Ingegneria del Software 2

Modulo Software Testing

Martina De Maio, 0296447

A.A. 2020/2021

Sommario

PROGETTO "1+"

In questo progetto il lavoro svolto è stato quello di proporre una nuova implementazione, per ogni classe di test, basata su ***JUnit4*** e che preveda esclusivamente la dichiarazione di test parametrici.

Le classi testate sono le seguenti:

1. ***JCSRemovalSimpleConcurrentTest***
2. ***JCSTrashTest***

1. JCSRemovalSimpleConcurrentTest

Nella fase di configurazione viene creata un'istanza della classe JCS, per la quale vengono testati i metodi *remove* e *clear*. In particolare, in ogni test si inseriscono inizialmente un certo numero di oggetti (indicato dal parametro *count*) all'interno del JCS creato nel setup. Successivamente, viene effettuata la *remove* di ogni oggetto e viene testato che la *get* su quell'oggetto restituisca *null*. In un altro test invece si effettua una *clear* del JCS, in modo da rimuovere ogni elemento al suo interno, e si verifica che la *get* su ogni elemento prima creato restituisca *null*.
Alla fine di ogni test, nel *tearDown*, si effettua la *clear* del JCS.

L'unico elemento che è stato parametrizzato è ***count***, ossia il numero totale di oggetti che si vuole inserire nel JCS.

I casi di test implementati sono i seguenti:

Caso 1: `{"/TestRemoval.ccf" ,"testCache1", 500}`

Caso 1: `{"/TestRemoval.ccf" ,"testCache1", 0}`

Caso 1: `{"/TestRemoval.ccf" ,"testCache1", -1}`

dove i primi due parametri servono nel setup per creare l'istanza della classe JCS.

2. JCSTrashTest

Il setup del test è identico a quello precedente.
In un test si verifica il metodo *put,* ossia l'aggiunta di una entry al JCS, in un altro test si verifica il metodo *remove*, ossia la rimozione di entry dal JCS. Un successivo test serve a verificare che, dopo aver fatto diverse attività sul JCS, (ossia dopo aver aggiunto al JCS una entry avente value = "*value*" e key = "*key*", creato un numero di threads che leggono le chiavi, creato un altro numero di threads che inseriscono un alto numero di keys con *values* aventi un grande numero di bytes) la memoria utilizzata non sia cresciuta di molto.
In questo caso gli elementi che sono stati parametrizzati sono il numero di *threads* e il numero di *keys* inserite da tali threads.

I casi di test implementati sono i seguenti:

Caso 1: `{"/TestThrash.ccf" ,"testcache", 15, 500}`

Caso 2: `{"/TestThrash.ccf" ,"testcache", 0, 0}`

Caso 3: `{"/TestThrash.ccf" ,"testcache", -1, 0}`

dove i primi due parametri servono nel setup per creare l'istanza della classe JCS come nel test precedente.

JaCoCo coverage

Per analizzare la coverage con JaCoCo sono stati seguiti i seguenti step:

1. Comando *mkdir -p target/jacocogen/jcs-coverage/* per creare la cartella *jcs-coverage*
2. Comando *java -jar ${PATH_JACOCO_CLI_JAR}/jacococli.jar report target/jacoco.exec --classfiles ${PATH_JCS_JAR}/jcs-1.3.jar --sourcefiles ${PATH_JCS_SRC} --html target/jacoco-gen/jcs-coverage/ --xml target/jacoco-gen/jcs-coverage/file.xml --csv target/jacoco-gen/jcs-coverage/file.csv*
   per l'estrazione con JaCoCo da cli del report.

I metodi che sono stati testati sono i metodi *put(Object name, Object obj), remove(Object name)* e *clear()* della classe *CacheAccess* appartenente al **package org.apache.jcs.access.**

Di seguito le schermate del report JaCoCo in cui sono riportate la branch coverage e la statement coverage dei metodi *CacheAccess. put(Object name, Object obj), CacheAccess.remove(Object name)* e *CacheAccess.clear().*

Si osserva che i metodi *put* e *remove* hanno una statement coverage del 100% e una branch coverage non disponibile, in quanto non sono presenti branch nel metodo.
Il metodo *clear* invece ha una branch coverage non disponibile e una statement coverage del 45%, che non si è riuscita a migliorare nonostante diverse prove.

CacheAccess

Source file "org/apache/jcs/access/CacheAccess.java" was not found during generation of report.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
freeMemoryElements(int)		0%		n/a	1	1	8	8	1	1
resetElementAttributes(Object, IElementAttributes)		0%		0%	2	2	5	5	1	1
putSafe(Object, Object)		0%		0%	2	2	4	4	1	1
ensureCacheManager()		0%		0%	3	3	5	5	1	1
put(Object, Object, IElementAttributes)		60%		50%	2	3	4	11	0	1
getElementAttributes(Object)		0%		n/a	1	1	6	6	1	1
destroy()		0%		n/a	1	1	5	5	1	1
defineRegion(String, ICompositeCacheAttributes, IElementAttributes)		0%		n/a	1	1	2	2	1	1
defineRegion(String, ICompositeCacheAttributes)		0%		n/a	1	1	2	2	1	1
getAccess(String, ICompositeCacheAttributes)		0%		n/a	1	1	2	2	1	1
defineRegion(String)		0%		n/a	1	1	2	2	1	1
getAccess(String)		0%		n/a	1	1	2	2	1	1
destroy(Object)		0%		n/a	1	1	2	2	1	1
clear()		45%		n/a	0	1	2	5	0	1
getCacheElement(Object)		0%		n/a	1	1	1	1	1	1
resetElementAttributes(IElementAttributes)		0%		n/a	1	1	2	2	1	1
setDefaultElementAttributes(IElementAttributes)		0%		n/a	1	1	2	2	1	1
setCacheAttributes(ICompositeCacheAttributes)		0%		n/a	1	1	2	2	1	1
getElementAttributes()		0%		n/a	1	1	1	1	1	1
getDefaultElementAttributes()		0%		n/a	1	1	1	1	1	1
getCacheAttributes()		0%		n/a	1	1	1	1	1	1
remove()		0%		n/a	1	1	2	2	1	1
static {...}		90%		50%	1	2	0	1	0	1
get(Object)		100%		100%	0	2	0	2	0	1
put(Object, Object)		100%		n/a	0	1	0	2	0	1
remove(Object)		100%		n/a	0	1	0	2	0	1
CacheAccess(CompositeCache)		100%		n/a	0	1	0	3	0	1
getStatistics()		100%		n/a	0	1	0	1	0	1
getStats()		100%		n/a	0	1	0	1	0	1
dispose()		100%		n/a	0	1	0	2	0	1
Total	242 of 327	25%	11 of 16	31%	27	38	63	87	20	30

JaCoCo Coverage Report > org.apache.jcs

org.apache.jcs

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
JCS		70%		75%	2	8	4	16	1	6	0	1
Total	12 of 40	70%	1 of 4	75%	2	8	4	16	1	6	0	1

PROGETTO "2"

1 INTRODUZIONE

L'obiettivo del seguente report di Software Testing è quello di presentare l'attività di testing svolta su due progetti Open source prodotti dalla Apache Software Foundation: ***BookKeeper*** e ***Syncope***.

Tale attività di testing è stata inserita in un contesto di *Continuous Integration*, sfruttando le integrazioni con ***Travis CI,*** per effettuare la build con ***Maven*** in remoto, ***SonarCloud*** per raccogliere e analizzare i risultati dei test e quindi misurare la qualità del software prodotto, e utilizzando la piattaforma ***GitHub*** per l'accesso alle repository.

È stato inoltre utilizzato il framework ***JUnit*** per l'implementazione dei casi di test, ***Jacoco*** per la generazione dei report di control-flow coverage dei casi di test sviluppati e ***Pit*** per l'esecuzione del mutation testing.

1.1 Note preliminari

Il setup dei due progetti è stato molto oneroso, e la maggior parte dell'effort è stato utilizzato per trovare una configurazione funzionante.

Per BookKeeper è stato necessario trovare una versione che non effettuasse la build con ***Gradle***, in quanto sono stati riscontrati diversi problemi nell'integrazione con TravisCI.

La release utilizzata è stata la 4.13.0, disponibile al seguente link:
https://github.com/apache/bookkeeper/releases/tag/release-4.13.0

Per quanto riguarda Syncope sono state riscontrate diverse difficoltà dovute anche al fatto che non si è trovata nessuna documentazione sia per quanto riguarda la configurazione dell'ambiente di lavoro, sia per una spiegazione dettagliata delle classi, e quindi è stato necessario disabilitare diversi plug-in e modificare alcuni *pom.xml* che creavano problemi, dovendo spesso scaricare diverse release del progetto per trovare quella funzionante.

I test presenti nelle repository originali effettuati dagli sviluppatori, dove possibile, sono stati eliminati. Inoltre, i test da effettuati sono lasciati parzialmente commentati per mostrare le scelte e le considerazioni effettuate nel category partition.

2 SCELTA DELLE CLASSI

Per la scelta delle classi si sono tenuti in considerazione i risultati delle attività ottenuti nell'altro modulo del corso, soffermandosi su metriche come ***NumberOfRevisions***, ***LOC touched*** e osservando anche la ***buggyness*** delle classi nelle varie release.

Le classi testate per il progetto BookKeeper sono state le seguenti:

- *BookKeeper :* classe risultata affetta da bug in quasi tutte le release del progetto e avente un numero di revisioni e di linee di codice aggiunte sempre molto alto rispetto alla media. Inoltre, tale classe è stata scelta per la grande disponibilità di documentazione e per una facile comprensibilità del codice.
- *WriteCache* e *ReadCache :* la scelta di queste classi non è stata guidata da uno studio sulle metriche ma si è tenuto conto della chiarezza della documentazione trovata.
- *LedgerEntriesImpl:* classe toccata da poche release.

A causa della scarsa documentazione, per il progetto Syncope sono state scelte classi il cui scopo era comprensibile senza l'utilizzo di una documentazione e il cui comportamento era facilmente deducibile osservando il codice.

Le classi scelte per il progetto Syncope sono:

- *FormatUtils*, classe presente solamente in un'unica release, in cui risultava buggata, che si occupa del parsing e della formattazione di date e numeri.
- *Encryptor*, classe presente in molte release, risultata buggata in ognuna di queste, che si occupa delle funzionalità di crittografia utilizzate nel sistema.

- ***RealmUtils****,* che fornisce le funzionalità per gestire i ***realms*****,** chiavi utilizzate per gestire l'accesso a risorse condivise e protette.

3 APACHE BOOKKEEPER

3.1 Category Partition

Per creare una Test Suite per ogni metodo di una classe scelta si è seguito il ***Category Partition Method***, che consiste nel partizionare il dominio di input in modo da avere un insieme di test con minore cardinalità possibile. Per ogni partizione poi, verrà scelto un campione con cui svolgere il test.

WriteCache

La classe *WriteCache* implementa una cache di scrittura, la quale alloca la dimensione richiesta dalla memoria diretta e la suddivide in più segmenti. Le entries vengono aggiunte in un buffer comune e indicizzate tramite una *hashmap*, fino a quando la cache non viene cancellata. Per tale classe sono stati testati i seguenti metodi:

- `public boolean put(long ledgerId, long entryId, ByteBuf entry)`
- `public ByteBuf get(long ledgerId, long entryId)`

WriteCache.put(long ledgerId, long entryId, ByteBuf entry)

Tale metodo copia nella cache di scrittura il contenuto del buffer *entry* con un certo *entryId* destinato ad un determinato ledger con il suo *ledgerId*. Il metodo restituisce ***true*** in caso di avvenuta scrittura e ***false*** in caso di fallimento.

Un ledger è una sequenza di entries, dove ogni entry è una sequenza di bytes. Le entries vengono scritte sequenzialmente e al massimo una volta su un ledger.

I parametri di input *ledgerId* e *entryId* sono dei long, e in quanto tali, non avendo ulteriori informazioni sui range ammissibili, si scelgono come partizioni {>=0, <0}, mentre per il ByteBuf *entry* si sceglie la partizione {null, new ByteBuf() }. Poiché il successo o il fallimento del metodo dipendono dalla quantità di bytes che si vuole scrivere nella cache, è importante dividere il caso in cui i byte da scrivere sono inferiori dei bytes disponibili nella cache, dal caso in cui sono maggiori. Per i due casi si avrà, rispettivamente, un successo e un fallimento della scrittura. Le partizioni del dominio di input individuate per tale metodo sono quindi:

- *ledgerId*: `{-1, 0, 1}`
- *entryId*: `{-1, 0, 1}`
- *entry:* {`null, entry with size<=available memory in cache, entry with size > available memory in cache}`

In una prima analisi i casi di test sviluppati sono stati i seguenti:

	ledgerId	**entryId**	**entry**
Caso 1:	{ 0,	-1,	null }
Caso 2:	{- 1,	1,	entry with size <= available memory in cache}
Caso 3:	{ 0,	0 ,	entry with size > available memory in cache }
Caso 4:	{ 0,	0 ,	entry with size <= available memory in cache }

Nei casi di test 1 e 2 si ha un fallimento del metodo *WriteCache.put(),* poiché i due casi sollevano, rispettivamente, una *NullPointerException* dovuta a un *entryId* negativa e una *IllegalArgumentException: "Keys and values must be >= 0"* dovuta a un *ledgerId* negativo. Il caso di test numero 3 invece porta a un fallimento della scrittura dovuto al fatto che si sta tentando di scrivere nella cache una quantità di byte superiore alla memoria disponibile in essa. Nel caso di test numero 4 invece si ha un successo nell'operazione di scrittura.

WriteCache.get(long ledgerId, long entryId)

Tale metodo legge dalla cache il contenuto relativo ad una entry avente ID *entryID* situato nel ledger con un ID uguale a *ledgerID* passato. Se i parametri forniti si riferiscono ad una entry valida viene restituito un *ByteBuf* con il contenuto della entry, mentre il metodo ritorna *null* se l'entry passata non è presente nella cache o se la coppia *ledgerId-entryId* non è valida.
Nella fase di setup si è inizializzata una *WriteCache* nella quale verranno scritte delle entry tramite il metodo *WriteCache.put().* Per implementare il test vengono divisi gli ID dei ledger e delle entries da inserire nella cache

tramite il metodo *put,* da quelli che si passano al metodo *get* per leggere la cache. Si avranno quindi, rispettivamente, *ledgerIdPut – entryIdPut* e *ledgerIdGet - entryIdGet*.

Le partizioni del dominio di input individuate per tale metodo sono:

- *ledgerId*: {-1,0,1}
- *entryId*: {-1,0,1}

In una prima analisi i casi di test sviluppati sono stati i seguenti:

	ledgerIdGet	entryIdGet	ledgerIdPut	entryIdPut
Caso 1:	{ -1,	-1,	-1,	-1}
Caso 2:	{ 1,	0,	0,	1}
Caso 3:	{ 0,	0,	0,	0}

Nel caso 1 si solleva una *IllegalArgumentException* poiché il metodo *put*(*ledgerIdPut* = -1, *entryIdPut* = 0) fallisce in quanto l'ID del ledger passato è negativo. Nel secondo caso invece il metodo *get* fallisce perché si sta tentando la *get*(*ledgerIdGet = 0, entryIdGet = 1*), a seguito di una *put*(*ledgerIdPut* = 1, *entryIdPut* = 0), ossia si sta tentando di leggere una entry non valida, non presente in cache.
Nel caso 3 invece si ha un successo del metodo *WriteCache.get().*

ReadCache

La classe ReadCache implementa una cache di lettura, che utilizza una quantità di memoria specificata e la associa a una *hashmap*. La memoria è suddivisa in più segmenti che vengono utilizzati in modo ring-buffer. Quando la cache di lettura è piena, il segmento più vecchio viene cancellato e ruotato per fare spazio alle nuove voci da aggiungere alla cache di lettura. Per questa classe è stato testato il seguente metodo:

- *public ByteBuf **get**(long ledgerId, long entryId)*

ReadCache.get(long ledgerId, long entryId)

Il metodo *ReadCache.get(long ledgerId, long entryId)* restituisce un buffer contenente l'entry nella cache di lettura con *entryId* specificato che deve essere letta dal ledger specificato da *ledgerId*. Il metodo ritorna *null* nel caso in cui l'entry non è stata trovata in nessun segmento.
Nel setup si è creata un'istanza *ReadCache* nella quale verranno inserite delle entry tramite il metodo *ReadCache.put().* Analogamente al test effettuato per il metodo *WriteCache.get(),* sono stati distinti gli ID del ledger e della entry da inserire nella cache da quelli da utilizzare nel metodo *get().*

Le partizioni del dominio di input individuate per tale metodo sono:

- *ledgerId*: {-1,0,1}
- *entryId*: {-1,0,1}

In una prima analisi i casi di test sviluppati sono stati i seguenti:

	ledgerIdGet	entryIdGet	ledgerIdPut	entryIdPut
Caso 1:	{ -1,	0,	-1,	0}
Caso 2:	{ 1,	0,	0,	1}
Caso 3:	{ 1,	-1,	1,	-1}

Il caso di test numero 3 è l'unico che non solleva alcuna eccezione e per il quale il metodo *ReadCache.get()* ha successo.

LedgerEntriesImpl

La classe *LedgerEntriesImpl* si occupa dell'implementazione di ledger entries. Per tale classe sono stati testati i seguenti metodi:

- *public static LedgerEntriesImpl **create**(List<LedgerEntry> entries)*
- *public LedgerEntry **getEntry**(long entryId)*

LedgerEntriesImpl. create(List<LedgerEntry> entries)

Metodo che crea *ledger entries*.
Per l'implementazione del test si è creato un metodo *createEntryList(long ledgerId, long entryId)* che permette di creare, dato un *ledgerId* e una *entryId*, una lista di *LedgerEntry* da usare come parametro al metodo.

Le partizioni individuate per tale metodo sono:

- *entries*: `{null, Lists.newArrayList(), createEntryList(ledgerId = 0, entryId = 0)}`

In una prima analisi i casi di test sviluppati sono stati i seguenti:

	entries
Caso 1:	`{ entries = null }`
Caso 2:	`{ entries = Lists.newArrayList() }`
Caso 3:	`{ entries = createEntryList(0,0)}`

Nel caso di test 1 il metodo *create*() da testare solleva una *NullPointerException*, mentre nel caso 2 una *IllegalArgumentException* con messaggio *"entries for create should not be empty."*
Nel caso 3 invece, essendo entries una lista valida, si crea un'istanza di *LedgerEntriesImpl*, e si verifica che la size della lista di entry, la classe di appartenenza di ogni entry ed il contenuto delle entry nella lista siano quelli attesi.

LedgerEntriesImpl.getEntry(long entryId)

Metodo che ritorna una *LedgerEntry*, ossia una entry in un ledger, specificata da *entryId*.
I parametri del metodo sono:

- *entryId* (long): Id dell'entry che si vuole ottenere

Per implementare il test, inizialmente si è creata un'istanza di *LedgerEntriesImpl* contenente alcune entries che dovranno essere lette. In particolare, sono state inserite 7 *LedgerEntries* aventi un *entryId* da 0 a 6. Il metodo *getEntry()* restituisce una *LedgerEntry* solamente se questa è stata preventivamente creata.

Le partizioni individuate per il metodo *LedgerEntriesImpl.getEntry(long entryId)*, essendo *entryId* un long, sono state:

- *entryId*: `{-1,0,1}`
-

In una prima analisi i casi di test sviluppati sono stati i seguenti:

	entryId
Caso 1:	`{ entryId = -1}`
Caso 2:	`{ entryId = 0 }`
Caso 3:	`{ entryId = 1}`

Nel caso in cui la entry specificata è -1 si genera una *IndexOutOfBoundsException* con messaggio *"required index: -1 is out of bounds: [0, 6]."*

BookKeeper

Tale classe rappresenta il client per l'interazione con il sistema. In questa classe vengono eseguite operazioni sui ledger, come la creazione di un nuovo ledger, la scrittura o lettura su di esso, o la cancellazione.

Per tale classe si è scelto di testare il metodo:

- *public LedgerHandle* ***openLedger****(long lId, DigestType digestType, byte[] passwd)*

LedgerHandle.openLedger

Tale metodo permette di aprire un ledger esistente per prepararlo alla lettura. Il metodo fallisce se la password inserita è sbagliata o se si tenta di aprire un ledger il cui ID non è corretto.

I parametri del metodo sono:

- *lId* (long), identificatore del ledger
- *digestType* (DigestType), il digest type utilizzato per verificare l'integrità delle entries, che può essere MAC / CRC32/ CRC32C / DUMMY
- *passwd* (*byte[]*), password.

Per l'implementazione del metodo, nel setup è stato creato un valido *LedgerHandle* tramite:

BookKeeper.createLedger(DigestType.CRC32, "password".getBytes()),

per poi verificare che il metodo *openLedger* ritorni esattamente il *LedgerHandle* creato.

Poiché il metodo fallisce se la password inserita non è corretta o se il ledger ID passato non corrisponde a nessun ledger esistente, per il dominio di input sono state individuate le seguenti partizioni:

- *lId*: `{existing ledger ID, not existing ledger ID, invalidLedgerID}`
- *digestType*: `{MAC, CRC32, CRC32C, DUMMY}`
- password: `{validPassword, invalidPassword, emptyString, null}`

In una prima analisi i casi di test sviluppati sono stati i seguenti:

```
            lId                              digestType                passwd

Caso 1: {LedgerHandle.INVALID_ENTRY_ID, BookKeeper.DigestType.MAC,  "password".getBytes()}
Caso 2: {          0,           BookKeeper.DigestType.MAC, "bad_password".getBytes()},
Caso 3: {          0,           BookKeeper.DigestType.MAC,               "".getBytes()},
Caso 4: {          0,           BookKeeper.DigestType.MAC,             null          },
Caso 5: {          0,           DigestType.MAC,                  "password".getBytes()},
Caso 6: {          0,           DigestType.CRC32C,               "password".getBytes()},
Caso 7: {          0,           DigestType.DUMMY,                "password".getBytes()},
Caso 8: {          1,           BookKeeper.DigestType.CRC32,     "password".getBytes()},
Caso 9: {          0,           BookKeeper.DigestType.CRC32,     "password".getBytes()},
```

I primi 8 casi portano a un fallimento del metodo *openLedger*, in particolare:

- **casi 1 e 8**: viene sollevata una *BKException* con messaggio "*No such ledger exists on Metadata Server*" a causa del ledger id invalido nel caso 1, e del tentativo di aprire un ledger non esistente nel caso 8;
- **casi 2-4** : viene sollevata una *BKException$BKUnauthorizedAccessException* con messaggio "*Attempted to access ledger using the wrong password*", a causa della password invalida;
- **casi 5-7**: viene sollevata una *BKException$BKDigestMatchException* con messaggio "*Entry digest does not match*", a causa di invalidi digest type.

In una prima implementazione si è cercato di testare il metodo confrontando il *LedgerHandle* ottenuto da una *LedgerEntriesImpl.createLedger(DigestType digestType, byte[] passwd)* con il *LedgerHandle* ottenuto dal metodo *openLedger*, in modo da verificare che l'apertura del ledger ritorni esattamente il ledger creato precedentemente. Purtroppo, tale metodologia di implementazione è risultata non valida in quanto i due metodi ritornano due tipi di *LedgerHandle* diversi. Infatti, in un tentativo di un'*AssertEquals* tra i due *LedgerHandle*, si è riscontrato il seguente errore:

```
java.lang.AssertionError:
Expected :org.apache.bookkeeper.client.LedgerHandle@b71c6b7
Actual   :ReadOnlyLedgerHandle(lid = 0, id = 82832594)
```

Per questo motivo si è deciso di testare il metodo utilizzando le eccezioni.

3.2 Adeguatezza e miglioramento dei casi di test

In questo paragrafo vengono mostrati i risultati ottenuti dallo studio della ***statement*** and ***branch coverage*** ricavati tramite il plugin ***JaCoCo***, insieme ad eventuali miglioramenti effettuati.

WriteCache

- ***put***: in *Figura1* si può osservare che si ha una statement coverage del 96% e una branch coverage del 62%.
- ***get:*** in *Figura5* si può notare che si ha una statement coverage e una branch coverage del 100%.

Per quanto riguarda il metodo *put(),* dalla *Figura2* si può notare che non si è mai analizzato il caso in cui l'entry che si vuole scrivere ha una dimensione superiore a quella di un segmento in cui è suddivisa la cache.

Introducendo un ulteriore caso di test, con questi miglioramenti si è arrivato a una statement coverage del 97% e una branch coverage del 75%, come mostra la *Figura3*, oltre ai quali non si è riusciti a migliorare.

ReadCache

- ***get:*** in *Figura7* si può notare che si ha una statement coverage e una branch coverage del 100%.

LedgerEntriesImpl

- ***create*****:** In *Figura9* si può notare che si ha una statement coverage e una branch coverage del 100%.
- **getEntry() in** *Figura11* si può notare che si ha una statement coverage del 100% e una branch coverage del 75%.

Per quanto riguarda il metodo *getEntry(),* dalla *Figura12* si può notare che non si è mai analizzato il caso in cui *l'entryId* sia maggiore del numero di LedgerEntry che sono state create (nel nostro caso sono state create 7 LedgerEntries aventi ID da 0 a 6), quindi si è aggiunto un ulteriore caso di test :

- `{ entryId = 7}`

che ha permesso di migliorare la branch coverage portandola al 100% (*Figura13*).

BookKeeper

- ***openLedger*****:** In *Figura15* si può notare che si ha una statement coverage del 100% e una branch coverage non definita.

4 APACHE SYNCOPE

Apache Syncope è un sistema Open source per la gestione delle identità digitali in ambienti aziendali, implementato in tecnologia Java EE e rilasciato sotto licenza Apache 2.0.

4.1 Category Partition

FormatUtils

La classe FormatUtils è una classe utilizzata per il parsing e la formattazione di date e numeri, per la quale sono stati testati i seguenti metodi:

- *public static String* ***format****(final Date date, final boolean lenient, final String conversionPattern)*
- *public static Date* ***parseDate****(final String source, final String conversionPattern)*
- *public static Number* ***parseNumber****(final String source, final String conversionPattern)*

FormatUtils.format

I parametri del metodo sono:

- *date* (Date), la data che si vuole formattare
- *lenient* (boolean), booleano che serve a definire se la formattazione è rigorosa o clemente ("lenient") nell'interpretare input che non corrispondono esattamente al pattern
- *conversionPattern* (string), pattern secondo cui effettuare la formattazione.

Tale metodo, presa una data e un pattern per la conversione, permette di formattare la data convertendola in una stringa.
Ad esempio, se come data si passa "*Tue Jul 06 13:33:57 CEST 2021"* , ottenuta da *calendar.getTime(),* se si utilizza il pattern di conversione "*dd/MM/yyyy*" si otterrà la stringa "*06/07/2021*".
In particolare, ci si è accorti che, a prescindere dal valore di *lenient*, l'output è sempre lo stesso. Non si è quindi riusciti a capire effettivamente quale sia l'utilizzo di tale parametro in questo metodo. Analizzando attentamente la scarsa documentazione disponibile, si è capito che tale parametro porta al fallimento del metodo se il formato di data passato

non è conforme al pattern utilizzato, ma, in questo caso, poiché il parametro *date* viene creato con *calendar.getTime(),* il formato è sempre corretto. Si è anche provato a passare come parametro *date* una data creata tramite *new Date(string),* in modo da inserire un formato di stringa non coerente con il pattern voluto, ma comunque tale metodo effettua il parse della stringa, quindi se il formato non è valido tale metodo fallisce.

Le partizioni individuate per tale metodo sono:

- *date*: `{calendar.getTime(), null}`
- *lenient*: `{true, false}`
- *conversionPattern*: `{pattern valido, pattern non valido, null}`

In una prima analisi i casi di test sviluppati sono stati i seguenti:

	Date	**Lenient**	**ConversionPattern**
Caso 1:	{calendar.getTime(),	true,	"dd/MM/yyyy"}
Caso 2:	{calendar.getTime(),	false,	"invalidPattern"}
Caso 3:	{calendar.getTime(),	true,	null}
Caso 4:	{ null,	true,	"dd/MM/yyyy"}

Nel caso in cui il conversion pattern è null, non si effettua il parse della data e il metodo *format* ha comunque successo.

FormatUtils.parseDate

Tale metodo effettua la trasformazione opposta a *format(),* ossia converte una stringa *source* in una data, secondo il pattern *conversionPattern*.

In particolare, data per esempio una *source* "*06-07-2021*", utilizzando il pattern "*dd-MM-yyyy*" si ottiene la data *Tue Jul 06 00:00:00 CEST 2021*.

I parametri del metodo sono:

- *source* (String), stringa che si vuole convertire in Date
- *conversionPattern* (String), pattern secondo cui effettuare la conversione

Le partizioni individuate per tale metodo sono:

- *source*: `{ formato stringa per data valido, formato stringa per data non valido, empty string, null}`
- *conversionPattern*: `{pattern valido, pattern non valido, null}`

In una prima analisi i casi di test sviluppati sono stati i seguenti:

	Source	**ConversionPattern**
Caso 1:	{"20-06-2021",	"dd-MM-yyyy"}
Caso 2:	{"aaa",	"dd/MM/yyyy"}
Caso 3:	{"",	"dd/MM/yyyy"}
Caso 4:	{null,	"dd-MM-yyyy"}
Caso 5:	{"20-06-2021",	"yyyy-MM-dd'T'HH:mm:ssZ"}
Caso 6:	{"20-06-2021",	""}
Caso 7:	{"20-06-2021",	null}
Caso 8:	{"20-06-2021",	"aaa"}
Caso 9:	{null,	null}

I casi di test 2, 3 ,5 e 6 ritornano una *java.text.ParseException* a causa di un formato di stringa non valido come formato data, nel caso dei test 2 e 3, oppure a causa di un conversionPattern non valido per la stringa in questione, nel caso dei test 5 e 6. Si è ritenuto opportuno cancellare quindi i test 3 e 6, poiché ridondante. I casi 4, 8 e 9 ritornano una *java.lang.IllegalArgumentException*, ma con messaggi diversi: il primo e l'ultimo ritornano il messaggio "*Date and Patterns must not be null*", mentre il secondo "*Format 'c' not supported*". Si è quindi ritenuto utile eliminare il caso di test numero 4, poiché ridondante. Il caso di test numero 7 ritorna invece una *java.lang.NullPointerException* dovuta al conversionPattern = null.

Riassumendo, dopo un'attenta analisi, i casi di test implementati sono stati:

Caso 1:	{"20-06-2021",	"dd-MM-yyyy"}

```
Caso 2: {"aaa",            "dd/MM/yyyy"}
Caso 3: {"20-06-2021",     "yyyy-MM-dd'T'HH:mm:ssZ"}
Caso 4: {"20-06-2021",     null}
Caso 5: {"20-06-2021",     "aaa"}
Caso 6: {null,             null}
```

FormatUtils.parseNumber

I parametri del metodo sono:

- *source* (String), stringa che si vuole convertire in *Number*
- *conversionPattern* (String), pattern secondo cui effettuare la conversione

Tale metodo converte la stringa *source* in un oggetto di tipo *Number*, secondo il pattern *conversionPattern*.

Con un'attenta analisi del metodo *FormatUtils.parseNumber* ci si è accorti che una source valida è una qualunque stringa avente come carattere iniziale almeno un numero, e i caratteri successivi possono essere un qualsiasi tipo di carattere alfanumerico. Infatti, il metodo prende come caratteri validi solo i caratteri numerici e ignora quelli alfabetici, a condizione che i caratteri numerici siano all'inizio della stringa. In altre parole, esempi di stringhe considerate valide sono "*123aaa*", "*1aaa*", "*123*", mentre una stringa non valida è la stringa "*aaa123*".

Le partizioni individuate per tale metodo sono:

- source: `{ formato stringa valido, formato stringa non valido, null}`
- conversionPattern: `{pattern valido, pattern non valido, null}`

In una prima analisi i casi di test sviluppati sono stati i seguenti:

```
              Source        ConversionPattern
Caso 1: {"12345",           "###,###"}
Caso 2: {"aaa123",          "###,###"}
Caso 3: {null,              "###,###"}
Caso 4: {"12345",           "invalidPattern"}
Caso 5: {"aaa123",          "invalidPattern"}
Caso 6: {"12345",           null}
Caso 7: {null,              null}
```

Ci si è resi conto che i casi di test 6 e 7 sono equivalenti, in quanto ritornano entrambi una *java.lang.NullPointerException* dovuta al pattern nullo, quindi il caso di test 6 è risultato ridondante e di conseguenza eliminato. Anche i casi 4 e 5 sono equivalenti, perché entrambi ritornano una *java.text.ParseException* dovuta al pattern invalido, quindi il caso 5 è stato eliminato.

Riassumendo, i casi di test finali implementati sono stati:

```
Caso 1: {"12345",           "###,###"}
Caso 2: {"aaa123",          "###,###"}
Caso 3: {"12345",           "invalidPattern"}
Caso 4: {null,              null}
```

Encryptor

Per la classe *Encryptor* sono stati testati i metodi:

- *public String **decode**(final String encoded, final CipherAlgorithm cipherAlgorithm)*
- *public boolean **verify**(final String value, final CipherAlgorithm cipherAlgorithm, final String encoded)*

Encryptor.decode

Tale metodo prende in input una stringa criptata e un algoritmo di cifratura, e restituisce in output il valore della stringa in plaintext, ossia in chiaro, decriptandola tramite l'algoritmo scelto.

I parametri del metodo sono:

- *encoded* (String), la password criptata

- *cipherAlgorithm* (*CipherAlgorithm*), algoritmo di cifratura utilizzato per decriptare la password

Per l'implementazione di tale test è stato creato il metodo *encryptPsswd(String value,CipherAlgorithm cipherAlgorithm)* il quale, data una password, la cripta e restituisce la stringa *encoded* che verrà usata come input.
In particolare, la stringa da criptare è "*password*", che verrà cifrata tramite *encryptPasswd* generando una stringa che verrà usata come un *encoded* valido. L'idea è quella di verificare che, decriptando la stringa *encoded,* il risultato ottenuto è proprio la stringa "*password*".
Le partizioni individuate per tale metodo sono:

- encoded: `{ stringa valida criptata, stringa non valida, null}`
- cipherAlgorithm: `{algoritmo AES, algoritmo SHA-256, algoritmo BCRYPT, null}`

In una prima analisi i casi di test sviluppati sono stati i seguenti:

	Encoded	**CipherAlgorithm**
Caso 1:	{ encryptPsswd(**"password"**),	CipherAlgorithm.***AES*** }
Caso 2:	{**"ciao"**,	CipherAlgorithm.***AES*** }
Caso 3:	{null,	null}
Caso 4:	{ encryptPsswd(**"password"**),	CipherAlgorithm.***SHA256*** }
Caso 5:	{ encryptPsswd(**"password"**),	CipherAlgorithm.***BCRYPT*** }

Encryptor.verify

Tale metodo prende in input due stringhe, *value* e *encoded*¸ e un algoritmo di cifratura, e controlla se *encoded* è la cifratura di *value* ottenuta tramite l'algoritmo di cifratura scelto.

I parametri del metodo sono:

- *value* (String), la password in chiaro
- *cipherAlgorithm* (CipherAlgorithm), algoritmo di cifratura utilizzato per decriptare la password
- *encoded* (String), la password criptata

Per generare stringhe *encoded* valide si è rifatto uso del metodo *encryptPsswd* descritto precedentemente.

Le partizioni individuate per tale metodo sono:

- *source*: `{"password", "password_bad", null}`
- *encoded*: `{ stringa valida criptata, stringa non valida, null}`
- *cipherAlgorithm*: `{algoritmo AES, algoritmo BCRYPT, null}`

In una prima analisi i casi di test sviluppati sono stati i seguenti:

	Value	**CipherAlgorithm**	**Encoded**
Caso 1:	{**"password"**,	CipherAlgorithm.***AES***,	encryptPsswd(**"password"**)}
Caso 2:	{**"password"**,	CipherAlgorithm.***AES***,	encryptPsswd(**"bad_password"**)}
Caso 3:	{**"password"**,	CipherAlgorithm. ***BCRYPT***,	encryptPsswd(**"password"**)}
Caso 4:	{null,	null,	null }

RealmUtils

Per la classe *RealmUtils* è stato testato il metodo:

- *public static boolean* ***normalizingAddTo****(final Set<String> realms, final String newRealm)*

Non essendo stata trovata alcuna documentazione, sono state effettuate diverse prove che hanno portato ad una interpretazione personale del metodo. Il metodo riceve come input un set di stringhe *realms* e una stringa *newRealm.* Se *newRealm* inizia con una delle stringhe contenute in *realms*, viene restituito un booleano *false* e il set di stringhe rimane invariato. Al contrario, se è una delle stringhe contenuta in *realms* a iniziare con *newRealm*, tale stringa viene rimossa da *realms* e al suo posto viene inserita *newRealm* e viene restituito *true.*

I parametri del metodo sono:

- *realms* (Set<String>), insieme di stringhe

- *newRealm* (String), stringa da inserire, eventualmente, in realms

Le partizioni individuate per tale metodo sono:

- *realms*: `{ set valido, set vuoto}`
- *newRealm*: `{stringa valida, null}`

Per l'implementazione del test il parametro *realms* è stato scelto con solamente due stringhe al suo interno, e si è testato inizialmente il caso in cui *newRealm* iniziasse con una delle due stringhe, e successivamente il caso in cui era una delle due stringhe ad iniziare con *newRealm.*

In una prima analisi i casi di test sviluppati sono stati i seguenti:

	Realm1	**Realm2**	**NewRealm**
Caso 1:	{ "realm1",	"realm2",	"realm123"}
Caso 2:	{ "realm123",	"realm2",	"realm1"}
Caso 3:	{null,	null,	null}

Dove **`Realm1`** e **`Realm2`** sono le stringhe da aggiungere in *realms.*

In tutti gli insiemi di casi di test visti fino ad ora, il comportamento atteso è stato sempre confermato.

4.2 Adeguatezza e miglioramento dei casi di test

In questo paragrafo vengono mostrati i risultati ottenuti dallo studio della **statement and branch** coverage ricavati tramite il plugin ***JaCoCo***, insieme ad eventuali miglioramenti effettuati.

FormatUtils

- ***format***: In *Figura 23* si può vedere che si ha una statement coverage e una branch coverage del 100% (infatti tutti i branch e gli statement sono raggiunti (*Figura24*))
- ***parseDate***: In *Figura 25* si può vedere che si ha una statement coverage del 100% e una branch coverage non definita, a causa di mancanza di branch nel metodo (*Figura26*)
- ***parseNumber***: In *Figura 27* si può vedere che si ha una statement coverage del 100% e una branch coverage non definita, a causa di mancanza di branch nel metodo (*Figura28*)

Encryptor

- **decode:** In *Figura 29* si può notare che si ha una statement coverage e una branch coverage del 100% (infatti tutti i branch e gli statement sono raggiunti (*Figura30*))
- **verify in** *Figura 31* si può notare che si ha una statement coverage del 82% e una branch coverage dell'87%.

Per quanto riguarda il metodo *verify()*, dalla *Figura 32* si può notare che non si è mai analizzato il caso in cui l'algoritmo di cifratura fosse diverso da *AES* o *BCRYPT*, quindi a tale scopo è stato introdotto un ulteriore caso di test:
`{ password", CipherAlgorithm.SHA256, encryptPsswd("password", CipherAlgorithm.SHA256)},`
che ha permesso di ottenere una statement e una branch coverage del 100% (*Figura 33*).

RealmUtils

- **normalizingAddTo**: In *Figura 35* si può vedere che si ha una statement coverage e una branch coverage del 100% (infatti tutti i branch e gli statement sono raggiunti (*Figura 36*))

5 MUTATION COVERAGE

La mutation coverage per **BookKeeper** è del 28%, nonostante la classe **LedgerEntriesImpl** abbia una buona mutation coverage del 76%, la migliore tra tutte le classi testate nei due progetti.

Nel metodo **create** della classe **LedgerEntriesImpl** vengono uccise **tutte le mutazioni**, mentre nel metodo **getEntry** solo 1 ne sopravvive. Per le classi **Bookkeeper** , **ReadCache** e **WriteCache** si ha una mutation coverage nettamente inferiore (23% per Bookkeeper , 22% per ReadCache e 31% per WriteCache).

Nel metodo **get** di ReadCache vengono uccise 5 mutazioni e ne sopravvivono 3.

Per quanto riguarda Syncope invece la classe **Encryptor** presenta una mutation coverage del 47%, **FormatUtils** 33% e **RealmUtils** 15%.

La classe Encryptor ha una buona mutation coverage del 47%, infatti *Figura43* e *Figura44* si può osservare che in entrambi i metodi **decode** e **verify** tutte le mutazioni sono state uccise.

Per quanto riguarda la classe **FormatUtils**, nel metodo **format** vengono uccise 3 mutazioni mentre 1 sopravvive, in **parseDate** 2 vengono uccise e 1 sopravvive. In **parseNumber** invece tutte le mutazioni sono state uccise.

6 LINKS

GitHub

JCS: *https://github.com/martina97/JCS-Tests*

BookKeeper: https://github.com/martina97/bookkeeper

Syncope: https://github.com/martina97/syncope

SonarCloud

JCS: *https://sonarcloud.io/dashboard?id=Martina97_JCS-Tests*

BookKeeper: https://sonarcloud.io/dashboard?id=martina97_bookkeeper

Syncope: https://sonarcloud.io/dashboard?id=martina97_syncope

TravisCI

JCS: *https://travis-ci.com/github/martina97/JCS-Tests*

BookKeeper: https://travis-ci.com/github/martina97/bookkeeper

Syncope: https://travis-ci.com/github/martina97/syncope

Per compilare ed eseguire i test si possono lanciare i comandi:
BookKeeper:

- ***PIT***: *mvn -DwithHistory org.pitest:pitest-maven:mutationCoverage surefire:test -Ppit*
- ***JaCoCo***: *mvn clean org.jacoco:jacoco-maven-plugin:prepare-agent verify*

Syncope:

- ***PIT***: *mvn -DwithHistory org.pitest:pitest-maven:mutationCoverage surefire:test -Dianal.skip=true -Drat.skip=true -Dcheckstyle.skip=true*
- ***JaCoCo***: *mvn clean org.jacoco:jacoco-maven-plugin:prepare-agent verify -Dianal.skip=true -Drat.skip=true -Dcheckstyle.skip=true*

7 IMMAGINI – BOOKKEEPER

WriteCache

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
forEach(WriteCache.EntryConsumer)		0%		0%	8	8	32	32	1	1
get(long, long)		0%		0%	2	2	10	10	1	1
lambda$forEach$0(long, long, long, long)		0%		0%	2	2	8	8	1	1
getLastEntry(long)		0%		0%	2	2	4	4	1	1
WriteCache(ByteBufAllocator, long, int)		92%		50%	3	4	1	25	0	1
isEmpty()		0%		0%	2	2	1	1	1	1
deleteLedger(long)		0%		n/a	1	1	2	2	1	1
size()		0%		n/a	1	1	1	1	1	1
count()		0%		n/a	1	1	1	1	1	1
put(long, long, ByteBuf)		96%		62%	3	5	3	20	0	1
clear()		100%		n/a	0	1	0	7	0	1
close()		100%		100%	0	2	0	3	0	1
alignToPowerOfTwo(long)		100%		n/a	0	1	0	1	0	1
static {...}		100%		n/a	0	1	0	2	0	1
align64(int)		100%		n/a	0	1	0	1	0	1
WriteCache(ByteBufAllocator, long)		100%		n/a	0	1	0	2	0	1
Total	321 of 617	47%	28 of 38	26%	25	35	63	120	8	16

Figura 1: Statement coverage e Branch coverage per il metodo WriteCache.put().

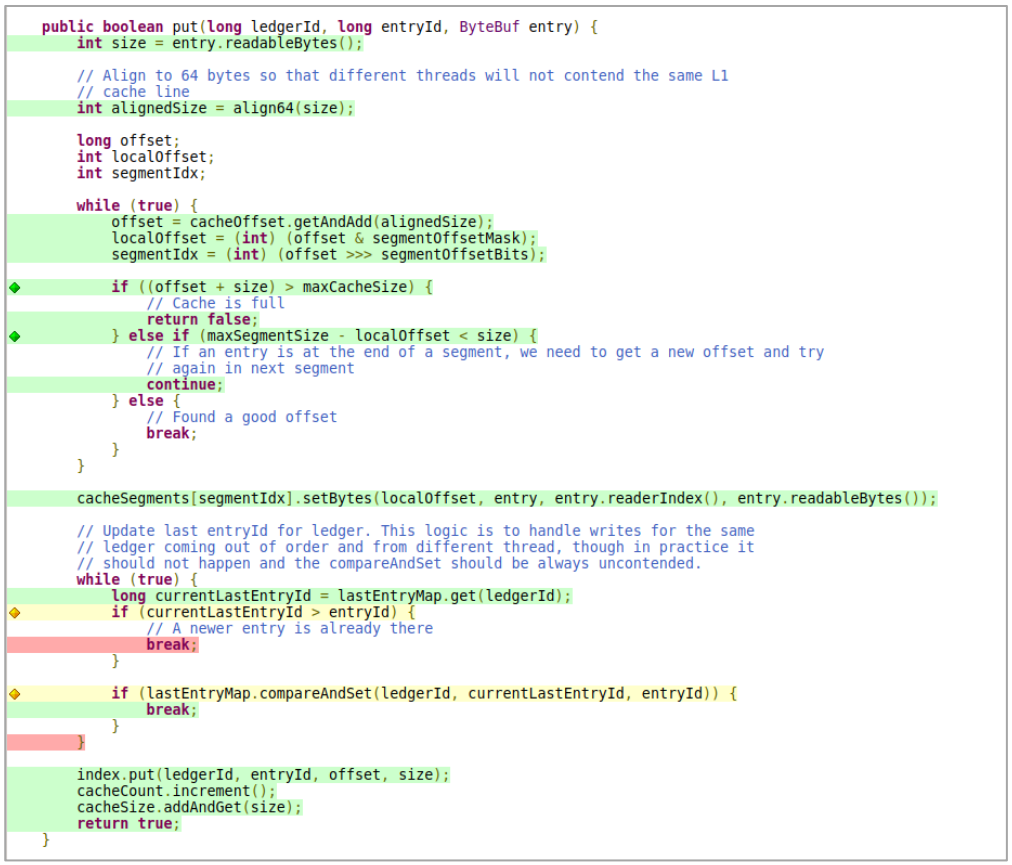
```
    public boolean put(long ledgerId, long entryId, ByteBuf entry) {
        int size = entry.readableBytes();

        // Align to 64 bytes so that different threads will not contend the same L1
        // cache line
        int alignedSize = align64(size);

        long offset;
        int localOffset;
        int segmentIdx;

        while (true) {
            offset = cacheOffset.getAndAdd(alignedSize);
            localOffset = (int) (offset & segmentOffsetMask);
            segmentIdx = (int) (offset >>> segmentOffsetBits);

            if ((offset + size) > maxCacheSize) {
                // Cache is full
                return false;
            } else if (maxSegmentSize - localOffset < size) {
                // If an entry is at the end of a segment, we need to get a new offset and try
                // again in next segment
                continue;
            } else {
                // Found a good offset
                break;
            }
        }

        cacheSegments[segmentIdx].setBytes(localOffset, entry, entry.readerIndex(), entry.readableBytes());

        // Update last entryId for ledger. This logic is to handle writes for the same
        // ledger coming out of order and from different thread, though in practice it
        // should not happen and the compareAndSet should be always uncontended.
        while (true) {
            long currentLastEntryId = lastEntryMap.get(ledgerId);
            if (currentLastEntryId > entryId) {
                // A newer entry is already there
                break;
            }

        index.put(ledgerId, entryId, offset, size);
        cacheCount.increment();
        cacheSize.addAndGet(size);
        return true;
    }
```

Figura 2: Analisi di Statement Coverage e Branch Coverage all'interno del metodo WriteCache.put().

WriteCache

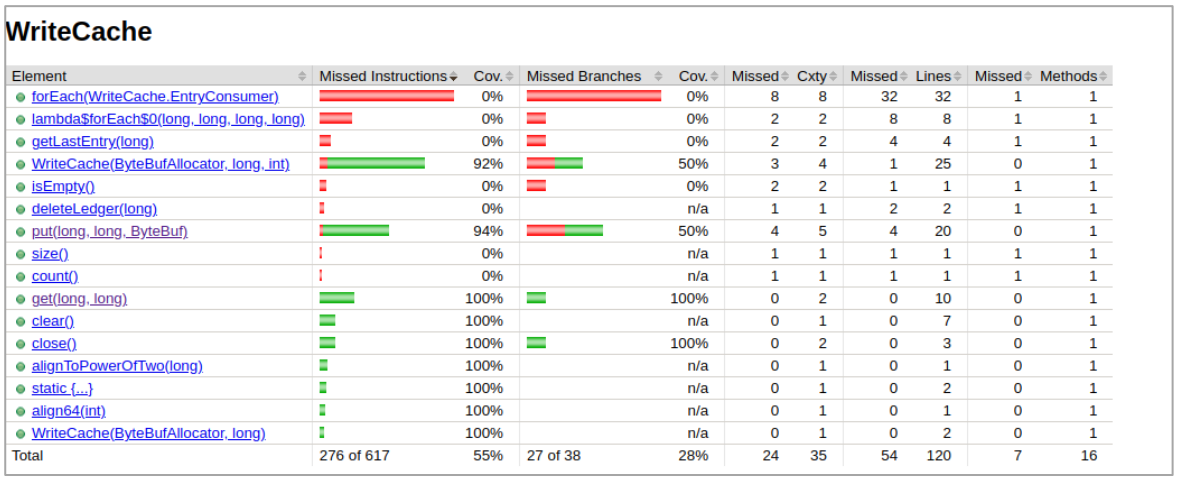
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
forEach(WriteCache.EntryConsumer)		0%		0%	8	8	32	32	1	1
get(long, long)		0%		0%	2	2	10	10	1	1
lambda$forEach$0(long, long, long, long)		0%		0%	2	2	8	8	1	1
getLastEntry(long)		0%		0%	2	2	4	4	1	1
isEmpty()		0%		0%	2	2	1	1	1	1
deleteLedger(long)		0%		n/a	1	1	2	2	1	1
size()		0%		n/a	1	1	1	1	1	1
count()		0%		n/a	1	1	1	1	1	1
WriteCache(ByteBufAllocator, long, int)		98%		66%	2	4	0	25	0	1
put(long, long, ByteBuf)		97%		75%	2	5	2	20	0	1
clear()		100%		n/a	0	1	0	7	0	1
close()		100%		100%	0	2	0	3	0	1
alignToPowerOfTwo(long)		100%		n/a	0	1	0	1	0	1
static {...}		100%		n/a	0	1	0	2	0	1
align64(int)		100%		n/a	0	1	0	1	0	1
WriteCache(ByteBufAllocator, long)		100%		n/a	0	1	0	2	0	1
Total	311 of 617	49%	26 of 38	31%	23	35	61	120	8	16

Figura 3: Statement coverage e Branch coverage per il metodo WriteCache.put() dopo opportuni miglioramenti.



```
public boolean put(long ledgerId, long entryId, ByteBuf entry) {
    int size = entry.readableBytes();

    // Align to 64 bytes so that different threads will not contend the same L1
    // cache line
    int alignedSize = align64(size);

    long offset;
    int localOffset;
    int segmentIdx;

    while (true) {
        offset = cacheOffset.getAndAdd(alignedSize);
        localOffset = (int) (offset & segmentOffsetMask);
        segmentIdx = (int) (offset >>> segmentOffsetBits);

        if ((offset + size) > maxCacheSize) {
            // Cache is full
            return false;
        } else if (maxSegmentSize - localOffset < size) {
            // If an entry is at the end of a segment, we need to get a new offset and try
            // again in next segment
            continue;
        } else {
            // Found a good offset
            break;
        }
    }

    cacheSegments[segmentIdx].setBytes(localOffset, entry, entry.readerIndex(), entry.readableBytes());

    // Update last entryId for ledger. This logic is to handle writes for the same
    // ledger coming out of order and from different thread, though in practice it
    // should not happen and the compareAndSet should be always uncontended.
    while (true) {
        long currentLastEntryId = lastEntryMap.get(ledgerId);
        if (currentLastEntryId > entryId) {
            // A newer entry is already there
            break;
        }

        if (lastEntryMap.compareAndSet(ledgerId, currentLastEntryId, entryId)) {
            break;
        }
    }

    index.put(ledgerId, entryId, offset, size);
    cacheCount.increment();
    cacheSize.addAndGet(size);
    return true;
}
```

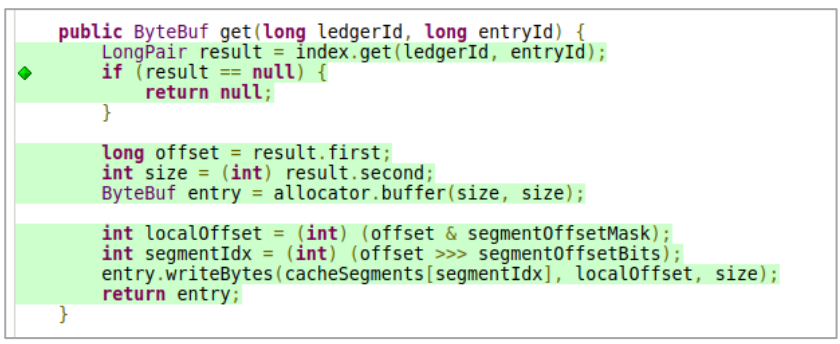
Figura 4: Analisi di Statement Coverage e Branch Coverage all'interno del metodo WriteCache.put() dopo opportuni miglioramenti.



WriteCache

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
forEach(WriteCache.EntryConsumer)		0%		0%	8	8	32	32	1	1
lambda$forEach$0(long, long, long, long)		0%		0%	2	2	8	8	1	1
getLastEntry(long)		0%		0%	2	2	4	4	1	1
WriteCache(ByteBufAllocator, long, int)		92%		50%	3	4	1	25	0	1
isEmpty()		0%		0%	2	2	1	1	1	1
deleteLedger(long)		0%		n/a	1	1	2	2	1	1
put(long, long, ByteBuf)		94%		50%	4	5	4	20	0	1
size()		0%		n/a	1	1	1	1	1	1
count()		0%		n/a	1	1	1	1	1	1
get(long, long)		100%		100%	0	2	0	10	0	1
clear()		100%		n/a	0	1	0	7	0	1
close()		100%		100%	0	2	0	3	0	1
alignToPowerOfTwo(long)		100%		n/a	0	1	0	1	0	1
static {...}		100%		n/a	0	1	0	2	0	1
align64(int)		100%		n/a	0	1	0	1	0	1
WriteCache(ByteBufAllocator, long)		100%		n/a	0	1	0	2	0	1
Total	276 of 617	55%	27 of 38	28%	24	35	54	120	7	16

Figura 5: Statement coverage e Branch coverage per il metodo WriteCache.get().



```
public ByteBuf get(long ledgerId, long entryId) {
    LongPair result = index.get(ledgerId, entryId);
    if (result == null) {
        return null;
    }

    long offset = result.first;
    int size = (int) result.second;
    ByteBuf entry = allocator.buffer(size, size);

    int localOffset = (int) (offset & segmentOffsetMask);
    int segmentIdx = (int) (offset >>> segmentOffsetBits);
    entry.writeBytes(cacheSegments[segmentIdx], localOffset, size);
    return entry;
}
```

Figura 6: Analisi di Statement Coverage e Branch Coverage all'interno del metodo WriteCache.get().

ReadCache

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
put(long, long, ByteBuf)		42%		25%	2	3	11	21	0	1
size()		0%		0%	4	4	9	9	1	1
count()		0%		0%	2	2	6	6	1	1
ReadCache(ByteBufAllocator, long, int)		100%		100%	0	2	0	12	0	1
get(long, long)		100%		100%	0	3	0	13	0	1
ReadCache(ByteBufAllocator, long)		100%		n/a	0	1	0	2	0	1
close()		100%		n/a	0	1	0	2	0	1
Total	152 of 356	57%	11 of 18	38%	8	16	26	65	2	7

Figura 7: Statement coverage e Branch coverage per il metodo ReadCache.get().

```
    public ByteBuf get(long ledgerId, long entryId) {
        lock.readLock().lock();

        try {
            // We need to check all the segments, starting from the current one and looking
            // backward to minimize the
            // checks for recently inserted entries
            int size = cacheSegments.size();
            for (int i = 0; i < size; i++) {
                int segmentIdx = (currentSegmentIdx + (size - i)) % size;

                LongPair res = cacheIndexes.get(segmentIdx).get(ledgerId, entryId);
                if (res != null) {
                    int entryOffset = (int) res.first;
                    int entryLen = (int) res.second;

                    ByteBuf entry = allocator.directBuffer(entryLen, entryLen);
                    entry.writeBytes(cacheSegments.get(segmentIdx), entryOffset, entryLen);
                    return entry;
                }
            }
        } finally {
            lock.readLock().unlock();
        }

        // Entry not found in any segment
        return null;
    }
```

Figura 8: Analisi di Statement coverage e Branch coverage all'interno del metodo ReadCache.get().

LedgerEntriesImpl

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
getEntry(long)		64%		50%	2	3	1	6	0	1
releaseByteBuf()		0%		0%	2	2	5	5	1	1
recycle()		0%		n/a	1	1	3	3	1	1
close()		0%		n/a	1	1	2	2	1	1
create(List)		100%		100%	0	2	0	4	0	1
iterator()		100%		n/a	0	1	0	2	0	1
LedgerEntriesImpl(Recycler.Handle)		100%		n/a	0	1	0	3	0	1
static {...}		100%		n/a	0	1	0	1	0	1
Total	46 of 123	62%	4 of 8	50%	6	12	11	26	3	8

Figura 9 : Statement coverage e Branch coverage per il metodo LedgerEntriesImpl.create().

```
    public static LedgerEntriesImpl create(List<LedgerEntry> entries) {
        checkArgument(!entries.isEmpty(), "entries for create should not be empty.");
        LedgerEntriesImpl ledgerEntries = RECYCLER.get();
        ledgerEntries.entries = entries;
        return ledgerEntries;
    }
```

Figura 10 : Analisi di Statement coverage e Branch coverage all'interno del metodo LedgerEntriesImpl.create().

LedgerEntriesImpl

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
iterator()		0%		n/a	1	1	2	2	1	1
create(List)		94%		50%	1	2	0	4	0	1
getEntry(long)		100%		75%	1	3	0	6	0	1
releaseByteBuf()		100%		50%	1	2	0	5	0	1
recycle()		100%		n/a	0	1	0	3	0	1
LedgerEntriesImpl(Recycler.Handle)		100%		n/a	0	1	0	3	0	1
static {...}		100%		n/a	0	1	0	1	0	1
close()		100%		n/a	0	1	0	2	0	1
Total	10 of 123	91%	3 of 8	62%	4	12	2	26	1	8

Figura 11 : Statement coverage e Branch coverage per il metodo LedgerEntriesImpl.getEntry().

```java
@Override
public LedgerEntry getEntry(long entryId) {
    checkNotNull(entries, "entries has been recycled");
    long firstId = entries.get(0).getEntryId();
    long lastId = entries.get(entries.size() - 1).getEntryId();
    if (entryId < firstId || entryId > lastId) {
        throw new IndexOutOfBoundsException("required index: " + entryId
            + " is out of bounds: [ " + firstId + ", " + lastId + " ].");
    }
    return entries.get((int) (entryId - firstId));
}
```

Figura 12 : Analisi del Branch Coverage e Statement Coverage all'interno del metodo LedgerEntriesImpl.getEntry().

LedgerEntriesImpl

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
iterator()		0%		n/a	1	1	2	2	1	1
create(List)		94%		50%	1	2	0	4	0	1
getEntry(long)		100%		100%	0	3	0	6	0	1
releaseByteBuf()		100%		50%	1	2	0	5	0	1
recycle()		100%		n/a	0	1	0	3	0	1
LedgerEntriesImpl(Recycler.Handle)		100%		n/a	0	1	0	3	0	1
static {...}		100%		n/a	0	1	0	1	0	1
close()		100%		n/a	0	1	0	2	0	1
Total	10 of 123	91%	2 of 8	75%	3	12	2	26	1	8

Figura 13: Statement coverage e Branch coverage per il metodo LedgerEntriesImpl.getEntry() dopo opportuni miglioramenti.

```java
@Override
public LedgerEntry getEntry(long entryId) {
    checkNotNull(entries, "entries has been recycled");
    long firstId = entries.get(0).getEntryId();
    long lastId = entries.get(entries.size() - 1).getEntryId();
    if (entryId < firstId || entryId > lastId) {
        throw new IndexOutOfBoundsException("required index: " + entryId
            + " is out of bounds: [ " + firstId + ", " + lastId + " ].");
    }
    return entries.get((int) (entryId - firstId));
}
```

Figura 14 : Analisi di Statement Coverage e Branch Coverage all'interno del metodo LedgerEntriesImpl.getEntry() dopo opportuni miglioramenti.

BookKeeper

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
validateZooKeeper(ZooKeeper)		0%		0%	2	2	5	5	1	1
validateEventLoopGroup(EventLoopGroup)		0%		n/a	1	1	2	2	1	1
static {...}		100%		n/a	0	1	0	1	0	1
scheduleBookieHealthCheckIfEnabled(ClientConfiguration)		21%		50%	1	2	2	4	0	1
openLedgerNoRecovery(long, BookKeeper.DigestType, byte[])		0%		n/a	1	1	4	4	1	1
openLedger(long, BookKeeper.DigestType, byte[])		100%		n/a	0	1	0	4	0	1

Figura 15: Statement coverage e Branch coverage per il metodo BookKeeper.openLedger().

```java
public LedgerHandle openLedger(long lId, DigestType digestType, byte[] passwd)
        throws BKException, InterruptedException {
    CompletableFuture<LedgerHandle> future = new CompletableFuture<>();
    SyncOpenCallback result = new SyncOpenCallback(future);

    /*
     * Calls async open ledger
     */
    asyncOpenLedger(lId, digestType, passwd, result, null);

    return SyncCallbackUtils.waitForResult(future);
}
```

Figura 16: Analisi di Statement Coverage e Branch Coverage all'interno del metodo BookKeeper.openLedger().

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
4	47%	258/545	28%	79/278

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
org.apache.bookkeeper.bookie.storage.ldb	2	58%	110/191	28%	35/127
org.apache.bookkeeper.client	1	37%	121/327	23%	31/134
org.apache.bookkeeper.client.impl	1	100%	27/27	76%	13/17

Report generated by PIT 1.5.1

Figura 17: Mutation coverage per il progetto BookKeeper.

Pit Test Coverage Report

Package Summary

org.apache.bookkeeper.bookie.storage.ldb

Number of Classes	Line Coverage		Mutation Coverage	
2	58%	110/191	28%	35/127

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
ReadCache.java	59%	41/70	22%	10/46
WriteCache.java	57%	69/121	31%	25/81

Report generated by PIT 1.5.1

Pit Test Coverage Report

Package Summary

org.apache.bookkeeper.client

Number of Classes	Line Coverage		Mutation Coverage	
1	37%	121/327	23%	31/134

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
BookKeeper.java	37%	121/327	23%	31/134

Report generated by PIT 1.5.1

Pit Test Coverage Report

Package Summary

org.apache.bookkeeper.client.impl

Number of Classes	Line Coverage		Mutation Coverage	
1	100%	27/27	76%	13/17

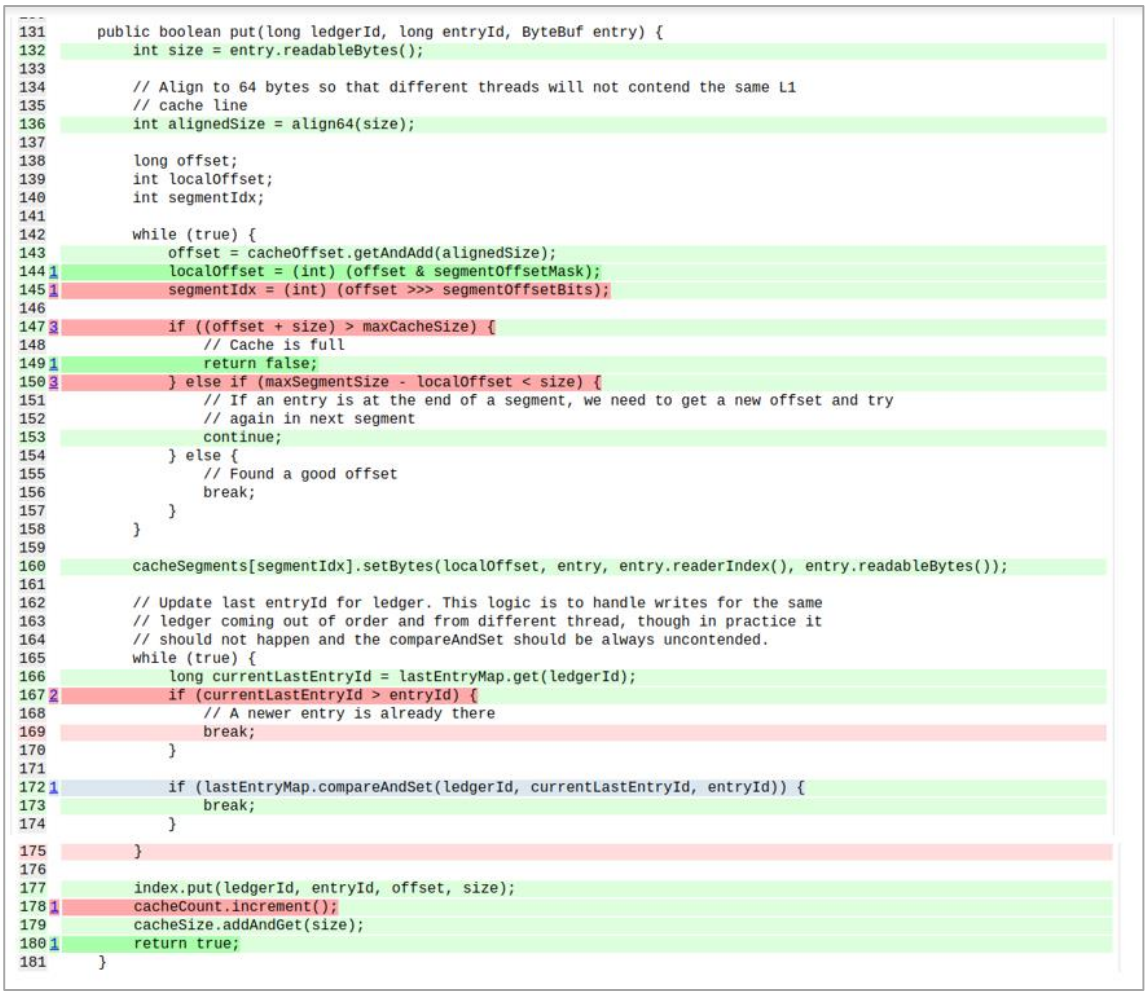
Breakdown by Class

Name	Line Coverage		Mutation Coverage	
LedgerEntriesImpl.java	100%	27/27	76%	13/17

Report generated by PIT 1.5.1

```
129      public ByteBuf get(long ledgerId, long entryId) {
130 1        lock.readLock().lock();
131
132          try {
133              // We need to check all the segments, starting from the current one and looking
134              // backward to minimize the
135              // checks for recently inserted entries
136              int size = cacheSegments.size();
137 3            for (int i = 0; i < size; i++) {
138 3                int segmentIdx = (currentSegmentIdx + (size - i)) % size;
139
140                  LongPair res = cacheIndexes.get(segmentIdx).get(ledgerId, entryId);
141 1                if (res != null) {
142                      int entryOffset = (int) res.first;
143                      int entryLen = (int) res.second;
144
145                      ByteBuf entry = allocator.directBuffer(entryLen, entryLen);
146                      entry.writeBytes(cacheSegments.get(segmentIdx), entryOffset, entryLen);
147 1                    return entry;
148                  }
149              }
150          } finally {
151 1            lock.readLock().unlock();
152          }
153
154          // Entry not found in any segment
155          return null;
156      }
```

Figura 18: Mutation coverage per ReadCache.get()



```
131     public boolean put(long ledgerId, long entryId, ByteBuf entry) {
132         int size = entry.readableBytes();
133
134         // Align to 64 bytes so that different threads will not contend the same L1
135         // cache line
136         int alignedSize = align64(size);
137
138         long offset;
139         int localOffset;
140         int segmentIdx;
141
142         while (true) {
143             offset = cacheOffset.getAndAdd(alignedSize);
144 1           localOffset = (int) (offset & segmentOffsetMask);
145 1           segmentIdx = (int) (offset >>> segmentOffsetBits);
146
147 3           if ((offset + size) > maxCacheSize) {
148                 // Cache is full
149 1               return false;
150 3           } else if (maxSegmentSize - localOffset < size) {
151                 // If an entry is at the end of a segment, we need to get a new offset and try
152                 // again in next segment
153                 continue;
154             } else {
155                 // Found a good offset
156                 break;
157             }
158         }
159
160         cacheSegments[segmentIdx].setBytes(localOffset, entry, entry.readerIndex(), entry.readableBytes());
161
162         // Update last entryId for ledger. This logic is to handle writes for the same
163         // ledger coming out of order and from different thread, though in practice it
164         // should not happen and the compareAndSet should be always uncontended.
165         while (true) {
166             long currentLastEntryId = lastEntryMap.get(ledgerId);
167 2           if (currentLastEntryId > entryId) {
168                 // A newer entry is already there
169                 break;
170             }
171
172 1           if (lastEntryMap.compareAndSet(ledgerId, currentLastEntryId, entryId)) {
173                 break;
174             }
175         }
176
177         index.put(ledgerId, entryId, offset, size);
178 1       cacheCount.increment();
179         cacheSize.addAndGet(size);
180 1       return true;
181     }
```

Figura 19: Figura 19: Mutation coverage per WriteCache.put().

```
183     public ByteBuf get(long ledgerId, long entryId) {
184         LongPair result = index.get(ledgerId, entryId);
185 1       if (result == null) {
186             return null;
187         }
188
189         long offset = result.first;
190         int size = (int) result.second;
191         ByteBuf entry = allocator.buffer(size, size);
192
193 1       int localOffset = (int) (offset & segmentOffsetMask);
194 1       int segmentIdx = (int) (offset >>> segmentOffsetBits);
195         entry.writeBytes(cacheSegments[segmentIdx], localOffset, size);
196 1       return entry;
197     }
```

Figura 20: Mutation coverage per WriteCache.get()

```
71      public static LedgerEntriesImpl create(List<LedgerEntry> entries) {
72  2       checkArgument(!entries.isEmpty(), "entries for create should not be empty.");
73          LedgerEntriesImpl ledgerEntries = RECYCLER.get();
74          ledgerEntries.entries = entries;
75  1       return ledgerEntries;
76      }
77
78      /**
79       * {@inheritDoc}
80       */
81      @Override
82      public LedgerEntry getEntry(long entryId) {
83          checkNotNull(entries, "entries has been recycled");
84          long firstId = entries.get(0).getEntryId();
85  1       long lastId = entries.get(entries.size() - 1).getEntryId();
86  4       if (entryId < firstId || entryId > lastId) {
87              throw new IndexOutOfBoundsException("required index: " + entryId
88                  + " is out of bounds: [ " + firstId + ", " + lastId + " ].");
89          }
90  2       return entries.get((int) (entryId - firstId));
91      }
```

Figura 21: Mutation coverage per LedgerEntriesImpl.create() e LedgerEntriesImpl.getEntry().

```
1273      public LedgerHandle openLedger(long lId, DigestType digestType, byte[] passwd)
1274              throws BKException, InterruptedException {
1275          CompletableFuture<LedgerHandle> future = new CompletableFuture<>();
1276          SyncOpenCallback result = new SyncOpenCallback(future);
1277
1278          /*
1279           * Calls async open ledger
1280           */
1281  1       asyncOpenLedger(lId, digestType, passwd, result, null);
1282
1283  1       return SyncCallbackUtils.waitForResult(future);
1284      }
1285
```

Figura 22: Mutation coverage per BookKeeper.openLedger().

8 IMMAGINI – SYNCOPE

FormatUtils

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
format(long, String)		0%		0%	2	2	7	7	1	1
format(double, String)		0%		0%	2	2	7	7	1	1
parseDate(String, String)		0%		n/a	1	1	4	4	1	1
parseNumber(String, String)		0%		n/a	1	1	3	3	1	1
lambda$static$0()		0%		n/a	1	1	3	3	1	1
format(Date, boolean)		0%		n/a	1	1	1	1	1	1
clear()		0%		n/a	1	1	3	3	1	1
format(Date)		0%		n/a	1	1	1	1	1	1
format(long)		0%		n/a	1	1	1	1	1	1
format(double)		0%		n/a	1	1	1	1	1	1
parseDate(String)		0%		n/a	1	1	1	1	1	1
format(Date, boolean, String)		100%		100%	0	2	0	6	0	1
static {...}		100%		n/a	0	1	0	2	0	1
Total	103 of 130	20%	4 of 6	33%	13	16	32	40	11	13

Figura 23: Statement coverage e Branch coverage per il metodo FormatUtils.format().

```
public static String format(final Date date, final boolean lenient, final String conversionPattern) {
    SimpleDateFormat sdf = DATE_FORMAT.get();

    if (conversionPattern == null) {
        sdf.applyPattern(SyncopeConstants.DEFAULT_DATE_PATTERN);
    } else {
        sdf.applyPattern(conversionPattern);
    }

    sdf.setLenient(lenient);

    return sdf.format(date);
}
```

Figura 24: Analisi del Branch Coverage e Statement Coverage all'interno del metodo FormatUtils.format().

FormatUtils

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
format(long, String)		0%		0%	2	2	7	7	1	1
format(double, String)		0%		0%	2	2	7	7	1	1
format(Date, boolean, String)		0%		0%	2	2	6	6	1	1
parseNumber(String, String)		0%		n/a	1	1	3	3	1	1
lambda$static$0()		0%		n/a	1	1	3	3	1	1
format(Date, boolean)		0%		n/a	1	1	1	1	1	1
clear()		0%		n/a	1	1	3	3	1	1
format(Date)		0%		n/a	1	1	1	1	1	1
format(long)		0%		n/a	1	1	1	1	1	1
format(double)		0%		n/a	1	1	1	1	1	1
parseDate(String)		0%		n/a	1	1	1	1	1	1
parseDate(String, String)		100%		n/a	0	1	0	4	0	1
static {...}		100%		n/a	0	1	0	2	0	1
Total	109 of 130	16%	6 of 6	0%	14	16	34	40	11	13

Figura 25: Statement coverage e Branch coverage per il metodo FormatUtils.parseDate().

```
public static Date parseDate(final String source, final String conversionPattern) throws ParseException {
    SimpleDateFormat sdf = DATE_FORMAT.get();
    sdf.applyPattern(conversionPattern);
    sdf.setLenient(false);
    return sdf.parse(source);
}
```

Figura 26: Analisi del Branch Coverage e Statement Coverage all'interno del metodo FormatUtils.parseDate().

FormatUtils

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
format(long, String)		0%		0%	2	2	7	7	1	1
format(double, String)		0%		0%	2	2	7	7	1	1
format(Date, boolean, String)		0%		0%	2	2	6	6	1	1
parseDate(String, String)		0%		n/a	1	1	4	4	1	1
format(Date, boolean)		0%		n/a	1	1	1	1	1	1
clear()		0%		n/a	1	1	3	3	1	1
format(Date)		0%		n/a	1	1	1	1	1	1
format(long)		0%		n/a	1	1	1	1	1	1
format(double)		0%		n/a	1	1	1	1	1	1
parseDate(String)		0%		n/a	1	1	1	1	1	1
parseNumber(String, String)		100%		n/a	0	1	0	3	0	1
lambda$static$0()		100%		n/a	0	1	0	3	0	1
static {...}		100%		n/a	0	1	0	2	0	1
Total	102 of 130	21%	6 of 6	0%	13	16	32	40	10	13

Figura 27: Statement coverage e Branch coverage per il metodo FormatUtils.parseNumber().

```
public static Number parseNumber(final String source, final String conversionPattern) throws ParseException {
    DecimalFormat df = DECIMAL_FORMAT.get();
    df.applyPattern(conversionPattern);
    return df.parse(source);
}
```

Figura 28: Analisi del Branch Coverage e Statement Coverage all'interno del metodo FormatUtils.parseNumber().

Encryptor

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
static {...}		57%		50%	6	7	14	31	0	1
verify(String, CipherAlgorithm, String)		0%		0%	5	5	11	11	1	1
getDigester(CipherAlgorithm)		57%		50%	2	3	6	16	0	1
Encryptor(String)		50%		50%	1	2	8	17	0	1
encode(String, CipherAlgorithm)		88%		62%	3	5	1	10	0	1
getInstance(String)		92%		50%	2	3	0	6	0	1
decode(String, CipherAlgorithm)		100%		100%	0	3	0	6	0	1
getInstance()		100%		n/a	0	1	0	1	0	1
Total	143 of 357	59%	22 of 42	47%	19	29	40	98	1	8

Figura 29 : Statement coverage e Branch coverage per il metodo Encryptor.decode().

```
public String decode(final String encoded, final CipherAlgorithm cipherAlgorithm)
        throws UnsupportedEncodingException, NoSuchAlgorithmException, NoSuchPaddingException, InvalidKeyException,
        IllegalBlockSizeException, BadPaddingException {

    String decoded = null;

    if (encoded != null && cipherAlgorithm == CipherAlgorithm.AES) {
        Cipher cipher = Cipher.getInstance(CipherAlgorithm.AES.getAlgorithm());
        cipher.init(Cipher.DECRYPT_MODE, keySpec);

        decoded = new String(cipher.doFinal(Base64.getDecoder().decode(encoded)), StandardCharsets.UTF_8);
    }

    return decoded;
}
```

Figura 30 : Analisi del Branch Coverage e Statement Coverage all'interno del metodo Encryptor.decode()

Encryptor

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
getDigester(CipherAlgorithm)		0%		0%	3	3	16	16	1	1
static {...}		57%		50%	6	7	14	31	0	1
Encryptor(String)		50%		50%	1	2	8	17	0	1
decode(String, CipherAlgorithm)		0%		0%	3	3	6	6	1	1
verify(String, CipherAlgorithm, String)		82%		87%	1	5	1	11	0	1
encode(String, CipherAlgorithm)		86%		75%	2	5	1	10	0	1
getInstance(String)		92%		50%	2	3	0	6	0	1
getInstance()		100%		n/a	0	1	0	1	0	1
Total	177 of 357	50%	20 of 42	52%	18	29	46	98	2	8

Figura 31: Statement coverage e Branch coverage per il metodo Encryptor.verify().

```
    public boolean verify(final String value, final CipherAlgorithm cipherAlgorithm, final String encoded) {
        boolean verified = false;

        try {
            if (value != null) {
                if (cipherAlgorithm == null || cipherAlgorithm == CipherAlgorithm.AES) {
                    verified = encode(value, cipherAlgorithm).equals(encoded);
                } else if (cipherAlgorithm == CipherAlgorithm.BCRYPT) {
                    verified = BCrypt.checkpw(value, encoded);
                } else {
                    verified = getDigester(cipherAlgorithm).matches(value, encoded);
                }
            }
        } catch (Exception e) {
            LOG.error("Could not verify encoded value", e);
        }

        return verified;
    }
```

Figura 32 : Analisi del Branch Coverage e Statement Coverage all'interno del metodo Encryptor.verify().

Encryptor

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
static {...}		57%		50%	6	7	14	31	0	1
getDigester(CipherAlgorithm)		57%		75%	1	3	6	16	0	1
Encryptor(String)		50%		50%	1	2	8	17	0	1
decode(String, CipherAlgorithm)		0%		0%	3	3	6	6	1	1
getInstance(String)		92%		50%	2	3	0	6	0	1
encode(String, CipherAlgorithm)		100%		87%	1	5	0	10	0	1
verify(String, CipherAlgorithm, String)		100%		100%	0	5	0	11	0	1
getInstance()		100%		n/a	0	1	0	1	0	1
Total	126 of 357	64%	15 of 42	64%	14	29	34	98	1	8

Figura 33: Statement coverage e Branch coverage per il metodo Encryptor.verify() in seguito a miglioramenti.

```
    public boolean verify(final String value, final CipherAlgorithm cipherAlgorithm, final String encoded) {
        boolean verified = false;

        try {
            if (value != null) {
                if (cipherAlgorithm == null || cipherAlgorithm == CipherAlgorithm.AES) {
                    verified = encode(value, cipherAlgorithm).equals(encoded);
                } else if (cipherAlgorithm == CipherAlgorithm.BCRYPT) {
                    verified = BCrypt.checkpw(value, encoded);
                } else {
                    verified = getDigester(cipherAlgorithm).matches(value, encoded);
                }
            }
        } catch (Exception e) {
            LOG.error("Could not verify encoded value", e);
        }

        return verified;
    }
```

Figura 34 : Analisi del Branch Coverage e Statement Coverage all'interno del metodo Encryptor.verify() dopo miglioramenti.

RealmUtils

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
getEffective(Set, String)		0%		0%	2	2	11	11	1	1
parseGroupOwnerRealm(String)		0%		0%	3	3	4	4	1	1
normalize(Collection)		0%		0%	2	2	5	5	1	1
lambda$normalize$0(Set, Set, String)		0%		0%	2	2	4	4	1	1
getGroupOwnerRealm(String, String)		0%		n/a	1	1	1	1	1	1
normalizingAddTo(Set, String)		100%		100%	0	6	0	12	0	1
Total	126 of 174	27%	10 of 20	50%	10	16	25	37	5	6

Figura 35: Statement coverage e Branch coverage per il metodo RealmUtils.normalizingAddTo().

```
    public static boolean normalizingAddTo(final Set<String> realms, final String newRealm) {
        boolean dontAdd = false;
        Set<String> toRemove = new HashSet<>();
        for (String realm : realms) {
            if (newRealm.startsWith(realm)) {
                dontAdd = true;
            } else if (realm.startsWith(newRealm)) {
                toRemove.add(realm);
            }
        }

        realms.removeAll(toRemove);
        if (!dontAdd) {
            realms.add(newRealm);
        }
        return !dontAdd;
    }
```

Figura 36: Analisi del Branch Coverage e Statement Coverage all'interno del metodo RealmUtils.normalizingAddTo().

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
1	71%	70/98	47%	17/36

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
org.apache.syncope.core.spring.security	1	71%	70/98	47%	17/36

Report generated by PIT 1.6.7

Figura 37: Mutation coverage per la classe Encryptor di Syncope.

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
1	24%	30/127	16%	12/77

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
org.apache.syncope.core.provisioning.api.utils	1	24%	30/127	16%	12/77

Report generated by PIT 1.6.7

Figura 38: Mutation coverage per le classi FormatUtils e RealmUtils di Syncope.

```
47      public static String format(final Date date, final boolean lenient) {
48  1       return format(date, lenient, null);
49      }
50
51      public static String format(final Date date, final boolean lenient, final String conversionPattern) {
52          SimpleDateFormat sdf = DATE_FORMAT.get();
53
54  1       if (conversionPattern == null) {
55  1           sdf.applyPattern(SyncopeConstants.DEFAULT_DATE_PATTERN);
56          } else {
57  1           sdf.applyPattern(conversionPattern);
58          }
59
60  1       sdf.setLenient(lenient);
61
62  1       return sdf.format(date);
63      }
```

Figura 39: Mutation coverage per FormatUtils.format().

```
103      public static Date parseDate(final String source) throws ParseException {
104 1        return DateUtils.parseDate(source, SyncopeConstants.DATE_PATTERNS);
105      }
106
107      public static Date parseDate(final String source, final String conversionPattern) throws ParseException {
108          SimpleDateFormat sdf = DATE_FORMAT.get();
109 1        sdf.applyPattern(conversionPattern);
110 1        sdf.setLenient(false);
111 1        return sdf.parse(source);
112      }
113
114      public static Number parseNumber(final String source, final String conversionPattern) throws ParseException {
115          DecimalFormat df = DECIMAL_FORMAT.get();
116 1        df.applyPattern(conversionPattern);
117 1        return df.parse(source);
118      }
119
```

Figura 40: Mutation coverage per FormatUtils.parseDate().

```
114      public static Number parseNumber(final String source, final String conversionPattern) throws ParseException {
115          DecimalFormat df = DECIMAL_FORMAT.get();
116 1        df.applyPattern(conversionPattern);
117 1        return df.parse(source);
118      }
```

Figura 41: Mutation coverage per FormatUtils.parseNumber().

```
43       public static boolean normalizingAddTo(final Set<String> realms, final String newRealm) {
44           boolean dontAdd = false;
45           Set<String> toRemove = new HashSet<>();
46           for (String realm : realms) {
47 1             if (newRealm.startsWith(realm)) {
48                 dontAdd = true;
49 1             } else if (realm.startsWith(newRealm)) {
50                 toRemove.add(realm);
51             }
52           }
53
54           realms.removeAll(toRemove);
55 1         if (!dontAdd) {
56             realms.add(newRealm);
57           }
58 2         return !dontAdd;
59       }
60
```

Figura 42: Mutation coverage per RealmUtils.normalizingAddTo().

```
193
194      public boolean verify(final String value, final CipherAlgorithm cipherAlgorithm, final String encoded) {
195          boolean verified = false;
196
197          try {
198 1            if (value != null) {
199 2                if (cipherAlgorithm == null || cipherAlgorithm == CipherAlgorithm.AES) {
200                      verified = encode(value, cipherAlgorithm).equals(encoded);
201 1                } else if (cipherAlgorithm == CipherAlgorithm.BCRYPT) {
202                      verified = BCrypt.checkpw(value, encoded);
203                  } else {
204                      verified = getDigester(cipherAlgorithm).matches(value, encoded);
205                  }
206              }
207          } catch (Exception e) {
208              LOG.error("Could not verify encoded value", e);
209          }
210
211 2        return verified;
212      }
```

Figura 43: Mutation coverage per Encryptor.verify().

```
213
214     public String decode(final String encoded, final CipherAlgorithm cipherAlgorithm)
215             throws UnsupportedEncodingException, NoSuchAlgorithmException, NoSuchPaddingException, InvalidKeyException,
216             IllegalBlockSizeException, BadPaddingException {
217
218         String decoded = null;
219
220 2       if (encoded != null && cipherAlgorithm == CipherAlgorithm.AES) {
221             Cipher cipher = Cipher.getInstance(CipherAlgorithm.AES.getAlgorithm());
222 1           cipher.init(Cipher.DECRYPT_MODE, keySpec);
223
224             decoded = new String(cipher.doFinal(Base64.getDecoder().decode(encoded)), StandardCharsets.UTF_8);
225         }
226
227 1       return decoded;
228     }
229
```

Figura 44: Mutation coverage per Encryptor.decode().