Report sul progetto di Software Testing

Filippo Muscherà

0338276

Contents

1 BookKeeper

1.1 Continuous Integration

Per il progetto Bookkeeper, dopo aver forkato la repository di Apache su GitHub, per fare in modo che quanto realizzato sulla mia macchina in locale fosse effettivamente eseguibile anche in ambiente configurato potenzialmente in maniera diversa dal mio sono stati realizzati una serie di worflows tramite l'utilizzo di GitHub Action. Per Bookkeeper in particolare ho realizzato tre diversi workflow:

- Uno che eseguisse il comando "mvn clean verify", per far si che i test di unità e di integrazione fossero eseguiti ogni volta che si eseguiva un push, per assicurarsi che il loro superamento non fosse legato alla specifica configurazione della mia macchina in locale.
- Uno che generasse il report di Ba-dua per il modulo bookkeeper-server (quello per cui sono stati realizzati i test), in modo che il report di ba-dua non fosse disponibile solamente in locale, ma che potesse essere consultato anche da GitHub, dato che alla fine del workflow l'.xml di ba-dua viene caricato come artifact, ed è disponibile al download.
- Un terzo workflow che si occupasse della coverage raggiunta dai casi di test. Questo workflow infatti esegue la build del progetto facendo girare i test di unità e di integrazione, e ne calcola la coverage tramite l'utilizzo del profilo che esegue anche il plugin di JaCoCo. Il report .html di JaCoCo viene poi caricato su GitHub come artifact, mentre l'.xml viene letto da SonarCloud, che lo sfrutta per calcolare la coverage raggiunta dai test realizzati. Il report di SonarCloud è visibile alla seguente pagina

1.2 SUT

Per il progetto Apache Bookkeeper, sono state individuate come classi da testare BookieImpl e Journal, all'interno del package "bookkeeper-server". La scelta è motivata dal fatto che, entrambe queste classi, rappresentano entità fondamentali al funzionamento di BookKeeper. BookieImpl infatti rappresenta il singolo nodo del sistema di database distribuito di Bookkeeper, mentre il Journal rappresenta in un certo senso il file di log del singolo Bookie.

1.3 Unit Testing

Lo scopo degli Unit test realizzati è quello di acquisire confidenza sul funzionamento della classe sotto esame in isolamento dal resto del sistema.

1.3.1 BookieImpl

Per la classe BookieImpl, il primo approccio per acquisire confidenza con il suo comportamento è stato quello di iniziare a testare i suoi principali metodi statici: *getBookieAddress, mountLedgerStorageOffline e format*.

getBookieAddress

{classi BookieImplAddressTest, BookieImplAddressMockTest, BookieImplAddressBaduaTest}

Questo metodo ha come unico input un oggetto di tipo ServerConfiguration, da cui estrae alcuni dei suoi attributi durante la sua esecuzione. Dunque, sono stati considerati come parametri di input al metodo tutti quegli attributi estratti dall'oggetto ServerConfiguration che il metodo riceve in input. I parametri quindi sono:

- advertisingAddress (String)
- bookiePort (int)
- listeningInterface (String)
- hostNameAsBookie (boolean)
- shortHostName (boolean)
- allowLoopback (boolean)

Per questi attributi si è proceduto ad eseguire il partizionamento in classi di equivalenza dei rispettivi valori. Sulla documentazione di Bookkeeper il significato dei vari parametri.

- Il parametro **advertisingAddress** è una stringa che rappresenta l'indirizzo IP associato al Bookie. Per questo parametro, nel definire le sue classi di equivalenza, teniamo conto sia del fatto che è un dato di tipo String, sia che rappresenta un IP. Fatte queste considerazioni, si ricavano le seguenti classi di equivalenza:
  {NULL}, {Stringa vuota}, {Stringa che rappresenta un IP valido}, {Stringa che rappresenta un IP non valido $\neq$ stringa vuota}.

- Per il parametro **bookiePort**, teniamo in considerazione il tipo di dato (intero) e la sua funzionalità (porta associata al server Bookie). Dal tipo di dato ricaviamo le classi di equivalenza "classiche" per un intero (maggiore, minore e uguale a 0). Dalla sua funzionalità ricaviamo invece le classi di partizioni "valido" e "non valido". Essendo il dato un numero di porta, un valore "valido" è quello compreso tra [1, 65535], mentre uno "non valido" è, di conseguenza, un intero che cade al di fuori di questo intervallo. Unendo queste suddivisioni del dominio del dato si ottengono le classi di equivalenza finali:
{$bookiePort$ <0}, {$bookiePort = 0$}, {0 <$bookiePort \leq 65535$}, {$bookieport > 65535$}.
- Il parametro **listeningInterface** è una stringa che rapprsenta:

  "The network interface that the bookie should listen on. If not set, the bookie will listen on all interfaces."

  Possiamo dunque considerare sia la natura del tipo di dato (String), da cui possiamo ricavare le classi di equivalenza {NULL}, {Stringa vuota}, {Stringa non vuota}, sia il significato del parametro in questione, grazie a quanto scritto sulla documentazione.
  Sapendo che questo parametro rappresenta un'interfaccia di rete, possiamo aggiungere anche le classi di equivalenza: {Network Interface valida}, {Network Interface non valida}.
  Combinando le due partizioni, otteniamo dunque quella finale:
  {NULL}, {Stringa vuota}, {Network Interface non valida $\neq$ ""}, {Network Interface Valida}.
  Una Network Interface valida sarà una stringa a cui effettivamente corrisponde un'interfaccia di rete del dispositivo, e quindi di conseguenza una non valida potrà essere una qualsiasi stringa (non vuota) che non trova corrispondenze nell'elenco di interfacce di rete.

- I parametri **hostNameAsBookie**, **shortHostName**, **allowLoopback** sono parametri booleani, pertanto gli unici valori possibili per questi saranno *true* o *false*. Di questi parametri però è importante tenere in conto le relazioni che hanno con gli altri input del metodo che si sta testando: hostNameAsBookie e shortHostName, come intuibile dai nomi stessi dei parametri, stabiliscono se usare l'hostname come nome del bookie e in tal caso se usare il FQDN o una versione "abbreviata" dell'hostname. Quindi è evidente come, nel caso in cui hostNameAsBookie sia false, shortHostName sia di fatto un "don't care". Per allowLoopback invece è evidente come ci sia un legame con l'interfaccia di rete scelta. Chiaramente se l'interfaccia di rete ha il loopback, non potrò settare allowLoopback a false, dato che BookieImpl non ha le responsabilità necessarie per poter "sovrascrivere" il comportamento dell'interfaccia di rete su cui si appoggia. Dunque se si proverà a settare allowLoopback = false, e si sceglierà un'interfaccia di rete che ha il loopback, si otterrà un'eccezione.

Boundary Values Definiamo ora i boundary values per ogni parametro del metodo:

- **advertisingAddress**: NULL, "", "192.168.56.102", "30000.568.1.2"
- **bookiePort**: -1, 0, 1, 65536
- **listeningInterface**: NULL, "notAnInterface", *"enp0s9"*

Nota: L'interfaccia di rete valida chiaramente varia in base al dispositivo su cui viene eseguito il test. In questo report è riportata *"enp0s9"*, che è l'interfaccia di rete che veniva selezionata sulla mia macchina locale. Il codice del test però è stato realizzato in modo da ricavare ogni volta un'interfaccia di rete valida, in modo che il codice del test fosse testabile anche attraverso la CI sulle Virtual Machine delle GitHub Actions, e in generale su qualsiasi macchina che potesse compiere la build di Bookkeeper.

Casi di Test Per lo sviluppo dei casi di test, si farà in modo che si verifichi uno dei due scenari:

1. Il caso di test ha esito positivo, e quindi viene restituito un bookieSocketAddress coerente con i parametri con cui il metodo è stato chiamato (dopo si definirà nel dettaglio cos'è un'istanza "coerente");
2. Oppure, se il metodo verrà chiamato con una combinazione di parametri che porterà al verificarsi del *throws* di un'eccezione, dovrà esserci un solo parametro "scorretto", e pertanto la causa dell'eccezione potrà essere una sola e facilmente individuabile.

Per la definizione dei casi di test si è deciso, in questa occasione, di avere un approccio ibrido tra una modalità di test multi-dimensionale e uni-dimensionale. Infatti, avendo un legame tra il parametro allowLoopback e advertisingAddress, per questi attributi vogliamo avere un caso di test per ognuna delle combinazioni date dal prodotto cartesiano {"", "192.168.56.102"} $\times$ {true, false}. Da questa combinazione abbiamo escluso il parametro "30000.568.1.2", e il parametro NULL. Il primo perchè chiaramente non rappresenta un indirizzo IP, per cui ci aspettiamo che non verrà utilizzato come tale, mentre il valore NULL è utilizzato solo in modo uni-dimensionale dato che viene trattato al pari del valore "", e quindi porterebbe solamente alla definizione di casi di test ridondanti.
Anche per gli altri attributi si sceglie un approccio uni-dimensionale, dato che un approccio multi-dimensionale non porterebbe ad avere un maggior livello di confidenza sul funzionamento del metodo, dal momento che sono attributi senza legami tra di loro. Per "**legame**" tra due parametri si intende che, assegnandoli due valori corretti se presi "individualmente", una loro combinazione incompatibile può portare a un'eccezione/errore nel metodo.
Nella tabella 1 i casi di test definiti fin'ora.

Commenti

- Un numero di porta = 0 non genera errori perché richiede una porta effimera.
- L'indirizzo IP scorretto viene semplicemente interpretato come un host name per il bookie. Il vero IP quindi verrà "ricalcolato" di volta in volta (come spiegato qui).
- L'istanza ritornata dal metodo, per essere considerata valida non deve aver generato eccezioni e deve avere il numero di porta che corrisponde a quanto richiesto tramite il parametro bookiePort.

Miglioramenti del test Tramite l'utilizzo di PIT, ci si accorge che questo test non uccide tutte le mutazioni che dovrebbe. In particolare, le mutazioni che PIT generava cambiando le condizioni negli if delle righe 258 e 278 non venivano intercettate, perchè gli unici controlli che venivano eseguiti sull'istanza ritornata riguardavano il numero di porta. Quindi, per rendere più robusto il test si sono introdotti un'ulteriore serie di controlli. Si è quindi andati a controllare anche che l'hostName del BookieSocketAddress fosse coerente con quanto richiesto in fase di chiamata del metodo rispetto ai parametri passati. Per esempio, se useHostNameAsBookieId = true, si va a controllare se il bookieSocketAddress ritornato dal metodo utilizzi effettivamente l'hostName (ed eventualmente la sua short version, in accordo con il parametro booleano useShortName). Oppure, se come parametro era stato passato un hostName utilizzabile (es.: $\neq$ null), si controlla se questo fosse stato effettivamente assegnato al BookieSocketAddress ritornato dal metodo.
Per come sono realizzati fin qui i test, analizzando il report di ba-dua, ci si rende conto che è stata tralasciata la coverage di alcune coppie definition-use. In particolare, tutti gli use delle variabili usate nel throws dell'eccezione a riga 271/272 di BookieImpl (hostName e iface). Per estendere la coverage di questi due casi, si è relizzato il metodo unresolvedHostNameTest nella classe BookieImplAddressBaduaTest . Il test in questa classe è stato realizzato tramite l'utilizzo di un mock della classe DNS di Bookkeeper, per far ritornare un hostName che facesse risultare il corrispondente InetSocketAddress irrisolto, portando quindi al lancio dell'eccezione di riga 271/272. A questo punto, eseguendo ba-dua anche sul nuovo test rimangono scoperti solamente tre coppie DU (questi). Per quanto riguarda il primo e il terzo, probabilmente la loro mancata coverage è dovuta al fatto che i relativi branch sono raggiunti tramite test che usano il runner di PowerMock, che entra in conflitto con ba-dua e interferisce con il suo calcolo della coverage, dal momento che dovrebbero risultare coperti, dato che un'analisi manuale (e l'utilizzo del debugger) mostra come il test testLoopbackException della classe BookieImplAddressMockTest copra quei due definiton-use. A riprova di questo c'è il fatto che, se il test inserito nella classe BookieImplAddressBaduaTest viene invece inserito nella classe con il runner di PowerMock, anche i relativi DU non risultano coperti, mentre quando è inserito in una classe senza questo particolare runner, la relativa coverage di ba-dua è corretta. Infine, il secondo DU dei tre rimasti scoperti a cui si faceva riferimento sopra, risulta ancora scoperto. Per avere coverage anche su questo DU, è stato aggiunto il metodo failDefaultHostNameTest alla classe BookieImplAddressBaduaTest. Questo test ha un meccanismo del tutto simile all'altro aggiunto per ba-dua, ma lo fa coprendo il DU relativo alla definizione di *iface* a riga 265 per il relativo uso a riga 271. Qui si è utilizzato un mock per la classe DNS, poiché per ottenere questo comportamento senza il mock bisognerebbe avere un sistema su cui la richiesta dell'indirizzo IP del local host fallisca, e inoltre non dovrebbe esserci, a livello di sistema, un indirizzo IP associato all'host name "localhost" (come si capisce da qui . Sono stati utilizzato i commenti nel codice sorgente perchè il relativo javadoc non è reperibile online e non viene generato nemmeno durante la build offline).

Quality Assurance A questo punto il test può ritenersi completato. La coverage di badua è stata portata al 100% (considerando "covered" anche quei due DU che vanno in conflitto con PowerMock), mentre tutte le mutazioni di PIT vengono eliminate, eccetto una che va in timeout. Questo metodo ha rispettato il comportamento previsto per tutti gli input che gli sono stati forniti.

N.B.: JaCoCo non restituisce una coverage del 100% per questo metodo, dal momento che se si entra in alcuni branch, l'unica istruzione presente è quella che lancia un'eccezione, e come riportano le FAQ di JaCoCo (Source code lines with exceptions show no coverage. Why?) , questo può causare problemi con il calcolo della coverage.

format

{classi FormatTest, FormatMockTest}

Questo metodo ha il semplice compito di formattare il Bookie, svuotando le cartelle dei journal, dei ledger, degli index e dei metadata. Queste cartelle sono anche i parametri che prende in input, assieme a due booleani: isInteractive e force.

Classi di equivalenza Il partizionamento in classi di equivalenza per questi parametri è piuttosto immediato: journalDirs, ledgerDirs e indexDirs sono String[], che rappresentano i path dei vari file corrispondenti. Per tutti e tre quindi le partizioni del dominio sono: {NULL}, {Array non valido}, {Array con path validi ed esistenti}, {Array con path validi ma non esistenti}.
Per array non valido intendiamo un array che contenga uno o più path scorretti. Un array valido invece è un array che

contenga tutti path validi (quindi path che possono, all'occorrenza, essere associati a un file senza errori). Il parametro GcEntryLogMetadataCachePath è invece una String, che rappresenta il path a cui sono conservati i metadati del Bookie. In modo simile a prima, per questo parametro definiamo le partizioni {NULL}, {Path non valido}, {Path valido non esistente}, {Path valido ed esistente}.
Infine, isInteractive e force sono booleani, quindi il loro partizionamento sarà semplicemente {true}, {false}.

Boundary Values Per i tre parametri di tipo String[] i valori scelti sono:
{NULL}, {Array contenenente il path "notAPath\0"}, {Array contenente path a cartelle temporanee esistenti}, {Array contenente path validi e in più il path "not/a/real/path"}.
Non specifichiamo esattamente i nomi delle cartelle temporanee poiché queste verranno create a runtime dal test stesso. In ogni caso, ci si assicurerà che le cartelle siano state create correttamente prima di lanciare il test, in modo che quest'ultimo sia sempre attendibile. Inoltre consideriamo il path scelto come non valido tale, poiché contiene il carattere di terminazione di stringa "\0", che non è consentito in un path.
Analogamente, per GcEntryLogMetadataCachePath avremo:
{NULL}, {"notAPath\0"}, {Path valido di una cartella esistente}, {"not/a/real/path"}.
Dal momento che, come si evince dalla segnatura del metodo, il metodo format(...) non solleva eccezioni, l'expected value potrà essere solamente un booleano: true nel caso in cui la formattazione avvenga con successo, false in caso contrario.

Casi di Test I casi di test sono stati definiti in maniera uni-dimensionale perché non essendoci evidenti legami tra i vari parametri, si è ritenuto che un approccio multi-dimensionale non avrebbe portato ad avere una maggiore confidenza sul corretto funzionamento del metodo. Nella tabella 2 si vedono tutti i casi di test realizzati e i rispettivi valori di ritorno attesi. I test sono stati realizzati in modo che ci fosse un solo parametro per volta che potesse far fallire, e quindi far ritornare false, al metodo format. Il valore NULL per journalDirs è gestito dalla ServerConfiguration (vedi listing 3), mentre per gli altri parametri (ledger, index e metadata dirs) rappresenta semplicemente il fatto che quelle cartelle non ci sono o non vanno formattate. Nei casi in cui si ha invece un path che è valido ma la relativa cartella non esiste, non c'è ragione di aspettarsi un false come ritorno, perché quella cartella potrà essere ignorata o considerata come già formattata. Dunque ci attendiamo un valore di ritorno false quando:

- Uno dei parametri che rappresenta delle cartelle è INVALID,
- isInteractive = false e force = false, perché non si procede alla formattazione,
- isInteractive = true ma l'interazione con l'utente (o con il mock nel caso del test) ritorna false quando si chiede conferma per procedere con il format,
- L'interazione con lo standard input genera un'eccezione.

Ci aspettiamo un valore di ritorno true in tutti gli altri casi.

Commenti Quando il parametro isInteractive è settato a true, verrà richiesta conferma tramite la command line prima di procedere al format. Testare questo branch così com'è chiaramente non sarebbe stato possibile, perché avrebbe richiesto ogni volta l'intervento di un utente. Si è quindi mockato il metodo della classe IOUtils di Bookkeeper che gestiva questa interazione, simulando i possibili comportamenti dell'utente.

Quality Assurance Con i test realizzati fono a questo momento, JaCoCo segnala una copertura del 100% per l'instruction coverage, ma non per il branch coverage, fermo al 93%. Anche il report di ba-dua segnala che ci sono due coppie DU non coperte (sempre relative alle stesse righe di codice). Per questo motivo si è deciso di realizzare ulteriori test, in modo da raggiungere il valore massimo di copertura anche per i branch condizionali. In particolare, rimanevano scoperti alcuni dei possibili branch condizionali di riga 1180 (condizione booleana per l'operatore ternario) e 1181 (blocco if). Sono quindi stati introdotti due metodi di test nella nuova classe FormatExtendedTest: emptyDirTest e notADirTest. Come intuibile dal nome, per coprire i branch mancanti si va a testare il metodo format anche quando la cartella dei journal è una cartella valida, esistente ma vuota, e nel caso in cui invece il path passato al metodo è valido e corrisponde a un file, che seppur esistente non rappresenta una cartella (nel test si è usato un semplice file .txt vuoto). Con l'aggiunta di questa nuova classe di test, ora il metodo format della classe BookieImpl ha 100% di coverage sia con il criterio data-flow di ba-dua, sia per la instruction coverage e la branch coverage di JaCoCo.

Infine, PIT riporta che tutte le mutazioni generate su questo metodo risultano killed, eccetto per una dove il risultato è TIMED_OUT. Questo accade quando PIT genera una mutazione scambiando una if condition (riga 1184) che porta il metodo a entrare nel branch "interattivo" quando non dovrebbe. Quando questa mutazione è eseguita su un test in cui il metodo a cui è delegata l'interattività non è mockato, questo rimane lì bloccato fino al timeout. La mutazione dunque non è "killed", ma è pur vero che il test alla fine finirebbe per fallire per timeout, e la build non risulterebbe avvenuta con successo.

mountLedgerStorageOffline

{classe BookieImplMountLedgerTest}

Questo metodo statico di BookieImpl ha lo scopo di effettuare il deploy di un un ledger storage. I parametri che prende in input sono un LedgerStorage, e tramite un oggetto di tipo ServerConfiguration legge i parametri diskSpaceThreshold e warnThreshold, entrambi dei float.

Classi di Equivalenza Il Ledger storage, come riportato dalla documentazione, essendo un'interfaccia ha tre diverse classi che la implementano, che in aggiunta al valore NULL compongono le classi di equivalenze scelte per questo parametro, che quindi sono:
{NULL}, {INTERLEAVED LedgerStorage}, {SORTED LedgerStorage}, {DB LedgerStorage}.
Per i parametri diskSpaceThreshold e warnThreshold, come intuibile dalla nomenclatura dei parametri stessi, e come confermato da un'analisi white-box del codice (sulla documentazione e sul Javadoc non è reperibile questa informazione), c'è il seguente vincolo:

> "Should be > 0 and < 1 and diskSpaceThreshold >= diskSpaceWarnThreshold"

Dunque, le classi di equivalenza per diskSpaceTreshold saranno:
{spaceTreshold < 0}, {spaceTreshold = 0}, { 0 < spaceTreshold < 1}, {spaceTreshold $\geq$ 1}.
Per warnTreshold invece sono state scelte, sulla base del vincolo appena visto:
{warnTreshold < spaceTreshold}, {warnTreshold = spaceTreshold}, {warnTreshold $\geq$spaceTreshold}.

Boundary Values I boundary value scelti sono immediati per il parametro LedgerStorage, dato che corrispondono con i valori delle partizioni scelte, mentre saranno:

- -1.0, 0.0, 0.1, 1.0 per spaceTreshold
- per warnTreshold avremo:
  (-1.1, -1.0, 0.0), (-1.0, 0.0, 1.0), (0.0, 0.1, 0.2), (0.9, 1.0, 1.1) rispettivamente ai valori di spaceTreshold.

Casi di Test In questo caso abbiamo un approccio multi-dimensionale per "spaceTreshold" e "warnTreshold", mentre riteniamo sufficiente un approccio uni-dimensionale per LedgerStorage, dato che il tipo di LedgerStorage non ha legami con gli altri due parametri. Per realizzare i vari casi di test, si vuole inserire quindi ogni ledgerStorage possibile almeno una volta insieme a dei parametri validi, per testare che l'esecuzione del metodo vada a buon fine con quello specifico ledgerStorage. Per farlo però "non bastano" le combinazioni di test che sono state selezionatoe fin'ora, poiché (vedi tabella 3), si hanno 12 casi di test al momento, ma solo 2 hanno delle combinazioni di warn e space treshold che possano far avere successo al metodo, ma ci sono 4 possibili valori per ledgerStorage da provare, e non si vuole, come detto, provare alcuni tipi di ledgerStorage solamente in test per cui è previsto in ogni caso il lancio di un'eccezione dovuto alla presenza di altri parametri scorretti.
Quindi si procede ad aggiungere due casi di test ulteriori con valori validi per spaceTreshold e warnTreshold. Per cercare di acquisire più confidenza sul funzionamento corretto del SUT, invece di riutilizzare parametri già scelti per altri test in precedenza, estriamo un nuovo boundary value per spaceTreshold. Infatti, abbiamo considerato nella partizione l'insieme { 0 < spaceTreshold < 1}, da cui per ora abbiamo estratto solo il valore "0.1". Quindi si decide di estrarre anche il boundary value "0.9", da usare in combinazione con l'unica scelta di warnTreshold che rende la combinazione di parametri valida, ovvero {warnTreshold < spaceTreshold}. Così arriviamo a tutti i test definiti nella tabella 3.

Commenti Come expected value ci si aspetta, tutte le volte che il vincolo descritto in precedenza non è rispettato, una IllegalArgumentException. Nel caso in cui il ledgerStorage passato è null, da un'analisi totalmente black-box ci si aspetterebbe una NullPointerException, o eventualmente un valore di ritorno null, dato che nella documentazione non è specificato il comportamento in questo caso. In realtà, come risulta poi da un'analisi white-box, se ledgerStorage = null, è il metodo stesso che ha la responsabilità di istanziare un ledgerStorage "di default". Siccome questo, anche se non è quello che ci si aspettava in un primo momento, è chiaramente un comportamento voluto dallo sviluppatore del metodo, si provvede ad adattare il test, ed è per questo che nella tabella 3 troviamo PASSED come expected value anche quando il ledgerStorage è null.

Miglioramenti del Test Inizialmente, per acquisire confidenza sul fatto che l'istanza che il metodo ritornava fosse effettivamente valida, ci si assicurava che il metodo per eseguire il "mount" non avesse generato eccezioni, ma poi si eseguiva solamente la chiamata al metodo *localConsistencyCheck()* sull'istanza che ci veniva ritornata, e si controllava che il numero di inconsistenze rilevate fosse pari a zero. Poi, sulla base di quanto visto dal report di PIT, si è scelto di controllare anche che un'istanza valida gestisse senza eccezioni il meccanismo dei checkpoint. Dal report di PIT, in particolare, si è notato che la mutazione che eliminava la chiamata a riga 368 (quella responsabile di settare

la CheckpointSource) poteva essere eliminata, senza che il test realizzato lo rilevasse. Questa era una grave lacuna del test, dal momento che si accettava come valido un LedgerStorage dove il meccanismo dei checkpoint poteva non funzionare, dato che non vi erano controlli a riguardo. Per rimediare sono stati estesi i controlli da fare per poter considerare l'istanza che veniva ritornata come valida. Nello specifico è stata aggiunta una chiamata al metodo *checkpoint* del LedgerStorage, per far si che questo sincronizzasse i suoi dati. Se il metodo sotto test non setta correttamente la CheckpointSource, questa chiamata fallisce. Infine, sempre guardando il report di PIT, rimaneva un'ultima eccezione che non veniva eliminata, quella che rimuoveva la chiamata al set del Checkpointer (riga 379 di BookieImpl). Per rendere più robusto il test anche riguardo a questo aspetto, ed eliminare questa mutazione, è stata apportata un ulteriore integrazione al metodo di test della classe BookieImplMountLedgerTest: si è fatto in modo che il LedgerStorage di cui veniva richiesto il "mount" tramite il metodo da testare, avesse un Checkpointer scorretto (lancia uncheked exception quando si invocano i suoi metodi). Questo LesgerStorage veniva poi passato come parametro al metodo da testare. Se mountLedgerStorageOffline non avesse settato correttamente un Checkpointer valido al posto di quello scorretto, la chiamata al metodo *onRotateEntryLog* (metodo di LedgerStorage che fa uso del Chekcpointer, aggiunta appositamente al test in fase di verifica di validità dell'istanza) sarebbe fallita, facendo fallire tutto il test.

Quality Assurance Dopo le integrazioni appena descritte, il test per questo metodo elimina tutte le mutazioni di PIT, ed ha il 100% di coverage se analizzato con ba-dua. Anche JaCoCo riporta una instruction coverage e una branch coverage del 100%. Non rimangono dunque ulteriori migliorie da apportare al test secondo l'analisi di questi criteri.

Costruttore

{classi BookieImplTest, BookieImplFailureTest, BookieImplMockTest}

A questo punto, avendo testato i principali metodi statici della classe BookieImpl, si vuole acquisire confidenza sull'istanza di BookieImpl, e quindi sul suo costruttore e su come, una volta eseguito, l'istanza risultante reagisce agli "stimoli esterni", ovvero all'invocazione di metodi su di essa.

Classi di Equivalenza Il costruttore di BookieImpl presenta nove parametri in input, che però diventano quattordici quando si vanno ad analizzare quelli presenti all'interno dell'oggetto ServerConfiguration. La segnatura del metodo è quella mostrate nel seguente snippet di codice.

Procediamo quindi con ordine per analizzare il category partition di ognuno dei parametri elencati. Bisogna notare che, per lanciare il costruttore di BookieImpl, viene fatto uso di un LedgerStorage (terzo parametro), e che per utilizzare questo oggetto con il bookie, bisogna prima effettuarne l'inizializzazione, che per essere eseguita in modo coerente con il BookieImpl a cui poi va passato come parametro, dovrà avere alcuni parametri in comune con il costruttore di BookieImpl. In particolare entrambi i costruttori fanno uso degli oggetti: conf, ledgerDirsManager, indexDirsManager, statsLogger, byteBufAllocator (vedi listing 4) .
Dunque, per questi oggetti non utilizzeremo il valore NULL come classe nella scelta delle classi di equivalenza, perché una scelta del genere porterebbe a generare una NullPointerException nell'inizializzazione del LedgerStorage, che è un'operazione preliminare alla chiamata alla costruzione del Bookie. L'assunzione che si sta facendo dunque è che valori NULL non possano rientrare nel dominio dei parametri citati sopra, poiché un loro valore NULL creerebbe una failure prima che si possa arrivare ad istanziare un oggetto BookieImpl tramite il suo costruttore.
L'assunzione che il ledgerStorage vada inizializzato con gli stessi parametri del BookieImpl non è purtroppo un'informazione che è possibile ricavare dalla documentazione. In qualche modo è intuibile dalla nomenclatura dei parametri di entrambi i metodi, ma non essendo una "prova" sufficiente, si è proceduto ad un'analisi più white-box. Andando a guardare nel codice sorgente gli utilizzi del costruttore di BookieImpl che si sta testando, si vede come tutte le volte che il costruttore viene invocato, in maniera preliminare c'è l'inizializzazione del LedgerStorage con gli stessi oggetti che vengono poi passati al costruttore di BookieImpl.
Tutta questa assunzione si basa a sua volta sulla "Competent Programmer Hypothesis": si sta considerando che il programmatore sappia che l'oggetto LedgerStorage vada effettivamente inizializzato prima di poter essere passato come argomento al costruttore di BookieImpl. È chiaro che si potrebbe passare uno dei parametri discussi prima come NULL al costruttore di BookieImpl, ma se si arriva a quel punto senza ancora aver avuto eccezioni/errori, vuol dire che non si è inizializzato il LedgerStorage che si andrà ad usare con il bookie, il che è un caso di programmazione scorretta.

- Per **ServerConfiguration** quindi una configurazione NULL o Invalida avrebbe reso impossibile l'inizializzazione del LedgerStorage, dunque le sue classi di equivalenza individuate sono state individuate sulla configurazione dello stato di una sua istanza valida. Sono stati presi in considerazione gli attributi bookiePort (int), flushInterval (int), logPerLedger (boolean), journalDirs (String[]), ledgerDirs (String[]). Tutte le classi di seguito elencate si riferiscono dunque a un'istanza valida di ServerConfiguration:
  {bookiePort < 0}, {bookiePort = 0}, {$0 < bookiePort \leq 65535$}, {bookiePort > 65535}, {flushInterval < 0}, {flushInterval = 0}, {flushInterval > 0}, {logPerLedger = true}, {logPerLedger = false}, {valid journalDirs}, {invalid JournalDirs}, {empty journalDirs}, {valid ledgerDirs}, {invalid ledgerDirs}, {empty ledgerDirs}.

- Per **RegistrationManager** le classi di equivalenza scelte sono:
  {NULL}, {Istanza Valida}, {Istanza non valida}.
  L'istanza valida è stata individuata nella classe di Bookkeeper *NullRegistrationManager*, un registration manager che consideriamo valido poichè è una sorta di dummy, dato che non compie realmente alcuna operazione quando i suoi metodi sono invocati, e dunque non può sollevare eccezioni. L'istanza non valida invece è stata implementata tramite la classe *InvalidRegistrationManager*, una classe realizzata appositamente per questo test, che lancia un'eccezione BookieException ogni qual volta un suo metodo viene invocato.

- Per **LedgerStorage**, come già fatto per mountLedgerStorageOffline sono state scelte le classi di equivalenza:
  {NULL}, {INTERLEAVED LedgerStorage}, {SORTED LedgerStorage}, {DB LedgerStorage},
  ricavate dalle classi che implementano questa Interface.

- Per **DiskChecker** sono state scelte:
  {NULL}, {istanza valida}, {istanza non valida}.
  L'istanza valida è un semplice DiskChecker che rispetta questo requisito osservato in precedenza. L'istanza non valida è stata realizzata con la classe *InvalidDiskChecker*, realizzata per questo test, e che ritorna valori del tutto scorretti quando i suoi metodi vengono chiamati (qui uno snippet del suo codice).

- Per **ledgerDirsManager** e **indexDirsManager** (entrambi oggetti di tipo LedgerDirsManager) sono state individuate le classi di equivalenza:
  {NULL}, {istanza valida}, {istanza non valida}.
  Un'istanza valida è un oggetto di tipo LedgerDirsManager istanziato in modo che il suo costruttore non lanci nessuna eccezione. Un'istanza non valida è quella realizzata tramite la classe InvalidLedgerDirsManager (qui il codice della classe su GitHub). Per come è istanziata in questo test ritornerà dei valori scorretti quando sono invocati i suoi metodi (per esempio una lista di file con un path scorretto).

- Per **StatsLogger** in maniera simile sono state scelte le classi di equivalenza:
  {NULL}, {istanza valida}, {istanza non valida}.
  L'istanza valida è quella implementata dalla classe di Bookkeeper NullStatsLogger, una sorta di logger dummy che non esegue vere operazioni e che quindi non solleva mai errori. Per l'istanza non valida è stata creata appositamente la classe InvalidStatsLogger (qui il suo codice), che fa il throw di una RuntimeException quando si invocano alcuni dei suoi metodi.

- Per il **ByteBufAllocator** sono state prese le classi di equivalenza:
  {PooledByteBufAllocator}, {UnpooledByteBufAllocator}.
  Sono entrambe istanze valide di ByteBufAllocator. Questa scelta è dovuta a quanto discusso in precedenza sugli oggetti usati sia da BookieImpl che da LedgerStorage. Avendo due diversi tipi di implementazione del ByteBufAllocator, si è ritenuto che testarle entrambe avrebbe aumentato il livello di confidenza sul corretto funzionamento della classe BookieImpl, poichè ci si aspetta che qualsiasi sia il ByteBufAllocator scelto, purché valido, il comportamento della classe BookieImpl sia lo stesso.

- Infine, per l'oggetto di tipo **Supplier<>** sono state scelte le classi di equivalenza:
  {NULL}, {Istanza valida}.
  L'istanza non valida non è stata considerata poiché la classe che va utilizzata è final, dunque non può essere estesa o mockata con Mockito. Sarebbe stato possibile probabilmente intervenire, magari tramite l'uso di tools come PowerMockito, ma ciò avrebbe richiesto ulteriore tempo che si è ritenuto non fosse speso in maniera del tutto efficiente dato che l'istanza scorretta avrebbe causato problemi, ma non direttamente alla classe BookieImpl, che al momento è il nostro SUT, perché questo oggetto non è utilizzato direttamente da BookieImpl, ma da altri oggetti. Dunque, soprattutto per motivi di tempo, si sono selezionate solamente queste due classi di equivalenza. L'utilizzo anche della classe di equivalenza {Istanza non valida} potrebbe comunque essere qualcosa che andrebbe ad aumentare la completezza del test, e che potrebbe essere implementata in un successivo raffinamento del test.

Boundary Value

- Per ServerConfiguration:
  - bookiePort: -1, 0, 1, 65536
  - flushInterval: -1, 0, 1000
    Si è scelto 1000 e non 1, che sarebbe stato il valore più vicino alla boundary poiché questo valore rappresenta in millisecondi ogni quanto devono essere flushati gli elementi del Bookie, e quindi 1 ms sembrava un valore poco realistico. Inoltre il default di questo valore è 10000, e la documentazione dice che questo valore può anche essere incrementato per migliorare le performance. Dunque il valore 1000 è sembrato un buon candidato per essere abbastanza vicino alla boundary, ma comunque più vicino all'ordine di grandezza del valore di default rispetto ad 1.
  - logPerLedger: true, false

  - journalDirs e ledgerDirs: [], [Array con path validi e cartelle esistenti], [Array che contiene il path ”notAPath\0]

- Per tutti gli altri parametri la scelta dei boundary value è ridondante, dal momento che corrispondono semplicemente alle classi di equivalenza, ed è stato già spiegato che cosa si intende per istanze valide e non valide per ognuno di essi.

Casi di Test Per il costruttore di BookieImpl si è scelto un approccio uni-dimensionale, dato che non ci sono particolari combinazioni di parametri che possono portare ad esiti diversi per il costruttore. La tabella 4 mostra i casi di test realizzati. Essendo il numero di parametri da utilizzare molto alto (14 parametri) si è scelto di appoggiarsi su una classe che facesse da bundle per tutti questi parametri, in modo da avere il costruttore della classe di test che prendesse in input solamente un oggetto di tipo InputBundle, da cui poi potesse estrarre i vari parametri settati nel metodo annotato con @Parameters. Inoltre, si è scelto di creare una configurazione di ”default” dell'InputBundle (con tutti valori validi), per poi di volta in volta andare a modificare solo i parametri di interesse per quello specifico bundle. I test sono stati divisi in tre diverse classi: BookieImplTest per quelli che devono ritornare un'istanza valida di BookieImpl, BookieImplFailureTest per quelli dove la costruzione dell'istanza non deve andare a buon fine, e BookieImplMockTest per tutti quei casi di test dove si sono usati dei mock.
In BookieImplTest ci si è posti il problema di assicurarsi che l'istanza di BookieImpl che viene generata, sia valida. Per farlo il test scritto procede a stimolare in diversi modi l'istanza creata, per controllare che ad ogni stimolo (chiamata a metodo) corrisponda il comportamento che ci si aspetta. Nello specifico il test:

- controlla che il bookie risulti effettivamente running
- se il bookie è stato creato con solamente array di cartelle vuote (quindi non ha cartelle su cui scrivere) ci si aspetta che il bookie risulti readOnly. In caso invece ci sia almeno una cartella su cui si può scrivere ci si aspetta che il bookie non sia readOnly
- controlla che lo spazio disponibile sul bookie sia minore o uguale a quello totale
- controlla che le cartelle dei ledger siano settate correttamente, controllando che ognuna di esse sia presente nella lista di cartelle dei ledger del bookie
- Se nel costruttore è stato settato un bookieId, controlla che quello del bookie istanziato corrisponda a quello settato nella ServerConfiguration (questo meccanismo viene eseguito una sola volta, ed è stato aggiunto in seguito, per coprire il branch di riga 245, che come si vedeva dal report di JaCoCo, non aveva coverage)
- Controlla che non si possa scrivere sui ledger qualora questi lancino una ”NoWritableLedgerDirException”
- Controlla che non si possa scrivere su un ledger se questo risulta *fenced*
- Controlla che se invece il ledger non lancia eccezioni, l'operazione di aggiunta di una entry su di un ledger vada a buon fine e che la callBack venga correttamente eseguita. Questo test deve generare un'eccezione solo nel caso in cui il bookie creato sia readOnly per le motivazioni viste al punto 2.

Se un'istanza supera tutti questi controlli con i comportamenti previsti allora l'istanza viene considerata valida, e il test ha successo.

Nella classe BookieImplFailureTest invece bisogna assicurarsi che tutte le istanze che il test parametrizzato va a generare non vengano istanziate, o se vengono istanziate sollevano eccezioni quando si provano a stimolare in modo simile a quello di BookieImplTest. Questa classe di test infatti fa in modo che il test fallisca se una delle istanze create con parametri scorretti riesce ad eseguire tutte queste operazioni senza sollevare eccezioni. Qui il codice del test, i metodi chiamati sull'istanza sono sostanzialmente gli stessi che devono però andare a buon fine in BookieImplTest. Infine, nella classe BookieImplMockTest si crea un'istanza valida di BookieImpl, e con l'ausilio dei mock si procedono a testare dei branch che non venivano toccati dalle altre due classi di test. Questa classe di test infatti è stata realizzata a seguito del report di JaCoCo. Riguardo a questa classe di test, si vuole porre un'attenzione maggiore su tre dei metodi di test, in quanto hanno evidenziato un potenziale bug. Partiamo dal metodo di test *localConsistencyCheckMockExtended*. Questo test, tramite l'utilizzo di mock, ha lo scopo di andare a testare la situazione in cui conf.isLocalConsistencyCheckOnStartup() = true, e dunque si entra nel branch di riga 674. Il mock è poi configurato in modo da lanciare una IOException sull'invocazione di ”ledgerStorage.localConsistencyCheck(Optional.empty())” (riga 678), per poi entrare nel catch alla riga successiva. Qui viene invocata la chiamata ”shutdown(ExitCode.BOOKIE_EXCEPTION)”. Dunque inizalmente il test era configurato per avere una assert che controllasse che, a seguito di questo flusso di esecuzione, il bookie terminasse la sua esecuzione, e che il codice di uscita fosse effettivamente ExitCode.BOOKIE_EXCEPTION. Nel test configurato in questo modo però, quest'ultimo controllo falliva. Dunque si è proceduto a una approfondita analisi white-box del flusso di esecuzione per capire il motivo del fallimento di questo test. Utilizzando anche il debugger ci si è accorti di un comportamento inaspettato del SUT. Infatti, seguendo il flusso di esecuzione, notiamo che:

- Viene chiamato il costruttore di BookieImpl
- Terminata la costruzione del bookie, si procede ad eseguire lo start()

- Per la configurazione dei mock si arriva a riga 681, dove a seguito della IOException verificatasi si invoca lo shutdown del bookie con codice BOOKIE_EXCEPTION
- Si entra nel metodo shutdown del bookie (riga 847). In questo momento è importante notare come l'attributo isRunning dell'istanza bookie sia = **false**, dal momento che l'esecuzione del metodo start() si è fermata a riga 681, ma il bookie sarebbe stato settato come running solamente a riga 719 del metodo start()
- Quindi, nel metodo shutdown, tutto il branch in cui si entrerebbe qualora isRunning() ritornasse true non viene eseguito
- Quindi il return di riga 890 ritorna un exitCode che non è mai stato modificato, dal momento che può essere settato solamente a riga 855, ovvero all'interno di quel branch in cui questo flusso di esecuzione non può entrare.

Questo porta alle seguenti considerazioni:

- Durante la start si verifica quella che anche il commento del log definisce una "fatal exception", ma il codice di uscita del bookie è semplicemente "OK", il che è decisamente un comportamento inaspettato e per certi versi poco sensato: perché se si verifica una "fatal exception", e viene richiesto quindi lo shutdown del bookie con un determinato codice di errore, poi ci si ritrova un codice di uscita "OK"?
- Si nota anche come, per questo flusso di esecuzione venga chiamata l'esecuzione del metodo shutdown con un parametro intero (il codice di uscita), che però non viene mai utilizzato, dal momento che il bookie non risulta ancora running. Infatti il parametro exitCode definito a riga 847, non ha uno use al di fuori del branch in cui entra solamente il bookie running. Questo, di fatto, rende inutile chiamare la shutdown con un qualsiasi codice di uscita nel metodo start() di BookieImpl prima di riga 719, dato che quel codice di uscita non verrà mai utilizzato. In sintesi significa che, durante l'esecuzione della start(), se si verifica un errore fatale prima che il bookie sia segnalato come running, il codice di uscita non potrà che essere "OK", anche se si richiede la shutdown con un codice diverso.
- Inoltre, altro fatto rilevante è quello che riguarda l'oggetto dirsMonitor di BookieImpl. Infatti, di questo oggetto viene chiamata la start a riga 651 del metodo start() del Bookie. In questo modo si lancia un thread per monitorare i vari Disk del bookie. Dunque questo thread viene eseguito prima che il bookie risulti running. Durante la shutdown si va anche a chiudere questo dirsMonitor (riga 876), ma c'è un problema: il dirsMonitor viene chiuso nel branch in cui si entra se il bookie è running. Dunque, con il flusso di esecuzione che abbiamo in questo test succede che:
  - Il bookie viene istanziato
  - Viene eseguita la start()
  - Viene lanciato il dirsManager
  - La start viene interrotta da una IOException e si chiama la shutdown(..) del bookie
  - Non si entra nel branch che chiude i vari componenti del bookie, perchè il bookie non risulta running
  - Il bookie termina la sua esecuzione.

  Quindi il dirsManager viene lanciato, ma non viene mai chiuso, anche dopo la terminazione del bookie. A riprova di questo si è provato a eseguire il seguente test: se si va a modificare temporaneamente il codice sorgente di LedgerDirsMonitor, rendendo public il suo executor, ci accorgiamo che, con il flusso di esecuzione che si sta discutendo, il controllo:

```
1                                   assertTrue(bookie.dirsMonitor.executor.isShutdown());
```

  fallisce anche se eseguito dopo che il bookie ha terminato la sua esecuzione a seguito della shutdown(..). Andando poi a chiamare manualmente "bookie.dirsMonitor.shutdown();", che è la chiamata che verrebbe effettuata, ma non viene eseguita, a riga 876 di BookieImpl, e ripetendo lo stesso controllo si vede che ora il controllo viene eseguito con successo, ed è quindi un controllo che dovrebbe passare. Come ulteriore controprova, se si fa la stessa cosa con i bookie della classe BookieImplTest (dove non ci sono eccezione e la start() va a buon fine), questo controllo è superato da ognuno di essi. Sostanzialmente in questa situazione il dirsManager viene lanciato, ma non viene mai chiuso. Questo può portare ad un uso scorretto dei thread e della memoria, ed è per questo che viene segnalato come possibile bug.

Nota: il catch attorno all'assert di riga 235 di questo test, è stato inserito per far passare comunque la build, ma per evidenziare come il valore di ritorno del codice di uscita non sia quello che ci si aspetterebbe. Il discorso fatto per questo test è del tutto simile anche per il metodo di test localConsistencyCheckMockWithErrors() e readJournalFailInStart(), dato che sostanzialmente seguono un flusso di esecuzione molto simile, registrando un'eccezione durante la start dopo l'invocazione di dirsManager.start() ma prima del metodo che setta il bookie come running.

Volendo provare a stimare la reliability del costruttore della classe BookieImpl, considerando il profilo operazionale che comprende ogni combinazione di input con cui si esegue un test, si hanno un totale di 52 input possibili. Quelli che riscontrano il possibile bug sono 3. Assumendo che ogni input sia equiprobabile, si ottiene che ognuno di essi ha una probabilità del 1.92% di essere utilizzato. Dunque, la reliability di questo profilo operazionale risulta essere $1 - P(failure) = 1 - 3 \cdot 0.0192$ = 0.9424 = 94.24%.

Quality Assurance Per quanto riguarda il costruttore di BookieImpl, dopo l'aggiunta della classe BookieImplMock-Test, creata per estendere la coverage di JaCoCo, si ha una instruction coverage del 99% e una branch coverage dell'83%. Ba-dua riporta una data-flow coverage di 104/107 (97%). Infine, sempre all'interno del costruttore, i test fin'ora realizzati eliminano tutte e 15 le mutazioni prodotte da PIT. Sebbene la data-flow e la control-flow coverage non abbiano raggiunto il 100%, si è scelto di fermarsi qui nel testare il costruttore di BookieImpl, in parte per motivi di tempo, in parte anche perché la qualità dei test sembra comunque essere soddisfacente, dato che le percentuali per JaCoCo e Ba-dua sono molto vicine al 100%, e con PIT si eliminano il 100% delle mutazioni.

A questo punto sostanzialmente è stata testata gran parte della classe BookieImpl, e quindi vengono riportate le statistiche a livello di classe. JaCoCo segnala una instruction coverage dell'83% e una branch coverage del 78%. PIT invece per questa classe riporta una mutation coverage del 69%, che diventa dell'76.5% se si escludono le mutazioni su porzioni di codice che non hanno coverage.

1.3.2 Journal

{classi JournalTest, JournalV5Test, setUpJournalTest}

L'idea che si è scelta per testare il journal nelle prime due classi è abbastanza semplice. Si costruiscono due Journal, uno con un certo numero di bytes scritti sopra al momento della creazione, l'altro vuoto. A questo punto si vuole vedere che, per ognuno dei due Journal il metodo scan(..) del Journal ritorni ciò che ci aspettiamo: nel caso del primo Journal l'esatto numero di bytes che c'erano scritti sopra, nel secondo caso invece un numero di bytes minore (ma maggiore di zero perché ogni Journal ha una sorta di header).

Classi di Equivalenza Per questo test sono molto semplici: il metodo scan prende in input un intero che rappresenta l'indice da cui iniziare la scansione e uno JournalScanner per eseguirla. Quindi:

- int scanPos: {< 0}, {= 0}, {> 0}, { scanPos > 0 && scanPos > lunghezza del Journal}
  Le prime tre sono banalmente le classi date dal tipo di dato intero, l'ultima è stata introdotta perché questo intero rappresenta la posizione da cui cominciare la scansione/lettura del Journal, e se ne vuole testare il comportamento nel caso in cui la posizione di inizio lettura fosse "oltre" l'ultima scrittura del Journal stesso (il Journal può essere visto come una serie sequenziale di bytes).
- JournalScanner scanner: {invalid}, {valid}, {null}

Boundary Values

- int scanPos: -1, 0, 1, Integer.MAX_INT
  Non potendo sapere con precisione la posizione dell'ultima scrittura, si prende il valoer massimo possibile
- JournalScanner scanner: {invalid}, {valid}, {null}

Casi di Test Qui (tabella 5) tutti i casi di test selezionati. Dato che scanPos determina dove inizia la scansione, per entrambi i Journal si vogliono provare tutte le possibili combinazioni di questo indice (dunque approccio multi-dimensionale). Solo il caso in cui scanPos = MAX_INT è stato considerato in modo uni-dimensionale, dal momento che essendo un indice che va oltre l'ultima scrittura sul Journal, sarebbe equivalente per entrambi i tipi di Journal. Per il tipo di scanner invece, ritenendo che un approccio multi-dimensionale non avrebbe accresciuto la confidenza nel funzionamento del Journal, si è scelto un approccio uni-dimensionale. Infatti, è abbastanza chiaro che se per esempio lo scanner è null, ci aspettiamo che sia lanciata una NullPointerException, a prescindere da quale possa essere il Journal o l'indice da cui iniziare la scansione. Da qui la scelta dell'approccio unidimensionale per questo caso di test. Da notare che se l'indice scanPos è negativo, giustamente, la scansione inizierà comunque da 0, dato che iniziare "da -1" non avrebbe senso. La classe JournalV5Test fa esattamente la stessa cosa ma usa un tipo di scrittura differente che fa uso anche di byte di padding. Si è scelto di implementarla a seguito dei report di JaCoCo e ba-dua, che segnalavano la presenza di un branch di scanJournal senza nessun tipo di coverage, proprio perché processava i byte di padding.

Costruttore Ovviamente si è andato a testare anche il comportamento del costruttore di Journal, e quello delle istanze da esso generate. Gran parte dei parametri che prende in input il costruttore sono stati già discussi qui, quando si è testato il costruttore di BookieImpl. Nella tabella 6 si trovano i casi di test selezionati (con approccio unidimensionale, poiché si è ritenuto che un approccio multi-dimensionale avrebbe aumentato il numero di casi di test, ma non la confidenza del funzionamento della classe stessa). Nella sua descrizione si trovano alcuni commenti sui parametri. In questo test si è affrontata la problematica di definire se l'istanza di Journal che viene ritornata fosse effettivamente un'istanza valida. Non potendo avere la certezza assoluta che un'istanza fosse valida o meno, si è stabilito che se un'istanza avesse superato i seguenti controlli, allora c'era sufficiente confidenza che fosse un'istanza valida:

- Non si generino eccezioni durante la sua creazione
- La journalDirectory è impostata correttamente
- Il Journal è in esecuzione
- La sua cartella abbia i corretti permessi di lettura e scrittura
- La size massima del Journal sia stata settata correttamente (è stato usato il default: 2 MB)
- La memoria che risulta attualmente utilizzata non sia superiore di quella totale disponibile
- Il meccanismo dei checkpoint funziona come previsto

Per quanto riguarda la spiegazione su che cos'è il meccanismo dei checkpoint, e come questi sono stati testati, si rimanda a questo commento nel codice stesso della classe. Questo commento (e i successivi) spiegano in maniera dettagliata cosa è stato fatto, e la comprensione risulta molto più semplice avendo sotto mano, oltre alla spiegazione, anche il codice che implementa quei passaggi. Per questo metodo, ba-dua segnala 72/78 DU coperti, e si sceglie di fermarsi qui poiché 5 dei 6 mancanti riguardano l'uso della variabile LOG, o l'utilizzo di alcune variabili all'interno delle "print" del log, che si è ritenuto superfluo testare. Il sesto riguarda la seconda delle variabili dell'*or* di riga 704. Con maggior tempo sarebbe stato possibile coprirlo, ma poiché si tratta di una semplice assegnazione di un valore booleano ad una variabile, non essendo così rilevante, si è deciso di tralasciarlo.

Commenti

- Per il test di setUpJournalTest è stata usata un anonymous class che estende il Journal e fa l'override dell'handleException, poiché la versione originale nel corpo di quel metodo chiama la exit() del processo, interrompendo il test senza dare l'opportunità di controllare che quella exit() fosse stata chiamata quando questo era effettivamente il comportamento previsto.
- In setUpJournal, nel caso di test in cui si usa un LedgerDirsManager invalido, se questo ritorna "solo" parametri scorretti, invece di lanciare un'eccezione unchecked, il comportamento scorretto viene evidenziato solo in maniera molto "blanda" dal log. Per chiarire meglio quanto appena detto, poniamoci nel caso di test in cui appunto si usa un InvalidLedgerDirsManager. Quando il journal gli richiede l'elenco delle direcotry dei ledger, se l'istanza invalida ritorna un array con un path scorretto (es.: contenente il path "notAPath\0"), si genera una IOException, ovviamente, di cui viene fatto il catch. A questo punto però, non c'è nessuna logica di gestione dell'errore, ma ci si limita a segnalare quanto accaduto all'interno del Log. Inoltre, il messaggio con qui questo viene segnalato è il seguente:

  "Problems reading from *$fileName* (this is okay if it is the first time starting this bookie)"

  Un messaggio di errore che quindi si può presentare anche in situazioni di normale esecuzione. Questo comportamento non viene segnalato come possibile bug perché sembra essere un comportamento voluto, ma sicuramente potrebbe essere raffinato, magari separando il caso in cui c'è una IOException ma solo perché è la prima volta che si avvia il bookie associato al Journal, e il caso in cui si genera perché c'è stato realmente un problema.
- Nel test setUpJournalTest, quando si va a istanziare un Journal con il parametro journalDirectory = null, si ha un potenziale comportamento indesiderato da parte del SUT. Il costruttore infatti non solleva nessuna eccezione e l'istanza viene ritornata "normalmente" al chiamante. Alcuni dei test che poi vengono eseguiti su questa istanza riescono anche a passare: per esempio la dimensione massima che il Journal sostiene di poter raggiungere è quella corretta, anche se questo risulta strano. Come può un Journal che non ha una cartella su cui scrivere, arrivare ad occupare 2 MB di memoria come dimensione massima? Il controllo di riga 283 per esempio fallisce, poiché chiaramente finisce per generare una NullPointerException. Il che conferma che si ha un Journal su cui non è possibile scrivere ne leggere. Un altro comportamento ambiguo da parte del Journal è quello che ha quando si prova a eseguire una logAddEntry su questa istanza. Quando infatti si invoca questo metodo per provare ad aggiungere una entry sul log del Journal, questa istanza entra in un'attesa infinita. Per far passare il test si è messo un countown tramite un latch (vedi riga 317). Il countodown attualmente ha un'attesa massima di 3 secondi, ma il comportamento risulta essere lo stesso anche con attese ben più lunghe. Si è provato anche

con timer di 60 secondi, e il risultato è lo stesso. Poi per velocizzare la build questo valore è stato abbassato. Dunque sembra chiaro che un Journal istanziato in questo modo non è utilizzabile, ma allora perché non impedirne la creazione, lanciando un'eccezione durante l'esecuzione del suo costruttore? Per questi motivi questo possibile bug è segnalato come comportamento indesiderato.

- Se si considera il profilo operazionale composto dagli input di tutti i test eseguiti sul costruttore del Journal, si può stimare la reliability relativa a questo profilo. Se si considera l'input appena discusso come un input che porta a un comportamento indesiderato, e quindi a una failure (attesa infinita quando si esegue una scrittura), si ottengono le seguenti stime. In totale si hanno 14 combinazioni di input che vengono utilizzati, ed una sola che porta ad avere il problema descritto. Assumendo che tutti gli input abbiano una probabilità uniforme dello 0.0714 (7.14%), si ottiene che la reliability, calcolata come $1 - P(failure) = 1 - 0.0714 = 0.9286 =$ 92.86%.

Quality Assurance Per la classe Journal JaCoCo riporta una instruction coverage dell'87% e una branch coverage del 69% (a livello di classe, per il costruttore sono 92% e 80% rispettivamente). Si va ora ad analizzare invece i risultati di PIT. Come già detto, questo tool è stato fondamentale per introdurre maggiori controlli sulle istanze che vengono considerate valide dai test realizzati per la classe Journal. Andando a guardare i risultati, però, troviamo che la classe Journal ha una mutation coverage del 43%, che sale solo al 46% escludendo le mutazioni senza coverage. Questo dato è in forte controtendenza rispetto a quello della classe BookieImpl, che aveva una mutation coverage ben più alta (76.5% contando solo quelle dove si ha coverage). Sopratutto se si considera che ci si trova a parità di instruction coverage per entrambe le classi (esattamente l'87% per entrambe). Sicuramente, almeno in parte, questi dati possono essere spiegati da una minore qualità dei test per la classe Journal rispetto a quella BookieImpl, ma è interessante analizzare alcuni dati ulteriori a riguardo. Infatti, dal report di JaCoCo risulta come, a parità di instruction coverage, la branch coverage di BookieImpl sia molto più alta di quella di Journal (81% contro il 69%). Questo ci può far intuire che la classe Journal ha, in proporzione, un codice con branch condizionali più frequenti, il che la renderebbe più difficile da testare.

A riprova di questo si può analizzare quanto prodotto dall'analisi del codice di SonarCloud. Come segnalato anche nel suo report, ci sono alcuni metodi della classe Journal con una cognitive complexity **molto** elevata: *checkpointComplete* con una complessità pari a 16, *scanJournal* con una complessità di 28 e addirittura il metodo *run* con una complessità di 107, quando il massimo (secondo gli standard di SonarCloud) dovrebbe essere 15! Inoltre, come segnalato sempre qui da SonarCloud, *run* risulta essere un "Brain Method", suggerendo:

> "Refactor it to reduce at least one of the following metrics: LOC from 203 to 64, Complexity from 52 to 14, Nesting Level from 7 to 2, Number of Variables from 31 to 6"

Come si può capire da queste metriche, questo metodo, che è per l'appunto quello che implementa quasi tutta la logica del Journal, è estremamente vasto in termini di branch, numero di variabili e numero di if condition annidate, ed è quindi molto complesso da testare. In confronto, il metodo con maggior complessità di BookieImpl ne ha una pari a 40: meno della metà rispetto al metodo run di Journal. In conclusione, non si può escludere che la qualità dei test realizzati per Journal sia più bassa, ma è anche abbastanza evidente che questa classe sia ben più complicata da testare, e questo spiega, almeno in parte, la grande differenza tra i due valori di mutation coverage.

(Qui una breve panoramica, qui il download del report completo).

1.4 Integration Test

classi BookieImplIT, JournalBufferedChannelIT

Sono state realizzate due classi per gli integration test:

- La prima, BookieImplIT, ha come SUT la classe BookieImpl e l'implementazione dell'interfaccia LedgerStorage associata al bookie utilizzato. Quello che si vuole testare è il corretto scambio di messaggi e la corretta interazione tra le due classi. BookieImpl infatti è la classe che rappresenta un nodo di Bookkeeper (una sorta di server), i Ledger sono i dischi su cui vengono raccolti i dati. Quello che si vuole testare dunque è lo scenario in cui al nodo rappresentato dal nostro bookie, venga richiesto di scrivere su uno dei suoi dischi una generica entry. La scrittura in memoria della entry quindi viene richiesta al bookie, che poi delega l'effettiva scrittura al Ledger. Per controllare che lo scambio di messaggi tra queste due entità sia avvenuto in maniera corretta, dopo aver richiesto la scrittura, si fa in modo che il bookie richieda ora di leggere dal Ledger la entry (e si specifica lo stesso id specificato in fase di scrittura per la entry e per il ledger). Il test poi confronta ciò che il bookie ha letto dal ledger con ciò che aveva richiesto al bookie di scrivere. Questi due dati potranno essere uguali se e solo se lo scambio di messaggi tra le due entità è andato a buon fine, e solo se lo scambio di dati è avvenuto secondo i protocolli stabiliti tra queste due classi.

- La seconda, JournalBufferedChannelIT, ha lo scopo di testare l'interazione tra alcune componenti della gestione dei Journal (che nell'architettura di Bookkeeper rappresentano una sorta di file di log per i Bookie). In particolare

qui il SUT è rappresentato dalla classe Journal, dalla classe BufferedChannel e dalla classe JournalChannel, che permette al possessore del Journal di ottenere l'oggetto di tipo BufferedChannel per scrivere sul Journal. Il meccanismo del test prevede di istanziare un Journal (che si assume aver superato tutti i test di unità), istanziare un oggetto di tipo JournalChannel relativo al Journal in questione, e da questo farsi ritornare il BufferedChannel. A questo punto si richiede di scrivere sul Journal 3 Long e una String tramite il BufferedChannel. Una volta che la scrittura è terminata, si chiede al Journal di eseguire una scanJournal, e tramite l'impiego di un apposito JournalScanner si vanno a leggere i byte che si trovano ora sul Journal. Il test passa solamente se:

- Si leggono correttamente i metadati scritti tramite il BufferedChannel sul Journal
- Il valore dei 3 Long e della String scansionati dalla scanJournal corrisponde con il valore di quelli passati al BufferedChannel
- Il numero di byte di padding allinea correttamente il Journal per la prossima scrittura.

Se tutto questo accade, significa che le due classi hanno interagito correttamente. Per dettagli più specifici su questa implementazione si rimanda alla classe di test stessa, e in particolare a questo e al successivo block comment. Come per il precedente, questo Integration Test viene superato con successo solamente se l'interazione tra le classi del SUT avviene con successo, e se vengono rispettati i protocolli per lo scambio di dati.

1.5 Note

- Sono stati integrati con le GitHub Actions tutti i tools utilizzati (JaCoCo e ba-dua) ed è stato integrato anche SonarCloud. L'unico tool che non è stato usato con le GitHub Actions, ma di cui si è fatto comunque ampio uso "offline", è stato PIT. Questo per delle problematiche riscontrate con il suo utilizzo in accoppiata con le GitHub Actions, dato che lanciando la sua esecuzione in un workflow, la virtual machine delegata alla sua esecuzione falliva sempre l'esecuzione finendo "out of memory" durante la prima fase di test della test suite, e da questa risultava poi che il progetto non disponesse di una "green suite" da utilizzare con PIT. Questa è una cosa mai successa offline ne con nessuno degli altri workflow. Per mancanza di ulteriore tempo per il troubleshooting quindi si è deciso di non integrare PIT con le GitHub Actions per questo progetto. (Questo problema si è riscontrato solo con BookKeeper, su OpenJPA è presente e funzionante il workflow che genera il report di PIT).

- La produzione del report .xml di Ba-dua avviene con successo eseguendo la build con il profilo "badua-profile", tuttavia per la generazione del report .xml si è dovuti ricorrere ad un workaround. Infatti, provando a strutturare il pom.xml come mostrato negli esempi dal team di badua, veniva generato il .ser, ma poi la classe Report di badua non riusciva a leggerlo. Dopo numerosi tentativi si è arrivati a trovare un workaround, utilizzando il .jar di ba-dua per il report da riga di comando, e facendo eseguire questa azione al plugin "exec-maven-plugin" di "org.codehaus.mojo". Qui l'implementazione di questa soluzione su OpenJPA, che è leggermente più elegante di quella di Bookkeeper. In OpenJPA infatti si esegue l'install di ba-dua, e poi si va a prendere il .jar necessario dalla cartella .m2 di Maven. In Bookkeeper invece c'è il .jar di ba-dua tra i sorgenti del progetto in una cartella separata. Chiaramente la soluzione adottata su OpenJPA è miglliore, e solo per mancanza di tempo non è stata trasposta anche su Bookkeeper.
  Questa soluzione, per quanto non ideale è stata comunque integrata con successo nei workflow di GitHub Actions per entrambi i progetti, sia in ambiente Unix (Bookkeeper), sia su Windows (OpenJPA).

2 OpenJPA

2.1 Continuous Integration

Come per Bookkeeper, anche per OpenJPA ci sono i workflow per JaCoCo, SonarCloud, ba-dua e su questo progetto anche per PIT. Tranne per quello di SonarCloud, dove il report (che sfrutta l'.xml di JaCoCo) è disponibile online, per tutti gli altri, nella pagina relativa all'esecuzione del workflow si trova, scaricabile come artifact, il relativo report. Inoltre, per questo progetto si è scelto di eseguire i workflow di JaCoCo, PIT e SonarCloud in ambiente Linux, per differenziare maggiormente l'ambiente di CI da quello locale di realizzazione dei test (Windows), in modo da comprovare il funzionamento dei test su ambienti e sistemi operativi diversi da questo.

2.2 SUT

Per OpenJPA sono state individuate come classi da testare:

- La classe AttachManager, del modulo openjpa-kernel
- La classe CacheMap del modulo openjpa-kernel

Durante i test di integrazione poi sono entrate a far parte del SUT anche le classi PreparedQueryCacheImpl e QueryImpl, rispettivamente dei moduli openjpa-jdbc e openjpa-persistence.

2.3 Unit Testing

2.3.1 AttachManager

{classi AttachManagerTest, AttachManagerAttachTest, AttachManagerAttachParamTest, AttachManagerAttachAllTest}

Per testare questa classe si è partiti dal suo costruttore, andando ad acquisire confidenza sulla validità delle istanze ritornate (classe AttachManagerTest. Nel paragrafo "Casi di Test" sarà poi spiegato come si decide se un'istanza ritornata è valida oppure no.)

Le classi AttachManagerAttachTest e AttachManagerAttachParamTest hanno lo scopo di testare il meccanismo di attach della classe, andandolo a testare in maniera parametrica (multi-dimensionale, classe AttachManagerAttachParamTest) e con l'ausilio di ulteriori mock per coprire branch che non ricevevano copertura (AttachManagerAttachTest, classe aggiunta a seguito dei report di PIT e JaCoCo). Infine, la classe AttachManagerAttachAllTest ha lo scopo di testare il meccanismo fornito dal metodo *attachAll*, analizzandone vari scenari (nessuna failure, failure per tutte le attach, failure solo per alcune delle attach) e verificando che i comportamenti dell'istanza di attachManager corrispondano a quelli che ci si asepterebbe.

Category Partition Per il costruttore di AttachManager si hanno i seguenti parametri:
un oggetto di tipo Broker, il boolean newCopy, l'Object callBack e la List toAttach.

- Broker: {NULL}, {Valid}, {Invalid}
- newCopy: {true}, {false}
- callBack: {NULL}, {Valid}, {Invalid}
- toAttach: {NULL}, {emptyList}, {lista con elemento null}, {|list| = 1}, {|list| > 1} Qui si scegli di testare dimensioni della lista uguali e maggiori di uno in due categorie separate poiché avere una lista di dimensione maggiore di uno può avere implicazioni diverse sul costruttore rispetto alla lista con un solo elemento, dal momento che, una lista con più elementi potrebbe portare a dover ripetere determinati comportamenti più volte , "stressando" in modo diverso il costruttore rispetto alla lista con un solo oggetto.
  La lista con elemento null è diversa dalla classe {NULL}, poichè non è toAttach ad essere null, ma il suo contenuto. Le istanze invalide sono istanze che eseguono il thorw di eccezioni unchecked quando vengono invocati i loro metodi.

Per il metodo attach si avranno anche i parametri:

- Object toAttach: {NULL}, {Valid Object}.
  Qui non c'è una classe {Invalid Object}, poiché si ritiene che questa "non esista", o al più coincida con la classe {NULL}. Infatti, il metodo attach non "usa" l'oggetto toAttach, ma fa solo in modo che si tenga traccia delle modifiche effettuate a questo oggetto per poi rifletterle sullo stato di persistenza (questo compito in realtà poi è in buona parte delegato alla classe AttachStrategy, che non è parte del SUT e sarà mockata). Dunque mentre null differisce dall'istanza valida poiché non è propriamente un oggetto (come riporta anche la documentazione di Java), tra un oggetto valido e uno che lancia RuntimeException a ogni chiamata dei suoi metodi (che potrebbe quindi sembrare un oggetto non valido), in questo ambito non c'è differenza, visti gli scopi del metodo. Dunque l'unico tipo di istanza che potrebbe essere invalido è l'istanza null.

- PersistanceCapable, OpenJPAStateManager, ValueMetaData: gli oggetti di queste tre classi vengono trattati insieme poiché hanno tutti la caratteristica di essere solamente passati a metodi di altre classi che non fanno parte del SUT. Quindi, siccome questo è uno Unit Test, le classi che non fanno parte del SUT vengono mockate, e quindi passare a un mock con un comportamento predefinito delle istanze non valide è sembrato poco utile. Per esempio, il test sul comportamento della classe AttachStrategy con un PersistanceCapable invalido, andrà testato quando si fa Unit Testing di quella classe, e non di AttachManager.
  Quindi per queste classi si sono scelte solo le classi {NULL}, {Valid}.
  Le istanze valide sono sempre state realizzate tramite mock.
- boolean explicit: {true}, {false}.

Il metodo attachAll ha come unico parametro:

- Collection instances: {NULL}, {empty collection}, {valid collection}, {totally invalid collection}, {partially invalid collection}. Le ultime due classi hanno la differenza che, se per esempio contengono tre elementi, nella ”totally invalid” nessuno dei tre elementi sarà un oggetto di cui si può fare l'attach, mentre nella ”partially invalid” ce ne sarà almeno uno di cui l'attach può andare a buon fine.

Boundary Values

- broker: {null}, {mock che ritorna oggetti e valori richiesti dai metodi, senza sollevare eccezioni}, {mock che lancia unchecked exception quando si invocano i suoi metodi}
- newCopy: {true}, {false}
- callBack: come broker
- List toAttach: {null}, {Collections.emptyCollection()}, {[null]}, {[”foo”]}, {[”foo”, ”bar”]}.
  Per semplicità, come oggetti della lista usiamo delle String.
- Object toAttach: {null}, {”foo”}
- PersistanceCapable, OpenJPAStateManager, ValueMetaData: {null}, {mock dell'istanza}
- explicit: {true}, {false}
- instances: {null}, {[]}, {[”foo”, ”bar”, ”test”]}. La terza lista è usata per tutti i casi di collection valida, interamente non valida e parzialmente non valida, dato che a definire se un oggetto può essere *attached* oppure no non è l'oggetto stesso, bensì il suo StateManager. Si avranno quindi due diversi mock dello state manager, uno che consente l'attach per quell'oggetto, e l'altro che lo impedisce. In base al caso che si vorrà simulare si farà il return di uno o dell'altro StateManager tramite Mockito.

Casi di Test In AttachManagerTest e AttachManagerAttachParamTest si è scelto di avere un approccio multi-dimensionale. Si è fatta questa scelta per poter acquisire confidenza sul fatto che, per ogni tipologia di oggetto toAttach, tutte le combinazioni di parametri si comportassero sempre in maniera prevedibile, a prescindere dal tipo di oggetto che la classe stava gestendo in quel momento.

Inoltre, nel test del costruttore, per avere maggior confidenza sul fatto che un'istanza ritornata fosse valida, si è voluto testare anche che quell'istanza fosse in grado gestire come ci si aspetterebbe l'attach di vari tipi di oggetto (null, lista vuota, lista con un oggetto, lista con più oggetti), e non semplicemente che ”superasse” il costruttore senza lanciare eccezioni. Infine, si è controllato che un'istanza da considerare valida si comportasse correttamente rispetto al meccanismo di set e get della attachedCopy. Si è poi voluta avere l'accortezza di non andare a lanciare test dove ci fosse più di un parametro che potesse generare un'eccezione. Questo per rendere i test il più prevedibili possibile, senza dover eseguire approfondite analisi white-box per capire, in caso di più parametri scorretti, chi avrebbe generato per primo l'eccezione, in modo da poter gestire correttamente l'expected value. La strategia adottata per realizzare questo meccanismo, è stata quella di generare tutti le possibili combinazioni di parametri tramite dei cicli for annidati, e poi scorrere la lista di Object[], rimuovendo quegli array che contenessero più si un valore scorretto. Qui (tabella 7) la lista dei casi di test per AttachManagerTest, qui (tabella 8) quella per AttachManagerAttachParamTest. Infine, nella tabella 9 i casi di test per AttachManagerAttachAllTest. La tabella per AttachManagerAttachTest non è mostrata poiché in realtà utilizza sempre gli stessi input, ma varia test per test il comportamento degli altri oggetti/mock, per riuscire a entrare e testare in un numero maggiore di branch. Qui il codice della classe. I commenti, abbinati al codice, spiegano il modo in cui si svolgono i vari test.

Commenti Nessuno dei test effettuatati ha riscontrato comportamenti inattesi o imprevisti della classe AttachManager. La classe presenta una buona suddivisione della logica (solo il metodo attachAll ha una complessità cognitiva > 15 (20)), e dunque da questo punto di vista non è stata particolarmente complicata da testare. Una dei fattori che invece complicano leggermente la fase di test è il fatto che la classe AttachManager faccia ampio uso di altri oggetti del progetto OpenJPA, quindi per gli Unit test si deve fare un uso esteso di mock. Questo ha in parte portato anche a un'analisi white-box dei metodi, per comprendere meglio come interagissero i vari oggetti, per poterli poi mockare correttamente.

Quality Assurance Con i test realizzati fino a questo momento JaCoCo riporta una line coverage di 69% e una branch coverage del 58%, mentre ba-dua riporta una defintion-use coverage dell'87% per il metodo attach() di riga 226, e il 100% di coverage per il metodo attachAll(). Per quanto riguarda il metodo attach() da entrambi i report viene evidenziata la mancata coverage del branch a riga 241. Infatti non c'è un flusso di esecuzione che entra in quel branch, come riporta JaCoCo, e di conseguenza ba-dua mostra come manchi coverage per le coppie DU che hanno uno use a riga 241, o che lo hanno a riga 240, ma con target riga 241. Si decide pertanto di introdurre un nuovo metodo di test visitedNodeContainsTest, strutturato in modo da far risultare che l'oggetto di cui stiamo eseguendo l'attach, fosse già stato visitato durante una attach con cascade per qualche altro oggetto. Con questo nuovo test, sia JaCoCo che ba-dua riportano una coverage del 100% per questo metodo. Con la test suite così definita, sia il costruttore che il metodo attach eliminano tutte le mutazioni generate da PIT, e dunque, per questi test si ritiene raggiunto un livello di qualità adeguato. Per quanto riguarda invece il metodo attachAll la situazione è diversa. Sebbene JaCoCo e ba-dua riportino coverage abbastanza alte (instruction coverage: 93%, branch coverage: 85%, DU-coverage: 100%), le mutazioni eliminate con PIT sono solamente il 47%. Si introduce quindi, nella classe AttachManagerAttachAllTest il metodo checkInvokefterException, che esegue controlli più stringenti sull'effettiva invocazione delle operazioni post attach, nello scenario in cui si entra nel branch di riga 155. Siccome l'effettiva invocazione delle chiamate post attach è un LifecycleEvent, gestito dal LifecycleEventManager, una classe che al momento non fa parte del SUT, utilizziamo un mock. Infatti, tramite l'oggetto di tipo BrokerImpl (che è un mock) con cui è istanziato l'AttachManager che si sta usando, si aggiunge un ulteriore comportamento per cui, ogni volta che gli verrebbe richiesto di richiedere al LifecycleEventManager di eseguire una chiamata post attach, questo incrementa semplicemente un contatore. All'inzio del test si inizializza il contatore a zero, e nella assert finale si controlla che sia stato incrementato esattamente una volta. Con questo ulteriore test si riesce ad eliminare una mutazione in più, portando la percentuale di mutazioni eliminate al 53%. Questa percentuale non è comunque molto elevata, ma c'è da notare che il metodo attachAll, come segnalato da SonarCloud è quello con la complessità cognitiva più alta, e l'unico a sforare il limite imposto da SonarCloud (complessità pari a 20, contro i 15 consentiti). Questo rende il metodo più complicato da testare, ed avrebbe richiesto ulteriore tempo per poter raffinare ulteriormente i test in modo da eliminare più mutazioni.

A livello di classe invece, con JaCoCo risulta una instruction coverage del 75%, e una mutation coverage del 61%, che sale al 78.5% se non si considerano le mutazioni generate in porzioni di codice senza coverage.

Tutti i report discussi sono disponibili nella sezione Actions della repository GitHub, come artifact dei relativi workflow.

2.3.2 CacheMap

{Classi CacheMapTest, CacheMapPutTest, CacheMapMaxDimensionTest, CacheMapLockTest}

CacheMap è una classe che ha il compito di mantenere in memoria vari oggetti (principalmente PreparedQuery) in modo da rendere più veloce la loro esecuzione qualora questa venisse richiesta più volte. Per farlo al suo interno ha tre diverse Map:

- cacheMap: la vera e proprio cache
- softMap: per gli oggetti per cui non c'è più spazio nella cache principale
- pinnedMap: per quegli oggetti che si vuole che rimangano sempre in cache, e che non possano essere scelti come oggetti da rimuovere quando bisogna fare spazio in cache.

Category Partition Il costruttore della CacheMap ha i seguenti parametri:

- boolean lru: {true}, {false}.
  Il suo scopo è quello di selezionare o meno una Map di tipo LRU (Least Recently Used), ovvero che rimuove dalla cache l'elemento che non viene usato da più tempo quando deve fare posto. Se settato su false invece usa una Map che sceglie l'elemento da rimuovere in modo pseudo-random.
- int max: {< 0}, {= 0}, {> 0}.
  Rappresenta la dimensione massima della cacheMap.
- int size: {< 0}, {= 0}, {> 0}.
  Rappresenta la dimensione iniziale della cacheMap.
- float load: {< 0.0}, {= 0.0}, {0 < load $\leq$ 1.0}, {> 1.0}.
  Rappresenta il valore dopo il quale (se size < max) la cacheMap viene allargata. Qui includiamo anche la partizione dove il load factor è maggiore di 1, poiché il load è il rapporto tra la dimensione totale della cache, e la memoria attualmente occupata, e quindi un valore > 1.0 è diverso da uno $\leq$ 1.0.
- int concurrencyLevel: {< 0}, {= 0}, {> 0}.
  Nota: come si evince semplicemente guardando il metodo, ma anche tramite i code smell segnalati su SonarCloud, questo parametro non è mai utilizzato all'interno del metodo. In seconda battuta si nota anche che nel

report di ba-dua non compare mai nessuna riga relativa a questo parametro per la definition-use coverage, dal momento che ha sì una definition, ma nessuno usage. Si è però scelto comunque di trattare questo parametro al pari di tutti gli altri. Questo infatti è stato fatto pensando al caso in cui, in futuro, l'implementazione di questo metodo possa cambiare, e questo parametro possa diventare parte integrante della logica del costruttore di CacheMap. In questo scenario, il test si rivelerebbe ancora efficace nel catturare eventuali errori, anche se legati a questo parametro, dal momento che è stato considerato nei vari casi di test. Al contrario, ignorando questo parametro, se un giorno l'implementazione del metodo dovesse cambiare e magari introdurre un bug proprio dovuto a un errato utilizzo di concurrencyLevel, il test non lo catturerebbe, facendo passare la build anche in presenza di un bug nel metodo che va a testare. Quindi, con l'idea di realizzare i casi di test con un approccio black-box, si è scelto di fare uso anche questo parametro. Chiaramente però, questo ha portato delle problematiche nella gestione degli expected values relativi a concurrencyLevel: infatti, al momento, un suo valore scorretto non porta nessuna failure, non venendo utilizzato. Qui il commento che spiega nel dettaglio come è gestito l'expected value per questo parametro.

Boundary Values La scelta dei boundary value è immediata per tutti i parametri, essendo questi tipi primitivi

- boolean lru: true, false
- int max: -1, 0, 1
- int size: -1, 0, 1
- float load: -0.1, 0.0, 0.1, 1.1
- int concurrencyLevel: -1, 0, 1

Casi di Test Per questa classe si è scelto un approccio multi-dimensionale, perché si è voluta acquisire confidenza sul suo corretto funzionamento in tutte le possibili situazioni, e ci si è voluto assicurare che la classe si comportasse allo stesso modo sia nel caso in cui la cacheMap fosse di tipo LRU, sia in caso contrario. Qui la (lunga) tabella con tutti i casi di test, realizzata tramite uno script. Per il discorso già affrontato in precedenza, è stato incluso nell'approccio multi-dimensionale anche il parametro concurrencyLevel. La classe CacheMapTest è quella in cui si è testata la costruzione della CacheMap, di cui si parlerà nel dettaglio più avanti. Nella classe CacheMapMaxDimensionTest invece ci si è voluti assicurare che la CacheMap si comportasse correttamente nei casi in cui venga istanziata una CacheMap con valori non ovviamente scorretti (quindi maggiori di 0), ma incompatibili tra loro. Per esempio una CacheMap con size iniziale 3 e size massima 1. In questo test dunque ci si è voluti assicurare che, per una cacheMap così costruita, fosse la siza massima a "dominare", poiché si ritiene che la size massima della cache si una condizione più stringente, dato che è immutabile, della size iniziale, che invece poi potrebbe crescere. Nella classe CacheMapPutTest si è testato il comportamento del metodo put(...) di CacheMap, e si è voluta acquisire confidenza sul corretto passaggio degli oggetti tra la cacheMap e la softMap. Si è anche testata la gestione degli elementi nella pinnedMap. *{Nota: per CacheMap si intende la classe, per cacheMap invece la cache primaria della CacheMap, quella descritta qui.}* Infine, nella classe CacheMapLockTest, tramite l'utilizzo di thread concorrenti, si è andati a testare il funzionamento dei lock() della cacheMap, assicurandosi che:

- Fossero possibili letture concorrenti
- Durante un writeLock non si potesse prendere un readLock finchè la scrittura non fosse terminata
- Se vengono richiesti più writeLock mentre c'è in corso un'altra scrittura, si vuole che poi la CacheMap rilasci il writeLock ai thread che erano in attesa, nell'ordine in cui questi sono arrivati a richiederlo.

Per quanto riguarda invece il costruttore di CacheMap, si è riscontrato un possibile bug. Supponiamo di essere nella situazione in cui abbiamo i seguenti parametri:

- int max > 0
- int size = 0
- 0 < float load $\leq$ 1 (un generico valore valido)

Quello che ci si aspetterebbe invocando il costruttore di CacheMap, è di ritrovarsi un'istanza valida della classe in cui la cacheMap abbia size attuale zero, e size massima > 0. Questo è infatti ciò che succede effettivamente quando la chiamata al costruttore di CacheMap avviene con il boolean lru = false. Se invece si richiede che la CacheMap sia di tipo LRU, quello che succede è che si genera un'eccezione di tipo IllegalArgumentException con il seguente messaggio:

"java.lang.IllegalArgumentException: LRUMap max size must be greater than 0"

Il che in un certo senso è assurdo, dato che il parametro max è stato definito proprio > 0, e dovrebbe quindi rispettare il vincolo imposto dalla LRUMap. Si prosegue dunque ad un'analisi white-box, per capire l'origine di questa incongruenza. Partiamo notando che la LRUMap viene istanziata tramite la chiamata a riga 140 di CacheMap (questo il codice).

Quindi sembra che si stia chiedendo una LRUMap con dimensione iniziale pari a "size". Andando pero a guardare il costruttore di LRUMap che si sta chiamando in CacheMap, si nota che è quello di LRUMap del modulo openjpa-lib\util. Il costruttore ha la forma qui mostrata.

Il *super* che si va a chiamare è quello di LRUMap del modulo openjpa-lib\collections, che ha invece questa forma. Si nota subito dunque, che il primo parametro si chiami "maxSize", mentre è stato invocato con il parametro initSize dalla sua sottoclasse. Alla fine, il costruttore che viene effettivamente chiamato in collections\LRUMap è questo, e in questo caso specifico i parametri che gli arrivano sono (size, size, load, false). Di fatto abbiamo richiesto da CacheMap la creazione di una LRUMap con initalSize = size, ma abbiamo ottenuto una lruMap che ha (anche) maxSize = size. Quindi il costruttore della LRUMap lancia giustamente un'eccezione, dato una cache LRU con size massima 0 ha poco senso di esistere. Il problema quindi non sta nel costruttore della LRUMap, ma nel modo in cui questo viene invocato. Infatti, quando si chiede a CacheMap una cache LRU con size iniziale 0 e size massima > 0, questa sta di fatto provando a costruirne una con size massima = 0.
{Qui un breve recap di tutte le chiamate nell'ordine in cui avvengono, per rendere più chiaro il flusso di esecuzione.}
Sorge dunque il dubbio che in cacheMap la LRU vada creata passando al suo costruttore il parametro max, e non size, con riga 140 che diventerebbe:

```
1 cacheMap = new LRUMap(max, load)
2 //invece che
3 cacheMap = new LRUMap(size, load)
```

e dovrebbe esserci, a questo punto, anche un refactor del costruttore della sottoclasse LRUMap del modulo openjpa-lib\util, con il primo parametro chiamato maxSize e non initSize.

Non è possibile avere la certezza che questo sia un bug, ma certamente è un comportamento che da un'analisi black-box risulta difficile da prevedere e contro intuitivo. Così come è molto contro intuitivo ricevere un messaggio d'errore che richiede una size massima > 0, quando il parametro max è effettivamente settato a un valore > 0. Da un'analisi white-box sembra effettivamente poterci essere un "typo" dove si usa la initSize per qualcosa che poi diventerà la maxSize, nonostante si abbia a disposizione un parametro che rappresenta la maxSize. Come inoltre si dimostra con il metodo showThatLRUShouldBePossible() , con gli stessi parametri con cui CacheMap lancia una IllegalArgumentException, è in realtà possibile costruire una LRUMap che non solleva questa eccezione.

In quel metodo si dimostra che la costruzione di una LRUMap è possibile sia usando il costruttore di collections\LRUMap (quello "vero e proprio") con initSize = size e maxSize = max, sia con entrambe le size = max (perché è così che verrebbe chiamato effettuando la modifica a riga 140 del costruttore di CacheMap)).

Inoltre, si dimostra direttamente anche che, se a riga 140 di CacheMap si chiamasse LRUMap(max, load) invece che LRUMap(size, load), la LRUMap verrebbe istanziata senza produrre alcuna IllegalArgumentException.

Commenti Per quanto riguarda questo possibile bug, tornando alla revision in cui la CacheMap è stata introdotta, si nota come la riga che invoca la creazione della LRUMap fosse già uguale a quella che si trova nella versione attuale di openJPA. In quel momento però non c'erano tutte le sottoclassi che estendono LRUMap che sono presenti adesso, ed il costruttore che veniva invocato prendeva come prametri:

- int maxSize
- float loadFactor

E come maxSize viene passato, appunto, size e non max. Dunque già qui sembra esserci un'incongruenza, poiché il parametro size, come già detto rappresenta la size iniziale e non quella massima (che corrisponde invece al parametro "max"). Se questo si rivelasse effettivamente un bug, sarebbe presente dalla prima introduzione di CacheMap dunque, che risale al 2006. Da un lato quindi sarebbe strano che un bug fosse dormiente così a lungo senza portare ad alcuna failure, dall'altro è pur vero che quel costruttore non viene mai invocato, al momento, all'interno del codice con i parametri che rappresentano questo caso limite. Va anche detto però, che questo possibile bug risulta più evidente con i parametri del caso limite discusso in precedenza, ma potenzialmente c'è comunque un comportamento indesiderato, anche se molto meno evidente:
Supponendo di avere i seguenti parametri:

- int $N \gg 0$
- int max = N
- int size = N/2

- 0 < load $\leq$ 1
- lru = true

Se invocassimo il costruttore di CacheMap con questi parametri non si genererebbe nessuna eccezione per la LRUMap, dato che size sarebbe comunque > 0. Però otterremmo una CacheMap con associata una LRUMap di size massima N/2.
Con gli stessi parametri, ma con lru = false invece otterremmo una CacheMap con associata una ConcurrentHashMap di size iniziale N/2, e size massima pari a N. Dunque ci sarebbe comunque un comportamento indesiderato, seppur molto meno evidente, perché nel caso con cache LRU, si ottiene una cache che, una volta raggiunta la sua size massima, sarà più piccola di quanto lo sarebbe stata la corrispettiva cache di tipo ConcurrentHashMap.

Nota: il catch di riga 243 è stato inserito per consentire il completamento con successo della build, e controlla che se quell'assert non è soddisfatto è solo e esclusivamente per la situazione descritta fin'ora.
In conclusione, si prova a stabilire la *reliability* del costuttore di CacheMap, considerando il profilo operazionale che coincide con gli input dei casi di test. Si hanno un totale di 216 possibili combinazioni di input. Quelle che rispettano i vincoli discussi in precedenza e che portano alla generazione del possibile bug sono solamente 3. Dunque, assumendo che tutti gli input di questo profilo operazionale siano equiprobabili, si otterrebbe una probabilità di utilizzo per ogni input dello 0.46%. Quindi la probabilità totale di scegliere un input che provoca una failure risulterebbe essere dell' 1.38%. Eseguendo quindi sempre il calcolo per la reliability già visto nelle precedenti occasioni si otterrebbe: $Reliability = 1 - P(failure) = 1 - 0.0138 = 0.9862$ e quindi una reliability del 98.62%.

Quality Assurance Per la classe CacheMap, JaCoCo riporta una line coverage dell'80% e una brach coverage del 67%. Ba-dua riporta una du-coverage del 100% per il costruttore di CacheMap e per il metodo put(..) della classe. Per il metodo get invece ba-dua riporta una coverage di 20 su 24 DU (83% circa). In realtà però, andando ad analizzare nel dettaglio i 4 DU con coverage = 0 (questi), ci si rende conto che possono essere considerati irraggiungibili:

- Il primo "richiede" che la variabile putcache definita a riga 360 venga usata a riga 378 con un esecuzione che abbia come target riga 379 (quindi che entri nell'if). Questo però non è possibile, dato che putcache a riga 360 è definito come false, quindi il uso utilizzo come condizione dell'if a riga 378, non porterà mai ad entrare nel corpo dell'if e quindi a eseguire riga 379. L'unica definizione di putcache che può avere quello use e quel target è quella di riga 371, che infatti risulta coperta.
- Il secondo è il caso in cui la variabile val, definita a riga 366 viene usata a riga 379. Questo non è possibile, dal momento che se l'esecuzione arriva a riga 366, non entrerà mai nell'else che imposta putcache a true, e quindi non potrà mai entrare nel corpo dell'if di riga 379.
- Per il terzo DU il discorso è del tutto analogo a quello del secondo DU.
- Il quarto DU che non ha coverage, riguarda la definition di putcache a riga 371, il suo utilizzo a riga 378 e un flusso di esecuzione che dopo l'utilizzo vada a riga 373 (sostanzialmente il flusso non deve entrare nel corpo dell'if di riga 379). Questo non è possibile, dal momento che la definizione di putcache a riga 371 è "putcache = true", ed è proprio quella che fa entrare il flusso d'esecuzione nel corpo dell'if a riga 379.

Quindi di fatto, risulta impossibile estendere ulteriormente la coverage di ba-dua. Infine, per la classe CacheMap PIT riporta una percentuale di mutazioni eliminate (a livello di classe) del 58%, che sale al 67.5% considerando solo le mutazioni che cadono in righe con coverage. Per migliorare la qualità dei test (e quindi eliminare più mutazioni) il punto fondamentale è stato quello di eseguire in tutti i test controlli sulla corretta gestione dei lock. Infatti, tramite la classe LockChecker , è stato implementato un meccanismo che controllasse il corretto uso dei lock da parte del metodo testato, anche se durante il test non vi fosse stata concorrenza. L'impiego di questo meccanismo ha permesso di aumentare la coverage di PIT, e di rendere i test più robusti, perché in questo modo, in tutti i metodi dove viene richiesto un lock, il test controlla che questi vengano gestiti nella maniera corretta da parte del SUT.

2.4 IntegrationTest

Per OpenJPA sono state realizzate le classi di Integration Test *PreparedQueryCacheIT* e *QueryImplIT*.

- In **PreparedQueryCacheIT**, il SUT è stato individuato nelle classi CacheMap e PreparedQueryCacheImpl. Quest'ultima classe è infatti il gestore della cache delle PreparedQuery per il modulo jdbc di OpenJPA. Questa classe realizza la logica che decide se una determinata prepared query vada *cachata* oppure no, e a quel punto delega a CacheMap il compito di mantenere in memoria la prepared query. Inoltre, utilizza due diverse CacheMap, una per la cache "vera e propria", e l'altra per tenere traccia di quelle prepared query che risultano non cachabili, in modo da non dover ripetere tutti i controlli ogni volta. In questo Integration Test si vuole quindi andare a testare l'interazione tra queste due classi, controllando che lo scambio di messaggi avvenga correttamente, e che anche lo scambio di dati avvenga seguendo i protocolli prestabiliti. Per fare ciò si costruisce una Query, e tramite la classe PreparedQueryCacheImpl se ne chiede la registrazione in cache. A quel punto si va a controllare lo

stato delle due CacheMap di PreparedQueryImpl, controllando che quella della cache contenga effettivamente la nostra query, mentre quella per gli oggetti non cachabili sia vuota.
Poi si ripete un procedimento simile, e dopo che la query è stata cachata si procede a segnalarla come uncachable tramite PrearedQueryCacheImpl. A quel punto ci si assciura che, la query che prima si trovava in cache, ora si trovi nella CacheMap relativa alle query uncachable.

- **QueryImplIT** invece, non è altro che un estensione del test precedente secondo una logica bottom-up. Infatti quello che si fa è estendere il SUT anche alla classe QueryImpl. A questa si associa l'istanza di PreparedQueryCacheImpl, che a sua volta avrà associate le due CacheMap. Dopodiché si procede a richiedere a QueryImpl di compilare ed eseguire la query. Durante l'esecuzione della query, QueryImpl procede anche a richiedere a PreparedQueryCacheImpl di inserire in cache la query. Questa richiesta poi verrà elaborata in maniera simile a quanto descritto per l'altro Integration Test. Per l'interazione tra queste tre classi, avendo settato che la cache potesse contenere solo una query per volta, sono stati eseguiti, nell'ordine, i seguenti test;
  - Inserimento con successo di una query in cache
  - Inserimento con successo di una seconda query in cache, controllando che la prima venisse spostata in softCache, e che la seconda prendesse il suo posto nella cache principale.
  - Richiesta di esecuzione di una terza query, che però fosse uncachable. Alla fine dell'esecuzione si è controllato che questa terza query non fosse in cache, che invece si trovasse nella CacheMap delle uncachable query, e che le precedenti query non fossero state rimosse dalla cache.

Per realizzare questi Integration Test si è fatto largo uso di Mockito (e dove necessario anche di PowerMock), per mockare quegli oggetti che non facevano parte del SUT, e per ottenere il flusso di esecuzione desiderato (query cachabile oppure no). Inoltre, per associare a PreparedQueryCacheImpl delle CacheMap istanziate dal test (in modo che poi questo potesse accedervi per controllarne il contenuto) si è fatto uso della reflection di Java. Questo è stato necessario per poter avere un maggiore controllo sulle CacheMap, dato che PreparedQueryCacheImpl le istanzia autonomamente, ed essendo poi oggetti privati per la classe, il test non vi avrebbe potuto avere accesso.

3 Tabelle e Codice

address	port	interface	hostNameAsBookie	shortHostName	allowLoopback	Expected Value
""	1	NULL	false	false	true	PASSED
""	1	"notAnInterface"	false	false	true	UnknownHostException
""	0	"enp0s9"	true	false	true	PASSED
""	0	"enp0s9"	false	false	false	UnknownHostException
""	1	""	false	false	true	UnknownHostException
"192.168.56.102"	1	"enp0s9"	true	true	true	PASSED
"192.168.56.102"	1	"enp0s9"	true	true	false	PASSED
NULL	1	"enp0s9"	false	false	true	PASSED
""	-1	"enp0s9"	true	false	true	IllegalArgumentException
""	65536	"enp0s9"	true	false	true	IllegalArgumentException
"300.568.1.2"	1	"enp0s9"	false	false	true	PASSED

Table 1: Casi di test per BookieImpl.*getBookieAddress()*. Il nome di alcuni attributi è stato abbreviato per migliorare la leggibilità della tabella.

Listing 1: costruttore di BookieImpl

```
public BookieImpl(ServerConfiguration conf,
                  RegistrationManager registrationManager,
                  LedgerStorage storage,
                  DiskChecker diskChecker,
                  LedgerDirsManager ledgerDirsManager,
                  LedgerDirsManager indexDirsManager,
                  StatsLogger statsLogger,
                  ByteBufAllocator allocator,
                  Supplier<BookieServiceInfo> bookieServiceInfoProvider)
          throws IOException, InterruptedException, BookieException
```

Listing 2: Frammento del report ba-dua per il metodo format(...) di BookieImpl

```
<du covered="0" def="1178" target="1189" use="1184" var="isInteractive"/>
```

journalDirs	ledgerDirs	indexDirs	metadataPath	isInteractive	force	expected
VALID	VALID	VALID	VALID	false	true	true
NULL	NULL	NULL	NULL	false	true	true
INVALID	VALID	VALID	VALID	false	true	false
VALID	INVALID	VALID	VALID	false	true	false
VALID	VALID	INVALID	VALID	false	true	false
VALID	VALID	VALID	INVALID	false	true	false
INVALID	VALID	VALID	VALID	false	false	false
INVALID	VALID	VALID	VALID	false	true	false
NOT_EX	NOT_EX	NOT_EX	NOT_EX	false	true	true
EMPTY	VALID	VALID	VALID	false	true	true
FILE	VALID	VALID	VALID	false	true	true
VALID	VALID	VALID	VALID	true(true)	false	true
VALID	VALID	VALID	VALID	true(false)	false	false
VALID	VALID	VALID	VALID	true(exception)	false	false

Table 2: Casi di test per BookieImpl.*format()*. ”NOT_EX” sta per ”Not existing”, e si riferisce a quei path validi, ma il cui relativo file/directory non esiste. EMPTY rappresenta una cartella valida ed esistente, ma vuota, mentre FILE rappresenta un path valido ed esistente, che però punta ad un file, e non a una directory. Dove isInteractive = true, è riportato tra parentesi il valore ritornato dal mock che sostituisce l’interattività.

Listing 3: Gestione dei null da parte di ServerConfiguration

```
1
2  ServerConfiguration configuration = new ServerConfiguration();
3         String[] jDirs = extractFileNames(journalList); //genera folder valide e ne estrae i nomi
4         jDirs[0] = null; //null sulla prima cartella
5         configuration.setJournalDirsName(jDirs);
6         System.out.println(Arrays.toString(configuration.getJournalDirNames()));
7         System.out.println(Arrays.toString(jDirs));
8         configuration.setJournalDirsName(null); //null a tutto l'oggetto journalDirs
9         System.out.println(Arrays.toString(configuration.getJournalDirNames()));
10
11 ------------------------------ OUTPUT ------------------------------
12 [temp-folder-journal3, temp-folder-journal6, temp-folder-journal9]
13 [null, temp-folder-journal3, temp-folder-journal6, temp-folder-journal9]
14 [/tmp/bk-txn]
15 ---------------------------- END OUTPUT ----------------------------
16
17 //Vediamo che stampando le cartelle "filtrate" da ServerConfiguration quella null viene eliminata.
18 //Nel caso in cui tutto l'oggetto sia null, ServerConfiguratin setta il path della cartella predefinita per i journal.
```

LedgerStorage	UsageThreshold	WarnThreshold	ExpectedValue
SORTED	-1.0	-1.1	ILLEGAL_ARGUMENT
SORTED	-1.0	-1.0	ILLEGAL_ARGUMENT
SORTED	-1.0	0.0	ILLEGAL_ARGUMENT
SORTED	0.0	-1.0	ILLEGAL_ARGUMENT
SORTED	0.0	0.0	ILLEGAL_ARGUMENT
SORTED	0.0	1.0	ILLEGAL_ARGUMENT
SORTED	0.1	0.0	PASSED
INTERLEAVED	0.1	0.1	PASSED
INTERLEAVED	0.1	0.2	ILLEGAL_ARGUMENT
INTERLEAVED	1.0	0.9	ILLEGAL_ARGUMENT
INTERLEAVED	1.0	1.0	ILLEGAL_ARGUMENT
INTERLEAVED	1.0	1.1	ILLEGAL_ARGUMENT
DB	0.9	0.8	PASSED
NULL	0.9	0.8	PASSED

Table 3: Casi di test per BookieImpl.*mountLedgerStorageOffline()*. Le ultime due righe sono quelle aggiunte a seguito dell'estrazione di un ulteriore boundary value, data la necessità di aggiungere delle combinazioni di parametri valide in più.

Listing 4: Coupling tra i parametri del LedgerStorage e quelli di BookieImpl

```
1
2  ledgerStorage.initialize(conf, new NullMetadataBookieDriver.NullLedgerManager(), ledgerDirsManager,
       indexDirsManager,
3                  statsLogger, byteBufAllocator);
4
5
6             bookieImpl = new BookieImpl(
7                    conf,
8                    reg,
9                    ledgerStorage,
10                   diskChecker,
11                   ledgerDirsManager,
12                   indexDirsManager,
13                   statsLogger,
14                   byteBufAllocator,
15                   bookieServiceInfo
16                   );
```

Listing 5: DiskChecker scorretto

```
public InvalidDiskChecker(float threshold, float warnThreshold) {
        super(threshold, warnThreshold);
    }

    @Override
    public long getTotalFreeSpace(List<File> dirs) {
        return -1L;
    }

    @Override
    public long getTotalDiskSpace(List<File> dirs) {
        return -1L;
    }

    @Override
    public float getTotalDiskUsage(List<File> dirs) {
        return -1.1f;
    }

    @Override
    public float checkDir(File dir) throws DiskErrorException {
        throw new DiskErrorException("Eccezione generata da InvalidRegistrationManager");
    }
```

port	jourDirs	ledgDirs	flushInt	logPerLedger	regManager	ledgStorage	diskCheck	ledgDirsManager	indexDirs	indexDirsManager	statsLogger	byteBufAllocator	bookieService	expectedValue
1	VALID	VALID	1000	true	NullRegistrationManager	SORTED	VALID	VALID	VALID	VALID	NullStatsLogger	Unpooled	VALID	PASSED
0	VALID	VALID	1000	true	NullRegistrationManager	SORTED	VALID	VALID	VALID	VALID	NullStatsLogger	Unpooled	VALID	PASSED
1	VALID	VALID	1000	false	null	SORTED	VALID	VALID	VALID	VALID	NullStatsLogger	Pooled	VALID	PASSED
1	Empty	Empty	1000	true	NullRegistrationManager	SORTED	VALID	VALID	Empty	VALID	NullStatsLogger	Unpooled	VALID	NO SPACE
1	VALID	VALID	1000	true	NullRegistrationManager	INTERLEAVED	VALID	VALID	VALID	VALID	NullStatsLogger	Unpooled	VALID	PASSED
1	VALID	VALID	1000	true	NullRegistrationManager	DB	VALID	VALID	VALID	VALID	NullStatsLogger	Unpooled	VALID	PASSED
1	VALID	VALID	1000	true	null	DB	VALID	VALID	VALID	VALID	NullStatsLogger	Unpooled	VALID	PASSED
-1	VALID	VALID	1000	true	NullRegistrationManager	SORTED	VALID	VALID	VALID	VALID	NullStatsLogger	Unpooled	VALID	IllegalArgument
1	INVALID	VALID	1000	true	NullRegistrationManager	SORTED	VALID	VALID	VALID	VALID	NullStatsLogger	Unpooled	VALID	IOExcep
1	VALID	INVALID	1000	true	NullRegistrationManager	SORTED	VALID	VALID	VALID	VALID	NullStatsLogger	Unpooled	VALID	IOExcep
1	VALID	VALID	0	true	NullRegistrationManager	SORTED	VALID	VALID	VALID	VALID	NullStatsLogger	Unpooled	VALID	IllegalArgument
1	VALID	VALID	-1	true	NullRegistrationManager	SORTED	VALID	VALID	VALID	VALID	NullStatsLogger	Unpooled	VALID	IllegalArgument
1	VALID	VALID	1000	true	InvalidRegistrationManager	SORTED	VALID	VALID	VALID	VALID	NullStatsLogger	Unpooled	VALID	Except
1	VALID	VALID	1000	true	NullRegistrationManager	null	VALID	VALID	VALID	VALID	NullStatsLogger	Unpooled	VALID	NullPointer
1	VALID	VALID	1000	true	NullRegistrationManager	SORTED	null	VALID	VALID	VALID	NullStatsLogger	Unpooled	VALID	NullPointer
1	VALID	VALID	1000	true	NullRegistrationManager	SORTED	InvalidDiskChecker	VALID	VALID	VALID	NullStatsLogger	Unpooled	VALID	DiskExcep
1	VALID	VALID	1000	true	NullRegistrationManager	SORTED	VALID	null	VALID	VALID	NullStatsLogger	Unpooled	VALID	NullPointer
1	VALID	VALID	1000	true	NullRegistrationManager	SORTED	VALID	INVALID	VALID	VALID	NullStatsLogger	Unpooled	VALID	IOExcep
1	VALID	VALID	1000	true	NullRegistrationManager	SORTED	VALID	VALID	INVALID	VALID	NullStatsLogger	Unpooled	VALID	IOExcep
1	VALID	VALID	1000	true	NullRegistrationManager	SORTED	VALID	VALID	VALID	VALID	null	Unpooled	VALID	NullPointer
1	VALID	VALID	1000	true	NullRegistrationManager	SORTED	VALID	VALID	VALID	VALID	InvalidStatsLogger	Unpooled	VALID	Excep
1	VALID	VALID	1000	true	NullRegistrationManager	SORTED	VALID	VALID	VALID	null	NullStatsLogger	Unpooled	VALID	NullPointer
1	VALID	VALID	1000	true	NullRegistrationManager	SORTED	VALID	VALID	VALID	INVALID	NullStatsLogger	Unpooled	VALID	Excep
1	VALID	VALID	1000	true	NullRegistrationManager	SORTED	VALID	VALID	VALID	VALID	NullStatsLogger	Unpooled	null	NullPointer
65536	VALID	VALID	1000	true	NullRegistrationManager	SORTED	VALID	VALID	VALID	VALID	NullStatsLogger	Unpooled	VALID	IllegalArgument

Table 4: Nomi attributi accorciati per leggibilità. I test di *BookieImplMockTest* sono tutti realizzati con la configurazione della prima riga, esplorando branch che non avevano coverage tramite l'impiego di mock

journalToScan	journalPos	scanner	Expected
0	-1	new DummyJournalScan()	EXACT_BYTES
0	0	new DummyJournalScan()	EXACT_BYTES
0	1	new DummyJournalScan()	PASSED
1	-1	new DummyJournalScan()	PASSED
1	0	new DummyJournalScan()	PASSED
1	1	new DummyJournalScan()	PASSED
0	0	null	NUllPointer
0	0	new InvalidJournalScan()	IOExcep
0	MAX_INT	new DummyJournalScan()	NO_READ

Table 5: Casi di test per JournalTest. ExactBytes significa che si sa esattamente quanti byte vogliamo ottenere che la scan() legga, perché la scansione inizia dall'inizio del Journal. Se inizia dal primo byte invece verranno letti diversamente gli header, e dunque è più complicato avere una stima esatta di quanti bytes aspettarsi, e ci si accontenta dunque di una stima. NO_READ sta a significare che il DummyJournalScanner non deve nemmeno essere invocato, dal momento che non può esserci nulla da leggere a quella posizione.

index	journalDir	config	ledgDirsManager	statsLog	byteBufAlloc	Expected
-1	VALID	VALID	VALID	NullStatsLogger	Unpooled	PASSED
-1	VALID	VALID	VALID	NullStatsLogger	Unpooled	PASSED
0	VALID	VALID	VALID	NullStatsLogger	Unpooled	PASSED
1	VALID	VALID	VALID	NullStatsLogger	Pooled	PASSED
1	INVALID	VALID	VALID	NullStatsLogger	Unpooled	IOExcep
1	VALID	INVALID	VALID	NullStatsLogger	Unpooled	RunTimexcep
1	VALID	VALID	INVALID	NullStatsLogger	Unpooled	LEDGExcep
1	VALID	VALID	VALID	null	Unpooled	NPExcep
1	VALID	VALID	VALID	InvalidStatsLogger	Unpooled	RunTimexcep
1	VALID	VALID	VALID	NullStatsLogger	null	BUFFExcep
1	VALID	VALID	VALID	NullStatsLogger	InvalidBBA	RUNTIMEBUFFExcep
1	VALID	VALID	null	NullStatsLogger	Unpooled	NPExcep
1	null	VALID	VALID	NullStatsLogger	Pooled	NULL_JOURNAL
1	VALID	null	VALID	NullStatsLogger	Pooled	NPExcep

Table 6: Casi di test per setUpJournalTest. L'index è semplicemente un intero che rappresenta il Journal, category partition e boundary values sono banali. Una config invalida fa il throws di una RuntimeException. Un byetBuf non valido fa il throws di una unchecked exception. NULL_JOURNAL sta a significare che l'istanza di Journal non sarà null, ma lo sarà la cartella dei log del Journal.La differenza tra la prime due righe sta nello stato dell'istanza valida di config. Nei commenti della classe ci sono maggiori dettagli.

broker	newCopy	callBack	toAttachList	Expected
null	true	VALID	[foo]	NullPointer
null	true	VALID	[null]	NullPointer
null	true	VALID	[]	NullPointer
null	true	VALID	[foo, bar]	NullPointer
null	true	VALID	null	NullPointer
null	false	VALID	[foo]	NullPointer
null	false	VALID	[null]	NullPointer
null	false	VALID	[]	NullPointer
null	false	VALID	[foo, bar]	NullPointer
null	false	VALID	null	NullPointer
VALID	true	null	[foo]	NullPointer
VALID	true	null	[null]	NullPointer
VALID	true	null	[]	NullPointer
VALID	true	null	[foo, bar]	NullPointer
VALID	true	null	null	NullPointer
VALID	true	VALID	[foo]	PASSED
VALID	true	VALID	[null]	PASSED
VALID	true	VALID	[]	PASSED
VALID	true	VALID	[foo, bar]	PASSED
VALID	true	VALID	null	PASSED
VALID	true	INVALID	[foo]	RuntimeExcep
VALID	true	INVALID	[null]	RuntimeExcep
VALID	true	INVALID	[]	RuntimeExcep
VALID	true	INVALID	[foo, bar]	RuntimeExcep
VALID	true	INVALID	null	RuntimeExcep
VALID	false	null	[foo]	NullPointer
VALID	false	null	[null]	NullPointer
VALID	false	null	[]	NullPointer
VALID	false	null	[foo, bar]	NullPointer
VALID	false	null	null	NullPointer
VALID	false	VALID	[foo]	PASSED
VALID	false	VALID	[null]	PASSED
VALID	false	VALID	[]	PASSED
VALID	false	VALID	[foo, bar]	PASSED
VALID	false	VALID	null	PASSED
VALID	false	INVALID	[foo]	RuntimeExcep
VALID	false	INVALID	[null]	RuntimeExcep
VALID	false	INVALID	[]	RuntimeExcep
VALID	false	INVALID	[foo, bar]	RuntimeExcep
VALID	false	INVALID	null	RuntimeExcep
INVALID	true	VALID	[foo]	RuntimeExcep
INVALID	true	VALID	[null]	RuntimeExcep
INVALID	true	VALID	[]	RuntimeExcep
INVALID	true	VALID	[foo, bar]	RuntimeExcep
INVALID	true	VALID	null	RuntimeExcep
INVALID	true	INVALID	[foo]	RuntimeExcep
INVALID	true	INVALID	[null]	RuntimeExcep
INVALID	true	INVALID	[]	RuntimeExcep
INVALID	true	INVALID	[foo, bar]	RuntimeExcep
INVALID	true	INVALID	null	RuntimeExcep
INVALID	false	VALID	[foo]	RuntimeExcep
INVALID	false	VALID	[null]	RuntimeExcep
INVALID	false	VALID	[]	RuntimeExcep
INVALID	false	VALID	[foo, bar]	RuntimeExcep
INVALID	false	VALID	null	RuntimeExcep
INVALID	false	INVALID	[foo]	RuntimeExcep
INVALID	false	INVALID	[null]	RuntimeExcep
INVALID	false	INVALID	[]	RuntimeExcep
INVALID	false	INVALID	[foo, bar]	RuntimeExcep
INVALID	false	INVALID	null	RuntimeExcep

Table 7: Casi di test per AttachManager()

toAttach	persistenceCapable	openJPAStateManager	valueMetaData	explicit	Expected
null	null	null	null	true	PASSED
null	null	null	null	false	PASSED
null	null	null	VALID	true	PASSED
null	null	null	VALID	false	PASSED
null	null	VALID	null	true	PASSED
null	null	VALID	null	false	PASSED
null	null	VALID	VALID	true	PASSED
null	null	VALID	VALID	false	PASSED
null	VALID	null	null	true	PASSED
null	VALID	null	null	false	PASSED
null	VALID	null	VALID	true	PASSED
null	VALID	null	VALID	false	PASSED
null	VALID	VALID	null	true	PASSED
null	VALID	VALID	null	false	PASSED
null	VALID	VALID	VALID	true	PASSED
null	VALID	VALID	VALID	false	PASSED
foo	null	null	null	true	PASSED
foo	null	null	null	false	PASSED
foo	null	null	VALID	true	PASSED
foo	null	null	VALID	false	PASSED
foo	null	VALID	null	true	PASSED
foo	null	VALID	null	false	PASSED
foo	null	VALID	VALID	true	PASSED
foo	null	VALID	VALID	false	PASSED
foo	VALID	null	null	true	PASSED
foo	VALID	null	null	false	PASSED
foo	VALID	null	VALID	true	PASSED
foo	VALID	null	VALID	false	PASSED
foo	VALID	VALID	null	true	PASSED
foo	VALID	VALID	null	false	PASSED
foo	VALID	VALID	VALID	true	PASSED
foo	VALID	VALID	VALID	false	PASSED

Table 8: Casi di test per attach()

instances	Expected
null	NullPointer
[]	attachList vuota
[”foo”, ”bar”, ”test”]	PASSED
[”foo”*, ”bar”*, ”test”*]	Exception Annidate
[”foo”, ”bar”*, ”test”]	Singola Exception
[”foo”, ”bar”, ”test”][1]	RuntimeExcep

Table 9: Casi di test per attachAll().
*: oggetto non attachable.
[1]: oggetti con CallBack non valida.

Listing 6: riga 140 di CacheMap che provvede a creare la LRUMap.

```
cacheMap = new LRUMap(size, load);
```

Listing 7: chiamata in openjpa-lib.util

```
public LRUMap(int initCapacity, float loadFactor) {
    super(initCapacity, loadFactor);
}
```

Listing 8: chiamata in openjpa-lib.collections

```
public LRUMap(final int maxSize, final float loadFactor) {
    this(maxsize, loadfactor, false);
}
```

Listing 9: costruttore effettivo di LRUMap

```
1
2  public LRUMap(final int maxSize, final int initialSize, final float loadFactor, final boolean
       scanUntilRemovable)
```

Listing 10: Serie di chiamate per la costruzione della LRUMap

```
1
2  //Recap della serie di chiamate per la creazione dell'LRUMap.
3
4  // ---------------------------- 1 ----------------------------------
5  //CacheMap.java riga 140:
6  cacheMap = new LRUMap(size, load);
7
8  // ---------------------------- 2 ----------------------------------
9
10 //Questa chiamata chiama il metodo della classe util\LRUMap.java, riga 46
11 public LRUMap(int initCapacity, float loadFactor) {
12        super(initCapacity, loadFactor);
13    }
14
15 // ---------------------------- 3 ----------------------------------
16
17 //Il "super" del passo 2 corrisponde al metodo a riga 128 di collections\LRUMap.java
18 public LRUMap(final int maxSize, final float loadFactor) {
19        this(maxSize, loadFactor, false);
20    }
21
22 // ---------------------------- 4 ----------------------------------
23
24 //che chiama, nella stessa classe
25 public LRUMap(final int maxSize, final float loadFactor, final boolean scanUntilRemovable) {
26        this(maxSize, maxSize, loadFactor, scanUntilRemovable);
27    }
28
29 // ---------------------------- 5 ----------------------------------
30
31 //che a sua volta chiama (sempre in collections\LRUMap.java) il costruttore
32 //"principale", a riga 174:
33
34 public LRUMap(final int maxSize,
35                final int initialSize,
36                final float loadFactor,
37                final boolean scanUntilRemovable) {
38
39        super(initialSize, loadFactor);
40        if (maxSize < 1) {
41            throw new IllegalArgumentException("LRUMap max size must be greater than 0");
42        }
43        if (initialSize > maxSize) {
44            throw new IllegalArgumentException("LRUMap initial size must not be greather than max size
                  ");
45        }
46        this.maxSize = maxSize;
47        this.scanUntilRemovable = scanUntilRemovable;
48    }
49
50 //In questo metodo viene eseguito il throw dell'eccezione di cui si sta discutendo.
```

Listing 11: Frammento del report ba-dua per il metodo get(...) di CacheMap

```
<du covered="0" def="360" target="379" use="378" var="putcache"/>
<du covered="0" def="366" use="379" var="val"/>
<du covered="0" def="368" use="379" var="val"/>
<du covered="0" def="371" target="373" use="378" var="putcache"/>
```

```

<du covered="0" def="258" target="293" use="292" var="conf"/>
<du covered="0" def="265" use="271" var="iface"/>
<du covered="0" def="290" use="293" var="addr"/>



}
```

4 Link

- Fork di BookKeeper
- Fork di OpenJPA
- Report PIT di BookKeeper
- SonarCloud di BookKeeper
- SonarCloud di OpenJPA