Università degli Studi di Roma Tor Vergata
Corso di Laurea Magistrale in Ingegneria Informatica
Ingegneria del Software II (9 CFU) A.A. 2020/2021
Modulo “Software Testing” - Docente Dott. Guglielmo De Angelis

Report sulle attività di sperimentazione dei progetti relativi al modulo di Software Testing

Massimo Stanzione, matr. 0304936

Introduzione

In questo documento si riportano i risultati ottenuti dai processi di sperimentazione delle tecniche di Software Testing apprese durante lo svolgimento del corso di *Ingegneria del Software II*, con particolare riferimento al modulo di *Software Testing* (3 CFU), tenuto dal docente Dott. Guglielmo De Angelis.
Le attività di sperimentazione hanno riguardato l’upgrading alla versione 4 e la parametrizzazione di casi di test del progetto software Apache BookKeeper™ [BK] prodotti per il framework *JUnit* versione 3 (progetto “1+”), nonché l’implementazione pratica delle tecniche di testing approfondite durante il corso allo stesso progetto [BK] e al progetto Apache OpenJPA™ [OJ] (progetti “2”). Nell’analisi si tiene anche conto, a scopo comparativo, di altre attività svolte durante lo svolgimento delle lezioni e dei progetti relativi al modulo *Machine Learning for SE* dello stesso corso.

Frameworks ed ambienti di sviluppo utilizzati

Le attività descritte sono state realizzate mediante scritture di test suites in linguaggio Java utilizzando *JUnit* 4 come framework di test. La copertura strutturale dei test è stata valutata mediante l’ausilio del framework *JaCoCo* (Java Code Coverage), i test di mutazione sono stati effettuati mediante framework *Pitest*.
A livello di sviluppo locale, il software è stato prodotto nell’ambiente *Eclipse* 2020-09 su *Debian* 9 e in *IntelliJ IDEA* 2021.2.3 su *Linux Mint* 20.2, e l’esecuzione è stata affidata a profili specifici specificati in *Maven*.

Integrazione CI/CT

A scopo di integrazione CI/CT, sono stati descritti appositi workflows in linguaggio YAML, eseguiti – in maniera differente a seconda del progetto considerato – con l’ausilio della piattaforma *CircleCI*, scelta in base ai criteri descritti nella *Deliverable 3* del modulo di *Machine Learning for Software Engineering* del corso [del3].

Progetto “1+”: Apache JCS™

Scopo del progetto è la trasposizione alla versione 4 del framework di testing *JUnit* di due classi di test del progetto Apache JCS™ 1.3 prodotte con la precedente versione 3, con implementazione di test parametrici e valutazione della variazione in copertura strutturale sulla base del variare dei parametri dei test.
La repository del progetto non contiene il codice sorgente di Apache JCS™, che è stato importato tramite dipendenza Maven a scopo di compilazione ed esecuzione del test. Per questo motivo, l’analisi della copertura strutturale con JaCoCo non ha avuto a disposizione i file Java delle classi in esame, e si è dunque proceduto ad una instrumentazione “in loco”, mediante lo script `jacocoPostExec.sh` nel quale viene invocata la JaCoCo CLI per procedere all’instrumentazione di un jar, allegato al codice, contenente i files necessari, per poi processare il “fat-jar” ottenuto insieme al report prodotto dall’esecuzione. Va notato che, comunque, la JaCoCo CLI accetta solamente il jar originale, in luogo di quello instrumentato.

Integrazione CI/CT

Il ciclo di test è stato totalmente integrato in un workflow automatico, azionato dai commit alla repository. In dettaglio, il ciclo prevede le fasi `clean` e `test` di maven (seguite da || true per garantire un output positivo a CircleCI e continuare con le fasi successive, ignorando il test *flaky* di seguito descritto) e l’esecuzione del citato script `jacocoPostExec.sh`. A valle di queste fasi, non essendo possibile eseguire l’integrazione con SonarCloud in assenza di codice sorgente da analizzare, i report prodotti vengono copiati in una cartella temporanea e salvati nella scheda *“Artifacts”* delle build di CircleCI [1+artifacts] per una durata di 30 giorni dall’ultima esecuzione del workflow.

Analisi delle classi e copertura strutturale

`org.apache.jcs.engine.EventQueueConcurrentLoadTest`

La classe prodotta con JUnit 3 presenta al suo interno una sottoclasse `CacheListenerImpl`, come implementazione *"dummy"* dell'interfaccia `ICacheListener` appositamente predisposta per il test. Si è deciso di mantenerla all'interno del progetto insieme alle classi di test, posizionandola in un file Java separato mantenuto nello stesso pacchetto, a scopo di accesso statico ad essa.
Una eventuale soluzione alternativa, considerata ma non attuata, sarebbe stata quella di utilizzare un *mock* dell'interfaccia `ICacheListener`, similmente a quanto eseguito nei progetti "2", ma si è deciso di non implementarla in quanto ciò sarebbe stato un inutile appesantimento del codice dal momento che `CacheListenerImpl` è già una implementazione sufficientemente utile allo scopo del progetto.
Il *porting* in JUnit 4 e la parametrizzazione dei test hanno incluso anche l'implementazione di una classe `EventQueueConcurrentLoadTestParams` per la gestione unitaria dei parametri di test, legati ai test stessi mediante il metodo `configure`, e l'implementazione del metodo `setup`, equivalente di `setUp`, annotato come `@BeforeClass`, da eseguirsi all'inizializzazione dell'ambiente di test.
Dal punto di vista della logica di test, la classe si presenta con un metodo centralizzato utilizzato per la chiamata degli effettivi metodi implementanti, sulla base di un parametro di tipo `String` utilizzato per individuare il metodo chiamato. Tutti i test implementanti agiscono con diverse operazioni su una coda condivisa, istanza di `CacheEventQueue`, con operazioni di aggiunta, rimozione, stop ed aggiunta con ritardo. Gli altri parametri, condivisi da tutti i metodi, sono di tipo `int`, rappresentati da `end`, riferito al numero di eventi Put/Remove da eseguire, ed `expectedPutCount`, valore minimo atteso per le operazioni di Put. Nei casi in cui i metodi implementati richiedano soltanto uno o nessuno dei due parametri, i valori non necessari sono stati sostituiti da un arbitario `int dontCare=-1`.

Test flakiness

Pur avendo lasciato inalterato il test program, nel passaggio a JUnit 4 si è notato che i test di tipo *RunPutDelay* presentavano fallimenti in maniera arbitraria, minando la stabilità della suite di test. Il problema sembra ricaderre nella classe di problematiche nota con il nome di *flakiness*.
Allo scopo di affrontare il problema è stata creata una branch dedicata [flakyBranch], a scopo di "sandbox" nella quale sperimentare alcune tra le tecniche di mitigazione considerate, di cui si è tenuto traccia nella issue [flakyIssue]. La problematica più importante da affrontare è stata costituita dall'inalterabilità della logica di test. Per tale motivo si è dovuto ricorrere ad un approccio più implementativo, di tipo *white-box*, cercando di verificare il comportamento dei metodi di test considerando la loro implementazione.
La causa del problema è probabilmente riferita al fatto che l'ordine di esecuzione dei test non è garantito, e ciò rappresenta una criticità dal momento che i test in esame sono sequenzialmente dipendenti. Le tecniche di mitigazione attuate sono state le seguenti:

- Implementazione di un **timeout**: nel caso in cui il test rimanga bloccato per un tempo indefinito, il test deve terminare all'interno una limitata finestra temporale.
- Implementazione di un numero di **test retry**: unito all'esecuzione Maven nel plugin Surefire, mediante il parametro `-Dsurefire.rerunFailingTestsCount`, fa sì che il test che presenta fallimento venga nuovamente inviato in esecuzione per un numero finito di tentativi. Soluzione scartata, in quanto comportava uno spreco di risorse e tempo di esecuzione.
- Uso dell'annotazione **`@ResourceLock`**, non realmente effettiva in quanto la risorsa è utilizzata in una sequenza di test e non in una effettiva concorrenza
- Uso dell'annotazione **`@FixMethodOrder`**`(MethodSorters.DEFAULT)`: non applicabile, in quanto l'ordine considerato è basato sui metodi annotati come `@Test`, e nella fattispecie si ha un solo metodo con tale annotazione, situazione non modificabile a scopo di lasciare invariata la logica di test originaria.

A margine, una possibile soluzione implementativa, che però andava a violare il requisito di inviolabilità della logica originaria dei test, consiste nell'anteporre l'uso del metodo `destroy()` esposto da `CacheEventQueue` all'esecuzione del blocco `synchronized`, a scopo di rendere sempre libera la coda, ed evitare così il *deadlock*.
Per quanto esposto, la problematica non è stata risolta, ma ciò ha comunque reso possibile l'esecuzione delle fasi successive del progetto.

Analisi della copertura strutturale

Analizzando il report prodotto tramite JaCoCo [1+artifacts] si nota una copertura strutturale di tipo *statement coverage* del **47%**, ed una di tipo *branch coverage* del **32%** per la classe in esame, `CacheEventQueue`.
A scopo di miglioramento di tali valori si è agito sui parametri `end` ed `expectedPutCount`. Nel dettaglio, si è notato che nei casi di test proposti non è mai stato utilizzato un valore del primo maggiore del secondo. Si

è dunque proceduto all'implementazione del caso di test, così come per l'introduzione di valori nulli o negativi, ma la copertura strutturale è rimasta invariata.

`org.apache.jcs.engine.memory.lru.LRUMemoryCacheConcurrentUnitTest`

La classe si presenta con una struttura molto più semplice rispetto alla precedente: vi è presente un solo test, con più asserzioni ma solo un parametro, di tipo `String`, riferito ad un identificativo di una particolare regione di memoria.
La classe sotto test è `LRUMemoryCache`, che nella versione JUnit 3 del test risulta essere inizializzata già all'interno del test e non in un metodo separato. Tale scelta, seppur poco ottimale (sarebbe stato preferibile utilizzare un metodo `setUp`) è stata preservata allo scopo di non mutare la logica di test. Il test, a livello implementativo, attua una inizializzazione, un inserimento ed una rimozione di elementi da una cache gestita con metodologia LRU.

Analisi della copertura strutturale

Analizzando il report prodotto da JaCoCo [1+artifacts] si nota una copertura strutturale di tipo *statement coverage* del **51%**, ed una di tipo *branch coverage* del **38%** per la classe in esame.
A scopo di miglioramento dei valori si è agito sull'unico parametro disponibile, `region`, cercando di inserire valori nulli, stringhe vuote o parametri palesemente non validi. La copertura strutturale, nonostante ciò, è rimasta invariata.

Progetto "2": Apache Bookkeeper™

La versione selezionata per il progetto in analisi è la release 4.6.0, rilasciata il 10 novembre 2017. Nonostante sia meno recente rispetto alle ultime disponibili e vi siano alcune differenze nella struttura dei progetti interni Maven, tale scelta è motivata dal voler includere anche l'analisi delle metriche utilizzate nella *Deliverable 2* del modulo *Machine Learning for Software Engineering* del corso.

Integrazione CI/CT

Il ciclo di test è stato totalmente integrato in un workflow automatico, azionato dai commit alla repository. In dettaglio, il ciclo prevede la fase `clean` di maven, seguita dall'invocazione del plugin jacoco, la fase `test` di Maven eseguita su profilo "coverage" e sul package di interesse (con specifica del package principale) e successivo inoltro del report all'analisi SonarCloud.

Test di mutazione

Per quanto riguarda i test di mutazione, essi sono stati volutamente esclusi dal ciclo di testing, in quanto esecuzioni locali hanno portato ad analisi di durata superiore ai 20 minuti, e aventi come risultato la presenza della quasi totalità di *time-out* nella validazione dei mutanti, rendendo l'analisi piuttosto onerosa e comunque non utilizzabile. Tale comportamento risulta comunque essere considerabile in questo tipo di analisi [PitTimeOut]. A riscontro di ciò, il report della più recente esecuzione di Pitest è stato reso disponibile manualmente nella cartella target del package bookkeeper-server.

Ulteriori note implementative

Rispetto al POM originario di Apache BookKeeper™ 4.6.0, sono state apportate le seguenti modifiche:

- upgrade del plugin `maven-surefire-plugin` alla versione 3.0.0-M4, per maggiore leggibilità
- esclusione della proprietà `reuseForks` allo stesso plugin
- upgrade del plugin `maven-shade-plugin` alla versione 3.2.1, a causa di problemi nell'esecuzione.

Scelta delle classi, metodologie e analisi

La scelta delle classi è stata effettuata focalizzandosi su quelle che, secondo la documentazione [BK1], sono le tre entità fondamentali alla base del progetto: *Entry*, *Ledger*, *Bookie*: si è cercato di prediligere classi relative alla diretta manipolazione di esse. I criteri utilizzati sono riportati in Tabella 1. Nel dettaglio l'analisi è stata impostata sulle classi `LedgerHandle` e `FileInfo`.
Per quanto riguarda il risultato atteso dall'esecuzione di ogni test, si considera un approccio di tipo "black-box", indicando un valore preimpostato `FAIL` in caso di errore atteso, non potendo specificare nel dettaglio la tipologia di errore specifica.
Il valore `FAIL` ed altre costanti predisposte sono contenute in una apposita classe `org.apache.bookkeeper.util.ISW2TestUtils`.

`org.apache.bookkeeper.client.LedgerHandle`

La classe è stata selezionata in quanto effettivamente manipolante due delle tre entità fondamentali (*Entry* e *Ledger*, criterio C1), e per questo motivo dispongono di una apposita *Ledger API* provvista di adeguata

documentazione concettuale (criterio C2 [BK2][BK3]), orientata alle funzionalità stesse, che permette di poter avere un approccio quanto più possibile di tipo *black-box* al problema. Ad una analisi delle metriche Machine Learning per la classe in questione ([del2]), si può notare che la classe è tra le più longeve, presente in tutte le versioni analizzate sin dalla prima versione del progetto, con una metrica *Size* consistente e quasi sempre in aumento, un numero di *LOC_Added* non elevato rispetto ad altre classi ma certamente non trascurabile, e sempre classificata come *buggy* nelle versioni analizzate in [del2]. Si nota, in particolare, che nella versione 4.4.0 tale classe è stata coinvolta in operazioni di modifica nei commit relativi alla versione stessa in misura ben più rilevante rispetto alle altre versioni e ad altre classi: essa risulta infatti tra quelle con il più alto numero di revisioni riferite, nonché, sempre per la stessa versione, tra quelle con più autori coinvolti nelle operazioni stesse. Questi elementi (criterio C3) rendono la classe in esame sicuramente meritevole di maggiore attenzione nell'ambito del testing. La scelta di questa classe è stata effettuata anche per affinità con l'attività di laboratorio [lab4], svolta durante il corso (criterio C4).
Si intende sottoporre ad attività di test le funzionalità messe a disposizione dalla classe per l'aggiunta e la lettura delle *Entry* per un *Ledger*, avendo cura di discernere i metodi mediante la documentazione esposta ed i criteri considerati.
La scrittura dei test è stata facilitata dalla presenza di una classe nativa `BookKeeperClusterTestCase`, cui è stato delegato il compito dell'inizializzazione dell'ambiente, e dalla presenza di appositi *snippets* di codice in documentazione, nonostante alcuni errori nella denominazione di alcuni tipi e metodi, per sopperire ai quali si è dovuto necessariamente inferire il codice sorgente del progetto.

```
long addEntry(byte[] data, int offset, int length)
```

Il parametro `data` rappresenta, secondo documentazione, un generico array di byte rappresentante il contenuto delll'*Entry* da scrivere sul *Ledger*. Non sono previste particolari limitazioni sul contenuto dell'array, né alcuna risulta desumibile dalla documentazione stessa.
I parametri interi `offset` e `length` rappresentano rispettivamente lo spiazzamento e la lunghezza della porzione di `data` da considerare. Anche in questo caso la documentazione non riporta limitazioni specifiche, perciò si possono solo desumere partizioni del dominio legate alla manipolazione di un array di bytes.

Category partitioning

Sulla base dei criteri riportati in Tabella 2 si individuano le seguenti classi di equivalenza
D(`data`)={`null`; vuoto; valido; fuori dal range `byte`} (critt. C1, C3, C6)
D(`offset`)={<0; ∈ [0, `len(data)`]; > `len(data)`} (critt. C1, C2, C6)
D(`length`)={<0; ∈ [0, `len(data)-offset`]; > `len(data)-offset`} (critt. C1, C2, C6).
Il criterio C3 non è stato considerato, in quanto è noto che per variabili di tipo `int` si ottiene sollvamento di eccezione.
Il criterio C5 è stato scartato, in quanto i valori estremi assumibili dal tipo `int` non soltanto sono assai poco probabili nel contesto in esame, ma rientrano nelle classi di equivalenza già considerate.

Boundary analysis

Si procede all'individuazione di valori limite per le classi di equivalenza indicate, osservando che ciò è valido per i parametri offset e length.
B(`data`) ≡ D(`data`)
B(`offset`) = {-1; 0; 1; `len(data)-1`; `len(data)`; `len(data)+1`}
B(`length`) = {-1; 0; 1; `len(data)-offset-1`; `len(data)-offset`; `len(data)-offset+1`}

Selezione finale

Allo scopo di ottenere una copertura quanto più possibile capillare dei risultati attesi, mantenendo allo stesso tempo l'obiettivo di giungere ad una test suite quanto più possibile minimale, si procede ad una selezione unidimensionale delle combinazioni dei parametri. Rimanendo in un approccio di tipo *"black box"* è possibile intuire che i risultati possibili possono includere, in caso di fallimento atteso, casistiche di entry il cui contenuto è vuoto, non determinabile o oltre il limite della lunghezza del buffer. Per tale motivo si cerca di ottenere, come ulteriore criterio, almeno una combinazione di parametri per ciascuna di queste categorie:

- Contenuto valido: {1, `len(data)-1`}
- Contenuto vuoto: {`len(data)-1`; 0}, {`len(data)`; 0}
- Contenuto non determinabile: {0; -1}, {-1; `len(data)`}
- Contenuto oltre la lunghezza del buffer: {`len(data)+1`; 1}, {1, `len(data)`}

Per i test sul parametro `data` si considererà sempre la scelta per il contenuto valido.

Coverage

Si desume dal codice che l'operazione di aggiunta della *Entry* ad un *Ledger* è effettuata operativamente dal metodo `AsyncAddEntry(byte[], int, int, Callback, Object)`, i cui primi tre parametri corrispondono a quelli in esame, e il cui valore di ritorno è coincidente.
Si osserva che la copertura strutturale corrisponde al **100%** sia per la *statement coverage* che per la *branch coverage*, situazione dovuta anche alla non elevata complessità del metodo.

Per quanto riguarda il metodo chiamante si osserva una *statement coverage* dell'**82** ed una *branch coverage* del **50%**, sui quali si è deciso di non intervenire in quanto il codice non raggiunto non ha influenza alcuna sulla strategia di test, essendo rilevante solo a scopo di *logging*.

```
Enumeration<LedgerEntry> readEntries(long firstEntry, long lastEntry)
```

I due parametri rappresentano rispettivamente l'indice della prima e dell'ultima *Entry* da prelevare da un *Ledger*. Non sono previste particolari limitazioni sui valori assumibile, né alcuna risulta desumibile dalla documentazione stessa.

Category partitioning

Si individuano le seguenti classi di equivalenza, secondo i criteri riportati in Tabella 2.
D(`firstEntry`)={<0; ≥ 0} (critt. C1, C6)
D(`lastEntry`)={<0; ∈ [0, `firstEntry`]; > `firstEntry`} (critt. C1, C6)
Un ulteriore raffinamento attuato è dato dal fatto che le prime due classi di equivalenza per `lastEntry` rientrano in realtà in un'unica classe coincidente: vista la stretta dipendenza funzionale tra i due parametri, ci si aspetta sempre un fallimento nel caso in cui `lastEntry` < `firstEntry`. Si considera dunque:
D(`lastEntry`)={< `firstEntry`; `firstEntry`; > `firstEntry`}
Il criterio C3 non è stato considerato, in quanto è noto che per variabili di tipo `long` si ottiene sollvamento di eccezione.
Il criterio C5 è stato scartato, in quanto i valori estremi assumibili dal tipo `long` non soltanto sono assai poco probabili nel contesto in esame, ma rientrano nelle classi di equivalenza già considerate.

Boundary analysis

Si procede all'individuazione di valori limite per le classi di equivalenza indicate.
B(`firstEntry`) = {-1; 0; 1}
B(`lastEntry`) = {-1; 0; 1; `firstEntry`-1; `firstEntry`; `firstEntry`+1}
Tenendo conto del raffinamento considerato nel paragrafo precedente:
B(`lastEntry`) = {`firstEntry`-1; `firstEntry`; `firstEntry`+1}

Selezione finale

Allo scopo di ottenere una copertura quanto più possibile capillare dei risultati attesi, mantenendo allo stesso tempo l'obiettivo di giungere ad una test suite quanto più possibile minimale, si procede ad una selezione unidimensionale delle combinazioni dei parametri. Rimanendo in un approccio di tipo *"black box"* è possibile intuire che i risultati possibili possono includere un quantitativo di *Entries* corretto e maggiore di 1, corretto e pari ad 1, non determinabile. Per tale motivo si cerca di ottenere, come ulteriore criterio, almeno una combinazione di parametri per ciascuna di queste categorie:

- Quantitativo valido: {1; 2}
- Quantitativo unitario: {1; 1}
- Quantitativo non determinabile: {-1; 0}, {0; -1}.

Coverage

Similmente al metodo analizzato precedentemente, si considera il metodo `AsyncReadEntries(long, long, ReadCallBack, Object)`, i cui primi due parametri corrispondono a quelli in esame e il cui valore di ritorno è coincidente.
Il report JaCoCo, riportato in Figura 1, fornisce l'indicazione che la copertura strutturale corrisponde al **63%** per la *statement coverage*, e all' **87%** per la *branch coverage*.
A scopo di miglioramento della copertura, in seguito ad una analisi del codice, si nota che nel partizionamento del dominio non sono state considerate partizioni comprendenti il numero di entries effettivamente scritte all'interno di un *Ledger*.
Per questo motivo si integra la test suite con i casi di test {0; numero di Entries − 1}, {0; numero di Entries}, {0, numero di Entries + 1}.
In seguito a tale adeguamento, la copertura raggiunge il **100%**.
Il metodo chiamante ha banalamente raggiunto sempre una copertura del **100%**.

`org.apache.bookkeeper.bookie.FileInfo`

Le motivazioni alla base della scelta della classe risiedono nella gestione a livello di astrazione poco elevato della persistenza: la classe funge da "adapter" rispetto al tipo nativo `File`, integrando parametri ed informazioni aggiuntive, rendendola così di particolare importanza (crit. C1). La documentazione disponibile è principalmente basata su Javadoc, mentre per quanto riguarda le metriche ML, la classe è presente sin dal primo rilascio, di Size non trascurabile e coinvolta in un elevato numero di commit, con numero di LOC Touched che raggiunge anche livelli rilevanti, particolarmente nella versione 4.2.2 (crit. C2). La non elevata complessità dei parametri ha inoltre contribuito alla selezione della classe (crit. C5).
Si intende sottoporre ad attività di test le funzionalità messe a disposizione dalla classe per la scrittura e lettura da/verso un file riferito da `FileInfo`.

Anche in questo caso la scrittura dei test è stata facilitata dalla presenza di una classe nativa `BookKeeperClusterTestCase`, cui è stato delegato il compito dell'inizializzazione dell'ambiente.
Tra le problematiche principali affronate si menzionano la gestione di *short write* e *short read*.

```
long write(ByteBuffer[] buffs, long position)
```

Il parametro `buffs` è relativo ad un array di buffer di bytes, che sarà scritto all'interno del file riferito da `FileInfo` a partire dalla posizione `position`.

Category partitioning

Si individuano le seguenti classi di equivalenza, secondo i criteri riportati in Tabella 2.
Secondo quanto appreso dall'analisi del metodo precedente, sarebbe opportuno considerare la dimensione del file:
D(`buffs`)={`null`; vuoto; valido; fuori dal range `byte`} (critt. C1, C3, C5)
D(`position`)={< 0; ∈ [0, lunghezza del file]; > lunghezza del file} (crit. C1, C6)
Tuttavia, non essendo definiti particolari limiti in documentazione, si pone:
D(`position`)={< 0; ≥ 0}
Per le motivazioni già esposte, al parametro `position` non sono stati applicati i criteri C3 e C5.

Boundary analysis

Si procede all'individuazione di valori limite per le classi di equivalenza indicate, osservando che ciò è valido per il parametro `position`.
B(`buffs`) ≡ D(`buffs`)
B(`position`) = {-1; 0; 1}

Selezione finale

Allo scopo di ottenere una copertura quanto più possibile capillare dei risultati attesi, mantenendo allo stesso tempo l'obiettivo di giungere ad una test suite quanto più possibile minimale, si procede ad una selezione unidimensionale delle combinazioni dei parametri. Vista l'esiguità dei casi di test, si decide di testare i valori {-1, 0, -1} per il parametro `position` in presenza di un `buffs` valido, e di provare le combinazioni dei parametri di `buffs` fissando position a 0.
Durante l'esecuzione dei test è emerso che, in presenza di valori di `position` negativi, essi vengono considerati come se valessero 0, non producendo così un atteso fallimento del test.

Coverage

Consultando il report prodotto da JaCoCo, riportato in Figura 2, è possibile osservare una copertura strutturale del **92%** per la *statement coverage*, e del **66%** per la *branch coverage*.
Dal report si nota che il codice non raggiunto si riferisce prevalentemente alle situazioni di *"short write"*, ossia di scritture che terminano prima di aver correttamente scritto tutto il contenuto del buffer, principalmente a causa del raggiungimento dell'EOF. In assenza di specifica documentazione si è provveduto alla consultazione dei dettagli implementativi, affrontando il problema su due fronti principali:

- rimanendo nell'ambito del category partitioning: dimensionamento opportuno di position e della dimensione di buff, con valori particolarmente elevati, e classi di equivalenza che tengono in considerazione il valore della costante START_OF_DATA;
- ancora più nel dettaglio: dimensionamento opportuno del file mediante l'ausilio della classe `RandomAccessFile`, ed assegnazione dei descrittori all'oggetto `FileInfo` sotto test.

Entrambi gli approcci, eseguiti sequenzialmente ed in combinazione tra loro, non hanno prodotto variazioni ai valori di copertura strutturale. Non potendo agire in modo diretto sulla classe `FileInfo`, si è deciso di considerare accettabili i valori ottenuti.

```
int read(ByteBuffer bb, long position, boolean bestEffort)
```

I parametri del metodo consistono, nell'ordine, in un buffer `bb` in cui inserire dati letti dal file riferito dall'istanza di `FileInfo`, letto a partire da `position`, e `bestEffort` che, se impostato al valore `true`, termina la lettura in caso di *short read* senza sollevare alcuna eccezione.

Category partitioning

Si individuano le seguenti classi di equivalenza, secondo i criteri riportati in Tabella 2.
Differente dai metodi precedenti, per il parametro corrispondente al tipo `ByteBuffer` si considerano soltanto un vettore di dimensione prefissata ed un vettore nullo, in quanto non avrebbe senso considerare il contenuto del buffer stesso, che sarà in ogni caso sovrascritto dalla lettura. I controlli di validità riguardano, infatti, principalmente il parametro `position`, opportunamente posto in combinazione con `bestEffort`.
Nel dettaglio, come per il metodo precedente si è in assenza di riferimenti per la dimensione del file, sia antecedentemente che susseguentemente a operazioni di scrittura. Per questo motivo, tale condizione non è

stata considerata in questa fase, lasciando eventuali raffinamenti della suite di test a fasi successive all'analisi del report JaCoCo.
D(`bb`)={`null`; valido} (critt. C1, C3, C6)
D(`position`)={< 0; ≥ 0} (critt. C1, C6)
D(`bestEffort`)={`true`, `false`} (critt. C1, C4, C6)
Il criterio C5 è stato scartato per il parametro `position`, per quanto visto in precedenza.

Boundary analysis

Similmente a quanto individuato in precedenza, si considerano i seguenti valori:
B(`bb`) ≡ D(`bb`)
B(`position`) = {-1; 0; 1}

Selezione finale

Allo scopo di ottenere una copertura quanto più possibile capillare dei risultati attesi, mantenendo allo stesso tempo l'obiettivo di giungere ad una test suite quanto più possibile minimale, si procede ad una selezione unidimensionale delle combinazioni dei parametri.
Allo scopo di ottimizzare l'insieme dei casi di test, per il test del buffer nullo sarà utilizzata la coppia di valori {0, `true`} per `position` e `bestEffort`, mentre per i test sui valori di `position` saranno considerati i valori {buffer valido, `true`} per i due parametri restanti.
I valori di `bestEffort` sono dunque sempre testati con valore `true`; il valore `false` sarà testato in condizioni di *short Read*, con `position` pari a 1.
Anche in questo caso, l'esecuzione dei test ha permesso di notare che, in presenza di valori di `position` negativi, essi vengono considerati come se valessero 0, non producendo così un atteso fallimento del test.

Coverage

Per la copertura strutturale si considera il metodo `readAbsolute(ByteBuffer bb, long start, boolean bestEffort)`, i cui primi tre parametri corrispondono a quelli in esame e il cui valore di ritorno è coincidente.
Il report prodotto da JaCoCo, riportato in Figura 3, mostra una copertura strutturale del **94%** per la *statement coverage*, e dell' **87%** per la *branch coverage*.
Si nota anzitutto che il caso di *short read* è stato raggiunto, differentemente da quanto accaduto con l'equivalente in scrittura.
Il codice non raggiunto consiste sostanzialmente nel contenuto di r. 348, non raggiunta dal branch di r. 347. Ad una analisi più approfondita, si riscontra che il parametro `fc`, istanza di `FileChannel`, è un parametro proprio di `FileInfo` riferente il file fisico, che per natura dei test prodotti non è stato preso in considerazione. Allo scopo di incrementare i valori di coverage senza dover aggiungere ulteriori parametri non richiesti e non direttamente rilevanti per il metodo in esame, si è tentato di eseguire una `close()` sull'oggetto `FileInfo` sotto esame, visibile a rr. 124-129, senza tuttavia ottenere maggiore copertura. Considerando la non diretta rilevanza dell'oggetto `FileChannel` rispetto all'obiettivo del progetto e alla gestione del test parametrico, si è deciso di accettare i valori di coverage ottenuti.

Progetto "2": Apache OpenJPA

La versione selezionata per il progetto in analisi è la release 3.1.2, rilasciata il 7 luglio 2020, in quanto la più recente con documentazione disponibile.
Questo progetto, rispetto al precedente, espone una documentazione molto più ampia e dettagliata, ma pressenta una struttura più complessa, in modo particolare per quanto riguarda le gerarchie di classi e parametri, con un più marcato orientamento alla definizione e realizzazione di interfacce.

Integrazione CI/CT

Il ciclo di test è stato totalmente integrato in un workflow automatico, azionato dai commit alla repository. In dettaglio, il ciclo prevede le fasi `clean` e `test` di Maven, seguita dall'esecuzione dell'analisi SonarCloud, con specificazione dei package di interesse (oltre al package principale).
I risultati dei test di mutazione, non potendo essere integrati in SonarCloud, sono stati automaticamente inseriti nella sezione *Artifact* delle build CircleCI [OJArtifacts], dove rimangono per una durata di 30 giorni dall'ultima esecuzione.

Ulteriori note implementative

Rispetto al POM originario di Apache BookKeeper™ 4.6.0, sono state apportate le seguenti modifiche:

- esclusione del plugin `apache-rat`
- esclusione delle proprietà `NewlineAtEndOfFile` e `LineLength` per il plugin `maven-checkstyle-plugin`, per maggiore praticità

Scelta delle classi, metodologie e analisi

La scelta delle classi è stata incentrata sul fatto che un processo rilevante alla base del progetto Apache OpenJPA è l'uso e la manipolazione degli identificatori, cui sono dedicate sezioni apposite in documentazione. I criteri utilizzati sono riportati in Tabella 1. Nel dettaglio l'analisi è stata impostata sulle classi `DBIdentifierImpl`, in quanto gestione ad elevato livello di astrazione di identificatori con strutture già definite, e `IdentifierUtilImpl`, per un approccio più "a basso livello" alla gestione degli identificatori.
Per quanto riguarda il risultato atteso dall'esecuzione di ogni test, si considera un approccio di tipo *"black-box"*, indicando un valore preimpostato `FAIL` in caso di errore atteso, non potendo specificare nel dettaglio la tipologia di errore specifica, oppure ci si avvale di un *oracolo* o un valore predefinito atteso nel caso vi sia la possibilità di poter definire valori attesi per i test.
Il valore FAIL ed altre costanti predisposte sono contenute in due apposita classe `org.apache.openjpa.jdbc.ISW2TestUtils` e `org.apache.bookkeeper.lib.ISW2TestUtils`, replicate a causa di problemi di dipendenze circolari tra i due moduli interessati.

`org.apache.openjpa.jdbc.identifier.DBIdentifierUtilImpl`

La classe è stata espone metodi per la gestione ad elevato livello di astrazione di operazioni riguardanti ogni aspetto degli identificativi (crit. C1) di varie tipologie supportate di basi di dati, che trovano riscontro nella sezione 4 del capitolo 4 della documentazione (crit. C2).
Tra i metodi esposti si intende sottoporre a test due metodi che riguardano funzionalità di trasformazione degli identificatori, in particolare il metodo per la standardizzazione di identificatori validi e il metodo di creazione di identificatori standardizzati.

```
DBIdentifier makeIdentifierValid(DBIdentifier sname, NameSet set, int maxLen,
    boolean checkForUniqueness)
```

Stando a quanto riportato in documentazione, il metodo preleva il nome dell'identificatore `sname`, lo tronca in lunghezza `maxLen`, ed effettua un controllo rispetto ad un elenco di parole riservate, giustapponendo un numero ordinale a partire da 0 in caso di esito positivo, per poi effettuare un ulteriore controllo di matching rispetto a nomi di identificatori contenuti in `set`, giustapponendo a sua volta un numero a partire da 0 in caso di matching.
Dall'esecuzione dei test e da un successivo controllo del metodo esaminato, tuttavia, si è pervenuti al riscontro di un errore nella documentazione citata: i numeri ordinali posti in append hanno inizio da 0 solamente in caso di controllo di parola riservata, mentre iniziano da 1 in caso di matching con elementi di `set`.

Category partitioning

Si individuano le seguenti classi di equivalenza, secondo i criteri riportati in Tabella 2.
D(`sname`)={ident. valido, ident. vuoto, ident. nullo, ident. con parole riservate, `null`} (critt. C1, C3, C6)
D(`set`)={vuoto, pieno, `null`} (critt. C1, C3, C6)
D(`maxLen`)={<0; ∈ [0, `len(idname)`-1]; ≥ `len(idname)`} (critt. C1, C6)
D(`checkForUniqueness`)={`true`, `false`} (critt. C1, C4, C6)
Per quanto già visto, per `maxLen` non si applicano i criteri C3 e C5.
A livello implementativo i nomi degli identificatori sono stati portati in uppercase a causa di un problema, documentato in [mivProblem], che portava a fallimenti non attesi dei test.

Boundary analysis

Per il parametro `maxLen` si procede ad una *boundary analysis*, individuando valori ai limiti estremi delle classi di equivalenza individuate.
B(`maxLen`)={-1; 0; 1; `len(idname)`-1, `len(idname)`; `len(idname)`+1}
Dall'esecuzione dei test è emerso che per `maxLen` i valori negativi, nulli e superiori a `len(idname)` vengono automaticamente considerati come `len(idname)`, diversamente da quanto atteso.

Selezione finale

Per la scelta dei valori da utilizzare nel test, oltre al criterio di copertura di tutte le classi di equivalenza e di ogni valore in *boundary analysis*, si considerano le seguenti regole per la scelta di una suite di test quanto più possibile minimale:

- per i test su `maxLen` si considerano fissati valori validi di `sname` e `set`, allo scopo di evidenziare le problematiche sollevate dai valori di `maxLen` con certezza;
- `checkForUniqueness`, se `true`, deve essere testato con identificativo valido (che si pone pari ad altro identificativo già esistente), ma in ogni caso per `set` sia pieno che vuoto. Per il valore `false` la scelta di `set` è indifferente.

Coverage

Si osserva dal report JaCoCo, riportato in Figura 4, che la copertura strutturale corrisponde al **73%** per la *statement coverage*, e al **53%** per la *branch coverage*. Analizzando il report i problemi individuati e le soluzioni attuate sono le seguenti:

(a) r. 160: non verificata la condizione per la quale un nome riservato ha lunghezza minore di `maxLen`;
(b) r. 167: non verificata la condizione in cui `set == null` e `checkForUniqueness == false`;
(c) rr. 171, 174: non sono stati provati identificatori di tipo TABLE e SEQUENCE;
(d) rr. 184, 187: non è stato verificato il caso in cui, per `checkForUniqueness == true`, vi siano più identificatori esistenti, ed in particolare in numero maggiore di 9, per poter aumentare il numero di caratteri posti in *append* ai nuovi identificatori;
(e) non è stato effettuato alcun controllo su identificatori delimitati.

Per cercare di ottenere copertura del codice relativo a questi punti di criticità ed allo stesso tempo cercare di modificare l'insieme dei parametri di test nel modo più minimale possibile, vengono messe in atto le seguenti misure:

- *problematica (b)*: dichiarazione esplicita delle condizioni `set == null` e `checkForUniqueness == false`: in luogo di aggiungere una nuova tupla nell'insieme di parametri di test si sceglie di riutilizzare opportunamente l'insieme avente `len(idname) - 1` come valore di `maxLen`, in modo da non alterare il caso già precedentemente specificato.
  ⇒ La *statement coverage* resta invariata, mentre la *branch coverage* raggiunge il **53%**.
- *problematica (c)*: inserimento, quali valori del parametro `sname`, di identificatori di tipo `DBIdentifierType.TABLE` e `DBIdentifierType.SEQUENCE`.
  ⇒ La *statement coverage* sale all'**82%**, la *branch coverage* raggiunge il **71%**.
- *problematica (d)*: ripetizione di un identificatore per almeno 9 volte in `set`: si sceglie di riutilizzare il valore relativo all'identificatore di tipo `DBIdentifierType.SEQUENCE` considerato nel punto precedente ed il set di default già in uso nell'insieme dei parametri.
  ⇒ La *statement coverage* raggiunge l'**85%**, la *branch coverage* l'**81%**.
- *problematica (e)*: implementazione di due *mock* `upperConf` e `lowerConf`, di tipo *"spia"* per l'interfaccia `IdentifierConfiguration`, in quanto tecnica utilizzata anche per altri test della stessa classe, ai quali vengono associati regole di delimitazione opportune applicabili per coprire i casi non raggiunti. I mock vengono utilizzati come parametro aggiuntivo rispetto ai parametri del test, ed impostati quale parametro di configurazione per l'istanza di `DBIdentifierUtil` sotto test. Nel dettaglio implementativo, il valore di `maxLen` deve tenere conto anche della presenza dei delimitatori.
  ⇒ La *statement coverage* raggiunge il **100%**, mentre la *branch coverage* sale al **96%**.
- *problematica (a)*: inserimento di una nuova tupla di valori – in quanto non risulta possibile riutilizzare tuple già presenti, opportunamente alterate – avente `sname` con nome appartenente all'insieme di parole riservate, e `maxLen` di valore maggiore rispetto alla lunghezza del nome stesso.
  Statement coverage e *branch coverage* raggiungono il **100%** di copertura.

Mutation testing

Al metodo in analisi vengono applicate **30** mutazioni, appertenenti alle tipologie Pitest `MATH`, `NEGATE_CONDITIONALS`, `INCREMENTS`, `CONDITIONALS_BOUNDARY` e `NULL_RETURNS`.
Dall'esecuzione dei test sui mutanti si ottiene che, ad esclusione di un *time-out* a r. 178 per una mutazione di tipo `NEGATE_CONDITIONALS`, si ha che **1** solo mutante, di tipo `CONDITIONALS_BOUNDARY`, non è stato rivelato a r. 189. Si ottiene dunque un *mutation score* pari a **0.96**.

```
DBIdentifier fromDBName(String name, DBIdentifierType id)
```

Il metodo, secondo documentazione, crea un nuovo identifiativo di tipo `id` a partire dal nome riportato nella stringa `name`.

Category partitioning

Si individuano le seguenti classi di equivalenza, secondo i criteri riportati in Tabella 2.
D(`name`)={valido, vuoto, `null`} (critt. C1, C3, C6)
D(`id`)={{tipologie di `DBIdentifier`}, null} (C1, C3, C4, C6)
In seguito all'esecuzione dei test, si nota che per name il valore `null` è trasformato nella stringa `"null"`, e per `id` il valore `null` è convertito automaticamente in `DBIdentifierType.NULL`. In entrambi i casi, quindi, i test non falliscono come inizialmente atteso.

Boundary analysis

Considerata la natura dei parametri esaminati, essi non si prestano ad una "reale" *boundary analysis*. Si procede pertanto alla selezione dei valori sulla base delle classi di equivalenza.

Selezione finale

Per la scelta dei valori da utilizzare nel test, oltre al criterio di copertura di tutte le classi di equivalenza e di ogni valore in boundary analysis, si considerano le seguenti regole per la scelta di una suite di test quanto più possibile minimale:

- per i test su name si considera fissato il parametrio id di tipo `DBIdentifierType.DEFAULT`;
- per i test su `id` si considera fissato il parametro `name` come stringa valida

Coverage

Il report prodotto da JaCoCo mostra una copertura strutturale molto bassa, corrispondente al **30%** per la *statement coverage*, ed appena al **17%** per la *branch coverage*.
Come si evince da Figura 5, possono essere individuate le seguenti problematiche che non hanno permesso di raggiungere una copertura strutturale soddisfacente:

(a) rr. 290, 300: non è stato considerato il non-supporto agli identificatori aventi nome delimitato: la configurazione di default comprende nativamente tale supporto;
(b) rr. 306 ~ 318: per gli identificatori avente nome delimitato, non sono state considerate le combinazioni di "case schema";
(c) rr. 322 ~ 324: in aggiunta al problema (b), non è stato considerata la regola di default di delimitazione per la configurazione legata a id.

Tutte le problematiche esposte sono legate a parametri interni alla classe sotto test, in particolare ad attributi dell'attributo di tipo `DBIdentifierConfiguration`. Data la scarsa copertura strutturale raggiunta, si decide di intervenire su tale attributo.
Per cercare di ottenere copertura del codice relativo si cerca di attuare strategie basate sul *mocking* dell'attributo in questione.

- *problematica (b)*: in seguito a verifica delle combinazioni di {`delimCase`, `nonDelimCase`} che possono soddisfare le condizioni non raggiungibili, e scelto un sottoinsieme minimale adatto allo scopo, si procede ad implementare un insieme di mock di tipo *"spia"* della classe `DefaultIdentifierConfiguration`, unica implementazione utilizzata, allo scopo di alterarne opportunamente il comportamento limitatamente ai soli metodi `getDelimitedCase` e `getSchemaCase`, in modo da poter assegnare tali mock come configurazioni dell'implementazione della classe sotto test mediante introduzione di un nuovo parametro nei casi di test, da passare all'oggetto `DBIdentifierUtil` all'inizio di ogni test.
- *problematica (c)*: con un approccio del tutto simile al caso precedente, viene implementato un mock di tipo *"spia"* avente come proprietà `delimitAll`.
  ⇒ La *statement coverage* e la *branch coverage* raggiungono il **96%**.
- *problematica (a)*: persistendo nell'approccio orientato ai *mock*, si agisce sulla proprietà `getSupportsDelimitedIdentifiers()`.
- ⇒ La *statement coverage* e la *branch coverage* raggiungono il **100%**.

Mutation testing

Al metodo in analisi vengono applicate **18** mutazioni, appertenenti alle tipologie Pitest `NEGATE_CONDITIONALS` e `NULL_RETURNS`.
Dall'esecuzione dei test sui mutanti non si ottiene alcun *time-out*, ma la test suite si dimostra assai meno robusta di quella considerata nel metodo precedente: ben **11** mutazioni, tutte di tipo `NEGATE_CONDITIONALS`, non vengono rivelate. Di conseguenza, il *mutation score* ottenuto risulta essere molto basso, circa **0.47**.

`org.apache.openjpa.lib.identifier.IdentifierUtilImpl`

La classe, differentemente da quanto valido per la precedente, espone metodi per la manipolazione degli identificatori ma ad un livello di astrazione meno elevato: le trasformazioni delle stringhe identificanti non è effettuata sulla base di standard di basi di dati pre-configurati, bensì a livello di gestione di delimitatori e operazioni dirette tra stringhe.
Si intende sottoporre a test il metodo per l'*append* di identificatori e il metodo per la rimozione dei delimitatori da essi.
Per entrambe le classi, non potendo utilizzare `DefaultIdentifierConfiguration` in quanto generante dipendenze circolari, si sceglie di avvalersi della già presente classe `IdConfigurationTestImpl`, applicando eventuali *mock* ove necessario.

```
String appendNames(IdentifierRule rule, String name1, String name2)
```

Il metodo, secondo documentazione, effettua una operazione di *append* tra `name1` e `name2`, preservando eventuali delimitatori, secondo la specifica regola `rule`.

Category partitioning

Si individuano le seguenti classi di equivalenza, secondo i criteri riportati in Tabella 2.

Per quanto riguarda il parametro `rule`, si è deciso di non considerare regole differenti da quanto di seguito riportato, in quanto prive di documentazione e/o non producenti alcun effetto su quanto eseguito dal metodo testato.
D(`rule`)={*rules* aventi le proprietà{`can_delimit`, `must_delimit`, `nullable`, `not_nullable`}, `null`}
D(`name1`)={senza delimitatore, inizia per delimitatore, con delimitatore nel mezzo, parola riservata, `null`}
D(`name2`)={senza delimitatore, termina per delimitatore, con delimitatore nel mezzo, parola riservata, `null`}
I criteri di scelta attuati sono C1, C3, C4, C6.

Boundary analysis

Considerata la natura dei parametri esaminati, essi non si prestano ad una "reale" *boundary analysis*. Si procede pertanto alla selezione dei valori sulla base delle classi di equivalenza.

Selezione finale

Rimanendo in un approccio di tipo *"black box"*, si considerano specifiche combinazioni di `name1` e `name2` atte alla costruzione di un insieme più possibile minimale. A tale scopo si intuiscono i comportamenti delle proprietà "can delimit" e "must delimit":

- "Can delimit": delimitazione effettuata se e solo se i delimitatori sono presenti ma posizionati in modo incorretto;
- "Must delimit": delimitazione sempre effettuata, a meno che la stringa in esame non sia già delimitata.

L'analisi di queste regole permetto di determinare un subset minimale di tutte le combinazioni di `name1` e `name2`, individuate come segue:
{`name1`, `name2`} = {{non delimitata, non delimitata}, {delimitata, delimitata}, {inizia per delim, finisce per delim.}, {non delmitata, delimitatore nel mezzo}, {parola riservata divisa tra i due parametri}}
Le regole nullable e not_nullable vengono analizzate ponendo:
{`name1`, `name2`} = {{stringa valida, null}, {null, null}}
Il valore `null` per il parametro rule viene provato con due stringhe valide per `name1` e `name2`.

Coverage

La copertura strutturale dei test prodotti risulta avere valore pari al **100%** sia per la *statement coverage* che per la *branch coverage*.

Mutation testing

Al metodo in analisi vengono applicate **5** mutazioni, appertenenti alle tipologie Pitest `NEGATE_CONDITIONALS` ed `EMPTY_RETURNS`.
Dall'esecuzione dei test sui mutanti non si ottiene alcun *time-out*, e non viene rilevata alcuna mutazione. Di conseguenza, il *mutation score* ottenuto risulta essere pari a **1**.

`String` **`removeDelimiters`**`(IdentifierConfiguration config, String rule, String name)`

Secondo documentazione, il metodo rimuove i delimitatori, relativi alla configurazione `config`, da name, seguendo la regola `rule`.
In seguito ad una successiva analisi del codice, si è pervenuti alla conclusione che, a livello implementativo, l'attributo `rule` non viene in realtà in alcun modo utilizzato, e che quindi non sortisce alcun tipo di cambiamento a livello di test suite.

Category partitioning

Si individuano le seguenti classi di equivalenza, secondo i criteri riportati in Tabella 2.
Avendo a disposizione solamente la citata `IdConfigurationTestImpl` come valore per il parametro `config`, si decide di individuarne un secondo artificialmente mediante mock della stessa.
Per l'attributo `rule` sono individuate regole specificatamente definite, principalmente basate su `canDelimit` e `mustDelimit`.
D(`config`)={di default, mock, `null`} (critt. C1,C3, C6)
D(`rule`)={rif. regola valida, rif. regola non esistente, `""`, `null`} (critt. C1, C3, C6)
D(`name`)={delim. secondo regola di default, delim. Secondo regola mock, `""`, `null`} (critt. C1, C3, C6)

Boundary analysis

Considerata la natura dei parametri esaminati, essi non si prestano ad una "reale" *boundary analysis*. Si procede pertanto alla selezione dei valori sulla base delle classi di equivalenza.

Selezione finale

Tra i parametri passati al test si considerano due tipi di `config`: `confTo`, assegnata all' istanza di `IdentifierUtilImpl`, secondo le cui regole si otterrà il valore di ritorno finale, e `confFrom`, passata come parametro a `removeDelimiters`, che servirà ad identificare i delimitatori presenti in name.
Per quanto riguarda i valori di `name`, è opportuno testare con entrambi i valori di `config` valori aventi delimitatori per entrambe le configurazioni.

I valori di rule vengono distribuiti utilmente in base alle regole assegnate ai due valori predisposti per config.
A scopo di selezione di un insieme di valori quanto più possibile minimale, i valori `""` vengono testati fissando valori di default per gli altri parametri, ed i valori `null` per le configurazioni vengono testati solo in sostituzione dell'istanza *mock* di config, in quanto già completamente manipolata nei casi precedenti.
In seguito all'esecuzione dei test è emerso che per valori nulli di `name` i test non falliscono come atteso, ma semplicemente utilizzano la stringa `"null"` in luogo del valore nullo.

Coverage

La copertura strutturale dei test prodotti risulta avere valore pari al **100%** sia per la *statement coverage* che per la *branch coverage*.

Mutation testing

Al metodo in analisi vengono applicate **6** mutazioni, appertenenti alle tipologie Pitest `NEGATE_CONDITIONALS`, `MATH` ed `EMPTY_RETURNS`.
Dall'esecuzione dei test sui mutanti non si ottiene alcun *time-out*, e non viene rilevata alcuna mutazione. Di conseguenza, il *mutation score* ottenuto risulta essere pari a **1**.

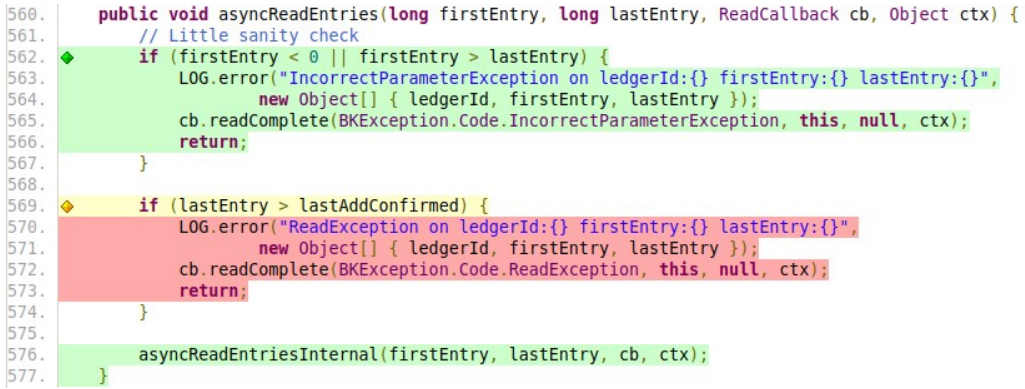
Illustrazioni e tabelle

ID	Descrizione
C1	Importanza concettuale
C2	Disponibilità di documentazione
C3	Metriche
C4	Affinità con lavori precedentemente svolti [lab4][proj1plus]
C5	Complessità dei parametri

Tabella 1: Criteri di selezione per le classi da analizzare nei progetti "2"

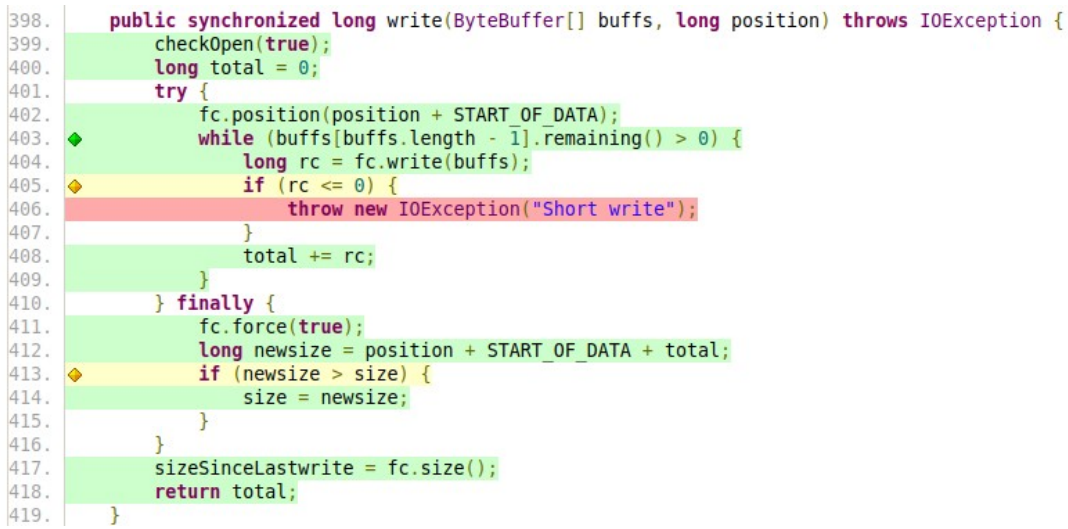
ID	Descrizione
C1	Contesto del dominio di input
C2	Condizioni dichiarate/ricavate in documentazione
C3	Valore nullo
C4	Definizione esplicita di tutti i valori ammissibili all'interno di una enumerazione
C5	Confronto con i valori di default assegati nell'implementazione
C6	Contesto ricavabile dalla documentazione

Tabella 2: Criteri di selezione per category partitioning



```
560.    public void asyncReadEntries(long firstEntry, long lastEntry, ReadCallback cb, Object ctx) {
561.        // Little sanity check
562.        if (firstEntry < 0 || firstEntry > lastEntry) {
563.            LOG.error("IncorrectParameterException on ledgerId:{} firstEntry:{} lastEntry:{}",
564.                    new Object[] { ledgerId, firstEntry, lastEntry });
565.            cb.readComplete(BKException.Code.IncorrectParameterException, this, null, ctx);
566.            return;
567.        }
568.
569.        if (lastEntry > lastAddConfirmed) {
570.            LOG.error("ReadException on ledgerId:{} firstEntry:{} lastEntry:{}",
571.                    new Object[] { ledgerId, firstEntry, lastEntry });
572.            cb.readComplete(BKException.Code.ReadException, this, null, ctx);
573.            return;
574.        }
575.
576.        asyncReadEntriesInternal(firstEntry, lastEntry, cb, ctx);
577.    }
```

Figura 1: JaCoCo report per asyncReadEntries (BookKeeper)



```
398.    public synchronized long write(ByteBuffer[] buffs, long position) throws IOException {
399.        checkOpen(true);
400.        long total = 0;
401.        try {
402.            fc.position(position + START_OF_DATA);
403.            while (buffs[buffs.length - 1].remaining() > 0) {
404.                long rc = fc.write(buffs);
405.                if (rc <= 0) {
406.                    throw new IOException("Short write");
407.                }
408.                total += rc;
409.            }
410.        } finally {
411.            fc.force(true);
412.            long newsize = position + START_OF_DATA + total;
413.            if (newsize > size) {
414.                size = newsize;
415.            }
416.        }
417.        sizeSinceLastwrite = fc.size();
418.        return total;
419.    }
```

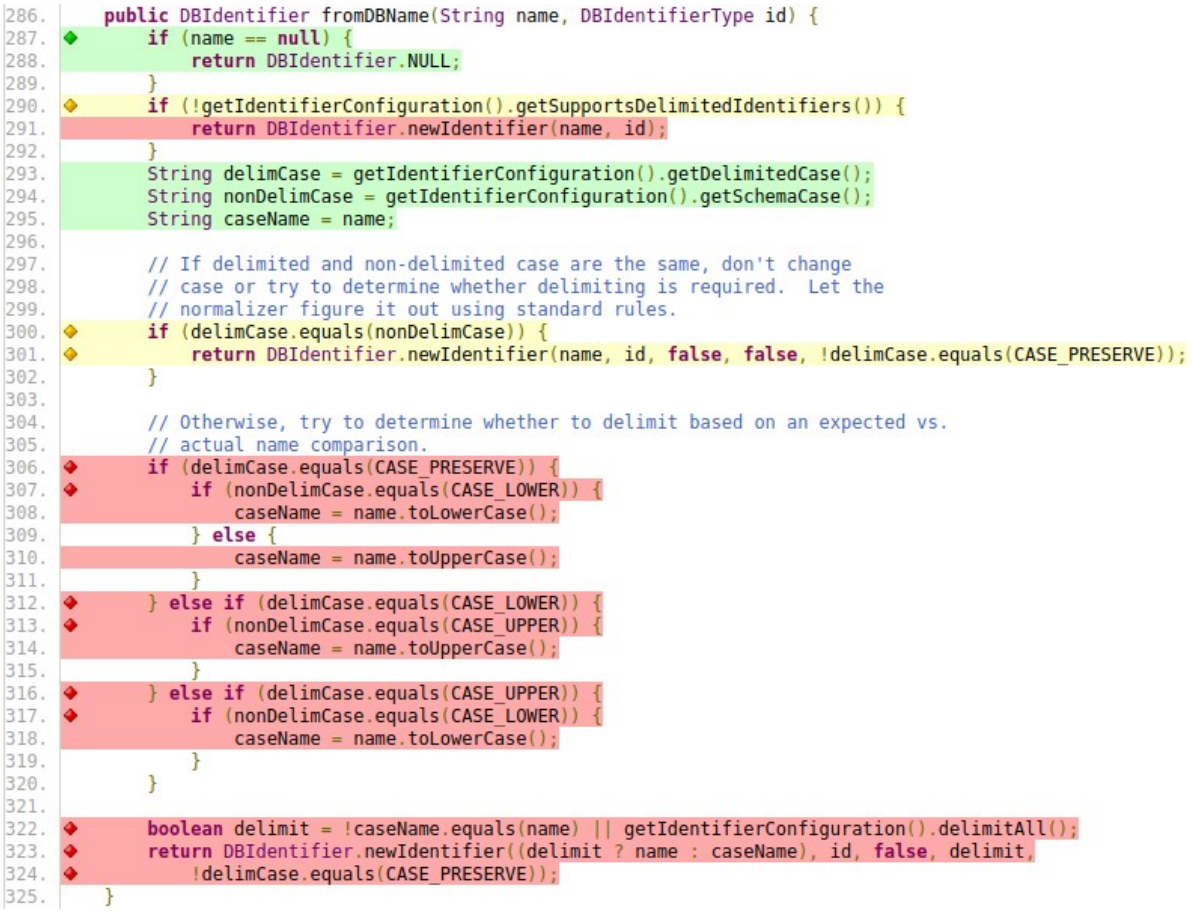
Figura 2: JaCoCo report per write (BookKeeper)

```
343.    private int readAbsolute(ByteBuffer bb, long start, boolean bestEffort)
344.            throws IOException {
345.        checkOpen(false);
346.        synchronized (this) {
347.            if (fc == null) {
348.                return 0;
349.            }
350.        }
351.        int total = 0;
352.        int rc = 0;
353.        while (bb.remaining() > 0) {
354.            synchronized (this) {
355.                rc = fc.read(bb, start);
356.            }
357.            if (rc <= 0) {
358.                if (bestEffort) {
359.                    return total;
360.                } else {
361.                    throw new ShortReadException("Short read at " + getLf().getPath() + "@" + start);
362.                }
363.            }
364.            total += rc;
365.            // should move read position
366.            start += rc;
367.        }
368.        return total;
369.    }
```

Figura 3: JaCoCo report per readAbsolute (BookKeeper)

```
146.    @Override
147.    public DBIdentifier makeIdentifierValid(DBIdentifier sname, NameSet set, int maxLen,
148.        boolean checkForUniqueness) {
149.        DBIdentifier validName = sname;
150.        String rule = sname.getType().name();
151.
152.        maxLen = getMaxLen(rule, validName, maxLen);
153.
154.        int nameLen = validName.getName().length();
155.        if (nameLen > maxLen) {
156.            validName = DBIdentifier.truncate(validName, nameLen - maxLen);
157.            nameLen = validName.getName().length();
158.        }
159.        if (isReservedWord(rule, validName.getName())) {
160.            if (nameLen == maxLen)
161.                validName = DBIdentifier.truncate(validName, 1);
162.            validName = DBIdentifier.append(validName, "0");
163.            nameLen = validName.getName().length();
164.        }
165.
166.        // now make sure the name is unique
167.        if (set != null && checkForUniqueness) {
168.            for (int version = 1, chars = 1; true; version++) {
169.                // for table names, we check for the table itself in case the
170.                // name set is lazy about schema reflection
171.                if (validName.getType() == DBIdentifierType.TABLE) {
172.                    if (!((SchemaGroup) set).isKnownTable(QualifiedDBIdentifier.getPath(validName)))
173.                        break;
174.                } else if (validName.getType() == DBIdentifierType.SEQUENCE) {
175.                    if (!((SchemaGroup) set).isKnownSequence(QualifiedDBIdentifier.getPath(validName)))
176.                        break;
177.                } else {
178.                    if (!set.isNameTaken(validName))
179.                        break;
180.                }
181.
182.                // a single char for the version is probably enough, but might
183.                // as well be general about it...
184.                if (version > 1) {
185.                    validName = DBIdentifier.truncate(validName, chars);
186.                }
187.                if (version >= Math.pow(10, chars))
188.                    chars++;
189.                if (nameLen + chars > maxLen) {
190.                    validName = DBIdentifier.truncate(validName, nameLen + chars - maxLen);
191.                }
192.                validName = DBIdentifier.append(validName, Integer.toString(version));
193.                nameLen = validName.getName().length();
194.            }
195.        }
```

Figura 4: JaCoCo report per makeIdentifierValid (OpenJPA)



```
286.    public DBIdentifier fromDBName(String name, DBIdentifierType id) {
287.        if (name == null) {
288.            return DBIdentifier.NULL;
289.        }
290.        if (!getIdentifierConfiguration().getSupportsDelimitedIdentifiers()) {
291.            return DBIdentifier.newIdentifier(name, id);
292.        }
293.        String delimCase = getIdentifierConfiguration().getDelimitedCase();
294.        String nonDelimCase = getIdentifierConfiguration().getSchemaCase();
295.        String caseName = name;
296.
297.        // If delimited and non-delimited case are the same, don't change
298.        // case or try to determine whether delimiting is required.  Let the
299.        // normalizer figure it out using standard rules.
300.        if (delimCase.equals(nonDelimCase)) {
301.            return DBIdentifier.newIdentifier(name, id, false, false, !delimCase.equals(CASE_PRESERVE));
302.        }
303.
304.        // Otherwise, try to determine whether to delimit based on an expected vs.
305.        // actual name comparison.
306.        if (delimCase.equals(CASE_PRESERVE)) {
307.            if (nonDelimCase.equals(CASE_LOWER)) {
308.                caseName = name.toLowerCase();
309.            } else {
310.                caseName = name.toUpperCase();
311.            }
312.        } else if (delimCase.equals(CASE_LOWER)) {
313.            if (nonDelimCase.equals(CASE_UPPER)) {
314.                caseName = name.toUpperCase();
315.            }
316.        } else if (delimCase.equals(CASE_UPPER)) {
317.            if (nonDelimCase.equals(CASE_LOWER)) {
318.                caseName = name.toLowerCase();
319.            }
320.        }
321.
322.        boolean delimit = !caseName.equals(name) || getIdentifierConfiguration().delimitAll();
323.        return DBIdentifier.newIdentifier((delimit ? name : caseName), id, false, delimit,
324.            !delimCase.equals(CASE_PRESERVE));
325.    }
```

Figura 5: JaCoCo report per fromDBName (OpenJPA)

Riferimenti bibliografici

[BK]: Apache BookKeeper, https://bookkeeper.apache.org/
[OJ]: Apache OpenJPA, https://openjpa.apache.org/
[del3]: Deliverable 3, (non ancora pubblicata)
[1+artifacts]: Rapporto JaCoCo per il progetto 1+, https://app.circleci.com/pipelines/github/massimostanzione/isw2-proj1plus/13/workflows/2da1726e-3cb4-4112-9d0e-85d02d6612e6/jobs/13/artifacts
[flakyBranch]: Branch "anti-flakiness" del progetto 1+, https://github.com/massimostanzione/isw2-proj1plus/tree/anti-flakiness
[flakyIssue]: Issue riguardante il test "flaky" nel progetto 1+, https://github.com/massimostanzione/isw2-proj1plus/issues/4
[PitTimeOut]: Pagina FAQ di Pitest, https://pitest.org/faq/
[BK1]: Documentazione di Apache BookKeeper, https://bookkeeper.apache.org/docs/4.14.0/getting-started/concepts/#basic-terms
[BK2]: Documentazione tecnica progetto Apache BookKeeper: Entries, https://bookkeeper.apache.org/docs/4.14.0/getting-started/concepts/#entries
[BK3]: Documentazione tecnica progetto Apache BookKeeper: Ledgers, https://bookkeeper.apache.org/docs/4.14.0/getting-started/concepts/#ledgers
[del2]: Deliverable 2, https://github.com/massimostanzione/isw2-deliverable2
[lab4]: Attività svolta durante il corso, https://github.com/massimostanzione/isw2-lab-activities/tree/master/src/it/uniroma2/isw2/massimostanzione/lab4
[OJArtifacts]: Test mutazionale OpenJPA, https://app.circleci.com/pipelines/github/massimostanzione/openjpa/10/workflows/6e310a2e-a494-452f-9413-3ca3678cc8c2/jobs/10/artifacts
[mivProblem]: Descrizione problematica riscontrata in makeIdentifierValid, https://www.mail-archive.com/users@openjpa.apache.org/msg09963.html
[proj1plus]: proj1plus,