

Report ISW2 – Modulo Software Testing – Prof. G. De Angelis

Author : Domenico Verde, 0289890

Introduzione

Il presente documento descrive, come da titolo, il lavoro svolto durante il corso di Ingegneria del Software 2 (9 CFU); in particolare, il report in questione si riferisce al modulo sul Software Testing (3 CFU), svolto dal professore Guglielmo De Angelis. Tale lavoro consiste, sostanzialmente, nell'applicare delle tecniche di software testing mostrate durante il corso a due progetti open-source di larga scala Apache, valutando la qualità del lavoro svolto tramite l'ausilio di diversi frameworks. Il primo progetto "Bookkeeper" è comune a tutti gli studenti del corso, mentre il secondo "Syncope" mi è stato assegnato mediante un algoritmo proposto dal prof. Falessi.

Configurazione dell'ambiente di lavoro locale

L'ambiente di lavoro è costituito da una macchina ASUS con sistema operativo Ubuntu 18.04 LTS, sulla quale sono stati installati i tools fondamentali per il lavoro in questione, quali Junit, Maven, Git ed Eclipse come IDE. Inizialmente si è preferito svolgere il lavoro in locale; quindi, si è provveduto a clonare le repository Github dei due progetti sulla Scrivania della macchina. Dopodiché, seguendo le indicazioni e le procedure presenti nella documentazione e nelle varie developers guide dei due progetti, si è proceduto a fare la build di entrambi tramite il comando `mvn clean package -DskipTests`, avendo cura quindi di non eseguire i tests presenti nei progetti. Risolti alcuni piccoli problemi di configurazione dell'ambiente locale (ad esempio, si è dovuto configurare propriamente alcune variabili d'ambiente tra le quali JAVA_HOME, che inizialmente non permetteva la corretta esecuzione della build poiché non settata), si è provveduto ad eliminare le cartelle di test create dagli sviluppatori originali dei due progetti e ad editare le configurazioni e le dipendenze di test presenti nel pom.xml dei vari moduli, così da avere un progetto pulito su cui poter poi includere i test che verranno sviluppati in futuro. Infine, è stata rieseguita la build per verificare la corretta compilazione dei progetti anche a seguito delle modifiche effettuate.

Il progetto BOOKKEEPER: la scelta delle classi e motivazioni

BOOKKEEPER viene definito da Apache come "A scalable, fault-tolerant, and low-latency storage service optimized for real-time workloads". Esso si basa su alcuni concetti chiave quali Entries, Ledgers e Bookies. In particolare, esso consiste in un sistema distribuito composto da alcuni Clients (i quali quindi possono crashare o corrompere alcune delle informazioni) che, eseguendo l'applicazione, possono creare o scrivere in determinati streams che vengono chiamati Ledgers. Il processo di scrittura in tali ledgers avviene solo in append-mode. Ogni Ledger è composto da una sequenza di Entries ed ognuna di esse è sostanzialmente una sequenza di bytes. L'Entry è quindi l'unità di memoria fondamentale in Bookkeeper. I server in cui vengono memorizzati i Ledgers (o, a volte, parte di essi – per motivi di efficienza) vengono chiamati Bookie.

Si è scelto quindi di testare le funzionalità che consentono di creare un nuovo ledger, funzionalità espressa tramite il metodo `createLedger()` della classe `Bookkeeper.java`, e di leggere stream di bytes da un determinato ledger esistente tramite il metodo `read()` della classe `LedgerInputStream.java`.

La scelta è ricaduta su queste classi anche perché, analizzate le metriche del dataset creato nella parte del corso svolta dal Prof. Falessi, si è notato che: la prima, classe fondamentale per il progetto in questione, è composta da un alto numero di LOC (416) ed è stata fortemente analizzata nel corso di sviluppo del progetto, infatti possiede 98 autori distinti che hanno aggiunto circa 1492 righe di codice alla classe stessa e ciò potrebbe far supporre che la classe scelta abbia una certa robustezza: in effetti, ad oggi non esistono bugs che influenzano tale classe, anche se in passato ne sono stati scoperti diversi; mentre la seconda, anche se presenta un numero di linee di codice molto minore (182 LOC), è una classe molto anziana, ovvero è presente nel progetto da più di 343 settimane e nel tempo ad essa sono state aggiunte solamente 21 righe di codice in 6 revisioni. Ciò potrebbe far supporre che la classe sia stata progettata bene sin dall'inizio del progetto. Anch'essa nell'ultima versione del progetto considerato (6.0.0) non presenta bugs noti, anche se nel passato ne sono stati scoperti diversi che influenzavano tale classe.

Il progetto SYNCOPÉ: la scelta delle classi e motivazioni

SYNCOPÉ, invece, è un sistema open-source per la gestione delle identità digitali utilizzato soprattutto in ambienti enterprise. Per gestione delle identità digitali (o Identity Management) s'intendono quei servizi che consentono di gestire ed al tempo stesso di controllare gli accessi di utenti a dati sensibili di un sistema, evitando quindi eventuali accessi non autorizzati che potrebbero causare gravi danni ad utenti oppure all'organizzazione stessa. L'entità fondamentale del progetto in questione è l'User, che rappresenta appunto l'identità virtuale di un determinato soggetto. Ogni user può avere associati zero o più Accounts, che in Syncope vengono differenziati in Mapped Account e Linked Accounts. In genere, ogni account contiene determinate informazioni sull'utente a cui si riferisce, quali ad esempio username, password o e-mail. Una Policy consiste in un set di regole (Rules) che impongono, per una qualche ragione, dei vincoli sul formato di alcuni campi di un account; imporre determinati vincoli può risultare di notevole importanza in alcuni casi, ad esempio per mantenere un certo grado di sicurezza delle informazioni.

Le classi da testare scelte DefaultAccountRule.java e DefaultPasswordRule.java consentono di imporre alcuni vincoli rispettivamente su username e password di un account; tali classi, inoltre, consentono anche di verificare se i vincoli imposti vengono rispettati.

Le classi scelte, molto simili sia in termini di responsabilità che di funzionalità implementate, presentano anche valori delle metriche molto simili tra loro; le uniche differenze sostanziali si riscontrano in termini di LOC, ovvero, la seconda risulta avere molte più linee di codice della prima (134 vs. 233) e presenta anche molte più LOC Added della prima (30 vs. 177). La motivazione principale per cui sono state scelte tali classi è quella di valutare se, applicando diverse tecniche di software testing su entrambe, si riscontrano notevoli differenze tra i risultati prodotti da tali tecniche. In tal caso, potrebbe essere ragionevole supporre che tali differenze siano dovute ai diversi valori delle metriche LOC e LOC added.

Category Partition Testing

Individuate le classi e le funzionalità da testare, si è proceduto effettuando del testing tramite il category partition method sulle operazioni di Bookkeeper: createLedger(int E, int Qw, int Qack, DigestType d, byte[] password) e read(byte[] b, int offset, int length).

In particolare per quanto riguarda il metodo createLedgers(), dopo aver dato una prima bozza delle classi di equivalenza partizionando E, Qw e Qack negli interi positivi e non positivi, non conoscendo a fondo i restanti parametri, si è preferito procedere leggendo la documentazione riportata nel Link 1: qui si è notato che E (Ensemble size) rappresenta il numero di bookie, ovvero il numero di nodi in cui deve essere memorizzato il ledger, Qw (Write Quorum) rappresenta il numero di nodi in cui ogni entry deve essere memorizzata e Qack (Acknowledged Quorum) il numero di nodi che devono dare l'ack della avvenuta scrittura della entry. Per questo motivo deve valere la relazione $E \geq Qw \geq Qack$, altrimenti il metodo fallisce. Inoltre, leggendo la documentazione riportata nel Link 2, si è notato che la password è necessaria per la creazione di un ledger ed il suo valore di default è la stringa vuota "". Infine, analizzando il codice della classe Bookkeeper.java, si è notato che DigestType è un'enumerazione e possiede 4 stati, e rappresenta l'algoritmo utilizzato per valutare l'integrità della password. A questo punto, le classi di equivalenza sono state ridefinite considerando i vincoli appena descritti ed è stata condotta la boundary analysis, per identificare i confini di ogni partizione. In particolare, le partizioni definite risultano per E: {interi ≤ 0 ; interi > 0 }, per Qw: {interi $\leq E$; interi $> E$ }, per Qack: {interi $\leq Qw$; interi $> Qw$ }, considerando la relazione tra di essi riportata nella documentazione. Per d, invece, essendo necessario testare ogni possibile caso dell'enumerazione le partizioni risultano tutti i possibili stati ed inoltre si è preferito considerare anche il valore null. Per il parametro password, infine, si è preferito utilizzare due valori validi ovvero i bytes corrispondenti alla stringa vuota ed alla stringa "password", ed un valore non valido ovvero null. I boundary values ricavati a partire dalle partizioni e l'insieme minimale di test sono riportati nella tabella 1. Il primo test-case, essendo i valori di E, Qw e Qack tutti negativi, dovrebbe sollevare un'eccezione dato che si riferiscono alla cardinalità di insiemi e pertanto non possono assumere valori negativi. Anche il secondo test case dovrebbe sollevare un'eccezione, dato che la relazione $E \geq Qw \geq Qack$ non viene rispettata. Gli ultimi due casi, invece, dovrebbero terminare con la creazione di un nuovo ledger (rappresentato mediante un oggetto LedgerHandle), dato che tutti gli input rispettano i vincoli riportati nella documentazione.

Per quanto riguarda il metodo `read()`, invece, analizzando la documentazione di `bookkeeper` riportata nel Link 3 ed analizzando il codice della classe `LedgerInputStream`, si è notato che tale metodo è un override del metodo `read()` della classe `java InputStream`, perciò si è preferito dare uno sguardo anche a tale documentazione che è riportata nel Link 4. Tale metodo consente di leggere `length` bytes da un dato ledger e di posizzionarli nel buffer `b` a partire da un dato `offset`. Tale metodo, inoltre, dovrebbe ritornare il numero di bytes effettivamente letti ed inseriti nel buffer oppure `-1` se lo stream ha raggiunto la propria fine. Analizzando la classe e fissate le partizioni di `length`: $\{\text{interi} < 0; \text{interi} \geq 0\}$, si è osservato che, per quanto concerne `b`, esso può essere scelto in modo che i bytes da leggere rientrino senza problemi nell'array (se `b.size() ≥ length`) oppure in modo che non vi sia spazio a sufficienza per accomodare tutti i bytes letti (se `b.size() < length`). Inoltre, si è preferito includere il caso in cui esso sia `null`. Le partizioni di `b` a questo punto risultano `b`: $\{\text{null}, b.size() \geq length, b.size() < length\}$. Per quanto concerne l'`offset`, invece, si è osservato che, supposto che il buffer abbia spazio a sufficienza per memorizzare tutti i bytes letti dallo stream, esso possa consentire di scrivere nel buffer `b` tutti i bytes letti (se `offset ≤ b.size() - length`), solamente una porzione di essi (se `offset > b.size() - length`) oppure nessuno, nel caso in cui esso sia maggiore della dimensione massima del buffer (se `offset ≥ b.size()`). Inoltre, si è ritenuto necessario includere il caso in cui esso sia negativo. Quindi, le partizioni di `offset`: $\{\text{interi} < 0, \text{interi} > b.size() - length, \text{interi} \leq b.size() - length, \text{interi} \geq b.size()\}$. Quanto detto si traduce nei boundary values e quindi nel test set minimale riportato nella tabella 2. Seguendo quanto riportato nella documentazione Java citata in precedenza, il primo test-case dovrebbe sollevare un'eccezione. In particolare, l'eccezione sollevata dovrebbe essere di tipo `NullPointerException` dato che `b` è `null` oppure di tipo `IndexOutOfBoundsException` dato che sia l'`offset` e sia `length` sono negativi. Anche il secondo ed il quarto test-case, seguendo la documentazione, dovrebbero sollevare un'eccezione dato che `length > b.size() - offset`. In entrambi i casi l'eccezione sollevata dovrebbe essere di tipo `IndexOutOfBoundsException`. Nel terzo e nel quinto test-case, invece, tutti i parametri scelti dovrebbero consentire la corretta esecuzione dei casi di test: il metodo scelto quindi dovrebbe ritornare il numero di bytes letti dallo stream, ovvero 1 nel test-case n.3 e 2 nel test-case n.5.

Dopo aver progettato i casi di test per il progetto `Bookkeeper`, si è proceduto analizzando le classi del progetto `Syncope`. In particolare, si è scelto di testare il metodo `enforce(String clear, String username, Set<String> wordsNotPermitted)` della classe `DefaultPasswordRule` ed il metodo `enforce(String username, Set<String> wordsNotPermitted)` della classe `DefaultAccountRule`. Tali metodi consentono di verificare se, dati una password o username, essi rispettino determinate regole definite in precedenza. In caso affermativo, entrambi non ritornano alcun valore, essendo metodi `void`; altrimenti, in caso negativo, essi lanciano un'eccezione. Analizzando la documentazione riportata nei Link 5,6 si è notato che entrambi i metodi fanno uso di un'interfaccia ausiliaria di tipo `DefaultAccountRuleConf` o `DefaultPasswordRuleConf`, le quali permettono di settare alcune regole di default, ad esempio sulla lunghezza di username/password o sui tipi di carattere che questi ultimi possono/devono contenere. Per entrambe le interfacce ausiliarie sono stati opportunamente creati dei mock al fine di poter implementare una policy del tutto personale.

In particolare, per quanto riguarda la classe `DefaultPasswordRule`, configurabile tramite l'interfaccia `DefaultPasswordRuleConf`, si è scelto di implementare una policy molto semplice che ammette l'inserimento di password lunghe almeno 8 caratteri e non più lunghe di 16 caratteri. Tale scelta può essere giustificata dalla necessità, in alcune applicazioni reali, di mantenere una certa robustezza per la password ed allo stesso tempo una totale compatibilità con sistemi legacy. A primo impatto, basandosi esclusivamente sul tipo di dato dei parametri del metodo da testare, sono state definite le seguenti classi di equivalenza: `clear`: $\{\text{null}, "", \text{anyOtherString}\}$; `wordsNotPermitted`: $\{\text{null}, \text{new Set<String>}\}$. Poi, analizzando il codice, si è notato che il parametro `clear` rappresenta la password candidata (in chiaro). Tenendo conto della policy implementata e quindi considerando che ogni password valida deve essere lunga tra gli 8 ed i 16 caratteri, si è preferito partizionare `clear` per lunghezza: `clear`: $\{\text{clear.size()} < 8, 8 \leq \text{clear.size()} \leq 16, \text{clear.size()} > 16\}$. Per quanto riguarda `wordsNotPermitted`, invece, analizzando la documentazione riportata nel Link 6, si è notato che tramite tale parametro è possibile specificare una lista di parole che la password non deve assolutamente contenere. Se la password contiene oppure è identica ad una di quelle parole, allora essa viene giudicata non valida. Quindi, è possibile scegliere tale set in modo che esso contenga una sottostringa della password oppure in modo che non ne contenga alcuna. Inoltre, si è preferito includere anche il valore `null` per tale parametro. Quindi, le partizioni di `wordsNotPermitted` scelte risultano: `wordsNotPermitted`: $\{\text{invalid: null; valid: set.contains(aSubstringOf(password)), !set.contains(aSubstringOf(password))}\}$. Per quanto riguarda `username`, invece, le sue partizioni non sono state ridefinite. I boundary values e i casi di

test progettati sono riportati nella tabella 3. I primi quattro casi di test riportati dovrebbero sollevare un'eccezione: il primo poiché tutti i valori dei parametri risultano pari a null, pertanto tale test non risulta significativo; il secondo test-case poiché presenta un valore di wordsNotPermitted nullo, pertanto non è possibile accedere al set di parole proibite; il terzo poiché presenta una password troppo corta, ovvero con meno di 8 caratteri; il quarto poiché presenta un set contenente la stringa scelta come password. Quinto e sesto caso di test, invece, essendo tutti i parametri del metodo testato validi, si aspetta che vada a buon fine. Dato che il metodo scelto non ha valore di ritorno, si definirà il test superato se non viene sollevata alcuna eccezione. Il settimo ed ultimo caso di test dovrebbe sollevare un'eccezione poiché la password utilizzata è più lunga di 16 caratteri.

Per quanto riguarda la classe DefaultAccountRule, invece, si è scelto di implementare una policy che consente l'inserimento di soli caratteri lower-case nello username. È noto che le query di ricerca in alcuni databases SQL (utilizzanti determinati set di caratteri) siano case in-sensitive; ciò potrebbe causare diversi problemi di ambiguità: ad esempio nel distinguere l'utente "UserName" dall'utente "username". Ragione per cui diverse applicazioni esistenti, al fine di eliminare tale ambiguità implementano una policy simile. Tale policy è stata implementata mediante l'ausilio della classe DefaultAccountRuleConf. Inizialmente, le categorie per i parametri username e wordsNotPermitted sono state definite in modo molto simile a quanto fatto nel paragrafo precedente: username: {null, "", anyOtherString}; wordsNotPermitted: {null, new Set<String>}. Poi, analizzando la documentazione riportata nel Link 5 e tenendo conto della policy implementata, si è preferito ridefinire le categorie della variabile username nel modo seguente: username: {null, "", stringsAllLowercase, !stringsAllLowercase}. Per quanto riguarda il set wordsNotPermitted, anche in questo caso, è possibile scegliere esso in modo che contenga l'username o una porzione di esso oppure in modo che non contenga nessuna sub-string dello username. Per tanto, in modo molto simile a quanto fatto nel paragrafo precedente, le categorie di tale parametro risultano: wordsNotPermitted: {invalid: null; valid: set.contains(aSubstringOf(username)), !set.contains(aSubstringOf(password))}. Boundary Values e test-cases sono riportati in tabella 4. Nel primo test case, il metodo enforce() presenta tutti i parametri di input invalidi, pertanto dovrebbe sollevare un'eccezione. Nel secondo, tale metodo ha come valore del parametro username la stringa vuota ed essendo tale valore invalido anche in questo caso esso dovrebbe sollevare un'eccezione. Il terzo, invece, avendo un username valido e non essendo contenuto nel set di parole proibite, non dovrebbe sollevare alcuna eccezione. L'ultimo test case, infine, presentando un username con una lettera maiuscola ed essendo tale username contenuto nel set dovrebbe sollevare un'eccezione.

Adequacy of Tests: Coverage

Dopo aver implementato i tests generati nel paragrafo precedente tramite l'ausilio di Junit ed Eclipse, essi sono stati inclusi nel ciclo di build dei progetti, è stata verificata la corretta compilazione a seguito di tale inclusione ed è stato configurato opportunamente il plugin Jacoco per valutare la bontà di tali test set. Analizzando il report prodotto da Jacoco, in parte riportato in figura 1, 2, 3, 4 si è notato che, scelte come metriche di riferimento branch condition coverage e statement coverage automaticamente calcolate dal plugin, il valore di tali metriche risulta nella maggior parte dei casi non soddisfacente e per tanto, ad ora, alcuni test set risultano inadeguati per il SUT corrispondente. Per questo motivo, si è deciso di allargare tali test set, includendo ulteriori tests volti ad aumentare i valori di tali metriche.

Per quanto riguarda la classe Bookkeeper.java, osservando il report si è notato che il metodo considerato è in overload con altri 3 metodi della classe stessa, ed in particolare esso richiama il metodo createLedger(int E, int Qw, int Qack, DigestType d, byte[] password, Map m). Trattandosi di implementazioni della stessa funzionalità, si è deciso di analizzare ed eventualmente di aumentare il valore delle metriche di tutti i metodi in overload. Il test set progettato consente di raggiungere una percentuale di statement coverage del 100% per 3 di questi 4 metodi, tra cui anche il metodo testato in precedenza applicando il category partition method. Per quanto riguarda l'ultimo metodo, createLedger(int E, int Qw, int Qack, DigestType d, byte[] password, Map m), il test-set consente di raggiungere una percentuale di statement coverage del 81% ed una percentuale di branch coverage del 50%, come riportato in figura 1.1. Si è proceduto analizzando quindi il codice di tale metodo al fine di rilevare le parti di codice non coperte finora dal test-set: così facendo si è notato che un ramo di un if non viene eseguito in nessun caso di test, come riportato in figura 1.1.2. Si è deciso quindi di analizzare il codice della classe SyncCallbackUtils per cercare di capire se ed in che modo fosse possibile eseguire tale ramo. L'obiettivo è quindi quello di rendere vera la condizione dell'if, ovvero ottenere un valore di ritorno null per l'operazione waitForResult(). Osservando il codice di quest'ultimo

metodo, riportato in figura 1.1.3, appare evidente che è possibile ottenere tale valore solo se il metodo `get()` della classe `Future` va in timeout, ovvero se, passati circa 292 anni esso non restituisce alcun risultato. Non potendo aspettare il tempo necessario e non potendo in alcun modo utilizzare un mock, si dirà che tale parte di codice risulta non raggiungibile e che pertanto il test set risulta adeguato per il SUT. Per questo motivo, non è stato ritenuto necessario aggiungere alcun caso di test all'insieme creato in precedenza.

Per quanto riguarda la classe `LedgerInputStream`, invece, osservando il report prodotto da Jacoco e riportato in figura 2 si è notato che il test set creato consente di raggiungere, per il metodo `read(byte[], int, int)` testato in precedenza, una percentuale di statement coverage del 95% ed una percentuale di branch coverage del 75%. Osservando il codice del metodo, riportato in figura 2.1, si è notato che non è stato previsto nessun caso di test nel quale si è provato a leggere da uno stream che ha raggiunto la propria fine. In altre parole, in nessun caso il metodo ha ritornato il valore -1. Si è deciso perciò di includere un nuovo caso di test nel test set per coprire tale mancanza. In particolare, il nuovo caso di test introdotto, chiamato `test1A()`, consiste nel leggere continuamente da un `LedgerInputStream` finché non si raggiunge la fine dello stream e nel verificare il numero dei cicli effettuati. Inoltre, al fine di aumentare la percentuale di statement e di branch coverage dell'intera classe, ed al fine di valutare l'equivalenza tra i metodi `read()` in overload presenti nella classe stessa, si è preferito applicare il medesimo algoritmo di test anche utilizzando gli altri metodi `read()`. Inoltre, sempre al fine di aumentare il valore delle metriche considerate, si è preferito utilizzare anche l'altro costruttore della classe. Dopo aver implementato il test-case, esso è stato incluso nel ciclo di build del progetto ed è stato nuovamente analizzato il report prodotto da Jacoco, riportato in figura 2.2. A questo punto, essendo entrambe le percentuali di statement coverage e di branch coverage pari al 100% per tutti i metodi `read()` della classe, in particolare per il metodo `read(byte[], int, int)`, si può definire il test set adeguato per il SUT. Come è possibile notare dalle figure 2 e 2.2, le strategie utilizzate hanno consentito un notevole aumento dei valori delle metriche statement coverage (passato dal 41% al 85%) e branch coverage (passato dal 27% al 88%).

Come è possibile notare dalla figura 3, il report creato da Jacoco evidenzia come, anche per la classe `DefaultPasswordRule`, il test set generato precedentemente non consente di raggiungere i massimi valori delle metriche scelte come riferimento. In particolare, tale set consente di ricoprire il 43% di statement coverage ed il 31% di branch condition coverage per il metodo testato. Analizzando il codice prodotto, si è notato che il problema principale è che i casi di test non ricoprivano numerose condizioni di ogni `if` presente nella classe e ciò a causa della policy implementata. Per questo motivo, si è deciso di includere nel set ulteriori casi di test volti ad aumentare il valore di entrambe le metriche e quindi a coprire tutte le condizioni non ricoperte finora dal test-set generato precedentemente. In particolare, si è deciso di implementare il caso di test `passwordLengthTest()` per ricoprire tutte le condizioni e gli statement presenti nelle righe di codice 64-72 della classe e sinora non ricoperti. Tale test simula una situazione in cui non sono stati settati dei limiti sulla dimensione della password. In tal caso, si è verificato che il programma non solleva eccezioni inserendo una password di default. Si è deciso di implementare il test `usernameTest()` per simulare una policy che consenta/non consenta di inserire una password uguale all'username. Tale caso consente di ricoprire tutte le condizioni e gli statement non coperti presenti nelle righe 73-77. Il caso di test `digitRequiredTest()` consente di simulare una situazione in cui è richiesto l'inserimento di almeno una cifra nella password; in tal caso se viene inserita una password senza cifra si aspetta che venga sollevata un'eccezione altrimenti no. Tale test case consente di ricoprire le condizioni e gli statement non ricoperti presenti nelle righe 84-88. Il test `requiredLowercaseTest()` simula una situazione in cui viene utilizzata una politica che accetti una password solo se essa contiene almeno un carattere lowercase; in tal caso si simula l'inserimento di una password che rispetta tale politica ed una che non la rispetti. Questo test case consente di ricoprire le condizioni e gli statement non ricoperti presenti nelle righe 89-93. Il test `requiredUppercaseTest()`, molto simile al precedente, simula una politica che accetti passwords aventi almeno un carattere Uppercase. Esso consente di ricoprire le condizioni presenti nelle righe 94-98. I tests `prefixTest()` e `suffixTest()` sono volti a verificare, nel caso in cui vengano considerati non ammissibili alcuni prefissi e/o suffissi, se l'inserimento di password con tali morfemi generi un'eccezione. Essi consentono di ricoprire le condizioni e gli statement presenti nelle righe 99-112. Il test `alphanumericOccurrenceTest()` simula una situazione in cui viene utilizzata una politica che obbliga l'inserimento di una password con almeno un carattere alfanumerico/non alfanumerico. In entrambi i casi, vengono immesse una password che rispetta tale politica ed una che non la rispetti. Il test case consente di ricoprire le condizioni presenti nelle righe 131-140. I test-case rimanenti, identificati col prefisso "must", simulano situazioni in cui le password giudicate valide devono iniziare/non

iniziare, terminare/non terminare con una cifra, un carattere alfanumerico o un carattere non-alfanumerico. In ogni caso, si sono utilizzate una password che rispetti la regola definita ed una password che la violi. Ciò per coprire le condizioni alle righe 113-130 e 141-175. Come mostrato in figura 3.1., I test appena discussi consentono di raggiungere una percentuale di statement e di branch condition coverage pari al 100% per il metodo in questione e per tanto il test set, dopo l'aggiunta di tali casi di test, risulta adeguato per il nostro SUT.

Infine, si discuterà l'adeguatezza del test-set creato per la classe `DefaultAccountRule`. Il report generato da Jacoco, riportato in figura 4, mostra le percentuali di statement coverage - 60% e di condition coverage - 40% raggiunte grazie a tale test-set. In modo molto simile a quanto fatto per l'altra classe del progetto Syncope, è stato analizzato il report per capire quali condizioni e quali statement non venivano ricoperti dal test set. I test case riportati di seguito, quindi, sono volti ad aumentare i valori delle metriche di riferimento scegliendo diverse ed opportune policy da implementare. In particolare, il test `usernameLengthTest()` consente di simulare una situazione in cui in atto vi sia una politica che consenta l'esclusivo inserimento di usernames con più di 4 caratteri e meno di 6 caratteri e simula l'inserimento da parte di un utente di un username troppo corto, troppo lungo o corretto secondo tali regole. Consente di ricoprire tutti gli statement e le condizioni sinora non ricoperti dal test set, ma presenti nelle righe 55-64 della classe in questione. I metodi `allUppercaseTest()` e `allLowerCaseTest()` simulano una politica in cui è possibile inserire nell'username solo caratteri maiuscoli o minuscoli e verifica il corretto comportamento dell'applicazione nel caso in cui vengano inseriti input che rispettino/non rispettino tale policy. Essi consentono di ricoprire gli statement e le condizioni non ricoperte alle righe 71-79. I casi di test `prefixTest()`, `suffixTest()` e `patternTest()`, come nel caso precedente, sono volti a verificare il corretto comportamento del sistema nel caso in cui l'username contenga determinati prefissi, suffissi e/o pattern non siano considerati ammissibili. Consentono di ricoprire gli statement e le condizioni presenti nelle righe 80-89. Anche in questo caso, il test set con l'aggiunta degli ulteriori casi di test appena descritti, consente di aumentare i valori delle metriche branch condition coverage e statement coverage per il metodo `enforce(string, string, set<string>)` fino al 100%, come mostrato in figura 4.1.

Tutte le classi scelte non hanno mostrato sinora difetti o particolari criticità.

Mutation Testing

Infine, si provvederà ad utilizzare la tecnica del mutation testing sulle classi finora considerate. Tale tecnica, consiste nell'applicare mutazioni a parti di codice delle classi testate e nel verificare se i tests creati in precedenza falliscano in presenza di tali mutazioni. Nel caso in cui esistano mutanti non uccisi dal plugin, potrebbe essere ragionevole supporre che vi siano difetti nelle classi oppure che i test-set non siano sufficientemente robusti. In particolare, tale tecnica verrà implementata tramite il framework Pitest, il quale è stato integrato nei pom.xml dei moduli `bookkeeper-server` e `syncope-core-spring`. Il report è stato generato eseguendo tramite maven il goal `mvn org.pitest:pitest-maven:mutationCoverage`.

Per quanto riguarda i metodi `createLedger()`, il report non evidenzia particolari problematiche riguardanti il test set. In particolare, come è possibile notare dalla figura 5, tutti i mutanti creati vengono uccisi ed una delle mutazioni applicate ha causato un time-out, rivelando quindi anche in questo caso il mutante creato. In effetti, eliminando la chiamata al metodo `asyncCreateLedger()` il programma resta in attesa indefinita poiché il metodo `waitForResult()` non può restituire un oggetto `LedgerHandle` che non verrà mai creato proprio a causa di tale rimozione. Si ritiene quindi che l'insieme di test progettato risulta sufficientemente robusto, dato che il mutation score calcolato considerando i soli metodi testati risulta essere pari ad 1 (6 mutanti rivelati su 6 mutanti totali).

Per quanto riguarda i metodi `read()`, invece, il report rivela numerose mancanze nel test set sinora generato. Osservando tale report prodotto e riportato in figura 6, esso evidenzia chiaramente come non siano stati effettuati controlli sui valori di ritorno di ognuno dei metodi `read()` testati. In particolare, si è provveduto quindi ad aggiornare il test-case `test1A()` aggiungendo ulteriori asserzioni: per quanto riguarda il metodo `read()`, si è provveduto a verificare che ritorni, ogni volta che viene chiamato, lo stesso carattere presente in una data posizione del ledger. Per quanto riguarda i metodi rimanenti, invece, si è provveduto a verificare che essi ritornino il numero di caratteri effettivamente letti e che in `b` siano presenti effettivamente i caratteri della stringa presente nel ledger. Inoltre, si è provveduto a verificare che tutti i metodi considerati ritornino -1 nel

caso di EOS. Tali strategie hanno permesso un netto miglioramento del mutation score che, se calcolato considerando le sole mutazioni applicate ai metodi testati, risulta essere pari ad 1, come è possibile notare da figura 6.1.

Per quanto riguarda il progetto Syncope, si è riscontrato, oltre a diverse carenze di robustezza nei test-set implementati, la presenza di mutanti equivalenti in entrambe le classi scelte. In ognuna di esse sono presenti dei controlli che permettono di verificare se un dato username/password rispetti determinati vincoli sulla lunghezza minima consentita. È possibile verificare se sono stati settati tali vincoli tramite il metodo `getMinLength()` della classe `Default*RuleConf` utilizzata per implementare le regole della policy: secondo la documentazione ufficiale del progetto, riportata nei Link 5,6, se non vi sono limiti sulla lunghezza minima di essi allora il metodo deve ritornare il valore 0, altrimenti un intero positivo che rappresenta la lunghezza minima ammissibile in caratteri. Osservando il report prodotto da Pitest ed in parte riportato nelle figure 7, 8 si è notato che, mutando la condizione `conf.getMinLength() > 0` in `≥ 0`, il mutante creato sopravvive ai test sinora implementati. Analizzando il codice di entrambe le classi si è notato che la condizione appena mutata è in AND con un'ulteriore condizione, ovvero `conf.getMinLength() > username.length()` in un caso e `conf.getMinLength() > clear.length()` nell'altro caso. Entrambe le condizioni considerate sono visibili alla riga 56 della classe `DefaultAccountRule` ed alla riga 65 della classe `DefaultPasswordRule`. Si è deciso perciò di provare a progettare un nuovo caso di test volto a distinguere il comportamento del programma originale dal mutante, per poterlo così poi uccidere. Innanzitutto, se il metodo `conf.getMinLength()` ritorna un valore > 0 allora esso è anche ≥ 0 , pertanto per ogni valore positivo il comportamento dei due programmi è identico; quindi, nell'eventuale caso di test che si andrà a creare si configurerà la relativa classe `Default*RuleConf` in modo che `conf.getMinLength()` ritorni il valore 0. In tal caso, il programma originale, essendo falsa la prima condizione non valterebbe nemmeno l'altra (a causa delle tecniche di lazy evaluation – short circuits implementate in Java), mentre il mutante, essendo la condizione vera in tal caso, verificherebbe l'altra condizione e ciò aprirebbe la strada ad un possibile differente comportamento dei due programmi. Tuttavia, se il valore ritornato dal metodo è 0, allora l'altra condizione sarà $0 < \text{username.length}()$ oppure $0 < \text{password.length}()$, a seconda della classe. Seguendo quanto affermato nella documentazione Java il metodo `String.length()` non può produrre un risultato negativo, dato che si riferisce al numero di caratteri presenti in una stringa; pertanto, tale condizione risulterà falsa per qualsiasi username o password inserito. Per questo motivo, quindi, si ritiene che il mutante creato sia equivalente al programma originale: non è possibile generare alcun caso di test che consenta di soddisfare la proprietà di infezione dello stato, essendo in ogni caso possibile lo stato del mutante identico a quello del programma originale.

Per quanto riguarda le ulteriori mutazioni applicate sul metodo `enforce()` della classe `DefaultAccountRule`, dal report Pitest in parte visibile in Figura 7 si è notato che alcuni dei mutanti sopravvissuti riguardavano prefissi e suffissi non ammissibili, come è possibile notare alle righe 88, 95 di tale figura. Per sopperire a tali carenze, si è preferito modificare i tests `prefixTest()` e `suffixTests()` aggiungendo in essi una parte in cui viene immesso un username non avente prefissi o suffissi proibiti, asserendo che il metodo testato non sollevasse alcuna eccezione. Inoltre, al fine di uccidere i mutanti generati mutando le boundary condition delle righe 56, 61, si è modificato il test `usernameLengthTest()`, il quale ora simula l'immissione di usernames di 3,4,6,7 caratteri nell'applicazione, utilizzando una policy che giudica validi esclusivamente gli usernames con un numero di caratteri compreso tra 4 e 6. I nuovi test così implementati consentono di raggiungere un mutation score pari ad 1, essendo 22 i mutanti uccisi, un mutante equivalente al SUT e 23 i mutanti totali creati, come è possibile notare in figura 7.1.

Infine, per quanto riguarda il metodo `enforce()` della classe `DefaultPasswordRule` si è agito in maniera molto simile a quanto fatto per l'omonimo metodo dell'altra classe del progetto: si sono modificati i due metodi `prefixTest()` e `suffixTest()` aggiungendo una parte di codice che permette di testare usernames non contenenti prefissi o suffissi ammissibili. Tali modifiche hanno consentito di uccidere i mutanti generati a seguito delle mutazioni effettuate alle righe 101, 108 della Figura 8. Inoltre, si è aggiunto il metodo `wordsNotPermittedTest()` per coprire l'ultima mutazione rimasta. Tale caso di test consente di valutare il comportamento del metodo quando viene inserito un username contenente/non contenente parole proibite, non sollevando/solevando eccezioni. Il mutation score finale, anche in quest'ultimo caso risulta essere pari ad 1, essendo 53 i mutanti uccisi, 1 mutante equivalente e 54 i mutanti totali.

Github, TravisCI & SonarCloud

E' possibile visionare le repositories Github contenenti i progetti ed i tests implementati utilizzando i Link 7,8. Tali repositories sono state aggiornate ogni qualvolta venivano apportate modifiche sostanziali ai progetti, mentre si è preferito configurare i tools TravisCI e SonarCloud solamente una volta applicate tutte le tecniche di software testing e dopo aver apportato tutte le modifiche suggerite da tali tecniche. Al fine di risparmiare notevole tempo nella compilazione dei progetti (non è stato possibile effettuare automaticamente la compilazione, l'esecuzione dei test e l'analisi dei report a causa di alcune limitazioni sulla lunghezza dei jobs in TavisCI), si è preferito caricare nelle repositories i binari dei moduli dei progetti considerati ed i reports generati da Jacoco, consentendo a SonarCloud di analizzare tali report e tali binari mediante TravisCI. Tale configurazione è stata implementata mediante il file .travis.yml, in cui si è preferito utilizzare gli Advanced Reactors messi a disposizione da maven per consentire a SonarCloud l'analisi dei soli moduli toccati dal testing effettuato. Inoltre, al fine di permettere una sicura e corretta comunicazione tra questi tools, si è preferito utilizzare un token generato da SonarCloud e criptato mediante l'applicazione desktop del tool TravisCI. Per quanto riguarda i metodi testati, i risultati delle analisi prodotte da SonarCloud sono in linea con quanto riportato dal tool Jacoco.

Conclusioni

Dal lavoro svolto è emerso che il Category Partition Method, metodo sistematico per la creazione di test-cases, è risultato un ottimo punto di partenza ed ha consentito di generare buoni casi di test per le funzionalità considerate. Tali casi di test generati, nella maggior parte dei casi, non hanno consentito di raggiungere i massimi valori ottenibili per le metriche di adeguatezza statement coverage e condition coverage. È risultato quindi necessario aggiungere ulteriori casi di test al fine di rendere i test-set adeguati secondo tali metriche. Inoltre, anche se i valori di entrambe le metriche risultano massimi, non è detto che tali test-set siano sufficientemente robusti. Infatti, il mutation testing è un'ottima tecnica che consente di valutare e quindi migliorare la robustezza dei test-set in maniera completamente automatizzata. Al termine del lavoro svolto, non sono stati riscontrati particolari bugs o difetti delle classi e delle funzionalità scelte. Non essendo stati riscontrati bugs, non è stato nemmeno possibile effettuare correlazioni tra tali bugs e le metriche del dataset utilizzate per la scelta delle classi.

Links

1. <https://bookkeeper.apache.org/docs/4.5.1/development/protocol/#ledger-metadata>
2. <https://bookkeeper.apache.org/docs/4.5.1/reference/config/>
3. <https://bookkeeper.apache.org/archives/docs/r4.0.0/bookkeeperStream.html>
4. [https://docs.oracle.com/javase/7/docs/api/java/io/InputStream.html#read\(byte\[\],%20int,%20int\)](https://docs.oracle.com/javase/7/docs/api/java/io/InputStream.html#read(byte[],%20int,%20int))
5. https://ci.apache.org/projects/syncope/2_1_X/reference-guide.html#default-account-rule
6. https://ci.apache.org/projects/syncope/2_1_X/reference-guide.html#default-password-rule
7. <https://github.com/DomenicoVerde/Bookkeeper>
8. <https://github.com/DomenicoVerde/Syncope>

Tabelle

Test Case	Function Parameters (int E, int Qw, int Qack, DigestType d, byte[] password)	Expected Result
1	-1,-2,-3, DigestType.DUMMY, null	Exception Thrown
2	0,1,2, DigestType.CRC32C, "".getBytes()	Exception Thrown
3	1,1,1, DigestType.CRC32, "password".getBytes()	New LedgerHandle Object
4	1,1,1, DigestType.MAC, "".getBytes()	New LedgerHandle Object
5	1,1,0, null, "password".getBytes()	Exception Thrown

Tabella 1. Category Partition Oracle for createLedgers(). I boundary values a cui si riferiscono i tests sono riportati di seguito: E = {-1, 0, 1}; Qw = {E - 1, E, E + 1}; Qack = {Qw - 1, Qw, Qw + 1}; d = {null, MAC, CRC32, CRC32C, DUMMY}; password = {valid: "", "password"; invalid: null}.

Test Case	Function Parameters (byte[] b, int offset, int length)	Expected Result
1	null, -1, -1	Exception Thrown
2	new byte[0], 1, 0	Exception Thrown
3	new byte[2], 1, 1	1
4	new byte[3], 0, 4	Exception Thrown
5	new byte[4], 1, 2	2

Tabella 2. Category Partition Oracle for read(). I boundary values sono riportati di seguito: b = {invalid: null; valid: b.size() = length+1; length; length-1}; offset = {-1, b.size() - length -1, b.size() - length; b.size() - length + 1, b.size() + 1}; length = {-1, 0, 1};

Test Case	Function Parameters (String clear, String username, Set<String> wordsNotPermitted)	Expected Result
1	null, null, null	Exception Thrown
2	"123456789", "", null	Exception Thrown
3	"1234567", "domenico", "<"abc", "def">	Exception Thrown
4	"12345678", "domenico", "<"abc", "def", "12345678",>	Exception Thrown
5	"123456789012345", "domenico", <>	No Exceptions Thrown
6	"1234567890123456", "domenico", <>	No Exceptions Thrown
7	"12345678901234567", "domenico", <>	Exception Thrown

Tabella 3. Category Partition Oracle for DefaultPasswordRule.enforce(). I boundary values sono riportati di seguito: clear = {invalid: null, clear.length() = 7, clear.length() = 17; valid: clear.length() = 8, clear.length() = 9, clear.length() = 15, clear.length() = 16}; username = {invalid: null, ""; valid: "domenico"}; wordsNotPermitted: {invalid: null; valid: set.contains(password), new Set()};.

Test Case	Function Parameters (String username, Set<String> wordsNotPermitted)	Expected Result
1	null, null	Exception Thrown
2	"", "<"word", "123">	Exception Thrown
3	"domenico", <>	No Exceptions Thrown
4	"Domenico", "<"Domenico">	Exception Thrown

Tabella 4. Category Partition Oracle for DefaultAccountRule.enforce(). I boundary values sono riportati di seguito: username: {invalid: null, "", "Domenico"; valid: "domenico"}; wordsNotPermitted: {invalid: null; valid: set.contains(username), !set.contains(username)};

Figure




BookieWatcherImpl.new RemovalListener() {...}		50%		n/a	1	2
BookKeeper		38%		25%	66	88
BookKeeper.Builder		0%		n/a	12	12

Figura 1. Report Jacoco per la classe Bookkeeper.java




































BookKeeper						
Element		Missed Instructions =	Cov. =	Missed Branches =	Cov. =	
asyncCreateLedger(int, int, BookKeeper.DigestType, byte[], AsyncCallback.CreateCallback, Object)			0%		n/a	
asyncCreateLedger(int, int, int, BookKeeper.DigestType, byte[], AsyncCallback.CreateCallback, Object, Map)			85%		75%	
asyncCreateLedgerAdv(int, int, int, BookKeeper.DigestType, byte[], AsyncCallback.CreateCallback, Object, Map)			0%		0%	
asyncCreateLedgerAdv(long, int, int, int, BookKeeper.DigestType, byte[], AsyncCallback.CreateCallback, Object, Map)			0%		0%	
asyncDeleteLedger(long, AsyncCallback.DeleteCallback, Object)			0%		0%	
asyncIsClosed(long, AsyncCallback.IsClosedCallback, Object)			0%		n/a	
asyncOpenLedger(long, BookKeeper.DigestType, byte[], AsyncCallback.OpenCallback, Object)			0%		0%	
asyncOpenLedgerNoRecovery(long, BookKeeper.DigestType, byte[], AsyncCallback.OpenCallback, Object)			0%		0%	
BookKeeper()			0%		n/a	
BookKeeper(ClientConfiguration)			100%		n/a	
BookKeeper(ClientConfiguration, ZooKeeper)			0%		n/a	
BookKeeper(ClientConfiguration, ZooKeeper, EventLoopGroup)			0%		n/a	
BookKeeper(ClientConfiguration, ZooKeeper, EventLoopGroup, ByteBufferAllocator, StatsLogger, DNSToSwitchMapping, HashedWheelTimer, FeatureProvider)			73%		50%	
BookKeeper(String)			100%		n/a	
checkForFaultyBookies()			0%		0%	
close()			72%		42%	
createLedger(BookKeeper.DigestType, byte[])			100%		n/a	
createLedger(int, int, BookKeeper.DigestType, byte[])			100%		n/a	
createLedger(int, int, int, BookKeeper.DigestType, byte[])			100%		n/a	
createLedger(int, int, int, BookKeeper.DigestType, byte[], Map)			81%		50%	
createLedgerAdv(int, int, int, BookKeeper.DigestType, byte[])			0%		n/a	
createLedgerAdv(int, int, int, BookKeeper.DigestType, byte[], Map)			0%		0%	
createLedgerAdv(long, int, int, int, BookKeeper.DigestType, byte[], Map)			0%		0%	

Figura 1.1. Report Jacoco dettagliato per alcuni metodi della classe Bookkeeper.java

```

public LedgerHandle createLedger(int ensSize, int writeQuorumSize, int ackQuorumSize,
                                DigestType digestType, byte[] passwd, final Map<String, byte[]> customMetadata)
    throws InterruptedException, BKException {
    CompletableFuture<LedgerHandle> future = new CompletableFuture<>();
    SyncCreateCallback result = new SyncCreateCallback(future);

    /*
     * Calls asynchronous version
     */
    asyncCreateLedger(ensSize, writeQuorumSize, ackQuorumSize, digestType, passwd,
        result, null, customMetadata);

    LedgerHandle lh = SyncCallbackUtils.waitForResult(future);
    if (lh == null) {
        LOG.error("Unexpected condition : no ledger handle returned for a success ledger creation");
        throw BKException.create(BKException.Code.UnexpectedConditionException);
    }
    return lh;
}

```

Figura 1.1.2 . Coverage del metodo createLedger() della classe BookKeeper.java

```

45     public static <T> T waitForResult(CompletableFuture<T> future) throws InterruptedException, BKException {
46         try {
47             try {
48                 /*
49                  * CompletableFuture.get() in JDK8 spins before blocking and wastes CPU time.
50                  * CompletableFuture.get(long, TimeUnit) blocks immediately (if the result is
51                  * not yet available). While the implementation of get() has changed in JDK9
52                  * (not spinning any more), using CompletableFuture.get(long, TimeUnit) allows
53                  * us to avoid spinning for all current JDK versions.
54                  */
55                 return future.get(Long.MAX_VALUE, TimeUnit.NANOSECONDS);
56             } catch (TimeoutException eignore) {
57                 // it's ok to return null if we timeout after 292 years (2^63 nanos)
58                 return null;
59             }
60         } catch (ExecutionException err) {
61             if (err.getCause() instanceof BKException) {
62                 throw (BKException) err.getCause();
63             } else {
64                 BKException unexpectedConditionException =
65                     BKException.create(BKException.Code.UnexpectedConditionException);
66                 unexpectedConditionException.initCause(err.getCause());
67                 throw unexpectedConditionException;
68             }
69         }
70     }
71 }
72

```

Figura 1.1.3. Codice del metodo WaitForResult() della classe SyncCallbackUtils.

Apache Bookkeeper :: Tests > bookkeeper-server > org.apache.bookkeeper.streaming > LedgerInputStream

LedgerInputStream

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods
● refill()	<div><div></div></div>	35%	<div><div></div></div>	33%	3 4	11 18	0 1
● LedgerInputStream(LedgerHandle, int)	<div><div></div></div>	0%	<div><div></div></div>	n/a	1 1	12 12	1 1
● read(byte[])	<div><div></div></div>	0%	<div><div></div></div>	0%	3 3	10 10	1 1
● read()	<div><div></div></div>	0%	<div><div></div></div>	0%	3 3	7 7	1 1
● read(byte[], int, int)	<div><div></div></div>	95%	<div><div></div></div>	75%	1 3	1 10	0 1
● close()	<div><div></div></div>	0%	<div><div></div></div>	n/a	1 1	1 1	1 1
● LedgerInputStream(LedgerHandle)	<div><div></div></div>	100%	<div><div></div></div>	n/a	0 1	0 12	0 1
Total	165 of 282	41%	13 of 18	27%	12 16	38 66	4 7

Figura 2 . Report Jacoco per la classe LedgerInputStream.java

```

157.  @Override
158.  public synchronized int read(byte[] b, int off, int len) throws IOException {
159.      // again dont need ot fully
160.      // fill b, just return
161.      // what we have and let the application call read
162.      // again
163.      boolean toread = true;
164.  ◆   if (bytebuff.remaining() == 0) {
165.          toread = refill();
166.      }
167.  ◆   if (toread) {
168.          int bcopied = bytebuff.remaining();
169.          int tocopy = Math.min(bcopied, len);
170.          System.arraycopy(bbytes, bytebuff.position(), b, off, tocopy);
171.          bytebuff.position(bytebuff.position() + tocopy);
172.          return tocopy;
173.      }
174.      return -1;
175.  }
176. }

```

Figura 2.1. Coverage del metodo read() della classe LedgerInputStream.java.

LedgerInputStream

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
● refill()		47%		66%	2	4	10	18	0	1
● LedgerInputStream(LedgerHandle)		100%		n/a	0	1	0	12	0	1
● LedgerInputStream(LedgerHandle, int)		100%		n/a	0	1	0	12	0	1
● read(byte[])		100%		100%	0	3	0	10	0	1
● read(byte[], int, int)		100%		100%	0	3	0	10	0	1
● read()		100%		100%	0	3	0	7	0	1
● close()		100%		n/a	0	1	0	1	0	1
Total	41 of 282	85%	2 of 18	88%	2	16	10	66	0	7

Figura 2.2 . Report Jacoco per la classe LedgerInputStream.java dopo aver rivalutato l'adeguatezza del test set.

DefaultPasswordRule

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
● enforce(String, String, Set)		43%		31%	39	42	18	50	0	1
● enforce(LinkedAccount)		0%		0%	4	4	27	27	1	1
● enforce(User)		0%		0%	3	3	12	12	1	1
● setConf(PasswordRuleConf)		47%		50%	1	2	2	5	0	1

Figura 3. Report Jacoco per la classe DefaultPasswordRule.

● enforce(String, String, Set)		100%		100%	0	42	0	50	0	1
--	--	------	--	------	---	----	---	----	---	---

Figura 3.1. Report Jacoco per il metodo enforce() della classe DefaultPasswordRule dopo aver rivalutato l'adeguatezza del test-set.

DefaultAccountRule

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
● enforce(String, Set)		60%		40%	9	11	4	21	0	1
● enforce(LinkedAccount)		0%		0%	2	2	13	13	1	1
● enforce(User)		0%		n/a	1	1	11	11	1	1

Figura 4. Report Jacoco per la classe DefaultAccountRule.

● enforce(String, Set)		100%		100%	0	11	0	21	0	1
--	--	------	--	------	---	----	---	----	---	---

Figura 4.1. Report Jacoco per il metodo enforce della classe DefaultAccountRule dopo aver rivalutato l'adeguatezza del test-set.

851	1. replaced return value with null for org/apache/bookkeeper/client/BookKeeper::createLedger → KILLED
870	1. replaced return value with null for org/apache/bookkeeper/client/BookKeeper::createLedger → KILLED
890	1. replaced return value with null for org/apache/bookkeeper/client/BookKeeper::createLedger → KILLED
915	1. removed call to org/apache/bookkeeper/client/BookKeeper::asyncCreateLedger → TIMED_OUT
919	1. negated conditional → KILLED
923	1. replaced return value with null for org/apache/bookkeeper/client/BookKeeper::createLedger → KILLED

Figura 5. Report Pitest per i metodi createLedger().

130	1. Replaced bitwise AND with OR → SURVIVED
131	1. replaced int return with 0 for org/apache/bookkeeper/streaming/LedgerInputStream::read → SURVIVED
133	1. replaced int return with 0 for org/apache/bookkeeper/streaming/LedgerInputStream::read → TIMED_OUT
142	1. negated conditional → KILLED
145	1. negated conditional → KILLED
150	1. removed call to java/lang/System::arraycopy → SURVIVED
151	1. Replaced integer addition with subtraction → KILLED
152	1. replaced int return with 0 for org/apache/bookkeeper/streaming/LedgerInputStream::read → SURVIVED
154	1. replaced int return with 0 for org/apache/bookkeeper/streaming/LedgerInputStream::read → TIMED_OUT
164	1. negated conditional → KILLED
167	1. negated conditional → KILLED
170	1. removed call to java/lang/System::arraycopy → KILLED
171	1. Replaced integer addition with subtraction → KILLED
172	1. replaced int return with 0 for org/apache/bookkeeper/streaming/LedgerInputStream::read → KILLED
174	1. replaced int return with 0 for org/apache/bookkeeper/streaming/LedgerInputStream::read → TIMED_OUT

Figura 6. Report Pitest per i metodi read().

125	1. negated conditional → KILLED
129	1. negated conditional → KILLED
130	1. Replaced bitwise AND with OR → KILLED
131	1. replaced int return with 0 for org/apache/bookkeeper/streaming/LedgerInputStream::read → KILLED
133	1. replaced int return with 0 for org/apache/bookkeeper/streaming/LedgerInputStream::read → KILLED
142	1. negated conditional → KILLED
145	1. negated conditional → KILLED
150	1. removed call to java/lang/System::arraycopy → KILLED
151	1. Replaced integer addition with subtraction → KILLED
152	1. replaced int return with 0 for org/apache/bookkeeper/streaming/LedgerInputStream::read → KILLED
154	1. replaced int return with 0 for org/apache/bookkeeper/streaming/LedgerInputStream::read → KILLED
164	1. negated conditional → KILLED
167	1. negated conditional → KILLED
170	1. removed call to java/lang/System::arraycopy → KILLED
171	1. Replaced integer addition with subtraction → KILLED
172	1. replaced int return with 0 for org/apache/bookkeeper/streaming/LedgerInputStream::read → KILLED
174	1. replaced int return with 0 for org/apache/bookkeeper/streaming/LedgerInputStream::read → KILLED

Figura 6.1. Report Pitest per i metodi read(), dopo aver migliorato la robustezza del test set.

56	1. changed conditional boundary → SURVIVED 2. changed conditional boundary → SURVIVED 3. negated conditional → KILLED 4. negated conditional → KILLED
61	1. changed conditional boundary → KILLED 2. changed conditional boundary → SURVIVED 3. negated conditional → KILLED 4. negated conditional → KILLED
67	1. replaced boolean return with false for org/apache/syncope/core/spring/policy/DefaultAccountRule::lambda\$enforce\$0 → KILLED 2. replaced boolean return with true for org/apache/syncope/core/spring/policy/DefaultAccountRule::lambda\$enforce\$0 → KILLED
68	1. removed call to java/util/stream/Stream::forEach → KILLED
73	1. negated conditional → KILLED 2. negated conditional → KILLED
76	1. negated conditional → KILLED 2. negated conditional → KILLED
81	1. negated conditional → KILLED
82	1. negated conditional → KILLED
88	1. replaced boolean return with false for org/apache/syncope/core/spring/policy/DefaultAccountRule::lambda\$enforce\$2 → KILLED 2. replaced boolean return with true for org/apache/syncope/core/spring/policy/DefaultAccountRule::lambda\$enforce\$2 → SURVIVED
89	1. removed call to java/util/stream/Stream::forEach → KILLED
95	1. replaced boolean return with false for org/apache/syncope/core/spring/policy/DefaultAccountRule::lambda\$enforce\$4 → KILLED 2. replaced boolean return with true for org/apache/syncope/core/spring/policy/DefaultAccountRule::lambda\$enforce\$4 → SURVIVED
96	1. removed call to java/util/stream/Stream::forEach → KILLED

Figura 7. Report Pitest per il metodo enforce() della classe DefaultAccountRule.

56	1. changed conditional boundary → SURVIVED
	2. changed conditional boundary → KILLED
	3. negated conditional → KILLED
	4. negated conditional → KILLED
61	1. changed conditional boundary → KILLED
	2. changed conditional boundary → KILLED
	3. negated conditional → KILLED
	4. negated conditional → KILLED
67	1. replaced boolean return with false for org/apache/syncope/core/spring/policy/DefaultAccountRule::lambda\$enforce\$0 → KILLED
	2. replaced boolean return with true for org/apache/syncope/core/spring/policy/DefaultAccountRule::lambda\$enforce\$0 → KILLED
68	1. removed call to java/util/stream/Stream::forEach → KILLED
73	1. negated conditional → KILLED
	2. negated conditional → KILLED
76	1. negated conditional → KILLED
	2. negated conditional → KILLED
81	1. negated conditional → KILLED
82	1. negated conditional → KILLED
88	1. replaced boolean return with false for org/apache/syncope/core/spring/policy/DefaultAccountRule::lambda\$enforce\$2 → KILLED
	2. replaced boolean return with true for org/apache/syncope/core/spring/policy/DefaultAccountRule::lambda\$enforce\$2 → KILLED
89	1. removed call to java/util/stream/Stream::forEach → KILLED
95	1. replaced boolean return with false for org/apache/syncope/core/spring/policy/DefaultAccountRule::lambda\$enforce\$4 → KILLED
	2. replaced boolean return with true for org/apache/syncope/core/spring/policy/DefaultAccountRule::lambda\$enforce\$4 → KILLED
96	1. removed call to java/util/stream/Stream::forEach → KILLED

Figura 7.1. Report Pitest per il metodo enforce() della classe DefaultAccountRule dopo aver valutato la robustezza del test-set.

65	1. changed conditional boundary → SURVIVED
	2. changed conditional boundary → KILLED
	3. negated conditional → KILLED
	4. negated conditional → KILLED
69	1. changed conditional boundary → KILLED
	2. changed conditional boundary → KILLED
	3. negated conditional → KILLED
	4. negated conditional → KILLED
74	1. negated conditional → KILLED
	2. negated conditional → KILLED
	3. negated conditional → KILLED
79	1. replaced boolean return with false for org/apache/syncope/core/spring/policy/DefaultPasswordRule::lambda\$enforce\$0 → KILLED
	2. replaced boolean return with true for org/apache/syncope/core/spring/policy/DefaultPasswordRule::lambda\$enforce\$0 → SURVIVED
80	1. removed call to java/util/stream/Stream::forEach → KILLED
85	1. negated conditional → KILLED
	2. negated conditional → KILLED
90	1. negated conditional → KILLED
	2. negated conditional → KILLED
95	1. negated conditional → KILLED
	2. negated conditional → KILLED
101	1. replaced boolean return with false for org/apache/syncope/core/spring/policy/DefaultPasswordRule::lambda\$enforce\$2 → KILLED
	2. replaced boolean return with true for org/apache/syncope/core/spring/policy/DefaultPasswordRule::lambda\$enforce\$2 → SURVIVED
102	1. removed call to java/util/stream/Stream::forEach → KILLED
108	1. replaced boolean return with false for org/apache/syncope/core/spring/policy/DefaultPasswordRule::lambda\$enforce\$4 → KILLED
	2. replaced boolean return with true for org/apache/syncope/core/spring/policy/DefaultPasswordRule::lambda\$enforce\$4 → SURVIVED
109	1. removed call to java/util/stream/Stream::forEach → KILLED
114	1. negated conditional → KILLED
	2. negated conditional → KILLED
118	1. negated conditional → KILLED
	2. negated conditional → KILLED
123	1. negated conditional → KILLED
	2. negated conditional → KILLED

Figura 8. Report Pitest per il metodo enforce() della classe DefaultPasswordRule.