap” | false |
| Stringa da 129 char (8 bit per char) | S-SHA-256 | “randomCrap” | false |
| null | S-SHA-512 | “randomCrap” | false |
| “” | S-SHA-512 | “randomCrap” | false |
| “ab” | S-SHA-512 | “randomCrap” | false |
| Stringa da 129 char (8 bit per char) | S-SHA-512 | “randomCrap” | false |
| null | BCRYPT | “randomCrap” | false |
| “” | BCRYPT | “randomCrap” | false |
| “ab” | BCRYPT | “randomCrap” | false |
| Stringa da 129 char (8 bit per char) | BCRYPT | “randomCrap” | false |
| null | null | “randomCrap” | false |
| “” | null | “randomCrap” | false |
| “ab” | null | “randomCrap” | false |
| Stringa da 129 char (8 bit per char) | null | “randomCrap” | false |

Tabella Test-cases per il metodo verify() della classe Encryptor

|  |  |  |
| --- | --- | --- |
| **encoded** | **cipherAlgorithm** | **expected output** |
| null | null | null |
| “” | null | null |
| encode(“myString”, AES) | null | null |
| “randomCrap” | null | null |
| null | AES | null |
| “” | AES | “” |
| encode(“myString”, AES) | AES | “myString” |
| “randomCrap” | AES | exception |
| null | BCRYPT | null |
| “” | BCRYPT | null |
| encode(“myString”, AES) | BCRYPT | null |
| “randomCrap” | BCRYPT | null |

Tabella Test cases per il metodo decode() della classe Encryptor



Figura 26 JaCoCo class coverage per Encryptor (iterazione 2)

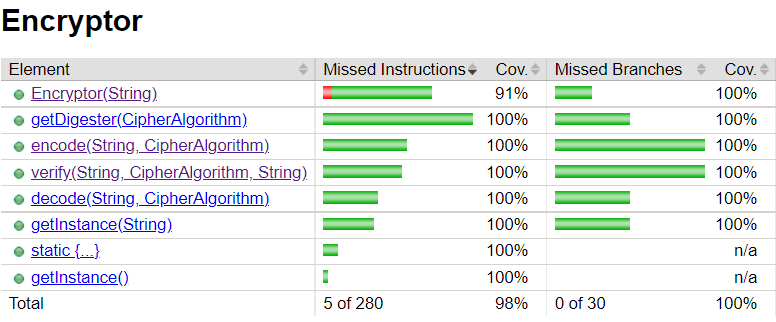


Figura JaCoCo method coverage per la classe Encryptor (iterazione 2)

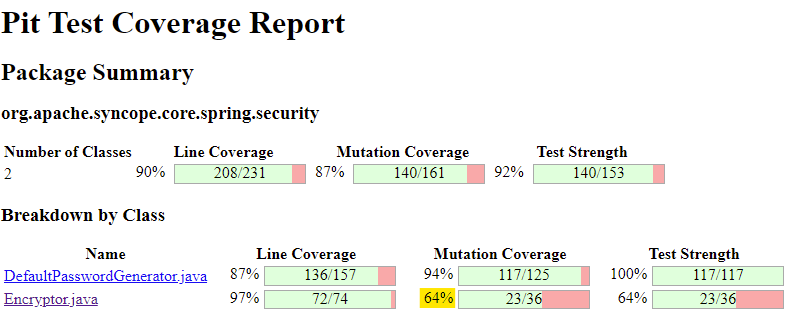


Figura PIT: mutation coverage della classe Encryptor incrementata al 64% (iterazione 2)

|  |  |  |  |
| --- | --- | --- | --- |
| **Tipo** | **Parametro** | **Classi di equivalenza** | **Boundary-values** |
| int | ensSize (E) | {<Qw}, {=Qw}, {>Qw} | Qw-1, =Qw, Qw+1 |
| int | writeQuorumSize (Qw) | {<Qa}, {=Qa}, {>Qa} | Qa-1, =Qa, Qa+1 |
| int | ackQuorumSize (Qa) | {<0}, {=0}, {>0} | -1, 0, 1 |
| DigestType | digestType | {MAC}, {CRC32}, {CRC32C}, {DUMMY} | MAC, CRC32, CRC32C, DUMMY |
| byte[] | passwd | {array vuoto}, {array non vuoto} | [], “passwd”.getBytes() |
| Map<String, byte[]> | customMetadata | {null}, {mappa vuota}, {mappa non vuota} | null, emptyMap(),  Map((“myMetadata”, “myCustomMetdata”)) |

Tabella Individuazione delle classi di equivalenza e boundary-value analysis per il metodo createLedger() della classe BookKeeper

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **ensSize** | **writeQuorumSize** | **ackQuorumSize** | **digestType** | **passwd** | **customMetadata** | **Output**  **atteso** |
| 3 | 2 | 1 | MAC | “passwd”.getBytes() | Map((“myMetadata”, “myCustomMetdata”)) | Handle valido |
| 2 | 2 | 1 | CRC32 | “passwd”.getBytes() | null | Handle valido |
| 1 | 2 | 1 | MAC | [] | null | BKException |
| 2 | 1 | 1 | MAC | “passwd”.getBytes() | null | Handle valido |
| 1 | 1 | 1 | CRC32 | [] | emptyMap() | Handle valido |
| 0 | 1 | 1 | MAC | “passwd”.getBytes() | null | BKException |
| 1 | 0 | 1 | MAC | “passwd”.getBytes() | null | BKException |
| 0 | 0 | 1 | CRC32C | “passwd”.getBytes() | null | BKException |
| -1 | 0 | 1 | MAC | “passwd”.getBytes() | null | BKException |
| 2 | 1 | 0 | DUMMY | [] | emptyMap() | Handle valido |
| 1 | 1 | 0 | MAC | “passwd”.getBytes() | null | Handle valido |
| 0 | 1 | 0 | MAC | “passwd”.getBytes() | null | BKException |
| 1 | 0 | 0 | CRC32 | “passwd”.getBytes() | null | Handle valido |
| 0 | 0 | 0 | CRC32C | “passwd”.getBytes() | Map((“myMetadata”, “myCustomMetdata”)) | Handle valido |
| -1 | 0 | 0 | MAC | [] | null | BKException |
| 0 | -1 | 0 | MAC | “passwd”.getBytes() | null | BKException |
| -1 | -1 | 0 | MAC | “passwd”.getBytes() | null | BKException |
| -2 | -1 | 0 | MAC | [] | null | BKException |
| 1 | 0 | -1 | MAC | “passwd”.getBytes() | null | Handle valido |
| 0 | 0 | -1 | DUMMY | “passwd”.getBytes() | null | Handle valido |
| -1 | 0 | -1 | MAC | “passwd”.getBytes() | null | BKException |
| 0 | -1 | -1 | MAC | “passwd”.getBytes() | null | Handle valido |
| -1 | -1 | -1 | MAC | [] | null | BKException |
| -2 | -1 | -1 | MAC | “passwd”.getBytes() | null | BKException |
| -1 | -2 | -1 | MAC | “passwd”.getBytes() | null | BKException |
| -2 | -2 | -1 | MAC | “passwd”.getBytes() | Map((“myMetadata”, “myCustomMetdata”)) | BKException |
| -3 | -2 | -1 | MAC | “passwd”.getBytes() | null | BKException |

Tabella Test-cases da category partition per createLedger() di BookKeeper

|  |  |
| --- | --- |
| **lId** | **expected output** |
| ledgerHandle.getId() | No exception |
| -1 (assunto non esistente) | Exception |

Tabella Test cases per deleteLedger() della classe BookKeeper



Figura JaCoCo coverage: classe BookKeeper, iterazione 1

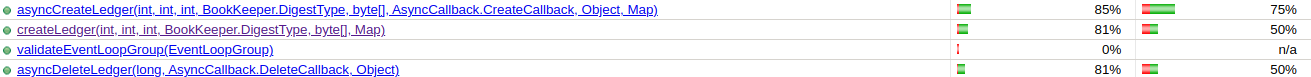


Figura Jacoco coverage, method coverage, iterazione 1

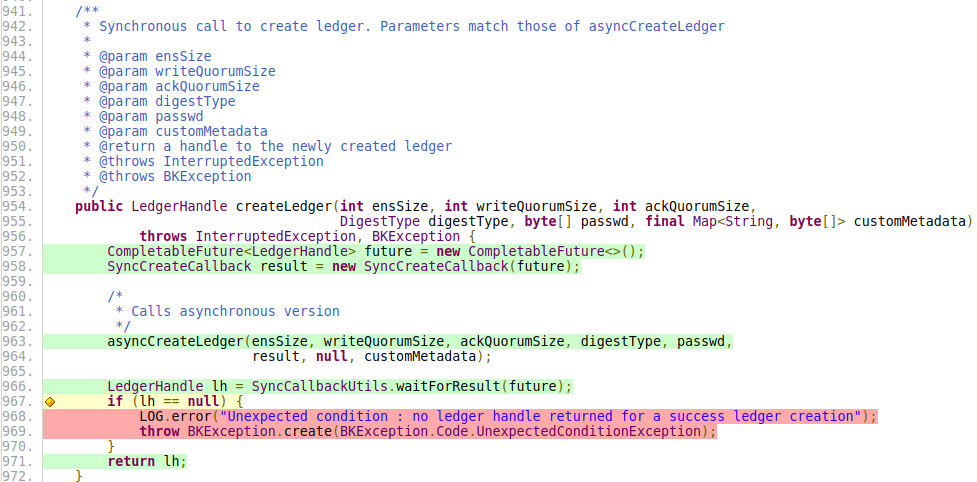


Figura Coverage del metodo createLedger() testato nella prima iterazione di test per la classe BookKeeper

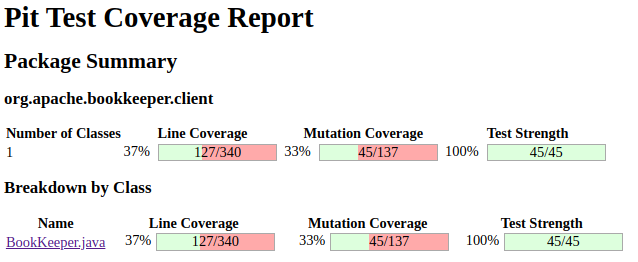


Figura Mutation coverage classe BookKeeper, iterazione 1



Figura L'invocazione del metodo deleteLedger() su un ledger non esistente ritorna con successo anziché lanciare un'eccezione

|  |  |  |
| --- | --- | --- |
| **Parametro** | **Classi di equivalenza** | **Boundary-values** |
| ensSize | {<qSize}, {=qSize}, {>qSize} | qSize -1, =qSize, qSize +1 |
| qSize | {<0}, {=0}, {>0} | -1, 0, 1 |
| digestType | {MAC}, {CRC32}, {CRC32C}, {DUMMY} | MAC, CRC32, CRC32C, DUMMY |
| passwd | {empty array}, {non empty array} | [], “passwd”.getBytes() |

Tabella Classi di equivalenza e boundary-values per il metodo createLedger(ensSize, qSize, digestType, passwd)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ensSize** | **qSize** | **digestType** | **passwd** | **Output atteso** |
| -2 | -1 | MAC | [] | exception |
| -1 | -1 | CRC32 | “passwd”.getBytes() | Exception |
| 0 | -1 | CRC32C | “passwd”.getBytes() | Handle valido |
| -1 | 0 | DUMMY | “passwd”.getBytes() | Exception |
| 0 | 0 | MAC | [] | Handle valido |
| 1 | 0 | MAC | “passwd”.getBytes() | Handle valido |
| 0 | 1 | MAC | “passwd”.getBytes() | Exception |
| 1 | 1 | MAC | “passwd”.getBytes() | Handle valido |
| 2 | 1 | MAC | “passwd”.getBytes() | Handle valido |

Tabella Casi di test per il metodo createLedger(ensSize, qSize, digestType, passwd)

|  |  |  |
| --- | --- | --- |
| **digestType** | **passwd** | **Output atteso** |
| MAC | [] | Handle valido |
| CRC32 | “passwd”.getBytes() | Handle valido |
| CRC32C | “passwd”.getBytes() | Handle valido |
| DUMMY | “passwd”.getBytes() | Handle valido |

Tabella Casi di test per il metodo createLedger(digestType, passwd)

|  |  |  |  |
| --- | --- | --- | --- |
| **lId** | **digestType** | **passwd** | **Output atteso** |
| ledgerHandle.getId() | MAC | “passwd”.getBytes() corretta | Handle valido |
| ledgerHandle.getId() | CRC32 | [] corretta | Handle valido |
| ledgerHandle.getId() | CRC32C | “wrongPassword”.getBytes() | BKexception |
| -1 | DUMMY | “passwd”.getBytes() corretta | BKException |
| -1 | MAC | [] corretta | BKException |
| -1 | CRC32 | “wrongPassword”.getBytes() | BKException |

Tabella Test cases per il metodo openLedger() della classe BookKeeper



Figura Class coverage JaCoCo per la classe BookKeeper, iterazione 2



Figura JaCoCo, method coverage classe BookKeeper, iterazione 2. Sono evidenziati i metodi complessivamente testati nelle due iterazioni

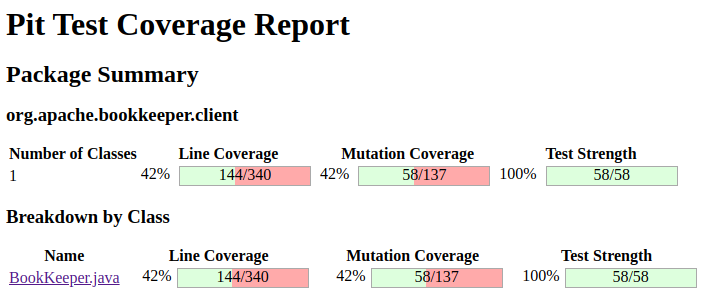


Figura Mutation coverage classe BookKeeper, iterazione 2

|  |  |
| --- | --- |
| **node** | **Output atteso** |
| null | Nodo non aggiunto, ma nessuna eccezione |
| DataNode(“127.0.0.1:4000”, “/root”) | Nodo correttamente aggiunto |
| InnerNode(“127.0.0.1:4001”, “/inner”) | IllegalArgumentException |
| DataNode(“127.0.0.1:4002”, “/rack/127.0.0.1:3999”) | IllegalArgumentException |

Tabella Test-cases per il metodo add() della classe NetworkTopologyImpl

|  |
| --- |
| @RunWith**(**Parameterized**.**class**)**  public class NetworkTopologyImplIT **{**  // SUT  private NetworkTopologyImpl sut**;**  private Node node**;**  public NetworkTopologyImplIT**(**Node node**)** **{**  configure**(**node**);**  **}**  public void configure**(**Node node**)** **{**  **this.**sut **=** **new** NetworkTopologyImpl**();**  **this.**node **=** node**;**  **}**  @Parameterized**.**Parameters  public static Collection**<**Object**[]>** parameters**()** **{**  **return** Arrays**.**asList**(new** Object**[][]{**  **{new** NodeBase**(**"127.0.0.1:9999"**,** "/rack"**)}**  **});**  **}**  @Test  public void testAddIT**()** **{**  node **=** Mockito**.**spy**(**node**);**  sut**.**add**(**node**);**  Mockito**.**verify**(**node**,** Mockito**.**times**(**1**)).**setParent**(**any**());**  **}**  **}** |

Codice Integration test tra Node e NetworkTopologyImpl, riguardante la correttezza del metodo add()

|  |  |
| --- | --- |
| **node** | **Output atteso** |
| null | Nessun valore di ritorno e nessun cambiamento della topologia |
| DataNode(“127.0.0.1:4000”, “/root”) | Nodo rimosso correttamente |
| InnerNode(“127.0.0.1:4001”, “/inner”) | IllegalArgumentException |

Tabella Casi di test per il metodo remove() di NetworkTopologyImpl



Figura JaCoCo, class coverage per NetworkTopologyImpl, iterazione 1

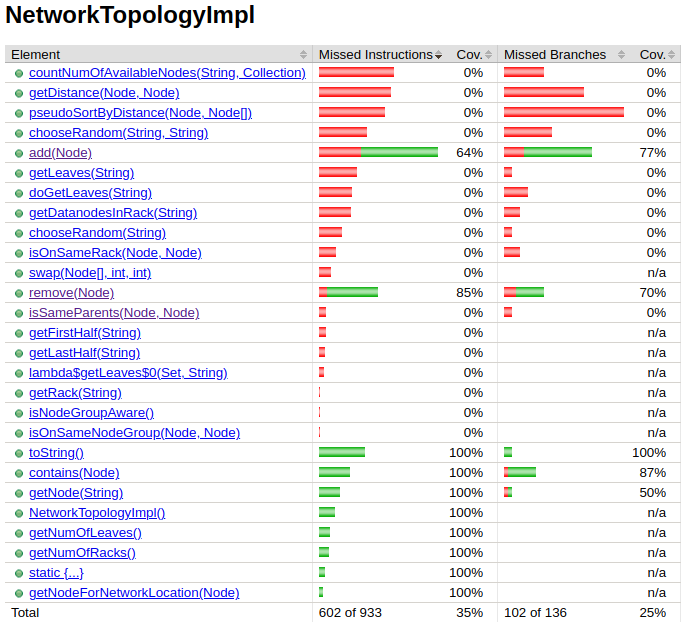


Figura JaCoCo method coverage, classe NetworkTopologyImpl, iterazione 1

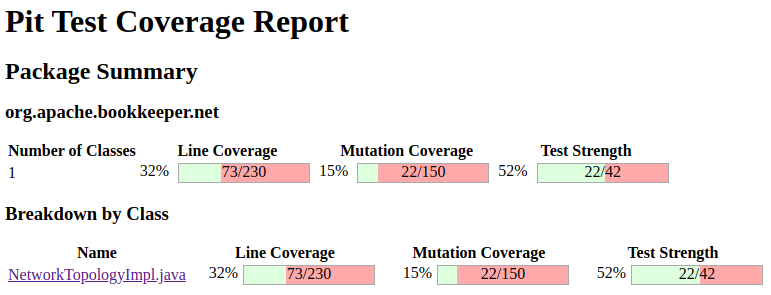


Figura Mutation coverage classe NetworkTopologyImpl, iterazione 1

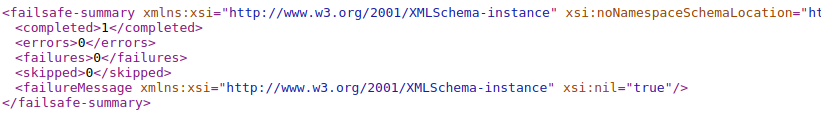


Figura Risultato esecuzione del test di integrazione con failsafe, andato a buon fine

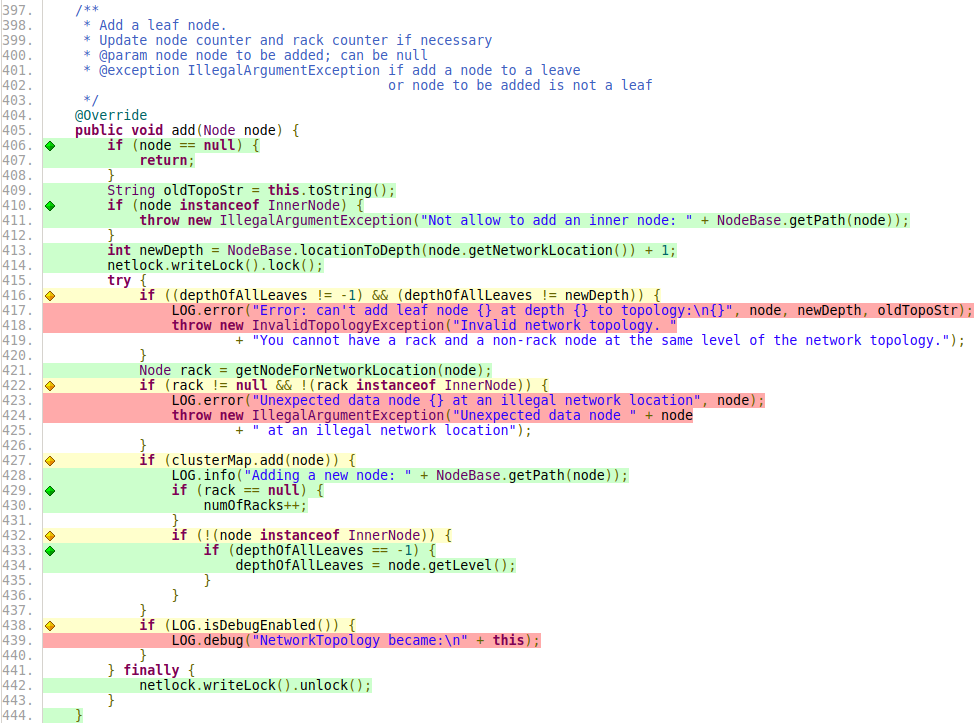


Figura JaCoCo coverage del metodo add() della classe NetworkTopologyImpl, iterazione 1



Figura Ba-dua coverage: metodo add(), classe NetworkTopologyImpl, iterazione 1