BookKeeper

Un primo semplice caso

Il primo file selezionato per sviluppare dei casi di test nel progetto BookKeeper è “Value.java”; la classe Value appartiene al package “org.apache.bookkeeper.metastore” (responsabile della gestione dello storage dei metadati per l’applicazione) contenuto nel modulo “bookkeeper-server”. La scelta di questa particolare classe è dovuta a molteplici fattori, in particolare essa risulta molto stabile nel tempo, subendo poche revisioni nonostante la sua età di certo non trascurabile (il primo rilevamento risale alla terza release in esame, ed escludendo l’inserimento subisce una sola revisione nella release successiva, con un totale di soli due autori); a conferma della suddetta stabilità questa classe non risulta “buggy” in nessuna delle release in esame. Le motivazioni dietro la caratteristica appena descritta sono da ricercarsi probabilmente nella sua relativa semplicità e probabilmente nella mancanza di utilizzi di quest’ultima in situazioni complesse che ne potrebbero mettere a dura prova la correttezza, oltre all’assenza di test. La semplicità della classe in esame consente di concentrare l’attenzione su alcune caratteristiche altrimenti difficilmente considerabili in toto e di raggiungere alti livelli di copertura del codice in maniera più diretta (lo studio di test per casi più complessi verrà in ogni modo effettuata in seguito con altre classi). La classe Value è incentrata sulla variabile di istanza “fields”, ovvero una map di campi/valori (nello specifico String/byte[]) necessaria alla gestione del metastore. I riferimenti ai test ed i rispettivi metodi sono indicati alla fine del report.

Metodo equals

Un primo caso esaminato è la copertura del metodo “equals”; la natura relativamente nota di questo metodo consente un’analisi preliminare basata su criteri funzionali, nonostante l’assenza di documentazione riguardo la classe in esame. Effettuata una partizione del dominio di input, dei primi casi di test riguardano il confronto di un’istanza con: (1) un valore null, (2) un oggetto di tipo diverso (per fare questo è stato utilizzato il tag “@suppressWarnings(unlikely-arg-type)”), (3) un’istanza con dei campi diversi del map ed (4) un’istanza uguale a quella in esame. I casi (1) e (2) forniscono, come previsto, un risultato false, mentre i restanti due (che da previsione avrebbero dovuto restituire rispettivamente false e true) finiscono invece per lanciare un’eccezione. Allo scopo di comprendere le cause di tale comportamento è necessaria un’analisi del codice, spostando quindi l’interesse anche sull’ aspetto strutturale; il motivo del lancio dell’eccezione è la scelta di un particolare map per il test. Nel tentativo di ottenere un insieme di test minimale in grado di esplorare i diversi comportamenti del programma è stato usato un map contenente anche valori nulli (sia nel campo chiave che in quello valore); la presenza di valori nulli causa il lancio dell’eccezione all’interno di un comparator di byte[] utilizzato nel metodo in esame. Un primo tentativo di ricercare le boundary di questo possibile difetto consiste nell’inserimento di un test (5) per confrontare due Value con map totalmente differenti ma senza valori nulli (una sola coppia chiave/valore ognuna), ma anche tale test ha lo stesso esito dei precedenti e lancia un’eccezione. Anche in questo caso il motivo è da ricercarsi in un valore null all’interno del comparator precedentemente “incriminato”; questo è dovuto al modo in cui il confronto avviene all’interno del metodo equals prima ancora di lanciare il comparator: per ogni chiave di una delle istanze viene ricercato il valore associato a tale chiave nella seconda istanza e questa viene passata al comparator assieme al valore associato alla chiave della prima istanza. Come conseguenza di tale modalità di confronto, nel caso i due Value abbiano map con campi chiavi differenti, la ricerca del campo chiave della prima istanza all’interno della seconda restituirà null e tale valore sarà passato al comparator, portando al comportamento osservato. Come controprova è stato effettuato un ulteriore test (6) in cui avviene il confronto di due istanze Value con stesse chiavi ma valori diversi e questo ha restituito il valore atteso (false) senza lanciare eccezioni; è stato anche costruito un nuovo test (7) per istanze uguali ma senza valori null all’interno di fields, che ha anch’esso restituito il risultato atteso (true). Viene a questo punto spontaneo, data l’età elevata e le scarse modifiche effettuate al file nel tempo, supporre che tali circostanze non si siano mai verificate finora e che forse non esiste alcuna possibilità che il SUT utilizzi un’istanza di questa classe con, ad esempio, dei valori nulli all’interno di fields; questo comunque si contrappone alla presenza di controlli specializzati alla ricerca di valori null nel suddetto map (sia nella key che nel valore) all’interno di altri metodi, quali toString e merge.

Metodo hashCode

Un secondo metodo analizzato è fortemente interconnesso al primo, ovvero hashCode; anche in questo caso la natura della funzione permette di effettuare delle considerazioni funzionali preliminari. Gli input

iniziali per il test di questo metodo sono molto simili a quelli di equals, anche se stavolta non sono presenti argomenti nella chiamata del metodo, verranno pertanto computati gli hash di due Value e poi questi verranno confrontati tra loro (questa struttura porta ad escludere i test con valori null o di tipo diverso per ovvi motivi); i test effettuati sono dunque: (1) due Value con fields uguali (è stato sfruttato il caso di test per equals), (2) due Value i cui fields hanno le stesse chiavi ma valori diversi, (3) due Value i cui fields hanno chiavi diverse. I casi (1) e (3) forniscono i risultati attesi (rispettivamente true e false), mentre il test (2) che in prima analisi era atteso essere false fallisce; il motivo del true restituito è facilmente intuibile passando ad una analisi strutturale, che evidenzia come per il calcolo degli hash vengano utilizzati esclusivamente le chiave del map fields ma non i valori. Ad ogni modo stando al contratto del metodo hashCode *"It is not required that if two objects are unequal according to the* `equals(Object)` *method, then calling the* `hashCode` *method on each of the two objects must produce distinct integer results"*, pertanto tale test viene trascurato (lo stesso si potrebbe in realtà fare con il caso (2) per lo stesso motivo, ma vista l'implementazione si può pensare che gli sviluppatori volessero ottenere valori diversi almeno per key diverse).

Metodo toString

Si passa poi a testare il metodo toString; in questo caso per cercare di mantenere un insieme di test minimale si possono considerare due casi: (1) un Value appena creato, (2) un Value la cui map fields contiene, tra gli altri, valori null, sia come chiave che come valore; entrambi i suddetti test restituiscono i risultati attesi, come era prevedibile dal momento che l'analisi di questo metodo viene effettuata esclusivamente con un approccio strutturale.

Metodo project

Un altro metodo testato è project; tale metodo prende come argomento un set di chiavi e restituisce il sottoinsieme di fields con chiavi uguali a quelle inserite. In questo caso i test pensati sono: (1) il set corrisponde ad un sottoinsieme di chiavi di fields, (2) il set corrisponde all'intero campo fields; entrambi i test rispondono come previsto, restituendo rispettivamente un Value con il sottoinsieme corretto di elementi ed uno con tutti gli elementi. A questi test sarebbe possibile aggiungerne uno che passi per argomento un set contenente chiavi non appartenenti a fields dell'istanza in esame, ma in assenza di documentazione riguardo questa classe non è possibile conoscere il comportamento atteso o se questo caso possa in ogni modo verificarsi.

Metodo merge

Un ultimo metodo esaminato per lo sviluppo di test è merge; questo metodo effettua la fusione del campo fields dell'istanza chiamante con quello dell'istanza passata per parametro. In questo caso al fine di ottenere una granularità fine dei controlli è stato utilizzato un mock; (1) viene passata al metodo una istanza di Value con fields contenente diverse coppie chiave/valore, tra cui una chiave ed un valore null e, tramite l'utilizzo di verify si controlla che i valori vengano correttamente inseriti (tramite put, se non sono nulli) o rimossi dall'istanza attuale (tramite remove, nel caso di valori nulli); questa distinzione di comportamento naturalmente è stato possibile effettuarla esclusivamente tramite il codice del metodo stesso.

Coverage e test aggiuntivi

Grazie all'utilizzo di strumenti come Jacoco e Sonar, quest'ultimo connesso tramite Travis CI, è stata dunque effettuata un'analisi del livello di copertura del codice con casi di test; i due criteri riportati dagli strumenti suddetti sono nello specifico line coverage e condition coverage, che risultano già notevolmente elevati, complice la semplicità della classe in esame e la copertura da parte dei test di tutti i metodi (esclusi setter e getter naturalmente). Un caso di test sfuggito comunque al partizionamento degli input effettuato risulta comunque presente, nello specifico nel metodo equals è presente un controllo sul numero di elementi delle variabili fields delle due istanze di Value sotto analisi; al fine di coprire tale condizione e le linee appartenenti al ramo non esplorato (una in realtà) è stato inserito un altro test al metodo equals: (8) confronto tra due Value con map fields di dimensioni diverse, viene aggiunto assieme al test (1); tale test restituisce naturalmente il valore atteso (false) e contribuisce ad incrementare ulteriormente i livelli di copertura a livello di linee e di condizioni, che raggiungono così i valori finali di, rispettivamente, 95,1% e 100% (i getter ed i setter non hanno condizioni). Un ulteriore strumento utilizzato a questo punto è PitTest, utilizzato per l'analisi del mutation coverage; ad una prima analisi su 22 mutazioni ne vengono uccise 16. Delle 6 rimaste 5 riguardano nuovamente setter e getter, pertanto l'attenzione viene concentrata su quella rimasta; questa

indica come i test non abbiano identificato la sostituzione del valore di ritorno del metodo merge con null. Al fine di aumentare questo tipo di copertura viene dunque utilizzato un nuovo test nel metodo suddetto: (2) si controlla che il risultato del merge di una istanza Value con una seconda appena creata fornisca un Value con lo stesso fields della prima istanza; tale controllo permette di uccidere la mutazione in esame e portare il livello di mutation coverage al 77% (17/22). Infine per una ulteriore analisi viene utilizzato Ba-dua, che consente di calcolare un criterio di adeguatezza basato su data flow; in questo caso, grazie ancora alla relativa semplicità del caso in esame la copertura risulta completa già da una prima analisi (70 su 70) e non è necessario prendere alcun provvedimento.

Secondo caso

Il secondo caso di studio da esaminare, decisamente più complesso del primo, riguarda il file "RoundRobinDistributionSchedule.java", appartenente al package "org.apache.bookkeeper.client", contenuto anch'esso nel modulo "bookkeeper-server". La classe RoundRobinDistributionSchedule contiene al suo interno altre classi, che all'interno di target/classes vengono gestite come file diversi di tipo class, tale distinzione deve pertanto essere gestita all'interno dei filtri di PitTest e Jacoco per poter ottenere una corretta valutazione dei livelli di coverage. La scelta di questo file per lo sviluppo di casi di test risiede nella sua profonda differenza rispetto al primo per quanto riguarda le informazioni reperite su BookKeeper. Il primo rilevamento del file risale già alla prima release ottenuta, pertanto si può supporre che le classi che vi appartengono siano di importanza centrale per il progetto; durante le prime release di vita subisce 5 revisioni (rispettivamente 2 nella prima release, 1 nella seconda e 2 nella terza), per poi stabilizzarsi e non ricevere più modifiche (almeno fino alla settima release, che è l'ultima rilevata). Il file riceve un bug fix durante la seconda release (ovvero nei commit tra la prima e la seconda), e risulta buggy nella prima release, per poi non esserlo più in tutte quelle successive. Un'altra differenza sostanziale tra Value.java e RoundRobinDistributionSchedule.java è che per il primo gli sviluppatori hanno realizzato a loro volta dei casi di test, introdotti nel progetto a partire dalla terza release (tali test hanno poi subito anch'essi una revisione nella release numero sei). Va comunque rimarcato come lo studio dell'evoluzione di BookKeper sia stato effettuato limitatamente alla prima metà delle release del progetto (per limitare il problema del bug snoring), e risulta chiaro come il file in esame sia stato modificato significativamente anche in un momento successivo (il numero di LOC nella settima release risultava di 109, mentre quello attuale è di 442). Siccome questo file contiene numerosi metodi appartenenti a diverse sottoclassi, soltanto alcuni di questi verranno esaminati.

Metodo moveAndShift

Il primo metodo per cui sono stati sviluppati dei test è moveAndShift appartenente alla sottoclasse WriteSetImpl; tale sottoclasse è in realtà privata ma è facilmente accessibile tramite il metodo wirteSetFromValues che si occupa di generare e ritornare tale classe; al suddetto metodo è associato dagli sviluppatori il tag "@VisibleForTesting" proprio allo scopo di consentire il test sulla classe in esame. Il metodo moveAndShift effettua lo spostamento di un valore all'interno dell'array di appartenenza (cambio di posizione); l'array su cui opera viene inizializzato prima dell'esecuzione del metodo tramite writeSetFromValues. Il metodo in esame prende in input due argomenti, che corrispondono, nell'ordine, alla posizione attuale dell'elemento dell'array da spostare ed alla posizione a cui è destinato; lo spostamento non avviene tramite lo scambio degli elementi ma tramite lo spostamento di una posizione degli elementi tra la posizione di partenza e quella di destinazione. La prima operazione effettuata dal metodo non appena invocato è la chiamata di un metodo privato (checkBounds) che controlla che gli argomenti di input siano contenuti nei bounds dell'array (il metodo viene quindi invocato due volte, una per ogni argomento); successivamente avviene lo spostamento tramite una variabile temporanea. La prima decisione da prendere per sviluppare dei test è l'array su cui operare; per considerare più variazioni possibili limitando il numero di test viene scelto un array che contiene sia numeri positivi che negativi, assieme ad uno zero (naturalmente tale scelta è stata effettuata a priori dello studio strutturale del metodo in esame), per la precisione l'array utilizzato è (-2, -1, 0, 1, 2) che ha quindi dimensione 5; a questo punto il passo successivo è la scelta dei due parametri in input (*from* e *to*), questi verranno passati al metodo ed il risultato verrà confrontato con l'oracolo. Effettuata la partizione dei domini di input, le prime coppie di valori di input su cui il metodo viene testato sono (1): (3, 1), (2): (1, 3), (3): (0, 4) e (4): (1, 1); le prime due coppie di valori controllano il comportamento del metodo con (to > from) e (from > to), con gli stessi valori degli argomenti scambiati tra i due test. Il terzo test esegue il controllo del comportamento nel caso di elementi ai bordi dell'array (entrambi i bordi con un solo test). Infine il quarto test si occupa di controllare che il metodo restituisca l'output corretto in caso

l'elemento venga "scambiato con se stesso" (from = to); tutti e quattro gli input considerati forniscono i risultati attesi dall'oracolo, confermando il corretto comportamento del metodo. A questo punto vengono inseriti due test il cui scopo è di aumentare la copertura del codice tramite il controllo dell'esecuzione nei casi in cui sia atteso un errore; gli argomenti scelti, mantenendo la politica della selezione degli elementi ai bordi che più spesso presentano problematiche, sono (5): (-1, 3) e (6): (5, 0). Il controllo della correttezza di questi due casi risiede nel catch di un'eccezione di tipo "InputOutOfBoundsException", tale eccezione viene lanciata dal metodo privato checkBounds nel caso gli indici non appartengano all'array; ad una verifica preliminare entrambi i test effettuati ottengono il risultato atteso e finiscono con il lanciare la suddetta eccezione. L'aggiunta di questi ultimi due test permette di aumentare la copertura sia a livello di linee di codice che di condizioni. Quest'ultima analisi però, assieme a quella con le mutazioni effettuate tramite PitTest, forniscono un risultato non immediatamente comprensibile: nonostante il controllo sui bounds sia eseguito in tutti i casi possibili, ovvero index < 0, index > size ed il caso corretto 0 <= index <= size (è naturalmente impossibile il verificarsi di index < 0 e index > size insieme), la copertura della condizione del metodo privato checkBounds non risulta comunque completa. Viene a questo punto effettuato un controllo più approfondito a livello strutturale per verificare le cause di questo comportamento, poiché i test attualmente effettuati non hanno, per qualche motivo, riscontrato alcun problema sul codice coperto. Un'analisi più accurata del metodo incriminato mostra come ci sia un problema all'interno del controllo di appartenenza dell'indice all'array; il bug risiede nel considerare come appartenente all'array un indice pari a size, ovvero alla dimensione dell'array stesso; il controllo effettuato è:

if (i < 0 || i > size) then throw new IndexOutOfBoundsException,

tale controllo dovrebbe invece essere sostituito con:

if (i < 0 || i >**=** size) then throw new IndexOutOfBoundsException.

Questo errore spiega perfettamente il motivo per cui la condizione non risultava completamente coperta (il branch i > size non viene tuttora esercitato) ed il perché anche PitTest non riuscisse ad uccidere tramite i test effettuati la mutazione riguardante una modifica dei bound del controllo. Inizia a questo punto uno studio più accurato per cercare di capire il motivo per cui i test attuali non abbiano individuato prima tale bug. Il test che più di ogni altro sembrava dover avere problemi nell'essere superato è naturalmente il numero (6), in cui la coppia di valori (5, 0) finisce comunque per lanciare l'eccezione nonostante il metodo appena esaminato non lo faccia. La causa di tale assenza di errori nei test è da ricercarsi nelle righe di codice del metodo moveAndShift immediatamente successive all'esecuzione delle due chiamate al metodo privato checkBounds. Il metodo in esame, per scambiare le posizioni degli elementi dell'array effettua varie operazioni, tra cui: "int tmp = array[from]" e "array[to] = tmp" il cui scopo è il salvataggio del valore da spostare tramite una variabile temporanea per poi inserirlo nella posizione opportuna al termine dello spostamento di tutti gli altri elementi tra la posizione di partenza e quella di destinazione; tali due operazioni se invocate con un indice (valore di to e from) non appartenente all'array finiscono anch'esse per lanciare un'eccezione. L'eccezione lanciata in questo caso è di tipo "ArrayIndexOutOfBoundException", che estende "IndexOutOfBoundException"; per questo motivo quando il test realizzato preliminarmente va ad analizzare la presenza dell'eccezione riesce in effetti a riscontrarla, nonostante non sia esattamente quella corretta voluta dagli sviluppatori di BookKeper. A primo impatto può sembrare che tale differenza non sia sostanziale (come appena visto un controllo sull'eccezione di quel tipo avrà comunque lo stesso effetto) ed è il motivo per cui nonostante l'età avanzata della classe e la presenza di diversi test realizzati dagli sviluppatori su quest'ultima tale problema non sia prima venuto alla luce; il punto chiave è che considerando questo un problema di poco conto si rende totalmente inutile la presenza di tali controlli effettuati dal metodo privato checkBounds, in quanto se la differenza tra tali eccezioni è da considerarsi ignorabile tale metodo risulta a tutti gli effetti completamente superfluo ed eliminabile. Uno dei motivi che può aver portato alla realizzazione del metodo buggato è la possibilità futura di estendere il comportamento dell'applicazione in caso di tale errore, ed in questo caso tale bug potrebbe risultare notevolmente più dannoso di quanto non lo sia attualmente. Un modo molto naive per verificare la discrepanza di comportamento (quello utilizzato in un primo momento per la ricerca del bug) è la stampa dei due messaggi di errore nei casi (-1, 3) e (5, 0); la differenza nei messaggi di errore è indicativo del lancio scorretto dell'eccezione. Una volta individuato esattamente il problema la distinzione può essere fatta in modo più solido riconoscendo le eccezioni di tipo ArrayIndexOutOfBoundsException, nello specifico se l'eccezione lanciata IndexOutOfBoundsException è di questo tipo allora il comportamento è scorretto, altrimenti il metodo si comporta come ci si aspetta. Modificata quindi la struttura del test seguendo tale linea guida viene quindi inserita un ultima coppia di argomenti di input (7): (0, 6); questo input è scelto tale per due diversi motivi, prima di tutto esso consente di coprire la condizione completamente (in quanto risulta to > size), confermando così la correttezza dell'ipotesi appena analizzata, ed in secondo luogo permette di testare il comportamento del metodo anche nel caso sia

il secondo argomento a non rientrare tra gli indici dell'array (i due test precedenti erano eseguiti esclusivamente sul primo). Questo controllo aggiuntivo anche sul secondo elemento comporta anche l'uccisione di una mutazione in più, ovvero la rimozione della seconda chiamata del metodo checkBounds che era prima sopravvissuta, consentendo così di aumentare anche la copertura a livello di mutazione; quest'ultimo test passa senza particolari complicanze e tutto si comporta come ci si aspettava (anche in relazione ai livelli di copertura). Il test (6) è stato al momento commentato per consentire la build del progetto senza errori, ma ad ogni modo qualora il problema venga risolto e si voglia riaggiungere quest'ultimo può essere fuso assieme al (7) alla ricerca di un test set minimale; un esempio potrebbe essere la coppia di valori (0, 5), in questo modo si avrebbero ancora sia la copertura del valore al bordo della partizione di input che la copertura dei casi con entrambi gli argomenti fuori dagli indici dell'array.

Metodo getEntriesStripedToTheBookie

Il secondo caso di studio in questa sezione è il metodo getEntriesStripedToTheBookie appartenente direttamente alla classe principale, non una di quelle interne. Tale metodo è il cuore dell'intera classe in quanto si occupa di quella che è la sua funzione principale, ovvero la restituzione, a partire da un insieme di bookie e di entry, di una mappa di bit che indica quali delle entry vengono assegnate al bookie in esame; l'assegnazione avviene con modalità round robin, come indicato dal nome della classe principale, e necessita della conoscenza di alcune informazioni aggiuntive, quali la dimensione del quorum da raggiungere tra i bookie (ovvero a quanti dei bookie deve essere inviata una determinata entry). Per fare un esempio semplice del meccanismo di funzionamento, avendo a disposizione un insieme costituito di 3 bookie, con dimensione del quorum pari a 2, la prima entry va ai bookie 0 e 1, la seconda a 1 e 2, la terza a 2 e 0 e così via; secondo questa logica al bookie numero 1 (il secondo) saranno assegnate la prima e la seconda entry, poi la quarta e la quinta e così via. Il BitSet restituito sarà quindi, in questo esempio [1101101…] fino alla dimensione massima che sarà pari al numero di entry. Il metodo in esame prende in input tre argomenti, nell'ordine: l'indice del bookie nell'insieme di questi ultimi, l'indice della prima entry e l'indice dell'ultima entry; la dimensione del set di entry sarà così naturalmente calcolata tramite la differenza delle due (+1 per considerare entrambi gli estremi) e sarà possibile partire da una entry con un numero arbitrario. Oltre ai valori in input il metodo ne utilizza altri due altrettanto necessari: la dimensione dell'insieme dei bookie e la dimensione del quorum sopra descritto; tali valori vengono settati direttamente dall'invocazione del costruttore della classe. La prima idea di partizionamento dei domini di input è di avere: un valore negativo, uno pari a zero ed almeno uno positivo per tutti e tre gli argomenti bookieIndex, startEntryId e lastEntryId; già prima di procedere all'analisi strutturale, guardando meglio i nomi degli ultimi due parametri si può notare come essi debbano necessariamente essere legati tra loro, pertanto un piccolo miglioramento dell'idea principale può essere: lasciare invariato quanto detto per bookieIndex e startEntryId ma modificare lastEntryId in modo di avere un valore minore di startEntryId, uno uguale ed uno maggiore di quest'ultimo. Partendo da questi presupposti vengono realizzati dei primi casi di test, da estendere poi, con valori degli argomenti (1): (0, 0, 3), (2): (2, 0 3), (3): (0, 1, 1), (4): (2, -1, 0), (5): (-1, 0, 3) e (6): (0, 1, 0); in un primo momento per semplicità le due variabili utilizzate dal metodo e non ottenute come argomento vengono lasciate fisse con valori writeQuorumSize (dimensione del quorum) = 2 ed ensembleSize (dimensione dell'insieme di bookie) = 3. I test consistono nell'invocazione del metodo ed il confronto dell'output e della cardinalità di quest'ultimo (il numero di bit del BitMap settati ad 1) con un oracolo precedentemente ricavato, oppure nel semplice controllo del lancio di un'eccezione attesa. Nello specifico i primi tre test portano ai risultati attesi di BitMap con cardinalità diverse tra loro, mentre i restanti tre portano al lancio di eccezioni di tipo IllegalArgumentException per i motivi che ci si aspettava, ovvero rispettivamente startEntryId < 0, bookieIndex < 0 e lastEntryId < startEntryId. Si procede a questo punto all'estensione dei casi di test, aggiungendone in primo luogo uno con gli stessi valori del test (3), ovvero (7): (0, 1, 1), ma con un ensembleSize settato a 0; ci si aspetta naturalmente che tale caso sollevi anch'esso un'eccezione, o per il valore minore del bookieId o comunque poiché questo non è positivo, ed in effetti anche questo caso ha il comportamento atteso. Per evitare di ottenere errori invece di fallimenti di test, nei casi ci si aspetti un'eccezione viene assegnato un oracolo con un array di dimensione 0 e viene poi gestito con un Assert.fail il caso di lancio di eccezione di tipo IndexOutOfBoundsException. Il metodo sotto test oltre a lanciare l'eccezione in caso di argomenti considerati non validi stampa anche un log che avvisa dell'errore a video stampando i parametri di input; per evitare tale verbosità e la continua stampa di errori durante i test è stato aggiunto un file di configurazione linkato dal pom in cui viene semplicemente specificato di escludere tali messaggi nella fase di test. Come in tutti i casi precedenti viene a questo punto effettuata l'analisi della copertura del codice dai casi di test, ponendo quindi maggiormente l'attenzione sugli aspetti strutturali. Il primo test aggiunto grazie a tali report consiste nella scelta di un lastEntryId < 0 (come originariamente era stato supposto) per esercitare una condizione altrimenti non coperta; i valori scelti in questo caso sono (8): (0, 1, -1) e l'eccezione attesa viene regolarmente sollevata. Ad una analisi più attenta comunque è possibile notare come questa condizione risulti completamente inutile, in quanto tale caso è già coperto da altre due condizioni:

risulta impossibile avere (lastEntryId < 0) senza avere (startEntryId < 0 || lastEntryId < startEntryId),

ovvero se lastEntryId è negativa allora o startEntryId è maggiore di quest'ultima o è anch'essa negativa. Da questo punto risulta decisamente più complesso riuscire a coprire maggiormente il codice, poiché le condizioni non si basano esclusivamente su argomenti presi direttamente da input ma anche su alcuni calcolati indirettamente tramite operazioni di modulo. Una condizione non completamente coperta è ad esempio alla riga 435 in un if annidato all'interno dell'else di quello esterno; tale condizione nei test finora eseguiti o non è stata raggiunta o quando lo è stata si è rivelata essere sempre vera a causa delle due condizioni atomiche che la compongono. Facendo diversi tentativi per seguire il flusso di esecuzione sono stati trovati dei valori che consentono la copertura anche di tale possibilità tramite il test (9): (2, 0, 3) con dei valori di writeQuorumSize = 3 ed ensembleSize = 4, in cui tale condizione viene raggiunta e risulta falsa per l'ultima entry; naturalmente anche questo test restituisce i risultati attesi e consente così di incrementare ulteriormente la condition coverage. Si passa a questo punto all'analisi della mutation coverage tramite PitTest, che rileva come diversi mutanti siano sopravvissuti ai test, alcuni dei quali risultano tremendamente difficili, se non impossibili, da eliminare; un primo esempio è una mutazione della prima condizione, che non subisce modifiche nel comportamento se mutata (probabilmente a causa della condizione inutile sopra discussa), ed un secondo mutante, forse ancora peggiore riguarda la creazione della BitMap, che come già indicato ha una dimensione pari a lastEntryId – startEntryId +1; la sostituzione del meno con il più non viene rilevata dai test effettuati ed il motivo è semplice: la creazione di una BitMap di dimensione maggiore non può essere riscontrata da nessun metodo, in quanto la lunghezza della stessa viene considerata esclusivamente fino al raggiungimento dell'ultimo bit posto ad 1 e non oltre, rendendo probabilmente impossibile il rilevamento di tale modifica; l'unico modo di rendere possibile l'individuazione di tale sostituzione sarebbe rendere lastEntryId o StartEntryId negative, ma in questo caso l'esecuzione non raggiungerebbe comunque questo punto. Alcune altre mutazioni risultano invece eliminabili e l'attenzione viene concentrata su due molto simili tra loro alle righe 432 e 436; tali mutazioni sostituiscono nuovamente un segno meno con un più, ma stavolta all'interno del metodo set della BitMap che nella versione corretta ha come argomento entryId - startEntryId. Questi due mutanti sono sopravvissuti finora per via della scelta effettuata di startEntryId nella maggior parte dei test; salvo alcuni test infatti, nella maggior parte dei casi tale parametro viene posto a zero, portando quindi al mancato riconoscimento della mutazione (viene sostituito "- 0" con "+ 0"). Individuata dunque la causa di tali mancati rilevamenti si procede alla creazione di un nuovo set di argomenti per il test (10): (0, 1, 3) con writeQuorumSize = 2 ed ensembleSize = 3; tale test raggiunge durante la sua esecuzione entrambe le linee di codice incriminate e consente dunque di eliminare due mutazioni in un colpo solo, incrementando quindi la mutation coverage.

Coverage e Ba-Dua

I livelli totali di coverage della classe risultano infine di 62,2% per la line coverage, 64,3% per la condition coverage ed il 46% per la mutation coverage; tali valori sensibilmente più bassi rispetto al primo caso di studi non devono sorprendere in quanto tengono conto di diversi metodi non esaminati, mentre limitatamente ai metodi per cui i test sono stati ideati le tre coperture risultano sensibilmente più alte dei valori attuali. Infine come per il primo caso di studi (e limitatamente a questi due) viene utilizzato Ba-dua come strumento per l'analisi con criteri basati su data-flow, nella speranza che questa volta ci fornisca più indicazioni del caso precedente, che risultava già completamente coperto. L'esecuzione riscontra infatti alcune coppie definition-use non coperte dall'attuale set di test, sia per il metodo moveAndShift che per getEntriesStripedToTheBookie. Lo studio dei test necessari per coprire tali coppie è tutt'altro che semplice, ed in alcuni casi può risultare impossibile senza modificare il metodo principale (esattamente come la mutazione precedentemente discussa). Per il metodo moveAndShift la copertura è di 46 su 48, con 2 sole coppie sfuggite ai test; queste ultime però fanno riferimento ad una variabile "i" dichiarata ed utilizzata in dei for loop alle righe 180 e 186, in cui l'eventualità che tale loop esca immediatamente dopo l'inizializzazione di "i" non è coperta; la definizione a cui si fa riferimento sembrerebbe dunque quella del ciclo, così come l'uso, mentre come target si ha l'istruzione immediatamente successiva al ciclo, in cui si effettua la sostituzione tramite l'indice stesso e la variabile temporanea. Tali coppie def-use risultano tuttavia impossibili da ottenere con tali target senza prima passare per l'interno del ciclo, questo a causa di controlli eseguiti prima dei cicli incriminati: il primo ciclo parte da "from" e procede verso "to" decrementando "i" ed è preceduto da un "if (from > to)", mentre il secondo ha un comportamento speculare, con il ciclo che parte da "from" e procede verso "to" incrementando "i" ed è preceduto da un "else if (from < to)"; in queste condizioni è dunque impossibile non passare mai all'interno dei cicli for (probabilmente un ciclo do-while eliminerebbe tale difficoltà). Si procede dunque ad una analisi del metodo getEntriesStripedToTheBookie, dove in un primo momento le coppie def-use che sono state coperte sono 69 su 71; una delle due non coperte è un caso similare a quello precedente, in cui la variabile sotto indagine è "entryId", la definizione è all'inizio di un ciclo for (riga 427), così come l'uso, mentre il target a cui si fa riferimento è il return (riga 440) subito dopo il ciclo stesso; anche in questo caso il ciclo è impossibile da non percorrere, in quanto la variabile "entryId" è inizializzata a startEntryId e viene incrementata finchè risulta minore o uguale a lastEntryId, tuttavia tale ciclo

si trova dopo aver controllato la condizione (lastEntryId >= startEntryId) e dovrà quindi essere necessariamente percorso. Rimane dunque da analizzare un'ultima coppia def-use non coperta; questa fa riferimento ancora una volta alla variabile "entryId" dichiarata all'inizio del ciclo for alla riga 427, mentre l'uso a cui ci si riferisce è alla riga 436 all'interno dell'if annidato nell'else di quello esterno. In questo caso risulta finalmente possibile coprire tale coppia def-use tramite un caso di test opportuno, dopo aver identificato la causa che ha portato alla mancata copertura fino a questo punto; in particolare analizzando i test precedenti e studiandone i flussi di esecuzione all'interno del metodo si può notare quello che tale mancanza di copertura sta ad indicare: non esiste alcun test che, una volta raggiunto il ciclo ed inizializzata la variabile "entryId", raggiunga immediatamente la riga di use indicata (ovvero non rispetti la condizione dell'if esterno, finendo così nell'else, e rispetti poi la condizione dell'if annidato). Vengono a questo punto identificati i valori degli argomenti da passare in input al metodo in esame per avere il comportamento del flusso di esecuzione sopra descritto, tramite il test (11): (0, 1, 1) come i test (3) e (7) ma questa volta con writeQuorumSize = 2 e ensembleSize = 3; anche quest'ultimo test si comporta come era atteso e, riavviando l'analisi di Ba-Dua si può notare come questo abbia consentito di coprire una ulteriore coppia def-use portando il totale di coppie coperte a 70 su 71.

Syncope

Terzo caso

Il terzo caso di studio è il file "SyncopeFiqlParser.java", appartenente al package "org.apache.syncope.common.lib.search" nel modulo "syncope-common-idrepo-lib". Uno dei fattori che ha portato alla scelta di questo file è che quest'ultimo è stato aggiunto al progetto in uno stato già notevolmente avanzato, che si contrappone ai precedenti file studiati di BookKeeper; SyncopeFiqlParser è così recente da non permettere di conoscere informazioni al riguardo tramite lo studio usato in precedenza, questo poiché si colloca dopo la metà del numero di release totali (la seconda metà come già indicato viene esclusa per limitare il problema dello snoring bug). Il primo commit relativo al file risale al 14 Dicembre 2018, ovvero più di 6 anni dopo la data di rilascio della prima release (6 Agosto 2012); in base alle date raccolte delle varie release tale file dovrebbe essere stato aggiunto al progetto indicativamente tra la release numero 47 e la 48 (su 51 totali riscontrate, quindi molto recente). Per la classe trattata ad ogni modo sono stati comunque già realizzati dei test, inseriti nel progetto nello stesso commit del file testato. La classe in esame rappresenta l'estensione di un'altra, "FiqlParser", dove Fiql sta per Feed Item Query Language e costruisce una sintassi per l'espressione di filtri in determinati campi; SyncopeFiqlParser si occupa di estendere tale sintassi con due nuovi operandi: IEQ (uguaglianza non case-sensitive, indicata con i simboli =~) e NIEQ (disuguaglianza non case-sensitive, indicata con i simboli !~), che vanno ad aggiungersi a quelli che erano già presenti nella superclasse (come maggiore, maggiore o uguale, ecc.). Il primo problema di cui occuparsi è l'istanziazione della classe stessa, questa è infatti definita come una classe generica con un parametro di tipo T (SyncopeFiqlParser<T>); per la realizzazione di una istanza valida (così da testare principalmente il metodo della classe senza troppi vincoli legati alla classe estesa) si è preso spunto, per questa volta, dai test già presenti realizzati dagli sviluppatori (tale aspetto non verrà pertanto approfondito oltre in questo contesto).

Metodo parseComparison

Il metodo analizzato per la costruzione di test è parseComparison, che sovrascrive l'omonimo metodo della superclasse e contribuisce alla fase di parsing aggiungendo i simboli logici sopra specificati. Il meccanismo di funzionamento di Fiql si basa sulla costruzione di filtri nella struttura "<nome_campo> <simbolo_logico> <valore_campo>" (ad esempio "età=gt=40" filtra solo i campi con età maggiore del valore specificato); tale struttura viene specificata tramite il parametro di tipo String "expr", che è anche l'unico in input al metodo stesso. Contrariamente ai test degli sviluppatori, i quali si sono concentrati sull'analisi del metodo "parse" che invoca poi indirettamente il metodo in esame, qui si è preferito studiare il comportamento direttamente di parseComparison. Il metodo in esame utilizza alcune chiamate ad altri metodi, i quali fanno però parte della superclasse di quella attuale, il loro comportamento viene pertanto ereditato da questa, e non viene considerata una buona pratica usare dei mock per questi ultimi; inoltre tali metodi nei test utilizzati restituiscono una stringa tale e quale a quella presa in input, rendendone superfluo il mock per questi casi. In caso di successo il metodo restituisce una istanza di classe SyncopeComparison, anch'essa definita all'interno dello stesso file; tale classe è tuttavia privata, pertanto per semplicità ci si è limitati a controllare che il metodo sotto test restituisse un oggetto non nullo nell'eventualità che l'impostazione del filtro vada a buon fine. Il comportamento in caso di esecuzione anomala invece (con un argomento in input che non rappresenta un filtro valido) è costituito dal lancio di un'eccezione di tipo SearchParseException. I primi casi

di test sviluppati seguendo la tecnica del category partition sono (1): “fieldname=~value”, (2): “fieldname!~value” e (3): “”. I primi due test rappresentano un input di tipo valido e controllano il corretto funzionamento del metodo in caso di occorrenza di entrambi i filtri inseriti dall’estensione in esame, mentre il terzo è una stringa vuota e costituisce pertanto, per forza di cose, un input non valido come filtro; tutti e tre i test hanno il comportamento atteso dall’oracolo, con i primi due che restituiscono una istanza non nulla ed il terzo che lancia l’eccezione di tipo SearchParseException. Come nei due casi precedenti vengono a questo punto avviate le analisi della copertura del codice a livello di linee, di condizioni e di mutazioni, rispettivamente tramite Jacoco/Sonar e PitTest. Le analisi riscontrano già un buon livello di copertura del metodo, ma comunque non completo; si decide dunque ancora una volta di aggiungere almeno un test in modo da aumentare tale copertura seguendo tali analisi (stavolta soprattutto linee e condizioni). Una parte del codice rimasta scoperta si occupa di lanciare un’eccezione nel caso, a seguito del parsing effettuato tramite i simboli logici all’interno delle stringhe, il campo valore risulti essere nullo; viene dunque aggiunto un nuovo test adatto a tale situazione, con una stringa non vuota, contenente un simbolo logico valido, ma senza campo valore, (4): “fieldName!~”. Il test, una volta avviato, lancia l’eccezione SearchParseException come era atteso e, confermandolo con il rilancio delle analisi precedenti, permette di esercitare la parte di codice precedentemente scoperta, incrementando tutti e tre i tipi di copertura di quest’ultimo. A questo punto, seguendo la filosofia dettata dall’ultimo test si è voluto aggiungerne un altro (5): “=~value” che rappresenta una stringa che come la precedente è non vuota e contiene un simbolo logico valido, ma stavolta ha un valore del campo che è valido ma un nome che non lo è (non è presente); anche tale test ha lo stesso esito del precedente e lancia la medesima eccezione come ci si attendeva. Va fatto notare comunque come quest’ultimo test può essere probabilmente considerato superfluo dal punto di vista del livello di copertura in quanto non va ad aumentarne nessun tipo (pertanto non si adatta perfettamente alla filosofia del test set minimale), e segue più che altro un approccio funzionale più che strutturale.

Coverage

Al termine della realizzazione dei test i livelli di copertura sono: line coverage 58%, condition coverage 50% e mutation coverage 29%; le cause dei valori discretamente bassi riscontrati sono ancora una volta da attribuire alla presenza nella classe in esame di numerosi metodi non coperti dai casi di test; nello specifico qui questi ultimi si trovano all’interno della classe privata (che si è comunque scelto di mantenere nell’analisi). Facendo riferimento esclusivamente al metodo in esame i livelli di copertura mancante sono molto esigui: una linea ed una condizione mancanti e zero mutazioni sopravvissute.

Quarto Caso

Il quarto ed ultimo caso di studi è il file “Encryptor.java”, appartenente al package “org.apache.syncope.core.spring.security” nel modulo “syncope-core-spring”. Uno dei fattori che ha portato alla scelta di questo file è l’alto numero di revisioni che quest’ultimo ha ricevuto; fa la sua comparsa nel progetto nella release numero 18, per poi essere modificato nella 20 ed in altre release successive dopo aver subito uno spostamento; il file presenta già dei test realizzati dagli sviluppatori, inseriti nel progetto nella stessa release di quest’ultimo. La classe in esame si occupa di gestire gli aspetti inerenti la crittografia di messaggi (o password), sfruttando librerie esterne per la realizzazione di diversi algoritmi di cifratura, indicati in una enumerazione (quella su cui si focalizza maggiormente è tuttavia la AES, ed è pertanto quella su cui la maggior parte dei test fanno leva). La classe sfrutta un blocco statico per eseguire dei controlli ed impostare, se necessario, dei valori di default per delle costanti di cui fa uso, ed un metodo statico che restituisce un’istanza della classe; il metodo “getInstance” prende in input una Stringa (secretKey) che rappresenta la chiave di cifratura utilizzata dell’istanza di Encryptor restituita. I test della classe sono suddivisi in due parti, grazie al runner “Enclosed”, una prima parte che sfrutta il runner “Parameterized” ed una seconda per test che verranno invece eseguiti una singola volta trascurando la parte parametrica.

Test parametrizzati

Questa sezione sfrutta il runner Parameterized per diminuire la replicazione del codice ed effettuare i medesimi test due volte con parametri diversi; tutti questi test si basano sulla previa creazione di una istanza valida della classe che ha per chiave di cifratura “secretKeyLongEnough” (la ragione del nome sarà spiegata nella sezione dei test non parametrizzati)

Metodo encode

Il primo metodo analizzato è “encode”, questo può essere considerato uno dei più importanti della classe, prende in input due parametri, una Stringa (“value”) che rappresenta il messaggio da cifrare ed un cipherAlgorithm che indica quale algoritmo di cifratura utilizzare sul messaggio (a scelta da una enum),

assieme alla chiave di cifratura scelta al momento della creazione dell'istanza della classe. I primi due test scelti, nella sezione parametrica, utilizzano (1): ("string to encode", CipherAlgorithm.AES) e (2): (null, CipherAlgorithm.AES); il primo test costituisce un messaggio valido da cifrare con AES, mentre il secondo una stringa nulla, quindi non valida. L'oracolo per il primo test è stato generato eseguendo a priori una codifica del messaggio con il medesimo algoritmo di cifratura tramite strumenti online e con questo il metodo restituisce il valore atteso; il secondo test quando eseguito ha anch'esso il comportamento atteso, restituendo in output il valore null ottenuto in input; in un primo momento è stato considerata, durante il partizionamento del dominio di input, la stringa vuota come eventualità oltre al valore null, ma per un motivo che verrà tra poco spiegato tale test è stato separato da questi parametrici.

Metodo decode

Il secondo metodo analizzato è "decode" con un comportamento speculare ad encode; anche questo metodo, come il precedente, prende in input due parametri, una stringa ("encoded") che rappresenta il valore codificato da decifrare ed un cipherAlgorithm che specifica il metodo di cifratura usato, in modo da poter eseguire la decifratura sfruttando ancora una volta la chiave di cifratura scelta al momento della creazione dell'istanza della classe. Sfruttando gli stessi parametri del primo metodo i test sono (1): (MSGC, CipherAlgorithm.AES) dove con MSGC si indica il messaggio cifrato, non riportato per leggibilità, e (2): (null, CipherAlgorithm.AES); anche in questo caso entrambi i test vanno a buon fine, con il primo che restituisce il messaggio originale "string to encode" ed il secondo che restituisce null.

Metodo verify

Il terzo metodo analizzato è "verify", che si occupa di verificare se una cifratura è avvenuta utilizzando un determinato algoritmo o meno; i parametri in input a questo metodo sono tre, una prima stringa ("value") rappresenta il messaggio non cifrato, un cipherAlgorithm che specifica ancora una volta l'algoritmo di cifratura in esame ed un'altra stringa ("encoded") che corrisponde al messaggio cifrato; il valore di ritorno è un booleano che indica se la prima stringa codificata tramite l'algoritmo scelto corrisponde alla terza. I test sfruttano ancora una volta gli stessi parametri e sono (1): ("string to encode", cipherAlgorithm.AES, MSGC) e (2): (null, cipherAlgorithm.AES, null); entrambi i test restituiscono il valore atteso, true nel primo caso e false nel secondo, questo perché il metodo restituisce sempre false in caso di valore null, comportamento che potrebbe essere considerato sbagliato ma è comunque un caso limite che meriterebbe di essere analizzato più in dettaglio studiando l'uso di questo metodo fatto dal sistema in generale. Va fatto notare come questo ultimo test riscontri qualche piccolo problema con PitTest che al momento della mutazione finisce con il lanciare un'eccezione, tuttavia il test risulta corretto ed incrementa la copertura anche secondo i criteri di Jacoco.

Test non parametrizzati

Questa sezione permette di realizzare dei test per casi particolari o che comunque vengono eseguiti una sola volta e non replicati per ogni lista di parametri.

Il primo test si occupa della questione rimasta in sospeso poco sopra, il controllo del comportamento del sistema in caso di una stringa vuota in input. In un primo momento questo test analizzava il metodo encode per verificare che la codifica di una stringa vuota (non nulla) restituisse ancora una stringa vuota, ma questo non è accaduto, poiché viene restituita una stringa di una certa lunghezza; per sicurezza è stato dunque aggiunto questo test che, dopo la cifratura, effettua anche l'operazione inversa, inserendo la stringa ottenuta nel metodo decode, che finisce per restituire nuovamente la stringa vuota originale. Permettere di cifrare una stringa vuota ottenendo un output non vuoto è un comportamento probabilmente piuttosto strano, ma l'ottenimento della stringa vuota dalla fase di decoding fa sì che tale caso limite possa in qualche modo essere ancora considerato corretto.

Il secondo test viene inserito con il contributo delle analisi effettuate da Jacoco/Sonar e PitTest, permettendo di coprire parti di codice finora scoperte. Il costruttore privato della classe, richiamato dal metodo getInstance sopra citato non si limita ad impostare la chiave di cifratura ottenuta in input, ma ne controlla anche la corretta lunghezza minima, questa deve infatti essere di 16 caratteri (da cui il nome della chiave di cifratura nella sezione precedente "secretKeyLongEnough" di lunghezza superiore al minimo). In questo test viene invece utilizzata una chiave di cifratura minore di 16 caratteri per confermare il corretto funzionamento del sistema in tali condizioni (vengono aggiunti caratteri random fino al raggiungimento della soglia minima); in aggiunta le analisi della coverage evidenziavano come una condizione che compara il cipherAlgorithm in input con null risultasse non coperta, pertanto in questo test viene inserito null come algoritmo. Anche questo test è strutturato con una cifratura tramite metodo encode ed una verifica della stessa tramite il metodo verify.

Il terzo ed ultimo test realizzato si occupa invece di una questione rimasta in sospeso dall'origine, al momento della partizione dei domini di input; finora infatti è stato utilizzato soltanto AES come algoritmo di cifratura (o null, che porta comunque alla scelta di quest'ultimo come comportamento di default), è ora quindi necessario controllare il corretto funzionamento anche con tutti gli altri algoritmi nella enumerazione. Questo test è strutturato come il precedente ma con un ciclo che esamina tutte le possibili alternative di algoritmo di cifratura, eseguendo prima una codifica con il metodo encode e verificandola poi con il metodo verify. La realizzazione di questo test permette inoltre di incrementare notevolmente i livelli di copertura, soprattutto a livello di linee e di condizione, poiché il codice testato esercita condizioni diverse a seconda dell'algoritmo utilizzato per la cifratura.

Coverage

Al completamento delle analisi, in seguito alla realizzazioni di questi test, i livelli di copertura raggiunti sono: 80,6% per la line coverage, 78,6% per la condition coverage e 50% per la mutation coverage. La maggior parte del codice non coperto dai test fa riferimento al blocco statico discusso all'inizio della sezione, che effettua una lettura di un file di configurazione (security.properties) ed imposta dei valori di default in caso di errore o file mancante; per il test di tale sezione potrebbe essere indicato un mock che restituisca un valore null, in modo da sollecitare le condizioni, ma la classe chiamata per la lettura è definita come static final, e sia Mockito che PowerMock hanno riscontrato problemi con tali caratteristiche (sarebbe probabilmente necessario trovare delle impostazioni particolari di questi ultimi).

Configurazione

Vengono qui descritte rapidamente le configurazioni utilizzate per poter utilizzare gli strumenti di analisi della copertura del codice dai test, in particolare Jacoco (insieme a Sonar), PitTest e Ba-Dua, mentre verranno omesse considerazioni su altri strumenti come Travis CI.

Jacoco/Sonar

Questi due strumenti vengono utilizzati per l'analisi dei livelli di copertura del codice al livello di linee e di condizioni; Sonar verrà collegato tramite Travis ed effettuerà l'analisi in automatico ad ogni build su quest'ultimo. La configurazione di Jacoco, dopo aver aggiunto il plugin nella sezione "plugin management" del parent pom (in modo che venga utilizzato da tutti i sottomoduli), è effettuata tramite due sezioni separate. In primo luogo avviene la fase di preparazione dell'agent, per maggiore chiarezza all'interno di un profilo dedicato, che risulta comunque attivo di default, sempre all'interno del pom principale; questo ne permette l'esecuzione a monte di tutti i sottomoduli. Viene dunque aggiunto un modulo apposito come ultimo modulo, chiamato tests, che sarà incaricato di gestire l'aggregazione dei risultati dei test degli altri moduli; all'interno del pom del modulo tests si inserisce il componente aggregatore, che si occuperà di mettere insieme gli esiti dei vari test e generare il report. La connessione agli altri moduli avviene tramite la specifica di questi ultimi effettuata nella sezione "dependencies" del pom stesso. Per maggiore chiarezza e semplicità di visualizzazione sono stati impostati dei filtri di visualizzazione all'interno del componente aggregatore, che consentono di mostrare nel report esclusivamente le classi volute (quelle coperte da almeno un caso di test). La configurazione di Sonar è invece dipendente da quella di Jacoco, poiché utilizza il file di report generato da quest'ultimo per computare i propri livelli di copertura; questo avviene tramite la specifica all'interno dei pom dei vari moduli di cui è necessario aggregare i risultati di una proprietà nella sezione "properties". Anche in Sonar, come era avvenuto per Jacoco è stato impostato un filtro per la visualizzazione esclusiva delle classi interessate per maggiore chiarezza; questo è stato effettuato direttamente utilizzando il sito internet di quest'ultimo.

PitTest

Questo strumento è utilizzato per l'analisi del livello di mutation coverage del codice. La sua configurazione avviene in maniera similare a quella di Jacoco, anche qui viene infatti sfruttata l'esecuzione sequenziale dei vari moduli del progetto. Viene inserito nel pom dei vari sottomoduli in cui inserire i test il plugin contenente varie indicazioni, tra cui la fase di esecuzione di quest'ultimo (verify) ed i filtri per la visualizzazione esclusiva dei livelli di copertura delle classi su cui sono stati realizzati i test, in maniera similare a quanto avvenuto nell'ultima fase di Jacoco; a questo punto nel pom del modulo tests viene inserito il componente aggregatore, che si occupa di generare il report unendo i risultati ottenuti dalle analisi dei vari sottomoduli. Risulta importante notare come l'esecuzione di tale plugin necessiti di informazioni salvate precedentemente

nel sistema dalle esecuzioni dei moduli rispetto a cui le dipendenze vengono specificate; ciò comporta un errore nel caso questi non vengano trovati, portando così alla necessità di specificare durante la compilazione (sia in locale che su Travis) l'utilizzo della fase di install (prima di inserire questo componente ci si fermava alla fase di verify).

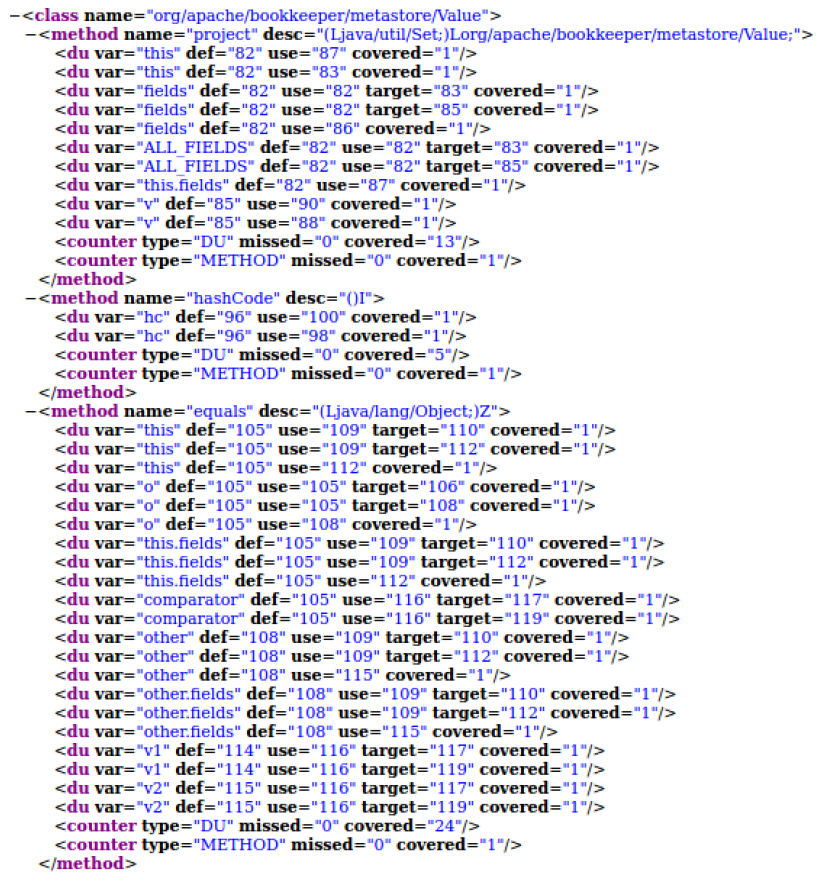
Ba-Dua

Quest'ultimo strumento viene utilizzato per l'analisi dei livelli di copertura del codice basati su criteri di tipo data-flow, in qualche modo considerabili più stringenti rispetto ai precedenti; tale strumento è comunque sperimentale e non è pertanto semplice trovare informazioni al riguardo per ottenere una configurazione funzionante che non entri in conflitto con altri strumenti. Per i motivi appena descritti l'uso di Ba-Dua è stato ristretto ai due casi di studio legati a BookKeeper, a causa di un problema di incompatibilità di uno dei comandi necessari all'uso del java agent con la jdk di Syncope (Xbootclasspath/p funziona esclusivamente fino alla jdk versione 8, mentre quest'ultimo progetto necessita della versione 11). Per la configurazione di Ba-Dua sono state aggiunte le indicazioni per l'esecuzione del java agent all'interno del pom principale, tra le configurazioni del plugin di Surefire. Tra le informazioni necessarie vi è il riferimento al jar del ba-dua-agent, che è stato necessario inserire all'interno del progetto; una volta ottenuto un file di coverage è stato utilizzato il programma report all'interno di ba-dua-cli per generare per l'appunto il report desiderato.

Immagini

Vengono qui riportate le immagini (screenshot) relativi alle coperture del codice analizzate con Ba-Dua; in generale tutti i livelli di copertura sono facilmente verificabili tramite il materiale allegato al progetto, ma la consultazione di queste può risultare utile per una migliore comprensione della sezione inerente l'analisi eseguita (soprattutto per la classe RoundRobinDistributionSchedule ed il metodo getEntriesStripedToTheBookie), che ha un test pensato appositamente.

Data-Flow classe Value:



```
−<method name="merge" desc="(Lorg/apache/bookkeeper/metastore/Value;)
    <du var="this" def="130" use="137" covered="1"/>
    <du var="this" def="130" use="134" covered="1"/>
    <du var="this" def="130" use="132" covered="1"/>
    <du var="this.fields" def="130" use="134" covered="1"/>
    <du var="this.fields" def="130" use="132" covered="1"/>
    <du var="entry" def="130" use="131" target="132" covered="1"/>
    <du var="entry" def="130" use="131" target="134" covered="1"/>
    <du var="entry" def="130" use="134" covered="1"/>
    <du var="entry" def="130" use="132" covered="1"/>
    <counter type="DU" missed="0" covered="12"/>
    <counter type="METHOD" missed="0" covered="1"/>
  </method>
−<method name="toString" desc="()Ljava/lang/String;">
    <du var="UTF_8" def="142" use="153" covered="1"/>
    <du var="sb" def="142" use="157" covered="1"/>
    <du var="sb" def="142" use="158" covered="1"/>
    <du var="sb" def="142" use="155" covered="1"/>
    <du var="entry" def="144" use="150" target="151" covered="1"/>
    <du var="entry" def="144" use="150" target="153" covered="1"/>
    <du var="entry" def="144" use="153" covered="1"/>
    <du var="f" def="145" use="146" target="147" covered="1"/>
    <du var="f" def="145" use="146" target="150" covered="1"/>
    <du var="f" def="145" use="155" covered="1"/>
    <du var="f" def="147" use="155" covered="1"/>
    <du var="value" def="151" use="155" covered="1"/>
    <du var="value" def="153" use="155" covered="1"/>
    <counter type="DU" missed="0" covered="16"/>
    <counter type="METHOD" missed="0" covered="1"/>
  </method>
  <counter type="DU" missed="0" covered="70"/>
  <counter type="METHOD" missed="0" covered="5"/>
  <counter type="CLASS" missed="0" covered="1"/>
</class>
```

Data-flow classe RoundRobinDistributionSchedule, metodo moveAndShift

```
−<method name="moveAndShift" desc="(II)V">
    <du var="this" def="176" use="185" covered="1"/>
    <du var="this" def="176" use="189" covered="1"/>
    <du var="this" def="176" use="187" covered="1"/>
    <du var="this" def="176" use="179" covered="1"/>
    <du var="this" def="176" use="183" covered="1"/>
    <du var="this" def="176" use="181" covered="1"/>
    <du var="from" def="176" use="178" target="179" covered="1"/>
    <du var="from" def="176" use="178" target="184" covered="1"/>
    <du var="from" def="176" use="184" target="185" covered="1"/>
    <du var="from" def="176" use="184" target="191" covered="1"/>
    <du var="from" def="176" use="185" covered="1"/>
    <du var="from" def="176" use="186" covered="1"/>
    <du var="from" def="176" use="179" covered="1"/>
    <du var="from" def="176" use="180" covered="1"/>
    <du var="to" def="176" use="178" target="179" covered="1"/>
    <du var="to" def="176" use="178" target="184" covered="1"/>
    <du var="to" def="176" use="184" target="185" covered="1"/>
    <du var="to" def="176" use="184" target="191" covered="1"/>
    <du var="to" def="176" use="186" target="187" covered="1"/>
    <du var="to" def="176" use="186" target="189" covered="1"/>
    <du var="to" def="176" use="189" covered="1"/>
    <du var="to" def="176" use="180" target="181" covered="1"/>
    <du var="to" def="176" use="180" target="183" covered="1"/>
    <du var="to" def="176" use="183" covered="1"/>
    <du var="this.array" def="176" use="185" covered="1"/>
    <du var="this.array" def="176" use="189" covered="1"/>
    <du var="this.array" def="176" use="187" covered="1"/>
    <du var="this.array" def="176" use="179" covered="1"/>
    <du var="this.array" def="176" use="183" covered="1"/>
    <du var="this.array" def="176" use="181" covered="1"/>
    <du var="tmp" def="179" use="183" covered="1"/>
    <du var="i" def="180" use="180" target="181" covered="1"/>
    <du var="i" def="180" use="180" target="183" covered="0"/>
    <du var="i" def="180" use="181" covered="1"/>
    <du var="i" def="180" use="180" covered="1"/>
    <du var="i" def="180" use="180" target="181" covered="1"/>
    <du var="i" def="180" use="180" target="183" covered="1"/>
    <du var="i" def="180" use="181" covered="1"/>
    <du var="i" def="180" use="180" covered="1"/>
    <du var="tmp" def="185" use="189" covered="1"/>
    <du var="i" def="186" use="186" target="187" covered="1"/>
    <du var="i" def="186" use="186" target="189" covered="0"/>
    <du var="i" def="186" use="187" covered="1"/>
    <du var="i" def="186" use="186" covered="1"/>
    <du var="i" def="186" use="186" target="187" covered="1"/>
    <du var="i" def="186" use="186" target="189" covered="1"/>
    <du var="i" def="186" use="187" covered="1"/>
    <du var="i" def="186" use="186" covered="1"/>
    <counter type="DU" missed="2" covered="46"/>
    <counter type="METHOD" missed="0" covered="1"/>
  </method>
```

Data-flow classe RoundRobinDistributionSchedule, metodo getEntriesStripedToTheBookie

```
−<method name="getEntriesStripedToTheBookie" desc="(IJJ)Ljava/util/BitSet;">
    <du var="this" def="418" use="423" covered="1"/>
    <du var="this" def="418" use="418" target="418" covered="1"/>
    <du var="this" def="418" use="418" target="420" covered="1"/>
    <du var="this" def="418" use="428" covered="1"/>
    <du var="this" def="418" use="429" covered="1"/>
    <du var="bookieIndex" def="418" use="423" covered="1"/>
    <du var="bookieIndex" def="418" use="418" target="418" covered="1"/>
    <du var="bookieIndex" def="418" use="418" target="420" covered="1"/>
    <du var="bookieIndex" def="418" use="418" target="418" covered="1"/>
    <du var="bookieIndex" def="418" use="418" target="420" covered="1"/>
    <du var="bookieIndex" def="418" use="435" target="435" covered="1"/>
    <du var="bookieIndex" def="418" use="435" target="436" covered="1"/>
    <du var="bookieIndex" def="418" use="435" target="436" covered="1"/>
    <du var="bookieIndex" def="418" use="435" target="427" covered="1"/>
    <du var="bookieIndex" def="418" use="431" target="431" covered="1"/>
    <du var="bookieIndex" def="418" use="431" target="427" covered="1"/>
    <du var="bookieIndex" def="418" use="431" target="432" covered="1"/>
    <du var="bookieIndex" def="418" use="431" target="427" covered="1"/>
    <du var="startEntryId" def="418" use="418" target="418" covered="1"/>
    <du var="startEntryId" def="418" use="418" target="420" covered="1"/>
    <du var="startEntryId" def="418" use="423" covered="1"/>
    <du var="startEntryId" def="418" use="418" target="420" covered="1"/>
    <du var="startEntryId" def="418" use="418" target="426" covered="1"/>
    <du var="startEntryId" def="418" use="426" covered="1"/>
    <du var="startEntryId" def="418" use="427" covered="1"/>
    <du var="startEntryId" def="418" use="436" covered="1"/>
    <du var="startEntryId" def="418" use="432" covered="1"/>
    <du var="lastEntryId" def="418" use="423" covered="1"/>
    <du var="lastEntryId" def="418" use="418" target="418" covered="1"/>
    <du var="lastEntryId" def="418" use="418" target="420" covered="1"/>
    <du var="lastEntryId" def="418" use="418" target="420" covered="1"/>
    <du var="lastEntryId" def="418" use="418" target="426" covered="1"/>
    <du var="lastEntryId" def="418" use="426" covered="1"/>
    <du var="lastEntryId" def="418" use="427" target="428" covered="1"/>
    <du var="lastEntryId" def="418" use="427" target="440" covered="1"/>
    <du var="this.ensembleSize" def="418" use="423" covered="1"/>
    <du var="this.ensembleSize" def="418" use="418" target="418" covered="1"/>
    <du var="this.ensembleSize" def="418" use="418" target="420" covered="1"/>
    <du var="this.ensembleSize" def="418" use="428" covered="1"/>
    <du var="this.ensembleSize" def="418" use="429" covered="1"/>
    <du var="LOG" def="418" use="420" covered="1"/>
    <du var="this.writeQuorumSize" def="418" use="429" covered="1"/>
    <du var="entriesStripedToTheBookie" def="426" use="440" covered="1"/>
    <du var="entriesStripedToTheBookie" def="426" use="436" covered="1"/>
    <du var="entriesStripedToTheBookie" def="426" use="432" covered="1"/>
    <du var="entryId" def="427" use="427" target="428" covered="1"/>
    <du var="entryId" def="427" use="427" target="440" covered="0"/>
    <du var="entryId" def="427" use="428" covered="1"/>
    <du var="entryId" def="427" use="429" covered="1"/>
    <du var="entryId" def="427" use="436" covered="0"/>
    <du var="entryId" def="427" use="427" covered="1"/>
    <du var="entryId" def="427" use="432" covered="1"/>
    <du var="modValOfFirstReplica" def="428" use="430" target="431" covered="1"/>
    <du var="modValOfFirstReplica" def="428" use="430" target="435" covered="1"/>
    <du var="modValOfFirstReplica" def="428" use="435" target="435" covered="1"/>

   <du var="modValOfFirstReplica" def="428" use="435" target="435" covered="1"/>
   <du var="modValOfFirstReplica" def="428" use="435" target="436" covered="1"/>
   <du var="modValOfFirstReplica" def="428" use="431" target="431" covered="1"/>
   <du var="modValOfFirstReplica" def="428" use="431" target="427" covered="1"/>
   <du var="modValOfLastReplica" def="429" use="430" target="431" covered="1"/>
   <du var="modValOfLastReplica" def="429" use="430" target="435" covered="1"/>
   <du var="modValOfLastReplica" def="429" use="435" target="436" covered="1"/>
   <du var="modValOfLastReplica" def="429" use="435" target="427" covered="1"/>
   <du var="modValOfLastReplica" def="429" use="431" target="432" covered="1"/>
   <du var="modValOfLastReplica" def="429" use="431" target="427" covered="1"/>
   <du var="entryId" def="427" use="427" target="428" covered="1"/>
   <du var="entryId" def="427" use="427" target="440" covered="1"/>
   <du var="entryId" def="427" use="428" covered="1"/>
   <du var="entryId" def="427" use="429" covered="1"/>
   <du var="entryId" def="427" use="436" covered="1"/>
   <du var="entryId" def="427" use="427" covered="1"/>
   <du var="entryId" def="427" use="432" covered="1"/>
   <counter type="DU" missed="2" covered="69"/>
   <counter type="METHOD" missed="0" covered="1"/>
</method>
```

Miglioramento della copertura rispetto all'immagine precedente dopo l'esecuzione del test (11)

```
<du var="entryId" def="427" use="429" covered="1"/>
<du var="entryId" def="427" use="436" covered="1"/>
<du var="entryId" def="427" use="427" covered="1"/>
```

Tabella dei test della classe Value

Vengono qui riportati i nomi dei test della classe Value di cui si è parlato durante la trattazione per poterli verificare direttamente; le altre classi fanno uso per lo più di test parametrici, i cui valori sono già stati discussi, risulterebbe pertanto superfluo trascriverli qui di seguito.

Metodo equals

Numero	Nome	Esito
(1) – (8)	testEqualsNotSameSize	positivo
(2)	testEqualsNotSameType	positivo
(3)	testEqualsNotSameEntries1	negativo
(4)	testEqualsTrue1	negativo
(5)	testEqualsNotSameEntries2	negativo
(6)	testEqualsNotSameValues	positivo
(7)	testEqualsTrue2	positivo

Metodo hashCode

Numero	Nome	Esito
(1)	testEqualsTrue2	positivo
(2)	testHashCodesNotSameValues	ignorato
(3)	testHashCodesNotSameEntries	positivo

Metodo toString

Numero	Nome	Esito
(1) – (2)	testToString	positivo

Metodo project

Numero	Nome	Esito
(1)	testProjectPartial	positivo
(2)	testProjectTotal	positivo

Metodo merge

Numero	Nome	Esito
(1) – (2)	testMerge	positivo