1 JCS

A livello di progetto, sono stati riscontrati problemi nella build, in particolare maven non è stato capace di fare il retrieve di alcune classi javax, la cui licenza non è compatibile con i progetti sull'hub di maven. Quindi la scelta è stato di scrivere il testing su un nuovo progetto maven, importando tramite lo stesso il jar della versione di jcs richiesta. ciò a consentito di scrivere il testing per il sistema, tuttavia ha dato non pochi problemi nell'integrazione dei test con il sorgente dell'applicazione.

1.1 EventQueueConcurrentLoadTest

La classe di Test EventQueueConcurrentLoadTest esegue delle operazioni di inserimento e rimozione di elementi, ed alla fine verifica la validità della sequenza di operazioni contando il numero di inserimenti registrati da una classe di cache interna. Il test è stato reso parametrico nel seguente modo:

1. Il numero di inserimenti e il numero di inserimenti registrati attesi sono i due parametri che il testing si aspetta.
2. Il numero di inserimenti viene contato facendo uno stub della classe di cache tramite mockito e contando il numero di volte che i suoi metodi di inserimento vengono chiamati, nell'originale veniva usato uno stub fatto a mano.
3. gli inserimenti poi vengono rimossi.
4. è stata implementata una versione che, seguendo il testing originale della classe, aggiunge un delay tra gli inserimenti.

Con questo test si è voluto:

1. mantenere le operazioni del test originale.
2. mantenere i parametri di input dei test originali.
3. utilizzare Mockito al posto degli stub fatti a mano, e utilizzarlo per contare il numero di volte che determinati metodi vengano chiamati.

1.2 LRUMemoryCacheConcurrentTest

La classe di test LRUMemoryCacheConcurrentTest esegue una serie di inserimenti e rimozioni su una cache LRU, ed alla fine verifica che la proprietà LRU (Least recently used) sia stata rispettata. Il test preso di riferimento il test originale esegue le seguenti operazioni:

1. Legge un file di configurazione per in-stanziare la classe, questa parte è stata mantenuta perché altrimenti il comportamento del SUT può essere influenzato da una cattiva inizializzazione dello storage della cache.
2. inserisce un numero di elementi input nella cache.
3. verifica che la cache rispetti la proprietà LRU scartando gli elementi più vecchi quando è piena.

4. vengono eseguite delle operazioni di rimozione, riportando la cache allo stato iniziale.

Nel testo originale i parametri del testing erano hard-coded all'interno del codice e dovevano essere compatibili con i valori nel file, questo è stato cambiato in modo che i parametri vengono ricavati dinamicamente, in modo da evitare errori. Inoltre sono stati provati vari numeri di inserimenti, scegliendo un numero di inserimenti minori, uguali, superiori alla capacità della cache.

2 Bookkeeper

Per il testing di Bookeeper sono state scelte le classi:

- `org.apache.bookkeeper.bookie.storage.ldb.ReadCache`
- `org.apache.bookkeeper.bookie.storage.ldb.WriteCache`

Il motivo per questa scelta sono:

1. La similitudine di dominio con le classi affrontate su JCS. (Cache in cui devi inserire e rimuovere oggetti).
2. Parte del dominio è condiviso, rendendo più facile sfruttare la conoscenza acquisita su una classe sull'altra.

2.1 ReadCache

2.1.1 Descrizione della Classe

La classe `ReadCache` fornisce un servizio di caching di basso livello, la sua proprietà caratterizzante è il modo in cui viene gestita la memoria, infatti la memoria gestita da questa classe viene allocata alla sua istanziazione è rimane costante per tutto il ciclo di vita della istanza.
Dalla documentazione leggiamo che la memoria data viene divisa in segmenti, i quali vengono acceduti seguendo una logica ring-buffer, in caso la memoria sia piena, la classe rilascia il segmento più anziano, cosi può fare spazio per nuove entries.
I metodi di questa classe sono:

- `void put(long ledgerId, long entryId, ByteBuf entry)`
- `ByteBuf get(long ledgerId, long entryId)`
- `long size()`
- `long count()`
- `void close()`

2.1.2 ReadCachePutTest

Questa classe implementa un test parametrico per testare un singolo inserimento nella ReadCache. Il test viene implementato nel seguente modo; i parametri del test sono i due id della entry, un `ByteBuf` buffer seguito da flag booleano che indica se la tripla precedente sia valida o meno; il test inserisce con i parametri dati il buffer nella cache, poi controlla in ordine:

1. La dimensione della cache data da `size()` sia aumentata di 1.
2. Il count degli oggetti dato da `count()` all'interno della cache sia aumentata di 1.
3. Che una `get()` trovi effettivamente l'elemento appena inserito.

La prima scelta implementativa è stata la scelta dei parametri; I parametri sono stati scelti prima considerando la validità dei parametri prima individualmente, infatti dalla documentazione evince che `ledgerId` è u `long` che deve essere positivo, mentre per il `ByteBuffer` mi aspetto un buffer di dimensione minore della dimensione massima di segmento.
I parametri scelti quindi sono stati, un cosidetto "happy path", in cui tutti e tre i valori sono validi, una tripletta in cui il `ledgerId` è negativo, una in cui il `entryId` è negativo, una in cui il buffer è `null`. Quest Testing suite è stata in seguito analizzata con Jacoco per verificarne la validità della suite di test implementata; Il risultato è che Jacoco é il seguente; circa meta della funzione non viene raggiunto da questa test suite, poiché si occupa di gestire l'inserimento nel caso in cui la cache sia piena, da Jacoco è stata calcolata una statement coverage di circa 0.5 e un branch converage di 0.25 (un solo branch su 4 viene raggiunto). Allo scopo di migliorare le metriche di coverage calcolate in precedenza sono state aggiunti due nuovi casi test, il caso in cui un `ByteBuffer` viene create con dimensione 0, (quindi è vuoto) e il caso in cui invece la dimensione del buffer è maggiore di quella massima supportata dalla cache, quindi dimensione di un segmento più uno.
La nuova test Suite riporta su jacoco uno statement coverage di 0.76 e un branch coverage dello 0.75 (3 su 4 branch coperti), tuttavia, per la statement coverage gli statement non coperti riguardano l'inserimento di entries in caso di cache piena, quindi non raggiungibili da questo test che al più inserisce un singolo elemento all'interno di una cache vuota, portando lo statement coverage a 1, stesso discorso si applica per il branch mancato, non raggiungibile con un singolo inserimento, per cui le due metriche prese come riferimento sono massime per questo test.

2.1.3 ReadCacheMultiplePutTest

Questa Classe implementa il testing di due inserimenti consecutivi, multipli inserimenti permettono di provare l'inserimento in caso la cache sia piena.
I parametri scelti sono una naturale estensione dei precedenti, adattandoli a due inserimenti invece che uno solo; In questo caso la cache è grande abbastanza per tenere al suo interno poco meno di due segmenti.
I casi di test scelti sono stati quindi; quattro test in cui vengono inverti l'ordine degli id di due entries valide (che entrano nella cache). Infine i due ultimi test provano il caso in cui, la seconda entry non entra nella cache, quindi si aspetta un

fallimento dell'inserimento. La differenza tra due test è la grandezza del primo inserimento, primo grande quanto un segmento, mentre il secondo grande meta di un segmento, la differenza sta nel numero di byte inseriti rispetto al parametro maxCacheSize, nel primo il numero di byte inseriti è uguale al parametro, mentre nel secondo sono minore. Studiano l'efficacia della test suite con Jacoco mostra, che ho uno statement coverage pari a 1 per il metodo put, ed un branch coverage di 0.75.

2.1.4 ReadCacheSizeAndCountTest

Questo Test verifica che i metodi `count()` e `size()` funzionano correttamente. Il test è stato implementato nel seguente modo, vengono eseguiti un numero arbitrario di inserimenti all'interno della cache, è si verifica che i valori riportati dai due metodi corrispondano a quelli attesi.
La parametrizzazione del test è stata gestita nel seguente modo, gli input del test sono i parametri di inizializzazione della cache, il numero di inserimenti, la dimensione degli elementi inseriti ed infine i valori attesi.
I primi test eseguiti sono stati:

1. Tanti inserimenti fino ad arrivare alla dimensione massima della cache.
2. gli stessi inserimenti precedenti, ma con le entries di dimensione dimezzata, la size deve essere legata ai segmenti richiesti è non solo alla dimensione degli elementi all'interno.
3. Eseguire più inserimenti di quanto la dimensione cache massima può supportare.

Usando Jacoco sulla seguente testing suite, otteniamo uno statement coverage di 1 ed un branch coverage allo 0.83, Jacoco fa notare che ci perdiamo un branch, ovvero il caso in cui inserisco un buffer vuoto (con dimensione 0); quindi è stato aggiunto un quarto caso di test in cui vengono inseriti degli elementi con dimensione 0; con il quarto test arrivo ad una branch coverage di 1.

2.1.5 ReadCacheGetTest e ReadCacheEmptyGetTest

Queste classi hanno lo scopo di testare il metodo get, la prima inserisce un elemento nella cache per poi riprenderlo con la get, verificando che la get ritorno l'elemento inserito. La seconda invece prova una get di un elemento non presente nella cache. Insieme queste due classi hanno statement e branch coverage a 1 per il metodo get.

2.2 WriteCache

WriteCache è un altro tipo di Cache all'interno di bookkeeper, la differenza rispetto alla ReadCache descritta in precedenza è che la classe allocherà memoria diretta (un array), e poi la dividerà in segmenti.

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

- `long count()`
- `void close()`
- `boolean isEmpty()`

2.2.1 WriteCachePutTest

Questo Test esegue un inserimento è controlla che l'intero stato della cache sia correttamente aggiornato. L'input del test sono gli id (ledger e entry id), la entry da inserire (definita dalla sua dimensione), ed il risultato atteso dal test, due booleani per indicare il successo della `put()` ed se l'input aspetta un eccezione. La prima decisione è stato l'insieme di testing, il primo caso presenta l'happy path, il secondo prende un ledger id negativo (che da documentazione è invalido), il terzo prende un entry id negativo (che invece da documentazione è valido); continuando sugli id sono stati scelti i valori massimi e minimi per i `long`, il motivo è il testing di incrementi o decrementi, che agiscono differentemente con i valori massimi è minimi, infine sono stati testati i casi limite per il buffer, un buffer null, un buffer vuoto, un buffer più grande della dimensione massima di un segmento. Da Jacoco abbiamo uno statement coverage allo 0.97, ed un branch coverage allo 0.75, che sono stati ritenuti accettabili per la test suite, questo perché i branch mancanti sono casi estremamente improbabili, ed gli statement che coprono solo il restante 0.03.

2.2.2 WriteCacheMultiplePutTest

Questo test esegue due inserimenti e verifica che lo stato della cache sia stato correttamente aggiornato. L'input del test è un estensione del test precedente, usando i parametri per due inserimenti. I parametri scelti per il test sono un happy path, il caso in cui i due inserimenti avvengono con gli stessi id, un caso in cui solo gli entryID siano uguali, un caso in cui i ledgerID siano uguali, ma gli entryID sono in ordine decrescente. Infine gli ultimi casi sono sui valori di entry, testando i casi in cui le due entry siano di dimensione inferiore alla dimensione della cache e di dimensione superiore alla dimensione della cache. Per valutare la bontà della suite test usiamo Jacoco per calcolarci le stamente coverage e la branch coverage che sono rispettivamente 0.97 e 0.75, tuttavia come sopra i casi mancati sono casi eccezionali, non facilmente riproducibili.

2.2.3 WriteCacheGetTest

Questo test esegue un inserimento e poi fa il retrieve della entry, verificando che lo stato dell'intera cache venga aggiornato correttamente. Gli input del test sono i valori per fare un inserimento, e tre flag booleani, se eseguire la put (per provare il caso in cui la cache sia vuota), se il retrieve ha successo, se viene attesa un eccezione. I casi di test scelti sono stati, il solito happy path, un inserimento e retrive di un entry null, provare a fare il retrieve di un elemento non inserito, l'inserimento e il retrieve di un buffer vuoto ma non nullo. lo statement coverage e il branch coverage sono entrambi 1, quindi la test suite è stata considerata adeguata.

2.2.4 WriteCacheGetLastEntryTest

Questo Test esegue vari inserimenti per poi verificare che il metodo getLastEntryTest recuperi effettivamente l'ultimo elemento inserito. L'input del test è stato scelto come il numero di inserimenti eseguito prima del retrieve, in particolari i casi scelti sono stati, nessun inserimento, un inserimento (meno del massimo), due inserimenti (contengono il massimo numero di elementi che la cache possiede), tre inserimenti (più del massimo, uno degli elementi viene scartato). Jacoco da un statement coverage e un branch coverage entrambi ad 1, quindi la test suite è stata considerata adeguata.

2.2.5 Mutation Testing e considerazioni finali

Considerando l'intero set di test come un unica test suite otteniamo per la classe ReadCache uno statement coverage di 0.98 e un branch coverage allo 0.94, mentre per la WriteCache da jacoco otteniamo uno statement coverage di 0.61 e un branch coverage al 0.50, tuttavia questi valori non contano che la test suite non comprende il metodo `forEach()` della classe, levando la classe dall'analisi otteniamo uno statement coverage globale di 0.95 ed un branch coverage di 0.86. PIT permette di eseguire mutation testing sulla nostra testing suite, i mutation test sono stati eseguiti in paro a jacoco per avere un secondo feedback sull'efficacia dei test. Pit da un Test Strenght di 0.8 per l'intera testing suite su `ReadCache` e 0.74 per `WriteCache`, alcune delle mutazioni sono assai difficili da rilevare oppure appaiono in metodi secondari non testati direttamente, quindi i test presenti sono stati considerati accettabili.

3 OpenJPA

Per il testing di OpenJPA sono state scelte le seguenti classi:

1. `org.apache.openjpa.lib.util.StringDistance`
2. `org.apache.openjpa.util.ProxyManagerImpl`

3.1 StringDistance

La classe StringDistance implementa il calcolo della distanza di Levenshtein tra stringhe, la classe è stata scelta perché ha delle caratteristiche interessanti, prima di tutto è una classe senza side-effect; quindi la scelta dell'input determina univocamente l'output dei metodi della classe. Non tutti i metodi sono stati testati, questo perché i metodi non testati sono wrapper sui metodi testati, la cui operazioni è una semplice operazione di conversione di parametri, spesso da array java a Collection; è stato pensato che duplicare l'intera test suite per testare una conversioni di parametri non sarebbe stata molto produttiva al fine di migliorare l'efficienza dei test della classe.

3.1.1 StringDistanceGetLevenshteinDistanceTest

Questo metodo statico prende due stringhe e calcola la loro distanza di Levenshtein che serve a determinare quanto due stringhe siano simili. I parametri scelti per il test sono le due stringhe di input per il metodo è il risultato atteso

dal test. I test prevedono un happy test, i vari test possibili con la stringa vuota, due stringhe uguali. Il valore null non è stato testato perché considerato invalido dalla documentazione del metodo, se al metodo viene passato un valore null il metodo termina con un `NullPointerException`, quindi non è stato trovato del valore aggiunto nel considerarlo come un valore ammissibile nei test. Provando con jacoco la testing suite ottiene un statement coverage di 1 ed un branch coverage di 1.

3.1.2 StringDistanceGetClosestLevenshteinDistanceTest

Questo metodo statico, data una stringa, una collection di stringhe è un intero, ritorna la string all'interno dell'array la quale distanza sia la minore e che non sia minore del threshold intero specificato. I parametri per il test sono, i parametri di input del metodo unito alla stringa risultato aspettata. La stringa deve essere presente all interno della collection. I parametri di test scelti partizionano l'insieme degli input per parametro, dividendolo in stringa vuota, stringa non vuota all'interno della collection, stringa non vuota all'interno della collection, una collection null, vuota, con degli elementi al suo interno, un threshold negativo, nullo, positivo, il valore massimo degli interi (come caso limite). Visto con Jacoco questo test arriva ad una statement coverage e branch coverage di 1.

3.1.3 StringDistanceGetClosestLevenshteinDistanceFloatTest

Questo metodo statico, data una stringa, una collection di stringhe è un float, ritorna la string all'interno dell'array la quale distanza sia la minore e che non sia minore del threshold float specificato. I parametri per il test sono, i parametri di input del metodo unito alla stringa risultato aspettata. La partizione dei parametri di testo è stata fatta considerando i parametri partizionati nel seguente modo, La stringa di input che è stata partizionata come null, stringa vuota, stringa nella collection, stringa fuori la collection. La collection come null, vuota, non vuota. Il threshold come 0.0, il massimo valore float, infinito, meno infinito, NaN. Provando con jacoco la testing suite ottiene un statement coverage di 1 ed un branch coverage di 1, motivo per cui è stato considerato adeguato come test suite.

3.1.4 Mutation Test

Con Pit eseguendo un Mutation Test sull'interezza dei test descritti in precedenza otteniamo un mutation coverage del 77% e un test strenght del 94%; tenendo in conto che molte delle mutazioni mancate avvengono in metodi non raggiungibili, la mutation coverage finale sul codice testato risulta più alta. I mutation test comunque rassicurano che l'adeguatezza della test suite è più che sufficiente.

3.2 ProxyManagerImpl

Questa classe è un implementazione dell'interfaccia ProxyManager, che dalla documentazione legge "Manager for copying and proxying second class objects". Lo scopo di questa classe è ritornare una copia esatta del suo input, con i vari metodi che si occupano di tipi di dato particolari come array, calendar, etc... Un altra delle features I metodi testati sono quelli dell'interfaccia ProxyManager, i

metodi che non sono parte di questa interfaccia non sono stati considerati per il testing.

3.2.1 ProxyManagerImplCopyArrayTest

Questo metodo prende un array come input 'e ritorna una copia. Gli input del programma sono l'array da copiare, è un flag per indicare che il test si aspetta un errore. I parametri provati sono stati tre array validi, di tipo Integer, Double e String, tre array vuoti dei tipi sopra elencati, un generico Object() che si aspetta di fallire, ed un parametro null. Jacoco ritorna uno statement e un branch coverage di 1.

3.2.2 ProxyManagerImplCopyCalendarTest

Questo metodo si occupa di copiare un classe che inerita l'interfaccia Calendar, gli input del test sono dati dal singolo elemento da copiare, il test deve provare che il risultato della copia sia identico all'input dato. I parametri scelti sono un calendario generico con qualche elemento al suo interno, un calendario proxy, generato dal ProxyManager, vuoto ed un calendario proxy con qualche elemento al suo interno. Nella sua semplicità il test ritorna uno statement coverage ed un branch coverage di 1, il che conclude i casi di test per questo metodo.

3.2.3 ProxyManagerImplCopyCollectionTest

Questo metodo si occupa di copiare una collection generica e tutti gli elementi al suo interno. Il test è stato parametrizzato solo con la collection da copiare, visto che se il test ha successo l'output del metodo deve essere identico all'input. I parametri scelti sono stati una `LinkedList` e un `ArrayList` con vari elementi al loro interno, una collection proxy vuota creata dall'interfaccia ProxyManager, ed infine una collection proxy non vuota. Il test ritorna uno statement coverage ed un branch coverage di 1, il che conclude i casi di test per questo metodo.

3.2.4 ProxyManagerImplCopyCustomTest

Dalla documentazione di openjpa abbiamo questo metodo descritto nel seguente modo " Return a copy of the given object with the same information, or null if this manager cannot copy the object.", rendendo questo metodo un metodo più generale per copiare un qualsiasi tipo di oggetto generico. I parametri di input del test sono l'oggetto da copiare è un flag per indicare se l'oggetto è proxyable, perché se non lo è il metodo ritorna null. I parametri sono stati scelti nel seguente modo, prima tutti i tipi proxy creati da ProxyManager, seguiti da implementazioni reali dei tipi proxy, quindi Collection, Map, Date, Calendar, un oggetto custom, definito da una classe CustomBean, una classe nonProxiable, creata estendendo la classe CustomBean con un costruttore non-copy; queste due classi sono state prese dai test ufficiali di questa classe, infine una referenza null. Con Jacoco si ha uno statement coverage di 0.96 e un branch coverage di 0.93, nonostante siano dei buoni valori per una test suite, in questo caso il problema risulta che non abbiamo coperto un caso di input possibile con la nostra analisi, un particolare insieme di classi che questo metodo non è capace di copiare chiamati `Manageable`; Una classe Manageable è una classe con delle particolari proprietà che la rendono impossibile da copiare per la logica di questa

classe, per imitarne il comportamento è stata creata una classe stub con Mockito dell'interfaccia `PersistenceCapable`, aggiunto quest'ultimo caso di test, Jacoco riporta uno statement coverage e un branch coverage di 1, dove finiamo la nostra analisi.

3.2.5 ProxyManagerImplCopyDateTest

Questo metodo si occupa di copiare una classe Date generica. Il test è stato parametrizzato solo con la classe Date da copiare, visto che se il test ha successo l'output del metodo deve essere identico all'input. I parametri scelti sono stati una classe `Date` e una classe `TimeStamp`, una Date proxy non inizializzata creata dall'interfaccia ProxyManager, ed infine una classe Date proxy inizializzata. Il test ritorna uno statement coverage ed un branch coverage di 1, il che conclude i casi di test per questo metodo.

3.2.6 ProxyManagerImplCopyMapTest

Questo metodo si occupa di copiare una Map generica e tutti gli elementi al suo interno. Il test è stato parametrizzato solo con la Map da copiare, visto che se il test ha successo l'output del metodo deve essere identico all'input. I parametri scelti sono stati una `HashMap`, un `LinkedMap` e una `TreeMap` con vari elementi al loro interno, una `HashMap` vuota, una Map proxy vuota creata dall'interfaccia ProxyManager, ed infine una Map proxy non vuota. Il test ritorna uno statement coverage ed un branch coverage di 1, il che conclude i casi di test per questo metodo.

3.2.7 Mutation Testing

Utilizzando Pit eseguiamo i Mutation Test per verificare l'adeguatezza dei test implementati, in questo caso ci sono state delle limitazioni dovute al fatto che sono stati testati solo i test dell'interfaccia ProxyManager, mentre la classe ProxyManagerImpl possiede molti metodi privati o protected non rilevanti alla nostra test suite ma comunque non escludibili da PIT. Con questa premessa PIT ritorna i seguenti valori: un mutation coverage allo 0.39 ed un test strenght all 0.64, tuttavia andando manualmente a verificare i metodi testati è stato possibile verificare che quasi tutti i mutanti generati che li coinvolgono vengono effettivamente uccisi, non è stato calcolato un mutation coverage corretto esplicitamente, tuttavia il risultato di questo mutation coverage risulta comunque soddisfacente.